Københavns Universitet: DL Assignment 1

Daniel Friis-Hasché - rcb933
Victor Vangkilde Jørgensen - kft410
Victor Abildgaard Cadier Pedersen - wdz535

November 28, 2025

Contents

1 Backpropagation pen and paper

1.1 1c) The equations for an FFNN as recursion

*Note: We refer to a neuron's value pre-activation function as its **activation**, while we refer to a neuron's value post-activation function as its **output**.*

We begin with the first layer of activations, $a_j^{(1)}$. These are given by the following expression:

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

We can rewrite this notation such that we don't have to explicitly write the 'bias' parameter, $w_{j0}^{(1)}$. We do this by extending the set of inputs x_i to include a constant x_0 with $x_0 = 1$. By extending our sum we get the following expression:

$$a_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$

Where the parameter $w_{j0}^{(1)}$ is multiplied by $x_0 = 1$ in the sum (and thus this expression is equivalent to the one with the bias). We keep a note that this is done for every layer. We now apply a non-linear function denoted h_1 to the activations $a_j^{(1)}$. This gives us the output of the first layer, which we write as

$$z_j^{(1)} = h_1(a_j^{(1)}) \ .$$

Now is when we get to the recursion. The activations for the next layer are the sum of the products of the weights ($w_{ji}^{(2)}$) and the output of the first layer ($z_j^{(1)}$). In fact, this continues for every layer, allowing us to express these values recursively:

$$a_j^{(l)} = \sum_{i=0}^{M} w_{ji}^{(l)} z_i^{(l-1)} \quad l = 2, \dots, L$$

Where L is the amount of layers, and M refers to amount of neurons in both the hidden layers and the output layer. We apply the activation function h_l for the l'th layer giving us the output of the $l'th$ layer, which we'll now also write recursively:

$$z_j^{(l)} = h_l(a_j^{(l)}) \quad l = 1, \dots, L$$

While we've now fully described our FFNN, we'll add another term y_j which will refer to the $j'th$ neuron of the last layer, which is the output layer:

$$y_j = z_j^{(L)}$$

Now let's write all this in a neat box.

Feed-forward neural network
We consider a FFNN with L layers, D neurons in the input layer, and M neurons in each hidden layer and the output layer. The equations for the neural network can be expressed as such:

$$\begin{aligned}
&\text{(Output layer)} && y_j = z_j^{(L)} && \\
&\text{(Output of the } l\text{'th layer)} && z_j^{(l)} = h_l(a_j^{(l)}) && l = 1, \dots, L \\
&\text{(Activation of the } l\text{'th layer)} && a_j^{(l)} = \sum_{i=0}^{M} w_{ji}^{(l)} z_i^{(l-1)} && l = 2, \dots, L \\
&\text{(Activation of the first layer)} && a_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} x_i &&
\end{aligned}$$

1.2 1j) The backpropagation rule for layer $l < L$

For this exercise we're asked to use the 'above results' to argue why the general back-propagation rule for $l < L$ is written as:

$$\frac{\partial E(w)}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)} ,$$

$$\text{With } \delta_j^{(l)} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)})$$

The 'above result' the exercise is referring to, is a derivation resulting in (among other things) this expression:

$$\delta_j^{(L-1)} = \frac{\partial E(w)}{\partial a_j^{(L-1)}} .$$

This expression is in terms of the second to last layer. Let's first generalize it, by allowing ourselves to rewrite $L-1$ as simply l, understanding that these results apply to all layers

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial a_j^{(l)}} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) .$$

This is already the expression of $\delta_j^{(l)}$ from the exercise text, indicating that we're on the right track. Let's take another look at the partial derivative of the error function with respect to a given weight of the second to last layer,

$$\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}} = \sum_{k=1}^{K} \frac{\partial E(w)}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial a_j^{(L-1)}} \frac{\partial a_j^{(L-1)}}{\partial w_{ji}^{(L-1)}} = \sum_{k=1}^{K} \delta_k^{(L)} w_{kj}^{(L)} h_{L-1}'(a_j^{(L-1)}) z_i^{(L-2)} .$$

(This was an intermediate step in the exercise text). Again, we generalize this expression by rewriting $L-1$ as l, giving us this expression:

$$\frac{\partial E(w)}{\partial w_{ji}^{(l)}} = \sum_{k=1}^{K} \frac{\partial E(w)}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) z_i^{(l-1)}$$

We can now see what happens when we take the product of $\delta_j^{(l)}$ and $z_i^{(l-1)}$:

$$\delta_j^{(l)} z_i^{(l-1)} = \left(\sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) \right) \left(z_i^{(l-1)} \right) = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) z_i^{(l-1)}$$

We see that $\delta_j^{(l)} z_i^{(l-1)}$ gives us the same sum, that was equivalent to $\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}}$. As such, we've shown that

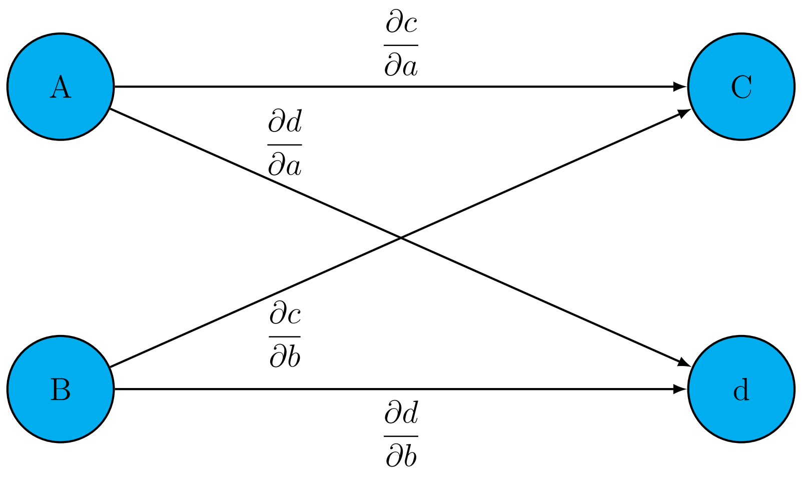
$$\frac{\partial E(w)}{\partial w_{ji}^{(L-1)}} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) z_i^{(l-1)} = \delta_j^{(l)} z_i^{(l-1)}$$

with $\delta_j^{(l)} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)})$. This is the result we were asked to argue for in this exercise, which we've now shown.

2 AutoDiff framework

2.1 2b)

The entire framework runs using the defined `Var` class. The class is easy to work with since it stores all gradients computed in memory. When we compute the gradient for a given neuron, we compute all gradients leading from the last neuron, through the network to the neuron, and add all pathways together. When doing backpropagation, we essentially "inverse" the network, and run a different kind of `forward()` on it, but instead of feeding the normal weights and biases, we feed the gradients - hence the reason we start from the last neuron and work our way backwards.
When we do backpropagation, we compute the gradients by finding the partial derrivatives of each neuron, and summing them together:


A
B
C
d
$\frac{\partial c}{\partial a}$
$\frac{\partial d}{\partial a}$
$\frac{\partial c}{\partial b}$
$\frac{\partial d}{\partial b}$

We would then get the final gradient for e.g. neuron A as

$$\sum_{i=0}^{1} \nabla a_i^{(l)} : \quad \frac{\partial c}{\partial c} \cdot \frac{\partial a}{\partial c} + \frac{\partial d}{\partial d} \cdot \frac{\partial a}{\partial d} = \frac{\partial a}{\partial c} + \frac{\partial a}{\partial d} \tag{1}$$

When we compute the new gradient for A (by using backpropagation), we then save the gradient in memory, so we can access it later on.

