Assignment 1

Exercise 1.1

1. Linear regression and a perceptron can be compared by looking at the activation function in the perceptron. In linear regression, we model the relationship between input features $\mathbf{x}_i = (x_{1,i}, x_{2,i}, \cdots, x_{d,i})$ and an output y_i using a linear equation $y_i = w_1 x_{1,i} + w_2 x_{2,i} + \cdots + w_d x_{d,i} + \beta$, optimizing weights and bias to minimize the sum of squared errors. For a perception the output $f(\mathbf{x}_i)$ is the weighted sum of inputs and applies an activation function as $f(\mathbf{x}_i) = \textbf{activation}\,(w_1 x_{1,i} + w_2 x_{2,i} + \cdots + w_d x_{d,i} + \beta)$. Here the similarities become quite apparent as if we make this activation function a linear or an identity function, the perceptron basically mimics a linear relationship between inputs and output. However this is not typically the case as for neural networks the activation function is a non-linear function such as sigmoid or ReLU. Nevertheless, by using a linear activation function, a perceptron could perform the same computations as a linear regression.

2. The function corresponding to the dataset where $y_i = -\sin(0.8\pi x_i)$ can be seen in Fig 1[1].

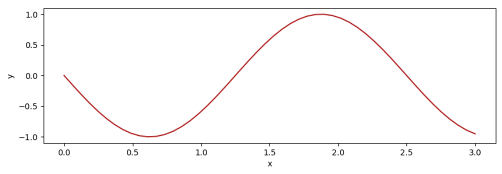


Figure 1: Graph of $y_i = -\sin(0.8\pi x_i)$

A linear model will not be able to correctly capture the relationship between the input and output data for the function $y_i = -\sin(0.8\pi x_i)$. This function is non-linear and a linear model, such as linear regression, assumes a linear relationship between the input and output variables.

With regards to over and underfitting, a linear model would likely underfit this data. Underfitting will likely happen when the model is too simple to capture the underlying dataset. Moreover, overfitting is quite unlikely as this happens when a model is too complex relative to the data and captures not only the underlying pattern but also the noise in the data. Here, a linear model is both too simple and the dataset has almost no noise, so overfitting is quite unlikely.

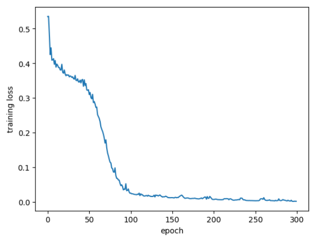
3. We can explore the model's learning efficiency when changing the learning rate. With the original learning rate of 0.2, the perceptron stabilizes at a loss of 0.4235, after 25 epochs. However, when the learning rate is decreased by a factor of 10 to 0.02, the model still achieves a similar loss of 0.4235, but it requires more epochs of 37, in this test. This slower convergence is simply due to the smaller step sizes in weight adjustments. Further decreasing the learning rate by a factor of 100 to 0.0002 shows an even more extreme example. The model only gets to a loss of 0.444, even after a significantly longer training duration of 2000 epochs. This higher loss and extended training period indicate inefficiency in learning at this very low learning rate. Additionally, the model takes about 10 epochs to reduce the loss by just 0.001, suggesting a very slow progression towards optimization.

If we, on the other hand, increase the learning rate, the learning behaviour changes notably. When the learning rate is set to 0.25, the model takes longer to settle compared to the original rate. It reaches the same loss of 0.4235 after 41 epochs. This shows that a learning rate of 0.25 is less efficient for this particular model and dataset. When the learning rate is increased even further to 0.3, the model's performance completely breaks, leading to a diverging loss. This is not unexpected behaviour when the learning rate is set too high, as it causes the model to overshoot the minimum of the loss function and fail to converge to an optimal solution.

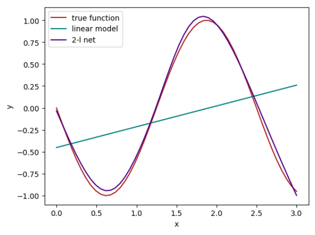
4. With the given setup, it is not guaranteed that each epoch will always decrease the loss. With this setup, we have 50 data points, so having `batch_size=25` we essentially divide the data into two batches, meaning that for each epoch the model's weights are updated two times. This larger batch

[1]The colab notebook had defined the dataset with $y_i = -\sin(0.8x_i)$ without π as specified in the exercises, so this was changed.

size can reduce the stochastic nature of mini-batch gradient descent. This means that we have a higher likelihood of observing a more consistent decrease in the loss function value with each epoch as each batch covers half the dataset and makes the gradient estimates more representative of the overall data. However, a consistently decreasing loss is still not guaranteed due to other factors such as the adaptive learning rate adjustments by the Adam optimizer and the possibility of overshooting.



(a) Training loss for the 2-layer model



(b) Predictions for the two models

Figure 2: Plots for the 1-layer and 2-layer model

We can see from the training loss in Fig. 2(a) that even though the 2-layer model becomes quite accurate at predicting, the training loss does not follow a nice curve as it stagnates slightly around 30-70 epochs. This can again be caused by the randomness of training these models.

Exercise 1.2

To calculate the new weights, we first have to do a forward pass with the data. To do this we first pass each datapoint to the hidden layer neurons (h_1 and h_2) separately to calculate their output. For the first datapoint $DP1 : (1, 1, -1)$ we get the outputs:

$$h_1 = \sigma(0.4 \cdot 1 + 0.2 \cdot 1 - 1.2 \cdot 1) = \sigma(-0.6) = \frac{1}{1 + \exp(0.6)} = 0.3543 \tag{1}$$

$$h_2 = \sigma(0.7 \cdot 1 - 0.7 \cdot 1 - 0.7 \cdot 1) = \sigma(-0.6) = \frac{1}{1 + \exp(0.7)} = 0.3318 \tag{2}$$

Now these can be passed as inputs to the final output neuron to predict $\hat{y}$.

$$\hat{y} = 0.4 \cdot 1 - 0.5 \cdot 0.3543 + 2.3 \cdot 0.3318 = 0.9859 \tag{3}$$

For the other two datapoints we get:

$DP2 : (0, -2, 1)$	$h_1 = 0.9427$	$h_2 = 0.8909$	$\hat{y} = 1.977$
$DP3 : (-1, 1, 1)$	$h_1 = 0.2689$	$h_2 = 0.6681$	$\hat{y} = 1.8022$

Now we can compute the loss L of the batch:

$$L = MSE = \frac{1}{3}\left((-1 - 0.9859)^2 + (1 - 1.9777)^2 + (1 - 1.8022)^2\right) = \frac{5.5432}{3} = 1.8477 \tag{4}$$

For the backpropagation, we want to compute the gradient of the loss L with respect to each weight w_k in the network.

$$\frac{\partial L}{\partial w_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_k} \tag{5}$$

To do this we use the chain rule to propagate the error from the output layer to the input layer.
We need the following equations

$$\frac{\partial L}{\partial \hat{y}_i} = -\frac{2}{N}(y_i - \hat{y}_i) \tag{6}$$

$$\frac{\partial \sigma(t)}{\partial t} = \sigma(t)(1 - \sigma(t)) \tag{7}$$

The first weight we calculate is weight from h_1 to $\hat{y}$, w_7. We update the weights by subtracting the partial derivative of the error with respect to w_7 multiplied by the learning rate (which is 1 in our case, so this will be ignored) so we get $\hat{w_7} = w_7 - \frac{\partial L}{\partial w_7}$. We first need to calculate $\frac{\partial L}{\partial \hat{y_i}}$ for all datapoints:

$$\frac{\partial L}{\partial \hat{y_1}} = -\frac{2}{3}(-1 - 0.9859) = 1.3239 \quad \wedge \quad \frac{\partial L}{\partial \hat{y_2}} = -\frac{2}{3}(1 - 1.977) = 0.6513 \quad \wedge \quad \frac{\partial L}{\partial \hat{y_3}} = -\frac{2}{3}(1 - 1.8022) = 0.5348$$

Now we calculate $\frac{\partial \hat{y_i}}{\partial w_k}$ which is equal to $h_1\sigma'(\hat{y}_{in})$ as all other inputs except from w_7 are removed by the derivative. In this case $\hat{y}_{in}$ is equal to the output for each datapoint as there output neuron is just a linear sum. We therefore calculate $\sigma'()$

$$\sigma'(\hat{y_1}) = 0.014 \quad \wedge \quad \sigma'(\hat{y_2}) = 0.045 \quad \wedge \quad \sigma'(\hat{y_3}) = 0.356$$

We can now finally calculate the new weight $\hat{w_7}$ as

$$\hat{w_7} = -0.5 - (1.3239 \cdot 0.3543 \cdot 0.014 + 0.6513 \cdot 0.9427 \cdot 0.045 + 0.5348 \cdot 0.2689 \cdot 0.356) = -0.5853 \tag{8}$$

We can do the same thing to w_8, the weight between h_2 and $\hat{y}$ to get: $w_8 = 0.6813$ and for the bias we can ignore the $h()$ as the value is 1 and get $w_9 = 0.79$ Now for the input layer we can calculate the weights as

$$\frac{\partial L}{\partial w_k} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{y_i}} \cdot \frac{\partial \hat{y_i}}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_i} \tag{9}$$

Where $\frac{\partial L}{\partial \hat{y_i}}$ is the same, $\frac{\partial \hat{y_i}}{\partial h_i}$ is $w_j\sigma'(y_{in})$ where w_j is the output weight from the hidden layer that this input neuron is connected to, e.g. w_7 or w_8. Finally the $\frac{\partial h_i}{\partial w_i}$ is equal to $x_i\sigma'(y_{in})$. We can now calculate the weight w_1 as the weight between the first input x_1 and the first hidden neuron h_1 as:

$$\begin{aligned}\hat{w_1} = 0.2 - (&1.3239 \cdot (-0.5 \cdot 0.014) \cdot (1 \cdot 0.014) \\ &+ 0.6513 \cdot (-0.5 \cdot 0.045) \cdot (0 \cdot 0.045) \\ &+ 0.5348 \cdot (-0.5 \cdot 0.356) \cdot (-1 \cdot 0.356)) = 0.1662\end{aligned}$$

We can do the same for the rest of the weights and get from x_1 to h_2 as $w_2 = -0.5456$. The weight from x_2 to h_1 as $w_3 = -1.1677$. The weight from x_2 to h_2 as $w_3 = -0.8502$. And for biases we can replace x_i with 1 as this is the input to get: The weight from *bias* to h_1 as $w_5 = 0.4352$. The weight from *bias* to h_2 as $w_6 = 0.5403$. To see if this improved the model we can do another a forward pass and calculate the MSE.

$$MSE_{new} = \frac{1}{3}\left((-1 - +0.1)^2 + (1 - 0.565)^2 + (1 - 0.36)^2\right) = 0.478 \tag{10}$$

As we can see, this MSE is lower than the original and the model is therefore improved on the training set.

