WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

Sprawozdanie

Zaawansowane metody uczenia maszynowego

Wielowarstwowe sieci neuronowe

Autor:

Karol Baranowski

Prowadzący:

mgr inż. Przemysław Czuba

Spis treści

Zadanie

1.Eksperymentalnie:

- wybrać liczbę neuronów oraz warstw sieci

- wybrać liczbę epok uczenia

- znaleźć miejsca na przyspieszenie oraz propozycje zmiany (mając na uwadze, że pętle 'for' są bardzo wolne)

- jak długo trwa uczenie dla różnych konfiguracji liczby neuronów oraz warstw?

2. Wypisać funkcje biblioteki PyTorch użyte w programie wraz z krótki opis działania

oraz dlaczego zostały zastosowane

3. Odpowiedzieć na pytania zadane w komentarzach.

Zadanie I

Epok: 1 Pierwsza warstwa: 100 neuronów Druga warstwa: 50 neuronów

```
Epoch: 01 | Epoch Time: 0m 15s
        Train Loss: 0.525 | Train Acc: 83.70%
         Val. Loss: 0.208 |  Val. Acc: 94.08%
Test Loss: 0.180 | Test Acc: 94.73%
```

Epok: 2 Pierwsza warstwa: 100 neuronów Druga warstwa: 50 neuronów

```
Epoch: 02 | Epoch Time: 0m 19s
        Train Loss: 0.182 | Train Acc: 94.45%
         Val. Loss: 0.125 |  Val. Acc: 95.96%
Test Loss: 0.108 | Test Acc: 96.74%
```

Epok: 10 Pierwsza warstwa: 100 neuronów Druga warstwa: 50 neuronów

```
Epoch: 10 | Epoch Time: 0m 20s
        Train Loss: 0.098 | Train Acc: 96.93%
         Val. Loss: 0.081 |  Val. Acc: 97.55%
Test Loss: 0.063 | Test Acc: 97.90%
```

Wraz ze wzrostem iteracji uczenia celność dla 100 i 50 neuronów rośnie. Dla 10 epok wynosi prawie 98%.

Epok: 20 Pierwsza warstwa: 50 neuronów Druga warstwa: 25 neuronów

```
Epoch: 20 | Epoch Time: 0m 24s
        Train Loss: 0.127 | Train Acc: 96.17%
         Val. Loss: 0.094 |  Val. Acc: 97.25%
Test Loss: 0.075 | Test Acc: 97.65%
```

Mimo, że neuronów w obu warstwach jest o połowę mniej czas epoki nie uległ znacznej poprawie. Aby osiągnąć wynik zbliżony dla 10 epok warstwy 100 i 50 neuronowej należy wykonać aż o 10 iteracji więcej, co wyraźnie pokazuje, że sieć uczy się skuteczniej dla większej ilości neuronów w warstwach.

Epok: 2 Pierwsza warstwa: 150 neuronów Druga warstwa: 75 neuronów

```
Epoch: 02 | Epoch Time: 0m 23s
        Train Loss: 0.194 | Train Acc: 94.14%
         Val. Loss: 0.126 |  Val. Acc: 95.97%
Test Loss: 0.104 | Test Acc: 96.59%
```

W tym przykładzie zwiększono liczbę neuronów o 1/3 w stosunku do ich wartości początkowych. Wynik po 2 epokach jest gorszy niż dla tych wartości. Znaczy to, że ilość neuronów można zwiększać tylko do pewnego stopnia, potem to nie wpływa na wynik, a nawet go pogarsza jak w tym przypadku.

Epok: 2 Pierwsza warstwa: 110 neuronów Druga warstwa: 60 neuronów

```
Epoch: 02 | Epoch Time: 0m 21s
        Train Loss: 0.208 | Train Acc: 93.76%
         Val. Loss: 0.134 |  Val. Acc: 95.70%
Test Loss: 0.114 | Test Acc: 96.21%
```

Delikatne zwiększenie liczby neuronów w stosunku do wartości początkowych nie poprawia uczenia sieci. Znaczy to, że wartości 100 i 50 neuronów można uznać za optymalne (metodą eksperymentalną, ponieważ ciągle mogą istnieć kombinacje, liczą lepiej). Natomiast wynik dla tych parametrów po 10 iteracjach – 98% dokładności dla zbioru testowego jest bardzo zadowalający.

