

POLITECHNIKA WROCŁAWSKA

SIECI NEURONOWE

**Sprawozdanie**

PROBLE IDentyfikacji  
w Sieciach GŁĘBOKICH

**Autorzy:**

Tobiasz Rolla

Piotr Ges

**Data:** 20 stycznia 2025

# Spis treści

1	Wstęp . . . . .	2
2	Opis działania sieci konwolucyjnej . . . . .	2
3	Opis Zestawu Danych . . . . .	3
4	Przetworzenie obrazów . . . . .	4
5	Podział obrazów na zestawy . . . . .	5
6	Funkcje uczenia oraz testowanie . . . . .	6
7	Opis sposobu prowadzenia badań . . . . .	8
8	Sieć 1 . . . . .	8
9	Sieć 2 . . . . .	10
10	Sieć 3 . . . . .	12
11	Sieć 4 . . . . .	15
12	Wprowadzenie augmentacji danych . . . . .	17
13	Sieć 5 . . . . .	18
14	Dodatkowe badania . . . . .	20
14.1	Funkcje optymalizacyjne . . . . .	20
14.2	Proporcje zbioru danych . . . . .	21
15	Wnioski . . . . .	21

# 1 Wstęp

Celem projektu była identyfikacja zestawu kart do gry przy użyciu sieci konwolucyjnych CNN. W projekcie zostały wykorzystany PyTorch do budowy oraz uczenia sieci neuronowej oraz gotowy zbiór danych zawierający karty do gry.

## 2 Opis działania sieci konwolucyjnej

Sieć konwolucyjna znajduje szerokie zastosowanie w identyfikacji obrazu. W standardowej architekturze sieci konwolucyjnej można wyodrębnić 4 rodzaje warstw są to:

### Warstwa Konwolucji

Warstwa ta wykonuje operację konwolucji(splotu) przesuwając niewielkie jądro(kernel) z reguły o rozmiarach 3x3 po kolejnych pikselach. Pozwala to wyodrębnić interesujące nas szczeguły karzda taka warstwa uczy się rozpoznawać inny aspekt obrazu.

### Pading

Ciekawym aspektem tej warstwy jest możliwość dodania paddingu. Jest to dodanie ramki do okoła obrazu co pozwala na zachowanie jego pierwotnego rozmiaru.

### Warstwa Poolingowa

Zadaniem tej warstwy jest redukcja rozmiaru z jednoczesnym zachowaniem interesujących szczegółów obrazu. Pooling również posiada swój kernel zregóły mniejszy o rozmiarach 2x2. Ponadto trzeba ustawić tak zwany stride odowiedzialny za przesuwanie okna z reguły przesunięcie jest równe 2. Nie stosowanie większych kerneli wynika z faktu zbyt dużej redukcji rozmiarów obrazu. Opiszmy teraz rodzaje poolingu.

- **Max Pool** Najpopularniejszy rodzaj poolingu. Max Pool wyodrębnia pixel z największą wartością w jądrze. Skupia się na kolorach białych a pozbywa się ciemnych
- **Average Pool** Również dość popularne rozwiązanie. Average Pool wylicza średnia wartość z pikseli znajdujących się w jądrze i tą wartość przypisuje obszarowi użyteczny gdy szczeguły obrazu nie są powiązane z konkretną wartością pikseli.
- **Min Pool** Najmniej popularny skupia się na najmniej znaczących pixelach faworyzuje kolory ciemne.

## Warstwa w pełni połączona

Warstwa w pełni połączona ang(Full Conected Neural-Network) FN to kluczowy aspekt sieci odpowiedzialny za klasyfikację. Neurony każdej warstwy są połączone z neuronami każdej następnej warstwy. Jeśli chodzi o wykorzystywane funkcje aktywacji są to

- **ReLU** funkcja tanh zwraca wartości wejścia gdy jest ono większe od zera i odzuca wartości mniejsze od zera z reguły znajduje się w wszystkich warstwach FC poza ostatnią
- **SoftMax** Znajdująca się w ostatniej warstwie przekształca wartości na prawdopodobieństwo.

$$\text{Softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^k \exp(z_j)}$$

- $z_i$  to wynik (logit) dla klasy  $i$ ,
- $\exp(z_i)$  to wartość wykładnicza  $e^{z_i}$ ,
- $\sum_{j=1}^k \exp(z_j)$  to suma wykładniczych wartości dla wszystkich klas.

## 3 Opis Zestawu Danych

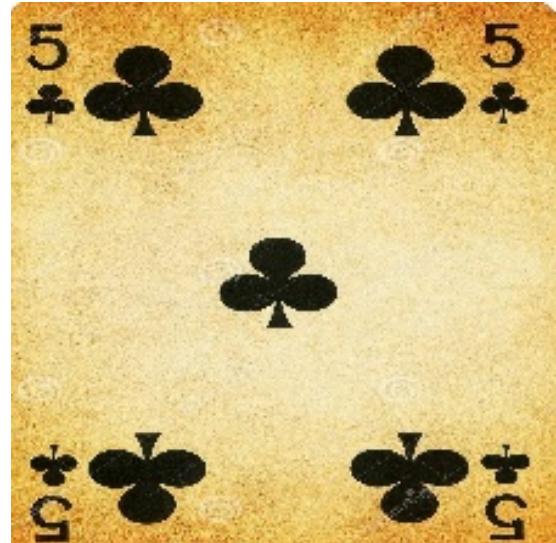
Zestaw składał się z 53 klas(13 figur każdego z 4 kolorów plus jocker). Zestaw składał się z 8154 kart. Posiada obrazy kart nie ich zdjęcia co ułatwia klasyfikację, ponieważ w tle nie znajdują się różne obrazy mogące wpływać na dokładność rozpoznawania. Karty posiadają różne style graficzne. Zostały one podzielone na:

- zestaw uczący 7624 kart odpowiadając 93.5%
- zestaw walidacyjny zawierał 265 kart odpowiadając 3.25%.
- zestaw testowy zawierał również 265 kart odpowiadając 3.25%

Tak duża przewaga kart zestawu uczącego wynika z dużej liczby klas i wielkości zbioru. Poniżej przedstawiono przykładowe obrazy z bazy:



Rysunek 1: Przykładowy obraz 1.



Rysunek 2: Przykładowy obraz 2.

## 4 Przetworzenie obrazów

W celu przygotowania danych do trenowania modelu, obrazy znajdujące się w bazie przetworzono poprzez transformację oraz utworzenie odpowiednich struktur danych. Wykonane transforamcje to skalowanie, normalizacja oraz w późniejszych sieciach augmentacja.

```
transform = transforms.Compose([
    transforms.Resize((128, 128)), #Resize to 128x128
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Rysunek 3: Transformacja obrazów.

W późniejszych modelach wprowadzono augmentacje obrazów zawartych w bazie. Pozwala to na poprawie jakości danych oraz zwiększenie możliwości generalizacji modelu.

```
transform_train = transforms.Compose([
    transforms.Resize((128, 128)),
    RandomChoice([
        transforms.RandomHorizontalFlip(p=1.0),
        transforms.RandomVerticalFlip(p=1.0),
        transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.2),
        transforms.RandomRotation(degrees=30)
    ]),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Rysunek 4: Transformacja obrazów oraz augmentacja.

