Ioniță Alexandru-Mihail
1231B
UNSTPB
FILS

Playing Card Classifier

Neural Networks &
Genetic Algorithms
2024

Contents:

I) About the Project

As a project for the NNGA course, I chose to create a Neural Network that would classify images of playing cards. The model is able to discern between 53 categories, which include all of the 52 standard cards we would find in a pack, and the two jokers as their own category.

The dataset we used can be found online. It originally contained around 100-170 training photos of playing cards per class, with only 5 validation photos. I decided to move some of the photos from the training dataset to the validation dataset, while trying to keep a ratio of 80% training data - 20% validation data. Our dataset contains in total 7624 images, each one of them being 224x224 pixels, with 3 channels for color (RGB).

After training the model, we get an accuracy of 59%, after 30 epochs, but more on that later.

II) Used Libraries

Our code is written in Python, and taking advantage of multiple external libraries. These are:

- **PyTorch:** For building the model of our neural network and its training
    - torch: the Core library
    - torch.nn: Loss functions and other modules
    - torch.optim: Optimization algorithms
    - torchvision.transforms: Preprocessing of the images before training
    - torch.utils.data: Utilities
- **Scikit-learn:** For evaluating the performance of the trained model
    - Sklearn.metrics: Confusion matrix and accuracy score
- **Matplotlib:** Plotting the training and validation losses through the epochs
    - matplotlib.pyplot: Graph plotting functionality
- **Seaborn:** Better visualization of our confusion matrix
    - seaborn: The heat map of the confusion matrix

III) About the Model

Since we are creating a model for image classification, I decided to go with a Convolutional Neural Network, or CNN. It's architecture is made by four convolutional layers, followed by four fully connected layers. We also use dropout layers, in order to try to prevent overfitting.

a) **Convolutional layers:** Each one of them extracts features from the input images, after which is followed by a ReLU activation and max-pooling.

b) **Dropout layers:** It selects a fraction of the input units to 0 during each upgrade, while training. This helps us by preventing the model from learning our train dataset too well, and not classifying correctly images it has never seen before.

c) **Fully Connected layers:** These last four layers perform the final classification of our data.

IV) Code Description

First things first, we check the device the code is running on for any CUDA cores, in order to get more computation power, which results in a shorter training time. If no CUDA cores are available, we just train the neural network on the CPU.

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

After that, I created a new class named "PlayingCardDataset", which helps us load the dataset from the drive. This class is based on the PyTorch dataset class, for better compatibility.

```
class PlayingCardDataset(torch.utils.data.Dataset):
    def __init__(self, data_dir, transform=None):
        self.data = ImageFolder(data_dir, transform=transform)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

    @property
    def classes(self):
        return self.data.classes
```

Next up, we define our model. Besides the code from the initialization function, it also has a "forward" function, which makes the data go through all of the layers of the Convolutional Neural Network.

```
class CardNet(nn.Module):
    def __init__(self):
        super(CardNet, self).__init__()

        # 4 layers of convolution to make the features more obvious
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)

        # Dropout function to counter overfitting
        self.dropout = nn.Dropout(p=0.2)

        # 2 fully Connected layers
        self.fc1 = nn.Linear(256 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 53)
```

We resize all of the photos to a dimension of 128x128 pixels, for a better standardization of our dataset, and then transform everything into an object of the Tensor class, so it can be analyzed with PyTorch.

```
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
])
```

The last step before training the model is to define the function loss, the optimizer, and the learning rate of our model. For the optimizer, we choose ADAM for its accuracy. If we increase the learning rate, we start to get overfitting after the first 2-3 iterations, with both the training loss and validation loss stagnating and staying close to each other.

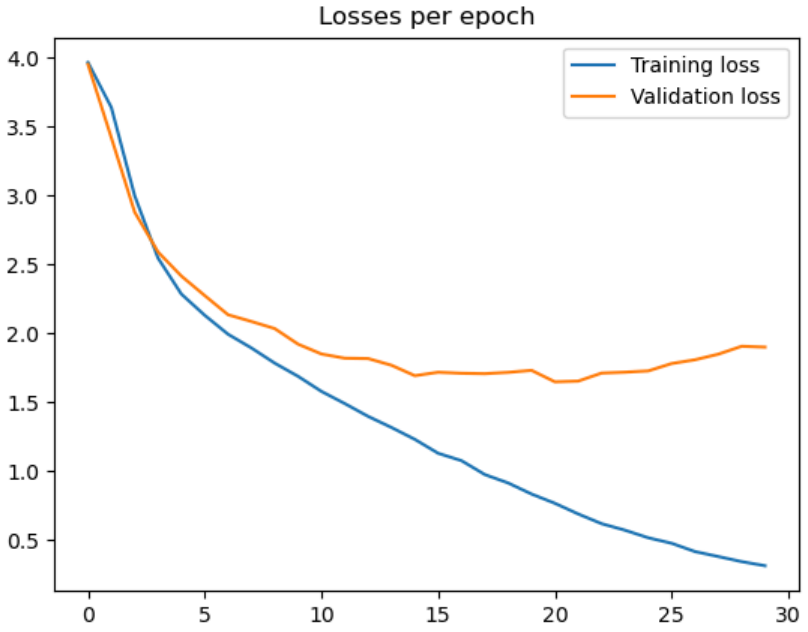
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

