Assignment 1: Building a Simple Neural Network on Iris Dataset

Objective: To understand the workflow of a Neural Network in TensorFlow/Keras using the Iris dataset, with fixed activation and optimizer settings.

Step 1: Import Libraries

Import the required libraries (tensorflow, keras.layers, sklearn, matplotlib).

Step 2: Load & Preprocess Data

Load the Iris dataset.

Normalize the features (mean=0, variance=1).

One-hot encode the labels (since Iris has 3 classes).

Split the data into training and testing sets.

Step 3: Build the Model

Create a Sequential model with:

Input layer: 4 features.

Hidden layers: 10 neurons (ReLU), 8 neurons (ReLU).

Output layer: 3 neurons (Softmax for 3 classes).

Step 4: Compile the Model

Compile the model with:

Optimizer: sgd

Loss: categorical_crossentropy

Metric: accuracy

Step 5: Train the Model Train the model for 50 epochs and store the history.

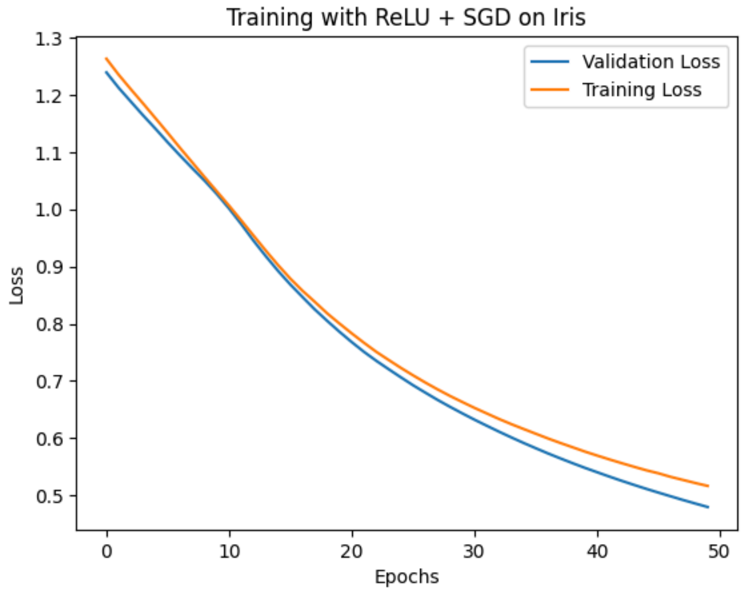
Step 6: Visualize Convergence Plot both training and validation loss curves against epochs.

Step 7: Final Evaluation Print the test accuracy of the model.

```
# Step 1: Import libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelBinarizer

# Step 2: Load & preprocess Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Normalize features
X = StandardScaler().fit_transform(X)

# One-hot encode labels
y = LabelBinarizer().fit_transform(y)

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

# Step 3: Build the model (activation + optimizer fixed for now)
model = keras.Sequential([
    layers.Dense(10, activation="relu", input_shape=(4,)),
    layers.Dense(8, activation="relu"),
    layers.Dense(3, activation="softmax")  # 3 classes in Iris
])

model.compile(
    optimizer="sgd",   # 🔹 one optimizer only (students can later change)
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

# Step 4: Train the model
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
```

```
    epochs=50,
    verbose=0
)

# Step 5: Plot validation loss curve
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.plot(history.history["loss"], label="Training Loss")
plt.title("Training with ReLU + SGD on Iris")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()

# Step 6: Final evaluation
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {acc:.4f}")
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` arg
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```




