Problem 3.1 (25 %)

Consider the simple two layer neural network with three neurons in total depicted in Figure 3.1.1. Here, the input consists of the vector $\mathbf{x} = [x_0, x_1]^T$, while the output is given by the scalar $\hat{y}$.

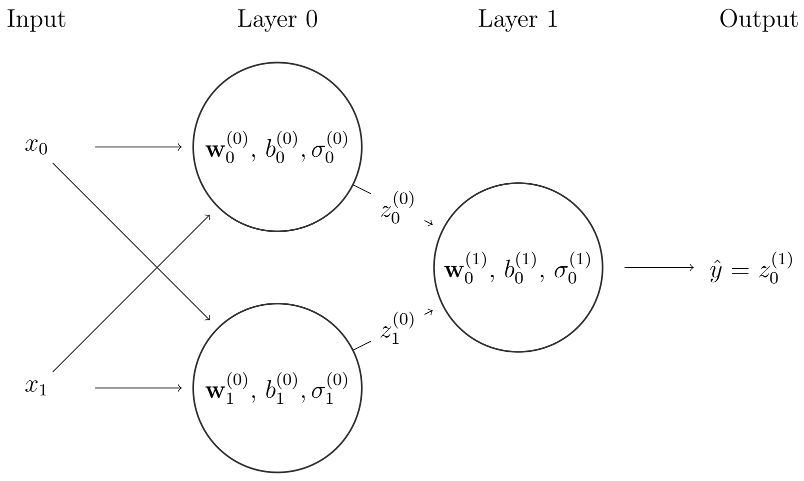

Input
Layer 0
Layer 1
Output
x_0
x_1
$\mathbf{w}_0^{(0)}, b_0^{(0)}, \sigma_0^{(0)}$
$\mathbf{w}_1^{(0)}, b_1^{(0)}, \sigma_1^{(0)}$
$z_0^{(0)}$
$z_1^{(0)}$
$\mathbf{w}_0^{(1)}, b_0^{(1)}, \sigma_0^{(1)}$
$\hat{y} = z_0^{(1)}$

Figure 3.1.1: Simple two layer neural network.

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

$$a_j^{(i)} = \mathbf{w}_j^{(i)} \mathbf{z}^{(i-1)} + b_j^{(i)} \tag{3.1.1}$$

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 the following, we will assume *linear* or *relu* activation functions. Hence, $\sigma_j^{(i)}(a_j^{(i)})$ is either $a_j^{(i)}$ or $\max(0, a_j^{(i)})$.

$\boxed{3.1.1}$ Find a configuration of the network in Figure 3.1.1 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 3.1.1, 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 3.1.1: Network output and activations.

$\boxed{3.1.2}$ Use the chain rule to find analytical expressions for the derivatives of the loss $L(\mathbf{w}, b) = (y - \hat{y})^2$, 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?

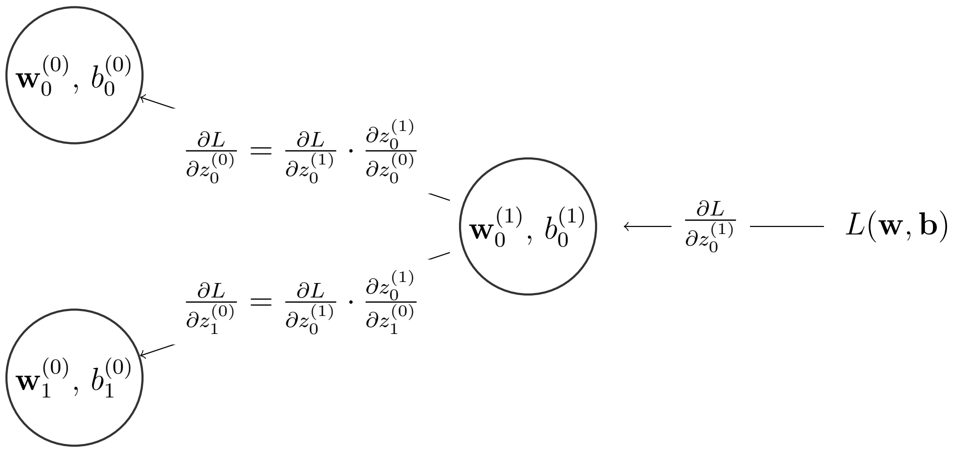

$\mathbf{w}_0^{(0)}, b_0^{(0)}$
$\frac{\partial L}{\partial z_0^{(0)}} = \frac{\partial L}{\partial z_0^{(1)}} \cdot \frac{\partial z_0^{(1)}}{\partial z_0^{(0)}}$
$\mathbf{w}_0^{(1)}, b_0^{(1)}$
$\frac{\partial L}{\partial z_0^{(1)}}$
$L(\mathbf{w}, \mathbf{b})$
$\frac{\partial L}{\partial z_1^{(0)}} = \frac{\partial L}{\partial z_0^{(1)}} \cdot \frac{\partial z_0^{(1)}}{\partial z_1^{(0)}}$
$\mathbf{w}_1^{(0)}, b_1^{(0)}$

Figure 3.1.2: Flow of gradient information during backpropagation.

