МІНІСТЕРСТВО ОСВІТИ I НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ   
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ БІОМЕДИЧНОЇ ІНЖЕНЕРІЇ

КАФЕДРА БІОМЕДИЧНОЇ КІБЕРНЕТИКИ

**Комп’ютерний практикум №7**

з дисципліни «**Нейронні мережі**»

на тему: «**РОЗРОБКА СІАМСЬКОЇ МЕРЕЖІ (SIAMESE**

# NETWORKS)»

**Виконав:**

студент гр. БС-03

Затуловський Г. А.

**Перевірив:**

ас. каф. БМК Дюмін О.Д.

Зараховано від \_\_\_.\_\_\_.\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис викладача)

Київ-2023

# Варіант 6

# Практична частина

1. Створити сіамську мережу для датасету зображень відповідно до варіанту завдань.
2. Дослідити вплив регуляризації та нормалізації на якість сіамської мережі.
3. Спробувати іншу архітектуру базової мережі.
4. Реалізувати використання triplet loss замість contrastive loss.
5. Аналіз результатів: Візуалізувати криві навчання та втрат, розрахувати метрики, проаналізувати результати, надати рекомендації щодо можливих покращень моделі.
6. Вимоги до звіту: опис використовуваного датасету, його особливостей та завдань, візуалізація попередньої обробки даних, детальний опис структури розробленої моделі, графіки навчання та втрат, висновки та рекомендації.

**Варіанти завдань:**

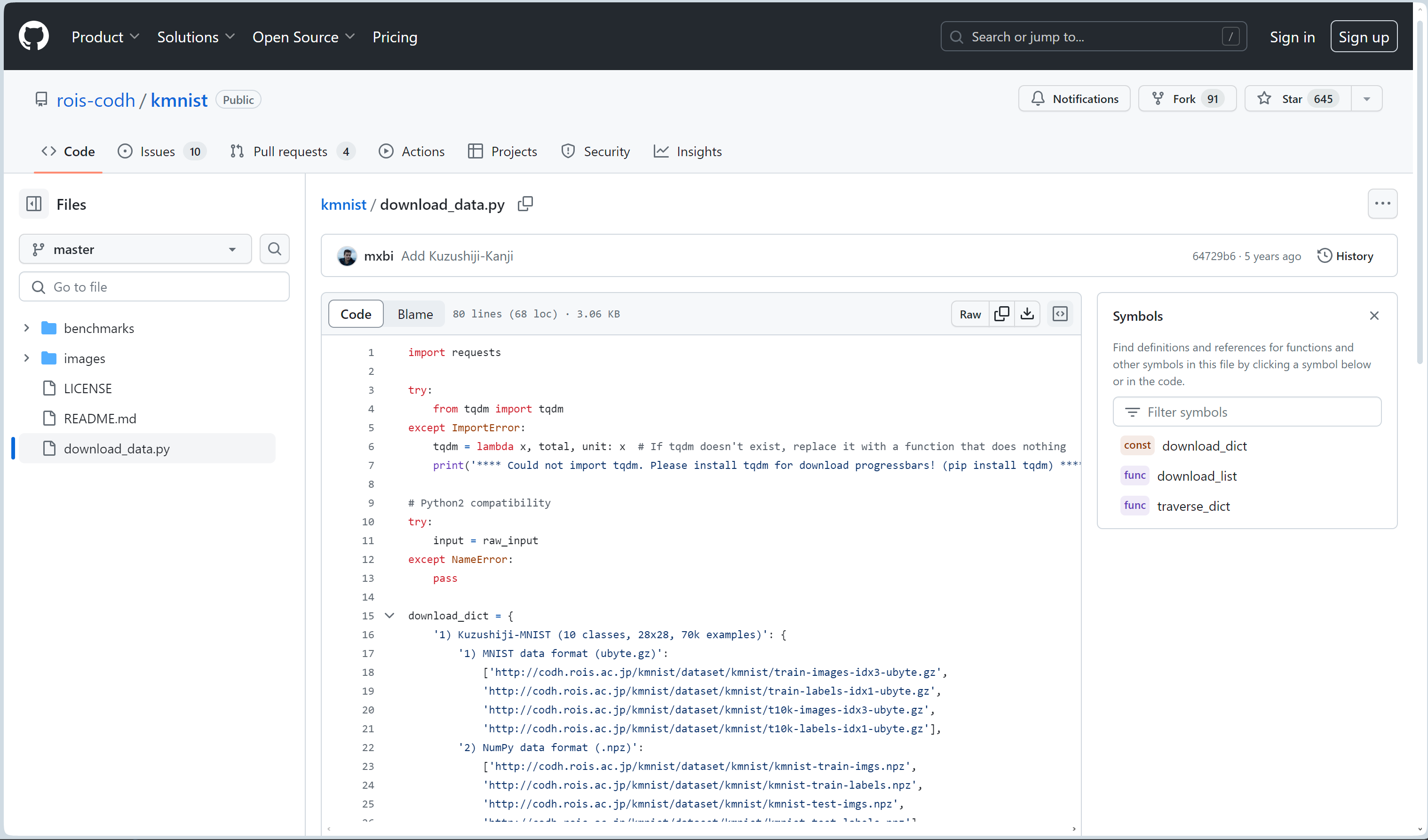
[3, **6**, 9, 12, 15, 18, 21, 24] KMNIST (Kuzushiji-MNIST):

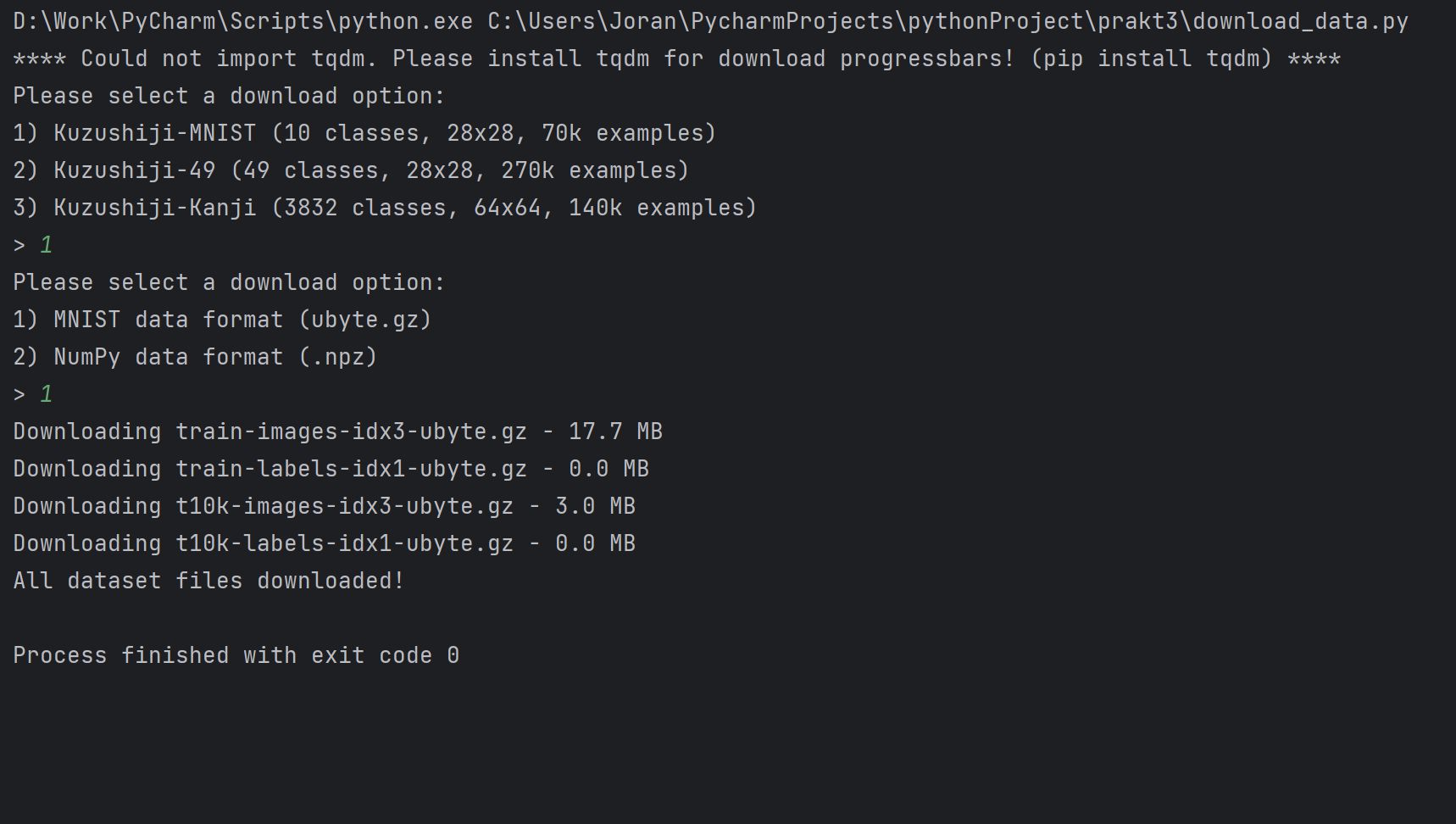
Датасет історичних японських рукописних символів.