## 5 Podział obrazów na zestawy

W celu przeprowadzenia efektywnego trenowania, walidacji i testów modelu dane należy podzielić na trzy części: zbiór treningowy, walidacyjny oraz testowy. W naszym przypadku używana baza obrazów posiadała taki podział. Przeprowadzono jednak testy na końcowym modelu dla których zmieniano proporcje podziału danych.

```
def merge_category_folders(input_dir, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    list_dir_input = [entry.name for entry in os.scandir(input_dir) if entry.is_dir()]
    for dir in list_dir_input:
        print(dir)
    list_train = [entry.name for entry in os.scandir(input_dir + '/' + list_dir_input[0]) if entry.is_dir()]
    list_valid = [entry.name for entry in os.scandir(input_dir + '/' + list_dir_input[1]) if entry.is_dir()]
    list_test = [entry.name for entry in os.scandir(input_dir + '/' + list_dir_input[2]) if entry.is_dir()]
    for dir in list_train:
        os.makedirs(output_dir + '/' + dir, exist_ok=True)
        train_path = input_dir + '/' + list_dir_input[0] + '/' + dir
        valid_path = input_dir + '/' + list_dir_input[1] + '/' + dir
        test_path = input_dir + '/' + list_dir_input[2] + '/' + dir
        category_output_dir = output_dir + '/' + dir
        print(train_path)
        for root, dirs, files in os.walk(train_path):
            for file in files:
                if "zone.identifier" in file.lower():
                    continue
                src_path = os.path.join(root, file)
                dst_file = f"train_{file}" # Dodanie prefiku
                dst_path = os.path.join(category_output_dir, dst_file)
                shutil.copy(src_path, dst_path)
        for root, dirs, files in os.walk(valid_path):
            for file in files:
                if "zone.identifier" in file.lower():
                    continue
                src_path = os.path.join(root, file)
                dst_file = f"val_{file}" # Dodanie prefiku
                dst_path = os.path.join(category_output_dir, dst_file)
                shutil.copy(src_path, dst_path)
        for root, dirs, files in os.walk(test_path):
            for file in files:
                if "zone.identifier" in file.lower():
                    continue
                src_path = os.path.join(root, file)
                dst_file = f"test_{file}" # Dodanie prefiku
                dst_path = os.path.join(category_output_dir, dst_file)
                shutil.copy(src_path, dst_path)

merge_category_folders(r'C:\Users\piotr\Desktop\data', r'C:\Users\piotr\Desktop\all_data')
input_folder1 = r'C:\Users\piotr\Desktop\data\train\joker'
input_folder2 = r'C:\Users\piotr\Desktop\data\validate\joker'
input_folder3 = r'C:\Users\piotr\Desktop\data\test\joker'
print(len(input_folder1))

splitfolders.ratio(r'C:\Users\piotr\Desktop\all_data', r'C:\Users\piotr\Desktop\95_2.5_2.5', seed=1337, ratio=(0.95, 0.025, 0.025))
splitfolders.ratio(r'C:\Users\piotr\Desktop\all_data', r'C:\Users\piotr\Desktop\90_7.5_2.5', seed=1337, ratio=(0.9, 0.075, 0.025))
splitfolders.ratio(r'C:\Users\piotr\Desktop\all_data', r'C:\Users\piotr\Desktop\90_5_5', seed=1337, ratio=(0.9, 0.05, 0.05))
splitfolders.ratio(r'C:\Users\piotr\Desktop\all_data', r'C:\Users\piotr\Desktop\80_10_10', seed=1337, ratio=(0.8, 0.1, 0.1))
splitfolders.ratio(r'C:\Users\piotr\Desktop\all_data', r'C:\Users\piotr\Desktop\70_15_15', seed=1337, ratio=(0.7, 0.15, 0.15))
splitfolders.ratio(r'C:\Users\piotr\Desktop\all_data', r'C:\Users\piotr\Desktop\60_20_20', seed=1337, ratio=(0.6, 0.2, 0.2))
```

Rysunek 5: Kod dzielący dane na odpowiednie zbiory w różnych proporcjach

Obrazy zostały następnie przekonwertowane do datasetów oraz załadowane za pomocą dataloaderów. Dane do loaderów podawane były w porcjach batch\_size, ustawione dla większości testów na 32. Co daje dobry kompromis pomiędzy stabilnością i szybkością.

```

train_dataset = PlayingCardDataset(train_folder, transform=transform_train)
val_dataset = PlayingCardDataset(valid_folder, transform=transform)
test_dataset = PlayingCardDataset(test_folder, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

```

Rysunek 6: Przekształcenie w datasety i dataloadery.

## 6 Funkcje uczenia oraz testowanie

W celu rozpoczęcia uczenia modelu, stworzono funkcję uczącą. Dostosowuje ona parametry modelu do danych treningowych działając w trybie trenowania (co pozwala na aktualizację wag). Dla każdej porcji w train\_loader obliczana jest strata oraz aktualizowane są wagi poprzez propagowanie wstecz gradientu.

```

epoch = 0
done = False
while epoch < num_epochs and not done:
    epoch += 1
    # Training phase
    model.train()
    running_loss, running_corrects = 0.0, 0
    for images, labels in tqdm(train_loader, desc='Training loop'):
        # Move inputs and labels to the device
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * labels.size(0)
        _, preds = torch.max(outputs, 1) # Pobieranie prognoz
        running_corrects += (preds == labels).sum().item()
    train_loss = running_loss / len(train_loader.dataset)
    train_accuracy = running_corrects / len(train_loader.dataset)
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

```

Rysunek 7: Pętla treningowa

Następnie przechodzi w tryb ewaluacji (model.eval()), obliczana jest strata i dokładność bez aktualizacji wag.

```

# Validation phase
model.eval()
running_loss, running_corrects = 0.0, 0
with torch.no_grad():
    for images, labels in tqdm(val_loader, desc='Validation loop'):
        # Move inputs and labels to the device
        images, labels = images.to(device), labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)
        running_loss += loss.item() * labels.size(0)
        _, preds = torch.max(outputs, 1) # Pobieranie prognoz
        running_corrects += (preds == labels).sum().item()
    val_loss = running_loss / len(val_loader.dataset)
    val_accuracy = running_corrects / len(val_loader.dataset)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)
    print(f'Epoch {epoch}/{num_epochs} - Train loss: {train_loss:.4f}, Validation loss: {val_loss:.4f}, Train acc: {train_accuracy:.4f}, Val acc: {val_accuracy:.4f}')

    # Aktualizacja harmonogramu
    #scheduler.step(val_loss)
    # Check early stopping criteria
    done = es(model, val_loss)
    print(f'Early Stopping: {es.status}')

