Assignment Code: DS-AG-019

Neural Network - A Simple Perceptron | Assignment

Instructions: Carefully read each question. Use Google Docs, Microsoft Word, or a similar tool to create a document where you type out each question along with its answer. Save the document as a PDF, and then upload it to the LMS. Please do not zip or archive the files before uploading them. Each question carries 20 marks.

Total Marks: 200

Question 1: What is Deep Learning? Briefly describe how it evolved and how it differs from traditional machine learning.

Answer:

- **Definition:** Deep Learning (DL) is a subfield of machine learning that uses artificial neural networks with many layers ("deep" networks) to learn hierarchical feature representations directly from data. DL models automatically extract features and learn complex mappings from inputs to outputs, enabling state-of-the-art performance in areas such as computer vision, natural language processing, speech recognition, and reinforcement learning.
- **Evolution (brief timeline):**
  - **1940s–1960s:** Early models — perceptron (Rosenblatt, 1958) and theoretical foundations of neural computation.
  - **1980s:** Backpropagation and multi-layer perceptron's (Rumelhart, Hinton, Williams) made training multi-layer networks practical.
  - **1990s–2000s:** SVMs and other classical ML methods dominated; neural networks faced optimization and data/compute limits.
  - **2006:** The term "deep learning" gained traction — unsupervised pertaining and deep belief nets (Hinton et al.) showed deeper networks were trainable.
  - **2010s:** Breakthroughs from large labelled datasets (ImageNet), GPUs, improved architectures (CNNs, RNNs), and regularization/optimization techniques (dropout, Adam) led to rapid progress.
  - **2020s:** Transformers and scaling laws produced dramatic gains in language and multimodal models.

➢ **How DL differs from traditional ML:**

- **Feature engineering:** Traditional ML often relies on handcrafted features; DL learns features automatically from raw data.
- **Model complexity:** DL models typically have far more parameters and layers, enabling representation of more complex functions.
- **Data requirements:** DL usually requires much larger datasets to reach best performance; classical ML can do well with smaller datasets.
- **Computation:** DL training is computationally intensive (GPUs/TPUs), whereas many classical algorithms are lighter.
- **End-to-end learning:** DL excels at end-to-end pipelines (e.g., image pixels -> labels) without intermediate manual steps.

Question 2: Explain the basic architecture and functioning of a Perceptron. What are its limitations?

Answer:

Architecture:

- A perceptron is the simplest type of artificial neuron used for binary classification. It takes a feature vector $x=[x_1,x_2,...,x_n]$, applies weights $w=[w_1,w_2,...,w_n]$ and a bias b, computes a weighted sum (net input), and passes it through a step activation function.

Mathematically:

$$\mathbb{w}.\mathbb{x} + \mathbb{b} = \sum_{i=0}^{n} \boldsymbol{w_i}\ \boldsymbol{x_i} + \boldsymbol{b}$$

Training (Perceptron learning rule):

- Initialize weights (often small random values or zeros).
- For each training example (x,y), compute prediction $\hat{y}$.
- Update weights if prediction is wrong: $w \leftarrow w+\eta(y-\hat{y})x$, bias updated similarly; where η is the learning rate.
- Repeat until convergence or max epochs.

Limitations:

1. **Linearly separable only:** Perceptron can only learn datasets that are linearly separable (e.g., cannot learn XOR).
2. **Binary output / step activation:** The hard step activation gives non-differentiable output making gradient-based training of deeper networks impossible unless replaced with differentiable activations.
3. **Limited representational power:** Single-layer perceptron cannot capture complex functions; needs multi-layer networks (MLPs) for non-linear decision boundaries.
4. **No probabilistic outputs:** Outputs are 0/1 instead of probabilities (unless modified with logistic/sigmoid output).

Question 3: Describe the purpose of activation function in neural networks. Compare Sigmoid, ReLU, and Tanh functions.

Answer:

Purpose of activation functions:

- Introduce non-linearity so networks can model complex, non-linear relationships. Without them, stacked layers' collapse into a single linear transformation. Activation functions also control gradient flow during training and influence convergence behaviour.

