Problem 4.1 (2.5 points)

Convolutional layers are specifically desgined to handle structure in the input data — as such, they are often used for the processing of time-series and images.

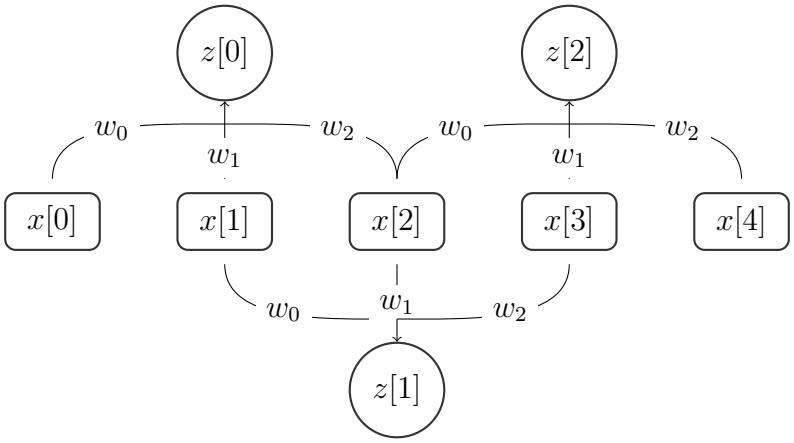

Figure 4.1.1: 1D-Convolutional Layer

At the core of convolutional layers is the concept of *weight sharing* — this means that the weights are reused for consecutive inputs. In a nutshell, the weights are collected in a filter $\mathbf{w}$ which is slided along the input $\mathbf{x}$ to generate the output $\mathbf{z}$ — see Figure 4.1.1.

4.1.1 The input-output relation of such a 1D convolutional layer consisting of a single filter $\mathbf{w}$, with a one-channel input $\mathbf{x}$ and a linear activation function is given by:

$$z_k = a_k = \sum_{m=0}^{M-1} w_m \cdot x_{k+m} + b. \tag{4.1.1}$$

Download the file *input_ output.py* from TUWEL and get familiar with the concept of multi-channel inputs and multiple filters by examining the *keras* template code and adapting the fixed weights and inputs.

Subsequently, consider the following layer configurations:

- 1D layer, nonlinear $\sigma(\cdot)$, two filters $\mathbf{w}^{(0)}$ and $\mathbf{w}^{(1)}$, one-channel input $\mathbf{x}$.
- 1D layer, nonlinear $\sigma(\cdot)$, single filter, two-channel input $\mathbf{x}^{(0)}$ and $\mathbf{x}^{(1)}$.
- 2D layer, nonlinear $\sigma(\cdot)$, single filter $\mathbf{W}$, one-channel input $\mathbf{X}$.

and derive respective analytical input-output relations. What changes for a different setting of *strides*?

4.1.2 During backpropagation the update for a single weight w_m or input element x_l is affected by multiple output elements z_k. As such, the gradients are given by the sum over all individual contributions:

$$\frac{\partial J}{\partial w_m} = \sum_k \frac{\partial J}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_m} \qquad \frac{\partial J}{\partial x_l} = \sum_k \frac{\partial J}{\partial z_k} \cdot \frac{\partial z_k}{\partial x_l}. \tag{4.1.2}$$

Consider the MSE loss function

$$J(\mathbf{y}, \mathbf{w}, \mathbf{x}) = \frac{1}{K} \sum_{k=1}^{K} (y_k - z_k)^2 \tag{4.1.3}$$

and find an analytical expression for the gradients in Equation 4.1.2. You can assume, that the input-output relation from Equation 4.1.1 is in place. Which mathematical operation do you obtain as the final result?

4.1.3 Finally, implement a network consisting of a single 2D-Convolution layer in *keras* and fix the weights of your network. Download the files *bike.png* & *edges.png* from TUWEL and convert them into *numpy* arrays. Use the images as inputs to your network and provide plots of the outputs for each of the filters $\mathbf{W}_{(0)}$ and $\mathbf{W}_{(1)}$. Briefly describe what you see.

$$\mathbf{W}_{(0)} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathbf{W}_{(1)} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{4.1.4}$$

Problem 4.2 (2.5 points)

In the following you will implement a CNN for the classification of handwritten digits from the MNIST dataset. The dataset is directly available in *keras* via the function *datasets.mnist.load_data()*.

