Deep learning for portfolio optimization

Dzmitry Sei

January 2024

1 Deep learning

In this section basic principles of deep learning will be described. Moreover, we will describe the type of layers used in the thesis.

1.1 General description of NN principles

This subsection will describe general ideas of neural networks in a manner adapted to the context of the master thesis.

All the NNs described in the thesis are implemented using the author's fork of an open-source Python package DeepDow [27]. DeepDow is a deep learning portfolio optimization library, written by Jan Krepl in 2020. The package integrates both stages of the investment process mentioned before. The package can be seen as a combination of auto differentiable convex optimization package CVXPYLAYERS (see later) and code that eases working with financial data. Since the extensive usage of the package, we will describe DeepDow's way of working with NNs.

An NN can be seen as just a function $F : I \to W$, which transforms input I to the output W.

The output W is a vector of weights, such as $W \in R^{assets}$ and $\Sigma w_i = 1$. Plus we can set any restriction on w_i that preserves convexity. Suppose we have 5 assets in total, where $asset_1, asset_2, asset_3$ are from the banking sector and assets $asset_4, asset_5$ are from the technological sector. Suppose we don't want to have too big exposure to the banking sector. In this case, we can make our NN not invest more than z faction of our money (considering both short and long allocation) into this sector by setting the following convex restriction on the output $-z < w_1 + w_2 + w_3 < z$. The set of the restriction, preserving convexity, we can set on the output is very broad [22], which gives us big flexibility for the formulation of investment ideas for our NN.

In the thesis, I is a 3D tensor of asset returns $I \in R^{channels,assets,time}$. The dimensions are:

1. Channel. The dimension containing relevant asset information can be relevant returns, volumes, financial ratios, or any other data. In the thesis, only information about asset returns will be used, so the size of the channel dimension is 1.
2. Time. Controlling the amount of data we are interested in, affecting both past and future. For example, if we are working with past daily data and the dimension is equal to five, this will mean the tensor we have information about the asset for the past five days.

3. Assets. Reflecting the amount of assets we are working with. If we are working with three ETFs, the dimension size is three.

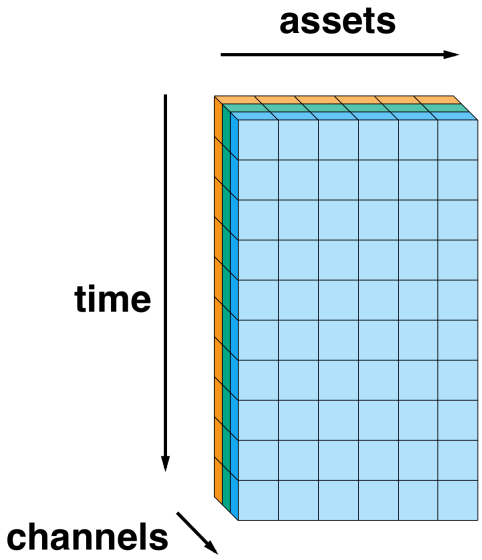

Рис. 1: DeepDow financial data representation [27].

This structure naturally allows us to split any financial data into past, irrelevant immediate future, and useful future as shown in the image below. Irrelevant immediate future can appear in the context of high-frequency portfolio rebalancing, where because of decision-making computational time we had to skip the nearest future. But in the thesis, we are working with daily data, so the time dimension of the immediate future will be zero.

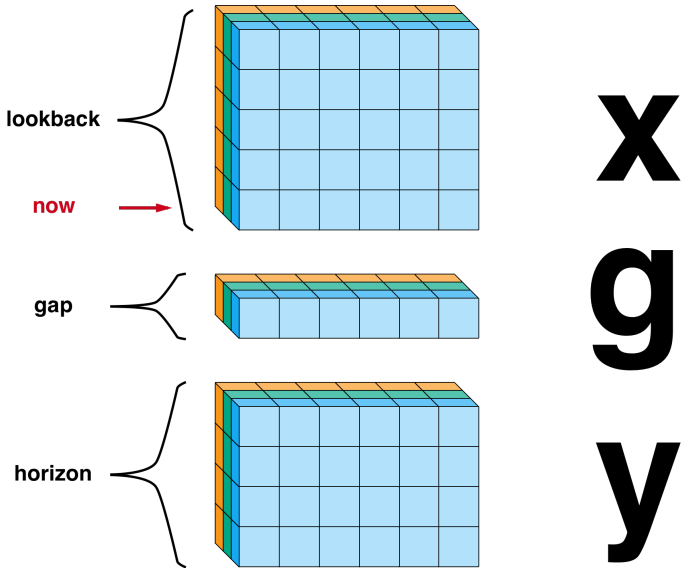

Рис. 2: Time through the eyes of DeepDow [27].

Figure 2 represents how DeepDow is working with time. Initially, tensor x encapsulates all

historical and current knowledge. The second tensor, g, embodies information about the immediate future, which is not accessible for making investment decisions. The last tensor, y, represents the future trajectory of the market.

The amount of data points in the past (x) is controlled by the *lookback* parameter. For example, suppose we are working with daily data and lookback = 3; then, we will look only at 3 days in the past. The (useless) immediate future (g) is controlled by the *gap* parameter, for instance, gap = 2 implies that the amount of days in the immediate future (g) is two. Similarly, the amount of days we are looking forward into the future (y) is controlled by the *horizon* parameter, following the same logic as for the other parameters.

In the figure below it is shown how this methodology combined with the rolling window principle is working with row data.

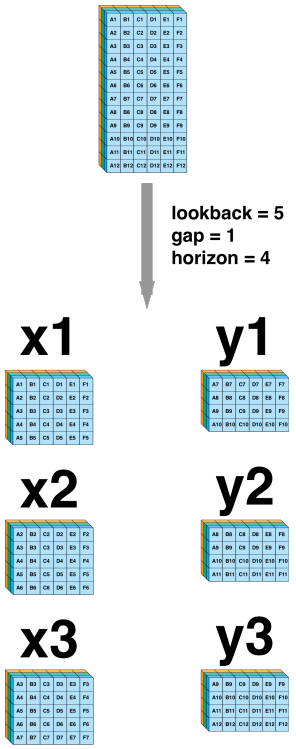

Рис. 3: DeepDow data example [27].

Diving deeply, an NN can be defined as a graph of embedded functions, depending on learnable parameters

$$F(x, \theta) = F_n(x, \theta, F_1, F_2 \ldots, F_{n-1}), \tag{1}$$

where θ is a sequence of learnable parameters and F_i is a layer. A layer is just a function, which transforms its input based on a subset of learnable parameters θ_i, such as $\theta_i \in \theta$.

In the simplest case, NN's graph can have a sequential form. In this case, an NN can be expressed in the following way:

$$F(x, \theta) = F_n\left(F_{n-1}\left(\ldots F_2\left(F_1(x, \theta_1), \theta_2\right)\ldots, \theta_{n-1}\right), \theta_n\right) \tag{2}$$

So in the context of the thesis, NN can be viewed as a function, that transforms a tensor (actually a matrix) of portfolio returns into a vector of weights, as visualized in the figure below.

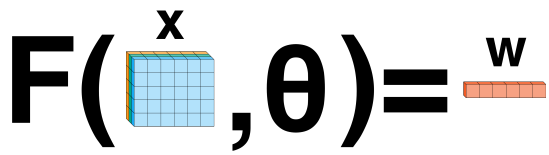

Рис. 4: From returns to portfolio weights based on learned θ[27].

An NN learns θ by minimizing a specific loss function L over the entire (or its subset) dataset used for training.

Suppose we have a training dataset (set of tensors obtained from the initial returns dataset in a way shown in Figure 3) $\{(x_i, y_i)\}_{i=1}^{m}$. In this case, the goal of the learning phase of our NN F is to solve the following optimization problem:

$$\theta^* = \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} L(F(x_i; \theta), y_i). \tag{3}$$

where L is a loss function. The exact optimization process and loss functions used will be described later. At this stage it should be noted that loss functions in the thesis are computed in 2 stages:

1. Based on portfolio weights (output of NN $F(x_i, \theta)$) and information about asset returns in future y_i we can compute a vector of portfolio returns in future r_i.
2. The loss function is computed as a scalar function of the vector of portfolio returns r_i.

