7. BUILDING A ANN/MLP USING PyTorch

24 August 2025 02:48 AM

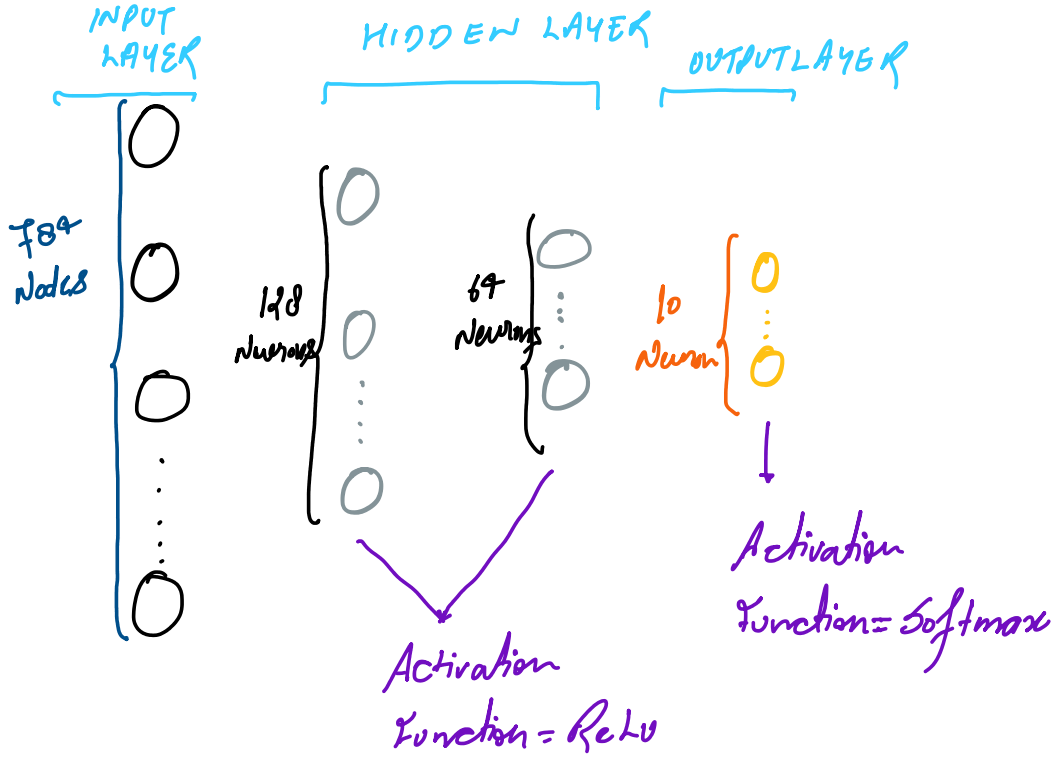
Dataset used: Fashion MNIST

↳ Total Images => 70,000

↓

We will use 6000 images

ARCHITETURE OF ANN:



WORK FLOW:

STEP 1:} Making DataLoader objects for both training data and test data

STEP 2:} Implementing training loop

STEP 3:} Model Evaluation Code on test data

Steps for PyTorch Training Pipeline

1) Reproducibility & Device

- **Set seeds**: *torch.manual_seed*, *np.random.seed*, *random.seed*.
- **Pick device**: *device = torch.device("cuda" if torch.cuda.is_available() else "cpu")*.

2) CustomDataset

Implement a class with **three required methods**:

- *__init__(self, data, targets, transform=None):*
  - Save arrays/paths, labels, and optional transforms/scalers.
- *__len__(self):*
  - Return number of samples.
- *__getitem__(self, idx):*
  - Load one sample (and label).
  - Return tensors (x, y).

3) Split Data

- Train/test split (e.g., *train_test_split* or index slicing).
- **Fit scalers on train only**; transform test with the same scaler.

4) DataLoaders

- Create *train_loader*, *test_loader* with appropriate *batch_size*, *shuffle=True* for train.
- Consider num_workers and *pin_memory=True* when on CUDA.

5) Define the Model

Create an nn.Module with:

- *__init__(self):*
  - Define layers (e.g., *nn.Linear*, *nn.ReLU*, *nn.Dropout*, etc.).

- *forward(self, x):*
  - Wire layers together and return predictions.

6) Hyperparameters

- Learning rate, epochs, batch size, weight decay, etc.

7) Loss, Optimizer

- Choose **criterion** (e.g., *nn.MSELoss* for regression, *nn.CrossEntropyLoss* for classification).
- Choose **optimizer** (e.g., *Adam* / *SGD*).

