Bachelor's Thesis

Deep Learning Approximation for Optimal Execution Costs for Portfolios

Frederik Stihler

Department of Mathematics
Ludwig-Maximilians-Universität München

supervised by
Prof. Dr. Ari-Pekka Perkkiö

September 5, 2019

Contents

1 Introduction

Nowadays, investors often administer a portfolio consisting of several hundred securities [3]. Individual positions of the securities held by the portfolio managers can be very large and make up a considerable portion of the average daily trading volume of the security. Due to changing market conditions, institutional investors are forced to rebalance their portfolios frequently, for example by selling unfavorable stocks or buying new picks. To preserve the desired risk and reward profile of the portfolio, large orders across many different stocks must be executed in a relatively short period of time. The costs of executing such orders can be substantial and negatively affect the performance of asset managers [21]. Consequently, there is a great need for investors to control execution costs. Especially, the price impact of trading contributes to the execution costs for sizeable orders. The price impact describes the unfavorable effect that executing an order has on the prices of the respective securities [6]. For instance, a large purchasing order can move prices up and lead to higher execution costs for the trade. As a result, it is common in practice to split large orders into smaller packages that are executed over several short periods [7]. Hence, the challenge that arises for the investors is to find the optimal split and trading strategy to minimize incurring execution costs.

From a mathematical perspective, finding the optimal trading strategy represents a high-dimensional stochastic control problem, since the stock price development involves uncertainty. The traditional way of solving such problems is dynamic programming, where the problem is broken down into simpler subproblems in a recursive manner. However, the technique runs into difficulties for high-dimensional problems. This issue is called the "curse of dimensionality" and describes the effect that the run time of the dynamic programming algorithm grows exponentially with the dimensionality of the problem [2]. One possible approach to overcome this curse of dimensionality is to use deep learning and approximate the solution with deep neural networks [13].

Deep learning is a subfield of machine learning and is based on the use of artificial neural networks. In general, machine learning deals with algorithms and statistical models that enable computer systems to perform specific tasks without being explicitly programmed, by relying on pattern recognition. Artificial neural networks consist of several layers of interconnected artificial neurons, where each connection is associated with a weight. Complex networks with multiple layers are called deep neural networks. A neural network can be viewed as a function that is parameterized by the weights. Learning of the network can be realized by modifying these weights based on training data in a way that a cost function is minimized. After training, a network is able to generalize, make predictions and produce reasonable outputs from a set of inputs that was not used during training. This information processing capability of deep neural networks is called deep learning and makes it possible to find good approximate solutions to complex and high-dimensional problems, such as our stochastic control problem of minimizing execution costs for portfolios.

In this thesis, a finite time horizon model for execution costs for portfolios is used. The model is proposed in [3]. We implement and improve a deep learning approach developed by [13] to approximate the optimal trading strategy and minimize execution costs of this model. The aim is to approximate the optimal trading decisions at each time step by using artificial neural networks. These so-called subnetworks are then connected through the model dynamics to form a very deep neural net and are trained simultaneously. Our numerical results show that the deep neural network can learn the optimal split of the investors order across multiple stocks over the available time periods very well and approximate the optimal solution with high accuracy. Furthermore, it is worth mentioning that the general approach to solve high-dimensional stochastic control problems with deep learning has a wide range of applications other than minimizing execution cost, as the research in [13] shows. For example, it can also be applied to optimize resource allocation with many sources and demands or to dynamic game theory with many agents. Consequently, we keep the discussion of the theory and the deep learning approximation algorithm for optimal execution costs as general as possible, to facilitate the transfer of the basic ideas and principles to other fields of research.

This thesis comprises a comprehensive overview of the deep learning approach for approximating optimal execution costs for portfolios. It includes the necessary machine learning theory, the model specifications for the execution costs and the programming realization: First, the theoretical basis for understanding the deep learning methods used is established. The basic concepts of artificial neural networks are described, including the network architecture, functionality and the training process. Subsequently, a short review of stochastic control problems and the model of the execution costs for portfolios is presented. This encompasses a detailed description of how the control problem of execution costs is modelled and an outline of the deep learning approximation approach that is used to solve it. In the implementation part, the general structure of the code that implements the programming of the model and the construction of the deep neural network is explained. Finally, the results of the conducted numerical experiments are analyzed and the performance of the neural network regarding its ability to approximate the optimal sequence of trades and minimize the execution costs of a portfolio is discussed.

2 Theory of Artificial Neural Networks

2.1 Network Architecture

Artificial neural networks (ANNs) are computational systems that can learn to perform various tasks by considering training data, generally without being explicitly programmed with task specific rules. This places the research in ANNs in the broader field of machine learning, which is in turn a discipline of artificial intelligence (AI) in computer science. We count any technique that enables computers to mimic human behavior to AI [25]. In this context, ANNs are inspired by the biological brain of humans. However, neurons in ANNs interact by sending signals in the form of mathematical functions instead of electrical and chemical signals. An artificial neural network consists of multiple layers that are built from an arbitrary number of neurons. Usually, it contains an input layer, an output layer and eventual so-called hidden layers in between. If an ANN has one or more of these hidden layers, we speak of a deep neural network (DNN). Each connection between the neurons of different layers is represented by a weight variable. A visualization of an example ANN with two hidden layers and twenty neurons in total can be seen in Figure 1. The different colors of the connections in the picture symbolize the different values of the associated weight variables.

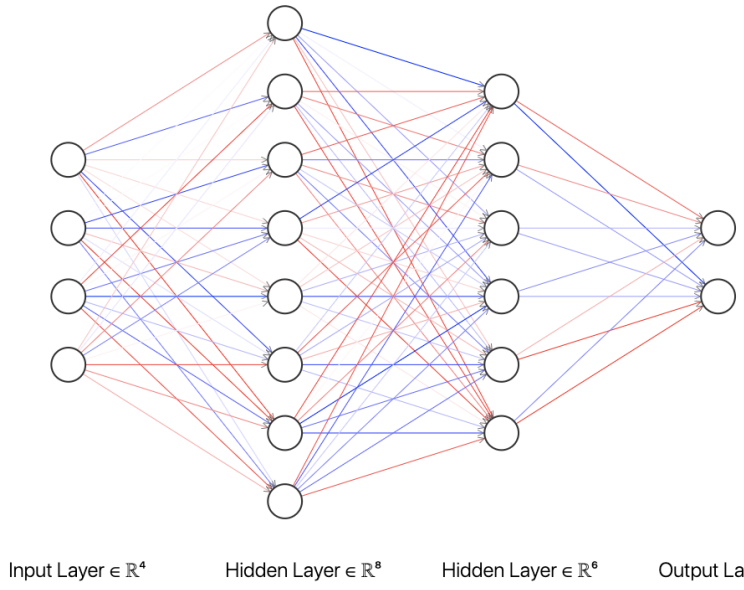


Figure 1: Illustration of a general artificial neural network with four layers and twenty neurons in total

In this thesis, we focus on one of the simplest types of ANNs: feed-forward neural networks (FFNNs). Feed-forward in this case means that information is only passed on forward

through the layers in the same direction. This flow of signals is represented in Figure 1 by the arrows pointing towards the outputs. We limit our attention to fully-connected FFNNs, meaning that each neuron of a layer in the net is connected to all neurons in the subsequent layer.

Nonetheless, it is worth mentioning that there is a lot of applications for network types that are not characterized by being fully-connected or passing information in one direction only. For instance, convolutional neural networks (CNNs) are emulating the structure of neurons in the visual cortex. In CNNs, neurons are only connected to a subset of nodes in the neighboring layers, making them able to learn local features of the inputs [12]. Consequently, these types of networks are well-suited for image recognition and video processing. Another very popular network type is the class of recurrent neural networks (RNNs). In RNNs, information does not flow in one direction only, which permits cycles in the network that allow internal memory of what has already been calculated before. Dealing with sequential information, such as sentences, is one of the best applications for RNNs. Hence, they are used for handwriting and speech recognition.

In the next section, we will have a closer look at the artificial neurons in the networks. In the human brain, biological neurons accumulate all incoming signals and pass on an output of a fixed amplitude only if an activation threshold is reached. The neurons remain inactive if this threshold is not reached. Artificial neurons were developed to act in a similar manner, but can be modified to be a little more flexible.

2.1.1 Artificial Neurons

Adaptive artificial neurons are the structural building blocks of ANNs and are sometimes called "simple perceptrons" in literature [26]. In Figure 2, one can see the basic structure of an artificial neuron. It consists of several inputs, a processing unit and an output.

The artificial neuron takes n inputs, denoted as $x_1, \ldots, x_n$ (see the blue nodes in Figure 2). We condense these inputs in a vector $\hat{x} \in \mathbb{R}^n$. Each input connection to the processing unit of the neuron is associated with a weight w_i for $i = 1, \ldots, n$. The weights are summarized in a vector $\hat{w} \in \mathbb{R}^n$. In addition, there is an option for adding a bias. The bias acts like an input that constantly signals the value 1 to the processing unit. However, it is also connected to the processing unit via a variable weight b. Within the processing unit, the weighted sum z of the inputs is calculated at first:

$$z = b + \sum_{i=1}^{n} x_i w_i. \tag{1}$$

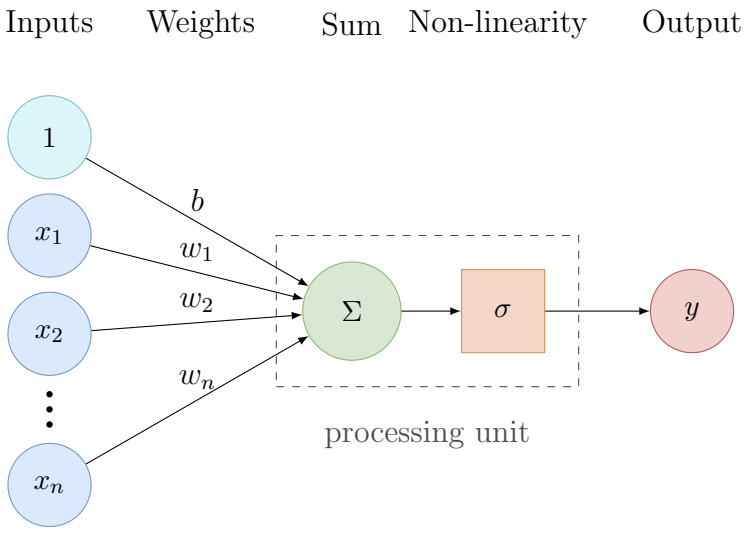


Figure 2: Illustration of an artificial neuron

In Figure 2, z is calculated in the green node labelled Σ. Here, the role of the bias becomes clear: It enables the shifting of this value, which will be processed by an activation function in the next step. Consequently, the bias allows to shift this function to the left or right, which may be critical for successful learning in the context of a complete neural network (see Section 2.2). Notice that we can also represent z in a vector notation:

$$z = b + \hat{x}^\intercal \hat{w}. \tag{2}$$

The next step in the processing unit of the artificial neuron is to apply a non-linear activation function σ to the weighted sum z in order to receive an output value y:

$$y = \sigma(b + \sum_{i=1}^{n} x_i w_i) = \sigma(b + \hat{x}^\intercal \hat{w}) = \sigma(z). \tag{3}$$

The application of the activation function is illustrated as an orange square in Figure 2 and the result is contained by the output node y, which is shown in red. This output is the value that the neuron can pass on to another neuron in a subsequent layer of the ANN. The exact mathematical formulation of this signal flow in an ANN is discussed in the following section.

2.1.2 Feed-forward Pass

As mentioned before, we focus on fully-connected feed-forward neural networks with one or more hidden layers that are made up of artificial neurons with non-linear activation functions. In this thesis, we may also refer to the neurons in a layer as nodes. This type of network is sometimes called a "Multilayer Perceptron (MLP)". In the following, we describe mathematically how the outputs of a complete ANN are derived from its inputs. Unfortunately, dealing with the theory of neural networks always involves a lot of indices, since there are several layers in a network, each containing many neurons and weights connecting them. For this reason, it is important to carefully handle this issue and precisely define the notation used in order to not get confused.

We assume that the network has $L \in \mathbb{N}$ layers in total with $L \geq 3$, where the first layer is the input layer and the L-th layer is the output layer. For $l = 1, \ldots, L$, let $N_l \in \mathbb{N}$ denote the number of nodes in the l-th layer. We label the accumulated incoming signal at the i-th node in the l-th layer by z_i^l for $l > 1$. This is just again the weighted sum of the inputs to the node, which in this case are the outputs of all nodes in the previous layer and the bias:

$$z_i^l = b_i^{l-1} + \sum_{j=1}^{N_{l-1}} y_j^{l-1} w_{ij}^{l-1}. \tag{4}$$

w_{ij}^{l-1} represents the weight associated with the connection from the j-th node in the $(l-1)$-th layer to the i-th node in the l-th layer. This order of indices will be useful for the matrix-vector notation we introduce in (7). The bias to the i-th node in the l-th layer is denoted by b_i^{l-1}. In total, there is one bias variable for each neuron with incoming signals in the network. The output of the i-th node in the l-th layer is called y_i^l. The outputs of the input layer are simply equal to the network's inputs $y_i^1 = x_i$. In the last layer L, the output layer, no activation function is applied in order to not restrict the range of output values of the entire ANN. Thus, the outputs in the final layer become $y_i^L = z_i^L$. The outputs of the nodes of the hidden layers, $1 < l < L$, are calculated as

$$y_i^l = \sigma(b_i^{l-1} + \sum_{j=1}^{N_{l-1}} y_j^{l-1} w_{ij}^{l-1}) = \sigma(z_i^l), \quad i = 1, \ldots, N_l, \tag{5}$$

where σ is the activation function. The activation function is assumed to be the same for all nodes in the hidden layers. In a more general setting, it is also possible to apply different activation functions, but will not be relevant in this thesis.

In total, we notice that after having computed all outputs of one layer we can calculate the values of the subsequent layer and so on. The information of the inputs is thus passed forward through the network, leading to the following recursive formulation of the output

of node i of a hidden layer l:

$$y_i^l = \sigma\left[b_i^{l-1} + \sum_{j=1}^{N_{l-1}} \sigma(b_j^{l-2} + \sum_{k=1}^{N_{l-2}} \sigma(\ldots \sigma(b_m^1 + \sum_{n=1}^{N_1} x_n w_{mn}^1)\ldots) w_{jk}^{l-2}) w_{ij}^{l-1}\right]. \tag{6}$$

This illustrates an important property of MLPs, which is that the only independent variables in the network are the inputs $x_1, \ldots, x_{N_1}$. In summary, an artificial neural network is despite its quite complex mathematical form just a mapping of real vectors $\hat{x} \in \mathbb{R}^{N_1}$ to $\hat{y}^L \in \mathbb{R}^{N_L}$, the outputs of the net. Furthermore, the expression in (6) is a nested sum of scaled activation functions. These functions can be shifted or the slope can be changed by adjusting the parameters of the ANN, which are the weights and biases. This leads to an enormous flexibility of the neural networks when being given the task to approximate a functional dependency between inputs and outputs.

In fact, it was shown by Cybenko [8] that "networks with one internal layer and an arbitrary continuous sigmoidal function can approximate continuous functions with arbitrary precision providing that no constraints are placed on the number of nodes or the size of the weights". This statement is known as the universal approximation theorem. Later on, it was proven that the theorem is not limited to sigmoidal functions only [17], but that the potential of approximating any continuous function is based on the architecture of the multilayer feed-forward neural nets. Some extensions to the popular *ReLU* function (see section 2.1.3) have also been developed [14].

As before, we can simplify the signal processing procedure, the so-called feed-forward pass of ANNs, by using a matrix-vector notation. We write $\hat{y}^l \in \mathbb{R}^{N_l}$ for the column vector that contains the outputs of layer $l = 1, \ldots, L$. The biases are also represented as a column vector $\hat{b}^l \in \mathbb{R}^{N_{l+1}}$ for $l < L$. The weights between layer l and $l+1$ are described by a weight matrix $W^l \in \mathbb{R}^{N_{l+1} \times N_l}$. Using this notation, we can rewrite the values of the accumulated incoming signals at layer l and summarize them in a column vector $\hat{z}^l$ for $l > 1$ as

$$\hat{z}^l = \begin{bmatrix} z_1^l \\ \vdots \\ \vdots \\ z_{N_l}^l \end{bmatrix} = \begin{bmatrix} b_1^{l-1} \\ \vdots \\ \vdots \\ b_{N_l}^{l-1} \end{bmatrix} + \begin{bmatrix} w_{11}^{l-1} & w_{12}^{l-1} & \ldots & w_{1N_{l-1}}^{l-1} \\ w_{21}^{l-1} & w_{22}^{l-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ w_{N_l 1}^{l-1} & \ldots & \ldots & w_{N_l N_{l-1}}^{l-1} \end{bmatrix} \begin{bmatrix} y_1^{l-1} \\ \vdots \\ \vdots \\ y_{N_{l-1}}^{l-1} \end{bmatrix} = \hat{b}^{l-1} + W^{l-1}\,\hat{y}^{l-1}. \tag{7}$$

Again, the outputs of the hidden layers are

$$\hat{y}^l = \sigma(\hat{b}^{l-1} + W^{l-1}\,\hat{y}^{l-1}) = \sigma(\hat{z}^l), \quad l = 2, \ldots, L-1. \tag{8}$$

Note that we use the vectorized form of the activation function σ, where σ is applied componentwise. A closer look at different activation functions is taken in the next section.