```

Rysunek 8: Faza walidująca pętli treningowej

Jeżeli strata walidacyjna nie poprawi się przez określoną ilość epok, trenowanie jest przerwane i model wraca do wartości wag z epoki przed którą wykryto brak spadku strat walidacji. Funkcja wczesnego zatrzymywania jest wykorzystywana w celu uniknięcia przetrenowania sieci. Można w niej dostosować parametry takie jak cierpliwość, czyli ile epok musi wystąpić pod rząd bez poprawy straty przed zatrzymaniem lub parametr, który określa czy przywrócić wagi z najlepszej dotychczas epoki. Implementacja funkcji wczesnego zatrzymywania poniżej:

```
class EarlyStopping:
    def __init__(self, patience=5, min_delta=0, restore_best_weights=True):
        self.patience = patience
        self.min_delta = min_delta
        self.restore_best_weights = restore_best_weights
        self.best_model = None
        self.best_loss = float('inf')
        self.counter = 0
        self.status = ""

    def __call__(self, model, val_loss):
        if val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
            self.best_model = copy.deepcopy(model.state_dict())
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.status = "Stopping training"
                if self.restore_best_weights:
                    model.load_state_dict(self.best_model)
                return True
            self.status = f"{self.counter}/{self.patience}"
        return False
```

Rysunek 9: Implementacja funkcji wczesnego zatrzymywania

Straty i dokładności dla zestawu treningowego i walidacyjnego są zapisywane w listach, które są później używane do stworzenia wykresów. Kod jest zaprojektowany w sposób modułowy, umożliwiający łatwe dostosowanie do różnych modeli, zbiorów danych i konfiguracji hiperparametrów.

W celu sprawdzenia skuteczności modelu stworzono funkcję obliczającą dokładność na podstawie danych testowych.

```
model.eval()
true_labels = []
predicted_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)

        outputs = model(images)
        _, predictions = torch.max(outputs, 1) # Get the class index with the highest probability
        true_labels.extend(labels.cpu().numpy())
        predicted_labels.extend(predictions.cpu().numpy())

# Calculate accuracy
accuracy = accuracy_score(true_labels, predicted_labels)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

Rysunek 10: Funkcja testowa

## 7 Opis sposobu prowadzenia badań

Badania przeprowadzono po przez testy różnych konfiguracji modeli sieci. Zmieniano ilości warstw oraz ich rodzaje w celu znalezienia optymalnej struktury. Następnie wprowadzono augmentacje danych w celu sprawdzenia, czy poprawi to dokładność sieci. Później testowano różne rodzaje funkcji optymalizacji. Jako podstawowego optymalizatora użyto Adam. Tempo uczenia w optymalizatorze było ustawione na 0.001. Jako podstawową funkcję aktywacji użyto relu. Funkcją strat we wszystkich testach było CrossEntropyLoss.

## 8 Sieć 1

Pierwsza najbardziej podstawowa architektura sieci składa się z 6 warstw. Wchodzą w nie 2 warstwy konwolucyjne, 2 warstwy poolingowe oraz 2 warstwy liniowe. W tej sieci nie użyto augmentacji danych, ani dodatkowych technik w celu poprawienia dokładności.

```
#Model
class CardClassifier(nn.Module):
    def __init__(self, num_classes=53):
        super(CardClassifier, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

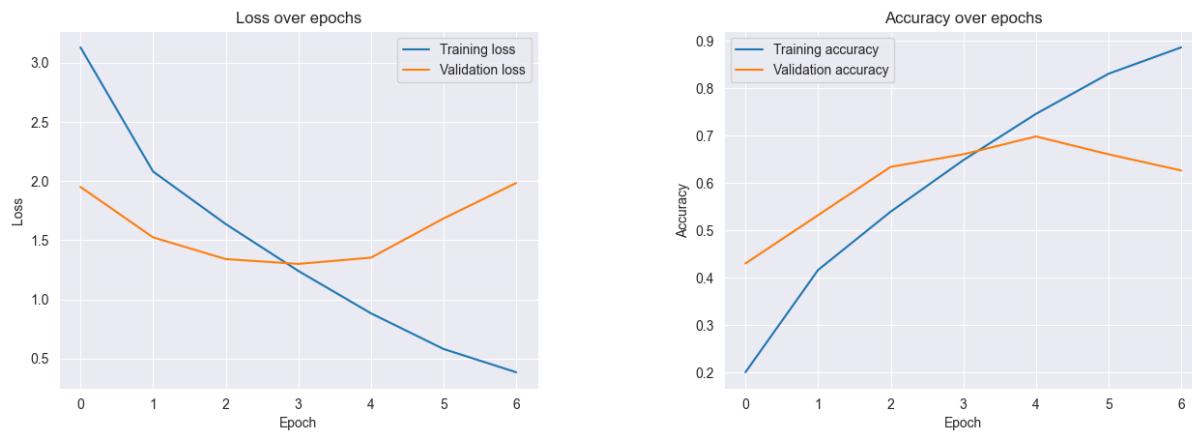
        self.fc1 = nn.Linear(128 * 32 * 32, 512)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.pool1(self.relu1((self.conv1(x))))
        x = self.pool2(self.relu2((self.conv2(x))))
        x = x.view(-1, 128*32*32)
        x = F.relu(self.fc1(x))
        output = self.fc2(x)