```
Test Accuracy: 0.8667
```

Assignment 2 : Building a Simple Neural Network on Iris Dataset

Objective: To understand the complete workflow of a Neural Network in TensorFlow/Keras using the Iris dataset, with fixed activation and optimizer settings.

Step 1: Import Libraries

Import all the required libraries for model development:

tensorflow.keras for building the neural network

sklearn for loading and preprocessing the dataset

matplotlib for plotting the training progress

Step 2: Load & Preprocess Data

Load the Iris dataset from sklearn.

Normalize the features (standardization → mean=0, variance=1).

Apply one-hot encoding on labels since Iris has 3 classes.

Split the dataset into training and testing sets.

Step 3: Build the Model

Create a Sequential model with:

Input layer: 4 features

Hidden layer 1: 10 neurons, ReLU activation

Hidden layer 2: 8 neurons, ReLU activation

Output layer: 3 neurons, Softmax activation (for multi-class classification)

Step 4: Compile the Model

Compile the model with:

Optimizer: SGD (Stochastic Gradient Descent)

Loss function: categorical_crossentropy (since we have multi-class labels)

Metric: accuracy

Step 5: Train the Model

Train the model for 50 epochs on the training set.

Store the training history to visualize convergence.

Step 6: Visualize Convergence

Plot both training loss and validation loss across epochs.

These plots help to understand how the model converges during training.

Step 7: Final Evaluation

Evaluate the model on the test set.

Print the final test accuracy.

```
# Step 1: Import libraries
# TensorFlow/Keras for deep learning
# Sklearn for dataset, preprocessing, splitting
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# -------------------------------
# Step 2: Load & preprocess dataset
# -------------------------------
iris = load_iris()
X, y = iris.data, iris.target   # Features and labels

# Normalize features (so all inputs are on the same scale)
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Convert labels into One-Hot vectors (e.g., 0 → [1,0,0], 1 → [0,1,0], 2 → [0,0,1])
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y.reshape(-1, 1))

# Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# -------------------------------
# Step 3: Build model
# -------------------------------
# Sequential model = layers stacked one after another
model = Sequential()

# First hidden layer
# Units = 10 neurons
# Activation = ReLU (students can replace with sigmoid, tanh, etc.)
model.add(Dense(10, activation='relu', input_shape=(X_train.shape[1],)))

# Output layer
# Units = 3 (since Iris has 3 classes)
# Activation = softmax (to get probabilities for each class)
model.add(Dense(3, activation='softmax'))

# -------------------------------
# Step 4: Compile model
# -------------------------------
# Optimizer = SGD (students can replace with Adam, RMSProp, etc.)
# Loss = categorical_crossentropy (used for multi-class classification)
# Metric = accuracy
model.compile(
    optimizer='sgd',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# -------------------------------
```

```
# Step 5: Train model
# ------------------------------
# The backpropagation happens automatically inside model.fit()
# Forward pass → Loss → Backward pass → Optimizer update
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=50, batch_size=8, verbose=1)

