SPRAWOZDANIE

Projekt nr 2

Piotr Olesiejuk

16 kwietnia 2019

1 Treść projektu

Zaprojektowanie konwolucyjnej sieci neuronowej do rozpoznawania obrazków ze zbioru CIFAR10.

2 Wykorzystane technologie

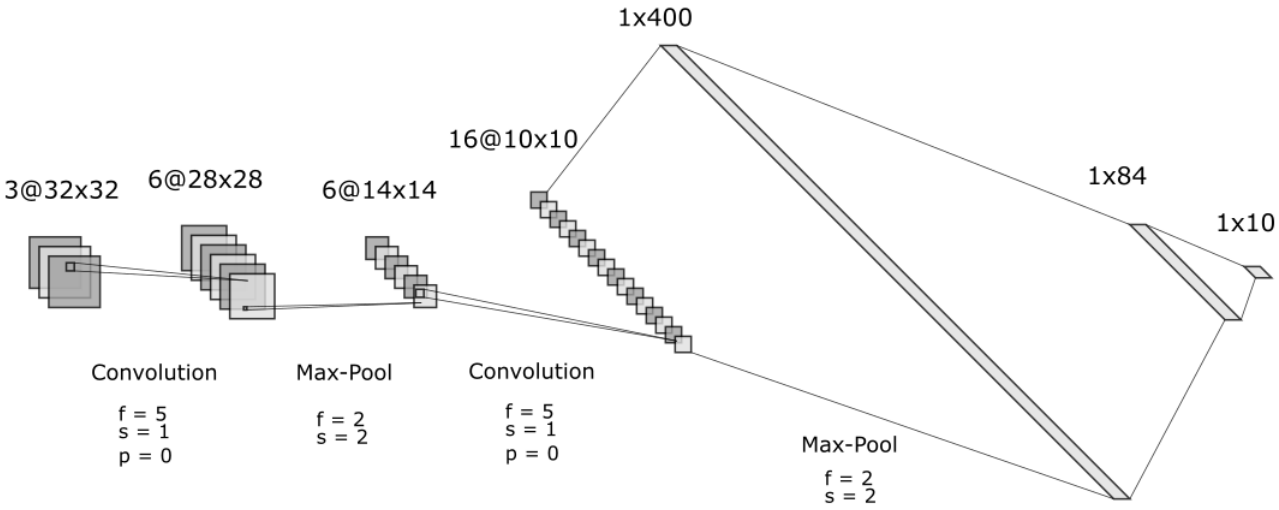
Projekt został zaimplementowany w języku Python w oparciu o technologię PyTorch. W projekcie zostały wykorzystane następujące pakiety:

1. torch
2. torchvision
3. matplotlib
4. numpy
5. IPython.core.debugger
6. os

3 Architektury

Zostały przetestowane trzy architektury. Były testowane z różnymi wartościami hiperparametrów oraz metodami augmentacji danych wejściowych. Pierwszą architekturą był LeNet-5, natomiast dwie kolejne to samodzielnie zaprojektowane architektury. W przypadku sieci LeNet-5 architektura została zaprezentowana w sposób graficzny, pozostałe ze względu na większą głębokość oraz zastosowanie dodatkowych metod regularyzacji zostały przedstawione w postaci tekstowej.

Model 1: LeNet-5



Rysunek 1: Architektura sieci LeNet-5

Model 2

Model drugi to niejako LeNet-5 z dodatkową warstwą konwolucyjną na początku oraz dropoutem w sieci gęstej.

```
(net):
(0): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU()
(2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU()
(4): MaxPool2d(kernel_size=2, stride=2, padding=0)
(5): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
(6): ReLU()
(7): MaxPool2d(kernel_size=2, stride=2, padding=0)
(8): Linear(in_features=1568, out_features=300, bias=True)
(9): ReLU()
(10): Dropout(p=0.5)
(11): Linear(in_features=300, out_features=10, bias=True)
```

