On Advances in Deep Learning with Applications in Financial Market Modeling

by

Xing Wang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 8, 2020

Keywords: Deep learning, Stock prediction, Convolutional neural network, Stock2Vec, Deep Q-network, Exploration, Overestimation, Cross Q-learning, DQN Trading

Approved by

Alexander Vinel, Chair, Assistant Professor of Industrial and Systems Engineering
Jorge Valenzuela, Distinguished Professor of Industrial and Systems Engineering
Daniel F. Silva, Assistant Professor of Industrial and Systems Engineering
Erin Garcia, Industrial and Systems Engineering

Abstract

This dissertation focuses on advancing the machine learning, with a particular focus on the application for financial trading. It is organized into two parts. The first part of this dissertation (Chapters 1-2) will be concerned with the application of predictive modeling on stock market prediction. Chapter 1 presents the basics of machine learning and deep learning. In Chapter 2, we combine several recent advances in deep learning to build a hybrid model to forecast the stock prices, that gives us the ability to learn from various aspects of the related information. In particular, we take a deep look at the representation learning and temporal convolutional network for sequential modeling. With representation learning, we derived an embedding called Stock2Vec, which gives us insight for the relationship among different stocks, while the temporal convolutional layers are used for automatically capturing effective temporal patterns both within and across series. Our hybrid framework integrates both advantages and achieves better performance on the stock price prediction task than several popular benchmarked models.

In the second part of this dissertation (Chapters 3 - 6), we turn our focus to the topics of reinforcement learning. In Chapter 3, we provide the necessary mathematical and theoretical preliminaries in reinforcement learning, as well as several recent advances in deep Q-networks (DQNs) that we would apply later. In Chapters 4 and 5, we aim at algorithmically improving the convergence of training in reinforcement learning, with theoretical analysis and empirical experiments. One prominent challenge in reinforcement learning is the tradeoff between exploration and exploitation. In deep Q-networks (DQNs), this is usually addressed by monotonically decreasing the exploration rate yet is often unsatisfactory. In Chapter 4, we propose to encourage exploration by resetting the exploration rate when it

is necessary. Another severe problem in training deep Q-networks involves the overestimation for the Q-values. In Chapter 5, we propose to bootstrap the estimates from multiple agents, and refer to this learning paradigm as cross Q-learning. Our algorithm effectively reduces the overestimation and significantly outperforms the state-of-the-art DQN training algorithms. In Chapter 6, we continue our studies on DQN with an application in real financial trading environment, by training a DQN agent that provides trading strategies. Finally, we summarize this dissertation in Chapter 7, and discuss the possible directions for future research.

Acknowledgments

The first person I am greatly indebted to during my Ph.D. studies is my advisor, Dr. Alexander Vinel. He is the kindest advisor a Ph.D. student could have asked for. Dr. Vinel fully supports every decision I made and gives me lots of freedom to grow as an independent researcher; he encourages me at every difficult moment and takes good care of me both academically and personally. I must express special thanks to Dr. Fadel Megahed and Dr. Daniel Silva. I was benefited greatly from many discussions with them to gain guidance and excellent insights, both of them have significant impact on my research. I would also like to thank the other committee members, Dr. Jorge Valenzuela, Dr. Erin Garcia, and Dr. Levent Yilmaz, for their time and effort on giving me invaluable feedback. Thanks also go to all my collaborators, Bing Weng, Lin Lu, Yijun Wang, and Waldyn Martinez, for their kind and brilliant help, and I had great pleasure to work with them. Looking back the long journey, I cannot forget to express my sincere gratitude to my masters advisors and mentors in other majors, Dr. Alvin Lim, Dr. Yujin Chung, and Dr. Henry Kinnucan, I am fortunate enough to be guided by them and have learned a lot from them as well. Last but not the least, I need to thank my parents for their long-lasting support and love during my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Machine Learning and Deep Learning Preliminaries

1.1 Introduction to Machine Learning

Machine learning is the study of establishing models, or machines that can learn from data. A commonly cited definition is "a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [1]. Upon the distinction of the tasks T, there are generally three subfields in machine learning: supervised learning, unsupervised learning, and reinforcement learning.

In supervised learning, we are given a set of experience E that consists of N input-output pairs $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, each represents a training example, in which the input $\mathbf{x}$ is described by a vector of features, and the output y is often called the label. The task T is to find a model f to predict the labels for a set of new test data. If the labels are discrete, $f(\mathbf{x})$ can be interpreted as an estimate of the category that $\mathbf{x}$ belongs to, and it is a classification task; on the contrary, it is called a regression task if the labels are continuous. Linear regression might be served as a simple form of the supervised learning problem, our model in which is a linear transformation of the inputs: $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, in which $\mathbf{w}$ is a vector of parameters, in the least squares setting, we aim at minimize the mean squared error (MSE) over the training data:

$$\frac{1}{N}\sum_{i=1}^{N} ||y_i - \mathbf{w}^T\mathbf{x}_i||^2. \tag{1.1}$$

In contrast, in unsupervised learning problems, the experience E does not include the labels, in other words, we are only given X, and the task T is to perform some transformation

or obtain some insight from the inputs, includes density estimation, dimensionality reduction, clustering, and representation learning, etc.

We will introduce reinforcement learning in details in Chapter 3, but here we would like to illustrate its distinction with supervised learning. In reinforcement learning, the actions can be seen as labels, but our task T is sequential decision making, instead of making only the decision once as in supervised learning. Moreover, the experience E is collected by interacting with the environment throughout learning, unlike in supervised learning, all the samples are given beforehand. Thus we would have the exploration-exploitation dilemma in reinforcement learning, which we would like to address in Chapter 4.

1.2 Support Vector Machine

To explain the learning process from statistical point of view, [2] proposed VC learning theory, and one of its major components characterizes the construction of learning machines that enable them to generalize well. Based on that, Vapnik and his colleagues in Bell laboratory developed the support vector machine (SVM) [3, 4] which has been shown as one of the most influential supervised learning algorithms til now. The key insight of SVM is that those points closest to the separator, called the support vectors, are more important than others. Assigning non-zero weights only to those support vectors while constructing the learning machine can lead to better generalization, and the separator is then called the maximum margin separator. [5] then expanded the idea to regression problems, by omitting the training points which deviate the actual targets less than a threshold ε while calculating the cost. These points with small errors are also called support vectors, and the corresponding learning machine for the classification or regression task is called the support vector machine (SVM).

The goal of training SVM is to find a hyperplane that maximize the margin, which is equivalent to minimize the norm of the weight vector for every support vectors, subject to the constrains that make each training sample valid, i.e., the optimization problem can be

written as

$$\begin{aligned} \min \quad & \tfrac{1}{2}||w||^2 \\ \text{s.t.} \quad & y_i - w^T x_i - b \leq \varepsilon \\ & w^T x_i + b - y_i \leq \varepsilon \end{aligned} \tag{1.2}$$

where x_i is a training sample with target y_i. We will not show the details here, but maximizing the Lagrangian dual is a much simpler quadratic programming optimization problem. It is convex thus would not stuck in local optima, and has well-studied techniques to solve, such as the squential minimal optimization (SMO) algorithm [6] that is specialized for minimizing SVM loss.

[4] also introduced the idea of soft margin which allows some misclassification by assigning them a penalty proportional to the distance to their correct class. The allowance of softness in margins dramatically reduces the computational work while training SVM, but more importantly, it captures the noisiness of real world data and could obtain more generalizable model, as in contrast, hard margin results in zero errors in training data, but the model is possibly overfitting. Another key technique that makes SVM successful is the use of so-called "kernel trick", which maps the non-linearly-separable original input into higher dimensional space so that the data become linearly-separable, thus greatly expand the hypothesis space [7].

However, SVM has its own disadvantages. The performance of SVM is extremely sensitive to the selection of the kernel function as well as the parameters. Another major drawback to kernel machines is that the computational cost of training is high when the dataset is large [8], and also suffers the curse of dimensionality and struggles to generalize well.

1.3 Boosting

Rooted in probably approximately correct (PAC) learning [9], [10] posed the question that whether a set of "weak" learners (i.e., learners that perform slightly better than random guessing) can be combined to produce a learner with accuracy arbitrarily high. [11] and [12] then showed the affirmative answer by giving a boosting algorithm, and the most popular boosting algorithm Adaboost was also developed by [13]. Adaboost addresses two fundamental questions in the idea of boosting: how to choose the distribution in each round, and how to combine the weak rules into a single strong learner [14]. It uses the "importance weights" to force the learner pay more attention on those examples having larger errors, that is, iteratively fits a learner using the weighted data and updates the weights using the error from the fitted learner, and lastly combines these weak learners together through a weighted majority vote. Boosting is generally computationally efficient and has no difficult parameters to set, it (theoretically) guarantees to provide desired accuracy given sufficient data and a reliable base learner. However, practically, the performance of boosting significantly depends on the sufficiency of data as well as the choice of base learner. Applying base learners that are too weak would definitely fail to work, overly complex base learners could result in overfitting on the other hand. It also seems susceptible to uniform noise [15], since it may over-emphasize on the highly noisy examples in later training and result in overfitting.

As an "off-the-shelf" supervised learning method, the decision tree method is used most common in the choice of base learners for boosting. It is one of the simplest to train yet powerful and easy to represent. It partitions the space of all joint predictor variable values into disjoint regions using greedy search, either based on the error or the information gain. However, due to its greedy strategy, the results obtained by the decision tree might be unstable and have high variance, thus often achieve lower generalization accuracy. One common way to improve its performance is boosting, which primarily reduces the bias as well as the variances [16].

1.3.1 Gradient Boosted Tree and XGBoost

The key idea of gradient boosting is to use gradient descent to find the optimal weak learner at each iteration that forms the final additive model, which requires the objective function to be differentiable to calculate the gradient. Consider the additive boosting, let f_t denote the weak learner obtained at iteration t, the predictive value of input x_i can be formed by the additive model as

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i), \tag{1.3}$$

The objective of the optimization problem is then

$$\begin{aligned} \min \quad \mathcal{J}^{(t)} &= \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) && (1.4) \\ &= \sum_{i=1}^{n} \mathcal{L}\big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\big) + \Omega(f_t) + \text{constant} && (1.5) \end{aligned}$$

where $\mathcal{L}$ denotes the loss function, Ω is the regularization penalty, and n is the number of samples. We can obtain Equation (1.5) since the weak learners from previous iterations are fixed in additive boosting. Let g_i and h_i denote the gradient and Hession for the loss function with respect to i-th sample, respectively, i.e.,

$$g_i = \frac{\partial \mathcal{L}(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}, \tag{1.6}$$

$$h_i = \frac{\partial^2 \mathcal{L}(y_i, \hat{y}_i^{(t-1)})}{\partial (\hat{y}_i^{(t-1)})^2}. \tag{1.7}$$

With Taylor expansion, we rewrite Equation (1.5) as

$$\begin{aligned} \min \quad \mathcal{J}^{t} &= \sum_{i=1}^{n} \big[\mathcal{L}(y_i, \hat{y}_i^{t-1}) + g_i f_t(x_i)) + + \frac{1}{2} h_i f_t^2(x_i)\big] + \Omega(f_t) + \text{constant} && (1.8) \\ &= \sum_{i=1}^{n} \big[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)\big] + \Omega(f_t) && (1.9) \end{aligned}$$

after removing all the constants. This optimization problem then can be solved with the gradient and Hessian for each sample as inputs.

As for gradient boosting tree, further define an individual tree can be parameterized as

$$f_t(x) = w_{q(x)}, \tag{1.10}$$

where $w \in \mathbb{R}^T$ denotes the score vector on leaves, T is the number of leaves, and $q : \mathbb{R}^d \to \{1, \cdots, T\}$ assigns sample x to the corresponding leaf. And define the regularization term as

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2. \tag{1.11}$$

Equation (1.9) can then be written as

$$\min \quad \mathcal{J}^t \approx \sum_{i=1}^{n} \left[g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2 \right] \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{1.12}$$

$$= \sum_{j=1}^{T} \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\lambda + \sum_{j \in I_j} h_i) w_j^2 \right] + \gamma T \tag{1.13}$$

where $I_j = \{i | q(x_i) = j\}$ is the index set for samples that are assigned to leaf j. We solve the problem iteratively with greedy algorithm, i.e., coordinate descent for each leaf, then each term in Equation (1.13) is of quadratic form, and has simple analytic solution that

$$w_j^* = -\frac{G_j}{H_j + \lambda} \tag{1.14}$$

in which $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$ for conciseness. And we can see how easily and efficiently the gradient boosted tree can be obtained during an iteration with the gradient and Hession information, according to Equation (1.14).

XGBoost [17] refers to a software package that implements the gradient boosting with extreme engineering effort. It was designed to be scalable, efficient, flexible and portable,

the common advantages of XGBoost include: cache-aware access for memory efficiency, out-of-core computation, efficient handling of sparse data, theoretical justified approximation for instance weights, built-in cross-validation for regularization and tree pruning, etc. The library is very user-friendly, training XGBoost models can be very fast and easily parallelized as well as distributed across cluster.

1.4 Bagging and Random Forest

The name "bagging" is abbreviated from bootstrap aggregating, which is another ensemble meta-algorithm in machine learning. Bagging averages the predicted value (or aggregates votes with equal weights as in classification task) over a collection of bootstrap subset of training samples (i.e., by sampling with replacement), thus reducing the variance. The bagging estimate would differ from the original estimate only if the latter is a nonlinear or adaptive function of the data [16].

Some supervised learning methods, such as the decision tree, are sensitive to outliers and have high variance, thus are often combined with bagging to avoid an individual model overfitting the training data. Random forest [18] further improves the bagged decision trees. While each decision tree selects optimal split on features with greedy algorithm, even with bagging, the outputs from different trees are often highly correlated, due to the similarity of trees. Random forest alleviates this issue by searching over only a random subset of features for each split, results in less correlation in the outputs.

1.5 Neural Networks and Deep Learning

Inspired by complex biological neuron system in our brain, the artificial neurons were proposed by [19] using the threshold logic. [20] and [21] independently discovered the back-propagation algorithm which could train complex multi-layer perceptrons effectively by computing the gradient of the objective function with respect to the weights, and made the complicated neuron networks widely used since then, especially since the reviving of deep

learning field from 2006 as the parallel computing emerged quickly. Neural networks have been shown as the most successful among machine learning models in stock market prediction, due to its ability to handle complex nonlinear systems over the complex stock market data.

In neural networks, the features are as input x and weighted summed ($z = w^T x$), the information are then transformed by the functions in each neuron and propagated through layers, finally to the output we desired. If there were hidden layers between the input and output layer, the network is called "deep", and the hidden layers could distort the linearity of the weighted sum of inputs, so that the outputs become linearly separable. Theoretically, we can approximate any function that maps the input to the output, if the number of neurons are not limited. And that gives the neural networks the ability to obtain higher accuracy in stock market prediction, where the model is extremely complicated. The functions in each neuron are called "activations", and could have many different types. The most commonly used activation before deep learning era is the sigmoid function, which is smooth and has easy-to-express first order derivative (in terms of the sigmoid function itself), thus is appropriate to train by using back-propagation. Furthermore, its bell-shaped curve is good for classification, but as for regression, this property might be a disadvantage. It is worth to note that the rectified linear unit (ReLU) [22], which takes the simple form $f(z) = \max(z, 0)$, has the advantage of less likely to have vanishing gradient but rather constant (when $z > 0$), thus results in faster learning in networks with many layers. Also, the sparcity of its weights arises as $z < 0$, thus could reduce the complexity of the representation on large architecture. Both properties allow the ReLU become one of the dominant non-linear activation functions in the last few years, especially in the field of deep learning [23].

1.5.1 Convolutional neural network

Convolutional neural networks (CNNs) are a special family of deep neural network model, and have been tremendously successful in practical applications, especially in the field

of computer vision for processing 2D image data. In contrast to standard fully-connected neural networks in which a separate weight describes an interaction between each input and output pair, CNN shares the parameters for multiple mappings. This is achieved by constructing a bunch of kernels (or filters) with fixed size (which is generally much smaller than that of the input), each consists of a set of trainable parameters, therefore, the number of parameters is greatly reduced. The size of the kernels is generally much smaller than that of the input, and each kernel is slide over the entire input to create a feature map. Multiple kernels are usually trained and used together, each is specialized in capturing a specific feature from the data. Note that the so-called convolution operation is technically a cross-correlation in general, which generates linear combinations of a small subset of input, thus focusing on local connectivity. Fundamentally, with CNNs we assume that the input data has some grid-like topology, and the same characteristic of the pattern would be the same for every location, i.e., yields the property of equivariance to translation [24]. The size of the output would then not only depend on the size of the input, also on several settings of the kernels: the stride, padding, and the number of kernels. The stride s denotes the interval size between two consecutive convolution centers, and can be thought of as downsampling the output. Whereas with padding, we add values (zeros are used most often) at the boundary of the input, which is primarily used to control the output size, but as we will show later, it can also be applied to manage the starting position of the convolution operation on the input. The number of kernels adds another dimensionality on the output, and is often denoted as the number of channels.

1.5.2 Recurrent neural network

Recurrent neural network (RNN) and its variants of sequence to sequence (Seq2Seq) framework [25] have achieved great success in many sequential modeling tasks, such as machine translation [26], speech recognition [27], natural language processing [28], and extended

to autoregressive time series forecasting [29, 30] in recent years. However, RNN suffers several major problems, for instance, due to its inherent temporal nature (i.e., the hidden state is propagated through time), the training cannot be parallelized; moreover, training with backpropagation through time (BPTT) [31], RNN can severely suffer the problem of gradient vanishing thus actually cannot capture long time dependency [32]. More elaborate architectures of RNN use gating mechanisms to alleviate the gradient vanishing problem, the long short-term memory (LSTM) [33] and its simplified variant, the gated recurrent unit (GRU) [34], are the two popular architectures commonly used in practice.

In LSTM, a memory cell is used to store the states at time step t, as the cell state vector $\mathbf{c}_t$ and a hidden state vector $\mathbf{h}_t$ would be propagated over time as the information flow. Inside each memory cell, the flow is controlled by parameterized gates: the forget gate $\mathbf{f}_t$, the input gate $\mathbf{i}_t$, and the output gate $\mathbf{o}_t$. The forward propagation of an LSTM cell can be summarized as follows:

$$\mathbf{f}_t = \sigma(\mathbf{W_f}\big[\mathbf{x}_t, \mathbf{h}_{t-1}\big] + \mathbf{b_f}) \tag{1.15}$$

$$\mathbf{i}_t = \sigma(\mathbf{W_i}\big[\mathbf{x}_t, \mathbf{h}_{t-1}\big] + \mathbf{b_i}) \tag{1.16}$$

$$\mathbf{o}_t = \sigma(\mathbf{W_o}\big[\mathbf{x}_t, \mathbf{h}_{t-1}\big] + \mathbf{b_o}) \tag{1.17}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W_c}\big[\mathbf{x}_t, \mathbf{h}_{t-1}\big] + \mathbf{b_c}) \tag{1.18}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{1.19}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \mathbf{c}_t \tag{1.20}$$

where $[\mathbf{x}_t, \mathbf{h}_{t-1}] \in \mathbb{R}^{d_h+d_x}$ denotes the concatenation of the current input $\mathbf{x}_t \in \mathbb{R}^{d_x}$ and the previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^{d_h}$, the activation σ is often a sigmoid function, while the operator $\odot$ denotes the Hadamard product. $\mathbf{W_f}, \mathbf{W_i}, \mathbf{W_o}, \mathbf{W_c} \in \mathbb{R}^{d_h \times (d_h+d_x)}$ and $\mathbf{b_f}, \mathbf{b_i}, \mathbf{b_o}, \mathbf{b_c} \in \mathbb{R}^{d_h}$ are the weights and bias parameters to be learned.

GRU combines the forget gate and input gate in LSTM to a single "update gate" $\mathbf{z}_t$, thus is simpler and has fewer parameters to learn. To eliminate confusion due to renaming,

we note that the reset gate $\mathbf{r}_t$ and hidden state $\mathbf{h}_t$ correspond to the output gate and the cell state in its LSTM counterpart, respectively. The update of a GRU cell is very similar to that of LSTM, and is summarized as follows:

$$\mathbf{z}_t = \sigma(\mathbf{W}_\mathbf{z}\big[\mathbf{x}_t, \mathbf{h}_{t-1}\big] + \mathbf{b}_\mathbf{z}) \tag{1.21}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_\mathbf{r}\big[\mathbf{x}_t, \mathbf{h}_{t-1}\big] + \mathbf{b}_\mathbf{r}) \tag{1.22}$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_\mathbf{h}\big[\mathbf{x}_t, \mathbf{i}_t \odot \mathbf{h}_{t-1}\big] + \mathbf{b}_\mathbf{h}) \tag{1.23}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \tag{1.24}$$

1.6 Regularization

In statistics, weight decay is often called Ridge, or simply L_2 regularization. As argubly the most common regularizer, weight decay is imposed for most of our models, in which the training objective function is assumed to include a regularization term that penalizes a perturbation of unknown parameters in terms of L_2 norm, i.e.,

$$\hat{w} = \arg\min_w \Big\{\mathcal{L}(w) + \lambda||w||_2^2\Big\}, \tag{1.25}$$

where $\lambda \geq 0$ is a complexity parameter that controls the amount of regularization. It is equivalent to adding a hard constrain of the weights to an Euclidean ball, with the radius decided by the amount of weight decay, i.e., an equivalent way of rewriting (1.25) is

$$\begin{aligned} \hat{w} &= \arg\min_w \mathcal{L} \\ \text{subject to} \quad & ||w||_2^2 \leq t, \end{aligned} \tag{1.26}$$

Note there is a one-to-one correspondence between λ in (1.25) and t in (1.26). We could also think of weight decay from Bayesian perspective, it then corresponds to using a symmetric multivariate normal distribution as prior for the weights, i.e., $p(w) \sim \mathcal{N}(w|\mathbf{0}, \lambda^{-1}\mathbf{I})$, as a result, $-\log\mathcal{N}(w|\mathbf{0}, \lambda^{-1}\mathbf{I}) \propto -\log\exp(-\frac{\lambda}{2}||w||_2^2) = \frac{\lambda}{2}||w||_2^2$.

Just as in weight decay, the Lasso (Least Absolute Shrinkage and Selection Operator) estimate [35] is defined as

$$\begin{aligned} \hat{w} &= \arg\min_{w} \mathcal{L} \\ \text{subject to} \quad & |w| \leq t, \end{aligned} \tag{1.27}$$

And its equivalent Lagrangian form is written as

$$\hat{w} = \arg\min_{w} \Big\{ \mathcal{L}(w) + \lambda |w| \Big\}, \tag{1.28}$$

In addition to the shrinkage effect as in Ridge, Lasso also performs variable selection. As the sparsity of the variables can be measured by the L_0 norm, however, the most straightforward optimization problem for imposing sparsity that

$$\arg\min_{w} \Big\{ \mathcal{L}(w) + \lambda ||w||_0 \Big\}, \tag{1.29}$$

is intractable. Lasso can be seen as a convex relaxation for (1.29), in which the L_1 norm is used to replace the non-convex L_0 norm to impose sparsity. While linear models with Lasso regularization are often solved with least angle regression (LAR) [36], more generally, such a convex optimization problem can be solved efficiently using proximal gradient descent such as iterative thresholding algorithm (ITA) [37], as long as $\mathcal{L}(w)$ is convex and has Lipschitz continuous gradient while the other term only needs to be convex, which is often the case in design of machine learning systems. In particular, the alternating direction method of multipliers (ADMM) [38] also often serves as the solver, especially for distributed model fitting with big data, or deriving heuristics when the objective is non-convex. For more details on proximal gradient methods, readers can refer to [39].

A number of simple yet powerful regularization methods have been proposed in recent years that dedicate to prevent overfitting in training deep learning models. Besides the general weight decay and Lasso, such methods include data augmentation [40], early stopping [41], dropout [42], etc. Dropout is a stochastic regularization method which imposes sparsity

constraints by randomness. During training, dropout masks out each element of a layer output randomly with a given dropout probability, this prevents units from excessive co-adapting since the dropped-out neurons can no longer affect other retained units. At test time, predictions are obtained by using the output of all neurons, but are scaled down by the dropout probability. Another way to interpret dropout is that it yields a very efficient form of model averaging where the number of trained models is exponential in that of units, and these models share the same parameters [43].

It also been shown that a bunch of recent advances on deep learning optimization techniques have implicit regularization effect, such as learning rate decay, batch normalization, etc. Batch normalization [44] has been proposed for resolving the internal covariant shift by normalizing layer inputs, in which the distribution of inputs of each layer changes during the training process. However, experimental studies has proven that it also induces both faster convergence and better generalization, by enabling large learning rate and preventing overfitting when training deep networks.

1.7 Principal Component Analysis (PCA) and Robust PCA

The principal component analysis (PCA) in some ways formed the multivariate data analysis and is probably the most commonly used multivariate technique. Its origin can be traced back to [45], who described the geometric view of this analysis as looking for lines and planes of closest fit to systems of points in space. [46] further developed this technique and came up with the term "principal component". The goal of PCA is to extract and only keep the important information of the data. To achieve this, PCA projects the original data into principal components (PCs), also called PC-scores, which are derived as linear combinations of the original variables so that the second-order reconstruction error is minimized. As we know, for normal variables with mean zero, the second-order covariance matrix contains all the information about the data. Thus the PCs provide the best linear approximation to the original data, the first PC is computed as the linear combination to capture the largest

possible variance, then the second PC is constrained to be orthogonal to the first PC while capture the largest possible variance left, and so on. This process can be obtained through the singular value decomposition (SVD), while indeed the PCA is often solved by truncated SVD. Since the variance depends on the scale of the variables, standardization (i.e., centering and scaling) is needed beforehand so that each variable has zero mean and unit standard deviation. Let X be the standardized data matrix, the covariance matrix can be obtained as $\Sigma = \frac{1}{n}XX^T$, which is symmetric and positive definite. By spectral theorem, we can write $\Sigma = Q\Lambda Q^T$, where Λ is a diagonal matrix consists of ordered eigenvalues of Σ, and the column vectors of Q are the correspondent eigenvectors which are orthonormal. The PCs then can be obtained as the columns of $Q\Lambda$. It can be shown [47] that the total variation is equal to the sum of the eigenvalues of the covariance matrix $\sum_{i=1}^{p} \mathrm{Var}(\mathrm{PC}_i) = \sum_{i=1}^{p} \lambda_i = \sum_{i=1}^{p} \mathrm{trace}(\Sigma)$, and the fraction $\sum_{i=1}^{k} \lambda_i / \mathrm{trace}(\Sigma)$ gives the cumulative proportion of the variance explained by the first k PCs. In many cases, the first a few PCs have captured most variation, so the remaining components can be disregarded only with minor information loss. Also it is important to note that PCA derives orthogonal components which are uncorrelated with each other. The optimization problem can then be written as

$$\begin{aligned} \arg\min_{A} \quad & ||X - A||_F^2 \\ \text{subject to} \quad & \mathrm{rank}(A) \leq k \end{aligned} \tag{1.30}$$

where F denotes the Frobenius norm, and A is the low rank representation for the centered data X, with the truncated SVD solution $A = \sum_{i \leq k} \sigma_i u_i v_i^T$.

However, just as other linear models, PCA is highly sensitive to outliers. Robust PCA [48, 49], also called Principal Component Pursuit, solves this issue by considering an additional structure which is assumed to present the sparse outliers, i.e., $X = A + Z + E$, where Z is the matrix of sparse outliers, and E denotes rest of the noise. The optimization problem

is then becomes

$$\begin{aligned} \min_{A} \quad & \text{rank}(A) + \lambda ||Z||_0 \\ \text{subject to} \quad & X = A + Z \end{aligned} \tag{1.31}$$

where λ is the regularization term for Z. While solving the above problem is intractable, two convex relaxations are made. First, the nuclear norm of A, i.e., the sum of singular values, $||A||_* = \sum_i \sigma_i(A)$ replaces the real rank (which can be seen as the L_0 norm of Σ), this relaxation is often applied in solving matrix completion problem. Second, as in Lasso, L_1 instead of L_0 norm of Z is used. We can then write it in augmented Lagrangian form as

$$\mathcal{L}(A, Z, Y) = ||A||_* + \lambda ||Z||_1 + < Y, X - (A + Z) > + \frac{\mu}{2} ||X - (A + Z)||_F^2 \tag{1.32}$$

where Y and $\frac{\mu}{2}$ are the coefficients for the Lagrangian term and the augmentation, respectively, and $< \cdot, \cdot >$ denotes the inner product. This convex problem can then be solved in the form of ADMM with further improvement borrowed elsewhere as follows:

- Given Z and Y, update A:

$$\arg\min_{A} ||A||_* + \frac{\mu}{2} ||X - A - Z + \frac{Y}{\mu}||_F^2.$$

  Note this step is very similar to the convex relaxation of matrix completion problem, and can be efficiently solved by singular value thresholding (SVT) [50], i.e., soft-thresholding on the singular values of matrix $X - S + \frac{Y}{\mu}$ by $\frac{1}{\mu}$.

- Given A and Y, update Z:

$$\arg\min_{Z} ||Z||_1 + \frac{\mu}{2} ||X - A - Z + \frac{Y}{\mu}||_F^2.$$

  While this step has the same form as Lasso, it has closed form solution with soft-thresholding the scalar entries by $\frac{\lambda}{\mu}$, i.e., $Z = \text{sgn}(X - A + \frac{Y}{\mu}) \max(|X - A + \frac{Y}{\mu}| - \frac{\lambda}{\mu}, 0)$.

- Given A and Z, update Y as in ADMM:

$$Y \longleftarrow Y + \mu(X - A - Z).$$

And the above procedure iterates until convergence. Note that the robust PCA can be seen as a special case of imposing regularization on PCA.

1.8 Summary

In this chapter, we introduced several widely applied machine learning models, which serve as a preliminaries for our later chapters, especially for our predictive modelling application in the stock market. The SVM model gives us a general illustration for the linkage between modeling and learning theory. As we will illustrate in Chapter 2, CNN is one of the major components in our proposed model, while its RNN counterpart serves as an important benchmark. We introduced the ensemble models, boosting and bagging in particular, not only since we use the XGBoost and random forest models for additional benchmarking, more importantly, the idea of combining useful and powerful components to form a strong model in order to addressing some particular issue in our need is applied throughout our dissertation. We also discussed several techniques of regularization, as different forms of regularization are applied in all of our models, also although it is not elaborated in this dissertation, developing regularization through optimization advances is one of our major research focus and interest. Finally, we discussed PCA in details, as we used this unsupervised learning technique not only to accomplish some dimensionality reduction tasks for visualization, also for the purpose of comparing with another major component in our proposed model, namely the neural embedding, which is shown in Chapter 2.

Chapter 2

Stock2Vec: A Hybrid Deep Learning Framework for Stock Market Prediction with Representation Learning and Temporal Convolutional Network

We have proposed to develop a global hybrid deep learning framework to predict the daily prices in the stock market. We used entity embedding for the stocks, and trained the models globally over the whole stock market. The trained embedding layer Stock2Vec can reveal some insight of the relationship among stocks, so that we can combine the market information which helps improving the model performance. In addition, 1-D dilated causal convolutional layers are used for capturing the temporal patterns both within and across series from historical data, lead to more accurate predictions. Finally, the models were evaluated on S&P500.

2.1 Introduction

In finance, the classic strong efficient market hypothesis (EMH) posits that the stock prices follow random walk and cannot be predicted [51]. Consequently, the well-known capital assets pricing model (CAPM) [52, 53, 54] serves as the foundation for portfolio management, asset pricing, among many applications in financial engineering. The CAPM assumes a linear relationship between the expected return of an asset (e.g., a portfolio, an index, or a single stock) and its covariance with the market return, i.e., for a single stock, CAPM simply predicts its return r_i within a certain market with the linear equation

$$r_i(t) = \alpha_i + \beta_i r_m(t),$$

where the Alpha (α_i) describes the stock's ability to beat the market, also refers to as its "excess return" or "edge", and the Beta (β_i) is the sensitivity of the expected returns of the stock to the expected market returns (r_m). Both Alpha and Beta are often fitted using simple linear regression based on the historical data of returns. With the efficient market hypothesis (EMH), the Alphas are entirely random with expected value of zero, and can not be predicted.

In practice, however, financial markets are more complicated than the idealized and simplified strong EMH and CAPM. Active traders and empirical studies suggest that the financial market is never perfectly efficient and thus the stock prices as well as the Alphas can be predicted, at least to some extent. Based on this belief, stock prediction has long played a key role in numerous data-driven decision-making scenarios in financial market, such as deriving trading strategies, etc. Among various methods for stock market prediction, the classical Box-Jenkins models [55], exponential smoothing techniques, and state space models [56] for time series analysis are most widely adopted, in which the factors of autoregressive structure, trend, seasonality, etc. are independently estimated from the historical observations of each single series. In recent years, researchers as well as the industry have deployed various machine learning models to forecast the stock market, such as k-nearest neighbors (kNN) [57, 58], hidden Markov model (HMM) [59, 60], support vector machine (SVM) [61, 62], artificial neural network (ANN) [63, 64, 65, 66, 67], and various hybrid and ensemble methods [68, 69, 70, 68, 71], among many others. The literature has demonstrated that machine learning models typically outperform traditional statistical time series models, which might be mainly due to the following reasons: 1) less strict assumption for the data distribution requirement, 2) various model architecture can effectively learn complex linear and non-liner from data, 3) sophisticated regularization techniques and feature selection procedures provide flexibility and strength in handling correlated input features and control of overfitting, so that more features can be thrown in the machine learning models. As the fluctuation of the stock market indeed depends on a variety of related factors, in addition to utilizing the

historical information of stock prices and volumes as in traditional technical analysis [72], recent research of stock market forecasting has been focusing on informative external source of data, for instance, the accounting performance of the company [73], macroeconomic effects [74, 71], government intervention and political events [75], etc. With the increased popularity of web technologies and their continued evolution, the opinions of public from relevant news [76] and social media texts [77, 78] have an increasing effect on the stock movement, various studies have confirmed that combining the extensive crowd-sourcing and/or financial news data facilitates more accurate prediction [79].

During the last decade, with the emergence of deep learning, various neural network models have been developed and achieved success in a broad range of domains, such as computer vision [80, 81, 82, 83] and natural language processing [84, 85, 86]. For stock prediction specifically, recurrent neural networks (RNNs) are the most preferred deep learning models to be implemented [87, 88]. Convolutional neural networks (CNNs) have also been utilized, however, most of the work transformed the financial data into images to apply 2D convolutions as in standard computer vision applications. For example, the authors of [89] converted the technical indicators data to 2D images and classified the images with CNN to predict the trading signals. Alternatively, [90] directly used the candlestick chart graphs (which uses candles to visually representing the open, close, low and high prices, and denoting the moves with different colors) as inputs to determine the Buy, Hold and Sell behavior as a classification task, while similarly in [91], the bar chart images were fed into CNN. The authors of [92] uses a 3D CNN-based framework to extract various sources of data including different markets for predicting the next day's direction of movement of five major stock indices, which showed a significant improved prediction performance compared to the baseline algorithms. There also exists research combining RNN and CNN together, in which the temporal patterns were learned by RNNs, while CNNs were only used for either capturing the correlation between nearby series (in which the order matters if there are more than 2 series) or learning from images, see [93, 94]. Deployment of CNN in all these studies differs

significantly from ours, since we aim at capturing the temporal patterns without relying on two-dimensional convolutions. In [95], 1D causal CNN was used for making predictions based on the history of closing prices only, while no other features were considered.