# ------------------------------
# Step 6: Evaluate model
# ------------------------------
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f"\nFinal Test Accuracy: {acc:.2f}")
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8528 - loss: 0.4325 - val_accuracy: 0.8667 - val_loss: 0.4261
Epoch 24/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8023 - loss: 0.4837 - val_accuracy: 0.8667 - val_loss: 0.4173
Epoch 25/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8731 - loss: 0.4641 - val_accuracy: 0.8667 - val_loss: 0.4090
Epoch 26/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8596 - loss: 0.4864 - val_accuracy: 0.8667 - val_loss: 0.4012
Epoch 27/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8733 - loss: 0.4394 - val_accuracy: 0.8667 - val_loss: 0.3938
Epoch 28/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8789 - loss: 0.4078 - val_accuracy: 0.8667 - val_loss: 0.3869
Epoch 29/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9168 - loss: 0.3752 - val_accuracy: 0.8667 - val_loss: 0.3802
Epoch 30/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8526 - loss: 0.4130 - val_accuracy: 0.8667 - val_loss: 0.3738
Epoch 31/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8255 - loss: 0.4263 - val_accuracy: 0.8667 - val_loss: 0.3677
Epoch 32/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8761 - loss: 0.4103 - val_accuracy: 0.8667 - val_loss: 0.3618
Epoch 33/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8844 - loss: 0.4146 - val_accuracy: 0.8667 - val_loss: 0.3561
Epoch 34/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8992 - loss: 0.3866 - val_accuracy: 0.9000 - val_loss: 0.3506
Epoch 35/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9035 - loss: 0.3735 - val_accuracy: 0.9000 - val_loss: 0.3454
Epoch 36/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8819 - loss: 0.3696 - val_accuracy: 0.9000 - val_loss: 0.3402
Epoch 37/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9052 - loss: 0.3778 - val_accuracy: 0.9000 - val_loss: 0.3353
Epoch 38/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8931 - loss: 0.3822 - val_accuracy: 0.9000 - val_loss: 0.3305
Epoch 39/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9273 - loss: 0.3168 - val_accuracy: 0.9000 - val_loss: 0.3258
Epoch 40/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9117 - loss: 0.3025 - val_accuracy: 0.9000 - val_loss: 0.3212
Epoch 41/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9103 - loss: 0.3262 - val_accuracy: 0.9000 - val_loss: 0.3168
Epoch 42/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9118 - loss: 0.3128 - val_accuracy: 0.9333 - val_loss: 0.3125
Epoch 43/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8974 - loss: 0.3383 - val_accuracy: 0.9333 - val_loss: 0.3083
Epoch 44/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9043 - loss: 0.3166 - val_accuracy: 0.9333 - val_loss: 0.3042
Epoch 45/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8812 - loss: 0.3804 - val_accuracy: 0.9333 - val_loss: 0.3003
Epoch 46/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9321 - loss: 0.3151 - val_accuracy: 0.9333 - val_loss: 0.2964
Epoch 47/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9343 - loss: 0.2867 - val_accuracy: 0.9333 - val_loss: 0.2927
Epoch 48/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9324 - loss: 0.2876 - val_accuracy: 0.9333 - val_loss: 0.2891
Epoch 49/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9308 - loss: 0.3127 - val_accuracy: 0.9333 - val_loss: 0.2855
Epoch 50/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9221 - loss: 0.3256 - val_accuracy: 0.9333 - val_loss: 0.2820

