Faculty of Computers and Information Technology

Machine Learning

Project Report

Analyzing & Comparing the Results of An FNN & a CNN on The PlantVillage Dataset

Students:

Harethah Abu Shairah.

Mazen Tuahrri.

Omar Abohmoud.

Instructor: Dr. Osama Alia

Introduction

In This project, our aim was to create a Neural Network and train on part of the PlantVillage dataset from Kaggle, a set of (3, 256, 256) images , so we opted to create two separate models, a fully connected feed forward NN and a Convolutional NN, and then compare their structures and see how they would score on the test data.

Required Python Libraries

- Numpy
- Pytorch
- Matplotlib
- PIL

Work Plan

In order to be able to complete our project, we had to follow a set of steps to achieve the desired results, these are:

1. Download the dataset from Kaggle
2. Create a dataset class to store the data
3. Split the data into Train, Test, and Validation datasets
4. Create Pytorch DataLoaders from the datasets
5. Create the structures of the FNN & CNN models
6. Create our loss function and optimization algorithm
7. Iterate through the train dataset to optimize the models
8. Test the models and calculate their accuracy

Creating and Managing the Dataset

After downloading the dataset from Kaggle, we needed to implement a dataset class to store it and manage it, we achieved that by implementing a class that inherits from Pytorchs own Dataset class, and then do the necessary modifications to match our need.

```
class MyData(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.samples = []

        for class_label, class_name in enumerate(os.listdir(root_dir)):
            class_dir = os.path.join(root_dir, class_name)
            if os.path.isdir(class_dir):

                for image_name in os.listdir(class_dir):
                    image_path = os.path.join(class_dir, image_name)
                    self.samples.append((image_path, class_label))

    def __len__(self):
        return len(self.samples)
```

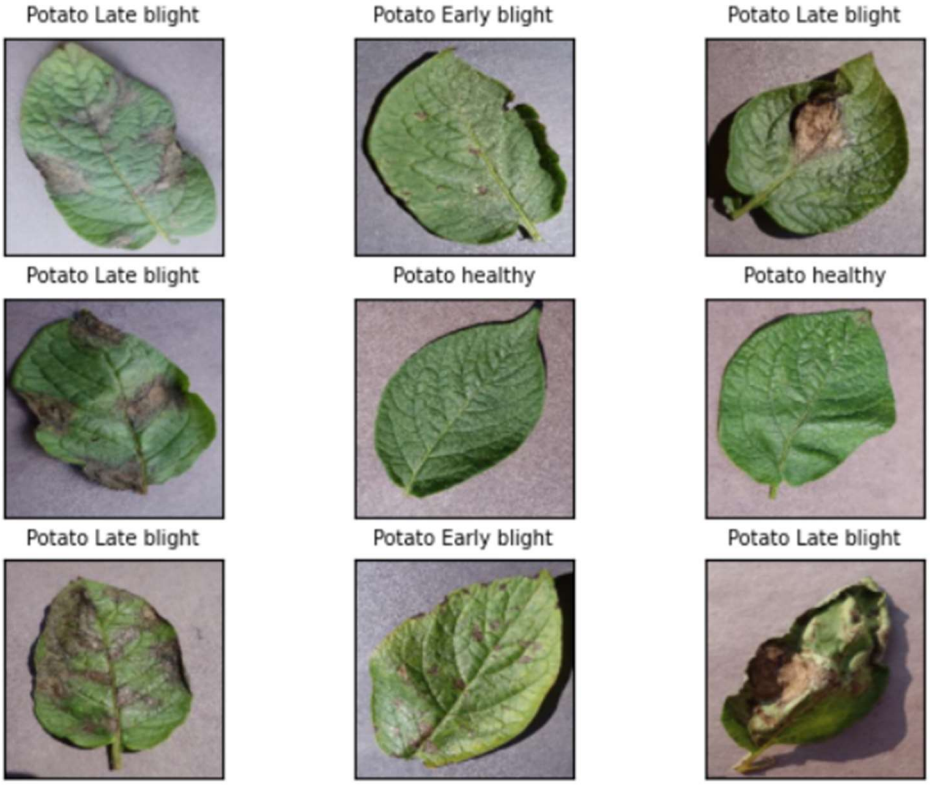
```
    def __getitem__(self, idx):
        image_path, class_label = self.samples[idx]

        image = Image.open(image_path)
        if self.transform:
            image = self.transform(image)

        return image, class_label
```

And after creating the dataset, we split it into Train, Test, and Validation sets, and form each set we created a DataLoader to manage iteration.