Epok: 2 Pierwsza warstwa: 200 neuronów Druga warstwa: 100 neuronów Trzecia warstwa: 50 neuronów

```
Epoch: 02 | Epoch Time: 0m 24s
        Train Loss: 0.189 | Train Acc: 94.13%
         Val. Loss: 0.135 |  Val. Acc: 95.64%
Test Loss: 0.109 | Test Acc: 96.40%
```

Dodano warstwę z 200 neuronami. Dla 2 epok nie widać znaczącej różnicy.

Epok: 10 Pierwsza warstwa: 200 neuronów Druga warstwa: 100 neuronów Trzecia warstwa: 50 neuronów

```
Epoch: 10 | Epoch Time: 0m 22s
        Train Loss: 0.076 | Train Acc: 97.60%
         Val. Loss: 0.067 |  Val. Acc: 97.90%
Test Loss: 0.057 | Test Acc: 98.19%
```

Dla 10 epok widać już poprawę dokładności o ~0.3%.

Epok: 10 Pierwsza warstwa: 500 neuronów Druga warstwa: 100 neuronów Trzecia warstwa: 50 neuronów

```
Epoch: 10 | Epoch Time: 0m 23s
        Train Loss: 0.075 | Train Acc: 97.63%
         Val. Loss: 0.062 |  Val. Acc: 98.27%
Test Loss: 0.055 | Test Acc: 98.36%
```

Zwiększenie ilości neuronów w pierwszej warstwie trójwarstwowego modelu do 500 jeszcze mocniej poprawiło uczenie. Po 10 epokach dokładność sieci wynosi 98.36%.

Epok: 10 Pierwsza warstwa: 1000 neuronów Druga warstwa: 500 neuronów Trzecia warstwa: 100 neuronów Czwarta warstwa: 50

```
Epoch: 10 | Epoch Time: 0m 41s
        Train Loss: 0.073 | Train Acc: 97.77%
         Val. Loss: 0.061 |  Val. Acc: 98.20%
Test Loss: 0.056 | Test Acc: 98.32%
```

Dodanie czwartej warstwy (na pierwszej pozycji) z 1000 neuronów nie poprawiło uczenia sieci w stosunku do modelu trójwarstwowego. Poziom dokładności mniej więcej taki sam. Natomiast znacznie wzrósł czas uczenia na epokę – o prawie 20s. W tym przypadku jest około 1,35 mln konfigurowalnych parametrów licząc wszystkie wagi, aktywacje i bias.

Aby zoptymalizować kod można zrównoleglić działania przy ewaluacji ponieważ poszczególne wagi nie są między sobą powiązane i nie trzeba wyznaczać gradientów. Niestety przy trenowaniu modelu nie można zrównoleglić obliczeń, ponieważ wyznaczane wagi w poszczególnych iteracjach są od siebie zależne.

Zadanie II

```
79  def evaluate(model, iterator, criterion, device):
80    epoch_loss = 0
81    epoch_acc = 0
82    model.eval()
83    with torch.no_grad():
84      for (x, y) in iterator:
85        x = x.to(device)
86        y = y.to(device)
```

with torch.no_grad() jest sposobem na wyłączenie części elementów z obliczeń gradientu w backward_propagation. W ten sposób pośrednie wyniki gradientów nie będą trzymane w pamięci, ponieważ dzięki tej funkcji deklarowane jest, że gradienty nie są użyte.

```
images = torch.cat(images, dim = 0)
labels = torch.cat(labels, dim = 0)
probs = torch.cat(probs, dim = 0)
return images, labels, probs
```

torch.cat() od concatenate, czyli konkatenacja inaczej łączenie poprzedniego z następnym. Łączy tensory tego samego typu w jeden tensor o zadanym wymiarze. W tym przypadki wszystkie pośrednio pozyskane tensory obrazków, ich wyników i próbek łączone są w pojedyncze jednowymiarowe tensory.

```
true_prob = probs[true_label]
incorrect_prob, incorrect_label = torch.max(probs, dim = 0)
ax.imshow(image.view(28, 28).cpu().numpy(), cmap='bone')
```

torch.max() zwraca maksymalną wartość ze wszystkich elementów zadanego tensora. W tym przypadku z tensora probs zwróci największą wartość, którą można uznać za błąd oraz przyporządkowaną do niej faktyczną wartość.

```
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
```

