Deep Learning 2025
Assignment 1

Emil Vang Klinker, dkz250
Simon Hvidfeldt, zgh166
William Saugmann, sgz655

October 29, 2026

1 Backpropagation pen and paper

This section contains solutions to exercises **c)** and **j)** from the notebook `DL_Backpropagation_pen_and_paper.ipynb`.

Exercise c:

For a feed-forward neural network with L layers, we eliminate all a's and z's and write the output directly as nested activation functions.

Let $y_m^{(0)} = x_m$ with $x_0 = 1$ (bias input). For each layer $l = 1, \ldots, L$ we define

$$y_j^{(l)} = h_l\left(\sum_{i=0}^{K_{l-1}} w_{ji}^{(l)}\, y_i^{(l-1)}\right).$$

Expanding this recursively yields the complete expression for the network output:

$$y_j^{(L)} = h_L\left(\sum_{i_{L-1}} w_{j\,i_{L-1}}^{(L)}\; h_{L-1}\left(\sum_{i_{L-2}} w_{i_{L-1}\,i_{L-2}}^{(L-1)}\; h_{L-2}\left(\cdots h_1\left(\sum_{m} w_{i_1\,m}^{(1)}\, x_m\right)\cdots\right)\right)\right).$$

This matches the structure from exercises a) and b), generalized to L layers.

Exercise j: Backpropagation rule for layer $l < L$

Exercise i introduced the error signal

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

the derivative of the loss E with respect to the pre–activation $a_j^{(l)}$ of neuron j in layer l. For the output layer L we showed that

$$\delta_j^{(L)} = y_j - t_j,$$

which is the discovery that the error signal is simply the prediction minus the target. In this exercise we extend the same idea to hidden layers.

We derive the two required expressions:

Gradient with respect to the weights. The pre-activation in layer l is

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

where $z_i^{(l-1)} = h_{l-1}(a_i^{(l-1)})$ is the activation from the previous layer. Using the chain rule, we obtain

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l)}}.$$

By definition of $\delta_j^{(l)}$ and by differentiating the linear expression for $a_j^{(l)}$ we get

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

Thus

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

which is the same structure found in Exercise i: error signal times input.

Backpropagation recursion for the error signal. For the output layer L, Exercise i showed

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

For a hidden layer $l < L$, the pre–activation $a_j^{(l)}$ influences the loss only *through the next layer*. Applying the chain rule over all units k in layer $l + 1$ gives

$$\frac{\partial E}{\partial a_j^{(l)}} = \sum_k \frac{\partial E}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}}.$$

We evaluate the three factors one by one:

$$\frac{\partial E}{\partial a_k^{(l+1)}} = \delta_k^{(l+1)}, \qquad \frac{\partial a_k^{(l+1)}}{\partial z_j^{(l)}} = w_{kj}^{(l+1)}, \qquad \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} = h_l'(a_j^{(l)}).$$

Substituting into the chain rule yields

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \, w_{kj}^{(l+1)} \, h_l'(a_j^{(l)}).$$

Reordering gives the standard backpropagation recursion:

$$\delta_j^{(l)} = h_l'(a_j^{(l)}) \sum_k \delta_k^{(l+1)} \, w_{kj}^{(l+1)}$$

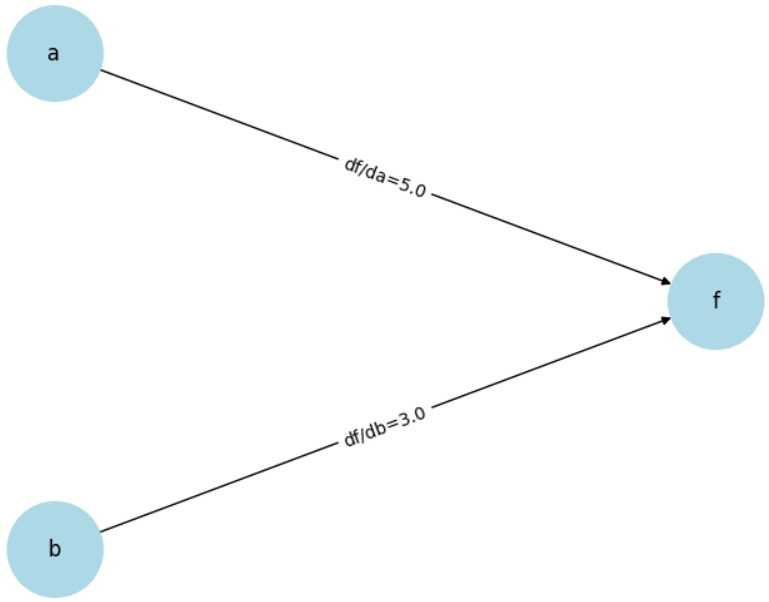
This formula shows that the "error" from layer $l+1$ is passed backward: it is weighted by the outgoing weights $w_{kj}^{(l+1)}$ and scaled by the derivative of the activation function $h_l'(a_j^{(l)})$. Together with the output-layer expression from Exercise i, this gives a complete recursive procedure for computing all error signals $\delta^{(l)}$.

