DEEP LEARNING PROJECT

LE Thi Hoa - p2310380

Wednesday, 15 November 2023

Part 1 : Perceptron Indicate the size of each tensor of the provided file perceptron_pytorch.py. Explain.

In file perceptron_pytorch.py, we have the tensors:

- "*data_train*" is the training data, which is a tensor containing training samples from the MNIST dataset. The size of "data_train" is (number of training samples, number of features).
    - The size of tensor data_train = torch.Size([63000, 784])
- "*label_train*" is the corresponding label for the training data. The size of "label_train" is (number of training samples, number of classes).
    - The size of tensor label_train = torch.Size([63000, 10])
- "*data_test"* is the testing data, containing test samples from the MNIST dataset. The size of "data_test" is (number of testing samples, number of features)
    - The size of tensor data_test = torch.Size([7000, 784])
- *"Label_test"* is the corresponding label for the testing data. The size of "label_test" is (number of testing samples, number of classes)
    - The size of tensor label_test = torch.Size([7000, 10])
- "*w*" is a tensor that contains the model's weights. The size of "w" is (the number of features, the number of classes). It is initialized with random values in the range from -0.001 to 0.001.
    - The size of tensor w = torch.Size([784, 10])
- *"b"* is a tensor that contains bias values of the model. The size of "b" is (1, number of classes), and it is also initialized with random values in the range from -0.001 to 0.001.
    - The size of tensor b = torch.Size([1, 10])

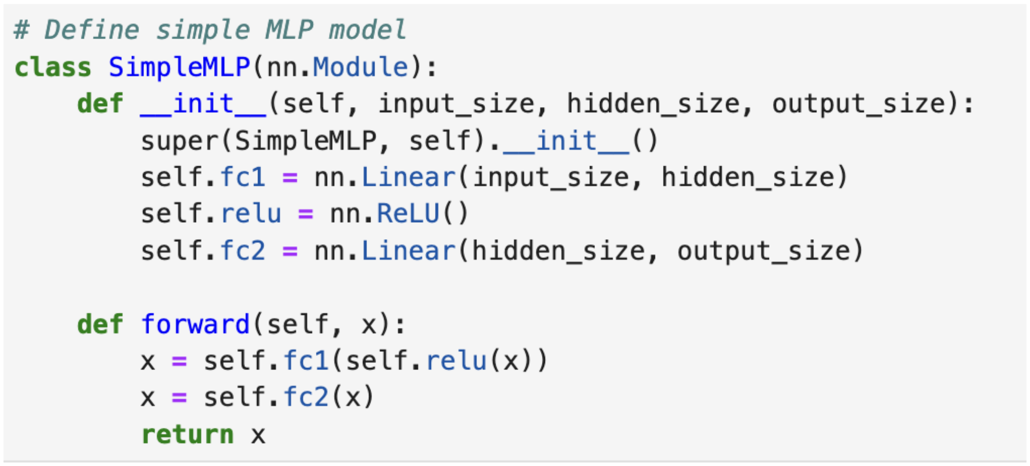
Part 2 : Shallow network In this part, you will implement a MLP with only one hidden layer and a linear output layer.

My code: Link github:

https://github.com/LeHoa98ptit/DeepLearning-Project-M2/blob/main/Report_LE_Thi_Hoa_ver_2.ipynb

File: Report_LE_Thi_Hoa_ver_2.ipynb

The model is built as a simple feedforward neural network with: an input layer, a hidden layer, and an output layer.



```python
# Define simple MLP model
class SimpleMLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(self.relu(x))
        x = self.fc2(x)
        return x
```

I use the ReLU activation function.

Hidden_size: Controls the model's capacity to capture complex patterns. A larger hidden_size can lead to better performance but may result in overfitting if not tuned properly.

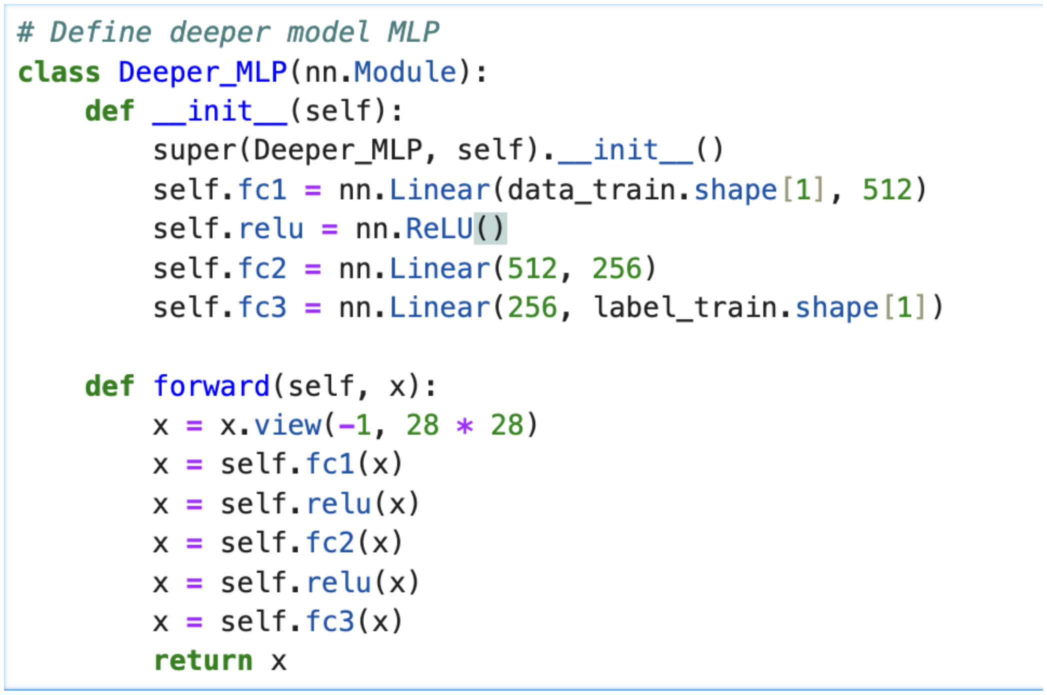
Learning_rate: Determines the training speed and stability. The choice of learning rate affects the convergence and optimization process. Selecting an appropriate learning rate is necessary for successful training.

So, I try different values of learning_rate and hidden_size to find the best hidden_size and learning_rate.

The code for the results: Best Hidden Size: **64**, Best Learning Rate: **0.001**, Best Accuracy: **86.1429**

Part 3 : Deep network

Model Deeper_MLP - It is a multi-layer perceptron (MLP) model with three fully connected layers. The layers in the model are defined as follows:



```python
# Define deeper model MLP
class Deeper_MLP(nn.Module):
    def __init__(self):
        super(Deeper_MLP, self).__init__()
        self.fc1 = nn.Linear(data_train.shape[1], 512)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, label_train.shape[1])

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x
```

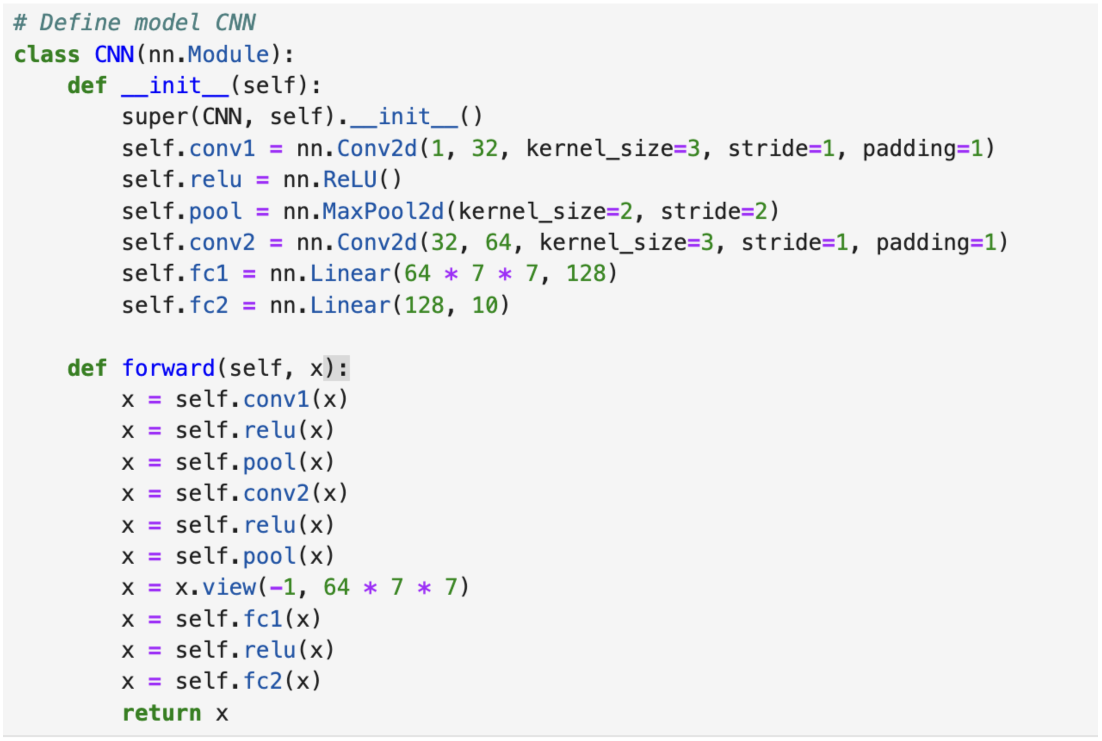
- fc1: Input layer with data_train.shape[1] input features and 512 output features.
- relu: ReLU activation function.
- fc2: Hidden layer with 512 input features and 256 output features.
- fc3: Output layer with 256 input features and label_train.shape[1] output features.
- The forward method is responsible for the forward pass of the model. It takes an input x, reshapes it to a 2D tensor, and passes it through the layers in the defined sequence, applying ReLU activation after each fully connected layer. The final output of the model is returned.

I use ReLU activation function

Similar to part 2, I try several learning_rate values to find the best value.

Best Learning Rate: **0.001**, Best Accuracy: **98.41%**

Part4: CNN: Implement a CNN architecture (more adapted to images).



```
# Define model CNN
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(-1, 64 * 7 * 7)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

The layers in the model are defined as follows:

- *conv1:* Convolutional layer with 1 input channel, 32 output channels, a kernel size of 3x3, stride of 1, and padding of 1
- *relu*: ReLU activation function.
- *pool:* Max pooling layer with a kernel size of 2x2 and stride of 2.
- *conv2:* Convolutional layer with 32 input channels, 64 output channels, a kernel size of 3x3, stride of 1, and padding of 1.
- *fc1*: Fully connected layer with an input size of 64x7x7 (output size from the previous convolutional layer) and an output size of 128.
- *fc2*: Fully connected layer with an input size of 128 and an output size of 10 (number of classes).
- The forward method is responsible for the forward pass of the model. It takes an input x, passes it through the convolutional layers with ReLU activation and max pooling, reshapes it to a 1D tensor, and passes it through the fully connected layers. The final output of the model is returned.