Nick Marks

Chess Pieces Classifier

A demonstration and walkthrough of the code can be found on YouTube: https://www.youtube.com/watch?v=GcBdZo7DuJU.

Additional code and materials like the dataset used can be found at https://github.com/nmarks98/chess-classifier. The `chess_classifier.py` script at this GitHub link is the script shown in the demo video.

The goal of this project is to create a program that takes an image of a single chess piece and output the name of that piece. This Jupyter notebook creates and trains the model used for the classification and the `chess_classifier.py` script uses the model to classify the chess piece in a provided image. . Originally, my project proposal suggested that I would make a program that classifies an entire chess board, however this has proven to take longer than expected and I needed to scale the project back slightly. However, this project could be adapted to classify an entire chess board without much trouble. The only additional step that is needed is to create another program that breaks an image of a chess board into 64 images (1 image for each square on the board) and classify each of the images.

In [ ]:
```
import torch
from torch import nn, optim
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder

import numpy as np
import warnings
```

In [ ]:
```
# Setup torch device, using GPU if its available
# Training with the CPU on my laptop is very very slow, so using a GPU (perhaps with Google colab) is preferred
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    warnings.warn("It is recommended to use train on a GPU, perhaps through Google colab, for performance")
    device = torch.device("cpu")
print(f"Using {device}")
```

```
Using cpu
/tmp/ipykernel_48160/4162326941.py:6: UserWarning: It is recommended to use train on a GPU, perhaps through Google colab, for performance
  warnings.warn("It is recommended to use train on a GPU, perhaps through Google colab, for performance")
```

Setup

To begin, we will start by importing the data. The dataset used in this project was found on Kaggle (link) and contains labelled images of chess pieces, including both digital and real images. This is good since I would like my model to have the ability to classify images of chess pieces from online games and well as live over-the-board chess games.

The code below imports the data from a chess_pieces directory in the same directory and this project, and separates it into training and validation data. It then creates PyTorch DataLoader objects from the data to be used later on. At this point, this is mostly "boilerplate" PyTorch.

Transforming the Data

Some *transformations* were applied to the data before proceeding which seek to improve the model's ability to classify the images, and make things easier later on. Here I have chosen to apply the RandomHorizontalFlip and RandomRotation transforms. The ToTensor transform is just necessary for using PyTorch.

In [ ]:
```
# Define transformations for training
input_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomHorizontalFlip(p = 0.4),
    transforms.RandomRotation(30),
])
```

In [ ]:
```
# Load in the dataset
dataset_path = "./datasets/chess_pieces"
dataset = ImageFolder(dataset_path, transform=input_transforms)

train_data, val_data = torch.utils.data.random_split(
    dataset,
    [int(len(dataset)*0.8), len(dataset) - int(len(dataset)*0.8)]
)

val_data, test_data = torch.utils.data.random_split(
    val_data,
    [int(len(val_data)*0.8), len(val_data) - int(len(val_data)*0.8)]
)

train_loader = DataLoader(train_data, batch_size = 16, shuffle = True)
val_loader = DataLoader(val_data, batch_size = 16, shuffle = True)
test_loader = DataLoader(test_data, batch_size = 1, shuffle = True)
test_loader_ordered = DataLoader(test_data, batch_size = 1, shuffle = False)
```

Defining a Model

Throughout our machine learning studies this quarter, we have not discussed deep learning, seeing as that is the topic of next quarter, however, in order to obtain reasonable results for a wide range of chess pieces, I have decided to use a convolutional neural network (CNN) for the model. This is because CNNs are notoriously good at classifying images and although we didn't learn much about them, many of the topics we did learn about still apply.

To implement a CNN in PyTorch, we define a Python class that inherits from the nn.Module class. In this case, I have chosen to use the resnet50 model from the torchvision library. Although the mathematics behind this model (a residual neural network, which is a special case of a convolutional neural network) is beyond the scope of my understanding and our EE475 course, I have chosen to use it since implementing it with PyTorch was no more difficult that using another type of model, and residual neural networks have shown to be extremely good at classifying images.