2 AutoDiff framework

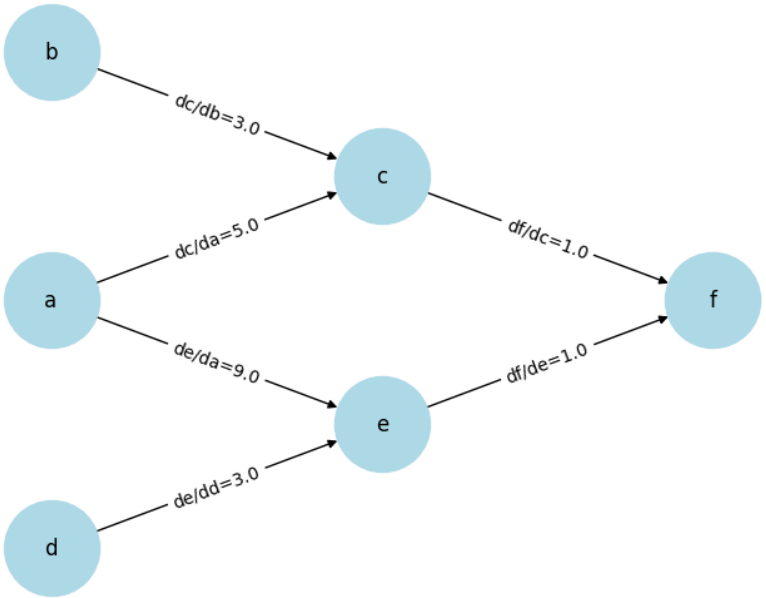
This section contains the required solutions to exercises **b)** and **l)** in `DL_AutoDiff_Nanograd.ipynb`.

Exercise b: How does the backward function work?

Below is the first two expressions as graphs. We have filled in the missing expressions for the different derivatives.



(a) Directed graph of example 1.



(b) Directed graph of example 2.

In the remainder we only consider the first expression.

Object	**Value** (`v`)	**Gradient** (`grad`) (before `backward()`)	**grad_fn**
a	3.0	0.0	`lambda: []`
b	5.0	0.0	`lambda: []`
f	15.0	0.0	`lambda: [(a, 5.0), (b, 3.0)]`

Table 1: Schematic of the data structure

The function `grad_fn` for `f` is essential. It returns a list of tuples, where each tuple contains a input, which is either `a` or `b` and the gradient of `f` with respect to the input.

We will now execute the backward function by hand, to convince ourself that it indeed calculates the gradients with respect to the variables.

Step 1:

`f.backwards()` is called. The function starts differentiating by calling `f.backprop(1.0)`. The value 1.0 represents $\frac{\partial f}{\partial f}$, because we want to differentiate the output with respect to itself.

Step 2:

`f`'s accumulated gradient `f.grad` is updated to $0.0 + 1.0 = 1.0$

`f` iterates through its `grad_fn`, which returns its local gradients and input, which are `[(a, 5.0), (b, 3.0)]`.

For each input `backprop` is called recursively with the updated gradient `grad * bp`.

`a.backprop(5.0 * 1.0) = a.backprop(5.0).`

`b.backprop(3.0 * 1.0) = b.backprop(3.0)`

Step 3:

`a`'s accumulated gradient `a.grad` is updated to $0.0 + 5.0 = 5.0$

`a`'s `grad_fn` is empty, so no further backpropagation.