$\boxed{3.1.3}$ Use *tensorflow.keras* to implement the neural network from Figure 3.1.1, 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 3.1.1. *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 3.2 (25 %)

Learning curves are particularly helpful to analyse the training process of a neural network. In the following we will implement the networks from Figures 3.2.1 and 3.2.2 and evaluate the obtained learning curves.

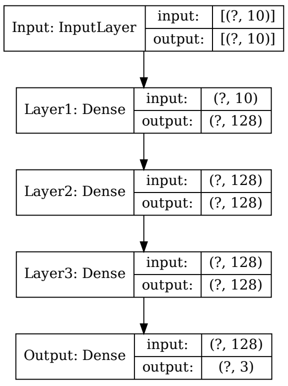

Input: InputLayer | input: [(?, 10)] | output: [(?, 10)]
Layer1: Dense | input: (?, 10) | output: (?, 128)
Layer2: Dense | input: (?, 128) | output: (?, 128)
Layer3: Dense | input: (?, 128) | output: (?, 128)
Output: Dense | input: (?, 128) | output: (?, 3)

Figure 3.2.1: Baseline Model

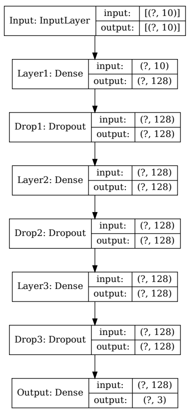

Input: InputLayer | input: [(?, 10)] | output: [(?, 10)]
Layer1: Dense | input: (?, 10) | output: (?, 128)
Drop1: Dropout | input: (?, 128) | output: (?, 128)
Layer2: Dense | input: (?, 128) | output: (?, 128)
Drop2: Dropout | input: (?, 128) | output: (?, 128)
Layer3: Dense | input: (?, 128) | output: (?, 128)
Drop3: Dropout | input: (?, 128) | output: (?, 128)
Output: Dense | input: (?, 128) | output: (?, 3)

Figure 3.2.2: Dropout Model

Note, that both networks consist of three densely connected hidden layers and an output layer of dimension 3. The model in Figure 3.2.2 differs from the one in Figure 3.2.1 by the inclusion of dropout layers.

$\boxed{3.2.1}$ 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 Fig. 3.2.2. Print the output of *model.summary()* and explain how the number of trainable parameters is calculated for each layer.

$\boxed{3.2.2}$ 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.

$\boxed{3.2.3}$ 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* as the loss function. Which model achieves a higher accuracy on T_{test}? *Hint: Make sure to select the correct crossentropy variant from tensorflow for the given problem.*

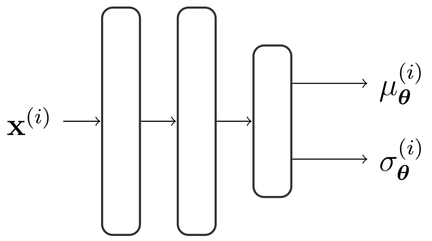
$\boxed{3.2.4}$ Adapt the parameters of *model.compile()* and *model.fit()* to keep track of *accuracy* and *loss* on the training T_{train} and the validation dataset T_{test} during training. Subsequently, repeat the training from Task 3.2.3 and plot the obtained learning curves. What do you observe?

$\boxed{3.2.5}$ Use the respective callback from *tensorflow.keras* to stop 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.

$\boxed{3.2.6}$ 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. *Hint: It might be beneficial to standardize the input data. Note, that transforming the outputs also changes the reported error metrics.*

Problem 3.3 (25 %)

It is often valuable to provide a measure of confidence together with the prediction of a neural network. For regression tasks, we can account for data uncertainty by following a probabilistic interpretation of the network output.


