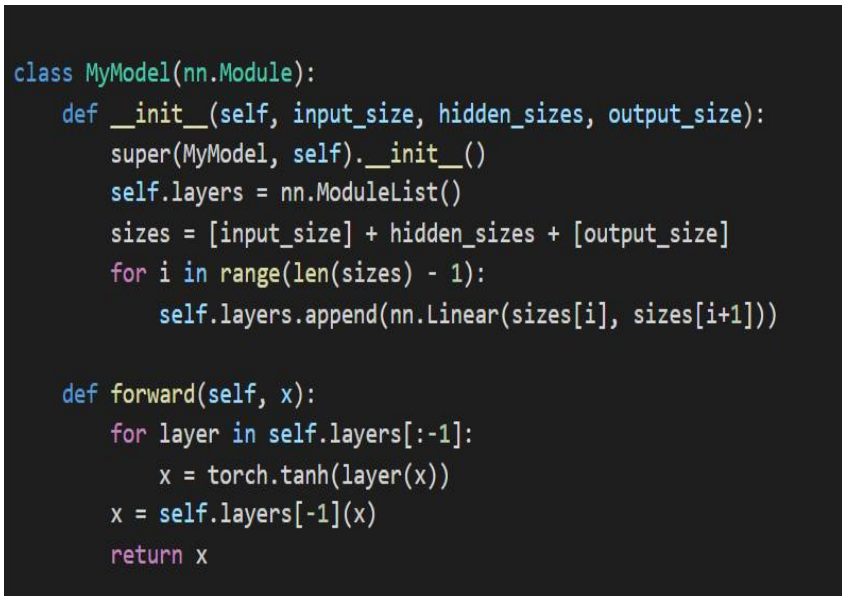
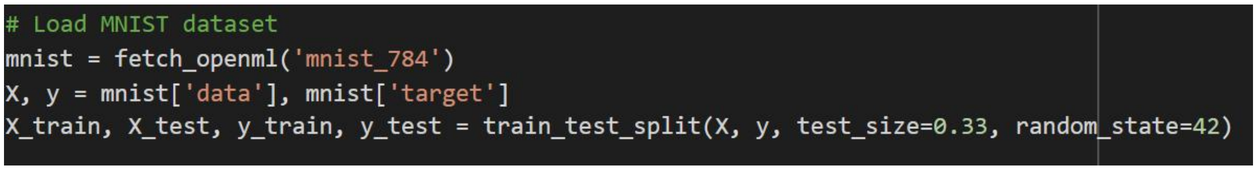
Neural Network



```python
class MyModel(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(MyModel, self).__init__()
        self.layers = nn.ModuleList()
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            self.layers.append(nn.Linear(sizes[i], sizes[i+1]))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.tanh(layer(x))
        x = self.layers[-1](x)
        return x
```

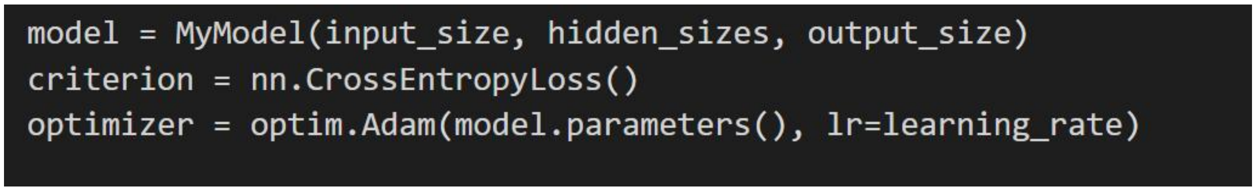
1) Here we implemented the Artificial Neural Network (ANN) model using the pytorch library.
2) In the above pic we have used tanh as the activation function between the input-hidden-output layers.
3)



```python
# Load MNIST dataset
mnist = fetch_openml('mnist_784')
X, y = mnist['data'], mnist['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

-Here we used the train-test-split of size 0.33 as the testing size.

4)

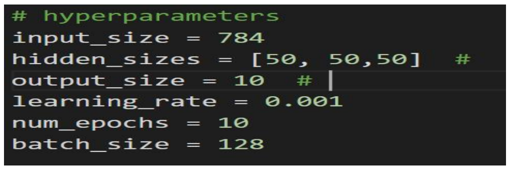


```python
model = MyModel(input_size, hidden_sizes, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

-Have used Cross-entropy loss as the loss criterion.

-Have used Adam Optimizer which is an optimization algorithm used for stochastic gradient descent (SGD) Optimization.