Note that all of the aforementioned work has put their effort into learning more accurate Alphas, and most of the existing research focuses on deriving separate models for each of the stock, while only few authors consider the correlation among different stocks over the entire markets as a possible source of information. In other words, the Betas are often ignored. At the same time, since it is natural to assume that markets can have nontrivial correlation structure, it should be possible to extract useful information from group behavior of assets. Moreover, rather than the simplified linearity assumed in CAPM, the true Betas may exhibit more complicated nonlinear relationships between the stock and the market.

In this chapter, we propose a new deep learning framework that leverages both the underlying Alphas and (nonlinear) Betas. In particular, our approach innovates in the following aspects:

1) from model architecture perspective, we build a hybrid model that combines the advantages of both representation learning and deep networks. With representation learning, specifically, we use embedding in the deep learning model to derive implicit Betas, which we refer to as Stock2Vec, that not only gives us insight into the correlation structure among stocks, but also helps the model more effectively learn from the features thus improving prediction performance. In addition, with recent advances on deep learning architecture, in particular the temporal convolutional network, we further refine Alphas by letting the model automatically extract temporal information from raw historical series.

2) and from data source perspective, unlike many time series forecasting work that directly learn from raw series, we generate technical indicators features supplemented with external sources of information such as online news. Our approach differs from

most research built on machine learning models, since in addition to explicit hand-engineered temporal features, we use the raw series as augmented data input. More importantly, instead of training separate models on each single asset as in most stock market prediction research, we learn a global model on the available data over the entire market, so that the relationship among different stocks can be revealed.

The rest of this chapter is organized as follows. Section 2.2 lists several recent advances that are related to our method, in particular deep learning and its applications in forecasting as well as the representation learning. Section 2.3 illustrates the building blocks and details of our proposed framework, specifically, Stock2Vec embedding and the temporal convolutional network, as well as how our hybrid models are built. Our models are evaluated on the S&P 500 stock price data and benchmarked with several others, Section 2.4 describes the sample data, while the evaluation results as well as the interpretation of Stock2Vec are shown in Section 2.5. Finally, we conclude our findings and discuss the meaningful future work directions in Section 2.6.

2.2 Related Work

Recurrent neural network (RNN) and its variants of sequence to sequence (Seq2Seq) framework [25] have achieved great success in many sequential modeling tasks, such as machine translation [26], speech recognition [27], natural language processing [28], and extensions to autoregressive time series forecasting [29, 30] in recent years. However, RNNs can suffer from several major challenges. Due to its inherent temporal nature (i.e., the hidden state is propagated through time), the training cannot be parallelized. Moreover, trained with backpropagation through time (BPTT) [31], RNNs severely suffer from the problem of gradient vanishing (i.e., the backpropagated gradients easily approach zero in deep models with chain rule, and the model can hardly learn from the backpropagated loss through gradient descent), thus often cannot capture long time dependency [32]. More elaborate architectures of RNNs use gating mechanisms to alleviate the gradient vanishing problem,

with the long short-term memory (LSTM) [33] and its simplified variant, the gated recurrent unit (GRU) [34] being the two canonical architectures commonly used in practice.

Another approach, convolutional neural networks (CNNs) [96], can be easily parallelized, and recent advances effectively eliminate the vanishing gradient issue and hence help building very deep CNNs. These works include the residual network (ResNet) [97] and its variants such as highway network [98], DenseNet [99], etc. In the area of sequential modeling, 1D convolutional networks offered an alternative to RNNs for decades [100]. In recent years, [101] proposed WaveNet, a dilated causal convolutional network as an autoregressive generative model. Ever since, multiple research efforts have shown that with a few modifications, certain convolutional architectures achieve state-of-the-art performance in the fields of audio synthesis [101], language modeling [102], machine translation [103], action detection [104], and time series forecasting [105, 106]. In particular, [107] abandoned the gating mechnism in WaveNet and proposed temporal convolutional network (TCN). The authors benchmarked TCN with LSTM and GRU on several sequence modeling problems, and demonstrated that TCN exhibits substantially longer memory and achieves better performance.

Learning of the distributed representation has also been extensively studied [108, 109, 110] with arguably the most well-known application being word embedding [28, 84, 85] in language modeling. Word embedding maps words and phrases into distributed vectors in a semantic space in which words with similar meaning are closer, and some interesting relations among words can be revealed, such as

$$\text{King} - \text{Man} \approx \text{Queen} - \text{Woman}$$

$$\text{Paris} - \text{France} \approx \text{Rome} - \text{Italy}$$

as shown in [84]. Motivated by Word2Vec, the neural embedding methods have been extended to other domains in recent years. The authors of [111] obtained item embedding for recommendation systems through a collaborative filtering neural model, and called it

Item2Vec which is capable of inferring relations between items even when user information is not available. Similarly, [112] proposed Med2Vec that learns the medical concepts with the sequential order and co-occurrence of the concept codes within patients' visit, and showed higher prediction accuracy in clinical applications. In [113], the authors mapped every categorical features into "entity embedding" space for structured data and applied it successfully in a Kaggle competition, they also showcased the learned geometric embedding coincides with the real map surprisingly well when projected to 2D space.

In the field of stock prediction, the term "Stock2Vec" has already been used before. Specifically, [114] trained word embedding that specializes in sentiment analysis over the original Glove and Word2Vec language models, and using such a "Stock2Vec" embedding and a two-stream GRU model to generate the input data from financial news and stock prices, the authors predicted the price direction of S&P500 index. The authors of [115] proposed another "Stock2Vec" which also can be seen as a specialized Word2Vec, trained using the co-occurences matrix with the number of the news articles that mention both stocks as entries. Stock2Vec model proposed here differs from these homonymic approaches and has its distinct characteristics. First, our Stock2Vec is an entity embedding that represent the stock entities rather than a word embedding that denotes the stock names with language modeling. As the difference between entity embedding and word embedding may seem ambiguous, more importantly, instead of training the linguistic models with the co-occurrences of the words, our Stock2Vec embedding is trained directly as features through the overall predictive model, with the direct objective that minimizes prediction errors, thus illustrating the relationships among entities, while the others are actually fine-tuned subset of the original Word2Vec language model. Particularly inspiring for our work are the entity embedding [113] and the temporal convolutional network [107].

2.3 Methodology

2.3.1 Problem Formulation

We focus on predicting the future values of stock market assets given the past. More formally speaking, our input consists of a fully observable time series signals $\mathbf{y}_{1:T} = (y_1, \cdots, y_T)$ together with another related multivariate series $\mathbf{X}_{1:T} = (\mathbf{x}_1, \cdots, \mathbf{x}_T)$, in which $\mathbf{x}_t \in \mathbb{R}^{n-1}$, and n is the total number of series in the data. We aim at generating the corresponding target series $\hat{\mathbf{y}}_{T+1:T+h} = (\hat{y}_{T+1}, \cdots, \hat{y}_{T+h}) \in \mathbb{R}^h$ as the output, where $h \geq 1$ is the prediction horizon in the future. To achieve the goal, we will learn a sequence modeling network with parameters θ to obtain a nonlinear mapping from the input state space to the predicted series, i.e., $\hat{\mathbf{y}}_{T+1:T+h} = f(\mathbf{X}_{1:T}, \mathbf{y}_{1:T}|\theta)$, so that the distribution of our output could be as close to the true future values distribution as possible. That is, we wish to find $\min_\theta \mathbb{E}_{\mathbf{X},\mathbf{y}} \sum_{t=T+1}^{T+h} \mathrm{KL}\big(y_t||\hat{y}_t\big)$. Here, we use Kullback-Leibler (KL) divergence to measure the difference between the distributions of the true future values $\mathbf{y}_{T+1:T+h}$ and the predictions $\hat{\mathbf{y}}_{T+1:T+h}$. Note that our formulation can be easily extended to multivariate forecasting, in which the output and the corresponding input become multivariate series $\hat{\mathbf{y}}_{T+1:T+h} \in \mathbb{R}^{k\times h}$ and $\mathbf{y}_{1:T} \in \mathbb{R}^{k\times h}$, respectively, where k is the number of forecasting variables. The related input series is then $\mathbf{X}_{1:T} \in \mathbb{R}^{(n-k)\times T}$, and the overall objective becomes $\min_\theta \mathbb{E}_{\mathbf{X}_{1:T},\mathbf{y}_{1:T}} \sum_{t=T+1}^{T+h} \sum_{i=1}^{k} \mathrm{KL}\big(y_{i,t}||\hat{y}_{i,t}\big)$.

2.3.2 A Distributional Representation of Stocks: Stock2Vec

In machine learning fields, the categorical variables, if are not ordinal, are often one-hot encoded into a sparse representation. i.e.,

$$e : x \mapsto \delta(x, c),$$

where $\delta(x, c)$ is the Kronecker delta, in which each dimension represents a possible category. Let the number of categories of x be $|C|$, then $\delta(x, c)$ is a vector of length $|C|$ with the

only element set to 1 for $x = c$, and all others being zero. Note that although providing a convenient and simple way of representing categorical variables with numeric values for computation, one-hot encoding has various limitations. First of all, it does not place similar categories closer to one another in vector space, within one-hot encoded vectors, all categories are orthogonal to each other thus are totally uncorrelated, i.e., it cannot provide any information on similarity or dissimilarity between the categories. In addition, if $|C|$ is large, one-hot encoded vectors can be high-dimensional and often sparse, which means that the model has to involve a large number of parameters resulting in inefficient computations. For the cross-sectional data that we use for stock market, the number of total interactions between all pairs of stocks increases exponentially with the number of symbols we consider, for example, there are approximately $\binom{500}{2} \approx 0.1$ million pairwise interactions among the S&P 500 stocks. This number keeps growing exponentially as we add more features to describe the stock price performance. Therefore, trading on cross-sectional signals is remarkably difficult, and approximation methods are often applied.

We would like to overcome the abovementioned issue by reducing the dimensionality of the categorical variables. Common (linear) dimensionality reduction techniques include the principal component analysis (PCA), singular value decomposition (SVD), which operate by maintaining the first few eigen- or singular vectors corresponding to the largest few eigen- or singular values. PCA and SVD make efficient use of the statistics from the data and have been proven to be effective in various fields, yet they do not scale well for big matrices (e.g., the computational cost is $\mathcal{O}(n^3)$ for a $n \times n$ matrix), and they cannot adapt to minor changes in the data. In addition, the unsupervised transformation based on PCA or SVD do not use predictor variable, and hence it is possible that the derived components that serve as surrogate predictors provide no suitable relationship with the target. Moreover, since PCA and SVD utilize the first and second moments, they rely heavily on the assumption that the original data have approximate Gaussian distribution, which also limits the effectiveness of their usage.

Neural embedding is another approach to dimensionality reduction. Instead of computing and storing global information about the big dataset as in PCA or SVD, neural embedding learning provides us a way to learn iteratively on a supervised task directly. In this paper, we present a simple probabilistic method, Stock2Vec, that learns a dense distributional representation of stocks in a relatively lower dimensional space, and is able to capture the correlations and other more complicated relations between stock prices as well.

The idea is to design such a model whose parameters are the embeddings. We call a mapping $\phi : x \to \mathbb{R}^D$ a D-dimensional embedding of x, and $\phi(x)$ the embedded representation of x. Suppose the transformation is linear, then the embedding representation can be written as

$$z = Wx = \sum_c w_c \delta_{x,c}.$$

The linear embedding mapping is equivalent to an extra fully-connected layer of neural network without nonlinearity on top of the one-hot encoded input. Then each output of the extra linear layer is given as

$$z_d = \sum_c w_{c,d} \delta_{x,c} = w_d x,$$

where d stands for the index of embedding layer, and $w_{c,d}$ is the weight connecting the one-hot encoding layer to the embedding layer. The number of dimensions D for the embedding layer is a hyperparameter that can be tuned based experimental results, usually bounded between 1 and $|C|$. For our Stock2Vec, as we will introduce in Section 2.5, there are 503 different stocks, and we will map them into a 50-dimensional space.

The assumption of learning a distributional representation is that the series that have similar or opposite movement tend to correlated with each other, which is consistent with the assumption of CAPM, that the return of a stock is correlated with the market return, which in turn is determined by all stocks' returns in the market. We will learn the embeddings as part of the neural network for the target task of stock prediction. In order to learn the intrinsic relations among different stocks, we train the deep learning model on data of

all symbols over the market, where each datum maintains the features for its particular symbol's own properties, include the symbol itself as a categorical feature, with the target to predict next day's price. The training objective is to minimize the mean squared error of the predicted prices as usual.

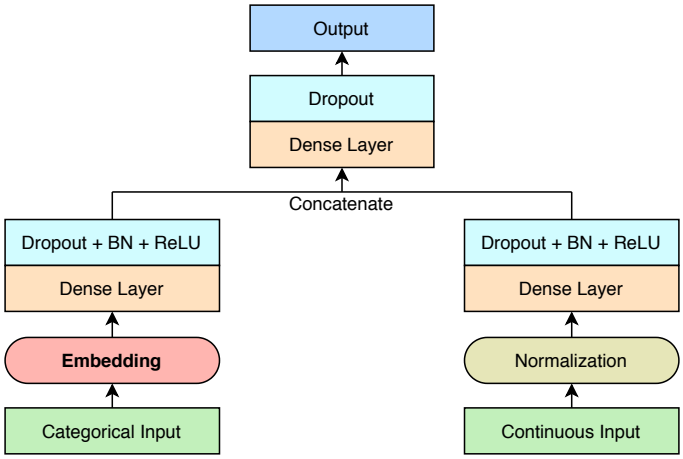


Figure 2.1: Model Architecture of Stock2Vec.

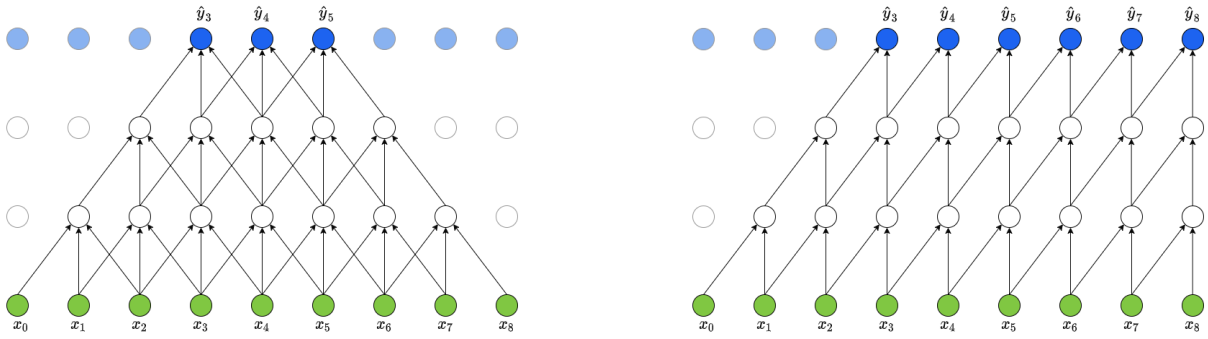
2.3.3 Temporal Convolutional Network

Sequential data often display long-term correlations and can be though of as a 1D grid with samples taken at regular time intervals. CNNs have shown success in time series applications, in which the 1D convolution is simply an operation of sliding dot products between the input vector and the kernel vector. However, we make several modifications to traditional 1D convolutions according to recent advances. The detailed building blocks of our temporal CNN components are illustrated in the following subsections.

Causal Convolutions

As we mentioned above, in a traditional 1D convolutional layer, the filters are slided across the input series. As a result, the output is related to the connection structure between the inputs before and after it. As shown in Figure 2.2(a), by applying a filter of width 2 without padding, the predicted outputs $\hat{x}_1, \cdots, \hat{x}_T$ are generated using the input series

$x_1, \cdots, x_T$. The most severe problem within this structure is that we use the future to predict the past, e.g., we have used x_2 to generate $\hat{x}_1$, which is not appropriate in time series analysis. To avoid the issue, causal convolutions are used, in which the output x_t is convoluted only with input data which are earlier and up to time t from the previous layer. We achieve this by explicitly zero padding of length $(kernel\_size - 1)$ at the beginning of input series, as a result, we actually have shifted the outputs for a number of time steps. In this way, the prediction at time t is only allowed to connect to historical information, i.e., in a causal structure, thus we have prohibited the future affecting the past and avoided information leakage. The resulted causal convolutions is visualized in Figure 2.2(b).



(a) standard (non-causal) (b) causal

Figure 2.2: Visualization of a stack of 1D convolutional layers, non-causal v.s. causal.

Dilated Convolutions

Time series often exhibits long-term autoregressive dependencies. With neural network models hence, we require for the receptive field of the output neuron to be large. That is, the output neuron should be connected with the neurons that receive the input data from many time steps in the past. A major disadvantage of the aforementioned basic causal convolution is that in order to have large receptive field, either very large sized filters are required, or those need to be stacked in many layers. With the former, the merit of CNN architecture is lost, and with the latter, the model can become computationally intractable. Following

[101], we adopted the dilated convolutions in our model instead, which is defined as

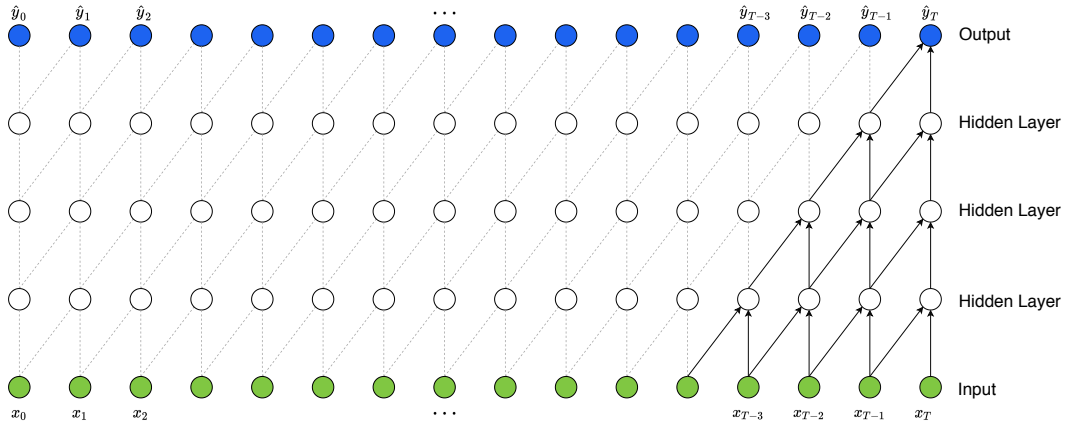
$$F(s) = (\mathbf{x} *_d f)(s) = \sum_{i=0}^{k-1} f(i) \cdot \mathbf{x}_{s-d\times i},$$

where $x \in \mathbb{R}^T$ is a 1-D series input, and $f : \{0, \cdots, k-1\} \to \mathbb{N}$ is a filter of size k, d is called the dilation rate, and $(s - d \times i)$ accounts for the direction of the past. In a dilated convolutional layer, filters are not convoluted with the inputs in a simple sequential manner, but instead skipping a fixed number (d) of inputs in between. By increasing the dilation rate multiplicatively as the layer depth (e.g., a common choice is $d = 2^j$ at depth j), we increase the receptive field exponentially, i.e., there are $2^{l-1}k$ input in the first layer that can affect the output in the l-th hidden layer. Figure 2.3 compares non-dilated and dilated causal convolutional layers.

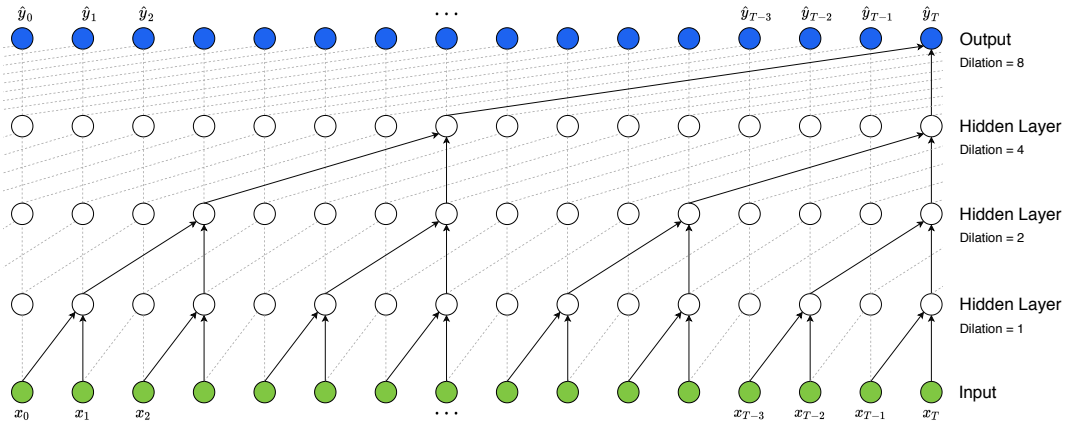
Residual Connections

In traditional neural networks, each layer feeds into the next. In a network with residual blocks, by utilizing skip connections, a layer may also short-cut to jump over several others. The use of residual network (ResNet) [97] has been proven to be very successful and become the standard way of building deep CNNs. The core idea of ResNet is the usage of shortcut connection which skips one or more layers and directly connects to later layers (which is the so-called identity mapping), in addition to the standard layer stacking connection $\mathcal{F}$. Figure 2.4 illustrates a residual block, which is the basic unit in ResNet. A residual block consists of the abovementioned two branches, and its output is then $g(\mathcal{F}(x) + x)$, where x denotes the input to the residual block, and g is the activation function.

By reusing activation from a previous layer until the adjacent layer learns its weights, CNNs can effectively avoid the problem of vanishing gradients. In our model, we implemented double-layer skips.



(a) Non-dilated

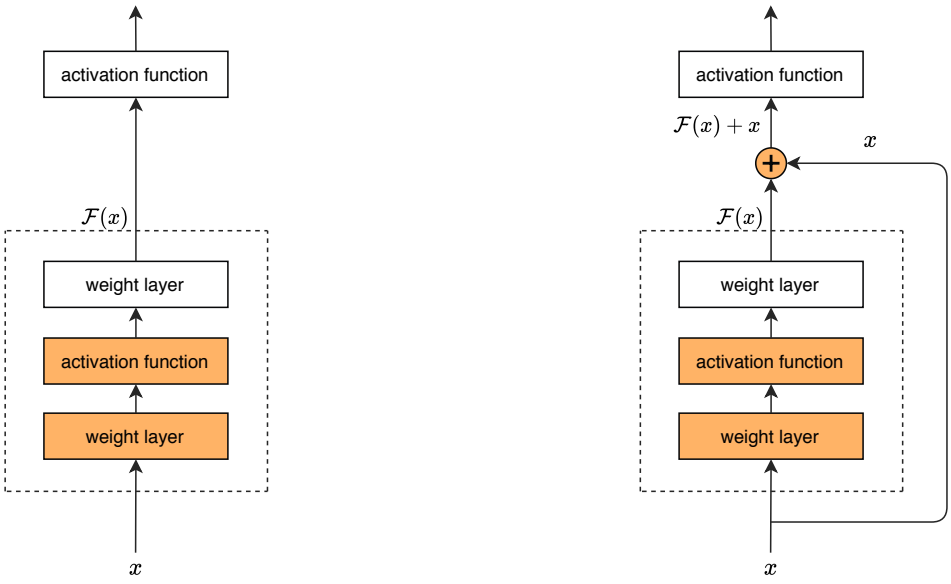


(b) Dilated

Figure 2.3: Visualization of a stack of causal convolutional layers, non-dilated v.s. dilated.

2.3.4 The Hybrid Model

Our overall prediction model is constructed as a hybrid, combining Stock2Vec embedding approach with an advanced implementation of the temporal convolutional network (TCN), schematically represented on Figure 2.5. Compared with Figure 2.1, it contains an additional TCN module. However, instead of producing the final prediction outputs of size 1, we let the TCN module output a vector as a feature map, as the size of the temporal module output can be easily and naturally controlled by the convolutional layer. As a result, it adds a new source of features, which contains information extracted from the temporal series. We then concatenate the temporal output vector with the learned Stock2Vec features. Note that the TCN module can be replaced by any other architecture that learns temporal patterns,



(a) A standard block

(b) A residual block

Figure 2.4: Comparison between a regular block and a residual block. In the latter, the convolution is short-circuited.

for example, LSTM-type network. Finally, a series of fully-connected layers (referred to as "head layers") are applied to the combined features producing the final prediction output. Implementation details are discussed in Section 2.5.1.

Note that in each TCN block, the convolutional layers use dropout in order to limit the influence that earlier data have on learning [42, 43]. It is then followed by a batch normalization layer [44]. The most widely used activation function, the rectified linear unit (ReLU) [22] is used after each layer except for the last one.

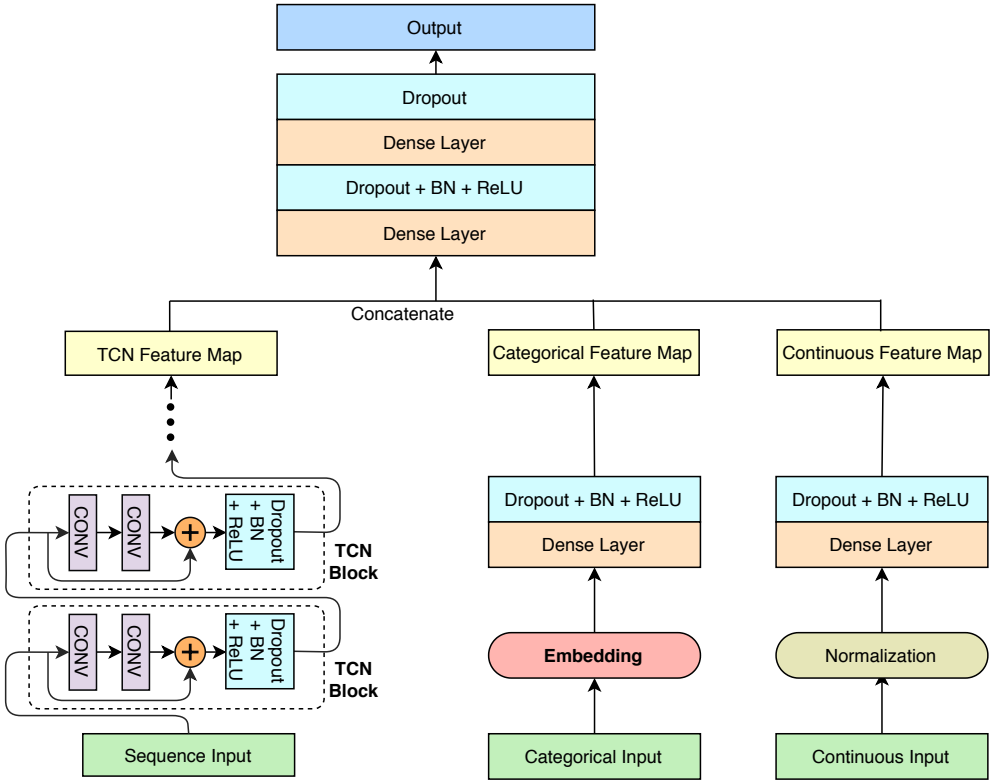


Figure 2.5: The full model architecture of hybrid TCN-Stock2Vec.

2.4 Data Specification

The case study is based on daily trading data for 485 assets listed on S&P 500 index, downloaded from Yahoo!finance for the period of 2015/01/01–2020/02/18 (out of 505 assets listed on `https://en.wikipedia.org/wiki/List_of_S%26P_500_companies`, two did not have data spanning the whole period). Following the literature, we use the next day's closing price as the target label for each asset, while the adjusted closing prices up until the current date can be used as inputs. In addition, we also use as augmented features the downloaded open/high/low prices and volume data for calculating some commonly used technical indicators that reflect price variation over time. In our study, eight commonly used technical indicators are selected, which are described in Table 2.1. As we discussed in Section 2.1, it has also been shown in the literature that assets' media exposure and the corresponding text sentiment are highly correlated with the stock prices. To account for

this, we acquired another set of features through the Quandl API. The database "FinSentS Web News Sentiment" (`https://www.quandl.com/databases/NS1/`) is used in this study. The queried dataset includes the daily number of news articles about each stock, as well as the sentiment score that measures the texts used in media, based on proprietary algorithms for web scraping and natural language processing.

We further extracted several date/time related variables for each entry to explicitly capture the seasonality, these features include month of year, day of month, day of week, etc. All of the above-mentioned features are all dynamic features that are time-dependent. In addition, we gathered a set of static features that are time-independent. Static covariates (e.g., the symbol name, sector and industry category, etc.) could assist the feature-based learner to capture series-specific information such as the scale level and trend for each series. The distinction between dynamic and static features is important for model architecture design, since it is unnecessary to process the static covariates by RNN cells or CNN convolution operations for capturing temporal relations (e.g., autocorrelation, trend, and seasonality, etc.).

Technical Indicators	**Category**	**Description**
Moving average convergence or divergence (MACD)	Trend	Reveals price change in strength, direction and trend duration
Parabolic Stop And Reverse (PSAR)	Trend	Indicates whether the current trend is to continue or to reverse
Bollinger Bands (BB®)	Volatility	Forms a range of prices for trading decisions
Stochastic Oscillator (SO)	Momentum	Indicates turning points by comparing the price to its range
Rate Of Change (ROC)	Momentum	Measures the percent change of the prices
On-Balance Volume (OBV)	Volume	Accumulates volume on price direction to confirm price moves
Force Index (FI)	Volume	Measures the amount of strength behind price move

Table 2.1: Description of technical indicators used in this study.

Note that the features can also be split into categorical and continuous. Each of the categorical features is mapped to dense numeric vectors via embedding, in particular, the vectors embedded from the stock name as a categorical feature are called Stock2Vec. We scale all continuous features (as well as next day's price as the target) to between 0 and 1, since it is widely accepted that neural networks are hard to train and are sensitive to input scale [116, 44], while some alternative approaches, e.g., decision trees, are scale-invariant [117]. It is important to note that we performed scaling separately on each asset, i.e., linear

transformation is performed so that the lowest and highest price for asset A over the training period is 0 and 1 respectively. Also note scaling statistics are obtained with the training set only, which prevents leakage of information from the test set, avoiding introduction of look-ahead bias.

As a tentative illustration, Figure 2.6 shows the most important 20 features for predicting next day's stock price, according to the XGBoost model we trained for benchmarking.

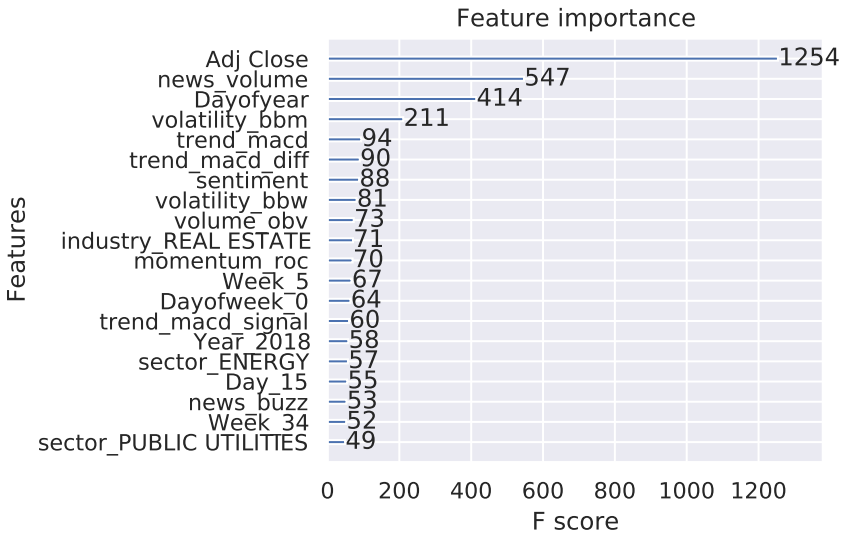


Figure 2.6: Feature importance plot of XGBoost model.

In our experiments, the data are split into training, validation and test sets. The last 126 trading days of data are used as the test set, cover the period from 2019/08/16 to 2020/02/18, and include 61000 samples. The rest data are used for training the model, in which the last 126 trading days, from 2019/02/15 to 2019/08/15, are used as validation set, while the first 499336 samples, cover the period from 2015/01/02 to 2019/02/14, form the training set. Table 2.2 provides a summary of the datasets we used in this research.

Table 2.2: Dataset summary.

	Training set	Validation set	Test set
Starting date	2015/01/02	2019/02/15	2019/08/16
End date	2019/02/14	2019/08/15	2020/02/18
Sample size	499336	61075	61000

2.5 Experimental Results and Discussions

2.5.1 Benchmark Models, Hyperparameters and Optimization Strategy

In the computational experiments below we compare performance of seven models.. Two models are based on time series analysis only (TS-TCN and TS-LSTM), two use static feature only (random forest [18] and XGBoost [17]), pure Stock2Vec model and finally, two versions of the proposed hybrid model (LSTM-Stock2Vec and TCN-Stock2Vec). This way we can evaluate the effect of different model architectures and data features. Specifically, we are interested in evaluating whether employing feature embedding leads to improvement (Stock2Vec vs random forest and XGBoost) and whether a further improvement can be achieved by incorporating time-series data in the hybrid models.

Random forest and XGBoost are ensemble models that deploy enhanced bagging and gradient boosting, respectively. We pick these two models since both have shown powerful predicting ability and achieved state-of-the-art performance in various fields. Both are tree-based models that are invariant to scales and perform split on one-hot encoded categorical inputs, which is suitable for comparison with embeddings in our Stock2Vec models. We built 100 bagging/boosting trees for these two models. LSTM and TCN models are constructed based on pure time series data, i.e., the inputs and outputs are single series, without any other feature as augmented series. In later context, we call these two models TS-LSTM and TS-TCN, respectively. The Stock2Vec model is a fully-connected neural network with embedding layers for all categorical features, it has the exactly same inputs as XGBoost and random forest. As we introduced in Section 2.3.4, our hybrid model combines the Stock2Vec model with an extra TCN module to learn the temporal effects. And for comparison purpose, we also evaluated the hybrid model with LSTM as the temporal module. We call them TCN-Stock2Vec and LSTM-Stock2Vec correspondingly.

Our deep learning models are implemented in PyTorch [118]. In Stock2Vec, the embedding sizes are set to be half of the original number of categories, thresholded by 50 (i.e., the

maximum dimension of embedding output is 50). These are just heuristics as there is no common standard for choosing the embedding sizes. We concatenate the continuous input with the outputs from embedding layers, followed by two layers of fully-connected layers, with sizes of 1024 and 512, respectively. The dropout rates are set to 0.001 and 0.01 for the two hidden layers correspondingly.

For the RNN module, we implement two-layer stacked LSTM, i.e., in each LSTM cell (that denotes a single time step), there are two LSTM layers sequentially connected, and each layer consists of 50 hidden units. We need an extra fully-connected layer to control the output size for the temporal module, depending on whether to obtain the final prediction as in TS-LSTM (with output size to be 1), or a temporal feature map as in LSTM-Stock2Vec. We set the size of temporal feature map to be 30 in order to compress the information for both LSTM-Stock2Vec and TCN-Stock2Vec. In TCN, we use another convolutional layer to achieve the same effect. To implement the TCN module, we build a 16-layer dilated causal CNN as the component that focuses on capturing the autoregressive temporal relations from the series own history. Each layer contains 16 filters, and each filter has a width of 2. Every two consecutive convolutional layers form a residual block after which the previous inputs are added to the flow. The dilation rate increases exponentially along every stacked residual blocks, i.e., to be $1, 2, 4, 8, \cdots, 128$, which allows our TCN component to capture the autoregressive relation for more than half a year (there are 252 trading days in a year). Again, dropout (with probability 0.01), batch normalization layer and ReLU activation are used for each TCN block.