Ustawienie ziarna do generowania pseudolosowych liczb aby wyniki były odtwarzalne, tzn dla danego ziarna zawsze losowało ten sam output. Generatory są dwa zarówno dla CUDA jak i CPU.

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

Wybór urządzenia do przeprowadzenia obliczeń. Jeżeli sprzęt wspiera CUDA to zostanie to wybrane i obliczenia będą mogły zostać przeprowadzone z użyciem kart graficznych. Jeżeli nie ma takiej możliwości to standardowo obliczaniem zajmie się procesor (CPU).

```
model.load_state_dict(torch.load('nn_model.pt'))
```

torch.load() wczytuje zapisany wytrenowany model (czyli wytrenowane parametry dla których model optymalizuje najlepiej) do przeprowadzenia dalszych analiz.

```
images, labels, probs = get_predictions(model, test_iterator, device)
pred_labels = torch.argmax(probs, 1)

plot_confusion_matrix(labels, pred_labels)
corrects = torch.eq(labels, pred_labels)
incorrect_examples = []
```

torch.argmax() - zwraca tensor ineksów maksymalnych wartości tensora wejściowego.

torch.eq() - porównuje czy tensory zawierają te same wartości. Zwraca tensor z wartościami True lub False dla każdych dwóch porównywanych tensorów wewnątrz tensorów wejściowych. W tym przypadku sprawdza czy labels mają takie same wartości jak pred_labels i uzupełnia wartościami True i False tensor corrects.

```
incorrect_examples.sort(reverse = True, key = lambda x: torch.max(x[2], dim = 0).values)
plot_most_incorrect(incorrect_examples, N_IMAGES)
# Dlaczego te obrazki mogły zostać nieprawidłowo sklasyfikowane?
```

torch.max() zwraca maksymalną wartość ze wszystkich elementów zadanego tensora. Key jest parametrem określającym funkcję, która ma zostać na każdym elemencie przy porównaniu elementów w celu ich sortowania. Ostatecznie incorrect_examples będzie tensorem posiadającym najbardziej niepoprawne przykłady na pierwszych indeksach.

```
def forward(self, x):
  batch_size = x.shape[0]
  x = x.view(batch_size, -1)
  h_1 = F.relu(self.input_fc(x))
  h_2 = F.relu(self.hidden_fc(h_1))
  y_pred = self.output_fc(h_2)
  return y_pred, h_2
```

```
y_prob = F.softmax(y_pred, dim = -1)
```

W torch.nn.functional znajdują się funkcje aktywacji. W kodzie zostały użyte ReLu na 2 ukrytych warstwach oraz softmax na warstwie wychodzącej.

```
optimizer = optim.Adam(model.parameters())
```

torch.optim.Adam() jest jednym z algorytmów optymalizacyjnych z paczki torch.optim. Konstruowany jest obiekt optimizer, który przechowuje obecny stan parametrów i na podstawie wyliczonych gradientów zaktualizuje parametry (wagi i bias). Adam jest algorytmem do optymalizacji gradientów pierwszego rzędu stochastycznych funkcji celu, opartym na adaptacyjnych oszacowaniach momentów niższego rzędu.

```
criterion = nn.CrossEntropyLoss()
```

torch.nn.CrossEntropyLoss() określa funkcję liczenia kosztu. Jest kombinacją funkcj nn.LogSoftmax() i nn.NLLLoss(). Funkcji tych używa się do określenia strat i kosztów w celu wytrenowania odpowiedniej klasyfikacji.