Step 4:

`b`'s accumulated gradient `b.grad` is updated to $0.0 + 3.0 = 3.0$

`b`'s `grad_fn` is empty, so no further backpropagation.

We will now write down the sequence of calls to `backprop`. The sequence of calls to `backprop` for the expression $f = a \cdot b$ is:

`f.backprop(1.0)`: The first call from `f.backward()`.

`a.backprop(5.0)`: The call from `f` with the accumulated gradient $\frac{\partial f}{\partial a} = 5$

`b.backprop(3.0)`: The call from `f` with the accumulated gradient $\frac{\partial f}{\partial b} = 3.0$.

Exercise l: Show overfitting, underfitting and just right fitting

For this exercise we tried to vary the architecture and other things to show clear signs of overfitting and underfitting.

Overfitting When trying to force the model to overfit we increased the models complexity. By using 100 neurons in the hidden layer for a simple 1 dimensional problem the network should have too many parameters. Which should result in the model beginning to memorize instead of learning.

Additionally, we used the activation function `identity()` along with `hyperbolic_tangent()` in our architecture.

```
# Overfitting
NN = [
    DenseLayer(1, 100, lambda x: x.hyperbolic_tangent()),
    DenseLayer(100, 1, lambda x: x.identity())
]

# Initialize training hyperparameters
EPOCHS = 200
LEARN_R = 2e-3
```

This is to introduce the non-linearity to the model.

Underfitting When trying to get the model to underfit we did the opposite. We decreased the amount of neurons in the hidden layer to two. We should see that the model performs poorly on both the training and test set.

This time we tried to use two `identity()` functions in our architecture.

```
# Underfitting

NN = [
    DenseLayer(1, 2, lambda x: x.identity()),
    DenseLayer(2, 1, lambda x: x.identity())
]

# Initialize training hyperparameters
EPOCHS = 200
LEARN_R = 2e-3
```

This should in theory restrict the model to learning only linear relationships.

Just right To hit the sweet-spot we kept the hidden layers at a moderate 8, which should result in the model capturing the general trend without memorizing noise.

Here we used the two activator functions `relu()` and `identity()`.

```
#Just right
NN = [
    DenseLayer(1, 8, lambda x: x.relu()),
    DenseLayer(8, 1, lambda x: x.identity())
```

```
]

# Initialize training hyperparameters
EPOCHS = 200
LEARN_R = 2e-3
```

We did this to get a balanced model between flexibility and generalization.

So, did we see any big difference between validation and test loss? The answer here is no.

Overfitting	**Underfitting**	**Just right**
Train loss: 12.182	Train loss: 12.031	Train loss: 11.878
Validation loss: 9.049	Validation loss: 8.979	Validation loss: 8.922
Test loss: 9.470	Test loss: 9.343	Test loss: 9.767

Table 2: Comparison between overfitting, underfitting and "just right"

The above table summarizes the final loss metrics for the three models architecture. The **Just right** architecture achieved the lowest training and validation losses with 11.878 and 8.922. This indicates that it achieved the most effective balance between model capacity and generalization, but the others are not far behind.

In reality, we don't see any big difference between the validation and test loss. This is most likely due to how our data was generated. To generate the data we sampled from the function $0.5\sin(3x) + x$, and then randomly shuffled the data, which means they have the same distribution.

Lastly, when choosing hyper parameters it is important we keep our training, and test data separate. In the validation set we use it to make decisions. Because we make decisions based on this data the model sees it and adapts to it. The training set must remain unseen as it is the final unbiased check to see how the model performs on data it has never seen.

3 Deep versus wide networks

The implementation for this problem is contained in `feedforwardAssignment.ipynb`. Note that we haven't used any random seeds, so the report may vary from our notebook.

3.1 Noisy XOR dataset and decision boundaries

Following the assignment, we generate data points according to

$$(x_1, x_2, y) = (m_1 + s\eta_1,\ m_2 + s\eta_2,\ m_1 \oplus m_2),$$

where $(m_1, m_2) \in \{(0,0), (0,1), (1,0), (1,1)\}$ are chosen uniformly, $\eta_1, \eta_2 \sim \mathcal{N}(0,1)$, and s is a user-defined noise level. The XOR label is $m_1 \oplus m_2$.

