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 adopted 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 [23]. 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 is just function $F : I \rightarrow W$, 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 [18], 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 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. Controlling 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.

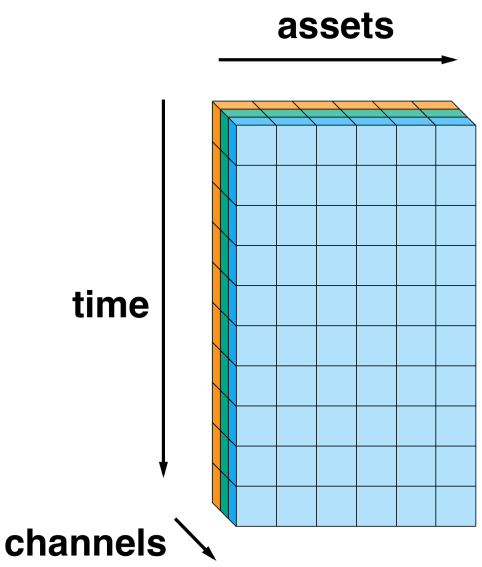

Рис. 1: DeepDow financial data representation [23].

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.

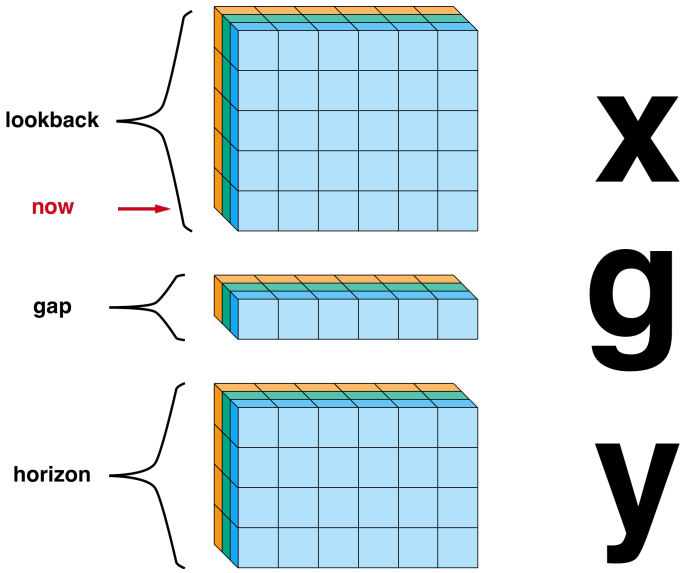

Рис. 2: Time through the eyes of DeepDow [23].

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.

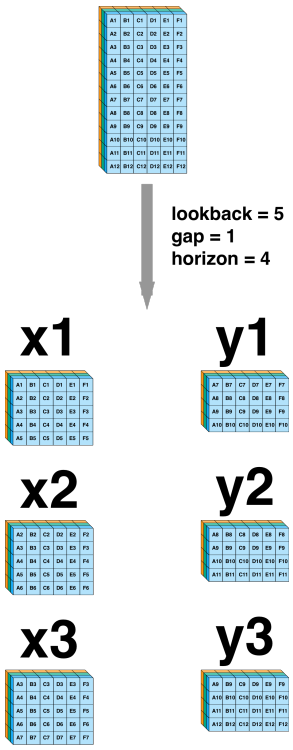

Рис. 3: DeepDow data example [23].

an NN is essentially a graph of embedded functions, depending on learnable parameters

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

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 following way.

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

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

So in the context of our 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.

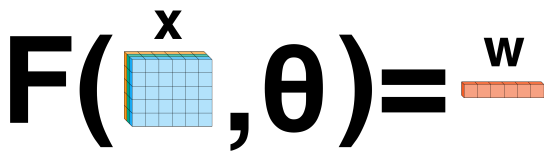

Рис. 4: From returns to portfolio weights based on learned θ[23].

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 $\{(x_i, y_i)\}_{i=1}^{m}$. In this case, the goal of the learning phase of our NN F is to solve the following optimization problem:

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

The exact loss function and the optimization process 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.
2. The loss function is computed as a scalar function of the vector of portfolio returns r.

The whole process is visualized in the figure below.

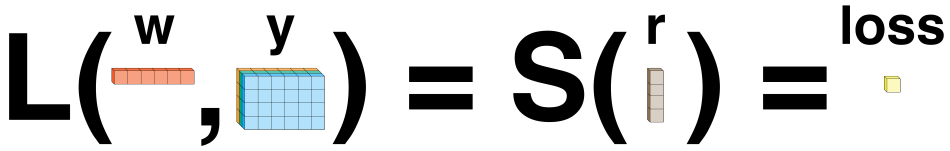

Рис. 5: Loss function computation process [23].

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:

$$F_i(y, \theta) = \sigma_i(W_i y + b_i), \tag{2}$$

where W_i is layer weights, b_i is layer bias and σ is an activation function [27]. 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 1 with $F_i(y, \theta) = W_i y + b_i$ for all i (σ_i, 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 [17].

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 [5]. 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 [4] 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 [12].

Another activation function capable of mitigating the dying ReLU problem is the Maxout activation function [6]. 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.

1.3 Dense NN as universal approximator

Despite its relative simplicity dense NNs (NNs from Equation 1, 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 $\mathbf{b}_2$ 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 $g : R^n \to R^m$ and for any $\epsilon > 0$, there exists a configuration of σ, W_1, W_2, $\mathbf{b}_1$, and $\mathbf{b}_2$ such that the neural network F can approximate g such that the maximum error across all outputs and for all inputs in a compact subset of R^n is less than ϵ if hidden layer size will be big enough (amount of rows in W_1 will be big enough)[7].

It was proven that the theorem works for both ReLu and Leak ReLu [11]. Since Maxout can imitate ReLu, this theorem also applicable for 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 in a respective way. 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

1.4 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.

1.4.1 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 [27]. 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 [28].

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 [15] 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.

1.4.2 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:

$$L_{\text{regularized}}(F(x_i;\theta)) = L(F(x_i;\theta)) + \lambda R(\theta) \tag{3}$$

where:

- $L(F(x_i;\theta))$ 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 [28].

1.4.3 Early Stopping

It was shown that if data labels have noise, NNs tend to learn correctly labeled data first and then learn the noise [20]. 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 into three subsets: training, validation, and testing.
2. During training, monitor the model's performance on the validation set at the end of each epoch.
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. Optionally, restore the model parameters to the state where the validation performance was at its best.

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

1.4.4 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:

$$r_j^{(l)} \sim \text{Bernoulli}(p) \tag{4}$$

$$\tilde{y}^{(l)} = r^{(l)} * y^{(l)} \tag{5}$$

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 [24].

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.

1.4.5 Data augmentation as a proposition of future research

The best way to increase the generalization gap is to increase the size of the training dataset (ideally we want NN to be trained of all possible data) [5]. 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 Dealing with the noise of covariance function

Estimating the covariance matrix of asset returns is a non-trivial task. Direct usage of the empirical covariance matrix is not always appropriate. This section will discuss the problems of an empirical covariance matrix, show the machinery for finding signals in the covariance matrix, discuss how this can be applied in neural networks, and show methods that will help to estimate the covariance matrix.