2.1.3 Activation Functions

A selection of common non-linear activation functions is presented in Figure 3. The non-linearity is important to enable the potential of the network to approximate non-linear functions too. Another desirable property of activation functions is continuous differentiability. When using gradient descent methods, this property can facilitate the optimization of the loss function (see Section 2.2.2). Note that the $ReLU$ function is not continuously differentiable. However, gradient based optimization can still be implemented with it, as it is nearly linear [10]. The $ReLU$ function is defined as follows:

$$ReLU(x) = max(0, x). \tag{9}$$

In comparison, the step function, which is the function that mimics best the activation signals in the human brain, is also not differentiable at 0. Yet, its derivative for all other values is 0, because it signals either 0 or 1, depending on whether the activation threshold is reached (see Figure 3, upper right). Thus, gradient descent algorithms with this binary function will not be able to decrease the loss function and the network cannot learn. Consequently, the sigmoid function

$$sig(x) = \frac{1}{1 + e^{-x}}$$

and the hyperbolic tangent

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

were traditionally used as activation functions in practice [24], because they resemble the binary step function and have bounded ranges, so that the signals cannot get too big. The $tanh$ function became a little more popular than the sigmoid function due to the fact that it produces zero centered outputs, which is beneficial for the training procedure (see Section 2.3.1).

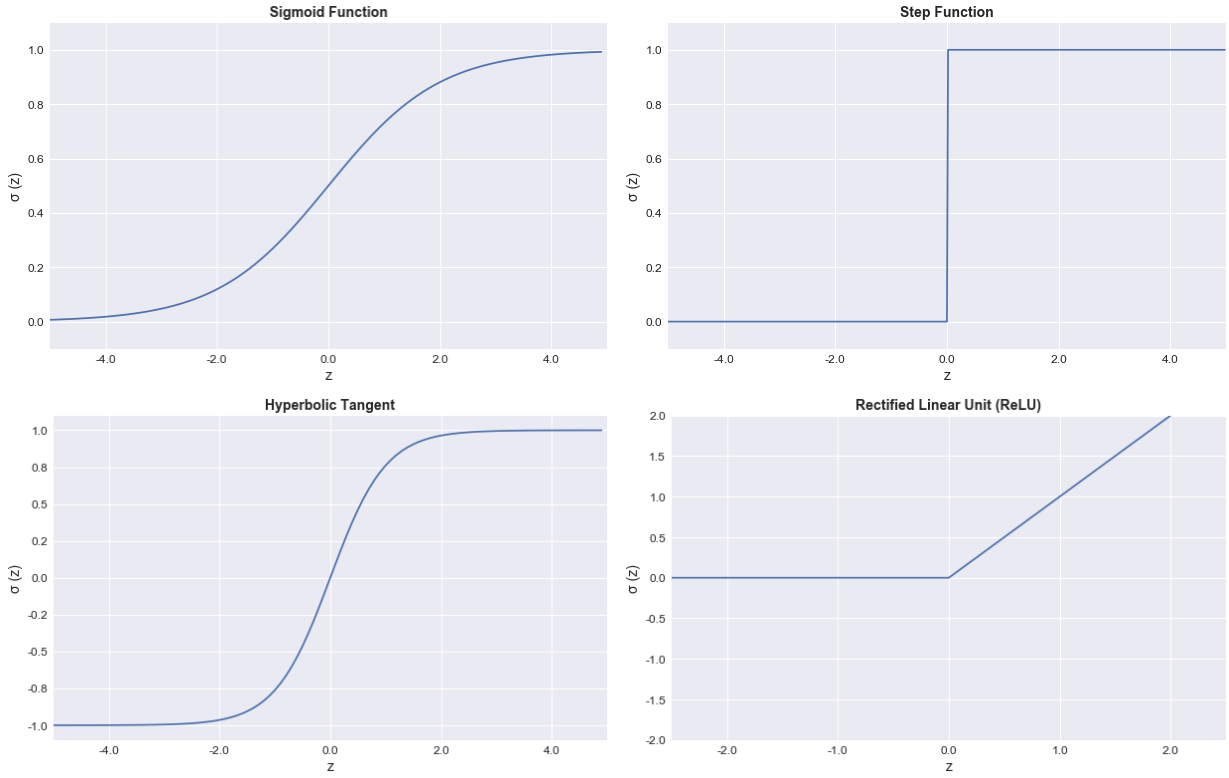


Figure 3: Graphs of four different non-linear activation functions

Unfortunately, both the sigmoid function and the hyperbolic tangent can suffer from the vanishing gradient problem when training the ANN (see Section 2.2.3). As a solution, *ReLU* is nowadays the most common activation function [24], although it is unbounded and signals can get arbitrarily large. This is because of the non-saturation of its gradients and its resulting ability to reduce the vanishing gradient problem. Moreover, it is obviously less expensive to compute, because it avoids divisions and exponentials. We conclude that the choice of activation function is very important for the overall ability of the ANNs to learn and can greatly affect the speed of training. The training of ANNs is discussed in the following section.

2.2 Training of Neural Networks

Training an ANN requires training data. The training data set usually consists of input data and target values that are associated with the input data, for instance when the network should solve a classification problem. Yet, there exist applications of neural networks that are trained on data sets without target values. We denote the training set as $X_{train} = \{{}_1\hat{x}, \ldots, {}_D\hat{x}\}$ with D samples and do not consider possible target values. In machine learning, the training data is used to fit a model. In the case of deep learning, this model is a DNN. So the network 'learns' based on the data of X_{train}, which means that the

parameters of the net, the weights and biases, are adjusted in a way that the performance of the DNN increases. In order to objectively measure the final performance and accuracy of the model, there exists a test data set that serves as unbiased evaluation data. The test data set ideally consists of 'unseen' data, meaning that it was not used during training. Besides the training and the test data, the use of an additional validation set is common in practice. It is also used to give an unbiased evaluation of the model. However, the model is evaluated on the validation set continuously during training. This is done to carefully tune the model's hyperparameters, which will be discussed later, and should help detect overfitting in an early stage.

2.2.1 Quantifying Loss

An artificial neural network can learn by adjusting its weights and biases, as mentioned before. This adjustment is not done arbitrarily, but in a systematic manner. For this purpose, we introduce a loss function L that quantifies the loss incurred from the outputs of the network. Thus, the loss function should allow a measurement of the performance of the network, whereby improvements of the loss value should indicate a better model performance. The loss is calculated based on the outputs $\hat{y}^L$ of the ANN. Recall that these outputs are generated by passing on the inputs $\hat{x}$ through the complete network and applying a series of matrix-vector multiplications, vector additions, as well as the activation function (see for example (8)). Hence, the outputs $\hat{y}^L$ are dependent on the inputs and parameters of the ANN. Consequently, we can formulate the loss for a single training sample as a function

$$\begin{aligned} &L : \mathbb{R}^{N_L} \to \mathbb{R}, \;\; \hat{y}^L \mapsto L(\hat{y}^L) \\ &L(\hat{y}^L) = L(\hat{y}^L({}_i\hat{x}, W, b)), \end{aligned} \tag{10}$$

where ${}_i\hat{x} \in \mathbb{R}^{N_1}$ is the i-th sample of the training set X_{train}, W is the collection of all weights in the network $W = \{W^1, \ldots, W^{L-1}\}$ and b are all biases $b = \{\hat{b}^1, \ldots, \hat{b}^{L-1}\}$. Sometimes, the loss function is also referred to as cost function and can additionally depend on possible target values. With this loss function for individual training samples, we define the empirical loss $\mathcal{L}$ as the average loss over the entire training data set:

$$\mathcal{L}(W, b) = \frac{1}{D} \sum_{d=1}^{D} L(\hat{y}^L({}_d\hat{x}, W, b)). \tag{11}$$

Notice that we do not see the empirical loss function $\mathcal{L}$ as a function of the inputs of the training set X_{train}, as we expect the entire training set to be fixed. It rather is a function with a parameter space consisting only of the weights and biases, because these are the parameters that we can modify, while input data and possible target values are not something that the ANN learns. From now on, we may call the empirical loss $\mathcal{L}$ simply loss while making sure that it is evident from the context, whether the loss of a single training

example L or over the entire data set $\mathcal{L}$ is meant.

Overall, the aim of training is to adjust the weights and biases in the network in a way that the empirical loss function is minimized. Therefore, we can interpret the training process as solving an optimization problem. The loss function represents the objective function and acts as a measure to evaluate a set of parameters as a candidate solution in the context of this optimization problem. To find the optimal weights W^* and biases b^* that achieve the lowest loss, we formulate the loss optimization problem:

$$(W^*, b^*) = \underset{(W,b)}{\operatorname{argmin}} \mathcal{L}(W, b) = \underset{(W,b)}{\operatorname{argmin}} \frac{1}{D} \sum_{d=1}^{D} L(\hat{y}^L({}_d\hat{x}, W, b)). \tag{12}$$

In the case of a classification problem, minimizing the loss function could lead for example to the outputs of the ANN being close to the target values in the training set, when feeding it with the associated input training data. The network can then classify the inputs in different categories or make certain predictions. However, we often encounter the problem of overfitting in practice, meaning that the ANN is trained too specifically on the training set and can perform poorly on 'unseen' test data. Preventing overfitting is therefore an important task in machine learning. The techniques commonly used to find the solution of the optimization problem are presented in the next section.

2.2.2 Optimizing the Loss Function

The optimization problem in (12) can be solved or approximated in several ways. The most common approach in practice however is to use a version of a gradient descent algorithm, due to its computational advantages over other optimization methods. For instance, finding the optimal solution analytically might not be possible or could be very difficult, because of the extremely high dimensionality of the neural network's parameter space. In the following section, the main principles, benefits and disadvantages of gradient descent methods are presented, as well as some extensions to the basic algorithm. Nevertheless, for a more detailed discussion of the topic and the algorithm for computing the gradients, see for example [4], [27] and [28].

The idea of the gradient decent algorithm is to take small steps towards a descent direction until a minimum is reached (see Algorithm 1). Hence, the method is an iterative optimization algorithm for minimization. It evaluates the objective function, in our case the loss function, at a random starting point. Then, a step towards the direction of the steepest descent of the function, which is the negative gradient, is taken. This procedure is repeated until the gradient is zero (or small enough to satisfy a stopping criterion) and a stationary point, ideally a global minimum, is reached. Here, some of the biggest challenges of descent algorithms already come to light. First, instead of converging to the global minimum, the algorithm may get stuck in a local minimum or saddle point. Moreover, the algorithm is sensitive to the starting point, which in our case is the initialization of the weights and bi-

ases. Depending on the randomly selected starting parameters, the algorithm may or may not converge. Finally, the choice of the step size can play a major role for the convergence and speed of the algorithm and thus has to be made carefully.

For clarity, we denote the parameters W and b of the network as $\theta = (W, b)$. One can think of θ as a column vector listing all of the parameters. In its most basic form, the gradient descent algorithm for neural networks to find a minimum then reduces to:

***Algorithm 1* : Regular Gradient Descent**
Require: Objective function $\mathcal{L}(\theta)$ with parameters θ
Require: Step size $\eta > 0$

---

1. Initialize parameters θ randomly
2. *While* θ not converged *do*:
   2.1 Compute gradient: $\nabla_\theta \mathcal{L}(\theta)$
   2.2 Update parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}(\theta)$

   end while
3. Return parameters θ

---

where η is called the learning rate, representing the step size in the descent algorithm. The gradient of the loss function $\nabla_\theta \mathcal{L}(\theta)$ is a column vector containing all partial derivatives of the loss function with respect to the parameters $\frac{\partial \mathcal{L}}{\partial \theta_i}$, for $i = 1, \ldots, \sum_{l=2}^{L}(N_{l-1}+1)N_l$. The total number of parameters is just the number of all biases and weights in the network, meaning that the gradient contains all partial derivatives $\frac{\partial \mathcal{L}}{\partial w_{ij}}$ and $\frac{\partial \mathcal{L}}{\partial b_i}$.

The regular gradient descent algorithm has a couple of disadvantages. Therefore, it is not used in practice very often. Yet, the basic idea of reducing the loss function step by step by moving towards a descent direction has proven to be useful. The main drawbacks of implementing regular gradient descent are that it can run into memory issues and be very slow. For each update in the algorithm, the gradient of the loss function for the whole training data set has to be calculated. If the training set is large, it might not fit into the computer memory. Consequently, it makes sense to look for alternatives to this algorithm that tackle some of the above mentioned problems.

One variant of Algorithm 1 is stochastic gradient descent (SGD). In contrast, the SGD algorithm performs a parameter update for each individual training example ${}_i\hat{x}$. The true gradient of $\mathcal{L}(\theta)$ is approximated by a gradient at a randomly selected single training example. The update in 2.2 of Algorithm 1 then becomes

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\hat{y}^L({}_i\hat{x}, \theta)).$$

This method is usually much faster, due to the frequent updates [27]. Additionally, the noise in the approximation of the true gradient can help the algorithm escape local minima. On the downside, this fluctuation can also lead the algorithm in unfavorable directions. Lastly, the SGD loses the advantage of vectorized operations, as it only deals with a single training sample at a time.

A compromise between computing the true gradient over all training data and the gradient for each individual training sample is offered by the so-called mini-batch gradient descent method, which can perform significantly better than regular gradient descent or SGD. The training set X_{train} with D data points is divided randomly into equally sized mini-batches, which are used to approximate the gradient of the loss function for the parameter update. If the batch size is chosen to be K, assuming that D is divisible by K, then there will be $\frac{D}{K}$ mini-batches, which we can call $B_i = \{{}_1\hat{x}_i, \dots, {}_K\hat{x}_i\}$ for $i = 1, \dots, \frac{D}{K}$. As the batches are created randomly from the training set, we have ${}_k\hat{x}_i \in X_{train}$ for all $k \leq K$. Moreover, we state that the mini-batches should form a disjoint union of the training set:

$$X_{train} = B_1 \dot{\cup} \dots \dot{\cup} B_{\frac{D}{K}}.$$

From the fact that the loss function $\mathcal{L}$ can be written as an average over all training data (see (11)) and by the linearity of the gradient, we can conclude that the gradient of the loss function can also be written as an average:

$$\nabla_\theta \mathcal{L}(\theta) = \nabla_\theta \left(\frac{1}{D} \sum_{d=1}^{D} L(\hat{y}^L({}_d\hat{x}, \theta)) \right) = \frac{1}{D} \sum_{d=1}^{D} \nabla_\theta L(\hat{y}^L({}_d\hat{x}, \theta)). \tag{13}$$

Hence, we can also calculate the gradient over some mini-batch data by averaging the gradients of each individual training sample in the batch. With this in mind, we formulate the following algorithm:

***Algorithm 2* : Mini-Batch Gradient Descent**
Require: Loss function L for individual training examples
Require: Step size $\eta > 0$

1. Initialize parameters θ randomly
2. *While* θ not converged *do*:
   2.1 Randomly create $\frac{D}{K}$ mini-batches $B_i = \{{}_1\hat{x}_i, \ldots, {}_K\hat{x}_i\}$ of size K from X_{train}
   2.2 *For* $i = 1, \ldots, \frac{D}{K}$ *do*:
      2.2.1 Define average loss function over mini-batch data: $\mathcal{L}_i(\theta) := \frac{1}{K} \sum_{k=1}^{K} L(\hat{y}^L({}_k\hat{x}_i, \theta)$
      2.2.2 Compute gradient: $\nabla_\theta \mathcal{L}_i(\theta) = \frac{1}{K} \sum_{k=1}^{K} \nabla_\theta L(\hat{y}^L({}_k\hat{x}_i, \theta))$
      2.2.3 Update parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}_i(\theta)$

   end while
3. Return parameters θ

As the algorithm does not process each training sample separately, it can make use of vectorized operations and at the same time does not have to memorize all of the training data at once. Furthermore, this method may result in a smoother convergence compared to SGD, because the computed gradient for each parameter update is averaged over K training examples. This means that the directions taken are less noisy, however, it can still help escape local minima. These are some of the reasons why using mini-batches is very common when implementing a form of gradient descent in the field of deep learning [27].

Nevertheless, the algorithm still has some disadvantages. The batch-size for example represents an extra hyperparameter that has to be tuned during training (see 4.3.2). Besides, there is no guarantee for convergence and choosing an appropriate learning rate can be difficult, a challenge that is discussed in Section 2.3.2.

2.2.3 The Vanishing Gradient Problem

One obstacle to the effective training of DNNs with gradient based optimization methods is the vanishing gradient problem [16]. As the name suggests, it deals with the effect that partial derivatives of the loss function with respect to the weights can get smaller and smaller for the early layers in a network. The more layers there are in a network, the worse this effect can be. The vanishing gradient problem results from the fact that calculating the gradients via back-propagation requires a repeated application of the chain rule, leading to the multiplication of the derivatives. If these are small in value, i.e. < 1, then we have an exponential decrease of the gradient. This is an undesirable phenomenon, as we

learned that small gradients lead to smaller updates in the gradient descent optimization algorithms and thus making the network very hard to train. The slow learning in the early layers can also lead to a poor accuracy of the whole network, because the initial layers are often crucial to recognize basic patterns in the input data [30].

Another factor responsible for this problem is the choice of the activation function [9]. Looking for example at the sigmoid function in Figure 3, we see that large changes in the incoming signals might not have a large effect on the output of the function if the signals are large in absolute value. This means that the derivative of the sigmoid activation function is very small at its tails. We can also see this by looking at

$$
\begin{aligned}
sig(x) &= \frac{1}{1+e^{-x}} \\
sig'(x) &= \underbrace{sig(x)}_{\in(0,1)}\underbrace{(1-sig(x))}_{\in(0,1)} = \frac{e^{-x}}{(1+e^{-x})^2}.
\end{aligned}
$$

Notice that the derivative of the sigmoid function is always less than one, which can cause the exponential decrease of the gradients described above. A possible solution that works effectively in practice is to use the $ReLU$ activation function, $ReLU(x) = max(0, x)$, because it does not have the problem of derivatives being too small for signals larger than zero:

$$
ReLU'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0. \end{cases}
$$

In addition, the derivative of the $ReLU$ function is very easy to compute. Nevertheless, using $ReLU$ also has some disadvantages. For example, neurons with the $ReLU$ activation function that receive a negative signal can die out due to the fact that they will only output 0 and the derivative will also become 0. This can prevent the neurons from further learning and is called the dying $ReLU$ problem [23]. Besides the choice of the activation function, it was found that the initialization of the weights can play a major role in tackling the vanishing gradient problem [9], as well as the use of batch normalization [18]. This technique is discussed in the following section.