For binary classification, the decision boundary is defined as the curve in input space where the model predicts equal probability for both classes:

$$p(y = 1 \mid x_1, x_2) = 0.5.$$

For a linear model

$$p(y = 1 \mid x) = \sigma(w_1 x_1 + w_2 x_2 + w_0),$$

the boundary is the straight line $w_1 x_1 + w_2 x_2 + w_0 = 0$. Since XOR is not linearly separable, a single linear decision boundary cannot separate the two classes. Neural networks with hidden units can solve this by creating non-linear transformations of the input space.

We implemented the dataset as a subclass of `torch.utils.data.Dataset`. Figure 2 shows an example of the generated data for $s = 0.1$.

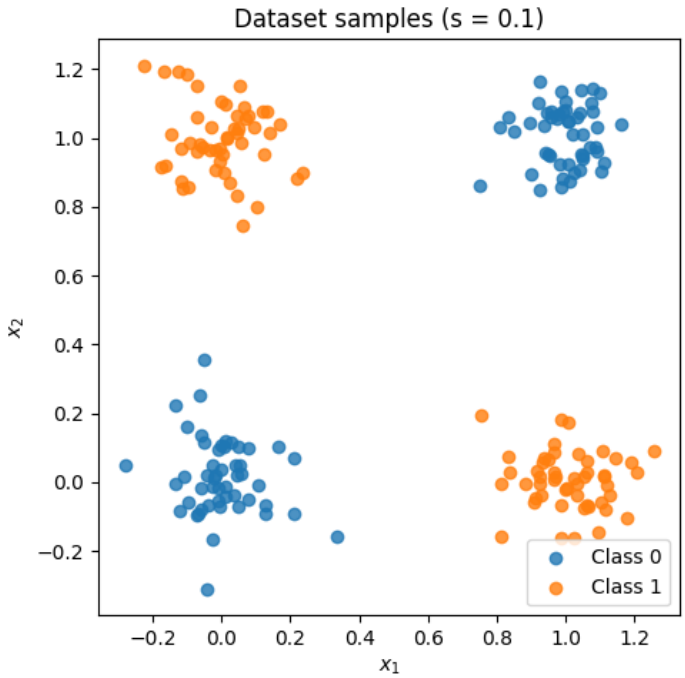


Figure 2: Example of the noisy XOR data ($s = 0.1$). Blue points correspond to class 0, orange points to class 1.

3.1 (a) Binary cross-entropy: BCELoss vs. BCEWithLogitsLoss

PyTorch provides two variants of the binary cross-entropy loss:

- `BCELoss` Expects probabilities in $[0, 1]$. The user must apply `sigmoid` manually. This can cause numerical instability when logits are large or small.
- `BCEWithLogitsLoss` Expects raw logits. Applies the sigmoid internally using the numerically stable log-sum-exp trick. This is the recommended choice in virtually all binary classification tasks.

We therefore use `BCEWithLogitsLoss` throughout this assignment.

3.2 Neural network model

We implemented a configurable feed-forward neural network of the form

$$2 \to H_1 \to H_2 \to \cdots \to H_L \to 1,$$

where we vary:

- number of hidden layers $L \in \{0, 1, 2, 3\}$,
- hidden width $W \in \{1, 2, 3\}$ (same in every layer),
- activation function in hidden layers: tanh,
- output activation: identity (logit).

The loss function is `BCEWithLogitsLoss`, and we train using mini-batch gradient descent with the Adam optimizer. We also implemented visualisation to match Figure 1 of the assignment for the training dataset and a held-out validation dataset.

3.3 Depth/width sweep and parameter count

To evaluate how depth and width affect the ability to learn the noisy XOR dataset, we systematically tried all combinations of

$$L \in \{0, 1, 2, 3\}, \qquad W \in \{1, 2, 3\}.$$

For each of the 36 architectures, we:

1. generated a fresh noisy XOR dataset,
2. trained the model for a fixed number of epochs,
3. repeated the training $n = 10$ times with different random seeds,
4. recorded the mean and standard deviation of the *validation loss*,
5. computed the number of trainable parameters.