Zadanie III

Co dokładnie wykonują transformacje?

```
# Co dokładnie wykonują transformacje?
train_transforms = transforms.Compose([
  transforms.RandomRotation(5, fill=(0,)),
  transforms.RandomCrop(28, padding = 2),
  transforms.ToTensor(),
  transforms.Normalize(mean = [mean], std = [std])
])

test_transforms = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize(mean = [mean], std = [std])
])
```

Transformacje dostosowują dane wejściowe odpowiednio dla modelu. W wewnątrz listy w transforms.Compose() można wpisać wszystkie transformacje, które chcemy wykonać. W przypadku użycia biblioteki PyTorch w należy przede wszystkim transformować tablice danych na tensory co robi funkcja ToTensor(). RandomRotation(5) obraca obrazy o losową wartość od -5 do 5 stopni. RandomCrop(28, padding=2) przycina obraz w losowej lokacji tak aby wyjściowy rozmiar wynosił 28x28. Padding (wolne miejsce) dodawany jest na każdą stronę po 2. normalize(mean, std) normalizuje obraz w tensorze według podanej średniej i odchylenia standardowego.

Definicja DataLoader'ów (czym są DataLoader'y)?

Co oznacza i do czego jest używany parametr shuffle = True w zbiorze treningowym?

```
# Definicja DataLoader'ów (czym są DataLoader'y)?
# Co oznacza i do czego jest używany parametr shuffle = True w zbiorze treningowym?
BATCH_SIZE = 64
train_iterator = data.DataLoader(train_data,
                                 shuffle = True,
                                 batch_size = BATCH_SIZE)

valid_iterator = data.DataLoader(valid_data,
                                 batch_size = BATCH_SIZE)

test_iterator = data.DataLoader(test_data,
                                batch_size = BATCH_SIZE)
```

DataLoadery są to obiekty służące do iterowania po zbiorach wejściowych wspierające różne przydatne funkcje. Batch_size oznacza po ile danych jest wczytywane w jednej iteracji ponieważ wczytywanie olbrzymiej ilości danych naraz mogłoby spowolnić sprzęt lub wręcz zapchać proces. Shuffle oznacza mieszanie danych treningowych. Dane treningowe z MNIST (obrazki z liczbami) dostarczane są posortowane tzn. na początku są wszystkie zera, potem jedynki, potem dwójki itd. W celu lepszej nauki i wyeliminowaniu optymalizacji kosztu w kierunku danej cyfry (tak może się dziać gdy sieć będzie trenowana na początku tylko na jednej cyfrze) dane warto przemieszczać aby rozkład występowania cyfr był losowy.

Jak zmiana ma wpływ na wyniki?

Czym jest nn.Linear?

```
"""
Defaultowe wartości:
 - 1 warstwa ukryta - 100 neuronów
 - 2 warstwa ukryta - 50 neuronów
 - funkcje aktywacji - ReLU

 Jak zmiana ma wpływ na wyniki?
 Czym jest nn.Linear?
"""

class MLP(nn.Module):
  def __init__(self, input_dim, output_dim, inp, out, out2, out3):
    super().__init__()
    self.input_fc = nn.Linear(input_dim, inp)
    self.hidden_fc = nn.Linear(inp, out)
    self.hidden_fc1 = nn.Linear(out, out2)
    self.hidden_fc2 = nn.Linear(out2, out3)
    self.output_fc = nn.Linear(out3, output_dim)
```

Jak zmiany neuronów i warstw wpływają na wyniki opisano w zad1. nn.Linear(x_size, y_size) jest funkcją realizującą transformację liniową do przychodzących danych: y = x * W ^ T + b. W parametrach przyjmuje wielkości każdej próbki wejścia i wyjścia czyli w tym przypadku ilość neuronów na warstwie.