2.3 Tuning of Neural Networks

2.3.1 Batch Normalization

Batch normalization is a technique developed to address the vanishing gradient problem (see Section 2.2.3) and speed up the training process [18]. Additionally, it deals with the general problem that the distribution of each layer's inputs changes during training, as the weights in the previous layers are adjusted. The changing input distributions require the layers to continuously adapt, slowing down training and requiring a lower learning rate, as well as a careful parameter initialization. To avoid these problems, batch normalization can be adopted just before applying the activation function in each layer of a DNN.

Implementing batch normalization means making normalization a part of the model architecture. It is performed for each training batch. Basically, the technique zero-centers the inputs for each layer and normalizes them, before scaling and shifting the results again using two new trainable parameters. Essentially, these two parameters, which we call the scaling parameter γ and the shifting parameter β, allow the model to learn the optimal scale and mean of each layer's inputs. In order to be able to perform the centering and normalization, the mean and standard deviation of the inputs of the entire training set have to be estimated. This is done by computing the mean and standard deviation of the current training batch, giving the technique its name.

Let B_i be a training batch with K input samples. Since the normalization transformation is applied at each layer independently, we focus on the accumulated incoming signals at one particular layer l. Hence, we have K values of these signals in the batch: ${}_1\hat{z}_i^l, \ldots, {}_K\hat{z}_i^l$. For clarity, we will drop the indices for the layer and the batch in this section. Then, the algorithm for the batch normalization transformation is:

***Algorithm 3* : Batch Normalization**

Require: Accumulated incoming signals ${}_1\hat{z}, \ldots, {}_K\hat{z}$ at layer l of batch B_i

Require: Scaling parameter γ and shifting parameter β

Require: Smoothing term $\epsilon > 0$

---

1. Calculate batch mean: $\mu_b = \frac{1}{K} \sum_{k=1}^{K} {}_k\hat{z}$
2. Calculate batch variance: $\sigma_b^2 = \frac{1}{K} \sum_{k=1}^{K} ({}_k\hat{z} - \mu_b)^2$
3. *For* $k = 1, \ldots, K$ *do*:
   - 3.1 Normalize the signals: ${}_k\hat{z} \leftarrow \frac{{}_k\hat{z} - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}$
   - 3.2 Scale and shift: ${}_k\hat{z} \leftarrow \gamma \, {}_k\hat{z} + \beta$
4. Return transformed signals ${}_k\hat{z}$

The distribution of values of the normalized signals in step 3.1 in Algorithm 3 for any sample ${}_k\hat{z}$ has an expected value of 0 and a variance of 1, as long as the samples of the batch are drawn from the same distribution and if we neglect ϵ. We use ϵ as a so-called smoothing term and choose it to be a very small number, e.g. 10^{-10}, to avoid division by zero.

Notice that when using batch normalization, we do not need a bias term anymore, such as described in Section 2.1.1, because the mean subtraction in the batch normalization would eliminate it. The shifting parameter β takes over the role of the bias in this case. Although batch normalization is found to significantly reduce the vanishing gradients problem, allow higher learning rates and make the networks less sensitive to weight initialization, it adds complexity to the model and can lead to longer running times when making predictions, due to the extra computations in each layer.

2.3.2 Learning Rate Scheduling

Previously, we have found out that the choice of the learning rate is important for the speed and convergence of the gradient descent algorithm (see Section 2.2.2). If the learning rate is set too high, the training algorithm may diverge and the cost function cannot be minimized. Another possibility is that the model learns quickly in the beginning, but does not converge to an optimal solution, because the algorithm jumps around the optimum. On the other hand, if the learning rate is set too low, the optimal solution is reached very slowly [12]. Consequently, if a constant learning rate is used, it has to be tuned carefully to find a good balance.

Learning rate scheduling offers an alternative to using a constant rate. The idea is to decay the learning rate during training such that the optimization algorithm can make fast progress in the beginning and gets more accurate towards the end. For instance, in a version called 'performance scheduling' one starts with a high learning rate and reduces it, when the loss function stops decreasing. With this technique, a good solution can be found faster than with constant rates. There are many different learning rate reduction strategies. An easier implementation of learning rate scheduling is to use a 'predetermined piecewise constant learning rate'. The lowering of the learning rate is fixed in advance and performed after a predetermined number of iterations. Although this might work very well, it also requires fine tuning. In addition to the learning rate decay, it was found that increasing the batch size can also help obtain a better learning curve [29].

2.3.3 Training Acceleration with Adam Optimizer

The final measure to ensure an even faster training process that we discuss is using an advanced gradient descent optimization algorithm, such as the Adam algorithm. Similar to regular gradient descent, SGD and mini-batch gradient descent, the algorithm aims to find a solution to problem (12) and only uses first-order partial derivatives. Although the mini-batch gradient descent presented in Algorithm 2 is good for understanding the use of training batches and the principles of the learning process of ANNs, the Adam algorithm is shown to perform much better for deep learning applications in practice [20]. Advantages of the Adam algorithm include its computational efficiency and its little memory requirements.

The Adam method keeps track of an exponentially decaying average of past gradients, which is known from the so-called momentum optimizer, and also of an exponentially decaying average of past squared gradients (see m and v in steps 2.2.4 and 2.2.5 of Algorithm 4). The idea is to use the gradient as acceleration, leading to the ability to take larger steps towards relevant directions with steep slopes and be able to escape plateaus [27]. As a result, it requires less tuning of the learning rate hyperparameter η.

The name Adam is derived from adaptive moment estimation, as Adam uses adaptive estimates of lower-order moments. For a compact presentation of the Adam algorithm, we first need to define the entrywise product $\circ$ and entrywise division $\oslash$ of vectors: For two real matrices or vectors A and B of the same dimension $n \times m$, the entrywise product $A \circ B$ is a matrix or vector of the same dimension with elements given by $(A \circ B)_{ij} = (A)_{ij}(B)_{ij}$, while the entrywise division $A \oslash B$ is a matrix or vector of the same dimension with elements given by $(A \oslash B)_{ij} = (A)_{ij}/(B)_{ij}$. Adding a constant to a vector or the application of the square root are also componentwise. Then, the Adam algorithm with mini-batches is:

***Algorithm 4* : Adam with Mini-batches**

Require: Loss function L for individual training examples
Require: Step size $\eta > 0$
Require: Exponential decay rates for the moment estimates: $\alpha_1, \alpha_2 \in [0, 1)$
Require: Smoothing term $\epsilon > 0$

1. Initialization

   1.1 Initialize parameters θ randomly

   1.2 Initialize first and second moment vectors: $m \leftarrow 0, \quad v \leftarrow 0$

   1.3 Initialize iteration number: $t \leftarrow 0$

2. *While* θ not converged *do*:

   2.1 Randomly create $\frac{D}{K}$ mini-batches $B_i = \{{}_1\hat{x}_i, \ldots, {}_K\hat{x}_i\}$ of size K from X_{train}

   2.2 *For* $i = 1, \ldots, \frac{D}{K}$ *do*:

      2.2.1 $t \leftarrow t + 1$

      2.2.2 Define average loss function over mini-batch data: $\mathcal{L}_i(\theta) := \frac{1}{K} \sum_{k=1}^{K} L(\hat{y}^L({}_k\hat{x}_i, \theta)$

      2.2.3 Compute gradient: $\nabla_\theta \mathcal{L}_i(\theta) = \frac{1}{K} \sum_{k=1}^{K} \nabla_\theta L(\hat{y}^L({}_k\hat{x}_i, \theta))$

      2.2.4 Update biased mean estimate: $m \leftarrow \alpha_1 \cdot m + (1 - \alpha_1) \cdot \nabla_\theta \mathcal{L}_i(\theta)$

      2.2.5 Update biased variance estimate: $v \leftarrow \alpha_2 \cdot v + (1 - \alpha_2) \cdot \nabla_\theta \mathcal{L}_i(\theta) \circ \nabla_\theta \mathcal{L}(\theta)$

      2.2.6 Compute bias-corrected mean estimate: $\hat{m} \leftarrow \frac{1}{1-\alpha_1^t} \cdot m$

      2.2.7 Compute bias-corrected uncentered variance estimate: $\hat{v} \leftarrow \frac{1}{1-\alpha_2^t} \cdot v$

      2.2.8 Update parameters: $\theta \leftarrow \theta - \eta \cdot \hat{m} \oslash \sqrt{\hat{v} + \epsilon}$

   end while

3. Return parameters θ

Observe that the hyperparameters α_1 and α_2 control the exponential decay rates of the weighted moving averages m and v. These moving averages estimate the mean and the uncentered variance of the gradient. However, the moment estimates are found to be biased towards zero in the beginning of the iteration and especially when using small decay rates [20]. This happens due to the fact that they are initialized with 0 (as a vector). Hence, a computation of bias-corrected estimates is introduced in steps 2.2.6 and 2.2.7. This will boost m and v at the start, avoiding this problem.

3 Optimal Execution Costs for Portfolios

In this section, we introduce a model for execution cost for portfolios and develop a deep learning approach to solve the problem of minimizing these costs. The execution costs are the costs that are associated with the execution of investment strategies and are composed of several components. They include explicit transaction costs, such as commissions and bid/ask spreads, as well as costs that are not directly measurable, such as opportunity costs of waiting and the price impact from trading [21]. Especially for institutional investors and portfolio managers with large positions, execution costs can have a substantial negative impact on the performance of investments and consequently need to be controlled carefully [3]. Using the execution cost model described in Section 3.2.1, this requires solving a stochastic control problem, which we will approach with a deep learning approximation algorithm.

In our cost model, we capture the price impact from trading in a stock on itself and on other stocks in the portfolio. The price impact describes the unfavorable effect that executing a buying or selling order has on the price of the respective security. For instance, a large purchasing order can move prices up and lead to higher execution costs. Moreover, the bigger the order, the stronger the price impact, which is also called market impact. As a result, it is common in practice to split large orders into smaller packages that are executed over several time periods [7]. We aim to find an optimal trading strategy that splits and executes orders across several stocks, using artificial intelligence.

3.1 Stochastic Control Problems

Before specifying the concrete model for the execution costs for portfolios, we introduce the main principles of stochastic control theory. In general, optimal control theory deals with optimizing a sequence of actions over a period of time to attain some future goal, for instance the minimization of a sum of path costs [19]. The optimal sequence of actions is dependent on a dynamic system. In stochastic control problems, the evolution of this system is not fully deterministic, but also driven by random noise that affects the state variables. We consider a stochastic control problem in discrete time within a finite horizon T on a general probability space $(\Omega, \mathcal{F}, P)$. In addition, a filtration $\mathcal{F}_0 \subset \mathcal{F}_1 \subset \ldots \subset \mathcal{F}_T = \mathcal{F}$ is required. For convenience, we adopt the convention throughout this thesis that any by t indexed variable is $\mathcal{F}_t$-measurable. Important variables of the stochastic control problem are written in bold.

Next, we introduce the state variable $\boldsymbol{s}_t$ and the control variable $\boldsymbol{a}_t$. The state variable describes the state of the dynamic system at time t. We let $\boldsymbol{s}_t \in \mathcal{S}_t \subset \mathbb{R}^d$, where $\mathcal{S}_t$ is the set of possible states. The n-dimensional control variable $\boldsymbol{a}_t \in \mathbb{R}^n$ represents the action that is decided at time t and will influence the state variable at the next point in time. We use both the terms controls and actions interchangeably for these variables $\boldsymbol{a}_t$. Moreover, it is assumed that the control variable $\boldsymbol{a}_t$ only depends on the current state $\boldsymbol{s}_t$,

meaning that the state variable fully describes the dynamic system. Hence, we define $\boldsymbol{a}_t$ to be an element of the set of admissible actions $\boldsymbol{a}_t \in \mathcal{A}_t = \{\boldsymbol{a}_t(\boldsymbol{s}_t) : \mathcal{S}_t \to \mathbb{R}^n\}$. Similar to the execution cost model that will be presented in Section 3.2.1, we do not consider state-dependent constraints on the controls.

Due to the fact that $\boldsymbol{s}_T$ is the final state, we only have controls at times $t = 0, 1, \ldots, T-1$. The evolution of the system can now be described as a stochastic model:

$$\boldsymbol{s}_{t+1} = \boldsymbol{s}_t + g_t(\boldsymbol{s}_t, \boldsymbol{a}_t) + \boldsymbol{\xi}_{t+1}, \quad t = 0, 1, \ldots, T-1. \tag{14}$$

In this model, g_t is the deterministic drift, which depends only on the current state and the action taken: $g_t(\boldsymbol{s}_t, \boldsymbol{a}_t) : \mathcal{S}_t \times \mathbb{R}^n \to \mathbb{R}^d$. The deterministic drift is noiseless and can be rewritten as $g_t(\boldsymbol{s}_t, \boldsymbol{a}_t(\boldsymbol{s}_t))$. However, as we are dealing with a stochastic model, an $\mathcal{F}_{t+1}$-measurable random variable $\boldsymbol{\xi}_{t+1} \in \mathbb{R}^d$ is added. This random variable represents the uncertainty in the model dynamics and comprises all noisy information arriving during the time period between t and $t+1$.

As mentioned before, the overall goal is to optimize an objective function by selecting the optimal control variables. For example, one can think of maximizing an expected reward or minimizing expected costs. In this thesis, we consider a minimization problem. We denote the intermediate cost that is associated with taking action $\boldsymbol{a}_t$ at time t in state $\boldsymbol{s}_t$ as $\boldsymbol{c}_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$. Additionally, we define the cumulative cost as

$$\boldsymbol{C}_t = \sum_{\tau=0}^{t} \boldsymbol{c}_\tau(\boldsymbol{s}_\tau, \boldsymbol{a}_\tau), \quad t = 0, 1, \ldots, T-1. \tag{15}$$

For simplicity, we do not take into account a final cost that is associated with ending up in state $\boldsymbol{s}_T$ at time T and is not directly related to a decision made. Consequently, $\boldsymbol{C}_{T-1}$ stands for the total cost of the problem. Note that the last state $\boldsymbol{s}_T$ can be ignored, since the calculation of $\boldsymbol{C}_{T-1}$ does not consider it. Overall, the stochastic control problem can now be formulated as

$$\min_{\substack{\boldsymbol{a}_t \in \mathcal{A}_t \\ t=0,\ldots,T-1}} \mathbb{E}_P[\boldsymbol{C}_{T-1} \mid \boldsymbol{s}_0] = \min_{\substack{\boldsymbol{a}_t \in \mathcal{A}_t \\ t=0,\ldots,T-1}} \mathbb{E}_P[\sum_{t=0}^{T-1} \boldsymbol{c}_t(\boldsymbol{s}_t, \boldsymbol{a}_t(\boldsymbol{s}_t)) \mid \boldsymbol{s}_0], \tag{16}$$

which means that the objective is to find the sequence of actions that minimizes the expected total cost, given the initial state $\boldsymbol{s}_0$. Traditionally, these problems are solved with dynamic programming [5]. However, we focus on the presentation of a deep learning approach to approximate the optimal solution, avoiding the need for a dynamic programming algorithm.

3.2 Mathematical Formulation of Execution Costs

3.2.1 The Basic Model

In this section, the mathematical model for the execution costs for portfolios is defined. To bring into focus the deep learning approach for solving stochastic control problems, the model is prevented from growing too complicated. We consider a finite time horizon T and a portfolio that consists of n stocks. The fixed blocks of shares of the n stocks of the portfolio to be purchased within T periods are denoted by $\bar{\boldsymbol{a}} = (\bar{a}_1, \bar{a}_2, \dots, \bar{a}_n)^\intercal \in \mathbb{N}^n$. This represents a purchasing order that an investor has to fulfill. Notice that without loss of generality we limit the scope of this model to a buying program, meaning that only the acquisition of a certain number of stocks within the determined portfolio is desired. A selling order or a combination of buying and selling could be implemented easily by using appropriate signs. So, after T periods, $\bar{a}_i$ is the number of shares that must have been bought of stock i of the portfolio.

We label the number of shares of each stock acquired in period t by $\boldsymbol{a}_t = (a_{1t}, \dots, a_{nt})^\intercal \in \mathbb{R}^n$ for $t = 0, \dots, T-1$. These are the control variables in the stochastic control problem. Note that for simplification reasons, we do not require these variables to be natural numbers, although in practice it might not be possible to buy fractions of a share. The execution prices of these stocks at the time of purchase are contained in $\boldsymbol{p}_t = (p_{1t}, \dots, p_{nt})^\intercal \in \mathbb{R}^n$. The execution price represents the amount of money that an investor actually has to pay for receiving the stocks (see Section 3.2.2). The objective of the investor now is to minimize the expected total cost of trading in the n stocks to complete the order $\bar{\boldsymbol{a}}$ in time T. We state this minimization problem of the investor as

$$\min_{\{\boldsymbol{a}_t\}_{t=0}^{T-1}} \mathbb{E}_P[\sum_{t=0}^{T-1} \boldsymbol{p}_t^\intercal \boldsymbol{a}_t] \tag{17}$$

subject to:

$$\sum_{t=0}^{T-1} \boldsymbol{a}_t = \bar{\boldsymbol{a}}. \tag{18}$$

For the sake of clarity, we omit the dependency on the initial state in the formulation of the stochastic control problem above. In principle, the aim is to find the optimal sequence of trades as a function of the state variables $\boldsymbol{s}_t$ (which we will define in the next section) that minimizes the expected execution costs of the order $\bar{\boldsymbol{a}}$.

In this model, we do not impose additional constraints, such as a no-sales condition $\boldsymbol{a}_t \geq 0$ or other institutional restrictions and tax considerations. Hence, we allow the sale of shares within the time horizon of the order, although we are considering a buying program. Yet,

the addition of constraints that are relevant to portfolio managers in practice could be incorporated in the approach without much effort by adding penalty terms [13]. Leaving out the constraints in this thesis should consequently not reduce the significance of the research, but rather help to keep a clear structure and prevent over-engineering.

In the following, we introduce the remaining shares to be bought at time t as $\boldsymbol{u}_t = (u_{1t}, \ldots, u_{nt})^\intercal$. This $\boldsymbol{u}_t$ will be part of the state variables $\boldsymbol{s}_t$. So $\boldsymbol{u}_t$ contains the part of the order that is still outstanding at time t, which means that $\boldsymbol{u}_0 = \bar{\boldsymbol{a}}$ and $\boldsymbol{u}_T = 0$, in order to ensure that all $\bar{\boldsymbol{a}}$ shares are purchased within time T. The development of the outstanding order over time obviously depends on the trades executed by the investor and can therefore be viewed as part of the deterministic drift of the system. We can summarize the development of $\boldsymbol{u}_t$ as

$$\begin{aligned} \boldsymbol{u}_0 &= \bar{\boldsymbol{a}} \\ \boldsymbol{u}_{t+1} &= \boldsymbol{u}_t - \boldsymbol{a}_t, \quad t = 0, \ldots, T-2 \\ \boldsymbol{u}_T &= 0. \end{aligned} \tag{19}$$

As presented in equation (14), there are more specifications required to fully describe the evolution of the system of a stochastic control problem, such as the random noise. In our model, this noise is incorporated in the price dynamics, as well as in the evolution of other state variables, which we will call $\boldsymbol{x}_t$. The price dynamics should capture the price impact of trading and the influence on the execution price by changing market conditions or information. Here, we consider a linear percentage price impact model as proposed in [3], which is discussed in the next section.

3.2.2 Linear Percentage Price Impact

The linear percentage price impact model is used to describe the price dynamics of the execution prices $\boldsymbol{p}_t$, which are assumed to be the sum of two components for $t = 0, \ldots, T-1$:

$$\boldsymbol{p}_t = \tilde{\boldsymbol{p}}_t + \boldsymbol{\delta}_t. \tag{20}$$

Here, $\tilde{\boldsymbol{p}}_t$ represents the "no-impact" price and $\boldsymbol{\delta}_t$ stands for the price impact. The no-impact price is the price of the stocks that would prevail on a financial market that is free of any market impacts. This means that $\tilde{\boldsymbol{p}}_t$ is independent of the trade size $\boldsymbol{a}_t$. On the contrary, the price impact refers to the price change of the stocks caused by the investors incoming order and other market impacts [6]. The influence of these effects on the execution price is represented by $\boldsymbol{\delta}_t$.

To ensure non-negative no-impact prices and for simplicity, $\tilde{\boldsymbol{p}}_t$ is modelled as multivariate geometric Brownian motion [3]:

$$\tilde{\boldsymbol{p}}_t = \exp(\boldsymbol{Z}_t)\tilde{\boldsymbol{p}}_{t-1}, \tag{21}$$

where $\boldsymbol{Z}_t = \text{diag}[\boldsymbol{z}_t]$ and $\boldsymbol{z}_t$ is a normal random vector with mean $\boldsymbol{\mu}_z$ and covariance matrix $\boldsymbol{\Sigma}_z$. The operator $\exp(\cdot)$ represents the matrix exponential, which reduces to the element-wise application of the exponential function to the diagonal entries in the case of a diagonal matrix.

The price impact is given by

$$\boldsymbol{\delta}_t = \tilde{\boldsymbol{P}}_t(\boldsymbol{A}\tilde{\boldsymbol{P}}_t\boldsymbol{a}_t + \boldsymbol{B}\boldsymbol{x}_t), \tag{22}$$

where $\tilde{\boldsymbol{P}}_t = \text{diag}[\tilde{\boldsymbol{p}}_t]$, $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is positive definite, $\boldsymbol{B} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{x}_t \in \mathbb{R}^m$. We let $m \in \mathbb{N}$ be the number of sources of information that affect the execution prices of the stocks. The vector that represents this information and the changing market conditions is labelled $\boldsymbol{x}_t$. For instance, $(\boldsymbol{x}_t)_i$, $i \leq m$, might be the return on a specific stock market index at time t, which is a common component in the price of stocks. The dynamics of these other state variables are described below in (23).

Note that we use $\boldsymbol{A}$ to measure the price impact's sensitivity to the current trade size, while $\boldsymbol{B}$ is a measure for the sensitivity to the market conditions and information in $\boldsymbol{x}_t$. The choice of $\boldsymbol{A}$ reveals that we are not only taking into account the price impact of trading in stock i on p_{it}, which is determined by $\boldsymbol{A}_{ii}$, but also cross market impact effects between different stocks if $\boldsymbol{A}$ is non-diagonal. This lets us model instances, where stocks are close substitutes and purchasing a large number of shares in one stock can not only drive the price in this stock up, but also induce similar price movements in other stocks of the portfolio. Or, in case of a negative correlation, the execution price of another stock can be reduced, revealing a diversification effect. Specifically, we observe that $\boldsymbol{A}_{ij}$ measures the sensitivity of the price impact of trading in stock j on p_{it}, while $\boldsymbol{A}_{ji}$ measures the influence of trading in stock i on p_{jt}. This also highlights the relevance of the research of [3], which shows that it is much more accurate to incorporate cross-stock effects compared to minimizing the expected cost of executing trades in each security in isolation. Notice that the functionality for $\boldsymbol{B}$ is similar. As an example, $\boldsymbol{B}_{ik}$ measures the sensitivity of the price impact on p_{it} to the k-th market condition.

The linear percentage price impact model has a couple of advantages that we briefly want to mention. The separation of the execution price into a no-impact price and a price impact component shown in (20) leads to the fact that price impact effects of trades are only temporary, instead of being permanently incorporated into the price. Moreover, the percentage price impact increases linearly with the trade size, which also seems realistic [22].

Lastly, we define the law of motion for the process of the information state variables $\boldsymbol{x}_t$ as

$$\boldsymbol{x}_t = \boldsymbol{C}\boldsymbol{x}_{t-1} + \boldsymbol{\phi}_t, \tag{23}$$

where we let $\boldsymbol{C} \in \mathbb{R}^{m\times m}$ and $\boldsymbol{\phi}_t$ be a vector white noise with mean 0 and covariance matrix $\boldsymbol{\Sigma}_\phi$. Thus, $\boldsymbol{x}_t$ is a VAR(1), a vector autoregressive process with one lag. This means that

the state variables in $\boldsymbol{x}_t$ depend linearly on their own values of the previous period and on the stochastic noise introduced by $\boldsymbol{\phi}_t$. Modelling $\boldsymbol{x}_t$ as a VAR(1) allows the application of varying degrees of predictability in the market conditions or of available information and is fairly easy to implement. To ensure the stationarity of $\boldsymbol{x}_t$, we must have $|{}_C\lambda_i| < 1$ for all eigenvalues ${}_C\lambda_1, \ldots, {}_C\lambda_m$ of $\boldsymbol{C}$ [3].

Altogether, we let the state variables of the stochastic control problem of optimal execution costs for portfolios be

$$\boldsymbol{s}_t = \begin{bmatrix} \tilde{\boldsymbol{p}}_t \\ \boldsymbol{x}_t \\ \boldsymbol{u}_t \end{bmatrix}. \tag{24}$$

3.2.3 Calculation of Costs

To implement a deep learning approximation algorithm for the stochastic control problem in (17), a detailed specification of the composition of costs needs to be included in the problem statement. Therefore, we first define the intermediate execution cost $\boldsymbol{c}_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$ that is associated with trading $\boldsymbol{a}_t$ shares of the stocks in the portfolio at time t in state $\boldsymbol{s}_t$:

$$\boldsymbol{c}_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{p}_t^\intercal \boldsymbol{a}_t = (\tilde{\boldsymbol{p}}_t + \boldsymbol{\delta}_t)^\intercal \boldsymbol{a}_t. \tag{25}$$

The intermediate cost represents the amount an investor actually has to pay for executing the trades $\boldsymbol{a}_t$ at execution prices $\boldsymbol{p}_t$, including market impact effects. Subsequently, the cumulative cost $\boldsymbol{C}_t$ can be defined as

$$\begin{aligned} \boldsymbol{C}_0 &= \boldsymbol{c}_0 \\ \boldsymbol{C}_{t+1} &= \boldsymbol{C}_t + \boldsymbol{c}_{t+1} \quad t = 0, \ldots, T-2. \end{aligned} \tag{26}$$

The total cost $\boldsymbol{C}_{T-1}$ is the cumulative cost after all trades have been executed and the order $\bar{\boldsymbol{a}}$ is fulfilled at time T. Hence, we can restate the stochastic control problem from (17) as

$$\min_{\{\boldsymbol{a}_t\}_{t=0}^{T-1}} \mathbb{E}_P[\boldsymbol{C}_{T-1}] \quad = \min_{\{\boldsymbol{a}_t\}_{t=0}^{T-1}} \mathbb{E}_P[\sum_{t=0}^{T-1} \boldsymbol{c}_t(\boldsymbol{s}_t, \boldsymbol{a}_t)]. \tag{27}$$

Notice the similarity to (16) and that the condition of (18) has to be met as well. An analytical solution to this problem can be found by using dynamic programming. For the analytical optimal cost and the corresponding expression of the optimal trading strategy, see [3]. Later, we use this analytical optimal solution to benchmark the results of our numerical experiments.

Due to the fact that the total execution cost $\boldsymbol{C}_{T-1}$ dependents heavily on the no-impact share prices of the stocks to be purchased and the order size, we introduce another measure for costs that makes the transactions more comparable for different portfolio structures. We define the total execution costs in cents per share above the no-impact cost $\tilde{\boldsymbol{p}}_0^{\mathsf{T}}\bar{\boldsymbol{a}}$ as

$$\mathcal{C}^{\$} = 100\,\frac{\boldsymbol{C}_{T-1} - \sum_{i=1}^{n}\tilde{p}_{i0}\bar{a}_i}{\sum_{i=1}^{n}\bar{a}_i}. \tag{28}$$

It is calculated by comparing the total execution cost with the cost that would arise if the complete order would be executed at $t = 0$ with no market impacts and other transaction costs and distributing this difference over the number of shares traded in total to fulfill the order. Finally, the approach for approximating the solution of this stochastic control problem with a deep neural network is presented in the next section.

3.3 Deep Learning Approximation Approach

As stated before, the challenge that arises is to find the optimal actions in each period in order to minimize the expected total cost. Our approach is to approximate this optimal trading strategy for each time period t using a feed-forward neural network $\texttt{SN}_t$ [13]. As the choice of the action $\boldsymbol{a}_t$ only depends on the current state $\boldsymbol{s}_t$ for each $t = 0, 1, \ldots, T-1$, the networks will take the current state variables as inputs. The aim is to find the functional dependencies between state variables and controls with the networks $\{\texttt{SN}_t\}_{t=0}^{T-2}$ by optimizing their parameters $\boldsymbol{\theta}_t$ such that

$$\boldsymbol{a}_t(\boldsymbol{s}_t) \approx \boldsymbol{a}_t(\boldsymbol{s}_t|\boldsymbol{\theta}_t).$$

Note that we only have $T-1$ subnetworks $\texttt{SN}_t$, for each $t = 0, \ldots, T-2$. This results from the fact that in the last period, between time $T-1$ and T, the trades that are executed are predetermined by the remaining order: $\boldsymbol{a}_{T-1} = \boldsymbol{u}_{T-1}$. Consequently, there is no need for a neural network to approximate the control variable at $t = T-1$. Hence, the optimization problem we want to solve becomes

$$\min_{\{\boldsymbol{\theta}_t\}_{t=0}^{T-2}} \mathbb{E}_P[\sum_{t=0}^{T-2}\boldsymbol{c}_t(\boldsymbol{s}_t, \boldsymbol{a}_t(\boldsymbol{s}_t|\boldsymbol{\theta}_t)) + \boldsymbol{c}_{T-1}(\boldsymbol{s}_{T-1}, \boldsymbol{u}_{T-1})]. \tag{29}$$

To create a deep neural network that outputs the total execution costs, we stack these subnetworks $\texttt{SN}_t$ together through the specified model dynamics (see Figure 4). This overall network is labelled $\texttt{DNN}$. The subnetworks can then be trained simultaneously by feeding the deep network with an initial state and samples of the stochastic process $\{\boldsymbol{\xi}_t\}_{t=1}^{T-1}$ that includes all noise incorporated in the evolution of $\tilde{\boldsymbol{p}}_t$ and $\boldsymbol{x}_t$. During this process, the $\texttt{DNN}$ learns to trade in a way that the total execution costs are minimized.

The exact architecture of the $\texttt{DNN}$ is presented in Figure 4. The state variables $\boldsymbol{s}_t$ are visualized as blue circles and fed into the first hidden layer of the appropriate subnetwork.

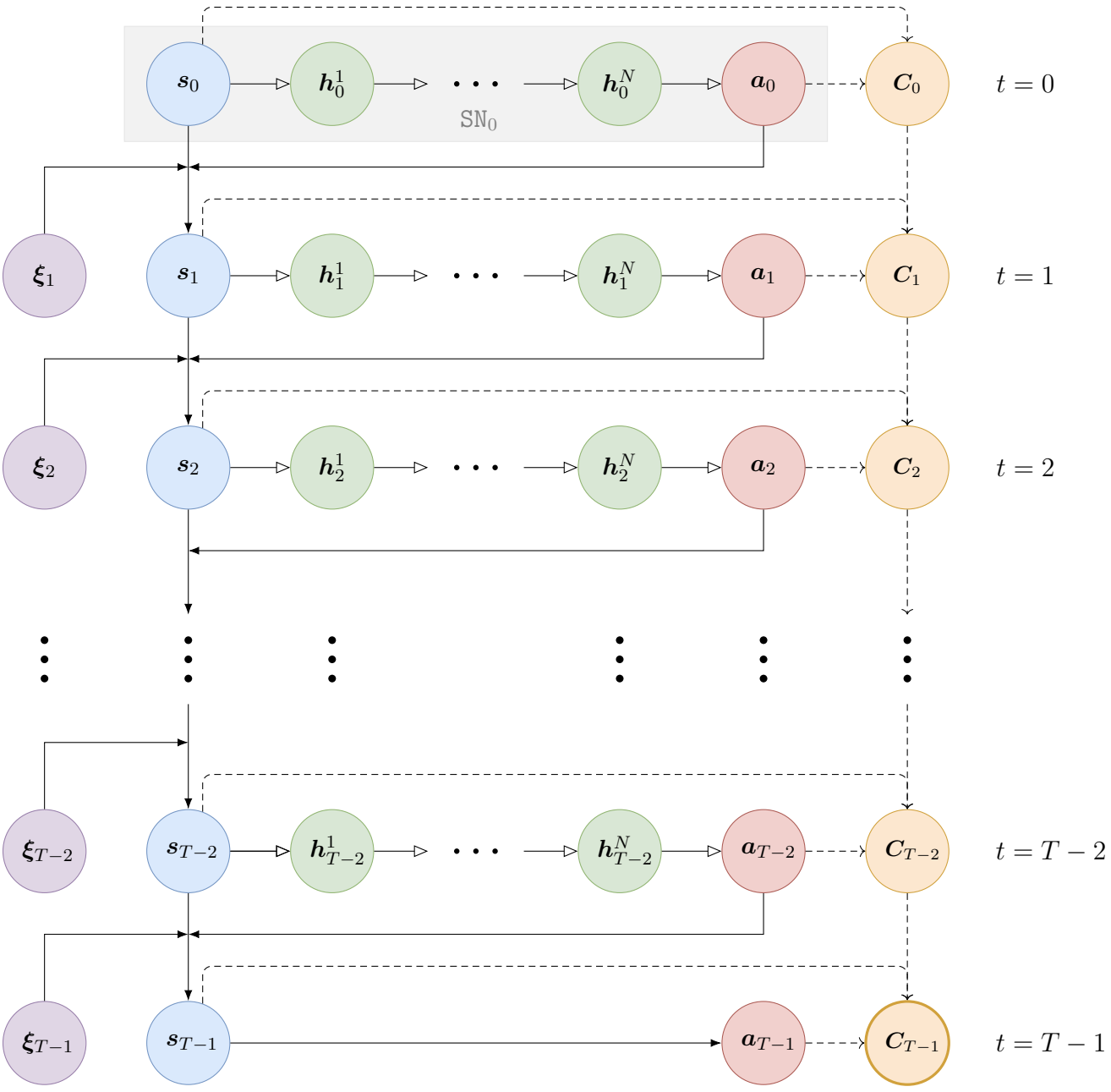


Figure 4: Illustration of the architecture of the deep neural network `DNN` for the stochastic control problem of optimizing execution costs for portfolios

The N hidden layers of the subnetwork $\mathtt{SN}_t$ are called $\{\boldsymbol{h}_t^l\}_{l=1}^N$ and are displayed in green. Next, the controls $\boldsymbol{a}_t$ at each time are approximated by the subnetworks and shown in red. All cumulative costs are presented in yellow, while the stochastic noise that enters the model and influences the evolution of states is denoted by $\boldsymbol{\xi}_t$ in purple.

Notice that there are three types of arrows in this network, representing different types of connections:

1. $\boldsymbol{s}_t \longrightarrow \boldsymbol{h}_t^1 \longrightarrow \ldots \longrightarrow \boldsymbol{h}_t^N \longrightarrow \boldsymbol{a}_t$ is the multilayer feed-forward subnetwork $\texttt{SN}_t$ that approximates the control $\boldsymbol{a}_t$ at time t, for $t = 0, \ldots, T-2$. The first subnetwork $\texttt{SN}_0$ is indicated in gray as an example. The inputs to the t-th subnetwork are given by the state variables $\boldsymbol{s}_t$. Note that these subnetworks contain the parameters $\boldsymbol{\theta}_t$ that we want to optimize. We highlight again that there is no subnetwork for the last period, as the remaining part of the order $\boldsymbol{u}_{T-1}$ will be executed at time $T-1$.

