MRO - zadanie 3: Pierwszy i prawdopodobnie ostatni raz w życiu (trenujemy od podstaw konwolucyjną sieć neuronową)

Piotr Zawiślan

6 Listopad 2022

1 Wykorzystane technologie

Kod pisałem w języku Python z wykorzystaniem frameworka Pytorch.

2 Dane, dane, me królestwo za dane

Na początku wczytałem zbiór *CIFAR10* (wersję treningową i testową) z wykorzystaniem wbudowanej klasy *torchvision.datasets.CIFAR10*. Przeskalowałem wartości obrazów do zakresu $0-1.0$ przy użyciu transformacji *ToTensor* oraz zamieniłem etykiety na wersję *OneHot* z wykorzystaniem funkcji *scatter*.

```
cifar_train_ds = datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(
        lambda y: torch.zeros(10, dtype=torch.float).scatter_(0, torch.tensor(y),
    value=1)
    )
)
```

Listing 1: Wczytanie datasetu (treningowego)

Część zbioru treningowego wydzieliłem na zbiór walidacyjny.

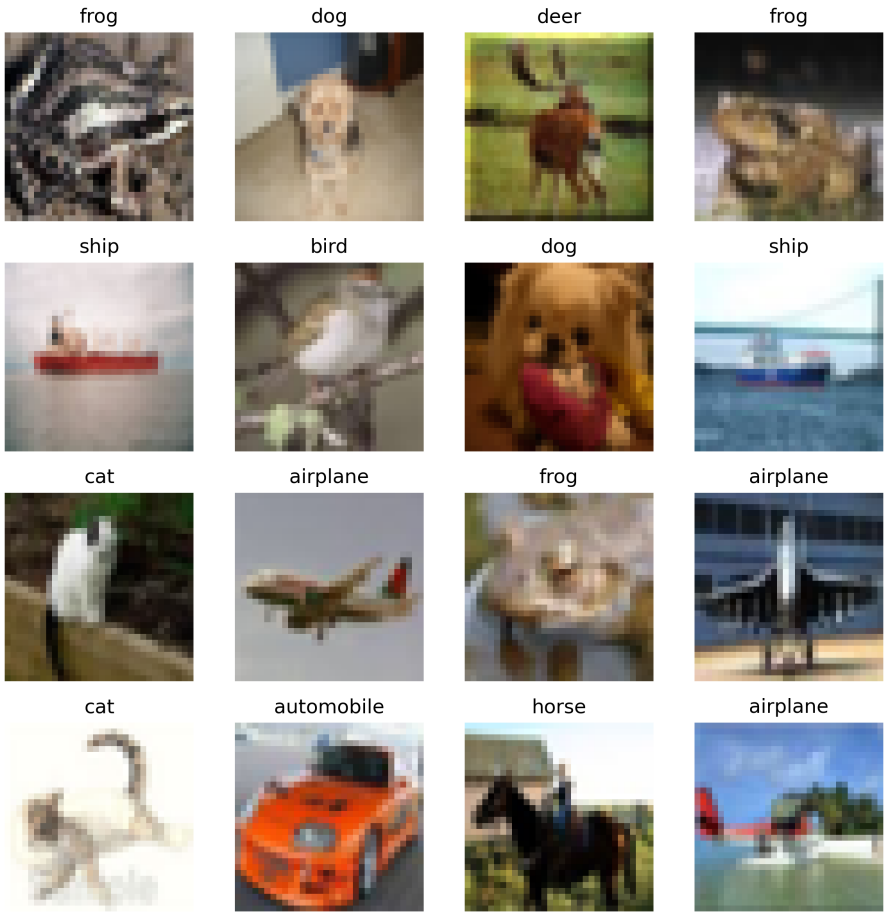
```
full_train_ds, full_val_ds = random_split(cifar_train_ds, (42000, 8000))
```

Listing 2: Podział na zbiór treningowy i walidacyjny

Aby przetestować działanie późniejszych modeli i funkcji trenujących/plotujących utworzyłem też mniejsze zbiory danych.

```
trimmed_train_ds = Subset(full_train_ds, indices=random.sample(range(len(full_train_ds
    )), 2048))
trimmed_val_ds = Subset(full_val_ds, indices=random.sample(range(len(full_val_ds)),
    1048))
```

Listing 3: Utworzenie mniejszych zbiorów



Rysunek 1: Losowe próbki i etykiety ze zbioru treningowego

3 Minimalna architektura

3.1 Model A

Na poniższym skrawku kodu zaimplementowałem startową mikro-architekturę.

```
class CNNModelA(nn.Module):
    def __init__(self):
        super().__init__()
        self.sequence = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=5, kernel_size=3, padding='same'),
            nn.Sigmoid(),
            nn.Conv2d(in_channels=5, out_channels=5, kernel_size=3, padding='same'),
            nn.Sigmoid(),
            nn.MaxPool2d(kernel_size=8),
            nn.Flatten(),
            nn.Linear(in_features=5*4*4, out_features=10),
            nn.Softmax(dim=1)
        )

    def forward(self, x: torch.Tensor):
        probs = self.sequence(x)
        return probs
```

