- $\hat{y} \in \mathbb{R}^{n_{N+1}}$, $\hat{y} = (\hat{y}_1, \hat{y}_2, ..., \hat{y}_{n_{N+1}})^\top$ be a vector of estimated output values obtained from the neural network by sending an input vector x through the network.

Remark 5.5. *Since the input layer has n_0 neurons, the input vector x must be an element of $\mathbb{R}^{n_0}$. Similarly, the estimated output $\hat{y}$ of the neural network is a vector which is an element of $\mathbb{R}^{n_{N+1}}$.*

Remark 5.6. *Since $a^{(i)}$ denotes the values that are transferred between neurons, it is both the output values of the neurons from layer i and the input values of the neurons from layer $i+1$.*

Remark 5.7. *The input values x for the neuronal network correspond to the first activation values $a^{(0)}$, e.g. $x = a^{(0)}$. Similarly, the estimated output $\hat{y}$ of the neural network corresponds to the last activation values $a^{(N+1)}$, e.g. $\hat{y} = a^{(N+1)}$.*

Remark 5.8. *For the training of a neural network many different sets of input and associated output vectors are needed. If a specific set of input or output data is to be referenced, this is done by adding a superscript, e.g. $x^{(k)}$ and $y^{(k)}$. The corresponding estimated values are also referred to as $\hat{y}^{(k)}$.*

Remark 5.9. *The weight defined as $W^{(i)}_{jk}$ connects the activation value from the k-th neuron in layer i with the j-th neuron from layer $i+1$.*

After a notation for the analysis of the neural network has been defined, the next section is devoted to the question of how data can flow through a network and how the network can learn.

5.2. Train a model

Figure (5.3) together with formula (5.1) already illustrated in the previous section how a single neuron is constructed and how it processes the inputs. Since the notation has been generalized in the previous section also formula (5.1) could be generalized.

Remark 5.10. *Let $a^{(i)} \in \mathbb{R}^{n_i}$ be the activation values from layer i, $b^{(i)}_j \in \mathbb{R}$ the bias term for the j-th neuron in layer $i+1$, $W^{(i)}_{j.}$ the weights and $g : \mathbb{R} \mapsto \mathbb{R}$*

any activation function. Then the linear transformation performed by the j-th neuron of layer $i+1$ called $s_j^{(i+1)}$ is given by:

$$z_j^{(i+1)} = \sum_{k=1}^{n_i} W_{jk}^{(i)} a_k^{(i)} + b_j^{(i)} \tag{5.6}$$

Using vector notation gives:

$$z_j^{(i+1)} = W_{j.}^{(i)} a^{(i)} + b_j^{(i)} \tag{5.7}$$

Remark 5.11. *Based on the previous remark, the activation value $a_j^{(i+1)}$ generated by neuron $s_j^{(i+1)}$ is given by:*

$$a_j^{(i+1)} = g(z_j^{(i+1)}) = g\left(\sum_{k=1}^{n_i} W_{jk}^{(i)} a_k^{(i)} + b_j^{(i)} \right) \tag{5.8}$$

Using vector notation gives:

$$a_j^{(i+1)} = g(z_j^{(i+1)}) = g(W_{j.}^{(i)} a^{(i)} + b_j^{(i)}) \tag{5.9}$$

Performing these transformations for all neurons from layer $i+1$ gives a system of n_{i+1} equations:

$$\begin{aligned} a_1^{(i+1)} &= g(W_{1.}^{(i)} a^{(i)} + b_1^{(i)}) \\ a_2^{(i+1)} &= g(W_{1.}^{(i)} a^{(i)} + b_1^{(i)}) \\ &\vdots \\ a_{n_{i+1}}^{(i+1)} &= g(W_{1.}^{(i)} a^{(i)} + b_1^{(i)}) \end{aligned}$$

Since the activation function is only defined for scalar values, a small extension must be made.

