Lab

Implementation of Artificial neural network for classification.

Design Neural network model with Keras Libraries

```
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np

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

model = Sequential()
model.add(Dense(2, input_shape=(2,)))
model.add(Activation('sigmoid'))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='mean_squared_error', optimizer='sgd', metrics=
['accuracy'])
model.summary()
```

Backpropagation using ANN

```
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Forward and backward propagation
def train(X, y, hw, hb, ow, ob, lr, epochs):
    for _ in range(epochs):
        # Forward pass
        h_out = sigmoid(np.dot(X, hw) + hb)
        p_out = sigmoid(np.dot(h_out, ow) + ob)
```

```
        # Backpropagation
        o_error = (y - p_out) * sigmoid_derivative(p_out)
        h_error = o_error.dot(ow.T) * sigmoid_derivative(h_out)

        # Update weights
        ow += h_out.T.dot(o_error) * lr
        hw += X.T.dot(h_error) * lr

    return hw, ow, p_out

# Input data (normalized)
X = np.array([[2, 9], [1, 5], [3, 6]], dtype=float)
X = X / np.amax(X, axis=0)  # Feature scaling

# Output data (normalized)
y = np.array([[92], [86], [89]], dtype=float) / 100

# Initialize parameters
np.random.seed(42)
hw = np.random.rand(2, 3)  # Hidden layer weights
hb = np.random.rand(1, 3)  # Hidden layer bias
ow = np.random.rand(3, 1)  # Output layer weights
ob = np.random.rand(1, 1)  # Output layer bias

# Training parameters
lr = 0.1      # Learning rate
epochs = 5000  # Number of training iterations

# Train the neural network
hw, ow, p_out = train(X, y, hw, hb, ow, ob, lr, epochs)

# Print final predictions
print("Predicted Output:\n", p_out)
```

Baseline NN - Regression - Boston

```
# Install required libraries
!pip install tensorflow
!pip install scikeras
```

```
# Import necessary libraries
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import initializers, optimizers
from tensorflow.keras.losses import MeanSquaredError
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import KFold, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Load dataset
try:
    dataset = pd.read_csv("housing.csv").values
    X, Y = dataset[:, :-1].astype(float), dataset[:, -1].astype(fl
oat)
except Exception as e:
    raise SystemExit(f"Error loading dataset: {e}")

# Define model function
def create_model(hidden_units=13):
    model = Sequential([
        Dense(hidden_units, input_shape=(13,),
              kernel_initializer=initializers.RandomNormal(), acti
vation='relu'),
        Dense(1, kernel_initializer=initializers.RandomNormal())
    ])
    model.compile(loss=MeanSquaredError(), optimizer=optimizers.Ad
am())
    return model

# Set seed and K-Fold cross-validation
seed = 7
np.random.seed(seed)
kfold = KFold(n_splits=10, random_state=seed, shuffle=True)

# Baseline Model Evaluation
baseline = KerasRegressor(model=create_model, epochs=1, batch_size
=5, verbose=0)
baseline_results = cross_val_score(baseline, X, Y, cv=kfold)

# Print results
```

```
print(f"Baseline: {baseline_results.mean():.2f} ({baseline_result
s.std():.2f}) MSE")
```