2.1 Problems of empirical covariance matrix

The empirical covariance matrix of X with $dim(X) = (m, n)$ is a maximum likelihood estimator of true covariance of X if rows of X (observations) are i.i.d. random variables and m is significantly bigger than n. Both of these conditions are often violated in the context of financial returns data. This leads to problems when empirical covariance and empirical mean are used

in Markowitz-style portfolio optimization. Michaud (1989) [14] called this portfolio optimizer estimation-error maximizer.

2.2 Macrenko-Pastur distribution

2.2.1 The distribution

Suppose we have a matrix X is a (m, n) matrix of i.i.d. random variables with zero mean and finite variance σ^2. Then according to Marchenko–Pastur theorem, if $m \to \infty$, $n \to \infty$ and $\frac{m}{n} \in (1, \infty)$, the density of eigenvalues of the empirical covariance matrix $C = \frac{X^T X}{m}$ (note that the eigenvalues are random variables in this case) converges to a specific density, known as the Marchenko–Pastur distribution:

$$f_{MP}(\lambda, \sigma) = \begin{cases} \frac{m}{n} \frac{\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}}{2\pi\lambda\sigma^2} & \text{if } \lambda \in [\lambda_-, \lambda_+], \\ 0 & \text{if } \lambda \notin [\lambda_-, \lambda_+], \end{cases}$$

where $\lambda_+ = max(E(\lambda)) = \sigma^2(1 + \sqrt{\frac{m}{n}})^2$ and $\lambda_- = min(E(\lambda)) = \sigma^2(1 - \sqrt{\frac{m}{n}})^2$ [30].

In the image below, the applicability of the theorem is shown for a X consisting of *i.i.d.* random normal variables with $\sigma^2 = 1$, $m = 1000$, $n = 1000$. The prediction of a Marcenko-Pastur density can be compared with empirical density obtained using kernel density estimator [21] with standard normal kernel function and manually selected bandwidth.

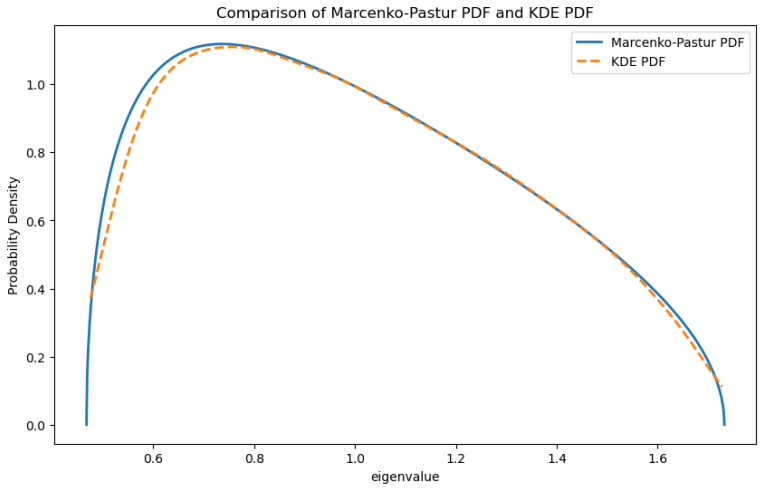

Рис. 6: Marcenko-Pastur density vs empirical density.

The importance of eigenvalues of a covariance matrix is explained by the fact that an eigenvalue controls the data variance across the relevant eigenvector [16]. Marchenko–Pastur distribution gives us the machinery for differentiation between eigenvalues (variance) related to noise in the data and eigenvalues related to the signal. We can associate $\lambda \in [\lambda_-, \lambda_+]$ with noise and $\lambda \in [0, \lambda_-] \vee [\lambda_+,)$ with signal [3].

In practice, we associate $\lambda \in [0, \lambda_+]$ with noise, and $\lambda > \lambda_+$ with signal, because $\lambda \in [0, \lambda_-]$ contains almost no information about the variability of the data.

2.2.2 How to determine the amount of relevant factors from the covariance matrix?

The mentioned differentiation between signal and noise allows us to define covariance matrix factors. Factors are empirical eigenvalues of a covariance matrix that exceed λ_+ and are often interpreted as signals or true underlying factors that substantially impact the data structure.

The idea of factors allows us to apply Marcenko-Pastur distribution to not fully random matrixes.

Suppose we have data matrix X and $dim(X) = (15000, 1000)$. Suppose it is an almost fully random matrix and its covariance is a weighted sum

of signal and noise. And noise weight is 0.999. By noise, we mean a matrix N, with $dim(S) = (1000, 1000)$, of i.i.d. standard normal random variables. And

Let $W \in R^{1000\times 50}$ be a matrix, where each element W_{ij} is drawn from a standard normal distribution $\mathcal{N}(0, 1)$ and 50 represents the number of structural factors in the data X. The signal part of X's covariance matrix is constructed as follows:

$$S = WW^T + diag(u)$$

Here, WW^T creates a symmetric, semi-positive definite matrix reflecting the covariance introduced by the signals in S. To ensure the covariance is of full rank and invertible, a diagonal matrix $diag(u)$ with entries u_i drawn from a uniform distribution $\mathcal{U}(0, 1)$ is added, adjusting the variances on the diagonal. In the end, we obtain a signal matrix, containing 50 factors.

So the covariance of X can be expressed in the following way $cov(X) = N + 0.999S$. To use Marcenko-Pastur for defining the amount of structural factors in $cov(X)$, we need λ_+. But, as was shown before, λ_+ depends on unknown σ, so we can't calculate it straightforwardly. The solution is to find σ, solving the following optimization problem:

$$\sigma = \arg\min_{\sigma} \int_0^{\infty} (f_{MP}(\lambda, \sigma) - g(\lambda))^2 \, d\lambda \tag{6}$$

,

where g is an empirical density function, obtained using kernel density estimator.

The result of fitting the Marcenko-Patstur distribution is shown below. We consider everything to the right of the green line to be non-random eigenvalues-factors.

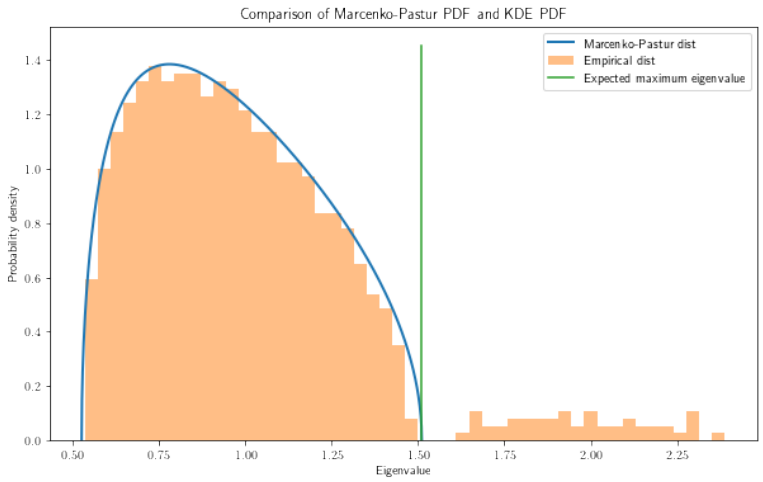

Рис. 7: Counting data factors using Marcenko-Pastur.