Model 3

```
(net):
(0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): BatchNorm2d(16)
(2): ReLU()
(3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(4): BatchNorm2d(32)
(5): ReLU()
```

```
(6): MaxPool2d(kernel_size=2, stride=2, padding=0)
(7): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): BatchNorm2d(32)
(9): ReLU()
(10): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): BatchNorm2d(32)
(12): ReLU()
(13): MaxPool2d(kernel_size=2, stride=2, padding=0)
(14): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): BatchNorm2d(64)
(16): ReLU()
(17): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
(18): BatchNorm2d(128)
(19): ReLU()
(20): MaxPool2d(kernel_size=2, stride=2, padding=0)
(21): Linear(in_features=1152, out_features=1300, bias=True)
(22): ReLU()
(23): Dropout(p=0.5)
(24): Linear(in_features=1300, out_features=300, bias=True)
(25): ReLU()
(26): Dropout(p=0.5)
(27): Linear(in_features=300, out_features=10, bias=True)
```

4 Szczegółowy opis implementacji

Funkcja straty

W projekcie wykorzystano Cross-entropy jako funkcję straty w przypadku wszystkich modeli. Również ostatnią warstwą w każdym modelu jest warstwa log-softmax. Nie widać tego w przypadku architektury, gdyż funkcja obliczająca stratę jest zintegrowana z log-softmax, przez co wyjście z modelu musi być zwykłym wyjściem liniowym.

Hiperparametry

Hiperparametrami, którymi można sterować są:

1. wielkość mini-batch
2. wartość początkowego współczynnika uczenia
3. wartość współczynnika regularyzacji l2

Optymalizator i proces uczenia

W każdym przypadku optymalizatorem stosowanym w procesie uczenia jest Adam. Początkowo stosowano dwustopniowy trening, gdzie w pierwszej fazie stosowanym algorytmem był Adadelta, a następnie w kolejnej fazie model był dotrenowywany przy pomocy Adama. Podejście to jednak zostało zmienione i do uczenia zastosowano algorytm adaptacyjnej zmiany współczynnika uczenia w zależności od charakteru

funkcji straty. Algorytm ten przedstawia się następująco (strata dotyczy zbioru testowego w poniższym algorytmie):

1. Wprowadź początkowe wartości lr, l2_reg oraz maksymalną liczbę epok
2. Wykonaj epokę treningu
3. Jeżeli model ma mniejszą stratę niż poprzedni najlepszy model to zapisz model
4. Wczytaj najlepszy model i zmniejsz współczynnik uczenia 3 razy jeżeli
   - po epoce, wartość funkcji straty jest o 5% większa niż strata najlepszego modelu do tej pory
   - w dwóch poprzednich epokach funkcja straty rośnie
5. Zakończ trening jeżeli:
   - lr $<$ 1e-7
   - wczytano dwa razy ten sam model z rzędu
   - liczba epok przekroczy maksymalną liczbę epok
6. Wróć do punktu 2.

Algorytm ten pozwala na stosunkowo szybkie uczenie, większość przeprowadzonych treningów nie przekraczała 60 epok. Problemem, może być zbieganie do minimum lokalnego, jednak na trenowanych sieciach wyniki były zadowalające i zgodne z oczekiwanymi. W obecnej implementacji jest to podstawowy algorytm, natomiast w kolejnych wersjach programu można w prosty sposób dodać go jako element opcjonalny.

Śledzenie treningu

W implementacji zastosowano bardzo szczegółowe śledzenie treningu wraz z zapisywaniem modeli częściowych. Każdy trening generuje plik .txt z architekturą modelu, zastosowanymi transformacjami danych, opisem hiperparametrów oraz przebiegiem treningu, podawane są tutaj informacje odnośnie straty na obu zbiorach treningowym i testowym w danej epoce oraz dokładność predykcji na obu zbiorach. Wspomniane pliki .txt stanowią załącznik do niniejszego raportu.

5 Wyniki

We wszystkich testowanych modelach, jeżeli nie jest sprecyzowane inaczej to zostały zastosowane następujące hiperparametry oraz transformacje na zbiorze treningowym.

Hiperparametry:

Startowy współczynnik uczenia: 0.001
Współczynnik regularyzacji l2: 0.002
Mini-batch: 100

Augmentacja na zbiorze treningowym:

RandomCrop(size=(32, 32), padding=None)
RandomHorizontalFlip(p=0.5)
RandomRotation(degrees=(-15, 15), resample=False, expand=False)
ColorJitter(brightness=[0.8, 1.2], contrast=[0.8, 1.2], saturation=[0.8, 1.2], hue=None)
ToTensor()
Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])

Transformacja zbioru testowego:

ToTensor()
Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])

W dalszej części, jeżeli jest mowa o braku augmentacji, to oznacza to zastosowanie do zbioru treningowego takiej samej transformacji jak do zbioru testowego, która jest podana powyżej.

