```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

```
# Hyperparameters
sequence_length = 28
input_size =28
hidden_size = 28
num_layers = 2
num_classes= 10
batch_size = 100
num_iters = 1200
learning_rate = 0.001  # More power so we can learn faster! previously it was 0.001

# Device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
'''
LOADING DATASET
'''
train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),  # Normalize the image to [0-1] from [0-255]
                            download=True)

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

'''
MAKING DATASET ITERABLE
'''
num_epochs = num_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,drop_last=True)   # It's better to shuffle the whole training dataset!

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

RNN: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

LSTM: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size= hidden_size
        self.num_layers = num_layers

        # self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True) # For uni Directional RNN
        # self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True,bidirectional=True) # For BiDirectional RNN
        # # self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True) # For uni Directional LSTM
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True,bidirectional=True) # For BiDirectional LSTM
        # self.fc = nn.Linear(hidden_size, num_classes) #For uni Directional
        self.fc = nn.Linear(hidden_size*2, num_classes) #For Bidirectional

    def forward(self, x):
        # set initial hidden and cell states
        # h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device) #For uni Directional
        # c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device) #For uni Directional
        h0 = torch.zeros(self.num_layers*2, x.size(0), self.hidden_size).to(device) #For Bidirectional
        c0 = torch.zeros(self.num_layers*2, x.size(0), self.hidden_size).to(device) #For Bidirectional

        #Forward Propagation
        # out, _  = self.rnn(x,h0)
        out, _  = self.lstm(x,(h0,c0)) #out: tensor of shape (batch size, seq_length, hidden_size)
        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out
```

input_size – The number of expected features in the input x

hidden_size – The number of features in the hidden state h

num_layers – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1

bias – If False, then the layer does not use bias weights b_ih and b_hh. Default: True

batch_first – If True, then the input and output tensors are provided as (batch, seq, feature) instead of (seq, batch, feature). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: False

dropout – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout. Default: 0

bidirectional – If True, becomes a bidirectional LSTM. Default: False

proj_size – If > 0, will use LSTM with projections of corresponding size. Default: 0

```
'''
INSTANTIATE MODEL CLASS
'''
model = RNN( input_size, hidden_size, num_layers, num_classes)
# To enable GPU
model.to(device)
```

```
RNN(
  (lstm): LSTM(28, 28, num_layers=2, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=56, out_features=10, bias=True)
)
```

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
'''
TRAIN THE MODEL
'''
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)

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

        # Forward pass to get output/logits
        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 % 300 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:

                images = images.reshape(-1, sequence_length, input_size).to(device)

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

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

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