Problem 3.1 (2.5 points)

In the following, we will examine two variants of gradient descent that vary in regards to how they traverse the training data set $T_{train} = \{\mathbf{x}_{(i)}, y_{(i)}\}_{i=0}^{N-1}$. In both variants, the estimate of the weight vector $\hat{\mathbf{w}}^{(k)}$ is updated by a scaled version of the gradient $\nabla_{\mathbf{w}} J(\hat{\mathbf{w}}^{(k-1)})$ for each iteration $k = [1, \ldots, K]$.

Batch-Gradient-Descent: In BGD the loss is evaluated over the complete training set, before the weights are updated with the sample mean of the obtained gradient information. As such, the update equation is given by 3.1.1.

$$\hat{\mathbf{w}}^{(k)} \leftarrow \hat{\mathbf{w}}^{(k-1)} - \alpha \cdot \frac{1}{N} \sum_{i=0}^{N-1} \nabla_{\mathbf{w}} J(\hat{\mathbf{w}}^{(k-1)}, \mathbf{x}_{(i)}, y_{(i)}). \tag{3.1.1}$$

Stochastic-Gradient-Descent: Here the gradient $\nabla_{\mathbf{w}} J(\hat{\mathbf{w}}^{(k-1)}, \mathbf{x}_i, y_i)$ is not averaged over the training batch. As such the weights are updated immediately while SGD iterates over the training samples. See 3.1.2 — where $i = (k-1) \mod N$.

$$\hat{\mathbf{w}}^{(k)} \leftarrow \hat{\mathbf{w}}^{(k-1)} - \alpha \cdot \nabla_{\mathbf{w}} J(\hat{\mathbf{w}}^{(k-1)}, \mathbf{x}_{(i)}, y_{(i)}) \tag{3.1.2}$$

In the following, we will implement BGD and SGD for a simple linear regressor with two features x_0 and x_1 given by

$$\hat{y} = w_0 x_0 + w_1 x_1 = \mathbf{w}^T \mathbf{x} \tag{3.1.3}$$

and the following squared loss function for a given sample $\{\mathbf{x}_{(i)}, y_{(i)}\}$:

$$J(\mathbf{w}, \mathbf{x}_{(i)}, y_{(i)}) = (y_{(i)} - \mathbf{w}^T \mathbf{x}_{(i)})^2. \tag{3.1.4}$$

3.1.1 Download the dataset *training.csv* from TUWEL and provide a surface plot of the MSE[1] for the complete training dataset over weight configurations in the interval $w_0 \in [0{,}100]$ and $w_1 \in [0{,}100]$. Is the error surface convex?

3.1.2 Provide an analytical expression for the gradient $\nabla_{\mathbf{w}} J(\mathbf{w}, \mathbf{x}^{(i)}, y^{(i)})$ of the loss function given in Equation 3.1.4.

3.1.3 Use the result from task 3.1.2 to implement BGD for the regressor from 3.1.4 and fit it to *training.csv*. Provide a plot of the MSE for the complete training dataset over the iterations $k = [1, \ldots, K]$.
Hint: Use the following configuration: $K = 200$, $\alpha = 10^{-2}$, $\hat{\mathbf{w}}^{(0)} = [1, 1]^T$

3.1.4 Extent the code from Task 3.1.3, to provide a 2D scatter plot showcasing the history of the weight estimates $\hat{\mathbf{w}}^{(k)}$ throughout the optimization process. What configuration do you obtain for $\hat{\mathbf{w}}^{(K)}$?

[1] Mean Squared Error

3.1.5 Repeat the Tasks 3.1.3 and 3.1.4 for SGD using the same configuration. What do you observe when comparing the plots for BGD and SGD?

3.1.6 Examine the influence of the learning rate α on the training process. For that, repeat the training of SGD and BGD with $\alpha = 10^{-1}$ and again provide plots of the weights and MSE history. Briefly discuss the benefits/drawbacks of high/low values for α — what do you observe for $\alpha \geq 1$?

3.1.7 One full iteration over the training dataset is denoted as an *epoch*. Specify the number of *epochs* for BGD and SGD in this example. Which gradient descent variant would you choose for a large training dataset? Explain your answer.