Na wykresach odpowiednie klasy oznaczone cyframi odpowiadają następującym klasom obiektów rzeczywistych:

Klasa numerycznie	Klasa rzeczywista
0	airplane
1	automobile
2	bird
3	cat
4	deer
5	dog
6	frog
7	horse
8	ship
9	truck

Przerywane linie na prezentowanych dalej krzywych uczenia rozdzielają etapy uczenia, w których sieć uczyła się z różnym learning rate.

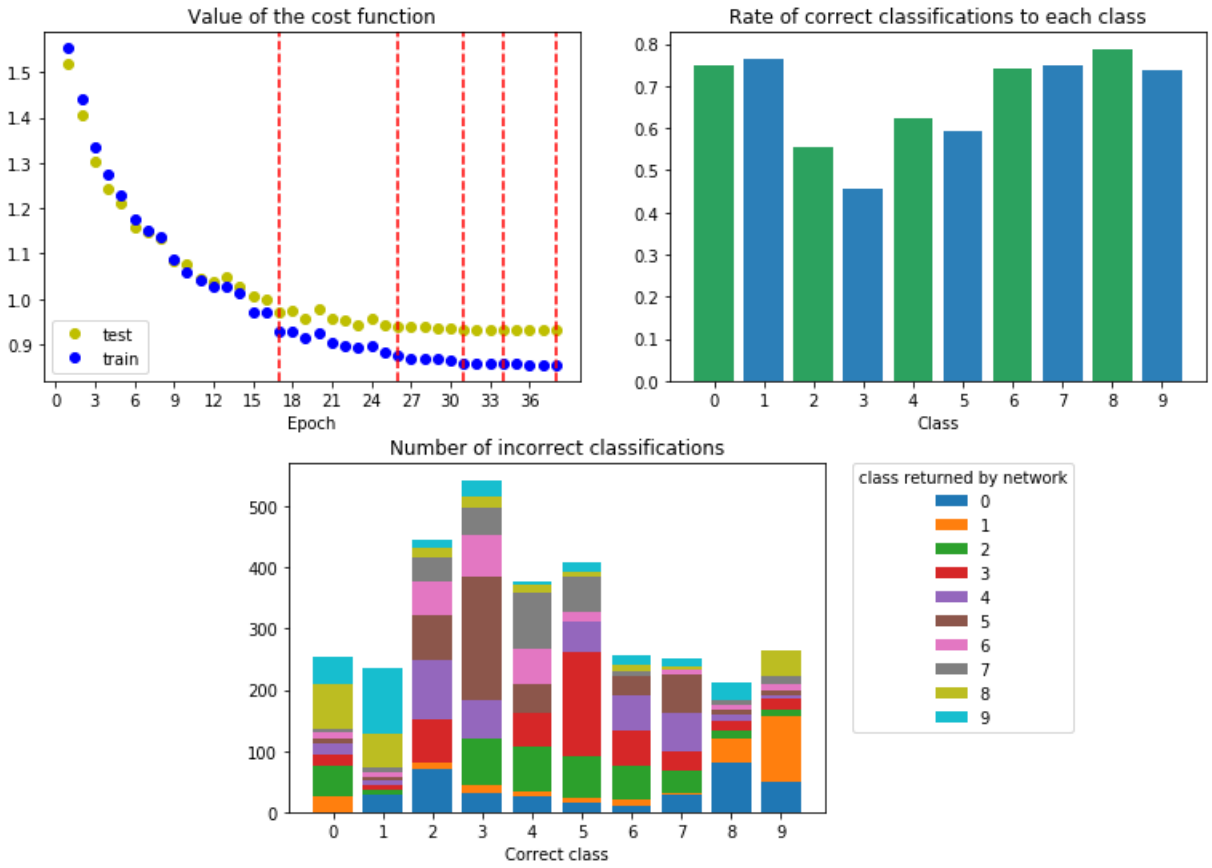
5.1 Model 1: LeNet-5

W poniższej sekcji przedstawiono wyniki na zbiorze LeNet-5. W pierszej części zostaną omówione najlepsze wyniki na tym zbiorze z zastosowaniem augmentacji danych oraz regularyzacji l2. W kolejnej zostaną zobrazowane krzywe uczenia i wyniki dla tej samej sieci z różnymi hiperparametrami i augmentacją danych wejściowych.