2. $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{\xi}_{t+1}) \longrightarrow \boldsymbol{s}_{t+1}$ describes the model dynamics. The evolution of the states is fully characterized by these connections. There are no parameters to optimize here, as the state equations are fixed. Recall that $\boldsymbol{s}_t$ consists of the variables $\tilde{\boldsymbol{p}}_t$, $\boldsymbol{x}_t$ and $\boldsymbol{u}_t$ (see Equation (24)). The evolution of all of these variables depends on their predecessors in time, making it obvious that $\boldsymbol{s}_{t+1}$ depends on $\boldsymbol{s}_t$. Furthermore, the trade decision $\boldsymbol{a}_t$ affects $\boldsymbol{s}_{t+1}$ by determining changes in the outstanding order $\boldsymbol{u}_{t+1}$ as described in (19). The random variable $\boldsymbol{\xi}_{t+1}$, which contains all noisy information that arrives between time t and $t+1$, influences $\tilde{\boldsymbol{p}}_{t+1}$ and $\boldsymbol{x}_{t+1}$. Thus, $\boldsymbol{\xi}_{t+1}$ must contain the white noise $\boldsymbol{\phi}_{t+1}$ of the autoregressive process for the market conditions $\boldsymbol{x}_t$, as detailed in (23), and must also include the information of the random normal vector $\boldsymbol{z}_{t+1}$ that models the no-impact price dynamics, which we outlined in Equation (21).

3. $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{C}_{t-1}) \dashrightarrow \boldsymbol{C}_t$ is the contribution to the total execution cost, which is the final output of the deep neural network $\texttt{DNN}$. For $t = 0$ we only have $(\boldsymbol{s}_0, \boldsymbol{a}_0) \dashrightarrow \boldsymbol{C}_0$, as there are no pre-existing costs. The composition of costs is described in detail in Section 3.2.3.

The scope of the $\texttt{DNN}$ in Figure 4 depends mainly on the choice of N, the number of hidden layers per subnetwork, and the time horizon T. Overall, the network has, including the input and output layers, $(T-1)(N+2)$ layers in total. Choosing for example two hidden layers in a model with time horizon $T = 25$, there will be 96 layers in total. To succeed with this deep learning approximation approach, the connections listed above must be implemented correctly, meaning the creation of the subnetworks and their careful linkage through model dynamics. The programming realization of the execution cost model and the implementation of the $\texttt{DNN}$ is discussed in the next section.

4 Implementation

This section gives an overview of how the execution cost model for portfolios is set up in python and how the network architecture is implemented using TensorFlow. The deep learning approach to solve the optimization problem of execution costs is realized on a MacBook Pro 2018 with a 2.3 GHz Intel Core i5 processing unit. The program for the approximation algorithm is written in python 3.7 on a Jupyter Notebook interface. Furthermore, the open-source software library TensorFlow, which was developed by the Google Brain team [11], is used to train the model. For this training, meaning the optimization of the weights in the neural network, no GPU acceleration is used.

After a general introduction to TensorFlow, the four main parts of the written code are presented. First, the execution cost model is built, including the initialization of the model parameters and the implementation of the model dynamics of Section 3.2. It follows a short part with configurations for the training, where the subnetwork architecture is defined and hyperparameters, such as the learning rate, are specified. In the third part, the overall architecture of the `DNN` is converted to a TensorFlow graph. This graph construction represents the crucial part for implementing a deep learning algorithm and requires the most work. The last part of the code runs a TensorFlow session that uses the predefined graph to perform training and minimize loss. The full code can be seen in Appendix A.

4.1 Machine Learning with TensorFlow

The primary interface of TensorFlow is python. However, the core functionality of the software is programmed in C++, ensuring a high performance. TensorFlow stores operations in a computation graph, which it can then run on a GPU or an external system. This allows fast calculations and short training times of machine learning models. The TensorFlow API is available at www.tensorflow.org.

TensorFlow utilizes dataflow graphs to perform the user's computations (see Figure 5). The graphs are composed of a set of nodes that represent operations. Each node can have an arbitrary number of in- and outputs. Dataflow and dependencies of the nodes are described by connecting edges [1]. The user's program interacts with the TensorFlow system by creating a TensorFlow session. A session is required to feed a graph with values for its computations and evaluate certain nodes. Consequently, working with TensorFlow is usually split in two phases. First, the graph construction phase, where the graph is defined. Second, an execution phase, where a session is created to run parts of the graph on local or remote devices. A major advantage of this dataflow model and the split of work is that it allows parallel computing. This can significantly speed up the calculations and increase the efficiency of the program.

In the case of building an ANN, the network architecture is converted to a TensorFlow graph. Every neuron of each layer, as well as the application of the activation function and the loss have to be considered. The training is then performed during a TensorFlow session by feeding the graph with training data, repeatedly evaluating the node that represents the loss and adjusting the weights according to an optimization algorithm.

4.1.1 Example of a TensorFlow Graph

An example of a TensorFlow graph is shown in Figure 5. In our example, the graph represents the calculation of the function

$$L(W, b, x) = ReLU(W \cdot x + b) \cdot 5.$$

Notice that there are different types of nodes in the graph. The node in gray represents a TensorFlow constant (*tf.constant*) with a fixed value. The green nodes are TensorFlow variables (*tf.Variable*). Later on, the parameters of the ANN, which means the weights and biases or the weights and the shifting and scaling parameters for batch normalization, will be of this type of nodes. This means that the *tf.Variables* hold the parameters that will be adjusted during training and therefore targeted by the descent algorithm used. Hence, we ultimately want to calculate the partial derivatives of the loss function with respect to these nodes. In our example, one can interpret the function above as a cost function that calculates the loss L based on the values of W, b and x (similar to (10)) by applying some basic operations. The operations performed are represented by the orange nodes in the graph.

Finally, the node in blue labelled x is a placeholder node (*tf.placeholder*). This type of nodes is usually used for the training data. In a computation graph, they act similar to a *tf.constant*, but can be assigned with different values for different runs of the graph. For example, during the iterations of the optimization algorithm, these nodes are fed with different data from the training batches.

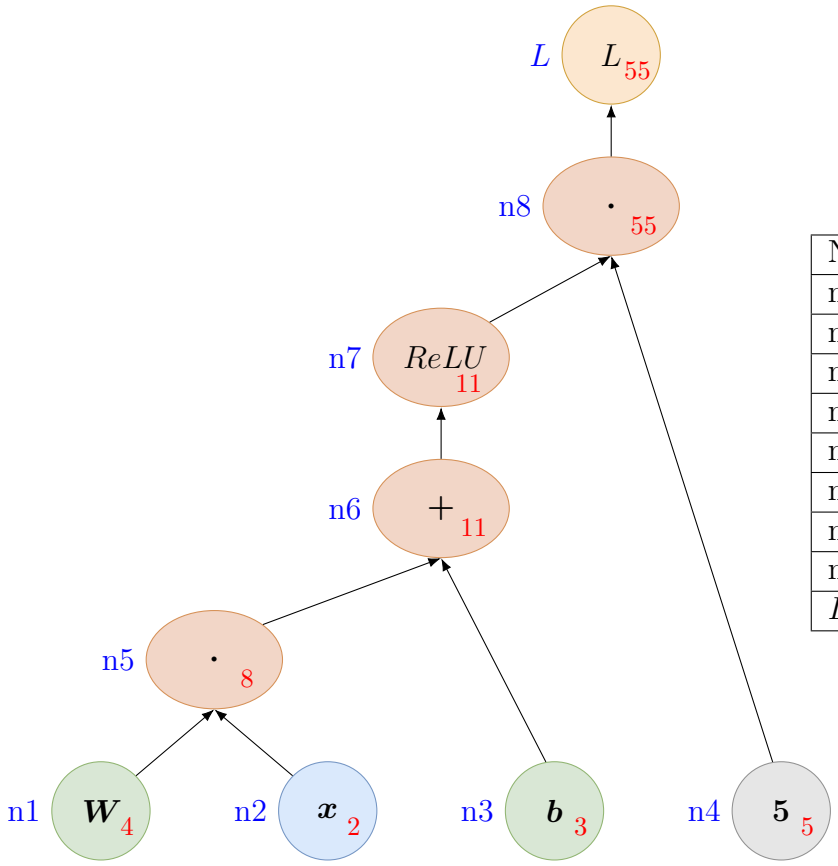


Figure 5: Example of a TensorFlow graph

Table 1: Values of nodes

Node	Expression	Value
n_1	W	4
n_2	x	2
n_3	b	3
n_4	5	5
n_5	$n_1 \cdot n_2$	8
n_6	$n_5 + n_3$	11
n_7	$max(0, n_6)$	11
n_8	$n_7 \cdot n_4$	55
L	n_8	55

Operation nodes can be evaluated in a TensorFlow session, after the initialization of the input nodes (*tf.Variables* and *tf.placeholders*). For example, if we initialize the graph with the values $W = 4$, $b = 3$ and feed the placeholder node x with the value 2, we receive a value of $L(4, 3, 2) = ReLU(4 \cdot 2 + 3) \cdot 5 = 11 \cdot 5 = 55$ for the output node L. The value of each individual node is shown at the bottom right in red and can also be found in Table 1. We label the nodes n_1 to n_8 for clarity.

Understanding how a TensorFlow computation graph works is essential for a successful implementation of a deep learning algorithm, because the architecture of the ANN must be converted to such a graph. If this is done correctly, the training of the network becomes comparably simple and can exploit all advantages of the software. For instance, the computation of the gradients of the loss function with respect to the network's parameters is done automatically by TensorFlow, which uses an automatic differentiation technique. The idea behind this method is explained in the next section by using the example graph from above.

4.1.2 Reverse Mode Automatic Differentiation

TensorFlow uses reverse mode automatic differentiation to compute gradients automatically. This method is especially efficient and accurate when there are many inputs and few outputs, which is usually the case for ANNs [12]. Its goal is to compute all partial derivatives of the output of a graph (the loss) with respect to relevant nodes (the weights) by relying on the chain rule.

Coming back to the example in Section 4.1.1, we can manually compute the partial derivatives of interest as

$$\frac{\partial L}{\partial W} = \frac{\partial (ReLU(W \cdot x + b) \cdot 5)}{\partial W} = x \cdot 5 \cdot \begin{cases} 0 & \text{if } W \cdot x + b < 0 \\ 1 & \text{if } W \cdot x + b > 0 \end{cases}$$

and

$$\frac{\partial L}{\partial b} = \frac{\partial (ReLU(W \cdot x + b) \cdot 5)}{\partial b} = 5 \cdot \begin{cases} 0 & \text{if } W \cdot x + b < 0 \\ 1 & \text{if } W \cdot x + b > 0. \end{cases}$$

Now, we look at how TensorFlow computes these partial derivatives automatically for given values of the input nodes. In a first traversal of the graph, the values of each node are computed in a forward direction, such as indicated in red in Figure 5. In a second pass, all partial derivatives of L with respect to the *tf.Variable* nodes in green are computed by going backwards through the graph, starting with the output L.

If we have any node n_i in the graph and the reverse path from L to n_i goes through the intermediate node n_p, we call this node a parent node of n_i. For example, if $L = f(\mathrm{n}_p)$ and $\mathrm{n}_p = g(\mathrm{n}_i)$, with f and g being simple functions, n_p is a parent of n_i. Using the multivariate chain rule, we can derive the following formula for finding the partial derivatives:

$$\frac{\partial L}{\partial \mathrm{n}_i} = \sum_{p \in \pi(\mathrm{n}_i)} \frac{\partial L}{\partial \mathrm{n}_p} \frac{\partial \mathrm{n}_p}{\partial \mathrm{n}_i}, \tag{30}$$

where $\pi(\mathrm{n}_i)$ contains the indices of all direct parent nodes of n_i, meaning all parent nodes that are directly connected to n_i. Hence, for finding the partial derivative of L with respect to any intermediate or input node, we only need the derivatives of its parents and the formula to calculate the derivative of the simple function $\mathrm{n}_p = g(\mathrm{n}_i)$. The standard derivatives for such basic operations (addition, subtraction, multiplication, division, exponential function, matrix operations etc.) are known by TensorFlow [1] or can be added manually by the user if needed.

In our example, we want to calculate $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. Consequently, we want to calculate $\frac{\partial L}{\partial \mathrm{n}_1}$ and $\frac{\partial L}{\partial \mathrm{n}_3}$ for the *tf.Variable* nodes. We begin with $\frac{\partial L}{\partial \mathrm{n}_1}$:

The reverse pass starts at the output node L. We trivially have that $\frac{\partial L}{\partial L} = 1$. Since $\mathrm{n}_8 = L$ we also have

$$\frac{\partial L}{\partial \mathrm{n}_8} = 1.$$

We continue down the graph towards n_1 to n_7. We know by (30) that

$$\frac{\partial L}{\partial \mathrm{n}_7} = \underbrace{\frac{\partial L}{\partial \mathrm{n}_8}}_{=1} \frac{\partial \mathrm{n}_8}{\partial \mathrm{n}_7} = \frac{\partial \mathrm{n}_8}{\partial \mathrm{n}_7}.$$

In Table 1, we see the expression $\mathrm{n}_8 = \mathrm{n}_7 \cdot \mathrm{n}_4$. Therefore, we know that the partial derivative is

$$\frac{\partial \mathrm{n}_8}{\partial \mathrm{n}_7} = \mathrm{n}_4,$$

so $\frac{\partial L}{\partial \mathrm{n}_7} = \mathrm{n}_4$. Note that n_4 has the value 5 (see Figure 5).

Similarly, we can proceed to n_6 with

$$\frac{\partial L}{\partial \mathrm{n}_6} = \underbrace{\frac{\partial L}{\partial \mathrm{n}_7}}_{=\mathrm{n}_4} \frac{\partial \mathrm{n}_7}{\partial \mathrm{n}_6}.$$

The partial derivative $\frac{\partial \mathrm{n}_7}{\partial \mathrm{n}_6}$ is

$$\frac{\partial \mathrm{n}_7}{\partial \mathrm{n}_6} = \frac{\partial ReLU(\mathrm{n}_6)}{\partial \mathrm{n}_6} = \begin{cases} 0 & \text{if } \mathrm{n}_6 < 0 \\ 1 & \text{if } \mathrm{n}_6 > 0, \end{cases}$$

which we know from Section 2.2.3. Due to the fact that the value of n_6 is $11 > 0$ and $\mathrm{n}_4 = 5$, the value of $\frac{\partial L}{\partial \mathrm{n}_6}$ is also 5.

The process continues until we reach the *tf.Variable* input node n_1. We have

$$\frac{\partial L}{\partial \mathrm{n}_5} = \frac{\partial L}{\partial \mathrm{n}_6} \frac{\partial \mathrm{n}_6}{\partial \mathrm{n}_5} \quad \text{and} \quad \frac{\partial L}{\partial \mathrm{n}_1} = \frac{\partial L}{\partial \mathrm{n}_5} \frac{\partial \mathrm{n}_5}{\partial \mathrm{n}_1},$$

because $\pi(\mathrm{n}_5) = \{6\}$ and $\pi(\mathrm{n}_1) = \{5\}$. Furthermore, we can compute $\frac{\partial \mathrm{n}_6}{\partial \mathrm{n}_5} = 1$ and $\frac{\partial \mathrm{n}_5}{\partial \mathrm{n}_1} = \mathrm{n}_2 = 2$.

Finally, we can calculate the value of the partial derivative of L with respect to the input W by substituting what we calculated and multiplying the partial derivatives:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \mathrm{n}_1} = \frac{\partial L}{\partial \mathrm{n}_8} \cdot \frac{\partial \mathrm{n}_8}{\partial \mathrm{n}_7} \cdot \frac{\partial \mathrm{n}_7}{\partial \mathrm{n}_6} \cdot \frac{\partial \mathrm{n}_6}{\partial \mathrm{n}_5} \cdot \frac{\partial \mathrm{n}_5}{\partial \mathrm{n}_1} = 1 \cdot \mathrm{n}_4 \cdot \begin{cases} 0 & \text{if } \mathrm{n}_6 < 0 \\ 1 & \text{if } \mathrm{n}_6 > 0 \end{cases} \cdot 1 \cdot \mathrm{n}_2 = 5 \cdot 2 = 10.$$

Now, $\frac{\partial L}{\partial b}$ is calculated the same way, by going down the edges in the graph from L to n_3:

$$\frac{\partial L}{\partial \mathrm{n}_3} = \frac{\partial L}{\partial \mathrm{n}_8} \cdot \frac{\partial \mathrm{n}_8}{\partial \mathrm{n}_7} \cdot \frac{\partial \mathrm{n}_7}{\partial \mathrm{n}_6} \cdot \frac{\partial \mathrm{n}_6}{\partial \mathrm{n}_3} = 1 \cdot \mathrm{n}_4 \cdot \begin{cases} 0 & \text{if } \mathrm{n}_6 < 0 \\ 1 & \text{if } \mathrm{n}_6 > 0 \end{cases} \cdot 1 = 5.$$

Notice that this aligns with the solution found by manual differentiation when inserting the same values for W, b and x. This technique allows TensorFlow to automatically calculate the gradients needed for the gradient descent optimization and is very efficient, because it only requires one forward pass and one backward pass to compute the partial derivatives of the output with respect to all inputs of a neural network. Additionally, the application of the chain rule illustrates again the risk of running into the vanishing gradient problem of Section 2.2.3. If the values of the derivatives of the activation function in an ANN are small, the gradient can diminish.

4.2 Setting up the Execution Cost Model

In this first part of the code, the parameters of the execution cost model that cannot be influenced by the investor and that are necessary for calculating the total execution costs are initialized, such as the initial no-impact prices of the stocks in the portfolio and the matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ (see Section 3.2). In addition, it includes the calculation of an analytical optimal solution to the execution cost model given the above mentioned parameters. This is done according to the dynamic programming algorithm presented by [3]. The analytical optimal solution is important to evaluate the performance of the neural network. However, it is not relevant for training. Finally, the sampling of training data is specified by implementing the model dynamics.

