In-Class Exercises: Solutions

Instructor copy – do not distribute before the exercise session.

Data Science with Python – BSc Course

Exercise 1: Forward Propagation by Hand — Solution

Network: 2-2-1 MLP with $W^{(1)} = \begin{pmatrix} 0.5 & -0.3 \\ 0.2 & 0.8 \end{pmatrix}$, $b^{(1)} = (0.1, -0.1)^\top$, $W^{(2)} = (0.7,\ -0.5)$, $b^{(2)} = 0.2$, input $x = (1, 2)^\top$, ReLU hidden, sigmoid output.

(a) Hidden pre-activation $z^{(1)}$

$$z_1^{(1)} = 0.5 \cdot 1 + (-0.3) \cdot 2 + 0.1 = 0.5 - 0.6 + 0.1 = 0.0,$$
$$z_2^{(1)} = 0.2 \cdot 1 + 0.8 \cdot 2 + (-0.1) = 0.2 + 1.6 - 0.1 = 1.7.$$

So $z^{(1)} = (0.0,\ 1.7)^\top$.

(b) Hidden activations $h^{(1)} = \mathrm{ReLU}(z^{(1)})$

$$h_1^{(1)} = \max(0,\ 0.0) = 0.0, \qquad h_2^{(1)} = \max(0,\ 1.7) = 1.7.$$

So $h^{(1)} = (0.0,\ 1.7)^\top$.

(c) Output pre-activation $z^{(2)}$

$$z^{(2)} = 0.7 \cdot 0.0 + (-0.5) \cdot 1.7 + 0.2 = 0 - 0.85 + 0.2 = -0.65.$$

(d) Output $y = \sigma(z^{(2)})$

$$y = \sigma(-0.65) = \frac{1}{1 + e^{0.65}} \approx \frac{1}{1 + 1.9155} \approx \frac{1}{2.9155} \approx 0.343.$$

(e) Predicted class

Because $y \approx 0.343 < 0.5$, the network predicts **class 0**.

Key takeaways

- Neuron 1 is *dead* on this example (pre-activation was exactly 0, ReLU passes through 0). Only neuron 2 contributes.
- The forward pass is a sequence of matrix-vector products followed by elementwise activations — this is exactly what `torch.nn.Linear` + `torch.nn.ReLU` or `tf.keras.layers.Dense` do internally.
- Sigmoid at the output maps any real number into $(0, 1)$ so it can be interpreted as a probability.

Exercise 2: Activation Function Match — Solution

	Scenario	Best activation	Reasoning
A	Deep network, fast training, avoid vanishing gradients	**ReLU**	Non-saturating on the positive side, cheap to compute, and the workhorse of modern deep networks.
B	10-class output, probabilities summing to 1	**softmax**	Normalizes logits to a proper probability distribution over K mutually exclusive classes.
C	Dying ReLUs — neurons stuck at zero	**Leaky ReLU / ELU**	Small negative slope keeps gradient alive for $z < 0$, so "dead" units can recover during training.
D	Binary output in $[0, 1]$	**sigmoid**	Single-output analog of softmax; maps any real number into $(0, 1)$ for probability interpretation.

Bonus: Why is softmax not used in hidden layers?

Softmax is a *competitive* activation — it forces the outputs of one layer to sum to 1. That is exactly what we want for the *final* layer of a multi-class classifier, but it is not what we want internally: hidden layers need to pass rich, multi-dimensional features onward, not a probability distribution over neurons. Also, softmax's gradient involves every neuron in the layer coupled together, which makes backprop more expensive and slower than componentwise activations like ReLU.

Notes for discussion

- tanh is sometimes a valid answer for A/C on shallow networks, but it still saturates and has the vanishing-gradient issue in deep networks.
- GELU (used in Transformers) and Swish are modern alternatives in the same family as Leaky ReLU/ELU.

Exercise 3: Backprop Gradient for One Weight — Solution

Given $x = 1$, $w_1 = 0.5$, $w_2 = 1.5$, $t = 2$.