$\mathbf{x}^{(i)}$
$\mu_{\boldsymbol{\theta}}^{(i)}$
$\sigma_{\boldsymbol{\theta}}^{(i)}$

$$\begin{aligned} J_{\mathbf{y}}(\boldsymbol{\mu}_{\boldsymbol{\theta}}, \boldsymbol{\sigma}_{\boldsymbol{\theta}}) &= -\log \mathcal{N}(\mathbf{y}; \boldsymbol{\mu}_{\boldsymbol{\theta}}, \mathbf{I}\boldsymbol{\sigma}_{\boldsymbol{\theta}}) \\ &= -\log \prod_{i=1}^{N_{\text{train}}} \mathcal{N}\left(y^{(i)}; \mu_{\boldsymbol{\theta}}^{(i)}, \sigma_{\boldsymbol{\theta}}^{(i)}\right) \end{aligned} \tag{3.3.1}$$

As sketched above, this can be achieved by estimating the moments of a Gaussian distribution through a neural network with parameterization $\boldsymbol{\theta}$. The loss function is then given by the negative log-likelihood of the targets y, given the network outputs $\mu_{\boldsymbol{\theta}}$ and $\sigma_{\boldsymbol{\theta}}$. After training, we can then interpret the mean $\mu_{\boldsymbol{\theta}}^{(i)}$ as the network's estimate for input i, while $\sigma_{\boldsymbol{\theta}}^{(i)}$ represents the uncertainty associated with that prediction.

$\boxed{3.3.1}$ Expand the loss function $J_{\mathbf{y}}(\boldsymbol{\mu}_{\boldsymbol{\theta}}, \boldsymbol{\sigma}_{\boldsymbol{\theta}})$ from (3.3.1) and derive an analytical expression for the gradient with respect to a single network output $\mu_{\boldsymbol{\theta}}^{(i)}$ and $\sigma_{\boldsymbol{\theta}}^{(i)}$. Briefly interpret the obtained expressions — when are the individual gradients zero?

$\boxed{3.3.2}$ Download the *data_ uncertainty* regression dataset from TUWEL and generate a 2D scatter plot of training and test data, highlighting the respective target values through colors. Then, fit a neural network with two hidden layers to the dataset, using a scalar deterministic output with a *MSE*[1] loss function. Provide learning curves and visualize the predictions as well as the error for the individual samples, again using scatter plots. *Hint: Make sure to use X_ train_ coords and X_ test_ coords for visualization, but the standardized, polynomial expansions X_ train, X_ test as model inputs.*

$\boxed{3.3.3}$ Extent the above model to a probabilistic formulation, adapting the output layer such that it generates $\mu_{\boldsymbol{\theta}}$ and $\sigma_{\boldsymbol{\theta}}$. Make sure to use a suitable activation function, mapping $\sigma_{\boldsymbol{\theta}}$ to a valid range. You can then implement $J_{\mathbf{y}}(\boldsymbol{\mu}_{\boldsymbol{\theta}}, \boldsymbol{\sigma}_{\boldsymbol{\theta}})$ as a custom loss function. Herefore, you can reuse parts of the reference implementation available on TUWEL. *Hint: A function with signature loss_fn(y_ true, y_pred) can directly be passed to model.compile().*

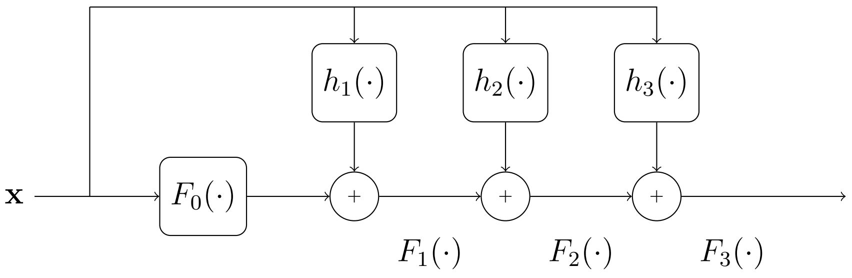
$\boxed{3.3.4}$ Fit the probabilistic model variant to the *data_ uncertainty* dataset, providing learning curves also including the average uncertainty $\sigma_{\boldsymbol{\theta}}$ and the *MSE* of the mean $\mu_{\boldsymbol{\theta}}$ through dedicated custom metrics. Then visualise the mean $\mu_{\boldsymbol{\theta}}$ and uncertainty $\sigma_{\boldsymbol{\theta}}$ outputs through separate scatter plots. Briefly interpret the results. *Hint: Note, that custom metrics are implemented equivalently to loss functions.*

[1]Mean Squared Error

$\boxed{3.3.5}$ Again, revisit the *California Housing* dataset and apply a neural network with sufficient capacity and the probabilistic treatment from above. Provide learning curves and compare the obtained MAE[2] to the results from the previous exercises. Then, select different quantiles of the predicted uncertainty $\sigma_{\boldsymbol{\theta}}$ as thresholds and compute the MAE only for predictions fulfilling these upper thresholds. Further compute the correlation coefficient between the absolute prediction error and $\sigma_{\boldsymbol{\theta}}$. *Hint: Use train_ test_ split with a test_ size of 0.2 and a random_ state of 0. Again, it can be helpful to standardize the dataset in advance.*

