Deep Learning 2025
Assignment 1

Emil Ramsbæk, Jonas Ellesøe & Nikolai Brinch

November 28, 2025

1 Backpropagation pen and paper

For the tasks in this section, we use the same notation as that presented in the provided notebook, DL_Backpropagation_pen_and_paper.ipynb.

Task c

The recursive equations for a feed-forward neural network (FFNN) with L layers can be written as follows

$$y_j = z_j^{(L)} \tag{1}$$

$$z_j^{(l)} = h_l\left(a_j^{(l)}\right) \qquad l = 1, \ldots, L \tag{2}$$

$$a_j^{(l)} = \sum_i^{M_{l-1}} w_{ji}^{(l)} z_i^{(l-1)} \qquad l = 2, \ldots, L \tag{3}$$

$$a_j^{(1)} = \sum_i^{D} w_{ji}^{(1)} x_i \tag{4}$$

The network output, y_j is given by the activations of the final layer, L, and each layer l applies its activation function h_l to the corresponding pre-activation $a_j^{(l)}$.

For layers $l = 2, \ldots, L$ each pre-activation is computed as a weighted sum of the activations from the previous layer. For the first layer, the pre-activations are linear combinations of the input variables, x_i.

Task j

The general backpropagation rule for $l < L$ can be written as

$$\frac{\partial E(w)}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)} \tag{5}$$

$$\text{where} \quad \delta_j^{(l)} = \sum_{k=1}^{K} \delta_k^{(l+1)} w_{kj}^{(l+1)} h_l'(a_j^{(l)}) \tag{6}$$

To understand why expression (5) holds, we examine its two components: the delta term $\delta_j^{(l)}$ and the activation $z_i^{(l-1)}$.

The delta term represents the error signal for neuron j in layer l and is defined as

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial a_j^{(l)}}$$

It measures how sensitive the loss is to changes in the pre–activation $a_j^{(l)}$, in other words, how much neuron j contributes to the total error, $E(w)$. In hidden layers ($l < L$), this dependence is indirect, so we apply the chain rule through the next layer. Each neuron k in layer $l+1$ contributes an error signal $\delta_k^{(l+1)}$, weighted by $w_{kj}^{(l+1)}$ and scaled by the activation derivative. This recursion in expression (6) illustrates the backward pass: the error signal from layer $l+1$ is propagated back to layer l.

The term $z_i^{(l-1)}$ comes from the structure of the pre–activation,

$$a_j^{(l)} = \sum_i w_{ji}^{(l)} z_i^{(l-1)}$$

which gives

$$\frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}} = z_i^{(l-1)}.$$

Thus, by the chain rule, $\delta_j^{(l)}$ tells us how the loss changes when $a_j^{(l)}$ changes, while $z_i^{(l-1)}$ tells us how $a_j^{(l)}$ responds to a change in the weight $w_{ji}^{(l)}$. Their product therefore gives the total effect of $w_{ji}^{(l)}$ on the loss, completing the general backpropagation rule for $l < L$.

2 AutoDiff framework

b)

1.

The first graph corresponds to

$$f = a \cdot b,$$

with missing derivatives

$$\frac{\partial f}{\partial a} = b, \qquad \frac{\partial f}{\partial b} = a.$$

The second graph represents the extended expression

$$c = ab, \qquad e = ad, \qquad f = c + e.$$

The missing derivatives are

$$\frac{\partial c}{\partial a} = b, \quad \frac{\partial c}{\partial b} = a, \quad \frac{\partial e}{\partial a} = d, \quad \frac{\partial e}{\partial d} = a, \quad \frac{\partial f}{\partial c} = 1, \quad \frac{\partial f}{\partial e} = 1.$$

These are simply the local chain-rule derivatives shown in the diagrams.

2.

For the simple expression $f = a \cdot b$, evaluating

```
a = Var(3.0)
b = Var(5.0)
f = a * b
```

creates three `Var` objects:

- a: value 3.0, gradient 0, no parents.
- b: value 5.0, gradient 0, no parents.
- f: value 15.0, gradient 0, and a `grad_fn` storing the two parents (a, b) together with their local derivatives b and a.

This object structure corresponds exactly to the first graph.

3.