The whole process is visualized in the figure below.

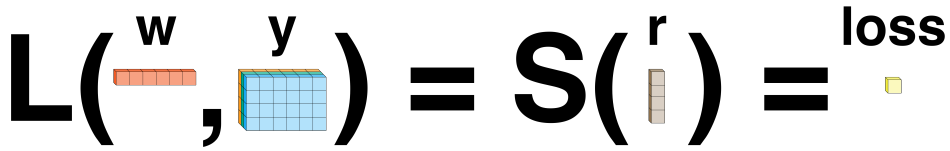

Рис. 5: Loss function computation process [27].

1.2 Dense layers

1.2.1 General description

We call a layer F_i a dense layer if it transforms an input in the following way:

$$F_i(y, \theta) = \sigma_i(W_i y + b_i), \tag{4}$$

where W_i is layer weights, b_i is layer bias and σ is an activation function [33]. Both W_i and b_i are part of learnable parameters θ.

Activation function σ_i is introduced to allow an NN to deal with nonlinear problems. As we know, the combination of linear functions is a linear function. For example, if we are working with sequential NN, described in Equation 2 with $F_i(y, \theta) = W_i y + b_i$ for all i (σ_i is identity function), the whole NN will be a linear function. In the end, a linear NN is undesirable, because it can't catch complex dependencies in the data (for example can't be used in XOR classification problems [21]).

1.2.2 Activation functions

As mentioned before, in essence, activation functions are just transformers of linear signals into non-linear ones. The set of plausible activation functions is enormous, and a lot of unpublished activation functions performs as well as conventional ones. For example, the unconventional *cos* activation function on the MNIST dataset gives an error rate that is comparable with one of the conventional activation functions [8]. In this section, we will describe only those activation functions among which we will be selecting for constructing our NNs.

In general activation functions are selected from the set of almost everywhere differential functions, which behave like linear, to ease the optimization process.

The most popular activation function is ReLu activation $\sigma(x) = max(0, x)$ [2]. Note that the function is not differentiable at 0, but this is not the problem because of floating-point arithmetic [7] the probability to get 0 as input of the activation function is almost zero, so this doesn't create a problem in most cases. But if the input is zero we set $\sigma'(0) = 0$ by convention. The second-order derivative of the function is zero, so all the information relevant to optimization is contained in the gradient.

The main drawback of ReLu is the inability to make the parameter updates using gradient in case of non-positive input. To overcome this limitation a lot of generalizations of ReLu were developed, which can find informative gradients everywhere. In the thesis, we are examining Leaky ReLu.

The Leaky ReLU function is defined as follows:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Where α is a small constant typically set to a small positive value, often around 0.01.

Compared to the standard ReLU function, which outputs 0 for any non-positive input, Leaky ReLU allows a small, non-zero gradient for non-positive inputs.

The introduction of this small slope for negative inputs helps address the "dying ReLU"problem, which occurs when neurons become inactive during training because they consistently output zero for negative inputs, effectively stopping the gradient flow and hindering the learning process.

By allowing a small gradient for negative inputs, Leaky ReLU helps to alleviate this issue and enables neurons to continue learning, even when the input is negative. This can lead to more robust training and improved performance, especially in deep neural networks where the dying ReLU problem can be prevalent [18].

Another activation function capable of mitigating the dying ReLU problem is the Maxout activation function [9]. Suppose an input to the layer is a $X \in R^d$ vector. Each element X_i of this vector in the context of neural networks is called a neuron.

Maxout considers that each X_i of the dense layer has its own set of learnable weights $\{(W_{ij}, b_{ij})\}$ for $j =$ 1 to k

The activation function is defined as follows:

$$\sigma(X)_i = \max(W_{i1}^T x + b_{i1}, W_{i2}^T X + b_{i2}, \ldots, W_{ik}^T X + b_{ik})$$

where W_{ij} and b_{ij} represent weight vectors and bias terms specific to neuron i, with k being the number of linear functions considered. Maxout provides the network with the ability to learn an arbitrary convex function, enhancing flexibility and robustness compared to traditional activation functions. However, it should be noted that the increased expressive power of Maxout comes at the cost of higher computational complexity.

1.2.3 Using activation functions for portfolio construction

We can construct an activation function, whose output is a vector, which sums up to 1. So the output can be seen as portfolio weights.

For constructing long-only portfolios we can use the softmax activation function. It is commonly used for classification, but we can treat a Multinoulli output of the activation function as portfolio weights.

The activation function transforms a vector of real-valued scores (logits) into a probability distribution over multiple classes. Mathematically, given an input vector z of length K, the softmax function computes the probability p_i of each class i as follows:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where z_i is the ith element of the input vector z. This function ensures that the output probabilities sum up to 1, making it suitable for representing a probability distribution over multiple classes or portfolio weights among several assets.

For incorporating short selling, we can utilize a normalized version of the hyperbolic tangent function (tanh), which outputs values in the range $(-1, 1)$. Given the same input vector z of length K, the normalized tanh function transforms each element z_i as follows:

$$t_i = \tanh(z_i)$$

After applying the tanh function, we normalize the outputs to ensure the sum of their absolute values equals 1, which is suitable for representing portfolio weights including short positions. The normalization can be expressed as:

$$w_i = \frac{t_i}{\sum_{j=1}^{K} t_j}$$

where w_i represents the weight of the ith asset in the portfolio, ensuring that the portfolio weights accommodate both long and short positions with their absolute values summing up to 1.

Dense NN as universal approximator Despite its relative simplicity dense NNs (NNs from Equation 2, with F_i being a dense layer) are a very powerful tool because of the Universal Approximation Theorem.

Consider a simple dense neural network defined by the function

$$y = W_2\sigma(W_1x + b_1) + b_2$$

where

- x is the input vector from R^n,
- W_1 is the weight matrix connecting the input layer to the hidden layer,
- W_2 is the weight matrix connecting the hidden layer to the output layer, facilitating the mapping to R^m,
- b_1 and b_2 are the bias vectors for the hidden and output layers, respectively,
- σ denotes the activation function applied element-wise to the hidden layer outputs,
- y is the output vector in R^m,

The theorem ensures that for any continuous function $g : \mathbb{R}^n \to \mathbb{R}^m$ and for any $\epsilon > 0$, there exists a configuration of σ, W_1, W_2, b_1, and b_2 such that the neural network F can approximate g with any precision, if hidden layer size will be big enough (amount of rows in W_1 will be big enough)[11].

It was proven that the theorem works for both ReLu and Leak ReLu [15]. Since Maxout can imitate ReLu, this theorem is also applicable to Maxout.

It should be noted that the theorem states the existence of an approximating NN for any continuous function, but doesn't guarantee that it is possible to train an NN respectively. There is no commonly accepted way to find a well-generalizing function from the training set.

Another problem with the theorem is that it doesn't say how big amount of rows in W_1 should be to find a good approximation, but it was shown that increasing the overall amount of layers in dense NN can, in general, reduce the amount of the required neurons (or amount of raws in W_i).

In our context, the theorem can say that if exists a continuous function, which can provide us with optimal portfolio weights, based on asset returns, there is a dense NN, which perfectly approximates it

1.3 Convolutional Layer

1.3.1 Convolution

Convolutional layers have a long history [5] and exist in different forms, in the thesis, the one based on a two-dimensional cross-correlation operation is used.

A convolutional layer can be seen as a function that accepts a two-dimensional matrix as an input and returns a predefined amount of two-dimensional matrixes of the same size as an output. The amount of matrixes returned in a convolutional context is called the amount of output channels.

