# Assignment 3 — Solutions

## Neural Network Fundamentals

Part A — Foundations · 26 questions with answers

### Q1. How does a neural network work? Explain with a single neuron first. ★

**Answer**

**Single neuron:** takes inputs $x_1 \ldots x_n$, multiplies each by a weight, adds a bias, and passes the result through an activation function:

$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$, output $a = f(z)$

**Network:** neurons are arranged in layers – input layer → hidden layers → output layer. Each layer's outputs are the next layer's inputs.

**Learning:**

1. **Forward pass** – compute predictions.
2. **Loss** – compare predictions with true labels.
3. **Backpropagation** – compute the gradient of the loss with respect to every weight (chain rule).
4. **Update** – an optimizer adjusts weights to reduce the loss.
5. Repeat over many batches and epochs until the loss stops decreasing.

Non-linear activations let the network learn complex, non-linear patterns.

### Q2. What is the weight update formula in a neural network?

**Answer**

**Gradient descent update rule:**

$w_new = w_old - \eta \times \partial L/\partial w$

$b_new = b_old - \eta \times \partial L/\partial b$

where $\eta$ is the learning rate and $\partial L/\partial w$ is the gradient of the loss with respect to the weight. We move **opposite to the gradient** because the gradient points in the direction of steepest increase of the loss.

### Q3. Define: epoch, sample, batch size, iteration, learning rate, weights, biases.

**Answer**

- **Epoch** – one complete pass of the entire training dataset through the network.
- **Sample** – a single data point (one row / one sentence / one image).
- **Batch size** – number of samples processed before one weight update (e.g. 32).
- **Iteration (step)** – one weight update = one batch. Iterations per epoch = dataset size / batch size. (1,000 samples, batch 100 → 10 iterations per epoch.)
- **Learning rate** – step size used when updating weights.
- **Weights** – learnable parameters that scale each input; they store what the network has learned.
- **Biases** – learnable offsets added to the weighted sum; let a neuron activate even when all inputs are zero (shifts the activation function).

## Q4. What is the difference between a loss function and a cost function?

### Answer

- **Loss function** – error for a **single** training example, e.g. $(y - \hat{y})^2$.
- **Cost function** – **average** (or sum) of the losses over the whole batch/dataset, possibly plus regularization terms, e.g. MSE = $(1/n)\ \Sigma\ (y_i - \hat{y}_i)^2$.

The optimizer minimizes the cost function. In practice the two words are often used interchangeably.

## Q5. Explain the learning rate. What happens if it is too high or too low? ★

### Answer

The **learning rate (η)** controls how big a step the optimizer takes in the direction of the negative gradient.

- **Too high:** the updates overshoot the minimum, the loss oscillates or **diverges** (becomes NaN).
- **Too low:** training is very **slow**, may get stuck in a poor local minimum/plateau, and needs many epochs.
- **Good practice:** start with typical values (Adam: 1e-3; fine-tuning Transformers: 1e-5 to 5e-5; LoRA: 1e-4 to 2e-4), use a **learning-rate finder**, **warm-up** and **schedulers** (decay).

## Q6. Explain forward propagation and backpropagation. ★

### Answer

**Forward propagation:** input data passes layer by layer. For each layer, $z = W \cdot x + b$ and $a = f(z)$. The last layer gives the prediction $\hat{y}$, and the loss $L(y, \hat{y})$ is computed.

**Backpropagation:** computes the gradient of the loss with respect to every weight using the **chain rule**, starting from the output layer and moving backward:

$\partial L/\partial w = \partial L/\partial a \times \partial a/\partial z \times \partial z/\partial w$

Each layer reuses the gradients already computed for the layer after it, which makes it efficient. The optimizer then uses these gradients to update weights. Frameworks (PyTorch autograd, TensorFlow GradientTape) do this automatically.

## Q7. What is data normalization/standardization and why is it needed for neural networks?

### Answer

**Normalization (min-max scaling):** $x' = (x - \min) / (\max - \min) \rightarrow$ range [0, 1].

**Standardization (z-score):** $x' = (x - \mu) / \sigma \rightarrow$ mean 0, std 1.

**Why needed:**

- Features on very different scales (age 0–100, salary 0–10,00,000) cause large-scale features to dominate gradients.
- Gradient descent converges much **faster and more stably** when features are on similar scales (the loss surface is less elongated).
- Avoids saturating activation functions (sigmoid/tanh) with very large inputs.