The described methodology allowed us to correctly count all the factors (eigenvalues to the right of the green line). The described experiment was repeated 100 times and each time amount of factors was correctly counted.

2.2.3 Applicability of Marcenko-Pastur to examining the performance of NNs and proposition of future research

As was shown before NNs in the presented research play the role of the transformers of the initial information about asset returns into Markowitz-style beliefs about future asset expected returns and expected covariance matrix, which are used as an input to an optimization (CVXPYLAYER) layer. So in the end a trained NN can provide us with the covariance matrix of the assets (for a certain period) based on certain input X with $dim(X) = (m, n)$.

The idea of factors, presented previously, can be naturally applied to the output covariance matrix of the NN. Suppose, using Marcenko-Pastur machinery, we defined that the amount of factors of $cov(X)$ is i. Then suppose that we used X as an input into the NN, after which we obtained the NN view on the future covariance matrix cov_{NN}. Applying the Marcenko-Pastur distribution to cov_{NN} we obtained the amount of factors j. If on average on validation set $i > j$, we can conclude that the NN learned a more structured representation from the data.

The problem with the implementation of this methodology are assumptions of Marcenko-Pastur distribtuion. As mentioned before Marcenko-Pastur assumes that $m \to \infty$ and $n \to \infty$. Although in practice we are working with finite m and n, they are still supposed to be relatively high.

Suppose we are working in the context of the experiment described in Section 2.2.2, but we have different fractions of noise in the covariance matrix: 0.999, 0.99, and 0.9. We are interested in the amount of m and n required to successfully find all the factors of generated *cov*. Specifically, we are interested in finding the values of m and n that minimize the expression $m \times n$, because the size of the input to the neural network in our research is represented by the product $m \times n$, and find all the factors (50 in the context of the experiment). The figure below shows the result of the experiments for different m and n and contamination fraction.

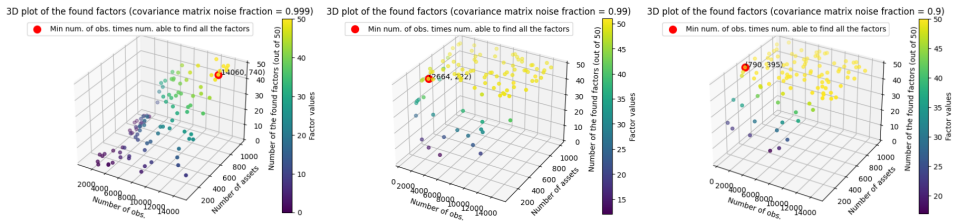

Рис. 8: (m, n) required by Marcenko-Pastur.

From the figure, we can conclude the following. If the covariance matrix contamination fraction is 0.999, we will need approximately $m \times n = 14060 \times 740 = 10404400$ inputs to our NN. If the covariance matrix contamination fraction is 0.99, we will need approximately $m \times n = 2664 \times 222 = 591408$ inputs to our NN. If the covariance matrix contamination fraction is 0.99, we will need at approximately $m \times n = 790 \times 395 = 312050$ inputs to our NN. Considering that NNs are in general computationally intensive [25], this creates a significant block for further research. The author attempted to train the RNN architecture described previously on synthetic data, with the neural network's input being a $(500, 250)$ matrix and utilizing a training dataset comprising $15,000$ samples. Note, according to the figure above, both m and n dimensions are still too small, even when the contamination fraction reaches 0.9. Despite leveraging Google Colab Pro, which provided access to substantial computational resources, including a high-memory environment with approximately 25 GB of RAM and a powerful GPU such as the NVIDIA

Tesla P100, the training process faced significant challenges. The computational requirements exceeded the capabilities provided by these enhanced resources. As a result, the author was unable to complete the training of the neural network within the 24-hour time limit imposed by Google Colab Pro's session duration. The author also tried to use AWS SageMaker but didn't manage to obtain the results.

The further problem creates the data setup per se. Due to the curse of dimensionality [26] the training data sample size requirements are significantly increasing with increasing of the NN's input dimensions. Also, if we assume that the contamination fraction of the covariance matrix is 0.9, as based on the experiment we select input to our neural network being a matrix of 395 assets and 790 observations, we will need at least hourly data about the performance of the assets to be able to form a training dataset. Considering that in open source this data is unavailable in general, further research will require significant data-acquiring costs.

Because of the computational limitations and cost requirements to obtain the needed hourly price data for a significant amount of assets, it was decided to stop the research in this direction. However, the author still thinks that the topic of utilizing Marcenko-Pastur for testing the ability of neural networks to find hidden structures in the data is very promising.

2.3 Estimating covariance matrix

2.3.1 Shrinkage

The concept of shrinkage in the context of covariance matrix estimation involves estimating the true covariance matrix as a weighted average between the empirical covariance matrix, which can be noisy, especially in high-dimensional settings, and a structural (or target) matrix, which represents a simpler structure or prior belief about the relationships between variables.

$$\hat{\Sigma}_{\text{shrink}} = \alpha\hat{\Sigma}_{\text{empirical}} + (1 - \alpha)S$$

where $\hat{\Sigma}_{\text{shrink}}$ is the shrunk estimator of the covariance matrix, $\hat{\Sigma}_{\text{empirical}}$ is the empirical covariance matrix, S is the structural matrix, and α is a shrinkage constant in the interval $0, 1$.

The selection of the shrinkage intensity parameter α is crucial for balancing the trade-off between the empirical covariance matrix and the target matrix

S. The optimal value of α minimizes the estimation error of the shrunk covariance matrix. In practice, α is determined through cross-validation, analytical methods, or by relying on estimators that assess the variability and structure of the data. One common approach, proposed by Ledoit and Wolf, involves estimating α as:

$$\alpha = \frac{\sum_{i \neq j} (\hat{\sigma}_{ij} - \rho \cdot \hat{\sigma}_i \cdot \hat{\sigma}_j)^2}{\sum_{i,j} (\hat{\sigma}_{ij} - S_{ij})^2} \tag{7}$$

where $\hat{\sigma}_{ij}$ represents the elements of the empirical covariance matrix $\hat{\Sigma}_{\text{empirical}}$, $\hat{\sigma}_i$ and $\hat{\sigma}_j$ are the standard deviations of the ith and jth variables, respectively, ρ is the average correlation coefficient, and S_{ij} are the elements of the target matrix S. This formula adjusts α based on the discrepancy between the empirical correlations and those implied by the target structure, aiming to minimize the overall prediction error of the shrunk covariance matrix.

The optimal α reflects the proportion of the total variance in the empirical covariance matrix that is attributable to noise as opposed to true signal. A higher α indicates greater reliance on the target matrix S, suitable when the empirical covariance matrix is expected to be highly noisy or the sample size is small relative to the number of variables.

The are several ways to create S. Below you can find the two most common. The edit-Wolf covariance estimator is using the following form of S:

$$S = \lambda I \tag{8}$$

where λ is the average variance of all the variables, calculated as the mean of the diagonal elements of the empirical covariance matrix $\hat{\Sigma}_{\text{empirical}}$, and I is the identity matrix of appropriate dimension [10].