Calling `f.backward()` starts backpropagation with seed value 1. The function `backprop(1)` on f adds 1 to f.grad and then propagates the two contributions

$$1 \cdot b \quad \text{to } a, \qquad 1 \cdot a \quad \text{to } b.$$

Since a and b have no parents, the recursion ends here. This reproduces the gradients predicted by the chain rule in the graph.

4.

The resulting call sequence for backpropagation is

$$\texttt{f.backprop(1)} \rightarrow \texttt{a.backprop(b)} \rightarrow \texttt{b.backprop(a)}.$$

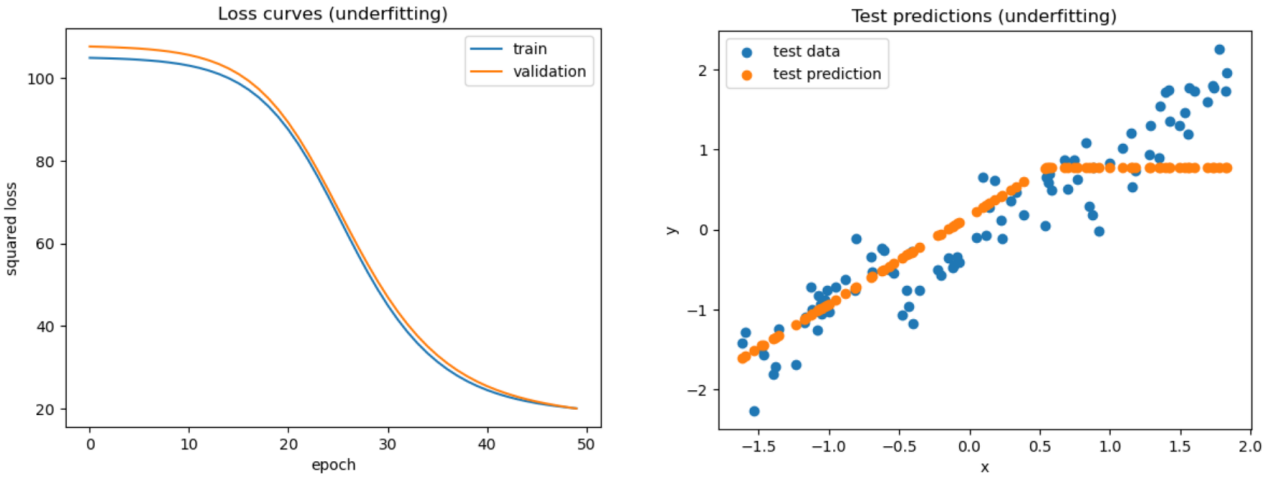
The last two calls terminate immediately because `a` and `b` have no parents.

l)

We used our feed-forward network on the 1D regression task with three different configurations and compared training, validation and test losses together with prediction–versus–target plots.

Underfitting.

For the underfitting case we used a very small network (one hidden layer with a single unit) and trained for only 50 epochs. The loss curves show that both training and validation losses decrease but remain relatively high (around 20 at the end of training), and they are almost identical. The test predictions form an almost piecewise linear curve that does not follow the scatter of the data very well, especially in the extreme regions of the input space. This indicates that the model does not have enough capacity to capture the underlying relationship in the data.



(a) Training and validation loss.

(b) Predictions vs. targets on the test set.

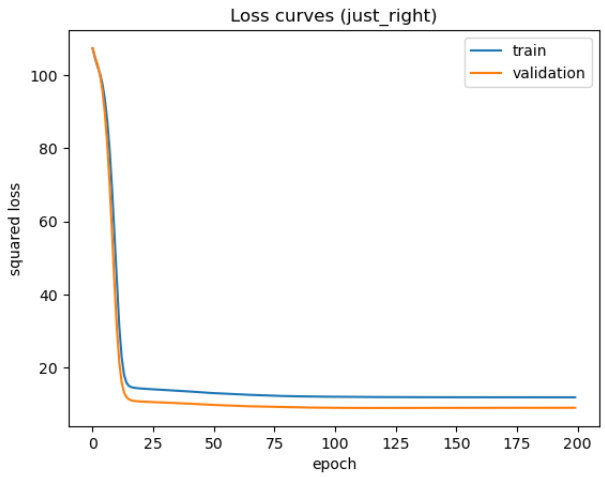
Figure 1: Performance of the underfitting configuration.

Just-right fitting.