Given a single-channel (remember that we are using only daily returns channel) input feature map X with dimensions (H_{in}, W_{in}), and considering a convolutional layer aiming to produce an output with multiple channels (C_{out}), the operation within the layer for each output channel can be described as follows. For each filter corresponding to an output channel c, where $c \in \{1, 2, \ldots, C_{out}\}$, with each filter having dimensions (H_f, W_f) and a bias term b_c, the output Y^c at position (i, j) for channel c is computed by the formula:

$$Y_{ij}^c = b_c + \sum_{m=1}^{H_f} \sum_{n=1}^{W_f} K_{mn}^c X_{i+m,j+n} \tag{5}$$

where:

- Y_{ij}^c represents the intensity of the c-th output feature map at the spatial location (i, j).
- b_c is the bias associated with the c-th filter.
- K_{mn}^c denotes the weight of the c-th filter at location (m, n).
- $X_{i+m,j+n}$ is the value of the input feature map at the location $(i + m, j + n)$, implicating the spatial overlap between the filter and the input feature map.

b_c and K_{mn}^c are learnable parameters, which are discovered by an NN during the training process.

This operation is independently applied for each output channel c, allowing the layer to extract and transform multiple features from the single-channel input based on the diverse set of filters K^c.

Note that there is a multichannel version of the described convolutional layer, which can work with a multichannel input.

Convolutional layers are more often used for working with image data [17], but in our work, we construct a new representation from the matrix of daily asset returns. The applicability of the image-centered approach for financial returns is questionable. So actually in our work a modified version of the Equation 5 is used.

$$Y_{ij}^c = b_c + \sum_{m=1}^{H_f} K_m^c X_{i+m,j} \tag{6}$$

.

For example, if we set $H_f = 3$ and $C_{out} = 1$, the Y_{ij}^C will be a weighted average of three elements of the input plus bias. An example is shown in the figure below.

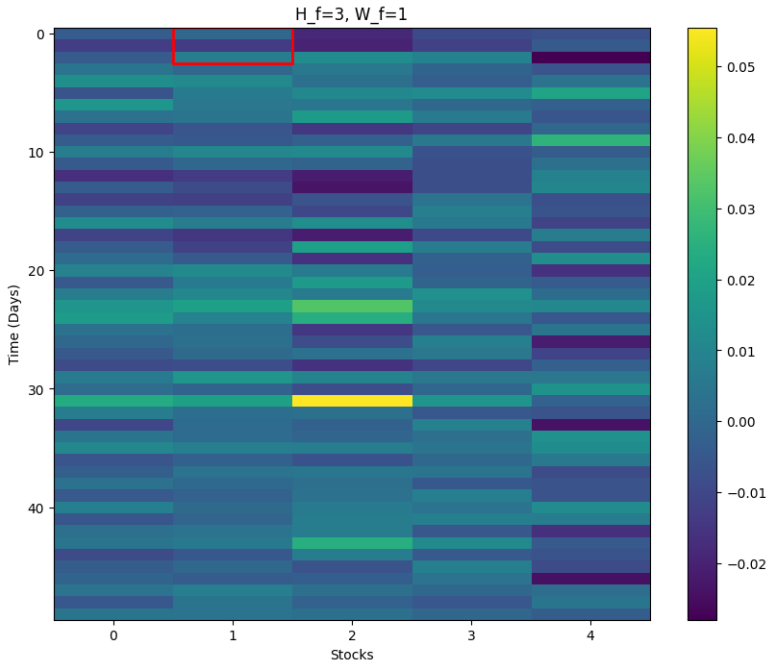

Рис. 6: $Y^c_{01} = b^1 + K^1_{00}X_{01} + K^1_{10}X_{11} + +K^1_{20}X_{21}$ (computer indexing used).

The motivation for this approach is the following: our input is a (H_{in}, W_{in}) matrix with H_{in} representing timesteps and W_{in} representing assets. While there is a certain order in the columns of the input matrix in the rows the order of returns is completely human-made. The author defined the order of assets while downloading the data. This is not the case for image data.

The figure below shows the importance of column positions for an image.

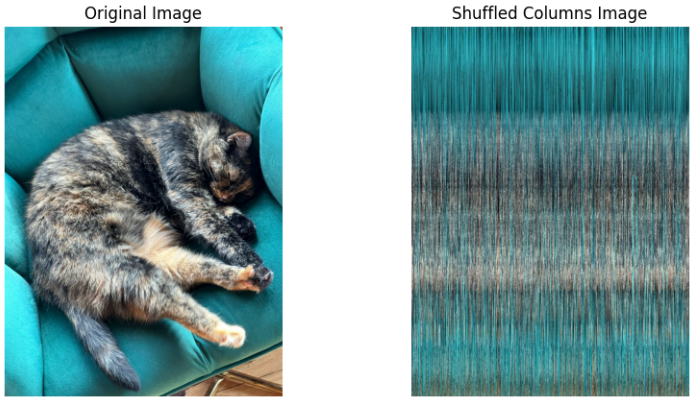

Рис. 7: The influence of columns shuffling on the image.

For asset returns, the original matrix is still informative after shuffling the columns even after removing the column names.

	Stock1	Stock2	Stock3
2024-01-01	-0.001353	0.014120	0.003198
2024-01-02	-0.001346	0.020830	-0.021771
2024-01-03	0.018512	-0.028073	0.002712
2024-01-04	0.028568	0.007888	0.007533
2024-01-05	0.007342	-0.027541	0.024717

	Stock3	Stock1	Stock2
2024-01-01	0.003198	-0.001353	0.014120
2024-01-02	-0.021771	-0.001346	0.020830
2024-01-03	0.002712	0.018512	-0.028073
2024-01-04	0.007533	0.028568	0.007888
2024-01-05	0.024717	0.007342	-0.027541

Рис. 8: The influence of columns shuffling on the stock returns matrix.

It can be claimed that for stock returns input the column's positions are not important, that is why horizontal neighbors of a stock, should be ignored during convolution.

1.3.2 Pooling

After applying the convolution layer to input matrix X, we obtain an output tensor O of dimension $(C_{out}, H_{out}, W_{cout})$. To work further with an output we need to somehow get rid of the channel dimension of the output tensor. The are a lot of aggregating functions, that can deal with the dimension, but in the thesis, we are using average pooling.

Given a tensor O from the convolutional operation with dimensions $(C_{out}, H_{out}, W_{out})$, the average pooling across channels operation is mathematically represented spatial location (i, j) as:

$$\hat{O}_{i,j} = \frac{1}{C_{out}} \sum_{c=1}^{C_{out}} O_{c,i,j}$$

This results in a tensor $\hat{O}$ with dimensions (H_{out}, W_{out}), where each element is the average of the corresponding spatial location across all output channels.

1.4 Recurrent layers

The layers described before have proved to be useful for a lot of tasks, but they have strong limitations: they assume that data points are independent, and while this is true for many data classes such as images or customer reviews it is not the case about financial returns data [16]. The asset performance at this week is definitely affected by asset performance during the previous week. So we can state the asset performance is not just a sequence of i.i.d. random variables but a stochastic process. It was caught by statisticians more than 70 years ago [23] that for modeling the future state of a process, we should somehow take into account its past state. Recurrent layers can be seen as a natural projection of these ideas on the NN field and can be defined as modified Dense layers, able to take into account states of the network in the past.

1.4.1 The recurrent layers

In the context of recurrent NNs, there is a terminology overlap. There are recurrent layers (networks) as a methodology to work with data taking into account the previous input into the NN. And there is a concrete function also called the recurrent layer. In this section, we will describe the concrete function.

In the thesis recurrent layer (or RNN cell) is a function which at time t updates its hidden state h_t and computes the output o_t at each time step t as follows [1]:

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$$

Where:

1. x_t is the input at time t.
2. $h_{(t-1)}$ is the function output at the previous time step $t-1$ (so-called hidden state), with h_0 being the initial state (in our case $h_0 = 0$).
3. W_{ih} and W_{hh} are the weight matrices for the input and hidden layers, respectively.
4. b_{ih} and b_{hh} are the bias terms for the input and hidden layers, respectively.