Exercise 1.3

1. Increasing the noise in the dataset will have a big impact on the optimization process. It can lead to noisier gradients, making each step in the gradient descent potentially less precise when looking for the optimal path towards minimizing the loss function. This added noise will therefore often result in a less precise model as it has to optimize in a more irregular loss landscape.

   We can see the effect of noise in Fig. 3, where an increasing amount of noise negatively impacts the training MSE of a model. The optimization for the first few epochs stays quite consistent for all datasets indicating that the models are learning some general features. However, with a higher noise coefficient, the models' training stagnates earlier and earlier, suggesting that they are not able to capture all the features of their respective datasets.

2. Vanilla Gradient Descent (GD) and its stochastic and accelerated counterparts, such as SGD, Momentum, and Adam, show different convergence behaviors. Vanilla GD processes the entire dataset in a single update per epoch, leading to stable but slower convergence and difficulties in complex loss landscapes

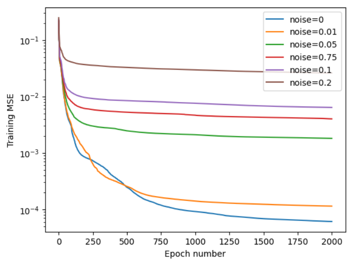


Figure 3: Training MSE with different values of `noise` for identical initial models.

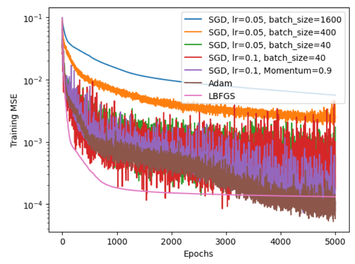


Figure 4: Training MSE for `Net3L(50, 50)` with different hyperparameters

as can be seen in Fig. 4. In contrast, SGD, particularly with smaller batches like size 40, updates more frequently, enabling faster but more variable convergence due to higher update variance, which can aid in escaping local minima and improving generalization. This frequent updating, however, increases computation time, especially with smaller batches, as seen in Table 1. Accelerated methods like Momentum and Adam, which build on SGD, introduce mechanisms for smoother and quicker convergence. Momentum accelerates SGD by leveraging past gradients, while Adam adjusts learning rates adaptively, making them more effective for practical applications. This effectiveness is evident in Fig. 4, where Adam and LBFGS show superior performance, with LBFGS leading initially up to 3000 epochs before Adam slightly overtakes, likely due to its noisier yet adaptive training approach.

3. The size of the neural network significantly influences the choice of optimizer, as different optimizers can have different behaves based on the model size. For smaller networks with e.g. `Net3L(10, 10)`, simpler optimizers might be sufficient. Training this model, the performance overall was worse than the original `Net3L(50, 50)` but the difference between optimizers was noticeably more similar. Even with the SGD with `momentum=0.9` outperformed Adam. However, as the network size increases, with more layers and neurons for `Net3L(100, 100)`, the complexity of the loss landscape increases. This required more sophisticated optimizers like Adam or LBFGS. These optimizers outperform in complex loss landscapes with more local minima Additionally, the computational time for each optimizer will vary with network size. As shown in table 1, larger networks generally require more time to train, regardless

Model Size	**SGD, lr=0.05 bs = 1600**	**SGD, lr=0.05 bs = 400**	**SGD, lr=0.05 bs = 40**	**SGD, lr=0.01 bs = 40**	**SGD, lr=0.01 M = 0.9**	**Adam**	**LBFGS**
10x10	113.87s	110.73s	286.26s	285.92s	109.59s	117.27s	146.11s
25x25	125.96s	112.85s	303.25s	304.35s	128.24s	135.20s	155.27s
50x50	123.45s	129.02s	317.63s	317.31s	129.41s	136.94s	171.31s
100x100	140.64s	140.86s	331.29s	326.77s	144.06s	154.67s	203.01s

Table 1: Training time comparison for different model sizes with various optimizers after 5000 epochs.

of the optimizer. However, the relative efficiency of different optimizers become more pronounced with larger models. We see that increasing the size of the hidden layers from 10×10 to 100×100 the Vanilla GD's training time is increased by 23.5% where the time for the LBFGS optimizer is increased by 38.9%. This makes the choice of optimizer more critical in achieving efficient training and convergence.

4. When evaluating the speed of different algorithms in training, it is important to realize the difference between the epochs and the actual. One epoch represents a full cycle of passing the entire training dataset through the learning algorithm. Convergence in terms of epochs, as shown in the training loss graph of Fig. 4, indicates how efficiently an algorithm processes and learns from the data. Fast convergence in epochs means the algorithm requires fewer passes through the data to achieve a certain level of accuracy or minimize loss. However, a smaller number of epochs doesn't always mean shorter training time, as the duration of each epoch varies significantly depending on the algorithm and other factors such as batch size and model complexity.

On the other hand, the actual time taken for training, as seen in the training time table 1, illustrates both the efficiency and computational requirements of the algorithm. This can illustrate many aspects such as the complexity of the algorithm's operations, the efficiency of the code, and the used hardware. It is important to note that time efficiency does not always correlate with a lower number of epochs, however, the example in the colab notebook has relatively linear correlation between epochs and time with e.g. the Adam model with hidden layer size of 50×50 of $[13s, 27s, 67s, 137s]$ for $[500, 1000, 2500, 5000]$ epochs respectively. In real world applications, the actual time is often more critical, which means that an algorithm that converges faster in terms of time rather than epochs is often more relevant.

5. The model has the following architecture:

$$conv2d \rightarrow max\_pooling2d \rightarrow conv2d \rightarrow max\_pooling2d \rightarrow flatten \rightarrow dropout \rightarrow dense$$

To count the total number of trainable parameters, we can ignore the pooling layers, the flatten layer and the dropout layer, as these layers do fixed operations and can not be trained. For a convolutional layer we can calculate the number of parameters as

$Conv_{params} = (KernelHeight \times KernelWidth \times InputChannels + 1) \times Filter_{num}$

So for the two convolutional layer we get:

$$Conv\_1_{params} = (3 \cdot 3 \cdot 1 + 1) \cdot 32 = 320 \quad \wedge \quad Conv\_2_{params} = (3 \cdot 3 \cdot 32 + 1) \cdot 64 = 18496$$

Finally, the number of parameters for the dense layer is:

$$Dense_{params} = (InputUnits + 1) \times OutputUnits$$

To find the input units we trace the spatial size through the model. The first convolution reduces the size to $28 - (3 - 1) = 26 \times 26$. The max pooling of 2×2 reduces it to 13×13. The next convolution reduces it to 11×11 and finally the max pooling gets it down to 5×5. Then finally we have 64 channels so the input to the dense layer is $5 \cdot 5 \cdot 64 = 1600$. The output units are simply the 10 classes so we get: $(1600 + 1) \cdot 10 = 16010$ Finally the total number of trainable parameters is the sum of these:

$$Total\_params = 320 + 18496 + 16010 = 34826$$

6. Replacing the Adam optimizer with SGD and then Adadelta in the model yielded interesting results. The model with Adam achieved a good test accuracy of 0.9905 and a low loss of 0.0275, switching to SGD still resulted in strong performance, with my best found being with a learning rate of 0.01 and momentum of 0.4 achieving a test accuracy of 0.9792 and a loss of 0.0665. This slight drop in performance indicates that while still good, SGD might not be as efficient as Adam in navigating the model's complex loss landscape or may require more fine-tuning. Subsequently, using the Adadelta optimizer with a learning rate of 0.1, rho of 0.95, and epsilon of 1e-07, got the best results for this optimizer. The model achieved a similar level of performance with a test accuracy of 0.9786 and a loss of 0.0659. Adadelta's particularity is with its approach of handling the diminishing learning rates of Adagrad. By accumulating a window of past squared gradients, Adadelta adapts the learning rate, making it less aggressive and more reliable. These results show that while Adam might offer the best initial performance in this case, other optimizers like SGD and Adadelta can still achieve great results and might be able to perform even better with the right hyperparameter tuning.

Exercise 1.4

1. The reason we are interested in having separate dataset for training and testing is in order to better evaluate the performance of a model accurately. The training dataset is used to develop a model by allowing it to learn patterns within the data. The testing dataset consists of data that the model has not previously seen or been trained on. By having a completely isolated dataset for training we can better ensure that the performance of the model is more generalized and not just works for the training data. This separation in data is useful for detecting potential overfitting, where a model learns the training dataset too well due to memorization of specific patterns, but this will make the model perform poorly on new data due to a lack of true learning and generalization capability. The plot of the surface associated with the training set can be seen in Fig. 5.

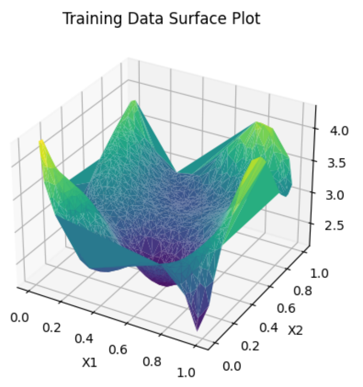


Figure 5: surface associated to the training set with `dataset_generation(9, 9, 7, 6, 5)`.

2. To find an optimal model I explored a wide combination of architectures, experimenting with different learning algorithms such as SGD, Adam, AdamW and RMSprop, and testing various transfer functions like ReLu, sigmoid and tanh. The model I found with the best results has three layers: an input layer with batch normalization, two hidden layers with 20 and 10 neurons respectively, and an output layer. The activation function for the hidden layers is ReLU. The optimizer used is SGD with a momentum of 0.9, which helps in accelerating the gradients vectors in the right direction. The model stopped training after 32 epochs as this hit the early stopping limit of `patience=5` For validation, I used a validation split of 20% to evaluate the model's performance.

3. The performance of the network on the test set was evaluated using the MSE. The final test error was found to be 0.01038, which indicates a good performing model. A visualization of the performance can be seen in Fig. 6. While there are some visual differences, ht is clear that the overall structure is

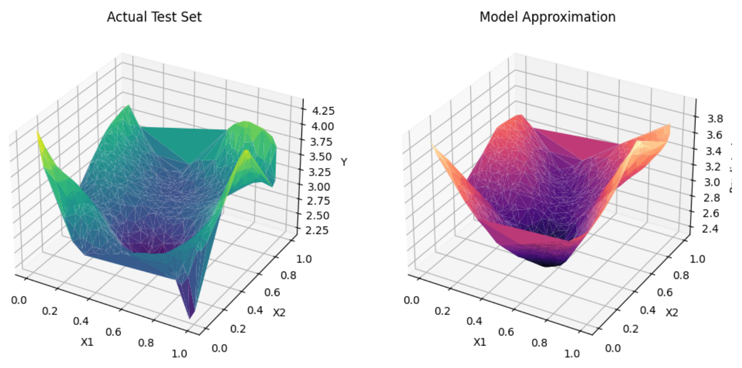


Figure 6: Testing data and the models prediction.

captured by the model.

