NEURAL NETWORKS FOR MNIST WITH PYTORCH

Ahmed Almohammed | 07/01/2025

ABOUT THE MNIST DATASET

MNIST consists of 28x28 pixel grayscale images of handwritten digits (0-9).

- Each image is labeled with its corresponding digit.
- We aim to build a model that classifies these images into the correct digit class (0-9).

LIBRARIES AND SETUP

Key libraries used for this project:

- **PyTorch** popular python framework for model building and training.
- **torchvision** popular datasets, model architecture, and common image transformations. We will be using it for loading the MNIST dataset.
- **matplotlib** for visualizing various graphs throughout the cycle.

CODE IMPORTS

```
import torch

import torch.nn as nn

from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from torch.optim import Adam

from torchvision.transforms.functional import to_tensor

import matplotlib.pyplot as plt
```

DATA PREPARATION

We load the MNIST dataset and convert images into tensors:

```
train_data = MNIST(root='./datasets', train=True, download=True, transform=to_tensor)
```

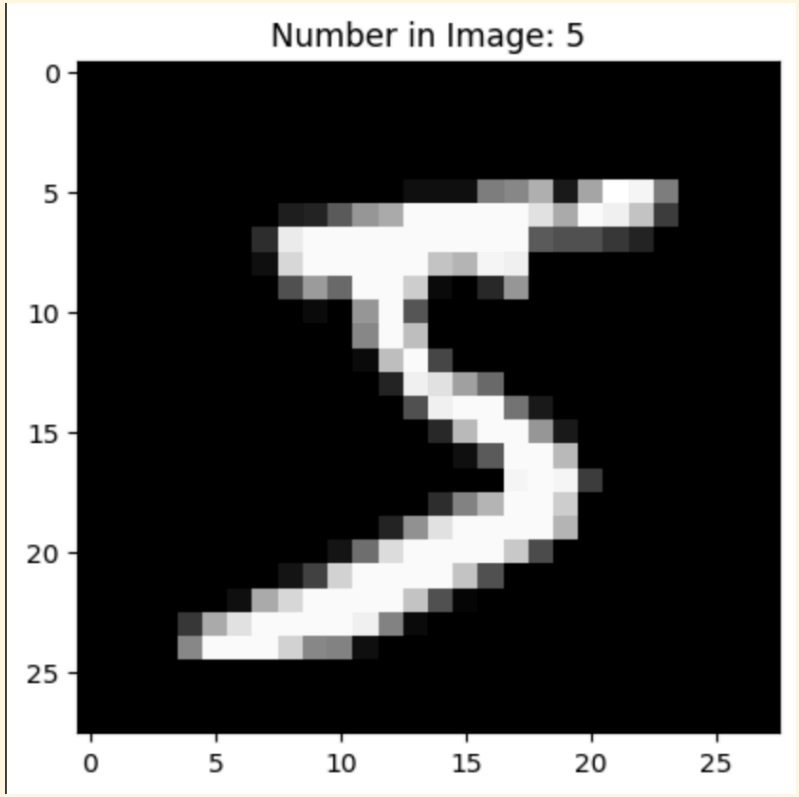
DataLoader is used to create batches of data for training:

```
train_loader = DataLoader(train_data, batch_size=64)
```

DATA PREPARATION

We load the MNIST dataset and convert images into tensors:

```
image = next(iter(train_loader))[0][0]
label = next(iter(train_loader))[1][0]
print(image.shape, label.shape)
# torch.Size([1, 28, 28]) torch.Size([])
# Image size is an array of [1, 28, 28]
# Label is an array of size []
# Size of [] mean single item
plt.title(f"Number in Image: {label.item()}")
plt.imshow(image.squeeze(), cmap='gray');
```



NEURAL NETWORK ARCHITECTURE

Our model is defined as a class, with the layers defined within. Each model (or class) inherits the forward function for the forward pass

```
class NN2Layer(nn.Module):
  def __init__(self, num_inp, num_hidden, num_out):
    super(NN2Layer, self).__init__()
    self.layer_1 = nn.Linear(num_inp, num_hidden)
    self.layer_2 = nn.Linear(num_hidden, num_out)
    self.hidden_activation = nn.ReLU()
  def forward(self, x):
    z1 = self.layer_1(x)
    a1 = self.hidden_activation(z1)
    z2 = self.layer_2(a1)
    return z2
```