(a) Forward pass

$$\begin{aligned} z_1 &= w_1 x = 0.5 \cdot 1 = 0.5, \\ h &= \sigma(z_1) = \sigma(0.5) \approx 0.622, \\ y &= w_2 h = 1.5 \cdot 0.622 \approx 0.934, \\ L &= (y - t)^2 = (0.934 - 2)^2 = (-1.066)^2 \approx 1.136. \end{aligned}$$

(b) Local derivatives

- $\dfrac{\partial L}{\partial y} = 2(y - t) = 2 \cdot (-1.066) \approx -2.132.$
- $\dfrac{\partial y}{\partial h} = w_2 = 1.5.$
- $\dfrac{\partial h}{\partial z_1} = h(1 - h) = 0.622 \cdot 0.378 \approx 0.235.$
- $\dfrac{\partial z_1}{\partial w_1} = x = 1.$

(c) Chain rule

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h} \cdot \frac{\partial h}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} \approx (-2.132) \cdot 1.5 \cdot 0.235 \cdot 1 \approx -0.752.$$

(d) Gradient descent update

With $\eta = 0.1$:

$$w_1 \leftarrow w_1 - \eta \cdot \frac{\partial L}{\partial w_1} \approx 0.5 - 0.1 \cdot (-0.752) \approx 0.5 + 0.0752 \approx 0.575.$$

Key takeaways

- The gradient is *negative*, so the update moves w_1 *up*. This makes y larger, pushing it toward the target $t = 2$ — exactly what we want.
- Notice the factor $h(1 - h) = 0.235 < 1$. In a deep network, chaining many such sub-unity factors gives exponentially small gradients: this is the **vanishing gradient problem**, and it is why ReLU (derivative 0 or 1) largely replaced sigmoid/tanh in hidden layers.
- The calculation generalizes: for each weight, backprop is just applying the chain rule from the output back to that weight, reusing intermediate quantities along the way.

Exercise 4: Regularization Match — Solution

	Symptom	Best technique	Reasoning
A	Validation loss rises while training loss still falls	**Early stopping**	Halt training at the validation minimum and restore best weights; cheapest, most effective first-line defence.
B	99.9% train vs. 70% held-out — memorization	**Dropout**	Randomly zeroing activations prevents co-adaptation and forces neurons to build robust features.
C	Exploding weights, unstable network	**L2 (weight decay)**	Adds $\lambda\|\mathbf{w}\|_2^2$ penalty, shrinks weights, limits worst-case sensitivity.
D	Unstable deep training, activation scales drift	**Batch normalization**	Normalizes pre-activations per mini-batch and layer; stabilizes training, often enables higher learning rates.

Bonus: Can you combine techniques?

Yes — and modern practice usually does. A standard deep-learning recipe stacks several complementary regularizers:

- **Early stopping** gives a safe default against overfitting without tuning.
- **L2 weight decay** (typically via the optimizer, e.g. AdamW) keeps weights small.
- **Dropout** on hidden layers breaks co-adaptation.
- **Batch normalization** stabilizes the scale of activations inside the network.
- **Data augmentation** (domain-dependent) expands the effective training set, which is the strongest defence of all.

They address *different* failure modes, so combining them is not redundant: L2 controls weight magnitudes, dropout controls neuron co-dependence, batch norm controls activation scales, and early stopping bounds training time. Empirically, the combination usually outperforms any single technique.

Notes

- L1 (sparsity) is a valid alternative answer for symptom C when you want to zero out many weights; L2 is the safer default.
- Data augmentation is the best answer for symptom B if the dataset is small and images/time-series can be augmented without changing labels.

Exercise 5: Build a Simple MLP — Solution

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier

X, y = make_classification(n_samples=2000, n_features=20,
                           n_informative=10, n_redundant=5,
                           random_state=42)