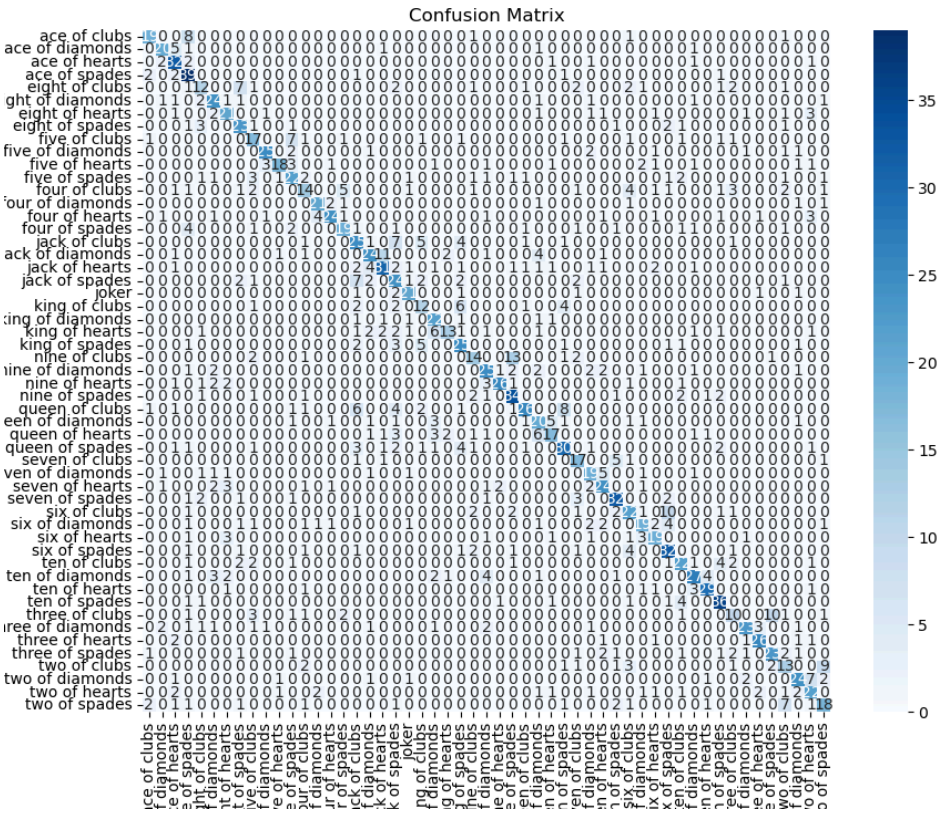
Finally, we train our model, and compute the losses of our training and validation in each epoch.

```
for epoch in range(number_of_epochs):

    # Training phase of the epoch
    model.train()
    running_train_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_train_loss += loss.item()

    # Evaluation phase of the epoch
    model.eval()
    running_valid_loss = 0.0
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for images, labels in valid_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_valid_loss += loss.item()

            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())

    # Compute the average losses
    average_train_loss = running_train_loss / len(train_loader)
    average_valid_loss = running_valid_loss / len(valid_loader)

    # Store the losses for plotting
    train_losses.append(average_train_loss)
    valid_losses.append(average_valid_loss)
```

V) Results Analysis

I have trained this neural network for 30 epochs, which took about 8 minutes. After the training and validating is completed, we obtain the following results:





Overall, the accuracy of our model is **only 59%**.

After we take a look at our graph for train loss and validation loss, we can observe that while our training dataset is more extensive, and can slowly but steadily train through these 30 epochs, the validation loss plateaus at around epoch 20, after which **it starts to rise**. This is a clear sign that our model is overfitting, which is most probably caused by our small dataset per class.

VI) Future Improvements

Firstly, I think that one of the best improvements that could be done in order to increase the accuracy of our model is to improve the dataset. A bigger dataset could certainly help the model train better and learn the patterns that define each one of our classes. We could also do a more aggressive augmentation on our data, in the transform function: putting the model into situations that could be encountered in normal usage of the program. Things like: rotating the image, applying grayscale, flipping the image horizontally or vertically, etc. could make the model better prepared.