XOR Problem

```
import random
import math

VARIANCE_W = 0.5

# Initialize weights randomly
w11 = random.uniform(-VARIANCE_W, VARIANCE_W)
w21 = random.uniform(-VARIANCE_W, VARIANCE_W)
b1 = 0

w12 = random.uniform(-VARIANCE_W, VARIANCE_W)
w22 = random.uniform(-VARIANCE_W, VARIANCE_W)
b2 = 0

w13 = random.uniform(-VARIANCE_W, VARIANCE_W)
w23 = random.uniform(-VARIANCE_W, VARIANCE_W)
b3 = 0

o1 = random.uniform(-VARIANCE_W, VARIANCE_W)
o2 = random.uniform(-VARIANCE_W, VARIANCE_W)
o3 = random.uniform(-VARIANCE_W, VARIANCE_W)
ob = 0

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

def sigmoid_prime(x):  # x is already sigmoid-activated
    return x * (1 - x)

def predict(i1, i2):
    s1 = sigmoid(w11 * i1 + w21 * i2 + b1)
    s2 = sigmoid(w12 * i1 + w22 * i2 + b2)
    s3 = sigmoid(w13 * i1 + w23 * i2 + b3)

    output = sigmoid(s1 * o1 + s2 * o2 + s3 * o3 + ob)
    return output
```

```
def learn(i1, i2, target, alpha=0.2):
    global w11, w21, b1, w12, w22, b2, w13, w23, b3
    global o1, o2, o3, ob

    # Forward pass
    s1 = sigmoid(w11 * i1 + w21 * i2 + b1)
    s2 = sigmoid(w12 * i1 + w22 * i2 + b2)
    s3 = sigmoid(w13 * i1 + w23 * i2 + b3)

    output = sigmoid(s1 * o1 + s2 * o2 + s3 * o3 + ob)

    # Error calculation
    error = target - output
    derror = error * sigmoid_prime(output)

    # Backpropagation
    ds1 = derror * o1 * sigmoid_prime(s1)
    ds2 = derror * o2 * sigmoid_prime(s2)
    ds3 = derror * o3 * sigmoid_prime(s3)

    # Update weights and biases
    o1 += alpha * s1 * derror
    o2 += alpha * s2 * derror
    o3 += alpha * s3 * derror
    ob += alpha * derror

    w11 += alpha * i1 * ds1
    w21 += alpha * i2 * ds1
    b1 += alpha * ds1

    w12 += alpha * i1 * ds2
    w22 += alpha * i2 * ds2
    b2 += alpha * ds2

    w13 += alpha * i1 * ds3
    w23 += alpha * i2 * ds3
    b3 += alpha * ds3

# XOR Dataset
INPUTS = [
    [0, 0],
    [0, 1],
    [1, 0],
```

```
    [1, 1]
]

OUTPUTS = [0, 1, 1, 0]

# Training Loop
for epoch in range(1, 10001):
    indexes = [0, 1, 2, 3]
    random.shuffle(indexes)

    for j in indexes:
        learn(INPUTS[j][0], INPUTS[j][1], OUTPUTS[j], alpha=0.2)

    # Print loss every 1000 epochs
    if epoch % 1000 == 0:
        cost = sum((OUTPUTS[j] - predict(INPUTS[j][0], INPUTS[j]
[1])) ** 2 for j in range(4)) / 4
        print(f"Epoch {epoch}, Mean Squared Error: {cost:.6f}")

# Predictions
for i in range(4):
    result = predict(INPUTS[i][0], INPUTS[i][1])
    print(f"For input {INPUTS[i]}, expected {OUTPUTS[i]}, predicte
d {result:.4f}, which is {'correct' if round(result) == OUTPUTS[i]
else 'incorrect'}")
```

MNIST

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.initializers import RandomNormal  # Import R
andomNormal
from tensorflow.keras.activations import relu, softmax  # Import r
elu and softmax
from tensorflow.keras.losses import categorical_crossentropy  # Im
port categorical_crossentropy
from tensorflow.keras.optimizers import Adam  # Import Adam

# Fix random seed for reproducibility
```

```
seed = 7
np.random.seed(seed)

# Load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Flatten 28x28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype(np.
float32)
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype(np.flo
at32)

# Normalize inputs from 0-255 to 0-1
X_train /= 255.0
X_test /= 255.0

# One-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]

# Define baseline model
def baseline_model():
    # Create model
    model = Sequential()

    # Use RandomNormal for weight initialization
    model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initi
alizer=RandomNormal(), activation=relu))
    model.add(Dense(num_classes, kernel_initializer=RandomNormal
(), activation=softmax))