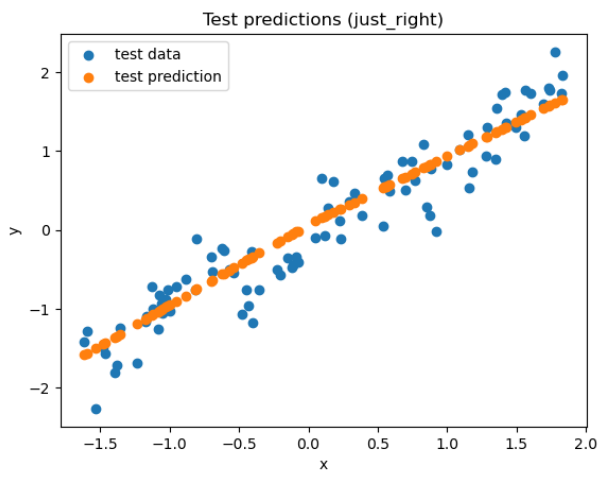
For the "just-right" configuration we used a moderate network with one hidden layer of five units and trained for 200 epochs. Here both training and validation losses drop quickly and then level off at considerably lower values (final training loss around 11.9 and validation loss around 9.0). The test predictions lie close to the trend in the data across the whole input range, and the prediction–versus–target scatter plot is concentrated near the diagonal. This suggests that the model has sufficient capacity to model the signal without fitting the noise too aggressively.

Overfitting.

To encourage overfitting we increased the capacity further (two hidden layers with 20 units each) and trained for 600 epochs. The training loss becomes slightly lower than in



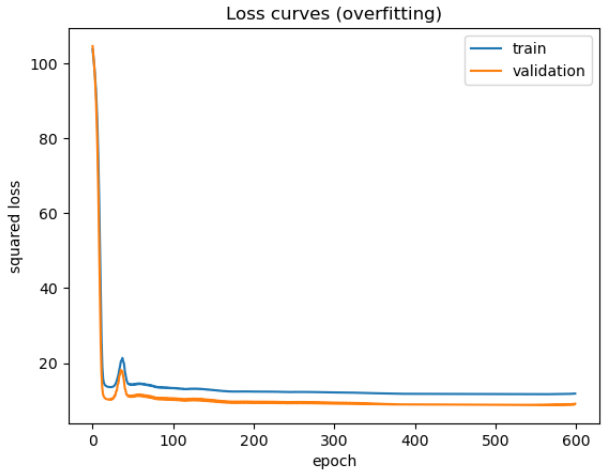
(a) Training and validation loss.



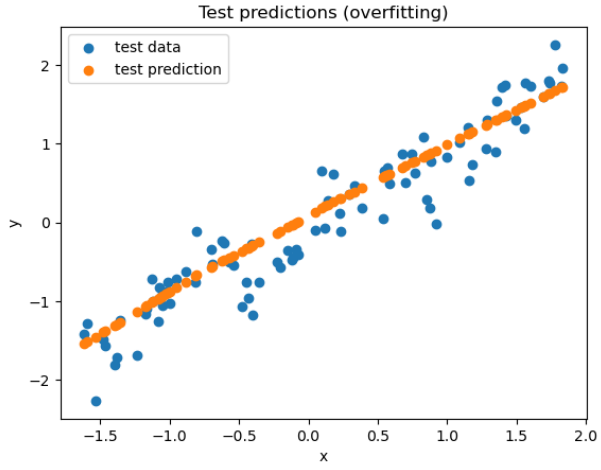
(b) Predictions vs. targets on the test set.

Figure 2: Performance of the ”just-right” configuration.

the underfitting case, but the improvement on the validation and test sets is small and in our runs the validation and test losses are in fact marginally worse than in the just-right configuration (final training loss $\approx$ 11.9, validation loss $\approx$ 9.2, test loss $\approx$ 10.2). The test predictions still follow the main trend well, but we see no dramatic improvement over the just-right model. Overall this configuration shows a mild tendency towards overfitting: the model is more complex, training loss is low, but the extra capacity does not translate into better generalization and slightly degrades validation and test performance.



(a) Training and validation loss.



(b) Predictions vs. targets on the test set.

Figure 3: Performance of the overfitting configuration.

Validation vs. test loss.