The MSE loss is used for all models. The deep learning models were trained using stochastic gradient descent (SGD), with batch size of 128. In particular, the Adam optimizer [119] with initial learning rate of 10^{-4} was used to train TS-TCN and TS-LSTM. To train Stock2Vec, we deployed the super-convergence scheme as in [120] and used cyclical learning rate over every 3 epochs, with a maximum value of 10^{-3}. In the two hybrid models, while the

weights of the head layers were randomly initialized as usual, we loaded the weights from pre-trained Stock2Vec and TS-TCN/TS-LSTM for the corresponding modules. By doing this, we have applied transfer learning scheme [121, 122, 123] and wish the transferred modules have the ability to effectively process features from the beginning. The head layers were trained for 2 cycles (each contains 2 epochs) with maximum learning rate of 3×10^{-4} while the transferred modules were frozen. After this convergence, the entire network was fine-tuned for 10 epochs by standard Adam optimizer with learning rate of 10^{-5}, during which an early stopping paradigm [41] was applied to retrieve the model with smallest validation error. We select the hyperparemeters based upon the model performance on the validation set.

2.5.2 Performance Evaluation Metrics

To evaluate the performance of our forecasting model, three commonly used evaluation criteria are used in this study: (a) the root mean square error (RMSE), (b) the mean absolute error (MAE), (c) the mean absolute percentage error (MAPE), (d) the root mean square percentage error (RMSPE):

$$\text{RMSE} = \sqrt{\frac{1}{H}\sum_{t=1}^{H}(y_t - \hat{y}_t)^2} \tag{2.1}$$

$$\text{MAE} = \frac{1}{H}\sum_{t=1}^{H}\left|y_t - \hat{y}_t\right| \tag{2.2}$$

$$\text{MAPE} = \frac{1}{H}\sum_{t=1}^{H}\left|\frac{y_t - \hat{y}_t}{y_t}\right| \times 100 \tag{2.3}$$

$$\text{RMSPE} = \sqrt{\frac{1}{H}\sum_{t=1}^{H}\left|\frac{y_t - \hat{y}_t}{y_t}\right|^2} \times 100 \tag{2.4}$$

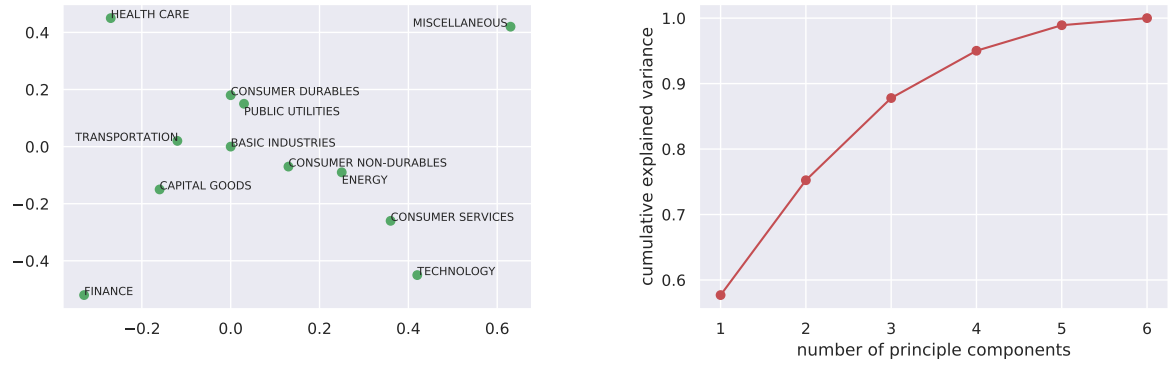
where y_t is the actual target value for the t-th observation, $\hat{y}_t$ is the predicted value for the corresponding target, and H is the forecast horizon.

The RMSE is the most popular measure for the error rate of regression models, as $n \to \infty$, it converges to the standard deviation of the theoretical prediction error. However, the quadratic error may not be an appropriate evaluation criterion for all prediction problems, especially in the presence of large outliers. In addition, the RMSE depends on scales, and is also sensitive to outliers. The MAE considers the absolute deviation as the loss and is a more "robust" measure for prediction, since the absolute error is more sensitive to small deviations and much less sensitive to large ones than the squared error. However, since the training process for many learning models are based on squared loss function, the MAE could be (logically) inconsistent to the model optimization selection criteria. The MAE is also scale-dependent, thus not suitable to compare prediction accuracy across different variables or time ranges. In order to achieve scale independence, the MAPE measures the error proportional to the target value, while instead of using absolute values, the RMSPE can be seen as the root mean squared version of MAPE. The MAPE and RMSPE however, are extremely unstable when the actual value is small (consider the case when the denominator or close to 0). We will consider all four measures mentioned here to have a more complete view of the performance of the models considering the limitations of each performance measure. In addtion, we will compare the running time as an additional evaluation criterion.

2.5.3 Stock2Vec: Analysis of Embeddings

As we introduced in Section 2.3, the main goal of training Stock2Vec model is to learn the intrinsic relationships among stocks, where similar stocks are close to each other in the embedding space, so that we can deploy the interactions from cross-sectional data, or more specifically, the market information, to make better predictions. To show this is the case, we extract the weights of the embedding layers from the trained Stock2Vec model, map the weights down to two-dimensional space with a manifold by using PCA, and visualize the entities to look at how the embedding spaces look like. Note that besides Stock2Vec, we also learned embeddings for other categorical features.

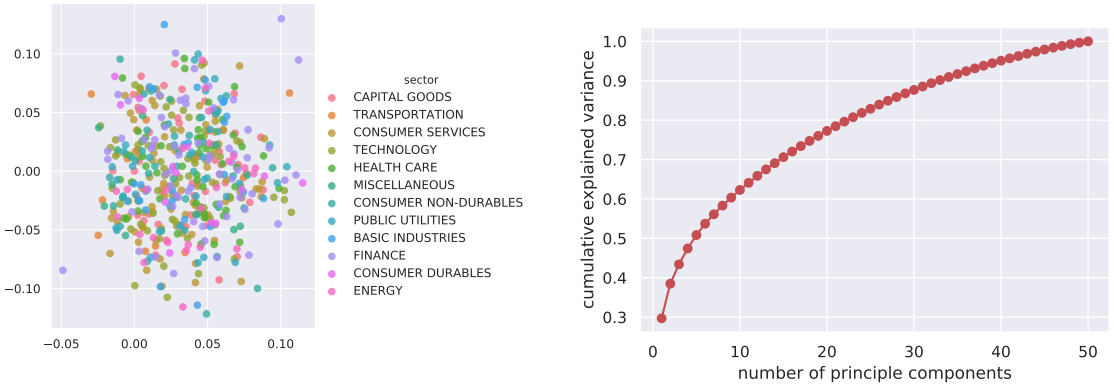
Figure 2.7(a) shows the first two principal components of the sectors. Note that here the first two components account for close to 75% of variance. We can generally observe that *Health Care, Technology/Consumer Services* and *Finance* occupy the opposite corners of the plot, i.e., represent unique sectors most dissimilar from one another. On the other hand a collection of more traditional sectors: *Public Utilities, Energy, Consumer Durables and Non-Durables, Basic Industries* generally are grouped closer together. The plot, then, allows for a natural interpretation which is in accordance with our intuition, indicating that the learned embedding can be expected to be reasonable.



(a) Visualization of learned embeddings for sectors, projected to 2-D spaces using PCA.

(b) The cumulative explained variance ratio for each of the principal components

Figure 2.7: PCA on the learned embeddings for Sectors



(a) Visualization of Stock2Vec (colored by sectors), projected to 2-D spaces using PCA.

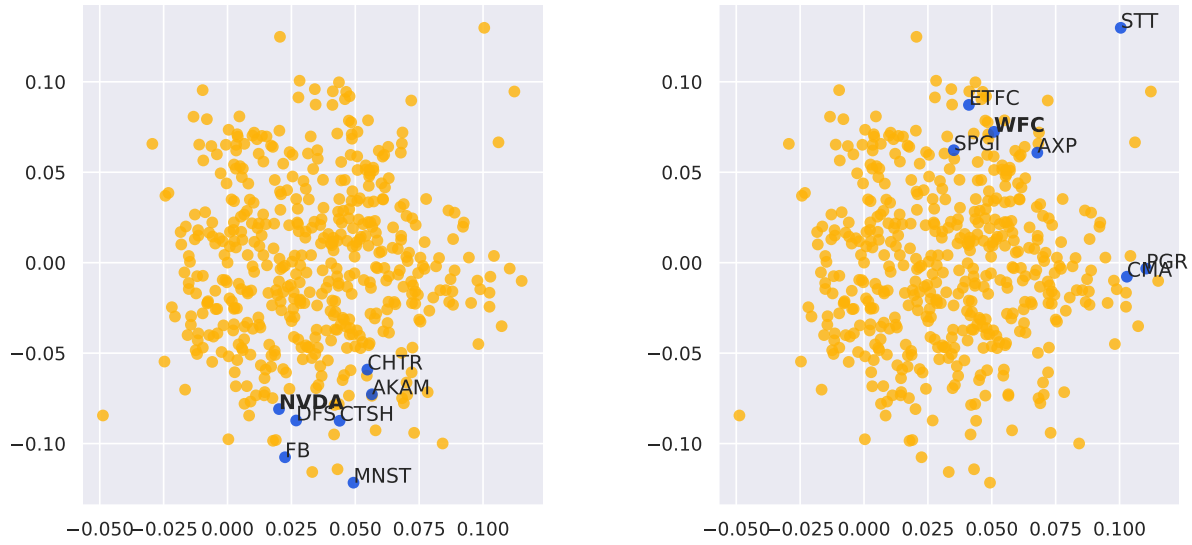
(b) The cumulative explained variance ratio for each of the principal components

Figure 2.8: PCA on the learned Stock2Vec embeddings

Similarly, from the trained Stock2Vec embeddings, we can obtain a 50-dimensional vector for each separate stock. We simialrly visualize the learned Stock2Vec with PCA in Figure 2.8(a), and color each stock by the sector it belongs to. It is important to note that in this case, the first two components of PCA only account for less than 40% of variance. In other words, in this case, the plotted groupings do not represent the learned information as well as in the previous case. Indeed, when viewed all together, individual assets do not exhibit readily discernible patterns. This is not necessarily an indicator of deficiency of the learned embedding, and instead suggests that two dimensions are not sufficient in this case. However, lots of useful insight can be gained from the distributed representations, for instance, we could consider the similarities between stocks in the learned vector space is an example of these benefits as we will show below,

To reveal some additional insights from the similarity distance, we sort the pairwise cosine distance (in the embedded space) between the stocks in the ascending order. In Figure 2.9a, we plot the ticker "NVDA" (Nvidia) as well as its six nearest neighbors in the embedding space. The six companies that are closest to Nvidia, according to the embeddings of learned weights, are either of the same type (technology companies) with Nvidia: Facebook, Akamai, Cognizant Tech Solutions, Charte; or fast growing during the past ten years (was the case for Nvidia during the tested period): Monster, Discover Bank. Similarly, we plot the ticker of Wells Fargo ("WFC") and its 6 nearest neighbors in Figure 2.9b, all of which are either banks or companies that provide other financial services. These observations are yet another indicator that Stock2Vec can be expected to learn some useful information, and indeed is capable of coupling together insights from a number of unrelated sources.

The following points must be noted here. First, most of the nearest neighbors are not the closest points in the two-dimensional plots due to the imprecision of mapping into two-dimensions. Secondly, although the nearest neighbors are meaningful for many companies as the results either are in the same sector (or industry), or present similar stock price trend in the last a few years, this insight does not hold true for all companies, or the interpretation



(a) Nearest neighbors of NVDA, which are: 1). 'MNST', Monster, fast growing; 2) 'FB', Facebook, IT; 3)'DFS', Discover Bank, fast growing; 4) 'AKAM', Akamai, IT; 5) 'CTSH', Cognizant Tech Solutions, IT; 6) 'CHTR', Charter, communication services.

(b) Nearest neighbors of WFC, which are: 1) 'ETFC': E-Trader, financial; 2) 'STT': State Street Corp., bank; 3) 'CMA': Comerica, bank; 4) 'AXP': Amex, financial; 5) 'PGR': Progressive Insurance, financial; 6) 'SPGI', S&P Global, Inc., financial & data

Figure 2.9: Nearest neighbors of Stock2Vec based on similarity between stocks.

can be hard to discern. For example, the nearest neighbors of Amazon.com (AMZN) include transportation and energy companies (perhaps due to its heavy reliance on these industries for its operation) as well as technology companies. Finally, note that there exist many other visualization techniques for projection of high dimensional vectors onto 2D spaces that could be used here instead of PCA, for example, t-SNE [124] or UMAP [125]. However, neither provided visual improvement of the grouping effect over Figure 2.8(a) and hence we do not present those results here.

Based on the above observations, Stock2Vec provides several benefits: 1) reducing the dimensionality of categorical feature space, thus the computational performance is improved with smaller number of parameter, 2) mapping the sparse high-dimensional one-hot encoded vectors onto dense distributional vector space (with lower dimensionality), as a result, similar categories are learned to be placed closer to one another in the embedding space, unlike in

one-hot encoding vector space where every pairs of categories yield the same distance and are orthogonal to each other. Therefore, the outputs of the embedding layers could be served as more meaningful features, for later layers of neural networks to achieve more effective learning. Not only that, the meaningful embeddings can be used for visualization, provides us more interpretability of the deep learning models.

2.5.4 Prediction Results

Table 2.3 and Figure 2.10 report the overall average (over the individual assets) forecasting performance of the out-of-sample period from 2019-08-16 to 2020-02-14. We observe that TS-LSTM and TS-TCN perform worst. We can conlude that this is because these two models only consider the target series and ignore all other features. TCN outperforms LSTM, probably since it is capable of extracting temporal patterns over long history without more effectively gradient vanishing problem. Moreover, the training speed of our 18-layer TCN is about five times faster than that of LSTM per iteration (aka batch) with GPU, and the overall training speed (given all overhead included) is also around two to three times faster. With learning from all the features, the random forest and XGBoost models perform better than purely timeseries-based TS-LSTM and TS-TCN, with the XGBoost predictions are slightly better than that from random forest. This demonstrates the usefulness of our data source, especially the external information combined into the inputs. We can then observe that despite having the same input as random forest and XGBoost, the proposed our Stock2Vec model further improves accuracy of the predictions, as the RMSE, MAE, MAPE and RMSPE decrease by about 36%, 38%, 41% and 43% over the XGBoost predictions, respectively. This indicates that the use of deep learning models, in particular the Stock2Vec embedding improves the predictions, by more effectively learning from the features over the tree-based ensemble models. With integration of temporal modules, there is again a significant improvement of performance in terms of prediction accuracy. The two hybrid models LSTM-Stock2Vec and TCN-Stock2Vec not only learn from features we give explicitly, but

also employ either a hidden state or a convolutional temporal feature mapping to implicitly learn relevant information from historical data. Our TCN-Stock2Vec achieves the best performance across all models, as the RMSE and MAE decreases by about 25%, while the MAPE decreases by 20% and the RMSPE decreases by 14%, comparing with Stock2Vec without the temporal module.

Table 2.3: Average performance comparison.

	RMSE	MAE	MAPE(%)	RMSPE (%)
TS-LSTM	6.35	2.36	1.62	2.07
TS-TCN	5.79	2.15	1.50	1.96
Random Forest	4.86	1.67	1.31	1.92
XGBoost	4.57	1.66	1.28	1.83
Stock2Vec	2.94	1.04	0.76	1.05
LSTM-Stock2Vec	2.57	0.85	0.68	1.04
TCN-Stock2Vec	**2.22**	**0.78**	**0.61**	**0.90**

Figure 2.10 shows the boxplots of the prediction errors of different approaches, from which we can see our proposed models achieve smaller absolute prediction errors in terms of not only the mean also the variance, which indicates more robust forecast. The median absolute prediction errors (and the interquartile range, i.e., IQR) of our TS-TCN model is around 1.01 (1.86), while they are around 0.74 (1.39), 0.45 (0.87), and 0.36 (0.66) for XGBoost, Stock2Vec and TCN-Stock2Vec, respectively.

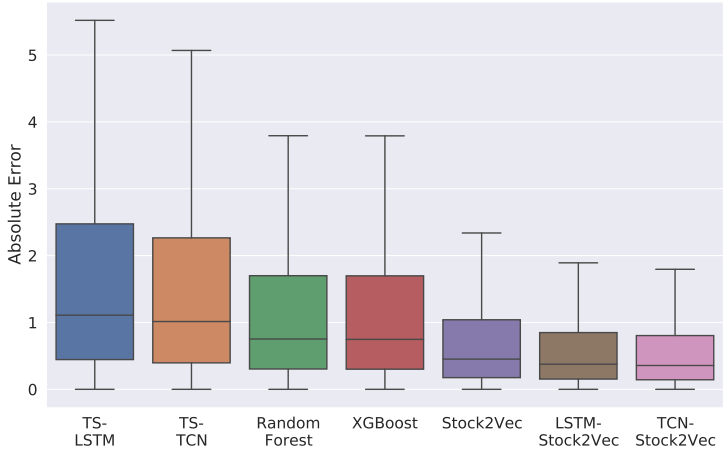


Figure 2.10: Boxplot comparison of the absolute prediction errors.

Similarly, we aggregate the metrics on the sector level, and calculate the average performance within each sector. We report the RMSE, MAE, MAPE, and RMSPE in Tables 2.4, 2.5, 2.6, and 2.7, respectively, from which we can see again our Stock2Vec performs better than the two tree-ensemble models for all sectors, and adding the temporal module would further improve the forecasting accuracy. TCN-Stock2Vec achieves the best RMSE, MAE, MAPE and RMSPE in all sectors with one exception. Better performance on different aggregated levels demonstrates the power of our proposed models.

We further showcase the predicted results of 20 symbols to gauge the forecasting performance of our model under a wide range of industries, volatilities, growth patterns and other general conditions. The stocks have been chosen to evaluate how the proposed methodologies would perform under different circumstances. For instance, Amazon's (AMZN) stock was consistently increasing in price across the analysis period, while the stock price of Verizon (VZ) was very stable, and Chevron's stock (CVX) had both periods of growth and decline. In addition, these 20 stocks captured several industries: (a) retail (e.g., Walmart), (b) restaurants (e.g., McDonald's), (c) finance and banks (e.g., JPMorgan Chase and Goldman Sachs), (d) energy and oil & gas (e.g., Chevron), (e) techonology (e.g., Facebook), (f) communications (e.g., Verizon), etc. Table 2.8, 2.9, 2.10, 2.11 show the out-of-sample RMSE, MAE, MAPE and RMAPE, respectively, from the predictions given by the five models we discussed above. Again, Stock2Vec generally performs better than random forest and XGBoost, and the two hybrid models have quite similar performance which is significantly better than that of others. While there also exist a few stocks on which LSTM-Stock2Vec or even Stock2Vec without temporal module produce most accurate predictions, for most of the stocks, TCN-Stock2Vec model performs the best. This demonstrates our models generalize well to most symbols.

Furthermore, we plot the prediction pattern of the competing models for the abovementioned stocks on the test set in 2.C, compared to the actual daily prices. We observe that the random forest and XGBoost models predict up-and-downs with a lag for most of the time,

as the current price plays too much a role as a predictor, probably mainly due to the correct scaling reason. And there occasionally exist several flat predictions over a period for some stocks (see approximately 2019/09 in Figure 2.15, 2020/01 in Figure 2.18, and 2019/12 in Figure 2.30), which is a typical effect of tree-based methods, indicates insufficient splitting and underfitting despite so many ensemble trees were used. With entity embeddings, our Stock2Vec model can learn from the features much more effectively, its predictions coincide with the actual up-and downs much more accurately. Although it overestimates the volatility by exaggerating the amplitude as well as the frequency of oscillations, the overall prediction errors are getting smaller than the two tree-ensemble models. And our LSTM-Stock2Vec and TCN-Stock2Vec models further benefit from the temporal learning modules by automatically capturing the historical characteristics from time series data, especially the nonlinear trend and complex seasonality that are difficult to be captured by hand-engineered features such as technical indicators, as well as the common temporal factors that are shared among all series across the whole market. As a result, with the ability to extract the autoregressive dependencies over long term both within and across series from historical data, the predictions from these two models alleviate wild oscillations, and are much more close to the actual prices, while still correctly predict the up-and-downs for most of the time with effective learning from input features.

2.6 Concluded Remarks and Future Work

Our argument that implicitly learning Alphas and Betas upon cross-sectional data from CAPM perspective is novel, however, it is more of an insight rather than systematic analysis. In this paper, we built a global hybrid deep learning models to forecast the S&P stock prices. We applied the state-of-the-art 1-D dilated causal convolutional layers (TCN) to extract the temporal features from the historical information, which helps us to refine learning of the Alphas. In order to integrate the Beta information into the model, we learn a single model that learns from the data over the whole market, and applied entity embeddings for the

categorical features, in particular, we obtained the Stock2Vec that reveals the relationship among stocks in the market, our model can be seen as supervised dimension reduction method in that point of view. The experimental results show our models improve the forecasting performance. Although not demonstrated in this work, learning a global model from the data over the entire market can give us an additional benefit that it can handle the cold-start problem, in which some series may contain very little data (i.e., many missing values), our model has the ability to infer the historical information with the structure learned from other series as well as the correlation between the cold-start series and the market. It might not be accurate, but is much informative than that learned from little data in the single series.

There are several other directions that we can dive deeper as the future work. First of all, the stock prices are heavily affected by external information, combining extensive crowd-sourcing, social media and financial news data may facilitate a better understanding of collective human behavior on the market, which could help the effective decision making for investors. These data can be obtained from the internet, we could expand the data source and combine their influence in the model as extra features. In addition, although we have shown that the convolutional layers have several advantages over the most widely used recurrent neural network layers for time series, the temporal learning layers in our model could be replaced by any other type, for instance, the recent advances of attention models could be a good candidate. Also, more sophisticated models can be adopted to build Stock2Vec, by keeping the goal in mind that we aim at learning the implicit intrinsic relationship between stock series. In addition, learning the relationship over the market would be helpful for us to build portfolio aiming at maximizing the investment gain, e.g., by using standard Markowitz portfolio optimization to find the positions. In that case, simulation of trading in the market should provide us more realistic and robust performance evaluation than those aggregated levels we reported above. Liquidity and market impacts

can be taken into account in the simulation, and we can use Profit & Loss (P&L) and the Sharpe ratio as the evaluation metrics.

2.A Sector Level Performance Comparison

Table 2.4: Sector level RMSE comparison

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
Basic Industries	1.70	1.61	1.06	0.85	**0.76**
Capital Goods	11.46	10.30	6.25	6.01	**5.10**
Consumer Durables	1.78	1.67	0.99	0.93	**0.83**
Consumer Non-Durables	1.57	1.55	0.98	0.87	**0.75**
Consumer Services	4.75	4.69	3.34	2.76	**2.30**
Energy	1.50	1.44	0.76	0.76	**0.67**
Finance	2.08	2.06	1.39	1.05	**1.00**
HealthCare	3.44	3.37	1.95	1.98	**1.60**
Miscellaneous	8.23	7.96	5.22	4.14	**3.73**
Public Utilities	0.94	0.95	0.64	**0.52**	0.52
Technology	4.20	4.23	2.91	1.90	**1.94**
Transportation	2.00	1.90	1.15	1.03	**0.88**

Table 2.5: Sector level MAE comparison

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
Basic Industries	1.06	1.03	0.64	0.52	**0.49**
Capital Goods	3.13	3.07	1.93	1.57	**1.47**
Consumer Durables	1.21	1.18	0.71	0.63	**0.57**
Consumer Non-Durables	0.96	0.93	0.57	0.52	**0.45**
Consumer Services	1.83	1.84	1.19	0.98	**0.88**
Energy	0.98	0.95	0.50	0.51	**0.45**
Finance	1.19	1.17	0.79	0.55	**0.54**
HealthCare	1.99	1.96	1.15	1.10	**0.92**
Miscellaneous	3.18	3.18	2.08	1.56	**1.44**
Public Utilities	0.63	0.64	0.44	**0.33**	0.34
Technology	1.95	1.98	1.26	0.92	**0.91**
Transportation	1.26	1.23	0.74	0.65	**0.56**

Table 2.6: Sector level MAPE (%) comparison

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
Basic Industries	1.34	1.31	0.74	0.65	**0.61**
Capital Goods	1.21	1.24	0.76	0.59	**0.56**
Consumer Durables	1.30	1.26	0.73	0.68	**0.60**
Consumer Non-Durables	1.48	1.32	0.76	0.85	**0.65**
Consumer Services	1.24	1.23	0.71	0.66	**0.59**
Energy	2.04	1.88	0.97	1.08	**0.92**
Finance	1.18	1.16	0.74	**0.53**	0.53
HealthCare	1.43	1.35	0.79	0.79	**0.65**
Miscellaneous	1.23	1.23	0.81	0.66	**0.60**
Public Utilities	0.88	0.90	0.57	0.49	**0.47**
Technology	1.44	1.43	0.83	0.68	**0.66**
Transportation	1.26	1.23	0.71	0.66	**0.57**

Table 2.7: Sector level RMSPE (%) comparison

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
Basic Industries	1.86	1.80	0.98	0.91	**0.83**
Capital Goods	1.63	1.65	1.01	0.83	**0.75**
Consumer Durables	1.79	1.68	0.96	0.95	**0.81**
Consumer Non-Durables	2.41	2.02	1.13	1.37	**1.01**
Consumer Services	1.88	1.82	0.99	1.07	**0.91**
Energy	2.89	2.66	1.29	1.51	**1.25**
Finance	1.60	1.56	1.00	0.78	**0.72**
HealthCare	2.17	2.00	1.15	1.24	**0.99**
Miscellaneous	1.66	1.63	1.05	0.95	**0.81**
Public Utilities	1.25	1.23	0.74	0.71	**0.64**
Technology	2.09	2.00	1.13	1.04	**0.95**
Transportation	1.76	1.68	0.98	0.95	**0.80**

2.B Performance comparison of different models for the one-day ahead forecasting on different symbols

Table 2.8: RMSE comparison of different models for the one-day ahead forecasting on different symbols

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
AAPL (Apple)	4.71	4.52	2.86	2.16	**1.81**
AFL (Aflac)	0.59	0.62	0.46	**0.31**	**0.27**
AMZN (Amazon.com)	29.91	28.47	23.80	17.73	**14.45**
BA (Boeing)	6.00	6.44	3.98	3.83	**3.49**
CVX (Chevron)	1.42	1.62	1.03	0.75	**0.65**
DAL (Delta Air Lines)	0.79	0.77	0.48	0.40	**0.32**
DIS (Walt Disney)	1.95	1.91	1.17	1.10	**0.92**
FB (Facebook)	3.51	5.54	2.15	1.72	**1.44**
GE (General Electric)	0.39	0.30	**0.14**	0.29	0.18
GM (General Motors)	0.58	0.57	0.30	0.30	**0.28**
GS (Goldman Sachs Group)	3.11	3.00	1.86	**1.27**	1.31
JNJ (Johnson & Johnson)	1.80	1.49	1.00	0.93	**0.80**
JPM (JPMorgan Chase)	1.72	1.63	1.59	**0.66**	0.68
MAR (Marriott Int'l)	2.02	2.02	1.52	**0.89**	1.07
KO (Coca-Cola)	0.49	0.50	0.32	0.26	**0.25**
MCD (McDonald's)	2.67	2.50	1.51	1.26	**1.16**
NKE (Nike)	1.27	1.23	1.01	**0.61**	0.62
PG (Procter & Gamble)	1.43	1.35	0.91	0.70	**0.61**
VZ (Verizon Communications)	0.54	0.55	0.46	0.29	**0.26**
WMT (Walmart)	1.34	1.43	1.06	0.55	**0.50**

Table 2.9: MAE comparison of different models for the one-day ahead forecasting on different symbols

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
AAPL (Apple)	3.63	3.56	2.15	1.72	**1.42**
AFL (Aflac)	0.45	0.44	0.35	**0.20**	0.21
AMZN (Amazon.com)	22.19	21.36	17.87	11.53	**10.29**
BA (Boeing)	4.59	5.10	2.87	2.87	**2.74**
CVX (Chevron)	1.07	1.22	0.75	0.57	**0.50**
DAL (Delta Air Lines)	0.59	0.58	0.36	0.29	**0.24**
DIS (Walt Disney)	1.37	1.40	0.87	0.77	**0.67**
FB (Facebook)	2.54	3.80	1.65	1.16	**1.06**
GE (General Electric)	0.30	0.22	**0.11**	0.25	0.15
GM (General Motors)	0.44	0.44	0.23	0.23	**0.22**
GS (Goldman Sachs Group)	2.48	2.37	1.31	**1.01**	1.05
JNJ (Johnson & Johnson)	1.21	1.04	0.72	0.64	**0.59**
JPM (JPMorgan Chase)	1.34	1.23	1.17	**0.51**	0.52
MAR (Marriott Int'l)	1.63	1.66	1.13	**0.65**	0.87
KO (Coca-Cola)	0.39	0.37	0.25	0.19	**0.19**
MCD (McDonald's)	1.99	1.96	1.26	0.89	**0.89**
NKE (Nike)	0.97	0.98	0.77	**0.46**	0.49
PG (Procter & Gamble)	1.14	1.03	0.70	0.52	**0.48**
VZ (Verizon Communications)	0.43	0.42	0.36	0.22	**0.20**
WMT (Walmart)	1.02	1.10	0.87	**0.41**	0.41

Table 2.10: MAPE (%) comparison of different models for the one-day ahead forecasting on different symbols

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
AAPL (Apple)	1.43	1.39	0.80	0.68	**0.54**
AFL (Aflac)	0.88	0.86	0.66	**0.39**	0.39
AMZN (Amazon.com)	1.21	1.17	0.97	0.63	**0.56**
BA (Boeing)	1.33	1.47	0.82	0.83	**0.80**
CVX (Chevron)	0.94	1.06	0.65	0.50	**0.43**
DAL (Delta Air Lines)	1.03	1.02	0.63	0.51	**0.43**
DIS (Walt Disney)	0.99	1.01	0.61	0.55	**0.48**
FB (Facebook)	1.29	1.92	0.82	0.59	**0.54**
GE (General Electric)	2.99	2.13	**1.10**	2.53	1.44
GM (General Motors)	1.22	1.23	0.63	0.63	**0.61**
GS (Goldman Sachs Group)	1.14	1.09	0.59	**0.46**	0.48
JNJ (Johnson & Johnson)	0.90	0.77	0.51	0.47	**0.43**
JPM (JPMorgan Chase)	1.08	1.00	0.90	**0.40**	0.42
MAR (Marriott Int'l)	1.21	1.23	0.81	**0.48**	0.63
KO (Coca-Cola)	0.72	0.68	0.45	**0.35**	0.35
MCD (McDonald's)	0.98	0.96	0.61	**0.44**	0.44
NKE (Nike)	1.05	1.06	0.80	**0.49**	0.53
PG (Procter & Gamble)	0.94	0.85	0.57	0.43	**0.40**
VZ (Verizon Communications)	0.73	0.71	0.60	0.37	**0.34**
WMT (Walmart)	0.88	0.94	0.73	**0.35**	0.35

Table 2.11: RMAPE (%) comparison of different models for the one-day ahead forecasting on different symbols

	Random Forest	XGBoost	Stock2Vec	LSTM-Stock2Vec	TCN-Stock2Vec
AAPL (Apple)	1.89	1.76	1.04	0.85	**0.68**
AFL (Aflac)	1.15	1.19	0.87	0.60	**0.53**
AMZN (Amazon.com)	1.60	1.55	1.28	0.95	**0.78**
BA (Boeing)	1.74	1.85	1.13	1.11	**1.02**
CVX (Chevron)	1.25	1.42	0.88	0.65	**0.57**
DAL (Delta Air Lines)	1.39	1.36	0.83	0.71	**0.57**
DIS (Walt Disney)	1.41	1.38	0.81	0.79	**0.66**
FB (Facebook)	1.77	2.75	1.06	0.85	**0.73**
GE (General Electric)	3.96	2.89	**1.35**	2.91	1.72
GM (General Motors)	1.62	1.60	0.82	0.84	**0.77**
GS (Goldman Sachs Group)	1.44	1.39	0.84	**0.58**	0.61
JNJ (Johnson & Johnson)	1.33	1.11	0.72	0.70	**0.60**
JPM (JPMorgan Chase)	1.40	1.33	1.20	**0.53**	0.54
MAR (Marriott Int'l)	1.49	1.50	1.07	**0.66**	0.78
KO (Coca-Cola)	0.90	0.92	0.57	0.47	**0.45**
MCD (McDonald's)	1.30	1.22	0.73	0.62	**0.57**
NKE (Nike)	1.38	1.34	1.03	**0.65**	0.67
PG (Procter & Gamble)	1.19	1.11	0.73	0.58	**0.50**
VZ (Verizon Communications)	0.93	0.93	0.76	0.49	**0.45**
WMT (Walmart)	1.15	1.23	0.89	0.47	**0.43**

2.C Plots of the actual versus predicted prices of different models on the test data

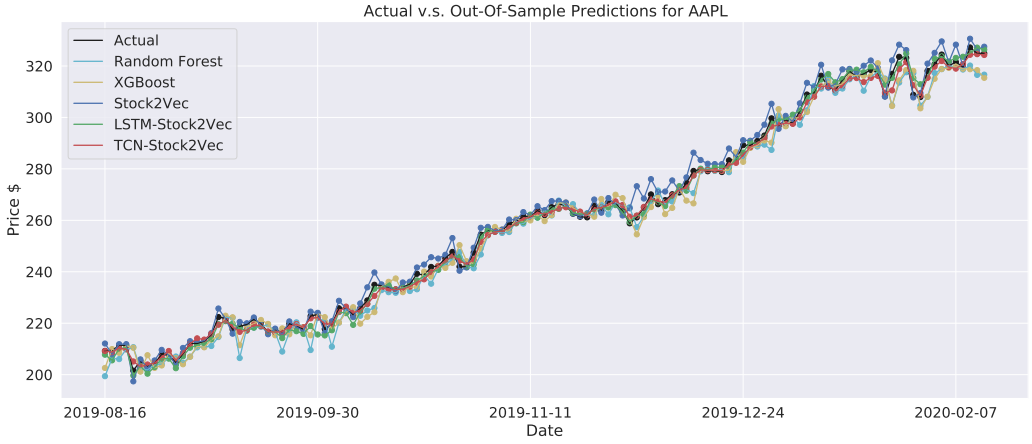


Figure 2.11: AAPL daily price predictions over test period, 2019/08/16-2020/02/14.

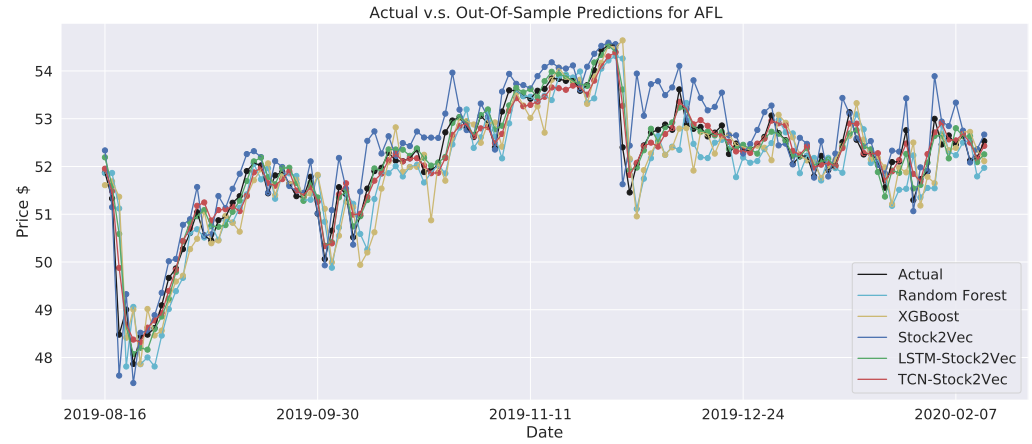


Figure 2.12: AAPL daily price predictions over test period, 2019/08/16-2020/02/14.