This formula encapsulates the core computation within an RNN cell, highlighting the recursive nature of the hidden state computation, which allows the network to maintain and update its "memory"of previous inputs through sequential data processing. The formula is visualized in the image below

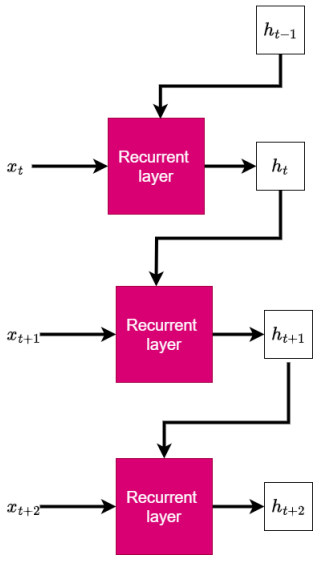

Рис. 9: Recurrent layer.

Note that the formula can be easily extended to include more hidden states (not only h_{t-1}, but h_{t-2}, h_{t-3} and so on), but in the thesis only h_{t-1} is used as a hidden state.

1.4.2 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a special kind of Recurrent Neural Network (RNN) capable of learning long-term dependencies, introduced by Hochreiter & Schmidhuber (1997) [10]. They were designed to overcome the limitations of traditional RNNs, particularly the problem of vanishing and exploding gradients [3], making them more effective for a range of tasks where understanding long-term dependencies is crucial (one can easily encounter the problem of vanishing gradients, while trying to train the RNN based NNs proposed by DeepDow [27]). It should be noted that LSTM just reduces the risk of vanishing and exploding gradient but does not fully eliminate it. During the search for the appropriate LSTM-based architectures the author often faced the problem of non-changing training loss, caused by near-zero gradients.

The key innovation of LSTM networks is the introduction of a memory cell c_t, which can maintain its state over time, and three gates (input i_t, forget f_t, and output o_t) that control the flow of information into and out of the cell. The equations governing the LSTM at time t are as follows:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{7}$$
$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{8}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{9}$$
$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{10}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{11}$$
$$h_t = o_t \odot \tanh(c_t) \tag{12}$$

Where:

1. x_t ($dim(x) = d$) is the input at time t.
2. h_{t-1} is the hidden state from the previous time step $t-1$, with h_0 being the initial state. $dim(h_i) = h$.
3. c_{t-1} is the memory cell state from the previous time step $t-1$, with c_0 being the initial state. $dim(c_i) = h$
4. W_{x*} and W_{h*} are the weight matrices for the input and previous hidden state, respectively. $dim(W_{x*}) = (h, d)$, $dim(W_{h*}) = (h, h)$
5. b_* are the bias terms, $dim(b_*) = h$.
6. σ denotes the sigmoid activation function, and $\odot$ denotes element-wise multiplication [24].

This structure allows LSTMs to effectively capture long-term dependencies and decide what information to store, forget, or pass through, making them highly effective for tasks such as language modeling, time series prediction, and sequence generation. The time dependence of an LSTM layer is visualized in the figure below.

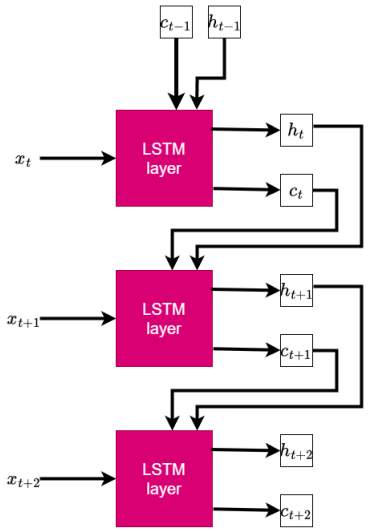

Рис. 10: Time dependence of an LSTM layer.

1.5 Regularizationion

In this section, we will describe what is regularization in general and specific techniques, that will be used for training NNs utilized for portfolio construction.

1.5.1 General idea

The process of learning for NNs (machine learning algorithms in general) consists of two stages:

1. Choosing learnable parameters θ by minimizing the loss function of other subsets of the entire dataset called training.
2. Estimating the generalization performance of the trained NN by estimating its performance (average loss function for example) over a subset of the initial dataset, called testing. Note for correct estimation testing and training subsets must be disjoint.

The difference between model performance on training and testing set is called generalization gap [33]. The bigger the gap, the lower the ability of our model to generalize well on unseen data. If the gap is too big (determined from the context) we say that the model overfit.

Regularization is a process of manipulating NN architecture aimed at decreasing the generalization gap.

The problems of NNs, if compared with other popular machine learning algorithms, is their expressiveness, ability to perfectly fit (or equivalently to memorize) any training set, and to find structure even in garbage data. For example, it was shown that a specific structure convolutional network, used for image classification, was able to memorize 1.2 million size training data with randomly generated labels [34].

In the context of statistical learning, it is a commonly accepted fact that the more complicated the model the higher its probability of overfitting. But in the context of NNs, it is not always

the case. It was shown [19] that under certain conditions increasing NN's model complexity the generalization gap follows the reverse U-shape, being big with small complexity models, big with moderate complexity models, and again small with high complexity problems. Moreover, the ability of NN to generalize is also a nonlinear function of several epochs. An epoch in the context of NN is the number of times the entire dataset is passed forward and backward through the NN. It was shown that after a certain amount of epochs, the generalization gap will increase.

1.5.2 Parameter Loss Penalties

This method aims to limit the magnitude of the learnable parameters θ, thus simplifying the model and helping it to generalize better to unseen data.

Given a loss function $L(F(x_i;\theta))$, where F represents the NN model's prediction for input x_i with parameters θ, regularization can be applied by modifying the loss function to include a penalty term. This results in a new loss function:

$$L_{\text{regularized}}(F(x_i;\theta), y_i) = L(F(x_i;\theta), y_i) + \lambda R(\theta) \tag{13}$$

where:

- $L(F(x_i;\theta), y_i)$ is the original loss function that measures the discrepancy between the NN predictions and the actual target values.
- λ is a regularization parameter that controls the strength of the penalty imposed on the magnitude of the parameters. Choosing an appropriate value for λ is crucial, as too high a value can lead to underfitting, whereas too low a value may not effectively prevent overfitting.
- $R(\theta)$ represents the regularization term. Common choices for $R(\theta)$. The most popular R are $L1$ and $L2$ norms.

It may be optimal to have a specific regularization parameter α for each layer, but the time cost of selecting optimal α is too big to be used in real life. Note it is also commonly used to penalize only a subset of θ.

Although this is a widely used technique in machine learning in general, penalization of the parameter size not always can save the model from overfitting [34].

1.5.3 Early Stopping

It was shown that if data labels have noise, NNs tend to learn correctly labeled data first and then learn the noise [26]. This gives the birth to idea of early stopping: stopping the training process before the NN learns the noise.

Early stopping involves monitoring the model's performance on a validation set at each epoch during training and stopping the training process when the performance on the validation set starts to deteriorate, indicating the beginning of overfitting.

The principle behind early stopping is to use a separate validation dataset that is not used for training the model. The model's performance is evaluated on this validation set at regular intervals during training, typically after each epoch. The process can be summarized as follows:

1. Divide the dataset, used for model training, into two disjoint subsets: training, and validation.
2. During training, monitor the model's performance on the validation set at the end of each epoch (an iteration of training, will be explained later).
3. Continue training as long as the performance on the validation set improves or remains within a certain tolerance.
4. Stop training when the validation performance begins to worsen, indicating that the model is starting to overfit to the training data.
5. Restore the model parameters to the state where the validation performance was at its best.

In the thesis, a modified approach of classical early stopping is used. When the validation performance starts to worsen, we don't immediately stop the training but wait for a certain period, to see if the validation error will improve. The waiting time depends on the specific model used, but on average it is nearly 15.

This technique not only helps in preventing overfitting but also saves computational resources by reducing unnecessary training time.