In [ ]:
```
# Define a neural network as a class that inherits from the torch.nn.Module class
class ChessCNN(nn.Module):
    def __init__(self):
        super(ChessCNN, self).__init__()

        # use Resnet, a deep neural network model, which is particularly good for image classification
        self.model = torchvision.models.resnet50(pretrained = True)

        for parameter in self.model.parameters():
            parameter.requires_grad = False

        # Define the model of each layer TODO: is this correct?
        self.model.fc = nn.Sequential(
            nn.Linear(2048, 1000),
            nn.ReLU(),
            nn.Linear(1000, 5)
        )

    # forward propogation step
    def forward(self, x):
        x = self.model(x)
        return x
```

Defining Training Parameters

The model is first instantiated for later use, then we define several parameters like the learning rate, number of iterations, the optimizer that we will use, and the loss function.

Learning Rate

Here we set the learning rate to 0.001. This was chosen after testing several learning rates both higher and lower. Higher learning rates result in faster training, however the loss oscillates or doesn't reach as small a value. An even smaller learning rate may be better, however I have access to only a limited amount of computing power (GPU access through Google Colab) so this learning rate is sufficient for this project.

Number of iterations The number of iterations was chosen mostly because of time considerations but also and you can see from the output of the training step later on, after several hundred iterations, there are very few new best weights that are found that make the loss function any smaller.

Optimizer There are many choices for the optimizer to use for this problem. I have chosen two different optimizers to try for this project and to compare performance. The first optimizer I tried was Stochastic Gradient Descent (SGD), which is much like the standard gradient descent (GD) algorithm we have been using in class, however it will update weights faster than basic gradient descent since it doesn't need to step through the entire training set to update weights. SGD results in many more oscillations but faster training than GD.

The second optimizer I tried was the Adam optimizer. This optimizer is an special implementation of SGD that is based on adaptive estimations of first and second order moments and can be shown to perform better for some problems.

In [ ]:
```
model = ChessCNN() # instantiate the neural net class
# learning_rate = 0.00001 # define the learning rate
learning_rate = 0.001
max_its = 1000

# Define the optimizer
# optimizer = optim.Adam(model.parameters(), lr = learning_rate, weight_decay = 0.001)
optimizer = optim.SGD(model.parameters(),lr=learning_rate,momentum=0.01)

# Define the loss function
loss_func = nn.CrossEntropyLoss() # use cross entropy loss function
min_loss = np.inf
model.to(device) # set model to use the appropriate defice (GPU or CPU)
```

Training

To train the model using the chosen loss function and optimizer (SGD or Adam) I created the function below to make it easier. The training function loops through the data in the training set, computes the loss, and updates the weights. It then uses the validation set to check the performance at each step and if its better than the previous best performing model, it will save the current model as the new best.

In [ ]:
```
def train(model, max_its, min_loss):
    for step in range(max_its):
        training_loss = 0
        model.train()
        for images, labels in train_loader:
            optimizer.zero_grad()
            images = images.to(device)
            labels = labels.to(device)
            yp = model(images)
            loss = loss_func(yp, labels)
            loss.backward()
            optimizer.step()
            training_loss += loss.item()

            del images, labels
            torch.cuda.empty_cache()


        valid_loss = 0
        valid_accuracy = 0
        model.eval()
        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                labels = labels.to(device)
                yp = model(images)
                loss = loss_func(yp, labels)
                valid_loss += loss.item()
                yp = nn.Softmax(dim = 1)(yp)
                _, top_class = yp.topk(1, dim = 1)
                num_correct = top_class == labels.view(-1,1)
                valid_accuracy += num_correct.sum().item()

                # clear data to ensure there isn't confusion
                del(images)
                del(labels)
                torch.cuda.empty_cache()

        # print("Step: {} \tTraining loss: {:.4f} \tValidation loss: {:.4f} \tAccuracy: {:.2f}%".format(step, training_loss, valid_loss, (valid_accuracy/len(val_data))*100))
        print(f"Step: {step} \tTraining loss: {training_loss:.4f} \tValidation loss: {valid_loss:.4f} \tAccuracy: {valid_accuracy/len(val_data)*100:.2f}%")

        # whenever a new minimum loss for the model is found replace the previous best model
        if valid_loss <= min_loss:
            print(f"New minumum loss found! = {valid_loss:.4f}\tSaving model...")
            torch.save(model.state_dict(), "trained_model.pt")
            min_loss = valid_loss # set new minumum loss
```

Training

Running the cell below will begin the training process. This can be very slow, especially on an average CPU. Therefore, to train the model I have chosen to use Google Colab which offers free (but limited) GPU usage.