In the notebook, we were asked to do these calculations, where we for the first example got:

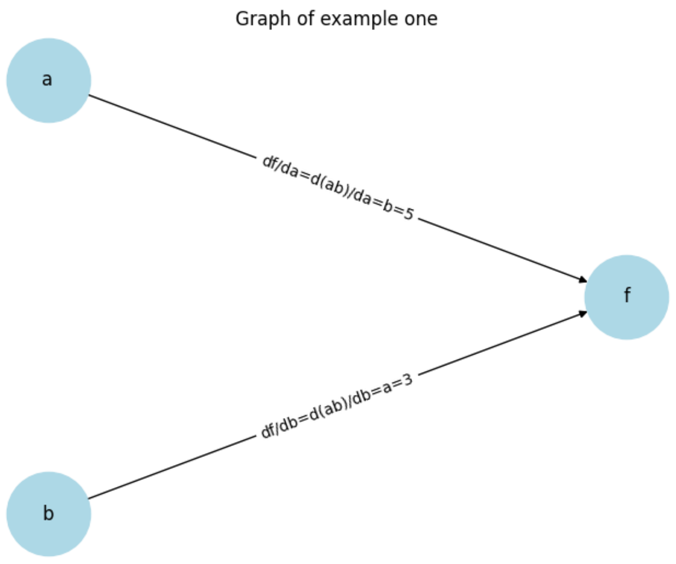


Figure 1: Gradients for example 1 in notebook 2 exercise *b*)

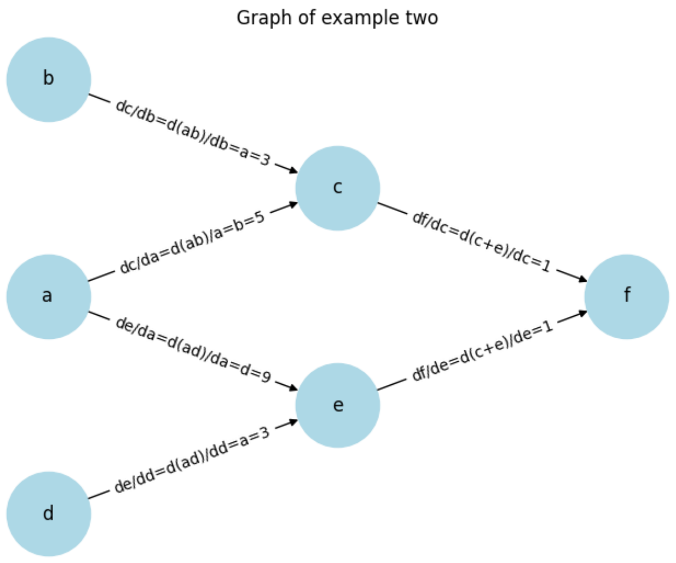


Figure 2: Gradients for example 2 in notebook 2 exercise *b*) (full)

In the two cells below the `Var` class definition, we have two examples for how to use the `Var` class. In the first example, the following expression is defined:

$$f = a \cdot b \quad \text{with a} = 3, \ \text{ and b} = 5 \tag{2}$$

The gradient is automatically calculated by the class when we call the `.backward()` method. Under the hood the following computations are being performed, which you can also see on two graphs.

Example 1:

$$\begin{aligned}
\text{f.grad} &= \frac{\partial f}{\partial f} = \frac{\partial(ab)}{\partial(ab)} = 1 \\
\text{b.grad} &= \frac{\partial f}{\partial b} = \frac{\partial(ab)}{\partial b} = 1 \cdot a = 3 \\
\text{a.grad} &= \frac{\partial f}{\partial a} = \frac{\partial(ab)}{\partial a} = 1 \cdot b = 5
\end{aligned}$$

For the next example, we perform a calculation defined by this expression:

$$c = ab, \quad e = ad, \quad f = c + e \quad \text{Or, equivalently} \quad f = a(b + d)$$

With respective variables being:

$$a = 3, \quad b = 5, \quad d = 9$$

Unlike the previous example, we now have neurons that connect to multiple neurons, which means we must compute the gradients slightly different. When calculating the gradient for a single neuron, we essentially calculate all gradient "paths", and sum them. If a neuron connected to two other neurons, the gradient for said neuron would be the gradients of both "paths" summed up. We could therefore write:

$$a_j^{(l)}.\text{grad} = \sum_k \frac{\partial Loss}{\partial a_k^{(l+1)}} \cdot \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} \tag{3}$$

In other words, we calculate all gradients throughout the network, by summing the gradients from any given neuron to the next (we go in reverse order), and so we have:
Example 2:

$$\begin{aligned}
\text{f.grad} &= \frac{\partial f}{\partial f} = \frac{\partial(c+e)}{\partial(c+e)} = 1 \\
\text{e.grad} &= \frac{\partial f}{\partial e} = \frac{\partial(c+e)}{\partial e} = 1 \\
\text{c.grad} &= \frac{\partial f}{\partial c} = \frac{\partial(c+e)}{\partial c} = 1 \\
\text{d.grad} &= \frac{\partial f}{\partial b} = \frac{\partial f}{\partial e}\frac{\partial e}{\partial d} = \frac{\partial(c+e)}{\partial e}\frac{\partial ad}{\partial d} = a = 3 \\
\text{b.grad} &= \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} = \frac{\partial(c+e)}{\partial c}\frac{\partial ab}{\partial b} = a = 3 \\
\text{a.grad} &= \frac{\partial f}{\partial a} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial f}{\partial e}\frac{\partial e}{\partial a} = \frac{\partial(c+e)}{\partial c}\frac{\partial ab}{\partial a} + \frac{\partial(c+e)}{\partial e}\frac{\partial ad}{\partial a} = 1 \cdot b + 1 \cdot d = 5 + 9 = 14
\end{aligned} \tag{4}$$

2.2 2l)

This exercise is split up into three parts: overfitting, underfitting, and just the right fitting.

2.2.1 Overfitting

For the task at hand, this turned out to be surprisingly difficult. We had tried various types of networks, and the same thing would happen each time: The model would either completely stagnate given long enough, or the learning rate would be so high that the loss would jump between every epoch. In either case, once the model had reached a fairly low test loss, training would effectively stop.
To make matters worse, the code was really quite slow. We were forced to lower the learning rate quite significantly, so that the model wouldn't completely overshoot the valleys of the loss landscape. But because of this necessary low learning rate, the model would continue to improve for easily thousands of epochs. While writing this report, we realize that we haven't tried to train it on smaller batches (instead of the entire training data per gradient update). In hindsight, we see that might have made the model learn a lot faster. Nonetheless, this is not what we did.
The trick we used was to have an **adaptive learning rate**. Our training code would have a baseline learning rate, which starts high and goes down throughout the epochs. Between epoch 75 and epoch 125, we would calculate a moving average of the change in training loss. At the end of this period, we take the average of the averages, and this becomes the 'target rate of change' of our model. For all epochs after 125, we compute the moving average of the training loss. The difference between the current rate of change in training loss, and our target rate of change, is used to calculate a value which we add to our learning rate. There are some additional details like how the target rate of change goes down throughout the training as well, but in the spirit of this exercise, our training algorithm, along with all the details of how it works, was of course 'overfitted' to this exact exercise. The algorithm was flimsy and would often enter these cycles of instability consisting of massive swings in learning rate and training loss.
The thesis here, and the logic behinds this algorithm, was that if we can force the model to keep improving its training loss through the entire training period, there will be a time where the model will simply be forced to overfit to the training data in order to keep improving. After many iterations of our training algorithm, this idea was finally validated as the trend of the validation loss falling in sync with the training loss was abruptly reversed after a little under 9000 epochs. The next day we tweaked the code and trained again from scratch. We trained our model for 10 000 epochs. The result is shown in figure 3 and figure 4.
In figure 3 we see how both the training loss and validation loss fall in the beginning of training. While the training loss falls down to nearly half of its from 100th epoch to the 10 000th epoch, the validation loss sees a significantly smaller proportional decrease throughout this period. But most intringuingly, the training loss decreases monotonically (except for some instability early on), while the validation loss certainly does not.
We see more precisely how the model overfits on figure 4. Here, we see that the validation loss achieves an all time low of approximately 6.94, around epoch 2300. Then, as the training loss continues to fall, the validation loss begins to climb higher, reaching approximately 7.27 near the end of training on eoch 10 000. This is admittedly a meagre example of overfitting, but we believe that this is nonetheless an example of it.
At the end of training, our training loss had fallen to 6.88 while the validation loss climbed itself up to 7.27. As the exercise defined overfitting as the training loss being significantly lower than test loss, we are perhaps stressing the definition of significant.