For this simple synthetic regression problem we did not observe a large systematic difference between validation and test loss. Both curves follow similar trends and end at comparable

values for all three configurations. This is expected, since the data are i.i.d. and the dataset is not extremely small. In more realistic scenarios, especially when doing extensive hyperparameter tuning or model selection, it is important to keep validation and test sets strictly separate. The validation set is used to choose architectures and training settings, while the test set should only be used once at the end to obtain an unbiased estimate of the final model performance.

3 Deep versus wide networks

1)

Parameter (value)	Meaning	If increased	If decreased
TOTAL (2000)	Total number of synthetic samples generated for train/val/test splits.	More data usually improves generalization and reduces variance, training takes longer and uses more memory.	Less data increases variance, potential underfitting, and unstable estimates, trains faster.
BATCH (64)	Number of samples per gradient update.	Larger batches give smoother gradient estimates, better throughput on GPUs, may require larger LR, can reduce generalization slightly.	Smaller batches give noisier gradients, can improve generalization and explore minima better but increase training time and overhead.
EPOCHS (100)	Number of full passes over the training set.	More epochs allow more training (risk of overfitting if too large) and can improve final accuracy if undertrained.	Fewer epochs may underfit (insufficient training) but reduce compute and overfitting risk.
LR (1e-2)	Learning rate for the optimizer (step size for parameter updates).	Larger LR speeds initial learning but can cause instability/-divergence or miss minima, may require smaller batch/regularization.	Smaller LR yields more stable but slower convergence and can get stuck in local minima, may need more epochs.

Table 1: Hyperparameters used and qualitative effects of changing them

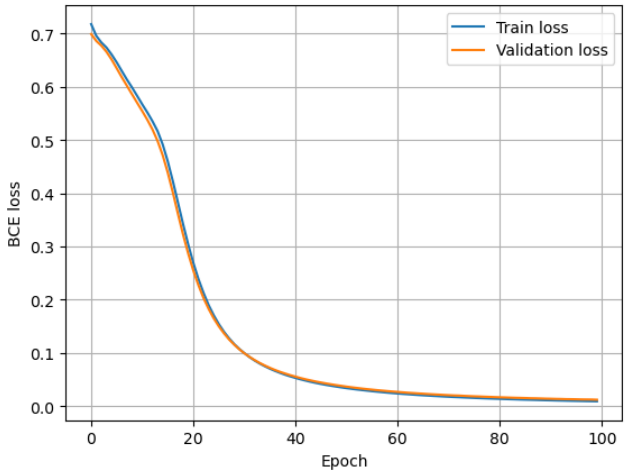


Figure 4: Visualization of Training and Validation Loss

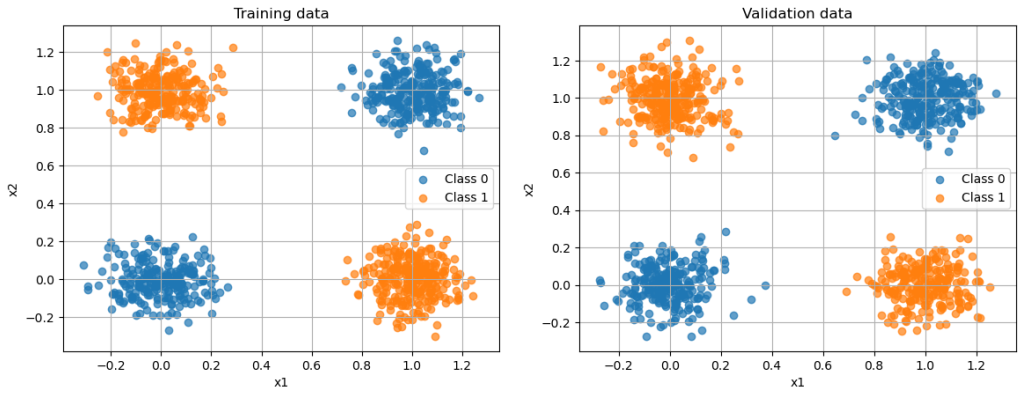


Figure 5: Visualization of Training and Validation Data

a)

PyTorch provides two common binary cross-entropy losses: `BCELoss`[1] and `BCEWithLogitsLoss`[2]. `BCELoss` expects probabilities in $[0, 1]$, so a sigmoid must be applied to the model's logits before calling it. `BCEWithLogitsLoss` accepts raw logits and internally applies a numerically stable sigmoid-plus-BCE computation (using log-sum-exp style operations). Therefore, when the model outputs logits directly, `BCEWithLogitsLoss` is preferred for numerical stability and convenience.