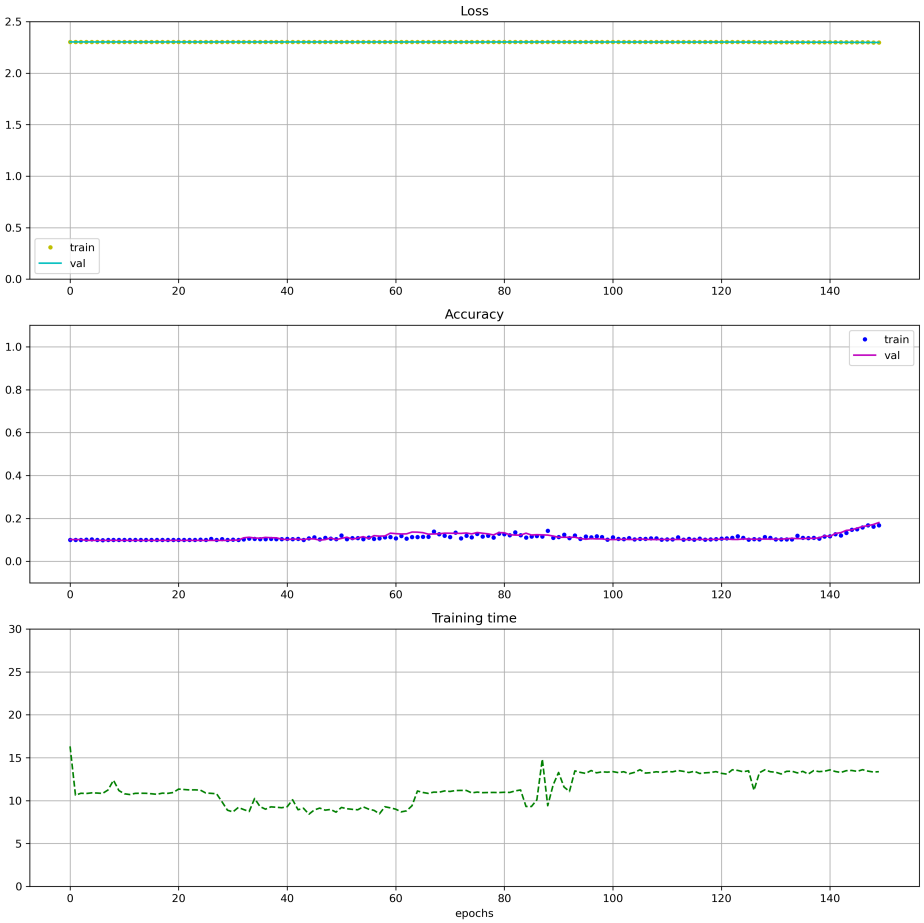
Listing 4: Implementacja modelu A

Na ten model składa się $(3 \cdot 3 + 1) \cdot 5 \cdot 2 + 5 \cdot 4 \cdot 4 + 1 = 181$ parametrów (jeśli dobrze liczę).
Bez treningu model przewiduje takie wektory prawdopodobieństw jak ten:

$$(0.132, 0.081, 0.131, 0.096, 0.064, 0.086, 0.100, 0.123, 0.091, 0.096)$$

Przy treningu wykorzystałem zaimplementowaną przez siebie klasę *Trainer* przyjmującej między innymi model i zbiory treningowy i walidacyjny. Klasa ta ma dwie metody: *train_step* oraz *val_step*. Po treningu można uzyskać z niej model oraz historię treningu.

Zaimplementowałem też funkcję plotującą tę historię treningu. Prezentuje ona jak wyglądała średnia wartość funkcji straty (entropii krzyżowej) oraz średnia dokładność na danych treningowych oraz walidacyjnych. Pokazuje również ile trwał trening każdej z epok w sekundach.



Rysunek 2: Wyniki treningu modelu A

Dokładność modelu na zbiorze walidacyjnym: 0.18

Z wykresu da się wywnioskować, że model zaczął się poprawiać dopiero w okolicach epoki 140 i to bardzo powoli. Może to wynikać z małej głębokości sieci, małej stałej uczenia (0.001) bądź aktywacji sigmoidą, a nie ReLU. Wyniki treningu raczej nie są zadowalające.