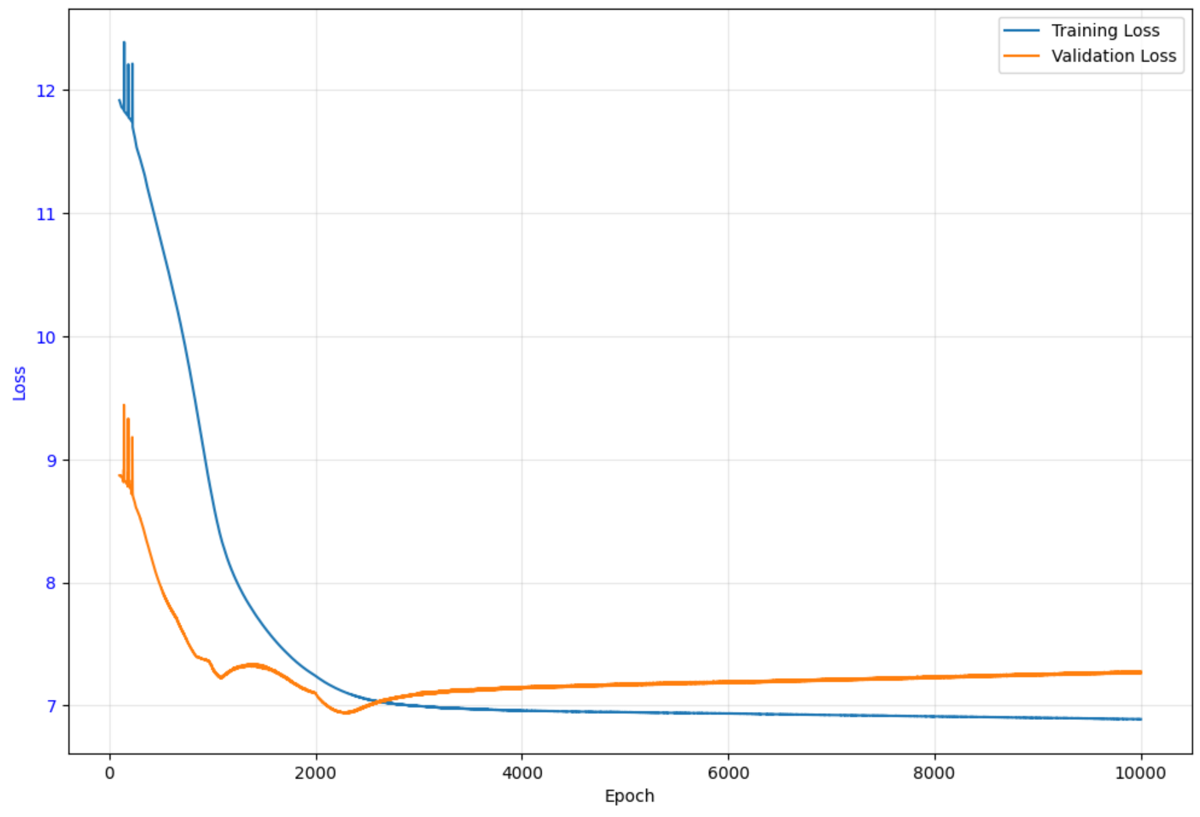


Figure 3: Loss of our overfitted model during training. From epoch 100 to epoch 10 000.

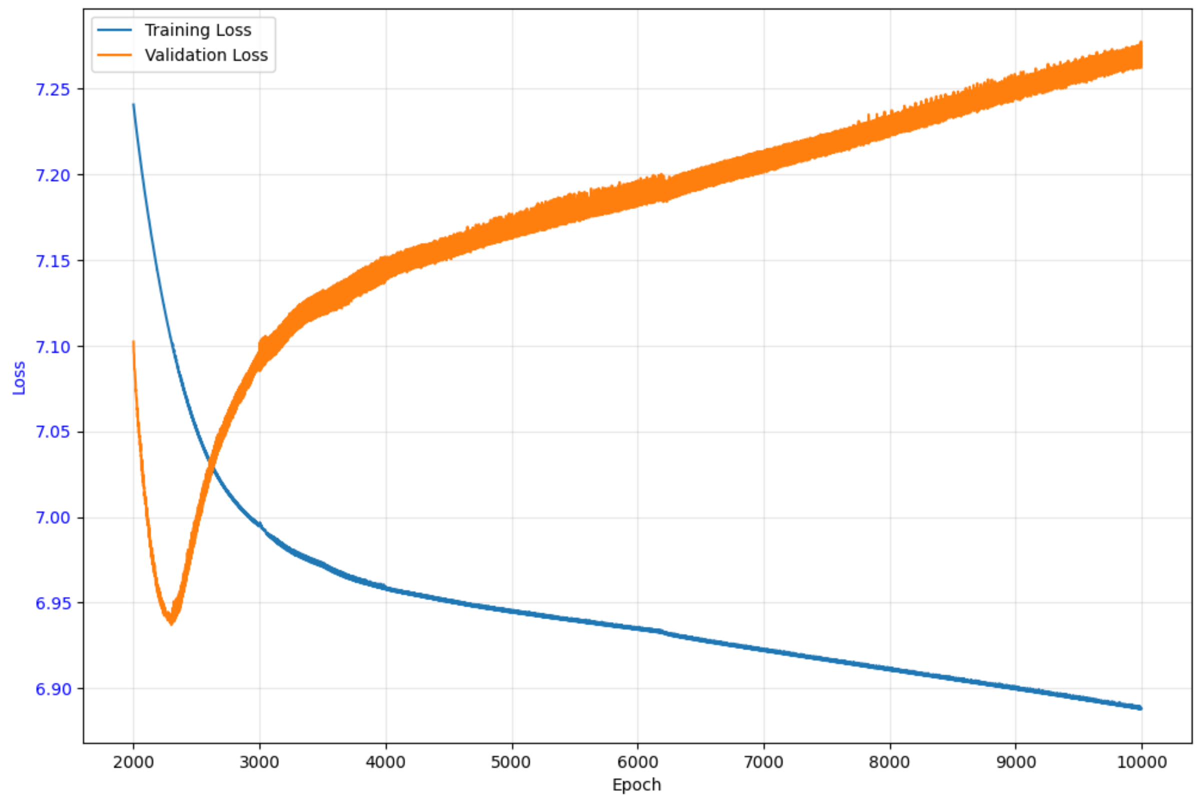


Figure 4: Loss of our overfitted model during training. From epoch 2000 to epoch 10 000.

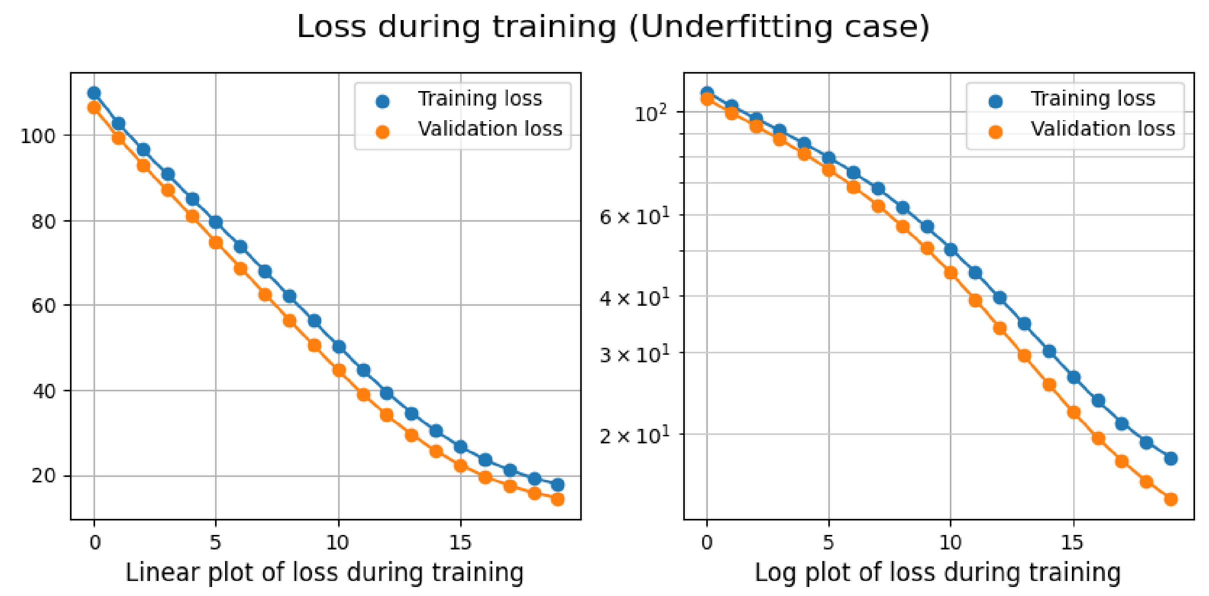


Figure 5: Loss during training, for the underfitting case

2.2.2 Underfitting

We define a new neural network, with a single hidden layer with a width of 32 neurons. We train this model for a only 20 epochs which is a lot less than we did before. The results can be seen on figure 5.
From the trajectory of the loss during training, we would expect to see a continued decline in loss for both sets if training kept going. Because the training loss is high (ending at 17.77) in comparison to what we've been able to achieve with more training, we can infer that we've underfitted.

2.2.3 Just the right fitting

For this example we wanted to demonstrate a case where, during learning, the loss for both the test set and validation set would plateau, as such seemingly converge onto specific values. For this, we made a very small model with only 31 parameters, and trained it for 5000 epochs. We plot the results on figure 6.
The plot shows that both the training and the validation loss of the model seems to plateau during training. On the loglog plot we see that both these losses seem to go down somewhat linearly, until their rate of change seems to approach zero. But we can see that they're not exactly zero. Nevertheless, training this model until 10 000 epochs reveals that there are no significant improvements before the validation loss begins to ever so slightly increase again.
Our model ends its training with a train loss of 9.43 and a validation loss of 6.93. Importantly here, if we simply compare the absolute values of these losses, our overfitted model just might be better for this case. Perhaps the real perfect fit would have been the overfitted model right before the validation loss begins to go up again. Nevertheless, we decided that we would attempt to produce a clear example of a 'perfect' fit specifically for a very small model, which we believe we've done.