The reason for not training further is due to the model reaching an optimal state where further training does not significantly improve performance on the validation loss. The patience was set to 5 to allow for slight progression once the performance stagnated but before potential overfitting.

4. The regularization strategy used in this model is batch normalization. This normalizes the inputs to each layer, which helps in stabilizing the learning process and works to reduce the chance of overfitting. As the inputs to a layer are normalized, the model becomes less likely to rely heavily on any particular feature or neuron, thus reducing overfitting.

   Another effective regularization strategy that also performed quite well in my testing was dropout. Here randomly selected neurons are ignored during training. This prevents the network from becoming too dependent on any specific neurons thus improving generalization. Additionally, techniques like L1/L2 regularization on weights are also effective in preventing overfitting.

Assignment 2

Exercise 2.1

Brussels

1. For time-series prediction on the climate dataset, I have chosen Brussels as the first city. To model and forecast the average temperatures, I have used Support Vector Regression (SVR) as this achieved good results. SVR is a type of Support Vector Machine that is used for regression tasks. The goal of SVR is to create a model that predicts the target values of our dataset, while maintaining a balance between the model's complexity and its ability to fit the data well. One of the benefits of SVR is its ability for fine-tuning with many hyperparameters. For this exercise I mainly focused on the kernel, C, epsilon, and Gamma.

2. For the kernel I used the Radial Basis Function (RBF). The RBF kernel is particularly good at handling non-linear data. Furthermore, as the dataset only consists of average temperatures, and therefore only has a single feature, many other kernels struggled to get a good performance, sot RBF was the best for this dataset. For the hyperparameters, I chose a regularization parameter (C) of 0.4, an epsilon (ϵ) of 0.2, and a 'scale' setting for gamma as this likewise achieved the best overall results. These parameters helped to make the model more robust to significant temperature trends while avoiding overfitting to minor changes.

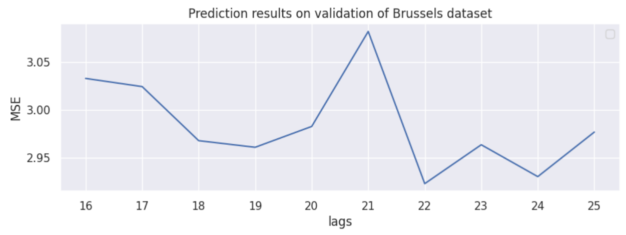


Figure 7: MSE for different lags on the training set

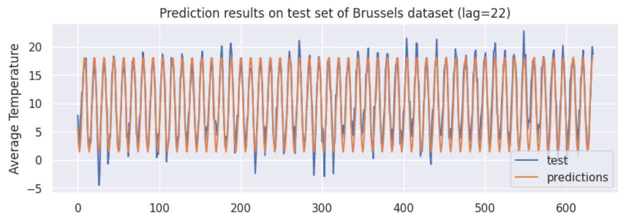


Figure 8: Prediction results on the test set

To optimize the model further I explored different lags and found that a lag of 22 gave the lowest MSE on the training set as shown in Fig. 7. After this point, the lag stagnated and did not improve the MSE much further. The performance of the SVR model, with these settings, was tested on the test set, and can be seen in Fig. 8 with a test MSE of 3.633. To achieve even better performance, it might help a lot to have more features from the dataset than just the average temperature so we have more time-series to work with for a more nuanced dataset. A more structured grid search of hyperparameters might also be able to give an improvement in performance.

Rio De Janeiro

1. For the 2nd city I have chosen Rio De Janeiro. This dataset seemed to have much more irregularities and was more noisy in between peaks and vallies in the data. For this dataset, I decided to use a Random Forest approach for modeling and forecasting. Random Forest work by constructing multiple decision trees during training and outputting the average prediction of the individual trees, which makes it highly suitable for handling complex, irregular datasets. Random Forest model can likewise handle non-linear data and capture irregular patterns, which could become important for this dataset. This approach also offers some hyperparameters for fine-tuning. In this exercise, I focused on adjusting parameters such as the number of trees in the forest and the maximum depth of the trees.

2. For the specific implementation of the Random Forest Regressor, I used the following hyperparameters: `n_estimators=100`, indicating 100 trees in the forest and `max_depth=10`, which limits the depth of each tree to 10. These parameters were able to achieve the best balance between the model's ability to capture complexity and preventing overfitting.

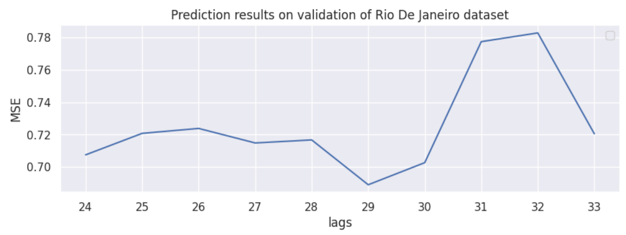


Figure 9: MSE for different lags on the training set

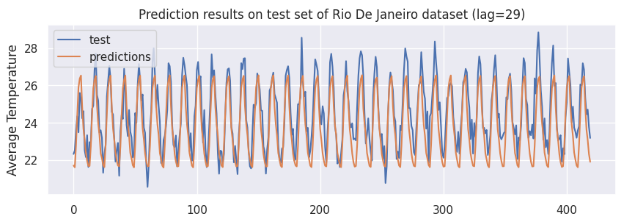


Figure 10: Prediction results on the test set

Further tuning of the model was done by trying different lag values. Here, I found that a lag of 29 provided the lowest MSE on the training set, as shown in Fig. 9. The performance of the Random Forest, set with these parameters and the optimal lag, was evaluated on the test set, which can be seen in Fig. 10. The model achieved a test MSE of 1.158, indicating the model's effectiveness. Again, this performance might further be enhanced by including additional features beyond average temperature. We can see in Fig. 9 that the prediction is very symmetric and repeating which could be because the data is too limited for a tree-based model.

Exercise 2.2

1. For the Ionosphere dataset, I trained a MLP classifier, where I focused on hyperparameter optimization for performance. This involved experimenting with the following configurations:

   - solvers = [”lbfgs”, ”sgd”, ”adam”]
   - activations = [”relu”, ”tanh”, ”logistic”, ”identity”]
   - hidden_layers = [(20, 20,), (20, 50,), (50, 50,), (50, 20,), (100, 100,), (50, 100, 100)]

   Each combination of solver type, activation function, and hidden layer size was evaluated for performance on a validation set. The results, shown in 2, highlight the best-performing models.

Solver	Activation	Hidden Layers	Validation accuracy
SGD	ReLU	(50, 20)	0.9271
Adam	ReLU	(20, 50)	0.9282
Adam	ReLU	(50, 100, 100)	0.9286
SGD	ReLU	(50, 20)	0.9432
Adam	ReLU	(100, 100)	0.9464

Table 2: Accuracy on Validation Set

   Looking at these results, its interesting that it is not completely similar models that perform the best, but different solvers can achieve great performance. Running the test set with the best performing model:

   `{Solver: Adam, Activation: ReLU, Hidden_dim: (100, 100)}`

   gave an accuracy of 0.930 which shows that the model is performing great.

2. Training on the Diabetes dataset, requires different considerations as it is multidimensional and somewhat imbalanced. For hyperparameter tuning I initially tried the ones that worked best for the Ionosphere dataset – consisting of Adam solver, ReLU activation function, and a hidden layer configuration of (100, 100). Interestingly, these parameters did not perform well with a validation accuracy of just 0.634. This illustrates that there are no configuration that works best for anything. To find better parameters I again did a grid search, exploring the same parameters as mentioned above. The best performing model for this dataset had the following hyperparameters:

   `{Solver: SGD, Activation: Tanh, Hidden_dim: (50, 20)}` with an accuracy of 0.724 on the validation set and 0.697 on the test set. This accuracy is worse than the one on the Ionosphere dataset which indicates that the Diabetes dataset is likely more complicated. Furthermore, it is worth noting that these results differ slightly depending on the model initialization. Even creating a deepcopy of the initial weights for all parameters will not be relevant for overall architecture. However, the specified hyperparameters consistently performed in the top of the tests.

3. Running the ordinary least squares (OLS) estimator and the Automatic Relevance Determination (ARD) on the synthetic dataset shows some interesting results. The ARD model clearly shows a better capability in identifying relevant features, as shown in the weight comparison Fig. (11). The estimates of OLS are very noisy which will likely make it less general for new samples. This shows its capabilities of effectively moving irrelevant feature weights towards zero. This aspect can be even more clearly shown in the histogram of the estimated weights (12), where the sparsity-inducing of ARD is has a high concentration of weights near zero. The relevant features, shown in the histogram with yellow dots, likewise clearly overlap with more weights being located at these values. By focusing on a limited number of relevant features, this model can get a high accuracy and increase the interpretability of the model.

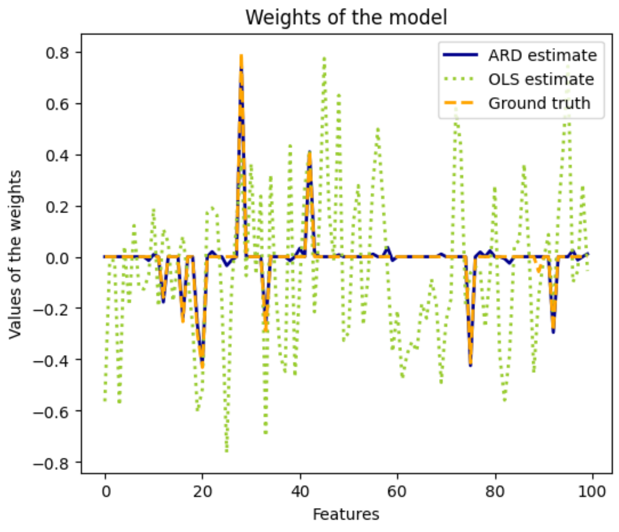


Figure 11: Weights of the models

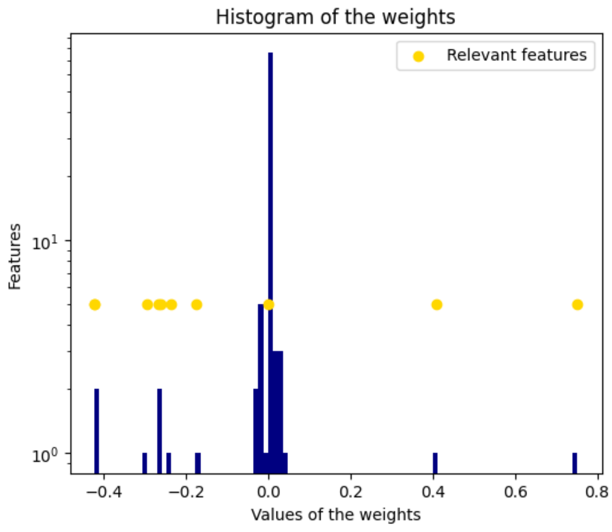


Figure 12: Histogram of the weights for the ARD

4. For the Diabetes dataset, we use the ARD for the binary classification task by treating it as a regression problem with targets [0,1]. This approach allowed for the identification of the most relevant input dimensions for the prediction task.