1.5.4 Dropout

Dropout is a regularization technique that addresses overfitting by temporarily and randomly removing neurons from the neural network during the training process. This method prevents units from co-adapting too much to the data, encouraging the network to learn more generalized representations. The key idea behind dropout is to randomly set a fraction of the input units to 0 at each update during training time, which helps to mimic the effect of training a large number of neural networks with different architectures in parallel. Dropout can be applied to each layer (except output) of an NN of to a subset of layers.

During training, each neuron (including input neurons but typically not the output neurons) has a probability p of being temporarily "dropped out,"meaning it will not contribute to the forward pass and its weight will not be updated during the backward pass. This probability p is a hyperparameter and is set prior to training, with common values ranging from 0.2 to 0.5. The effect of dropout is that the network becomes less sensitive to the specific weights of neurons, leading to a more robust model that is less likely to overfit to the training data.

The dropout procedure can be mathematically represented as follows:

$$r_j^{(l)} \sim \text{Bernoulli}(p) \tag{14}$$

$$\tilde{y}^{(l)} = r^{(l)} * y^{(l)} \tag{15}$$

where $r_j^{(l)}$ is a random variable drawn from a Bernoulli distribution with probability p for each neuron j in layer l, $y^{(l)}$ is the output of neuron j before dropout, and $\tilde{y}^{(l)}$ is the output after applying dropout. The '*' operator denotes element-wise multiplication.

At test time, dropout is not applied; instead, the neuron's output weights are scaled down by a factor equal to the dropout rate p to account for the larger number of active units during testing

compared to training. This ensures that the magnitude of the output through any neuron in testing is roughly the same as it would be on average during training [28].

Dropout has been shown to significantly improve the performance of neural networks on a wide variety of tasks by reducing overfitting, leading to models that generalize better to unseen data.

1.5.5 Data augmentation as a proposition of future research

The best way to decrease the generalization gap is to increase the size of the training dataset (ideally we want NN to be trained on all possible data) [8]. But the amount of data we have in real life is limited. The possible solution is to generate synthetic data with similar properties to the original data. For example in the context of image classification different rotations of an image are used to generate several new images, which can be used during the training process.

The problem is that for financial time series, there is no straightforward approach for data generating. But with the development of generative deep learning shortly, we can use generative models like Generative Adversarial Networks (GANs) or Variational Autoencoders (VAEs), which can generate synthetic time series data that is statistically similar to the original dataset. This method can significantly expand the dataset with new, unseen market scenarios, helping models to learn a wider range of patterns.

2 Optimization

As was shown in Equation 1, an NN depends on a set of learnable parameters θ. The goal of the training stage for an NN is to find the set of optimal θ. Optimality was defined in Problem 3.

The purpose of this chapter is to explain the process of searching for optimal parameters (θ by NN.

2.1 How do the NNs learn?

Initially, researchers viewed the process of training a neural network simply as an optimization problem [4]. From mathematical point of view this very untrivial. General shape of a loss function is very tricky, even on simple NNs. Moreover the majority of loss functions are non-convex. An interesting idea for proof of non-convexity of NNs in general is based on the Universal Approximation Theorem introduced in Section 1.2.3. An NN with certain amount of neurons can approximate any function even non-convex with any precision. But we can't approximate non-convex function with convex, so the NN is non-convex [30]. As a result the training process was seen as the process of finding a solution of very tricky generally non-convex problem, which earned NNs a reputation of being unpredictable and unreliable[4].

While approach focused on finding best solution of Problem 3 is natural, it showed not be useful for application. In reality, we don't train NN to find a global minimum of Problem 3, but to find θ, which will allow an NN to be productive on unseen data [33].

It turned out that neural network optimization is very different from standard mathematical optimization. First, it was shown [4] that finding the global minimum of the dataset used for training leads to increasing of generalization error. Another unobvious fact is that for big NNs (and, as will

be shown later, NNs implemented in the thesis are relatively big) there is no difference between the performance of local minimums in terms of the loss function on unseen data. This facts makes a painstaking search for the global minimum useless and even harmful.

So in the context of neural networks, optimization is not just about the minimization of a cost function, but is a mixture of mathematics and engineering tricks (for example early stopping described before)[32] aimed to find θ, which can be used for dealing with unseen data.

2.1.1 Stochastic gradient descent and minibatch stochastic gradient descent

As discussed in the previous Section 2.1, while minimizing the loss function a local solution is acceptable, so a simple method as gradient descent is applicable for this task.

The problem with vanilla gradient descent is computational complexity. Given a training dataset $X, Y = \{(x_i, y_i)\}_{i=1}^m$ an NN F, a typical loss function over the entire dataset X, Y is computed as an average of losses of individual points:

$$L(F(X;\theta), Y) = \frac{1}{m}\sum_{i=1}^{m} L(F(x_i;\theta), y_i)$$

.

Given that $\nabla_\theta L(F(X;\theta), Y) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(F(x_i;\theta), y_i)$, the computational complexity of each step of gradient descent is $O(m)$ complex. Making the process very ineffective when big datasets are used.

The natural solution to reduce computational complexity is the usage of a subset of X for gradient computation. In extreme cases, only one sample is used. This modification is called SGD.

The algorithm works in the following way[31]. :

1. Shuffle the dataset comprising m observations randomly.
2. Set a value for the learning rate, denoted by η.
3. Initialize the parameters, θ, to start the optimization.
4. For a single data point (x_i, y_i), update θ as follows: $\theta_{next} = \theta_{current} - \eta \cdot \nabla_\theta L(F(x_i; \theta_c urrent), y_i)$.
5. Repeat updating θ till a convergence condition is satisfied.

This reduces the optimization step's computational complexity to constant time one $O(1)$.

It should be noted that $\nabla_\theta L(F(x_i;\theta), y_i)$ is an unbiased estimator of $\nabla_\theta L(F(X;\theta), Y)$:

$$E\left[\nabla_\theta L(F(X;\theta), Y)\right] = \frac{1}{m}\sum_{i=1}^{m} E\left[L(F(x_i;\theta), y_i)\right] = E\left[L(F(x_i;\theta), y_i)\right].$$

The last part of the equation is obtained from classical for NNs analysis assumption that (x_i, y_i) are i.i.d. random variables. However, it should be noted that this assumption is often violated if X represents information about financial asset returns.

The problem of classical SGD is the increased variance of the updates. Suppose we are minimizing $x_1^3 + 2x_2^2$. The function's minimum is point $(0, 0)$. In the figure below the result of default, GD is compared to SGD. The SGD is unstable even after reaching the minimum point.

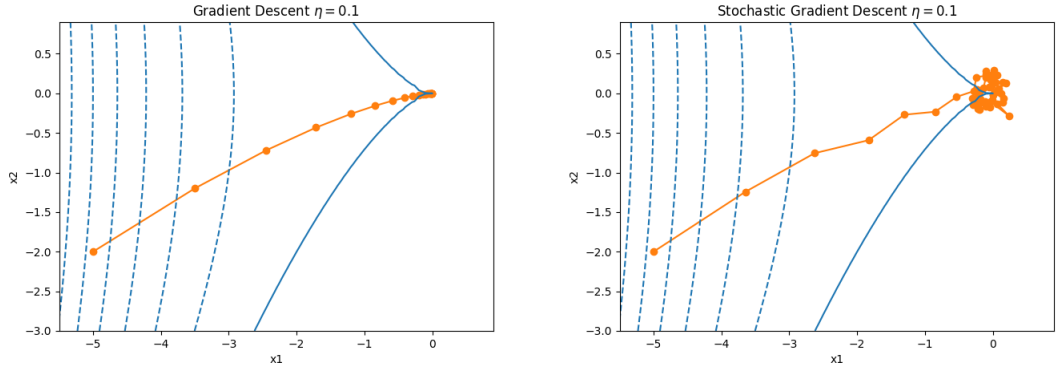

Рис. 11: GD and SGD optimization path.