Remark 5.12. *Let $g : \mathbb{R} \mapsto \mathbb{R}$ be an activation function which operates on scalars only. In case that a vector is supplied the scalar activation function g is applied element by element.*

With the help of the previous remark it is now possible to represent the flow of information through the network in vector notation.

$$\begin{aligned} a^{(0)} &= x \\ z^{(i)} &= W^{(i-1)} a^{(i-1)} + b^{(i-1)} \\ a^{(i)} &= g(z^{(i)}) \\ a^{(N+1)} &= \hat{y} \end{aligned} \tag{5.10}$$

This step, where the neural network takes an input vector and transforms it to a prediction is the first step to train the network and is called forward propagation [48]. The process shown in figure (5.2) illustrates that after the generation of the predictions $\hat{y}$ the deviation between those and the real outputs y must be measured. A way of measuring the deviation between actual and estimated values is the L^2 norm.

Definition 5.1. *Let $x^{(i)}$ and $y^{(i)}$ be the vector of the input and the corresponding output values, respectively. Let W be the matrices of weights and b the vectors of biases used in the entire network for a single forward propagation. If $\hat{y}^{(i)}_{W,b,x^{(i)}}$ denotes the estimated outputs from the network then the loss score is given by:*

$$L(b, W, x^{(i)}, y^{(i)}) := \frac{1}{2} \left\| \hat{y}^{(i)}_{W,b,x^{(i)}} - y^{(i)} \right\|_2^2 \tag{5.11}$$

By using the notation given in equation (5.11) it is explicitly stated, that the loss of a single forward pass depends on the four parameters $b, W, x^{(i)}$ and $y^{(i)}$. In order to increase the readability in the following paragraphs from now on it is not longer explicitly stated that the estimated output depends on the three parameters W, b and $x^{(i)}$, e.g. $\hat{y}^{(i)}_{W,b,x^{(i)}} = \hat{y}$. Since in most cases not only a single pair of input and output data is used for the calculation of the loss function but several, the loss must also be defined for a sample of multiple input and output vectors. In the field of machine learning such a sample of multiple input and output vectors is called a batch. The batch size is one of many so-called hyperparameters of a neural network and specifies how many samples must be sent through the network before the model parameters are updated.

Definition 5.2. *Let $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$ be a sample of m input and the corresponding output vectors, e.g. the batch size is m. Then the total loss is given by:*

$$\begin{aligned} L(b, W) &= \frac{1}{m} \sum_i^m \frac{1}{2} \left\| \hat{y}^{(i)} - y^{(i)} \right\|_2^2 \\ &= \frac{1}{m} \sum_i^m L(b, W, x^{(i)}, y^{(i)}) \end{aligned} \tag{5.12}$$

Remark 5.13. *In order to reduce the complexity of the model by using smaller weights and to reduce overfitting to some extend, a regularization term can be*

used in equation (5.12).

$$\begin{aligned} L(b, W) &= \frac{1}{m} \sum_{i}^{m} \frac{1}{2} \left\| \hat{y}^{(i)} - y^{(i)} \right\|_2^2 + \frac{\lambda}{2} \sum_{i=0}^{N} \left\| W^{(i)} \right\|_F^2 \\ &= \frac{1}{m} \sum_{i}^{m} L(b, W, x^{(i)}, y^{(i)}) + \lambda R(W) \end{aligned} \tag{5.13}$$

If a regularization term is used in a neural network, the choice of the regularization parameter λ plays an important role. When the regularization term λ is too high the weight matrices W are close to zero and this can lead to simple networks and underfitting. However, if the parameter is set too low and therefore higher weights of the matrices are not penalized enough, regularization has no effect at all.

Once a loss function is defined, the information of the loss score can be used to take the next step as shown in figure (5.2). The job of the optimizer is to evaluate how the weights of W and b have to be adjusted so that the loss score becomes smaller in the next forward pass. A reduction of the loss score indicates that the deviation between predicted output values and actual outputs is reduced. This iterative improvement of the loss score based on the insights given by the optimizer is the so called learning procedure of a neural network. The task of minimizing $L(b, W)$ with respect to the weights b and W can be solved by using the gradient descent method which is an first-order iterative optimization algorithm for finding the local minimum of a function [9]. This method calculates, using the gradients, for each weight of the network what effect a change would have on the loss score. The algorithm used to calculate the gradients is called backpropagation. Once all gradients have been calculated after a forward pass, one way to update the weights of the neural network is to adjusted them by a constant learning rate of α.