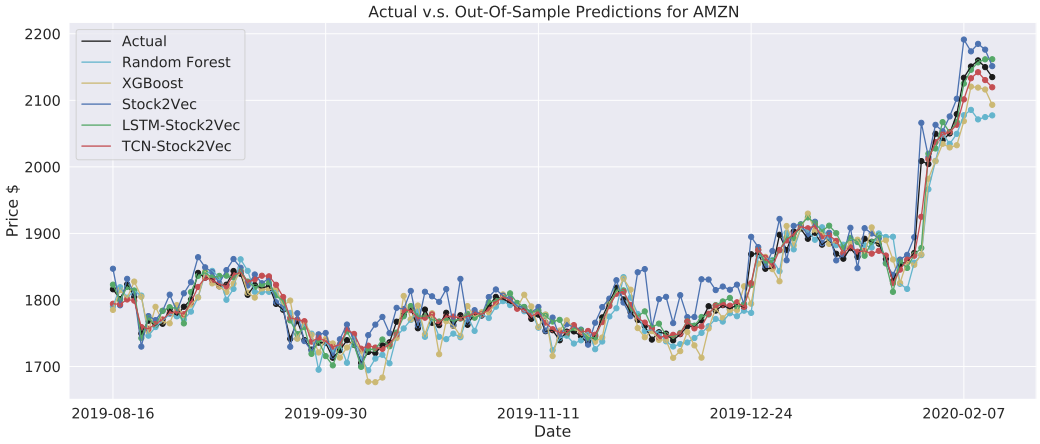


Figure 2.13: AAPL daily price predictions over test period, 2019/08/16-2020/02/14.

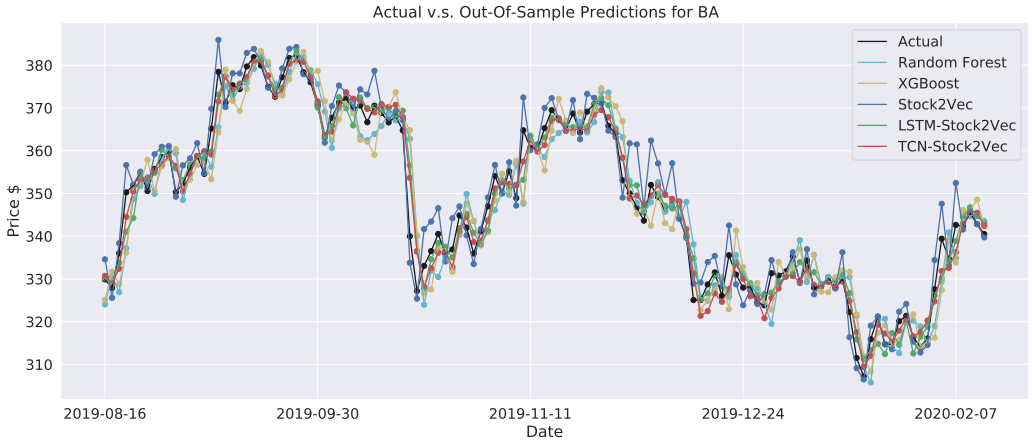


Figure 2.14: AAPL daily price predictions over test period, 2019/08/16-2020/02/14.

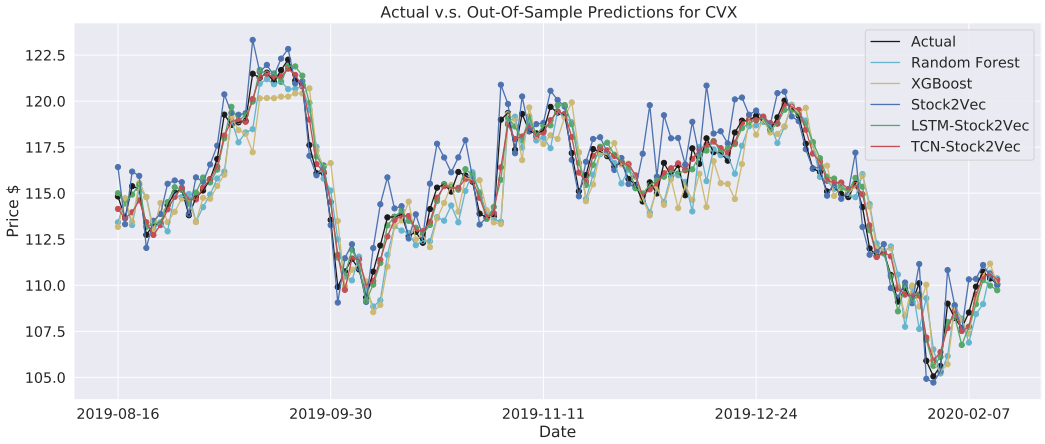


Figure 2.15: AAPL daily price predictions over test period, 2019/08/16-2020/02/14.

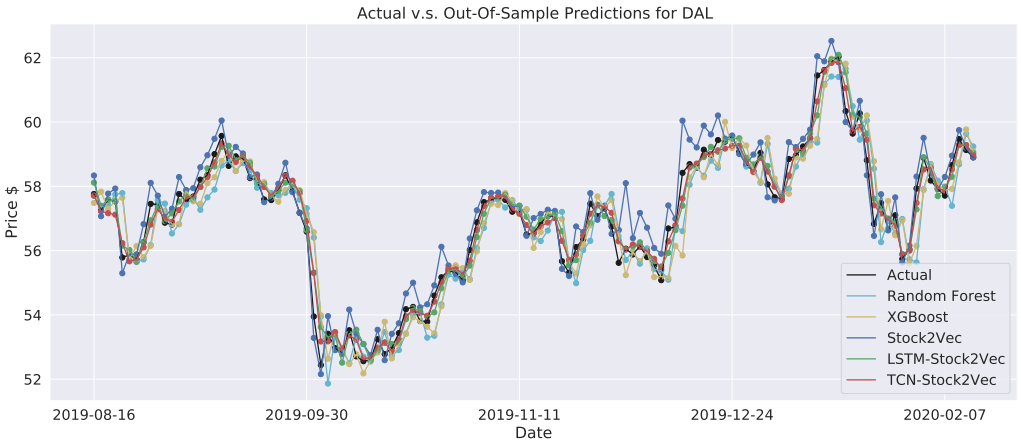


Figure 2.16: Showcase DAL of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

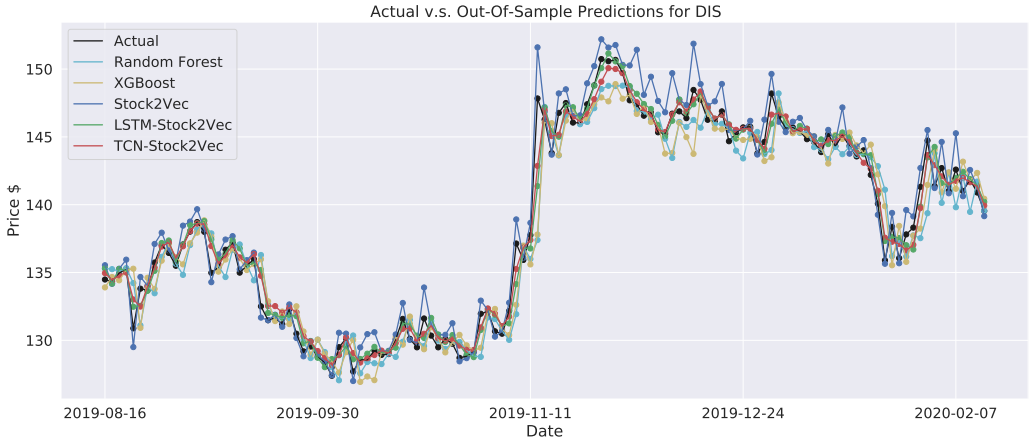


Figure 2.17: Showcase DIS of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

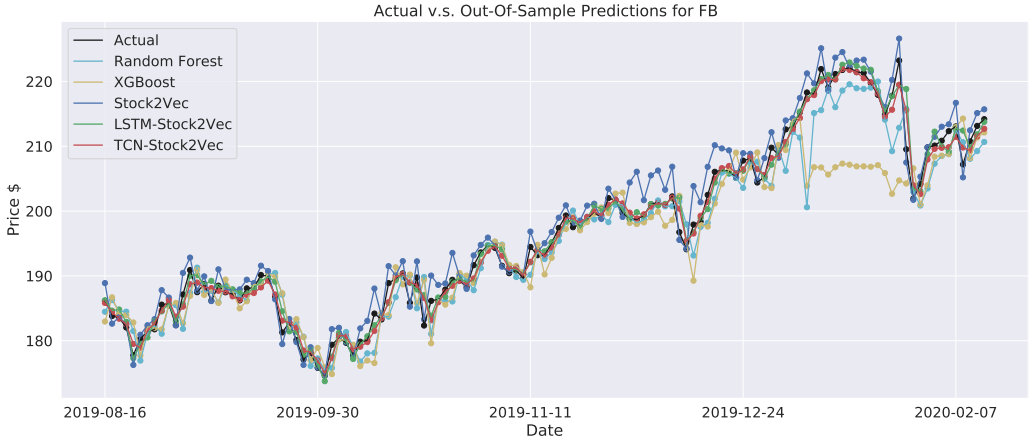


Figure 2.18: Showcase FB of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

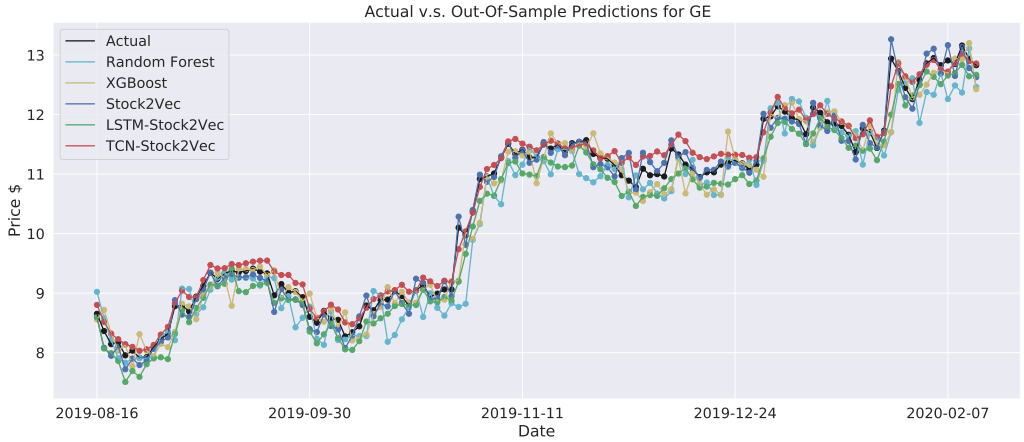


Figure 2.19: Showcase GE of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

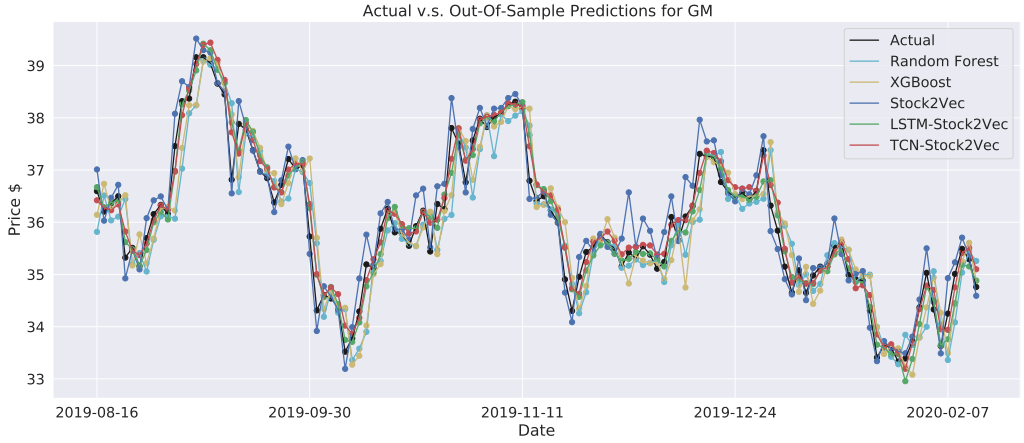


Figure 2.20: Showcase GM of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

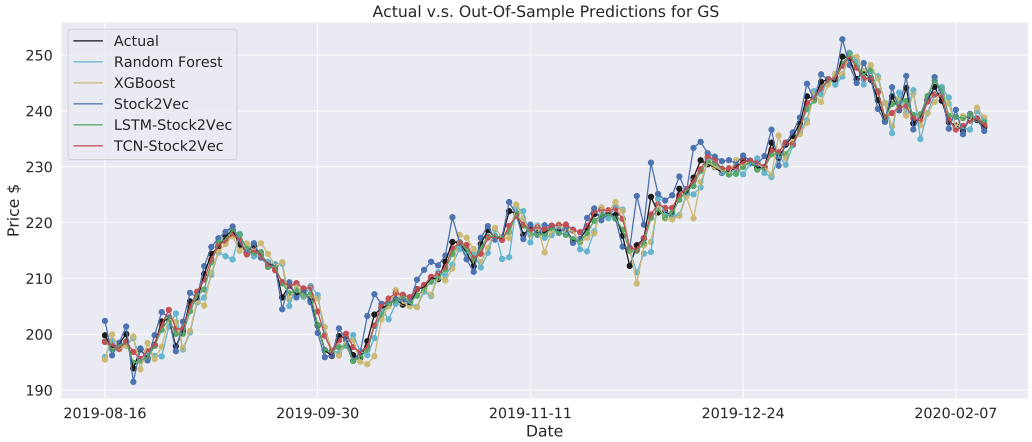


Figure 2.21: Showcase GS of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

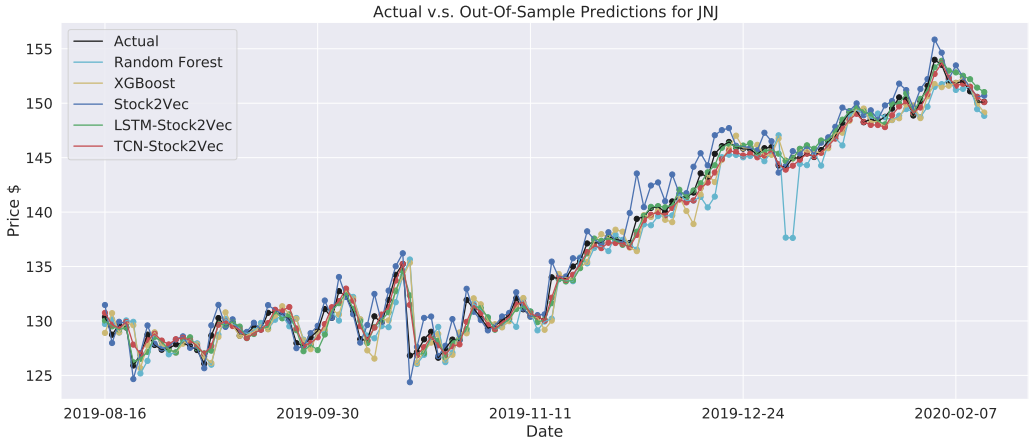


Figure 2.22: Showcase JNJ of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

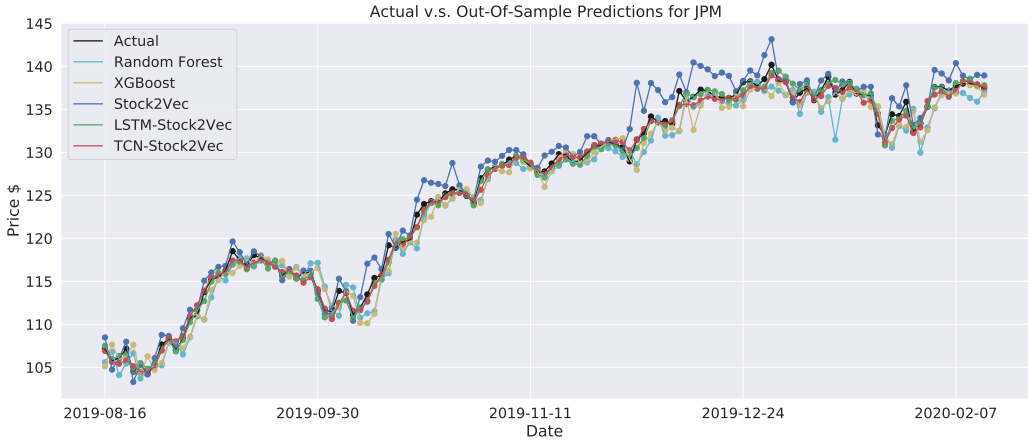


Figure 2.23: Showcase JPM of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

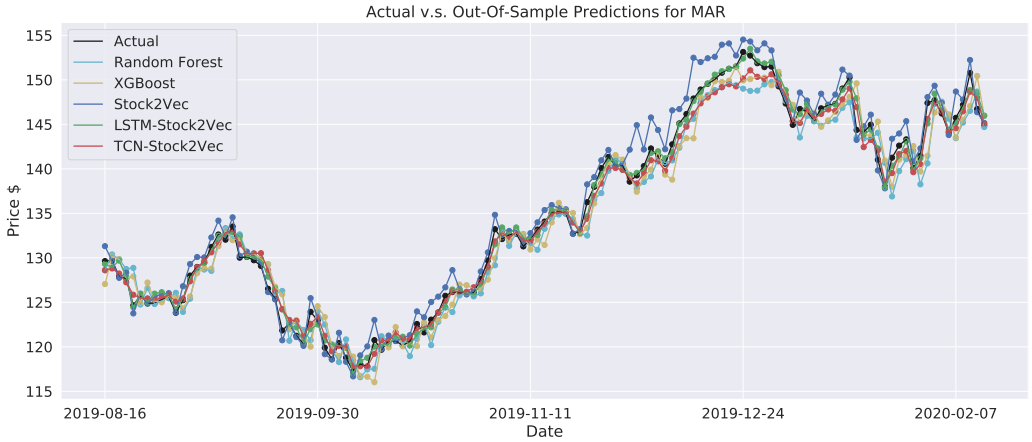


Figure 2.24: Showcase MAR of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

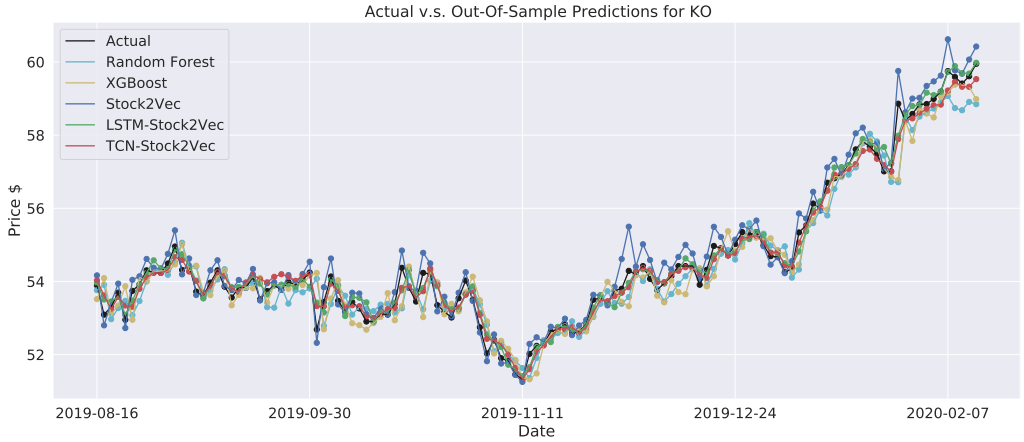


Figure 2.25: Showcase KO of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

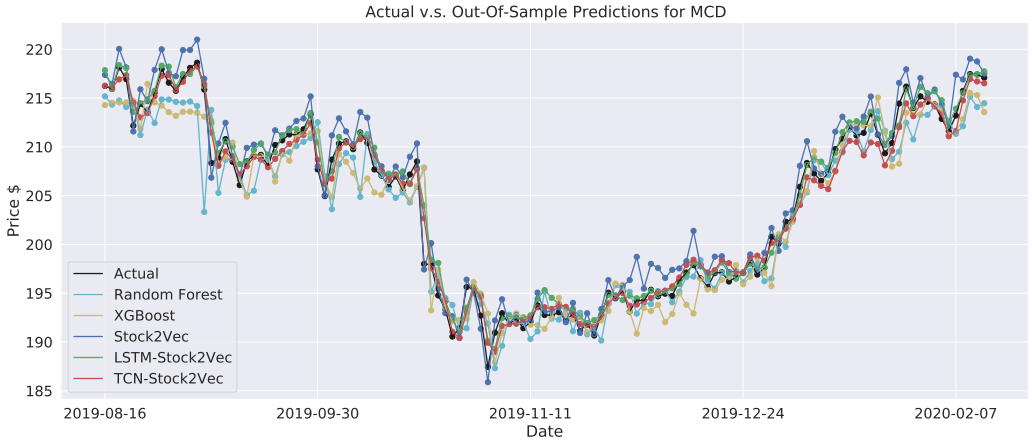


Figure 2.26: Showcase MCD of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

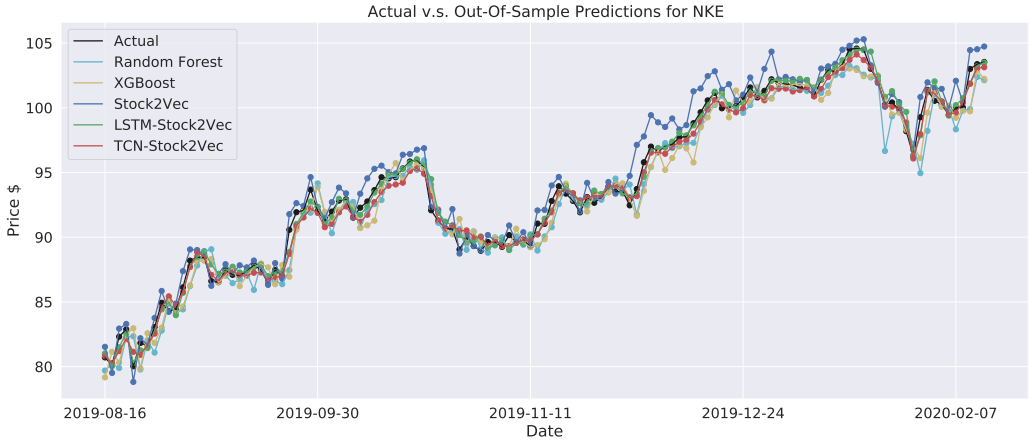


Figure 2.27: Showcase NKE of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

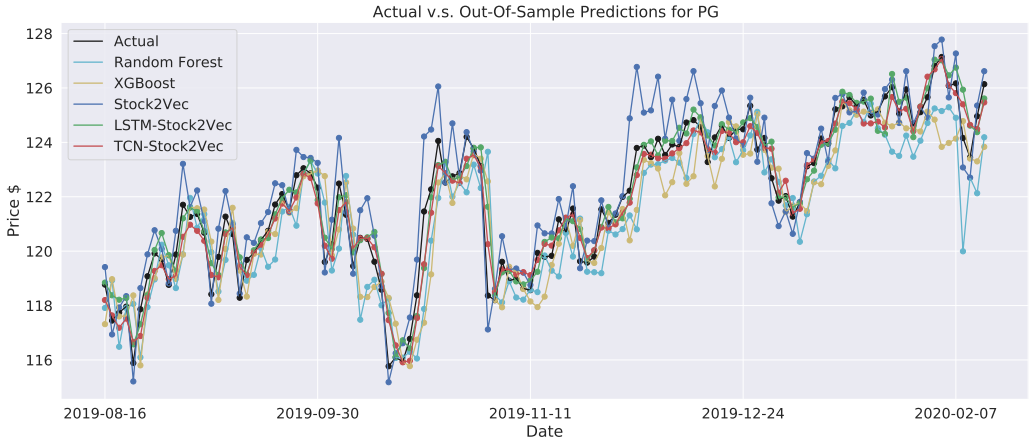


Figure 2.28: Showcase PG of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

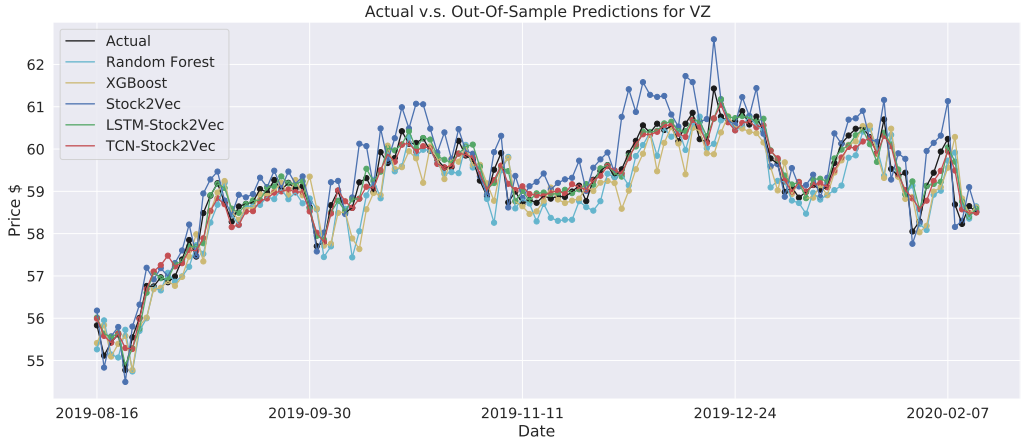


Figure 2.29: Showcase VZ of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

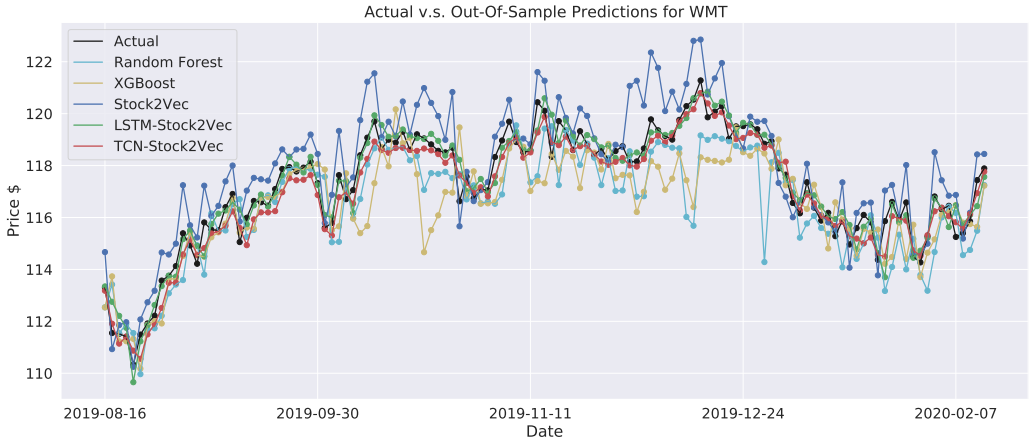


Figure 2.30: Showcase WMT of predicted v.s. actual daily prices of one stock over test period, 2019/08/16-2020/02/14.

Chapter 3

Reinforcement Learning Preliminaries

3.1 Markov Decision Processes

A natural abstraction for many sequential decision-making problems is to model the system as a *Markov Decision Process* (MDP) [126], in which the agent interacts with the environment over a sequence of discrete time steps. It is often represented as a 5-tuple: $M =< \mathcal{S}, \mathcal{A}, T, R, \gamma >$, where $\mathcal{S}$ is a set of *states*; $\mathcal{A}$ is a set of *actions* that can be taken; $T : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{P}_{\mathcal{S}}$ is the *transition function* such that $\int_{s' \in \mathcal{S}} T(s'|s, a) = 1$, which denotes the (stationary) probability distribution over $\mathcal{S}$ of reaching a new state s', after taking action a in state s; R is the reward function, which can take the form of either $R : \mathcal{S} \mapsto \mathbb{R}$, $R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, or $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$; and $\gamma \in [0, 1)$ is the *discount factor*.

A policy $\pi : \mathcal{S} \mapsto \mathcal{P}_{\mathcal{A}}$ defines the conditional probability distribution of choosing each action while in state s. For an MDP, once a stationary policy is fixed, the distribution of the reward sequence is then determined. Thus to evaluate a policy π, it is natural to define the *action value function under* π as the expected cumulative discounted reward by taking action a starting from state s and following π thereafter:

$$Q^{\pi}(s, a) \equiv \mathbb{E}_{\pi}\Big[\sum_{\tau=0}^{\infty} \gamma^{\tau} R_{t+\tau} | S_t = s, A_t = a\Big] = R(s, a) + \gamma \int_{s'} T(s'|s, a) Q^{\pi}(s', \pi(s')). \quad (3.1)$$

The goal of solving an MDP is to find an *optimal policy* π^* that maximizes the expected cumulative discounted reward in all states. The corresponding *optimal* action values satisfy $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$, and Banach's fixed-point theorem ensures the existence and

uniqueness of the fixed-point solution of *Bellman optimality equations* [**126**]:

$$Q^*(s,a) = R(s,a) + \gamma \int_{s'} T(s'|s,a) \max_{a'} Q^*(s',a') \tag{3.2}$$

from which we can derive a deterministic optimal policy by being greedy with respect to Q^*, i.e., $\pi^* = \text{argmax}_{a \in \mathcal{A}} Q^*(s,a)$.

3.2 Value-based Reinforcement Learning

In reinforcement learning problems, the agent must interact with the environment to *learn* the information about the transition and reward functions, meanwhile trying to produce an optimal policy. While interacting with the environment, at each time step t, the agent senses some representation of current state s, selects an action a, then receives an immediate reward r from the environment and finds itself in a new state s'. The *experience tuple* $< s, a, r, s' >$ summarizes the observed transition for a single step. Based on the experiences through interacting with the environment, the agent can either learn the MDP model first by approximating the transition probabilities and reward functions, and then plan in the MDP to obtain an optimal policy (this is called the *model-based* approach in reinforcement learning); or without learning the model, directly learn the optimal value functions and upon which the optimal policy is derived (this is called the *model-free* approach).

As a *model-free* approach, Q-learning [127] updates one-step bootstrapped estimation of Q-values from the experience samples over time steps. The update rule upon observing $< s, a, r, s' >$ is

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big) \tag{3.3}$$

in which α is the learning rate, $r + \max_{a'} Q(s',a')$ serves as the update target of the Q-value, which can be seen as a sample of the expected value of one-step look-ahead estimation for state-action pair (s,a), based on the the maximum estimated value over next state s', and the last term $Q(s,a)$ is simply the current estimation. The difference $\delta = r + \gamma \max_{a'} Q(s',a') -$

$Q(s,a)$ is referred to as temporal difference (TD) error, or Bellman error. Note that one can bootstrap more than one step when estimating the target, often by using the *eligibility trace* as in $TD(\lambda)$ [128]. Q-learning is guaranteed to converge to the optimal values in probability as long as each action is executed in each state infinitely often, s' is sampled following the distribution $T(s,a,s')$, r is sampled with mean $R(s,a)$, variance is bounded and given appropriately decaying α.

3.3 Deep Q-Networks

For environments with large state spaces, the Q-values are often represented by a function of state-action pairs rather than the tabular form, i.e., $Q_\theta(s,a) = f(s,a|\theta)$, where θ is a parameter vector. We consider Q-learning with function approximation in this paper. To update parameter vector θ, first-order gradient methods are usually applied to minimize the mean squared error (MSE) loss:

$$\theta \leftarrow \theta + \alpha\delta\nabla_\theta Q_\theta. \tag{3.4}$$

However, with function approximation, the convergence guarantee can no longer be established in general. Neural networks, while attractive as a powerful function approximator, were well known to be unstable and even to diverge when applied for reinforcement learning until deep Q-network (DQN) [129] was introduced to show great success, in which several important modifications were made. *Experience replay* [130] was used to address the non-stationary data problem, by storing and mixing the samples (i.e., experiences) into a replay memory for the updates. During training a batch of experiences is randomly sampled each time and the gradient descent is performed on the sampled batch. This way the temporal correlations could be alleviated. In addition, a separate *target network*, which is a copy of the learned network parameters (θ) is employed. This copy is frozen for a period of time and is only updated periodically (denoted as θ^-), and is applied to calculate the TD error, with the aim of improving stability.

3.3.1 Double DQN

A variety of extensions and generalizations have been proposed and shown successes in the literature. Overestimation due to the max operator in Q-learning may significantly hurt the performance. To reduce the overestimation error, double DQN (DDQN) [131] decouples the action selection from estimation of the target, that is, choosing the maximizing action according to the original network (Q_θ), and evaluate the current value using the other one (Q_{θ^-} from the target network), i.e.,

$$Q_\theta(s,a) \leftarrow r + \gamma Q_{\theta^-}(s', \arg\max_a Q_\theta(s',a)). \tag{3.5}$$

The procedures of double DQN is shown in Algorithm 1.

Algorithm 1 Double DQN

Initialize policy network Q_θ and target network $Q_{\theta\_}$ with random parameters.
Initialize replay buffer $\mathcal{B}$.
for each episode **until** end of learning **do**
  Initialize state s
  for step $t = 1, \cdots$ **until** s is terminal state of an episode **do**
    Select action $a_t = \text{argmax}_a Q_\theta(s,a)$ with exploration
    Take action a_t, observe reward r and next state s'
    Store experience tuple $< s, a_t, r, s' >$ into $\mathcal{B}$
    Sample a mini-batch of experiences from $\mathcal{B}$.
    for all sampled experience in the mini-batch **do**
      To train network Q_θ, compute $a' = \text{argmax}_a Q_\theta(s',a)$
      Estimate TD target with target network $y = r + Q_{\theta\_}(s',a')$
      Backpropagate TD error $\delta = y - Q_\theta(s,a_t)$ through Q^k, update θ with learning rate α
    end for
    $s \leftarrow s'$
    Update target network $\theta\_ \leftarrow \theta$ in a fixed frequency
  end for
end for

3.3.2 Dueling DQN

[132] proposed the dueling network architecture, in which lower layers of a deep neural network are shared and followed by two streams of fully-connected layers, that are used to represent two separate estimators, one for the state value function $V(s)$ and the other for

the associated state-dependent action advantage function $A(s, a)$. The two outputs are then combined to estimate the action value $Q(s, a)$:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \tag{3.6}$$

Note here the average of advantage values across all possible actions are used to achieve better stability, instead of the max operator in the other form proposed in [132], i.e.,

$$Q(s, a) = V(s) + A(s, a) - \max_{a'} A(s, a') \tag{3.7}$$

The dueling factoring often leads to faster convergence and better policy evaluation, especially in the presence of similar-valued actions. The deployment of advantage values is more robust to noise, since it emphasizes the gaps between Q-values of different actions given the same state, which are usually tiny thus small amount of noise may results in reordering of actions. In addition, the subtraction of an action-irrelevant baseline in Equation (3.6) also effectively reduces variance, which helps stabilize learning and thus is more often used. The shared feature learning module also generalizes learning across actions, in which more frequent updating of the value stream V leads to more efficient learning of state values, contrasts with that in DQNs of a single stream output, only one of the action values is updated while other action values remain untouched.

3.3.3 Bootstrapped DQN

The main purpose of Bootstrapped DQN [133] is to provide efficient "deep" exploration inspired by *Thompson sampling* or as *probability matching* in Bayesian reinforcement learning [134], but instead of maintaining a distribution over possible values and intractable exact posterior update, it takes a single sample from the posterior. Bootstrapped DQN maintains a Q-ensemble, represented by a multi-head deep neural network in order to parameterize a set of $K \in \mathbb{N}_+$ different Q-value functions. The lower layers are shared by the K "heads",

and each head represents an independent estimate of the action value $Q^k(s, a|\theta^k)$. For each episode at training, Bootstrapped DQN picks a single head uniformly at random, and follows the greedy policy with respect to the selected Q-value estimates, i.e., $a_t = \text{argmax}_a Q^k(s_t, a)$, until the end of the episode.