   The ARD regression identified dimensions [0, 5, 6] as the most significant contributors to the prediction results as can be seen in Fig. 13. This results can then be used for an MLP classifier where we only consider a reduced dataset that only contains the relevant dimensions found by ARD.

   Training two models with identical hyperparameters gave some interesting results, as the MLP model trained on the full dataset achieved an accuracy of 0.617 while the model train on the dimension-reduced dataset (of dimensions [0, 5, 6]) achieved a better accuracy of 0.708. These results illustrate that by choosing the most relevant features from a dataset, we can achieve better performance as we reduce potential noise or irrelevant correlations.

Exercise 2.3

1. In this example we have a 2-dimensional input where we want to represent the underlying distribution with our centroids being the `som_shape = (4, 4)`. With the Self-Organizing Map (SOM), the centroids

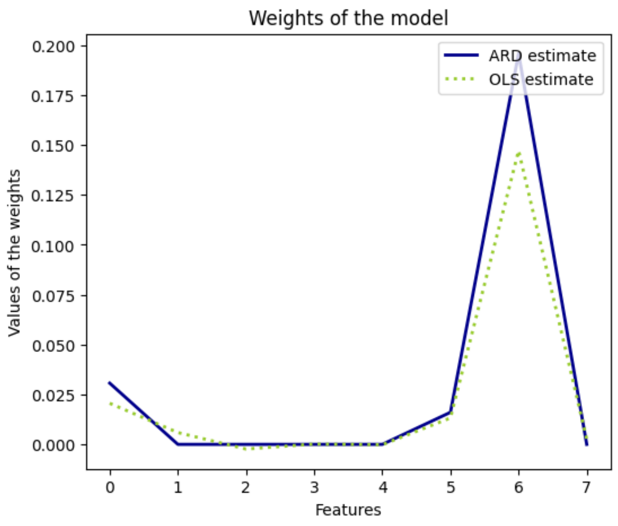


Figure 13: Relevant weights of the model

are initially places randomly within the specified feature space boundaries of [-1,1][-1,1] for both x and y coordinates.

As the SOM trains, these centroids move to represent the underlying distribution of the data. During training, each data point is assigned to its nearest centroid, and then the centroid positions are updated. However, some centroids do not move onto the two moons' structures, as can be seen in Fig. 14. For all iterations – even 1000 – we have some centroids that are not converging on the moons.

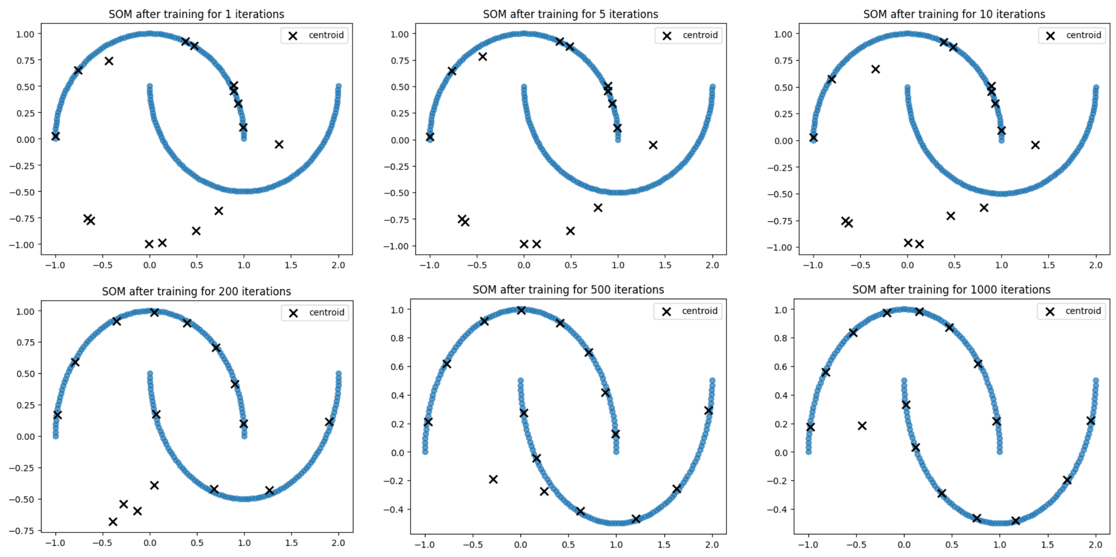


Figure 14: SOM after different iterations

This is likely because no data points are assigned to them, resulting in no updates to their positions. This is a common aspect of SOMs where some neurons may not effectively capture any portion of the data distribution, especially in cases of unevenly distributed data.

2. The U-matrix is a great way of visualizing the distances between neurons and revealing the underlying data structure. The U-matrix in Fig. 15 shows two significant dark lines. This likely shows a distinct separations between clusters in the dataset. These darker lines represent areas with few neurons,

suggesting that these are boundaries between different classes or clusters in the dataset.

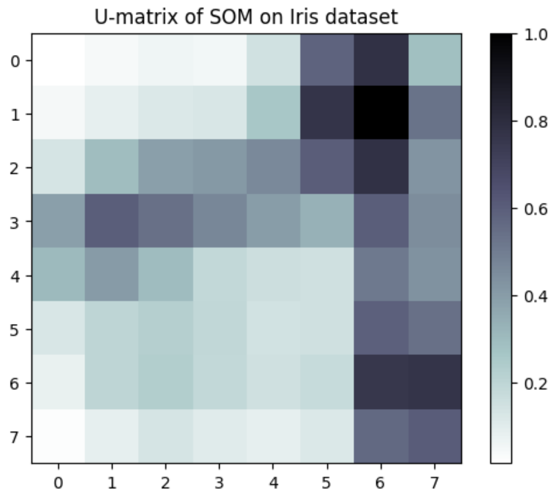


Figure 15: U-matric for the Iris dataset

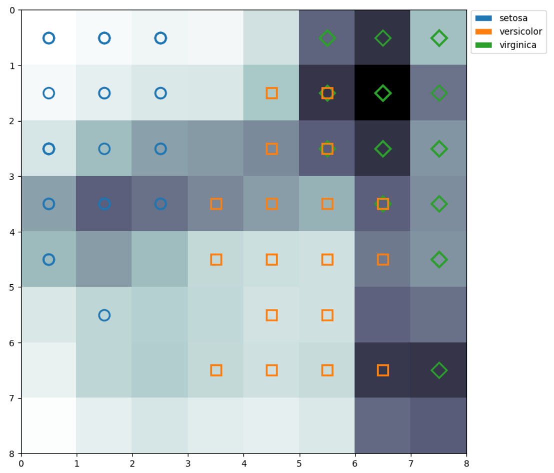


Figure 16: The samples on the SOM output layer

When looking at the overlaid training samples on the SOM's output layer in Fig. 16, it becomes more clear that the samples from the three Iris classes are placed in separate regions. Furthermore, the lighter areas in the U-matrix are where neurons are closely placed to their neighbors, indicate regions with many samples. This separation of dark lines between the light regions shows the effectiveness of the SOM in capturing the distribution and features of different classes within the dataset.

3. The SOM is configured into a 1x3 grid (`som_shape = (1, 3)`), effectively creating three clusters. For clustering, each centroid of the SOM was used as a separate cluster. In the plot in Fig. 17, each cluster is colored, and the centroids of these clusters are marked with 'x'. This clearly shows how the data points are distributed among the three clusters generated by the SOM.

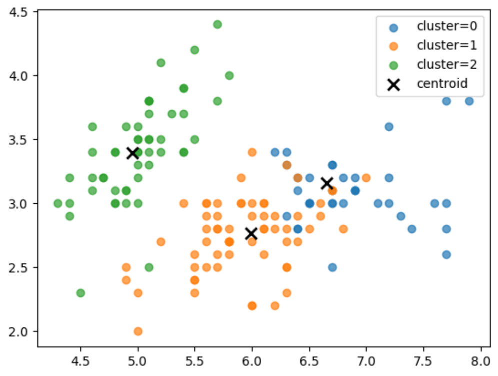


Figure 17: Clusters in SOM

To evaluate the clustering performance we can use Normalized Mutual Information (NMI). NMI is a measure based on mutual information where 1 indicates perfect clustering and 0 indicates no mutual information between the clustering and the true labels. NMI is a useful measure of clustering when the true labels are known, as it gets the amount of information obtained about one random variable through observing the other random variable. In my tests, the best NMI score I was able to achieve between the true labels (`y`) and the clustered indices (`cluster_index`) was 0.813 with `{'neighborhood_func': 'gaussian', 'sigma': 1.0, 'lr': 0.1}`. This score shows a good correlation between the clusters formed by the SOM and the actual classes in the Iris dataset. The closer the NMI score is to 1, the better the clustering performance is in terms of accurately depicting the data. Here, 0.813 indicates a good but not perfect alignment between the SOM's clusters and the Iris dataset's true class distribution.

4. Classification using SOMs is achieved through first mapping and then labeling. First, the SOM is trained with unlabeled data, where it organizes itself as it is typically used for. After training, each data point is classified based on the closest neuron's class. If a neuron doesn't have a specific class, the data point is assigned to the most common class from the training data.