Remark 5.14. *Let $W^{(i)}$ and $b^{(i)}$, $i = 0, 1, ..., N$ be the weights used in the $i+1$-th layer and $\alpha \in \mathbb{R}_+$ the user defined learning rate. Then the weights are updated in every optimization step such that:*

$$b_j^{(i)} = b_j^{(i)} - \alpha \frac{\partial}{\partial b_j^{(i)}} L(b, W) \tag{5.14}$$

$$W_{jk}^{(i)} = W_{jk}^{(i)} - \alpha \frac{\partial}{\partial W_{jk}^{(i)}} L(b, W) \tag{5.15}$$

Since the learning rate α is a hyperparameter that can be chosen freely, the only parts that need to be analyzed are the partial derivatives. By using definition 5.2 those can be written as:

$$\frac{\partial}{\partial b_j^{(i)}} L(b, W) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_j^{(i)}} L(b, W, x^{(i)}, y^{(i)}) \tag{5.16}$$

$$\frac{\partial}{\partial W_{jk}^{(i)}} L(b, W) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{jk}^{(i)}} L(b, W, x^{(i)}, y^{(i)}) \tag{5.17}$$

This shows that the partial derivative of the loss function based on a batch of size m is given by the mean value of the partial derivatives of all single loss functions based on the individual input and output vectors used in this batch. If the regularization term introduced in remark 5.13 is used for the weight matrices W, this does not change anything for the partial derivatives with respect to the bias terms b, but the partial derivatives of the weight matrices given in formula (5.17) changes to:

$$\frac{\partial}{\partial W_{jk}^{(i)}} L(b, W) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{jk}^{(i)}} L(b, W, x^{(i)}, y^{(i)}) + \lambda W_{jk}^{(i)} \tag{5.18}$$

If the regularization term is used, just an additive expression of the form $\lambda W_{jk}^{(i)}$ is added. Since the construction of the partial derivatives with respect to $W_{jk}^{(i)}$ and $b_j^{(i)}$ are very similar, only the former will be explained step by step in the following. Before the partial derivative can be analysed in more detail a distinction must be made between two different cases.

Case 1: The first case deals with the special situation in which the weights $W^{(N)}$ and $b^{(N)}$ are directly related to the output. These weights are used in the transformation that results in the estimated output values $\hat{y}$ and thus directly influences the loss function $L(b, W, x^{(i)}, y^{(i)})$. Because of their direct influence on the loss function, the calculation of their gradients is easier and can be used as a template for the other case.

Remark 5.15. *Let $W^{(N)}$ be the matrix of weights used in the last layer of the neural network. Then the gradient is given by:*

$$\frac{\partial L(b, W)}{\partial W_{jk}^{(N)}} = \delta_j^{(N+1)} a_k^{(N)}$$

Proof.

$$\begin{aligned}
\frac{\partial L(b,W)}{\partial W_{jk}^{(N)}} &= \frac{\partial}{\partial W_{jk}^{(N)}} \frac{1}{2} \sum_{l=1}^{n_{N+1}} (\hat{y}_l - y_l)^2 \\
&= \frac{\partial}{\partial W_{jk}^{(N)}} \frac{1}{2} \sum_{l=1}^{n_{N+1}} (a_l^{(N+1)} - y_l)^2 \\
&= (a_j^{(N+1)} - y_j) \frac{\partial}{\partial W_{jk}^{(N)}} (a_j^{(N+1)} - y_j) \\
&= (a_j^{(N+1)} - y_j) \frac{\partial}{\partial W_{jk}^{(N)}} g\big(z_j^{(N+1)}\big) \\
&= (a_j^{(N+1)} - y_j) g'\big(z_j^{(N+1)}\big) \frac{\partial}{\partial W_{jk}^{(N)}} \sum_{s=1}^{n_N} W_{js}^{(N)} a_s^{(N)} + b_j^{(N)} \\
&= (a_j^{(N+1)} - y_j) g'\big(z_j^{(N+1)}\big) a_k^{(N)} \\
&= \delta_j^{(N+1)} a_k^{(N)}
\end{aligned}$$

Whereby in the last step all terms with index j were summarized in δ_j^{N+1}. □

Remark 5.16. *Let $b^{(N)}$ be the vector of biases used in the last layer of the neural network. Then the gradient is given by:*

$$\frac{\partial L(b,W)}{\partial b_j^{(N)}} = \delta_j^{(N+1)}$$

Case 2: The second case deals with those situations where the weights W and b are not directly but indirectly associated with the loss function. These are all weights that are used in the hidden layers 1 to N. The calculation of those gradients relies on the first case and requires several steps.