Najlepszy wynik

Skuteczność na zbiorze testowym	Skuteczność na zbiorze treningowym
69,86%	70,52%

Poniżej prezentacja wyników w postaci graficznej.

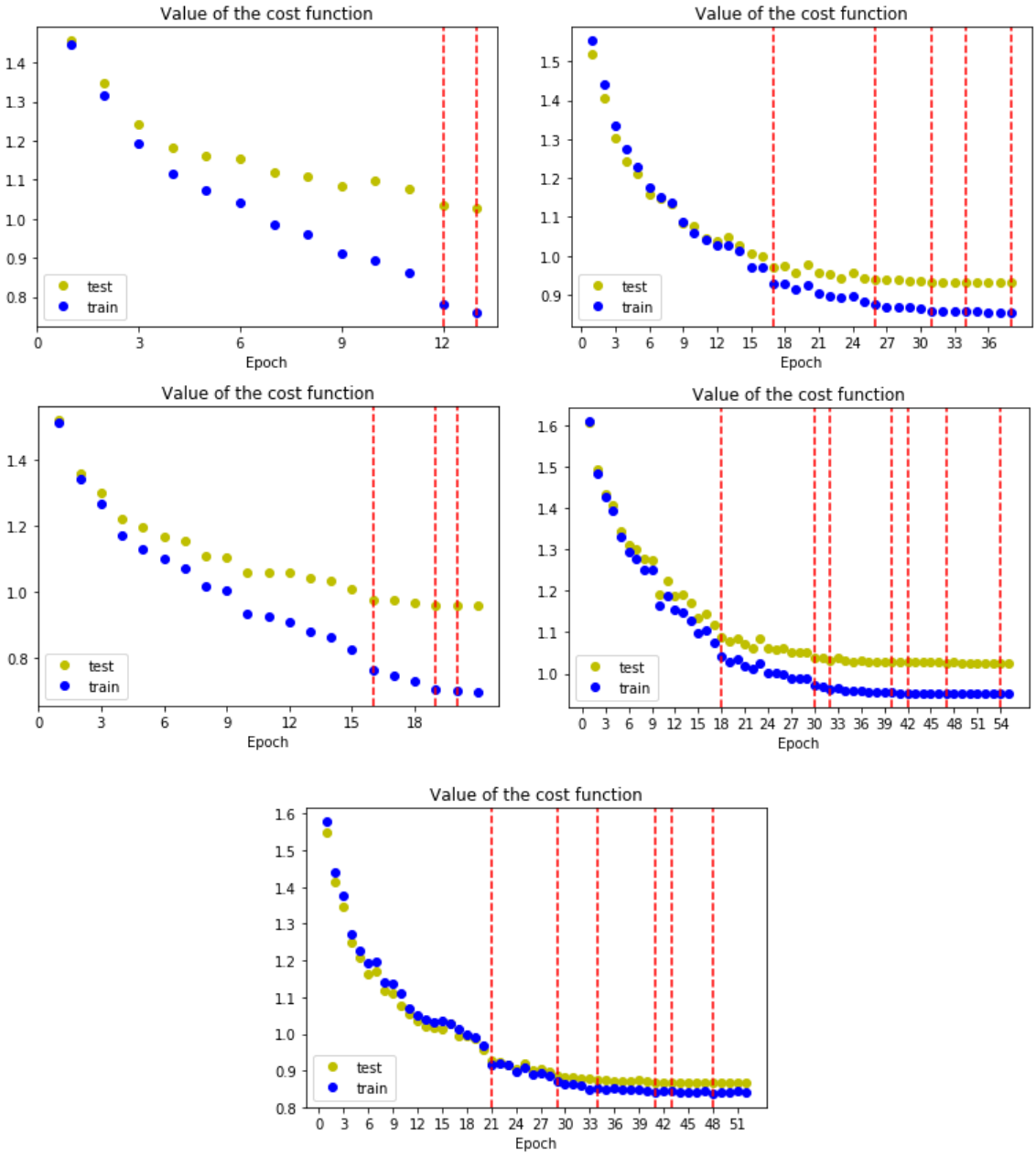


Rysunek 2: Wyniki dla sieci z zastosowaną augmentacją danych oraz wsp. regularyzacji l2 = 0.002

Model LeNet-5 przetrenowano sumarycznie w pięciu konfiguracjach:

1. l2_reg = 0 bez augmentacji
2. l2_reg = 0 z augmentacją
3. l2_reg = 0.002 bez augmentacji
4. l2_reg = 0.01 bez augmentacji
5. l2_reg = 0.002 z augmentacją

Poniżej zaprezentowano wykresy uczenia dla kolejnych konfiguracji (w kolejności wierszowej) oraz procent poprawnych klasyfikacji na zbiorach: treningowym oraz testowym.