Another widely recognized method for constructing S is the constant correlation model. This model posits that all pairwise correlations between variables are identical. The rationale behind this model is to simplify the covariance matrix by assuming a uniform correlation, denoted by ρ, among all pairs of variables, while retaining the individual variances from the empirical data. The estimation of ρ is straightforward: it is the average of all the sample correlations. This common correlation, along with the vector of sample variances,

forms the basis for the shrinkage target matrix S in the constant correlation model:

$$S_{ij} = \begin{cases} \sigma_i^2 & \text{if } i = j, \\ \rho \cdot \sigma_i \cdot \sigma_j & \text{if } i \neq j, \end{cases}$$

where S_{ij} represents the element of S for the ith row and jth column, σ_i^2 is the variance of the ith variable, and σ_i and σ_j are the standard deviations of the ith and jth variables, respectively. The structural matrix S thus incorporates a constant correlation ρ across all variable pairs, except for the diagonal elements that reflect the variables' variances [9].

In the thesis approach, described in Equation 8, is used.

2.3.2 Minimum-Covariance Determinant estimator and why it is not applicable for asset returns data

The Minimum-Covariance Determinant estimator is designed to robustly estimate a dataset's covariance matrix by focusing on a subset of observations that excludes outliers. The Minimum-Covariance Determinant method calculates an empirical covariance matrix from this subset by selecting a proportion h of the dataset deemed as "good"observations. This matrix is then rescaled in a consistency step to adjust for the selection bias, and a reweighting step may follow, where observations are weighted by their Mahalanobis distance to improve the estimate [22].

Although widely used in practice for dealing with financial time series, this method is not applicable for working with asset returns because of the following:

1. Financial data often contain important extreme movements that are not outliers in the traditional sense but are crucial for risk assessment.
2. Minimum-Covariance Determinant estimator assumes that the data is i.i.d. As mentioned before this is not the case for our data.
3. Minimum-Covariance Determinant estimator assumes that the data is generated by elliptical distribution. While the non-normality of stock returns is widely accepted [8]

2.3.3 Denoising covariance matrix using Marcenko-Pastur

In this subsection denoising algorithm for the correlation matrix will be shown, but, given that the transformation between covariance and correlation matrix is trivial, this can be seen as a covariance matrix denoising algorithm.

The core idea of denoising a correlation matrix is simple: replacing noise-related eigenvalues in eigenvector decomposition with constants

Specifically, given a descending ordered set of eigenvalues $\{\lambda_n\}_{n=1}^{N}$, the procedure identifies a cutoff point i such that $\lambda_i > \lambda^+$ and $\lambda_{i+1} \leq \lambda^+$. This demarcation (for example greenline on Figure 9) distinguishes between signal-bearing eigenvalues and those attributed to noise as described in Section (as described in Section 2.2.1).

Subsequently, for eigenvalues positioned beyond the ith (for $j = i + 1, \ldots, N$), each λ_j is set to the mean of the eigenvalues from $i + 1$ to N, specifically, $\lambda_j = \frac{1}{N-i} \sum_{k=i+1}^{N} \lambda_k$. This adjustment evenly distributes the aggregate variance of noise across all corresponding eigenvalues, thus preserving the matrix's trace while mitigating noise.

To construct the denoised correlation matrix, given the eigenvector decomposition $VW = W\Lambda$, where W and Λ represent the eigenvector and diagonal eigenvalue matrices respectively, a revised diagonal matrix Λ_e incorporating the adjusted eigenvalues is formed. The denoised matrix $C_{e1} = W\Lambda_e W'$ is then calculated, where W' signifies the transposition of W.

The final step involves normalizing C_{e1} to ensure unity along the main diagonal of the resultant matrix C_1, reaffirming its correlation matrix properties. This normalization is achieved by rescaling C_{e1} through the division by the square roots of the diagonal elements' product, thereby adjusting the matrix to exhibit unit variance diagonally [3].

The influence of denoising on eigenvalues is shown in the figure below. As we can see all randomness-related eigenvalues are constant in the denoised matrix.

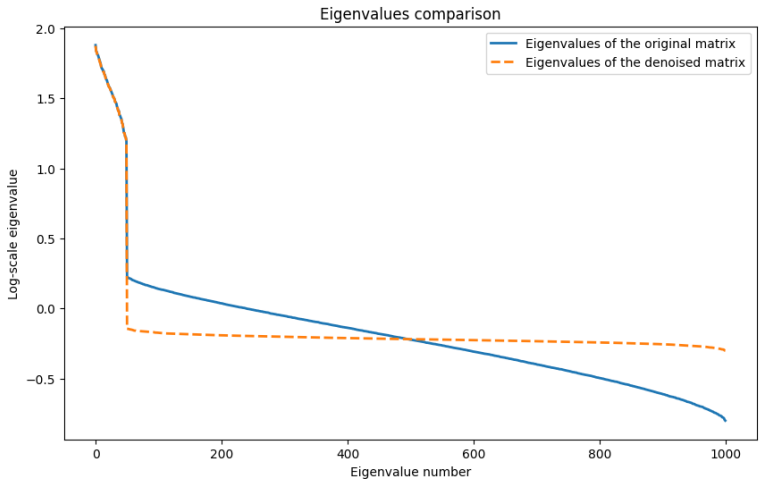

Рис. 9: Eigenvalues before and after denoising.

2.4 Comparing shrinkage efficiency with denoising for constructing minimum variance portfolio

This section will compare the performance of usage shrinkage and denoising techniques when selecting a minimum variance portfolio.

Suppose we have a data X. X is generated from $N(\mu, \Sigma)$. Where (μ, Σ) are true characteristics of the data formed in the following way. Suppose our investment universe consists of 5 sectors with 125 assets in each sector. The covariance matrix encapsulates the intricate web of risk and return interdependencies among assets, systematically arranged to reflect sector-based market dynamics. Its structure is informed by predefined intra-sector correlations set at 0.5, symbolizing a moderate level of return similarity within each sector. The variances, occupying the matrix's diagonal, are uniformly sampled between 5% and 20%, illustrating the spectrum of risk inherent to different assets. This serves as a simplified yet insightful representation of market behavior, echoing the diversification and risk-return profiles characteristic of segments within broader financial indices such as the S&P 500.

The vector of expected returns represents the anticipated profitability for each asset, calculated under the assumption that each asset's expected return is proportional to its risk, adhering to the efficient market hypothesis. Each element is generated from a normal distribution, where the mean and standard deviation are equal to the asset's own standard deviation derived

from the covariance matrix, underscoring the principle that, in an efficient market, all securities are expected to offer the same Sharpe ratio.

The goal is to minimize the portfolio variance subject to certain constraints. The optimization problem can be formulated as follows:

$$\begin{aligned} \text{minimize} \quad & w^T \Sigma w \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i = 1, \\ & w_i \geq 0, \quad \forall i = 1, \ldots, n, \end{aligned}$$

where Σ is an estimator of the covariance matrix of assets.

We will solve this problem in five setups: using true Σ, using shrunk and denoised estimator of Σ, using shrunk not-denoised estimator of Σ, using not-shrunk, not-denoised estimator of Σ and not-shrunk, denoised. The weights obtained using the true value of the covariance matrix will be used as the benchmark. We will compare the $L2$ distance between the benchmark and the weights obtained using other estimators.