70,000 зображень у 10 категоріях.

Зображення мають розмір 28x28 пікселів, монохромні.

Використовуйте це посилання та відповідний репозиторій (посилання на код додане) для завантаження даних: [http://codh.rois.ac.jp/kmnist/,](http://codh.rois.ac.jp/kmnist/) [https://github.com/rois-codh/kmnist/blob/master/download\_data.py,](https://github.com/rois-codh/kmnist/blob/master/download_data.py) при завантаженні треба обрати необхідний набір даних. Після завантаження можна користуватись даними наступним чином попередньо встановивши idx2numpy:

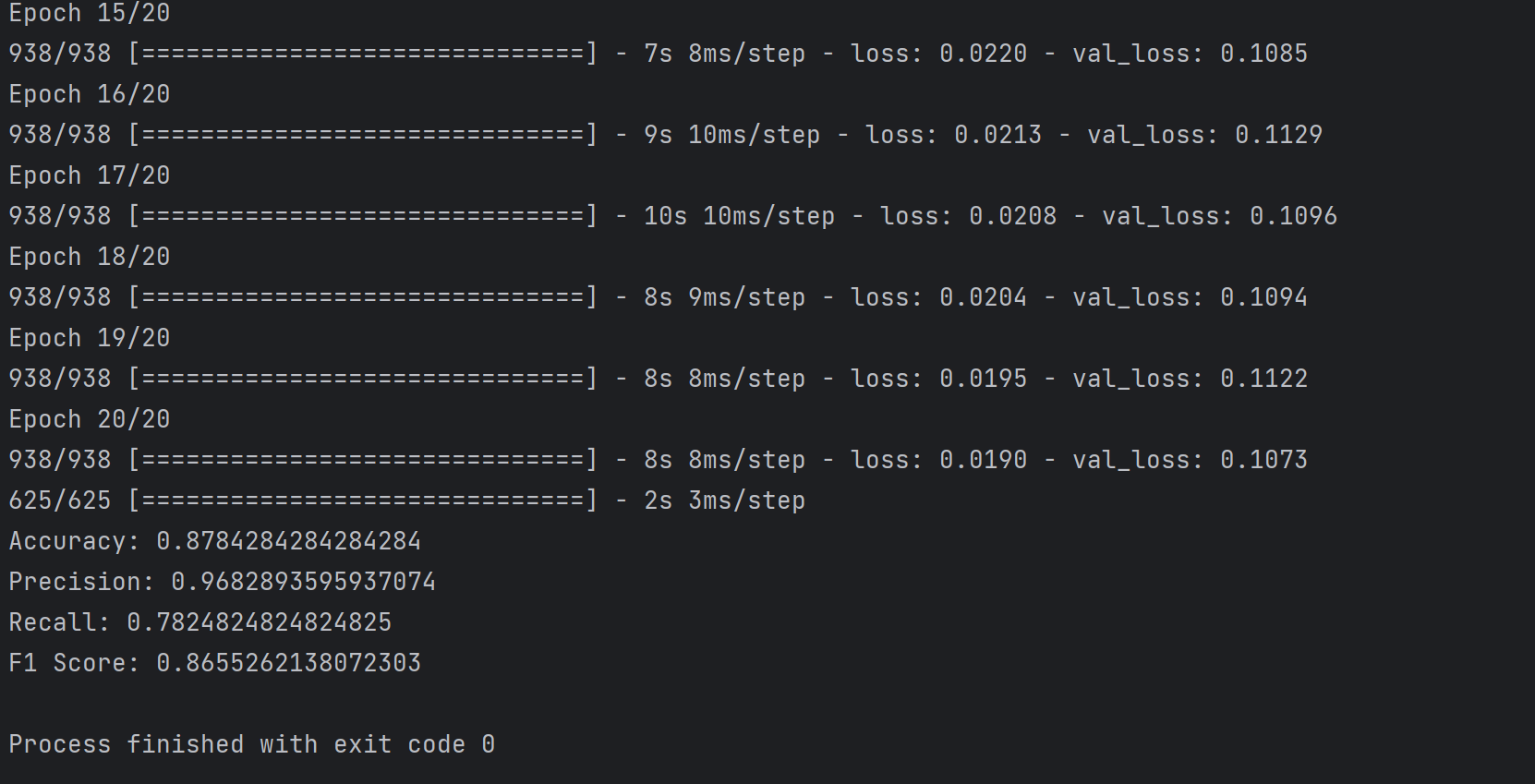


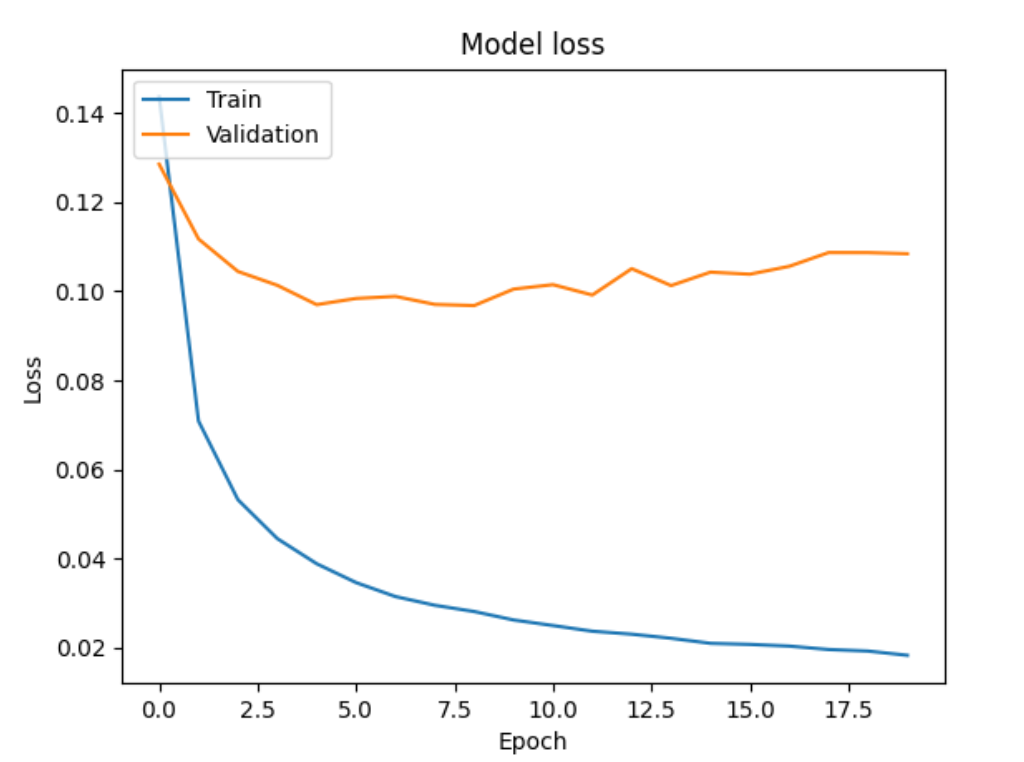


Код програми - contrastive loss:

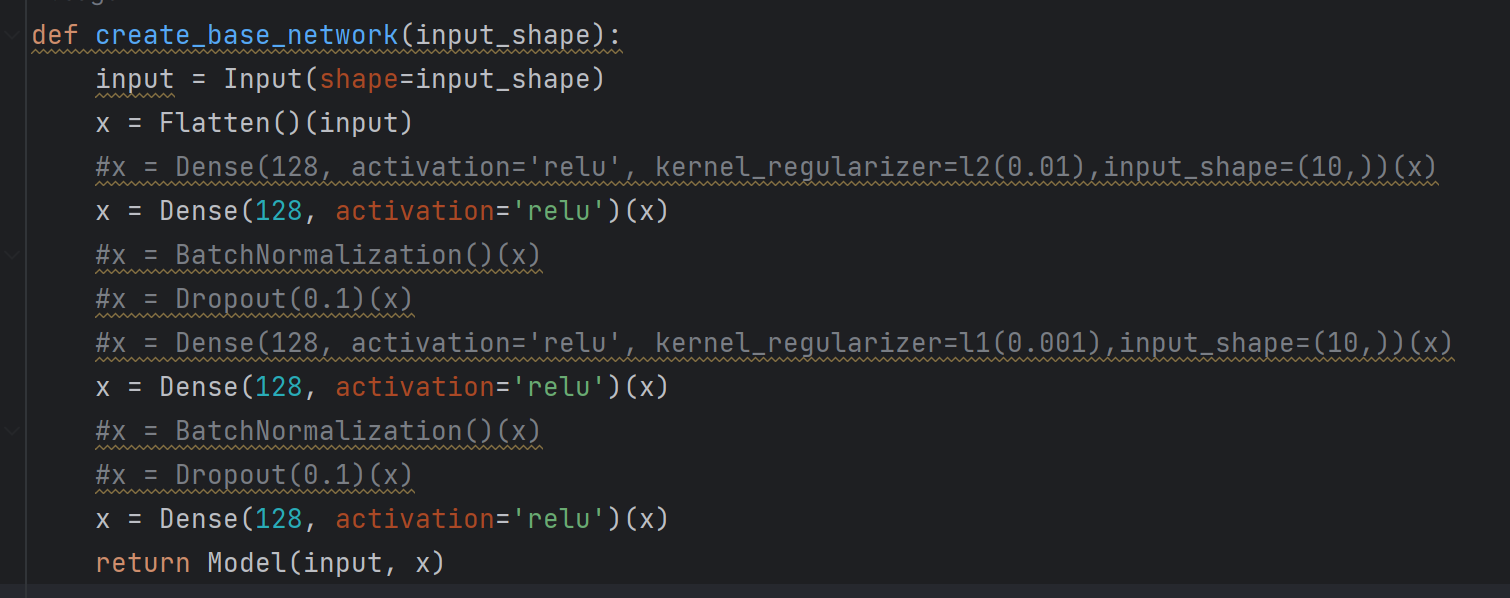
import gzip  
import idx2numpy  
import numpy as np  
import tensorflow as tf  
from tensorflow import keras  
from tensorflow.keras.datasets import mnist  
from tensorflow.keras.models import Model  
from tensorflow.keras.layers import Input, Flatten, Dense, Dropout, Lambda  
from tensorflow.keras.optimizers import RMSprop  
from tensorflow.keras import backend as K  
import matplotlib.pyplot as plt  
from sklearn.metrics import accuracy\_score, f1\_score, recall\_score, precision\_score  
  
  
def load\_kmnist():  
 with gzip.open('train-images-idx3-ubyte.gz', 'r') as f:  
 x\_train = idx2numpy.convert\_from\_file(f)  
 with gzip.open('train-labels-idx1-ubyte.gz', 'r') as f:  
 y\_train = idx2numpy.convert\_from\_file(f)  
 with gzip.open('t10k-images-idx3-ubyte.gz', 'r') as f:  
 x\_test = idx2numpy.convert\_from\_file(f)  
 with gzip.open('t10k-labels-idx1-ubyte.gz', 'r') as f:  
 y\_test = idx2numpy.convert\_from\_file(f)  
 return (x\_train, y\_train), (x\_test, y\_test)  
  
(x\_train, y\_train), (x\_test, y\_test) = load\_kmnist()  
x\_train = x\_train.astype('float32') / 255.  
x\_test = x\_test.astype('float32') / 255.  
x\_train = np.reshape(x\_train, (len(x\_train), 28, 28, 1))  
x\_test = np.reshape(x\_test, (len(x\_test), 28, 28, 1))  
  
  
def create\_pairs(x, digit\_indices):  
 pairs = []  
 labels = []  
 n = min([len(digit\_indices[d]) for d in range(10)]) - 1  
 for d in range(10):  
 for i in range(n):  
 z1, z2 = digit\_indices[d][i], digit\_indices[d][i + 1]  
 pairs += [[x[z1], x[z2]]]  
 inc = np.random.randint(1, 10)  
 dn = (d + inc) % 10  
 z1, z2 = digit\_indices[d][i], digit\_indices[dn][i]  
 pairs += [[x[z1], x[z2]]]  
 labels += [1, 0]  
 return np.array(pairs), np.array(labels)  
  
  
digit\_indices = [np.where(y\_train == i)[0] for i in range(10)]  
tr\_pairs, tr\_y = create\_pairs(x\_train, digit\_indices)  
  
digit\_indices = [np.where(y\_test == i)[0] for i in range(10)]  
te\_pairs, te\_y = create\_pairs(x\_test, digit\_indices)  
  
  
  
def create\_base\_network(input\_shape):  
 input = Input(shape=input\_shape)  
 x = Flatten()(input)  
 x = Dense(128, activation='relu')(x)  
 x = Dropout(0.1)(x)  
 x = Dense(128, activation='relu')(x)  
 x = Dropout(0.1)(x)  
 x = Dense(128, activation='relu')(x)  
 return Model(input, x)  
  
base\_network = create\_base\_network([28, 28, 1])  
input\_a = Input(shape=(28, 28, 1))  
input\_b = Input(shape=(28, 28, 1))  
  
processed\_a = base\_network(input\_a)  
processed\_b = base\_network(input\_b)  
  
  
def euclidean\_distance(vects):  
 x, y = vects  
 sum\_square = K.sum(K.square(x - y), axis=1, keepdims=True)  
 return K.sqrt(K.maximum(sum\_square, K.epsilon()))  
  
  
distance = Lambda(euclidean\_distance)([processed\_a, processed\_b])  
model = Model([input\_a, input\_b], distance)  
  
  
def contrastive\_loss(y\_true, y\_pred):  
 margin = 1  
 y\_true = K.cast(y\_true, 'float32')  
 sqaure\_pred = K.square(y\_pred)  
 margin\_square = K.square(K.maximum(margin - y\_pred, 0))  
 return K.mean(y\_true \* sqaure\_pred + (1 - y\_true) \* margin\_square)  
  
model.compile(loss=contrastive\_loss, optimizer=RMSprop())  
history = model.fit([tr\_pairs[:, 0], tr\_pairs[:, 1]], tr\_y, epochs=20, batch\_size=128,validation\_data=([te\_pairs[:, 0], te\_pairs[:, 1]], te\_y))  
predicted = model.predict([te\_pairs[:, 0], te\_pairs[:, 1]])  
  