2)

Depth	Width	Mean test loss	Std test loss
0	1	0.695142	0.002363
0	2	0.694236	0.000843
0	3	0.694663	0.001265
1	1	0.483318	0.021258
1	2	0.306683	0.176301
1	3	0.076872	0.156942
2	1	0.482193	0.008123
2	2	0.073978	0.156446
2	3	0.071154	0.156839
3	1	0.485881	0.012889
3	2	0.072803	0.158735
3	3	0.001503	0.002095

Table 2: Mean and standard deviation of test loss for different network depths and widths on the noisy XOR task.

3)

The network with zero hidden layers corresponds to logistic regression and consistently fails to separate the four xor clusters, which is reflected in a much higher loss and clearly misaligned decision boundary. Once at least one hidden layer is introduced, the model can bend the decision boundary and the loss drops markedly. Increasing the width from 1 to 2 or 3 units typically improves the quality of the classification because the network can represent more complex nonlinear transformations, but the main gain comes from the first hidden layer. Adding additional hidden layers beyond what is needed for the xor structure tends to give only small improvements and in some runs slightly worse validation performance, which is consistent with overfitting and with the fact that deeper networks are harder to optimize on a small synthetic data set.

For a fully connected network with two inputs, one output and L hidden layers of width h, all with biases, the total number of parameters is $2h + h$ for the first layer, $(h^2 + h)$ for each additional hidden layer, and $h + 1$ for the output layer. This gives a total of $3h + (L-1)(h^2 + h) + (h + 1)$ parameters for $L \geq 1$, while the purely linear model with $L = 0$ has only $2 \cdot 1 + 1 = 3$ parameters. Substituting $L \in \{0, 1, 2, 3\}$ and $h \in \{1, 2, 3\}$ yields the concrete counts for the tested architectures: depth 0 (any width) has 3 parameters; depth 1 with widths $1, 2, 3$ has $5, 9, 13$ parameters; depth 2 with widths $1, 2, 3$ has $7, 15, 25$ parameters; depth 3 with widths $1, 2, 3$ has $9, 21, 37$ parameters. Thus the deeper and wider networks quickly become much more expressive but also much more parameter rich than the minimal xor architecture.

The conclusions above are conditioned on the specific training hyperparameters summarised in the table of TOTAL, BATCH, EPOCHS and LR. With TOTAL fixed at 2000 points the deeper and wider networks have relatively few samples per parameter, so increasing TOTAL

would likely stabilise the estimates and reduce overfitting, making the differences between architectures clearer. The chosen batch size and learning rate balance stability and speed of convergence; for example, a much smaller learning rate or many fewer epochs could make deeper networks appear worse simply because they did not converge, while a much larger learning rate could destabilise training especially for the largest models. Similarly, changing the batch size modifies gradient noise and may slightly affect which architectures generalise best. In other words, the apparent effect of depth and width should be interpreted together with these training choices rather than in isolation.

4)

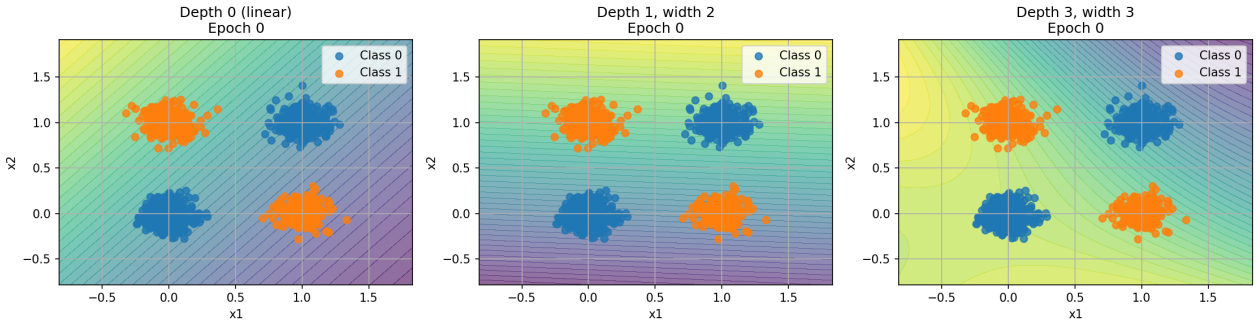


Figure 6: Animation works in Adobe Acrobat Reader (Windows / MacOS) or Okular (Linux). Can also be found in the code handin.