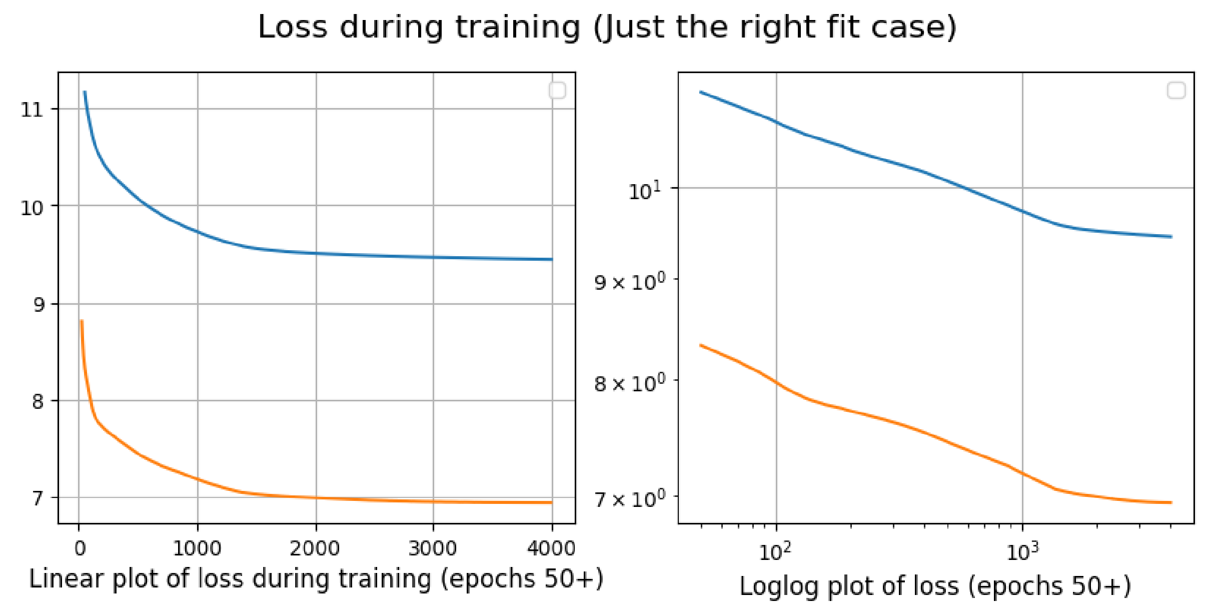


Figure 6: Plots of loss during training, for the 'just the right fit' case

2.2.4 Difference between validation and test loss

Throughout this exercise we noticed that a change on the training loss correlated quite strongly with a change on the validation loss. In fact, it was difficult for us to get a model to perform well on one and not the other (for the overfitting part of the exercise). Though, interestingly, the training loss was consistently higher than the validation loss almost all the times we trained our models, except for when we had overfitted. As the only real difference between the two sets is that the validation set consisted of 70% of the generated dataset while the training set was only 35% of it, we believe that this may have something to do with it.
The exercise text also asks for us to discuss cases where it is important to keep validation loss and test loss separate. We imagine a case where the validation set has been intentionally chosen to be made up of the particularly difficult data we know our model might struggle to generalize onto. This may be done e.g. because most of the data we have to train on pertains to a relatively easy class for the model to predict; a class we might care less about compared to a more difficult (and more rare) class. Having a greater distribution of these in the validation set may allow us to experiment with which model type that is better at learning about the class we want from our abundant data about the class we don't care about. In this case, the difference between the test and training loss as well as the absolute value of the validation loss would be far more important than the absolute value of the training loss (and so it would be important to keep the validation loss and test loss separate as they effectively measure different things).

3 Deep versus wide networks

After making our data generation class, we use it to generate a training and validation set:

3.1

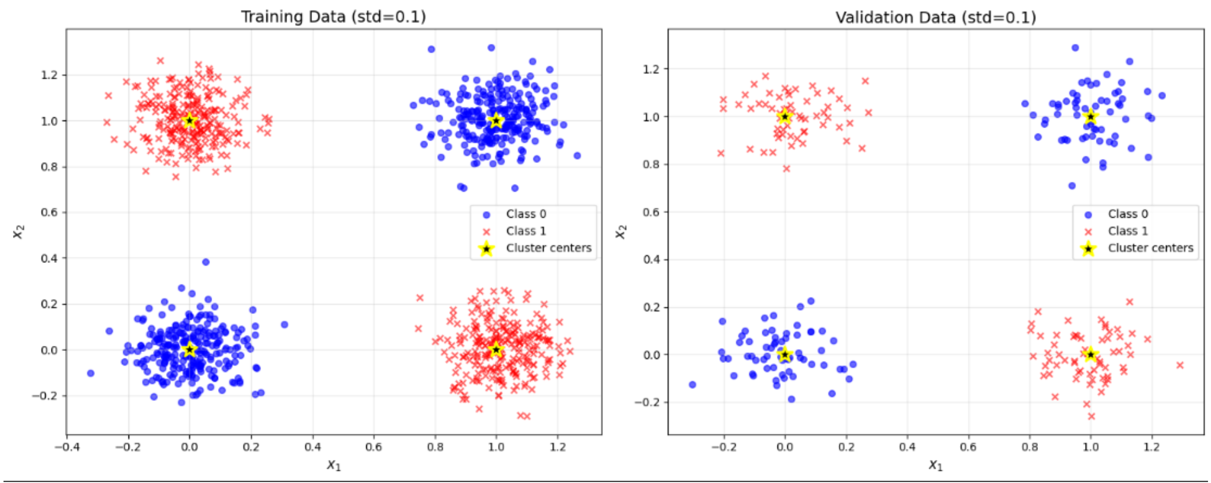


Figure 7: Generated data using the generator class we made

The nn.BCELoss function expects input probabilities in the range $[0, 1]$. We therefore have to use the sigmoid activation layer as the final step of our network. It will therefore probably not work if predictions are extremely close to 0 or 1.

The nn.BCEWithLogitsLoss function expects raw logits (unbounded scores). We should not use a sigmoid layer in our final output of the network. The function applies the sigmoid function internally, which makes it always work because it combines the sigmoid and loss calculation into one step. In our model we use the nn.BCEWithLogitsLoss function, by removing the sigmoid function from the output of our model. We do this because we want to avoid those cases where we have extremely close values to 0 and 1. After generating the data, we use our model class to train on our training set, and to validate on our validation set:

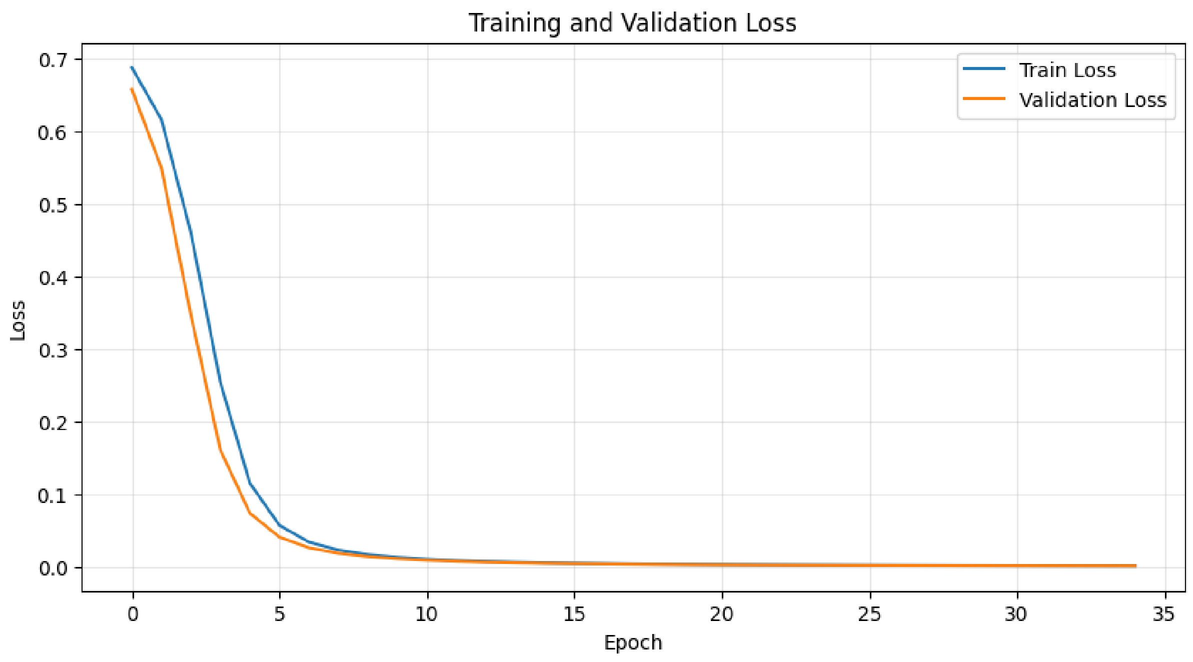


Figure 8

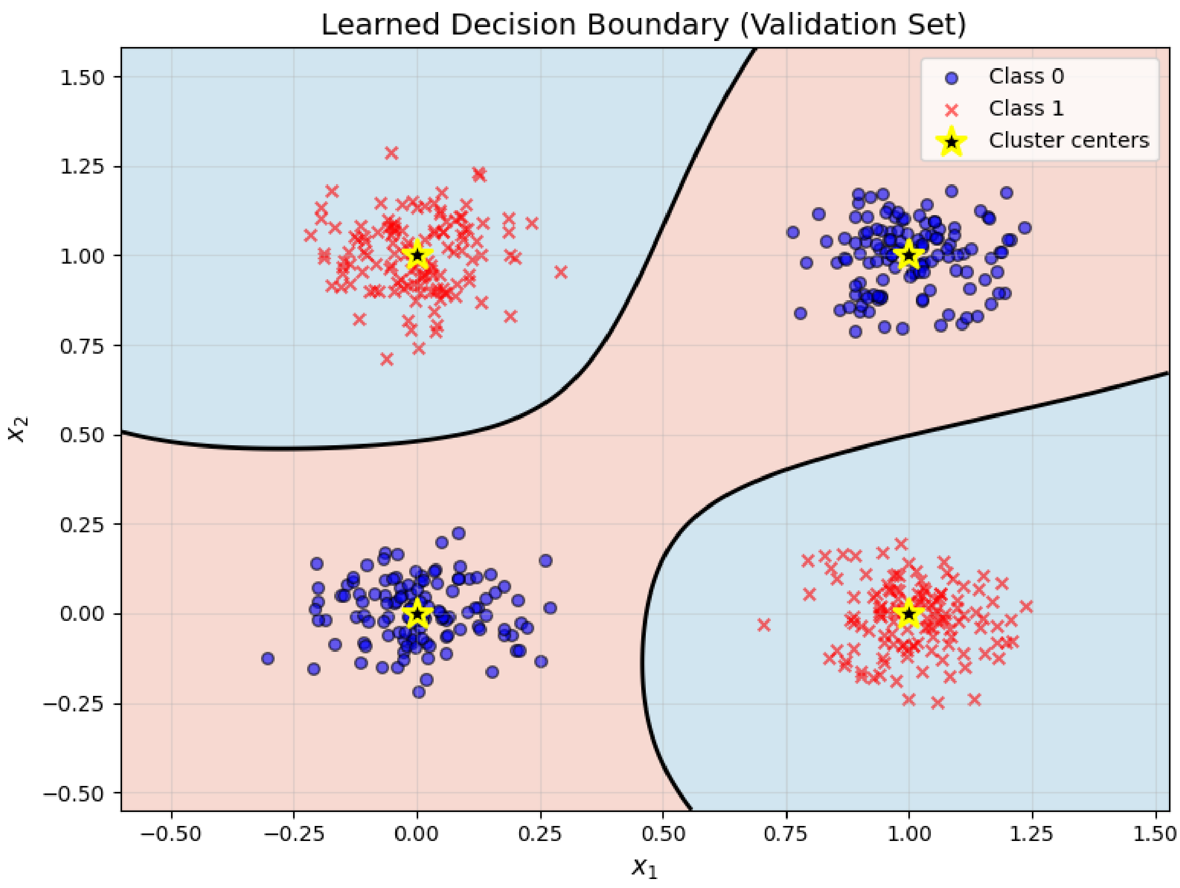


Figure 9

We see that the boundaries we get are successful at classifying the two classes.

3.2

We define some hyperparameter values for the depth and width of layers that we want to test:

```python
depths = [0, 1, 2, 3, 5, 10, 20]
widths = [1, 2, 3, 5, 10, 20]
n_repetitions = 6  # Number of runs per configuration
epochs = 25
```

We thereby end up with 42 different model hyperparameters, trained 6 times each for 25 epochs.

This gives us the following results:

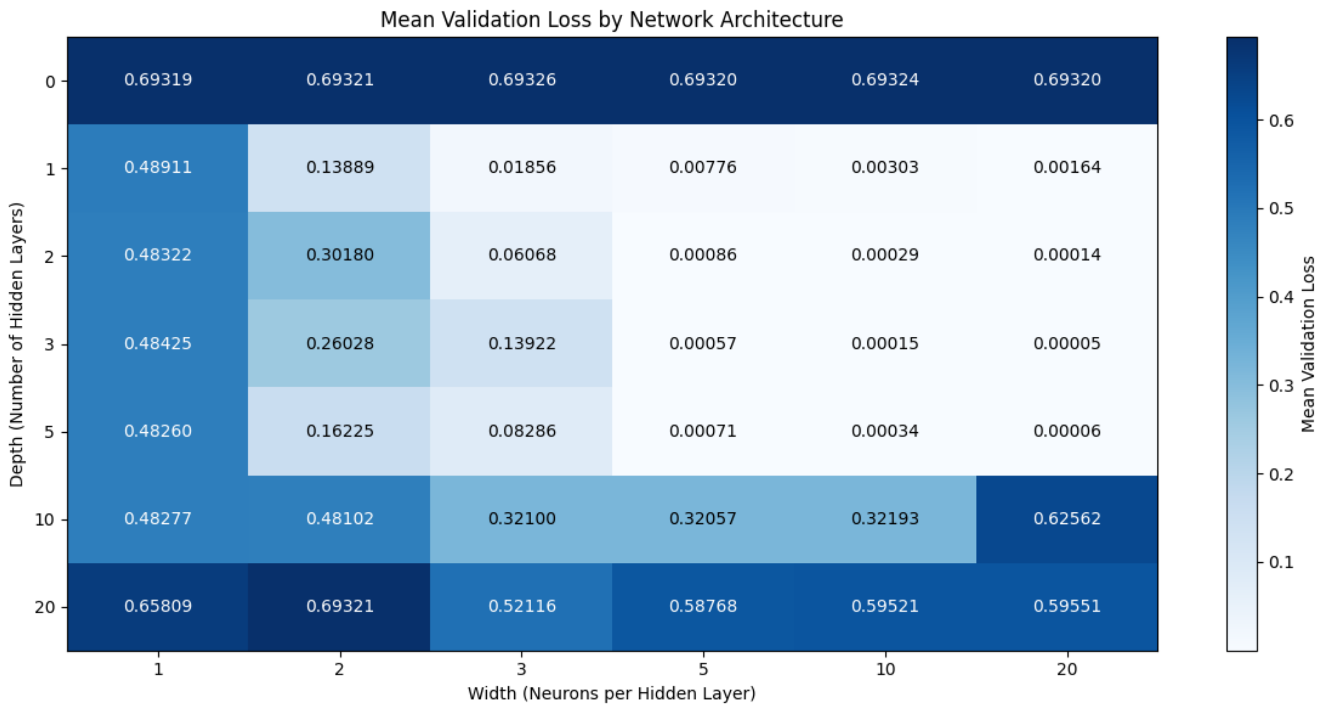
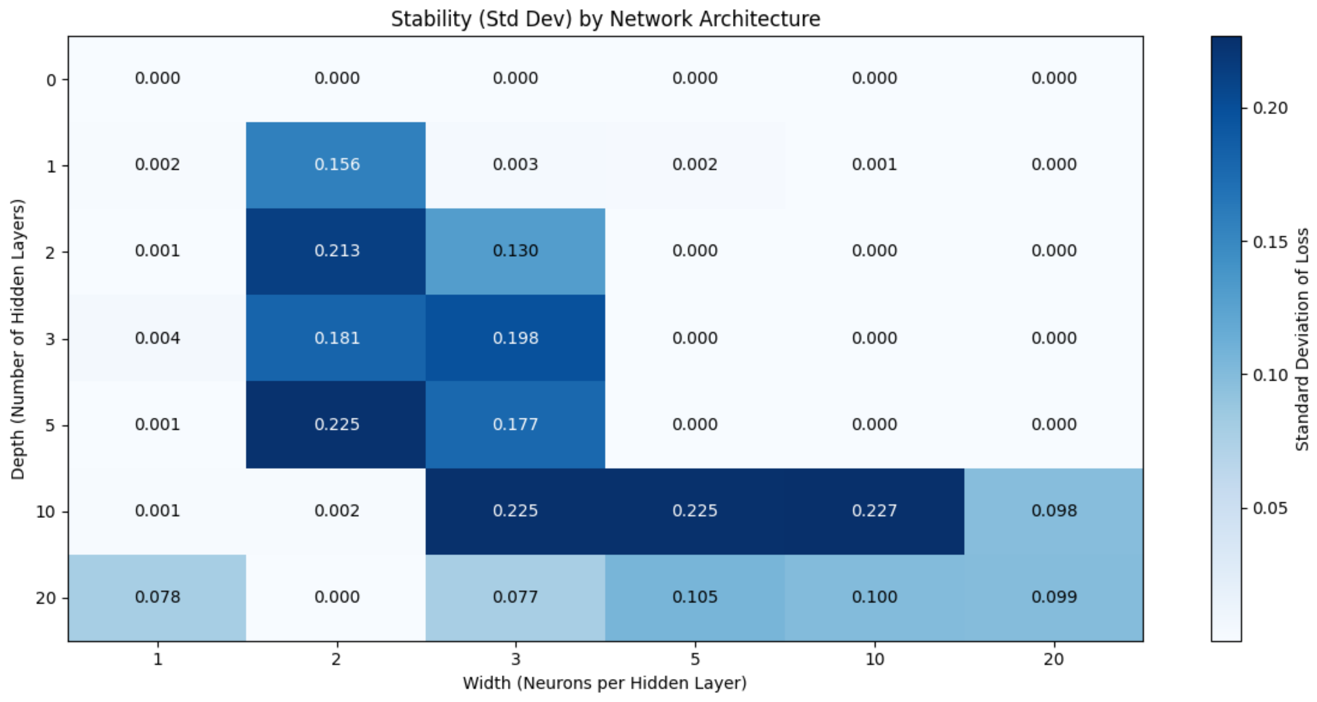