TRAINING SETUP

Key parameters for training:

Epochs	Learning Rate	Optimiser	Loss Function	Device
12	1e-4	Adam	CrossEntropyLoss	GPU/CPU

```
num_epochs = 12
lr = 1e-4

device = 'cuda' if torch.cuda.is_available() else 'cpu'
train_losses = []
val_losses = []

model = NN2Layer(28*28, 32, 10)
optimizer = Adam(model.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()

model.to(device)
```

THE TRAINING LOOP

For each epoch, we train the model and compute the loss:

```
for epoch_no in range(num_epochs):

  model.train()
  epoch_weighted_loss = 0

  for batch_X, batch_y in train_loader:
    batch_X = batch_X.view(-1, 28*28).to(device)
    batch_y = batch_y.to(device)
    batch_y_probs = model(batch_X)
    loss = criterion(batch_y_probs, batch_y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    epoch_weighted_loss += (len(batch_y)*loss.item())

  epoch_loss = epoch_weighted_loss / len(train_loader.dataset)
  train_losses.append(epoch_loss)
```

EVALUATION

After training, we evaluate the model on the test set:

```
model.eval()
correctly_labelled = 0

with torch.no_grad():
  val_epoch_weighted_loss = 0

  for val_batch_X, val_batch_y in test_loader:
    val_batch_X = val_batch_X.view(-1, 28*28).to(device)
    val_batch_y = val_batch_y.to(device)
    val_batch_y_probs = model(val_batch_X)
    loss = criterion(val_batch_y_probs, val_batch_y)
    val_epoch_weighted_loss += (len(val_batch_y)*loss.item())
    val_batch_y_pred = val_batch_y_probs.argmax(dim=1)
    correctly_labelled += (val_batch_y_pred == val_batch_y).sum().item()

val_epoch_loss = val_epoch_weighted_loss/len(test_loader.dataset)
val_losses.append(val_epoch_loss)
```

Accuracy is calculated by comparing predicted and actual labels.

VISUALIZING LOSS

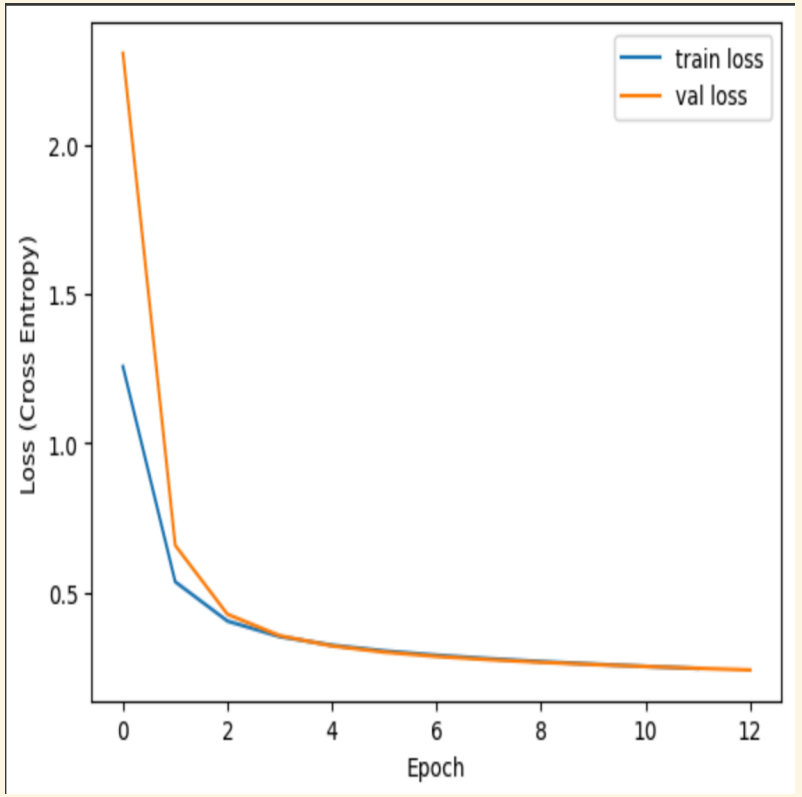
We track the loss over time to observe the model's learning:

```
plt.plot(train_losses, label='train loss')
plt.plot(val_losses, label='val loss')
plt.xlabel('Epoch')
plt.ylabel('Loss (Cross Entropy)')
plt.legend()
plt.show()
```