# 'history' - це історія, яка повертається після навчання моделі  
plt.plot(history.history['loss'])  
plt.plot(history.history['val\_loss'])  
plt.title('Model loss')  
plt.ylabel('Loss')  
plt.xlabel('Epoch')  
plt.legend(['Train', 'Validation'], loc='upper left')  
plt.show()  
threshold = 0.5  
binary\_predictions = [1 if dist < threshold else 0 for dist in predicted]  
accuracy = accuracy\_score(te\_y, binary\_predictions)  
precision = precision\_score(te\_y, binary\_predictions)  
recall = recall\_score(te\_y, binary\_predictions)  
f1 = f1\_score(te\_y, binary\_predictions)  
print(f"Accuracy: {accuracy}")  
print(f"Precision: {precision}")  
print(f"Recall: {recall}")  
print(f"F1 Score: {f1}")

Результати

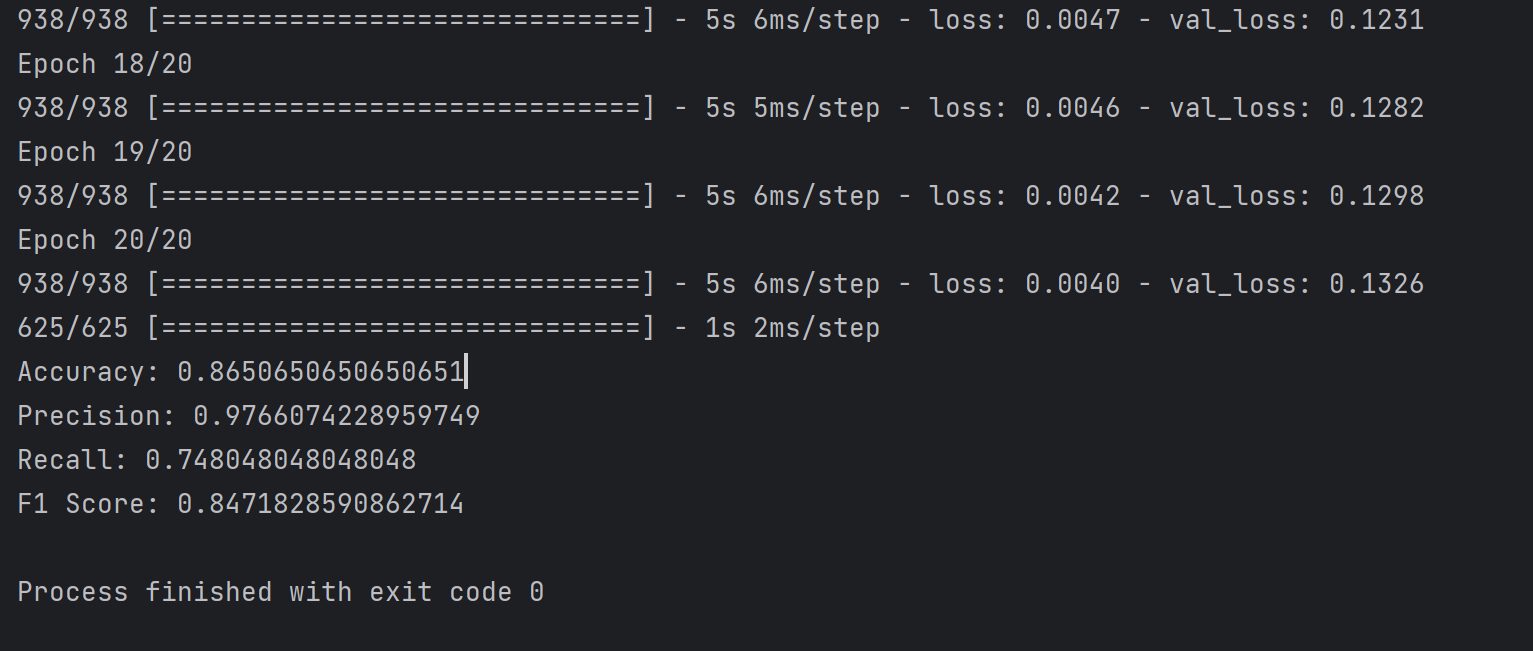


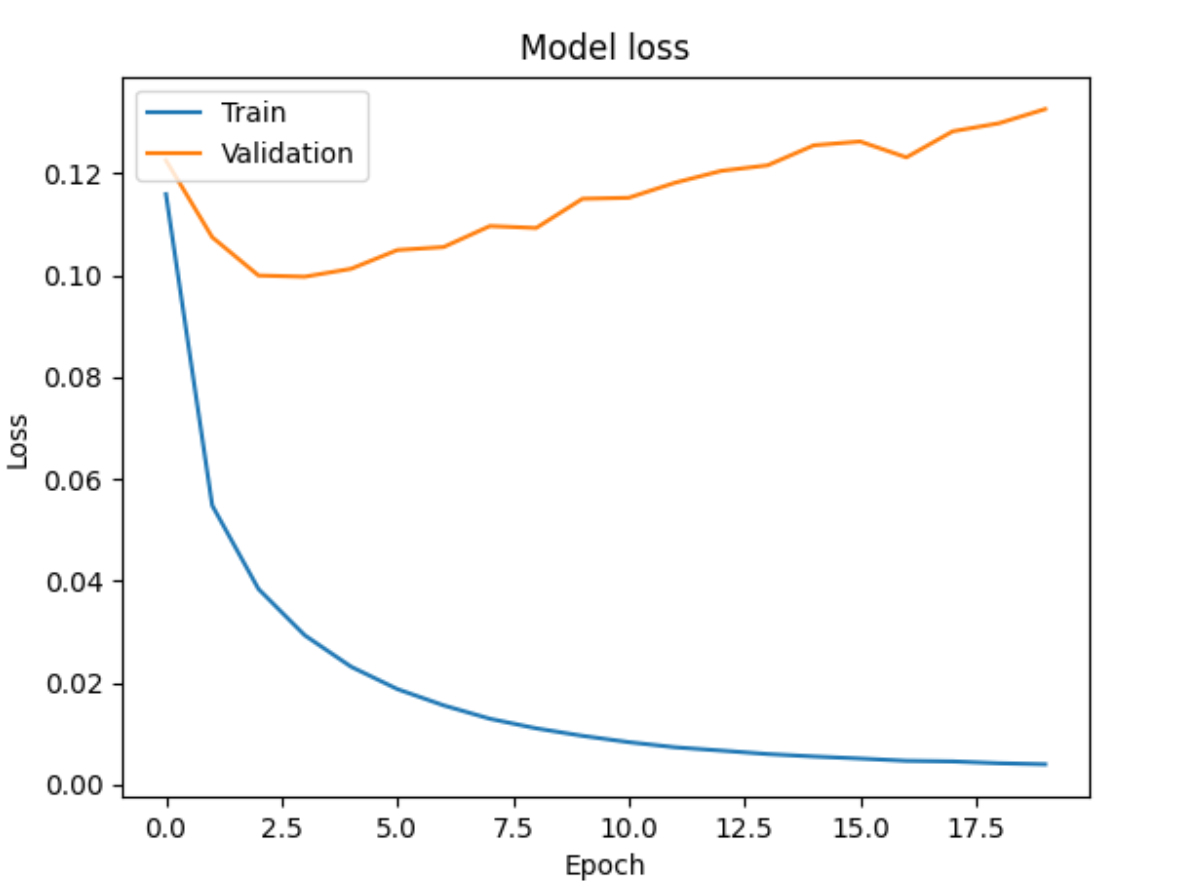


Приберемо регуляризацію dropout (без регулізації, без нормалізації)