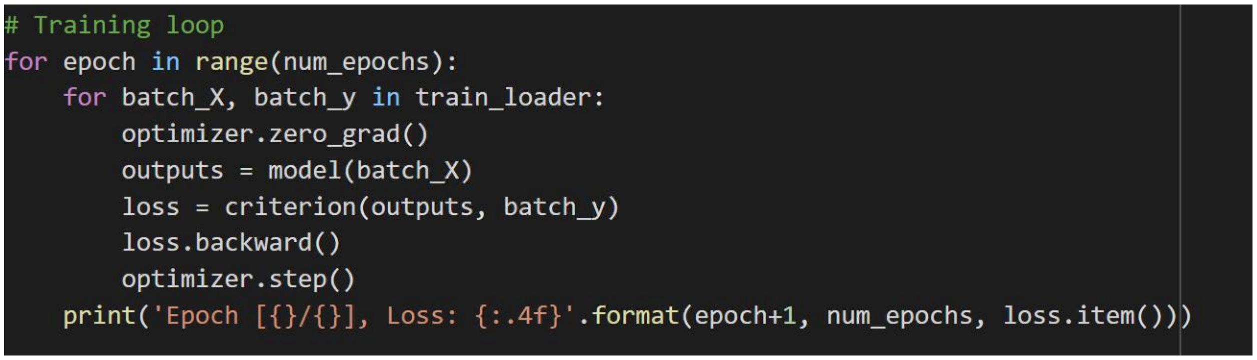
5) Have altered the hyper parameter- hidden layer sizes and number of hidden layer to check the variations of the model with respect to different values of hyper parameter.



```python
# hyperparameters
input_size = 784
hidden_sizes = [50, 50,50]   #
output_size = 10   #
learning_rate = 0.001
num_epochs = 10
batch_size = 128
```

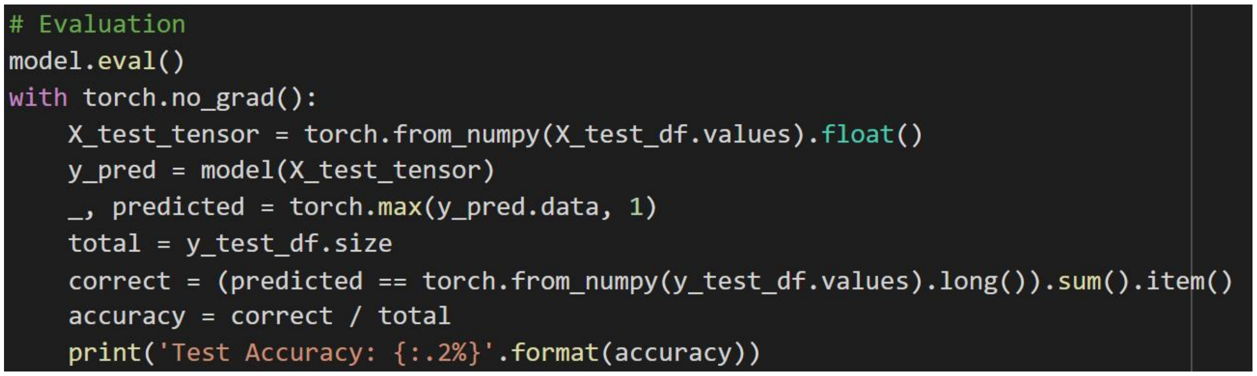
6) **Model Training :**

Within each mini-batch iteration, the optimizers gradients are reset to zero using optimizer.zero_grad() to prevent gradient accumulation. The model's forward pass is then computed using model(batch_X) , which applies the neural network model to the input features to generate output predictions. The loss between the predicted outputs and the ground truth labels is computed using a loss function defined as "criterion, and the gradients are computed during the backward pass using loss Backward() . Finally, the optimizers "step() " function is called to update the model's parameters based on the computed gradients and the chosen optimization algorithm (in this case, Adam).



```
# Training loop
for epoch in range(num_epochs):
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

7) **Model Evaluation:**



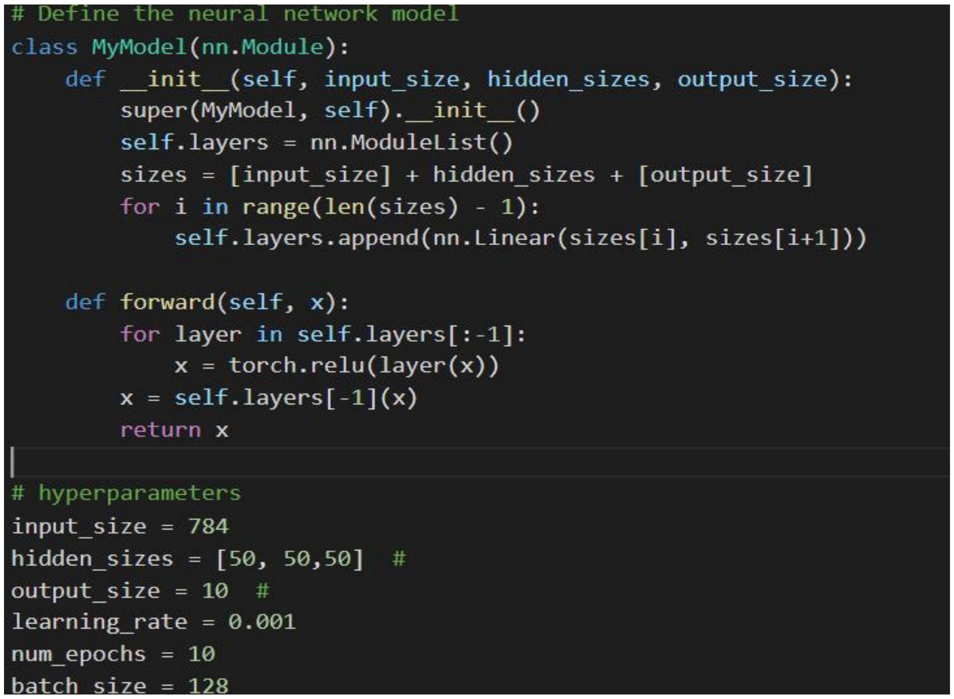
```
# Evaluation
model.eval()
with torch.no_grad():
    X_test_tensor = torch.from_numpy(X_test_df.values).float()
    y_pred = model(X_test_tensor)
    _, predicted = torch.max(y_pred.data, 1)
    total = y_test_df.size
    correct = (predicted == torch.from_numpy(y_test_df.values).long()).sum().item()
    accuracy = correct / total
    print('Test Accuracy: {:.2%}'.format(accuracy))
```

First, the model is put into evaluation mode using model.eval() . This is important as it disables any layers that might behave differently during training, such as dropout or batch normalization, and ensures that the model is ready for inference. Next, a torch.no_grad() block is used to disable gradient computation, which helps to reduce memory usage and computation time during evaluation, as gradients are not needed for inference. The test data, represented by "X_test_df" and "y_test_df", are converted to PyTorch tensors using "torch. from_numpy() ~ and cast to the appropriate data type ("float forinput features and "long " for target labels).

The model is then applied to the test data using model1(X_test_tensor) to generate predicted outputs. The predicted outputs are then used to compute the accuracy of the model's predictions.

8) **Conclusion :**

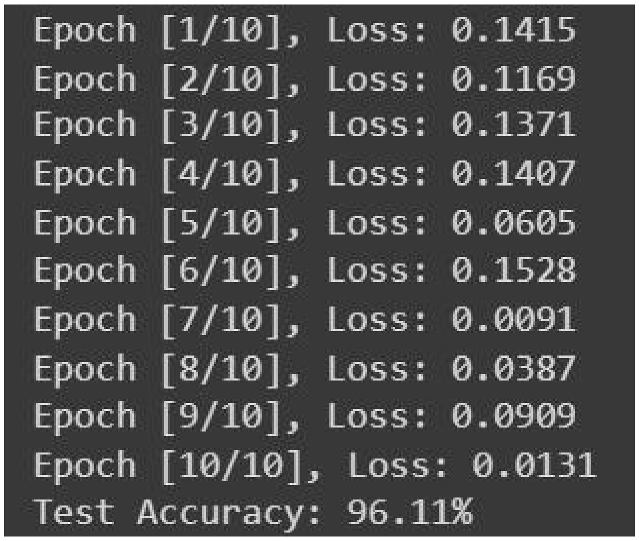
The model with three hidden layers of size 50,50,50 with an activation function of ReLu gives the highest accuracy among all other combinations.



```
# Define the neural network model
class MyModel(nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(MyModel, self).__init__()
        self.layers = nn.ModuleList()
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            self.layers.append(nn.Linear(sizes[i], sizes[i+1]))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = torch.relu(layer(x))
        x = self.layers[-1](x)
        return x

# hyperparameters
input_size = 784
hidden_sizes = [50, 50,50]  #
output_size = 10  #
learning_rate = 0.001
num_epochs = 10
batch_size = 128
```



```
Epoch [1/10], Loss: 0.1415
Epoch [2/10], Loss: 0.1169
Epoch [3/10], Loss: 0.1371
Epoch [4/10], Loss: 0.1407
Epoch [5/10], Loss: 0.0605
Epoch [6/10], Loss: 0.1528
Epoch [7/10], Loss: 0.0091
Epoch [8/10], Loss: 0.0387
Epoch [9/10], Loss: 0.0909
Epoch [10/10], Loss: 0.0131
Test Accuracy: 96.11%
```

X--X