Keep in mind that losses is an array of numbers corresponding to the loss at each epoch. Example:

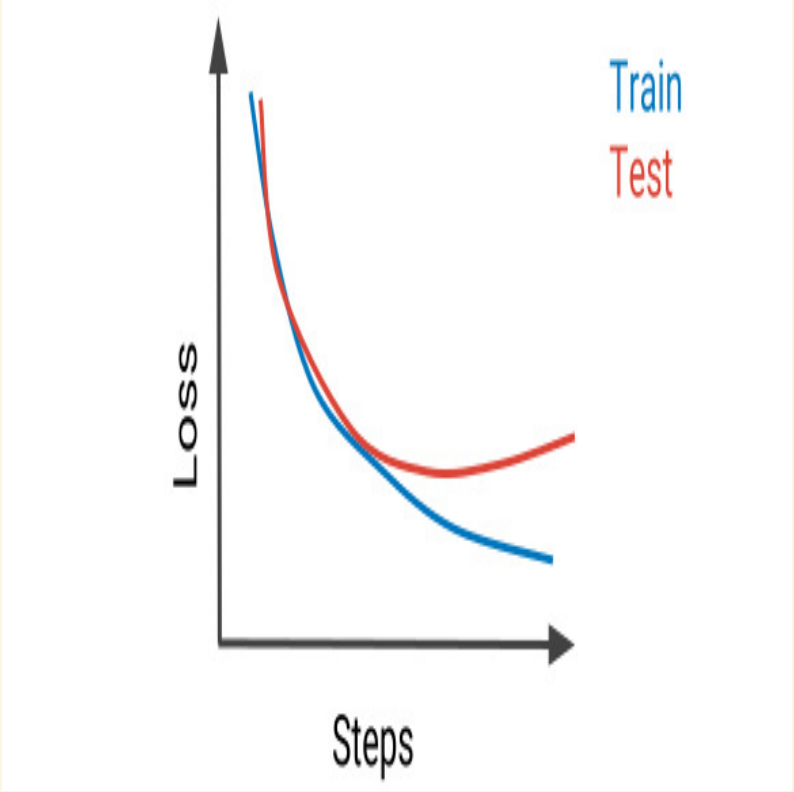
$$[2.1, 1.9, 1.7, 1.2, 0.8]$$

GENERATED LOSS



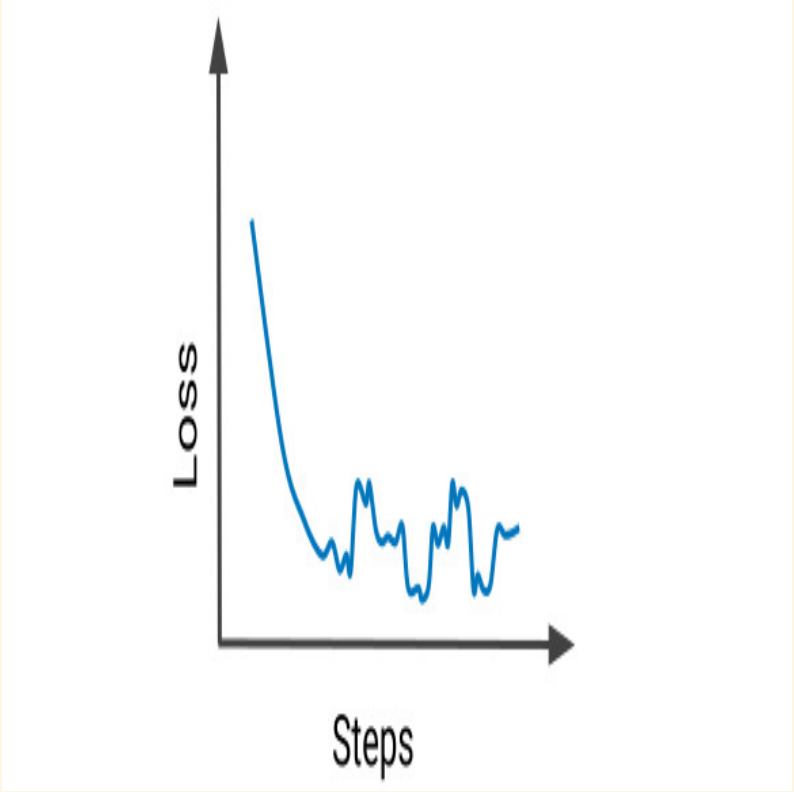
Looks promising!