Important: **fit the scaler on training data only**, then transform validation/test data with it (avoid data leakage).

### Q8. What are the different types of optimizers (SGD, Momentum, RMSProp, Adam, AdamW)? ★

**Answer**

- **SGD** – basic gradient descent on mini-batches.
- **SGD with Momentum** – adds a velocity term to accelerate and smooth updates.
- **Nesterov Accelerated Gradient** – momentum that "looks ahead".
- **AdaGrad** – per-parameter learning rate, decays with accumulated squared gradients (good for sparse data, but LR shrinks too much).
- **RMSProp** – fixes AdaGrad using an exponential moving average of squared gradients.
- **Adam** – Momentum + RMSProp (first and second moment estimates with bias correction). Default choice for most problems.
- **AdamW** – Adam with **decoupled weight decay**; standard for Transformers/LLMs.
- Others: Adadelta, Nadam, Lion, Adafactor (memory-efficient for large models).

### Q9. Explain Batch Gradient Descent, Stochastic Gradient Descent and Mini-batch Gradient Descent.

**Answer**

- **Batch Gradient Descent** – uses the **entire dataset** to compute one gradient and make one update per epoch. Stable, accurate gradient, but slow and memory-heavy for large data.
- **Stochastic Gradient Descent (SGD)** – updates weights after **every single sample**. Very fast updates and can escape local minima, but the loss is very noisy and it does not use hardware parallelism well.
- **Mini-batch Gradient Descent** – updates after a **small batch** (e.g. 32–512 samples). Best of both: efficient on GPUs and reasonably stable. This is what is used in practice (and frameworks usually call it "SGD").

### Q10. Explain momentum and learning-rate decay / schedulers (warm-up, cosine decay).

**Answer**

**Momentum:** keeps a moving average of past gradients (velocity) and moves in that direction:

$v = \beta \cdot v + \eta \cdot \nabla L$; $w = w - v$ ($\beta \approx 0.9$)

Like a ball rolling downhill – it speeds up in consistent directions, dampens oscillations and helps pass through small local minima and plateaus.

**Learning-rate decay / schedulers:** gradually reduce the learning rate during training – big steps early for fast progress, small steps later for fine convergence. Types: step decay, exponential decay, ReduceLROnPlateau, **cosine decay**.

**Warm-up:** start with a very small LR and increase it linearly for the first few hundred/thousand steps; prevents unstable updates early in Transformer training. "Linear warm-up + cosine decay" is the standard LLM schedule.

### Q11. Which optimizer is most commonly used today, and why is AdamW preferred for Transformers?

**Answer**

**Adam** is the most commonly used optimizer – it adapts the learning rate per parameter, converges fast and needs little tuning.

**AdamW is preferred for Transformers** because in original Adam, L2 regularization is added to the gradient and then gets scaled by the adaptive learning rate, so weight decay becomes inconsistent. AdamW **decouples weight decay** from the gradient update ($w = w - \eta \cdot (\text{adam_update} + \lambda \cdot w)$), giving proper regularization and better generalization. It is the default in Hugging Face Trainer.

### Q12. What is the difference between parameters and hyperparameters? Name important hyperparameters.

**Answer**

- **Parameters** – learned **by the model** during training: weights and biases.
- **Hyperparameters** – set **by us** before training and control the learning process.

**Important hyperparameters:** learning rate, batch size, number of epochs, optimizer type, number of layers, neurons per layer, activation functions, dropout rate, weight decay (L2), learning-rate schedule, weight initialization, early-stopping patience. For LLMs: context length, LoRA rank, temperature (at inference).

### Q13. What is the use of an activation function? What happens without one? ★

**Answer**

An activation function adds **non-linearity**, so the network can learn complex patterns. It also controls the output range (e.g. probabilities with sigmoid/softmax).

**Without activation functions**, every layer is a linear transformation, and a stack of linear layers collapses into **one linear layer** – a 100-layer network would be no more powerful than linear/logistic regression and could not learn non-linear patterns like XOR.

