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:

Deep Learning (DL) is a subset of machine learning that uses artificial neural networks with multiple layers (hence *deep*) to learn hierarchical representations of data. Deep networks learn multiple levels of abstraction: raw input → low-level features → higher-level concepts, allowing them to model very complex, non-linear relationships.

Evolution (brief):

- **1940s–1950s:** Early ideas: McCulloch–Pitts neuron, perceptron (Rosenblatt).
- **1960s–1980s:** Limitations recognized (Minsky & Papert). Backpropagation rediscovered 1986 (Rumelhart, Hinton, Williams).
- **1990s–2000s:** Neural nets used but limited by compute/data.
- **2010s:** Big datasets, GPUs, improved architectures (CNNs, RNNs, LSTMs), practical success (image/speech/NLP).
- **2012 onward:** AlexNet, then many modern deep architectures and frameworks (TensorFlow, PyTorch).

How DL differs from traditional ML:

- **Feature learning:** DL automatically learns features from raw data; traditional ML often relies on hand-crafted features.
- **Model complexity & capacity:** Deep networks have many parameters and can model highly non-linear relationships; typical traditional models (SVM, logistic regression, random forests) have less raw parameter capacity.

- **Data requirements:** DL usually requires much more data to generalize well; classical ML can work with smaller datasets.
- **Compute:** DL often requires GPUs and heavy compute; classical ML is usually less compute-intensive.
- **Interpretability:** Traditional ML models can be easier to interpret (e.g., linear models), while DL is often more opaque.

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

Answer:

Perceptron (single neuron) — architecture and operation:

- Inputs: x=[x1,x2,…,xn]x = [x_1, x_2, \dots, x_n]x=[x1,x2,…,xn].
- Weights: w=[w1,w2,…,wn]w = [w_1, w_2, \dots, w_n]w=[w1,w2,…,wn] and bias bbb.
- Net input: z=w·x+b=∑iwixi+bz = w \cdot x + b = \sum_{i} w_i x_i + bz=w·x+b=∑iwixi+b.
- Activation (step): y={1if z≥00otherwisey = \begin{cases}1 & \text{if } z \ge 0 \\ 0 & \text{otherwise}\end{cases}y={10if z≥0otherwise.
- Learning: weights updated by perceptron learning rule: w←w+η(ytrue−ypred)xw \leftarrow w + \eta (y_{true} - y_{pred}) xw←w+η(ytrue−ypred)x, b←b+η(ytrue−ypred)b \leftarrow b + \eta (y_{true} - y_{pred})b←b+η(ytrue−ypred).

Limitations:

1. **Linearly separable only:** A single perceptron can only solve linearly separable problems (e.g., AND, OR), not XOR.
2. **Non-differentiable activation:** Step function is non-differentiable; backpropagation requires differentiable activations (sigmoid/ReLU/tanh) for multilayer nets.
3. **Limited representational capacity:** A single layer perceptron cannot learn complex hierarchical features.

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

Answer:

Purpose of activation functions:
They introduce non-linearity into the model so that neural networks can learn complex mappings beyond linear transformations. Without non-linear activations, multiple layers collapse to an equivalent single linear transformation.

Compare Sigmoid, ReLU, Tanh

- **Sigmoid:** σ(x)=11+e−x\sigma(x) = \frac{1}{1+e^{-x}}σ(x)=1+e−x1
  - Range: (0, 1).
  - Pros: Output interpretable as probability-like (good for final binary output).
  - Cons: Saturates for large |x||x||x| → vanishing gradients; not zero-centered (can slow convergence).
- **Tanh:** 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
  - Range: (-1, 1).
  - Pros: Zero-centered (helps optimization), steeper gradient than sigmoid.
  - Cons: Still saturates → can suffer vanishing gradients for deep nets.
- **ReLU (Rectified Linear Unit):** ReLU(x)=max⁡(0,x)\text{ReLU}(x) = \max(0,x)ReLU(x)=max(0,x)
  - Range: [0, ∞).
  - Pros: Simple, efficient, alleviates vanishing gradient for positive region; sparse activations.
  - Cons: "Dead ReLU" problem (neurons stuck at 0 if inputs produce negative values and weights update poorly); not zero-centered.

Usage summary: ReLU (or its variants) is typically used for hidden layers in modern networks. Sigmoid/Tanh may be used in outputs depending on task (sigmoid for binary classification, softmax for multi-class).

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

Answer:

Purpose of activation functions:
They introduce non-linearity into the model so that neural networks can learn complex mappings beyond linear transformations. Without non-linear activations, multiple layers collapse to an equivalent single linear transformation.