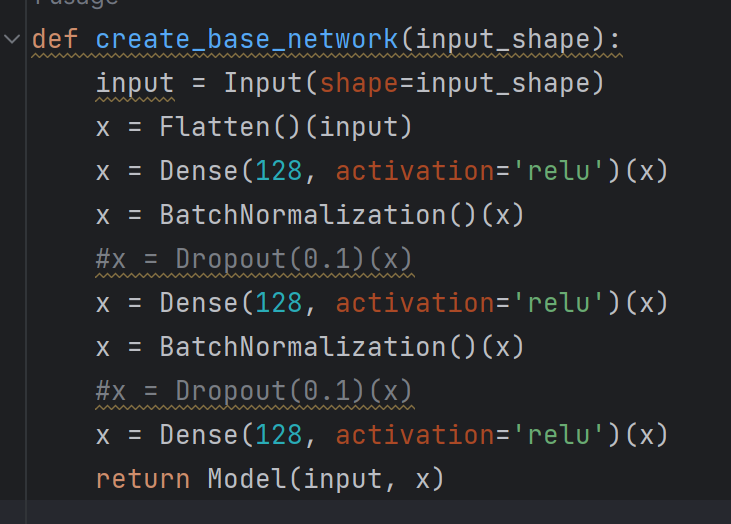


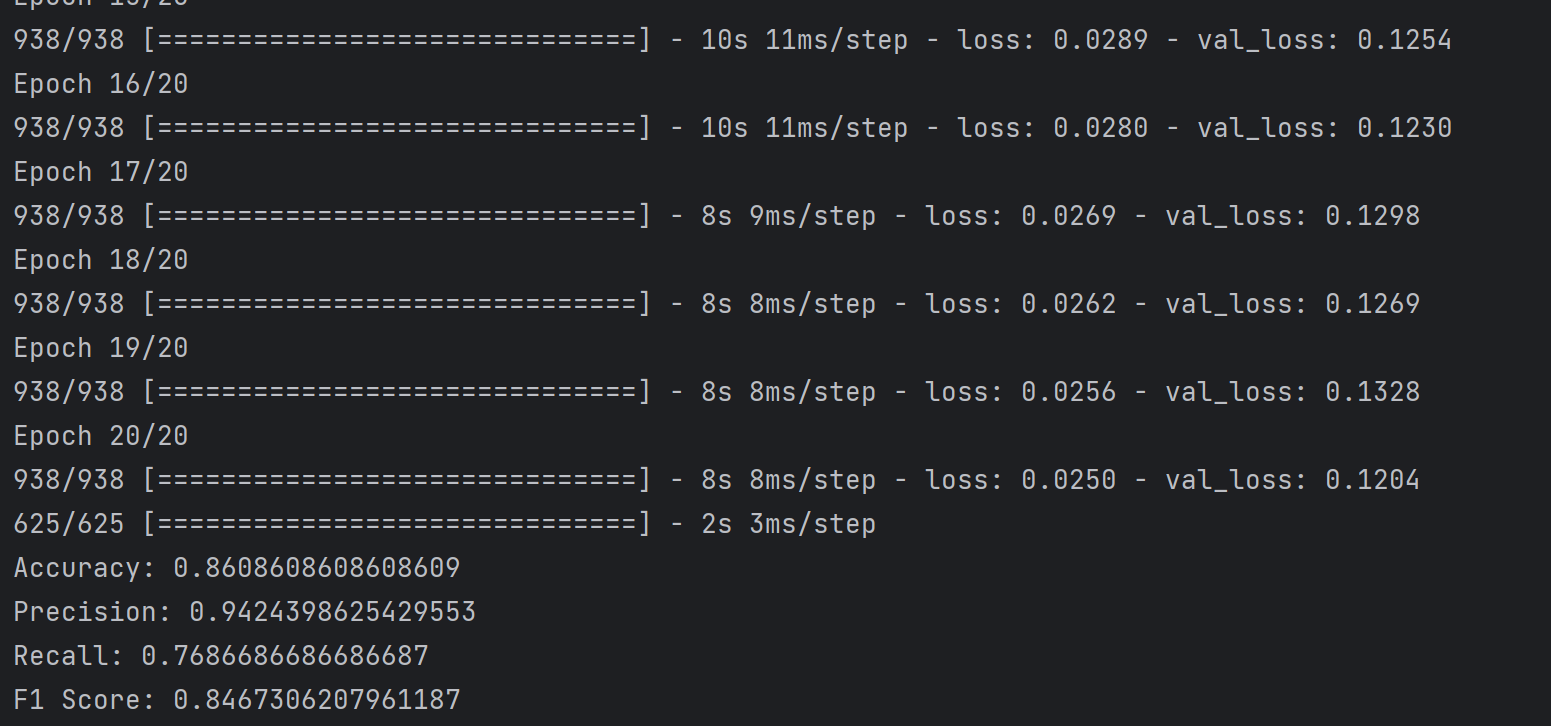
Результати

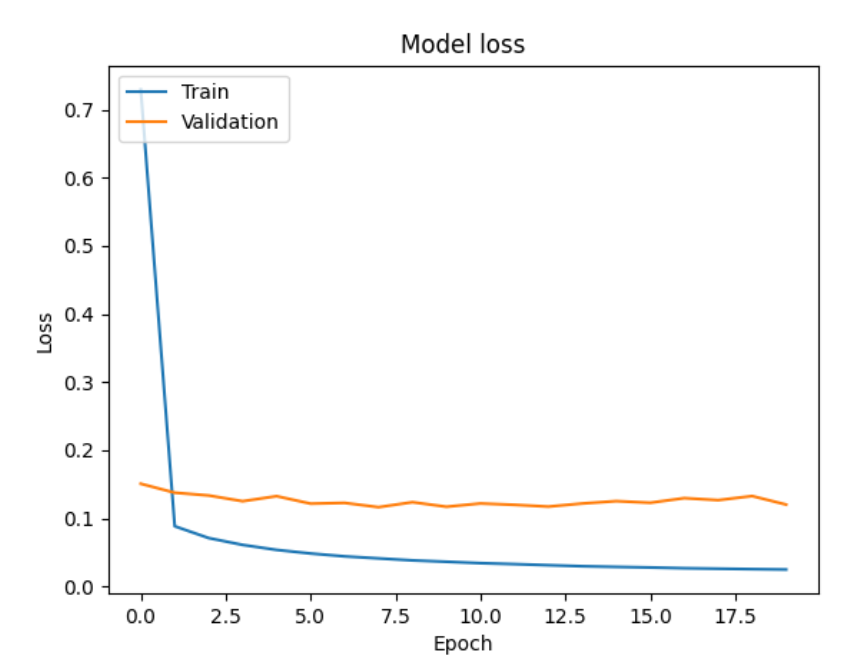