8) Training Loop (per epoch)

- *model.train()*.
- Training Loop:

```
for epoch in range(epochs):
    for batch_features, batch_labels in train_loader:
```

  - **Forward pass**: *pred = model(x)*.
  - **Compute loss**: *loss = criterion(pred, y)*.
  - **Zero grads**: *optimizer.zero_grad()*.
  - **Backward**: *loss.backward()*.
  - **Step**: *optimizer.step()*.

9) Test / Final Evaluation

- *model.eval()*.
- Wrap in *torch.no_grad()*.
- Iterate over *test_loader*, compute metrics (Accuracy, MAE, RMSE, etc.).

Building A ANN Using PyTorch

September 3, 2025

0.1 Fashion MNIST Classification with PyTorch

0.2 1. Import Required Libraries

[1]:
```
# Import pandas for data manipulation
import pandas as pd

# Import matplotlib for visualization
import matplotlib.pyplot as plt

# Import train_test_split for splitting data
from sklearn.model_selection import train_test_split

# Import torch and related modules for deep learning
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.optim as optim
```

0.3 2. Set Random Seed

Set the random seed for reproducibility of results.

[2]:
```
# Set random seed for reproducibility
torch.manual_seed(42)
```

[2]:
```
<torch._C.Generator at 0x19bfc627c30>
```

0.4 3. Load and Inspect Dataset

Load the Fashion MNIST dataset from a CSV file and inspect its structure.

[3]:
```
# Load Fashion MNIST dataset from CSV file
df = pd.read_csv('fmnist_small.csv')

# Display first few rows
df.head()
```

[3]:
```
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0      9       0       0       0       0       0       0       0       0
1      7       0       0       0       0       0       0       0       0
2      0       0       0       0       0       0       1       0       0
3      8       0       0       0       0       0       0       0       0
4      8       0       0       0       0       0       0       0       0

   pixel9  …  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0       0  …         0         7         0        50       205       196
1       0  …         0         0         0         0         0         0
2       0  …       142       142       142        21         0         3
3       0  …         0         0         0         0         0         0
4       0  …       213       203       174       151       188        10

   pixel781  pixel782  pixel783  pixel784
0       213       165         0         0
1         0         0         0         0
2         0         0         0         0
3         0         0         0         0
4         0         0         0         0

[5 rows x 785 columns]
```

[4]:
```
# Show info about the DataFrame (columns, types, nulls)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6000 entries, 0 to 5999
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 35.9 MB
```

0.5 4. Visualize Sample Images

Display a grid of the first 16 images from the dataset to understand the data visually.

[5]:
```
# Create a 4x4 grid of images
fig, axes = plt.subplots(
    4,
    4,
    figsize=(10, 10)
)

fig.suptitle("First 16 Images", fontsize=16)

# Plot the first 16 images from the dataset
for i, ax in enumerate(axes.flat):
    # Get image data and reshape to 28x28
```

```
    img = df.iloc[i, 1:].values.reshape(28, 28)
    # Show image in grayscale
    ax.imshow(img)
    # Hide axis for cleaner look
    ax.axis('off')
    # Show label as title
    ax.set_title(f"Label: {df.iloc[i, 0]}")

plt.tight_layout(rect=[0, 0, 1, 0.96])  # Adjust layout for title
plt.show()
```

0.6 5. Train-Test Split

Split the dataset into training and testing sets

[6]:
```
# Split features and labels
X = df.iloc[:, 1:].values
y = df.iloc[:, 0].values

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42
)
```

0.7 6. Feature Scaling

Scale the pixel values to the [0, 1] range for better model performance.

[7]:
```
# Scale features to [0, 1] range
X_train = X_train/255.0
X_test = X_test/255.0
```

0.8 7. Steps Overview

A summary of the steps involved in building and training the neural network.

0.9 STEPS

- ***Creating a CustomDataset Class***
- ***Creating train_dataset object***
- ***Creating test_dataset object***
- ***Creating train and test loader***
- ***Define Neural Network Class***
- ***Setting Learning Rate And Epochs***
- ***Instantiating The Model***
- ***Loss Function***
- ***Optimizer***
- ***Impliment Training Loop***
  - ***forward pass***
  - ***Loss Calculation***
  - ***Backward pass***
  - ***Updating Gradients***
- ***Set Model To eval Mode***
- ***Perform Evaluation***

0.10 8. Define Custom Dataset Class

Create a custom PyTorch Dataset class to handle feature and label access for batching.