To make the learning path smoother can make η dynamic, to allow the algorth gradually reduce the step size. The default approaches for making η dynamics are the following [33]:

1. Exponentially decaying η. $\eta(t) = \eta_0 \cdot e^{-\lambda t}$;
2. Polynomially decaying θ: $\eta(t) = \eta_0 \cdot (\beta t + 1)^{-\alpha}$

The effect of decaying η is shown in the figure below.

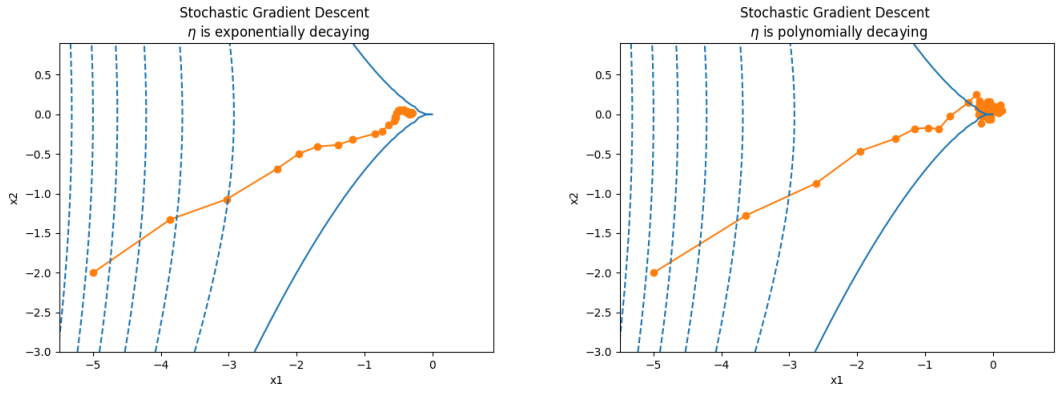

Рис. 12: Decaying η effect.

As we can see, when exponential decay is used, the algorithm does not have time to converge within the given iterations. In both cases the variance is lower less than SGD.

In practice pure SGD is rarely used. The first reason is the high variance depicted in the figures above. But also there is a computational problem. Although for complexity of one step of SGD is $O(1)$, if we want to go through the entire dataset of size m, we are still required to perform m steps of vector-to-vector computations.

The solution is to divide the training dataset into batches (subsets of training dataset) and make the iteration other the batches. A step of a GD with batches can be expressed as follows:

$$\theta_{next} = \theta_{current} - \eta \frac{1}{|B_t|} \sum_{i \in B_t} \nabla_{\theta_{current}} L(F(x_i; \theta), y_i),$$

where $B_t = \{(x_i, y_i) \mid i \in \text{batch indexes}\}$ is a batch. The division by $|B_t|$ is reducing variance. Moreover, the bigger batch size, the less optimization steps we need to go through the entire dataset. Selecting the appropriate batch size for training a NN is delicate. On one hand, a larger batch size can lead to faster computation by fully utilizing the parallelism of modern hardware. On the other hand, it can adversely affect the convergence properties of the optimization algorithm. As put by an expert in the field:

> Training with large minibatches is bad for your health. More importantly, it's bad for your test error. Friends don't let friends use minibatches larger than 32 [14].

During training of different models (see later) it was noted that batch size 32 is appropriate for use, in our context.

There is also a term mini-batch, which causes ambiguity in the terminology. Some people define mini-batch as subset of training set, and define batch as entire training set[12]. But in PyTorch documentation batch term is used for subset of training set. In the thesis the PyTorch approach is followed. Batch is defined as a subset of training set used for one optimization step. Also the term epoch is used. Epoch is the run of a training algorithm over entire dataset. Suppose we have a dataset consisting from 100 points. If we set batch size equal to 10, this will mean that we will have 10 bathces and the one epoch will consist from 10 steps.

2.1.2 Momentum

The problem with GD SGD and batch-based GD is that they work bad when the gradient of the cost functions are disproportional. Suppose we want to minimize

$$0.05x_1^2 + 2x_2^2. \tag{16}$$

The gradient of the function is $0.1x_1 + 4x_2$. The figures below show, the effect of the disproportionality in the gradient. On the

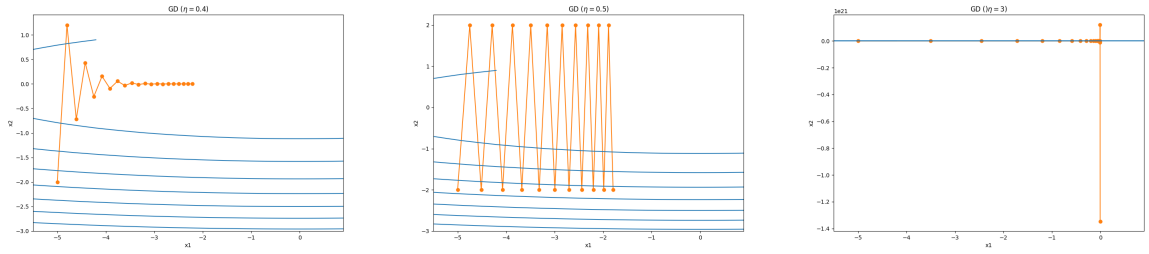

Рис. 13: 20 steps of GD with different η.

As we can see, because of the big effect of x_2 coorinat of the gradient, it is very hard to find the solution for different learning rates η. If η is too small the algorithm is unable to reach the minimum because of the small gradient over x_1, but when the η is becoming, the step size becoming too big over x_2 dimension, making the optimization path very unstable.

The solution to this problem is to replace gradient in GD step with a weighted sum of gradient at the current point and the sum of the past gradients. Namely, replace $\nabla_\theta L(F(X;\theta_t), Y)$ (θ_t is a vector of learnable parameters after t optimization steps) with

$$v_t = \beta v_{t-1} + \nabla_\theta L(F(X;\theta_{t-1}), Y),$$

where $\beta \in (0, 1)$ controls the effect of the previous history. After unwinding the recursion we get the following expression [33]:

$$v_t = \beta^2 v_{t-2} + \beta \nabla_\theta L(F(X;\theta_{t-2}), Y) + \nabla_\theta L(F(X;\theta_{t-1}), Y) = \cdots = \sum_{\tau=0}^{t-1} \beta^\tau \nabla_\theta L(F(X;\theta_\tau), Y).$$

When we replace the GD step's gradient with v_t we obtain the so-called momentum GD. Overall the momentum GD step is following:

$$\begin{aligned} v_t &\leftarrow \beta v_{t-1} + \nabla_\theta L(F(X;\theta_{t-1}), Y), \\ \theta_t &\leftarrow \theta_{t-1} - \eta_t v_t. \end{aligned} \tag{17}$$

An interesting analogy can be found among several authors. The process of gradient descent is compared with an individual making their way down a hillside, always opting for the path that slopes most sharply. Momentum is compared with a sphere that rolls along the same decline. The momentum it gathers serves to even out the ride and propel it forward, reducing zigzag motions [6].

The effect of replacing gradient in GD with v_t for the function is shown in the figure below:

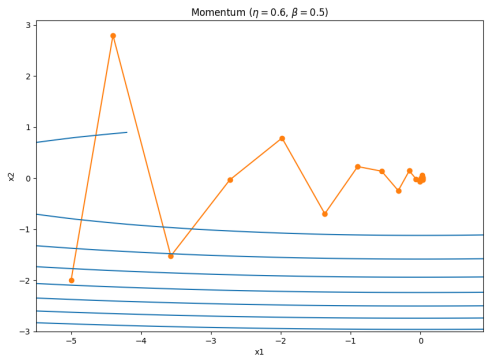

Рис. 14: 20 steps of momentum gradient descent.

As we can see the momentum usage makes the optimization path much more productive.