Compare Sigmoid, ReLU, Tanh

- **Sigmoid:** σ(x)=11+e−x\sigma(x) = \frac{1}{1+e^{-x}}σ(x)=1+e−x1
  - Range: (0, 1).
  - Pros: Output interpretable as probability-like (good for final binary output).
  - Cons: Saturates for large |x||x||x| → vanishing gradients; not zero-centered (can slow convergence).
- **Tanh:** 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
  - Range: (-1, 1).
  - Pros: Zero-centered (helps optimization), steeper gradient than sigmoid.
  - Cons: Still saturates → can suffer vanishing gradients for deep nets.
- **ReLU (Rectified Linear Unit):** ReLU(x)=max⁡(0,x)\text{ReLU}(x) = \max(0,x)ReLU(x)=max(0,x)
  - Range: [0, ∞).
  - Pros: Simple, efficient, alleviates vanishing gradient for positive region; sparse activations.
  - Cons: "Dead ReLU" problem (neurons stuck at 0 if inputs produce negative values and weights update poorly); not zero-centered.

Usage summary: ReLU (or its variants) is typically used for hidden layers in modern networks. Sigmoid/Tanh may be used in outputs depending on task (sigmoid for binary classification, softmax for multi-class).

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

Answer:

Role: Optimizers update model weights to minimize the cost function by following gradients. They determine step size, momentum, and adaptivity.

Gradient Descent (GD) variants:

- **Batch Gradient Descent (vanilla GD):** Compute gradient on whole dataset each step. Pros: accurate gradient; Cons: slow for large datasets.
- **Stochastic Gradient Descent (SGD):** Update per example; noisy updates can escape shallow minima but noisy.
- **Mini-batch SGD:** Practical compromise: update per small batch; used widely.

RMSprop:

- Keeps exponentially decaying average of squared gradients E[g2]E[g^2]E[g2] and divides the learning rate by the root of this average.
- Adapts learning rate per-parameter, good for non-stationary objectives.
- Works well for RNNs.

Adam (Adaptive Moment Estimation):

- Combines ideas of RMSprop (adaptive learning rates) and momentum (first moment estimate).
- Maintains running averages of gradients (m) and squared gradients (v), with bias correction.
- Very popular: often robust, requires less tuning, generally fast convergence.

Comparison summary:

- **GD/SGD:** simple; may need careful learning rate scheduling and momentum.
- **RMSprop:** adaptive; good for problems with noisy gradients.
- **Adam:** adaptive + momentum; often default choice for many deep learning tasks.

- **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:

```
# Perceptron for AND gate (from scratch using NumPy)
import numpy as np

# Training data for AND gate
# Inputs: [x1, x2], Output: x1 AND x2
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]], dtype=float)

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

# Perceptron parameters
np.random.seed(42)
weights = np.random.randn(2) * 0.1
bias = 0.0
lr = 0.1
epochs = 20

def step(x):
    return 1 if x >= 0 else 0

# Training loop
for epoch in range(epochs):
    total_loss = 0
    for xi, yi in zip(X, y):
        z = np.dot(weights, xi) + bias
        y_pred = step(z)
        error = yi - y_pred
        # Perceptron weight update
        weights += lr * error * xi
        bias += lr * error
        total_loss += abs(error)
    print(f"Epoch {epoch+1}/{epochs} - Total classification errors: {total_loss}")

print("\nTrained weights:", weights, "bias:", bias)

# Testing
print("\nPredictions after training:")
for xi in X:
```

```
z = np.dot(weights, xi) + bias
print(xi, "->", step(z))
```

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:

```
# Activation function plots
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 400)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

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

plt.figure(figsize=(12,4))

plt.subplot(1,3,1)
plt.plot(x, sigmoid(x))
plt.title("Sigmoid")
plt.grid(True)

plt.subplot(1,3,2)
plt.plot(x, tanh(x))
plt.title("Tanh")
plt.grid(True)

plt.subplot(1,3,3)
plt.plot(x, relu(x))
plt.title("ReLU")
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:

```
# Keras simple MLP on MNIST
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.utils import to_categorical

# Load MNIST
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess
x_train = x_train.reshape(-1, 28*28).astype("float32") / 255.0
x_test = x_test.reshape(-1, 28*28).astype("float32") / 255.0
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# Model
model = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(128, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")
])

model.compile(optimizer="adam",
              loss="categorical_crossentropy",
              metrics=["accuracy"])

# Train
history = model.fit(x_train, y_train_cat, validation_split=0.1, epochs=5, batch_size=128)

# Print final training accuracy
train_acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]
print(f"Final training accuracy: {train_acc:.4f}")
print(f"Final validation accuracy: {val_acc:.4f}")

# Evaluate on test set
```

```
test_loss, test_acc = model.evaluate(x_test, y_test_cat, verbose=0)
print(f"Test accuracy: {test_acc:.4f}")
```

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:

```
# Train on Fashion MNIST and visualize loss & accuracy
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Load Fashion MNIST
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

# Preprocess
x_train = x_train.reshape(-1, 28*28).astype("float32") / 255.0
x_test = x_test.reshape(-1, 28*28).astype("float32") / 255.0
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# Model (same simple MLP)
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="categorical_crossentropy",
          metrics=["accuracy"])

history = model.fit(x_train, y_train_cat, validation_split=0.1, epochs=10, batch_size=256)

# Plot training & validation loss
plt.figure(figsize=(12,4))
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.legend()

# Plot training & validation accuracy
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.legend()

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:

Design & reasoning:

1. **Data pipeline / real-time workflow (high-level):**
   - **Data ingestion**: stream transactions from source (Kafka, API) into feature store.
   - **Feature store / preprocessing**: standardize numeric features (amount), encode categorical features (merchant ID) using target/embedding or one-hot/ordinal; compute time-based features (hour/day), device/risk features; maintain running aggregates per customer.
   - **Feature engineering**: create features like transaction frequency, avg amount per customer, merchant risk score, geolocation distance, velocity features (transactions per minute).
   - **Model serving**: the trained model runs in real-time (low latency) to score each transaction; if score > threshold, trigger human review or block.
   - **Monitoring & retraining pipeline**: monitor drift, label feedback loop, periodic retraining.
2. **Model selection & architecture:**
   - Use a **multilayer feed-forward network** or tree-based models. For NN: an MLP with embedding layers for high-cardinality categorical features (merchant/customer), numeric inputs normalized.
   - Example MLP: Input numeric → dense layers with ReLU → dropout → output with Sigmoid.
3. **Activation & loss functions:**
   - **Hidden activations:** ReLU (efficient, good practice).
   - **Output activation:** Sigmoid (binary probability).
   - **Loss:** Binary cross-entropy (log loss): `binary_crossentropy`.
4. **Handling class imbalance:**
   - Use **class weighting** in loss (higher weight for minority fraud class).
   - Or use **resampling**: SMOTE (oversample minority) or undersample majority, or combined.
   - For neural nets, prefer **class_weight** or focal loss (reduces weight of easy negatives).
   - Evaluate with **precision/recall/F1 and ROC-AUC / PR-AUC**, not only accuracy.
5. **Optimizer & overfitting prevention:**
   - **Optimizer:** Adam (good default).
   - **Regularization:** Dropout, L2 weight decay.
   - **Early stopping** based on validation PR-AUC or validation loss.
   - **Batch normalization** may help stability.
   - Use cross-validation or time-based split for evaluation if data has temporal order.

```
# Fraud detection model skeleton (Keras)
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models, callbacks
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, roc_auc_score, precision_recall_curve,
auc

# Example synthetic placeholder: replace with your actual preprocessed features & labels
# X: numpy array (n_samples, n_features), y: binary labels (0/1)
# For demonstration, we create a toy imbalanced dataset
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=20000, n_features=20, weights=[0.98, 0.02],
                           n_informative=6, n_redundant=2, random_state=42)

# Train/val split (if data is temporal, use time-based split instead)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, stratify=y,
random_state=42)

# Build simple MLP
def build_model(input_dim):
    inp = layers.Input(shape=(input_dim,))
    x = layers.Dense(128, activation='relu')(inp)
    x = layers.Dropout(0.3)(x)
    x = layers.Dense(64, activation='relu')(x)
    x = layers.Dropout(0.2)(x)
    x = layers.Dense(32, activation='relu')(x)
    out = layers.Dense(1, activation='sigmoid')(x)
    model = models.Model(inputs=inp, outputs=out)
    return model

model = build_model(X_train.shape[1])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['AUC', 'Precision',
'Recall'])

# Compute class weights: inverse frequency
from sklearn.utils.class_weight import compute_class_weight
classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight='balanced', classes=classes,
y=y_train)
class_weight_dict = {k: v for k, v in zip(classes, class_weights)}
print("Class weights:", class_weight_dict)
```

```
# Callbacks
es = callbacks.EarlyStopping(monitor='val_auc', mode='max', patience=5,
restore_best_weights=True)
reduce_lr = callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3,
min_lr=1e-6)

# Train
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50,
batch_size=1024,
                    class_weight=class_weight_dict, callbacks=[es, reduce_lr])

# Evaluate
y_pred_proba = model.predict(X_val).ravel()
y_pred = (y_pred_proba >= 0.5).astype(int)

print("ROC AUC:", roc_auc_score(y_val, y_pred_proba))
print("\nClassification report (val):")
print(classification_report(y_val, y_pred, digits=4))

# Compute PR-AUC
prec, rec, _ = precision_recall_curve(y_val, y_pred_proba)
pr_auc = auc(rec, prec)
print("PR AUC:", pr_auc)
```