GENERATED LOSS



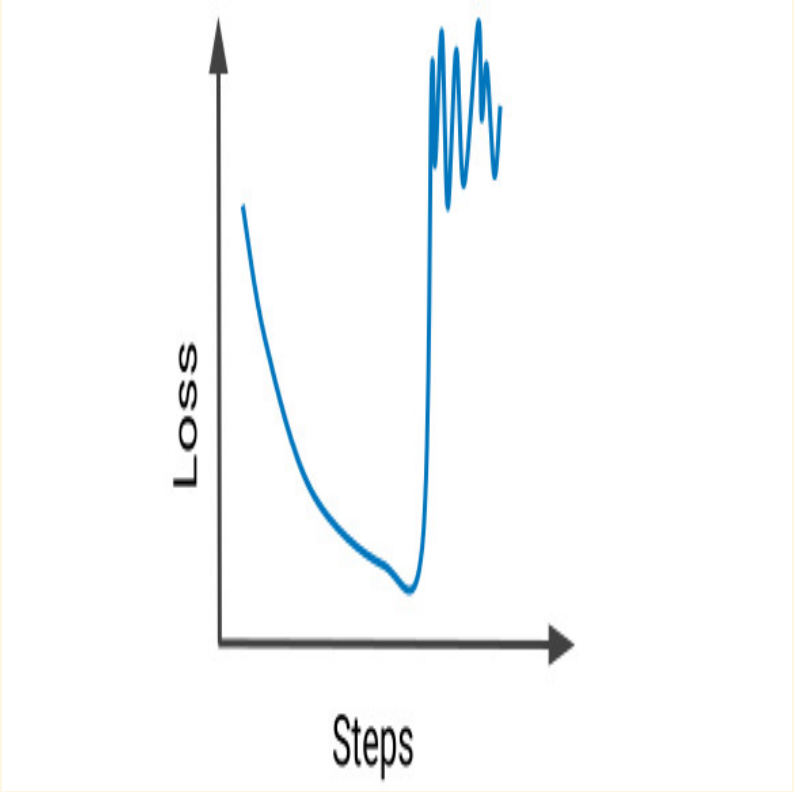
Overfitting!