Bootstrapped DQN diversifies the Q-estimates and improves exploration through independent initialization of the K heads as well as the fact that each head is trained with different experience samples. The K heads can be trained together with the help of so-called bootstrap mask m_k^τ, which decides whether the k-th head should be trained, i.e., the transition experience τ updates Q_k only if m_τ^k is nonzero. In addition, bootstrapped DQN adapts double DQN in order to avoid overestimation, i.e., the the estimates of TD targets are calculated using the target network $Q_{\theta^k_-}$. The loss backpropagated to k-the head is then

$$L(\theta^k) = \mathbb{E}_\tau[m_\tau^k(r + \gamma Q^k(s', a'|\theta_-^k) - Q^k(s, a|\theta^k))^2] \text{ where } a' = \text{argmax}_a Q^k(s', a|\theta^k) \quad (3.8)$$

Note the gradients should be further aggregated and normalized for updating the lower layers of the network.

3.A A Simple Proof of Policy Invariance under Reward Transformation From Linear Programming Perspective

In this section, we focus on the linear programming (LP) perspective of MDP. By looking at the LP dual form of MDP, it is possible to derive the policy invariance under reward transformation property with little proof, which serves as the theoretical foundation of the inverse reinforcement learning (IRL).

3.A.1 Encoding MDP as LP

Recall that *Bellman optimality equation* is a system of nonlinear equations:

$$V^*(s) = \max_a \left(R(s,a) + \gamma \int_{s'} T(s'|s,a) V^*(s') \right) \tag{3.9}$$

If the state space $\mathcal{S}$ and the action space $\mathcal{A}$ are finite, we can also encode the problem in the linear programming (LP) formulation:

$$\begin{aligned} \min \quad & \sum_{s \in \mathcal{S}} V(s) \\ \text{subject to} \quad & V(s) \geq R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s') \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{3.10}$$

For each decision variable $V(s)$, its optimal value should be no smaller than whichever action we take in state s, and that forms a set of $|\mathcal{A}|$ constraints. The minimization implies that it chooses the smallest upper bound for each $V(s)$, and the summation over every s gives us a single objective function.

Its dual is then:

$$\begin{aligned} \max \quad & \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \mu_{s,a} R(s,a) \\ \text{subject to} \quad & \sum_{a \in \mathcal{A}} \mu_{s',a} = 1 + \gamma \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \mu_{s',a} T(s,a,s'), \qquad \forall s' \in \mathcal{S} \\ & \mu_{s',a} \geq 0, \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{3.11}$$

in which the decision variables $\mu_{s',a}$ can be thought of the amount of "policy flow" by taking action a to land in the next state s', and the constraints can be illustrated as the flow conservation law, i.e., for each state s', the total outgoing flow (L.H.S.) equals the incoming flow from all possible (s,a) pairs weighted by their transition probabilities, plus the existing amount in state s' (which is assumed to be 1 for all states). The objective is to maximize

the total rewards by taking a stationary stochastic policy, which takes action a in state s with probability $\frac{\mu_{s,a}}{\sum_{a' \in \mathcal{A}} \mu_{s,a'}}$.

The primal has $|\mathcal{S}|$ decision variables and $|\mathcal{S}| \times |\mathcal{A}|$ constraints, and the dual has $|\mathcal{S}| \times |\mathcal{A}|$ decision variables and $|\mathcal{S}|$ constraints excluding the non-negativity ones.

3.A.2 Policy Invariance under Reward Transformation

[135] derived the result that under certain circumstances of reward shaping, we can change the reward function of MDP without changing the optimal policy.

Let $\Phi : \mathcal{S} \mapsto \mathbb{R}$ be some function over states, the so-called **potential-based shaping function** $F : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is defined as the difference of (discounted) potentials, i.e., $F = \gamma\Phi(s') - \Phi(s)$. Then the reward is reshaped as

$$R'(s, a, s') = R(s, a, s') + F(s, a, s') = R(s, a, s') - \Phi(s) + \gamma\Phi(s').$$

For example, $R'(s, a)$ could be an affine transformation of $R(s, a)$ with positive coefficient, i.e., $R'(s, a) = \alpha R(s, a) + \beta$, in which $\alpha \in \mathbb{R}^+, \beta \in \mathbb{R}$.

Theorem 3.1. *F is a potential-based shaping function is a necessary and sufficient condition for it to guarantee consistency with the optimal policy (when learning from $M' = (\mathcal{S}, \mathcal{A}, T, \gamma, R + F)$ rather than $M = (\mathcal{S}, \mathcal{A}, T, \gamma, R)$), in the following sense:*

- *(Sufficient) If F is a potential-based shaping function, then every optimal policy in M' will also be an optimal policy in M (and vice versa).*
- *(Necessary) If F is not a potential-based shaping function, then there exist (proper) transition function T and a reward function R such that no optimal policy in M' is optimal in M.*

The authors proved the sufficiency of the theorem by looking at the Q-values and using infinite telescoping sum (in the expectation). We notice that this result is more straightforward from the LP point of view by looking at the dual as in Equation (3.11), where the reward function only appears in the objective function but is not shown in the constraints. The sufficiency of Theorem 3.1 can then be directly derived without telescoping expansion as follows.

- (Multiply by a positive scalar) $R'(s,a) = \alpha R(s,a)$, the coefficients of the objective function is multiplied by α, so the optimal solution won't change.
- (Add a scalar) $R'(s,a) = R(s,a) + \beta$, add a constant to the objective function, so the optimal solution won't change.
- (General Potential-based shaping) $R'(s,a,s') = R(s,a,s') - \Phi(s) + \gamma\Phi(s')$, the objective is then

$$
\begin{aligned}
&\max \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \mu_{s,a} R'(s,a) \\
=&\max \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \mu_{s,a} \left[R(s,a) - \Phi(s) + \gamma\Phi(s')\right] \\
=&\max \left(\sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \mu_{s,a} R(s,a) - \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \mu_{s,a} \Phi(s) + \sum_{s\in\mathcal{S}}\sum_{a\in\mathcal{A}} \mu_{s,a} \gamma\Phi(s') \right)
\end{aligned}
$$

  in which the last two term forms a constant, and we go back to the case of adding a scalar as above.

Chapter 4

Re-anneal Decaying Exploration in Deep Q-Learning

Existing exploration strategies in reinforcement learning (RL) often either ignore the history or feedback of search, or are complicated to implement. There is also a very limited literature showing their effectiveness over diverse domains. We propose an algorithm based on the idea of reannealing, that aims at encouraging exploration only when it is needed, for example, when the algorithm detects that the agent is stuck in a local optimum. The approach is simple to implement. We perform an illustrative case study showing that it has potential to both accelerate training and obtain a better policy.

4.1 Introduction

The goal of a reinforcement learning agent is to try to make best decisions, based on the information it gathers along the way. Unlike supervised learning tasks, however, the agent can only have access to the environment through its own actions. It needs to explicitly explore its environment and gather information for decision making. Simultaneous exploitation (making best decisions) and exploration (gathering of information) tasks create a dilemma, and balancing the two is one of the core challenges in reinforcement learning. An early survey [136] of exploration strategies made a distinction between two categories: undirected and directed. The key idea behind the former is to add randomness, in the hope that a random action might lead towards better actions compared to the suboptimal policy which is viewed as the best given current information. Directed strategies take a "global" view and measure some statistics of the past experiences, and utilize these measures to guide efficient exploration mainly by adding an exploration bonus to the reward function, so that the less visited (in terms of pseudo-count [137] using a fitted density model or hash-count

[138] with locality sensitive hashing) state-action pairs, or those with larger information gain ([139], VIME [140]) or prediction error [141], are favored. Such strategies often allow for theoretical analysis, usually based on multi-armed bandit (MAB) problem theory [142]. In spite of their appealing mathematical formalism and theoretical guarantees in finite case, directed exploration strategies have not shown effectiveness over domains and thus have not played an important role in the recent success of reinforcement learning [138].

In many real-life learning applications, RL agents are trained to achieve optimal performance in certain specific tasks. Hence, it is often simple to distinguish when the learned behavior of the agent is acceptable. It is well described in the literature that if insufficient focus has been placed on exploration, then the agent can learn to stay in a "comfort zone" of a local optimum. In this case, one needs to force the agent to leave the "comfort zone" and try new actions which would take it to new states that have not been well-learned yet, so that it can explore more information about the environment, in the hope of finding better policy. In this paper, we propose a heuristic to overcome this problem, and in general, to speed up learning procedure.

Our main contribution is an easy yet efficient and scalable method that encourages exploration in complex reinforcement learning domains when it is needed. To be more specific, we emphasize the model dynamics and the agent's behavior rather than uncertainty estimates while measuring the need for exploration. This can be accomplished by training a supervised model to make predictions based on existing experiences, which would require extensive computation as well as the effective representation of the supervised model. In this paper, we focus on using a simple heuristic measure and an annealing-based method to redirect the agent. Our approach could be extended to serve as a general framework for interactively training in complex RL domains, to aid the agent in finding better policies.

Another contribution is that we abstract the learning procedure with the view of general optimization/search, and apply a generic method that attempts to improve search algorithms on hard problems, specifically a modified version of simulated annealing and thus our method

is referred to as *exploration reannealing*. A number of other metaheuristic approaches can be applied in similar fashion to better balance the exploration-exploitation tradeoff in reinforcement learning.

It is worth noting that exploration reannealing itself is not a complete learning algorithm, and in fact needs to be combined with other reinforcement learning tools. In this paper, we emphasize and evaluate its application to deep Q-learning, but the combination of exploration reannealing with other methods can be considered.

Section 4.2 provides some widely-used exploration strategies in DQN, also explains the trade-off between exploration and exploitation. In Section 4.3, we present the motivation of our reannealing method and discuss the appropriate ways to use it, as well as the specific algorithm. We perform empirical studies and showcase the improvement by exploiting our reannealing strategy on a large scale challenging domain, the Lunar Lander model, in Section 4.4. Finally, we outline conclusions in Section 4.5.

4.2 Exploration in DQN

4.2.1 Exploration Strategies

ε-Greedy Exploration. The most commonly-used strategy for exploration is the ε-greedy method, in which the agent selects the action it believes to be the best according to current $Q(s, a)$ values for the most of the time, and occasionally acts randomly. That is, it takes the greedy action $\arg\max_\pi \int Q(s, a)\pi(a|s)da$ with probability $1 - \varepsilon$, and selects (uniformly) randomly among all actions with probability ε. Then after infinitely many steps, every state-action pair will be visited infinitely often, thus all $Q(s, a)$ converge to the true action values $Q^*(s, a)$ almost surely [143]. However, deficiencies of ε-greedy are also often discussed and new RL algorithms can be proposed. For example, the time complexity of ε-greedy learning is exponential with respect to the size of the state space, which leads to PAC-learning ideas for RL [144]. Moreover, ε-greedy selects actions with equal probability. Intuitively, we would expect the agent to pay more attention to more "promising" actions, i.e., those with maybe

slightly lower Q-values than the current greedy action, rather than those with really low Q-values, which have less potential to be optimal. Moreover, it might be a waste to explore those actions with low Q-values, which have been selected many times since we may be confident that these are "bad" actions. Despite its deficiencies, due to its simplicity, practical effectiveness, and the ease with which it can be embedded into Q-learning, ε-greedy strategy has been prevalent in most value-based algorithms in reinforcement learning, including DQN and its variants.

Softmax (or Boltzmann) Exploration. The (variational) free energy for an RL agent can be defined as

$$F(\pi) = -\int Q(s,a)\pi(a|s)da + T\int \pi(a|s)\log\pi(a|s)\mathrm{d}a, \tag{4.1}$$

in which the first term represents the energy of the agent, and the second term is the standard form of negative entropy. Coefficient T of the negative entropy is referred to as *temperature*. Free energy principle claims that a self-organizing agent would act on the environment by minimizing its free energy, by which it reaches an equilibrium with the environment (or more accurately, a sampling of sensory data) [145]. Minimization of free energy gives us

$$\pi(a|s) = \frac{\exp\left(\frac{Q(s,a)}{T}\right)}{\int \exp\left(\frac{Q(s,a')}{T}\right)\mathrm{d}a'} \tag{4.2}$$

which is called the softmax policy or Boltzmann policy. Note that hyperparameter T controls the exploration [146]. If the temperature is high, the action selection according to π approaches uniform distribution, which yields more randomness and thus encourages exploration. On the other hand, low temperature would reduce the randomness and enhance exploitation. As an extreme case, if the temperature is zero, the negative entropy term in Eq. (4.1) goes away, and the corresponding policy becomes deterministic which takes the greedy action $\arg\max_\pi \int Q(s,a)\pi(a|s)da$ given the current estimate of $Q(s,a)$. With the

softmax probability, the possibilities for each action to be selected are ranked and weighted relevant to their estimated Q-values, instead of equal probabilities for all actions in ε-greedy approach.

4.2.2 Exploration Decay

Theoretical analysis of exploration strategies is usually performed through the Multi-Armed Bandit (MAB) model [142]. ε-Greedy strategy has been well studied through regret analysis in MAB, in which the regret is often defined as a measure of the difference in value between taking an action a and the optimal action a^* at time t, i.e., $\rho(t) = \mathbb{E}_a[V^* - Q(a_t)]$, that is, the opportunity loss of taking a_t for one step. The total regret is then the overall opportunity loss over time until time t, i.e., $L(t) = \sum_{\tau=1}^{t} \rho(\tau) = \mathbb{E}_a[\sum_{\tau=1}^{t}(V^* - Q(a_\tau))]$. As shown in [142], if we set $\varepsilon = 0$, that is, always choose the action with the largest Q-value greedily without exploration attempt, then the greedy action could lock onto a suboptimal policy forever, in that case, a linear bound ($\mathcal{O}(t)$) on total regret is achieved. On the other hand, if we take ε-greedy action with constant $\varepsilon > 0$, the agent would keep exploring with probability ε even if the optimal policy is found, thus also resulting in linear total regret.

A natural approach, then, is to encourage exploration early and exploitation later, which is achieved with decaying ε over time. Decaying-ε-greedy can achieve asymptotically logarithmic bound on total regret, by defining $\varepsilon_t = \min(1, \frac{c}{\delta^2 t})$, where c is a constant and δ is the gap between the best and second best action values, both are unknown however. Thus it is often hard to derive an efficient decay schedule. Nevertheless, it is important to emphasize the decay strategy on exploration. We note that the exploration of stochasticity for softmax strategy could also be annealed during training by changing the temperature T over time.

4.3 Exploration Reannealing

4.3.1 Local Optima in DQN

In theory, Q-learning converges to the optimal policy if all state-action pairs are visited infinitely often. However, this condition cannot be met in practice if the state-space is too large or continuous. A neural network in DQN approximates large or continuous state space and thus suffers from this problem. A deep neural network in general is of very high dimensinality, and popular practical optimization techniques, such as stochastic gradient descent, only consider first-order gradient information of the loss function. Such optimization algorithms may get stuck at local optima or saddle points. In practice, for regular neural networks, saddle points of the loss function can be escaped by applying special optimization techniques [147], and it is often the case that a local optima is good enough for many supervised learning problems. However, this might not be the case in DQN. A local optimum in DQN arises from both the complicated structure of loss function itself, and the limited representation and inference ability of a neural network for the search space. The latter is especially pervasive for state-space segments that are not well-explored. As a result, the learning agent cannot make progress for a long time and might waste learning resources by updating information for irrelevant parts of state space. Therefore, it is more common in practice that a DQN learning agent gets stuck in poor local optima, due to the difficulty of handling the exploration-exploitation trade-off well.

4.3.2 Exploration Reannealing

Simulated annealing (SA) is a classic heuristic optimization approach used to escape local optima. At the heart of it is an analogy with thermodynamics. Boltzmann probability distribution again is used to analogically represent the (variational) free energy, with a control hyperparameter, referred to as temperature T. The free energy determines the stochasticity for the search direction, which aids the local search to escape from local optima. The

concepts are fundamentally based on the same principle as those in exploration strategies we mentioned above, in which decaying the exploration is fundamentally the same as tuning the temperature in SA. In some variations of simulated annealing search, re-anneaing [148] is a quite common idea for the anneal schedule, that is, the temperature is periodically set to a high value in order to encourage exploration.

Similar idea can be naturally employed in our problem for enabling exploration in RL. Note that the act of reannealing itself can be implemented in a straightforward way for both ε-greedy and softmax strategies we introduced above. In ε-greedy, we can easily reset the exploration rate ε to a large value (note that $0 \leq \varepsilon < 1$ and ε decays over time) if poor local optima is met. Similarly, for softmax action selection, we can more directly reset the temperature to a high value (e.g., close to the initial temperature) whenever it is necessary. We note here that with finitely many reannealing events, the theoretic guarantee of asymptotically logarithmic bound on total regret will still hold as for decaying ε-greedy. A more significant challenge relates to the timing of reannealing events. We will discuss this question below, but first we discuss additional reasons why we believe reannealing can bring benefits for learning in DQN.

The key advantage of reannealing exploration is that it could substantially improve the sample efficiency. We know that collecting data by interacting with the environment is usually expensive for RL systems. While stuck in local optima, it is usually the case that the TD errors being backpropagated are small, and the agent could learn little information thus gain little learning progress. With reannealing, the agent would tend to take random actions in this case and is more likely to experience unacquainted states thereafter. Those state-action pairs are usually visited much less often than those obtained by taking greedy policy, thus have larger TD errors in general and from which the agent can learn more.

Another advantage is that reannealing exploration could substantially alleviate the data imbalance problem. Without reannealing exploration, large amounts of samples are collected around local optima, resulting in data distribution biased in favor of samples that may not

be relevant. As a result, a notable portion of model parameters are dedicated to describing states around (poor) local optima, and much of the training work is hence in vain. By reannealing the exploration instead of exploiting around the local optima, random actions are taken with much higher probabilities, the agent are more likely to jump out of the local optima and experience with unacquainted states, gather significantly more useful information about the entire environment as well as the training overall.

Finally, a training episode is often designed to have a finite horizon for computational simulation purpose. Each episode finishes when either certain criteria are met (in this case, a success or a failure on the task is defined and final reward is given), or the time step exceeds a fixed period. When a local optimum is encountered, the agent tends to wander around until exceeding the time limit of an episode. It is important to note that using a time limit makes the environment non-stationary, since in this case the final reward is never actually assigned, and hence, the agent may not be able to recognize a suboptimal policy. Exploration reannealing can enable the agent to actually achieve either success or failure, making sure that appropriate reward is assigned. Consequently, more episodes finish with more concrete information gain.

4.3.3 Defining Poor Local Optima

Given the intuitive advantages of applying reannealing to DQN, we next describe our proposed algorithm. As described above, the mechanism of reannealing is straghtford for both ε-greedy and softmax strategies. On the other hand, detemining the appropriate times for reannealing is more difficult. Clearly, we must reanneal, when the agent is stuck in a poor local optimum. Unfortunately, in high-dimensional spaces formally determining local optimum is challenging. In practice, an often used empirical way is to track variation of loss function across iterations. When the loss stops improving, it is often the case that the search reaches local optimum. However, simply looking at the change in loss does not tell

us whether the local optimum is acceptable. It might be the case that near-global optimum has already been achieved, and hence there is no need to escape from it.

On the other hand, sometimes a poor local optimum can be easily observed and distinguished by human from the outside perspective, in which case the observer utilizes some *a priori* knowledge that has not been integrated into the reward function. In RL, the agent's learned policy as well as its behavior are determined by optimizing the discounted cumulative rewards, thus an ideal reward function should capture the goal and measure the performance exactly, which requires perfect knowledge of all states and transitions in the environment. Except for some human designed games in which the rules are entirely understood, it often takes considerable effort to tweak the rewards until desired behavior is learned. This then means, that in many applications the reward function is already overloaded in a way as to result in favorable agent's behavior, and, hence, attempting to also use the same reward function to distinguish the quality of local optimum may be either impossible or very complicated with unexpected side-effects (see also inverse reinforcment learning, [149]).

Alternatively, we propose to consider a separate criterion for initiating reannealing. This criterion can be viewed as a supervised model, which makes predictions based on existing experiences. The training labels could be as simple as a categorical signal to denote the need to explore, or as complicated as representation of next state, which would require extensive computation as well as the effective representation of the supervised model. In this paper, we use a simplified version of this supervision idea. We explicitly measure the easily distinguished feature as an a priory defined *heuristic*, representing the fact that the agent's bottleneck behavior due to sub-optimal policy can often be described with some undesirable characteristics from an outside observer's perspective. We, then, can explicitly extract such a feature as a useful heuristic independent from the reward function. Once defined, we can keep track of the heuristic along the learning process and use it to control the learning behavior. See Section 4.4 for an example based on Lunar Lander problem.

4.3.4 Algorithm

The objective of reannealing exploration is to explicitly inform the learning agent that it should be exploring rather than exploiting with a heuristic measure. We set up a heuristic variable called `stuck` to represent if the agent has been stuck in poor local optima. The variable `stuck` should be a global statistic for some aspect of the agent's performance information. If some threshold of "stuck" has been reached, we reanneal the exploration. In ε-greedy learning, we reset ε to 1 and force the agent do pure exploration. The exploration rate ε then is decayed over time. The pseudo-code of our proposed procedure for DQN is shown in Algorithm 2. And similar reannealing strategy applies for softmax, in which we reset the "temperature" T to its initial value, and anneal it again to smaller value over time.

As argued above, the candidates of the `stuck` variable should be some performance measure that might have not been integrated (or not been integrated well) in the reward function. The chosen feature as the explicit heuristic should be a representative bottleneck for learning. We expect that the RL agent could jump out of the local optima by applying reanneal strategy, and be able to learn better policy than the one it obtained before reannealing when it stucks. As a result, acceptable behavior and good policy could be learned faster. We also expect that with reannealing exploration, we could worry less about poor local optima and spend less time on tuning the hyperparameters (such as the annealing schedule, learning rate, etc.) while training.

Algorithm 2 Exploration Reannealing in DQN

Initialize $\varepsilon_t = 1$ and `stuck`, as well as DQN parameters θ.
repeat {for each episode}
  initialize state s
  repeat {for each step in an episode}
    Generate a random number $u \in [0, 1]$
    if $u < \varepsilon_t$ **then**
      Randomly select $a \in A$
    else
      $a \leftarrow \arg\max_{a' \in A} Q(s, a'|\theta)$
    end if
    Take action a, observe reward r and next state s'
    Store experience tuple $< s, a, r, s' >$ into memory
    Sample a batch of experiences from memory
    for all sampled experience in the batch **do**
      Compute the TD error $\delta = r + \max_{a'} Q(s', a'|\theta) - Q(s, a|\theta)$
      Backpropagate δ through the DQN, update θ with learning rate α_t
    end for
    $s \leftarrow s'$
  until s is terminal state
  Update `stuck` according to performance
  if `stuck` meets some threshold **then**
    Reset $\varepsilon_t = 1$
    Reset `stuck` to its initial value
  else
    Decay ε_t
  end if
until end of learning

4.4 Experimental Results

4.4.1 Testbed Setup

We conducted an experiment by implementing a reinforcement learning agent to solve the Lunar Lander task in Box2D [150], interfaced through OpenAI gym environment [151]. In each step, the agent is provided with the current state s of the lander in $\mathbb{R}^8$, in which 6 of the dimensions are in continuous space denoting the position, speed, and angular speed, whereas the other 2 are dummy variables in discrete space, indicating the severity of collision. The agent is allowed to make one of the 4 possible actions (i.e., the action space is discrete).

At the end of each step, the agent receives a reward and moves to a new state s'. An episode finishes if the lander rest on the ground at zero speed (receives additional reward of $+100$), or hits the ground and crashes (receives additional -100 reward), or flies outside the screen, or reaches the maximum of 1000 time steps of one episode. The agent aims for successful landing which is defined as reaching the landing pad (between two flags) centered at the ground at the speed of zero, and receives an additional reward in range $[100, 140]$, while landing outside the pad would cause some penalty. Figure 4.1 provides a snapshot of the task environment.

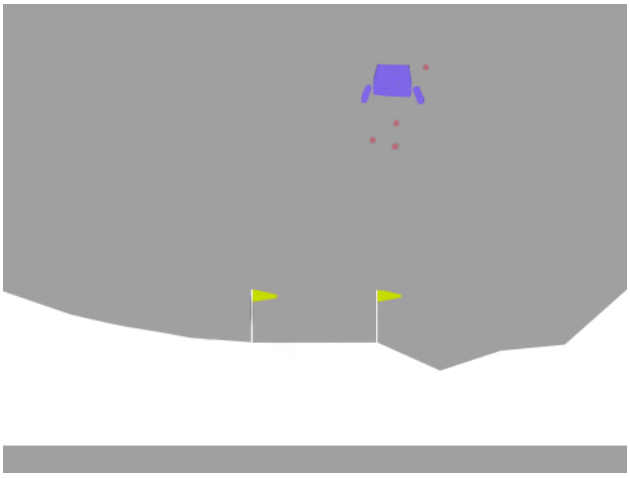


Figure 4.1: Lunar Lander Environment

We use a neural network with two fully-connected hidden layers (which consist of 200 and 60 neurons, respectively) as our function approximator. ReLU nonlinearity is utilized as the activate function for each hidden neuron. The network takes the 8-dimensional vector s which describes the state as the input, and outputs the approximated Q-values for the 4 possible actions. We train the neural network with a FIFO memory of size 10^6 for experience replay. A target neural network for double learning is updated every 20 episodes, so that the original network has enough time to converge. The adaptive moment estimation (Adam) optimizer with initial learning rate set to 0.01 is used to train the network, since it is in general less sensitive to the choice of the learning rate than other stochastic gradient descent algorithms [119]. We apply the pseudo-Huber loss instead of MSE as the loss function, as it

is less sensitive to outliers and is more commonly used in DQN [129]. The discount factor γ is set to 0.99, and ε-greedy policy is used for choosing actions throughout interacting with the environment. For comparison purpose, we used two different exploration decay rates, $\rho_{decay} = 0.99$ and 0.985. These hyperparameters are empirically tuned in the aim of achieving better performance.

4.4.2 Implementation of Exploration Reannealing

As in Q-learning, a simple ε-greedy policy is applied while choosing actions to interact with the environment during training. With large exploration rate ε, the agent fails in exploitation and refining its policy, while with small ε, the agent would have a problem in exploration. For example, if we simply pick $\varepsilon = 0.01$, the agent soon learns to hover above the ground forever but hesitates to land. Annealing strategy for exploration rate is considered and tried, in which ε gradually decreases from 1 to 0.01 during say, half of the training episodes, and $\varepsilon = 0.01$ for the rest of training time. However, this cannot solve the hovering problem. This annealing strategy adds randomness at the early stage of training, but the pretrain step (in order to fill the memory for experience replay) has already provided the agent enough exploration stored in the memory at the beginning. Even if the agent learns to land occasionally, it prefers hovering for most of the time. This is probably because the neural network is dealing with continuous state space, learning through some unknown states with bad decisions would also affects the values of well-learned states. As a result, the agent again learns to hover forever.

In order to escape from such hovering local optima, we carefully engineered a reannealing strategy for the exploration rate. The idea is to encourage the agent to explore while it is hovering. We define a variable `hover` to count the hovering number which starts from 0. Whenever an episode finishes exceeding the time limit (i.e., the maximum 1000 step in an episode), we increase the hovering number by 1. If the next episode finishes within 1000 steps, we halve the hovering number (using integer division). We will reset ε back to 1 (for

fully exploration) and recount if the hovering number reaches 10 (in this case, the agent tends to hover forever). Otherwise, ε anneals to 0.01 as described above.

4.4.3 Results

The network was trained over 10,000 episodes. Figures 4.2a and 4.2b illustrate the results for the ordinary DQN without applying the exploration reannealing strategy, using two different exploration decay rates. We plot the cumulative rewards for each episode shown with grey, and the smoothed moving averages of the last 100 episodes shown with the blue line. Notice that higher rate means slower decay, which results in more exploration at the beginning. In the case that reannealing strategy is not applied, the agent with exploration decay rate $\rho_{decay} = 0.985$ explores less at the beginning than the one with $\rho_{decay} = 0.99$, and performs worse, i.e., its average episodic total rewards are significantly lower, and also the learning process is slower. For instance, with $\rho_{decay} = 0.985$, the agent barely learns to avoid crashing (i.e., with episodic rewards above zero) within 3000 episodes, while with $\rho_{decay} = 0.99$, the agent can obtain the same level in about 2000 episodes. Also, to achieve average episodic rewards above 100, it takes less than 4000 episodes with $\rho_{decay} = 0.99$, and more than 7000 episodes with $\rho_{decay} = 0.985$. This coincides with our intuition, and emphasizes the importance of sufficient exploration.

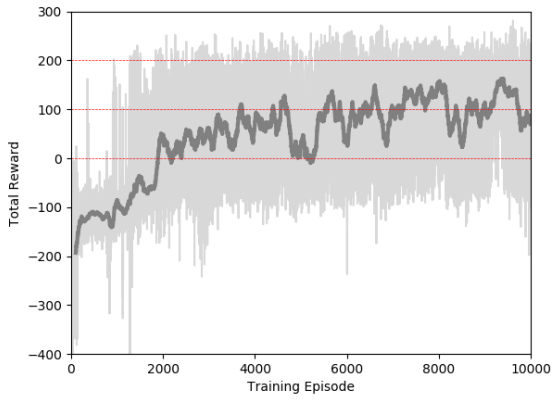
As shown in Figures 4.2c and 4.2d, applying the reannealing strategy improves our result significantly. We could achieve an average value of episodic total reward as high as 200 (in this case, reward 200 means that the agent could land smoothly at the right position on the ground). Without reannealing, however, the agent never achieves such level in either cases (see Figures 4.2a and 4.2b). An interesting observation is the steep falls of the moving average along the training while reannealing is applied, clearly these are the moments when ε is reset to 1. Note that at those times, the falling of episodic total rewards value does not mean the agent is doing worse in general. Q-learning is an off-policy algorithm, which means the learned target policy is not the same as the behavior policy (ε-greedy) it uses

while interacting with the environment and accumulating the samples. The induced greedy policy has not changed much in such a short period of time from the recent ε resetting, so the agent can still do as well as before falling if it acts greedily. At the same time, the target policy keeps learning while exploring. We can see from Figures 4.2c and 4.2d, that for most of the time, it can soon get back to the previous best performance, and often its new peaks are higher, which indicates that it jumps out of the previous local optima.

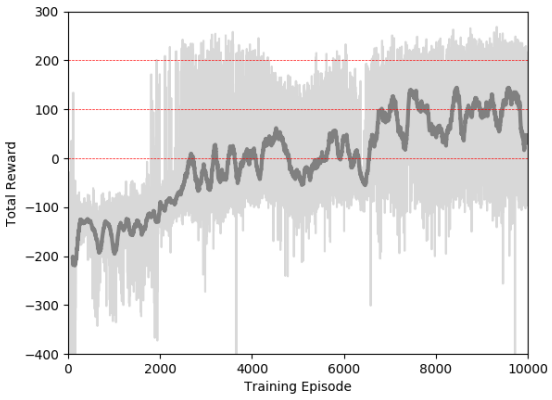
We deliberately choose the sliding window size to be not too big, nor do we show the average values over multiple training runs, so that the curves are not over-smoothed, thus allowing us to discern the occurrences of reannealing. Over-smoothed curves would give us the illusion that the learning is be slower with reannealing strategy, especially at the early training stages. We claim it is not true, using the same argument that Q-learning is off-policy. We cannot compare the derived policies early on since the exploration rate differ a lot, however, while ε reaches its minimum value, we can compare the performance of all "near-greedy" behavior polices. We see that with reannealing, the agent could reach higher values much faster, thus we claim that reannealing accelerated the training.

We also plot the varying ε values along training with reannealing strategy in Figures 4.2e and 4.2f, from which we can directly observe the moments when reannealing was initiated. There is no need to plot such patterns for the cases without reannealing, since ε decays to 0.01 in a few hundred episodes. From Figures 4.2e and 4.2f we can see the frequent reannealing early on, since the agent generally can learn to hover very quickly and frequently. Note that reannealing occurs more frequently with $\rho_{decay} = 0.985$ than that with $\rho_{decay} = 0.99$. We can surmise then that our reannealing strategy serves as a remedy for poor hyperparameter tuning, specifically the exploration decay rate, as long as the reannealing criterion (aka the heuristic measure) is appropriately picked. With insufficient exploration at the beginning, the learning would get stuck in poor local optima more often, but reannealing strategy can force the agent to explore later on when it is necessary, and help find similarly good policy as when training with better hyperparameter. Also notice that there is a long flat tail in

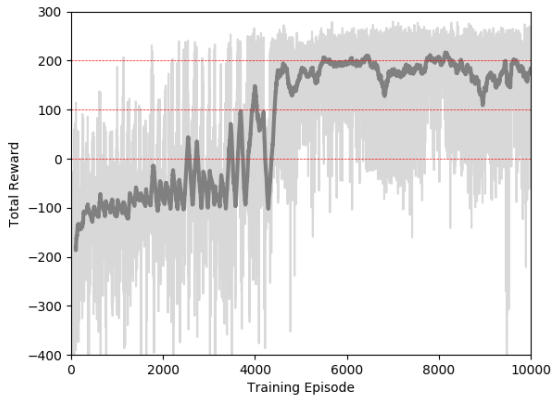
Figure 4.2e after episode 4500. During this period of training, the agent did not reanneal, and the total reward values stays at that level with smaller variance, compared with no reannealing graphs on Figures 4.2a and 4.2b . In fact, we can see that with reannealing, the variance is smaller when near-greedy policy is applied, i.e., when ε stays at its minimum for a while. Upon this, we could expect the (greedy) policy learned with reannealing strategy to be superior both in terms of higher total reward and smaller variance.



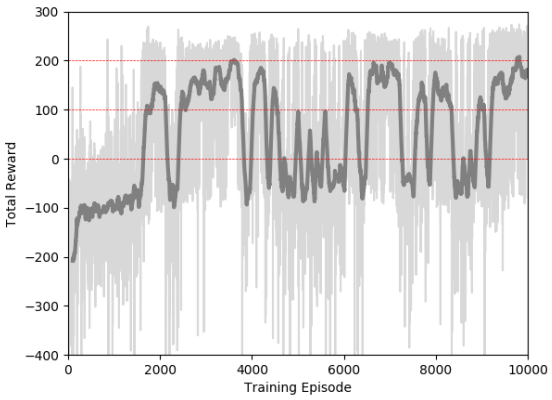
(a) no reannealing, $\rho_{decay} = 0.99$



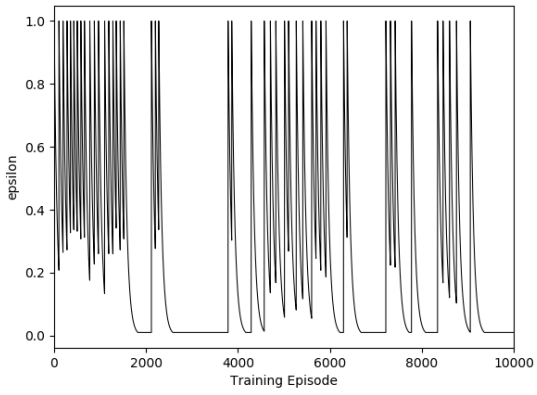
(b) no reannealing, $\rho_{decay} = 0.985$



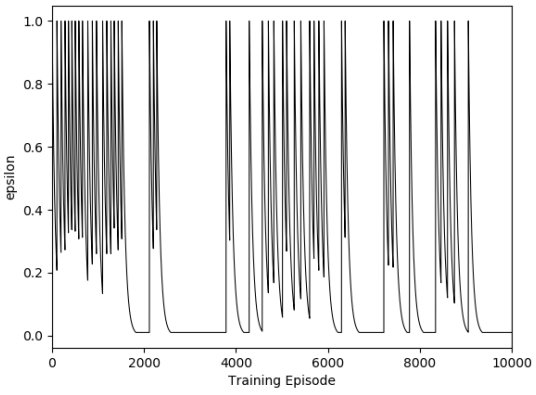
(c) with reannealing, $\rho_{decay} = 0.99$



(d) with reannealing, $\rho_{decay} = 0.985$



(e) progress of ε, $\rho_{decay} = 0.99$



(f) progress of $\rho_{decay} = 0.985$

Figure 4.2: Performances measured during training. The upper two rows illustrate the total rewards during each episode and moving averages; (a) and (b) correspond to training without reannealing, while (c) and (d) are with exploration reannealing. The bottom row plots the varying ε values along training with reannealing. In all cases the left column corresponds to exploration decay rate $\rho_{decay} = 0.99$, and the right column corresponds to $\rho_{decay} = 0.985$.