Problem 3.2 (2.5 points)

Consider the simple two layer neural network with 3 neurons in total depicted in Figure 3.2.1. Here, the input consists of the vector $\mathbf{x} = [x_0, x_1]^T$, while the output is given by the scalar $\hat{y}$.

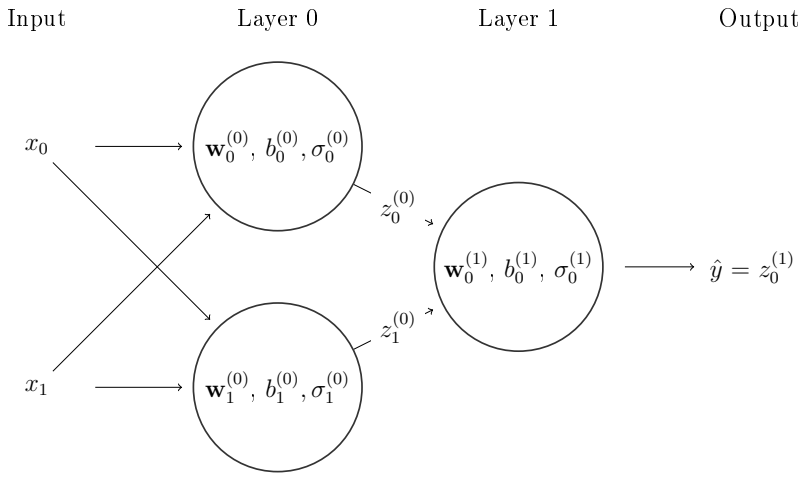

Figure 3.2.1: Simple two layer neural network.

For each neuron j in layer i, the activation is computed according to the following affine transformation:

$$a_j^{(i)} = \mathbf{w}_j^{(i)} \mathbf{z}^{(i-1)} + b_j^{(i)} \tag{3.2.1}$$

before the activation function $\sigma_j^{(i)}(\cdot)$ is applied to form the final output $z_j^{(i)} = \sigma_j^{(i)}(a_j^{(i)})$. In general we will assume *linear* or *relu* activation functions — as such, $\sigma_j^{(i)}(a_j^{(i)})$ is either $a_j^{(i)}$ or $\max(0, a_j^{(i)})$.

3.2.1 Find a configuration of the network in Figure 3.2.1 that approximates a logical *XOR* without any error. I.e., manually select activation functions (*relu* or *linear*) and weights and biases for which the input output relation mimics a logical *XOR*. Report your configuration and fill out Table 3.2.1, specifying the respective activations a and outputs z of each neuron. *Hint: Note, that the configuration is not unique.*

$\mathbf{x}^\top$	y	$a_0^{(0)}$	$z_0^{(0)}$	$a_1^{(0)}$	$z_1^{(0)}$	$a_0^{(1)}$	$\hat{y} = z_0^{(1)}$
$[0,0]$	0						
$[0,1]$	1						
$[1,0]$	1						
$[1,1]$	0						

Table 3.2.1: Network output and activations.

3.2.2 Use the chain rule to find analytical expressions for the derivatives of the loss $J(\mathbf{w}, b) = (y - \hat{y})^2$, with respect to $w_{0,0}^{(0)}, w_{0,1}^{(0)}, b_0^{(0)}$ and $w_{0,0}^{(1)}$, $w_{0,1}^{(1)}$, $b_0^{(1)}$ for an arbitrary choice of y and $\mathbf{x}$. Assume that each neuron has a *linear* activation function, but highlight the factors of the gradient that would change for a *nonlinear* one. Briefly describe the flow of gradient information throughout the network during optimzation — why can *backpropagation* be implemented efficiently?