Layers	Width	Mean val. loss	Std	Params
0	1	0.693795	0.000770	3
0	2	0.693345	0.000427	3
0	3	0.694028	0.000426	3
1	1	0.515078	0.029975	5
1	2	0.288591	0.210250	9
1	3	0.055189	0.017212	13
2	1	0.511108	0.064127	7
2	2	0.111326	0.197281	15
2	3	0.008792	0.005870	25
3	1	0.487364	0.002143	9
3	2	0.151200	0.229470	21
3	3	0.004351	0.004827	37

Table 3: Validation loss across architectures for the depth/width sweep (final run).

Interpretation of depth

- With $L = 0$ (purely linear), the network cannot learn XOR — loss stays near 0.693.
- With a single hidden layer ($L = 1$), the model becomes expressive enough to separate the classes, provided the width is at least 2.
- Increasing depth to $L = 2$ or $L = 3$ improves performance further, but the improvement is modest on such a simple task.

Interpretation of width

- Width $W = 1$ is insufficient unless the network is deep.
- Width $W = 2$ already solves XOR (matching the minimal XOR network).
- Width $W = 3$ works well but mostly increases parameter count.

Effect of training hyperparameters We used:

- Adam optimizer,
- fixed learning rate,
- fixed batch size,
- identical number of epochs for all runs.

If these were changed (e.g. larger learning rate or fewer epochs), the conclusions could differ because some architectures converge more slowly than others. Fixing the same training hyperparameters ensures a fair comparison.

3.4 Minimal XOR network and evolving decision boundaries

The Absalon notebook `XOR_NNsandbox.ipynb` shows that a minimal network with architecture

$$2 \to 2 \to 1,$$

and tanh activation can solve XOR by forming two linear hidden-unit boundaries whose combination yields the non-linear XOR separator.

We reimplemented this model, trained it on the noisy XOR dataset, and saved snapshots of the weights every 20 epochs. From these snapshots we plotted how the hidden-unit decision boundaries evolve.

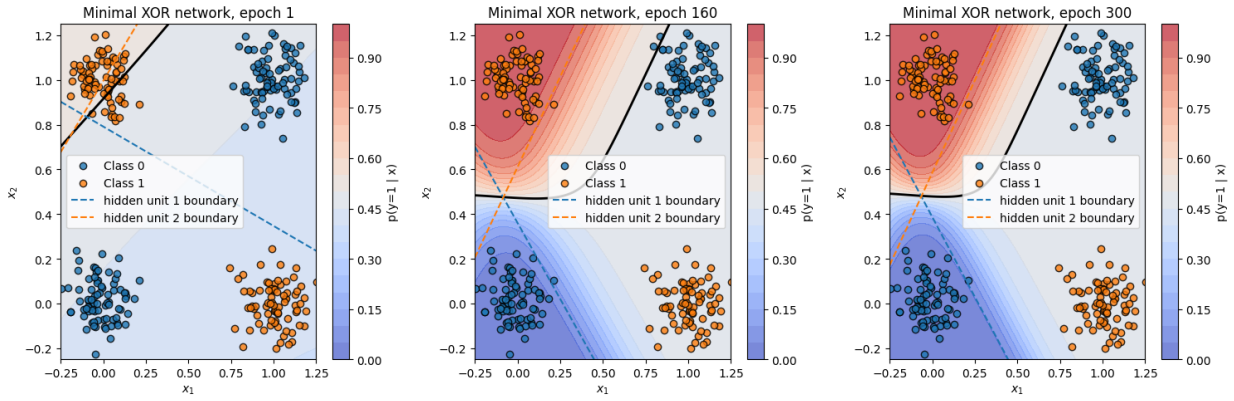


Figure 3: Evolution of decision boundaries for the minimal XOR network at epochs 1, 160, and 300. Each plot shows the two hidden-unit linear boundaries (dashed) and the final network decision boundary (0.5-contour).

Interpretation

- **Early training (epoch 1):** The hidden-unit boundaries are roughly random and do not align with any XOR structure.
- **Middle training (epoch 160):** The hidden units have rotated and begun to separate the diagonal clusters.
- **Late training (epoch 300):** The two hidden units form the characteristic "X" shape, and the output unit combines them into the correct non-linear XOR boundary.

This experiment provides intuition for how small neural networks solve non-linear tasks: by creating multiple linear partitions in transformed space and combining them through subsequent layers.

4 MLOps

The goal of this part of the assignment was to use an MLOps platform to monitor training, log metrics, and perform a small hyperparameter sweep. We used Weights & Biases (W&B), as it integrates easily with PyTorch and provides convenient plotting tools.