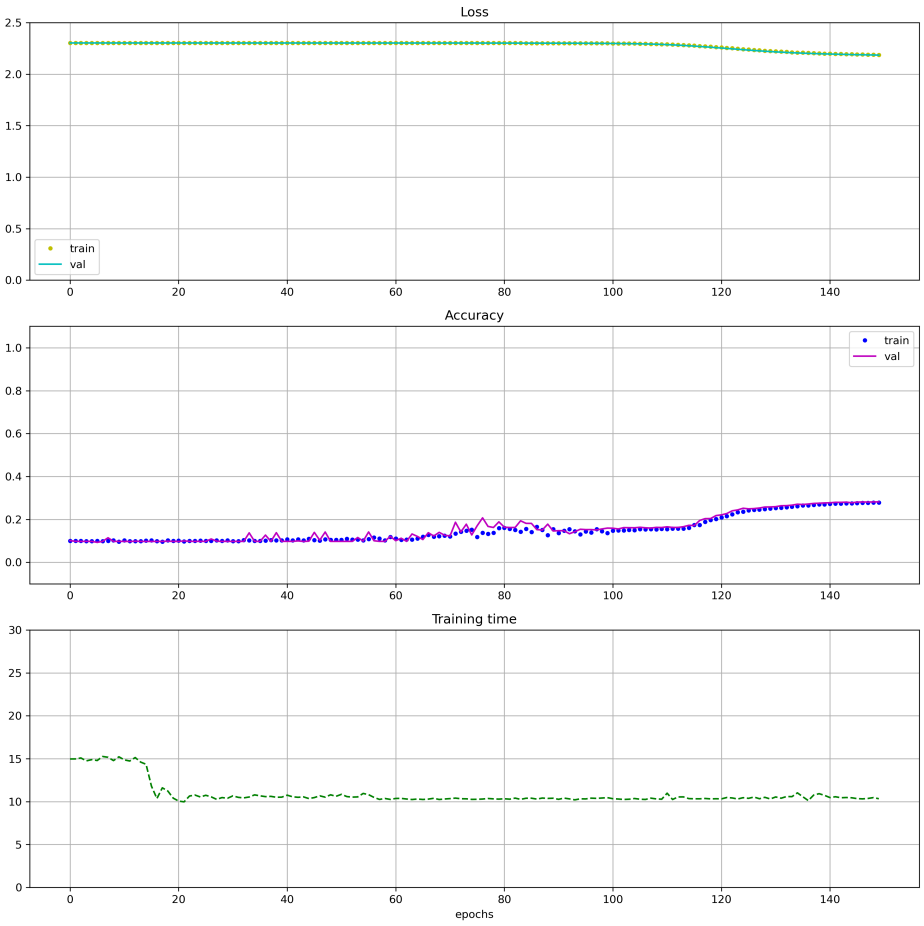
Po treningu model przewiduje takie wektory prawdopodobieństw jak ten:

$$(0.126, 0.087, 0.074, 0.098, 0.081, 0.101, 0.111, 0.077, 0.119, 0.116)$$

4 What? NETWORK is evolving!

4.1 Model B

Po zwiększeniu liczby fitrów w warstwach konwolucyjnych z 5 do 20 i 150 epok treningu otrzymałem takie wyniki:



Rysunek 3: Wyniki treningu modelu B

Dokładność modelu na zbiorze walidacyjnym: 0.28

Widzimy, że dokładność modelu zaczęła się poprawiać nieco wcześniej (w okolicy 110-tej epoki zaczęła się piąć w górę). To wciąż jednak dość wolno.

4.2 Funkcja zwracająca blok konwolucyjny

```python
def get_conv_block(
    in_ch: int, mid_ch: int, out_ch: int,
    kernel_a_size: int = 3, kernel_b_size: int = 3,
    pool_size: int = 2, activation_func_str: Literal['relu', 'sigmoid'] = 'relu',
    batch_norm: bool = False,
    dropout_rate: Union[float, None] = None):

    if activation_func_str == 'relu':
        activation_func = nn.ReLU
    elif activation_func_str == 'sigmoid':
        activation_func = nn.Sigmoid
    else:
        raise ValueError("No such activation func to choose.")

    seq_conv_1 = [
        nn.Conv2d(
            in_channels=in_ch, out_channels=mid_ch,
            kernel_size=kernel_a_size, stride=1, padding='same'),
        activation_func()
    ]

    seq_conv_2 = [
        nn.Conv2d(
            in_channels=mid_ch, out_channels=out_ch,
            kernel_size=kernel_b_size, stride=1, padding='same'),
        activation_func()
    ]

    if batch_norm:
        seq_conv_1.append(nn.BatchNorm2d(num_features=mid_ch))
        seq_conv_2.append(nn.BatchNorm2d(num_features=out_ch))

    seq_max_pool = [
        nn.MaxPool2d(kernel_size=pool_size)
    ]

    if dropout_rate is not None:
        seq_max_pool.append(nn.Dropout(p=dropout_rate))

    return nn.Sequential(*seq_conv_1, *seq_conv_2, *seq_max_pool)
```