Figure 10



Depth \ Width	1	2	3	5	10	20
0	0.000	0.000	0.000	0.000	0.000	0.000
1	0.002	0.156	0.003	0.002	0.001	0.000
2	0.001	0.213	0.130	0.000	0.000	0.000
3	0.004	0.181	0.198	0.000	0.000	0.000
5	0.001	0.225	0.177	0.000	0.000	0.000
10	0.001	0.002	0.225	0.225	0.227	0.098
20	0.078	0.000	0.077	0.105	0.100	0.099

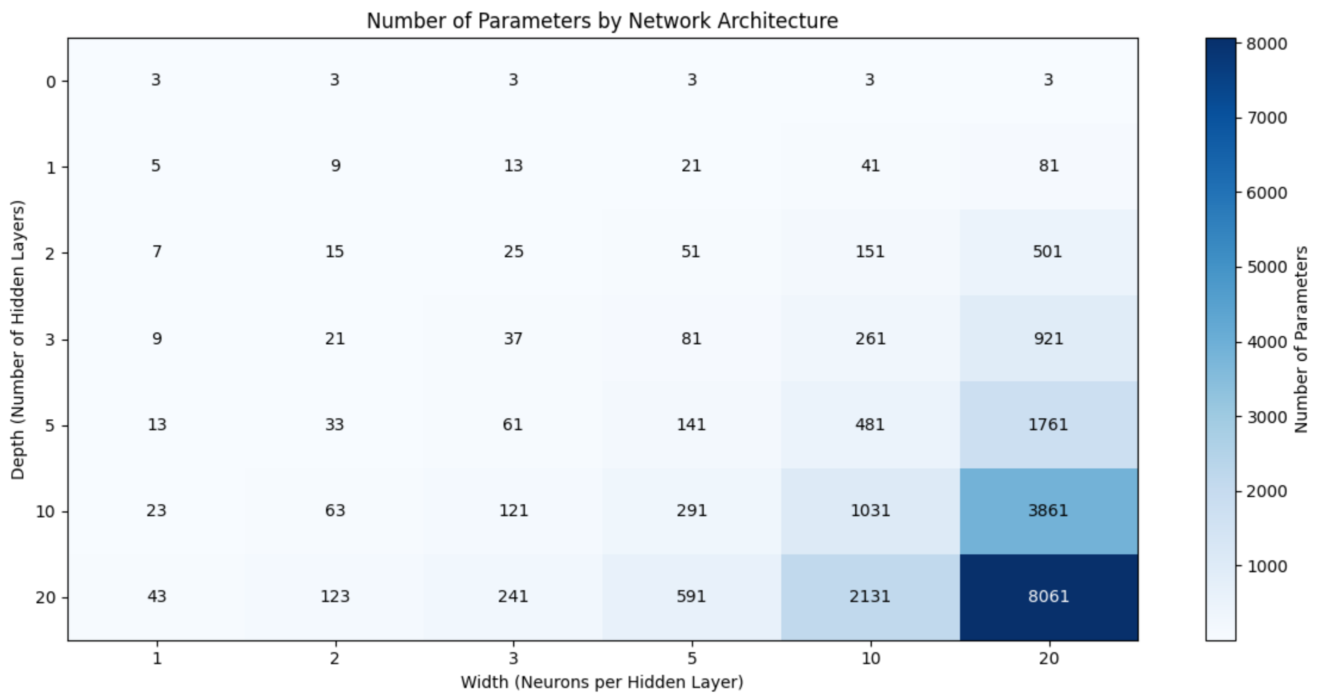
Figure 11

3.3

From the above results we see that it is generally better to have wide networks (more neuron in each layer), since it improves performance. We see this in the heat-map when we move from left to right (width 1 to 20), the loss drops dramatically (e.g., at Depth 2, it goes from 0.48 to 0.0001). We also notice that networks with a width of only 1 to 2 neurons perform relatively poorly regardless of how deep they are. The standard deviation of the loss also decreases as we use wider hidden layers, which must mean that the results get more stable as we widen the layers.

The other thing that we notice is that adding hidden layers helps initially. Going from 0 layers to 1, 2, 3 or 5 layers results in a massive drop in loss, provided the network is wide enough. Performance degrades significantly as the network becomes very deep (10 or 20 layers). The loss increases back up to high levels (e.g., > 0.5), likely due to optimization issues like vanishing gradients which make deep networks harder to train.

The best performance (lowest loss, represented by the lightest colors) is found in the depth of 2-5 layers and larger width of 5-20 neurons and probably even more. The model fails to learn effectively (high loss, dark blue) if it is either too shallow (0 layers), too narrow (1 neuron), or too deep (10+ layers).



Depth \ Width	1	2	3	5	10	20
0	3	3	3	3	3	3
1	5	9	13	21	41	81
2	7	15	25	51	151	501
3	9	21	37	81	261	921
5	13	33	61	141	481	1761
10	23	63	121	291	1031	3861
20	43	123	241	591	2131	8061

Figure 12

We use the Adam optimizer, since the theory tells us it is good at converging fast and probably better at finding the global minima in our problem than using sgd for example. We use a learning rate of 0.01. A higher learning rate might make us converge faster, but also risks overfitting.
We train for 25 epochs. This was primarily because we noticed that the loss does not reduce much more after 25 epocs, and we need to train a lot of models, so shorter training runs saves computation.
We use the tanh activation function for hidden layers, since it flattens out the outputs. Using relu might make us waste information if it does not activate the neuron.

We think that the parameters we have chosen are the best for the problem.