4.1 Logging a single feed-forward network

We selected one promising architecture from our Problem 3 experiments:

$$L = 2 \text{ hidden layers}, \qquad W = 3 \text{ hidden units per layer}.$$

This model achieved consistently low validation loss, making it a good candidate for detailed monitoring.

To enable W&B logging, we inserted three short modifications in `feedforwardAssignment.ipynb`:

1. Initialising a run:

```
wandb.init(project="deep-learning-a1",
           name="example_net_baseline")
```

2. Logging metrics inside the training loop:

```
wandb.log({
    "epoch": epoch + 1,
    "train_loss": train_loss,
    "val_loss": val_loss
})
```

3. Finishing the run:

```
wandb.finish()
```

Only these small additions were required because the training function already returns losses per epoch.

Figures 4, 5, and 6 show the logged metrics from W&B. Training and validation loss decrease smoothly, and the validation loss closely tracks the training loss, indicating little to no overfitting.



Figure 4: Epoch counter logged to W&B for the single training run.

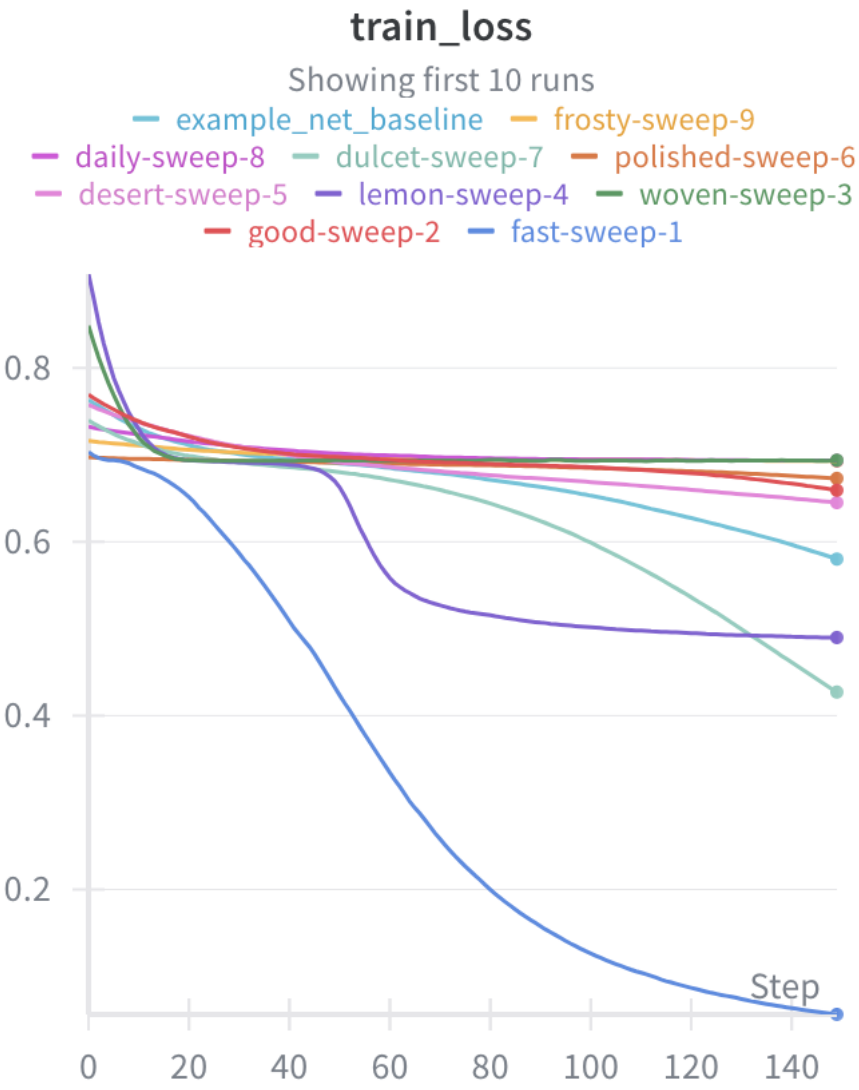


Figure 5: Training loss logged to W&B.