X_tr, X_va, y_tr, y_va = train_test_split(X, y,
                                          test_size=0.2,
                                          random_state=42)
scaler = StandardScaler().fit(X_tr)
X_tr = scaler.transform(X_tr)
X_va = scaler.transform(X_va)

mlp = MLPClassifier(
    hidden_layer_sizes = (64, 32),      # (1)
    activation         = 'relu',        # (2)
    solver             = 'adam',        # (3)
    alpha              = 1e-4,          # (4)  small L2 regularization
    max_iter           = 100,
    random_state       = 42,
    early_stopping     = True,
    validation_fraction= 0.1,
)

mlp.fit(X_tr, y_tr)
print("Train accuracy:", mlp.score(X_tr, y_tr).round(3))
print("Val   accuracy:", mlp.score(X_va, y_va).round(3))
print("Iterations used:", mlp.n_iter_)
```

Blank answers

Blank	Answer
(1)	`(64, 32)` — two hidden layers, widths 64 and 32
(2)	`'relu'` — sensible default for deep nets
(3)	`'adam'` — adaptive optimizer, sklearn default
(4)	`1e-4` — small L2 regularization strength

(b) Why scale features first?

Gradient-based optimizers take steps proportional to the gradient, and the gradient's magnitude scales with the feature magnitudes. Without scaling, a feature in the thousands dominates the gradient and pulls weight updates in its direction, while small-scale features are effectively ignored. The result is slow, zig-zagging training and poor conditioning. StandardScaler (zero mean, unit variance) makes all features contribute on the same scale so the optimizer can take well-shaped steps.

(c) What does `early_stopping=True` do?

With `early_stopping=True`, sklearn reserves `validation_fraction` of the training set as an internal validation split, checks the validation score each epoch, and *stops* training when it has not improved for a "patience" number of epochs. It then restores the best-seen weights. This corresponds directly to **Task 4** / **worksheet** ("stop when validation loss starts rising"): a cheap, effective regularizer that is almost always worth turning on.

Notes

- Widths of $(100,)$, $(128, 64)$, or similar are also acceptable — there is no unique right answer.
- `solver='lbfgs'` can be faster for very small datasets (under a few hundred samples); `'adam'` is the right default here.
- `alpha` is the L2 coefficient in sklearn's MLP. A value between `1e-5` and `1e-3` is typical.

Exercise 6: Dropout Effect — Solution

(a) Train vs. validation gap

With the noisy data generated by `flip_y=0.10` and the overparameterised 30-128-64-1 network, the **dropout = 0.0** model typically reaches train accuracy ≈ 0.99 but validation accuracy ≈ 0.80 — a gap of nearly 20 percentage points, the fingerprint of overfitting.

The **dropout = 0.3** model usually has lower training accuracy (≈ 0.90) but *higher* validation accuracy (≈ 0.83 – 0.85), a much smaller gap, and less volatile epoch-to-epoch behaviour. Exact numbers depend on the random seed but the qualitative picture is robust.

(b) Validation loss curves

Typical behaviour:

- The `dropout = 0.0` validation loss drops for the first 10-20 epochs, reaches a minimum around epoch 15–25, and then *rises* steadily for the remaining epochs — the model is memorizing training noise.
- The `dropout = 0.3` validation loss drops more slowly but keeps decreasing (or plateaus) throughout training; the curve is smoother and does not exhibit the late-epoch rise.

This is the same overfitting pattern as Network B in worksheet Task 4 — and the same fix (stop early, or regularize) applies.

(c) What if dropout is too high?

Increasing dropout to 0.7 typically *hurts* both training and validation accuracy:

- Each forward pass only uses 30% of each hidden layer, so the effective model capacity is very small.
- The network struggles to fit even the training set; both training and validation loss stay high.
- Training is slower because the effective gradient signal is diluted by the random zeroing.

There is a **sweet spot**: common choices are dropout rates in the range 0.1–0.5, with 0.5 being Srivastava's original default for fully-connected hidden layers. Treat dropout rate as a hyperparameter to tune on validation data.

Notes

- At *evaluation* time Keras automatically disables dropout and scales activations so that the expected output matches the training-time expectation.
- Dropout can be combined with L2 and early stopping. In `keras.Sequential`, add `kernel_regularizer=keras.regularizers.l2(1e-4)` to each Dense layer and pass `callbacks=[keras.callbacks.EarlyStoppi restore_best_weights=True)]` to `fit`.
- For very deep networks, **batch normalization** is usually more important than dropout; the community has moved somewhat away from high-rate dropout in CNNs and Transformers.