4.5 Conclusions

In this work we present a method to organize exploration in RL algorithms. In particular, we focus on its application to model-free value-based approaches, such as DQN. Our method is particularly suited to problems which suffer from poor local optima, and that have sparse rewards as well as long horizons which can trigger the termination criterion earlier. Poor local optima can often be easily distinguished from an outside perspective, yet it may be hard to encode this additional information into reward function or state variables due to complexity of the underlying system. Instead we propose to use a separate, heuristic measure, independent from the agents reward and state, aimed at detecting the local optima that need to be avoided. With such a measure, we can then organize the learning process using a reannealing framework, previously used to solve hard optimization problems.

We highlight some intuitive benefits of applying exploration reannealing, and demonstrate its performance on a standard RL task. In our experiments, reannealing method, indeed helps the agent avoid poor local optima and gather more useful information. The sample efficiency for the reinforcement learning is improved, and the data imbalance problem alleviated. As a result, the training procedure can is accelerated, and the derived policies have superior performance. In addition, we hypothesize that it can serve as a remedy for imperfect hyperparameter tuning.

It is worth noting that the simple framework presented here can be extended to use more sophisticated supervised learning-based heuristic measures for reannealing initiation. If trained properly, such strategies can result in even better performance, due to improved timing of reannealing.

Chapter 5

Cross Q-Learning in Deep Q-Networks

In this work, we propose a novel *cross Q-learning* algorithm, aim at alleviating the well-known overestimation problem in value-based reinforcement learning methods, particularly in the deep Q-networks where the overestimation is exaggerated by function approximation errors. Our algorithm builds on Double Q-learning, by maintaining a set of parallel models and estimate the Q-value based on a randomly selected network, which leads to reduced overestimation bias as well as the variance. We provide empirical evidence on the advantages of our method by evaluating on some benchmark environment, the experimental results demonstrate significant improvement of performance in reducing the overestimation bias and stabilizing the training, further leading to better derived policies.

5.1 Introduction

Overestimation has been identified as one of the most severe problems in value-based reinforcement learning (RL) algorithms such as Q-learning [152], where the maximization of value estimates induces a consistent positive bias, and the error of the estimates is accumulated by the nature of temporal difference (TD) learning. In the function approximation setting such as deep Q-networks (DQN), the issue of value overestimation is more severe, given the noise induced by the inaccuracy of the approximation. As a result, learning DQN tends to have instability and variability for estimated Q-values, the derived policies accroding to the overestimated Q-values tend to be not optimal and often diverge.

To overcome this issue, double Q-learning [131] has become a standard approach for training DQNs. The main purpose of double Q-learning is to avoid the overestimation problem for the target Q-value, by introducing negative bias from the double estimates.

The usual way to realize it in DQN is to maintain a target network which is a copy of the policy DQN which is either frozen for a period of time, or softly updated with exponential moving average. The target network then is used to estimate the TD target. This may alleviate the issue, however, double DQN still often suffer from overestimation in practice, partially because the policy and target estimates of Q-values are usually too similar, while the noise from high variance is propagated through the network and occasional large reward can produce great overestimation in the future. Another approach sometimes proposed is to impose a bias-correction term on the estimates for Q-learning [153], however, the error correction term is complicated to derive for deep networks, in which the finiteness of state space is no longer true. A more recent modification over double DQN favors underestimation and clips the Q-value estimates [154], that is, always chooses the minimum of the estimated targets over the two networks. The clipped double Q-learning is used on the critics in actor-critic methods for the deterministic policy gradient, which is referred to as TD3 (twin delayed deep deterministic policy gradient) and has shown state-of-the-art results on multiple tasks. However, the intentionally engineered underestimation lacks of rigorous theoretical guide, in addition, it may induce bias in the other direction, e.g., the underestimation can also accumulate through TD learning and derive suboptimal policies. Further, excessive underestimation can naturally lead to slower convergence.

Another direction to alleviate overestimation is through reducing the variance during training. For example, [155] uses the average of the learned estimated Q-values from multiple networks, which is designed to help reduce the target approximation error. There also exist various variance reduction techniques [156, 157, 158, 159] that focus on the general non-convex optimization procedure for accelerating the stochastic gradient descent, or their direct application on DQNs [160], in which the agent could obtain smaller approximated gradient errors. Reducing the variance can effectively stabilize the DQN training procedure, and overestimation alleviation can be seen as a by-product. However, these are indirect methods

for overestimation control, and the positive bias due to the max operator in TD update are not taken care of.

To address these concerns, we propose a cross DQN algorithm, which can be seen as a direct extension of an earlier variant of double DQN, but can be more flexible. In cross DQN, we maintain more than two networks, and update them one at a time based on the estimation from another randomly selected one. As mentioned above, the averaged DQN [155] calculates the average of K estimated Q-values, with the primary purpose of the overall variance reduction. For all K networks, each step of TD updates as well as action selections are based on combining the K estimates. Consequently, the networks are tangled together and cannot be implemented with a parallel simulation. In bootstrapped DQN [133], one of the K networks (or heads) is bootstrapped for each action selection step during training, aiming at encouraging exploration early on. Thus the simulation is not independent among networks, while the TD updates are totally independent within each of the networks, by using its own estimation of Q-values as in standard (double) DQN. [161] investigates more general applications of traditional ensemble reinforcement learning on policies, i.e., majority voting, rank voting, Boltzmann addition, etc. to combine the different policies derived from multiple networks, by which they called the target ensembles, in addition to the averaged DQN which they called the temporal ensemble. All of the above-mentioned work that maintain multiple networks have achieved better performance by addressing different issues through some particular settings. Our method focuses on the variation of TD updates, in which the target Q-values are estimated with a bootstrapped network for calculating the gradients, with the direct goal of reducing overestimation. Each of the K networks would perform its own TD updates, while maintaining flexibility in action selections: the networks can either interact with the environment independently, or through any other ensemble strategy. The detailed implementation options would be discussed in Section 5.4.

In supervised learning, ensemble strategies such as bagging, boosting, stacking, and hierarchical mixture of experts, etc. are commonly applied to achieve better performance, by

simultaneously learning and combining multiple models. All of the abovementioned algorithms that maintain multiple models, including ours, can be seen as special cases of general ensemble DQNs. But our method has a deeper root in resampling and model selection. By bootstrapping another model to assess the values of current model, we introduce model bias for in-sample estimations, but reduce the variance of out-of-sample estimations (i.e., the squares of out-of-sample bias), in other words, the trained model can generalize better and alleviate overfitting. For squared errors, this can be expressed as the well-known bias-variance trade-off: MSE = Irreducible Error^2 + Bias^2 + Variance. In value-based reinforcement learning, the model easily overfits due to overestimation (which is caused by the max operator) during learning. Cross Q-learning introduces underestimation bias, and further reduces the variance, thus improves the generalization of the trained model.

Like in [154], our work can be naturally extended to the state of the art actor-critic methods in continuous action space, such as the deep deterministic policy gradient [162], in which the critic network(s) are learned to give an estimate of the Q-value for the actor network to update its gradient and derive policies. Usually multiple critic networks are applied, however, rather than accumulating their learned gradients (either synchronously or asynchronously [163]) and optionally sharing network layers, no other information is shared among the critics. The extension of our method allows the critics to share their value estimates and utilize that of others, which leads to more accurate estimation of each critics, thus can improve the performance of these models. Similar to these actor-critic algorithms, our work can be implemented for parallel training easily, and the exchange of information among networks could take place either synchronously or asynchronously like the accumulation of gradients, as there is always tradeoff between synchronous and asynchronous update.

The rest of this chapter is organized as follows. In Section 5.2, we formally define the estimators for the maximum expected values, along with their theoretical properties. The convergence of our cross estimator is shown in Section 5.3. Section 5.4 illustrates our cross

DQN algorithm directly derived from the double DQN in details. We show some empirical results in Section 5.5. Finally, Section 5.6 draws conclusions and discusses future work.

5.2 Estimating the Maximum Expected Values

For Q-learning, the action is selected according to the estimated target Q-values. This is an instance of a more general maximum expected value estimation problem, which is formed as follows. Consider a set of $|\mathcal{A}|$ random variables $Q = \{Q_{a_1}, \cdots, Q_{a_{|\mathcal{A}|}}\}$, we are interested in finding the maximum expected value among the set of variables, which is defined as

$$\max_a \mu_a = \max_a \mathbb{E}[Q_a]$$

while each $\mathbb{E}[Q_a]$ is usually estimated from samples. Let Ω_a denote the sample space for estimating Q_a, for $a \in \mathcal{A}$, and we further assume that the samples in Ω_a are i.i.d. The sample mean $\hat{\mu}_a = \frac{1}{|\Omega_a|} \sum_{x \in \Omega_a} x$ is then an unbiased estimator for $\mathbb{E}[Q_a]$.

Let $f_a : \mathbb{R} \to \mathbb{R}$ be the probability density function (PDF) for the variable Q_a, and $F_a(x) = \int_{-\infty}^{x} f_a(x)dx$ be the cumulative density function (CDF). The maximum expected value is then

$$\max_a \mathbb{E}[Q_a] = \max_a \int_{-\infty}^{\infty} x f_a(x) dx. \tag{5.1}$$

5.2.1 (Single) Maximum Estimator

The most straightforward way to approximate $\max_a \mathbb{E}[Q_a]$ is to take the maximum over the sample mean for each a, i.e., $\max_a \mathbb{E}[Q_a] \approx \max_a \bar{q}_a$. Note that the sample means $\bar{q}_a$ are unbiased estimates of the true means, thus $\max_a \bar{q}_a$ is an unbiased estimate for $\mathbb{E}[\max_a \mu_a] = \int_{-\infty}^{\infty} x f_{\max}(x) dx$, however, it is a biased estimate for $\max_a \mathbb{E}[Q_a]$.

Consider its CDF $F^{\mu}_{\max} = P\{\max_a \hat{\mu}_a \leq x\} = \Pi_a P\{\mu_a \leq x\} = \Pi_a F^{\mu}_a(x)$, we can write

$$\mathbb{E}[\max_a \hat{\mu}_a] = \int_{-\infty}^{\infty} x \frac{d}{dx} \Pi_a F^{\mu}_a(x) dx = \sum_{a'} \int_{-\infty}^{\infty} x f^{\mu}_{a'}(x) \Pi_{a' \neq a} F^{\mu}_a(x) dx. \tag{5.2}$$

Comparing equations (5.1) and (5.2), clearly $\max_a \mathbb{E}[Q_a]$ and $\mathbb{E}[\max_a \hat{\mu}_a]$ are note equivalent. Moreover, the product term $\Pi_{a' \neq a} F_a^{\mu}(x)$ in the integral introduces positive bias (since CDFs are monotonically increasing, the sum of their derivatives will be positive, the integral value would be monotonically increasing while more product terms are added). Therefore, we say that the expected value of the single estimator for the maximum is an overestimation of the maximum expected value.

5.2.2 Double Estimator

Consider the case that we use two sets of estimators $\hat{\mu}^A = \{\hat{\mu}^A_{a_1}, \cdots, \hat{\mu}^A_{a_{|\mathcal{A}|}}\}$ and $\hat{\mu}^B = \{\hat{\mu}^B_{a_1}, \cdots, \hat{\mu}^B_{a_{|\mathcal{A}|}}\}$, in which each $\hat{\mu}^A_a$ and is estimated from a set of samples independent of the one to estimate $\hat{\mu}^B_a$, i.e., $\hat{\mu}^A_a = \frac{1}{|\Omega^A_a|} \sum_{x \in \Omega^A_a} x$, $\hat{\mu}^B_a = \frac{1}{|\Omega^B_a|} \sum_{x \in \Omega^B_a} x$, and $\Omega^A_a \cap \Omega^B_a = \emptyset$. For all a, both $\hat{\mu}^A_a$ and $\hat{\mu}^B_a$ are unbiased estimator for $\mathbb{E}[Q_a]$, assuming all the samples in both sets are independently drawn from the population. That means $\mathbb{E}[\hat{\mu}^A_a] = \mathbb{E}[Q_a]$ for all a, including $a^*_B = \text{argmax}_a \hat{\mu}^B_a$, the action that maximizes the sample mean $\hat{\mu}^B$. Therefore, $\hat{\mu}^A_{a^*_B}$ can be used to estimate $\max_a \mathbb{E}[\hat{\mu}^A_a]$ as well as $\max_a \mathbb{E}[Q_a]$, i.e.,

$$\max_a \mathbb{E}[Q_a] = \max_a \mathbb{E}[\hat{\mu}^A_a] \approx \hat{\mu}^A_{a^*_B}.$$

The same argument holds for the opposite way considering the best action over Ω^A and the sample mean $\hat{\mu}^B_{a^*_A}$. The selection of a^* means that all other a gives lower estimation, i.e., $P(a = a^*) = \Pi_{a \neq a^*} P(\mu^A_a < \mu^A_{a^*})$. Let f^A_a and F^A_a be the PDF and CDF of μ^A_a, respectively. Then

$$P(a = a^*) = \int_{-\infty}^{\infty} P(\mu^A_a = x) \Pi_{a' \neq a} P(\mu^A_A < x) dx = \int_{-\infty}^{\infty} x f^A_a(x) \Pi_{a' \neq a} F^A_a(x) dx.$$

The expected value of double estimator is a weighted sum of the sample means' expected values in one sample space, weighted by the probability of each sample mean to be the

maximum in the other sample space, i.e.,

$$\sum_a P(a = a^*)\mathbb{E}[\mu_a^B] = \sum_a \mathbb{E}[\mu_a^B] \int_{-\infty}^{\infty} x f_a^A(x) \Pi_{a' \neq a} F_a^A(x) dx.$$

Double estimator gives us negative bias, since the weights $P(a = a^*)$ are probabilities, which are positive and sum to 1, the maximum expected value then serves as an upper bound for the weighted sum, as some weights may also be given to variables whose expected value is less than the maximum.

5.2.3 Cross Estimator

We can easily extend the double estimator to a more general case, in which instead of using two sets of estimators, suppose now we have K independent sets of estimators $\hat{\mu^1}, \cdots, \hat{\mu^K}$. We call it the cross estimator. The double estimator can be seen as a special case of the more general cross estimator. Similar argument as analyzing the double estimator can be applied here, for any two estimators $\hat{\mu^i}$ and $\hat{\mu^j}$, as

$$\max_a \mathbb{E}[Q_a] = \max_a \mathbb{E}[\hat{\mu}_a^i] \approx \hat{\mu}_{a_j^*}^A.$$

The cross estimator finally uses a convex combination of the K sample means,

$$\sum_a P(a = a^*)\mathbb{E}[\mu_a^j] = \sum_a \mathbb{E}[\mu_a^j] \int_{-\infty}^{\infty} x f_a^i(x) \Pi_{a' \neq a} F_a^i(x) dx,$$

thus also underestimates the maximum expected value.

Theorem 5.1 (Van Hasselt [164])**.** *There does not exist an unbiased estimator for maximum expected values.*

5.3 Convergence in the Limit

In this section, we first present a lemma which claims the convergence of SARSA from [165], and then use it to prove convergence of cross Q-learning. Note that this part heavily borrows the proof of the convergence of double Q-learning [166], but serves as a more general case.

Lemma 5.2 (Singh et al. [165]). *Consider a stochastic process* $(\alpha_t, \Delta_t, F_t), t \geq 0$, *where* α_t, Δ_t *and* $F_t : X \to \mathbb{R}$ *satisfy the equation*

$$\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \alpha_t(x)F_t(x), \qquad \text{where } x \in X, t = 0, 1, 2, \cdots$$

Let P_t *be a sequence of increasing* σ*-fields such that* α_0 *and* Δ_0 *are* P_0*-measureable and* α_t, Δ_t *and* F_{t-1} *are* P_t*-measurable, for* $t = 1, 2, \cdots$.

Δ_t *converges to zero with probability one (w.p.1) if the following hold:*

1. *the set* X *is finite.*
2. $0 \leq \alpha_t(x) \leq 1, \sum_t \alpha_t(x) = \infty$, *and* $\sum_t \alpha_t^2(x) < \infty$ *w.p. 1.*
3. $||\mathbb{E}[F_t|P_t]|| \leq \kappa||\Delta_t|| + c_t$, *where* $\kappa \in [0, 1]$ *and* c_t *converges to zero w.p. 1.*
4. $Var(F_t|P_t) \leq K(1 + ||\Delta_t||)^2$, *where* K *is a constant.*

in which $||\cdot||$ *denotes the maximum norm.*

Theorem 5.3. *In a given ergodic MDP, suppose that we have a set of* K *Q-value functions,* $Q^1, Q^2, \cdots, Q^K$, *as updated by cross Q-learning, will converge to the optimal value function* Q^* *with probability 1, if the following conditions hold:*

1. *The MDP is finite, i.e.,* $|\mathcal{S} \times \mathcal{A}| < \infty$.
2. $\gamma \in [0, 1)$.

3. *The Q-values are stored in a lookup table.*

4. *Each state-action pair is visited infinitely often.*

5. *Each Q^k receives an infinite number of updates, for all $k = 1, \cdots, K$.*

6. $0 \leq \alpha_t(s,a) \leq 1, \sum_t \alpha_t(s,a) = \infty$, *and* $\sum_t \alpha_t^2(x) < \infty$ *w.p. 1. Moreover,* $\alpha_t(s,a) = 0, \forall (s,a) \neq (s_t, a_t)$.

7. $Var(R(s,a)) < \infty, \forall s, a$

Proof. Let $k, j \in \{1, \cdots, K\}$ are randomly picked with $k \neq j$. Apply Lemma 5.2 by letting $P_t = \{Q_0^1, Q_0^2, \cdots, Q_0^K, s_0, a_0, \alpha_0, r1, s1, \cdots, s_t, a_t\}$, $X = \mathcal{S} \times \mathcal{A}$, $\Delta_t = Q_t^k - Q^*$, and $F_t(s_t, a_t) = r_t + \gamma Q_t^j(s_{t+1}, a^*) - Q_t^*(s_t, a_t)$, where $a^* = \text{argmax}_a Q^k(s,a)$. The first two conditions of Lemma 5.2 hold immediately from conditions 1 and 6 of Lemma 5.3, respectively. And since condition 7 of Theorem 5.3 gives us the bounds for the variance of rewards, the fourth condition of Lemma 5.2 holds.

To show the third condition of Lemma 5.2, we write

$$\begin{aligned} F_t(s_t, a_t) &= r_t + \gamma Q_t^j(s_{t+1}, a^*) - Q_t^*(s_t, a_t) \\ &= \left(r_t + \gamma Q_t^k(s_{t+1}, a^*) - Q_t^* s_t(s_t, a_t)\right) + \gamma \left(Q_t^j(s_{t+1}, a^*) - Q_t^k(s_{t+1}, a^*)\right) \\ &= F_t^Q(s_t, a_t) + \gamma c_t \end{aligned}$$

in which we define $F_t^Q = r_t + \gamma Q_t^k(s_{t+1}, a^*)$ as the estimated Q-value for (s,a) under the standard (single) Q-learning. While the convergence of standard Q-learning in finite MDP is well-known, i.e., $\mathbb{E}[F_t^Q | P_t] \leq \gamma ||\Delta_t||$, it suffices to show that $c_t = Q_t^j(s_{t+1}, a^*) - Q_t^k(s_{t+1}, a^*) \to 0$, so that the condition on the expected contraction of F_t holds.

Let $\Delta_t^{jk}(s_t, a_t) = Q_t^j(s_t, a_t) - Q_t^k(s_t, a_t)$. It is important to note that at each step, the choice of j, k is random, all with equal probability $p_{jk} = p_{jk} = 1/\binom{K}{2}$. Consider the case

that Q^k is updated using Q_t^j at time t, the update of Δ^{kj} is

$$\begin{aligned}\Delta_{t+1}^{jk}(s_t, a_t) &= \Delta_t^{jk}(s_t, a_t) + \alpha_t(s_t, a_t)\left(r_t + \gamma Q_t^j(s_{t+1}, a^*) - Q_t^k(s_t, a_t)\right) \\ &= \Delta_t^{jk}(s_t, a_t) + \alpha_t(s_t, a_t) F_t^k(s_t, a_t)\end{aligned}$$

Or, again with probability $p_{kj} = 1/\binom{K}{2}$, we use Q^k to update Q^j, in this case we have

$$\begin{aligned}\Delta_{t+1}^{jk}(s_t, a_t) &= \Delta_t^{jk}(s_t, a_t) - \alpha_t(s_t, a_t)\left(r_t + \gamma Q_t^k(s_{t+1}, a^*) - Q_t^j(s_t, a_t)\right) \\ &= \Delta_t^{jk}(s_t, a_t) - \alpha_t(s_t, a_t) F_t^j(s_t, a_t)\end{aligned}$$

Otherwise, this particular (j, k) pair is not selected at time t, and the update of Δ^{kj} is then zero. Then

$$\begin{aligned}\mathbb{E}\left[\Delta_{t+1}^{jk} | P_{t+1}\right] &= \\ &= (1 - 2p_{jk})\mathbb{E}\left[\Delta_t^{jk} | P_t\right]\end{aligned}$$

Clearly $\mathbb{E}\left[\Delta_{t+1}^{jk} | P_t\right]$ converges to 0 since the coefficient on the R.H.S. is less than 1. Therefore we have shown that $c_t \to 0$ since $\Delta_t^{jk} \to 0$ in expectation and j, k are randomly chosen. It in turn ensures condition 3 of Lemma 5.2 holds, which completes our proof. □

Finally, we rephrase Theorem 5.3 as follows:

Proposition 5.4. Cross estimation converges in the limit, given finite and ergodic MDP.

5.4 Cross DQN

In this section, we elaborate our proposed cross Q-learning method and its variants. Cross DQN serves as an extension to the double DQN algorithm [131], which has been used as the default setting for most state-of-art DQN training.

Double DQN was proposed in the aim of reducing overestimation bias, in which the target network simply is a delayed-updated copy of the current network. Note that the original vanilla DQN also uses two networks, the purpose of periodic frozen and update of the target network is to stabilize learning. Specifically, in vanilla DQN, the target network is used to evaluate both the action and the value, i.e.,

$$y \leftarrow r + \gamma Q_{\theta'}(s', a'_*) \qquad \text{where } a'_* = \text{argmax}_{a'} Q_{\theta'}(s', a') \tag{5.3}$$

On the other hand, in double DQN, the current network is used to evaluate the action and select a', while the target network is used for evaluate the value, so that action selection is decoupled from estimation of the target:

$$y \leftarrow r + \gamma Q_{\theta'}(s', a'_*) \qquad \text{where } a'_* = \text{argmax}_{a'} Q_{\theta}(s', a') \tag{5.4}$$

In practice however, it is common the case that little improvement can be gained by using double DQN, since the current and target networks are usually too similar due to slowly changed parameters in neural network models with SGD optimization. We can neither set the period of updating target too long, otherwise the derived policy would not exhibit learning and progress. As a result, double DQN does not entirely eliminate the overestimation bias. In Section 5.5, we will further experimentally show the elimination of overestimation is not effective nor sufficient in double DQN.

Instead of maintaining only two separate networks, we will use a set of K models for estimating Q-values and selecting actions in our cross Q-learning. While update each network's parameters, we will calculate its TD target Q-value using one of the other $K-1$ models. More specifically, let the network with parameters we are about to adjust be our current network (θ_i), and we randomly pick another network to be our target network (θ_j, e.g., $j \in U[1, K]$). To compute the target Q-value, we will use the current network to evaluate the actions and select a' in the next state s', while the value is evaluated by using

the target network, i.e.,

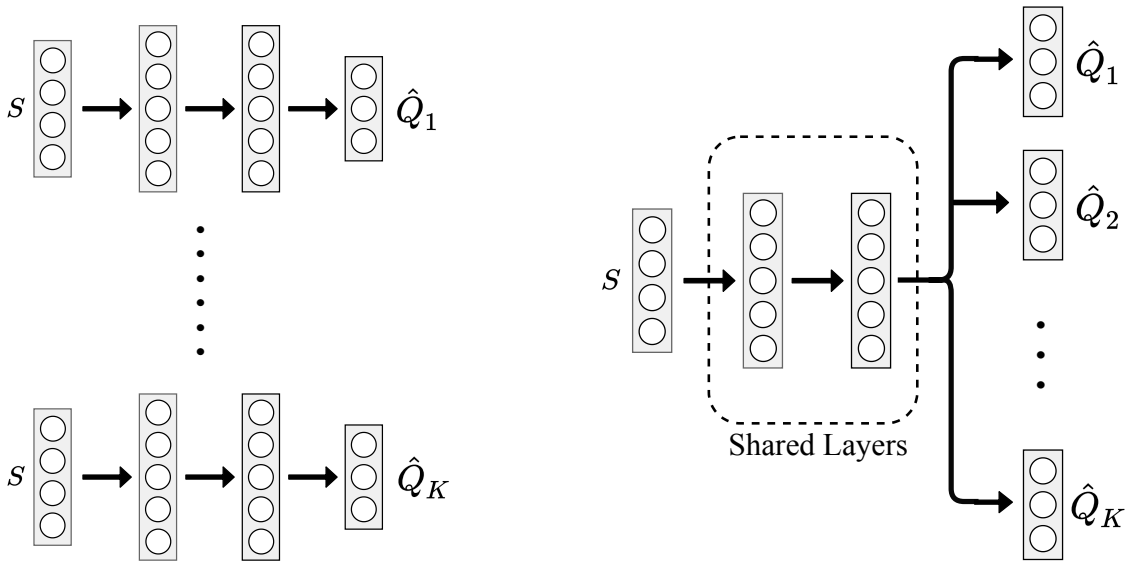
$$y \leftarrow r + \gamma Q_{\theta_j}(s', a'_*) \qquad \text{where } a'_* = \text{argmax}_{a'} Q_{\theta_i}(s', a') \tag{5.5}$$

Algorithm 3 Cross-Learning DQN

Initialize $K \in \mathbb{N}_+$ different Q-functions $Q(s, a|\theta^k)$ with random parameters θ^k for $k = 1, \cdots, K$.
Initialize replay buffer $\mathcal{B}$.
for each episode **until** end of learning **do**
  Initialize state s
  for step $t = 1, \cdots$ **until** s is terminal state of an episode **do**
    Select action a_t according to Q with exploration, e.g., $a_t = \text{MajorityVote}\{\text{argmax}_a Q_k(s, a)\}_{k=1}^K$
    Take action a_t, observe reward r and next state s'
    Store experience tuple $< s, a_t, r, s' >$ into $\mathcal{B}$
    Sample a mini-batch of experiences from $\mathcal{B}$.
    for all sampled experience in the mini-batch **do**
      To train network Q^i, compute $a' = \text{argmax}_a Q^i(s', a|\theta^i)$
      Randomly pick another network Q^j to estimate TD target $y = r + Q^j(s', a'|\theta^j)$
      Backpropagate TD error $\delta = y - Q^i(s, a|\theta^i)$ through Q^i, update θ^i with learning rate α_t
    end for
    $s \leftarrow s'$
  end for
end for

In implementation, we have flexibility and various options in how to utilize the K different Q-networks. There always exist tradeoffs among different choices that we need to consider in order to pick the one that meets our goal most. For example, we can have different design of neural network architectures. A natural choice of having K independent models is to maintain a list of separate neural networks with the same architecture. With K separate models, the difference between their outputs (i.e., K streams of Q-values derived from the same (s, a)-pair as the input) comes from different random parameter initialization of each model, also is due to that different data that each model is trained upon, i.e., for each step of backpropagation, each model randomly samples a mini-batch of experiences and performs SGD optimization with the mini-batch. Moreover, maintaining K model copies implies that not only the storage for the models would be K times large as a single network, also the forward propagation would take K times amount of computations. Instead, we can utilize the shared network design for the K models, in which the K models shared their weights

except for the last layer, which consists of K value function heads from which the value functions $Q_k(s, a|\theta_k)$ are derived, and the weights on the last layer are generally different. Thus we have much less parameters in total to be trained, and the computational burden can be greatly alleviated. Moreover, as recent deep learning research reveals, the first few layers of neural network are mainly about representations learning, the shared layers provide the same features expressed for computing Q, this can be seen as online transfer of learned knowledge among models. Note that in shared learning settings, in order to avoid premature learning and suboptimal convergence, the gradients of the network except the last layer are usually normalized by $1/K$, but this also results in slower learning early on. On the other hand, the separate models are simpler yet provide more variability in Q-values, also are more stable during training. In addition, when we train the networks in distributed system, the separate networks do not depend on others' weights thus can be learned independently, which requires much less information exchange and this could be a huge advantage for distributed learning. The comparison of the separate and shared network architectural design is shown in Figure 5.1.



(a) Separated Network Design (b) Shared Network Design

Figure 5.1: Separate and Shared Network Architecture

With K different models (or heads), while each could derive a possibly different policy, there is no doubt that during test phase we should take advantage of ensembles, for instance by choosing the action with the majority votes across the outputs. However, we can make choices on how to combine action selections into a single policy during training. With ensemble action selection such as majority voting, the derived policy is often superior than any individual one, thus greatly reduces the variance during training, as we will experimentally show in Section 5.5. This in turn refines exploitation, results in great variance reduction of Q-values and speeds up learning. Note that to deal with exploration-exploitation dilemma, ε-greedy strategy is needed to encourage exploration. On the other hand, we may also randomly pick a single network from the K models, and act as it suggests during training. This falls into the paradigm of Bootstrapped DQN [133], which encourages exploration, in the cost of slower early learning (see Section 5.5), but may learn better policy later with more exploration. Another advantage of bootstrapped action selection is that it can slightly reduce computational burden, since instead of forward passing and computing all K of the Q-values for action selection, we can calculate only one of them. The procedure of bootstrapped version of cross DQN is presented in Algorithm 4.

Algorithm 4 Bootstrapped Cross DQN

Initialize $K \in \mathbb{N}_+$ different Q-functions $Q(s, a|\theta^k)$ with random parameters θ^k for $k = 1, \cdots, K$.
Initialize replay buffer $\mathcal{B}$.
for each episode **until** end of learning **do**
    Initialize state s
    Randomly pick a network Q^k to act, where $k \in \{1, \cdots, K\}$.
    for step $t = 1, \cdots$ **until** s is terminal state of an episode **do**
        Select action $a_t = \text{argmax}_{a'} Q^k(s, a')$ with exploration
        Take action a_t, observe reward r and next state s'
        Store experience tuple $< s, a_t, r, s' >$ into $\mathcal{B}$
        Sample a mini-batch of experiences from $\mathcal{B}$.
        for all sampled experience in the mini-batch **do**
            To train network Q^i, compute $a' = \text{argmax}_{a'} Q^i(s', a'|\theta^k)$
            Randomly pick another network Q^j to stimate TD target $y = r + Q^j(s', a'|\theta^j)$
            Backpropagate TD error $\delta = y - Q^i(s, a_t|\theta^i)$ through Q^i, update θ^i with learning rate α_t
        end for
        $s \leftarrow s'$
    end for
end for

Another choice we can make is the training frequency. In our cross DQN settings, when backpropagation occurs, we can either choose to train on a single network (e.g., the single model that provides the action selection), or each of the K networks could independently sample a mini-batch of experiences and perform SGD optimization. The latter would increase the sample efficiency and speed up learning, while the former would reduce the computational burden, in which the number of backpropagation (which is the most computational expensive) remains the same as in a single DQN. In addition, with the former setting, our cross Q-learning does not require maintaining copies of the networks as the target. Experimentally, we found that freezing targets merely has any effect on stabilization of learning, but only costs doubled memory for model storage. This is due to two reasons. First, we bootstrap a model that is different than the current one, when $K \geq 2$, the variety of models ensures the difference in parameter initialization, as well as the difference of mini-batch data their learning based upon, which in turn ensures the independence of Q-value estimates. Secondly, with less frequent update of each network, the bootstrapped target Q-value changes less as well, also helps stabilize learning.

5.5 Experimental Results

In this work, we conducted experiments on two classical control problems, CartPole and LunarLander, for extended tests. We selected these testbeds in the aim of covering different challenges, especially in terms of complexity. As both environments interfaced through OpenAI gym environment [151], unless specified otherwise. The neural networks have a number of hyperparameters. The combinatorial space of hyperparameters is too large for an exhaustive search, therefore we have performed limited tuning. For each component, we started with the same settings as in [167] in order to make comparisons with states of the art results.

5.5.1 CartPole

Experimental Setup

The CartPole, also known as an inverted pendulum, in which a pole (or pendulum) is attached by an un-actuated joint to a cart (i.e., the pivot point). The pendulum starts upright at the center of a 2D track but is unstable since the center of gravity is above the pivot point. The goal of this task is to keep the pole balanced and prevent it from falling over, by applying appropriate force to the pivot point, while the force could move the cart along the frictionless track with finite length of 4.8 units. An immediate reward of $+1$ is provided for every timestep that the pole remains not falling over, and the maximum cumulative rewards in an episode are clipped to 200. An episode also ends when the pole is slanted with degree $> 15^\circ$ from vertical, or the cart moves out of the track [168]. In each timestep, the agent is provided with current state $s \in \mathbb{R}^4$, which represents cart position, cart velocity, pole angle, and pole angular velocity, respectively. A unit force either from left or right can be applied, thus the actions are discrete with $a \in \{-1, 0, +1\}$.

As in [167], we approximate the Q-values using a neural network with two fully-connected hidden layers (which consist of 64 and 32 neurons, respectively). We train each of the neural networks for 1000 episodes (approximately a little less than 200000 steps), with a FIFO memory of size 5×10^4 transitions for experience replay. A target network is updated every 500 steps to further stabilize learning. The adaptive moment estimation (Adam) optimizer with learning rate $\alpha = 0.001$ is used to train the network, since it is in general less sensitive to the choice of the learning rate than other stochastic gradient descent algorithms [119]. The optimization is performed on mini-batches of size 32, sampled uniformly from the experience replay. The discount factor γ is set to 0.99, and ε-greedy policy is used for choosing actions throughout interacting with the environment, which starts with exploration $\varepsilon = 1$, and annealed to 0.02 in the first 10000 steps.

After every 20 training episodes, we conduct a performance test that plays 10 full episodes using the greedy policy deterministically derived from the current network. For the models with $K > 1$, majority voting is used for the action selection disregard whether or not bootstrapped Q-value head is used during training. The cumulative rewards of each test episode are used for comparison among different models. Moreover, in order to comparing the estimation of Q-values among models, every 20 training episodes, we randomly sample a batch of historical 1024 (s, a)-pairs from the replay buffer and compute their Q-values using current network. More than one thousand samples ensure that their mean is somewhat representative for Q-values under current model.