In [ ]:
```
train(model, max_its, min_loss)
```

Results

In the cells below we evaluate the models, (one with the Adam optimizer and one with the SGD optimizer) with the testing data set. As you can see from the printout of the results, both models perform very similarly, so for this project, the SGD and Adam optimizer are comparable.

The final accuracy of the trained models tops out at just under 80%. This is decent performance considering the dataset and the training time. Better performance most likely could be achieved with a larger more comprehensive dataset and longer training time to allow the loss to decrease even further. The training time and computational requirements were definitely a limitation on this project, since to achieve these results, training took at least 1 hour running on Google Colab's provided GPU.

In [ ]:
```
total_correct = 0
count = 0
classes = dataset.classes
model.load_state_dict(torch.load('./model_SGD.pt',map_location=device))

print("************************")
print("Results with SGD optimizer:")
print("************************")

with torch.no_grad():
    model.eval()
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        yp = model(images)
        yp = nn.Softmax(dim = 1)(yp)
        top_p, top_class = yp.topk(1, dim = 1)
        eq = top_class == labels.view(-1, 1)
        # print(classes[top_class.item()])
        total_correct += eq.sum().item()

        if count % 1 == 0:
            print("Prediction: {}\t Confidence: {:.2f}%\t Actual label: {}".format(classes[top_class.item()], top_p.item() * 100, classes[labels.item()]))
        else:
            print(f"count%1 = {count % 1}")
        count += 1

print("Accuracy:\n========")
print(f"Score {total_correct}/{len(test_data)}")
print(f"Accuracy: {(total_correct/len(test_data)) * 100:.2f}%")
```

```
************************
Results with SGD optimizer:
************************
Prediction: Rook-resize  Confidence: 43.09%      Actual label: pawn_resized
Prediction: knight-resize        Confidence: 99.70%      Actual label: knight-resize
Prediction: Rook-resize  Confidence: 86.18%      Actual label: Rook-resize
Prediction: Queen-Resized        Confidence: 89.27%      Actual label: Queen-Resized
Prediction: bishop_resized       Confidence: 92.43%      Actual label: bishop_resized
Prediction: knight-resize        Confidence: 72.78%      Actual label: knight-resize
Prediction: knight-resize        Confidence: 76.24%      Actual label: knight-resize
Prediction: Queen-Resized        Confidence: 85.56%      Actual label: Queen-Resized
Prediction: Queen-Resized        Confidence: 89.55%      Actual label: Queen-Resized
Prediction: Queen-Resized        Confidence: 58.92%      Actual label: bishop_resized
Prediction: Queen-Resized        Confidence: 68.47%      Actual label: Queen-Resized
Prediction: bishop_resized       Confidence: 78.29%      Actual label: bishop_resized
Prediction: knight-resize        Confidence: 77.08%      Actual label: knight-resize
Prediction: Rook-resize  Confidence: 99.44%      Actual label: Rook-resize
Prediction: knight-resize        Confidence: 86.49%      Actual label: knight-resize
Prediction: knight-resize        Confidence: 91.18%      Actual label: knight-resize
Prediction: Queen-Resized        Confidence: 74.00%      Actual label: bishop_resized
Prediction: bishop_resized       Confidence: 79.27%      Actual label: Rook-resize
Prediction: Rook-resize  Confidence: 86.44%      Actual label: Rook-resize
Prediction: Rook-resize  Confidence: 57.77%      Actual label: bishop_resized
Prediction: pawn_resized         Confidence: 52.18%      Actual label: pawn_resized
Prediction: Queen-Resized        Confidence: 95.31%      Actual label: Queen-Resized
Prediction: knight-resize        Confidence: 81.42%      Actual label: knight-resize
Prediction: pawn_resized         Confidence: 84.91%      Actual label: pawn_resized
Prediction: Queen-Resized        Confidence: 58.78%      Actual label: bishop_resized
Prediction: knight-resize        Confidence: 85.80%      Actual label: knight-resize
Prediction: Queen-Resized        Confidence: 56.27%      Actual label: Queen-Resized
Accuracy:
========
Score 21/27
Accuracy: 77.78%
```