The described experiment was repeated 20 times and the following average distance metrics were obtained.

Таблица 1: average $L2$ distance to the optimal portfolio of the portfolios, which are based on different estimators

	Shrunk	**Not-shrunk**
Denoised (after the shrinkage)	0.0515	0.0273
Not-Denoised	0.0501	0.0414

As we can see denoising is more effective than shrinkage, specifically applied to not-shrunk data. Denoising after shrinkage is not very effective. The reason is that shrinkage makes signal eigenvalues wicker, making differentiation between signal eigenvalues and noise eigenvalues harder.

Although denoising is more effective than shrinkage, the problem is that denoising requires solving an optimization problem (Equation 6) to find σ of the data-generating process, which is computationally ineffective, restricting its applicability in the context of NN. So in our covariance matrix estimation layer *CovLayer* we are only using shrinkage techniques.

3 Architectures used

This section will describe the architectures used for portfolio composition.

3.1 How the networks were trained

After selecting the type of NNs, that we want to train, the process of training NNs consists of several steps:

1. Data preparation.
2. Setting the selected NN architecture setup. Training the NN.

All our networks were trained on the daily returns data of the selected assets. The training period starts on January 3, 2007, and ends on December 15, 2020.

To set up NN architecture, we are supposed to use hyperparameters. Hyperparameters are non-trainable parameters of an NN, which are selected before the training process starts such as type of activation function, amount of layers, optimizer, regularization techniques, and so on.

To select hyperparameters, the hold-out validation method was utilized. Specifically, the original training dataset was divided into a training subset and a validation subset, with proportions of 80% for training and 20% for validation [19]. For each proposed set of hyperparameters, a model was trained on the training data. The performance of each trained model was then recorded on the validation set. The set of hyperparameters was selected based on achieving the best performance on the validation set. To mitigate overfitting to the validation set, the division into 80% training and 20% validation was executed in a random manner, ensuring that each model was validated against non-identical datasets.

There is a popular alternative to the hold-out validation method, known as *k-fold cross-validation*. The core idea of this method is to divide the original training dataset into k disjoint subsets (or "folds"). The model is then trained and validated k times, with each iteration using a different fold as the validation set and the remaining $k-1$ folds combined as the training set. The model's performance is assessed by taking the average of the performance metrics across all k iterations. This approach ensures

that every data point is used for both training and validation exactly once, providing a comprehensive evaluation of the model's performance.

However, training neural networks is time-consuming, and the k-fold cross-validation process can significantly increase the computational burden because it requires training k separate models. Due to the time demands of both neural network training and the cross-validation process itself, it was decided to utilize the hold-out validation method.

Each of the NN was trained using a modified early-stopping regularization technique, described in Section 1.4.3. We don't stop the training at the moment the validation error starts to worsen but wait for 15 epochs to see if the validation performance will enhance. In case there is no positive effect from waiting, we stop training and restore the model where validation error is the best.

3.2 Layers

In this subsection, we will describe the building blocks of the implemented NNs

3.2.1 Markowitz layer

This is the most important block in our NNs. It encapsulates the essence of Markowitz portfolio optimization within a neural network paradigm, integrating this optimization directly into the neural network architecture.

This layer accepts as inputs expected returns, the square root of the covariance matrix of asset returns, a parameter to control the risk-return tradeoff, and a regularization parameter to enforce L2 regularization on the portfolio weights. The output of this layer is a tensor of optimal portfolio weights, selected by solving an optimization problem described below.

Unlike traditional neural network layers, the Markowitz layer does not have its own parameters through backpropagation. It utilizes the CVXPYLAYERS package to construct portfolio weights based on the provided inputs, solving an optimization problem. This optimization problem is formulated to maximize the expected portfolio return, adjusted for risk and regularization, as follows:

$$\max_{\mathbf{w}} \quad \mathbf{w}^\top \mathbf{r} - \lambda \mathbf{w}^\top \sqrt{\Sigma} \mathbf{w} - \alpha ||\mathbf{w}||^2 \tag{9}$$

subject to $\sum_i w_i = 1$, and $-1 \leq w_i \leq 1$, where $\mathbf{w}$ denotes the portfolio weights, $\mathbf{r}$ the expected returns, Σ the covariance matrix, λ the risk aversion coefficient, and α the regularization parameter. It should be noted that α and λ are learnable parameters, that are found during the NN training phase.

The layer enables the development of sophisticated end-to-end models for financial applications, allowing for investment decisions to be both informed by deep learning predictions and optimized through established financial theories.

3.2.2 Covariance layer

This layer transforms an input "data"matrix (the result of the previous transformations made by NN) into an empirical covariance matrix using the shrinkage technique, described in Equation 8, in such a way, that the ability of NN to autdifferentiate is preserved.

As a proposition of further research, it would be interesting to test the effectiveness of the denoising technique described in Section 2.3.3 in the context of the Covariance layer.

3.2.3 Normalization Layer

The Normalization Layer is a crucial component of modern neural network architectures, aimed at enhancing the stability and efficiency of the training process. This layer adjusts the input of a given layer so that they have a standard distribution, typically with a mean of zero and a standard deviation of one. This adjustment is applied per feature across the inputs it receives, ensuring a consistent and normalized data distribution that feeds into subsequent layers.

Normalization is achieved by calculating the mean and variance of the layer's inputs and using these statistics to normalize the input features. The normalization for each feature is mathematically represented as follows:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

where x_i is the input feature, μ is the mean of the inputs, σ^2 is the variance, ϵ is a small constant added for numerical stability, and $\hat{x}_i$ is the normalized output.

Additionally, the layer incorporates trainable scale (γ) and shift (β) parameters, allowing the network to dynamically adjust the normalization effect:

$$y_i = \gamma \hat{x}_i + \beta,$$

where y_i represents the final output of the normalization layer. This flexibility ensures that the normalization process can adapt to the needs of the specific data and learning task at hand.

The inclusion of trainable parameters γ and β enables the layer to learn an optimal scale and shift during training, effectively allowing the model to undo the normalization if that is what best serves the learning task.

3.3 Other layers

We also utilize the following layers:

1. Dense layer. The layer which implements the function from Equation 2.
2. Dropout layer. The layer from Section 1.4.4

3.4 Architectures

In this subsection, the implemented architectures will be described. Note these are 'the best' (in terms of validation error) architectures, we could find for our problem.

All the networks are implemented with a lookback parameter equal to 50, a number of assets equal to 5, and a horizon equal to 5.

3.4.1 DenseNet

DenseNet is a simple combination of dense layers with regularization dropout layers. Its architecture is shown in the image below. n represents the input sample size. The text in the lines represents the dimension of a layer output.