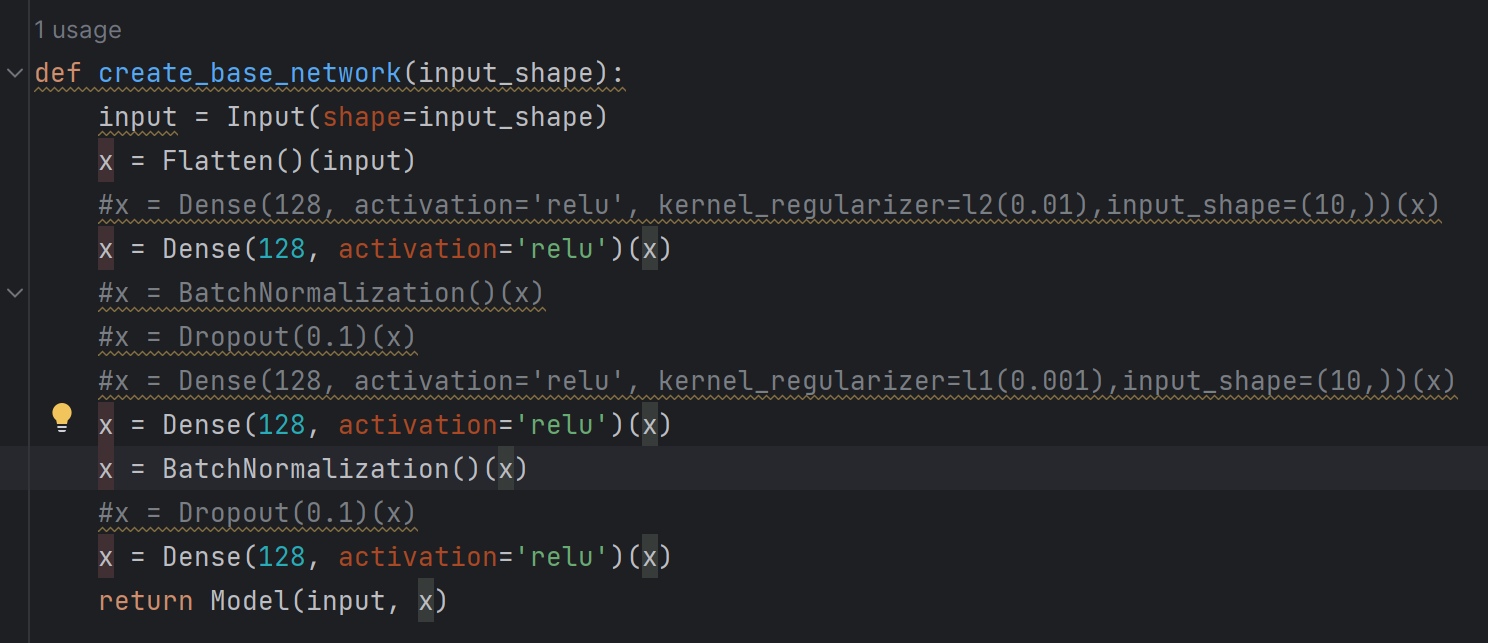
Додамо нормалізацію



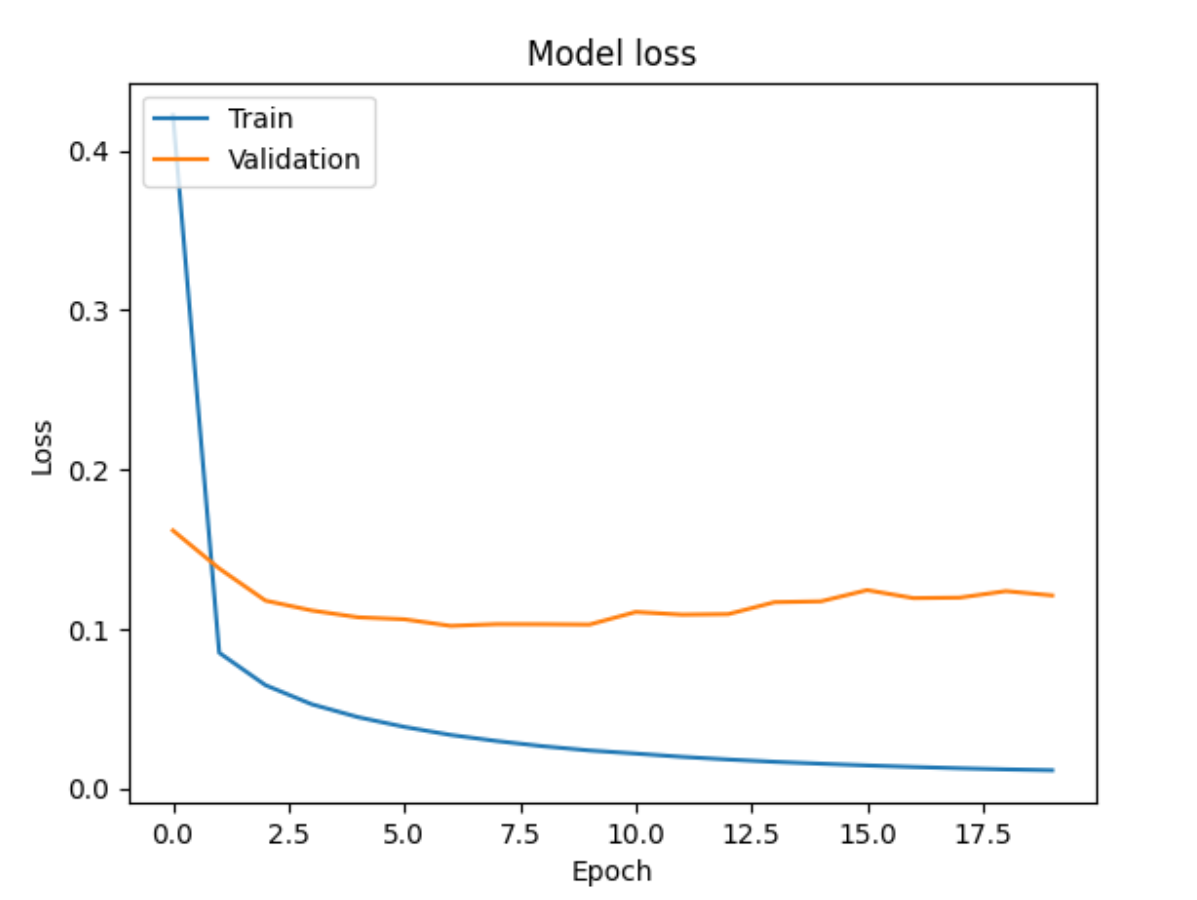
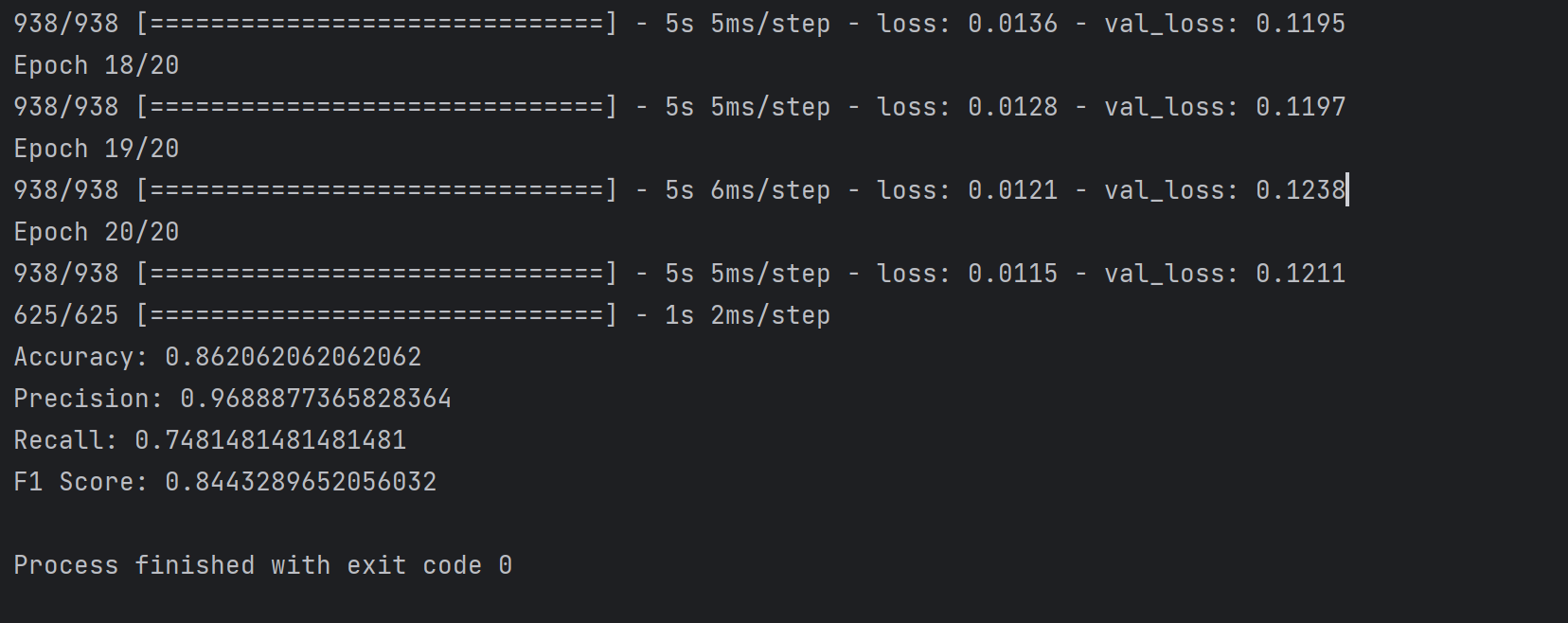
Результати