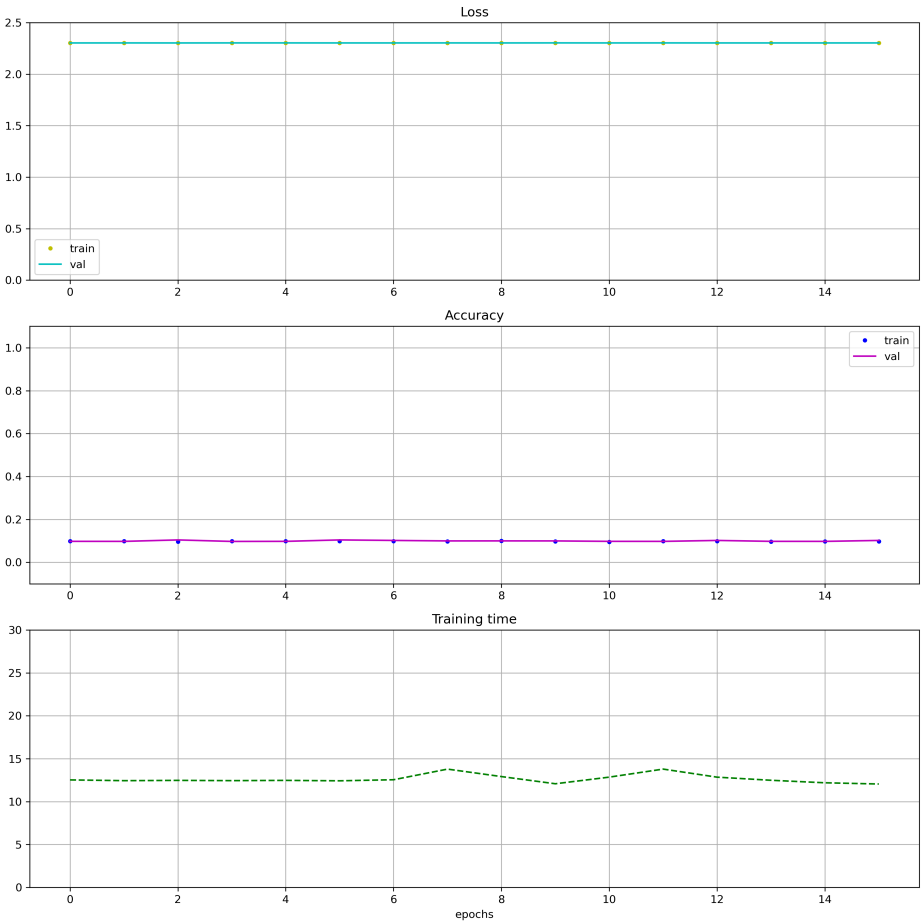
Listing 5: Funkcja get_conv_block

4.3 Model C

```python
self.sequence = nn.Sequential(
    get_conv_block(in_ch=3, mid_ch=20, out_ch=20, activation_func_str='sigmoid'),
    get_conv_block(in_ch=20, mid_ch=40, out_ch=40, activation_func_str='sigmoid'),
    nn.Flatten(),
    nn.Linear(in_features=40*(32 // 4)**2, out_features=10),
    nn.Softmax(dim=1)
)
```

Listing 6: Warstwy modelu C

Po 16-tu epokach treningu nie obserwujemy zaskakujących wyników.



Rysunek 4: Wyniki treningu modelu C

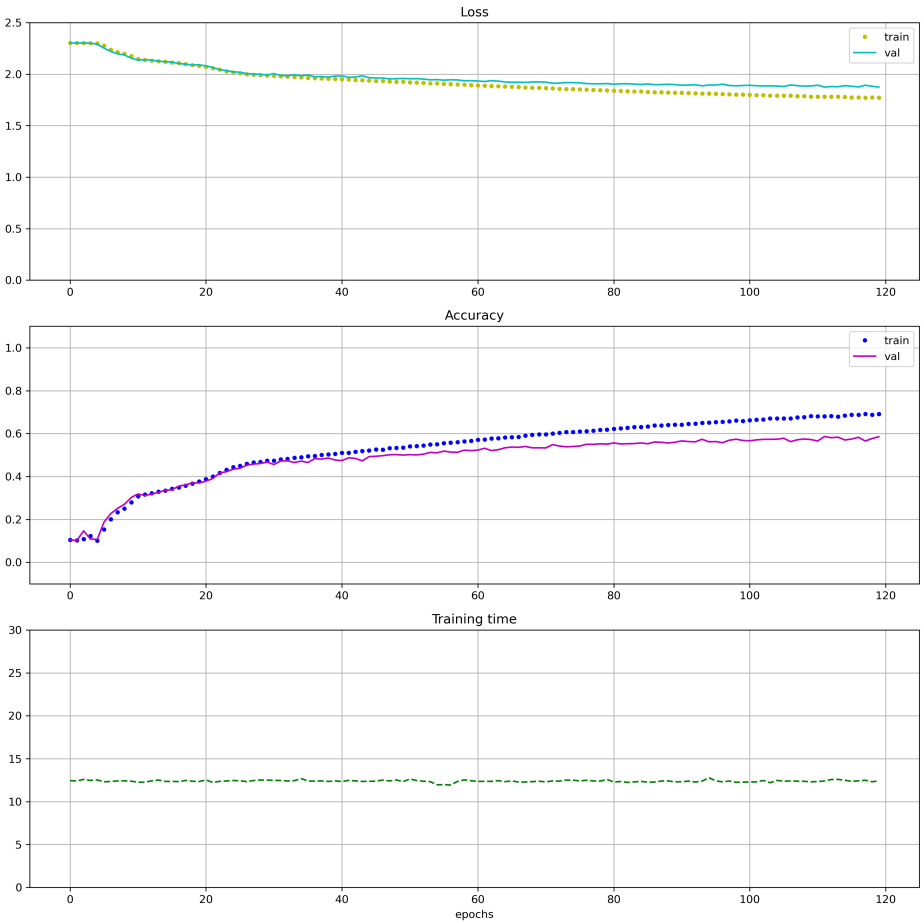
Dokładność modelu na zbiorze walidacyjnym: 0.10

W modelu D zamienimy funkcje aktywacji softmax na ReLU.