4 MLOps

We use the Weights Biases (WB) MLOps platform, as recommended, to monitor the training of our noisy-XOR classifier and to run a small hyperparameter sweep.

1.

We consider the implementation in `feedforwardAssignment.ipynb` and based on the results from the the previous part (part 3), we select a promising network configuration: one hidden layer with three hidden units, `depth = 1`, `width = 3`, and train it on the noisy-XOR data source.

To integrate W&B, we wrap the existing training loop in a W&B run and log a few scalar metrics per epoch. The only code changes are

```
#Initialising a run with the configuration:
wandb.init(project="noisy-xor-mlp", config=config)
```

and

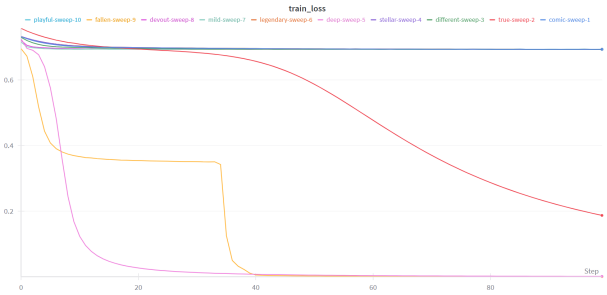
```
#Logging the metrics in each epoch:

wandb.log({
    "epoch": epoch,
    "train_loss": train_loss_epoch,
    "val_loss": val_loss_epoch,
    "val_accuracy": val_acc_epoch,
    "best_val_loss": best_val_loss
})
```

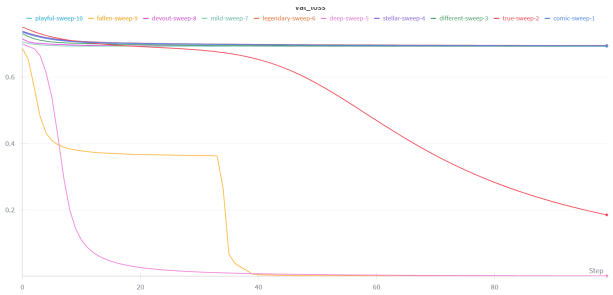
Apart from these additions, the training code is identical to the implementation used earlier (same data generation, architecture, optimiser, and binary cross-entropy loss). The WB dashboard then automatically produces the corresponding loss and accuracy curves for the run.

Figure 7 shows the W&B training curves for all runs in our experiment, including the single-run configuration required in this task. Each coloured curve corresponds to one configuration from the later sweep.

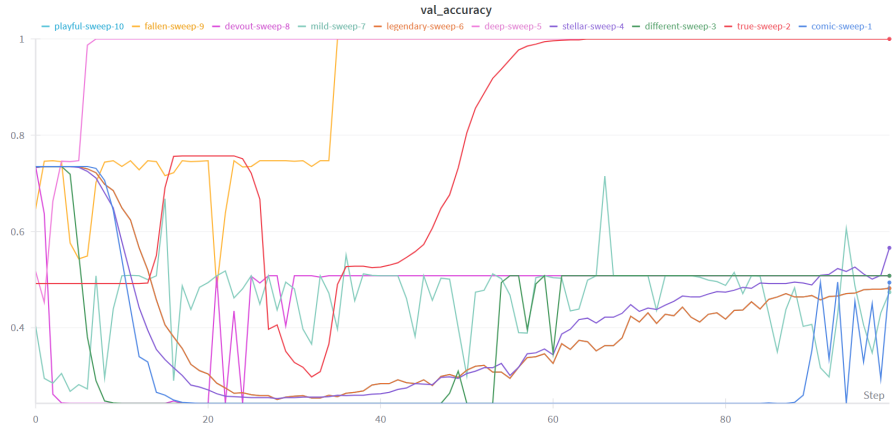
Each colour corresponds to one run with a different configuration. Well-performing runs quickly drive both training and validation loss towards zero while validation accuracy approaches one, whereas poor configurations remain near a loss of ≈ 0.69 and an accuracy around 0.5 (just like random guessing).



(a) Training loss for all runs.



(b) Validation loss for all runs.



(c) Validation accuracy for all runs.