Comparisons:

- **Sigmoid:**
  - Formula: $\sigma(x)= \frac{1}{1+e^{-x}}$.
  - Range: (0, 1).
  - Pros: Historically used, gives output interpretable as probability for binary outputs.
  - Cons: Vanishing gradients for large |x| (saturates), not zero-centred, slower convergence.
- **Tanh (hyperbolic tangent):**
  - Formula: $\tanh(x)= \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
  - Range: (-1, 1).
  - Pros: Zero-centred which often helps optimization; stronger gradients near zero compared to sigmoid.
  - Cons: Still suffers from saturation/vanishing gradients for large |x|.

- **ReLU (Rectified Linear Unit):**
  - Formula: ReLU(x)=max(0,x).
  - Range: [0, \infinity].
  - Pros: Simple, computationally efficient, mitigates vanishing gradient for positive side, often leads to faster convergence and sparser activations.
  - Cons: **Dying ReLU** — neurons can become inactive (output zero) for all inputs if weights push them to negative region; not zero-centred; unbounded positive side.

Question 4: What is the difference between Loss function and Cost function in neural networks? Provide examples.

Answer:

- **Loss function (per-example loss):** Measures error for a single training example. Denoted as L(y, ŷ) where y is true label and y^ is prediction. Examples:
  - **Mean Squared Error (MSE)** for regression: $L=(y-\hat{y})^2$.
  - **Binary Cross-Entropy (Log Loss)** for binary classification: $L=-[y\log(\hat{y})+(1-y)\log(1-\hat{y})]$.
  - **Categorical Cross-Entropy** for multi-class classification.

- **Cost function:** Aggregate of the loss over the training set (often average or sum). It is the quantity optimized by training algorithms.
  - Example: For dataset of N examples, cost = $J(\theta)=\frac{1}{N}\sum_{i=1}^{n} L^{(i)}$. Cost can include regularization terms, e.g., weight decay: $J(\theta)=\frac{1}{N}\sum L^{(i)} + \lambda||\theta||^2$.

Question 5: What is the role of optimizers in neural networks? Compare Gradient Descent, Adam, and RMSprop.

Answer:

Role of optimizers:

- Optimizers update model parameters (weights and biases) to minimize the cost function by following gradients. They determine step sizes, adapt learning rates, and incorporate momentum or second-order information to improve convergence.

Comparison:

- **Gradient Descent (GD):**
  - **Batch GD:** computes gradient over entire dataset each step — stable but slow and memory-heavy for large datasets.
  - **Stochastic GD (SGD):** updates per example — noisy updates, can escape shallow minima.
  - **Mini-batch SGD:** compromise; most commonly used.
  - **Learning rate:** fixed or scheduled. Simple but may require careful tuning; can be slow to converge and get stuck.
- **RMSprop:**
  - Keeps a moving average of squared gradients to normalize the parameter updates, effectively adapting the learning rate per parameter.
  - Update rule scales learning rate by $1/\sqrt{E[g^2]}$.
  - Works well for non-stationary problems and usually converges faster than vanilla SGD.
- **Adam (Adaptive Moment Estimation):**
  - Combines ideas from RMSprop (adaptive learning rates) and momentum (moving average of gradients).
  - Keeps estimates of first moment (mean of gradients) and second moment (uncentered variance) with bias-correction terms.
  - Often provides fast convergence and performs well out-of-the-box. Common default choices: learning rate=0.001, β1=0.9, β2=0.999.

- **Use NumPy, Matplotlib, and Tensorflow/Keras for implementation.**

Question 6: Write a Python program to implement a single-layer perceptron from scratch using NumPy to solve the logical AND gate.

(*Include your Python code and output in the code box below.*)
Answer:

Code:

```
import numpy as np

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=10):
        # initialize weights and bias
        self.weights = np.zeros(input_size)
        self.bias = 0
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation(self, x):
        # step activation function
        return 1 if x >= 0 else 0

    def predict(self, x):
        # linear combination + activation
        z = np.dot(x, self.weights) + self.bias
        return self.activation(z)

    def fit(self, X, y):
        # training process
        for epoch in range(self.epochs):
            total_error = 0
            for inputs, label in zip(X, y):
                prediction = self.predict(inputs)
                error = label - prediction
                # update weights and bias
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error
                total_error += abs(error)
            print(f"Epoch {epoch+1}: total errors = {total_error}")
            if total_error == 0:
                break


# Input and output data for AND gate
# Truth table: (x1, x2) → y
# (0,0)=0, (0,1)=0, (1,0)=0, (1,1)=1
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
```

```
          [1, 1]])
y = np.array([0, 0, 0, 1])

# Create perceptron object and train
p = Perceptron(input_size=2, learning_rate=0.2, epochs=10)
p.fit(X, y)

# Display final weights and bias
print("Final Weights:", p.weights)
print("Final Bias:", p.bias)

# Test predictions
print("\nPredictions for AND gate:")
for i in range(len(X)):
    print(f"{X[i]} -> {p.predict(X[i])}")
```