In [ ]:
```
total_correct = 0
count = 0
classes = dataset.classes
model.load_state_dict(torch.load('./model_adam.pt',map_location=device))

print("************************")
print("Results with Adam optimizer:")
print("************************")

with torch.no_grad():
    model.eval()
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        yp = model(images)
        yp = nn.Softmax(dim = 1)(yp)
        top_p, top_class = yp.topk(1, dim = 1)
        eq = top_class == labels.view(-1, 1)
        # print(classes[top_class.item()])
        total_correct += eq.sum().item()

        if count % 1 == 0:
            print("Prediction: {}\t Confidence: {:.2f}%\t Actual label: {}".format(classes[top_class.item()], top_p.item() * 100, classes[labels.item()]))
        else:
            print(f"count%1 = {count % 1}")
        count += 1

print("Accuracy:\n========")
print(f"Score: {total_correct}/{len(test_data)}")
print(f"Accuracy: {(total_correct/len(test_data)) * 100:.2f}%")
```

```
************************
Results with Adam optimizer:
************************
Prediction: knight-resize        Confidence: 98.64%      Actual label: knight-resize
Prediction: knight-resize        Confidence: 99.38%      Actual label: knight-resize
Prediction: pawn_resized         Confidence: 41.99%      Actual label: pawn_resized
Prediction: knight-resize        Confidence: 88.99%      Actual label: knight-resize
Prediction: Queen-Resized        Confidence: 81.18%      Actual label: Queen-Resized
Prediction: bishop_resized       Confidence: 47.86%      Actual label: Rook-resize
Prediction: bishop_resized       Confidence: 76.42%      Actual label: bishop_resized
Prediction: Rook-resize  Confidence: 75.67%      Actual label: Rook-resize
Prediction: knight-resize        Confidence: 99.30%      Actual label: knight-resize
Prediction: knight-resize        Confidence: 95.13%      Actual label: knight-resize
Prediction: Queen-Resized        Confidence: 96.16%      Actual label: Queen-Resized
Prediction: bishop_resized       Confidence: 37.09%      Actual label: bishop_resized
Prediction: bishop_resized       Confidence: 44.53%      Actual label: Queen-Resized
Prediction: pawn_resized         Confidence: 97.79%      Actual label: pawn_resized
Prediction: knight-resize        Confidence: 72.79%      Actual label: knight-resize
Prediction: Rook-resize  Confidence: 86.53%      Actual label: Rook-resize
Prediction: knight-resize        Confidence: 98.90%      Actual label: knight-resize
Prediction: Queen-Resized        Confidence: 41.97%      Actual label: bishop_resized
Prediction: Queen-Resized        Confidence: 59.26%      Actual label: Queen-Resized
Prediction: bishop_resized       Confidence: 59.03%      Actual label: bishop_resized
Prediction: Queen-Resized        Confidence: 51.27%      Actual label: Queen-Resized
Prediction: Rook-resize  Confidence: 37.03%      Actual label: knight-resize
Prediction: Rook-resize  Confidence: 98.09%      Actual label: Rook-resize
Prediction: Rook-resize  Confidence: 47.88%      Actual label: pawn_resized
Prediction: pawn_resized         Confidence: 61.92%      Actual label: knight-resize
Prediction: Rook-resize  Confidence: 62.42%      Actual label: bishop_resized
Prediction: bishop_resized       Confidence: 90.83%      Actual label: bishop_resized
Accuracy:
========
Score: 20/27
Accuracy: 74.07%
```