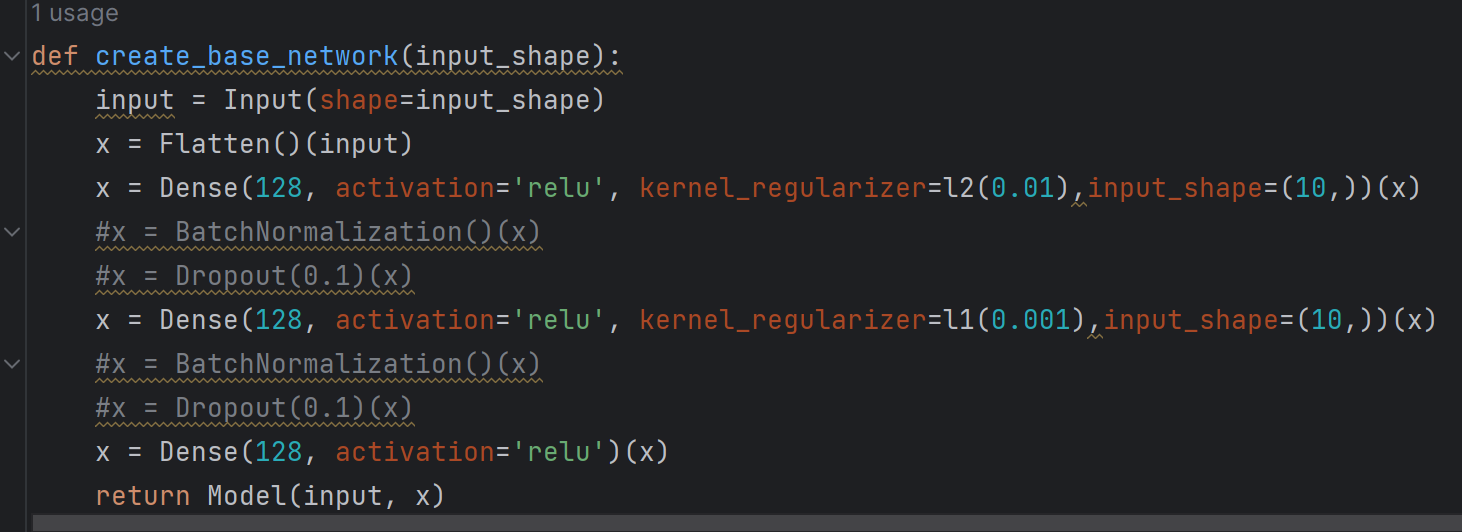
Приберемо одну нормалізацію



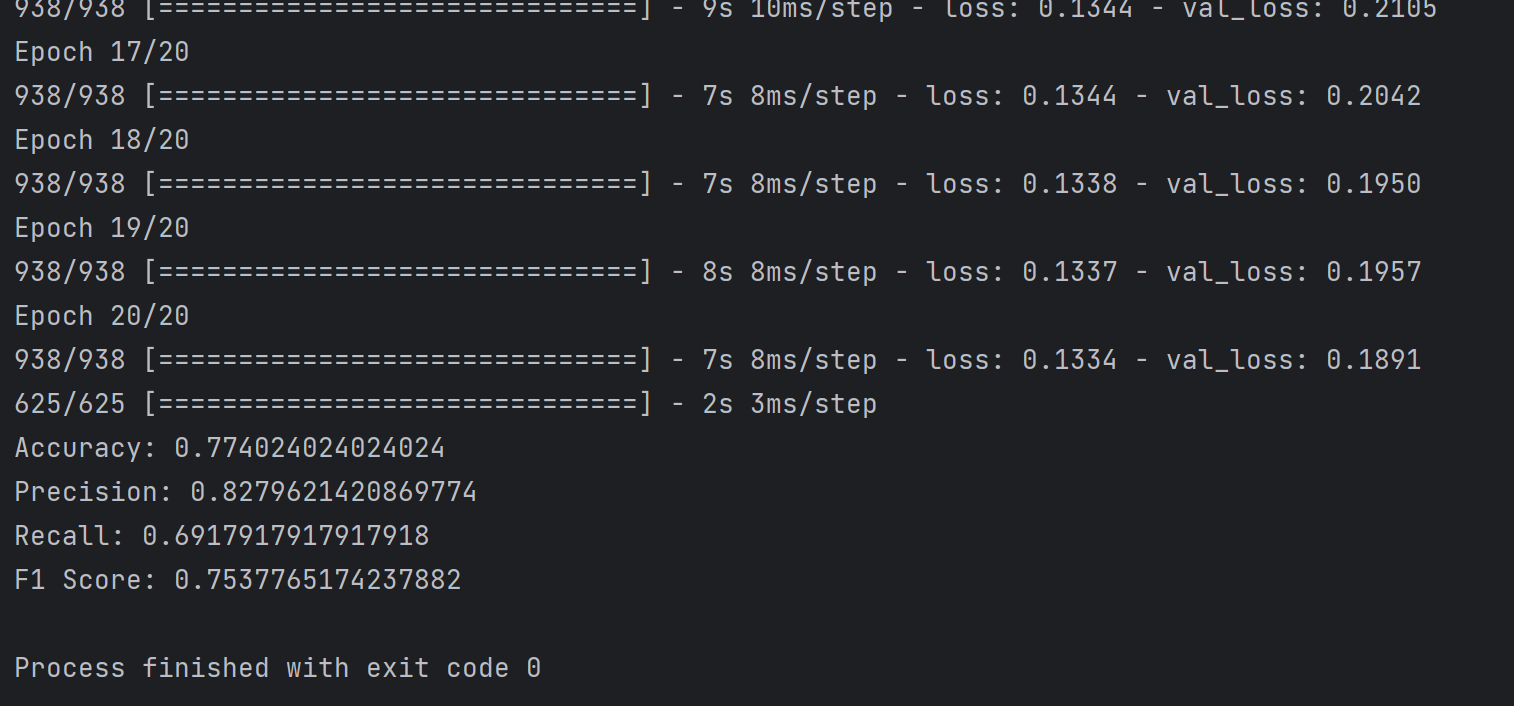
Результати