[2]Mean Absolute Error

Problem 3.4 (25 %)



Ensemble methods are machine learning techniques that combine the predictions of multiple individual models to improve overall predictive accuracy. In the following, we study the gradient boosting method to construct a strong learner $F_m(\cdot)$ out of an ensemble of $m \in \{1, \dots, M\}$ weak learners $h_m(\cdot)$. Starting from a base model $F_0(\cdot)$, we update the ensemble under a stagewise procedure with step size α:

$$F_m(\mathbf{x}_i) = F_{m-1}(\mathbf{x}_i) + \alpha \cdot h_m(\mathbf{x}_i). \tag{3.4.1}$$

Thereby, we select $h_m(\cdot)$, such that $F_m(\cdot)$ improves upon the previous model $F_{m-1}(\cdot)$. We interpret (3.4.1) as a gradient descent scheme, which yields that the negative gradient:

$$r_i^{(m-1)} = -\nabla_{F_{m-1}} L(y_i, F_{m-1}(\mathbf{x}_i)), \tag{3.4.2}$$

is a suitable update direction for $F_{m-1}(\cdot)$. For each step, the newly added weak learner $h_m(\cdot)$ is thus fitted to the so-called *pseudo residuals* $r_i^{(m-1)}$ from (3.4.2), before the ensemble is updated according to (3.4.1).

$\boxed{3.4.1}$ Consider a regression framework with inputs $\mathbf{x}$ and targets y under a simple squared loss function $L(\cdot) = (y_i - F_{m-1}(\mathbf{x}_i))^2$. Show that the negative gradient from (3.4.2) does indeed resemble the residuals up to a constant factor. Briefly interpret how the ensemble model $F_m(\cdot)$ is improved for each added weak learner $h_m(\cdot)$.

$\boxed{3.4.2}$ Download the *gradient_ boosting_ regression.csv* dataset from *TUWEL* and implement a gradient boosting regressor that can use arbitrary *scikit-learn* models as the weak learners $h_m(\cdot)$. Select the constant mean estimator $F_0(\cdot) = \frac{1}{N}\sum_{i=1}^{N} y_i$ as the base model and iteratively increase m to $M = 100$, by fitting $h_m(\cdot)$ to the residuals $r_i^{(m-1)}$, using a *DecisionTreeRegressor* with *max_ depth=1*. Visualize the regressor estimates $F_m(\mathbf{x}_i)$ for increasing ensemble sizes m, comparing it to the targets y_i. Additionally, provide learning curves of the MSE over m. *Hint: Select a constant step size of* $\alpha = 0.5$. *Consider an object-oriented style following the scikit-learn API.*

$\boxed{3.4.3}$ Switch the *DecisionTreeRegressor* for a *LinearRegression* scheme and compare the obtained ensemble to the results from Task. 3.4.2. Discuss whether a linear model is a reasonable choice for a boosting algorithm.

The gradient boosting method can also be extended to classification use cases by mapping the regressor output $F_m(\mathbf{x}_i)$ to class probabilities. For binary classification with the targets $y \in \{0, 1\}$, we compute

$$\hat{y}_i^{(m)} = \frac{1}{1 + e^{-F_m(\mathbf{x}_i)}} \in (0, 1), \tag{3.4.3}$$

and use the cross-entropy loss function:

$$L(\cdot) = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)). \tag{3.4.4}$$

$\boxed{3.4.4}$ Compute the gradient of the cross-entropy loss with respect to F_{m-1} and show that the *pseudo residuals* for the binary classification tasks are given by $r_i^{(m-1)} = y_i - \hat{y}_i^{(m-1)}$. *Hint: The derivative of the sigmoid function* $\sigma(\cdot)$ *can be written as* $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\cdot(1 - \sigma(x))$.

$\boxed{3.4.5}$ Download the *gradient_ boosting_ classification.csv* dataset from *TUWEL* and extend your implementation to the binary classification use case. As such, select the constant $F_0(\cdot) = 0$ estimator as the base model and iteratively increase the ensemble size by fitting a weak learner $h_m(\cdot)$ to the residuals from Task 3.4.4, using a *DecisionTreeRegressor* with *max_ depth=1*. Map the ensemble output to the class probabilities and report the accuracy over increasing ensemble sizes m with $M = 1000$. Further, visualize the decision surface through a heatmap plot.

$\boxed{3.4.6}$ Again, switch the *DecisionTreeRegressor* for a *LinearRegression* scheme and compare the obtained ensemble to the results from Task. 3.4.5. Briefly discuss how the obtained decision surface differs.