4.4 Model D

```
self.sequence = nn.Sequential(
    get_conv_block(in_ch=3, mid_ch=20, out_ch=20, activation_func_str='relu'),
    get_conv_block(in_ch=20, mid_ch=40, out_ch=40, activation_func_str='relu'),
    nn.Flatten(),
    nn.Linear(in_features=40*(32 // 4)**2, out_features=10),
    nn.Softmax(dim=1)
)
```

Listing 7: Warstwy modelu D



Rysunek 5: Wyniki treningu modelu D

Dokładność modelu na zbiorze walidacyjnym: 0.59

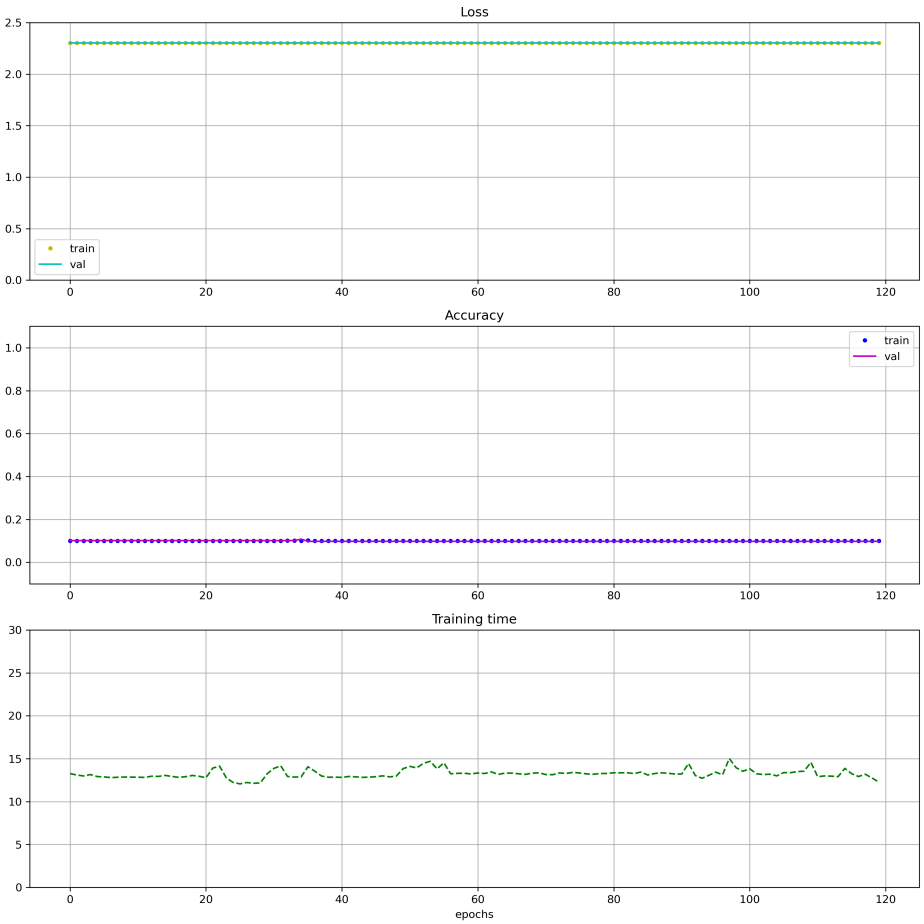
Nareszcie model zaczyna się uczyć :) Mimo to daje się zauważyć duża różnica pomiędzy dokładnością na zbiorze treningowym i walidacyjnym. Może to znaczyć, że model ulega przetrenowaniu (czy raczej nadmiernemu dopasowaniu).

4.5 Model E

Teraz zamiast dwóch bloków konwolucyjnych użyjemy czterech.

```
self.sequence = nn.Sequential(
    get_conv_block(in_ch=3, mid_ch=20, out_ch=20, activation_func_str='relu'),
    get_conv_block(in_ch=20, mid_ch=40, out_ch=40, activation_func_str='relu'),
    get_conv_block(in_ch=40, mid_ch=80, out_ch=80, activation_func_str='relu'),
    get_conv_block(in_ch=80, mid_ch=160, out_ch=160, activation_func_str='relu'),
    nn.Flatten(),
    nn.Linear(in_features=160*(32 // 16)**2, out_features=10),
    nn.Softmax(dim=1)
)
```