4.2.1 Initialization of Model Parameters

The parameters of the model are assigned with realistic values. However, we do not use real world data, but create the values and the training data artificially, based on the empirical example in [3] as a benchmark. Consequently, some assumptions regarding the underlying distributions of the parameters have to be made. Parameters that can be influenced by the investor, such as the number of stocks in the portfolio n or the order that has to be executed $\bar{\boldsymbol{a}}$, are defined deterministically in the configurations part (see Section 4.3).

We make the following assumption about the matrix $\boldsymbol{A}$ in the execution cost model, which measures the price impact's sensitivity to the current trade size:

Assumption 1 (Elements of $\boldsymbol{A}$)**.** *Let $\Lambda > 0$ be a constant. We assume the elements of $\boldsymbol{A}$ to be distributed uniformly according to*

$$\boldsymbol{A}_{ij} \sim \mathcal{U}(-\Lambda, \Lambda), \quad i, j = 1, \ldots, n,\ i \neq j,$$

$$\boldsymbol{A}_{ii} \sim \mathcal{U}(4\Lambda, 6\Lambda), \quad i = 1, \ldots, n.$$

Making the size of the elements of $\boldsymbol{A}$ dependent on the factor Λ allows the testing of varying degrees of influence of the trade size on the price impact. In the numerical experiments, we find out that this factor plays an important role for the overall range of execution costs. Moreover, note that the price impact from trading on the same stock is assumed to be significantly higher than the cross-stock price impact. The cross-stock price impact can also be negative.

Looking at the network architecture in Figure 4, we require an initial state $\boldsymbol{s}_0$. Hence, the initial no-impact prices $\tilde{\boldsymbol{p}}_0$ of the n stocks and the initial market conditions $\boldsymbol{x}_0$, which are part of the initial state, have to be defined. The following assumptions are made:

Assumption 2 (Initial No-Impact Prices)**.** *The initial prices $\tilde{\boldsymbol{p}}_0$ of the n stocks are assumed to be distributed uniformly between 1 and 100:*

$$\tilde{p}_{i0} \sim \mathcal{U}(1, 100), \quad i = 1, \ldots, n.$$

The starting point for the autoregressive process $\boldsymbol{x}_t$ is assumed to be distributed as follows:

Assumption 3 (Initial Market Conditions)**.** *The elements of the initial vector for the market conditions $\boldsymbol{x}_0$ are drawn randomly from a normal distribution with mean 0 and variance 0.5.*

$$x_{i0} \sim \mathcal{N}(0, 0.5), \quad i = 1, \ldots, m.$$

For the initialization of the other parameters, such as the matrices $\boldsymbol{B}$ and $\boldsymbol{C}$, as well as the vector $\boldsymbol{\mu}_z$, we refer to Appendix B.

4.2.2 Sampling of Training Data

To train the model, we also need samples of the process $\{\boldsymbol{\xi}_t\}_{t=1}^{T-1}$ as input to the `DNN`, besides the initial state $\boldsymbol{s}_0$ (see again Figure 4). To be more precise, we need python functions that create samples of the processes $\{\boldsymbol{x}_t\}_{t=0}^{T-1}$ and $\{\tilde{\boldsymbol{p}}_t\}_{t=0}^{T-1}$ when given a start value. The modelling of the evolution of these state variables is done according to equations (21) and (23). The exact code for the two python functions can be found in Appendix A.2.

In Figure 6, we show three samples of the no-impact price development of one stock with fixed initial no-impact price $\tilde{p}_{i0} = 50$. For this visualization, the parameters $\boldsymbol{\mu}_z$ and $\boldsymbol{\Sigma}_z$ are also chosen manually. A time horizon of $T = 25$ and $T = 500$ is selected.

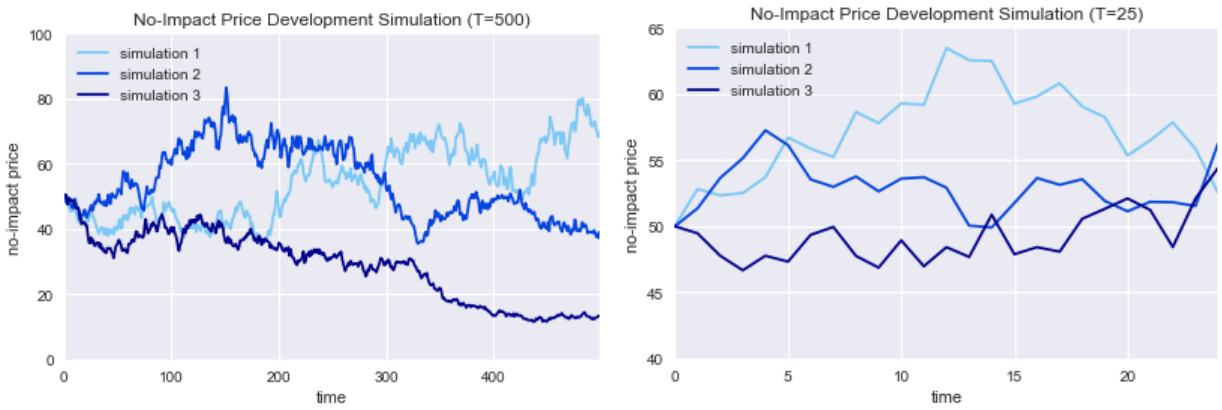


Figure 6: Simulations of the no-impact price development of one stock with initial price 50. The values $\boldsymbol{\mu}_z = 10^{-6}$ and $\boldsymbol{\Sigma}_z = 0.001$ are fixed deterministically.

The graphs in Figure 6 show the simulation of the no-impact price development created by a python function. Similarly, a function that creates samples of the vector autoregressive process of $\{\boldsymbol{x}_t\}_{t=0}^{T-1}$ is created. During training, these two functions are used to create the training batches that are fed to the `DNN`.

4.3 Training Configurations

The configurations part exists in order to let the user change fundamental parameters of the execution cost model or the training process quickly (see Appendix A.3). Examples for these parameters are the number of stocks in the portfolio n, the order size $\bar{\boldsymbol{a}}$, the number of market conditions m and the time horizon T. Furthermore, the network architecture for the subnetworks $\{\mathtt{SN}_t\}_{t=0}^{T-2}$ is decided here. Lastly, this part includes some training specifications, such as the batch size and the learning rate.

4.3.1 Architecture of Subnetworks

Regarding the architecture of the subnetworks, the number of hidden layers N and the number of neurons per hidden layer can be chosen. An example of such a subnetwork $\mathtt{SN}_t$ is shown in Figure 7. For this illustration, we choose $N = 2$ hidden layers with 100 neurons per layer. This is the same number of layers and neurons that we select for the final numerical experiments. The state variabes $\boldsymbol{s}_t \in \mathbb{R}^d$ with $d = 2n + m$ represent the inputs (see (24)) and the controls $\boldsymbol{a}_t$ are the outputs of the subnetwork.

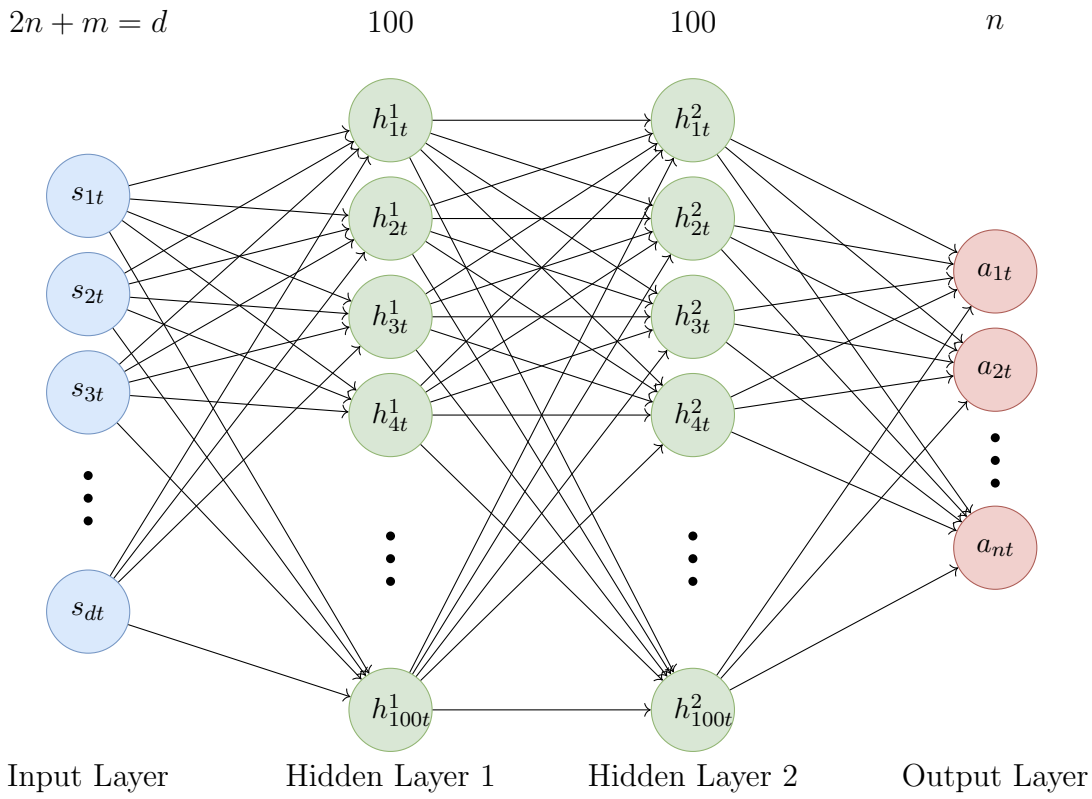


Figure 7: Architecture of a fully connected feed-forward subnetwork $\texttt{SN}_t$ that approximates the controls at time t

Choosing for instance $n = 10$ and $m = 3$, each subnetwork has $13.300 = (2n + m)100 + 100^2 + 100n$ weights. When n is 100, we even get 40.300 weights. This shows again the high-dimensionality of the parameter space of the loss optimization problem in (12), which we want to solve during training.

Due to the fact that we use batch normalization to speed up training and reduce the vanishing gradient problem described in Section 2.2.3, we do not include a bias term in the architecture of the subnetworks. Batch normalization is adopted for each subnetwork and is performed just before applying the activation function (see Figure 8). The activation function applied at all hidden layers of the subnetworks is the *ReLU* function (see (9)). This choice is made based on the findings of Section 2.1.3.

4.3.2 Other Training Specifications

The last configurations needed for the training are, for instance, the size of the training batches K and the size of the validation sample. In addition, the maximum number of iteration steps for the optimization algorithm is defined here, as well as a learning rate or a learning rate schedule (see Section 2.3.2).

We decide to use the Adam optimization method described in Algorithm 4 for training. Due to the fact that Adam is an adaptive learning rate optimization algorithm, we assume that it is sufficient to use a simple implementation of learning rate scheduling that requires less fine tuning. Hence, we apply a predetermined piecewise constant learning rate to accelerate learning and at the same time converge to a good solution. In this scheduling method, the learning rates and its reductions are fixed before training. Although this might work very well, the timing and the decrease rate have to be analyzed carefully. The outcome of the fine tuning of the learning rate for our model can be found in 5.3 in the results section. In addition to the learning rate scheduling, we double the batch size after each learning rate change, choosing the initial batch size to be $K = 64$ and the size of the validation sample to be 512.

All of the above configurations are contained in an object of a python class that can be passed on to other python functions. For example, the function that creates samples of the process $\{\tilde{\boldsymbol{p}}_t\}_{t=0}^{T-1}$ is fed with the batch size K, as well as n and T. It requires this information to produce the right number of sample paths of length T for n stocks. Moreover, the configurations are needed for the construction of the TensorFlow graph, which is described in the next part.

4.4 TensorFlow Graph Construction

In this part of the code, the architecture of the deep neural network presented in Figure 4 is implemented as a TensorFlow graph. In the beginning, all fixed parameters of the execution cost model are converted to TensorFlow constants, so that they can be called in the graph when needed. The same applies for the parameters defined in the configuration section 4.3.

To keep the construction of this TensorFlow graph clear, several python functions are defined that can be interpreted as operation nodes in the overall graph. However, these functions apply multiple basic operations to their inputs in order to arrive at their outputs, which means that they can be represented as TensorFlow graphs themselves. As an example, batch normalization is defined in a function according to Algorithm 3. It takes as inputs the incoming signals of a training batch at a specific layer in a given subnetwork $\texttt{SN}_t$, the respective trainable shifting and scaling parameters, β_t^l and γ_t^l, as *tf.Variables* and a smoothing term as a *tf.constant*. The function's outputs are the transformed signals. In the overall graph, we can then call this function as one operation node, without having to specify the subgraph for the complete batch normalization algorithm again.

We know that our DNN consists of subnetworks, which in turn consist of $N + 2$ layers. Hence, we implement the overall network by defining three main python functions:

1. A python function $next\_layer()$ that passes on signals from one layer to the next layer in any of the subnetworks. This is very similar to (5) combined with the application of batch normalization.

2. A python function $Subnetwork\_t()$ that performs the signal pass-through for a whole subnetwork, by repeatedly calling the function $next\_layer()$.

3. A python function $DNN\_cost()$ that calculates the total cost $\boldsymbol{C}_{T-1}$ and $\mathcal{C}^{\$}$ of executing the order $\bar{\boldsymbol{a}}$ in time T, by trading $\{\boldsymbol{a}_t\}_{t=0}^{T-1}$. It calculates the trades $\boldsymbol{a}_t$ by repeatedly calling the function $Subnetwork\_t()$ and adding up the intermediate costs, such as illustrated in Figure 4.

The definition of these functions in the code can be found in the graph construction part of Appendix A.4. Note that each of these functions can handle a full training batch in the code simultaneously. However, for an easier explanation of their functionality, we use one training example only and set the number of hidden layers as $N = 2$.

The $next\_layer()$ function takes as input the outputs from the previous layer, calculates the weighted sum of these signals and transforms them via batch normalization. Afterwards, it produces the outputs of the layer by applying the $ReLU$ activation function if it is not the final layer in a subnetwork. The code for this python function and the corresponding created TensorFlow graph is presented in the following figure:

```python
def next_layer(in_, trainable_params, layer=None):
    weighted_sum = tf.matmul(in_, trainable_params['w'])
    BN = batch_normalization(weighted_sum, trainable_params['beta'], trainable_params['gamma'])
    if layer == 'final':
        return BN
    else:
        out_ = tf.nn.relu(BN)
        return out_
```

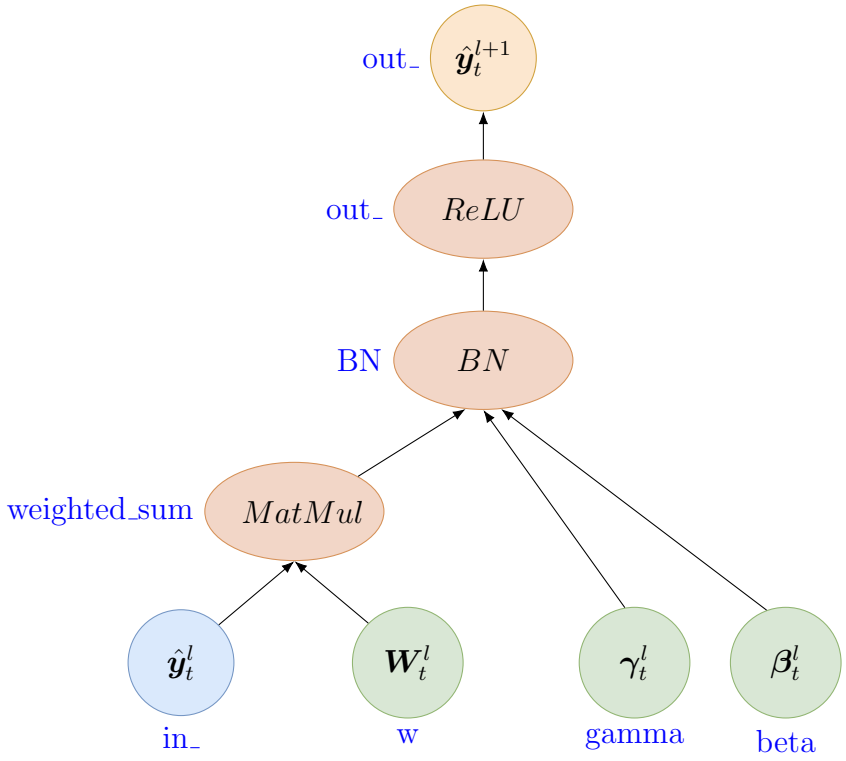


Figure 8: Code and visualization of the TensorFlow graph for the *next_layer*() function applied at a hidden layer in an arbitrary subnetwork

Notice that the input *trainable params* to the *next_layer*() function is a python dictionary containing the trainable parameters for this layer and subnetwork, which are the weights, as well as the batch normalization parameters (see *tf.Variable* nodes in green in Figure 8). The index t and l show that the parameters belong to the l-th layer of subnetwork t. Moreover, as mentioned before, *batch_normalization*() is a separately defined function that is just represented as one operation node in this illustration of the corresponding TensorFlow graph. The full code with the *batch_normalization*() function can be found in Appendix A.4.