    # Compile model, use imported objects
    model.compile(loss=categorical_crossentropy, optimizer=Adam(),
metrics=['accuracy'])

    return model

# Build the model
model = baseline_model()

# Fit the model
```

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epoc
hs=10, batch_size=200, verbose=2)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100 - scores[1] * 100))
```

https://colab.research.google.com/drive/15AqF_u47wxvstAYzTHNBt9D8VqmH6Tbn?usp=sharing → Classification

https://colab.research.google.com/drive/1rxkAmfjZ8CJW4PQLazUNUWCuMuTMgn49?usp=sharing → ANN

https://colab.research.google.com/drive/1SEy320gzI7NgBqD3qBQwPgVYqRQQNBdJ?usp=sharing → Regression

https://colab.research.google.com/drive/1ezW8Qa3bVPaxXkGT0gIstDQb-aAdx7Sg?usp=sharing → XOR

https://colab.research.google.com/drive/1FTYKdmtnvdBM0rIWVamlOxFjdvwGT4Bs?usp=sharing → MNSIT

Tensorflow

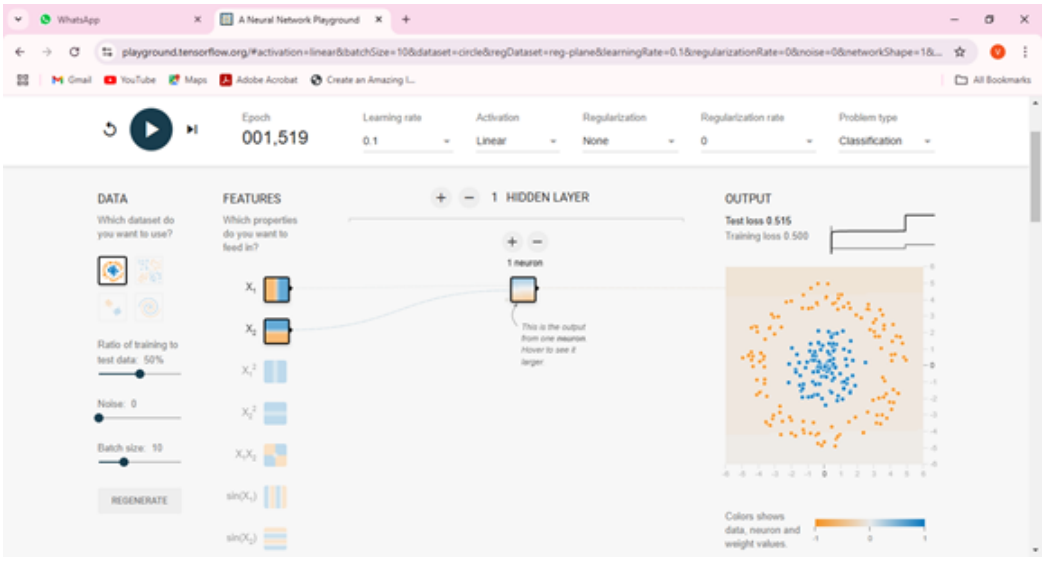
Experimentation and Observations

In this experiment, I used the TensorFlow Playground tool to analyze how the neural network performs with a nonlinear dataset. The initial settings were:

- Dataset: Circular dataset (nonlinear)
- Activation Function: Linear (in initial test)
- Hidden Layers: Single hidden layer with 1 neuron

Task 1: One Neuron with Linear Activation

With a single neuron in the hidden layer and a Linear activation function, the model achieved a test loss of 0.515. This high loss indicates that the model struggles with nonlinear data when only one neuron and a linear activation are used or with just a single neuron using a linear activation function (like a single unit with Linear in PyTorch or no activation in Keras), the model behaves like a simple linear regression model. This means it will only learn linear relationships between the inputs and the output. Running the model with a single neuron and linear activation should confirm that it cannot learn any nonlinearities and may struggle if the data has any nonlinear patterns. The loss is very high, and we say the model **underfits** the data.



Test 2: Two Neurons with ReLU Activation

Next, I increased the hidden layer to 2 neurons and changed the activation to ReLU to introduce nonlinearity. The test loss reduced to 0.268. This significant drop in loss demonstrates that the addition of neurons, coupled with a nonlinear activation function, improves the model's ability to capture the complex, nonlinear patterns in the data.

Or increasing the hidden layer size to 2 neurons and using a ReLU activation allows the model to learn nonlinear patterns in the data. The ReLU activation introduces nonlinearity by setting all negative inputs to zero while allowing positive inputs to pass through. With two neurons and a ReLU activation, the model should perform better in learning nonlinear data patterns, though the complexity of the data will determine if two neurons are enough to model it effectively.

- Does increasing the number of neurons improve the model's ability to learn nonlinearity?

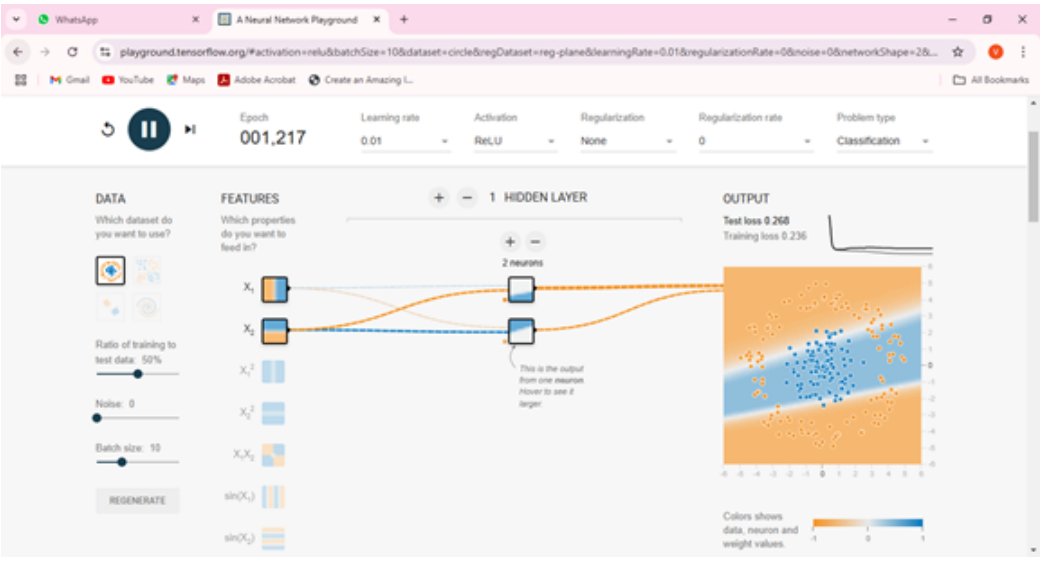
Yes, adding neurons improved the model's performance on the nonlinear dataset. This is shown by the decrease in test loss, as multiple neurons allow the model to better represent the underlying complexity of the data.

- Can the model effectively classify nonlinear data with a nonlinear activation like ReLU?

Absolutely. Using ReLU, the model was able to capture the nonlinear structure, as shown by the improved test loss and the clearer decision boundary in the output visualization

However, a single hidden layer with 2 neurons cannot reflect all the nonlinearities in this data set, and will have high loss even without noise: it still **underfits** the data.