Output:

```
Epoch 1: total errors = 2
Epoch 2: total errors = 3
Epoch 3: total errors = 3
Epoch 4: total errors = 0
Final Weights: [0.4 0.2]
Final Bias: -0.4000000000000001

Predictions for AND gate:
[0 0] -> 0
[0 1] -> 0
[1 0] -> 0
[1 1] -> 1
```

Question 7: Implement and visualize Sigmoid, ReLU, and Tanh activation functions using Matplotlib.

(*Include your Python code and output in the code box below.*)

Answer:

Code:

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
```

```
def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

x = np.linspace(-6, 6, 400)
y_sig = sigmoid(x)
y_tanh = tanh(x)
y_relu = relu(x)

plt.figure(figsize=(8,5))
plt.plot(x, y_sig, label='Sigmoid')
plt.plot(x, y_tanh, label='Tanh')
plt.plot(x, y_relu, label='ReLU')
plt.axhline(0, color='gray', linewidth=0.5)
plt.axvline(0, color='gray', linewidth=0.5)
plt.title('Activation functions: Sigmoid, Tanh, ReLU')
plt.xlabel('x')
plt.ylabel('activation(x)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Question 8: Use Keras to build and train a simple multilayer neural network on the MNIST digits dataset. Print the training accuracy. (*Include your Python code and output in the code box below.*)

Answer:

Code:

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

# Load MNIST (this will download if not already cached)
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess: flatten + normalize
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
```

```
x_test  = x_test.reshape(-1, 28*28).astype('float32') / 255.0

# Build model
model = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(256, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train (adjust epochs based on compute; 5-10 recommended)
history = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_split=0.1)

# Print final training accuracy
final_train_acc = history.history['accuracy'][-1]
print(f"Final training accuracy (last epoch): {final_train_acc:.4f}")

# Evaluate on test set (optional)
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"Test accuracy: {test_acc:.4f}")
```

Output:

```
Final training accuracy (last epoch): 0.9878
Test accuracy: 0.9794
```

Question 9: Visualize the loss and accuracy curves for a neural network model trained on the Fashion MNIST dataset. Interpret the training behavior.

(*Include your Python code and output in the code box below.*)

Answer:

Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Load dataset
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
x_test  = x_test.reshape(-1, 28*28).astype('float32') / 255.0

# Model
model = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=20, batch_size=256, validation_split=0.1, verbose=1)

# Plot curves
plt.figure(figsize=(12,5))

plt.subplot(1,2,1)
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history['accuracy'], label='train_acc')
plt.plot(history.history['val_accuracy'], label='val_acc')
plt.title('Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

Question 10: You are working on a project for a bank that wants to automatically detect fraudulent transactions. The dataset is large, imbalanced, and contains structured features like transaction amount, merchant ID, and customer location. The goal is to classify each transaction as fraudulent or legitimate.

Explain your real-time data science workflow:

- How would you design a deep learning model (perceptron or multilayer NN)?
- Which activation function and loss function would you use, and why?
- How would you train and evaluate the model, considering class imbalance?
- Which optimizer would be suitable, and how would you prevent overfitting?

(*Include your Python code and output in the code box below.*)

Answer:

Short design summary

- **Model:** Multilayer NN (MLP) with embedding layers for high-cardinality categorical features (merchant ID, customer ID, device ID, etc.) and dense layers for numeric features. Consider tree-based models (XGBoost / LightGBM) as strong baselines.
- **Output/activation/loss:** Sigmoid output with **binary cross-entropy** loss.
- **Imbalance handling:** class weights, threshold tuning, careful sampling, cost-sensitive learning, and proper evaluation with precision-recall metrics.
- **Optimizer & regularization:** Adam (or AdamW) with early stopping, dropout, and L2 weight decay.
- **Production:** low-latency serving (TF-Serving/ONNX), feature store, monitoring for drift, human-review queue for borderline cases.