Rysunek 3: Porównanie procesu uczenia dla różnych hiperparametrów w architekturze LeNet-5

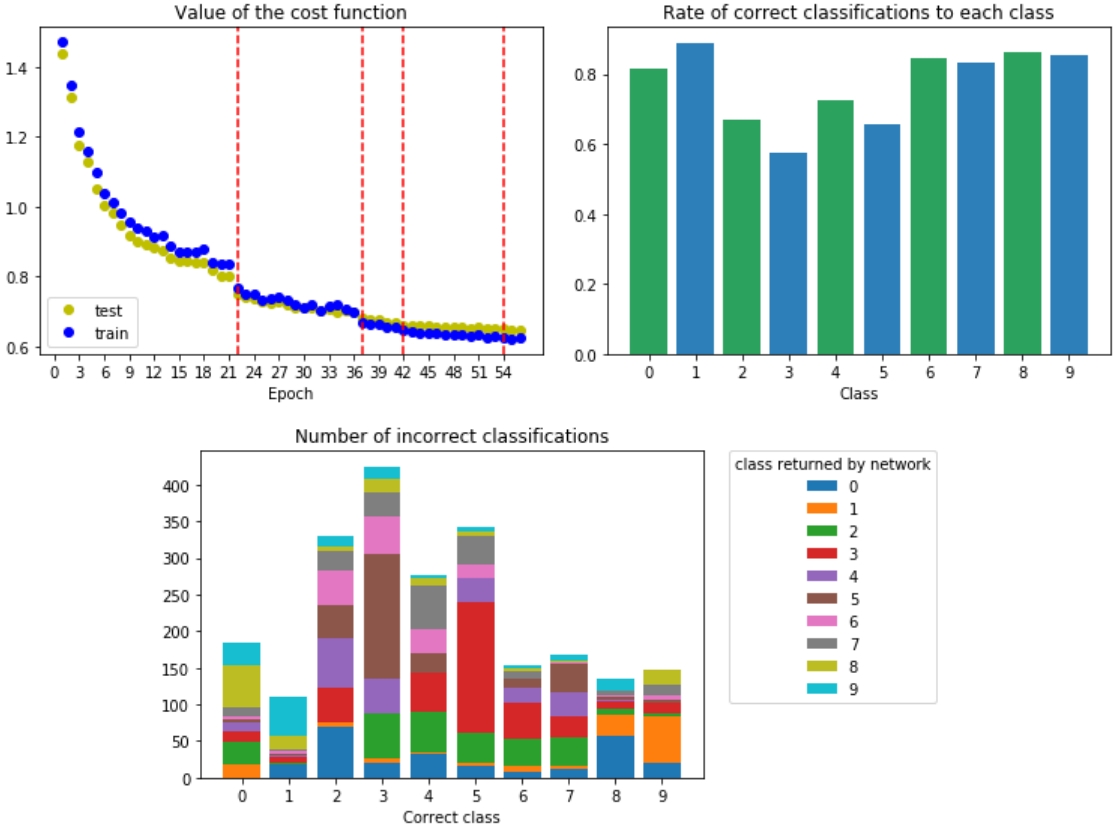
Model	Skuteczność na zbiorze testowym	Skuteczność na zbiorze treningowym
l2_reg = 0 bez augmentacji	64,31%	73,22%
l2_reg = 0 z augmentacją	67,54%	69,96%
l2_reg = 0.002 bez augmentacji	67,02%	75,99%
l2_reg = 0.01 bez augmentacji	64,29%	66,88%
l2_reg = 0.002 z augmentacją	69,86%	70,52%

5.2 Model 2

Model drugi to przede wszystkim LeNet-5 z dodatkową warstwą konwolucyjną oraz dropoutem. Wyniki jakie udało się uzyskać dla tego modelu przedstawione zostały w poniższej tabeli.