To combat overfitting, we could also increase the dropout of the Dropout Layer, so the model wouldn't memorize the dataset, decrease the learning rate during the training, or even stop the training altogether if the model starts to overfit, with the help of an early stop function.

VII) Conclusions

While the project is not perfect, it gives us more insight in the world of neural networks, and teaches us a few things.

I think that the importance of a good dataset is best exemplified here, due to the large number of classes the model has to work with. Data augmentation, normalization of the input data, and the need of a more complex model are just other highlights that should be taken into consideration in future works.

VIII) Sources & References

Dataset & Training Sequence

How the Adam Optimizer Works

IX) Source Code

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder

# Device configuration; I have a 4060 so we use the CUDA cores, the CPU
is put here as a failsafe
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("The device being used is:", device)

# Dataset class for easier access
class PlayingCardDataset(torch.utils.data.Dataset):
    def __init__(self, data_dir, transform=None):
        self.data = ImageFolder(data_dir, transform=transform)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

    @property
    def classes(self):
        return self.data.classes

# Define our model for the neural network
# Since we process images, we use a CNN (Convolution Neural Network)
class CardNet(nn.Module):
    def __init__(self):
        super(CardNet, self).__init__()

        # 4 layers of convolution to make the features more obvious
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1,
padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
padding=1)
```

```
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1,
padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1,
padding=1)

        # Dropout function to counter overfitting
        self.dropout = nn.Dropout(p=0.2)

        # 4 fully Connected layers
        self.fc1 = nn.Linear(256 * 8 * 8, 4096)
        self.fc2 = nn.Linear(4096, 1024)
        self.fc3 = nn.Linear(1024, 512)
        self.fc4 = nn.Linear(512, 53)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.max_pool2d(x, 2)
        x = self.dropout(x)
        x = nn.functional.relu(self.conv2(x))
        x = nn.functional.max_pool2d(x, 2)
        x = self.dropout(x)
        x = nn.functional.relu(self.conv3(x))
        x = nn.functional.max_pool2d(x, 2)
        x = self.dropout(x)
        x = nn.functional.relu(self.conv4(x))
        x = nn.functional.max_pool2d(x, 2)
        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = nn.functional.relu(self.fc1(x))
        x = self.dropout(x)
        x = nn.functional.relu(self.fc2(x))
        x = self.dropout(x)
        x = nn.functional.relu(self.fc3(x))
        x = self.dropout(x)
        x = self.fc4(x)
        return x

# Define the preprocessing we use on the dataset in order to counter
overfitting
# We want the CNN to learn the features, not memorize our dataset
transform = transforms.Compose([
    transforms.Resize((128, 128)),
```

```
    transforms.ToTensor(),
])

# Load the train dataset
train_dataset_folder = './downloaded_dataset/train'
train_dataset = PlayingCardDataset(train_dataset_folder,
transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Load the valid dataset
valid_dataset_folder = './downloaded_dataset/valid'
valid_dataset = PlayingCardDataset(valid_dataset_folder,
transform=transform)
valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=False)

# Initialize the network and define the loss function and optimizer
model = CardNet().to(device)

# Define our loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

# Number of epochs
number_of_epochs = 30

# Arrays in which we store our train / valid losses
train_losses = []
valid_losses = []

# Implement early stop after 3 steps

# Training loop
for epoch in range(number_of_epochs):

    # Training phase of the epoch
    model.train()
    running_train_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

        running_train_loss += loss.item()

    # Evaluation phase of the epoch
    model.eval()
    running_valid_loss = 0.0
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for images, labels in valid_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_valid_loss += loss.item()

            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())

    # Compute the average losses
    average_train_loss = running_train_loss / len(train_loader)
    average_valid_loss = running_valid_loss / len(valid_loader)

    # Store the losses for plotting
    train_losses.append(average_train_loss)
    valid_losses.append(average_valid_loss)

    # Print the stats of the epoch
    print(f'Epoch [{epoch+1}/{number_of_epochs}], Train Loss:
{average_train_loss:.4f}, Validation Loss: {average_valid_loss:.4f}')

# Plot training and validation loss
plt.plot(train_losses, label='Training loss')
plt.plot(valid_losses, label='Validation loss')
plt.legend()
plt.title("Losses per epoch")
plt.show()

# Calculate and print confusion matrix and accuracy
conf_matrix = confusion_matrix(all_labels, all_preds)
accuracy = accuracy_score(all_labels, all_preds)
```

```
print(f'Validation Accuracy: {accuracy:.4f}')

plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=train_dataset.classes, yticklabels=train_dataset.classes)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```