The momentum idea can be naturally extended on batch gradient descent, we need just to replace the expression for v_t in the following way:

$$v_t = \beta v_{t-1} + \frac{1}{|B_t|} \sum_{i \in B_t} \nabla_\theta L(F(x_i; \theta_{_{current}}, y_i).$$

2.1.3 Root Mean Square Propagation

An alternative way to update default GD to deal with the difficulties of minimizing Cost 16 shown in Figure 2.1.2 is to dynamically update the learning rate η for each coordinate of the gradient.

The most popular algorithm to make the update in this way is RMSProp (Root Mean Square Propagation) introduced by Geoff Hinton in his Neural Networks for Machine Learning course [29] (it was not published, before the introduction during the Coursera video lecture).

The fundamental principle of RMSProp revolves around the modification of the gradient by a running average of its recent magnitude, normalizing the update steps. The mathematical framework of RMSProp is outlined as follows:

$$\begin{aligned} S_t &\leftarrow \gamma S_{t-1} + (1-\gamma)(\nabla_\theta L(F(X; \theta_{t-1}), Y))^2, \\ \theta_{t+1} &\leftarrow \theta_t - \frac{\eta}{\sqrt{S_t + \epsilon}} \odot \nabla_\theta L(F(X; \theta_{t-1}), Y). \end{aligned} \tag{18}$$

Where:

- S_t is a vector that holds the exponential moving average of the squared gradients.
- γ is the decay rate, which is a hyperparameter that determines how quickly the influence of the previous squared gradients decays.
- η is a scalar value representing the learning rate.
- ϵ is a small constant added to enhance numerical stability.
- The square operation applied to the gradient $(\nabla_\theta L(F(X; \theta_{\text{t-1}}), Y)))^2$ and the division by the vector $\sqrt{S_t + \epsilon}$ are performed element-wise.
- $\odot$ denotes the element-wise (Hadamard) product.

RMSProp's mechanism of adaptive learning rate adjustments is particularly beneficial in scenarios characterized by noisy or sparse gradients. The effectiveness of RMSProp for minimizing Cost 16 is shown in the figure below.

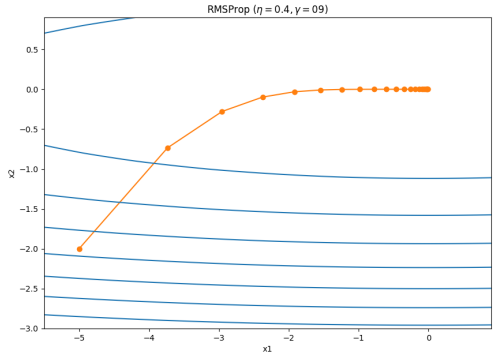

Рис. 15: 20 steps of GD with RMSProp update η.

2.1.4 Two in one, or Adam optimizer

Adam optimizer introduced in 2014 [13] can be seen as a mix of momentum GD ideas described in Section 2.1.2 and RMSProp.

Algorithm 1 Descriptive Caption of Your Algorithm

Require: η (learning rate), β_1, β_2, θ_0, $f(\theta) = L(F(X;\theta), Y)$ (objective)

Initialize: $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\hat{v}_0^{max} \leftarrow 0$
for $t = 1$ to $\ldots$ **do**
    $g_t \leftarrow \nabla f(\theta_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\hat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}$
    $\hat{v}_t \leftarrow \frac{v_t}{(1-\beta_2^t)}$
    $\theta_t \leftarrow \theta_{t-1} - \eta \frac{\hat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)}$
end for
return θ_t

For the reader, the algorithm should look like a natural continuation of the previously described methods. The only curiosities are $\hat{v}_t$ and $\hat{m}_t$. The terms $\frac{m_t}{(1-\beta_1^t)}$ and $\frac{v_t}{(1-\beta_2^t)}$ in the algorithm serve as bias corrections for the first and second-moment estimates, respectively. Initially, m_t and v_t are initialized to 0. Because of this initialization and the update rules that are weighted averages, the estimates are biased towards 0 at the start, especially when t is small.

The bias correction terms, $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$, counteract this bias. They adjust the estimates to account for their initialization. Without these corrections, the estimates would be too low at the beginning of training, which can significantly slow down the convergence, especially for high values of β_1 and β_2.

- For $\hat{m}_t$, the bias-corrected first moment estimate, the division by $(1-\beta_1^t)$ increases the value of the moving average estimate, counteracting the initialization bias. As t increases, β_1^t gets closer to 0, reducing the correction impact, which is appropriate since more gradient information has been included over time.

- Similarly, for $\hat{v}_t$, the bias-corrected second moment estimate, the division by $(1 - \beta_2^t)$ corrects the underestimate of the squared gradients. This is crucial for the adaptive learning rate component, ensuring that it is not too small at the beginning.

By applying these corrections, the algorithm adjusts its steps more accurately, especially during the initial phase, leading to more efficient and reliable convergence.

The difference between Adam perfomance with and without the corrections for minimizing Function 2.1.2 is shown in the figure below.

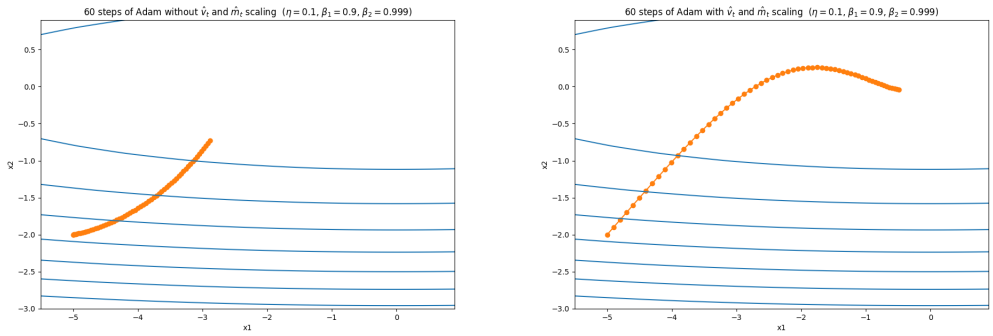

Рис. 16: Perfomance of Adam with and without the corrections.

As we can see Adam without correction can't even reach the optimum $(0, 0)$ in the given number of steps. The path of the Adam with correction looks very nice, especially when compare with GD path from Figure 2.1.2. The values β_1 and β_2 shown on the figure are default for the optimizer and will use them through the thesis.

Although under certain conditions Adam converges poorly [25]. During experiments on building models (to be described later), the algorithm showed itself to be worthy, and therefore only one was used to train all models.

2.2 Gradients computation

As was shown in the previous section gradient of the loss function $\nabla_\theta L(F(X;\theta), y_i)$ is extensively used during the training process.

For automatic computation of gradient in NN extensively used so-called backward propagation. A method used in artificial neural networks to adjust the model's parameters (weights and biases) by propagating the error backward from the output layer to the input layer to minimize the difference between the actual output and the predicted output. This method is based on so-called forward propagation. Foreward propagation is just a fancy way to name the prediction process. When based on an NN F, we are doing a prediction $Y = F(X;\theta)$, we are doing forward propagation.

This section will focus on backpropagation for NN consisting from fully connected layers and for NNs containing recurrent layers.

2.2.1 Backpropagation for a fully connected NN

Suppose we are working with a sequential form NN

$$F(x,\theta) = F_n\left(F_{n-1}\left(\dots F_2\left(F_1(x,\theta_1),\theta_2\right)\dots,\theta_{n-1}\right),\theta_n\right)$$

$$F(x,\theta) = F_n\left(F_{n-1}\left(\dots F_2\left(F_1(x,\theta_1),\theta_2\right)\dots,\theta_{n-1}\right),\theta_n\right)$$

with dense layers $F_i(y,\theta) = \sigma_i(W_i y + b_i)$. So $\theta_i = (W_i, b_i)$. Given the loss function $L(F(X;\theta),Y) = \frac{1}{m}\sum_{i=1}^{m} L(F(x_i;\theta),y_i)$, our goal is to compute $\frac{\partial L(F(x_i;\theta),y_i)}{\partial W_j}$ and $\frac{\partial L(F(x_i;\theta),y_i)}{\partial b_j}$ for each layer of F (since gradient of sum is sum of gradient the total L is easily computed, if we having losses for the individual points).To ease the notation let's replace $L(F(x_i;\theta),y_i)$ with $l(\theta)$ ((x_i,y_i) is a data point, so is fixed).