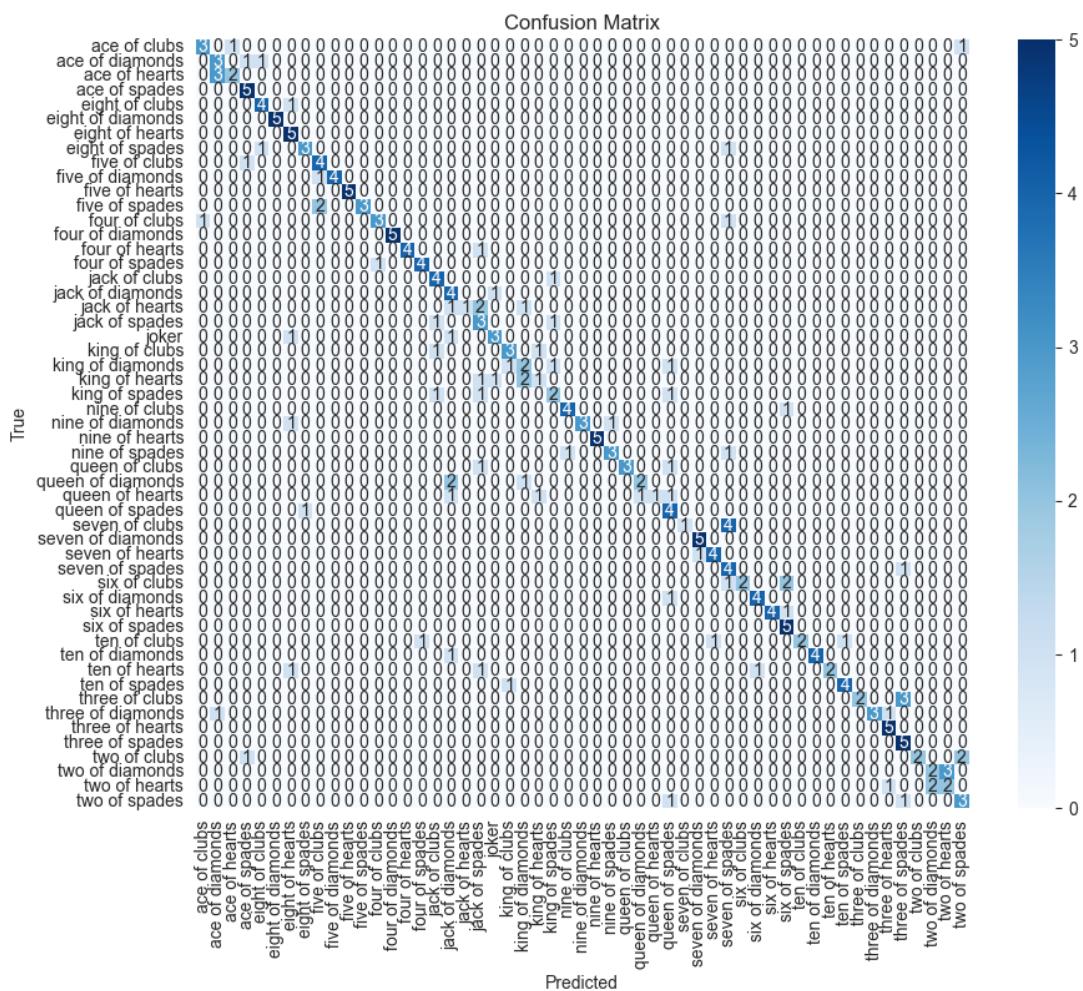
        return output
```

Rysunek 11: Model sieci 1

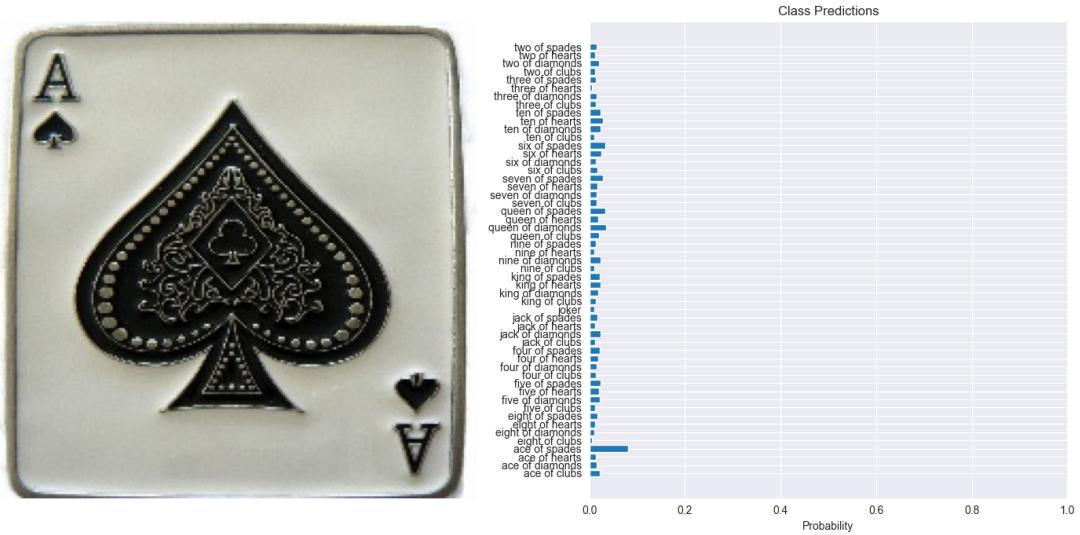
Sieć uczyła się bardzo szybko oraz była podatna na przeuczenie. Po 4 epokach błąd walidacyjny zaczął rosnąć co spowodowało zatrzymanie trenowania po 7 epokach i przywróceniu wartości wag do stanu z 4 epoki. Osiągnęła zaledwie 66.04% dokładności na danych testowych.



Rysunek 12: Przebieg procesu uczenia się.



Rysunek 13: Tablica pomyłek sieci 1.



Rysunek 14: Pewność w klasyfikacji w sieci 1 na przykładowej karcie.

## 9 Sieć 2

W drugiej sieci dodano jedną warstwę konwolucyjną oraz poolingową oraz jedną warstwę w pełni połączoną. Sieć składa się więc łącznie z 9 warstw 3 konwolucyjnych, 3 poolingowych oraz 3 liniowych. Model nadal trenowany na danych bez augmentacji.

```
#Model
class CardClassifier(nn.Module):
    def __init__(self, num_classes=53):
        super(CardClassifier, self).__init__()

        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.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(128 * 16 * 16, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, num_classes)

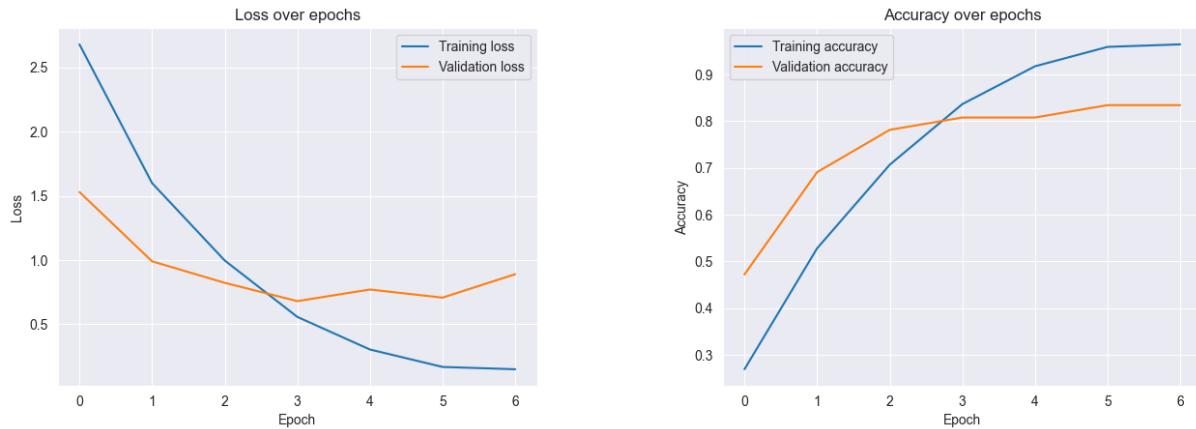
    def forward(self, x):
        x = self.pool(F.relu((self.conv1(x))))
        x = self.pool(F.relu((self.conv2(x))))
        x = self.pool(F.relu((self.conv3(x))))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        output = self.fc3(x)