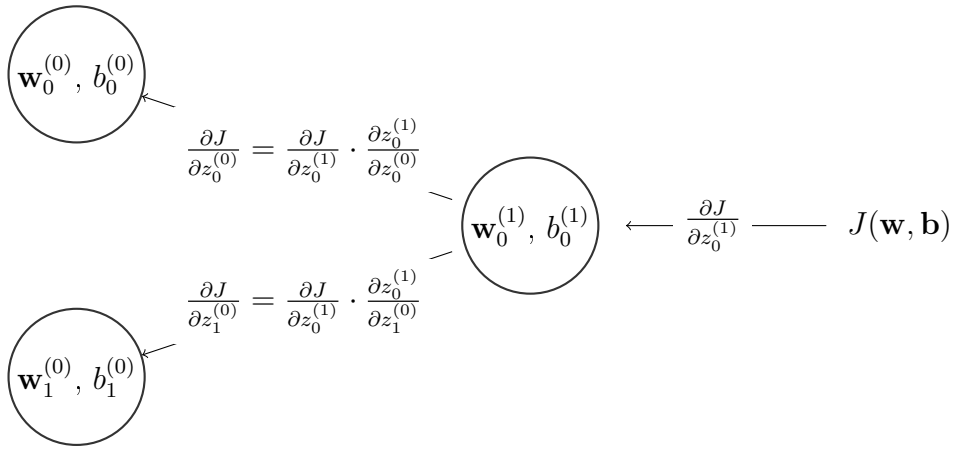

Figure 3.2.2: Flow of gradient information during backpropagation.

3.2.3 Use *keras* to implement the neural network from Figure 3.2.1, with a *sigmoid* activation function for the second layer. Train the network to approximate a logical *AND*. Finally, extract the obtained weights and biases from the trained network and again fill out Table 3.2.1. *Hint: You are free in your choice of an optimizer and loss function. Note, that it might be helpful to increase the learning rate.*

Problem 3.3 (2.5 points)

Learning curves are a particularly helfup tool to analyse the training process of a neural network. In the following we will implement the networks from Figures 3.3.1 and 3.3.2 and evaluate the obtained learning curves.

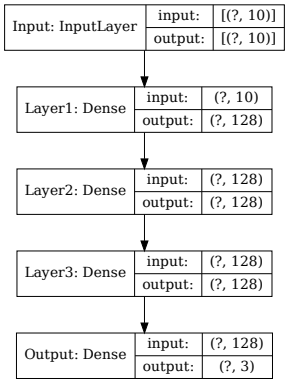

Figure 3.3.1: Baseline Model

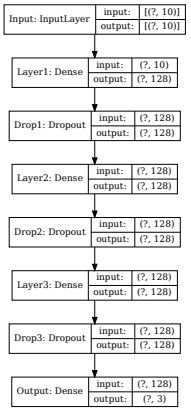

Figure 3.3.2: Dropout Model

Note, that both networks consist of 3 densely connected hidden layers and an output layer of dimension 3. The model in Figure 3.3.2 differs from the one in Figure 3.3.1 by the inclusion of dropout layers.

3.3.1 Implement both models using *relu* activation functions for the hidden layers and *softmax* for the output. Set the dropout rates to 0.5 for the model in 3.3.2. Print the output of *model.summary()* and explain how the number of trainable parameters is calculated for each layer.

3.3.2 Download the file *classification.csv* from *TUWEL* and transform the targets y into a one hot encoding. Subsequently split the data into T_{train} and T_{test} using a test size of 30% of the samples. For that, use *sklearn.model_selection.train_test_split* and set the *random_state* to 0.

3.3.3 Train each of the models for 200 epochs and report the accuracy of the trained networks on T_{train} and T_{test}. For that, use the *Adam* optimizer with default settings and a batch size of 64. Select *crossentropy* for the loss. Which model achieves a higher accuracy on T_{test}?

3.3.4 Adapt the parameters of *model.compile()* and *model.fit()* to keep track of *accuracy* and *loss* for T_{train} and T_{test} during training. Subsequently, repeat the training from Task 3.3.3 and plot the obtained learning curves. What do you observe?

3.3.5 Use the respective callback from *keras* to stopp the training of the baseline model shortly after the peak of the T_{test} *accuracy*. Report the configuration of *min_delta* and *patience* you used and plot the new learning curves.

3.3.6 Finally, revisit the *California Housing Dataset* from the first exercise and implement a neural network that achieves a MAE below 0.35 on T_{test}. For that, use *train_test_split* from *sklearn* with a *test_size* of 0.2 and set *random_state* to 0. Provide your network configuration and plot the learning curves of *loss* and *accuracy*. *Hint: It might be beneficial to standardize the input data.*