With all that being done, we reviewed the dataset to make everything as working as intended



Designing Model Structures

As we discussed earlier, we wanted to create two models and compare their results, the first was a Feed Forward Neural Network, we created a 4 layers network with Sigmoid as the activation function.

```
class FNN(nn.Module):
    def __init__(self):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(256*256*3, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 128)
```

```
        self.fc4 = nn.Linear(128, 3)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x.view(-1, 256*256*3)
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        x = self.fc4(x)
        return x
```

The second was a Convolutional Neural Network that uses 3 convolutional layers and 2 linear layers, with Relu for activation, we also implemented Batch Normalization and Dropout for the model to improve results.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, 5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(16)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(32)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(32, 64, 5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(64)
        )
        self.fc1 = nn.Linear(64*32*32, 512)
        self.fc2 = nn.Linear(512, len(classes))
        self.dropout = nn.Dropout(0.2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)

        x = torch.flatten(x, 1)
        x = self.dropout(self.relu(self.fc1(x)))
```

```
        x = self.fc2(x)
        return x
```

While these models might seem simple, the fact that we are dealing with (3, 256, 256) images meant our models would have a large number of trainable parameters, with the CNN model have more then 30 million parameters, and the FNN having even more than that.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 16, 128, 128]        --
|    └─Conv2d: 2-1                       [-1, 16, 256, 256]        1,216
|    └─ReLU: 2-2                         [-1, 16, 256, 256]        --
|    └─MaxPool2d: 2-3                    [-1, 16, 128, 128]        --
|    └─BatchNorm2d: 2-4                  [-1, 16, 128, 128]        32
├─Sequential: 1-2                        [-1, 32, 64, 64]          --
|    └─Conv2d: 2-5                       [-1, 32, 128, 128]        12,832
|    └─ReLU: 2-6                         [-1, 32, 128, 128]        --
|    └─MaxPool2d: 2-7                    [-1, 32, 64, 64]          --
|    └─BatchNorm2d: 2-8                  [-1, 32, 64, 64]          64
├─Sequential: 1-3                        [-1, 64, 32, 32]          --
|    └─Conv2d: 2-9                       [-1, 64, 64, 64]          51,264
|    └─ReLU: 2-10                        [-1, 64, 64, 64]          --
|    └─MaxPool2d: 2-11                   [-1, 64, 32, 32]          --
|    └─BatchNorm2d: 2-12                 [-1, 64, 32, 32]          128
├─Linear: 1-4                            [-1, 512]                 33,554,944
├─ReLU: 1-5                              [-1, 512]                 --
├─Dropout: 1-6                           [-1, 512]                 --
├─Linear: 1-7                            [-1, 3]                   1,539
==========================================================================================
Total params: 33,622,019
Trainable params: 33,622,019
...
Forward/backward pass size (MB): 17.50
Params size (MB): 128.26
Estimated Total Size (MB): 146.51
```

Model Parameters at each layer for the CNN model

Training and Optimization

To be able to train our models, we need 2 very important things:

1. A loss function to measure how well our model is doing
2. An optimizing algorithm to be able to reduce the loss with each training iteration

For the loss function we used the Cross Entropy loss function, which is one of the best for multi-class classification, and for optimization we used an algorithm called ADAM which is an improved version of Gradient Decent that implements Momentum and Adaptive Learning Rate.

```
loss_function = nn.CrossEntropyLoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

Training and Validation

In order to train a Neural Network algorithm, we have to follow a set of steps:

1. Decide on our hyper parameters, like learning rate and number of epochs
2. Run a forward pass through the network to get y hat
3. Calculate the loss on y hat with respect to the actual labels
4. Go through Backward Propagation and calculate the derivatives of the loss with respect to each parameter
5. Update the parameters by taking a small step in the direction of the negative gradient.
6. Keep track of training and validation losses to avoid over and under fitting

Here is how a training loop can be implemented using Pytorch:

```
epochs = 5
train_losses, valid_losses = [], []

for e in range(epochs):
    running_loss = 0
    val_loss = 0
    for i, data in enumerate(trainloader):
        model.train()
        images, labels = data
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()

        outs = model(images)
        loss = loss_function(outs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    with torch.no_grad():
        model.eval()
        for j, valdata in enumerate(valloader):
            images, labels = valdata
            images, labels = images.to(device), labels.to(device)
            outs = model(images)
            loss = loss_function(outs, labels)
            val_loss += loss.item()
        print(f"Epoch: {e+1}/{epochs}.. "
        f"Training Loss: {running_loss/len(trainloader):.3f}.. "
        f"Validation Loss: {val_loss/len(valloader):.3f}.. ")
        train_losses.append(running_loss/len(trainloader))
        valid_losses.append(val_loss/len(valloader))
        running_loss = 0
```

```
        val_loss = 0
```

Testing and Analyzing the Results

To get a better understanding of how well our model actually preforms, it must be tested on unseen data, to make sure that our model can generalize well, that is why we split the data earlier.

Now to compare the results of each model, we need to look at two metrics, the first is the Loss, as we train our model, we expect the loss to decrease until we get close to a minima, of we achieve that we know that our model has trained well.

Before we review the results, it as expected that the Dense model would preform worse than the Convolutional one, and that is because dense NNs are not well suited to work with images, on the other hand, CNNs we designed with primary goal of working on images.

The loss as we iterate through the FNN model:

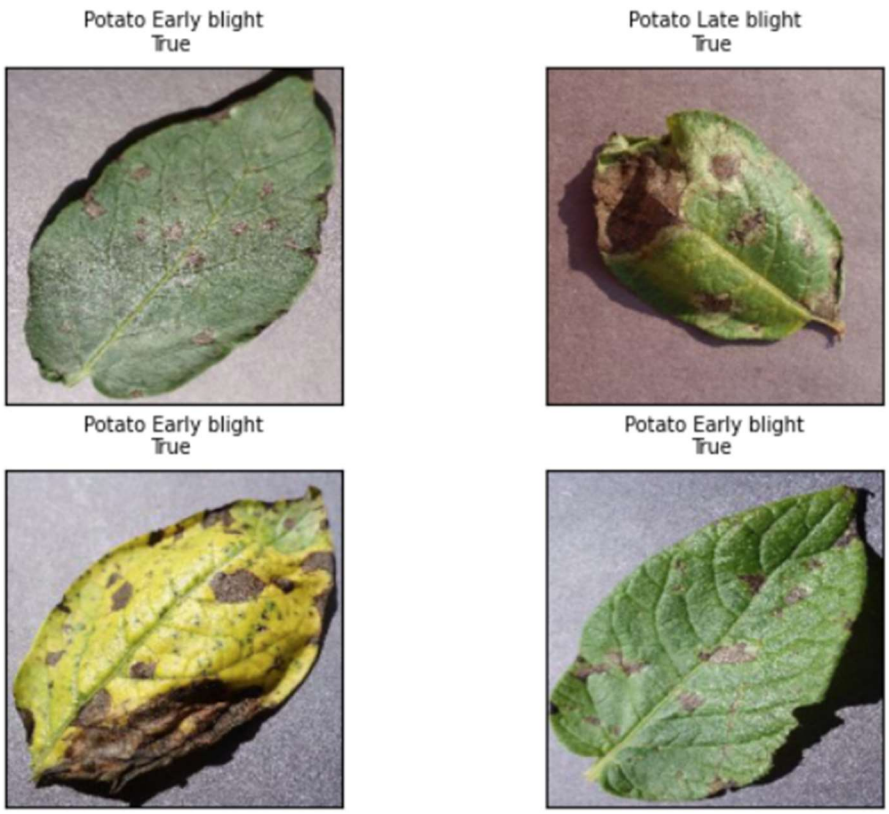
```
Epoch: 1/5.. Training Loss: 1.084.. Validation Loss: 1.084..
Epoch: 2/5.. Training Loss: 1.084.. Validation Loss: 1.084..
Epoch: 3/5.. Training Loss: 1.084.. Validation Loss: 1.083..
Epoch: 4/5.. Training Loss: 1.085.. Validation Loss: 1.084..
Epoch: 5/5.. Training Loss: 1.086.. Validation Loss: 1.083..
```

The loss as we iterate through the CNN model:

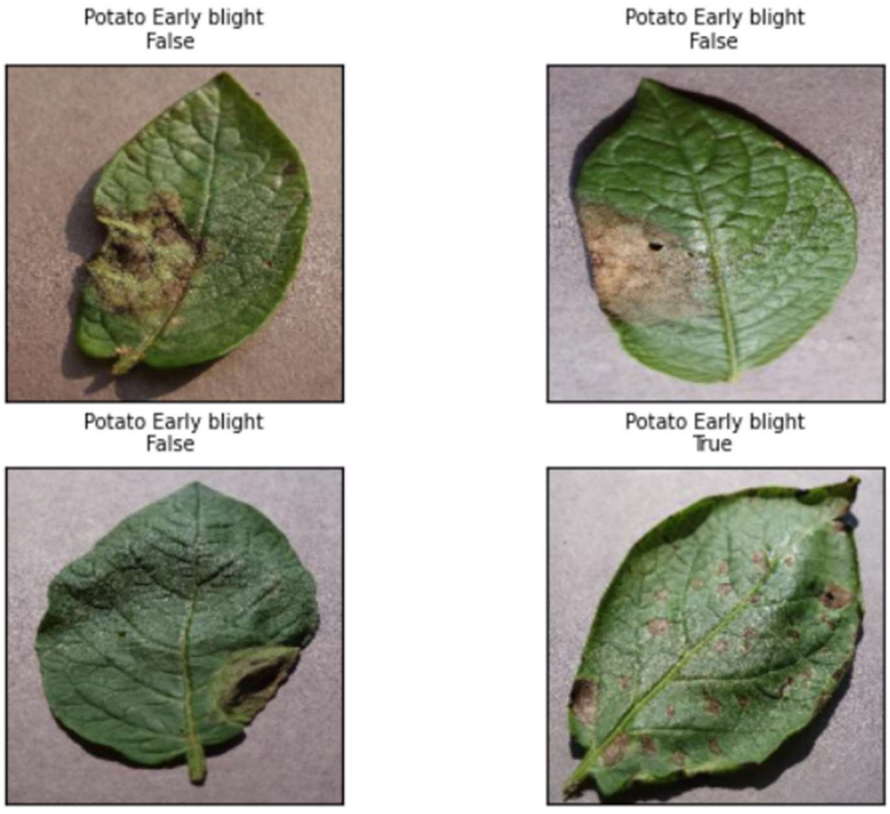
```
Epoch: 1/5.. Training Loss: 0.335.. Validation Loss: 7.721..
Epoch: 2/5.. Training Loss: 0.057.. Validation Loss: 3.152..
Epoch: 3/5.. Training Loss: 0.017.. Validation Loss: 0.386..
Epoch: 4/5.. Training Loss: 0.008.. Validation Loss: 0.113..
Epoch: 5/5.. Training Loss: 0.008.. Validation Loss: 0.162..
```

we can see from the output above how the CNN model trains really well, with the loss decreasing each iteration, on the other hand, our FNN models does not, the loss is barely changing and the model is not making any progress.

These results are also reflected in the accuracy of the models, our CNN model managed to score 97% accuracy on the testing dataset, while the FNN model only manged 41%



The predicted labels form the CNN model and their truth value



The predicted labels form the FNN model and their truth value