    return output
```

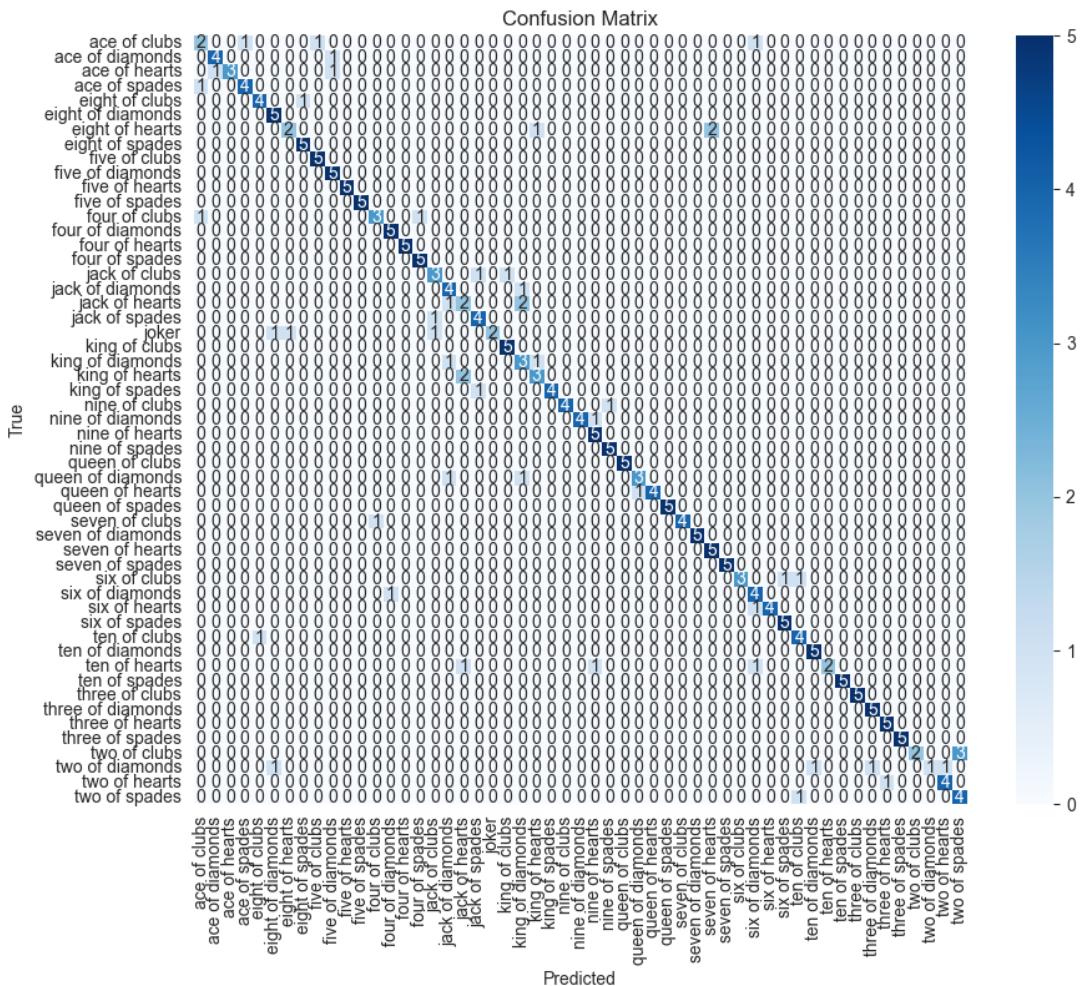
Rysunek 15: Model sieci 2

Po dodaniu dodatkowych warstw sieć nadal uczy się bardzo szybko oraz łatwo się przeucza. Poprawiła się jednak dokładność, która wzrosła z 66.04% do 80.75%. Na rysunku pokazującym pewność w klasyfikacji można również zauważyc, że zmiany poprawiły ten aspekt. Widać, że wskazywana jest prawidłowa klasa z dużym prawdopodobieństwem oraz

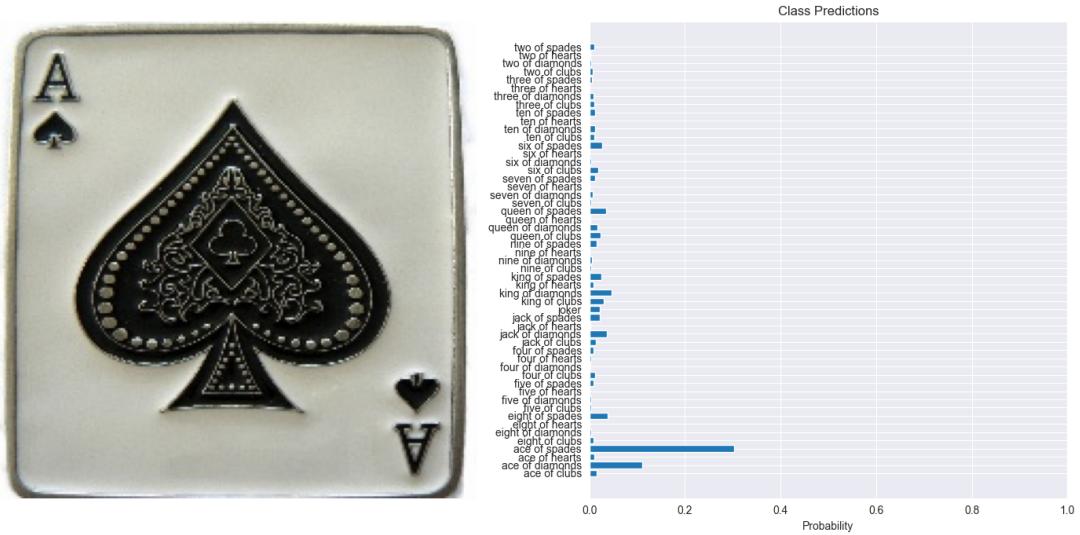
częściowo wyeliminowano efekt wskazywania na wszystkie klasy z niewielkim prawdopodobieństwem.



Rysunek 16: Przebieg procesu uczenia się.



Rysunek 17: Tablica pomyłek sieci 2.



Rysunek 18: Pewność w klasyfikacji w sieci 2 na przykładowej karcie.

## 10 Sieć 3

Siec 3 uzyskano na podstawie modyfikacji poprzedniej. Po każdej warstwie konwolucyjnej dodano Batch Normalization. Rozszerzając tym samym liczbę warstw do 12. W funkcji optymalizującej Adam dodano również parametr weight decay = 0.0001.

### Batch Normalization

Technika stosowana w sieciach neuronowych, która poprawia szybkość uczenia, stabilność i wydajność modeli. Działa poprzez normalizację danych wejściowych do warstwy, redukując efekt tzw. "shifting internal covariate" (przesunięcie kowariaty wewnętrznej). Efekt odnosi się do zmiany rozkładu aktywacji neuronów w miarę postępu treningu, co utrudnia proces uczenia.

### Weight decay

Polega na dodaniu do funkcji kosztu (loss function) dodatkowego składnika karnego, który penalizuje duże wartości wag modelu. Celem jest zmniejszenie ryzyka przeuczenia (overfitting) i poprawa generalizacji modelu.