Jak działa obliczanie dokładności modelu?

```
"""
Jak działa obliczanie dokładności modelu?
"""
def calculate_accuracy(y_pred, y):
  top_pred = y_pred.argmax(1, keepdim = True)
  correct = top_pred.eq(y.view_as(top_pred)).sum()
  acc = correct.float() / y.shape[0]
  return acc
```

Funkcja przyjmuje tensory wyliczonych wyjść dla danych walidacyjnych i dostarczonych poprawnych wyjść walidacyjnych, określonych z góry. Dzięki temu

każdy pojedynczy wynik można porównać czy został zaklasyfikowany dobrze. Dokładność to stosunek ilości policzonych poprawnie wyników do wszystkich wyników.

Opisać algorytm uczący - co można poprawić i jak?

```python
def train(model, iterator, optimizer, criterion, device):
  epoch_loss = 0
  epoch_acc = 0

  # Niektóre warstwy inaczej zachowują się podczas treningu a inaczej podczas ewaluacji.
  # Dobrą praktyką jest zaznaczenie, że sieć jest w "trybie uczenia"
  model.train()
  for (x, y) in iterator:
    x = x.to(device)
    y = y.to(device)
    optimizer.zero_grad()
    y_pred, _ = model(x)
    loss = criterion(y_pred, y)
    acc = calculate_accuracy(y_pred, y)
    loss.backward()
    optimizer.step()
    epoch_loss += loss.item()
    epoch_acc += acc.item()
  return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Trenowanie modelu polega na powtarzaniu pewnych kroków w pętli przez zadaną liczbę iteracji, które mają na celu zmianę parametrów (wag i biasu) na takie które zminimalizują koszt uczenia aby dokładność klasyfikacji była jak największa.

optimizer.zero_grad() - oznacza wyzerowanie wartości gradientów. Co wykonanie pętli podczas następnych funkcji gradienty są wyliczane (loss.backward() i optimizer.step()). Jeśli nie wyczyścilibyśmy gradientów po optimizer.step(), co jest krokiem aproksymacyjnym przed propagacją wsteczną, gradienty nagromadzały by się.

y_pred, _ = model(x) – wyliczenie tensora wartości przewidywanych przez model na podstawie tensora danych treningowych czyli propagacja do przodu.

loss = criterion(y_pred, y) – obliczanie funkcji kosztu na podstawie porównania wartości wyliczonych przez model z poprawnymi wartościami treningowymi.

acc = calculate_accuracy(y_pred, y) – wyliczenie dokładności predykcji sieci

loss.backward() - propagowanie do tyłu. Wyliczenie gradientów poprzez obliczenie pochodnych wag, biasu, wartości dla neuronów w poprzednich. Każdy neuron ma wartość wyliczoną z poprzednich w warstwach parametrów stąd do tyłu.

optimizer.step() - Uaktualnienie parametrów (wag i biasu) sieci aby zminimalizować funkcje kosztu.

Aby poprawić trenowanie sieci można zasilić sieć w więcej danych treningowych, dodać liczbę ukrytych warstw i neuronów, zmienić funkcje aktywacji, wyliczania kosztu i optymalizacji oraz zmiana learning rate. Również obróbka danych na początku może mieć wpływ np. jeszcze większa losowa zmiana kąta nachylenia obrazka.

Na czym polega ewaluacja modelu?

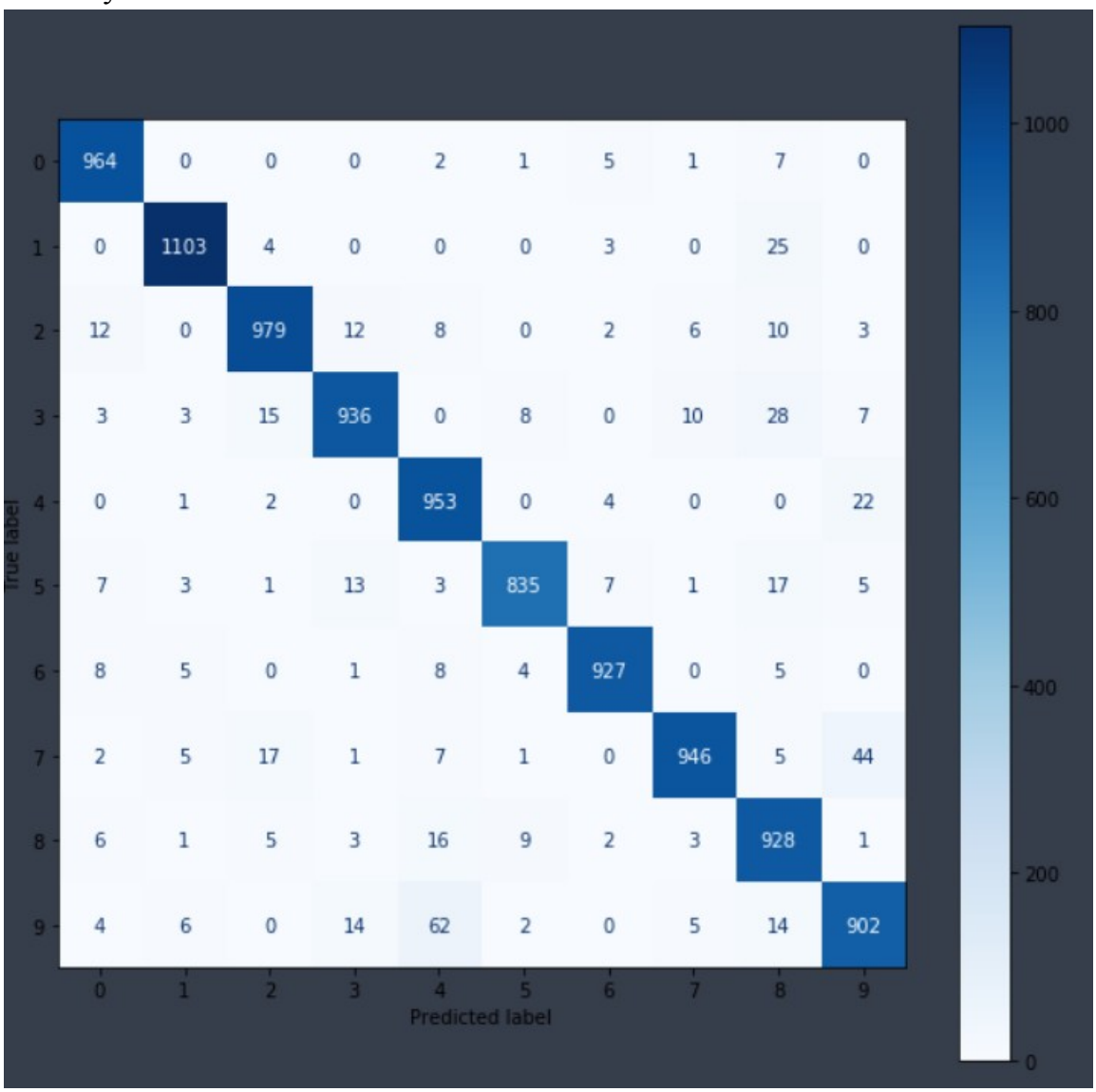
```
"""
Na czym polega ewaluacja modelu?
"""
def evaluate(model, iterator, criterion, device):
  epoch_loss = 0
  epoch_acc = 0
  model.eval()
  with torch.no_grad():
    for (x, y) in iterator:
      x = x.to(device)
      y = y.to(device)
      y_pred, _ = model(x)
      loss = criterion(y_pred, y)
      acc = calculate_accuracy(y_pred, y)
      epoch_loss += loss.item()
      epoch_acc += acc.item()
  return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Ewaluacja wykonuje się szybciej niż algorytm uczący, ponieważ jest wykonywana wewnątrz with torch.no_grad(), w którym nie są liczone żadne gradienty, a parametry nie są uaktualniane. W odróżnieniu od zbioru treningowego jest zwracany koszt i dokładność dla danych walidacyjnych a nie treningowych.