Skuteczność na zbiorze testowym	Skuteczność na zbiorze treningowym
77,29%	78,67%

Poniżej zestawiono wykresy przedstawiające proces uczenia oraz ostateczne rezultaty na zbiorze testowym.



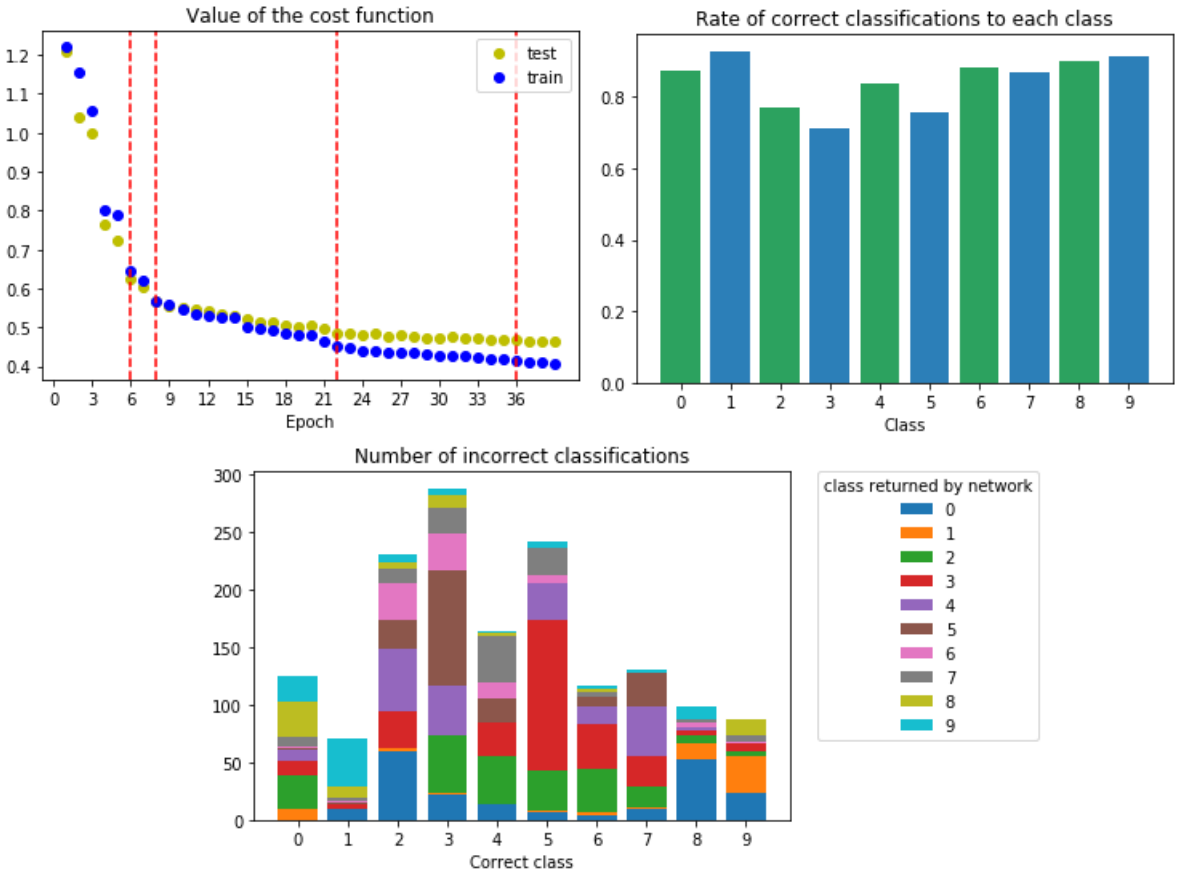
Rysunek 4: Wyniki dla modelu 2 z zastosowaną augmentacją danych oraz wsp. regularyzacji l2 = 0.002

5.3 Model 3

Model trzeci jest bardziej skomplikowany, posiada sumarycznie sześć warstw konwolucyjnych, trzy warstwy maxpool oraz trzy warstwy gęste.Wyniki jakie udało się uzyskać dla tego modelu przedstawione zostały w poniższej tabeli.