[8]:
```python
# Define CustomDataset class for PyTorch
class CustomDataset(Dataset):

    def __init__(self, features, labels):
        # Convert features to float tensor
        self.features = torch.tensor(features, dtype=torch.float32)

        # Convert labels to long tensor
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        # Return number of samples
        return len(self.features)

    def __getitem__(self, index):
        # Return feature and label at given index
        return self.features[index], self.labels[index]
```

0.11 9. Create Dataset Instances

Instantiate training and testing datasets using the custom Dataset class.

[9]:
```python
# Create train_dataset and test_dataset objects
train_dataset = CustomDataset(
    X_train,
    y_train
)

test_dataset = CustomDataset(
    X_test,
    y_test
)
```

0.12 10. DataLoader for Batching

Wrap the datasets with PyTorch DataLoader to enable efficient batching and shuffling during training and evaluation.

[10]:
```python
# Create DataLoader for training and testing
train_loader = DataLoader(
    train_dataset,
    batch_size=32,
    shuffle=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=32,
```

```
    shuffle=False
)
```

[11]:
```
# Display the number of batches in the train DataLoader.
len(train_loader)
```

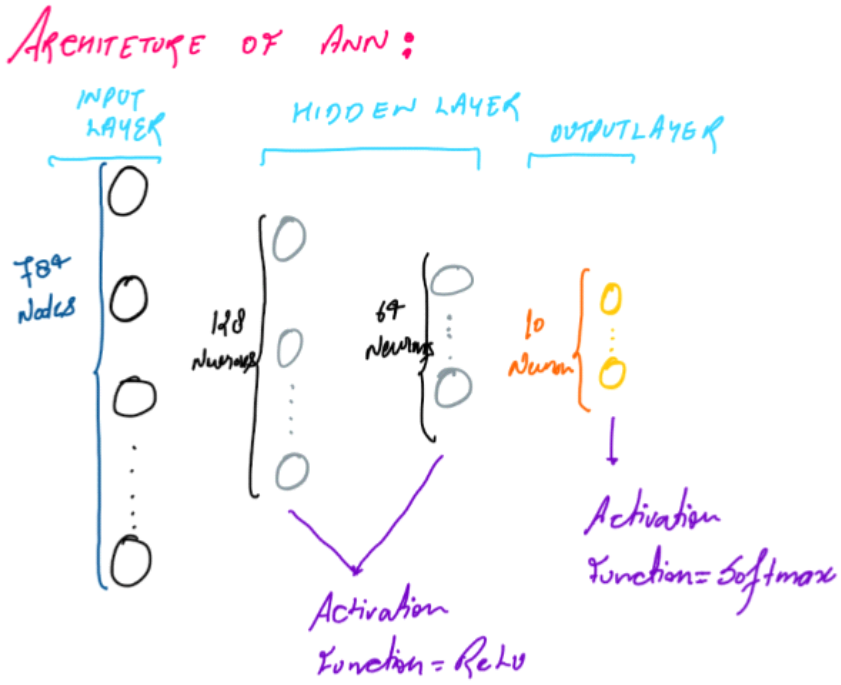
[11]: 150

[12]:
```
# Display the number of batches in the test DataLoader.
len(test_loader)
```

[12]: 38

0.13 11. Neural Network Architecture

Define the architecture of the artificial neural network using PyTorch's nn.Module and nn.Sequential.



[13]:
```
# Define neural network class
class MyNN(nn.Module):

    def __init__(self, num_features):
```

```
        # Call parent constructor
        super().__init__()
        # Define model architecture using nn.Sequential
        self.model = nn.Sequential(
            nn.Linear(num_features, 128),  # Input to first hidden layer
            nn.ReLU(),
            nn.Linear(128, 64),           # First hidden to second hidden
            nn.ReLU(),
            nn.Linear(64, 10)             # Second hidden to output layer (10 classes)
            # Softmax is handled internally by CrossEntropyLoss
            # Here explicityly we do not need to define the Softmax activation function for the output layer
            # It is implimented by deafult in the CrossEntropyLoss, internally.
        )

    def forward(self, x):
        # Forward pass through the model
        return self.model(x)
```

0.14 12. Set Training Hyperparameters

Specify the learning rate and number of epochs for model training.

[14]:
```
# Set number of epochs and learning rate
epochs = 100

learning_rate = 0.1
```

0.15 13. Model, Loss Function, and Optimizer

Instantiate the model, define the loss function (CrossEntropyLoss), and set up the optimizer (SGD).

[15]:
```
# Instantiate the model
model = MyNN(X_train.shape[1])

# Define loss function (CrossEntropy for multi-class classification)
criterion = nn.CrossEntropyLoss()

# Define optimizer (Stochastic Gradient Descent)
optimizer = optim.SGD(
    model.parameters(),
    lr=learning_rate
)
```

0.16 14. Training Loop

Iterate over epochs and batches, performing forward and backward passes, updating model parameters, and printing the average loss for each epoch.