```

#Model
class CardClassifier(nn.Module):
    def __init__(self, num_classes=53):
        super(CardClassifier, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(128 * 16 * 16, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, num_classes)

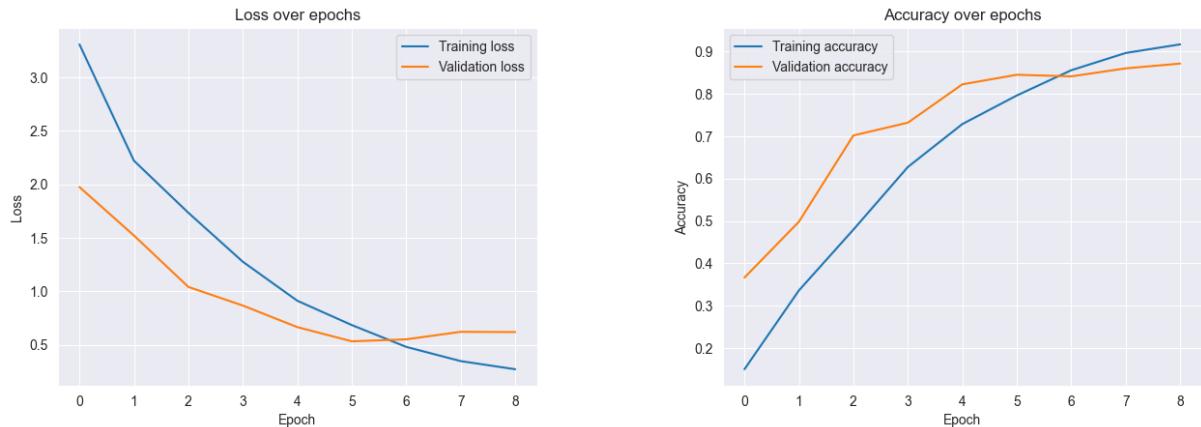
    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        output = self.fc3(x)

    return output

```

Rysunek 19: Model sieci 3

Po dodaniu Batch Normalization oraz Weight decay widać poprawę dokładności modelu z 80.75% do 84.53%. Ponadto widać poprawę pewności w klasyfikacji. Pomimo popraw sieć uczy się nadal relatywnie szybko i jest podatna na overfitting, jednakże w mniejszym stopniu niż poprzednio.



Rysunek 20: Przebieg procesu uczenia się.



## 11 Sieć 4

W cieci czwartej dodano dodatkową warstwę w pełni połączoną oraz użyto technik dropout. Liczba warstw wzrosła do 16. Zmiany w strukturze sieci pozwoliły na uzyskanie dokładności na poziomie 89.81% na zbiorze testującym. Podatność na przeuczenie sieci zmalała, a czas uczenia się wzrósł.

### Dropout

Dropout z reguły wykorzystuje się podczas treningu sieci w celu zapobiegnięcia przeuczeniu. Technika ta polega na wyłączaniu w losowy sposób neuronów sieci FC.

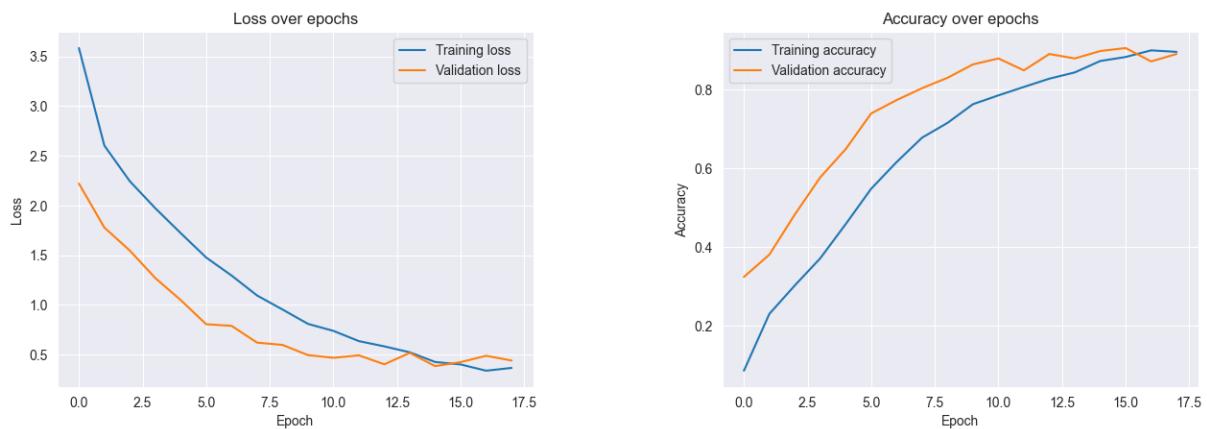
```
#Model
class CardClassifier(nn.Module):
    def __init__(self, num_classes=53):
        super(CardClassifier, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(128 * 16 * 16, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        output = self.fc3(x)