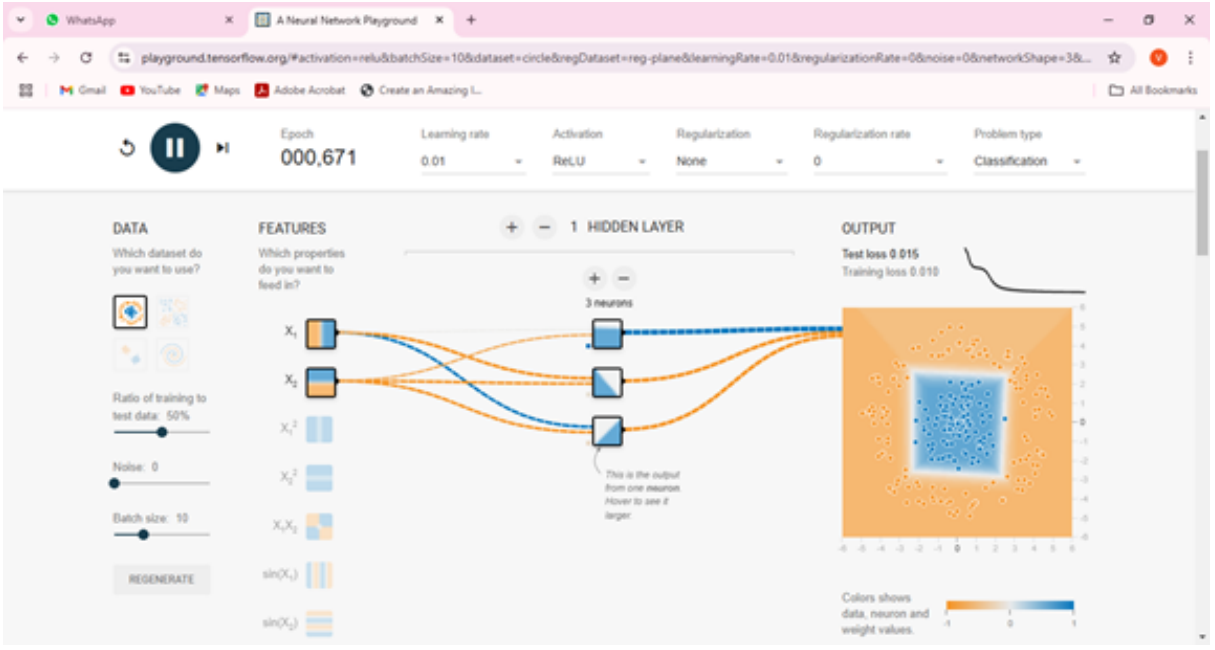
Adding neurons and using a ReLU activation enables the neural network to classify nonlinear data effectively. This experiment highlights the importance of nonlinearity in network structure to handle complex datasets.



Task 3

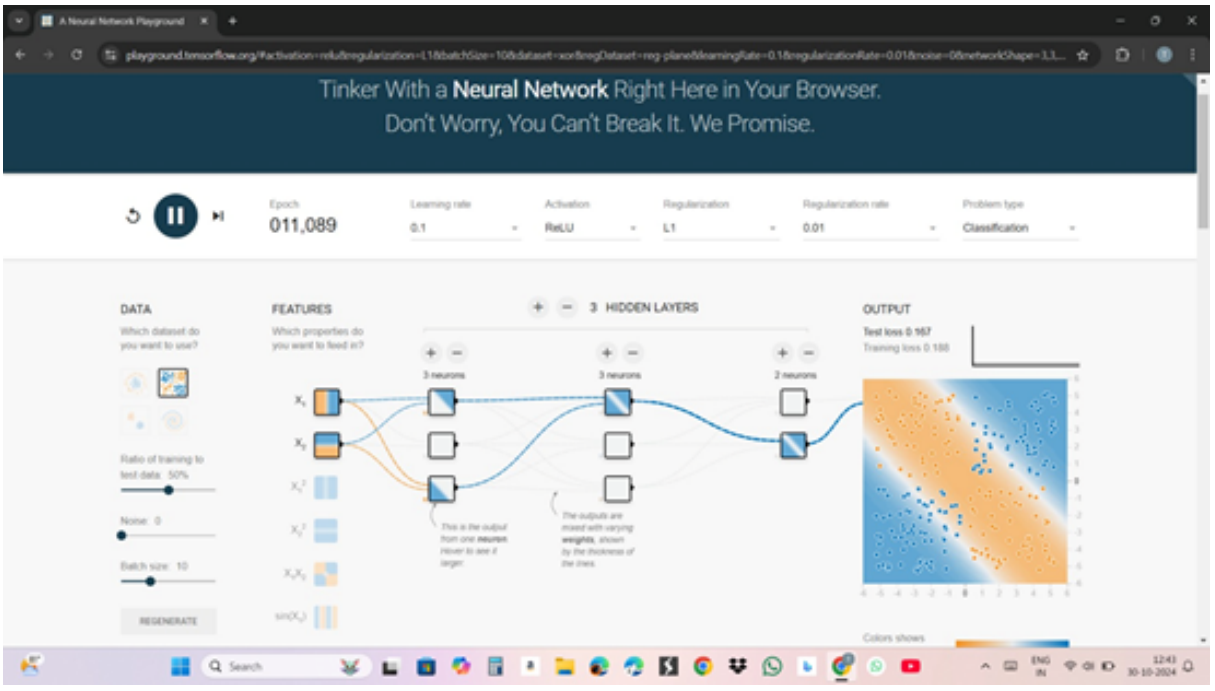
The experiment involves increasing the number of neurons in the hidden layer from 2 to 3 and applying a nonlinear activation function, ReLU, to the model with three neurons in the hidden layer, the model has a larger capacity to learn even more complex nonlinearities. Each additional neuron allows the model to capture more intricate relationships between the input features and the output. However, because the weights are often initialized randomly, you might observe some variability in the model quality from one run to the next. This is due to the random starting points and the potential for getting stuck in local minima during training.