[16]:
```
# Training loop
for epoch in range(epochs):

    total_epoch_loss = 0

    for batch_features, batch_labels in train_loader:

        # Forward pass: compute outputs
        outputs = model(batch_features)

        # Compute loss
        loss = criterion(outputs, batch_labels)

        # Zero gradients before backward pass
        optimizer.zero_grad()
        # Backward pass: compute gradients
        loss.backward()

        # Update model parameters
        optimizer.step()

        # Accumulate batch loss
        total_epoch_loss = total_epoch_loss + loss.item()

    # Calculate average loss for the epoch
    avg_loss = total_epoch_loss/len(train_loader)

    # Print epoch and average loss
    print(f'Epoch: {epoch + 1} , Loss: {avg_loss}')
```

```
Epoch: 1 , Loss: 1.3216368520259858
Epoch: 2 , Loss: 0.7793365579843521
Epoch: 3 , Loss: 0.6427524695793788
Epoch: 4 , Loss: 0.5751657489935557
Epoch: 5 , Loss: 0.5281801910201709
Epoch: 6 , Loss: 0.4952874990304311
Epoch: 7 , Loss: 0.46024329950412113
Epoch: 8 , Loss: 0.43594589988390603
Epoch: 9 , Loss: 0.4182921428481738
Epoch: 10 , Loss: 0.3984092238545418
Epoch: 11 , Loss: 0.38502645591894785
Epoch: 12 , Loss: 0.372268552283446
Epoch: 13 , Loss: 0.34773620883623757
Epoch: 14 , Loss: 0.34672420596083003
```

```
Epoch: 63 , Loss: 0.09160653349012136
Epoch: 64 , Loss: 0.06587979760952294
Epoch: 65 , Loss: 0.08712220632781585
Epoch: 66 , Loss: 0.06363527670657883
Epoch: 67 , Loss: 0.06208949364721775
Epoch: 68 , Loss: 0.05667706207217028
Epoch: 69 , Loss: 0.06815310258573543
Epoch: 70 , Loss: 0.0754167199631532
Epoch: 71 , Loss: 0.14631125033212206
Epoch: 72 , Loss: 0.08233193612347046
Epoch: 73 , Loss: 0.12243214213754981
Epoch: 74 , Loss: 0.05412741618386159
Epoch: 75 , Loss: 0.04562157000259807
Epoch: 76 , Loss: 0.03563245271798223
Epoch: 77 , Loss: 0.030208402726178366
Epoch: 78 , Loss: 0.05329750189868113
Epoch: 79 , Loss: 0.03239929866666595
Epoch: 80 , Loss: 0.036115992741736894
Epoch: 81 , Loss: 0.03408303061965853
Epoch: 82 , Loss: 0.03221908880048432
Epoch: 83 , Loss: 0.019485433438482382
Epoch: 84 , Loss: 0.021556298608581225
Epoch: 85 , Loss: 0.015287858942853443
Epoch: 86 , Loss: 0.01173053629405331
Epoch: 87 , Loss: 0.014901724323184075
Epoch: 88 , Loss: 0.016724679251007426
Epoch: 89 , Loss: 0.021482379210065117
Epoch: 90 , Loss: 0.06205682110429431
Epoch: 91 , Loss: 0.18101994422924084
Epoch: 92 , Loss: 0.10467932317095498
Epoch: 93 , Loss: 0.045464055612683296
Epoch: 94 , Loss: 0.04182172857923433
Epoch: 95 , Loss: 0.024459266438304136
Epoch: 96 , Loss: 0.033169551788208386
Epoch: 97 , Loss: 0.050397921473486346
Epoch: 98 , Loss: 0.018161742905310044
Epoch: 99 , Loss: 0.04232000354114765
Epoch: 100 , Loss: 0.014427155489684082
```

0.17 15. Set Model to Evaluation Mode

Switch the model to evaluation mode to disable dropout and other training-specific layers.