### Q14. Explain the activation functions: Sigmoid, Tanh, ReLU, Leaky ReLU, GELU, Softmax. Which is used where?

**Answer**

- **Sigmoid:** $1/(1+e^{-x})$, range (0, 1). Used in the **output layer for binary / multi-label classification**. Problems: vanishing gradient, not zero-centred.
- **Tanh:** range $(-1, 1)$, zero-centred. Used in RNN/LSTM hidden states. Still saturates.
- **ReLU:** $\max(0, x)$. Fast, no saturation for $x > 0$. **Default for hidden layers** in CNNs/MLPs. Problem: "dying ReLU" (neurons stuck at 0).
- **Leaky ReLU:** $\max(0.01x, x)$. Fixes dying ReLU.
- **GELU:** $x \cdot \Phi(x)$, a smooth ReLU. **Used in Transformers (BERT, GPT).** (SwiGLU is used in Llama-style models.)
- **Softmax:** converts logits into a probability distribution summing to 1. Used in the **output layer for multi-class classification** and inside attention.
- **Linear (no activation):** output layer for **regression**.

### Q15. Why are Sigmoid/Tanh not preferred in hidden layers? Which activation is most common in hidden layers?

**Answer**

- Sigmoid and Tanh **saturate**: for large |x| their gradient is almost 0. In deep networks, multiplying many small gradients causes **vanishing gradients**, so early layers stop learning.
- The maximum derivative of sigmoid is only **0.25**, so gradients shrink at every layer.
- Sigmoid outputs are **not zero-centred** (always positive), which makes gradient updates zig-zag.
- They require computing exponentials (slower).

**Most commonly used in hidden layers: ReLU** (and its variants Leaky ReLU/GELU; GELU/SwiGLU in Transformers).

### Q16. Which loss function should be used for regression, binary classification, multi-class and multi-label problems? ★

**Answer**

- **Regression: MSE** (penalizes large errors), **MAE** (robust to outliers), **Huber** (mix of both). Output: linear.
- **Binary classification: Binary Cross-Entropy** (BCE). Output: 1 neuron + sigmoid.
- **Multi-class (one label per sample): Categorical Cross-Entropy** (one-hot labels) or **Sparse Categorical Cross-Entropy** (integer labels). Output: N neurons + softmax.
- **Multi-label (several labels per sample): Binary Cross-Entropy per class**. Output: N neurons + sigmoid.
- Others: **Hinge** loss (SVM-style), **KL divergence** (distillation), **contrastive / triplet loss** (embeddings), **Focal loss** (heavy class imbalance, e.g. fraud detection).
- LLM pre-training = cross-entropy on next-token prediction.

### Q17. What are vanishing and exploding gradients? Why do they occur and how do you fix them? ★

**Answer**

- **Vanishing gradients:** gradients become extremely small as they are propagated back through many layers (or time steps), so early layers learn very slowly or not at all.
- **Exploding gradients:** gradients become extremely large, causing huge weight updates, unstable training and NaN loss.

**Why they occur:** backprop multiplies many derivatives together (chain rule). If they are $< 1$ (sigmoid/tanh derivatives, small weights) the product shrinks exponentially; if $> 1$ (large weights) it grows exponentially. Very deep networks and RNNs over long sequences are most affected.

**Fixes:**

- ReLU-family activations instead of sigmoid/tanh.
- Proper weight initialization (He/Xavier).
- Batch Normalization / Layer Normalization.
- Residual (skip) connections (ResNet, Transformers).
- LSTM/GRU instead of vanilla RNN.
- **Gradient clipping** (for exploding gradients).
- Lower learning rate.

## Q18. What are the weight initialization methods (Xavier/Glorot, He)? Why does initialization matter?

### Answer

- **Zero initialization** – bad: all neurons compute the same thing (symmetry is never broken).
- **Random normal/uniform (small values)** – breaks symmetry but may cause vanishing/exploding gradients in deep nets.
- **Xavier / Glorot** – variance = 2 / (fan_in + fan_out). Best for **sigmoid/tanh**.
- **He / Kaiming** – variance = 2 / fan_in. Best for **ReLU** family.
- **LeCun** – variance = 1 / fan_in. Used with SELU.
- **Orthogonal** – often for RNNs.

**Why it matters:** good initialization keeps the variance of activations and gradients roughly constant across layers, so the signal neither dies nor explodes and training converges faster.

## Q19. What is overfitting? List regularization methods (dropout, L1/L2, early stopping, data augmentation).

### Answer

**Overfitting:** the model learns the training data (including noise) too well – high training accuracy, poor validation/test accuracy.

**Regularization methods:**

- **Dropout** – randomly turn off a fraction of neurons during training.
- **L1 / L2 regularization (weight decay)** – penalize large weights.
- **Early stopping** – stop when validation loss stops improving.
- **Data augmentation** – more varied training data (back-translation, synonym replacement for text).
- **More data**, simpler model (fewer layers/neurons).
- **Batch normalization** (mild regularizing effect), label smoothing, cross-validation.