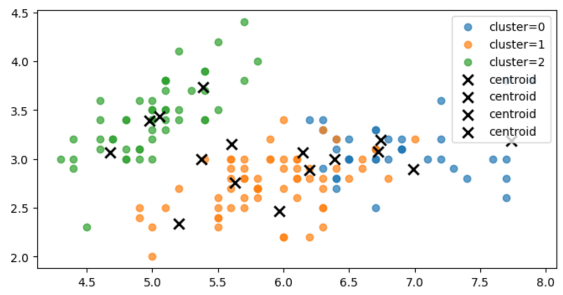
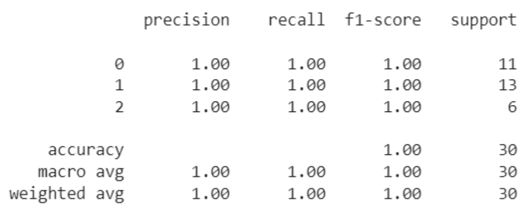


Figure 18: U-matric for the Iris dataset



```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         6

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Figure 19: The samples on the SOM output layer

For the classification of the Iris dataset i used the following parameters to achieve the best results: `{'x': 4, 'y': 4, 'neighborhood_func': 'triangle', 'sigma': 2, 'lr': 0.5}`

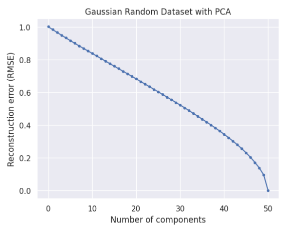
Using this 4x4 SOM, I achieved 100% accuracy, as shown in Fig. 19. However, this SOM has 16 centroids, which are much more than the three classes in the dataset. This will likely lead to overfitting and reduced generalizability, as can be clearly seen in Fig. 18. So for this example, a worse training performance would actually be a better choice.

Assignment 3

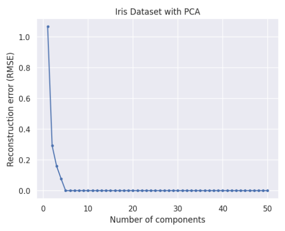
Exercise 3.1

1. In the reconstruction error curve – displayed in Fig. 20(A) – for the Gaussian random dataset, we can see what looks to be a generally linearly decreasing trend in the Root Mean Square Error (RMSE) as the number of components increases from 0 to 50. This trend shows that as more principal components are included in the reconstruction of the dataset, the accuracy of the reconstruction improves, leading to a lower RMSE as we project the data onto a dimension that resembles the original data more and more. With zero components, the RMSE is at its maximum, close to 1, signifying a high error in reconstruction as this is not able to capture any variance in the data. With 50 components – the same dimensionality as the original data – the RMSE is at 0 as the reconstructed data is essentially protected onto the same dimensionality as the original

   The linear decrease in RMSE suggests that each additional component contributes almost equally to reducing the reconstruction error. This is characteristic of datasets where the variance is spread out relatively evenly across all dimensions, as is often the case with Gaussian random data. With this Gaussian dataset, the data in all 50 dimensions are generated the same way and all resemble a Gaussian distribution, meaning that the variance is essentially equally spread out over all dimensions. This means that by reducing the dimensionality the RMSE is linearly affected by the number of dimensions removed.



(A) Components for the Gaussian dataset



(B) Components for the Iris dataset

Figure 20: Reconstruction error for different number of PCA components on datasets

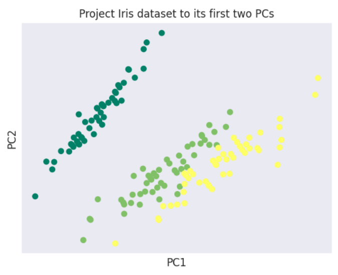
2. As the Iris dataset only has 4 features, any higher value in the number of components in the reconstructed dataset has an RSME of 0 as we can perfectly recreate the data. This can be seen in 20(B) With 0 components the RSME is close to 1, then it drops down and from *components* ≥ 4 the RSME is 0.

   We can also observe here that the decrease in RSME is inversely proportional to the components. Meaning that the closer we get to the true number of features (4) the less impact adding another component has as we add the least impactful component last.

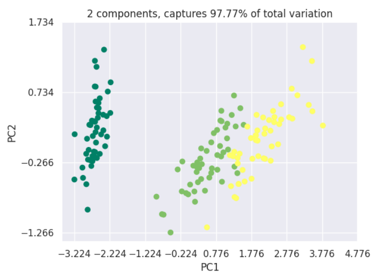
   To determine the most suitable number of components for the Iris dataset using PCA we need to optimize between dimensionality reduction and the preservation of information in the variance. The Iris dataset has only 4 features, so the maximum number of principal components that can be extracted is also 4. However, the goal of PCA is often to reduce the number of dimensions while retaining as much of the significant variance in the data as possible. Choosing fewer components simplifies the model but likely leads to more information loss. On the other hand, more components can capture more details but may include noise and reduce the model's generalizability. For this dataset, a suitable number of components could be 2 as this reduces the dimensionality by a lot while still having a relatively low RMSE for the reconstructed dataset. Furthermore reducing the dimensionality to 2 dimensions makes it convenient for development as this is easy to visualize.

   Comparing the reconstruction error curve for the Gaussian dataset and the Iris dataset shows the most noticeable difference in how the removal of dimensions affects the two completely different. As mentioned above, RMSE of the reconstructed error curve for the Gaussian dataset decreases linearly for each component added, whereas for the Iris dataset, the RMSE is inversely proportional to the number of components. What we see in the Iris dataset is a more typical feature of PCA as it is a real dataset, so the variance can drastically differ from dimension to dimension. For a real-life dataset of n dimensions, reducing the dimensionality to $n-1$ will likely not affect the reconstruction that much, as the removed dimension is the one that shows the least variance over the entire dataset. The closer we get to a dimensionality of $n=1$ the more impact each removed dimension will have on the RMSE as we remove drastically more and more impactful features.

When applying PCA to the Iris dataset using scikit-learn we can observe general similarities to the manual implementation, but with slight variations. Although the overall shape is quite similar, as shown in Fig. 21, the principal components identified by scikit-learn are slightly skewed compared to those from a manual model. This variation is not surprising and is likely due to differences in computational methods or data preprocessing.



Manual PCs



Scikit-Learn PCs

Figure 21: Iris data for the first two PCs for the manual implementation and the Scikit-Learn implementation

Exercise 3.2

Autoencoders

To explore the performance of image reconstruction on the MNIST dataset different hyperparameters were used. More specifically various combinations of training epochs (ranging from 1 to 10) and the encoding dimensions (ED) (ranging from 16 to 128) to analyze their effects on image reconstruction quality. The configurations and their training outcome are summarized in Table 3.

Encoding Dims	Epochs	Train Loss	Training Time
16	20	0.635804	3m 34s
16	100	0.629790	17 m51s
64	1	0.438360	0m 11s
64	5	0.107437	0m 55s
64	20	0.096065	3m 43s
128	20	0.037145	3m47s

Table 3: Experimental Configurations for the Autoencoder

From this data, several observations can be made.
First of all, increasing the encoding dimension greatly improves performance across all testing epochs This becomes most evident when considering that for the 128 ED model, the training loss is already at 0.075 after just the first epoch. This observation illustrates that the more we narrow the 'bottleneck' the more difficult it becomes to reconstruct the data.
We can also see that increasing the epochs from a low number (of 1 or 5) to around 20 leads to great performance enhancement as can be seen in the training loss. However, for the 16 ED model, extending the epochs from 20 to 100 only seems to minimally affect the performance. Notably, for all model configurations, the training loss appears to stabilize around 15-20 epochs which suggests that this range is sufficient.

Moreover, regardless of the encoding dimension, models that is trained with the same number of epochs show similar training times of approximately 3 minutes and 40 seconds. This shows that the primary factor in the training time is the number of epochs.
The training loss can only reveal so much about the models' performance, as we are dealing with images in the MNIST dataset. For a more intuitive assessment, we can look at a visual representations of the results, as depicted in Fig. 22.

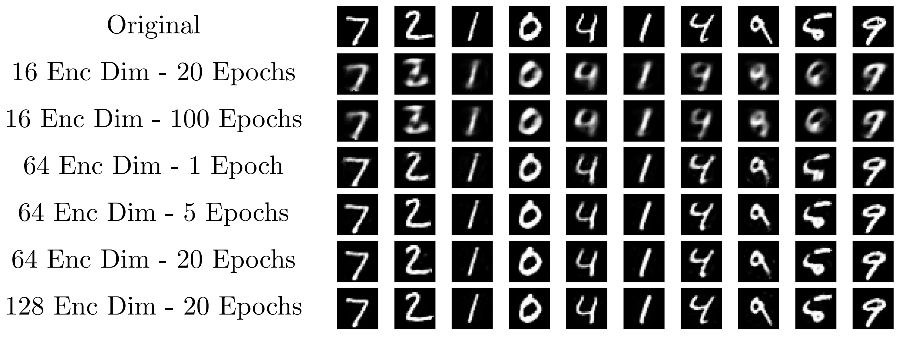


Figure 22: Evaluation Results for Different Configurations

Visual inspection of the results allows us to intuitively assess the performance of different configurations. The worst-performing configurations tend to show a more blurry reconstruction, which is visually evident. Specifically, the configuration with 16 encoding dimensions and 20 epochs unsurprisingly produces the least visually accurate, characterized by significant blurriness.
In comparison, the 64 encoding dimensions model with just 1 epoch shows a noticeable degree of blurriness, but as the number of epochs increases to 5, 20, or even 128 encoding dimensions, the visual quality becomes nearly indistinguishable. There may be subtle improvements in terms of slightly sharper edges with the 128 encoding dimensions configuration, although these differences are relatively minor.
To further improve reconstruction results – beyond adjusting the encoding dimension (ED) or training epochs – several alternative approaches can be explored. Employing deeper architectures for both the encoder and decoder can help the model capture more intricate features in the data. Additionally, more advanced techniques such as batch normalization, dropout, or L2 regularization can help to prevent overfitting and contribute to improved reconstruction quality. Lastly, Experimenting with different loss functions or optimizers could also enhance model performance.

Stacked Autoencoders

For the SAE, initially training the sub-encoders layerwise resulted in poor reconstruction quality. However, a big improvement in image clarity was obtained after training the entire network with these pre-trained sub-encoders. Despite this, the performance boost from sequentially training the smaller encoders and then the entire stacked autoencoder was not significantly greater than simply training the whole model for an equivalent duration. This might be due to the simple nature of the model and dataset which might limit the extent of improvement gained from this sequential approach. Nevertheless, conducting the 'whole training' after the layerwise training is a great way of fine-tuning the model.

Exercise 3.3

Convolutional Neural Networks

In order to construct the Toeplitz matrix $\mathbf{A}$ for the convolution operation, we need to arrange the elements of kernel $\mathbf{k} = [1, 0, 1]$ in the correct way in the matrix.
Since we have no padding and unit strides, the size of the output from the convolution operation is given by $n - m + 1$, where n is the length of the input vector $\mathbf{x}$ and m is the length of the kernel $\mathbf{k}$. Therefore, the Toeplitz matrix $\mathbf{A}$ is an $(n - m + 1) \times n$ matrix, which in this case is a 4×6 matrix.