The python function $Subnetwork\_t()$ performs the feed-forward pass for one subnetwork $\texttt{SN}_t$. It is basically the implementation of the architecture in Figure 7. It takes as inputs the state variables $\boldsymbol{s}_t$ and applies the $next\_layer()$ function $N+1$ times, until the outputs $\boldsymbol{a}_t$ are produced. In addition, it outputs the intermediate cost $\boldsymbol{c}_t$ for trading $\boldsymbol{a}_t$ of the stocks in the portfolio. The intermediate costs are calculated according to (25), where we again call a separately defined python function. The code for the $Subnetwork\_t()$ function can be found below.

```
def Subnetwork_t(tfg, ecm_tf, t, u_t, p_t_tilde, x_t, trainable_params):
        if t>2:
            u_t = batch_normalization(u_t, beta=None, gamma=None)
        s_t = tf.concat([u_t, p_t_tilde, x_t],1)
        hidden_1 = next_layer(s_t, trainable_params[0])
        hidden_2 = next_layer(hidden_1, trainable_params[1])
        a_t = next_layer(hidden_2, trainable_params[2], layer='final')
        c_t = intermediate_cost(ecm_tf, p_t_tilde, x_t, a_t)
        return c_t, a_t
```

Finally, we define the function $DNN\_cost()$ that represents the complete TensorFlow graph for the calculation of $\mathcal{C}^\$$ (see (28)), the total execution costs above the no-impact cost. Hence, the function takes many different inputs. For example, it takes as *tf.constants* all parameters of the execution cost model, such as $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$. Furthermore, it is fed with a collection of all weight matrices $\boldsymbol{W} = \{W_0^1, W_0^2, W_0^3, W_1^1, \ldots\ldots, W_{T-2}^1, W_{T-2}^2, W_{T-2}^3\}$ and all scaling and shifting parameters as *tf.Variables*. These are the parameters of the DNN that we want to train and we use TensorFlow and automatic differentiation to calculate the gradient of the output $\mathcal{C}^\$$, which is our loss, with respect to all of these trainable parameters.

In addition, the graph has *tf.placeholder* nodes, which we can use to feed it with batches of sample simulations of the processes $\{\boldsymbol{x}_t\}_{t=0}^{T-1}$ and $\{\tilde{\boldsymbol{p}}_t\}_{t=0}^{T-1}$ as described in Section 4.2.2. Then, the $DNN\_cost()$ function calculates the controls $\boldsymbol{a}_t$ and the intermediate costs $\boldsymbol{c}_t$ for each time step by repeatedly applying the $Subnetwork\_t()$ function described above. It also keeps track of the remaining order $\boldsymbol{u}_t$ according to (19) and transfers the value to the next subnetwork as part of the state variables. Lastly, it adds up all intermediate costs to arrive at the total cumulative cost $\boldsymbol{C}_{T-1}$ and calculate the loss $\mathcal{C}^\$$.

In the end of the graph construction phase, $\mathcal{C}^\$$ is the output of the TensorFlow graph created by the $DNN\_cost()$ function and serves as the loss function that will be targeted by the Adam optimization method during training, which is discussed in the next section.

4.5 Creating a TensorFlow Session

In the final part of the code, a TensorFlow session is created, where the parameters of the `DNN` are initialized and the training is conducted by using an optimization algorithm that adjusts the weights and other trainable parameters based on the data of sampled training batches.

4.5.1 Initialization of Trainable Parameters

The number of weight matrices required for the overall network depends on the number of hidden layers in each subnetwork N and the time horizon T. As we choose $N = 2$, we need $2 + 1$ weight matrices per subnetwork. In total, we have $3(T - 1)$ weight matrices, as well as one scaling and one shifting parameter per weight matrix. The dimensions of the weight matrices depend on the number of stocks in the portfolio n, the sources of market information m and the number of neurons per hidden layer. The information of these specifications is received from the configuration part (see Section 4.3). With the choice of $N = 2$ and 100 neurons per hidden layer, we conclude that for each $t = 0, \dots, T - 2$ the matrix W_t^1 is a $100 \times d$ matrix, $W_t^2 \in \mathbb{R}^{100\times100}$ and $W_t^3 \in \mathbb{R}^{n\times100}$ (see Figure 7 and Section 2.1.2). Recall that the matrix W_t^l holds the weights for the connections between layer l and $l + 1$ in subnetwork $\mathtt{SN}_t$ for $l = 1, 2, 3$ and that $d = 2n + m$.

In Section 2.2.3 we learned that the initialization of the weights is very important for preventing the vanishing gradient problem [9]. Therefore, we use the weight initialization strategy for the *ReLU* activation function suggested by [15] and [12], which is found to perform best for deep neural networks. This means that all weights are initialized using a truncated normal distribution. The mean of this distribution is 0 and the standard deviation is $\frac{2}{\sqrt{N_l+N_{l+1}}}$, where N_l is the number of neurons in layer l.

4.5.2 Running the Adam Optimizer

The initialized weights are passed on to the TensorFlow graph as *tf.Variables*. We choose the Adam algorithm to optimize these parameters and solve the problem shown in (29). Furthermore, we use the configurations of the hyperparameters α_1 and α_2 suggested by [20] and set $\alpha_1 = 0.9$ and $\alpha_2 = 0.999$, as well as $\epsilon = 10^{-8}$.

Next, the loss $\mathcal{C}^{\$}$ of the computation graph is defined as output of the function $DNN\_cost()$ and passed on to the Adam optimizer as objective function:

```
C_T_D, RCE = DNN_cost(tfg, ecm_tf, ecm, ones, p_tilde_t, x_t, trainable_params)
optimizer = tf.train.AdamOptimizer(learning_rate=config.learning_rate).minimize(C_T_D)
```

Note that this is very simple to implement in TensorFlow and that we label $\mathcal{C}^{\$}$ as C_T_D in the code. Before the training starts, a validation sample is created in order to enable the monitoring of the learning curve:

```
p_tilde_t_validation = ecm.P_simu(config.valid_sample_size)
x_t_validation = ecm.X_simu(config.valid_sample_size)
ones_validation = np.ones((config.valid_sample_size,1))
feed_dict_validation = {p_tilde_t: p_tilde_t_validation, x_t: x_t_validation, \
                        ones: ones_validation}
```

Here, the functions $P\_simu()$ and $X\_simu()$ are the python functions discussed in Section 4.2.2 that sample the training data or validation set. During the training process, training batches are created similarly by these functions and are fed to the graph into the corresponding *tf.placeholder* nodes. Then, the output node for the loss $\mathcal{C}^{\$}$ is evaluated by conducting a forward pass through the complete graph. Subsequently, the gradient of the loss with respect to all trainable parameters is calculated by TensorFlow using automatic differentiation (see Section 4.1.2). Using this gradient, the Adam optimizer performs one parameter update according to the rules in Algorithm 4. All of these actions are condensed in only one line of code:

```
sess.run(optimizer, feed_dict={p_tilde_t: p_tilde_t_train, x_t: x_t_train, \
                               ones: ones_batch})
```

Here, we see how easy the training of the network becomes after having carefully set up the computation graph and execution cost model. The procedure of updating the trainable parameters is repeated until the maximum number of iteration steps is reached. Progress made is displayed continuously with the validation data after a fixed number of iterations.

At the end, a summary of the results is presented and the training duration is measured. Then, the fully trained weights are extracted from the TensorFlow graph and the session is closed. Finally, a test sample is drawn and the average total execution cost above the no-impact cost for the test sample is calculated to evaluate the performance of the model and check whether it was overfitted or not. The full code can be seen in Appendix A.5. Numerical results for different configurations of the experiment can be found in the next section.

5 Results of Numerical Experiments

5.1 Performance of the Deep Neural Network

5.1.1 Relative Execution Costs

We first analyze the DNN's performance by investigating the relative execution cost compared to the optimal analytical solution of the dynamic programming algorithm for different time horizons. For this purpose, we fix $n = 10$ and $m = 3$. Hence, we get $\boldsymbol{a}_t \in \mathbb{R}^{10}$ for the control variables and $\boldsymbol{s}_t \in \mathbb{R}^{23}$ for the state variables, which determines a high-dimensional stochastic control problem according to our execution cost model. For simplicity, we choose $\bar{\boldsymbol{a}}_i = 10$ for all $i = 1, \dots, n$ and $\Lambda = 0.0001$. The number of hidden layers per subnetwork and the number of neurons per layer are kept the same as in Figure 7: $N = 2$ hidden layers consisting of 100 neurons each. Furthermore, we specify the initial batch size as 64 and the learning rate as 0.001. In Figure 9, we plot the learning curves for three different time horizons $T = 20, 25, 30$ and the above mentioned configurations over 15000 iterations.

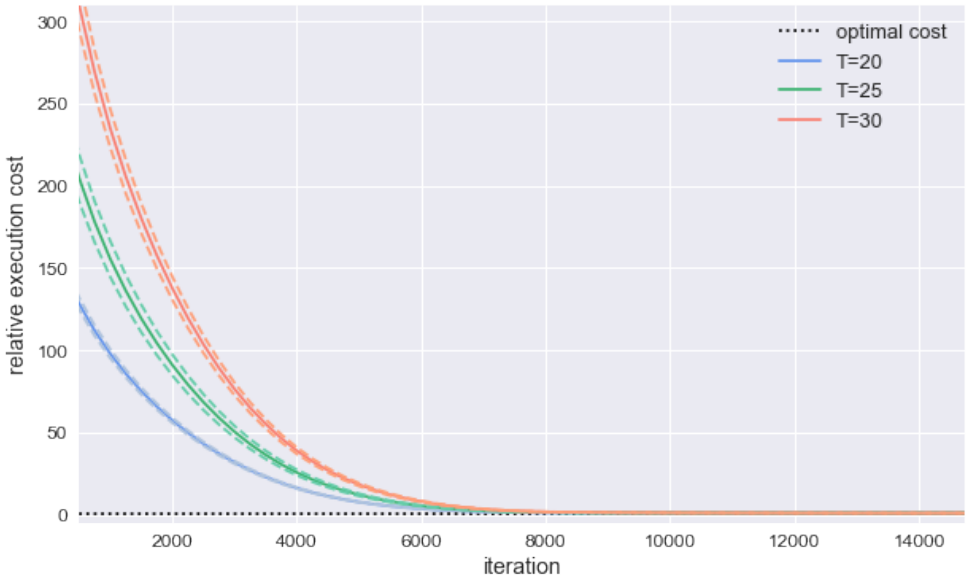


Figure 9: Relative execution costs on the validation sample during training compared to the optimal analytical solution over different time horizons. One curve represents the average costs for one time horizon over five different random seeds and training runs. The dashed lines show the mean ± the standard deviation of the five runs.

The average running times for one training run is 2732 s, 3499 s, 4315 s for $T = 20, 25, 30$. The black dotted line in Figure 9 represents the optimal analytical execution cost $\mathcal{C}^{\$}_{optimal}$ (see (28)) rescaled to 1. Thus, the learning curves represent the value

$$\mathcal{C}^{\$}_{relative} = \frac{\mathcal{C}^{\$}_{validation}}{\mathcal{C}^{\$}_{optimal}} \tag{31}$$

at the current iteration during training, where $\mathcal{C}^{\$}_{validation}$ is the average execution cost of the validation sample in cents per share above the no impact cost. After training, the `DNN`'s performance is evaluated on a test sample of size $32 \cdot 64 = 2048$ (a multiple of the training batch size). The average relative trading costs over five random seeds on test data are 1.001, 1.004, 1.005 for $T = 20, 25, 30$ respectively.

Overall, the `DNN` performs very well and can execute the order $\bar{\boldsymbol{a}}$ at near-optimal execution costs. Moreover, we observe that the accuracy declines a little with increasing time horizon, which is to be expected due to the fact that unfavorable deviations from the optimal solution accumulate in the cost over time.

5.1.2 Relative Control Error

To further investigate the deviations between the trading decisions of the `DNN` and the optimal trading strategy, we define the relative control error (RCE) for one training sample as follows:

$$RCE = \frac{1}{T-1} \sum_{t=0}^{T-2} \frac{|\boldsymbol{a}_t - \boldsymbol{a}_t^{optimal}|}{\boldsymbol{a}_t^{optimal}}. \tag{32}$$

Note that the trade in the last period is not taken into account, as it is just the remaining part of the order. The average RCE over the validation sample during training for one time horizon $T = 20$ is monitored in Figure 10.

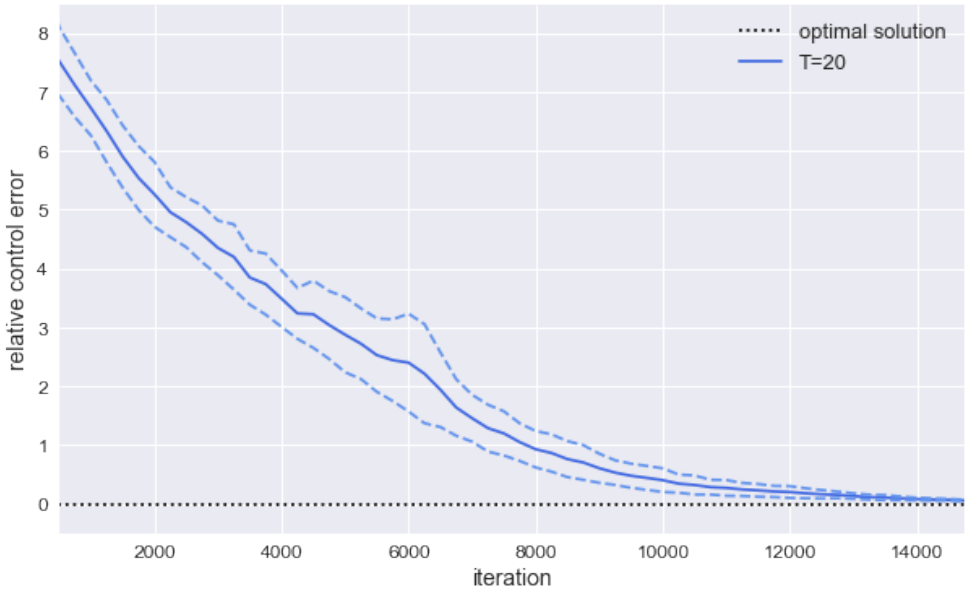


Figure 10: Relative control error RCE on the validation sample during training compared to the optimal trading strategy for $T = 20$ over five different training runs. The dashed lines show the mean $\pm$ the standard deviation of the five runs.

We observe that the trading strategy computed by the network approximates the exact solution well. This means that the trades that are executed by the network mimic the optimal trading strategy to a certain degree. On test samples, the average RCE over five different random seeds is 6.2%, 11.4%, 15.6% for $T = 20, 25, 30$.

In Figure 10, we see that there are some bumps in the curve for $T = 20$ and that there are moderate differences between different training runs. This is assumed to result from the fact that the deep neural network is not specifically asked to minimize the relative error of the trades compared to the exact solution. It is only trained to minimize total execution costs for completing the order over the fixed time horizon, independently of the trading strategy used. Hence, the curve for the RCE is not as smooth as the learning curve for the relative costs, where the gradient descent algorithm gradually moves towards a minimum on the surface of the loss function. The reason why the network mimics the optimal trading strategy, instead of developing a completely different strategy that achieves near-minimal cost, is assumed to lie in the nature of the execution cost model and of the cost function.

5.2 Sensitivity Analysis

5.2.1 Varying Time and Portfolio Size

After looking at relative execution costs and having tested the performance of the network in terms of its ability to approximate the optimal solution for one configuration of the model, we conduct a sensitivity analysis of the absolute optimal execution costs. Subsequently, we evaluate whether the network can reproduce its high performance for a more realistic and complex setting of the execution cost model. In a first step, we compute and compare the optimal execution costs for different portfolio sizes and time horizons, which can be seen in Figure 11.

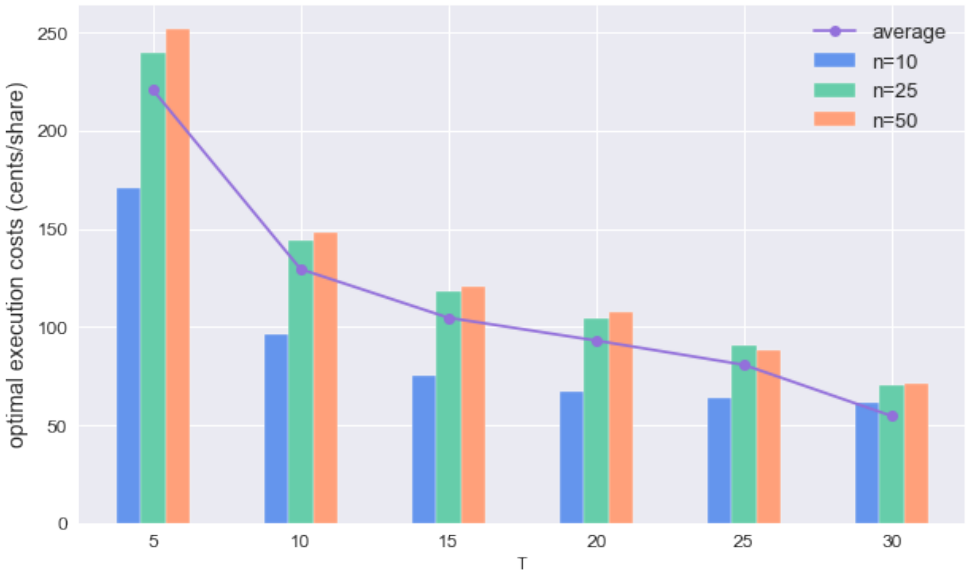


Figure 11: Optimal expected execution costs in cents per share above no-impact costs for different time horizons and numbers of stocks in the portfolio. The purple line represents the average execution costs for one time horizon over the three portfolio sizes $n = 10, 25, 50$. Other configurations for this experiment are $m = 3$, $\bar{\boldsymbol{a}}_i = 10$ and $\Lambda = 0.0001$.

The primary observation that we make is that expected execution costs fall with increasing time horizon. The line plot of the average visualizes this trend. This happens due to the fact that trading can be split over more time periods and because of the resulting flexibility to wait for more favorable conditions to trade [3]. Second, we notice that a larger portfolio leads to higher expected execution costs in most cases. Moreover, while the costs for $n = 25$ and $n = 50$ are similar, the costs for portfolios with only 10 different stocks are considerably lower. However, it is expected that this trend is not necessarily true in general, but only for our configuration of the execution cost model. It could be, for

instance, that including more stocks that are negatively correlated to the existing stocks in the portfolio decreases overall execution costs per share. Trading in one stock could lead to lower execution prices in another stock via the price impact and therefore lead to lower expected execution costs in total compared to a portfolio without this security. This phenomenon is known as diversification effect.