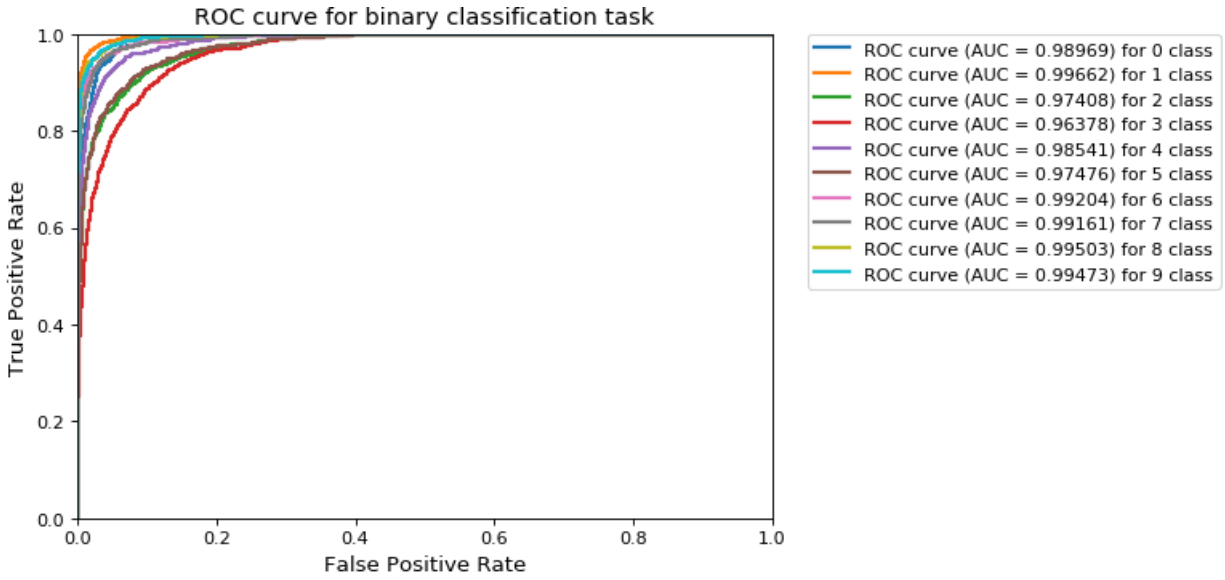
Skuteczność na zbiorze testowym	Skuteczność na zbiorze treningowym
84,51%	85,98%

Poniżej zestawiono wykresy przedstawiające proces uczenia oraz ostateczne rezultaty na zbiorze testowym.



Rysunek 5: Wyniki dla modelu 3 z zastosowaną augmentacją danych oraz wsp. regularyzacji l2 = 0.002

Program umożliwia również rysowanie krzywych ROC dla każdego modelu. W raporcie ograniczono się do wrysowania krzywych ROC dla najlepszego finalnego modelu.



Rysunek 6: Krzywa ROC i miary AUC dla problemu binarnej klasyfikacji w modelu 3

6 Wnioski

Dla zastosowanej sieci maksymalny wynik wyniósł ok. 84,5% co stanowi dobry wynik zważając na fakt, że nie zostały wykorzystane sieci pretrenowane ani komitety klasyfikatorów. Przykład modelu pierwszego wskazuje na istotny wpływ augmentacji danych oraz regularyzacji w procesie treningu rozważając przeuczenie modelu, ale również dokładność klasyfikacji. Wszystkie przykłady pokazują, że stosunkowo dobry wynik można uzyskać na niedużej sieci, natomiast wzrost głębokości sieci powoduje malejący wzrost dokładności klasyfikacji.
Jako wnioski dotyczące ewentualnego dalszego rozwoju projektu należy wskazać zwrócenie uwagi na większą przenośność kodu. Obecny program nie jest w pełni przenośny, niemniej zapewnia dość dużą modularność. Kolejnym aspektem jest uwzględnienie sieci pretrenowanych oraz komitetów sieci w projekcie. Jako szersze przetestowanie działania zaproponowanego algorytmu uczenia, sugerowane jest wytrenowanie większej liczby sieci przy zastosowaniu wspomnianego algorytmu oraz samego Adama (gdyż jest to domyślny optymalizator w projekcie).

7 Załączniki

Do niniejszego raportu załączone zostały następujące pliki:

1. Kod programu w postaci notatnika Jupyter
2. Ostateczne modele opisanych sieci, możliwe do wczytania przez funkcję load_model(path)
3. Logi z treningu opisanych modeli