Final Test Accuracy: 0.93
```

Assignment Question 3

In this task, you will explore the effect of different optimizers on training performance.

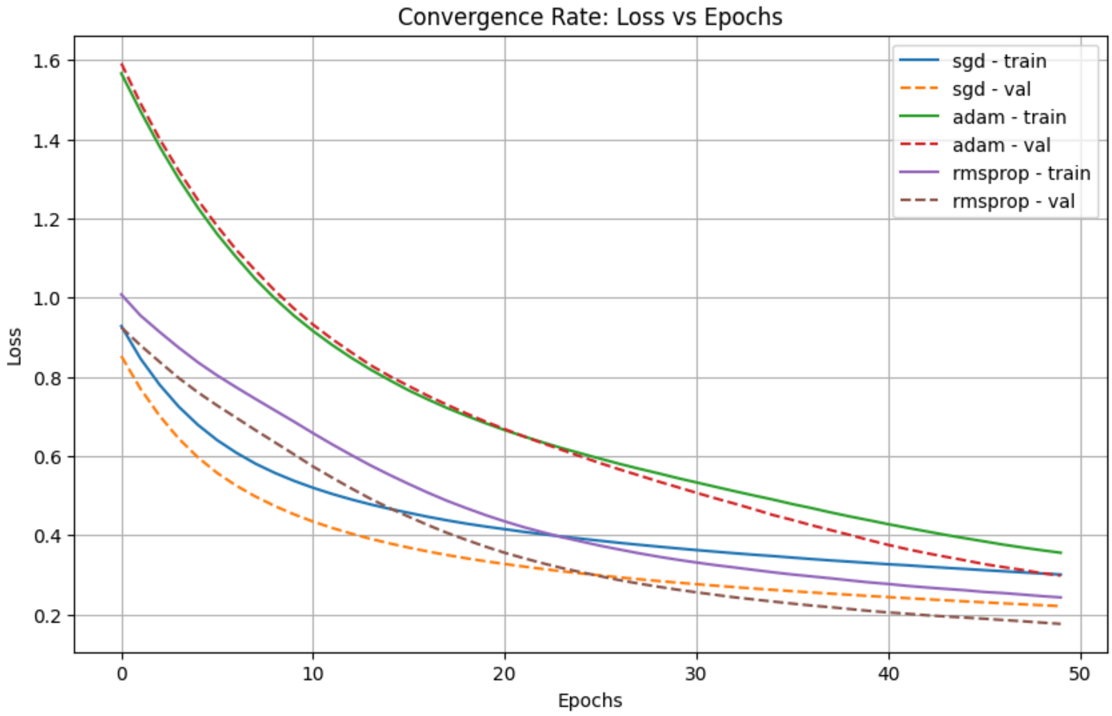
Use the given Iris dataset code where three optimizers (SGD, Adam, and RMSprop) are compared.

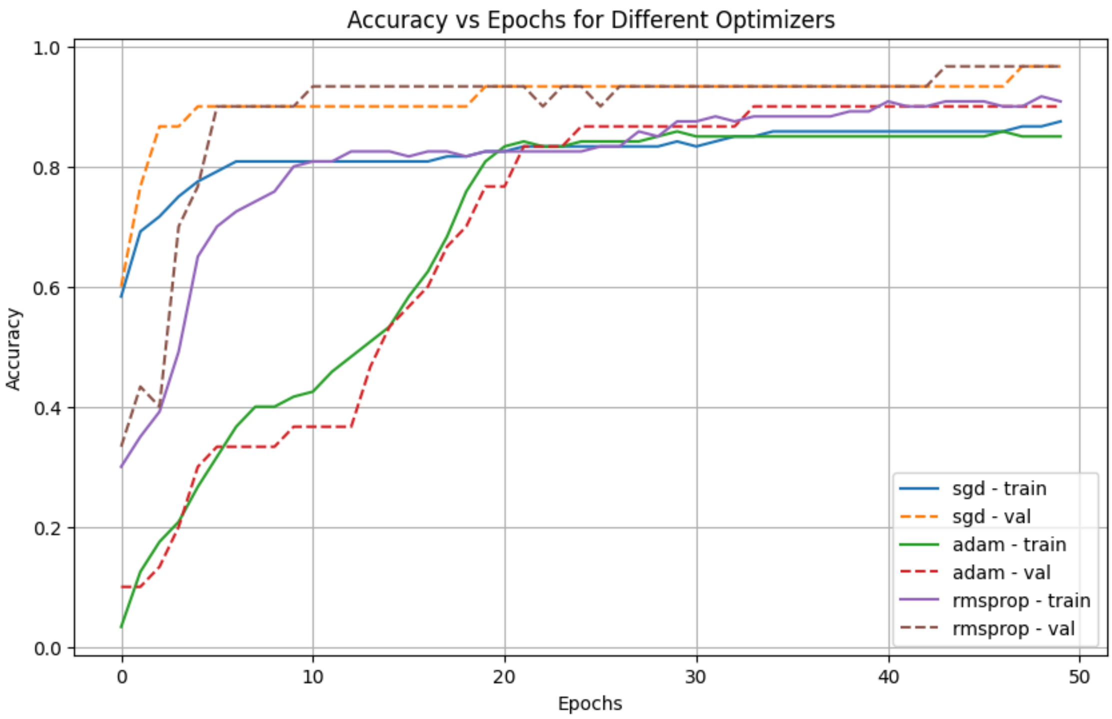
Carefully observe the loss curves and accuracy curves obtained for each optimizer.