Listing 8: Warstwy modelu E



Rysunek 6: Wyniki treningu modelu E

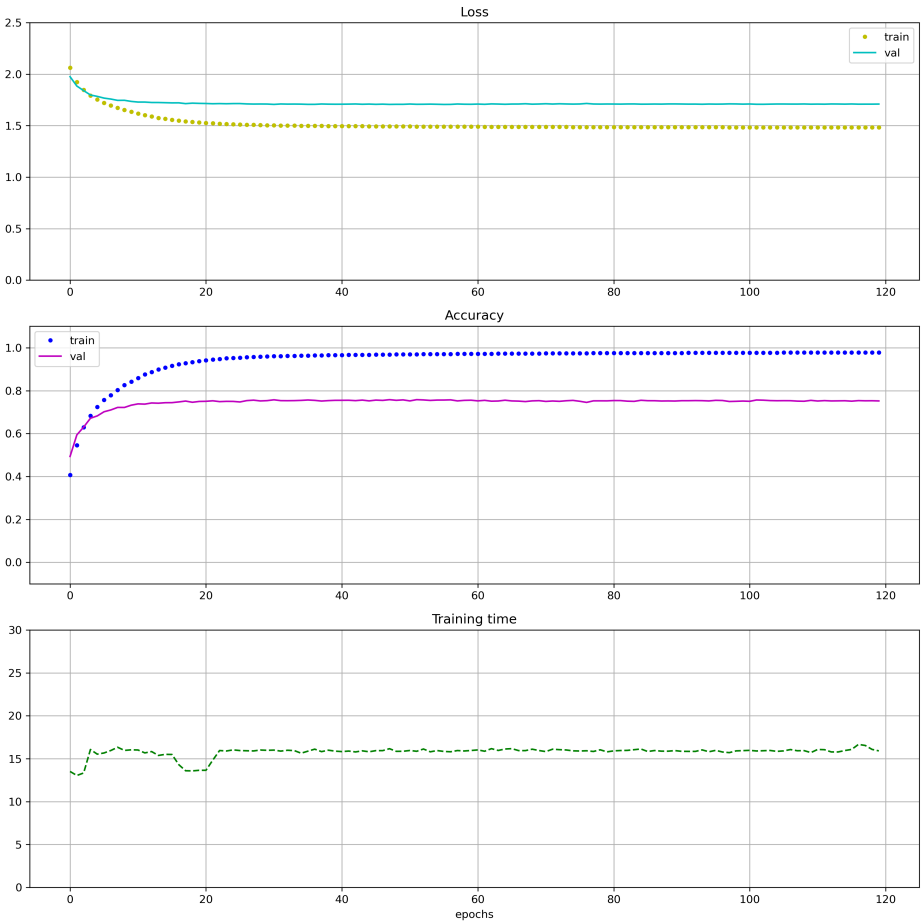
Dokładność modelu na zbiorze walidacyjnym: 0.01

Model nie uczy się w ogóle. Może to wynikać z tego, że jest za głęboki i potrzebuje warstw normalizujących. To właśnie dodamy w modelu F.

4.6 Model F

```
self.sequence = nn.Sequential(
    get_conv_block(in_ch=3, mid_ch=20, out_ch=20, batch_norm=True),
    get_conv_block(in_ch=20, mid_ch=40, out_ch=40, batch_norm=True),
    get_conv_block(in_ch=40, mid_ch=80, out_ch=80, batch_norm=True),
    get_conv_block(in_ch=80, mid_ch=160, out_ch=160, batch_norm=True),
    nn.Flatten(),
    nn.Linear(in_features=160*(32 // 16)**2, out_features=10),
    nn.Softmax(dim=1)
)
```

Listing 9: Warstwy modelu F



Rysunek 7: Wyniki treningu modelu F

- Dokładność na zbiorze treningowym: 0.978
- Dokładność na zbiorze walidacyjnym: 0.75
- Dokładność na zbiorze testowym: 0.75

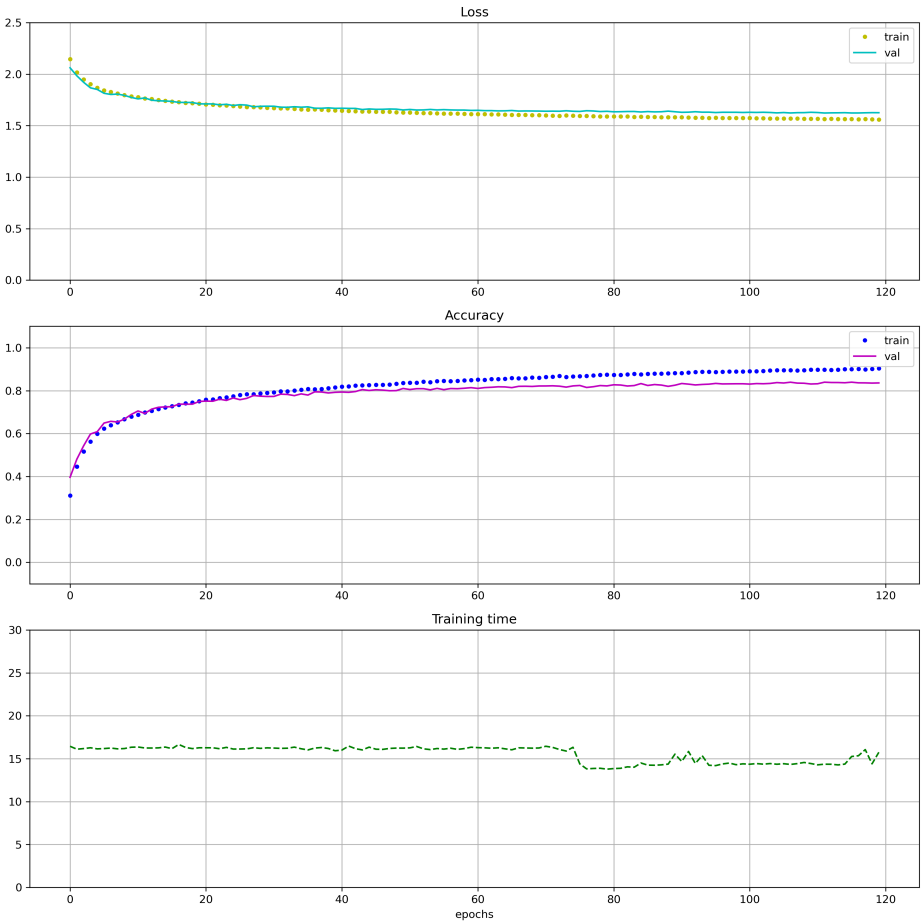
Model się wyuczył niemal perfekcyjnie danych treningowych, ale może mieć problemy z generalizacją, co wnioskujemy na podstawie różnicy pomiędzy dokładnością na zbiorze treningowym i walidacyjnym wynoszącej ponad 0.2.