[17]:
```
# Set model to evaluation mode (disables dropout, etc.)
model.eval()
```

[17]:
```
MyNN(
  (model): Sequential(
```

```
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

0.18 16. Model Evaluation

After training, set the model to evaluation mode and compute the accuracy on the test set. For each batch, the model predicts the class labels, which are compared to the true labels to calculate the overall accuracy.

[18]:
```python
# Evaluation code
total = 0
correct = 0

with torch.no_grad():
    # Iterate over test batches
    for batch_features, batch_labels in test_loader:

        # Forward pass: get outputs
        outputs = model(batch_features)

        # Get predicted class (highest score)
        _, predicted = torch.max(outputs, 1)

        # Update total and correct counts
        total = total + batch_labels.shape[0]
        correct = correct + (predicted == batch_labels).sum().item()

# Print accuracy
print(correct/total)
```

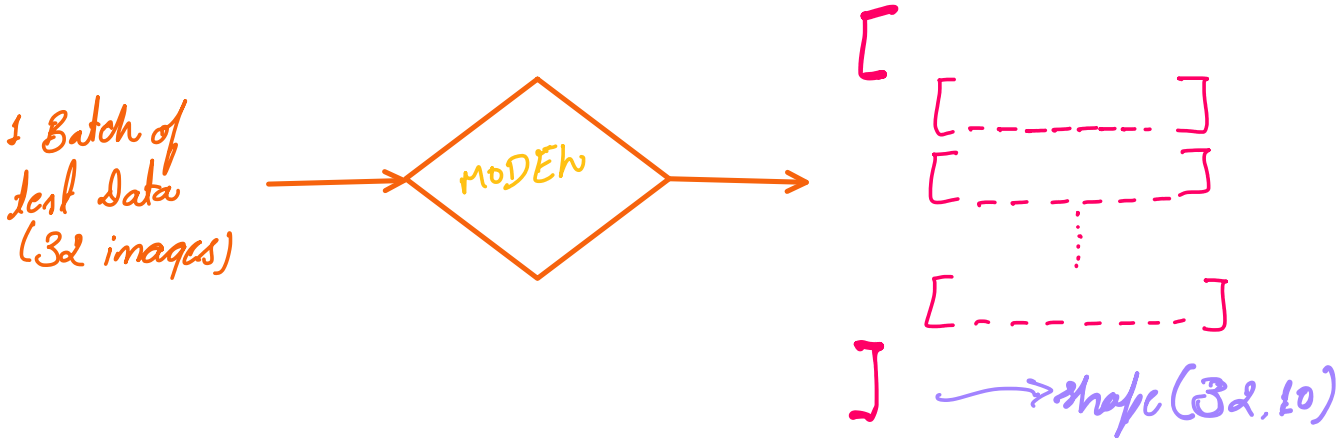
```
0.8175
```

Adding 32 each time → Batch Size

outputs = model(Batch_features)

so now we have a model, which has been trained as of now.

and now we are sending 1 batch of test data in the model i.e. we are sending 32 images in a go



now the model will give an output corresponding to each of the 32 images

and this output will be a tensor consisting of 10 values for every image

Because our data has 10 classes

so it will give a probability value corresponding to each class for each one of the image.

something like:

[
Probability value corresponding to 1st image
2nd image
3rd image
[-5.1, -6.9, 3.5, 0.4, 8.9, -1.8, 12.9
-8.4, -2.5, -3.09]
⋮

:

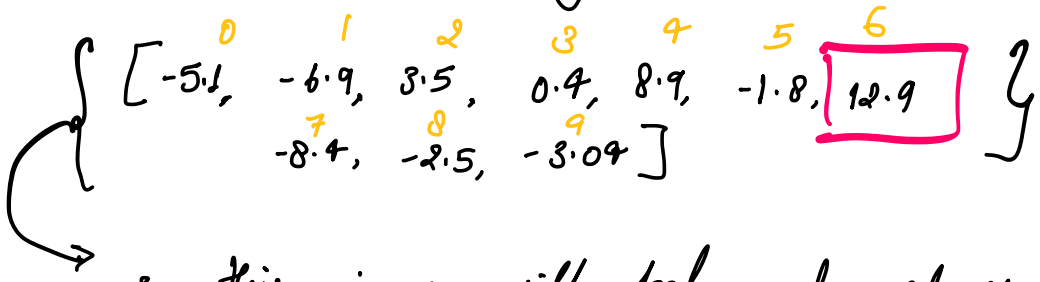
[ ]

]

(32, 10)

Now next we have to do extract labels from these tensors, that is, a particular image belongs to which class

Therefore for this we'll compare the probability value of each class and take out the class whose probability value is maximum



Now Different ways to increase our achieved Accuracy :-

1.) We can use full dataset
↳ till now we used only 6000 images out of 60,000

2.) Try out different optimizers

3.) Try out different learning rate values

4. Try out training for more epoches

5. We can also use different weight initialization techniques

6. We can also use the concept of regularization

7. We can also use the concept of drop-out, Batch normalization

etc. etc. etc.

8. We can also experiment with model's architecture by doing hyperparameter tuning