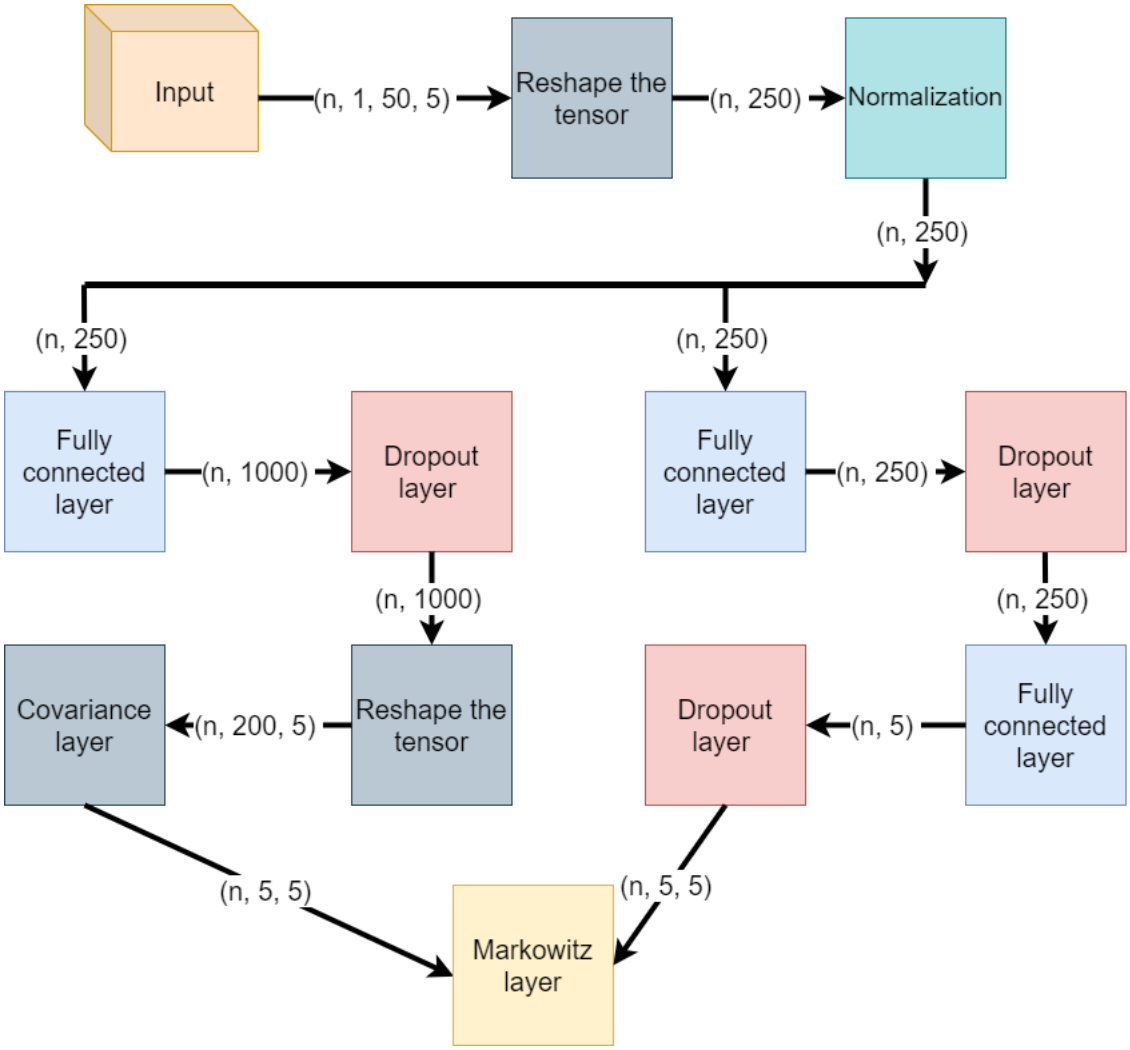

Рис. 10: Architecture of DenseNet.

It is interesting to note that despite testing the architecture utilizing Leaky ReLu and Maxout activation functions, the best NN was the one with the simplest ReLu activations.

The results of model training are shown in the image, below.

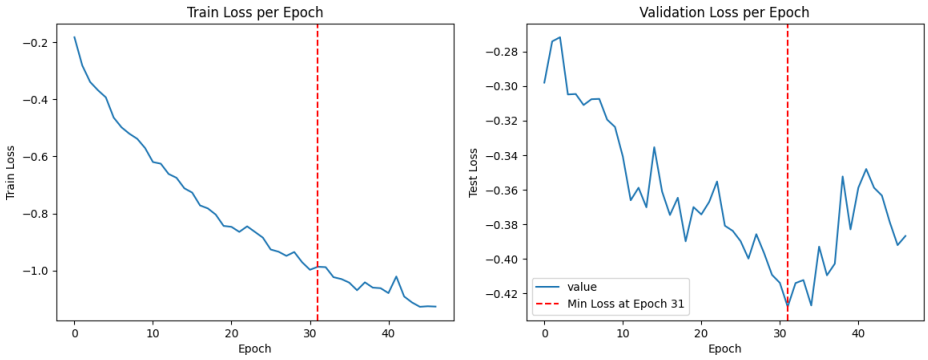

Рис. 11: DenseNet training progress.

As we can see even after the training the training loss function is significantly less the the validation one, despite all the used regularization. And this fact will remain true for all the trained networks. This can be explained by the data itself (it is very hard to generalize on financial data) and by the fact that we are using daily data. Possibly if the amount of data points is ten thousand the networks will be able to generalize better.

To initially test the applicability of the trained neural networks we also compare the performance of the trained NNs with the benchmark. As a benchmark, we are using an equally weighted portfolio. The comparison of the Sharpe ratio over the validation set of the trained network and of the benchmark is shown below.

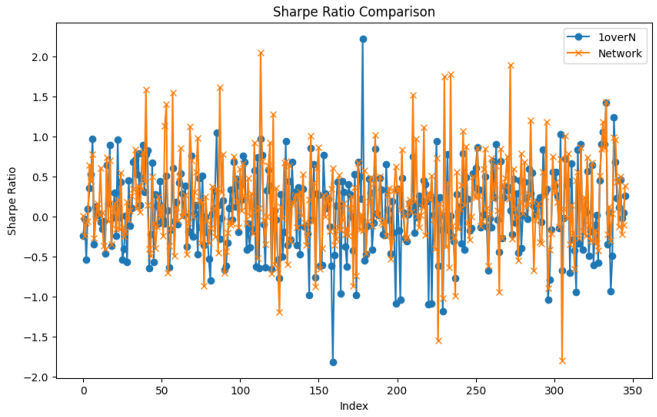

Рис. 12: Comparison of Sharpe ratio between the trained network and equally weighted portfolio benchmark.

The trained NN Sharpe ratio over the validation set on average is 31% times higher than the one of the equally weighted portfolio.

4 Other implemented strategies

This section will describe other (not end-to-end deep learning-based strategies).

4.1 Equally weighted portfolio

4.2 SPY long only

5 Backtest

The goal of our work is to compare the performance of classical algorithms and deep learning-based ones. To perform this comparison we will utilize backtesting. Backtesting is just running the rebalancing algorithm on specific data to gain an understanding of how the algorithm performs in general and how it generalizes on unseen market conditions (relevant for deep learning-based algorithms).