In this matrix, each row corresponds to a different position of the sliding window of the kernel over the input vector. Therefore, we go through each index in the kernel and add these to the diagonal. The first element of the **k** (which is 1) is placed at the 0×0 position of the matrix and then continuously along this diagonal. The second element of **k** (which is 0) is placed starting from the second column in the first row, then moving diagonally down and to the right, and so on for the rest of **k**.
This gives us the Toeplitz matrix **A** in Equation 11. From this, we can see that by multiplying this matrix

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{11}$$

$$y = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 5 \\ 4 \\ 1 \\ 3 \\ 7 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \\ 7 \\ 8 \end{bmatrix} \tag{12}$$

with **X** we get the same **Y** as before in Equation 12. Transposing the matrix **A** and multiplying it with the output vector **y** (as $\mathbf{A}^T\mathbf{y} = \hat{\mathbf{x}}$) does not necessarily recover the original input vector **x**. This is primarily because the convolution operation is not an invertible process in this context. The convolution involves overlapping sums and reduces the dimensionality of the original input, which leads to a loss of information. In order to retrieve **x** from **A** (where $\mathbf{y} = \mathbf{A}\mathbf{x}$) we need **A** to be invertible. To show that this is the case we can substitute $\mathbf{A}\mathbf{x} = \mathbf{y}$ back into the equation to get:

$$\mathbf{A}^T\mathbf{y} = \mathbf{A}^T\mathbf{A}\mathbf{x} = \hat{\mathbf{x}} \tag{13}$$

Where $\hat{\mathbf{x}}$ is attempted recovery of **x**.
For $\hat{\mathbf{x}}$ to be equal to **x** it must be the case that there exists a matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}^{-1}\mathbf{y} = \mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{x}$ However, in these convolution scenarios, with no padding and stride greater than one, the resulting Toeplitz matrix is not square and often not invertible due to the loss of information and reduction in dimensionality. In essence, the operation of convolution followed by transposition and multiplication is not an identity operation, and therefore, it cannot recover the original vector **x** from the convolution result **y**.
To illustrate that this does not work, we can try this with our example.

$$\hat{x} = A^T y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 6 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \\ 13 \\ 14 \\ 7 \\ 8 \end{bmatrix} \tag{14}$$

Here we can see that $\hat{x} = [6, 6, 13, 14, 7, 8]^T$ is not the same as the original input vector $\mathbf{x} = [2, 5, 4, 1, 3, 7]^T$.

Running different CNN architectures on the dataset showed some results. The original model from the `cnn.ipynb` consists of four convolutional layers followed by two fully connected layers. The convolutional layers are all followed by batch normalization and a ReLU activation function.
Notably, the first convolutional layer uses a 2D convolution with just one input channel, which is fitting as the MNIST images are greyscale. Furthermore, it uses 16 output channels, a kernel size of 3, stride of 2, and padding of 1. The following 2 layers follow a similar configuration with similar kernel size, stride and padding but with different output channels of 32 and 64 respectively. The last layer doesn't change the output channels from the previous layer but modifies the kernel to a size of 2 with no padding. This design works by progressively increasing depth and reducing spatial dimensions, which is a way of extracting spatial features from images.
To test some different variations, I created four different models with varying modifications to the original implementation
Model 1 was a slight variation of the original, which had an increased kernel size of 5 in the first convolutional layer. It also included an increase in the depth of the third layer where the output channels jumped to 128. Otherwise, this model kept the same configuration of four convolutional layers.

Model 2 introduced several distinctive features compared to the original. The first convolutional block expanded the number of filters to 32, coupled with batch normalization and a ReLU activation function, followed by a MaxPooling layer to reduce spatial dimensions. The second block increased the filter count to 64. A notable change in this model was the introduction of a third convolutional block with dilated convolution, which increases the receptive field without increasing the number of parameters. This block used 128 filters, providing a significant increase in depth. The model also employed Global Average Pooling, to try to reduce overfitting.
Model 3 has an initial convolution block that combines multiple kernel sizes to capture a variety of spatial features. This was followed by a MaxPooling layer. It then increased the convolutional depth significantly, using two sequential layers with 128 filters each before transitioning to fully connected layers. This architecture was designed to assess the effects of a mixed kernel approach and deeper convolutional layers on feature extraction and model efficacy.
Model 4 focused on simplicity and regularization. It started with a convolutional layer of 16 filters, followed by a ReLU activation and a dropout layer for regularization. A MaxPooling layer was used for spatial reduction. The second convolutional block increased the filter count to 32 and also incorporated a dropout layer. In the fully connected stage, the network used a single linear layer of 128, before the final output layer that maps to the 10 classes. This model aimed to explore the impact of reduced complexity and increased regularization on performance.
The accuracy curves w.r.t. epochs for each model can be seen in Fig. 23.

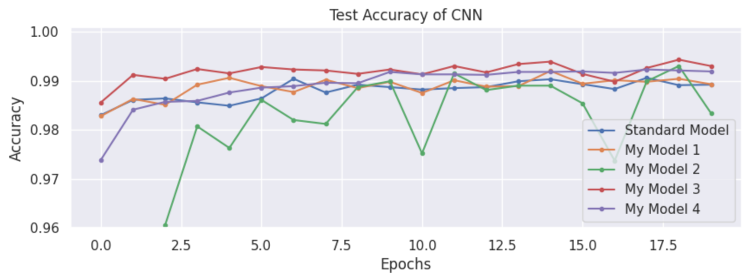


Figure 23: Model performance for the original model and my 4 variations

Here we can see the performance in general is pretty similar. Most notably the 2nd model is quite unreliable but the performance does become acceptable with an accuracy of 98.34%. We can also see that some models are more stable than others. Even though the 3rd and 4th models performer quite similarly through the entire training, model 4 is way more stable for each epoch than model 3.

Residual Neural Networks

After running the ResNet-18 model on the CIFAR-10 dataset we see that the general performance is pretty good with a test accuracy of 95.24%. We can see a visualization in the result in Fig. 24. Here we can see

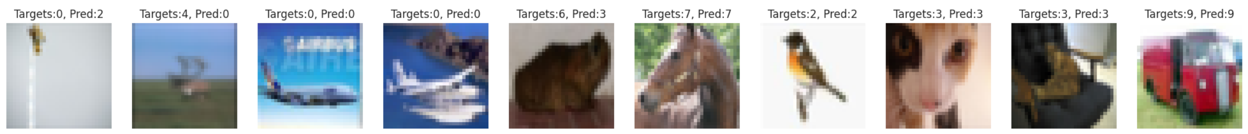


Figure 24: Sample output from the test data of the model
that most predictions are correct but there are some outliers[2]. We have a prediction 2 for a target 0, a prediction 0 for a target 4 and a prediction 3 for a target 6.
However, the sample size in these 10 images is quite small so to get a more general view of the behaviour I took 10,000 predictions and collected all the wrong ones. The results from these can be seen in Table 4.
Here we can see that the two most outlying predictions are for labels 3 and 5 which represent 'Cats' and 'Dogs' respectively. It is worth noticing that for these two particular labels, the most common misclassification

[2]The accuracy in the test differed somewhat when running the code multiple times and often it was correct for all 10 images

Label	0	1	2	3	4	5	6	7	8	9
Wrong Predictions	48	15	62	100	38	94	25	21	35	38

Table 4: Wrong Predictions for Each Label

involved confusion between the two: the majority of images labelled as 'Cats' were incorrectly predicted as 'Dogs', and conversely, most of 'Dogs' were misidentified as 'Cats'.
The reason for this inaccuracy may be caused by the similarity in the features these two labels share. Cats and dogs have quite similar proportions, colours and poses compared to many of the other labels such as Airplanes and birds.

Exercise 3.4

The dimensions in a self-attention implementation are designed by the need to transform the input data into query, key, and value representations.
In the NumPy implementation, the input **X** consists of 32 samples, each with a dimension of 256. This means that we have a batch of 32 vectors in a 256-dimensional space. The projection weights W_q, W_k, and W_v, are used to transform the input data into queries Q, keys K, and values V. The projection weights are of size 256×256 to make the projections result in all queries having dimensions of 32×256, to preserve the batch size and transform each vector into the respective space. The attention scores **A** are then computed as a 32×32 matrix, obtained by performing a dot product between Q and the transpose of K, scaled by the square root of the dimension, and applying a softmax function. This matrix represents how all the different samples are related to one another. The output is the multiplication of the attention scores **A** with the values V, which gives us a 32×256 matrix.
In the PyTorch implementation, the input **x** is a 3D tensor with a dimensionality of $batch\_size \times seq\_len \times input\_dim$ which are specified in the constructor of the `SelfAttention` class. In this example, the input **x** has dimensions $32 \times 20 \times 128$, meaning a batch of 32 sequences, each with 20 elements, and each element being a 128-dimensional vector. This input is then linearly projected to form queries q and keys k which yields ($batch\_size \times seq\_len \times dim\_\{q/k/v\}$) being $32 \times 20 \times 64$ for q and k and $32 \times 64 \times 32$ for v. Through batch matrix multiplication of q and k^T the attention matrix is computed giving dimensions $32 \times 20 \times 20$ as it has the shape ($batch\_size \times seq\_len \times seq\_len$), where each element indicates the attention score of a query vector over a key vector for each sequence in the batch. The final attention output is the multiplication of the attention matrix with v, yielding a tensor of dimensions $32 \times 64 \times 32$ being $batch\_size \times seq\_len \times dim\_v$.
To train the Transformer I ran the code with the following configurations shown in table 5.

Model	**Dim**	**Depth**	**Heads**	**MLP Dim**	**Validation Acc**	**Time (s)**
Baseline	64	6	8	128	98.43%	293.55 sec
Increased Capacity	128	8	16	256	98.67%	332.59 sec
Reduced Capacity	32	4	4	64	97.87%	250.83 sec
Increased Depth	64	12	8	128	98.62%	421.16 sec
Fewer Heads, Higher Dim	128	6	4	128	98.48%	298.24 sec
High MLP Dimension	64	6	8	256	98.39%	297.36 sec

Table 5: Different Configurations of Vision Transformer and Their Performance

