Lab 03

202238002 Changsub Yun

1. **Define NNs with 5 layers. Compare performance with/without weight regularization and dropout.**

- Code explanation

Define NNs with 5 layers >
class Net(nn.Module): Different neural network architectures are defined using PyTorch's nn.

Define Hyperparameters >
device = The device to run the training on, it can be "cpu" or a specific GPU device denoted by a number (e.g., "0" for the first GPU).
model_type = Specifies the architecture of the neural network to use, can be “Net” or “Net_dropout”.
BATCH_SIZE = The batch size for training the neural network.
LR = The learning rate for the optimizer.
EPOCH = The number of epochs to train the model.
WD = Setting Weight Regularization. (As the value of weight_decay increases, the weight value becomes smaller, which can solve the overfitting phenomenon. However, if the weight_decay value is too large, the underfitting phenomenon occurs, so an appropriate value must be used.)

Load MNIST dataset >
Load train and test MNIST data. using batch size 100 (Use a small subset of the dataset to encourage overfitting)

Define Train >
This loop iterates over the dataset for a specified number of epochs. An epoch is one complete forward and backward pass of all the training examples. The following step of code evaluate the model on the test data without computing gradients to save memory and computation. After each epoch, the average training and validation losses are printed. Accuracy is calculated in the evaluate the model section. Loss function is CrossEntropyLoss and optimizer is SGD.

- Result report

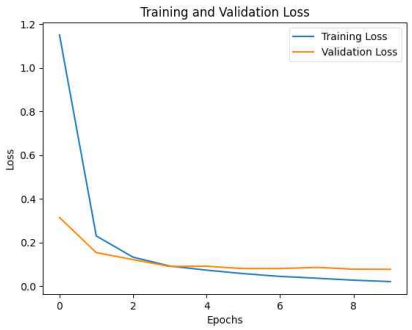
1. without weight regularization and dropout >

```
Net(
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=400, bias=True)
  (fc3): Linear(in_features=400, out_features=300, bias=True)
  (fc4): Linear(in_features=300, out_features=200, bias=True)
  (fc5): Linear(in_features=200, out_features=10, bias=True)
)
```

Epoch 1, Training loss: 1.1501556520660718, Validation loss: 0.31388237530365587, accuracy: 90.83%
Epoch 2, Training loss: 0.22961427450180052, Validation loss: 0.15315966428257524, accuracy: 95.57%
Epoch 3, Training loss: 0.13231080531763534, Validation loss: 0.12119127460056917, accuracy: 96.34%
Epoch 4, Training loss: 0.09166157468687743, Validation loss: 0.08977456093067303, accuracy: 97.41%
Epoch 5, Training loss: 0.07285196819963555, Validation loss: 0.09084834895795212, accuracy: 97.18%
Epoch 6, Training loss: 0.056969769747617346, Validation loss: 0.08010049454227555, accuracy: 97.61%
Epoch 7, Training loss: 0.044180446781295665, Validation loss: 0.07997091331490083, accuracy: 97.71%
Epoch 8, Training loss: 0.03595338291081134, Validation loss: 0.08569464960601181, accuracy: 97.48%
Epoch 9, Training loss: 0.02685114326052523, Validation loss: 0.07720756345428527, accuracy: 97.73%
Epoch 10, Training loss: 0.020610496335624097, Validation loss: 0.07679608027086943, accuracy: 97.86%

resutl : In machine learning, overfitting means learning the training data too well. In general, training data is usually a subset of actual data. Therefore, as shown in the graph below, there may be a point where the error decreases for the training data, but the error increases for the actual data.

If we look at overfitting from this perspective, overfitting is a phenomenon in which the error in the actual data increases due to excessive learning on the training data. For example, a phenomenon similar to overfitting where a person who learned the characteristics of a cat by looking at a yellow cat fails to recognize it as a cat when seeing a black or white cat.



2. with weight regularization and dropout >

```
Net_dropout(
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=400, bias=True)
  (fc3): Linear(in_features=400, out_features=300, bias=True)
  (fc4): Linear(in_features=300, out_features=200, bias=True)
  (fc5): Linear(in_features=200, out_features=10, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

Epoch 1, Training loss: 2.055663070480029, Validation loss: 1.2375012958049774, accuracy: 53.70%

Epoch 2, Training loss: 0.8271718637645245, Validation loss: 0.5674089422821998, accuracy: 82.89%

Epoch 3, Training loss: 0.47817958794534204, Validation loss: 0.3982996094971895, accuracy: 89.26%

Epoch 4, Training loss: 0.3885335473219554, Validation loss: 0.34427203588187694, accuracy: 90.78%

Epoch 5, Training loss: 0.34273793049156664, Validation loss: 0.32229476045817135, accuracy: 91.48%

Epoch 6, Training loss: 0.3221699864168962, Validation loss: 0.310721049644053, accuracy: 91.87%

Epoch 7, Training loss: 0.3063480562965075, Validation loss: 0.3085976406186819, accuracy: 91.92%

Epoch 8, Training loss: 0.29463990883280833, Validation loss: 0.27680017441511157, accuracy: 92.47%

Epoch 9, Training loss: 0.28853324268013236, Validation loss: 0.2881440767645836, accuracy: 92.20%

Epoch 10, Training loss: 0.2812580600256721, Validation loss: 0.2882515605539083, accuracy: 92.03%

result:

Weight Regularization:

Reason for Use:

Prevent Overfitting: Weight regularization introduces a penalty on the magnitude of the weights in the neural network. By doing this, it discourages the model from fitting the training data too closely, which can lead to overfitting.

Simplicity: Regularization tends to push the weights towards smaller values, leading to a simpler model. A simpler model is less likely to overfit to the training data.

Types:

L1 Regularization: Adds a penalty proportional to the absolute value of the weights.

L2 Regularization: Adds a penalty proportional to the square of the weights.

Potential Decrease in Accuracy:

Training Accuracy: Regularization might reduce the training accuracy because the model is restricted from fitting the training data too closely.

Generalization: However, while training accuracy might decrease, the model's ability to generalize to new, unseen data (test data) might improve.

Dropout:

Reason for Use:

Random Deactivation: Dropout randomly deactivates a fraction of neurons during training. This means that during each training iteration, a different set of neurons is used.

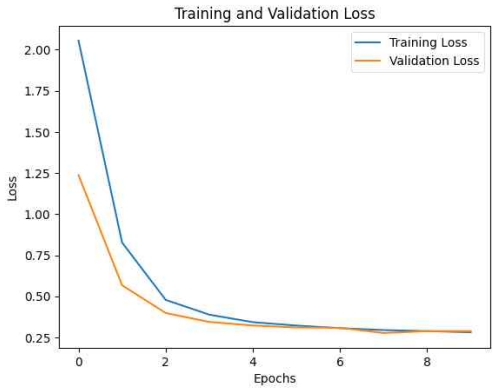
Prevent Co-adaptation: By randomly dropping out neurons, dropout prevents neurons from co-adapting too much. This means that no single neuron relies too heavily on the output of another neuron.

Ensemble Effect: Dropout can be thought of as training a pseudo-ensemble of neural networks. The final model is an average of these networks, leading to a more robust model.

Potential Decrease in Accuracy:

Training Accuracy: Because dropout deactivates neurons randomly during training, the training accuracy might be lower.

Test Accuracy: However, during testing (or inference), all neurons are used (dropout is turned off). This often results in better test accuracy compared to a model trained without dropout.



2. **Define NNs with 7 layers. Compare performance with different initializations such as random normal and He initializations. You can try other initializations as well.**

- Code explanation

Define NNs with 7 layers >
class SevenLayerNN(nn.Module): neural network architectures is defined using PyTorch's nn.

Define Weight Initializations >
Define three different Weight Initializations functions. (random normal, he initialization, xavier initialization)

Define Hyperparameters >
device = The device to run the training on, it can be "cpu" or a specific GPU device denoted by a number (e.g., "0" for the first GPU).
BATCH_SIZE = The batch size for training the neural network.
LR = The learning rate for the optimizer.
EPOCH = The number of epochs to train the model.

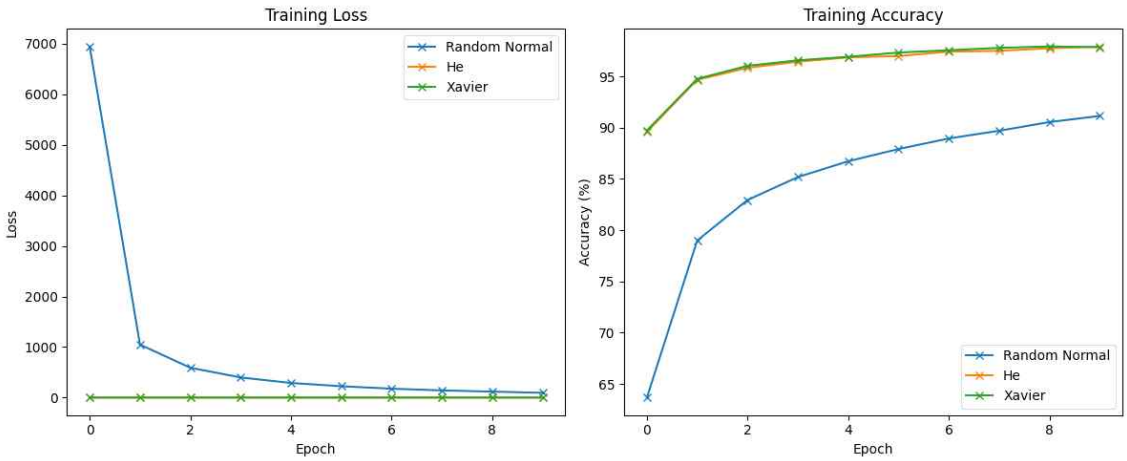
Load MNIST dataset >
Load train and test MNIST data.(transformation normalizes the tensor image with a given mean and standard deviation. In this case, both the mean and standard deviation are set to 0.5)

Define Train >
This loop iterates over the dataset for a specified number of epochs. An epoch is one complete forward and backward pass of all the training examples. After each epoch, the average training loss and accuracy are printed.

main loop >
Learn using 3 different initializations and output loss and accuracy. Loss function is CrossEntropyLoss and optimizer is Adam.

- Result report

```
SevenLayerNN(
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=400, bias=True)
  (fc3): Linear(in_features=400, out_features=300, bias=True)
  (fc4): Linear(in_features=300, out_features=200, bias=True)
  (fc5): Linear(in_features=200, out_features=100, bias=True)
  (fc6): Linear(in_features=100, out_features=50, bias=True)
  (fc7): Linear(in_features=50, out_features=10, bias=True)
)
Random Normal
Epoch 1, Training loss: 6946.857758561198, Accuracy: 63.66%
Epoch 2, Training loss: 1047.8455708007812, Accuracy: 79.00%
Epoch 3, Training loss: 589.8534116699219, Accuracy: 82.95%
Epoch 4, Training loss: 398.2370047566732, Accuracy: 85.19%
Epoch 5, Training loss: 288.8685953328451, Accuracy: 86.73%
Epoch 6, Training loss: 223.92520488688152, Accuracy: 87.92%
Epoch 7, Training loss: 175.60352443440755, Accuracy: 88.95%
Epoch 8, Training loss: 141.64729008382162, Accuracy: 89.70%
Epoch 9, Training loss: 117.43682438354492, Accuracy: 90.55%
Epoch 10, Training loss: 94.32807568359375, Accuracy: 91.15%
He
Epoch 1, Training loss: 0.005197410242445767, Accuracy: 89.60%
Epoch 2, Training loss: 0.002681199097291877, Accuracy: 94.68%
Epoch 3, Training loss: 0.0020954033503658136, Accuracy: 95.85%
Epoch 4, Training loss: 0.0017783049438575593, Accuracy: 96.45%
Epoch 5, Training loss: 0.0015700201970835527, Accuracy: 96.86%
Epoch 6, Training loss: 0.0015161304803487535, Accuracy: 97.00%
Epoch 7, Training loss: 0.0012855040827028765, Accuracy: 97.43%
Epoch 8, Training loss: 0.0012424013406135298, Accuracy: 97.51%
Epoch 9, Training loss: 0.0010740734415070619, Accuracy: 97.76%
Epoch 10, Training loss: 0.0010172352959401906, Accuracy: 97.92%
Xavier
Epoch 1, Training loss: 0.005078459588966022, Accuracy: 89.73%
Epoch 2, Training loss: 0.0026347343685726326, Accuracy: 94.77%
Epoch 3, Training loss: 0.002039169267145917, Accuracy: 96.04%
Epoch 4, Training loss: 0.0017488827089779078, Accuracy: 96.58%
Epoch 5, Training loss: 0.0015693913870607503, Accuracy: 96.93%
Epoch 6, Training loss: 0.0013781544770036513, Accuracy: 97.34%
Epoch 7, Training loss: 0.001280579921467385, Accuracy: 97.56%
Epoch 8, Training loss: 0.0011306551373757732, Accuracy: 97.79%
Epoch 9, Training loss: 0.0010654866560842492, Accuracy: 97.94%
Epoch 10, Training loss: 0.0010468025361469093, Accuracy: 97.86%
```



result:
Random Normal Initialization:
Advantages:
Simple and easy to implement.
Can work well for small networks.
Disadvantages:
For deeper networks, it can cause vanishing or exploding gradients, making training unstable.
Not tailored to any specific activation function, which can lead to suboptimal training.

He Initialization:
Advantages:
Designed specifically for ReLU (Rectified Linear Unit) and its variants, making it more suitable for networks using these activations.
Helps mitigate the vanishing gradient problem associated with ReLU activations.
Disadvantages:
Not ideal for non-ReLU activations.
Assumes that the distribution of inputs and outputs of each layer is the same, which might not always be the case.

Xavier (Glorot) Initialization:
Advantages:
Designed to keep the scale of gradients roughly the same in all layers.
Works well with Sigmoid and Hyperbolic Tangent (tanh) activation functions.
Helps in faster convergence.
Disadvantages:
Not tailored for ReLU activations; using it with ReLU can lead to vanishing gradients.
Assumes that the distribution of inputs and outputs of each layer is the same, which might not always be the case.

In practice, the choice of initialization often depends on the specific architecture of the neural network and the activation functions used. It's always a good idea to experiment with different initializations to find the one that works best for a given problem.

3. **Define NNs with 7 layers. Compare performance with different optimization methods such as stochastic gradient descent and Adam optimization. You can try other optimization methods as well.**

- Code explanation

Define NNs with 7 layers >
class Net(nn.Module): neural network architectures is defined using PyTorch's nn.

Define Hyperparameters >
device = The device to run the training on, it can be "cpu" or a specific GPU device denoted by a number (e.g., "0" for the first GPU).
BATCH_SIZE = The batch size for training the neural network.
LR = The learning rate for the optimizer.
EPOCH = The number of epochs to train the model.

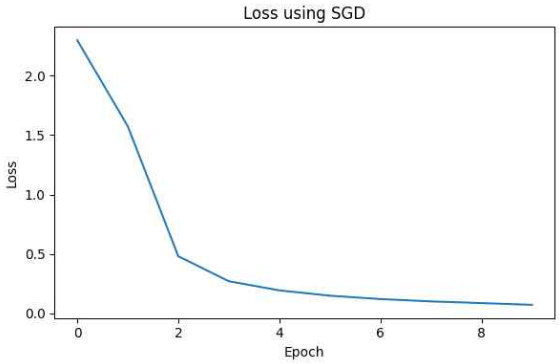
Load MNIST dataset >
Load train and test MNIST data.(transformation normalizes the tensor image with a given mean and standard deviation. In this case, both the mean and standard deviation are set to 0.5)

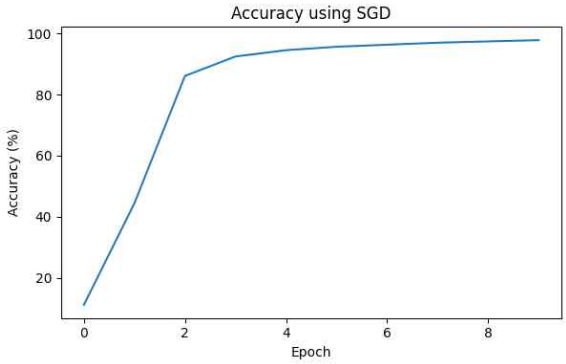
Define Train >
This loop iterates over the dataset for a specified number of epochs. An epoch is one complete forward and backward pass of all the training examples. After each epoch, the average training loss and accuracy are printed. and Loss and accuracy graph output for 3 different optimizers.

- Result report

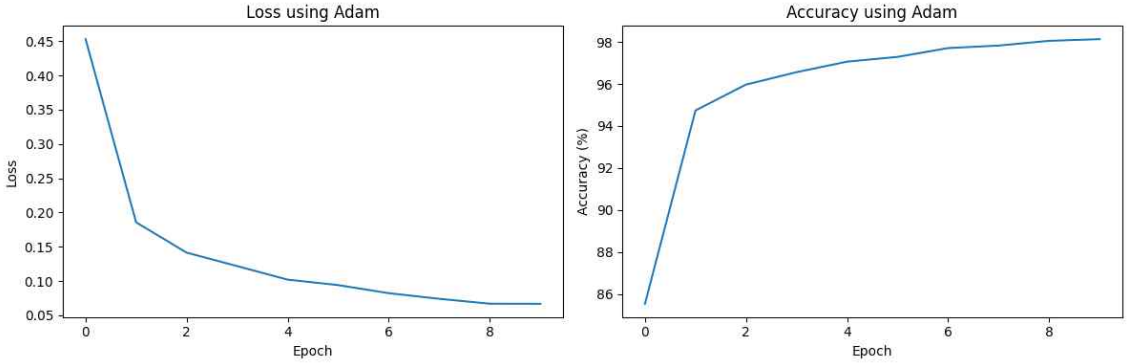
Net(
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=400, bias=True)
  (fc3): Linear(in_features=400, out_features=300, bias=True)
  (fc4): Linear(in_features=300, out_features=200, bias=True)
  (fc5): Linear(in_features=200, out_features=100, bias=True)
  (fc6): Linear(in_features=100, out_features=50, bias=True)
  (fc7): Linear(in_features=50, out_features=10, bias=True)
)

Training with SGD:
Epoch 1, Loss: 2.2971843163808185, Accuracy: 11.2%
Epoch 2, Loss: 1.5739082083861033, Accuracy: 44.43333333333333%
Epoch 3, Loss: 0.48124897013107937, Accuracy: 86.11%
Epoch 4, Loss: 0.2711700500736634, Accuracy: 92.46666666666667%
Epoch 5, Loss: 0.19418471899479628, Accuracy: 94.53333333333333%
Epoch 6, Loss: 0.14967245710591476, Accuracy: 95.65333333333334%
Epoch 7, Loss: 0.1213306133856376, Accuracy: 96.32166666666667%
Epoch 8, Loss: 0.10170420869328081, Accuracy: 96.98%
Epoch 9, Loss: 0.08737746587550889, Accuracy: 97.405%
Epoch 10, Loss: 0.07291311577831705, Accuracy: 97.81333333333333%

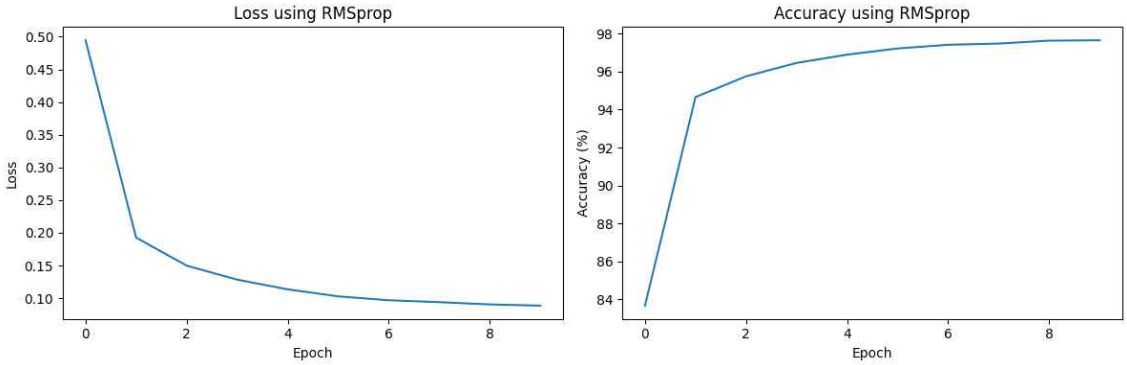




Training with Adam:
Epoch 1, Loss: 0.45285748569741846, Accuracy: 85.54166666666667%
Epoch 2, Loss: 0.18566056743649145, Accuracy: 94.74%
Epoch 3, Loss: 0.14139458602269492, Accuracy: 95.97%
Epoch 4, Loss: 0.12178764755769322, Accuracy: 96.56%
Epoch 5, Loss: 0.10201348260830467, Accuracy: 97.06166666666667%
Epoch 6, Loss: 0.09405997926080599, Accuracy: 97.28833333333333%
Epoch 7, Loss: 0.08222435852377676, Accuracy: 97.70666666666666%
Epoch 8, Loss: 0.07408531284738953, Accuracy: 97.82666666666667%
Epoch 9, Loss: 0.06691174149105791, Accuracy: 98.05166666666666%
Epoch 10, Loss: 0.06673808459994228, Accuracy: 98.13166666666666%



Training with RMSprop:
Epoch 1, Loss: 0.4945583731867373, Accuracy: 83.66333333333333%
Epoch 2, Loss: 0.19275979615536828, Accuracy: 94.66%
Epoch 3, Loss: 0.14989885459262878, Accuracy: 95.755%
Epoch 4, Loss: 0.1284599139738828, Accuracy: 96.46166666666667%
Epoch 5, Loss: 0.11362221709575193, Accuracy: 96.89833333333333%
Epoch 6, Loss: 0.10288893980864086, Accuracy: 97.225%
Epoch 7, Loss: 0.09692733215772702, Accuracy: 97.41833333333334%
Epoch 8, Loss: 0.09401208625731136, Accuracy: 97.48333333333333%
Epoch 9, Loss: 0.09055409720596799, Accuracy: 97.63666666666667%
Epoch 10, Loss: 0.08869045826967922, Accuracy: 97.65666666666667%



result:
1. Stochastic Gradient Descent (SGD):
Advantages:
Simplicity: It's straightforward and has been around for a long time. This makes it a good choice for a baseline method.
Convergence: Under appropriate conditions, it's guaranteed to converge to the global minimum for convex loss surfaces and to a local minimum for non-convex surfaces.
Scalability: It's computationally efficient and can handle large datasets since it updates the weights using only one data point at a time.

Disadvantages:
Learning Rate: The learning rate remains constant and doesn't change during training. This can lead to slow convergence or overshooting the optimal point.
Local Minima: SGD can get stuck in local minima in cases of non-convex loss functions.

Oscillations: Without momentum or other techniques, SGD can oscillate and take longer paths to the optimal point.

2. Adam (Adaptive Moment Estimation):
Advantages:
Adaptive Learning Rate: Adam adjusts the learning rate for each parameter during training, which can lead to faster convergence.
Momentum: It combines the benefits of two other extensions of SGD, AdaGrad and RMSProp. This helps in navigating the ravines (areas where the surface curves much more steeply in one dimension than in another) which are common around local optima.
Bias Correction: It uses moving averages of the parameters to ensure that they are biased towards zero initially, making the updates more reliable.

Disadvantages:
Memory: Adam stores an exponentially decaying average of past squared gradients, which can increase its memory requirements.
Hyperparameters: Although it's less sensitive to hyperparameters than other methods, the default values might not be optimal for all problems.

3. RMSprop (Root Mean Square Propagation):
Advantages:
Adaptive Learning Rate: Like Adam, RMSprop also adjusts the learning rate during training, which can prevent the aggressive and diverging updates.
Convergence: It has been shown to be effective for non-stationary objectives and problems with noisy and sparse gradients.
Less Sensitive to Hyperparameters: Compared to other methods, RMSprop is often less sensitive to the choice of hyperparameters.

Disadvantages:
Memory: Like Adam, RMSprop also uses past squared gradients to compute the adaptive learning rate, which can increase its memory requirements.
No Bias Correction: Unlike Adam, RMSprop doesn't include bias correction, which can sometimes lead to suboptimal convergence.

In practice, the choice of optimization algorithm often depends on the specific problem and the nature of the data. It's common to try multiple optimizers and select the one that performs best for a given task.

4. **Define NNs with 7 layers. Compare convergence speed with/without batch normalization or layer normalization. We haven't learned the batch normalization yet, but coding is not difficult. Please refer to the last page of the class materials or browse webpages.**

- Code explanation

Define Hyperparameters >
device = The device to run the training on, it can be "cpu" or a specific GPU device denoted by a number (e.g., "0" for the first GPU).
BATCH_SIZE = The batch size for training the neural network.
LR = The learning rate for the optimizer.
EPOCH = The number of epochs to train the model.

Load MNIST dataset >
Load train and test MNIST data.(transformation normalizes the tensor image with a given mean and standard deviation. In this case, both the mean and standard deviation are set to 0.5)

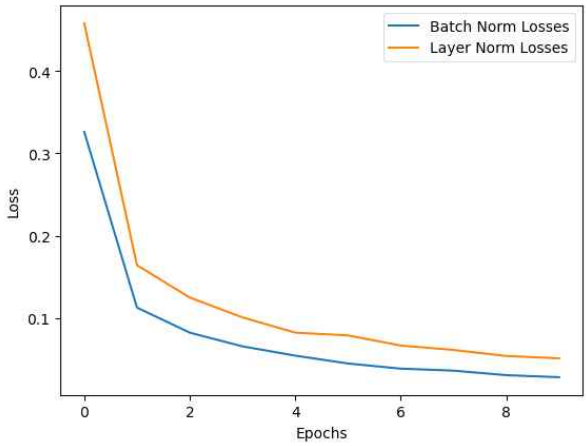
Define NNs with 7 layers >
class normalization networks(batch and layer) architectures are defined using PyTorch's nn.
and layer consist of 7 layers and using ReLU.

Define Train >
This loop iterates over the dataset for a specified number of epochs. An epoch is one complete forward and backward pass of all the training examples. After each epoch, the average training loss is printed. and Training is performed on the two different nets declared above(batch and layer normalization), and the time is measured at the beginning and end of learning.

- Result report

```
BatchNormNN(
  (main): Sequential(
    (0): Linear(in_features=784, out_features=500, bias=True)
    (1): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=500, out_features=250, bias=True)
    (4): BatchNorm1d(250, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=250, out_features=100, bias=True)
    (7): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): Linear(in_features=100, out_features=50, bias=True)
    (10): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): Linear(in_features=50, out_features=25, bias=True)
    (13): BatchNorm1d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU()
    (15): Linear(in_features=25, out_features=10, bias=True)
  )
)
LayerNormNN(
  (main): Sequential(
    (0): Linear(in_features=784, out_features=500, bias=True)
    (1): LayerNorm((500,), eps=1e-05, elementwise_affine=True)
    (2): ReLU()
    (3): Linear(in_features=500, out_features=250, bias=True)
    (4): LayerNorm((250,), eps=1e-05, elementwise_affine=True)
    (5): ReLU()
    (6): Linear(in_features=250, out_features=100, bias=True)
    (7): LayerNorm((100,), eps=1e-05, elementwise_affine=True)
    (8): ReLU()
    (9): Linear(in_features=100, out_features=50, bias=True)
    (10): LayerNorm((50,), eps=1e-05, elementwise_affine=True)
    (11): ReLU()
    (12): Linear(in_features=50, out_features=25, bias=True)
    (13): LayerNorm((25,), eps=1e-05, elementwise_affine=True)
    (14): ReLU()
    (15): Linear(in_features=25, out_features=10, bias=True)
  )
)
```

Batch Normalization
Epoch 1, Loss: 0.32604133253936957
Epoch 2, Loss: 0.11267276650973793
Epoch 3, Loss: 0.08229187840987037
Epoch 4, Loss: 0.06561633639200441
Epoch 5, Loss: 0.05435531423625741
Epoch 6, Loss: 0.044813328541379406
Epoch 7, Loss: 0.03856553038188032
Epoch 8, Loss: 0.036138370266007476
Epoch 9, Loss: 0.030648145469777816
Epoch 10, Loss: 0.028191000534117775
Layer Normalization
Epoch 1, Loss: 0.45790536362908163
Epoch 2, Loss: 0.16410615726876487
Epoch 3, Loss: 0.12493620903344392
Epoch 4, Loss: 0.10081158172903157
Epoch 5, Loss: 0.08225565497576794
Epoch 6, Loss: 0.07894457729239422
Epoch 7, Loss: 0.0666230501828473
Epoch 8, Loss: 0.061261784051855735
Epoch 9, Loss: 0.05395263072593745
Epoch 10, Loss: 0.051074860289619405



Compare Train Time
Training time with Batch Normalization: 148.50 seconds
Training time with Layer Normalization: 149.34 seconds

Neural Network Architectures:
BatchNormNN: This neural network uses Batch Normalization. It consists of several linear layers, each followed by a Batch Normalization layer and a ReLU activation function.
LayerNormNN: This neural network uses Layer Normalization. Similarly, it has multiple linear layers, each followed by a Layer Normalization layer and a ReLU activation function.

Training Results:
Batch Normalization:
The loss starts at 0.3260 in the first epoch and decreases steadily to 0.0282 by the 10th epoch.
This indicates that the model is learning and improving its performance over time.
Layer Normalization:
The loss starts at a higher value of 0.4579 in the first epoch compared to Batch Normalization. It decreases to 0.0511 by the 10th epoch, which is also a significant improvement, but the final loss is slightly higher than that of Batch Normalization.

Training Time:
Batch Normalization: The training time is 148.50 seconds.
Layer Normalization: The training time is slightly longer at 149.34 seconds.

Analysis:

Performance: Both normalization techniques help in stabilizing the learning process and improving convergence. However, based on the provided loss values, Batch Normalization seems to perform slightly better than Layer Normalization in this specific scenario. By the 10th epoch, the model with Batch Normalization achieved a lower loss compared to the one with Layer Normalization.

Training Time: The training times for both models are almost identical, with only a minor difference. This suggests that in terms of computational efficiency, there isn't a significant difference between the two normalization techniques for this specific dataset and architecture.

General Observation: Normalization techniques like Batch Normalization and Layer Normalization are essential for deep neural networks as they help in stabilizing the activations and gradients, leading to faster convergence. The choice between them often depends on the specific problem, dataset, and network architecture. In some cases, one might outperform the other, but in other scenarios, the differences might be negligible.

In conclusion, while both normalization techniques have their merits, Batch Normalization slightly outperformed Layer Normalization in this particular experiment in terms of loss values. However, the training times were almost identical. It's essential to consider the specific requirements and characteristics of each problem when choosing a normalization technique.