5.1 Data description

For this research, a strategic selection of five Exchange-Traded Funds (ETFs) was made to construct a diversified portfolio for backtesting. The chosen ETFs are: VWO (Vanguard FTSE Emerging Markets ETF), SPY (SPDR SP 500 ETF Trust), VNQ (Vanguard Real Estate ETF), LQD (iShares iBoxx $ Investment Grade Corporate Bond ETF), and DBC (Invesco DB Commodity Index Tracking Fund). These ETFs were selected based on pivotal criteria:

- Thes ETFs belong to different asset classes;
- Uniform trading across all assets on the New York Stock Exchange to guarantee liquidity;
- Availability of comprehensive data for the designated training and testing periods.

Economic Rationality The selection is grounded on the economic rationale to support the diversification strategy:

- **VWO:** Provides exposure to a broad range of large and mid-sized companies in emerging markets, aiming for diversification across different economic conditions.
- **SPY:** Represents a wide swath of the U.S. equity market, capturing the performance of the SP 500.
- **VNQ:** Focuses on real estate investment trusts (REITs), reflecting the dynamics of the real estate sector.
- **LQD:** Comprises investment-grade corporate bonds, offering sensitivity to interest rate changes.
- **DBC:** Tracks a commodities index, subject to global economic trends and geopolitical events.

5.1.1 Data Utilization and Strategy Retraining Protocol

The study employs daily price data of the selected ETFs from January 3, 2007, to December 15, 2020, for the model training phase. The subsequent

testing phase begins from December 15, 2020, extending to December 15, 2023. Crucially, to maintain the models' responsiveness to market changes, all neural network-based strategies are subject to annual retraining on December 15, 2021, and December 15, 2022.

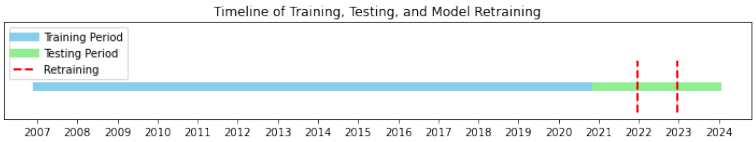

Рис. 13: Timeline of Training and Testing data.

5.1.2 Economic Rationality for Period Division

The allocation of training and testing periods incorporates critical economic events, equipping the models to navigate diverse market conditions:

- **Training Period (2007-2020):** This extensive period covers major financial upheavals, including the Global Financial Crisis and the European Debt Crisis, providing a rich dataset for training the models to identify and adapt to financial market volatilities.
- **Testing Period (2020-2023):** Encompassing the onset and aftermath of the COVID-19 pandemic, as well as the ongoing geopolitical and economic repercussions of the Russia-Ukraine conflict, this period tests the models' performance against a backdrop of unprecedented market disruptions and sustained uncertainties.

5.2 Portfolio metrics used

This section contains information about the performance of end-to-end deep learning-based strategies and an overall comparison of the backtest performance of all implemented strategies.

The performance of the end-to-end strategies is compared with the performance of the baseline strategies: equally weighted portfolio and SPY long-only portfolio.

5.2.1 Daily Value at Risk (97.5%)

Daily Value at Risk at the 97.5% confidence level estimates the potential loss in value of a portfolio over a defined period for a given confidence interval, calculated using the empirical quantile.

$$\text{VaR}_{97.5\%} = \text{Quantile}_{0.025}(\text{Portfolio Returns})$$

5.2.2 Expected Shortfall (97.5%)

Expected Shortfall at the 97.5% confidence level is the average loss during periods when the portfolio incurs losses beyond the VaR threshold, calculated using the empirical mean.

$$\text{ES}_{97.5\%} = \text{Mean}(\text{Returns} < \text{VaR}_{97.5\%})$$

5.2.3 Max Drawdown

Max Drawdown quantifies the largest single drop from peak to trough in the value of a portfolio before a new peak is achieved [13].

The formula to calculate Max Drawdown is given by:

$$\text{Max Drawdown} = \max_{\tau \in [0,T]} \left(\max_{t \in [0,\tau]} (P_t) - P_\tau \right) / \max_{t \in [0,\tau]} (P_t)$$

where:

- P_t represents the portfolio's value at any time t.
- P_τ denotes the portfolio's value at time τ, at which the trough is observed.
- The term $\max_{t \in [0,\tau]}(P_t)$ identifies the peak value of the portfolio up to time τ.
- The expression $\left(\max_{t \in [0,\tau]}(P_t) - P_\tau\right)$ calculates the drawdown at time τ, which is the decline from the peak value to the trough value at τ.
- T signifies the total observation period.

This calculation seeks the maximum value of these drawdown percentages over the entire observation period, effectively identifying the most significant percentage decline from a peak to a subsequent trough before the portfolio reaches a new peak value.

5.2.4 Stability

Stability measures the consistency of a portfolio's returns over time, aiming to capture the predictability of performance.

$$\text{Stability} = \text{Standard Deviation of the Portfolio's Annual Returns}$$

The Calmar Ratio assesses the risk-adjusted returns of an investment by comparing the annual return to the maximum drawdown over a specified period.

$$\text{Calmar Ratio} = \frac{\text{Annualized Return}}{\text{Maximum Drawdown}}$$

5.2.5 Sortino Ratio

The Sortino Ratio enhances the assessment of risk-adjusted returns by focusing exclusively on downside risk, differentiating between harmful and total volatility. It is defined by the formula:

$$\text{Sortino Ratio} = \frac{\text{Expected Return} - \text{Risk-Free Rate}}{\text{Downside Deviation}}$$

where *Downside Deviation* measures the volatility of returns falling below a target threshold (zero in our case), emphasizing the negative returns that investors are particularly keen to minimize.

The main advantage of the ratio over the Sharpe ratio is the fact that it does not penalize positive volatility, making it preferable for strategies with asymmetric return profiles.

5.2.6 Calmar Ratio

The Calmar Ratio assesses the risk-adjusted returns of an investment by comparing the annual return to the maximum drawdown over a specified

period.

$$\text{Calmar Ratio} = \frac{\text{Annualized Return}}{\text{Maximum Drawdown}}$$

5.2.7 Alpha and Beta

In the Capital Asset Pricing Model (CAPM), Alpha (α) and Beta (β) are metrics used to evaluate an investment's return relative to its risk. CAPM posits a linear relationship between the expected risk of an asset and its expected return.

Beta represents the sensitivity of an asset's returns to the returns of the market. It is the slope in the regression of the asset's returns on the market's returns, indicating how much the asset's return is expected to change with a change in the market return:

$$\beta = \frac{\text{Cov}(R_a, R_m)}{\text{Var}(R_m)}$$

where R_a is the asset return, and R_m is the market return.

Alpha measures the excess return of an asset relative to the return predicted by its Beta. It is the intercept in the regression, representing the asset's return that is not explained by market movements, thus indicating the portfolio performance:

$$\alpha = R_a - [R_f + \beta \times (R_m - R_f)]$$

where R_f stands for the risk-free rate.

While Beta assesses the asset's market-related risk and return, Alpha seeks to capture the asset's unique, risk-adjusted performance against its benchmark [1].

Not as a market benchmark we are using returns of SPY ETF.

5.2.8 Average interweight distance

Average interweight distance is the average L2 distance between portfolio weights before and after rebalancing.