From these results, the Baseline model achieved 98.43% accuracy in 293.55 seconds. The increased capacity model – with larger dimensions across all parameters – slightly outperformed the Baseline with an accuracy of 98.67% but took longer to train (332.59 seconds), showing a trade-off between capacity and efficiency. The Reduced Capacity model followed this trend with faster training (250.83 seconds), but with a drop in accuracy to 97.87%. Increasing the Depth of the model led to higher accuracy (98.62%) but had way longer training time (421.16 seconds). The Fewer Heads, Higher Dimension configuration got a similar accuracy to the Baseline (98.48%) with a similar training time (298.24 seconds). Lastly, the High MLP Dimension model, showed a slight decrease in accuracy (98.39%) with almost the same training duration as the Baseline (297.36 seconds). This shows that MLP size alone does not necessarily lead to performance gains. However all of these models performer quite similar. This might be due to the small dataset, so for this example we can get away with using smaller models.

Assignment 4

Exercise 4.1 - Energy-Based Models

Restricted Boltzmann Machines

1. When working with generative models, we often deal with high-dimensional data as it can involve images or text sequences. Therefore the use of pseudo-likelihood is primarily used for computational efficiency. Trying to calculate the actual likelihood is typically really computationally expensive which is why it is more efficient to use the pseudo-likelihood. It works by simplifying the estimation process by breaking don't the likelihood calculation into smaller parts that focuses on subsets of variables at a time. This is a more practical way of handling the learning process.
   However, there are multiple issues with the pseudo-likelihood that must be considered. Arguably the biggest issues is the introduction of bias in the estimation of the true likelihood, as the pseudo-likelihood is an approximation instead of an exact calculation. Additionally, as pseudo-likelihood can converges quicker than the true likelihood – due to the simplified computations – we run the risk of the model weighs local dependencies instead of discovering more global structures in the data.

2. The `n_components` refers to the number of hidden units in the RBM. The higher the number of components, the more complexity the RBM is able to capture from the data. However, this can also lead to overfitting. The `learning_rate` controls how much the weights are updated during training. If we increase the learning rate the more quickly the model can converge. The risk of increasing the learning rate is that the training can become more unstable and even overshoot the optimal solution. Conversely, having the learning rate too low will slow down the training and this can also run the risk of getting stuck in local minima. The `n_iter` determines how many times the training data is passed to the RBM. Having too few iterations can lead to a model that will not converge towards a good solution in time. On the other hand, more iterations can lead to a better performance, however, beyond a certain point the improvement might stagnate and we run the risk of overfitting.

3. Gibbs is used to approximate the distribution of data by iteratively sampling from the hidden and visible layers. Increasing the number of Gibbs sampling steps will generally lead to a more accurate approximation of the model's learned data distribution. As the number of steps increases, the visible and hidden states are sampled back and forth in a process that allows the model to refine its reconstruction which converges towards the true distribution. However, this refinement reduces the influence of the initial random data input, leading to the generation of samples that are more representative of the dataset. Notably, our model tends to generate more of the same digits, i.e. 6 and 7, more frequently. This may suggest a bias in the model's learned distribution.

4. The RBM's reconstruction shown in 25 shows the predictive abilities. By sampling hidden units and initializing parts of the visible layers, the RBM infers the missing top portions of digits. These are undeniably noisy reconstructions but the general shape of e.g. the '5' and the '9' have a high concentration of white where they should be.

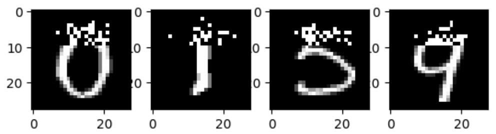


Figure 25: Image reconstruction

5. Removing more rows from the images makes the RBM's ability to accurately reconstruct the images more difficult, as it relies on less original data. The effect of removing rows from the bottom or the sides is based on the dataset. The general features of digits play a role in the RBM's ability to reconstruct. If we e.g. remove pixels from the bottom, the reconstruction tends to slope down and left like a '7' or '9' even for a straight '1'.

Deep Boltzmann Machines

1. Running the code in `DBM.ipynb` we can get the two filters from the model shown in Fig. 26 and Fig. 27. These filters are essentially the weights that have been adjusted during the training process to capture patterns within the dataset. Here, Fig 26 represents the weights of the first layer of the DBM

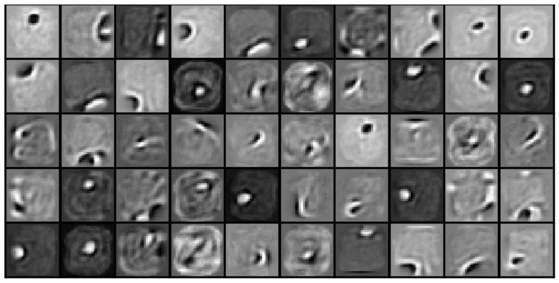


Figure 26: Filters extracted from the RBM

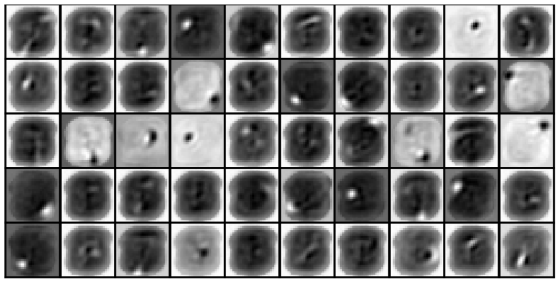


Figure 27: Filters from the DBM

which is directly connected to the input data. This layer, captures basic features from the input data. We can see that many of the features are on a very local basis. What can be seen in Fig 27 is the product of `W1_joint` and `W2_joint`, representing the effective weights of the second layer of the DBM. These filters are one step further in feature abstraction, which means that they can capturing more complex and high-level features compared to the first layer. As this layer is further removed from the original input data, it looks more abstract with less visual edges and shapes that one might be able to make out in the first layer.

2. When sampling new images from the DBM and comparing them to those generated by a RBM, the quality of the images produced by the DBM are generally more clear. This improvement is likely due to the DBM's deeper architecture, which allows for the capture of more complex and abstract features of the data. As the DBM can model more complex depemdencies between the observed variables than the RBM the DBM shows more clear digits with less noise.

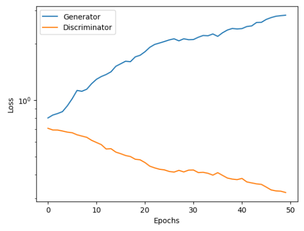
Exercise 4.2

1. Generative Adversarial Networks (GANs) have two separate neural networks – the generator and the discriminator – that are set up to compete against each other. The goal of the generator is to create fake data, while the discriminator aims to accurately distinguish between real data and generated data by the generator. This means that we have multiple perspectives when it comes to losses and results in a GAN. The generator's loss is when it fails to produce data that is convincing enough to '*fool*' the discriminator. Essentially, this loss reflects the generator's current effectiveness in mimicking real data. Meanwhile, the discriminator's loss is when it incorrectly classifies data as either real or generated. The results of the generator are measured in how often it can make the discriminator incorrectly classify one of its generated data as real. The discriminator's results are based on its accuracy in correctly classifying the data it receives.

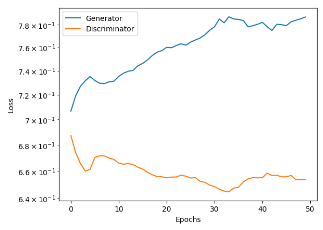
2. If the discriminator performs disproportionately much better than the generator, it can lead to stagnation in the training process. In this scenario, it becomes so much better at distinguishing data than the generator is at creating it, that it rarely gets '*fooled*'. This behaviour will lead to a situation where the generator will continuously get a high loss as it is not able to deceive the discriminator. Without being able to fool the discriminator, the generator will lack any useful feedback that can be used to improve its weights. This will stall the training as the generator is no longer able to improve.

3. Stability in GANs is one of the model's biggest downsides as it can be challenging to tune the training correctly since it is required that the generator and discriminator improve at a similar pace. If this balance is not maintained it will lead to divergence in the losses of the two models. The discriminator may become too proficient and easily identify all fake data, or the generator might fall into mode

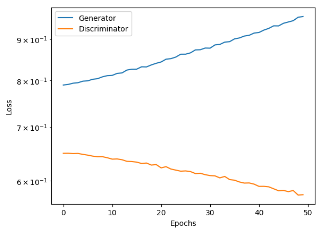
collapse, where it produces a limited variety of outputs as it has found a consistent way of '*fooling*' the discriminator. Examples of loss progressions with respect to epochs can be seen in Fig 28. From Fig



(a) Original



(b) Decreased learning rate for discriminator



(c) CNN-based model

Figure 28: Loss for the generator and discriminator with different training configurations

28a and Fig 28b we can see that the discriminator outperforms the generator the more the models are trained. Decreasing the learning rate for the discriminator does have an effect of slightly mitigating the divergence, but this illustrates that tuning these parameters to achieve a stable training process is not trivial. Ideally, we want to see the generator's loss gradually decrease meaning that it becomes better at creating convincing data. Simultaneously, the discriminator's loss should show an initial decrease followed by stabilization, indicating it's learning but not outpacing the generator. Convergence in a GAN is when this behaviour is realized and both models settle. In this state, the generator produces data that is realistic enough to often *fool* the discriminator, and the discriminator is capable of distinguishing real from fake data, but not so perfect that it always succeeds.

4. Each point in the latent space represents the required value to generate a potentially new output. Looking at the latent space in our GAN can give insights into how the latent variables influence the generated output. As we interpolate between the two latent variables `lambda_1` and `lambda_2` we can see the impact on the result. Shown in Fig. 29, we can see that gradually transitioning from `lambda_1` to `lambda_2` with an increasing λ, we see a smooth transition from one intelligible output (number 9) to another (number 8)

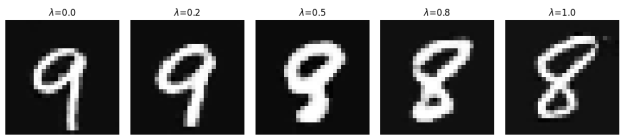


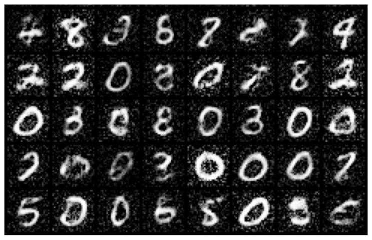
Figure 29: Image output for different λ-values moving from `latent_1` to `latent_2` of the GAN

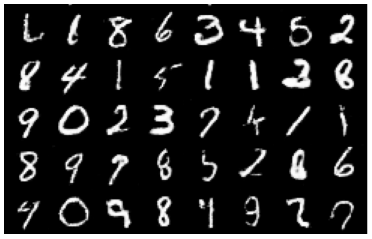
