In [1]:

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

In [2]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [3]:

```
input_size = 784 # 28x28
hidden_size_1 = 500
hidden_size_2 =250
hidden_size_3 = 100
num_classes = 10
num_epochs = 15
batch_size = 64
learning_rate = 0.01
```

In [4]:

```
# Import MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

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

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


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

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


Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNI
ST/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


Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIS
T/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/MNIS
T/raw
```

In [5]:

```
# Data loader
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)
```

In [6]:

```
# Fully connected neural network
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size_1,hidden_size_2,hidden_size_3, nu
m_classes):
        super(NeuralNet, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size_1)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size_1, hidden_size_2)

        self.l3 = nn.Linear(hidden_size_2, hidden_size_3)

        self.l4 = nn.Linear(hidden_size_3, num_classes)
        self.softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        out = self.relu(out)
        out = self.l4(out)
        out = self.softmax(out)

        return out


model = NeuralNet(input_size, hidden_size_1,hidden_size_2,hidden_size_3, num_cla
sses).to(device)
```

In [7]:

```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=0.001, betas=(0.9, 0.999), ep
s=1e-08 )
```

In [8]:

```
# Train the model
loss_log = []
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


        if (i+1) % 200 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}
], Loss: {loss.item():.4f}')
            loss_log.append(loss.item())
```