Czym jest "Confusion matrix"? Opisać dla wyników modelu.

```
"""
 Czym jest "Confusion matrix"? Opisać dla wyników modelu.
"""
def plot_confusion_matrix(labels, pred_labels):
  fig = plt.figure(figsize = (10, 10))
  ax = fig.add_subplot(1, 1, 1)
  cm = metrics.confusion_matrix(labels, pred_labels)
  cm = metrics.ConfusionMatrixDisplay(cm, range(10))
  cm.plot(values_format = 'd', cmap = 'Blues', ax = ax)
  fig.show()
```

Wynik Confusion Matrix:



Macierz ta pokazuje jak wyliczone wartości obrazów mają się do faktycznych wyników. Na przekątnej widać poprawne dopasowania. Natomiast widać też błędy w rozpoznawaniu obrazów takie jak: 4 pomylona z 9 62 razy czy 9 z 7 44 razy.

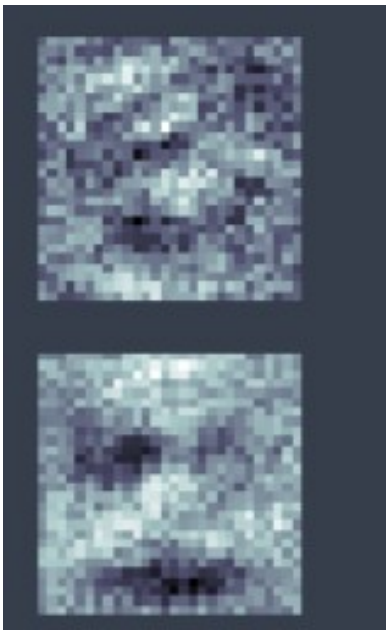
Zobrazowanie wag pierwszej warstwy modelu.

Może jakiś neuron nauczył się rozpoznawać konkretny wzór?

Podać przykład.

```
"""
 Zobrazowanie wag pierwszej warstwy modelu.
 Może jakiś neuron nauczył się rozpoznawać konkretny wzór?
 Podać przykład.
"""
def plot_weights(weights, n_weights):
  rows = int(np.sqrt(n_weights))
  cols = int(np.sqrt(n_weights))
  fig = plt.figure(figsize = (20, 10))
  for i in range(rows*cols):
    ax = fig.add_subplot(rows, cols, i+1)
    ax.imshow(weights[i].view(28, 28).cpu().numpy(), cmap = 'bone')
    ax.axis('off')
  fig.show()
```

Z wyniku funkcji widać, że neurony nauczyły się rozpoznawać konkretne wzory. Wiele z nich jest ciężkich do rozpoznania widać jednak, że są jakieś prawidłowości. Przykładowo poniżej obrazy przypominające ósemkę i dwójkę:



Jaki model został zapisany w pliku 'nn_model'?

```
# Jaki model został zapisany w pliku 'nn_model'?
model.load_state_dict(torch.load('nn_model.pt'))
```

Do pliku został zapisany wytrenowany model, który najlepiej działał na zbiorze walidacyjnym.

Dlaczego te obrazki mogły zostać nieprawidłowo sklasyfikowane?

```
incorrect_examples.sort(reverse = True, key = lambda x: torch.max(x[2], dim = 0).values)
plot_most_incorrect(incorrect_examples, N_IMAGES)
# Dlaczego te obrazki mogły zostać nieprawidłowo sklasyfikowane?
```

Obrazki wyglądają tak:

Możliwe przyczyny nieprawidłowego zaklasyfikowania obrazków to:

- Dobór nieodpowiednich parametrów modelu.
- Obrazki różnią się bardzo od większości danych treningowych np. ósemki w górnym rzędzie są inne niż większość ósemek treningowych dla których sieć minimalizowała koszt. Wpływa na to również kąt nachylenia co widać w pierwszej dziewiątce drugiego rzędu jak i dodatkowe znaki jak kropka w dziewiątce w pierwszym rzędzie.

- Narysowana na obrazku cyfra mogła być bardzo podobna do innej przez co klasyfikator przyporządkował ją do takiego wyniku.

Wnioski

Cel zadania został osiągnięty. Wszystkie polecenia udało się wykonać. Kod programu dokładnie przeanalizowano i zapoznano się z biblioteką PyTorch oraz ogólnym sposobem tworzenia w tym frameworku sieci neuronowych złożonych z wielu warstw.