## Q20. What is Batch Normalization? How does it differ from Layer Normalization?

### Answer

**Batch Normalization** normalizes each feature **across the samples in a batch** (mean/var over the batch dimension), then scales/shifts with learnable $\gamma$, $\beta$. Speeds up training and allows higher learning rates. Depends on batch size, behaves differently at train vs inference (uses running statistics). Common in CNNs.

**Layer Normalization** normalizes **across the features of a single sample** (each token vector separately). Independent of batch size, works the same during training and inference, and handles variable-length sequences. This is why **Transformers and RNNs use LayerNorm** (LLMs often use its cheaper variant RMSNorm).

## Q21. How would you change a pre-trained classification network into a regression network?

### Answer

1. Keep the pre-trained base layers (feature extractor).
2. **Replace the output layer:** remove the softmax/sigmoid classification head and add a **Dense layer with 1 neuron (or k neurons for k targets) and linear activation**.
3. **Change the loss function** from cross-entropy to **MSE/MAE/Huber**.
4. Change metrics to MAE/RMSE/$R^2$.

5. Freeze the base layers at first, train the new head, then optionally unfreeze and fine-tune with a small learning rate. Scale the target variable if needed.

### Q22. If the previous layer has 8 neurons and the current Dense layer has 512 neurons, how many parameters does the current layer have? (Show the calculation.) ★

**Answer**

For a Dense (fully connected) layer: **parameters = (inputs × neurons) + biases**

= (8 × 512) + 512 = 4,096 + 512 = **4,608 parameters**.

Each of the 512 neurons has 8 weights (one per input) plus 1 bias.

### Q23. What are the steps to build an ANN using TensorFlow/Keras? What is the Sequential API?

**Answer**

**Steps:**

1. Load and preprocess data (clean, encode categories, split train/val/test, scale features).
2. Define the model architecture (`Sequential` or Functional API).
3. Compile – choose optimizer, loss and metrics.
4. Train with `model.fit()` (epochs, batch size, validation data, callbacks such as EarlyStopping).
5. Evaluate with `model.evaluate()` on the test set.
6. Predict with `model.predict()`; tune hyperparameters; save with `model.save()`.

**Sequential API:** a linear stack of layers where each layer has exactly one input and one output tensor – the simplest way to build a model. For multiple inputs/outputs or skip connections, use the Functional API or model subclassing.

### Q24. Scale data for a neural network (StandardScaler / MinMaxScaler) — write the code. [CODE]

**Answer**

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()                 # or MinMaxScaler()
X_train_s = scaler.fit_transform(X_train) # fit ONLY on training data
X_test_s  = scaler.transform(X_test)      # reuse the same mean/std

# Keras alternative: a Normalization layer inside the model
import tensorflow as tf
norm = tf.keras.layers.Normalization()
norm.adapt(X_train)
```

## Q25. Define a model, add layers, compile it and train it (Keras or PyTorch). [CODE]

### Answer

```
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Input(shape=(X_train_s.shape[1],)),
    layers.Dense(64, activation="relu"),
    layers.Dropout(0.3),
    layers.Dense(32, activation="relu"),
    layers.Dense(1, activation="sigmoid")      # binary classification
])
# Alternative: model = models.Sequential(); model.add(layers.Dense(64, activation="relu")) ...

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss="binary_crossentropy", metrics=["accuracy"])
model.summary()

early = tf.keras.callbacks.EarlyStopping(patience=5, restore_best_weights=True)
history = model.fit(X_train_s, y_train, validation_split=0.2,
                    epochs=50, batch_size=32, callbacks=[early])
print(model.evaluate(X_test_s, y_test))
```

## Q26. Set a custom weight initializer for a layer in TensorFlow. [CODE]

### Answer

```
import tensorflow as tf
from tensorflow.keras import layers, initializers

layer = layers.Dense(
    64, activation="relu",
    kernel_initializer=initializers.HeNormal(),     # weights
    bias_initializer=initializers.Zeros())           # biases

# Shortcut with strings
layers.Dense(64, activation="tanh", kernel_initializer="glorot_uniform")

# Custom constant / random initializers
layers.Dense(10, kernel_initializer=initializers.RandomNormal(mean=0.0, stddev=0.05))

# Set weights manually after building
# layer.set_weights([W_numpy, b_numpy])
```