In addition, we find the absolute optimal execution cost of our model to be significantly higher than the costs calculated in the empirical example in [3]. This is of course a result of the choice of the model parameters. In Figure 11, for each time horizon, the expected optimal execution costs are more than 50 cents per share, going up to 250 for $T = 5$. In comparison, execution costs in the research in [3] only range from -8 to 14 cents per share. A major factor that influences the range of execution costs is the sensitivity of the price impact to the order size in our model. Thus, we analyze the optimal costs for varying values of $\bar{\boldsymbol{a}}$ and configurations of the matrix $\boldsymbol{A}$ in the next section.

5.2.2 Varying Order Size and Price Impact

To arrive at more realistic execution costs, we need to select a reasonable scale for the elements of $\boldsymbol{A}$, which measures the sensitivity of the price impact in the execution price to the trade size (see Section 3.2.2). Remember that in our implementation of the model, the scale of the elements of $\boldsymbol{A}$ is determined by Λ (see Section 4.2.1). In the previous experiments, we chose $\Lambda = 0.0001$ and $\bar{\boldsymbol{a}}_i = 10$. However, this led to unrealistically high execution costs. Moreover, the approach to solve the execution cost problem is developed for large institutional investors, who's order size can reach 10000 or more. So, the costs would be even higher for these large trades. Hence, we develope a heatmap in Figure 12 that shows optimal expected execution costs for different order sizes and values of Λ. Based on this table, we can make a more realistic choice of Λ.

Λ \ $\bar{\boldsymbol{a}}_i$	10	100	1000
0.001	450.23	4253.76	42286.45
0.0001	67.34	450.23	4253.76
0.00001	3.46	67.34	450.23

Figure 12: Optimal expected execution costs in cents per share above no-impact costs for different order sizes and values of Λ. Low execution costs are indicated in green, while high costs are shown in red. Other configurations for this experiment are $m = 3$, $n = 10$ and $T = 20$.

We see in Figure 12 that increasing the order size or Λ results in higher expected optimal execution costs. By using the same random seed, we notice that scaling $\bar{\boldsymbol{a}}_i$ or Λ by the same factor has the same effect on costs. This is no surprise, because we use a linear percentage price impact model and the scalar has the same influence on the price impact $\boldsymbol{\delta}_t$ in (22).

Setting for example $\Lambda = 0.00001$ and $\bar{\boldsymbol{a}}_i = 10$, we receive 3.46 as execution costs in cents per share, which is closer to the solutions in the empirical example of [3]. Nevertheless, for larger trade sizes, the price impact is still too high. Hence, we change $\bar{\boldsymbol{a}}_i$ to 1000 and decrease Λ to 1e-7. Therefore, we reduce the values of $\boldsymbol{A}$, which now lie in a similar range as in [3]. Although not shown in the heatmap, this produces optimal execution costs of 3.46 as well. We then repeat the experiment from above and test the networks performance on these more realistic and adjusted configurations. This time, we plot the learning curves in absolute values. The results for three different training runs can be found in Figure 13.

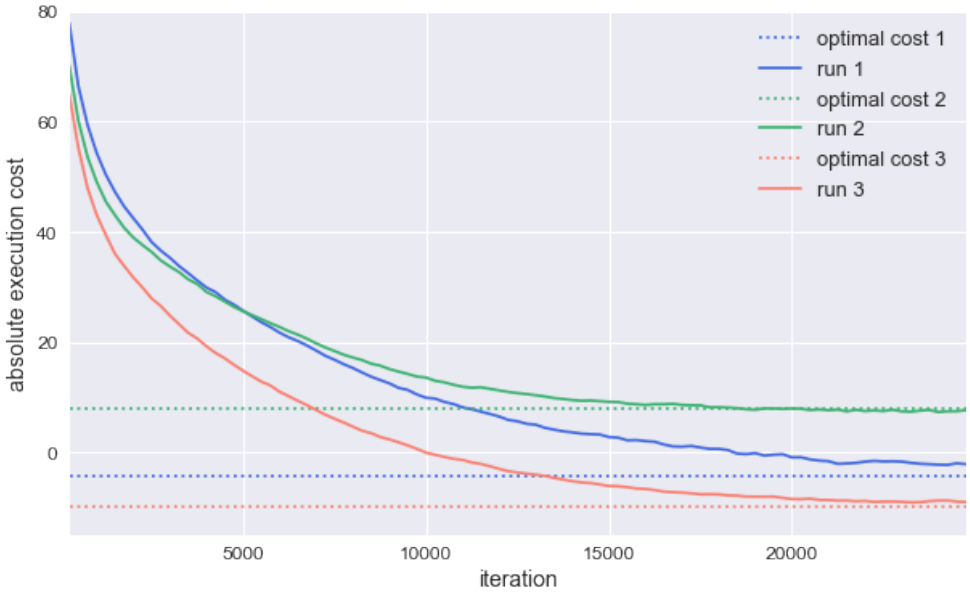


Figure 13: Absolute execution costs on the validation sample of three different training runs. The respective optimal analytic solution is shown as a dotted line. The configurations for this experiment are $m = 3$, $n = 10$, $T = 20$, $\bar{\boldsymbol{a}}_i = 1000$ and $\Lambda = 0.0000001$. We use 25000 iteration steps and a learning rate of 0.05 for training.

We first observe that optimal absolute execution costs for two of the three training runs on randomly drawn validation samples of our experiment are negative. Remember that due to the fact that we do not use no-sales constraints, the network can decide to sell stocks over the given time periods, as long as it completes the full order by time T. Consequently, if the price impact and the trade size are low, execution costs can turn negative and the networks trading strategy can optimally use the information available and execute orders

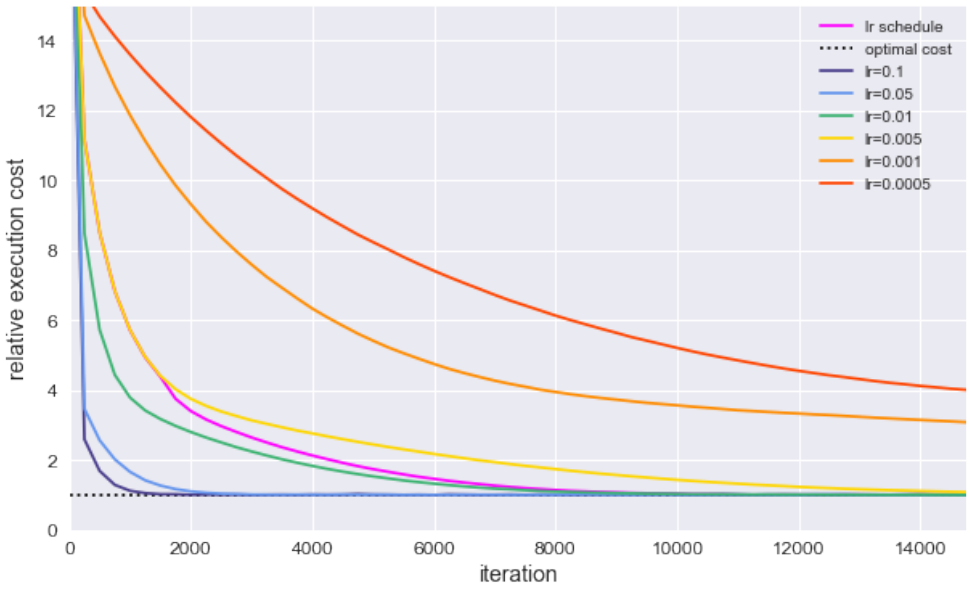
at a profit. In this case, the impact of information dominates the price impact from the shares traded, which are both components of the execution price.

Next, we notice that at only 15000 iterations, the difference between the learning curves and the corresponding optimal cost is big. In order to still receive a good convergence, we increase the learning rate to 0.05 and the iteration steps to 25000. As a result, the average running time for these three training runs grows to 4491s. Using the same learning rate as in the experiment in Figure 9 would take even longer to train the model for this configuration. However, using a higher constant learning rate prevents the optimization algorithm during training to accurately approximate the minimum of the loss function. We see this, for instance, in Figure 13 at the blue learning curve, which does not reach the optimum. Altogether, this highlights again the importance of the fine tuning of training specifications, which have to be re-calibrated for each new configuration of the model. An application of a learning rate schedule, where the learning rate is decreased towards the end of training, can help improve the convergence (see below in Section 5.3).

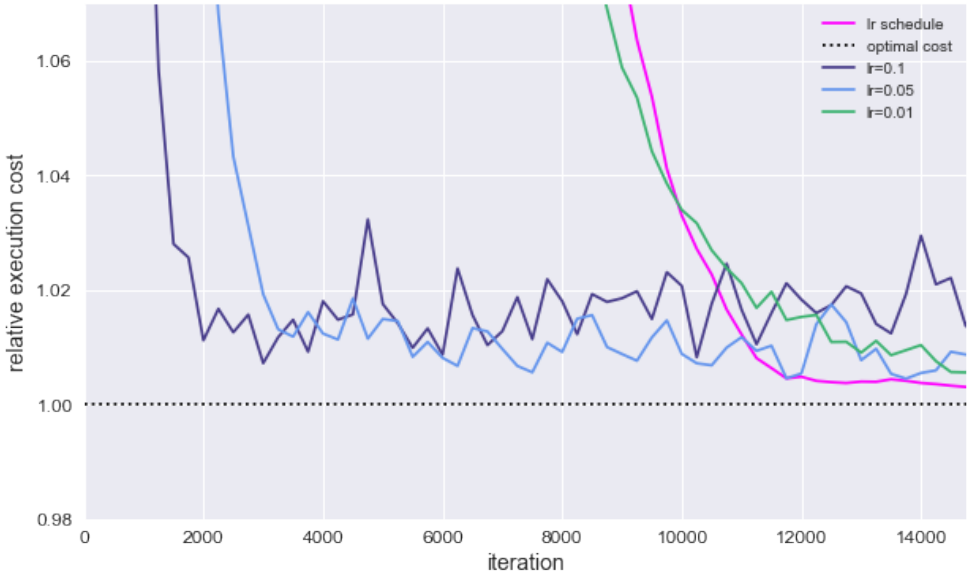
Lastly, we evaluate the trained network for each of the three training runs on test data. For all test samples drawn, the execution costs achieved with the optimal analytical trading strategy are found to be negative. Consequently, we can speak of profits realized by trading. The network achieves 98.5%, 96.3% and 96.0% of these profits. Compared to the performance of the network for the configurations in Section 5.1.1, this is slightly more inaccurate. Yet, the network still demonstrates the ability to approximate well the optimal execution costs (or profits) for a more complex configuration of the problem with a considerably higher order size. It is expected that this approximation can even be improved by using a smaller learning rate. To find a good balance between running time and accuracy, we conduct the following analysis of the learning rate.

5.3 Fine Tuning of the Learning Rate

For the next experiment, we choose $n = 10$, $T = 25$ and $m = 3$. In addition, we set $\bar{\boldsymbol{a}}_i = 100$ and $\Lambda =$ 1e-5, to get a compromise between the fast running times of the configurations in 5.1.1 and the complexity of the model in 5.2.2. In order to determine an appropriate learning rate schedule, we first visualize the learning curves for different values of a fixed learning rate. These learning curves can be found in Figure 14.



(a)



(b)

Figure 14: Relative execution costs on the validation sample during training compared to the optimal analytical solution for different constant learning rates and a learning rate schedule. Plot (a) shows the full learning curves during training, while (b) only shows the costs close to the optimal solution. 15000 iteration steps are monitored.

We observe that the orange colored learning rates 0.0005 and 0.001 are too small. Learning of the network with these rates happens too slowly and the Adam algorithm cannot converge during the 15000 training iterations. The learning rates shown in blue are too big (0.1 and 0.05). Although the relative execution cost decreases very quickly during training for these rates, the optimal solution cannot be approximated adequately, because the step size is too large. Hence, the curves oscillate heavily and jump around the optimum as seen in the zoomed illustration in Figure 14 (b).

The ideal learning curve for this configuration of the execution cost model lies in between 0.01 and 0.005, as we can obtain fast progress in the beginning and still converge to a good solution in time. Therefore, we choose a learning rate schedule with a predetermined piecewise constant learning rate, where we incrementally decrease the learning rate when the curve starts to flatten. The learning curve with the learning rate schedule is visualized in pink in Figure 14. We select an initial rate of 0.005 and decrease it after 15% of the training to 0.001. Finally, towards the end of training after 80% and 90%, we decrease it again in order to prevent oscillations and approximate the optimum as accurately as possible. In Figure 14 (b) we can observe that this works better than just choosing a learning rate of 0.01 and convergence is smoother.

6 Conclusion

In this thesis, we implement a deep learning approach to solve the high-dimensional stochastic control problem of minimizing execution costs for portfolios in a finite time horizon. The idea is to approximate the optimal trading strategy at each time step using feed-forward neural networks. Stacking these networks together forms a very deep neural network that can learn to trade efficiently and minimize loss, which is defined as the total execution costs.

The challenge in the implementation of the deep learning algorithm is the correct interconnection of the subnetworks based on the dynamics of the stochastic execution cost model. The architecture of the overall deep neural network must be specified very carefully. Moreover, the incorporation of several training acceleration techniques adds complexity to the model. However, this prevents slow learning and facilitates the evaluation of the network's performance.

Numerical results of our experiments suggest that the deep learning approach approximates the optimal solution very well. The network learns to trade with near-optimal execution costs and can handle the high-dimensionality of the portfolio problem. However, it requires the fine tuning of training parameters, such as the learning rate, in order to do so in a reasonable running time. Furthermore, it not only learns to trade with minimal execution costs, but also to mimic the optimal trading strategy.

The training data, as well as the model parameters are predominantly created artificially in this thesis. Therefore, it is important to make realistic assumptions in order to arrive at appropriate execution costs. For a real-world application in practice, however, many of the parameters are unknown and must be estimated in a costly procedure. This applies especially to the parameters of the price modelling, such as the cross-stock price impact from trading. In addition, changing the underlying portfolio, for example adding new stocks, requires a frequent re-calibrating of these parameters. Hence, the integration of the deep learning approach in the investment process of an institutional investor can be challenging.

Nevertheless, the general idea of solving high-dimensional stochastic control problems with the use of artificial intelligence is very promising. The deep learning approach to the execution cost problem could be adapted to other fields of research, such as a resource allocation problem. Approximating time dependent controls is shown to work very well with fully connected feed-forward neural networks in this thesis. Depending on the application, the model dynamics that define the exact connection of these subnetworks need to be adjusted. Then, the presented approach is expected to avoid the curse of dimensionality that many real stochastic problems are suffering of and to offer an accurate and fast approximation of the optimal solution.

7 Appendix

Appendix A: Code

Appendix B: Assumptions for Execution Cost Model

Assumption 4 (Mean of $\boldsymbol{z}_t$)**.**

$$(\boldsymbol{\mu}_z)_i \sim \boldsymbol{\mathcal{N}}(0, 1\text{e}{-5}), \quad i = 1, \ldots, n.$$

Assumption 5 (Elements of $\boldsymbol{B}$)**.**

$$\boldsymbol{B}_{ij} \sim \boldsymbol{\mathcal{U}}(0.5\text{e}{-4}, 1.5\text{e}{-4}), \quad i = 1, \ldots, n,\ j = 1, \ldots, m.$$

Assumption 6 (Elements of $\boldsymbol{C}$)**.**

$$\boldsymbol{C}_{ij} \sim \boldsymbol{\mathcal{N}}(0, 0.5), \quad i, j = 1, \ldots, m.$$

8 Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, et al. Tensorflow : Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467, 01 2015.

[2] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] D. Bertimas, A. Lo, and P. Hummel. Optimal control of execution costs for portfolios. *Computing in Science & Engineering*, 1:40 – 53, 12 1999.

[4] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 01 2003.

[5] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 4th edition, 2017.

[6] J. P. Bouchaud. Price impact. arXiv:0903.2428, 2009.

[7] L. K. C. Chan and J. Lakonishok. The behavior of stock prices around institutional trades. *The Journal of the American Finance Association*, 50, 1995.

[8] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 12 1989.

[9] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics*, 9, 2010.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] Google AI. Ai for everyone: inside tensorflow, our open-source machine learning platform, 2019. `https://ai.google/stories/tensorflow/`.

[12] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017.

[13] J. Han and W. E. Deep learning approximation for stochastic control problems. *Deep Reinforcement Learning Workshop, NIPS 2016*, 2016.

[14] B. Hanin. Universal function approximation by deep neural nets with bounded width and relu activations. arXiv:1708.02691, 2017.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. arXiv:1502.01852v1, 2015.

[16] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty Fuzziness and Knowledge-Based Systems*, 6:107–116, 1998.

[17] K. Hornik. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[18] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167, 2015.

[19] H. J. Kappen. Stochastic optimal control theory. International Conference on Machine Learning, July 2008.

[20] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. arXiv:1412.6980, 2015.

[21] A. Lo and D. Bertsimas. Optimal control of execution costs. *Journal of Financial Markets*, 1:1–50, 02 1998.

[22] T. F. Loeb. Trading cost: The critical link between investment information and results. *Financial Analysts Journal*, 39(3):39–44, 1983.

[23] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis. Dying relu and initialization: Theory and numerical examples. arXiv:1903.06733, 2019.

[24] C. E. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. arXiv:1811.03378, 2018.

[25] Oxford University Press. artificial intelligence, 2019. https://www.lexico.com/en/definition/artificial_intelligence.

[26] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[27] S. Ruder. An overview of gradient descent optimization algorithms. arXiv:1609.04747, 2016.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[29] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le. Don't decay the learning rate, increase the batch size. *Google Brain*, 2018.

[30] C.-F. Wang. The vanishing gradient problem, 2019. https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.