```
# Step 1: Import Libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler, OneHotEncoder
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Step 2: Load & Preprocess Dataset
iris = load_iris()
X, y = iris.data, iris.target

# Normalize features
X = StandardScaler().fit_transform(X)

# One-hot encode labels
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y.reshape(-1,1))

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Step 3: Build a Function to Create Model
def create_model(activation="relu", optimizer="sgd"):
    model = Sequential()
    model.add(Dense(10, input_shape=(4,), activation=activation))
    model.add(Dense(3, activation="softmax"))  # 3 classes
    model.compile(optimizer=optimizer,
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model

# Step 4: Train with Different Optimizers
optimizers = ["sgd", "adam", "rmsprop"]  # students can try one at a time
history_dict = {}

for opt in optimizers:
    print(f"\nTraining with optimizer: {opt}")
    model = create_model(activation="relu", optimizer=opt)
    history = model.fit(
        X_train, y_train,
        validation_data=(X_test, y_test),
        epochs=50, batch_size=8, verbose=0
    )
    history_dict[opt] = history

# Step 5: Plot Loss Curves for Comparison
plt.figure(figsize=(10,6))
for opt, history in history_dict.items():
    plt.plot(history.history["loss"], label=f"{opt} - train")
    plt.plot(history.history["val_loss"], linestyle="--", label=f"{opt} - val")

plt.title("Convergence Rate: Loss vs Epochs")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()

# Step 6: Plot Accuracy Curves
plt.figure(figsize=(10,6))
for opt, history in history_dict.items():
    plt.plot(history.history["accuracy"], label=f"{opt} - train")
    plt.plot(history.history["val_accuracy"], linestyle="--", label=f"{opt} - val")

plt.title("Accuracy vs Epochs for Different Optimizers")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()
```

```
Training with optimizer: sgd
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` a
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Training with optimizer: adam

Training with optimizer: rmsprop
```





Assignment Question 4: Effect of Activation Functions and Optimizers

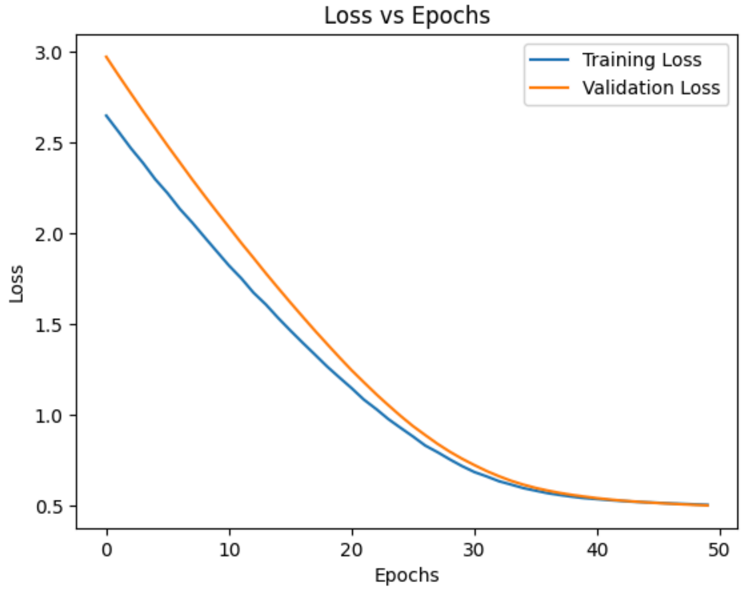
In this exercise, you will analyze how different activation functions and different optimizers affect the training of a simple neural network on the Iris dataset.

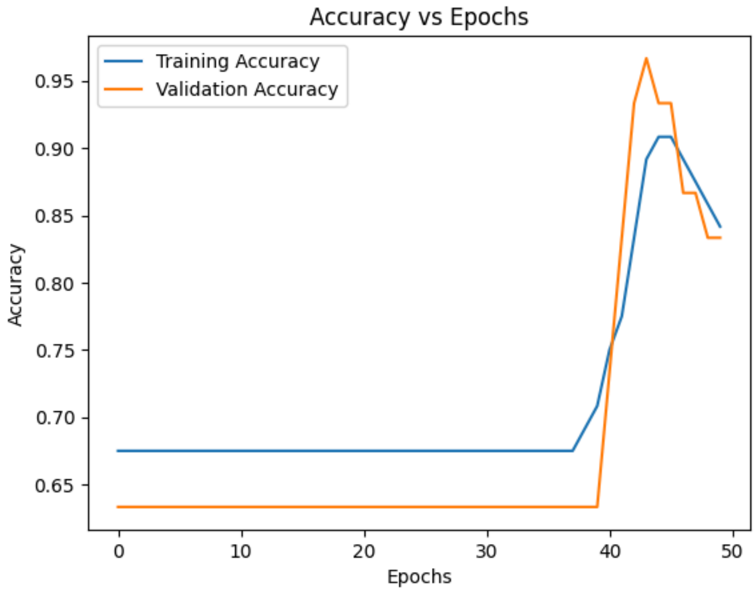
```
# ========================
# Neural Network on Iris Dataset
# Showing importance of Activation Functions & Optimizers
# ========================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD, Adam

