**1. Logistic Regression vs. Perceptron**

* **Why is Logistic Regression preferable to a classical Perceptron?**
  + The Perceptron algorithm only works if the classes are linearly separable; it fails on non-linearly separable data.
  + Logistic Regression provides probabilistic outputs (values between 0 and 1), making it useful for tasks requiring confidence scores.
  + Logistic Regression is trained using gradient-based optimization (e.g., gradient descent), whereas the Perceptron uses an update rule that doesn’t work well for noisy data.
* **How to tweak a Perceptron to make it equivalent to Logistic Regression?**
  + Use the sigmoid activation function instead of a step function.
  + Train it using gradient descent with a loss function like binary cross-entropy instead of the Perceptron learning rule.

**2. Importance of the Logistic Activation Function in Early MLPs**

* The logistic activation function (sigmoid) was important in early Multilayer Perceptrons (MLPs) because:
  + It introduced non-linearity, allowing networks to approximate complex functions.
  + It allowed for smooth gradients, enabling gradient-based optimization (e.g., backpropagation).
  + The sigmoid’s derivative is easy to compute, which helped in early implementations.

**3. Three Popular Activation Functions and Their Graphs**

1. **Sigmoid (Logistic function)**

σ(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}σ(x)=1+e−x1​

* + Output range: (0,1)
  + Used in binary classification**.**

1. **ReLU (Rectified Linear Unit)**

f(x)=max⁡(0,x)f(x) = \max(0, x)f(x)=max(0,x)

* + Output range: (0, ∞)
  + Helps prevent vanishing gradients.

1. **Tanh (Hyperbolic Tangent)**

tanh⁡(x)=ex−e−xex+e−x\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}tanh(x)=ex+e−xex−e−x​

* + Output range: (-1,1)
  + Centered around zero, better for hidden layers than sigmoid.

**4. MLP Structure and Matrix Shapes**

**Given:**

* Input layer: 10 neurons
* Hidden layer: 50 neurons (ReLU)
* Output layer: 3 neurons (ReLU)

**Matrix Shapes**

* Input matrix XXX: (m,10)(m, 10)(m,10), where mmm is the batch size.
* Hidden layer weights WhW\_hWh​: (10,50)(10, 50)(10,50)
* Hidden layer bias bhb\_hbh​: (1,50)(1, 50)(1,50)
* Output layer weights WoW\_oWo​: (50,3)(50, 3)(50,3)
* Output layer bias bob\_obo​: (1,3)(1, 3)(1,3)
* Output matrix YYY: (m,3)(m, 3)(m,3)

Equation for network output:

H=ReLU(XWh+bh)H = \text{ReLU}(X W\_h + b\_h)H=ReLU(XWh​+bh​) Y=ReLU(HWo+bo)Y = \text{ReLU}(H W\_o + b\_o)Y=ReLU(HWo​+bo​)

**5. Output Layer Configuration for Different Tasks**

* **Spam vs. Ham Classification (Binary Classification)**
  + Output neurons: 1
  + Activation function: Sigmoid
  + Loss function: Binary cross-entropy
* **MNIST Classification (10-digit classification)**
  + Output neurons: 10
  + Activation function: Softmax
  + Loss function: Categorical cross-entropy

**6. Backpropagation vs. Reverse-Mode Autodiff**

* Backpropagation:
  + Algorithm for computing gradients in neural networks.
  + Uses the chain rule to propagate errors backward.
  + Updates weights using gradient descent.
* Reverse-Mode Autodiff:
  + More general method for computing derivatives.
  + Used in modern deep learning frameworks (e.g., TensorFlow, PyTorch).
  + Backpropagation is a special case of reverse-mode autodiff.

**7. MLP Hyperparameters and Overfitting Solutions**

Hyperparameters:

* Network Architecture: Number of layers, neurons per layer.
* Activation Functions: ReLU, sigmoid, etc.
* Learning Rate: Controls how fast weights are updated.
* Batch Size: Number of samples per training step.
* Optimizer: Adam, SGD, RMSprop, etc.
* Regularization: Dropout, L2 regularization.
* Epochs: Number of training iterations.

To prevent overfitting:

* Use regularization (L2, dropout).
* Reduce network complexity (fewer neurons/layers).
* Early stopping (monitor validation loss).
* Use more training data (data augmentation).

**8. Train a Deep MLP on MNIST with Over 98% Accuracy**

**Answer:**

Here’s a TensorFlow/Keras implementation:

import tensorflow as tf

from tensorflow import keras

# Load MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = keras.datasets.mnist.load\_data()

# Normalize pixel values

X\_train, X\_test = X\_train / 255.0, X\_test / 255.0

# Flatten input images

X\_train = X\_train.reshape(-1, 28 \* 28)

X\_test = X\_test.reshape(-1, 28 \* 28)

# Define model

model = keras.models.Sequential([

keras.layers.Dense(128, activation="relu", input\_shape=(28\*28,)),

keras.layers.Dropout(0.2),

keras.layers.Dense(64, activation="relu"),

keras.layers.Dropout(0.2),

keras.layers.Dense(10, activation="softmax")

])

# Compile model

model.compile(optimizer="adam",

loss="sparse\_categorical\_crossentropy",

metrics=["accuracy"])

# Train model with checkpointing

checkpoint\_cb = keras.callbacks.ModelCheckpoint("mnist\_model.h5", save\_best\_only=True)

early\_stopping\_cb = keras.callbacks.EarlyStopping(patience=5, restore\_best\_weights=True)

history = model.fit(X\_train, y\_train, epochs=30, validation\_data=(X\_test, y\_test),

callbacks=[checkpoint\_cb, early\_stopping\_cb])

# Evaluate on test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print(f"Test accuracy: {test\_acc:.4f}")