Analysis of Cross Q-learning Effects

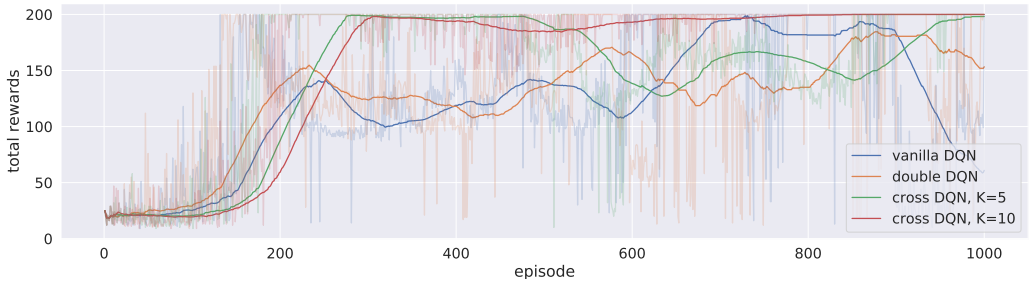
We compared our cross Q-learning algorithms with vanilla DQN and double DQN. Note that vanilla DQN uses single estimators, while double DQN uses double estimators, and our cross DQN uses cross estimators. $K = 5$ and $K = 10$ are used in cross DQNs. Figure 5.2(a) illustrate the training history of episodic total rewards of the four models, from which we can see that although with a single network (vanilla and double DQNs), the agent starts to learn early on with less samples, in particular, double Q-learning helps the single network to learn even faster, however, the learned models are not stable. With cross Q-learning, although the networks learn slower at the beginning, in particular, cross DQN with $K = 10$ started to learn even later than cross DQN with $K = 5$, once cross DQNs start to learn, the performance improvement is substantial. Not only the total rewards are higher, the learning is also much more stable. After 300 episodes, the training total rewards converge to 200 for $K = 10$ cross DQN, with little variation (due to ε exploration). $K = 5$ cross DQN has more variation, but it also seems to converge after 900 episodes, while vanilla DQN and double DQN are easily deteriorated, and have much larger variations.

The performance improvement can be more clearly seen in Figure 5.2(b). After 300 episodes of training, the policies derived cross DQN with $K = 10$ become more and more

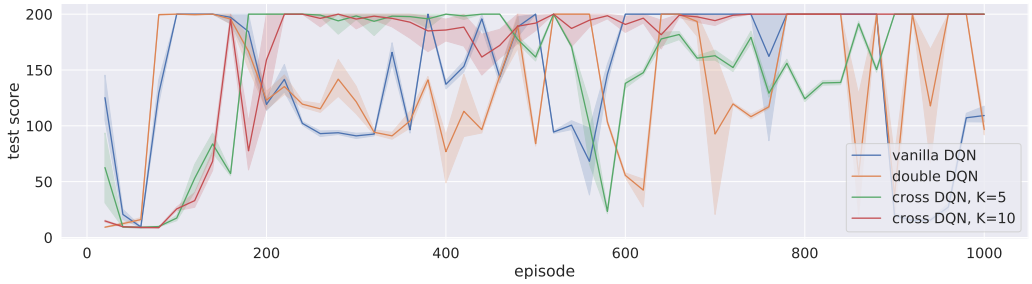
stable, the variance of test total rewards become zero close to the end of training. Cross DQN with $K = 5$ deteriorates after 500 episodes of training, but later it also learns to derive stable policy that has total rewards of 200 with tiny variances. Whereas the policies derived from vanilla DQN and double DQN can only get score which is approximately half of cross DQNs, and with large variances. The policy derived from double DQN seems to be a little better than that from vanilla DQN, but the improvement is not as significant as that of using cross Q-learning.

Furthermore, part of the reason for slower start of cross DQN is due to our learning settings, in which we only perform SGD optimization on one of the networks (or heads). In other words, we reduce the learning frequency of each network (or head) down to $1/K$ to alleviate the computational effort, at the cost of slower start on learning. If we increase the learning frequency (i.e., backpropagate for each of the K networks/heads every time), the learning should be faster.

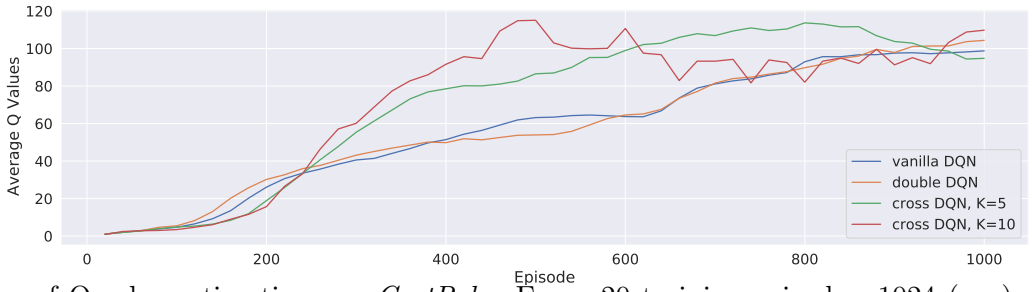
We also plot the average Q-values from bootstrapped 1024 (s, a)-pairs as shown in Figure 5.2(c). We observe that the beginning of learning, vanilla DQN has highest estimates of Q-values, which is an evidence of overestimation. The estimates from double DQN is lower, but only for limited amount, therefore we say that double Q-learning may have not solve the overestimation problem completely. Cross DQNs have quite smaller estimations at the beginning, in particular, as K gets larger, the estimates of Q-values become even lower. Overestimation is clearly an obstacle of effective learning, as a result, the estimated Q-values from cross DQNs are substantially higher than that from vanillar or double DQNs, since cross DQNs has derived better policies and obtained higher rewards. The Q-values estimates from cross DQNs start to converge after the derived policies stabilized, At the end of training, the estimated Q-values from the four different models are about at the same level, however, note that the estimates from vanilla and double DQNs continue increasing, and their derived policies are not stable, also have lower rewards. Our cross Q-learning algorithm has addressed the overestimation problem better.



(a) Learning curves



(b) Model test performance. Every 20 training episodes, 10 full test episodes were conducted.

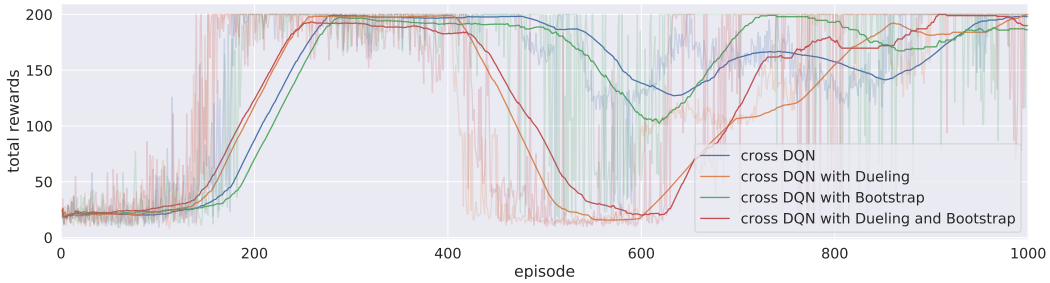


(c) Mean of Q-value estimations on *CartPole*. Every 20 training episodes, 1024 (s, a)-pairs were bootstrapped.

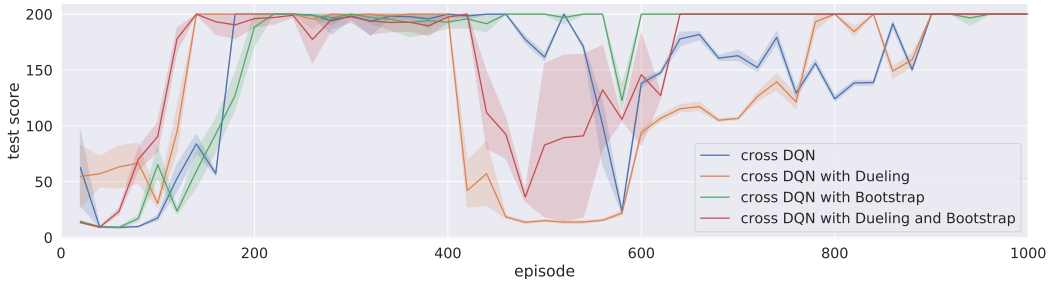
Figure 5.2: Comparison of vanillar DQN, double DQN and cross DQNs of $K = 5$, $K = 10$ on *CartPole*.

Effects of dueling DQN & Bootstrapped DQN

As the cross learning architecture shares the same input-output interface with standard DQN, we can recycle many recent advances in DQN research. We have mentioned one variant in Section 5.4 that it can combined with Bootstrapped DQN for action selection during training, while in Secction 5.5.1, our experiments for cross DQN are based on majority voting



(a) Learning curves

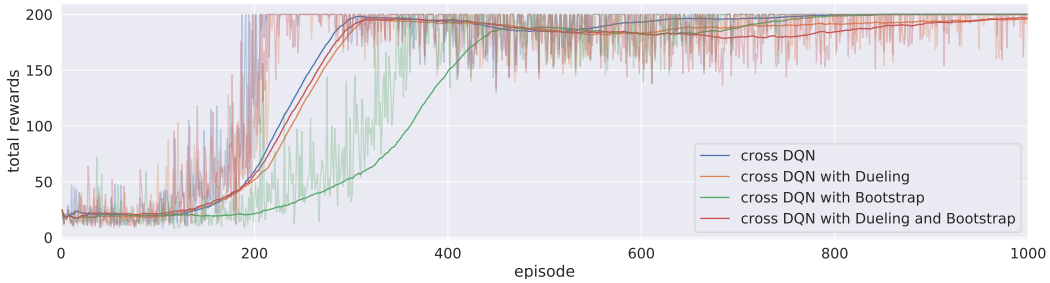


(b) Model test performance. Every 20 training episodes, 10 full test episodes were conducted.

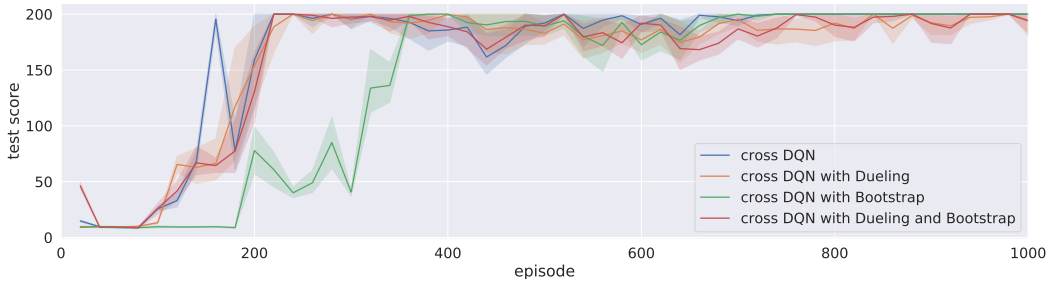
Figure 5.3: Comparison of cross DQNs of $K = 5$. Cross DQN with ensemble voting, with dueling DQN and voting, with bootstrapped DQN, and with both dueling & bootstrapped DQN on *CartPole*.

from K different Q-functions. Furthermore, it is convenient to combine the dueling architecture into each of the K networks. The goal of dueling DQN is to reduce variance for Q-value estimation, by subtracting a baseline and emphasizing the advantages among different actions, thus accelerates learning effectively. The variance reduction is performed on a single network's estimation, while our cross Q-learning reduces variance from a different perspective. For each network, the target values were calculated with other models by bootstrapping from multiple Q-values, thus introduces some bias. Due to the bias-variance tradeoff, however, the variance of our estimates decreases, and thus the overall error becomes smaller. In addition, the maximum operator induces overestimation bias, while cross-estimator tends to introduce bias in the other direction, thus greatly alleviates overestimation problem.

Figure 5.3 and Figure 5.4 illustrate the training and testing performance of cross DQN with different architectures, for the cases of $K = 5$ and $K = 10$, respectively. We can see that dueling architecture speeds up early on learning effectively, without hurting the



(a) Learning curves



(b) Model test performance. Every 20 training episodes, 10 full test episodes were conducted.

Figure 5.4: Comparison of cross DQNs of $K = 10$. Cross DQN, with dueling DQN, with bootstrapped DQN, with both dueling & bootstrapped DQN on *CartPole*.

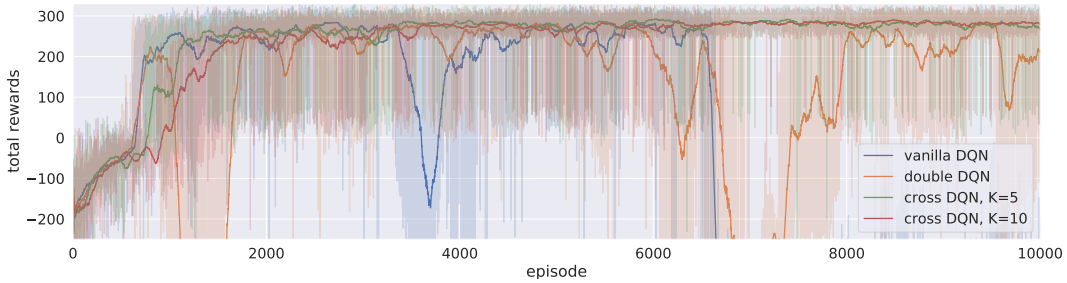
model performance later in general. On the other hand, Bootstrapped DQN slows learning at beginning, especially when K is large, since the selected actions varies among networks at beginning quite a bit. For example, the $K = 10$ cross DQN with bootstrap converges around 400 episodes while the other cross learning agents converges before 200 episodes. But after learned something, the bootstrapped action selection won't hurt the model. In fact, it might help learning for more complicated tasks because of more exploration early on. At least, using bootstrapped DQN can help our cross DQN agent make faster action selection during training and reduce computational burden slightly, since instead of calculate all K Q-values, we can calculate only one of them. Moreover, by comparing the learning curves of bootstrapped cross DQNs with different Ks, we can conclude that it is primarily our cross Q-learning rather than policy ensemble that greatly reduces the variance, as with $K = 10$ the variations are much smaller that that with $K = 5$, though policy ensemble further reduces the variance greatly, and during testing phase, our agent can definitely benefit from ensemble

of multiple models. Naturally combined crossed Q-learning with dueling and bootstrapped DQN, our model aggregates the merits from all three perspectives.

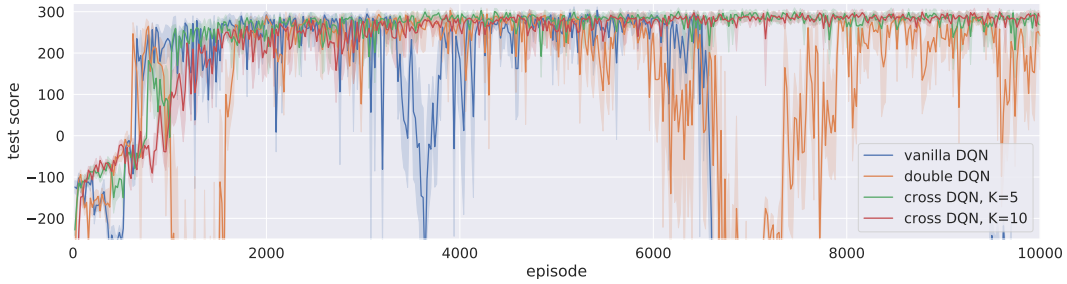
5.5.2 Lunar Lander

The task of Lunar Lander in Box2D [150] is to land the spaceship between the flags smoothly. The details of this task is discussed in Section 4.4.1. To solve this problem with cross DQNs, we build each network with two fully-connected hidden layers, which consist of 128 and 64 neurons, respectively. We train each of the neural networks for 10000 episodes for the LunarLander task, with a much larger replay buffer of size 10^6. The target network update is set to every 1000 steps for vanilla and double DQN, and learning rate $\alpha = 0.001$ and batch size of 64 are used for Adam optimizer to train all the models. The discount factor γ is again 0.99, and exploration rate ε is set to annealed to 0.02 in the first 100000 steps. And again, Q-values for bootstrapped 1024 (s, a)-pairs are evaluated and 10 episodes of performance tests with current policy are conducted every 20 training episodes.

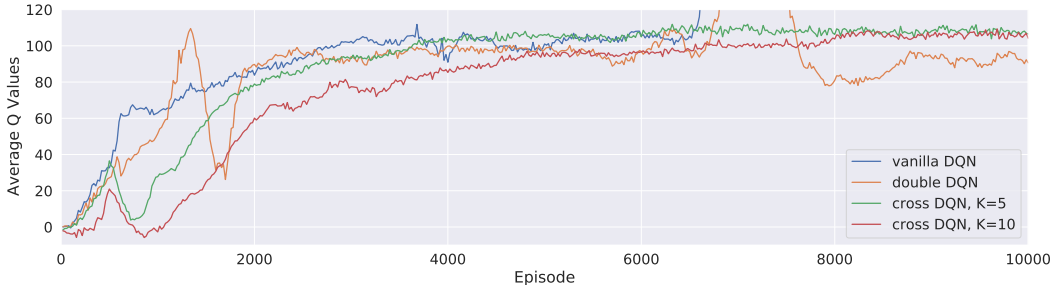
In Figure 5.5, We compare our cross Q-learning algorithms with vanilla DQN and double DQN. With slower learning in the first a few hundreds of episodes due to our experimental design of the learning frequencies, cross DQNs learned much better and more stable policies, while vanilla and double DQN have large variances in both learning curves and performance testing. Figure 5.5(c) clearly shows that from the beginning, vanilla DQN optimistically gathers the occasional large rewards which are due to the high variance, and produces great overestimations. Double DQN slightly allivates the problem, but cannot avoid the overestimation effectively. The derived policies from these two networks are then not optimal nor stable. As learning going on, the estimated Q-values from both vanilla and double DQN explode, resulting in that the derived policies are no better than random actions. On the other hand, cross DQNs have much lower Q-value estimations at the beginning, and the estimates from model with $K = 10$ are even lower than that from model with $K = 5$.



(a) Learning curves



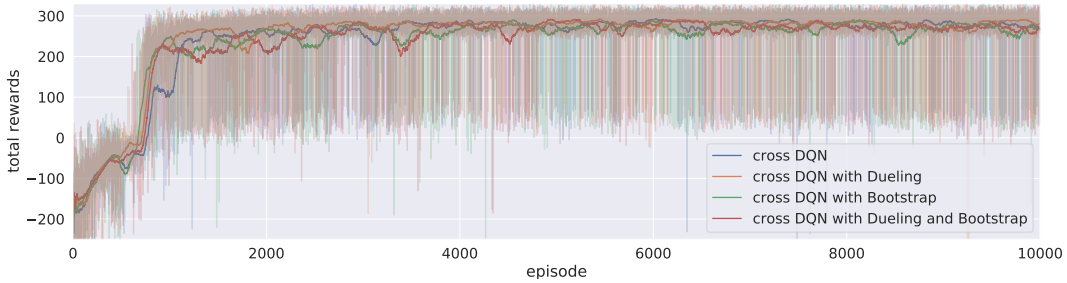
(b) Model test performance. Every 20 training episodes, 10 full test episodes were conducted.



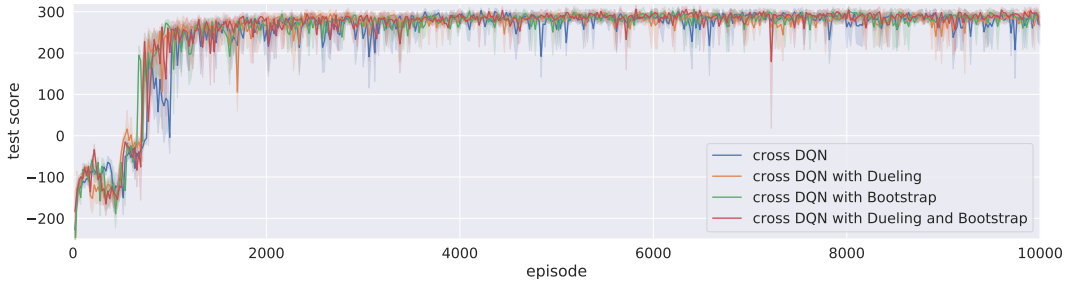
(c) Mean of Q-value estimations on LunarLander. Every 20 training episodes, 1024 (s, a)-pairs were bootstrapped.

Figure 5.5: Comparison of vanillar DQN, double DQN and cross DQNs of $K = 5$, $K = 10$ on *LunarLander*.

After 1000 episodes, the estimates continue growing until convergence, and their values converge to a same level at about 105. The derived policies are very stable, with total rewards close to 300 and also have little variance. Note that double DQN has lower estimates of Q-values than cross DQNs after 8000 episodes of training. The reason is that the corresponding policies from double DQN are much worse, and it does not indicate that double DQN addresses overestimation better.



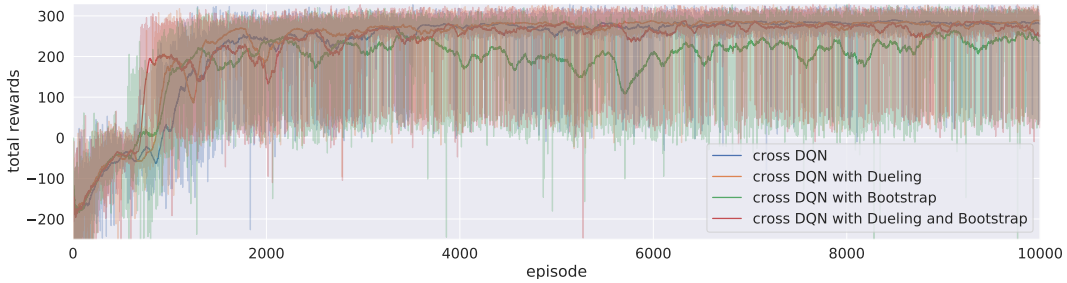
(a) Learning curves



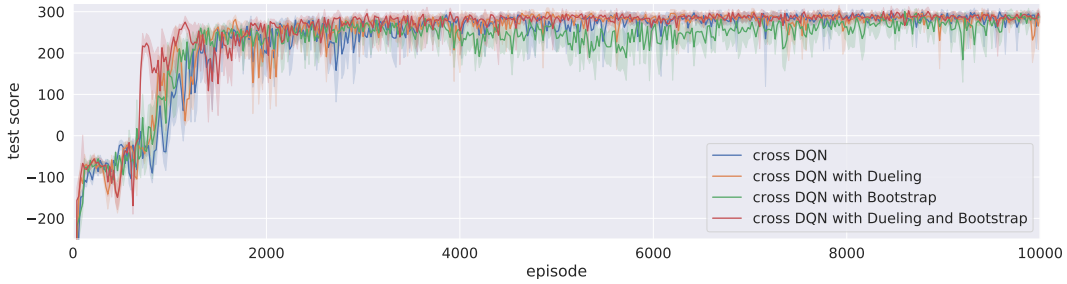
(b) Model test performance. Every 20 training episodes, 10 full test episodes were conducted.

Figure 5.6: Comparison of cross DQNs of $K = 5$. Cross DQN, with dueling DQN, with bootstrapped DQN, and with both dueling & bootstrapped DQN on *LunarLander*.

Comparing Figure 5.6 and Figure 5.7, $K = 5$ seems works even better than $K = 10$ for most of time. Especially for $K = 10$ bootstrapped cross DQN, both the learning curve and the test scores are lower than other cross DQN models. This indicates that it is not always the larger K the better, since cross estimator would induce underestimate bias, and too much underestimation may also hide the real better actions and thus hurt the model performance. In fact, $K = 10$ cross DQN might have too much underestimation at the beginning, which slows down the learning process significantly. But overall, the $K = 10$ bootstrapped cross learning with dueling architecture performs best among all models, including all $K = 5$ cross DQNs. We say that the DQN architectures are too complicated, and the aggregated effect may significantly change the performance of a particular model. Generally speaking, our cross DQNs favor underestimation, which should be much better than overestimation if no unbiased estimation can be achieved, since underestimations do not tend to propagate too much during training, as lower valued actions are avoided by the greedy action selection



(a) Learning curves



(b) Model test performance. Every 20 training episodes, 10 full test episodes were conducted.

Figure 5.7: Comparison of cross DQNs of $K = 10$. Cross DQN, with dueling DQN, with bootstrapped DQN, and with both dueling & bootstrapped DQN on *LunarLander*.

mechanism. And the bias-variance tradeoff tells us that the overall error can be reduced when the variance of our estimates is greatly decreased, by introducing slight negative bias, this in tern leads to better model performance.

Note that the derived policies from cross DQNs are much more stable in general, and hard to deteriorate. There are at least two reasons for this phenomena. First, cross Q-learning effectively addressed overestimation problem, thus premature policy would be more difficult to derived from cross DQN. In addition, we always ensemble policies using methods such as majority voting during test time, which in general is superior and has a stabilizing effect for action selections. The improved stability comes from larger barrier for altering the decision boundaries, and we could care much less about the early termination as an additional hyperparameter during training. This is yet another advantage of using multiple networks as in cross DQN.

5.6 Conclusions and Future Work

In this paper, we have presented the cross Q-learning algorithm, an extension to DQN that effectively reduces overestimation, stabilizes training, and improves performance. Cross DQN is a simple extension that can be easily integrated with other algorithmic improvement such as dueling network and bootstrapped DQN, leads to dramatic performance enhancement. We have both shown in theory and demonstrated in several experiments of classical control problems that the proposed scheme is superior in reducing overestimation and leads to better policies derivation, compared to widely used approaches such as double DQN. Cross learning favors underestimation, the introduced negative bias can greatly help variance reduction. We analyze this effect from the famous bias-variance tradeoff point of view. However, this also indicates that it is not the case the larger K the better model performance in cross DQN. Nevertheless, DQN models tolerate underestimation much more than overestimation, as lower valued actions can be avoided by the greedy action selection mechanism.

It is noted that the computation complexity of cross DQN is generally higher, comparing with that of single network DQNs. We can, however, greatly reduce the complexity given the flexibility provided by our model. In addition, ensemble policies from multiple networks help stabilize the decision space, which can be utilized optionally in stablizing learning and definitely during testing.

As future work, we would apply cross learning to the state-of-the-art actor-critic methods in continuous control, further reduce the overestimation and stabilize those algorithms. Also, analysis from statistical learning theory could be helpful for us to derive more advanced cross learning strategies, for instance, better bootstrap estimations may be obtained by mimicking the K-fold cross validation [164], or from Bayesian perspective [169].

Moreover, it worth noting that in each step of Q-learning (and more general value-based RL), we utilize Q-values in several different places. Now that a set of K different Q-functions are applied, we can make different choices for picking particular one to use. We call them generalized cross learning in DQNs, and some existing work can be fell into a particular

subclass of our generalized method. The first place that Q-values are utilized is when the agent makes decision for choosing an action a_t at time step t while observing s_t. We can pick a random Q-function for action selection, and this is exactly what bootstrapped DQN [133] does. We say the bootstrapped DQN is a special case of our generalized cross DQN. The next place is at TD update when the target Q-values need to be evaluated for choosing the next action a', which might not be executed, but is used to evaluate the current target Q-value and derive the max operator. Recall in Q-learning we use the maximum estimator. Finally, after picking the next action a', its value can be evaluated, again we have choices here for picking a Q-function to use. In the version of our cross DQN we presented in this work, which is directly derived from double DQN, we decoupled the selection and evaluation of the next action a', where the current network is used for evaluating a' while another target network is used for selecting a'. We could try to do the opposite in certain circumstances, i.e., select a' with the current network and bootstrap another network to evaluate a', which should have the effect of decrease bias but increase variance due to bias-variance tradeoff in general statistical learning scheme. One can further analyze and experiment with other generalized cross Q-learning variants.

Chapter 6

An Application of Deep Q-Network for Financial Trading

In this chapter, we formulate and train a simple reinforcement learning (RL) trading agent with deep Q-network (DQN). By observing the states from the real trading environment, the agent is able to directly make sequential decisions through the allocation of a portfolio, aiming at maximizing the cumulative profit according to the derived trading strategies such as to `LONG`, `SHORT` or `CLOSE` for certain assets. The action space of the DQN agent is discrete and finite, which greatly simplifies the portfolio optimization problem in real world. The computational cost would grow exponentially as the number of allowed actions increases, due to the curse of dimensionality in MDP. Therefore, recent advances focus on policy gradient algorithms that form the actions in parametric continuous space, in which only a few parameters need to be learned. In particular, consider the case that there is only a single asset other than cash in the portfolio, and the amount of buying or selling is fixed, the optimal strategy would be as simple as to buy when the asset's price increases, and to sell while the price decreases. As training is based upon all the historical information in the market, the best a trading agent can perform is to obtain the most accurate predictions for the future prices, whereas the RL trading agent needs to learn from errors through temporal difference update, given the credit is reasonably assigned through the reward functions during the design of the RL system. In other words, the same cross-entropy or mean squared error loss (for the classification or regression problem, respectively) for supervised learning is simply decomposed in the temporal sense and learned by the RL agent in the Bellman form. Therefore, the learning of the RL agent is less effective, and supervised learning system for more accurate predictions in prices is more widely applied, and we have developed such a framework as introduced in chapter 2. Certain optimal trading strategies can then be derived

by forming an optimization problem, based on the predicted prices for the future. However, RL trading agent has its own value as the system is end-to-end, rather than the two stages framework (prediction then optimization) we introduced above.

6.1 Introduction and Related Work

Deep reinforcement learning has achieved remarkable success in a wide range of research areas such as playing game Go [170], World of Warcraft or Starcraft [171], etc. Recently, it has also been applied in financial analysis and investment by a multitude of researchers. For instance, [172] proposed a recurrent reinforcement learning (RRL) algorithm to optimize security portfolios. [173] used the relative risk-adjusted profit (Sharp ratio) as performance function to train the trading system based on Q-learning. [174] compared the performance of DQN and RRL, and reported that DQN achieved better performance in stock trading. [175] applied deep reinforcement learning on portfolio management with cryptocurrencies. [176] combined DQN with a regressor that predicts the number of shares to trade. In this chapter, we are trying to employ deep Q-learning to build an deep Q-trading system which can automatically determine what position to hold at each trading time. As we will illustrate later, our approach is not the same as those state-of-art work.

This chapter is organized as follows. In Section 6.2, we carefully formulate and establish the trading problem as solving a DQN. We perform empirical studies on the Bitcoin trading market using our implemented DQN trading agent, as in Section 6.3. Finally, we conclude in Section 6.4.

6.2 Problem Formulation for Trading

6.2.1 State Space

In the problem of algorithmic trading, the agent performs trading actions in the environment of a financial market. It is impossible for the agent to get full information of such a

complex environment, which involves all activities in the human society as well as subjective emotions. Nevertheless, in the context of technical analysis, it is believed that all relevant information is reflected in the prices of financial assets. Under this point of view, a state s_t can be roughly represented by the prices throughout the market's history up its current moment t. Full history information, however, is either not available or too large for computation in practice. People instead discretize the time into periods, and use only a number of recent periods to represent the current state. Recent research of reinforcement learning for trading focus on using complicated convolutional or recurrent neural networks as the first few layers, in the hope that the relevant features could be automatically extracted by the state-of-art deep learning frameworks. In our practical experience, however, the representation power of these deep neural networks for algorithmic trading, especially in the context of reinforcement learning, may require extra tuning. On the other hand, various technical indicators serve as the foundation for existing technical analysis in the finance industry. These technical indicators were summarized by financial experts throughout many years, and have shown to be effective as trading signals. In our experience, we also found that these indicators can represent much more ambient information of historical trend and changes, and several indicators combined together could be served as more effective features in the RL trading framework than those automatically learned by deep neural networks. There are hundreds of technical indicators exist and people utilize them in different circumstances. For simplicity purpose, we only apply a few of them. More detailed discussion of applied technical indicators in this project would be provided in Section 6.3.2.

6.2.2 Action Space

The dynamics of trading depends on the actions taken by the agents. Depending on how complex we wish our RL agent to be, the action space could be either discrete or continuous. For each symbol in the portfolio, at a time step, one can either `BUY`, `HOLD`, or `SELL`. Both `BUY` and `SELL` would result in changing on the *positions* of the symbol, in which case we say

an *order* or *trade* occurs. The *positions* of the symbol can be either `LONG` (which means to possess positive number shares of the symbol), `CLOSE` (0 shares), or `SHORT` (meaning negative shares possession). We further assume the amount of shares each order can take to be some fixed values for simplicity purpose, so that the action space is discrete and fits the DQN framework.

The trading signals often represent appropriate position to take at the moment in the market, thus we will derive the actions from changes of the recommended positions by the RL agent, in the hope that our system can be more stable. Remark this strategy is different from some state-of-art work of deep RL trading. Also note that there are many other choices of actions. The agent could determine the proportion of each symbol (including cash) in a portfolio, which would involving continuous action space. Even more complex scenarios might involving various constraints imposed on the orders; or multiple objectives, such as placing limit orders in reality, in which case the agent needs to determine not only the trading amount also the price level.

6.2.3 Reward Function

Reward function directly reflects the goal of RL and determines the derived policy of the agent, and the design for reward function is tricky. The *Profit and Loss* (PnL) is a realistic choice, which is defined as the net profit from a trade when `CLOSE` a position, i.e., `SELL` previously `LONG`, or `BUY` previously `SHORT`. However, this type of reward signals is sparse since trading could be relatively rare during a period of time, and assigning the sparse reward in the temporal sense is a critical challenge in RL, i.e., the credit assignment problem. Instead, a more common choice is the net profit or return between two consecutive trading time slots, i.e., $p_t - p_{t-1}$ or $p_t/p_{t-1} - 1$ while the position is holding `LONG`, and conversely, taking the negatives if holding `SHORT`. This gives the agent more frequent feedback signals. In this project, we will use this return-typed reward.

There are various other choices for RL trading systems in literature. One may be risk-averse in some circumstances and take into account the risk as the objective, depending on how the profit and risk would result in his/her overall *utility*. *Sharpe ratio* and its variations, while consistent with Markowitz's mean-variance portfolio optimization theory, are commonly used. We will not use *Sharpe ratio* as the reward, but calculate its value as a measure of performance for the derived policies.

Transaction Cost

In reality, to `BUY` or `SELL` an order will incur some transaction cost. Usually a constant commission fee for the brokers will be put on each order, also the activity would cause the buying or selling pressure on the market and result in some impact on the prices, especially when the amount of order is huge. To simplify the model, we consider the transaction cost to be proportional to the trading symbol's prices, i.e., the price would be $(1 + c)p_t$ if you `BUY`, and become $(1 - c)p_t$ while you `SELL`, in which c is the transaction cost factor, and p_t is the close price at the moment. In this way, we also combine the transaction cost into the reward function, since an order would cause the return changes. We will evaluate the effect of transaction cost on the RL agent's learned policy in Section 6.3.4.

Learning Algorithm

In this paper, we use the DDQN update for training the trading agent. The learning procedure is illustrated in Algorithm 5.