# ----------------------------
# Step 1: Load dataset
# ----------------------------
iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)

# One-hot encoding for labels (0 → [1,0,0], etc.)
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y)

# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# ====================================================
# Teaching Note:
# Step 2: Build a simple NN with ONE activation & ONE optimizer
#   → Students will later change activation (sigmoid, tanh, relu, leaky relu)
#   → and optimizer (SGD, Adam, RMSprop, Momentum)
# ====================================================
model = Sequential([
    Dense(10, activation='relu', input_shape=(4,)),   # Try changing 'relu' to 'sigmoid', 'tanh', etc.
    Dense(3, activation='softmax')                    # Output layer for 3 classes
])

# Compile model with ONE optimizer initially
# Students can replace 'adam' with 'sgd' or 'sgd with momentum'
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


# ----------------------------
# Step 3: Train the model
# ----------------------------
history = model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test), verbose=0)


# ====================================================
# Step 4: Plotting Results
# ====================================================

# Plot 1: Loss vs Epochs
# ----------------------------
# This plot explains:
# - With different ACTIVATION FUNCTIONS → some converge slowly (sigmoid),
#   some fast (ReLU).
# - With different OPTIMIZERS → SGD may zigzag and converge slower,
#   Adam converges faster and smoother.
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot 2: Accuracy vs Epochs
# ----------------------------
# This shows how quickly the model reaches high accuracy depending
# on the optimizer. Adam usually reaches good accuracy faster.
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# ====================================================
# change only ACTIVATION FUNCTION (keep optimizer same):
#       Compare Loss vs Epochs plots → ReLU usually converges faster.
#       Optional: Plot decision boundaries for better visualization.
#
# change only OPTIMIZER (keep activation same):
#       Compare Loss & Accuracy plots → Adam, RMSprop faster than SGD.
#
```

```
#       First run with ReLU + Adam (default).
#       Then swap activation functions.
#       Then swap optimizers.
# =====================================================
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` arg
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```





Assignment 5: Backpropagation from Scratch with NumPy

In this assignment, you will implement a simple feedforward neural network on the Iris dataset using only NumPy, without relying on TensorFlow or Keras. You are required to perform all the essential steps manually, including forward propagation, loss calculation, and backpropagation using explicit derivative computations. The network should have one hidden layer, use the sigmoid or ReLU activation function, and output probabilities using softmax for multi-class classification. Train the model for a fixed number of epochs, plot the training loss convergence, and finally report the classification accuracy on the test set.