GENERATED LOSS



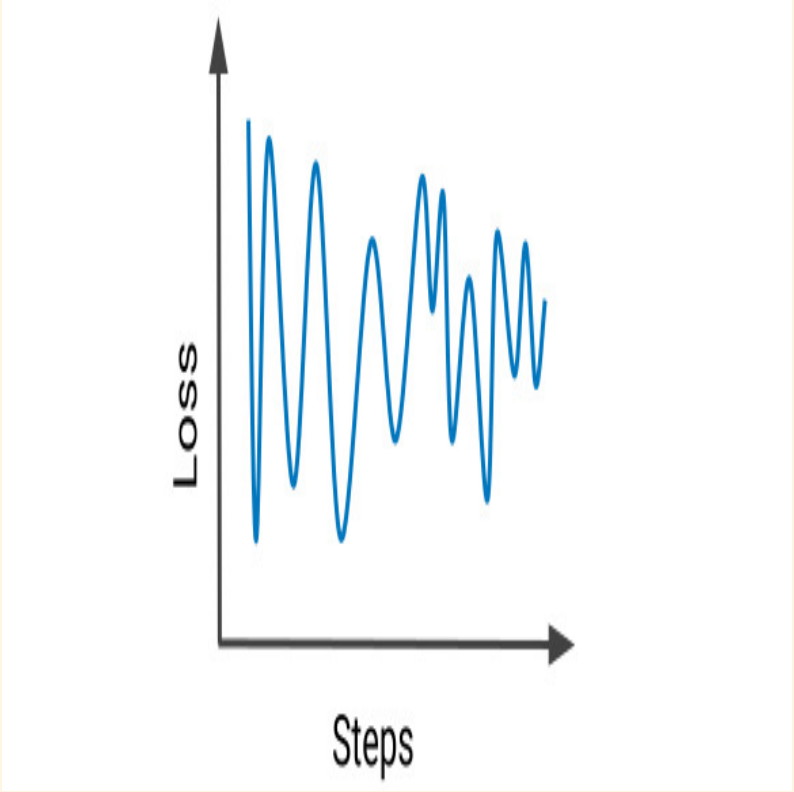
Training set with repetitions!

GENERATED LOSS



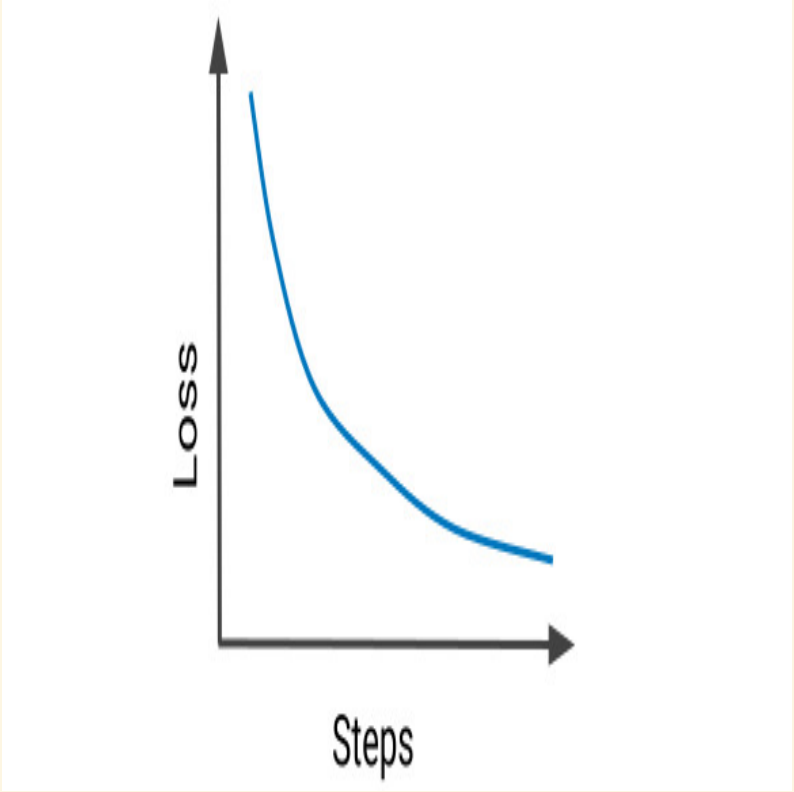
Dirty data 🤢 (outliers, NaNs)

GENERATED LOSS



Lower the learning rate!

GENERATED LOSS



Aim for this 😎!

SAVING AND LOADING THE MODEL

We save the model's state dict for later use:

```
torch.save(model.state_dict(), 'MNIST_classifier.pt')
```

To load the model:

```
loaded_model.load_state_dict(torch.load('MNIST_classifier.pt'))
```

CONCLUSION

ML probelm approached as follows:

- Problem Definition
- Data Collection
- Data Preprocessing
- Model Selection & Training
- Model Evaluation and & Validation

THE END

- Will push this as PDF for your reference in GitHub!
- Thank you for your attention.