Remark 5.17. *Let $z_l^{(N+1)}$ be the result of the linear transformation of neuron l in the $N+1$-th layer and $W_{jk}^{(N-1)}$ the weight for the j-th neuron in the N-th layer. Then the partial derivative of $z_l^{(N+1)}$ with respect to $W_{jk}^{(N-1)}$ is given by:*

$$\frac{\partial z_l^{(N+1)}}{W_{jk}^{(N-1)}} = \frac{\partial z_l^{(N+1)}}{\partial a_k^{(N)}} \frac{\partial a_k^{(N)}}{\partial W_{jk}^{(N-1)}} = W_{lk}^{(N)} g'\big(z_k^{(N)}\big) a_k^{(N-1)} \tag{5.19}$$

Proof.

$$\frac{\partial z_l^{(N+1)}}{\partial a_k^{(N)}} = \frac{\partial}{\partial a_k^{(N)}} \sum_{s=1}^{n_N} W_{ls}^{(N)} a_s^{(N)} + b_l^{(N)} = W_{lk}^{(N)}$$
$$\frac{\partial a_k^{(N)}}{\partial W_{jk}^{(N-1)}} = g'\big(z_k^{(N)}\big) \frac{\partial}{\partial W_{jk}^{(N-1)}} \sum_{s=1}^{n_{N-1}} W_{ks}^{(N-1)} a_s^{(N-1)} + b_k^{(N-1)} = g'\big(z_k^{(N)}\big) a_k^{(N-1)}$$

□

Remark 5.18. *Let $W^{(N-1)}$ be the matrix of weights used in the last hidden layer of the neural network. Then the gradient is given by:*

$$\frac{\partial L(b,W)}{\partial W_{jk}^{(N-1)}} = g'\big(z_k^{(N)}\big) a_k^{(N-1)} \sum_{l=1}^{n_{N+1}} \delta_l^{(N+1)} W_{lk}^{(N)}$$

Proof.

$$\begin{aligned}
\frac{\partial L(b,W)}{\partial W_{jk}^{(N-1)}} &= \frac{\partial}{\partial W_{jk}^{(N-1)}} \frac{1}{2} \sum_{l=1}^{n_{N+1}} (a_l^{(N+1)} - y_l)^2 \\
&= \sum_{l=1}^{n_{N+1}} (a_l^{(N+1)} - y_l) \frac{\partial}{\partial W_{jk}^{(N-1)}} (a_l^{(N+1)} - y_l) \\
&= \sum_{l=1}^{n_{N+1}} (a_l^{(N+1)} - y_l) \frac{\partial}{\partial W_{jk}^{(N-1)}} g\big(z_l^{(N+1)}\big) \\
&= \sum_{l=1}^{n_{N+1}} (a_l^{(N+1)} - y_l) g'\big(z_l^{(N+1)}\big) \frac{\partial}{\partial W_{jk}^{(N-1)}} z_l^{(N+1)} \\
&= \sum_{l=1}^{n_{N+1}} \delta_l^{(N+1)} \frac{\partial}{\partial W_{jk}^{(N-1)}} z_l^{(N+1)} \\
&= g'\big(z_k^{(N)}\big) a_k^{(N-1)} \sum_{l=1}^{n_{N+1}} \delta_l^{(N+1)} W_{lk}^{(N)}
\end{aligned}$$

Where in the last step the result from remark 5.17 was used. □

Remark 5.19. *Let $b^{(N-1)}$ be the vector of biases used in the last hidden layer of the neural network. Then the gradient is given by:*

$$\frac{\partial L(b,W)}{\partial b_j^{(N-1)}} = g'\big(z_j^{(N)}\big) \sum_{l=1}^{n_{N+1}} \delta_l^{(N+1)} W_{lk}^{(N)}$$

Those results derived in the last section can be used for any number of hidden layers so that the update process described in remark 5.14 can be slightly adopted and summarized in algorithm 5.

