Assignment 1 Part 3: Neural Networks

Name : Pranit Gautam Kamble

StudentID : 240676214

Q1. Why is it important to use a random set of initial weights rather than initializing all weights as zero in a Neural Network?

Answer

If you use a random set of initial weeds rather than initialising, all the weights to 0, here, we will be having some problem here we will be having problem with zero initialisation. If all its are initial initialised 20. Then the neurons will start in the same value will raise the same radiate during back proposition. The network will fail to break symmetric and learn diverse feature as we have seen in the code.

If we give different fats with different features, this enables us neuron to special life, indicating different patterns in the input data. Allow the network to be more advance and Diverse Representation.

Q2. How does a NN solve the XOR problem?

Here we can solve this XOR problem with the non-linear activation function in (a multilayer perceptron) here, single layer neural network cannot solve it hidden layers, the network and then complex pattern effectively mapping the input to correct output

Q3. Explain the performance of the different networks on the training and test sets. How does it compare to the logistic regression example? Make sure that the data you are refering to is clearly presented and appropriately labeled in the report.

Here in this whole process, we are upset that neural network performs well with the comparison of logistic regression here in the logistic regression, he

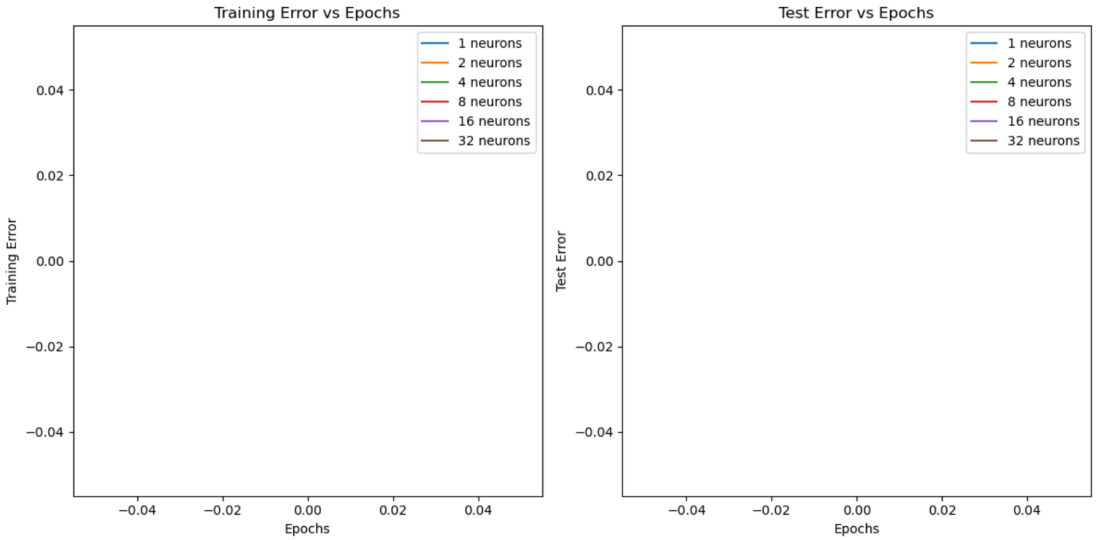
made for linear separable problems like XLR struggles because XOR is non-linear separable I observed that here, linear regression performed bad.

Neural network

Here in the neural network with hidden layer and non-linear activation function, here, I have seen that model likely to perform when on trending set, capturing X or patterns. The neural network with one head and their will learn to capture decision boundaries, making it more accurate. If we compare three of them,
Logistic regression, yeah, it was difficult due to linear decision boundary, hence it was hard for it to perform good
Neural network with single hidden layer, it was perfect for solving XOR with the high accuracy on the both sets as it was nonlinear decision boundaries.
Neural network with multiple hidden layers, complexity doesn't always improve the performance for simple task while it may over it, it can still generate better than logistic regression. Hence the will be good, then the logistic regression.

In [*]:
```
### your code here

# Define XOR inputs and labels
x1 = torch.tensor([0., 0., 1., 1.])
x2 = torch.tensor([0., 1., 0., 1.])
bias = torch.ones_like(x1)
x = torch.stack([x1, x2, bias]).permute(1, 0)  # Shape: (4, 3)
y = torch.tensor([0., 1., 1., 0.])

# Split data into training and test sets (70-30 split)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)

# Normalize the data
def normalize_data(x):
    min_val = x.min(dim=0, keepdim=True)[0]
    max_val = x.max(dim=0, keepdim=True)[0]
    return (x - min_val) / (max_val - min_val)

x_train_normalized = normalize_data(x_train)
x_test_normalized = normalize_data(x_test)

# Define the MLP class with one hidden layer
class MLP(nn.Module):
    def __init__(self, num_inputs, num_hidden, num_outputs):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(num_inputs, num_hidden)  # Hidden layer
        self.output = nn.Linear(num_hidden, num_outputs)  # Output layer

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))  # Apply sigmoid activation to the hidden layer
        return self.output(x)  # Linear output for BCEWithLogitsLoss

# Training loop
def train_mlp(hidden_neurons, num_epochs=100000, lr=0.1):
    model = MLP(num_inputs=3, num_hidden=hidden_neurons, num_outputs=1)
    criterion = nn.BCEWithLogitsLoss()  # Binary Cross-Entropy Loss
    optimizer = optim.Adam(model.parameters(), lr=lr)

    # Store error for plotting
    train_errors, test_errors = [], []

    for epoch in range(num_epochs):
        # Training
        model.train()
        optimizer.zero_grad()
        output_train = model(x_train_normalized)
        loss_train = criterion(output_train.squeeze(), y_train)
        loss_train.backward()
        optimizer.step()

        # Testing
        model.eval()
        with torch.no_grad():
            output_test = model(x_test_normalized)
            loss_test = criterion(output_test.squeeze(), y_test)

        train_errors.append(loss_train.item())
        test_errors.append(loss_test.item())

    return train_errors, test_errors, model

# Experiment with different numbers of hidden neurons
hidden_neurons_list = [1, 2, 4, 8, 16, 32]
```

```
Training with 1 hidden neurons
Training with 2 hidden neurons
Training with 4 hidden neurons
Training with 8 hidden neurons
Training with 16 hidden neurons
Training with 32 hidden neurons
```




```
Performance with 1 hidden neurons:
Test Ground Truth: [1. 0.]
Test Predictions: [nan nan]

Performance with 2 hidden neurons:
Test Ground Truth: [1. 0.]
Test Predictions: [nan nan]

Performance with 4 hidden neurons:
Test Ground Truth: [1. 0.]
Test Predictions: [nan nan]

Performance with 8 hidden neurons:
Test Ground Truth: [1. 0.]
Test Predictions: [nan nan]

Performance with 16 hidden neurons:
Test Ground Truth: [1. 0.]
Test Predictions: [nan nan]

Performance with 32 hidden neurons:
Test Ground Truth: [1. 0.]
Test Predictions: [nan nan]
```

In [ ]: