Long Short-Term Memory Models with PyTorch

Matthew Harper
University of Houston Victoria

Deep Learning Topic Areas:
Recurrent Networks Feedforward MLPs Principal Component Analysis Deep Networks
Linear Factor Models Non-linear Activation Long Short-Term Memory Sparse Coding

Abstract: Linear factor models arguably present some of the more simplistic methods for developing generative models with latent variables [2]. Linear factor models utilize probabilistic inference to predict a variable value given other independent variables and are often used to build larger deep probabilistic models [1, 2]. Moreover, a linear factor model can be further defined as a stochastic model or a model well described by a random probability distribution which uses a linear decoder function to generate value x by adding noise to a linear transformation. As a result, principal component analysis (PCA) and sparse coding are special cases of linear factor models which have demonstrated promising results for variable selection, reduced dimensionality, and noise reduction [2]. In addition, Lab 6 extends the rigors of recurrent networks and introduces Long Short-Term Memory models for applications in image recognition training and test models using PyTorch with optimizations methods to reduce long term dependencies and gradient anomalies [1].

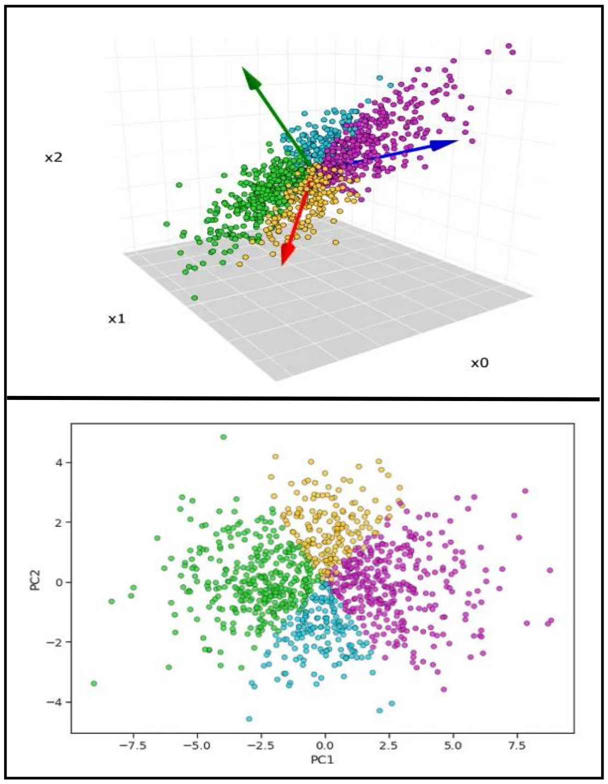


***Figure 1** – Example of a principle component analysis applied to a 3-dimensional hyperplane reduced to 2-dimensions with PCA [3].*

Linear Factor Models

A linear factor model is defined as a stochastic linear decoder type function. Stochastic refers to a random process or a process being well-described by a random probability distribution. In terms of Bayesian interpretation, this stochastic random probability distribution of a variable or function value is often generalized as a Gaussian distribution [2]. Linear factor models have significant importance for deep learning and modeling in general due to their ability to identify explanatory factors and predict values that have joint distributions other than closed form functions.

If we let h be some linear transformation having a combination of independent latent factors, then a linear factor model can be said to describe the data generation process formally with the relationship,

$$\mathbf{h} \sim p(\boldsymbol{h}),$$

where p(h) us a factorial distribution [2].

It follows that a set of data with an observable finite set of variables has an observed data vector output x represented as,

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} + \text{noise}$$

where noise has a random Gaussian distribution and is diagonal or independent across dimensions [2].

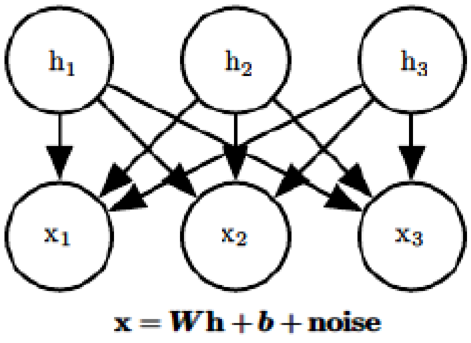
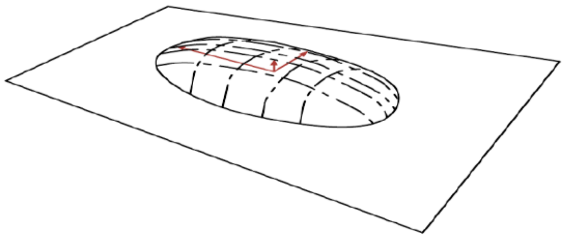

h_1
h_2
h_3
x_1
x_2
x_3
$\mathbf{x} = \boldsymbol{W}\mathbf{h} + \boldsymbol{b} + \textbf{noise}$

Figure 2 – *Linear factor model where x is obtained by a linear combination of independent latent factors (also transformed) by h [2].*

<u>*Principle Component Analysis (PCA)*</u>

A special case linear factor model, probabilistic principal component analysis or simply principal component analysis (PCA) is an orthogonal projection or transformation of the data into a (usually a lower dimensional) subspace so that the variance of the projected data is maximized [1, 2]. Principle component analysis and linear factor models can be interpreted as filters in a hyperplane often referred to as a learning manifold. PCA helps define a thin hyperplane of dimensionality having high probabilistic latent variable non-isotropic values having high variance. Figure 3 below illustrates an example of an orthogonal projection [2].



Principle component analysis recognizes that most variations in the data can be identified by the latent variable h with some small observable error referred to as reconstruction error. This characteristic is very useful for data modeling and determining which variables contribute to the most change in the parameterized value. If little of no observable change is identified, then that variable class is often labeled as noise and can be targeted for a dimensionality reduction or reduction as a reduction in feature extraction. This process improves both efficiency and accuracy in model training, testing, and inference.

Computation of recurrent neural networks can be divided into three significant blocks of parameters and transformations as follows:

1. Flow of tensors from the input to the hidden state.
2. Tensor flow from the previous hidden state to the next hidden state.
3. From the hidden state to the output.

Independent Component Analysis (ICA)

One of the oldest representation learning techniques, Independent Component Analysis is a statistical and computational technique for revealing hidden factors that underline sets of random variables [1,2,4]. In addition, according to Goodfellow et.al (2013d), ICA is an approach to linear factor models with an objective to separate observed signals or variables into many underlying signals that are scaled and added together to form observed data [2].

Independent component analysis is a linear factor model relatable to principal component analysis and defines a generative model for the observed multivariate data, which is typically given as a large database of samples [4]. Similar to PCA which can be generalized to nonlinear autoencoders, ICA can be generalized to nonlinear generative models. In this regard, a nonlinear function f is used to generate observable data.

In addition, ICA can be extended to nonlinear independent components estimation (NICE) which is used to transform data into a dimensioned space where it has a factorized marginal distribution [2]. In the NICE approach, variables are assumed to consist of linear mixtures of some unknown latent variables *h*, and the mixing system function is also unknown. The latent variables are assumed non-gaussian and mutually independent, and they are called the independent components of the observed data.

Another important feature of ICA is to identify groups of features with statistical independence between groups. Typically, groups are defined with non-overlapping relatable units in an approach known as independent subspace analysis. In topographic ICA, ICA is applied to natural images with overlapping groups of spatially neighboring units to learn Gabor filters. These Gabor filters are utilized so that pooling over small regions yields little or no translation variance. This is beneficial for edge detection and feature extraction in image processing. The figure below illustrates an example of an applied Gabor filter applied to an original image for processing.

Slow

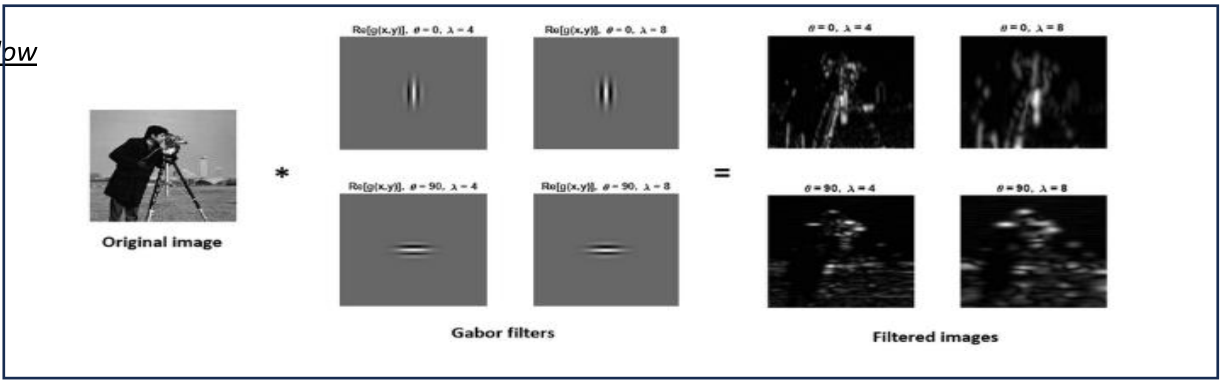


Figure 3 – *Gabor filter used for image processing and edge detection [8].*

Feature Analysis (SFA)

Slow feature analysis (SFA) is an unsupervised learning method utilized to extract the slowest or smoothest underlying functions or features from a time series where time is used as an index [2, 5]. This is often used for dimensionality reduction, regression, and classification [5]. Motivated in general by the slowness principle, slow feature analysis attempts to recognize and separate slow moving objects or variables from faster moving ones. An example of this is given by Goodfellow et.al. (2013d) in which computer vision utilizes SFA to observe and detect pixel values which can change rapidly if a zebra move across an image from left to right. This would be a good application for motion detection versus facial recognition which may be better served with a convolutional neural network among others [2].

Of particular interest, when SFA is trained on small patches of videos of natural scenes, SFA with quadratic basis expansions learns features similar to characteristics resembling complex neural network cells in the V1 cortex when trained on videos with random motion within 3-D computer rendered environments. In addition, SFA is also known for its ability to predict which features of the model SFA will learn, even in a deep, nonlinear environment [2].

Sparse Coding

Sparse coding (Olshausen and Field, 1996) is a linear factor model that has been heavily studied as an unsupervised feature learning and feature extraction mechanism. Sparse coding interprets insignificant values of the projected vector in the new representation being zeros [2]. This makes computation more efficient and allows the model to concentrate on the variables with the most variance. This is illustrated in the figure below in which sparse coding is utilized to identify and filter out the significant eigenvalues in the represented image matrices [6].

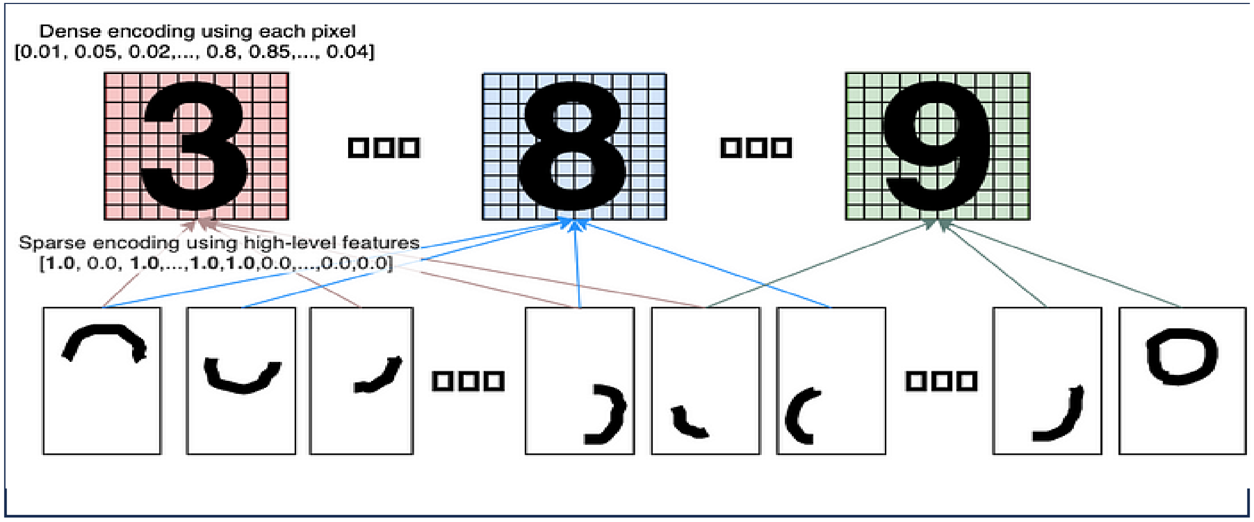


Figure 4 – *Example of sparse coding where significant eigenvalues are project while others are set to 0 [6].*

The term sparse coding refers to the process of inferring the value of h in which it uses a linear decoder plus noise to reconstruct the value of x. With sparse coding models, linear factors [2] are assumed to have Gaussian or randomly distributed noise with isotropic or symmetric procession B such that,

$$p(\boldsymbol{x} \mid \boldsymbol{h}) = \mathcal{N}(\boldsymbol{x}; \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}, \frac{1}{\beta}\boldsymbol{I}).$$

Sparse coding is a non-parametric [2] encoder that solves an optimization problem in which we seek the single most like code value as,

$$h^* = f(x) = \arg\max_{h} p(h \mid x).$$

This equation can be reduced if the terms not dependent on h are dropped and can be divided by a positive scaling factor to simplify the equation as follows:

$$= \arg\min_{h} \lambda ||h||_1 + \beta ||x - Wh||_2^2,$$

Advantages of sparse coding is that in principle one can minimize reconstruction error better than any parametric encoder. Another advantage is that there is no generalization error associated with sparse coding [2]. However, because sparse coding in non-parametric, each value and variable must be iterated thus the computational time required is significantly increased [2]. In addition, sparse coding can produce poor results when the model is unable to reconstruct properly due to the model generating random subsets of all features in the model.

Long Short-Term Memory
The LSTM networks can be viewed as a variation of recurrent neural networks that are considered one of the most utilized sequence modeling techniques which can be applied to many practical applications [2,3]. Long short-term memory (LSTM) models are special cases of recurrent neural networks in which special feedback loops are used to produce paths for the gradient to flow for longer periods of time or durations than its RNN counterpart. In a typical RNN, the information in the short-term memory is often replaced as new information is fed back into the network over time. This is typically an RNNs model can perform well when the gap between the relevant information and the place that is needed is small. An example would be for language processing to determine the next word in a sentence [7].

For applications requiring relevant information or to perform a task that is far away or that requires larger memory. RNNs typically have problems for long term memory tasks because their feedback loop is on a fixed time scale and ,as a result, the needed information is typically replaced by other information in the short-term memory by the time it is needed for the creation of longer acquired contextual information [2]. Moreover, cases where contextual information is required over a paragraph or perhaps a page of words and sentences and other large data sets require longer term memory for optimal performance. RNNs usually do not perform as well as LSTM models in these scenarios [7].

Long Short-Term Memory (LSTM) models on the other hand perform well when longer short-term memory is required for applications such as handwriting recognition, speech recognition, and machine translation among others [1,2,7].

LSTM model variation is largely a result of a weighted self-loop that allows dynamically conditioned information flow on the context rather than a fixed loop compared to RNNs. This is accomplished with the use of three primary gate layers which include: input gate, forget gate (feedback gate), and the output gate [7].

The gates in the LSTM model determine the flow of data and information through the network and are able to dynamically change the time scale based on the input sequence. This translates into LSTM networks able to learn long-term dependencies among the sequence of inputs more easily than the regular RNNs [2].

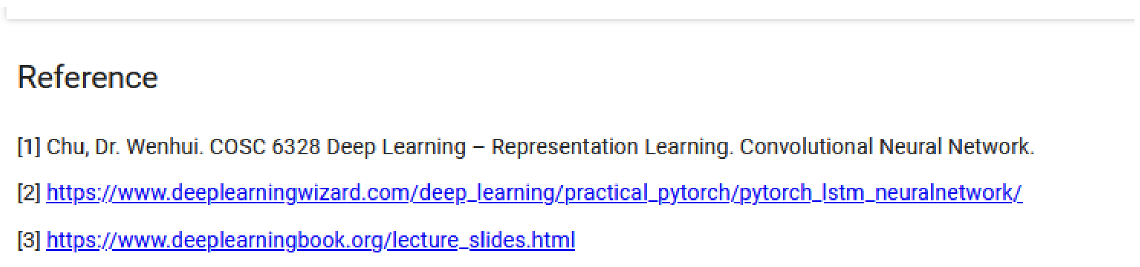
Project Steps with Screenshots for Building a Long Short-Term Memory Model with PyTorch

The process for the Recurrent Neural Network is as follows:

Step 1 – Load the Dataset and libraries
Step 2 – Make the Dataset Iterable
Step 3 – Create the model Class (LSTMModel() for the LSTM based on the RNN)
Step 4 – Instantiate the Model Class (as model = LSTMModel())
Step 5 – Instantiate the Loss Class (nn.CrossEntropyLoss() cross-entropy loss function built-in function)
Step 6 – Instantiate Optimizer Class (utilize the stochastic gradient descent SGD with lr = .01)
Step 7 – Train Model

Model A Parameters	**Model B Parameters**	**Model C Parameters**
ReLU	ReLU	Tanh
1 Hidden Layer	2 Hidden Layer	2 Hidden Layer
100 Hidden Units	100 Hidden Units	100 Hidden Units

Recurrent Neural Nework Models		
Model A	**Model B**	**Model C**
ReLU	ReLU	Tanh
1 Hidden Layer	2 Hidden Layer	2 Hidden Layer
Acc. 92.12%	Acc. 63.44%	Acc. 95.14%
Wall time = 1min 23s	Wall time = 2min 02s	Wall time = 1min 47s
Long Short-Term Memory Models		
Model A	**Model B**	**Model C**
LSTM	LSTM	LSTM
1 Hidden Layer	2 Hidden Layer	3 Hidden Layer
Acc. 96.37%	Acc. 95.74%	Acc. 93.14%
Wall time = 1min 11s	Wall time = 1min 32s	Wall time = 1min 58s



Reference

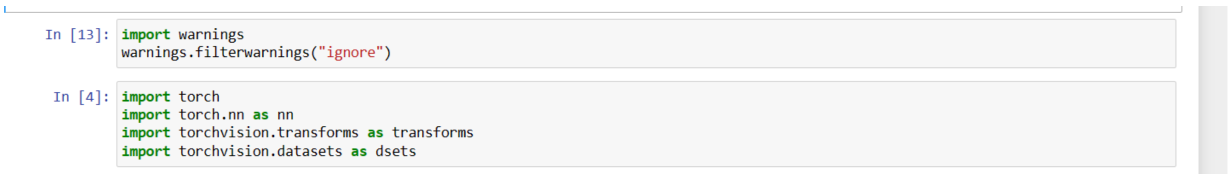
[1] Chu, Dr. Wenhui. COSC 6328 Deep Learning – Representation Learning. Convolutional Neural Network.

[2] https://www.deeplearningwizard.com/deep_learning/practical_pytorch/pytorch_lstm_neuralnetwork/

[3] https://www.deeplearningbook.org/lecture_slides.html

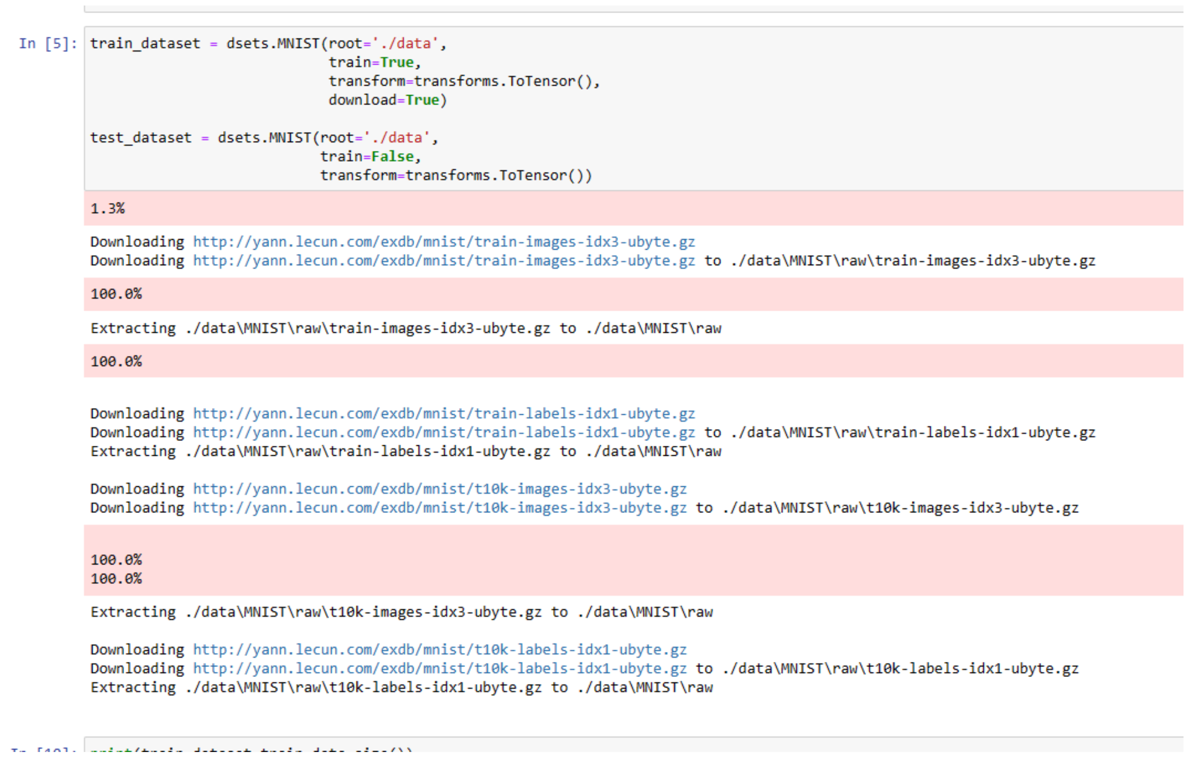
Step 1 – Load the MNIST Train Dataset

The first of the lab is to import the required libraries. First I recommend to use the following script to import the warnings to ignore anything that may arise which occurred the first time I executed the program. Then import the following torch libraries for transforms, neural networks, and datasets.



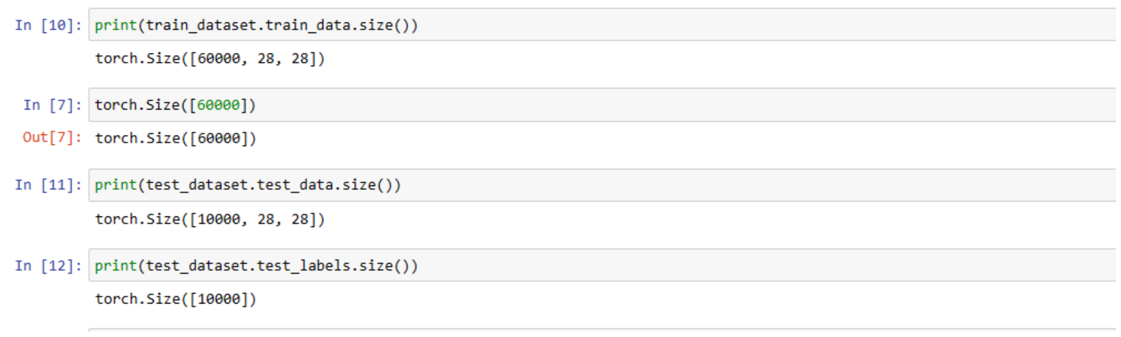
```
In [13]: import warnings
         warnings.filterwarnings("ignore")

In [4]:  import torch
         import torch.nn as nn
         import torchvision.transforms as transforms
         import torchvision.datasets as dsets
```

Now its time to train the dataset using the built-in MNIST data set we can load the data set to a root folder dame data (./data) and assign the train and test data sets with the appropriate train = True/False.



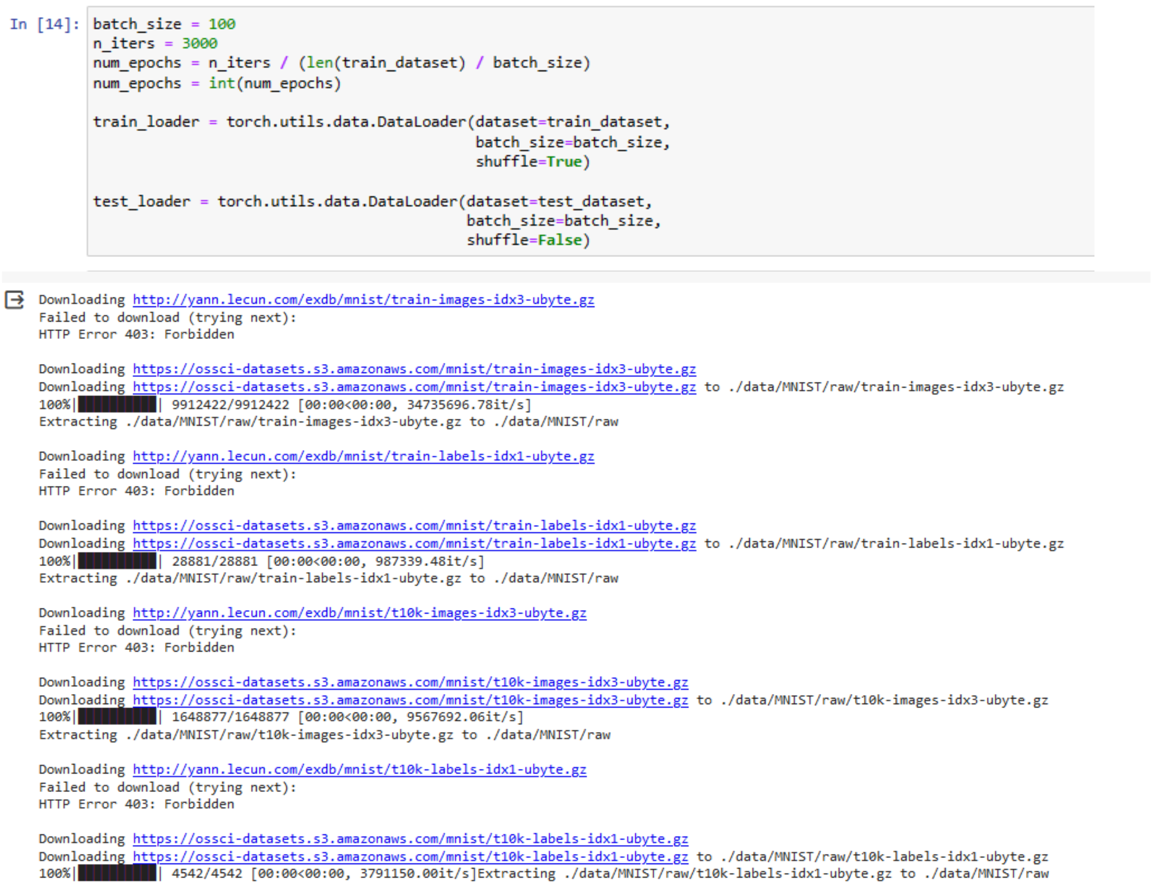
```
In [5]: train_dataset = dsets.MNIST(root='./data',
                                    train=True,
                                    transform=transforms.ToTensor(),
                                    download=True)

        test_dataset = dsets.MNIST(root='./data',
                                   train=False,
                                   transform=transforms.ToTensor())

        1.3%

        Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
        Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data\MNIST\raw\train-images-idx3-ubyte.gz

        100.0%

        Extracting ./data\MNIST\raw\train-images-idx3-ubyte.gz to ./data\MNIST\raw

        100.0%

        Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
        Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data\MNIST\raw\train-labels-idx1-ubyte.gz
        Extracting ./data\MNIST\raw\train-labels-idx1-ubyte.gz to ./data\MNIST\raw

        Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
        Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data\MNIST\raw\t10k-images-idx3-ubyte.gz

        100.0%
        100.0%

        Extracting ./data\MNIST\raw\t10k-images-idx3-ubyte.gz to ./data\MNIST\raw

        Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
        Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw\t10k-labels-idx1-ubyte.gz
        Extracting ./data\MNIST\raw\t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw

In [10]: print(train_dataset.train_data.size())
```

Verify the datasets loaded properly by checking the size.



```
In [10]: print(train_dataset.train_data.size())

         torch.Size([60000, 28, 28])

In [7]:  torch.Size([60000])
Out[7]:  torch.Size([60000])

In [11]: print(test_dataset.test_data.size())

         torch.Size([10000, 28, 28])

In [12]: print(test_dataset.test_labels.size())

         torch.Size([10000])
```

Step 2 – Make the dataset Iterable
Assign the batch size and the number of iterations with epochs.

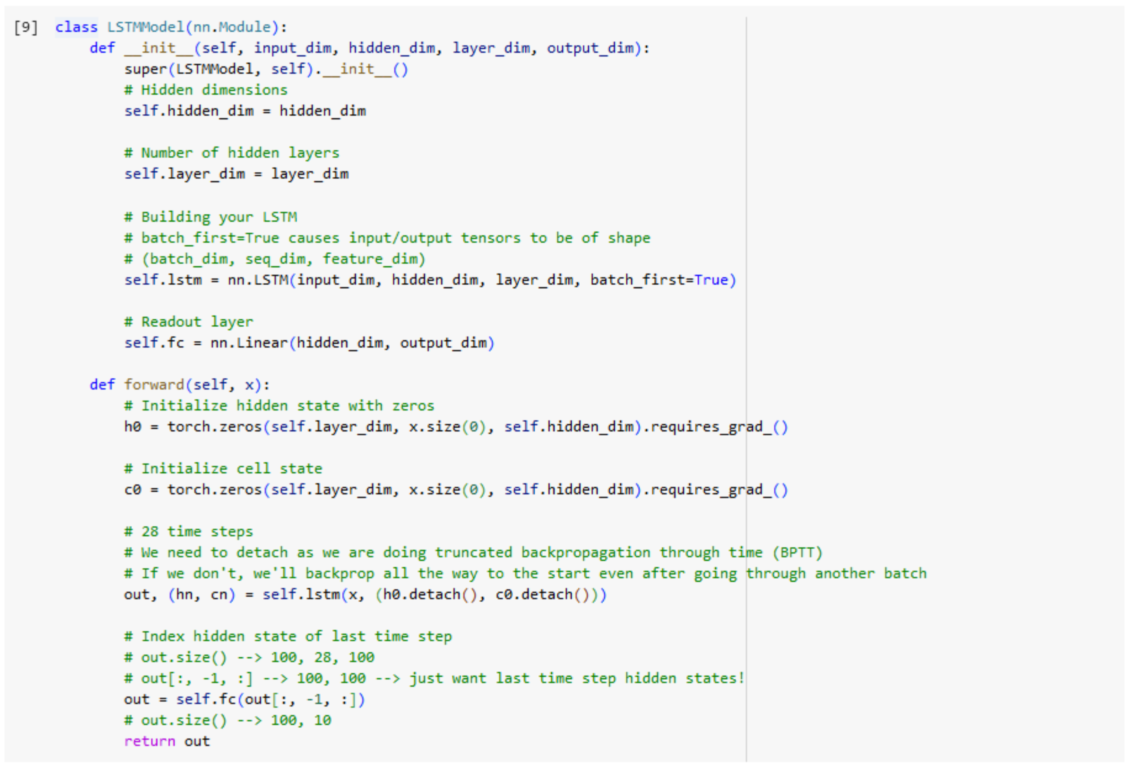


```
In [14]: batch_size = 100
         n_iters = 3000
         num_epochs = n_iters / (len(train_dataset) / batch_size)
         num_epochs = int(num_epochs)

         train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                                    batch_size=batch_size,
                                                    shuffle=True)

         test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                                   batch_size=batch_size,
                                                   shuffle=False)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 34735696.78it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 987339.48it/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 9567692.06it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 3791150.00it/s]Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

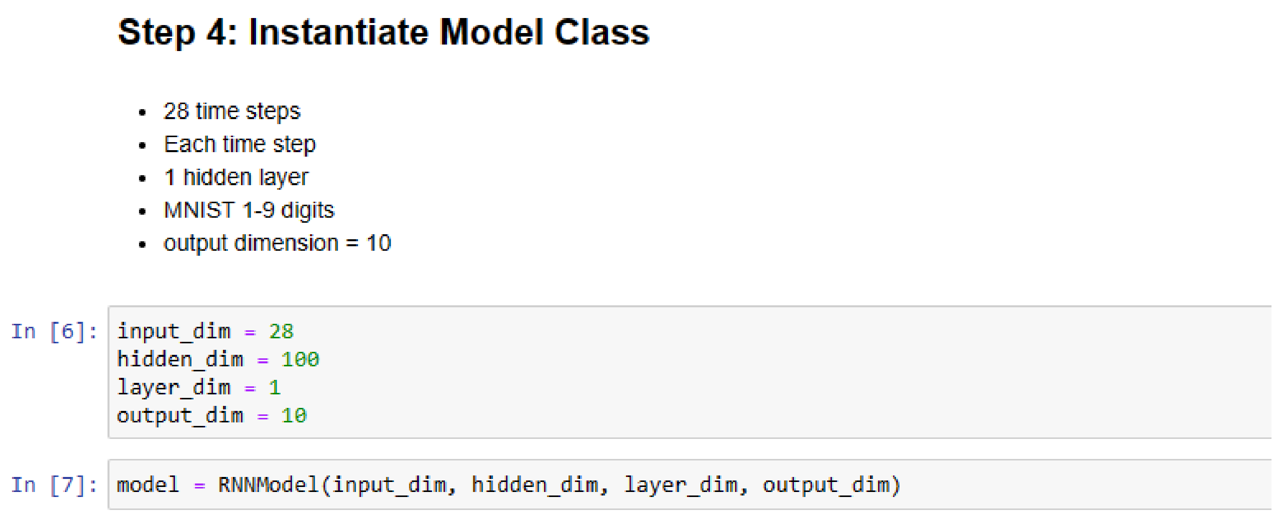
Step 3 – Create the Model Class

Main items to watch for here is that the dimensions are correct for the model. Also we have to create a zeros matrix h for the feedforward function and then detach to prevent exploding and vanishing gradient problems.



```
[9] class LSTMModel(nn.Module):
        def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
            super(LSTMModel, self).__init__()
            # Hidden dimensions
            self.hidden_dim = hidden_dim

            # Number of hidden layers
            self.layer_dim = layer_dim

            # Building your LSTM
            # batch_first=True causes input/output tensors to be of shape
            # (batch_dim, seq_dim, feature_dim)
            self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

            # Readout layer
            self.fc = nn.Linear(hidden_dim, output_dim)

        def forward(self, x):
            # Initialize hidden state with zeros
            h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

            # Initialize cell state
            c0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

            # 28 time steps
            # We need to detach as we are doing truncated backpropagation through time (BPTT)
            # If we don't, we'll backprop all the way to the start even after going through another batch
            out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

            # Index hidden state of last time step
            # out.size() --> 100, 28, 100
            # out[:, -1, :] --> 100, 100 --> just want last time step hidden states!
            out = self.fc(out[:, -1, :])
            # out.size() --> 100, 10
            return out
```

Step 4 – Instantiate the Model



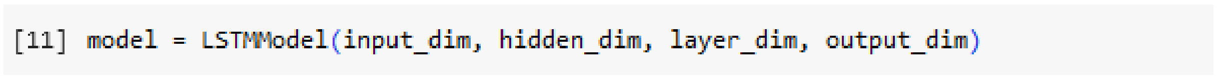
Step 4: Instantiate Model Class

- 28 time steps
- Each time step
- 1 hidden layer
- MNIST 1-9 digits
- output dimension = 10

```
In [6]: input_dim = 28
        hidden_dim = 100
        layer_dim = 1
        output_dim = 10
```

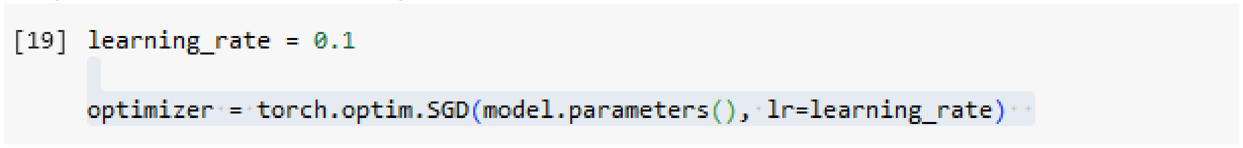
```
In [7]: model = RNNModel(input_dim, hidden_dim, layer_dim, output_dim)
```

Step 5 – Instantiate Loss Class



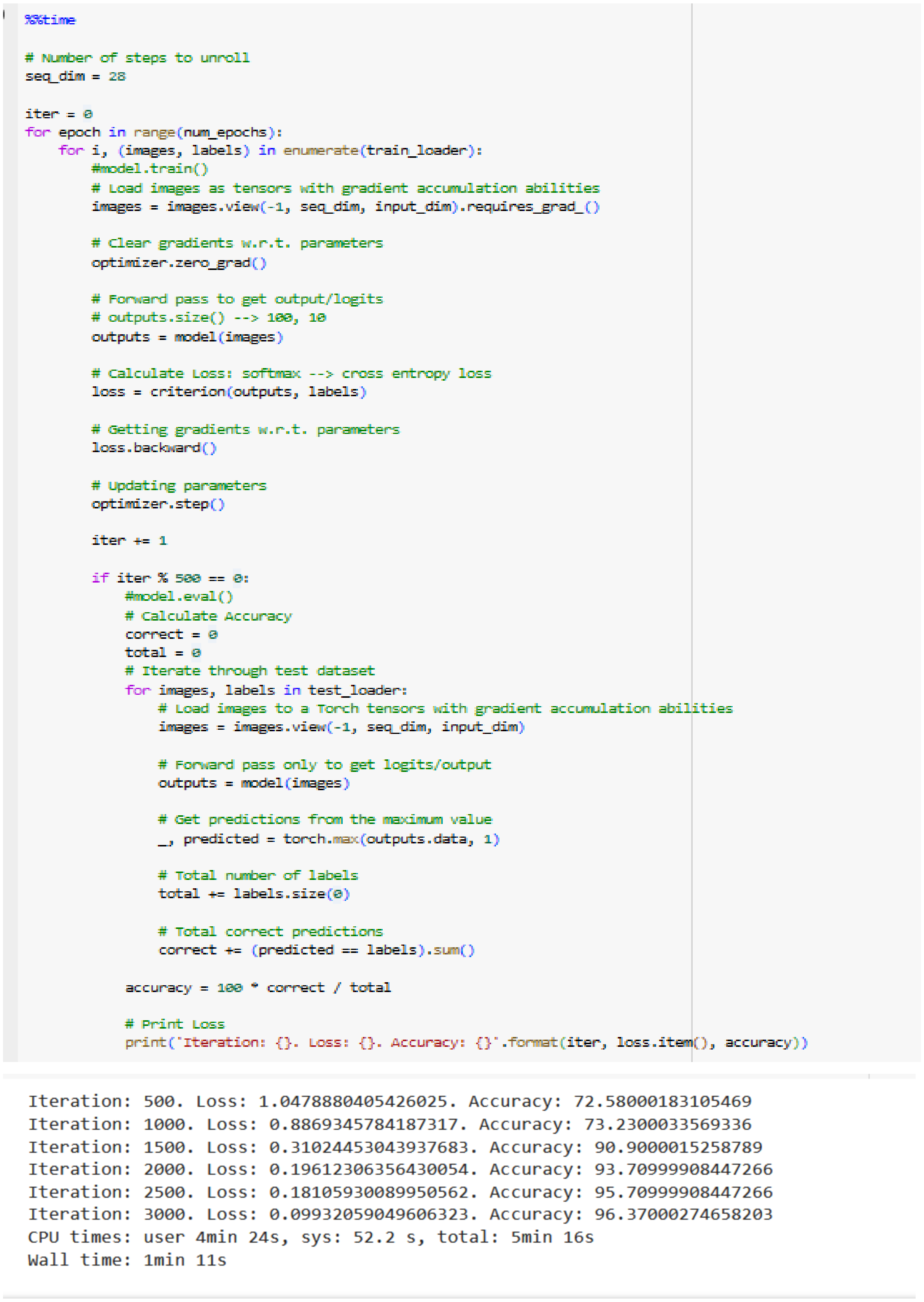
```
[11] model = LSTMModel(input_dim, hidden_dim, layer_dim, output_dim)
```

Step 6 – Instantiate the Optimizer Class



```
[19] learning_rate = 0.1

     optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Step 7 – Train Model



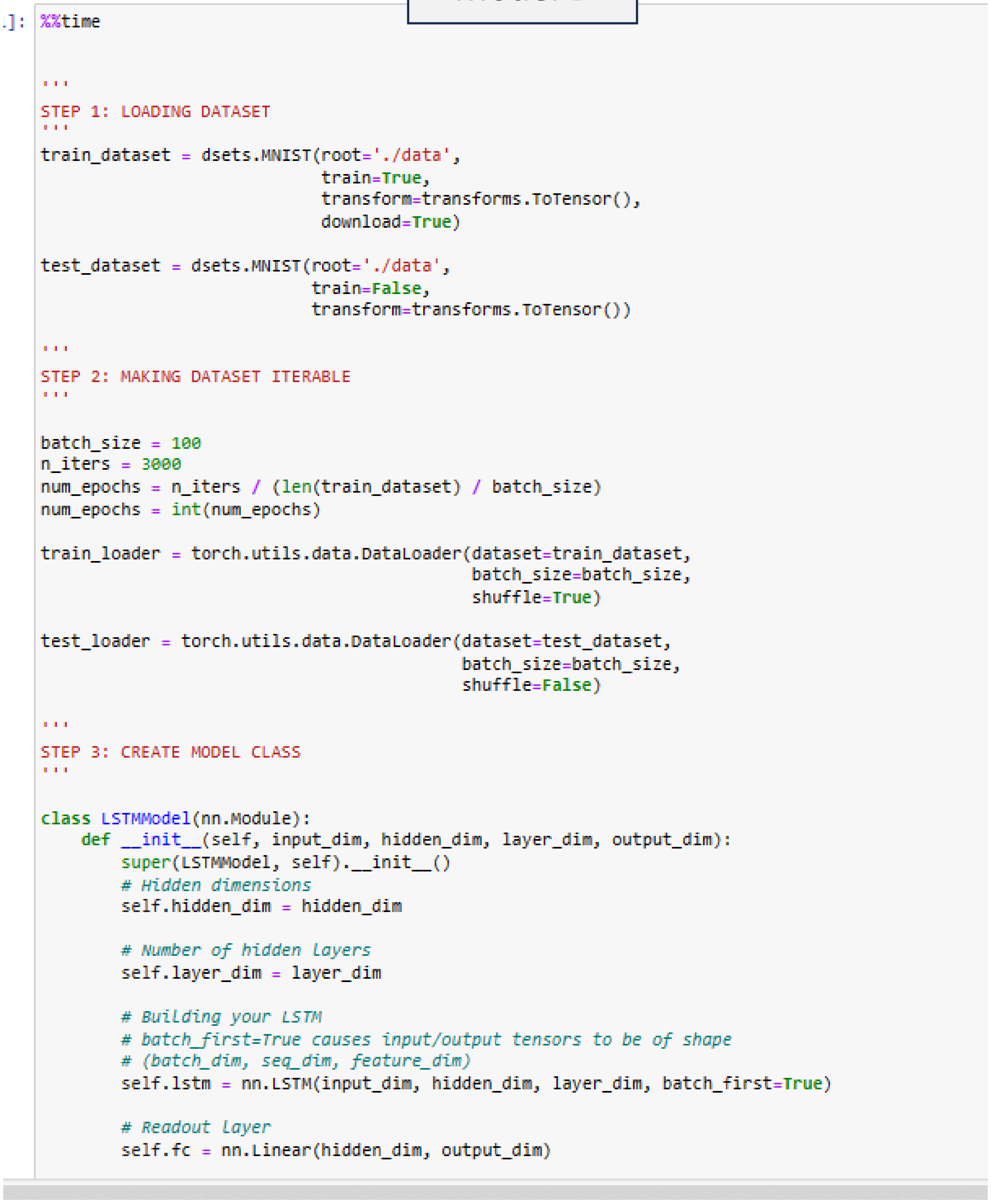
```
%%time

# Number of steps to unroll
seq_dim = 28

iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        #model.train()
        # Load images as tensors with gradient accumulation abilities
        images = images.view(-1, seq_dim, input_dim).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        # outputs.size() --> 100, 10
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            #model.eval()
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to a Torch tensors with gradient accumulation abilities
                images = images.view(-1, seq_dim, input_dim)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
```

```
Iteration: 500. Loss: 1.0478880405426025. Accuracy: 72.58000183105469
Iteration: 1000. Loss: 0.8869345784187317. Accuracy: 73.2300033569336
Iteration: 1500. Loss: 0.31024453043937683. Accuracy: 90.9000015258789
Iteration: 2000. Loss: 0.19612306356430054. Accuracy: 93.70999908447266
Iteration: 2500. Loss: 0.18105930089950562. Accuracy: 95.70999908447266
Iteration: 3000. Loss: 0.09932059049606323. Accuracy: 96.37000274658203
CPU times: user 4min 24s, sys: 52.2 s, total: 5min 16s
Wall time: 1min 11s
```

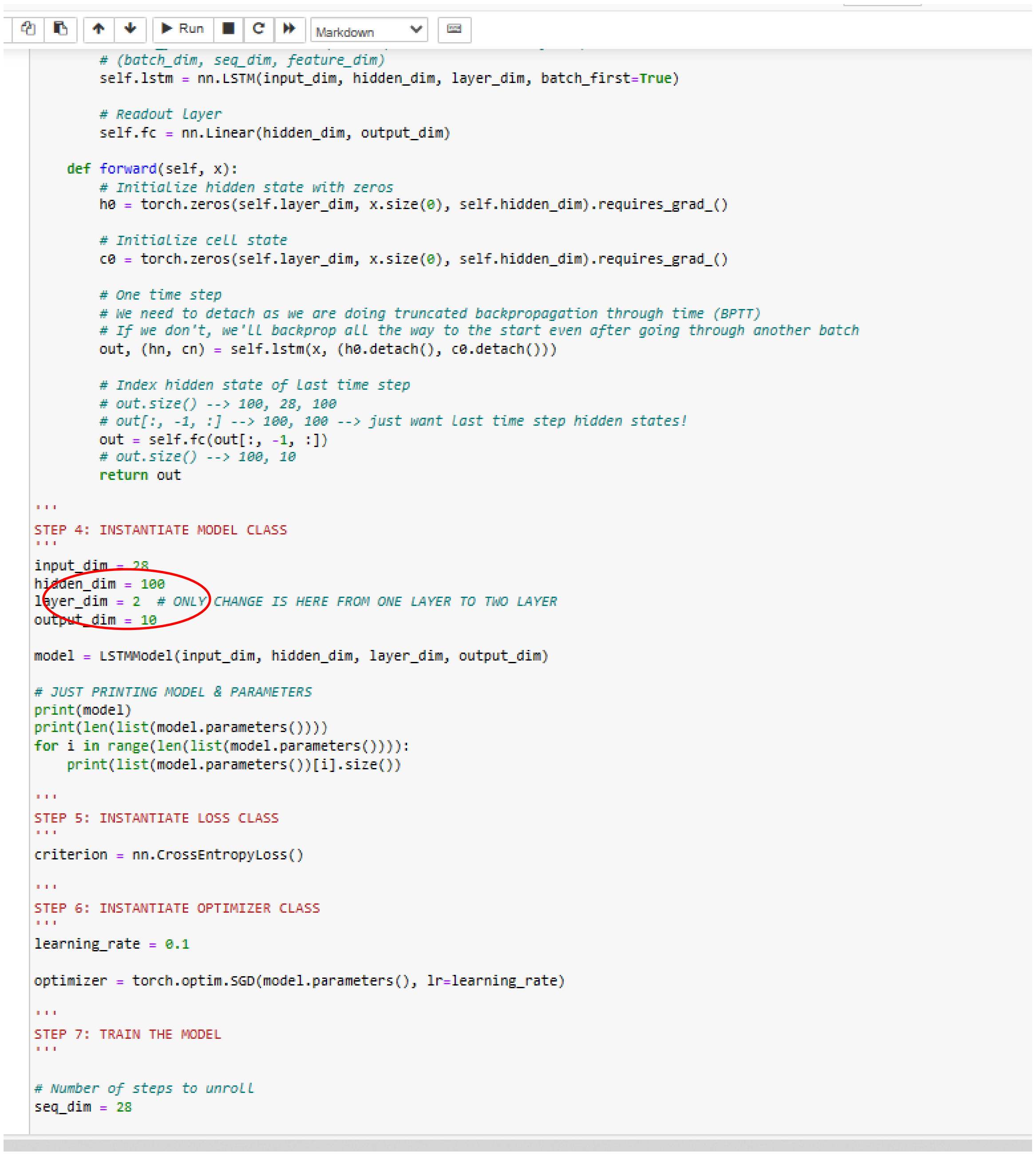
Repeat Steps for Models B, C and GPU

2 Hidden Layers

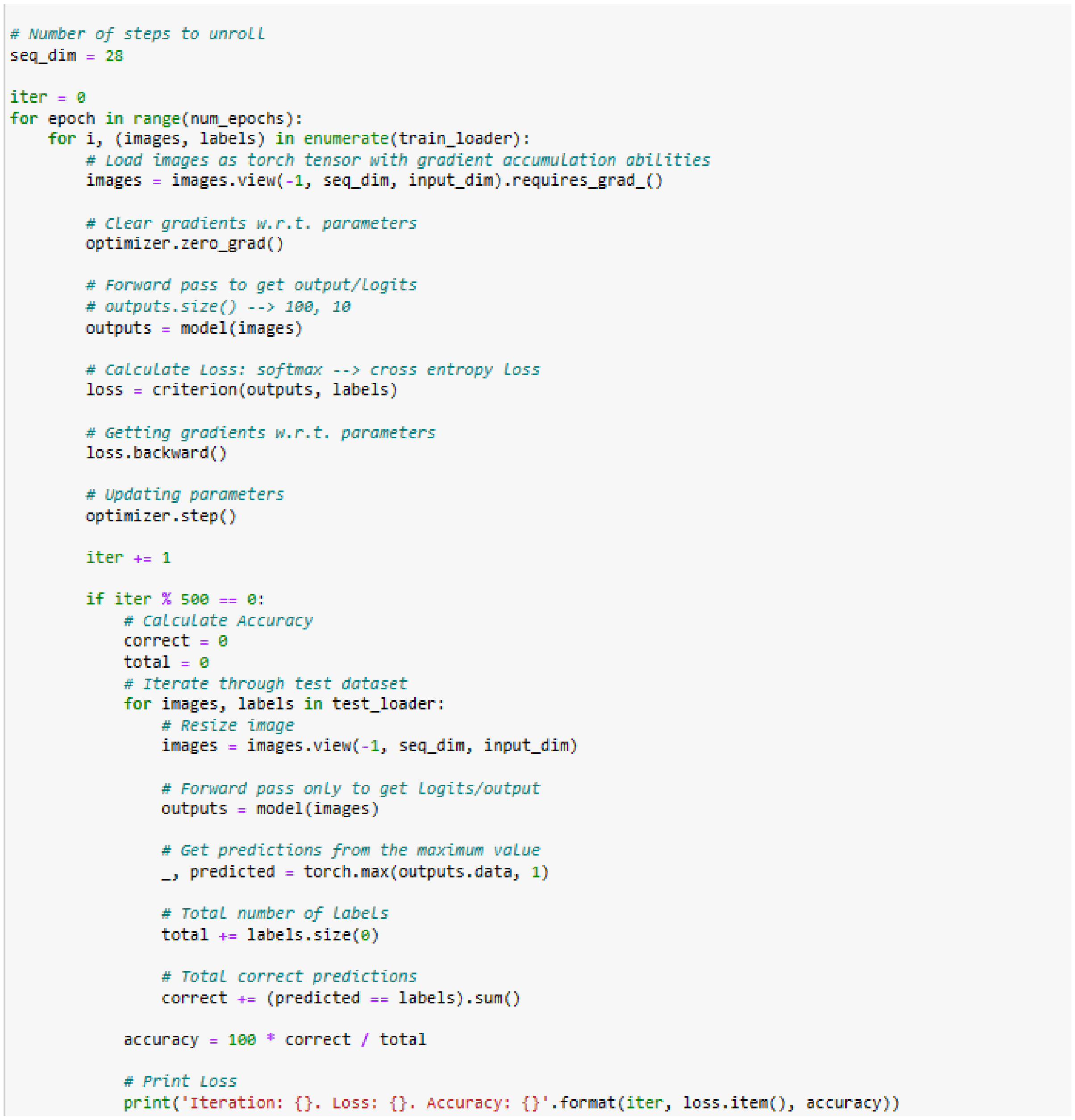
Model B



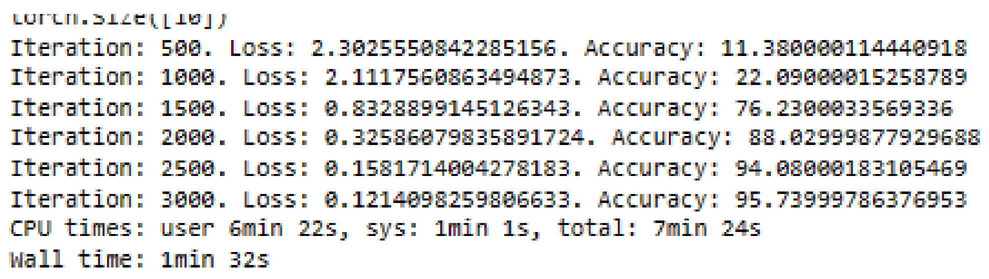
```
.]: %%time

    '''
    STEP 1: LOADING DATASET
    '''
    train_dataset = dsets.MNIST(root='./data',
                                train=True,
                                transform=transforms.ToTensor(),
                                download=True)

    test_dataset = dsets.MNIST(root='./data',
                               train=False,
                               transform=transforms.ToTensor())

    '''
    STEP 2: MAKING DATASET ITERABLE
    '''

    batch_size = 100
    n_iters = 3000
    num_epochs = n_iters / (len(train_dataset) / batch_size)
    num_epochs = int(num_epochs)

    train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                               batch_size=batch_size,
                                               shuffle=True)

    test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                              batch_size=batch_size,
                                              shuffle=False)

    '''
    STEP 3: CREATE MODEL CLASS
    '''

    class LSTMModel(nn.Module):
        def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
            super(LSTMModel, self).__init__()
            # Hidden dimensions
            self.hidden_dim = hidden_dim

            # Number of hidden layers
            self.layer_dim = layer_dim

            # Building your LSTM
            # batch_first=True causes input/output tensors to be of shape
            # (batch_dim, seq_dim, feature_dim)
            self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

            # Readout layer
            self.fc = nn.Linear(hidden_dim, output_dim)
```



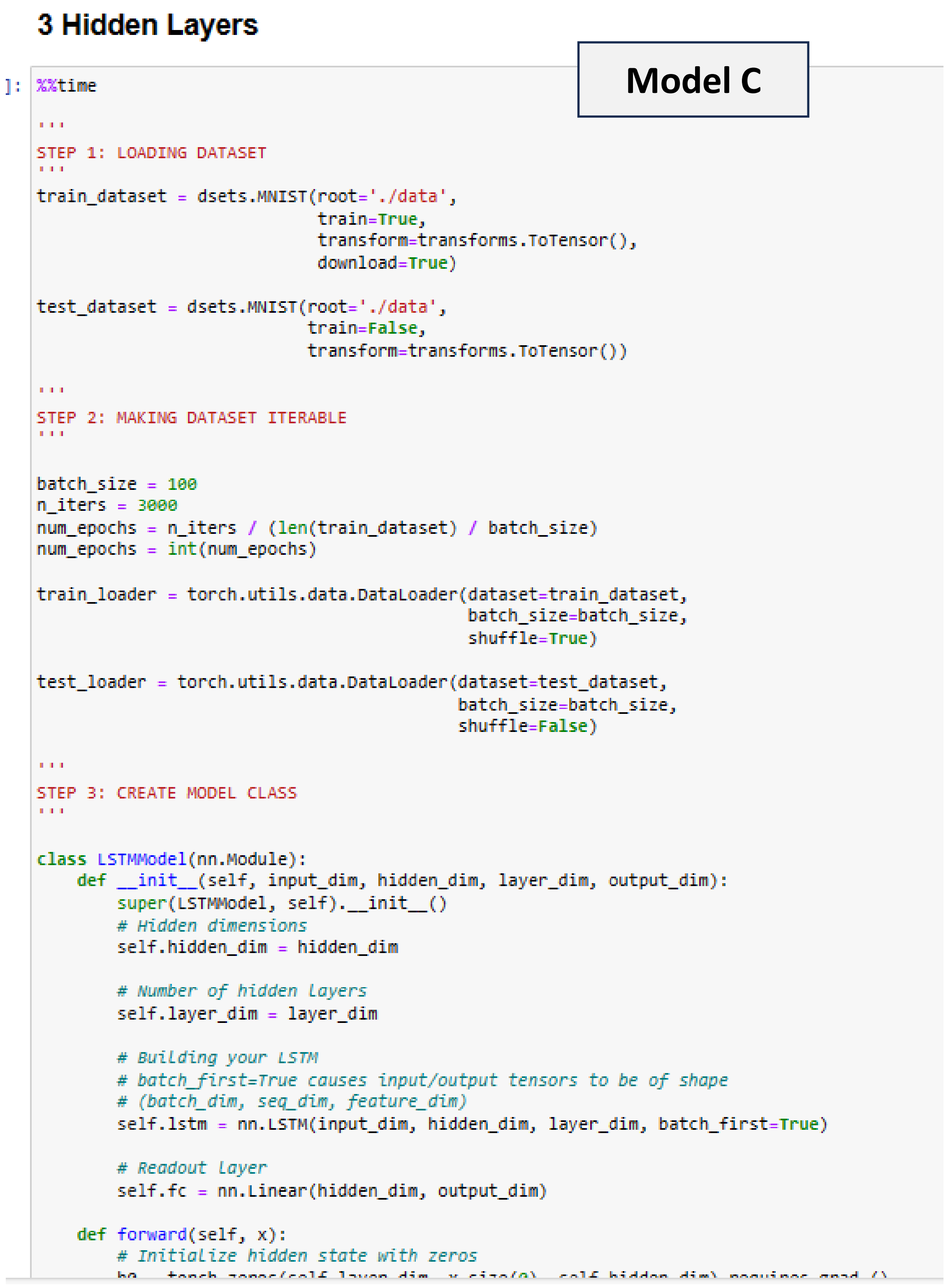
```
        # (batch_dim, seq_dim, feature_dim)
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

        # Readout layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        # Initialize cell state
        c0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        # One time step
        # We need to detach as we are doing truncated backpropagation through time (BPTT)
        # If we don't, we'll backprop all the way to the start even after going through another batch
        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

        # Index hidden state of last time step
        # out.size() --> 100, 28, 100
        # out[:, -1, :] --> 100, 100 --> just want last time step hidden states!
        out = self.fc(out[:, -1, :])
        # out.size() --> 100, 10
        return out

'''
STEP 4: INSTANTIATE MODEL CLASS
'''
input_dim = 28
hidden_dim = 100
layer_dim = 2  # ONLY CHANGE IS HERE FROM ONE LAYER TO TWO LAYER
output_dim = 10

model = LSTMModel(input_dim, hidden_dim, layer_dim, output_dim)

# JUST PRINTING MODEL & PARAMETERS
print(model)
print(len(list(model.parameters())))
for i in range(len(list(model.parameters()))):
    print(list(model.parameters())[i].size())

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()

'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
learning_rate = 0.1

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

'''
STEP 7: TRAIN THE MODEL
'''

# Number of steps to unroll
seq_dim = 28
```



```python
# Number of steps to unroll
seq_dim = 28

iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as torch tensor with gradient accumulation abilities
        images = images.view(-1, seq_dim, input_dim).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        # outputs.size() --> 100, 10
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Resize image
                images = images.view(-1, seq_dim, input_dim)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
```



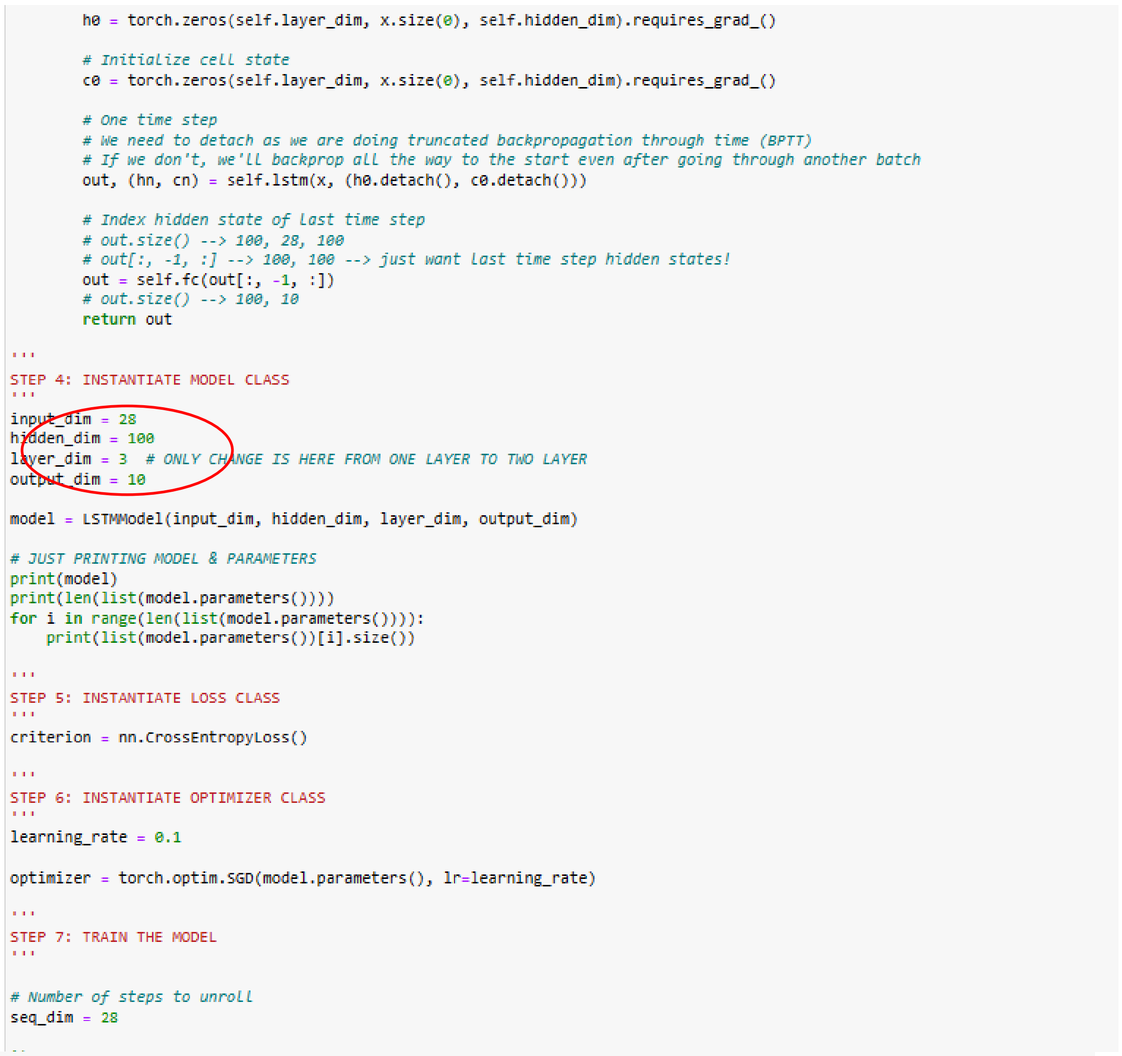
```
torch.Size([10])
Iteration: 500. Loss: 2.3025550842285156. Accuracy: 11.380000114440918
Iteration: 1000. Loss: 2.1117560863494873. Accuracy: 22.09000015258789
Iteration: 1500. Loss: 0.8328899145126343. Accuracy: 76.2300033569336
Iteration: 2000. Loss: 0.32586079835891724. Accuracy: 88.02999877929688
Iteration: 2500. Loss: 0.1581714004278183. Accuracy: 94.08000183105469
Iteration: 3000. Loss: 0.1214098259806633. Accuracy: 95.73999786376953
CPU times: user 6min 22s, sys: 1min 1s, total: 7min 24s
Wall time: 1min 32s
```



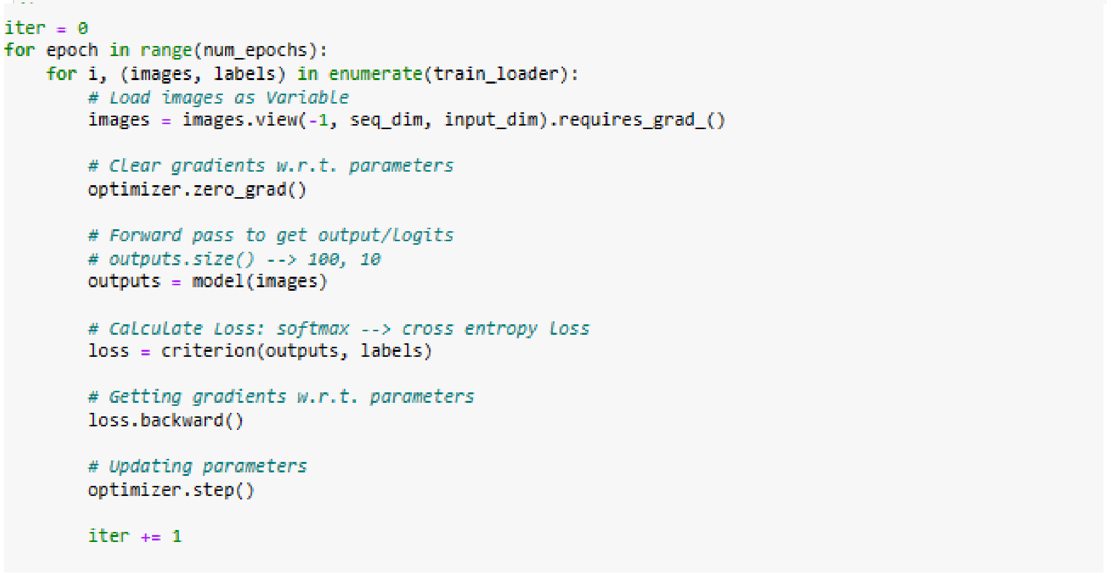
3 Hidden Layers

Model C

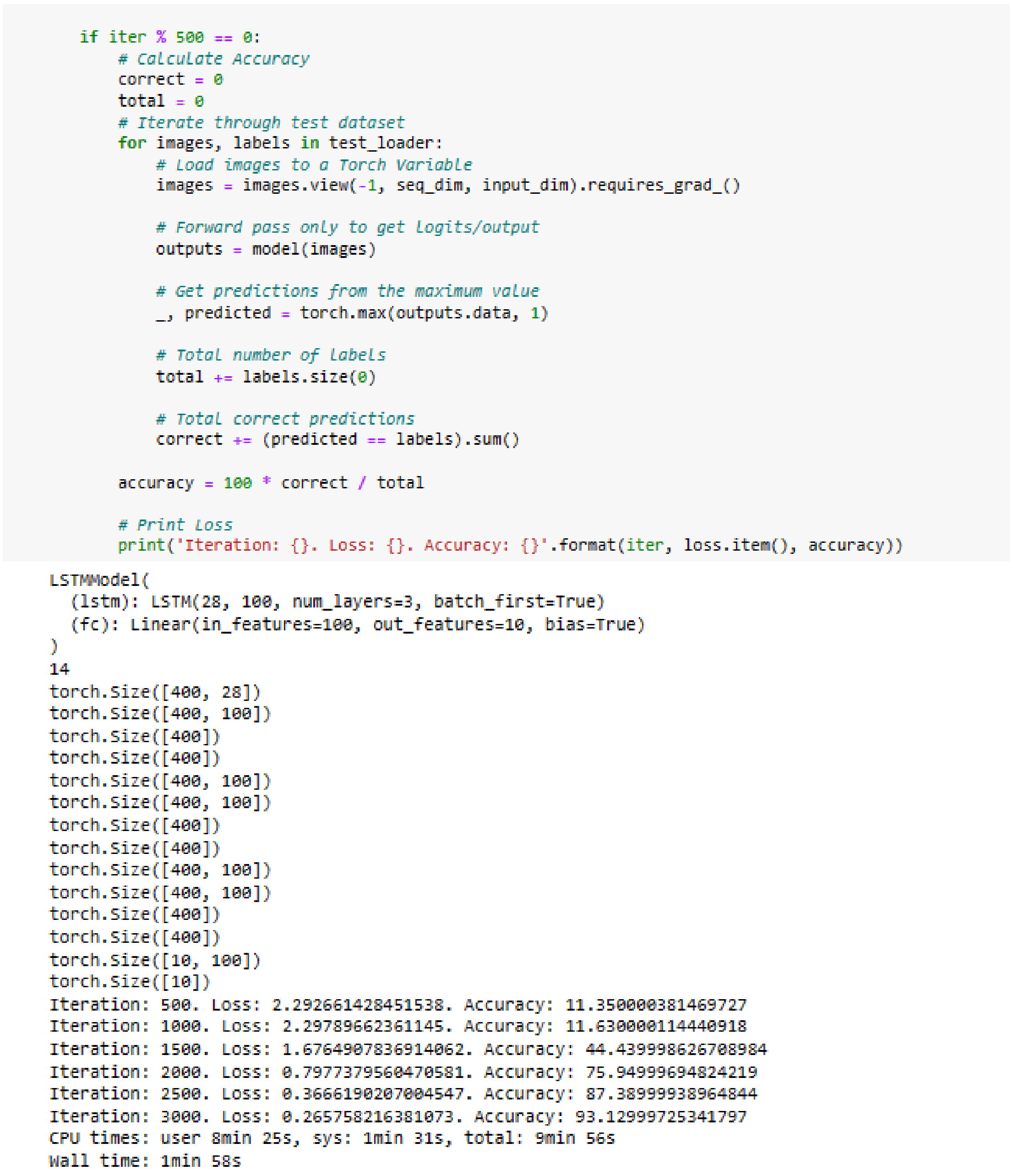
```
]: %%time

'''
STEP 1: LOADING DATASET
'''
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

'''
STEP 2: MAKING DATASET ITERABLE
'''

batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

'''
STEP 3: CREATE MODEL CLASS
'''

class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(LSTMModel, self).__init__()
        # Hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        # Building your LSTM
        # batch_first=True causes input/output tensors to be of shape
        # (batch_dim, seq_dim, feature_dim)
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

        # Readout layer
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()
```



```
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        # Initialize cell state
        c0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        # One time step
        # We need to detach as we are doing truncated backpropagation through time (BPTT)
        # If we don't, we'll backprop all the way to the start even after going through another batch
        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

        # Index hidden state of last time step
        # out.size() --> 100, 28, 100
        # out[:, -1, :] --> 100, 100 --> just want last time step hidden states!
        out = self.fc(out[:, -1, :])
        # out.size() --> 100, 10
        return out

'''
STEP 4: INSTANTIATE MODEL CLASS
'''
input_dim = 28
hidden_dim = 100
layer_dim = 3  # ONLY CHANGE IS HERE FROM ONE LAYER TO TWO LAYER
output_dim = 10

model = LSTMModel(input_dim, hidden_dim, layer_dim, output_dim)

# JUST PRINTING MODEL & PARAMETERS
print(model)
print(len(list(model.parameters())))
for i in range(len(list(model.parameters()))):
    print(list(model.parameters())[i].size())

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()

'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
learning_rate = 0.1

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

'''
STEP 7: TRAIN THE MODEL
'''

# Number of steps to unroll
seq_dim = 28
```



```
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as Variable
        images = images.view(-1, seq_dim, input_dim).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        # outputs.size() --> 100, 10
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1
```



```
        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to a Torch Variable
                images = images.view(-1, seq_dim, input_dim).requires_grad_()

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
```

```
LSTMModel(
  (lstm): LSTM(28, 100, num_layers=3, batch_first=True)
  (fc): Linear(in_features=100, out_features=10, bias=True)
)
14
torch.Size([400, 28])
torch.Size([400, 100])
torch.Size([400])
torch.Size([400])
torch.Size([400, 100])
torch.Size([400, 100])
torch.Size([400])
torch.Size([400])
torch.Size([400, 100])
torch.Size([400, 100])
torch.Size([400])
torch.Size([400])
torch.Size([10, 100])
torch.Size([10])
Iteration: 500. Loss: 2.292661428451538. Accuracy: 11.350000381469727
Iteration: 1000. Loss: 2.29789662361145. Accuracy: 11.630000114440918
Iteration: 1500. Loss: 1.6764907836914062. Accuracy: 44.439998626708984
Iteration: 2000. Loss: 0.7977379560470581. Accuracy: 75.94999694824219
Iteration: 2500. Loss: 0.3666190207004547. Accuracy: 87.38999938964844
Iteration: 3000. Loss: 0.265758216381073. Accuracy: 93.12999725341797
CPU times: user 8min 25s, sys: 1min 31s, total: 9min 56s
Wall time: 1min 58s
```

Conclusion

Overall, this exercise provides machine learning engineers with a comprehensive introduction and tutorial for deep learning model development of recurrent neural networks using PyTorch. The lab provided steps for 3 different models for comparison each which varying hidden layers and activation functions with varying results. Model C exhibited the lowest performance overall when compared to models A and B. Model A had the best performance when compared to the LSTM models. This is largely due to the model becoming over fit with the additional layers for each model A,B,C having hidden layer amounts of 1,2,3 respectively with accuracy of 96.4,95.7,93.1. Overall, this was a great lab in which I learned an extensive amount of knowledge in the subjects of deep learning, feedforward networks, neural networks, objective functions, and parameter sharing with activation transfer functions.

References

[1] Chu, Dr. Wenhui. COSC 6328 Deep Learning – Representation Learning. Convolutional Neural Network.

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.

[3] Cheng, C. (2023, May 6). Principal Component Analysis (PCA) Explained Visually with Zero Math. Medium. https://towardsdatascience.com/principal-component-analysis-pca-explained-visually-with-zero-math-1cbf392b9e7d

[4] What is Independent Component Analysis ? (n.d.). Dept. of CS, University of Helsinki. https://www.cs.helsinki.fi/u/ahyvarin/whatisica.shtml

[5] Hlynsson, H. D. (2021, February 16). A quick introduction to Slow Feature Analysis - Towards Data Science. Medium. https://towardsdatascience.com/a-brief-introduction-to-slow-feature-analysis-18c901bc2a58

[6] P, A. (2021, December 15). Sparse Coding simplified - Aleksandr P - Medium. Medium. https://medium.com/@apatsekin/sparse-coding-simplified-dd09b73eca14

[7] Sharda, R., Delen, D., & Turban, E. (2020b). Analytics, data science, and artificial intelligence: Systems for Decision Support, Global Edition.

[8] Science, B. O. C., & Science, B. O. C. (2024, March 18). How to use Gabor filters to generate features for machine learning | Baeldung on Computer Science. https://www.baeldung.com/cs/ml-gabor-filters