Algorithm 5 Stochastic gradient descent (SGD) with mini batches [21]

1. Define learning rates $\alpha_1, \alpha_2, ...$
2. Initialize weight parameters W and b pooled in θ.
3. Set a counter: $k = 1$
4. While stopping criterion is not met
   a) Sample a minibatch of m examples from the training set with corresponding output values $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$
   b) Compute gradient estimate: $g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
   c) Apply update: $\theta = \theta - \alpha_k g$.
   d) Increase counter: $k = k + 1$

Remark 5.20. *The notation for the computed gradient* $\frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ *in algorithm 5 is just a short form of formulas (5.16) and (5.17).*

Remark 5.21. *By setting all learning rates equal (e.g.* $\alpha = \alpha_1 = \alpha_2 = ... = \alpha_k$*), exactly the update process described in remark 5.14 is obtained.*

Remark 5.22. *In practice, the learning rates are often chosen in such a way that a linear decrease takes place up to a certain number of iterations and then the learning rates are kept constant.*

$$\alpha_k = \begin{cases} (1 - \frac{k}{\tau})\alpha_0 + \frac{k}{\tau}\alpha_\tau & k < \tau \\ \alpha_\tau & k \geq \tau \end{cases}$$

With α_0, α_τ *and* τ *set such that:*

- α_τ *is roughly 1 percent of the initial learning rate* α_0*.*
- τ *is set to the number of iterations required to pass the whole training set a few hundred times through the neural network.*
- α_0 *does not make the learning process of the model oscillate too much by setting it too high. Gentle oscillations are okay, but setting* α_0 *properly is part of a hyperparameter tuning.*

Remark 5.23 ([21]). *A sufficient condition to guarantee that the stochastic gradient descent algorithm converges is that:*

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \textit{and} \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

Based on SGD, the algorithm can be optimized in terms of time consumption. By adding a momentum, the learning process can be accelerated and the undesired case that the algorithm runs into a local minimum is partly counteracted.

Algorithm 6 Stochastic gradient descent (SGD) with momentum [21]

1. Initialize learning rate α and momentum parameter γ
2. Initialize weight parameter θ and velocity v.
3. Set a counter: $k = 1$
4. While stopping criterion is not met
   a) Sample a minibatch of m examples from the training set with corresponding output values $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$
   b) Compute gradient estimate: $\hat{g} = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
   c) Compute velocity update: $v = \gamma v - \alpha g$
   d) Apply update: $\theta = \theta + v$.
   e) Increase counter: $k = k + 1$

Remark 5.24. *The more consecutive gradients point in the same direction, the greater the updates of the weights become.*

Remark 5.25. *Usually the momentum parameter γ is initialized with values like 0.5, 0.9 or 0.99. Adopting γ over time is possible but typically less important than the linear decay of α shown previously.*

In the two algorithms 5 and 6 presented so far, the learning rate α is specified globally. As mentioned in remark 5.22 a sequence of learning rates α_i can be defined so that the learning rate is decreasing over time and is then kept constant after a certain number of iterations. If the idea of globally falling learning rates α_i is taken a step further, a group of algorithms can be presented which individually adapts the learning rates of every single model parameter. This then leads to a rapid drop in the individual learning rate for parameters

Algorithm 7 Adaptive Gradients (AdaGrad) [21]

1. Initialize learning rate α.
2. Initialize Small constant δ used for numerical stabilization. (Suggested default: 10^{-7})
3. Initialize weight parameter θ.
4. Initialize gradient accumulation variable: $r = 0$.
5. While stopping criterion is not met
   a) Sample a minibatch of m examples from the training set with corresponding output values $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$
   b) Compute gradient estimate: $\hat{g} = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
   c) Accumulate squared gradient: $r = r + g \odot g$
   d) Compute update: $\Delta\theta = -\frac{\alpha}{\sqrt{r}+\delta} \odot g$ (Division and square root applied element-wise)
   e) Apply update: $\theta = \theta + \Delta\theta$.