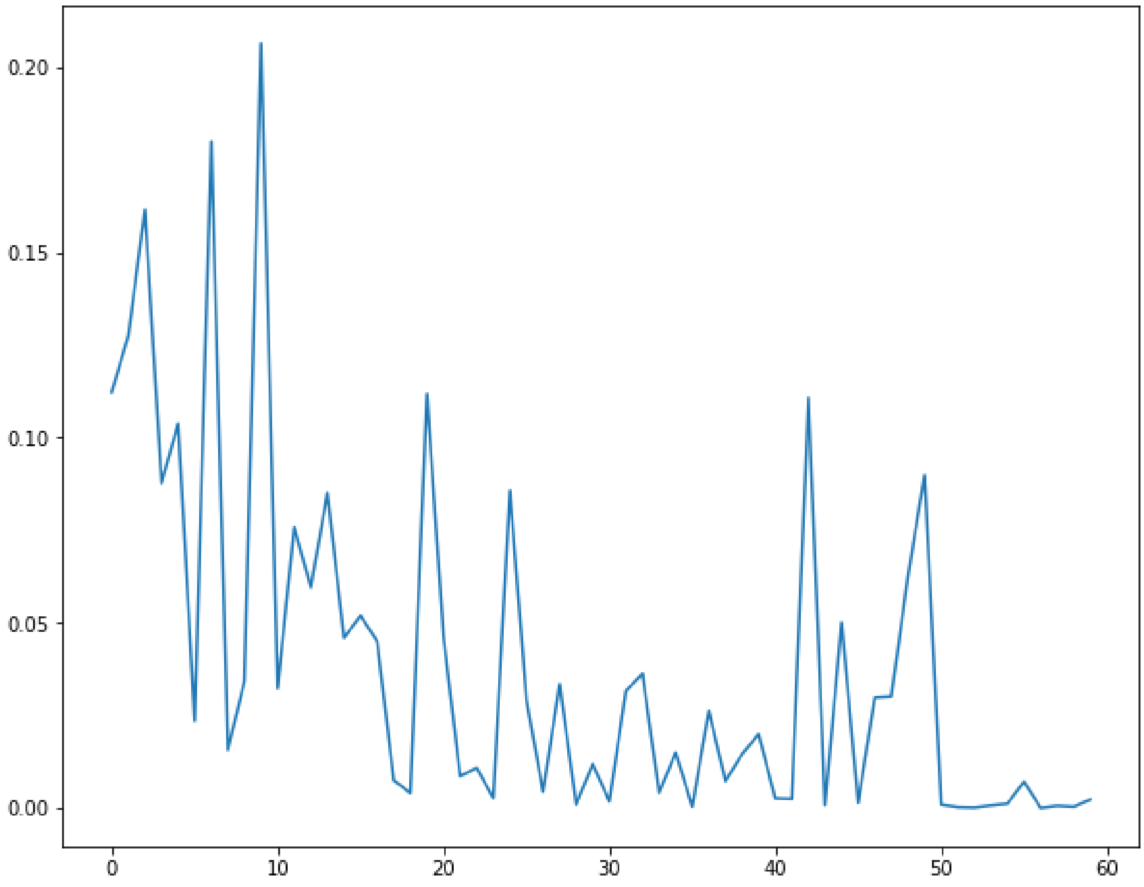
```
Epoch [1/15], Step [200/938], Loss: 0.1122
Epoch [1/15], Step [400/938], Loss: 0.1275
Epoch [1/15], Step [600/938], Loss: 0.1615
Epoch [1/15], Step [800/938], Loss: 0.0877
Epoch [2/15], Step [200/938], Loss: 0.1037
Epoch [2/15], Step [400/938], Loss: 0.0235
Epoch [2/15], Step [600/938], Loss: 0.1799
Epoch [2/15], Step [800/938], Loss: 0.0157
Epoch [3/15], Step [200/938], Loss: 0.0343
Epoch [3/15], Step [400/938], Loss: 0.2063
Epoch [3/15], Step [600/938], Loss: 0.0323
Epoch [3/15], Step [800/938], Loss: 0.0758
Epoch [4/15], Step [200/938], Loss: 0.0595
Epoch [4/15], Step [400/938], Loss: 0.0850
Epoch [4/15], Step [600/938], Loss: 0.0460
Epoch [4/15], Step [800/938], Loss: 0.0519
Epoch [5/15], Step [200/938], Loss: 0.0450
Epoch [5/15], Step [400/938], Loss: 0.0075
Epoch [5/15], Step [600/938], Loss: 0.0041
Epoch [5/15], Step [800/938], Loss: 0.1118
Epoch [6/15], Step [200/938], Loss: 0.0462
Epoch [6/15], Step [400/938], Loss: 0.0086
Epoch [6/15], Step [600/938], Loss: 0.0108
Epoch [6/15], Step [800/938], Loss: 0.0027
Epoch [7/15], Step [200/938], Loss: 0.0857
Epoch [7/15], Step [400/938], Loss: 0.0292
Epoch [7/15], Step [600/938], Loss: 0.0044
Epoch [7/15], Step [800/938], Loss: 0.0334
Epoch [8/15], Step [200/938], Loss: 0.0010
Epoch [8/15], Step [400/938], Loss: 0.0119
Epoch [8/15], Step [600/938], Loss: 0.0018
Epoch [8/15], Step [800/938], Loss: 0.0316
Epoch [9/15], Step [200/938], Loss: 0.0362
Epoch [9/15], Step [400/938], Loss: 0.0041
Epoch [9/15], Step [600/938], Loss: 0.0150
Epoch [9/15], Step [800/938], Loss: 0.0003
Epoch [10/15], Step [200/938], Loss: 0.0262
Epoch [10/15], Step [400/938], Loss: 0.0072
Epoch [10/15], Step [600/938], Loss: 0.0145
Epoch [10/15], Step [800/938], Loss: 0.0200
Epoch [11/15], Step [200/938], Loss: 0.0026
Epoch [11/15], Step [400/938], Loss: 0.0025
Epoch [11/15], Step [600/938], Loss: 0.1108
Epoch [11/15], Step [800/938], Loss: 0.0008
Epoch [12/15], Step [200/938], Loss: 0.0501
Epoch [12/15], Step [400/938], Loss: 0.0014
Epoch [12/15], Step [600/938], Loss: 0.0299
Epoch [12/15], Step [800/938], Loss: 0.0301
Epoch [13/15], Step [200/938], Loss: 0.0625
Epoch [13/15], Step [400/938], Loss: 0.0898
Epoch [13/15], Step [600/938], Loss: 0.0009
Epoch [13/15], Step [800/938], Loss: 0.0002
Epoch [14/15], Step [200/938], Loss: 0.0001
Epoch [14/15], Step [400/938], Loss: 0.0007
Epoch [14/15], Step [600/938], Loss: 0.0012
Epoch [14/15], Step [800/938], Loss: 0.0072
Epoch [15/15], Step [200/938], Loss: 0.0000
Epoch [15/15], Step [400/938], Loss: 0.0006
Epoch [15/15], Step [600/938], Loss: 0.0003
Epoch [15/15], Step [800/938], Loss: 0.0023
```

In [9]:

```
plt.figure(figsize=(10,8))
plt.plot(loss_log)
```

Out[9]:

```
[<matplotlib.lines.Line2D at 0x7f410f4f9e50>]
```



In [10]:

```
# Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
```

```
Accuracy of the network on the 10000 test images: 98.13 %
```

Inference: This model using ADAM optimiser instead of gradient descent outperforms all unregularised and the regularised relu activation model. This shows that ADAM optimiser is better than Gradient descent for learning parameters.