4.7 Model G

Tym razem dodamy warstwy *Dropout*, aby uniknąć overfitting'u.

```
self.sequence = nn.Sequential(
    get_conv_block(in_ch=3, mid_ch=20, out_ch=20, batch_norm=True, dropout_rate=0.1),
    get_conv_block(in_ch=20, mid_ch=40, out_ch=40, batch_norm=True, dropout_rate=0.2),
    get_conv_block(in_ch=40, mid_ch=80, out_ch=80, batch_norm=True, dropout_rate=0.3),
    get_conv_block(in_ch=80, mid_ch=160, out_ch=160, batch_norm=True, dropout_rate
    =0.4),
    nn.Flatten(),
    nn.Linear(in_features=160*(32 // 16)**2, out_features=10),
    nn.Softmax(dim=1)
)
```

Listing 10: Warstwy modelu G



Rysunek 8: Wyniki treningu modelu G

- Dokładność na zbiorze treningowym: 0.90
- Dokładność na zbiorze walidacyjnym: 0.83
- Dokładność na zbiorze testowym: 0.83

Tym razem trening modelu zadziałał o wiele lepiej. Nie ma aż tak dużej różnicy pomiędzy dokładnością na zbiorach treningowym i walidacyjnym/testowym, więc można stwierdzić, że model będzie się dobrze generalizował.

5 Mind the GAP!

5.1 Model GAP

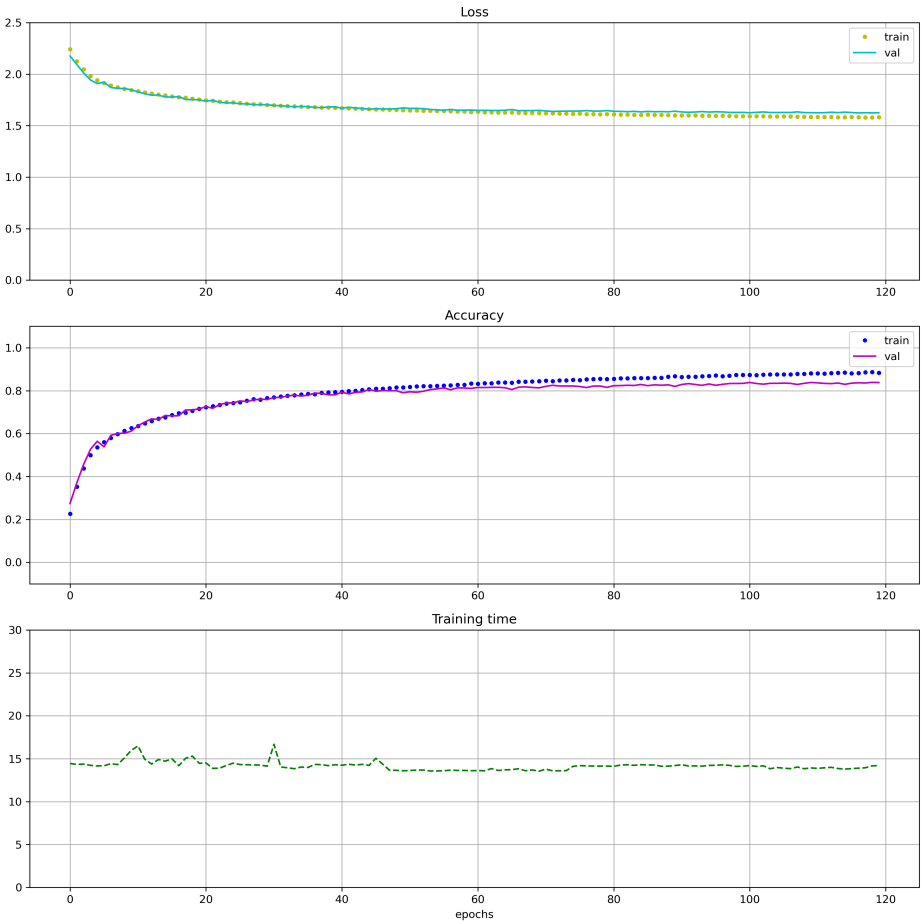
Na koniec, zamiast ostatniej warstwy MaxPooling zastosujemy warstwę GlobalAveragePooling.

```
self.sequence = nn.Sequential(
    get_conv_block(in_ch=3, mid_ch=20, out_ch=20, batch_norm=True, dropout_rate=0.1),
    get_conv_block(in_ch=20, mid_ch=40, out_ch=40, batch_norm=True, dropout_rate=0.2),
    get_conv_block(in_ch=40, mid_ch=80, out_ch=80, batch_norm=True, dropout_rate=0.3),

    nn.Conv2d(
        in_channels=80, out_channels=160,
        kernel_size=3, stride=1, padding='same'),
    nn.ReLU(),
    nn.BatchNorm2d(num_features=160),
    nn.Conv2d(
        in_channels=160, out_channels=160,
        kernel_size=3, stride=1, padding='same'),
    nn.ReLU(),
    nn.BatchNorm2d(num_features=160),
    nn.AdaptiveAvgPool2d(output_size=(1, 1)),  # GAP
    nn.Dropout(p=0.4),

    nn.Flatten(),
    nn.Linear(in_features=160, out_features=10),
    nn.Softmax(dim=1)
)
```