Algorithm 8 Root mean square propagation (RMSProp) [21]

1. Initialize learning rate α and decay parameter ρ.
2. Initialize Small constant δ used for numerical stabilization. (Suggested default: 10^{-6})
3. Initialize weight parameter θ.
4. Initialize gradient accumulation variable: $r = 0$.
5. While stopping criterion is not met
   a) Sample a minibatch of m examples from the training set with corresponding output values $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$
   b) Compute gradient estimate: $\hat{g} = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
   c) Accumulate squared gradient: $r = \rho r + (1 - \rho)g \odot g$
   d) Compute update: $\Delta\theta = -\frac{\alpha}{\sqrt{r}+\delta} \odot g$ ($\frac{1}{\sqrt{r}+\delta}$ applied element-wise)
   e) Apply update: $\theta = \theta + \Delta\theta$.

with large partial derivatives. In contrast, the learning rates of parameters with comparatively small partial derivatives are changed only slightly. This approach therefore allows an individual adjustment of the learning rate for each individual parameter depending on how the corresponding gradient behaves. Algorithms 7 and 8 show possible implementations of such adaptive learning approaches.

Remark 5.26. *Algorithm 7 implements an adoptive learning rate for each individual model parameter by summing all previous squared gradients. As the sum increases with each iteration, this means that the learning rate decreases monotonically.*

The strength of algorithm 7 lies in the fast convergence for convex functions. Since the entire history of the gradients is always used for the adjustment of the learning rates through summation, there may be cases in which the algorithm is trapped in convex sinks. This may be due to the fact that when such a sink occurs, the individual learning rate has already been reduced so much that this local minimum cannot be left. Algorithm 8 addresses this problem by making the adoptive learning rate more flexible.

Remark 5.27. *Algorithm 8 uses an exponentially weighted moving average to adjust the individual learning rates. The decay of the moving average is then controlled by a new hyperparameter called ρ, with small values increasing the magnitude of decay.*

By using the exponentially weighted moving average, previous gradients are considered less for the adaptive learning rate the further back in time they appeared. This means that, in contrast to the AdaGrad method, the individual learning rate does not have to fall monotonously but can develop more flexible.

The last algorithm presented here can be seen as a combination of RMSProp (algorithm 8) and SGD with momentum (algorithm 6). This algorithm, called Adam, shown in algorithm 9 is fairly new in the field of neural networks and uses momentum and adaptive learning rates to converge faster and more robust [27].

Remark 5.28. *Since the update process of the biased first moment estimate in step d) of algorithm 9 can be expressed as $s_t = (1 - \rho_1) \sum_{i=1}^{t} \rho_1^{t-i} g_i$ the correction term can be derived by:*

Algorithm 9 Adaptive Moments (Adam) [21]

1. Initialize learning rate α. (Suggested default: 0.001)
2. Initialize exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
3. Initialize Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
4. Initialize weight parameter θ.
5. Initialize 1st and 2nd moment variables $s = 0$, $r = 0$.
6. Initialize time step $t = 0$.
7. While stopping criterion is not met
   a) Sample a minibatch of m examples from the training set with corresponding output values $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$
   b) Compute gradient estimate: $\hat{g} = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
   c) Increase time step: $t = t + 1$
   d) Update biased first moment estimate: $s = \rho_1 s + (1 - \rho_1) g$
   e) Update biased second moment estimate: $r = \rho_2 r + (1 - \rho_2) g \odot g$
   f) Correct bias in first moment: $\hat{s} = \frac{s}{1 - \rho_1^t}$
   g) Correct bias in second moment: $\hat{r} = \frac{r}{1 - \rho_2^t}$
   h) Compute update: $\Delta\theta = -\alpha \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$
   i) Apply update: $\theta = \theta + \Delta\theta$. (operations applied element-wise)
   j) Increase counter: $k = k + 1$

$$\begin{aligned}\mathbb{E}[s_t] &= \mathbb{E}\left[(1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i}g_i\right] \\ &= \mathbb{E}[g_t](1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i} + \zeta \\ &= \mathbb{E}[g_t](1-\rho_1^t) + \zeta\end{aligned}$$

Where $\zeta = 0$ if the true first moment is stationary and otherwise ζ can be kept small because of the exponential decay [27].

Remark 5.29. *Similarly to remark 5.28 the correction term for the biased second moment can be derived.*

After the basic concepts have been explained, the last section of this chapter is dedicated to the question how well cash flows can be replicated with neural networks using different optimizer and layer structures.

5.3. Cash flow replication