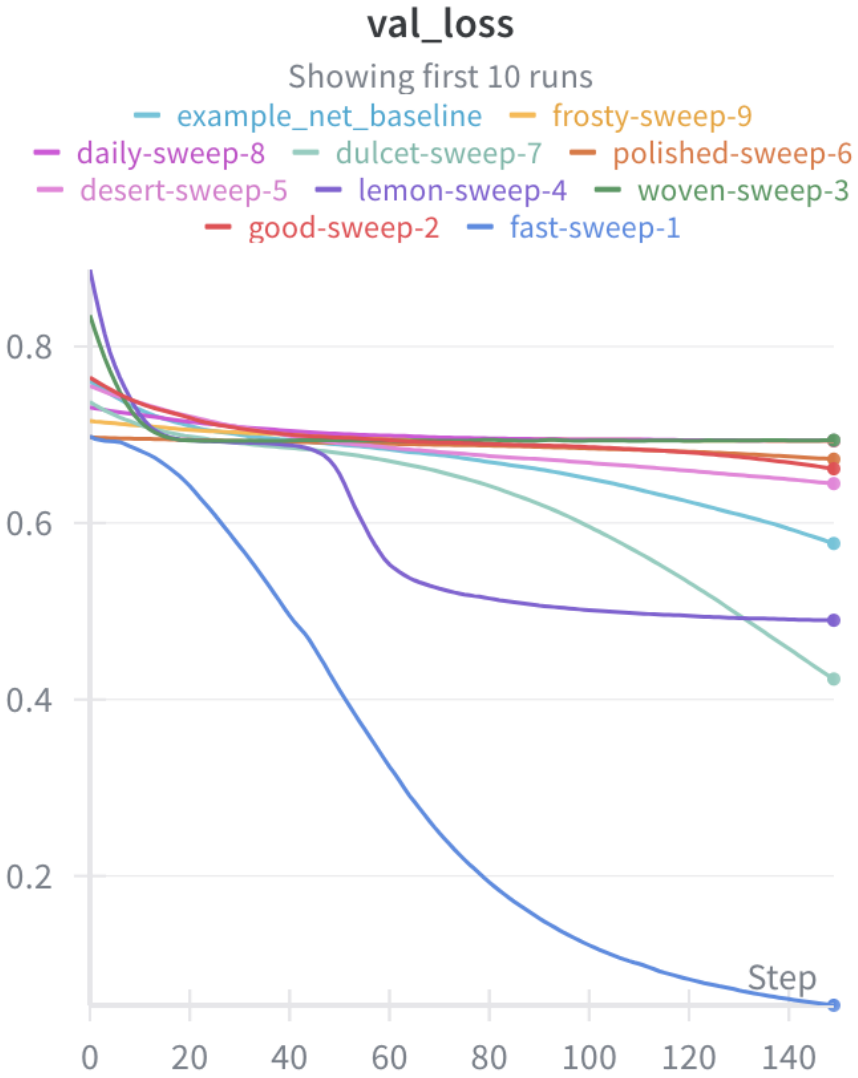


Figure 6: Validation loss logged to W&B.

4.2 Hyperparameter sweep

Next, we implemented a W&B sweep to explore how hyperparameters affect validation performance. We kept the sweep size modest due to compute limits.

We swept over the following parameters:

$$L \in \{1,2,3\}, \qquad W \in \{1,2,3\}, \qquad \eta \in \{10^{-3}, 10^{-2}, 10^{-1}\}.$$

A minimal train function compatible with W&B sweeps was added:

```
def train_sweep(config=None):
    with wandb.init(config=config,
                    project="deep-learning-a1") as run:
        cfg = wandb.config

        train_loader, val_loader = make_loaders()
        net = FeedforwardNet(cfg.run_hidden_layers,
                             cfg.width).to(device)

        history = train_one_model(
            net, train_loader, val_loader,
            epochs=EPOCHS_SWEEP, lr=cfg.lr,
            verbose=False
        )
        wandb.log({"final_val_loss": history["val_loss"][-1]})
```

We used W&B's default parallel-coordinates visualisations and bar charts to analyse the results. Figure 7 shows the final validation losses from the first runs of the sweep.