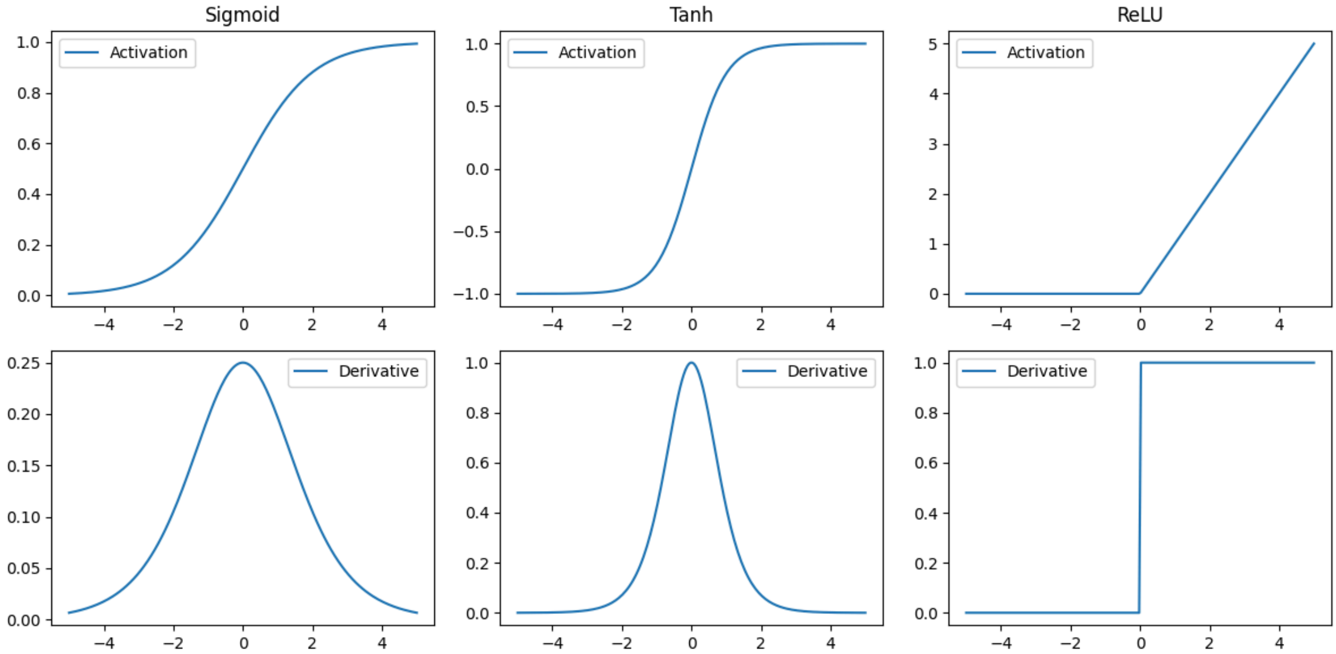
```
import numpy as np

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

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

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

def relu_derivative(x):
    return np.where(x > 0, 1, 0)
```

```
# ---------- Data ----------
x = np.array([[1, 0]])   # Input
y = np.array([[1]])      # Target output

# Initialize weights + bias
np.random.seed(0)
W1 = np.random.randn(2, 2)  # Input -> Hidden (2 neurons)
b1 = np.zeros((1, 2))
W2 = np.random.randn(2, 1)  # Hidden -> Output (1 neuron)
b2 = np.zeros((1, 1))

# Learning rate
lr = 0.1

# ---------- Forward Pass ----------
z1 = np.dot(x, W1) + b1
a1 = sigmoid(z1)  # Try relu(z1) here for comparison
z2 = np.dot(a1, W2) + b2
a2 = sigmoid(z2)

# Loss (MSE)
loss = np.mean((y - a2) ** 2)
print("Initial Loss:", loss)

# ---------- Backward Pass ----------
# Output layer
d_loss_a2 = -(y - a2)       # dL/da2
d_a2_z2 = sigmoid_derivative(z2)  # da2/dz2
d_z2_W2 = a1.T              # dz2/dW2

dL_dz2 = d_loss_a2 * d_a2_z2
dL_dW2 = np.dot(d_z2_W2, dL_dz2)
dL_db2 = np.sum(dL_dz2, axis=0, keepdims=True)

# Hidden layer
d_z2_a1 = W2
dL_da1 = np.dot(dL_dz2, d_z2_a1.T)
d_a1_z1 = sigmoid_derivative(z1)  # change to relu_derivative(z1) if using ReLU
dL_dz1 = dL_da1 * d_a1_z1
dL_dW1 = np.dot(x.T, dL_dz1)
dL_db1 = np.sum(dL_dz1, axis=0, keepdims=True)

# ---------- Update weights ----------
W1 -= lr * dL_dW1
b1 -= lr * dL_db1
W2 -= lr * dL_dW2
b2 -= lr * dL_db2

print("Updated Loss after 1 step:", np.mean((y - sigmoid(np.dot(sigmoid(np.dot(x, W1)+b1), W2)+b2))**2))
```