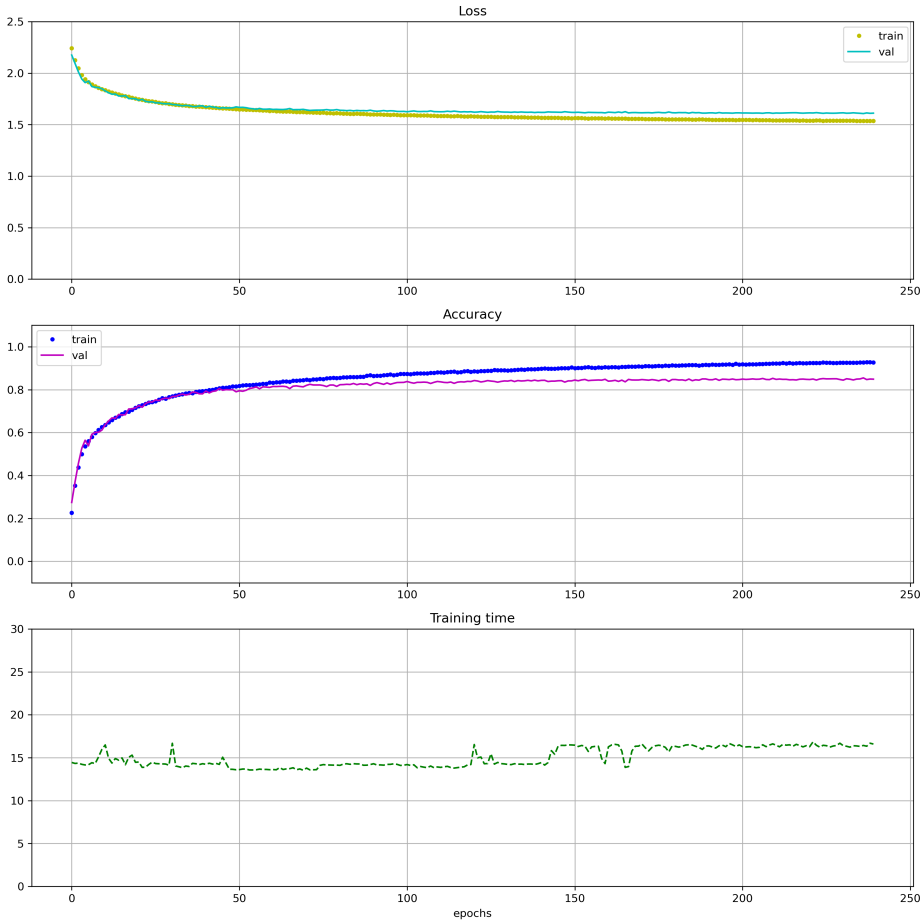
Listing 11: Warstwy modelu GAP



Rysunek 9: Wyniki treningu modelu GAP po 120 epokach treningu.

- Dokładność na zbiorze treningowym: 0.88
- Dokładność na zbiorze walidacyjnym: 0.83
- Dokładność na zbiorze testowym: 0.83

Poniżej wyniki treningu po kolejnych 120-tu epokach.



Rysunek 10: Wyniki treningu modelu GAP po 240 epokach treningu.

- Dokładność na zbiorze treningowym: 0.93
- Dokładność na zbiorze walidacyjnym: 0.85
- Dokładność na zbiorze testowym: 0.84

5.2 Test na zdjęciach o rozmiarze innym niż 32×32

Przygotowałem 2 zdjęcia: kota oraz żabę.

Rysunek 11: Kot: 32×20

Rysunek 12: Żaba: 32×18

W przypadku kota model GAP myślał, że to samolot na 97.3%. Kot miał dopiero 2.5%. Może to być przez fakt, że model nie był trenowany na tak długich, leżących kotach :).

Jeżeli chodzi o żabę - model był pewny (99.9999%), że to rzeczywiście żaba.

6 Wnioski

Wioski są takie:

- przy rozpoznawaniu obrazów *CNN* musi mieć pewną głębokość zanim zacznie dawać dobre wyniki,
- *ReLU* działa o wiele lepiej (zwalcza problem zanikającego gradientu) niż sigmoida jako funkcja aktywacji wewnątrz sieci,
- zastosowanie *Batch normalization* po warstwach konwolucyjnych może sprawić, że model będzie w stanie zacząć się odpowiednio dopasowywać,
- użycie warstw *Dropout* sprawia, że model się o wiele lepiej generalizuje,
- warstwa *Global Average Pooling* sprawia, że model może obsługiwać obrazy o różnych rozmiarach, a liczba aktywacji na wejściu końcowej warstwy liniowej nie zależy wtedy od rozmiaru obrazu, tylko od liczby kanałów.