    return output
```

Rysunek 23: Model sieci 4

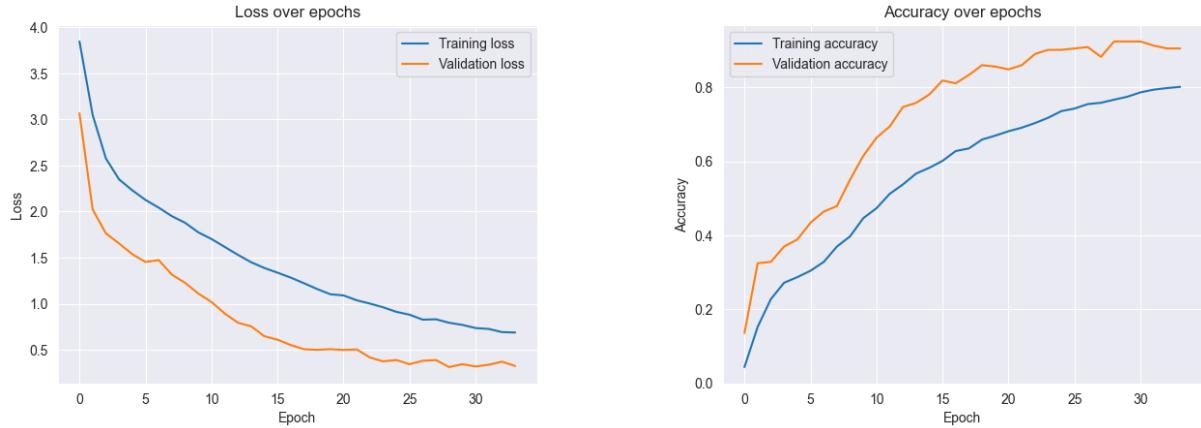


Rysunek 24: Przebieg procesu uczenia się.

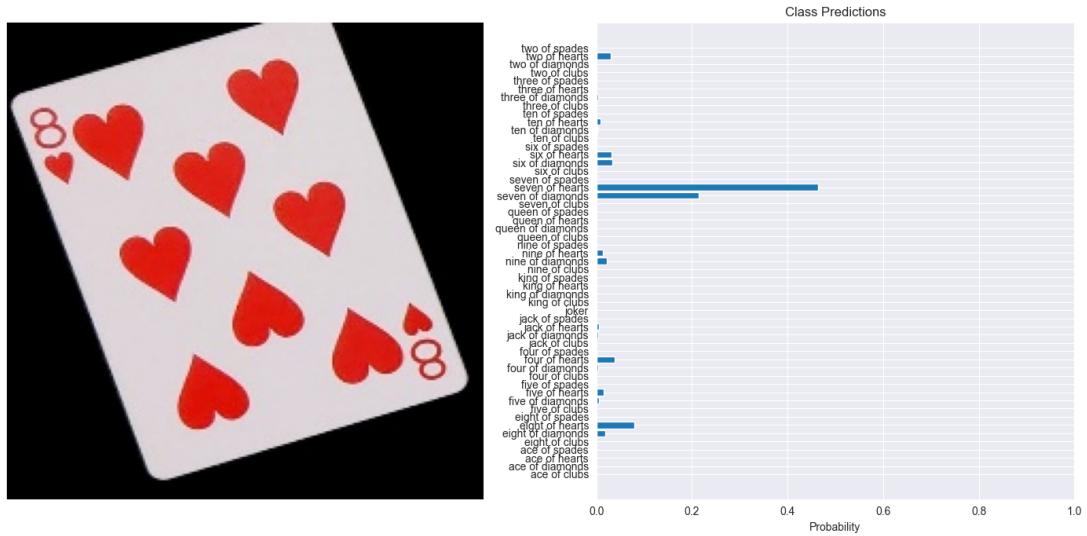


## 12 Wprowadzenie augmentacji danych

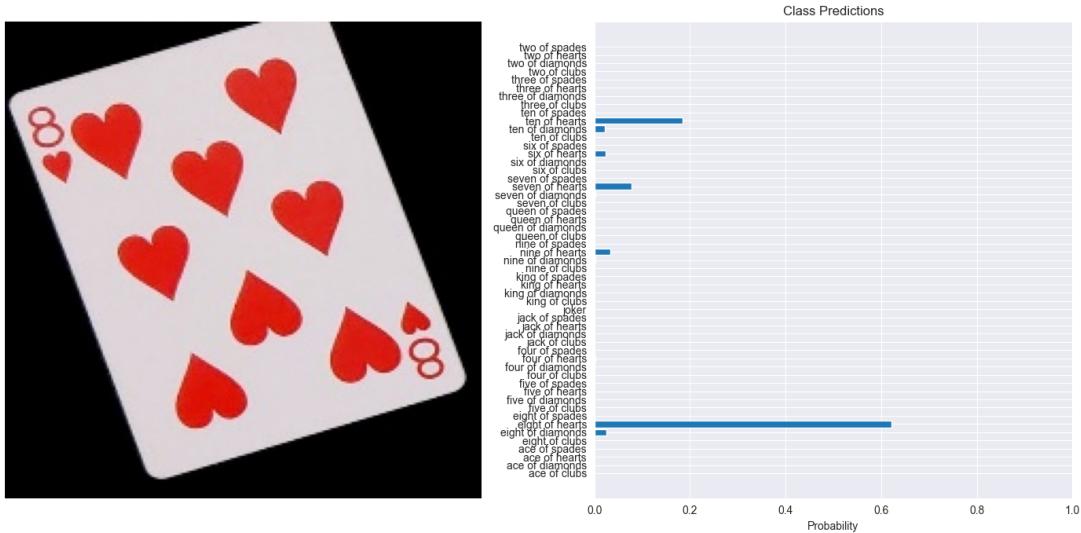
W celu poprawienia możliwości generalizacji modelu oraz poprawieniu jakości danych wprowadzono augmentacje. Obrazy dla zestawu trenującego są losują z równym prawdopodobieństwem, czy będą obracane poziomo lub pionowo, zmieni się ich kolor lub zostaną obrócone o 30 stopni. Implementacja jest pokazana wyżej na Rysunku 4. Po dodaniu augmentacji danych model sieci 4 uzyskał dokładność na poziomie 92.45%.



Rysunek 27: Przebieg procesu uczenia się.



Rysunek 28: Klasyfikacja obróconej karty bez augmentacji danych

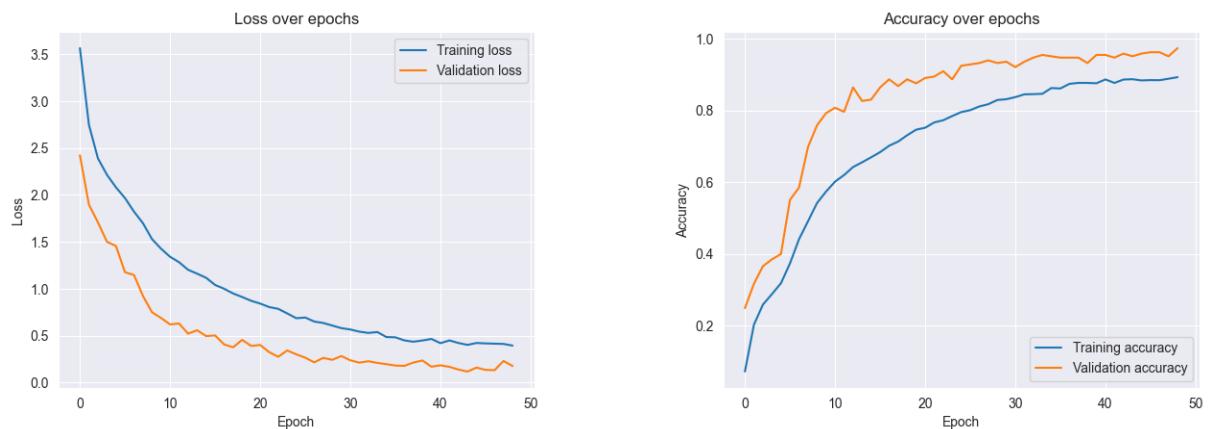


Rysunek 29: Klasyfikacja obróconej karty po augmentacji danych

Dzięki dodaniu augmentacji danych model jest w stanie rozpoznawać obrazy przechylone, które w niewielkich ilościach znajdują się w bazie. Pokazuje to lepszą generalizację modelu. Spowodowało to również spowolnienie procesu uczenia się.

## 13 Sieć 5

Jako ostatnie usprawnienie sieci postanowiono dodać dodatkową warstwę konwolucyjną wraz ze stosowanymi wcześniej technikami. Dokładność modelu po wprowadzeniu zmian oraz z zastosowaniem augmentacji wynosiła 95.07%. Jest to najlepszy wynik jaki udało nam się osiągnąć podczas realizacji projektu.



Rysunek 30: Przebieg procesu uczenia się.

```

class CardClassifier(nn.Module):
    def __init__(self, num_classes=53):
        super(CardClassifier, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(256)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(256 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, num_classes)

        self.dropout1 = nn.Dropout(p=0.1)
        self.dropout2 = nn.Dropout(p=0.2)
        self.dropout3 = nn.Dropout(p=0.3)

    def forward(self, x):
        x = self.pool(F.gelu(self.bn1(self.conv1(x))))
        x = self.pool(F.gelu(self.bn2(self.conv2(x))))
        x = self.pool(F.gelu(self.bn3(self.conv3(x))))
        x = self.pool(F.gelu(self.bn4(self.conv4(x))))