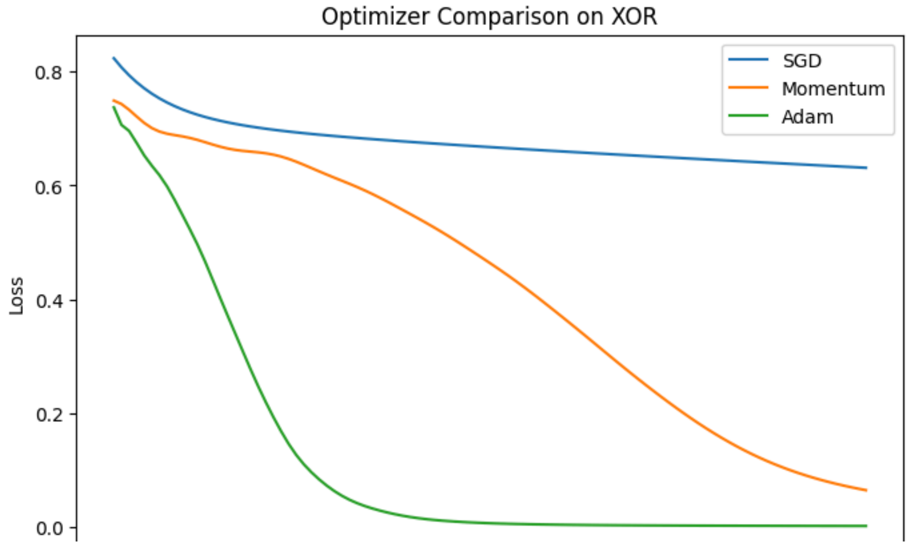
```
Initial Loss: 0.07135728861485516
Updated Loss after 1 step: 0.0701049633900343
```

```
import matplotlib.pyplot as plt

# Input range
x_vals = np.linspace(-5, 5, 200)

# Functions
def tanh(x): return np.tanh(x)
def tanh_derivative(x): return 1 - np.tanh(x)**2

# Plot activations & derivatives
functions = {
    "Sigmoid": (sigmoid, sigmoid_derivative),
    "Tanh": (tanh, tanh_derivative),
    "ReLU": (relu, relu_derivative)
}

plt.figure(figsize=(12, 6))
for i, (name, (f, f_prime)) in enumerate(functions.items()):
    plt.subplot(2, 3, i+1)
    plt.title(name)
    plt.plot(x_vals, f(x_vals), label="Activation")
    plt.legend()
    plt.subplot(2, 3, i+4)
    plt.plot(x_vals, f_prime(x_vals), label="Derivative")
    plt.legend()
plt.tight_layout()
```

```
plt.show()
```




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

# XOR Dataset
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

optimizers = {
    "SGD": tf.keras.optimizers.SGD(learning_rate=0.1),
    "Momentum": tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9),
    "Adam": tf.keras.optimizers.Adam(learning_rate=0.1)
}

histories = {}

for name, opt in optimizers.items():
    model = Sequential([
        Dense(4, input_dim=2, activation='tanh'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    history = model.fit(X, y, epochs=100, verbose=0)
    histories[name] = history.history['loss']

# Plot Loss Curves
plt.figure(figsize=(8,5))
for name, loss in histories.items():
    plt.plot(loss, label=name)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Optimizer Comparison on XOR")
plt.legend()
plt.show()
```


Optimizer Comparison on XOR
SGD
Momentum
Adam
Loss
0.8
0.6
0.4
0.2
0.0