Observations: Increasing the neurons and applying ReLU allowed the model to capture nonlinear patterns more effectively. This change resulted in a lower test loss, indicating improved generalization capability.



Task 4

Experimentation by varying the number of hidden layers, neurons per layer, and activation functions. The configuration includes:- First hidden layer with 3 neurons- Second hidden layer with 3 neurons- Third hidden layer with 2 neuronsAdditionally, regularization was applied with an L1 rate of 0.01 and learning rate of 0.01 This setup aims to find the smallest model size with a test loss of 0.277 or lower.



Observations: The model with three hidden layers achieved a test loss below 0.277, indicating an effective fit with a relatively compact architecture. This configuration shows that adding layers and neurons can improve the model's ability to capture complex patterns while still maintaining low test loss.

1. Adding More Layers and Neurons: Adding a second and third layer, each with 3, 3, and 2 neurons, respectively, should improve the model's ability to capture complex patterns by deepening the network and increasing its capacity.
2. Convergence Speed and Model Quality: Increasing the model size (more layers and neurons) can indeed improve convergence speed as it allows the model to find the optimal solution faster, given enough training data and iterations. However, a larger model may also lead to overfitting if it becomes too complex for the data.
3. Smallest Model for Target Loss: Experimentation will reveal the smallest configuration that achieves a test loss of 0.277. Often, a simpler architecture like two hidden layers with around 2-4 neurons each, using ReLU, can achieve a good fit if the data is not too complex.

I am summarizing as

- A single neuron with linear activation cannot learn nonlinearities.

- Adding neurons and using nonlinear activation like ReLU improves the model’s ability to learn nonlinear patterns.
- Multiple layers and neurons further enhance the model’s complexity and can reduce test loss, though too large a model can overfit.
- The model’s performance can vary slightly between runs due to random initialization, which affects the convergence quality.

I am willing to conclude as by observing Task3 and Task4

According to my observation Task 3 and Task 4 reveal that increasing model complexity, either by adding neurons or layers, generally improves the model's ability to fit complex data. However, adding regularization and adjusting learning rates are also important to prevent overfitting. The configuration with three hidden layers achieved a test loss below 0.277, demonstrating an optimal balance between model complexity and performance.

Problem 5: Neural Net Initialization

This problem explores the importance of weight initialization in training a neural network on the XOR dataset. XOR is a non-linearly separable problem, making it a good test case for neural networks.

Task 1: Running the Model Multiple Times with Different Initializations

Observations:

1. When running the neural network multiple times, each with a fresh initialization, the final decision boundary differs significantly.
2. The decision boundary typically converges to a **diagonal shape**, forming two distinct regions that separate the XOR classes.
3. However, the specific positioning and orientation of the decision boundary vary from run to run.
4. Some runs achieve a **smooth** and well-separated boundary, while others have **irregular decision boundaries**, which sometimes fail to generalize well.