3.4

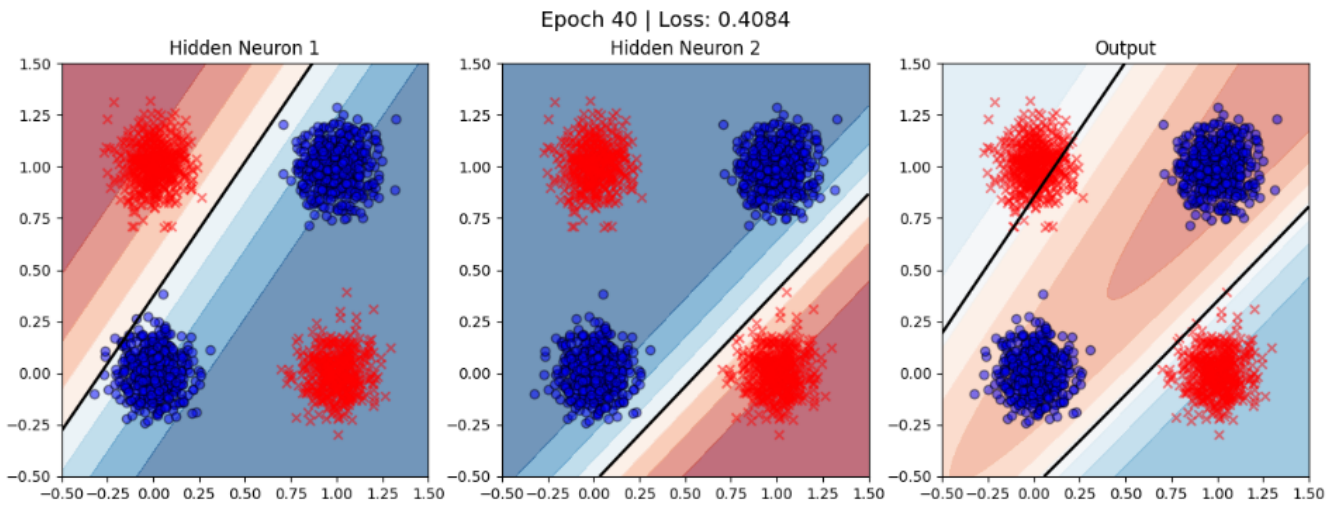


Figure 13

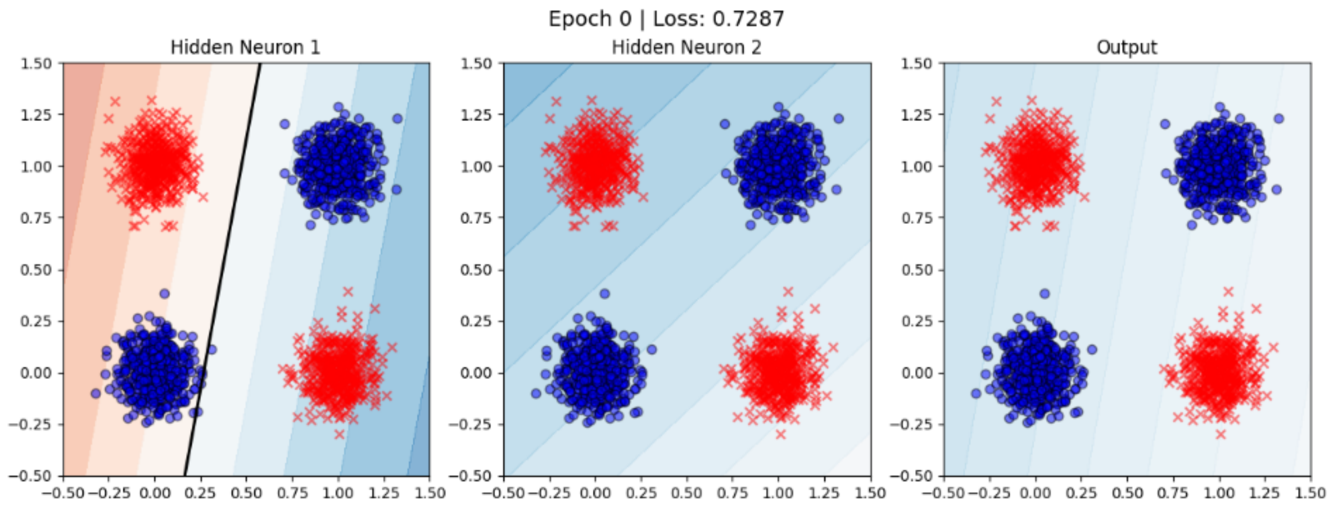


Figure 14

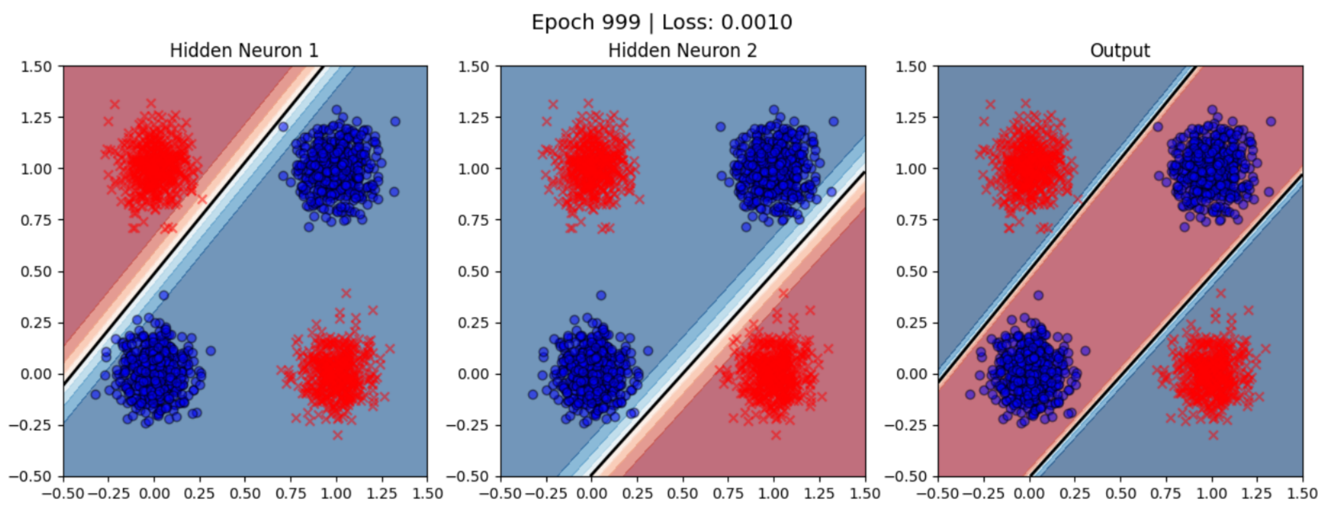


Figure 15

4 MLOps

4.1

To train, evaluate and log results to wandb we use:

```
wandb.agent(sweep_id, train_sweep, count=30)
```

which uses a training loop *train_sweep* that is very similar to our previous training function, but it uses the *.log* method to log the training results to wandb:

```
wandb.log({
    "epoch": epoch + 1,
    "train_loss": epoch_train_loss,
    "train_accuracy": epoch_train_acc,
    "val_loss": epoch_val_loss,
    "val_accuracy": epoch_val_acc,
})
```

We also use the *.watch* method to look at the gradiet at each computation in PyTorch:

```
wandb.watch(model, criterion, log="all", log_freq=10)
```

Using this we get the following results:

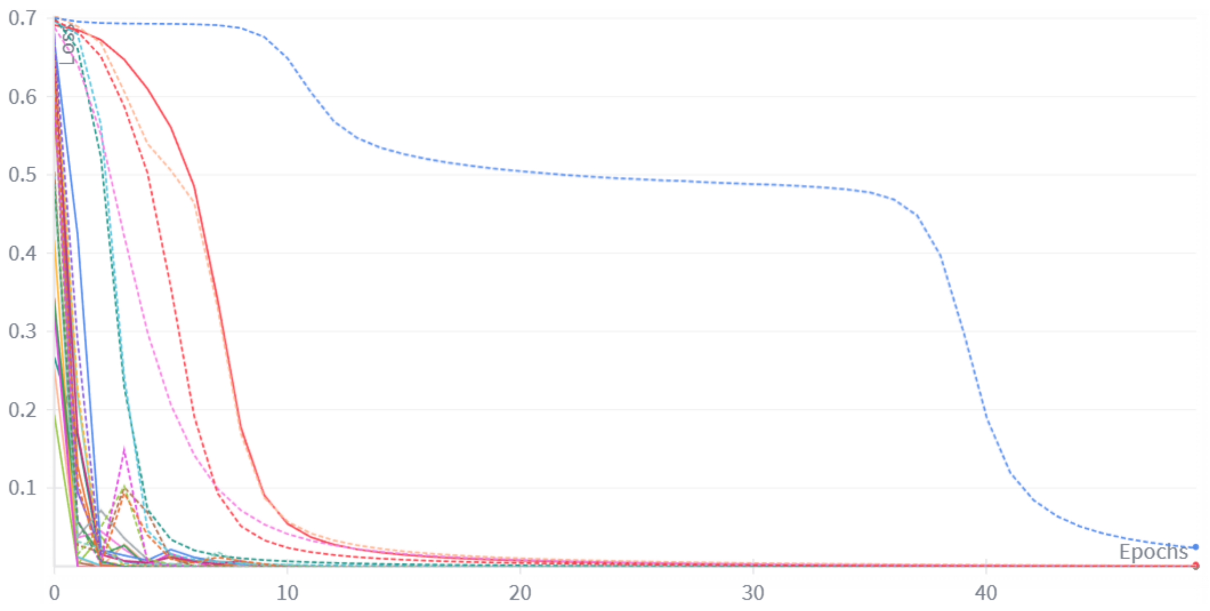


Figure 16: 30 training runs. y: Training loss, x: Epochs

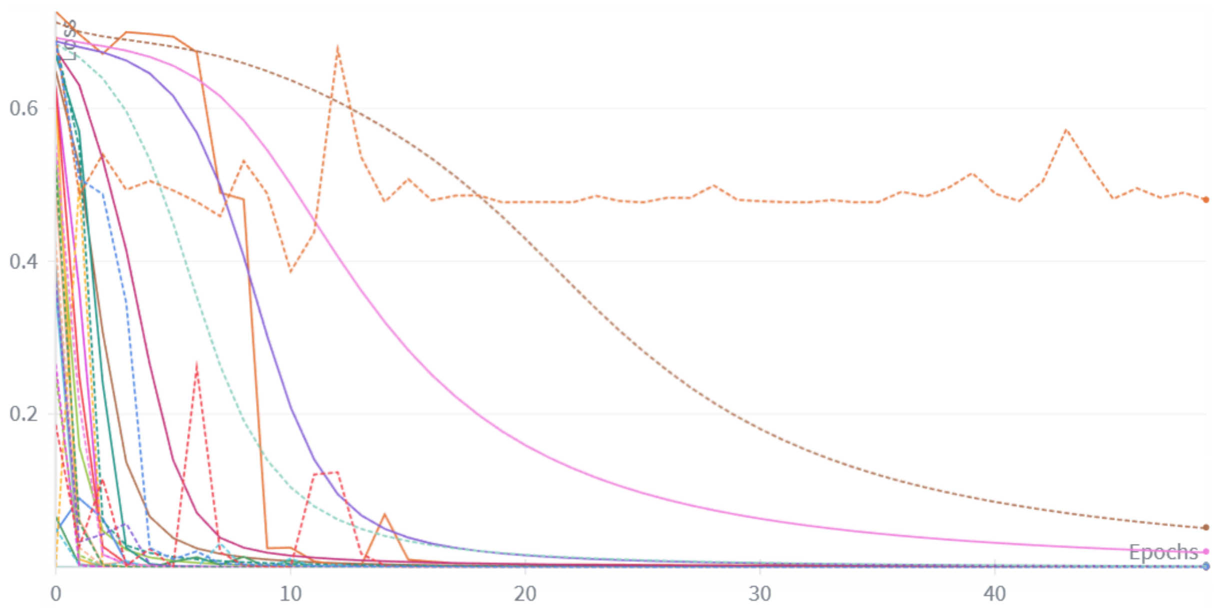


Figure 17: 30 training runs. y: Validation loss, x: Epochs

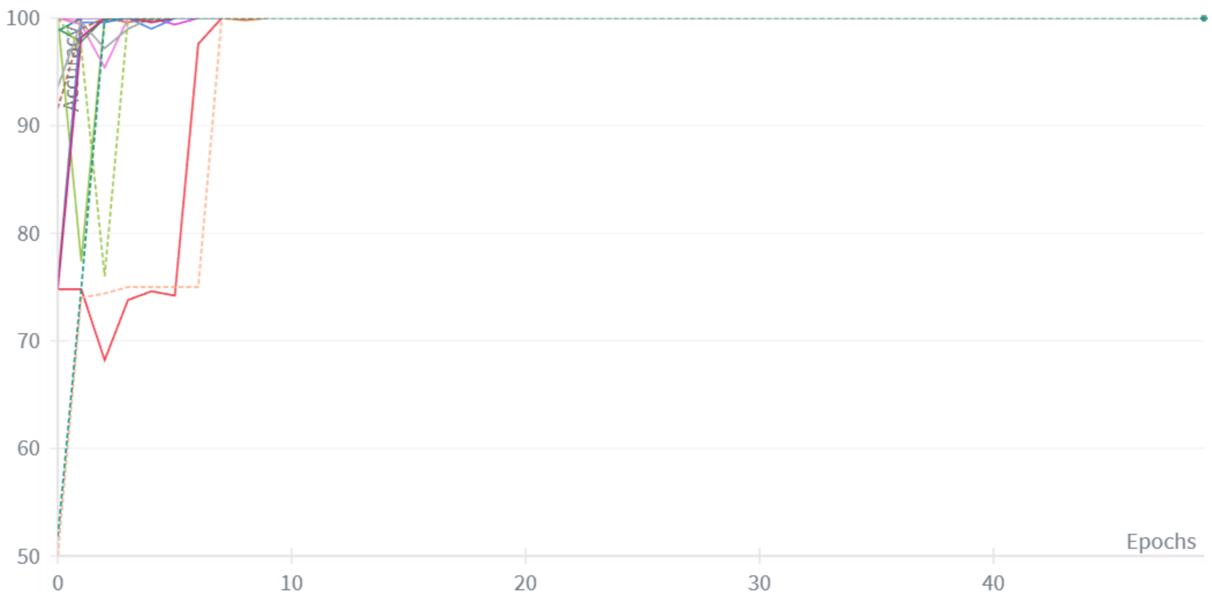


Figure 18: 30 training runs. y: Validation accuracy, x: Epochs

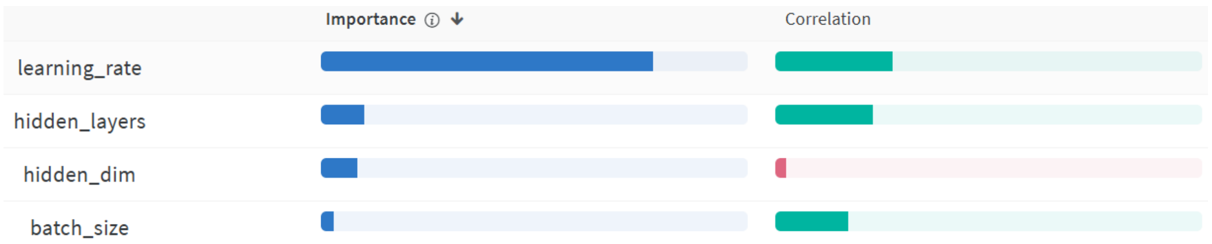


Figure 19: Importance and correlation between hyperparameters and validation loss

4.2

After sweeping over a range of hyperparameter values and get the following result:

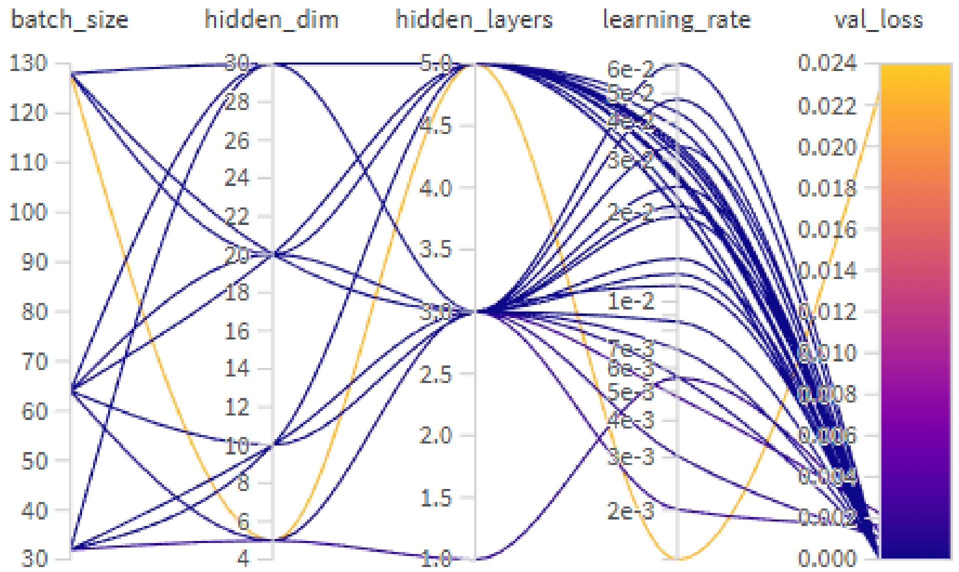


Figure 20: parallel coordinate plot for the 30 training runs over all hyperparameters

4.3

By using wandb we were able to sweep over a somewhat wide range of hyperparameters and log the results to our platform. The ranges we looked are defined in our code with the following:

```
sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'val_loss',
        'goal': 'minimize'
    },
    'parameters': {
        'hidden_layers': {
            'values': [1, 2, 3, 5]
        },
        'hidden_dim': {
            'values': [5, 10, 20, 30]
        },
        'learning_rate': {
            'distribution': 'log_uniform_values',
            'min': 0.001,
            'max': 0.1
        },
        'batch_size': {
            'values': [32, 64, 128]
        },
        'epochs': {
            'value': 50
        }
    }
}
```

We then train our models on different hyperparameters from the ranges on our training function from before.

This is how we ended up with our parrallel coordinate plot, which lets us know, which hyperparameters work the best together. We can roughly conclude from the parrallel coordinate plot and the importance plot that a higher learning rate results in a much lower validation loss. We also see that a lot of the best performing models had 3 hidden layers, while the models with 1 and 5 layers performed worse.

A rough conclusion is that a large amount of neurons in 3 hidden layers with a high learning rate results in the lowest loss, and since batch size did not have a great importance, we can let that vary.