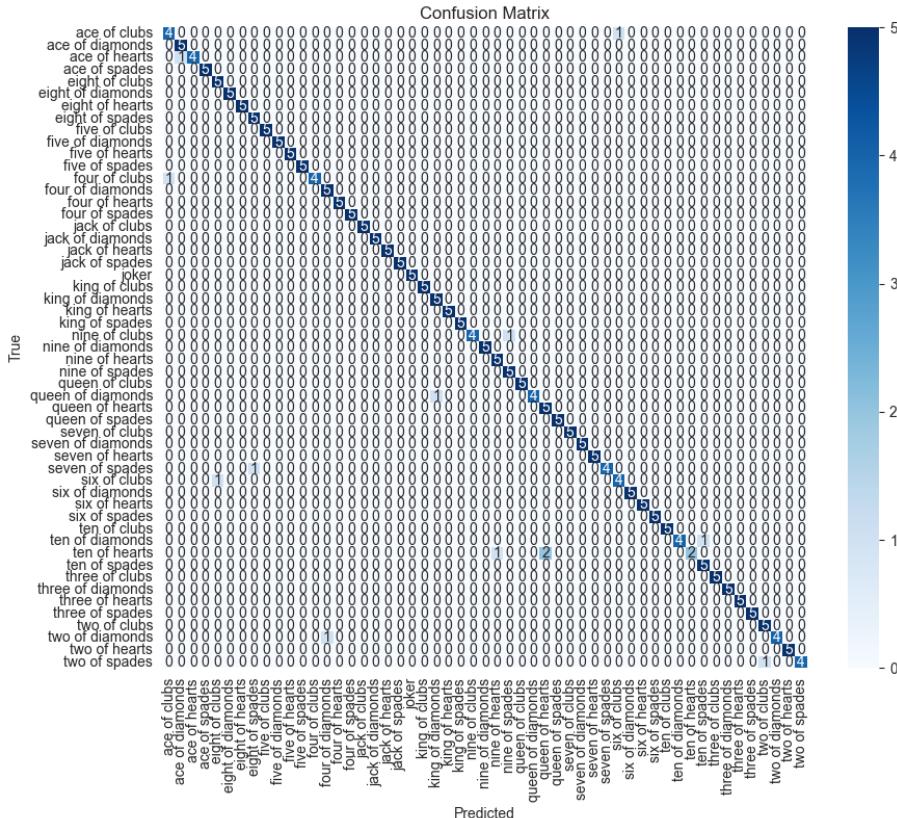
        x = x.view(x.size(0), -1)
        x = F.gelu(self.fc1(x))
        x = self.dropout1(x)
        x = F.gelu(self.fc2(x))
        x = self.dropout2(x)
        x = F.gelu(self.fc3(x))
        x = self.dropout3(x)

        output = self.fc4(x)

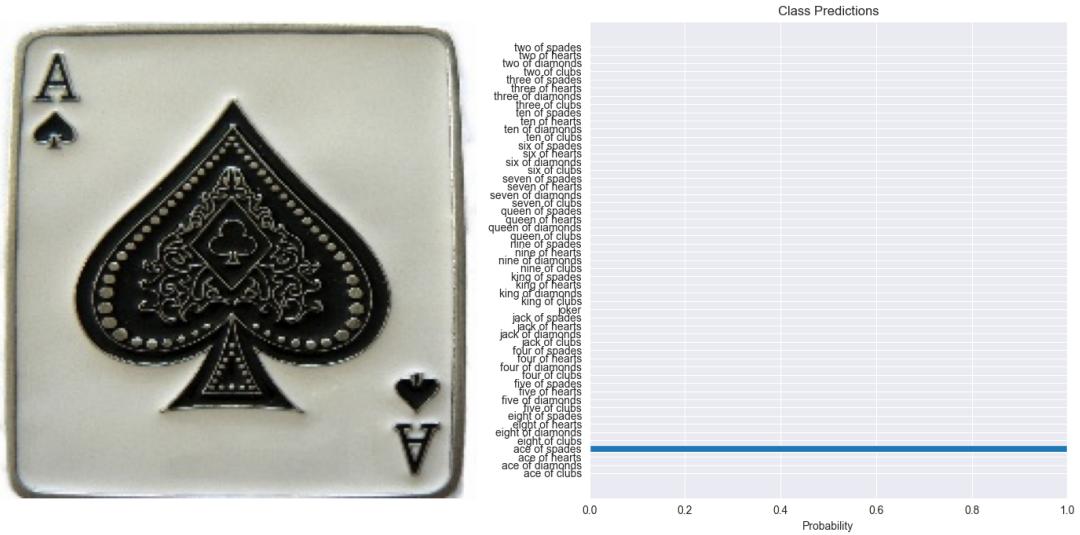
        return output

```

Rysunek 31: Model sieci 5



Rysunek 32: Tablica pomyłek sieci 5.



Rysunek 33: Pewność w klasyfikacji w sieci 5 na przykładowej karcie.

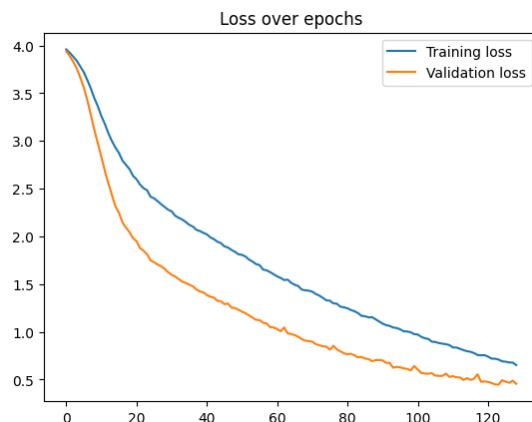
## 14 Dodatkowe badania

### 14.1 Funkcje optymalizacyjne

Optimizer	Adam	AdamW	RMSprop	SGD
Accuracy	95.07%	93.7%	92.83%	85.66%

Rysunek 34: Dokładność dla różnych funkcji optymalizujących

Zmiana funkcji optymalizującej w większości przypadków wpływała na wyniki w niewielkim stopniu lub wcale. Wyjątkiem była funkcja SGD (Stochastic gradient descent), która osiągnęła najgorszy wynik. Znacząco wydłużała również czas uczenia się sieci. Aby działała poprawnie zalecane było również użycie schedulera, który dynamicznie zmienia tempo nauki.



Rysunek 35: Strata sieci 5 z funkcją optymalizującą SGD.

## 14.2 Proporcje zbioru danych

Proporcje	60-20-20	70-15-15	80-10-10	bazowy
Accuracy	78.19%	79.47%	85.73%	95.07%

Rysunek 36: Dokładność dla różnych proporcji podziału danych

Podział danych na inne proporcje niż bazowe wpływa negatywnie na wyniki sieci. Może być to spowodowane złym dobraniem obrazów do zestawu trenującego, walidującego lub testowego. Bazowy podział danych pozwolił osiągnąć największą dokładność. innym powodem może być zbyt mała ilość danych co zmusza nas do przydzielenia większego procenta do danych uczących.

## 15 Wnioski

- Dobranie odpowiedniej ilości epok jest kluczowe, ponieważ pozwala na uniknięcie przeuczenia sieci. Zamiast dobierać ręcznie odpowiednią ilość epok można zastosować funkcję wczesnego zatrzymywania, która automatycznie przerwie pętlę uczącą po spełnieniu określonych kryteriów.
- Ze względu na dużą liczbę klas sieć osiąga w najlepszym przypadku wyniki na poziomie około 95% dokładności.
- Wprowadzenie augmentacji danych przyniosło niewielki wzrost dokładność modelu, ale znacznie poprawiło generalizację. Sieć była w stanie klasyfikować obrazy z którymi wcześniej sobie nie radziła, jak pokazano na przykładzie Rysunku 27 i Rysunku 28.
- Zwiększenie ilości warw konwolucyjnych z w znaczący sposób wpłynęło na uzyskane wynik. W szczególności przejście z 2 na 3 warstwy konwolucyjne pokazało znaczną poprawę dokładności oraz pewności modelu.
- Technika dropout zapobiegła w pewnym stopniu przeuczeniu sieci pozwoliła na osiągnięcie większej dokładności