5.3 Zipline

Backtesting is implemented using *Zipline*, a Python library for backtesting trading algorithms, is utilized. Preferred for its ability to simulate realistic market scenarios, Zipline is chosen despite its limitations in performance compared to vectorized backtesting frameworks [29]. As an event-driven system processing data sequentially, Zipline demands more computational resources and time but offers a high degree of simulation accuracy.

The following Zipline setting was used to replicate actual trading conditions:

- **Commission Structure**: The adopted commission structure imposes a $0.00075 fee per share traded, with a transaction minimum of $0.01.
- **Slippage Model**: The slippage model addresses the inevitable variance between the anticipated execution price of an order and the price at which it is actually filled. To capture this effect, the backtesting environment incorporates a slippage model tailored to reflect the volume impact of trades. The slippage model is calibrated to allow a trade to consume a defined fraction of the total available volume per time unit, with each share traded exerting a specified percentage impact on the market price.

.

5.4 Backtest results for the implemented strategies

5.4.1 DenseNet

This section will describe the result of the backtest for the network architecture described in Section 3.4.1.

The cumulative returns, rolling Share ratio and rolling volatility are shown in the figures below.

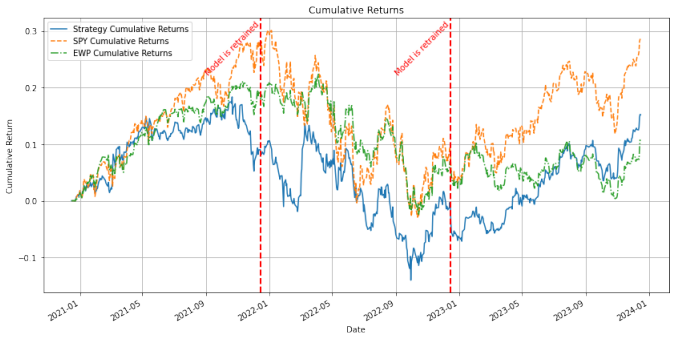

Рис. 14: DenseNet cumulative returns.

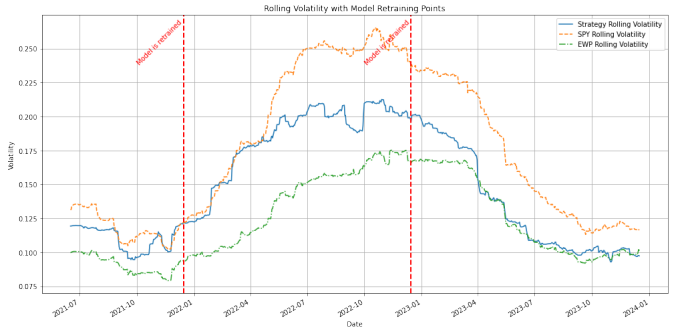

Рис. 15: DenseNet six months annualized rolling volatility.

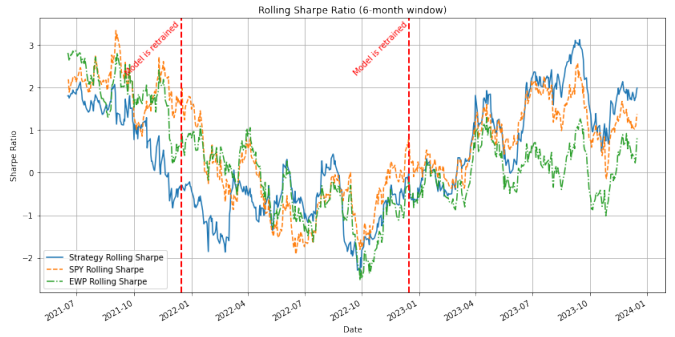

Рис. 16: DenseNet six months annualized rolling Sharpe ratio.

The overall strategy performance is summarized in the table below.

Metric	EWP	Column 1	Column 2	SPY
Annual return	3.447%	3.573%	4.123%	8.701%
Cumulative returns	10.731%	11.139%	12.923%	28.525%
Annual volatility	12.38%	13.706%	13.702%	17.586%
Sharpe ratio	0.34	0.32	0.36	0.56
Calmar ratio	0.17	0.13	0.16	0.34
Stability	0.18	0.21	0.16	0.02
Max drawdown	-20.226%	-26.528%	-26.1%	-25.36%
Omega ratio	1.06	1.06	1.07	1.10
Sortino ratio	0.48	0.47	0.53	0.80
Tail ratio	1.02	1.04	1.04	1.02
Daily value at risk	1.543%	-1.709%	-1.707%	2.176%
Expected Shortfall	-0.02027	-0.02406	-0.02405	-0.0305
Alpha	-0.02	-0.01	-0.01	0.00
Beta	0.60	0.57	0.57	1.00
Average inter weight distance	0	0.1382	0.1383	0

Таблица 2: Updated performance metrics comparison.

The overall performance of the strategy is average. It has lower returns than SPY and bigger volatility than EWP and vice versa. Alpha and beta in this case are good characterises of the portfolio: lower volatility than the one of the benchmark and significantly lower returns.

Список литературы

[1] P. Asquith and L. A. Weiss. *Lessons in corporate finance: A case studies approach to financial tools, financial policies, and valuation.* John Wiley & Sons, 2019.

[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] M. M. L. de Prado. *Machine learning for asset managers.* Cambridge University Press, 2020.

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

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

[6] 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.

[7] 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.

[8] M. Karoglou. Breaking down the non-normality of stock returns. *The European journal of finance*, 16(1):79–95, 2010.

[9] O. Ledoit and M. Wolf. Honey, i shrunk the sample covariance matrix. *UPF economics and business working paper*, (691), 2003.

[10] O. Ledoit and M. Wolf. A well-conditioned estimator for large-dimensional covariance matrices. *Journal of multivariate analysis*, 88(2):365–411, 2004.

[11] 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.

[12] 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.

[13] M. Magdon-Ismail and A. F. Atiya. Maximum drawdown. *Risk Magazine*, 17(10):99–102, 2004.

[14] R. O. Michaud. The markowitz optimization enigma: Is 'optimized'optimal? *Financial analysts journal*, 45(1):31–42, 1989.

[15] 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.

[16] M. Nezafat, M. Kaveh, and W. Xu. Estimation of the number of sources based on the eigenvectors of the covariance matrix. In *Processing Workshop Proceedings, 2004 Sensor Array and Multichannel Signal*, pages 465–469. IEEE, 2004.

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

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

[19] S. Raschka. *Python machine learning*. Packt publishing ltd, 2015.

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

[21] G. J. Rosa. The elements of statistical learning: Data mining, inference, and prediction by hastie, t., tibshirani, r., and friedman, j., 2010.

[22] P. J. Rousseeuw. Least median of squares regression. *Journal of the American statistical association*, 79(388):871–880, 1984.

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

[24] 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.

[25] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

[26] M. Verleysen and D. François. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer, 2005.

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

[28] 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.

[29] Zipline Development Team. Zipline documentation. `https://zipline.ml4trading.io/`, 2023. Accessed: November 21, 2023.

[30] Марченко and Пастур. Распределение собственных значений в некоторых ансамблях случайных матриц. *Математический сборник*, 72(4):507–536, 1967.