A little reminder that each layer F_i is a vector to vector function. Elements of the output vectors are called neurons. Let's define the neuron output a layer i as activation a_i. Since we are discussing sequential architecture for each activation the following equation holds true. $a_i = \sigma(W^i a^{i-1} + b^i)$. To further lets introduce $z_i = W^i a^{i-1} + b^i$, so $a^i = \sigma(z^i)$.

Following notation taken from Nielsen [20] let's define the backpropagation error of neuron i of layer $j\delta_i^j = \frac{\partial l}{\partial z_i^j}$. Intuitively the error δ_i^j measures the sensitivity of the loss to slight perturbation into input to the neuron i of layer j.

Let O be the index of the output layer, we can show, that

$$\delta^O = \nabla_a l \odot \sigma'(z_j^O),$$

where $\sigma'(z_j^O) = \nabla_{z^j} a^j$. The equation can be easily be proved using chain rule $\delta_j^O = \sum_k \frac{\partial l}{\partial a_k^O}\frac{\partial a_k^O}{\partial z_j^O}$, and the fact that $\frac{\partial a_k^O}{\partial z_j^O} = 0 \quad \text{for} \quad k \neq j$.

We can also prove that

$$\delta^i = \left((W^{i+1})^T \delta^{i+1}\right) \odot \sigma'(z^i). \tag{19}$$

Using chain rule, we can obtain:

$$\delta_j^i = \frac{\partial l}{\partial z_j^i} = \sum_k \frac{\partial l}{\partial z_k^{i+1}}\frac{\partial z_k^{i+1}}{\partial z_j^i}.$$

Now, we consider the weighted input to a neuron in the next layer $i+1$:

$$z_k^{i+1} = \sum_j w_{kj}^{i+1} a_j^i + b_k^{i+1},$$

which, when applying the activation function σ, becomes:

$$a_k^{i+1} = \sigma(z_k^{i+1}) = \sigma\left(\sum_j w_{kj}^{i+1}\sigma(z_j^i) + b_k^{i+1}\right).$$

Differentiating z_k^{i+1} with respect to z_j^i, we obtain:

$$\frac{\partial z_k^{i+1}}{\partial z_j^i} = w_{kj}^{i+1}\sigma'(z_j^i).$$

Based on this fact, we have:

$$\delta_j^i = \sum_k \frac{\partial l}{\partial z_k^{i+1}} w_{kj}^{i+1}\sigma'(z_j^i).$$

By the definition of matrix multiplication, this can be rewritten as:

$$\delta^i = (w^{i+1})^T\delta^{i+1} \odot \sigma'(z^i).$$

Thus, we have shown the matrix form of backpropagation for the error term δ^i.

For each neuron j of layer i the derivative with respect to bias can be expressed as follows:

$$\frac{\partial l}{\partial b_j^i} = \delta_j^i. \tag{20}$$

This equation can be proved in the following way:

$$\frac{\partial l}{\partial b_j^i} = \sum_k \frac{\partial l}{\partial z_k^i}\frac{\partial z_k^i}{\partial b_j^i} = \frac{\partial l}{\partial z_j^i}\frac{\partial z_j^i}{\partial b_j^i} = \delta_j^i \cdot 1$$

In the same way, we can prove, that

$$\frac{\partial l}{\partial w_{jk}^i} = a_k^{i-1}\delta_j^i \tag{21}$$

2.3 How to solve a convex optimization problem in a differentiable way?

Список литературы

[1] torch.nn.RNN – pytorch 1.x documentation. `https://pytorch.org/docs/stable/generated/torch.nn.RNN.html`, 2024.

[2] C. Banerjee, T. Mukherjee, and E. Pasiliao Jr. An empirical study on generalizations of the relu activation function. In *Proceedings of the 2019 ACM Southeast Conference*, pages 164–167, 2019.

[3] bayerj (https://stats.stackexchange.com/users/2860/bayerj). How does lstm prevent the vanishing gradient problem? Cross Validated. URL:https://stats.stackexchange.com/q/263956 (version: 2017-12-30).

[4] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pages 192–204. PMLR, 2015.

Algorithm 2 Backpropagation algorithm for training a neural network

Input a set of training examples
for each training example x **do**
    Set the corresponding input activation $a^i(x)$, and perform the following steps:
    Feedforward:
    for $i = 2, 3, \ldots, L$ **do**
        $z^i(x) = w^i a^{i-1}(x) + b^i$ and $a^i(x) = \sigma(z^i(x))$
    end for
    Output error $\delta^L(x)$:
    Compute the vector $\delta^L(x) = \nabla_a C \odot \sigma'(z^L(x))$
    Backpropagate the error:
    for $i = L-1, L-2, \ldots, 2$ **do**
        $\delta^i(x) = ((w^{i+1})^T \delta^{i+1}(x)) \odot \sigma'(z^i(x))$
    end for
    Gradient descent:
    for $i = L, L-1, \ldots, 2$ **do**
        Update the weights according to the rule $w^i \leftarrow w^i - \frac{\eta}{m} \sum_x \delta^i(x)(a^{i-1}(x))^T$
        Update the biases according to the rule $b^i \leftarrow b^i - \frac{\eta}{m} \sum_x \delta^i(x)$
    end for
end for

[5] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[6] G. Goh. Why momentum really works. *Distill*, April 2017.

[7] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

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

[9] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In *International conference on machine learning*, pages 1319–1327. PMLR, 2013.

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

[11] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[12] T. (https://stats.stackexchange.com/users/56984/tim). What are the differences between 39;epoch39;, 39;batch39;, and 39;minibatch39;? Cross Validated. URL:https://stats.stackexchange.com/q/117919 (version: 2021-02-17).

[13] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] Y. LeCun. Training with large minibatches is bad for your health. more importantly, it's bad for your test error. friends don't let friends use minibatches larger than 32. Twitter, 2018. Tweet.

[15] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[16] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[17] L. Ma, Z. Lu, and H. Li. Learning to answer questions from image using convolutional neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[18] A. L. Maas, A. Y. Hannun, A. Y. Ng, et al. Rectifier nonlinearities improve neural network, acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.

[19] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, 2021.

[20] M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[21] T. Nitta. Solving the xor problem and the detection of symmetry using a single complex-valued neuron. *Neural Networks*, 16(8):1101–1105, 2003.

[22] P. Patrinos. Optimization lecture notes. `https://www.kuleuven.be/english`, 2023. Lecture Notes for Course on Optimization.

[23] D. S. Poskitt. A note on autoregressive modeling. *Econometric Theory*, 10(5):884–899, 1994.

[24] PyTorch. torch.nn.lstm - pytorch documentation. `https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html`, 2023. Accessed: 2024-03-25.

[25] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[26] D. Rolnick, A. Veit, S. Belongie, and N. Shavit. Deep learning is robust to massive label noise. *arXiv preprint arXiv:1705.10694*, 2017.

[27] J. Siebert, J. Groß, and C. Schroth. A systematic review of python packages for time series analysis. *arXiv preprint arXiv:2104.07406*, 2021.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[29] T. Tieleman. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26, 2012.

[30] C. University. Lecture 7: Advanced topics in machine learning and data science. `https://www.cs.cornell.edu/courses/cs6787/2017fa/Lecture7.pdf`, 2017. Accessed: 2024-04-02.

[31] C. University. Stochastic gradient descent. `https://optimization.cbe.cornell.edu/index.php?title=Stochastic_gradient_descent`, 2024. Accessed: 2024-04-02.

[32] R. Zadeh. The hard thing about deep learning. https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/, 11 2016. Accessed: 2024-04-02.

[33] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning, 2021. Online; accessed February 26, 2024.

[34] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.