This gradual morphing effect shows the continuity and smoothness of the latent space. The image in between the two initial latent values contains features from both images, which shows the GAN model's ability to generate new data that does not exist in the initial training dataset.

One issue with the latent space of GANs e.g. compared to variational autoencoders (VAE), is the difficulty in controlling the generated output. Since the GAN does not have an encoder, it is extremely difficult to give a predictable input in the latent space that might generate e.g. a "4" in the case of the MNIST dataset.

5. The architecture of the CNN-based backbone for both the Generator and Discriminator uses a series of convolutional and transposed convolutional layers, combined with batch normalization and non-linear activation functions (ReLU and Leaky ReLU).

From the images in Fig. 30, we can see that the CNN-based model in Fig. 30(c) provide slightly sharper images compared to the original fully connected model in Fig. 30(a) and the optimized training version of that model in Fig. 30(b). None of the models in this exercise turned out to have a converging


(a) Original


(b) Decreased learning rate for discriminator

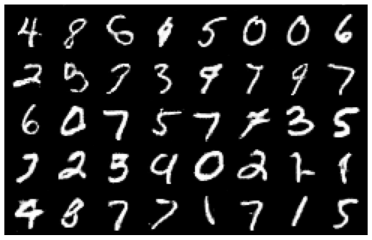

(c) CNN-based model

Figure 30: Generated output for the fully connected models and the CNN-based model

performance as was shown in Fig. 28 and this can again be seen in that some of the outputs don't represent actual numbers. Nevertheless, the CNN-based model is much more consistent and even for the unintelligible outputs, the features are less blurry and look more like hand-written images.

6. GANs have notable advantages over other generative models like auto-encoders and diffusion models, especially in generating higher-quality images. However, this comes with the trade-off of much more difficulty in training as it has stability issues due to the need for careful balancing of the generator and discriminator. In comparison, auto-encoders typically have much more stable training processes but at the cost of struggling to achieve the same generative qualities. Diffusion models, on the other hand, also have the ability to generate high-quality outputs like GANs, but they typically require longer training times and more computational resources. Overall, GANs work great in unsupervised learning from complex distributions, but their sensitivity to hyperparameter settings can make them more challenging to work with compared to the more straightforward training processes of auto-encoders and diffusion models.

Exercise 4.3

1. The goal of VAEs is to learn a latent representation of data. In theory, VAEs are designed to maximize the log-likelihood of the data, but in practice, directly maximizing the log-likelihood is computationally challenging. Therefore, alternative approaches are often used. In the code example from the `VAE.ipynb`, the optimization is done with an approach involving two main components: the reconstruction loss and the KL divergence loss. The reconstruction loss, `recon_loss`, uses binary cross-entropy to minimize the difference between the original data (`batch`) and its reconstruction (`batch_recon`). The KL divergence loss, `kl_loss`, measures the divergence of the learned latent distribution (`z_mu` and `z_var`) from a standard normal distribution. By minimizing the total loss, `total_loss = recon_loss + kl_loss`, the code gives accurate data reconstruction while maintaining a regularized latent space with a normal distribution, which is needed for the VAE's generation.

2. There are both similarities and differences in the stacked auto-encoder (SAE) and VAE implementations. The goal of both models is to reconstruct the input data as closely as possible. The overall architecture of both, consisting of encoders and decoders, show that in the forward pass we first transform the input to smaller layers before reconstructing it back to the original. The most obvious difference is that the VAE uses a probabilistic approach to encoding with the latent space represented by mean (`z_mu`) and variance (`z_var`) parameters. This leads to a regularization component (`kl_loss`) in the loss function, which is not a part of the SAE.

   With regards to the reconstructing error the VAE measures loss using binary cross-entropy (BCE) loss as `recon_loss = bce()`. This is used to quantify the difference between the original data and its reconstruction. For the SAE in the previous assignment, the reconstruction error for the training of individual layers also used BCE loss where criterion is defined as `BCELoss()`. However, for training the whole network, MSE loss was used.

3. Exploring the latent space in the VAE through interpolation between two random points shows the model's capacity for smooth transitions between different data representations. In the code, the two latent vectors (`latent_1` and `latent_2`) are sampled from a standard normal distribution and interpolated using increasing values of λ. The output which can be seen in Fig 31, where images gradually morph from resembling the digit '3' to '9', shows how the VAE's latent space encodes meaningful features.

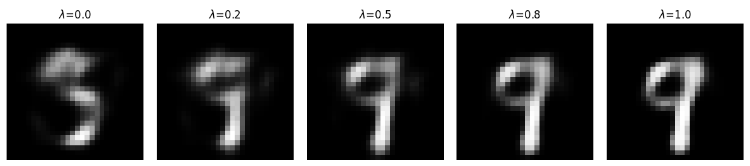
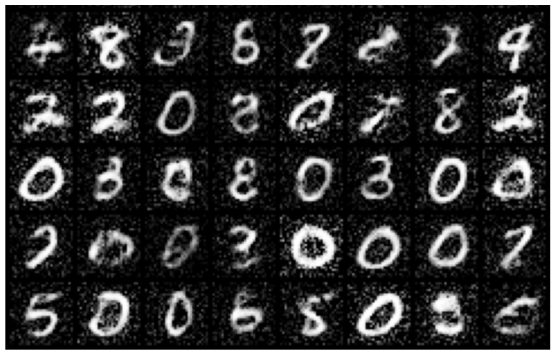


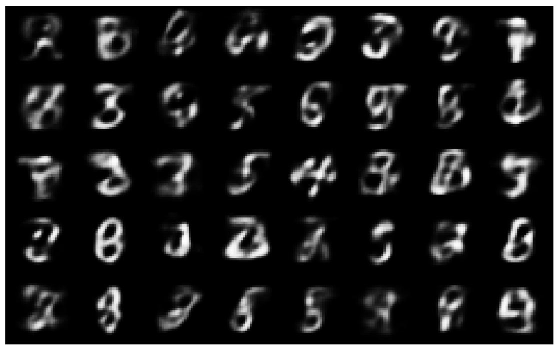
Figure 31: Image output for different λ-values moving from `latent_1` to `latent_2` of the VAE

This continuity and smoothness in the latent space is one of the main properties of VAEs, as it allows for the generation of data from values in the latent space that has not been directly trained on due to the probabilistic approach. This example is similar to what we observed in exercise 4.2 using GANs, indicating a shared feature among these generative models. Both VAEs and GANs effectively cover the latent space which can capture the underlying structure of the data distribution.

4. Comparing the outputs of the original GAN model and the VAE model, as shown in Fig. 32, reveals interesting characteristics in their generative outputs. GANs tend to produce sharper and more defined



(a) GAN model



(b) VAE model

Figure 32: Output from the GAN model and the VAE model

images, but with noticeable noise, especially in areas requiring uniformity, such as the background of the dataset. In contrast, VAEs generate much more blurrier images with significantly less noise, resulting in smoother but less detailed outputs. This difference might be caused by GANs' adversarial training, which focuses on detail but can introduce artifacts, where VAEs have a more probabilistic approach, emphasizing overall data structure over fine details.

Exercise 4.4

For this exercise, I have chosen to implement a Variational Autoencoder. This choice is based on VAE's ability to learn latent representations of data, which is an interesting concept in generative models.
For the dataset, I chose the CIFAR-10 dataset, where I specifically focused on the 'ship' class. This choice is due to the higher complexity compared to the simpler MNIST dataset we worked with previously as the images are larger and are coloured. This allows for developing a more complex model, while still keeping the training relatively light as the images are 32×32. Although one of the advantages of using VAE is that

it should be able to handle more different labels of data in its latent space, and therefore could in theory handle all labels from the CIFAR dataset, I chose to focus on the ships to try to achieve a clearer result.
The architecture of the VAE consists of the two primary components: the encoder (Q) and the decoder (P). The encoder is designed with convolutional layers, each followed by batch normalization, to try to capture the spatial features in images. It consists of three convolutional layers with increasing filter sizes (64, 128, 256) and a kernel size of 4. The output of these layers is then flattened and passed through fully connected layers to produce the mean and variance in the latent space distribution. The encoder users Leaky ReLU activation as this was found to be optimal for this design and dataset.
The decoder uses transposed convolutional layers, again, with batch normalization in order to reconstruct the image from the latent space representation provided by the encoder. It starts with a fully connected layer to expand the latent dimension. This is followed by upsampling and convolutional layers to return to the original image size gradually. The final layer of the decoder uses a Tanh activation function, as it works great for output normalization in the range of [-1, 1] and was found to perform well.
Additionally, multiple combinations of hyperparameters were tested and the following resulted in the best performance:

- Batch Size: 128
- Latent Dimension: 100
- Middle Dimension: 256
- Learning Rate: 1e-4

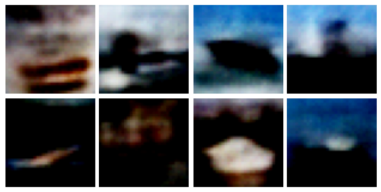


Figure 33: Reconstructed images during training

The training process was run for 250 epochs. Throughout the training, the model's performance was observed by looking at the reconstruction quality. One of the final samples can be seen in Fig. 33.
After training the following results in Fig. 34 and Fig. 35 were achieved.

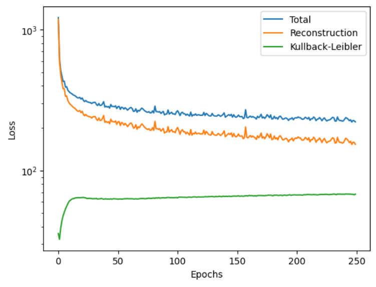


Figure 34: Trainig loss for the model

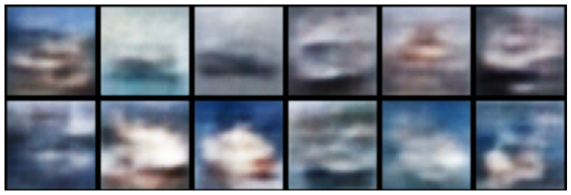


Figure 35: Image generation of the model

Here in Fig. 34 we can see that the training of the model is very stable and nicely lowered the loss. Arguably the 250 epochs were too few and an even better performance might have been possible with this setup if we increased the number of epochs. For the generation of new images in Fig. 35 it is clear that the model has learned some overall features to generate ships. Furthermore, the images show quite different variations of ships, indicating that the training was successful in developing a capable latent space for new image generation. The images are quite blurry however which is an expected challenge when working with VAEs. This blurry result is one of the typical downsides of VAE due to the tradeoff between quality and regularization in the latent space.
One additional factor that might influence the observed performance issues, is the noticeable blurriness of the reconstructed images during training in Fig. 33. Although these images appear slightly sharper than the models generated images, they still struggle to accurately reconstruct the training data. This might suggest that the current architecture of both the encoder and decoder may not be able to capture the dataset to its fullest.