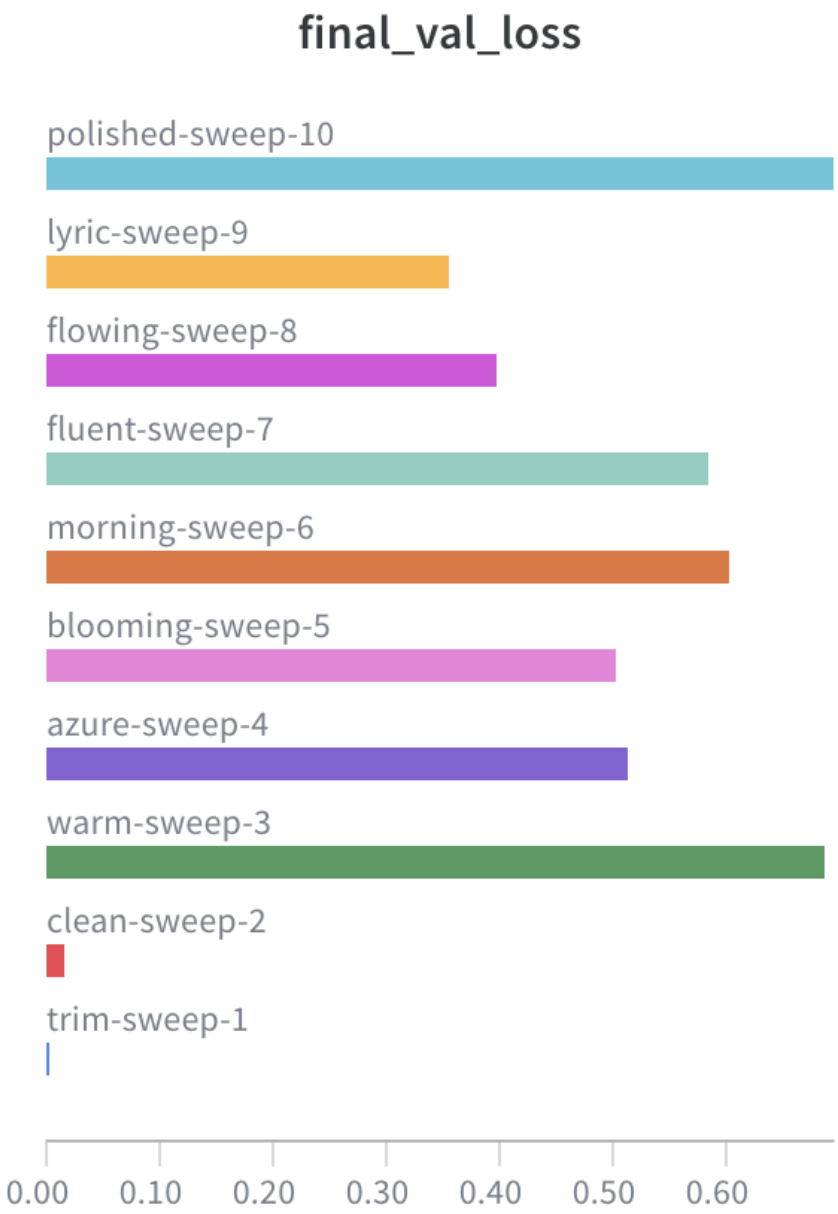


Figure 7: Validation loss of the first sweep runs logged to W&B.

4.3 Experimental findings and preliminary conclusions

Sweep observations. Even with a small sweep, several patterns emerged:

- Architectures with both *moderate depth and moderate width* tended to outperform the smallest networks.
- Very shallow or very narrow networks often failed to reduce the validation loss significantly.
- Learning rate had a substantial impact: runs with too high a learning rate diverged early, visible in increasing or unstable validation curves.

Comparison with Problem 3. These observations are consistent with the controlled sweep in Problem 3. Depth and width both help, but returns diminish beyond $W = 2$ or $L = 2$ for this simple task.

MLOps framework benefits. Using W&B made it straightforward to:

- track loss curves across runs,
- compare architectures side by side,
- visualise the behaviour of unstable or poorly tuned hyperparameters,
- and store results reproducibly.

Conclusion. The MLOps framework provided valuable insight into how sensitive training is to hyperparameter choices. The best-performing networks combined enough depth and width to model the XOR structure while avoiding excessive parameter counts. These results align with the theoretical expectations for non-linear classification tasks such as noisy XOR.