Algorithm 5 Training DDQN Trading Agent with Experience Replay

Require: Initialize replay memory $\mathcal{B}$ to capacity N
Require: Initialize action-value function Q with random weights
for episode = 1, M **do**
Initialise sequence $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
for $t = 1, \cdots, T$ **do**
With probability ϵ select a random action at a_t
otherwise select $a_t = \max_a Q_\theta(\phi(s_t), a)$
Execute action at in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi(t), a_t, r_t, \phi(t+1)$ in $\mathcal{B}$
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{B}$

$$y_i = \begin{cases} r_j & \text{for terminal}\phi_{j+1} \\ r_j + \gamma Q_{\theta^-}(\phi_{j+1}, \text{argmax}_a Q_\theta(\phi_{j+1}, a)) & \text{for non-terminal}\phi_{j+1} \end{cases}$$

Perform a gradient descent step on loss function to update θ
Update $\theta^- \leftarrow \theta$ every τ time steps.
end for
end for

6.3 Experiment

6.3.1 Environment Setup

Cryptocurrencies are electronic and decentralized alternatives to money issued by the governments. While the Bitcoin as the best known example and the dominant representative, there are various other cryptocurrencies traded in the electronic market. One advantage of using the cryptocurrencies is their openness, all the trading transactions are accessible on the orderbook, which is transparent to anyone through the Internet. Also most exchanges are open 24/7 without restriction on the amount. These are different from the traditional stock market, where the transaction records for high frequency trading (HTF) are not free (although some institutions such as Yahoo! Finance provide daily data) and some are hidden due to regulation requirements, also exchanges close during particular periods. We download the historical data from `https://www.cryptodatadownload.com/`, which has already converted all the transaction records into minute, hourly, or daily format by providing the

columns of open, close, high, and low prices similar to the stock market data. For our project, we will trade only a single cryptocurrency, the Bitcoin, which means the portfolios will consist of only cash and one symbol, `BTC`. Moreover, we choose data for the recent year for training and testing, one reason again is to reduce the computation time, also because that the prices experience significant rise and drop during the past year, and comparing with the past year, the previous price trends are simple and flat. Specifically, we choose one-year daily prices data between Oct 08, 2017 and Oct 08, 2018 for training (i.e., in-sample period), and the out-of-sample period is from Oct 09, 2018 to Dec 09, 2018 upon which we test/validate the derived policy from the learned DQN agent.

Now assume you are a millionaire starting with $1M cash that could be invested in the cryptocurrency market, which allows you to buy 100 coins for most of the time except the only month, namely Dec 2017, when the Bitcoin was at its peak price around $20,000. For simplicity purpose, we assume that there are only 3 allowable positions throughout the period of time: 100 coins `LONG`, 100 coins `SHORT`, and without holdings. As a result, different trading activities are allowed, depending on current position: you can either `CLOSE`, `SHORT` or hold still when you are holding `LONG`, which involves in selling 100, 200, and 0 coins, respectively. Similarly, you can either buy 100, 200 or 0 coins when you are holding `SHORT`. And while holding cash only, you have a choice among buying 100 coins, short selling 100 coins, and keeping the cash.

A benchmark portfolio will be used for comparison purpose, which is generated by the buy-and-hold (BH) strategy, i.e., the portfolio starts with the same $1M cash, investing in 100 coins and holding until the end of the period. This simple benchmark portfolio will partially reflect the price change of the coin.

In order to evaluate the derived policies, we need to simulate the performance of corresponding portfolios in the market. The portfolio values need to be calculated for each time step, considering the current price of Bitcoin as well as current position. Upon the history of the portfolio values, various metrics can be used to evaluate its performance, such as the

cumulative return, daily return (mean and standard deviation), and Sharpe ratio, etc. We focus on the cumulative return, since the objective of our agent is to maximize the profit, according to our designed reward. But we record some other measures as well.

6.3.2 DQN Agent Setup

Rather than using a sliding window of raw price data as the features, we preprocess the features with technical indicators, since the former one gave us poor performance. Specifically, we use the simple moving average index (SMA), Bollinger bands %B, relative strength index (RSI), as well as Williams %R, together with the daily price data as the input features. According to financial experts, a combination of technical indicators from different categories would give us nice trading signals. In our case, SMA serves as a trend indicator; Bollinger bands are simply the upper and lower bands which are 2σ away from SMA where σ is the standard deviation, thus %B is a volatility indicator; RSI on the other hand, is a momentum indicator; while Williams %R is an oscillator indicator. They together represent if the current price is in "overbought" or "oversold" condition, thus give signals for trading. Interested readers are referred to [72] for further reading of the technical indicators.

The features need to be normalized, since they are in different scales. The normalized features then serve as the states and are continuous, thus we used a neural network to approximate the action value function Q. A small fully-connected neural network which has one hidden layer of 10 neurons with ReLU actionvation was used, and ReLU nonlinearity is used as the activate function for each hidden neuron. The 5-dimensional vector s which describes the state would be the input for the network, and the final layer is simply linear combinations since we are trying to approximate Q, which can be any real value in essence. It would have multiple outputs, one for each of the 3 possible actions.

We trained a neural network with experience replay. We have not put restrictions on the size of the replay buffer since the samples we have gathered are of moderate size and can be fit into the memory. A batch of 64 samples are randomly sampled from the replay memory

each time for fitting the network. The adaptive moment estimation (Adam) optimizer is used to train the network, since it is in general less sensitive to the choice of the learning rate than other stochastic gradient descent algorithms [119], where the initial learning rate is set to 0.0025 after some tuning. We use the pseudo-Huber loss instead of MSE as the loss function, since it is less sensitive to outliers and is more commonly used in DQN [129]. ε-greedy policy is used for action selection, and the ε value decays with rate 0.999, which could be approximately decay from 1 to its minimum value 0.01 in about a full episode. The discount factor γ is set to 0.95. We do not consider the modeling horizon explicitly, but the discount factor implicitly limits the effect that the experienced reward can impose in the temporal sense.

For the training phase, our reward function actually assumes the price of next trading day is known. If the stock price goes up today, i.e., $p_{t+1} > p_t$, then the learning agent should better to assign larger Q-value on the action of *going long*, and small or negative Q-value on *going short* for yesterday; on the contrast, if the price decreases today, then the best choice for yesterday should be to *go short*, and should not choose to *go long*. We assume a single cryptocurrency in the portfolio, namely the Bitcoin. And 3 possible positions can then be taken: long 100 coins, short 100 coins, or hold cash only. An order would be placed by taking the holding difference of every adjacent trading days. The transaction cost is assumed to be zero in Section 6.3.3, and would be compared with different settings in Section 6.3.4.

6.3.3 Results

We show the training progress in Figure 6.1. Note that the total reward values are different from the cumulative returns we use to evaluate the portfolio performance, although they are closely related. We trained the DQN from the begining using the same in-sample data for 10 runs, in each run we trained 20 episodes, and an episode was trained using the whole in-sample data from 2017/10/08-2018/10/08, following the time sequence. We record the sum of rewards for each episode. Figure 6.1 illustrates the trend of reward sums over

Figure 6.1: Total rewards for each episode throughout training

training, in which the dark line represents the means of episodic total rewards for 10 experiments, and the transparent region shows the variances (i.e., mean$\pm 2\sigma$ confidence range), from which we see the DQN agent is consistently making progress throughout training. In each run, the first episode always results in negative total rewards (and also has negative cumulative return), which is due to the random action selection at the beginning of training. Recall that we use ε-greedy action selection strategy in DQN, in the hope that the agent could have enough exploration. As training proceeds, the agent tends to stick to the best action it can take, based on its estimation of the Q-values. The TD-errors give it the signal to update the weights of DQN, and the Q-value estimations could be improved, thus better policy would be derived. We emphasize the deployment of experience replay, by which the sample efficiency has been significantly improved, especially for training in such a trading environment, where the samples are actually quite limited. As a result, it also dramatically speedup the training. We are not claiming our DQN training has converged, on the contrary, we expect it could keep improving after 20 episodes, and could give us more refined results if we use smaller learning rate. However, due to lacking of computational resource and time expenses, we only trained for 20 episodes.

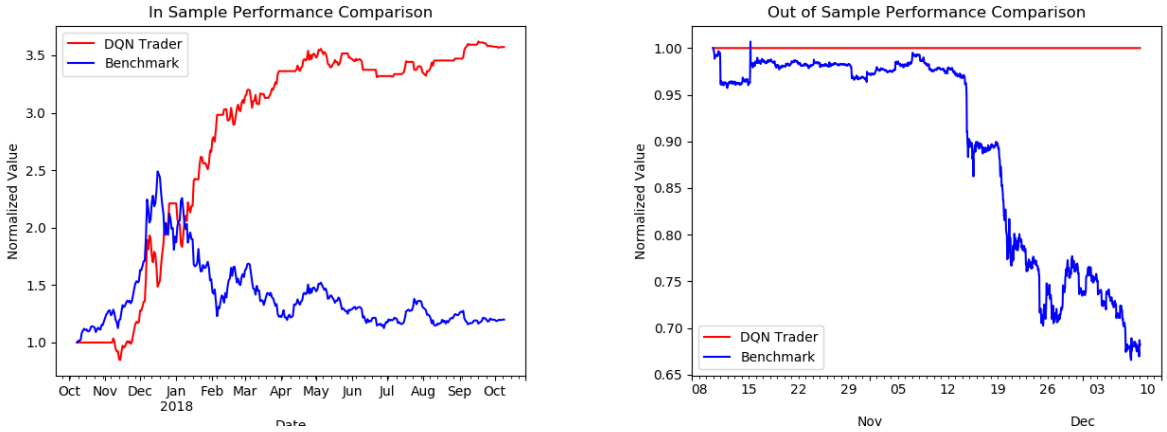
In Figure 6.2(a), we show the performance of the portfolio derived from our DQN policy (trained after 20 episodes), compared with the above-mentioned benchmark portfolio, during the in-sample period between 2017/10/08 abd 2018/10/08. The red line represents the normalized value of the portfolio using DQN policy during the period, while the blue

	DQN	Benchmark
Cumulative Return	3.5053	0.1991
Mean Daily Return	0.0118	0.0010
Std. Daily Return	0.0664	0.0313
Sharpe Ratio	2.8236	0.4979

Table 6.1: Some statistics of in-sample performance, DQN derived portfolio v.s. benchmark, 2017/10/08-2018/10/08

line represents that of the benchmark. We say the value of benchmark portfolio partially illustrates the change of the Bitcoin prices. And we see that the learned DQN policy performs much better than the benchmark for in-sample data. According to Table 6.1, the cumulative return is 350.53%, which means that the value of the portfolio become 4.5 times as its beginning in one year period. This is an extremely high return. We also record some other detailed in-sample performance statistics for both portfolios, as shown in Table 6.1, from which we see that our DQN derived portfolio obtains much higher cumulative return and Sharpe ratio. In fact, as we will show in Section 6.3.4, the DQN policy executes orders very frequently throughout the period, in the sense that taking advantage of the oscillations of the Bitcoin prices. As a result, its value almost always increases during the in-sample period. On the other hand, notice there are quite a lot of horizontal intervals on the red line, which represents our DQN agent chooses to `CLOSE` the position and holds cash only, and the cash value would not change during that period of time. In this way it could avoid loss during the rapid fall of Bitcoin price. This shows that its picked action is from the signals provided by those technical indicators, and probably could generalize.

Figure 6.2(b) illustrates performance of the learned DQN policy on out-of-sample data, specifically the recent 2 months (2018/10/09-2018/12/09). During this period, the Bitcoin price almost halves and the drop starts since mid November, which we can see from the benchmark performance. The strategy derived by our DQN agent is simple: hold the cash, and do not trade on Bitcoin. This would definitely result in a cumulative return of zero, but it is much better than losing money by `LONG` the Bitcoin. Unlike the different derived



(a) In-sample performance (from 2017/10/08 to 2018/10/08)

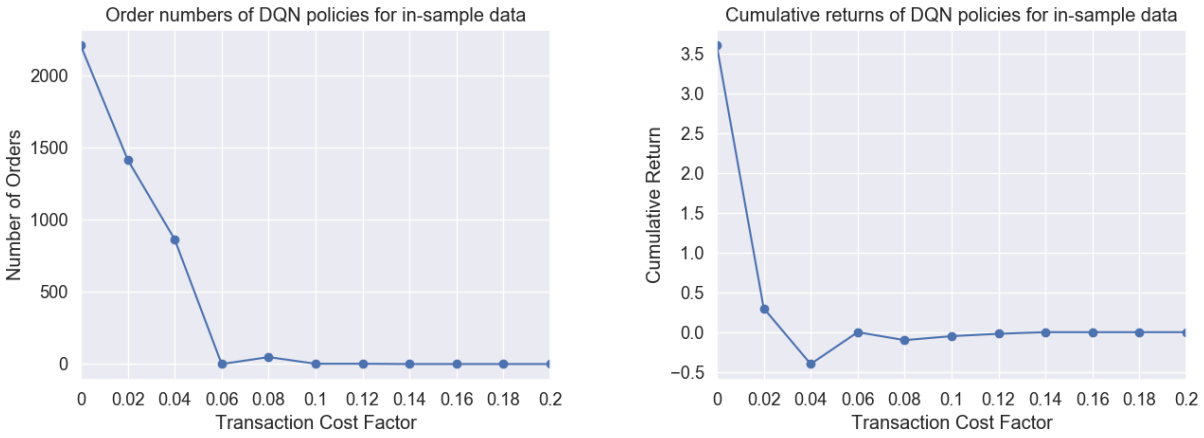
(b) Out-of-sample performance (from 2018/10/09 to 2018/12/09)

Figure 6.2: Effects of different transaction cost factor values on DQN in-sample policies

strategy and thus performance on in-sample data, this derived policy for the out-of-sample period is actually very consistent accordiing to our DQN agent, and can be learned in 2 episodes. However, we need to note that our DQN agent has not learned to take the advantage of shorting during the out-of-sample period, which we allow in the environment settings. According to the in-sample performance, our agent has successfully learned to take such an advantage during the dramatic falls in January and February, 2018, but for smaller falls, it chooses to hold cash rather than to short, since a small rise might occur after a short period of time. The drop since mid November is not that significant comparing with those in in-sample period, and our agent seems to be afraid of later risks thus choose to hold cash. From this we can see how the domain matters. The training and test domain should be of the same distribution (in expectation), which is not the case for our Bitcoin prices, and actually is not the case for trading environment in general.

6.3.4 Effect of Transaction Cost

As we discussed in Section 6.2.3, our reward function implicitly takes the transaction cost into account, and also our implemented market simulator would illustrate the effect of transaction cost. How would the transaction cost affect our DQN agent's learned trading



(a) Number of orders in the derived policies (b) Cumulative return for the derived policies

Figure 6.3: Effects of different transaction cost factor values on DQN in-sample policies

behavior and performance? In this section, we conduct an experiment on our DQN trading agent for the in-sample bitcoin data, with 6 different values of transaction cost factors c, ranging from 0 to 0.20. As c increases, the updated Q-values might be more inaccurate, thus the learned policy may perform worse. However, if the transaction cost is too much, that execution of a trading order has larger effect than taking advantage of the price changes, there is no necessity to trade. Our DQN agent has successfully learned that.

We first look at the number of orders from learned strategies with different transaction cost factors, as shown in Figure 6.3(a). We see that when the transaction cost is 0, the DQN agent executes quite a lot of orders, by taking the advantage of price changes. As transaction cost gets higher, the number of orders from derived policy decreases significantly. As the transaction cost factor ≥ 0.10, the agent learns to not to trade at all, since the cost exceeds the return, and executing orders can only result in a loss from now on.

Figure 6.3(b) shows the cumulative returns of learned polices with different transaction cost factors, on the same in-sample data, i.e, between the period Oct 08, 2017 and Oct 08, 2018. We see that when the transaction cost is 0, the agent learns to derive a strategy with quite high in-sample cumulative return. As the transaction cost increases, the learned policies perform worse. While the transaction cost factor reaches 0.04, the derived policy have

negative cumulative return. After that, as the factor grows, the learned policies performs a little bit better, for the agent learns to execute very few orders. When the factor ≥ 0.1, as we see in Figure 6.3, our DQN trading agent chooses to hold cash only from then on, results in a cumulative return of zero.

6.4 Summary

In this work we trained a DQN trading agent that applies the deep Q-learning approach to algorithmic trading. DQN trading is able to detect market status from raw and noisy data, and pays attention to long-term returns. Comparing with the state-of-art works, we emphasized deploying technical indicators for feature preprocessing while feeding the data for training rather than using the end-to-end deep learning framework. Our experiments on Bitcoin portfolio demonstrated that our DQN trading system performs well consistently. We also showed our DQN trading agent correctly responds to some environmental factors such as the transaction cost. Despite these interesting results, our study is still in a preliminary stage. In future work, we will investigate the contributions of other novel approaches from reinforcement learning research community, especially those with policy gradient which could be applied on continuous action spaces.

Chapter 7

Conclusion

In this dissertation, we have explored several subfields in deep learning and reinforcement learning. We first focus on predictive modeling. In Chapter 2, we built a hybrid deep learning model that can efficiently and effectively predict the future returns in stock market. Our work combines the recent advances in representation learning and temporal convolutional neural networks for sequential modeling. In addition to improving learning Alphas for stock prediction as usual research, we also combined the market information for learning the Betas. The hybrid training paradigm is motivated by stacking, but instead of combining the final predicted results from different models, we combined the learning feature maps to learn in the last a few layers to improve the forecast performance. There are several directions can go from where we achieved. First, the stock prices are heavily affected by external information, such as the macroeconomics factors, politic events, online exposure and sentiment on Internet such as in social media, etc. We could expand the data source and combine their influence in the model. Moreover, systematic feature selection could be performed for more effective learning, in addition to the ordinary L1 and L2 regularization, advanced methods such as grouped Lasso may have a chance to combine with deep learning models as well. In addition, although we have shown that the convolutional layers have several advantages over the most widely used recurrent neural network layers for time series, the temporal learning layers in our model could be replaced by any other type, for instance, the recent advances of attention models could be a good candidate. More importantly, systematic analysis on learning the Betas from the market could give us insight of more effective usage of the cross-sectional data, thus further improve the model performance. We believe this could be greatly related

to the field of unsupervised learning, in particular the decomposition of high dimensional data.

We then turn our focus to reinforcement learning. In Chapter 4, we encourage the exploration in deep Q-networks by reanneal the exploration rate when it stucks at poor local optima, measured by heuristic measures. As a few possible directions of future work, first of all, the idea of reannealing exploration can be easily and naturally extended to other reinforcement learning algorithms, not limited to DQN. Second, the measure of poor local optima could come from monitoring the performance of the learning system, in this way, we probably could use supervised learning to define and learn such a measure, instead of heuristics. It is also attractive to systematically reanneal the exploration rate, instead of adaptively responding to the heuristic measure. In the field of deep learning optimization, we have seen various forms of cyclical learning rate. Similarly, we could experiment on cyclical exploration rate as well.

In Chapter 5, we effectively alleviate the overestimation problem in value-based reinforcement learning, by cross learning the Q-values with multiple agents, resulting in faster convergence of training deep Q-networks. Our cross learning paradigm can be extended to deterministic policy gradient algorithms, and then naturally generalize to actor-critic algorithms, in which multiple critics are usually used for estimation of the action values. The difference between synchronized and asynchronized versions of cross learning in actor-critics is worth for further study.

We demonstrate the effective application of reinforcement learning algorithms in real trading environment in Chapter 6. We discussed the difference between the reinforcement learning trading and the general two steps portfolio optimization, and mentioned that the reinforcement trading system decomposes the supervised learning loss in the temporal sense through the reward function, and learns in the Bellman update form, thus is less effective. However, the system is end-to-end thus is more convenient to use. In future work, we could train reinforcement learning agents in more complex trading environment, for instance, to

establish the strategy for the allocation of a portfolio that consists of many stocks. It would be computationally expensive or even infeasible to training such a DQN agent that considering discrete actions, due to the curse of dimensionality. In that case, we could consider the actions in parametric continuous space, and the new advances of policy gradient algorithms might be more suitable.

This dissertation has presented novel models and algorithms in predictive modeling and reinforcement learning. One lesson that can be gleaned from the studies is that, in the design of a machine learning system, sometimes it is powerful to combine different components and significant advances to achieve better performance. Ensemble as a meta-method improves the performance in general, in the cost of computational complexity, but the idea of ensemble can be synthesized to a single model also, as we combining multiple components. And we can benefit from imposing other meta-heuristics as well. Despite the many successes that machine learning has seen, many interesting and important problems in learning, or more generally in artificial intelligence, remain. We believe that further efforts to study in this field will pay off richly.

Bibliography

[1] Thomas M Mitchell et al. Machine learning, 1997.

[2] Vladimir N Vapnik and Alexey J Chervonenkis. Theory of pattern recognition. 1974.

[3] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[4] C Cortes and V Vapnik. Support vector machine [j]. *Machine learning*, 20(3):273–297, 1995.

[5] Harris Drucker, Christopher JC Burges, Linda Kaufman, Alex Smola, Vladimir Vapnik, et al. Support vector regression machines. *Advances in neural information processing systems*, 9:155–161, 1997.

[6] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[7] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[10] Michael J Kearns and Leslie G Valiant. *Learning Boolean formulae or finite automata is as hard as factoring*. Harvard University, Center for Research in Computing Technology, Aiken Computation Laboratory, 1988.

[11] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[12] Yoav Freund. Boosting a weak learning algorithm by majority. In *COLT*, volume 90, pages 202–216, 1990.

[13] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[14] Robert E Schapire. The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer, 2003.

[15] Thomas G Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 40(2):139–157, 2000.

[16] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[17] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[18] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[19] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[20] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[21] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[22] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[26] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[27] Hasim Sak, Andrew W Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. 2014.

[28] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[29] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 2019.

[30] Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. In *Advances in neural information processing systems*, pages 7785–7794, 2018.

[31] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[32] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[34] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[35] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[36] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[37] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[38] Stephen Boyd, Neal Parikh, and Eric Chu. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[39] Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):127–239, 2014.

[40] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[41] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

[42] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[43] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[44] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[45] K Peason. On lines and planes of closest fit to systems of point in space. *Philosophical Magazine*, 2(11):559–572, 1901.

[46] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[47] Imola K Fodor. A survey of dimension reduction techniques. *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*, 9:1–18, 2002.

[48] John Wright, Arvind Ganesh, Shankar Rao, Yigang Peng, and Yi Ma. Robust principal component analysis: Exact recovery of corrupted low-rank matrices via convex optimization. In *Advances in neural information processing systems*, pages 2080–2088, 2009.

[49] Emmanuel J Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3):1–37, 2011.

[50] Jian-Feng Cai, Emmanuel J Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on optimization*, 20(4):1956–1982, 2010.

[51] Eugene F Fama. The behavior of stock-market prices. *The journal of Business*, 38(1):34–105, 1965.

[52] William F Sharpe. Capital asset prices: A theory of market equilibrium under conditions of risk. *The journal of finance*, 19(3):425–442, 1964.

[53] John Lintner. The valuation of risk assets and the selection of risky investments in stock portfolios and capital budgets. In *Stochastic optimization models in finance*, pages 131–155. Elsevier, 1975.

[54] Michael C Jensen, Fischer Black, and Myron S Scholes. The capital asset pricing model: Some empirical tests. 1972.

[55] George EP Box and Gwilym M Jenkins. Some recent advances in forecasting and control. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 17(2):91–109, 1968.

[56] Rob Hyndman, Anne B Koehler, J Keith Ord, and Ralph D Snyder. *Forecasting with exponential smoothing: the state space approach*. Springer Science & Business Media, 2008.

[57] Khalid Alkhatib, Hassan Najadat, Ismail Hmeidi, and Mohammed K Ali Shatnawi. Stock price prediction using k-nearest neighbor (knn) algorithm. *International Journal of Business, Humanities and Technology*, 3(3):32–44, 2013.

[58] Yingjun Chen and Yongtao Hao. A feature weighted support vector machine and k-nearest neighbor algorithm for stock market indices prediction. *Expert Systems with Applications*, 80:340–355, 2017.

[59] Md Rafiul Hassan, Baikunth Nath, and Michael Kirley. A fusion model of hmm, ann and ga for stock market forecasting. *Expert systems with Applications*, 33(1):171–180, 2007.

[60] Md Rafiul Hassan, Kotagiri Ramamohanarao, Joarder Kamruzzaman, Mustafizur Rahman, and M Maruf Hossain. A hmm-based adaptive fuzzy inference system for stock market forecasting. *Neurocomputing*, 104:10–25, 2013.

[61] Haiqin Yang, Laiwan Chan, and Irwin King. Support vector machine regression for volatile stock market prediction. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 391–396. Springer, 2002.

[62] Cheng-Lung Huang and Cheng-Yi Tsai. A hybrid sofm-svr with a filter-based feature selection for stock market forecasting. *Expert Systems with Applications*, 36(2):1529–1539, 2009.

[63] Jian-Zhou Wang, Ju-Jie Wang, Zhe-George Zhang, and Shu-Po Guo. Forecasting stock indices with back propagation neural network. *Expert Systems with Applications*, 38(11):14346–14355, 2011.

[64] Erkam Guresen, Gulgun Kayakutlu, and Tugrul U Daim. Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, 38(8):10389–10397, 2011.

[65] Werner Kristjanpoller, Anton Fadic, and Marcel C Minutolo. Volatility forecast using hybrid neural network models. *Expert Systems with Applications*, 41(5):2437–2442, 2014.

[66] Lin Wang, Yi Zeng, and Tao Chen. Back propagation neural network with adaptive differential evolution algorithm for time series forecasting. *Expert Systems with Applications*, 42(2):855–863, 2015.

[67] Mustafa Göçken, Mehmet Özçalıcı, Aslı Boru, and Ayşe Tuğba Dosdoğru. Integrating metaheuristics and artificial neural networks for improved stock price prediction. *Expert Systems with Applications*, 44:320–331, 2016.

[68] Jigar Patel, Sahil Shah, Priyank Thakkar, and Ketan Kotecha. Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques. *Expert systems with applications*, 42(1):259–268, 2015.

[69] Ash Booth, Enrico Gerding, and Frank Mcgroarty. Automated trading with performance weighted random forests and seasonality. *Expert Systems with Applications*, 41(8):3651–3661, 2014.

[70] Sasan Barak and Mohammad Modarres. Developing an approach to evaluate stocks by forecasting effective features with data mining methods. *Expert Systems with Applications*, 42(3):1325–1339, 2015.

[71] Bin Weng, Waldyn Martinez, Yao-Te Tsai, Chen Li, Lin Lu, James R Barth, and Fadel M Megahed. Macroeconomic indicators alone can predict the monthly closing price of major us indices: Insights from artificial intelligence, time-series analysis and hybrid models. *Applied Soft Computing*, 71:685–697, 2018.

[72] John J Murphy. *Technical analysis of the financial markets: A comprehensive guide to trading methods and applications*. Penguin, 1999.

[73] Eugene F Fama and Kenneth R French. Common risk factors in the returns on stocks and bonds. *Journal of*, 1993.

[74] Paul C Tetlock, Maytal Saar-Tsechansky, and Sofus Macskassy. More than words: Quantifying language to measure firms' fundamentals. *The Journal of Finance*, 63(3):1437–1467, 2008.

[75] Qing Li, Yuanzhu Chen, Li Ling Jiang, Ping Li, and Hsinchun Chen. A tensor-based information framework for predicting the stock market. *ACM Transactions on Information Systems (TOIS)*, 34(2):1–30, 2016.

[76] Bin Weng, Lin Lu, Xing Wang, Fadel M Megahed, and Waldyn Martinez. Predicting short-term stock prices using ensemble methods and online data sources. *Expert Systems with Applications*, 112:258–273, 2018.

[77] Johan Bollen, Huina Mao, and Xiaojun Zeng. Twitter mood predicts the stock market. *Journal of computational science*, 2(1):1–8, 2011.

[78] Nuno Oliveira, Paulo Cortez, and Nelson Areal. The impact of microblogging data for stock market prediction: Using twitter to predict returns, volatility, trading volume and survey sentiment indices. *Expert Systems with Applications*, 73:125–144, 2017.

[79] Qili Wang, Wei Xu, and Han Zheng. Combining the wisdom of crowds and technical analysis for financial market prediction using deep random subspace ensembles. *Neurocomputing*, 299:51–61, 2018.

[80] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[83] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[84] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[85] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[86] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[87] Akhter Mohiuddin Rather, Arun Agarwal, and VN Sastry. Recurrent neural network and a hybrid model for prediction of stock returns. *Expert Systems with Applications*, 42(6):3234–3241, 2015.

[88] Thomas Fischer and Christopher Krauss. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2):654–669, 2018.

[89] Omer Berat Sezer and Ahmet Murat Ozbayoglu. Algorithmic financial trading with deep convolutional neural networks: Time series to image conversion approach. *Applied Soft Computing*, 70:525–538, 2018.

[90] Guosheng Hu, Yuxin Hu, Kai Yang, Zehao Yu, Flood Sung, Zhihong Zhang, Fei Xie, Jianguo Liu, Neil Robertson, Timpathy Hospedales, et al. Deep stock representation learning: From candlestick charts to investment decisions. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2706–2710. IEEE, 2018.

[91] Omer Berat Sezer and Ahmet Murat Ozbayoglu. Financial trading model with stock bar chart image time series with deep convolutional neural networks. *arXiv preprint arXiv:1903.04610*, 2019.

[92] Ehsan Hoseinzade and Saman Haratizadeh. Cnnpred: Cnn-based stock market prediction using a diverse set of variables. *Expert Systems with Applications*, 129:273–285, 2019.

[93] Wen Long, Zhichen Lu, and Lingxiao Cui. Deep learning-based feature engineering for stock price movement prediction. *Knowledge-Based Systems*, 164:163–173, 2019.

[94] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. A deep reinforcement learning framework for the financial portfolio management problem. *arXiv preprint arXiv:1706.10059*, 2017.

[95] Luca Di Persio and Oleksandr Honchar. Artificial neural networks architectures for stock price prediction: Comparisons and applications. *International journal of circuits, systems and signal processing*, 10(2016):403–413, 2016.

[96] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[97] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[98] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[99] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[100] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[101] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[102] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941, 2017.

[103] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[104] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 156–165, 2017.

[105] Mikolaj Binkowski, Gautier Marti, and Philippe Donnat. Autoregressive convolutional neural networks for asynchronous time series. In *International Conference on Machine Learning*, pages 580–589, 2018.

[106] Yitian Chen, Yanfei Kang, Yixiong Chen, and Zizhuo Wang. Probabilistic forecasting with temporal convolutional neural network. *Neurocomputing*, 2020.

[107] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[108] Yoshua Bengio and Samy Bengio. Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in Neural Information Processing Systems*, pages 400–406, 2000.

[109] Alberto Paccanaro and Geoffrey E. Hinton. Learning distributed representations of concepts using linear relational embedding. *IEEE Transactions on Knowledge and Data Engineering*, 13(2):232–244, 2001.

[110] Geoffrey E Hinton et al. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA, 1986.

[111] Oren Barkan and Noam Koenigstein. Item2vec: neural item embedding for collaborative filtering. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE, 2016.

[112] Edward Choi, Mohammad Taha Bahadori, Elizabeth Searles, Catherine Coffey, Michael Thompson, James Bost, Javier Tejedor-Sojo, and Jimeng Sun. Multi-layer representation learning for medical concepts. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1495–1504, 2016.

[113] Cheng Guo and Felix Berkhahn. Entity embeddings of categorical variables. *arXiv preprint arXiv:1604.06737*, 2016.

[114] Dang Lien Minh, Abolghasem Sadeghi-Niaraki, Huynh Duc Huy, Kyungbok Min, and Hyeonjoon Moon. Deep learning approach for short-term stock trends prediction based on two-stream gated recurrent unit network. *Ieee Access*, 6:55392–55404, 2018.

[115] Qiong Wu, Zheng Zhang, Andrea Pizzoferrato, Mihai Cucuringu, and Zhenming Liu. A deep learning framework for pricing financial instruments. *arXiv preprint arXiv:1909.04497*, 2019.

[116] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[117] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[118] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[119] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[120] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

[121] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[122] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 17–36, 2012.

[123] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I Jordan. Deep transfer learning with joint adaptation networks. In *International conference on machine learning*, pages 2208–2217, 2017.

[124] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[125] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[126] ML Puterman. Markov decision processes. 1994. *Jhon Wiley & Sons, New Jersey*, 1994.

[127] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[128] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[129] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[130] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[131] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[132] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[133] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034, 2016.

[134] Malcolm Strens. A bayesian framework for reinforcement learning. 2000.

[135] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. 1999.

[136] Sebastian B Thrun. Efficient exploration in reinforcement learning. 1992.

[137] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[138] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2753–2762, 2017.

[139] Richard Y Chen, John Schulman, Pieter Abbeel, and Szymon Sidor. Ucb and infogain exploration via q-ensembles. *arXiv preprint arXiv:1706.01502*, 2017.

[140] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[141] Bradly C Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015.

[142] Giuseppe Burtini, Jason Loeppky, and Ramon Lawrence. A survey of online experiment design with the stochastic multi-armed bandit. *arXiv preprint arXiv:1510.00757*, 2015.

[143] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[144] Alexander L Strehl, Lihong Li, and Michael L Littman. Reinforcement learning in finite mdps: Pac analysis. *Journal of Machine Learning Research*, 10(Nov):2413–2444, 2009.

[145] Karl Friston. The free-energy principle: a unified brain theory? *Nature Reviews Neuroscience*, 11(2):127, 2010.

[146] Shin Ishii, Wako Yoshida, and Junichiro Yoshimoto. Control of exploitation–exploration meta-parameter in reinforcement learning. *Neural networks*, 15(4-6):665–687, 2002.

[147] Animashree Anandkumar and Rong Ge. Efficient approaches for escaping higher order saddle points in non-convex optimization. In *Conference on Learning Theory*, pages 81–102, 2016.

[148] Lester Ingber. Very fast simulated re-annealing. *Mathematical and computer modelling*, 12(8):967–973, 1989.

[149] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, pages 663–670, 2000.

[150] Erin Catto. Box2d: A 2d physics engine for games, 2011.

[151] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[152] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. 1993.

[153] Donghun Lee, Boris Defourny, and Warren B Powell. Bias-corrected q-learning to control max-operator bias in q-learning. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 93–99. IEEE, 2013.

[154] Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[155] Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 176–185. JMLR. org, 2017.

[156] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[157] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in neural information processing systems*, pages 1646–1654, 2014.

[158] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[159] Zeyuan Allen-Zhu. Katyusha: The first direct acceleration of stochastic gradient methods. *The Journal of Machine Learning Research*, 18(1):8194–8244, 2017.

[160] Zengqiang Chen, Beibei Qin, Mingwei Sun, and Qinglin Sun. Q-learning-based parameters adaptive algorithm for active disturbance rejection control and its application to ship course control. *Neurocomputing*, 2019.

[161] Xi-liang Chen, Lei Cao, Chen-xi Li, Zhi-xiong Xu, and Jun Lai. Ensemble network architecture for deep reinforcement learning. *Mathematical Problems in Engineering*, 2018.

[162] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[163] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[164] Hado Van Hasselt. Estimating the maximum expected value: an analysis of (nested) cross validation and the maximum sample average. *arXiv preprint arXiv:1302.7175*, 2013.

[165] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308, 2000.

[166] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[167] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[168] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, 5:834–846, 1983.

[169] Carlo D'Eramo, Marcello Restelli, and Alessandro Nuara. Estimating maximum expected value through gaussian approximation. In *International Conference on Machine Learning*, pages 1032–1040, 2016.

[170] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[171] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[172] John Moody, Lizhong Wu, Yuansong Liao, and Matthew Saffell. Performance functions and reinforcement learning for trading systems and portfolios. *Journal of Forecasting*, 17(5-6):441–470, 1998.

[173] Xiu Gao and Laiwan Chan. An algorithm for trading and portfolio management using q-learning and sharpe ratio maximization. In *Proceedings of the international conference on neural information processing*, pages 832–837, 2000.

[174] Yang Wang, Dong Wang, Shiyue Zhang, Yang Feng, Shiyao Li, and Qiang Zhou. Deep q-trading. *cslt. riit. tsinghua. edu. cn*, 2017.

[175] Zhengyao Jiang and Jinjun Liang. Cryptocurrency portfolio management with deep reinforcement learning. In *Intelligent Systems Conference (IntelliSys), 2017*, pages 905–913. IEEE, 2017.

[176] Gyeeun Jeong and Ha Young Kim. Improving financial trading decisions using deep q-learning: Predicting the number of shares, action strategies, and transfer learning. *Expert Systems with Applications*, 117:125–138, 2019.