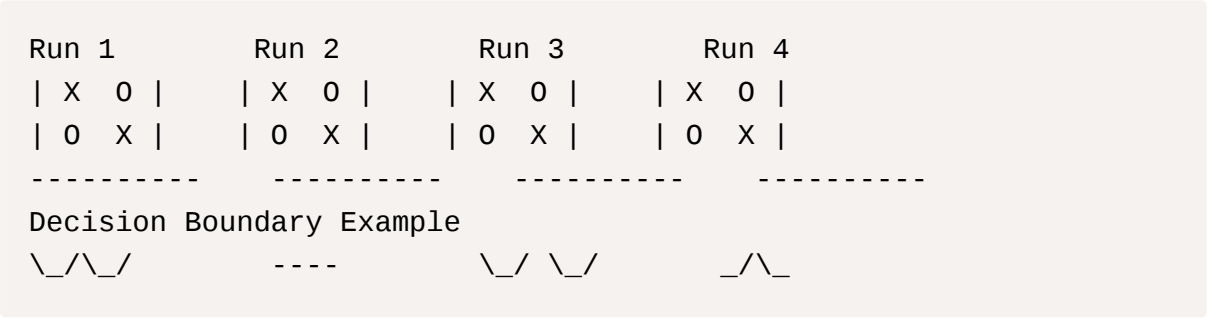
What This Says About Initialization in Non-Convex Optimization

- Neural networks train using **gradient descent**, which optimizes a highly non-convex loss function. Because of this:
    - The optimization process can get stuck in different **local minima** depending on the initial weights.
    - Different weight initializations lead to different **convergence paths** and **final solutions**.
    - Poor initialization can slow down training or cause the model to converge to **suboptimal solutions**.

Diagram Representation

Below is a conceptual representation of how different initializations lead to different decision boundaries:

Figure 1: Different Decision Boundaries Due to Initialization



- "X" and "O" represent different classes.
- The decision boundaries vary in shape, showing how initialization impacts training.

Problem 6: Increasing Model Complexity

This problem explores whether adding extra layers and neurons improves the stability of training.

Task 2: Adding a Layer and Extra Nodes

Observations:

1. Increasing the model complexity (adding an extra hidden layer and a few more neurons) **generally improves stability** in training.
2. The decision boundaries are more **consistent** across multiple runs.
3. While variations still exist, the network is more **likely** to converge to a good solution.
4. The decision boundary appears more **smoother and well-formed**, better capturing the XOR separation.

Why Does Extra Complexity Improve Stability?

- **More neurons and layers provide greater representational capacity**, allowing the model to fit the XOR pattern more effectively.
- **Weight initialization becomes less sensitive** as more neurons provide redundancy, reducing the impact of poor initial weights.
- A **deeper network smooths the optimization landscape**, reducing the risk of getting stuck in poor local minima.

Diagram Representation

Below is a conceptual representation of how increasing complexity improves stability.

Figure 2: Decision Boundaries with More Complexity

```
Run 1        Run 2        Run 3        Run 4
| X  O |    | X  O |    | X  O |    | X  O |
| O  X |    | O  X |    | O  X |    | O  X |
----------    ----------    ----------    ----------
More Stable Boundaries
 /---\       /---\       /---\       /---\
```

- The additional complexity results in more **stable** and **consistent** decision boundaries across runs.

Conclusion

1. **Initialization plays a crucial role** in neural network training, especially in non-convex problems like XOR.
2. **Different initializations can lead to vastly different decision boundaries**, affecting model performance.
3. **Adding layers and neurons improves stability**, making training less sensitive to initialization.
4. **While variations still exist, deeper networks provide more reliable decision boundaries**.

Would you like me to generate actual visual plots using Python for a simulated XOR classification experiment?