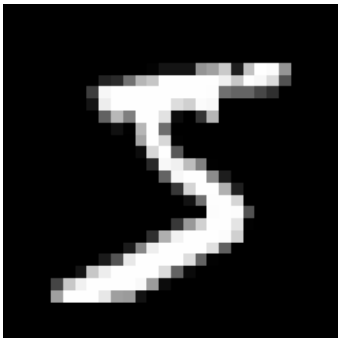

Figure 4.2.1: Handwritten 5

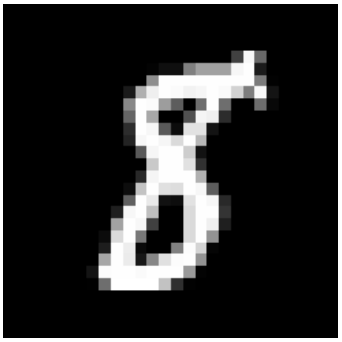

Figure 4.2.2: Handwritten 8

4.2.1 Download the MNIST dataset and plot one randomlly selected sample from each class. Subsequently, train a SVC with an RBF kernel and a regularization parameter of C=1 on the training dataset and report the accuracy on T_{test}.
Hint: You might want to normalize the data first. In case your machine is too slow — it is sufficient to only train with the first 10.000 samples from T_{train}.

4.2.2 Implement and train a neural network consisting of — amongst others — at least two 2D-Convolutional layers which outperforms the SVC on T_{test}. You are free in your model configuration and the choice of the optimizer and batch size. Provide a plot showing the learning curves for accuracy and loss on T_{train} and T_{test}.

4.2.3 Print the output of *model.summary()* and elaborate on how the number of free parameters and the output dimension is calculated for each layer of your model. Additionally, briefly discuss your choice for the activation function of the final layer.

4.2.4 Select 10 samples from T_{test} which your model failed to classify correctly. Plot the samples and report the respective predicted labels. Additionally, calculate the multiclass confusion matrix for your predictions and interpret the results.

4.2.5 Finally, select a random sample from T_{test} and extract the output of your trained model at the first and last 2D-Convolutional layer. At both layers, plot a heatmap of the output for a selection of up to 5 filters. Briefly describe how the outputs for the foremost and last convolutional layer differ.

Problem 4.3 (2.5 points)

kMeans (Lloyd's algorithm) is one of the most widely used clustering algorithms. It assigns each sample from the dataset $T = \{\mathbf{x}_n\}_{n=1}^{N}$ a cluster-label $i \in [1, \ldots, k]$, where k is the predefined number of cluster centers. The complete scheme consists of the following three steps:

1. **Initialization**
   Select an initial set of centroids $\mathbf{m}_1^{(0)}, \mathbf{m}_i^{(0)}, \ldots, \mathbf{m}_k^{(0)}$ uniformly at random from the data.

2. **Assignment**
   Each element $\mathbf{x}_n$ is assigned to the cluster defined by the set $S_i^{(t)}$ at iteration t:

   $$S_i^{(t)} = \left\{ \mathbf{x}_n : \left\| \mathbf{x}_n - \mathbf{m}_i^{(t)} \right\|^2 \leq \left\| \mathbf{x}_n - \mathbf{m}_{i^*}^{(t)} \right\|^2 \text{ for all } i^* = 1, \ldots, k \text{ with } i^* \neq i \right\}.$$

3. **Update Centroids**
   The new centroids are calculated according to $\mathbf{m}_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{\mathbf{x}_n \in S_i^{(t)}} \mathbf{x}_n$.

Note, that steps 2 & 3 are repeated until the algorithm converges to a state where the assignments do not change any more.

4.3.1 Write your own implementation of the *kMeans* algorithm. Consider writing your code in an object-oriented manner such that it is consistent with the *sklearn* API.

4.3.2 Download the file *blobs.csv* from TUWEL and fit your *kMeans* implementation with $k \in [2, 3, 4]$ to the dataset. For each variant, provide a scatterplot showcasing the clustering result. Make sure to highlight the centroids and their assigned samples. Consider a fixed choice of k — does *kMeans* always converge to the same solution?

4.3.3 Repeat Task 4.3.2 with $k = 3$ and visualize the training process of your implementation by providing a separate scatter plot for each iteration t. For that, manually choose samples from the region $x_0 > 4$ as the initial means $\mathbf{m}_1^{(0)}$, $\mathbf{m}_2^{(0)}$ and $\mathbf{m}_3^{(0)}$.

kMeans++ is a variant of *kMeans*, which utilizes a dedicated initialization algorithm. As such the first step of *kMeans* is replaced by the following scheme:

1. Choose the first mean $\mathbf{m}_1^{(0)}$ uniformly at random from the data.

2. Select the sample $\mathbf{x}'$ for $\mathbf{m}_i^{(0)}$ with probability $\frac{D(x')^2}{\sum_{x \in T} D(x)^2}$.

3. Repeat step 2 until all k centroids are initialized.

Here, $D(\mathbf{x})$ denotes the shortest distance from sample $\mathbf{x}$ to an already chosen centroid.

4.3.4 Extend your implementation to include initialization via *kMeans++* and again visualize the training process on *blobs.csv* by providing a separate scatter plot for each iteration t. What is the benefit of using the dedicated initialization scheme?

4.3.5 Finally, download the MNIST dataset via *keras* and cluster the samples using your *kMeans* implementation with *kMeans++* initialization and $k = 10$. Provide a histogram of the assigned cluster labels and plot each of the centroids. Note, that it is sufficient to cluster only the first 10.000 samples.