Figure 7: Training progress tracked with W&B.

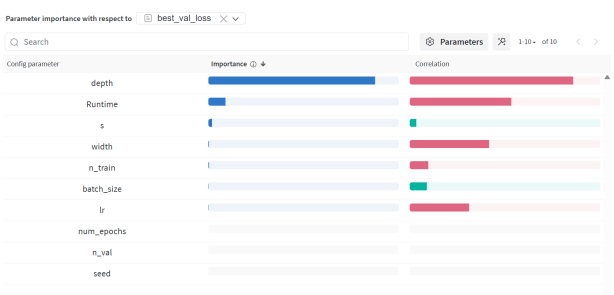
2.

For the second MLOps task we implement a W&B sweep to explore the influence of architectural and optimisation hyperparameters on performance. We use a random search over the following parameters:

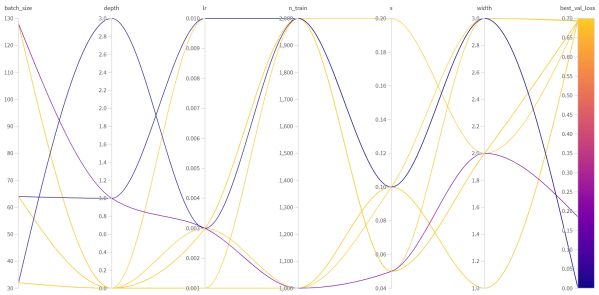
- **Depth** (`depth`): number of hidden layers $\in \{0, 1, 2, 3\}$.
- **Width** (`width`): hidden units per layer $\in \{1, 2, 3\}$.
- **Learning rate** (`lr`): $\{10^{-3}, 3 \cdot 10^{-3}, 10^{-2}\}$.
- **Batch size** (`batch_size`): $\{32, 64, 128\}$.
- **Training set size** (`n_train`): $\{1000, 2000\}$.
- **Noise level** (`s`): $\{0.05, 0.1, 0.2\}$.

The number of epochs is fixed to 100, and each sampled configuration is run once, giving ten runs in total. We use the best validation loss (`best_val_loss`) as the sweep metric to be minimised.

The W&B provides several built-in plots for analysing the sweep. In this report we include a parameter-importance plot and a parallel-coordinates plot, shown in Figure 8. The former estimates how strongly each hyperparameter influences the sweep metric, while the latter visualises all runs in a single panel, with each polyline coloured by its `best_val_loss`.



(a) Parameter importance for best validation loss.



(b) Parallel-coordinates plot coloured by best validation loss.

Figure 8: W&B visualisations of the hyperparameter sweep.

The importance plot suggests that depth is the most influential parameter in the explored range. The parallel-coordinates plot shows that low best validation loss is achieved by deeper networks with sufficient width and appropriate learning rate, while hallow (or linear) networks tend to perform poorly.

3.

The sweep results confirm the intuition from part 3: "Deep versus wide networks" experiment. Configurations with `depth = 0` (no hidden layers) behave like linear models and

are unable to solve the noisy-XOR problem: they remain close to a validation loss of ≈ 0.69 and an accuracy around 0.5, corresponding to random guessing. In contrast, networks with at least one hidden layer and moderate width can achieve very low validation loss and near-perfect accuracy.

In our sweep, we systematically varied *depth*, *width*, *learning rate*, *batch size*, the noise level s, and the number of training samples. The parameter-importance analysis (Figure 8a) indicates that depth has the largest effect on the best validation loss, followed by the noise level s and, to a lesser extent, the learning rate. Within the tested ranges, batch size and training set size have comparatively smaller impact on the final performance. The parallel-coordinates plot (Figure 8b) supports this: the lowest-loss runs correspond to deeper networks (typically `depth = 2` or 3) with width around three units and learning rate 0.01.

Overall, the MLOps experiments illustrate how W&B can be used to track training behaviour, analyse hyperparameter importance through tools such as parallel-coordinates plots, run systematic hyperparameter sweeps, and quickly identify promising regions of the hyperparameter space for further study.

References

[1] PyTorch Documentation. *torch.nn.BCELoss*. Online; accessed 2025-11-24. 2025. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.BCELoss.html.

[2] PyTorch Documentation. *torch.nn.BCEWithLogitsLoss*. Online; accessed 2025-11-24. 2025. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html.