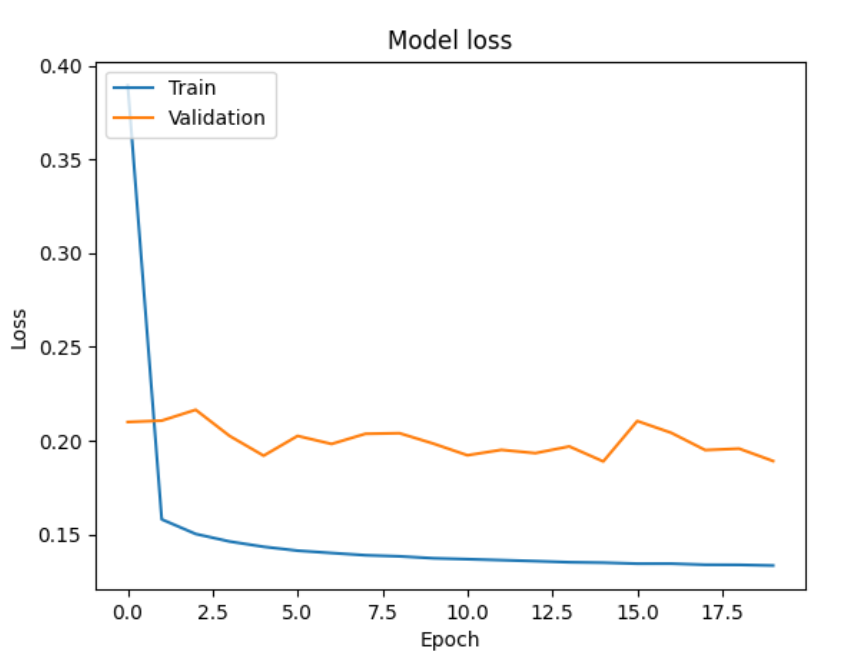
 

Додамо регуляризацію L1, L2

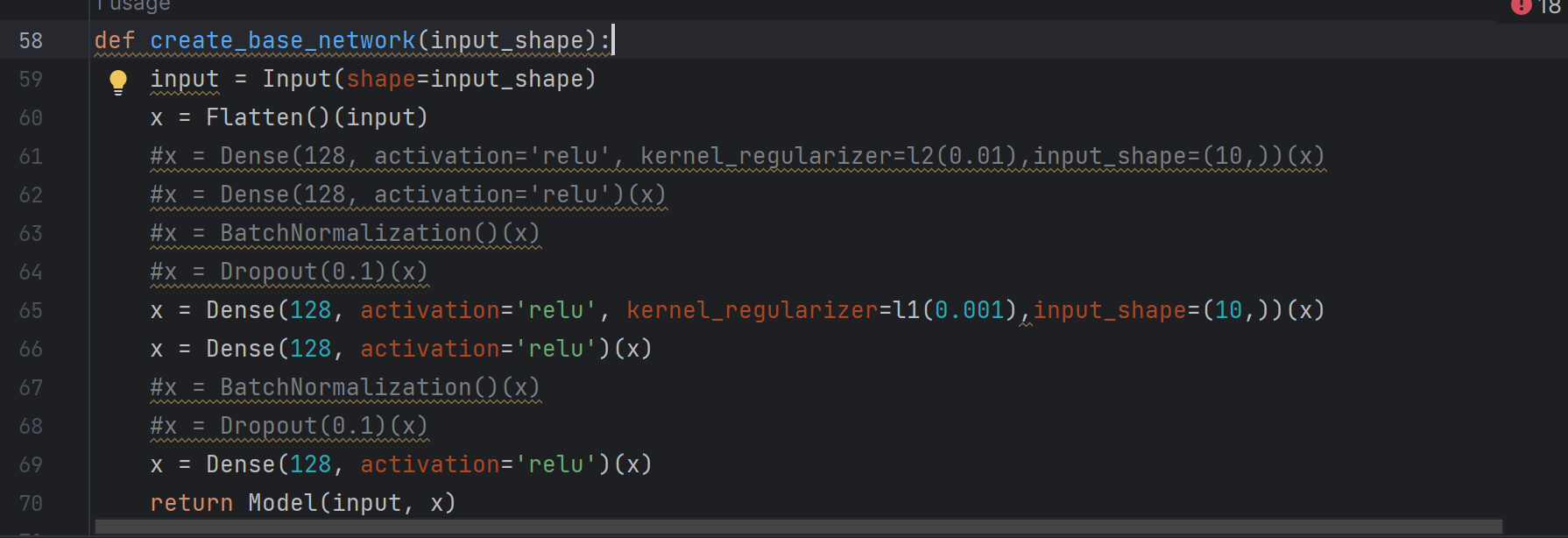


Результати

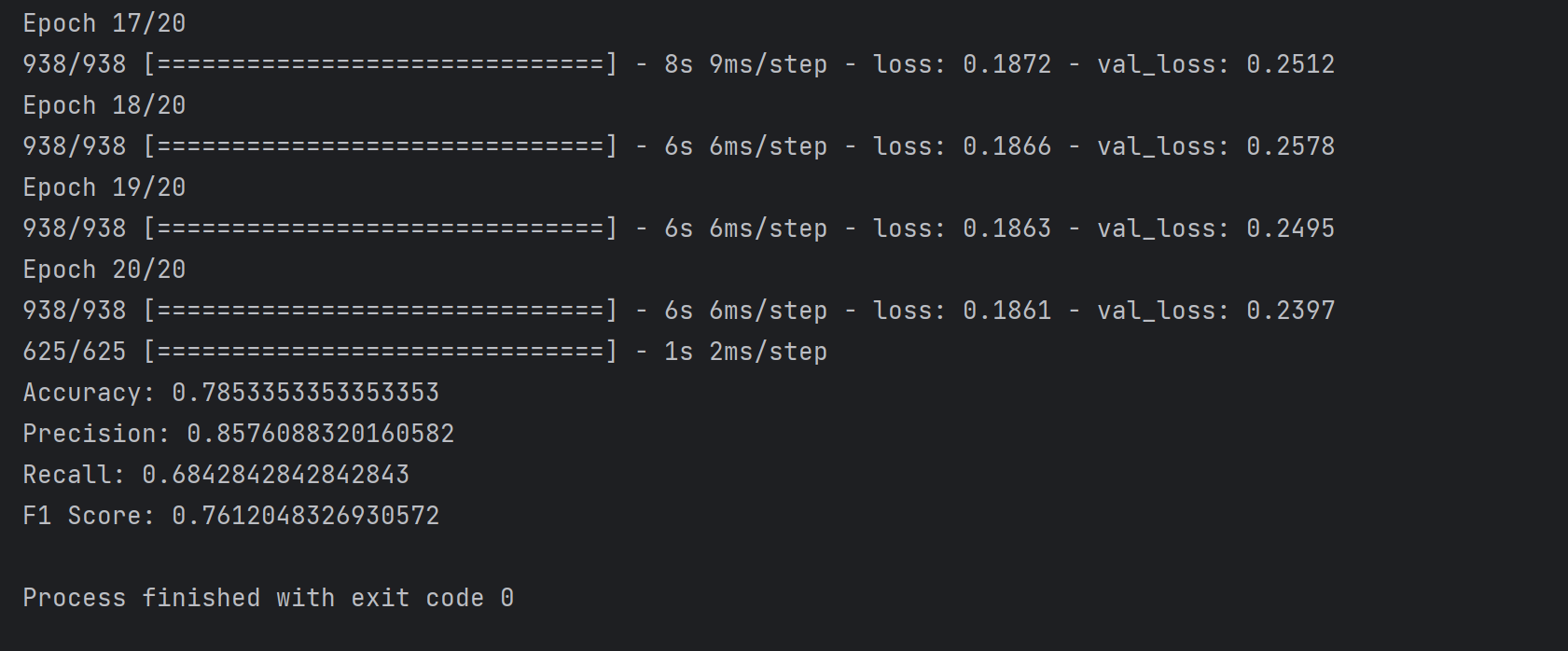
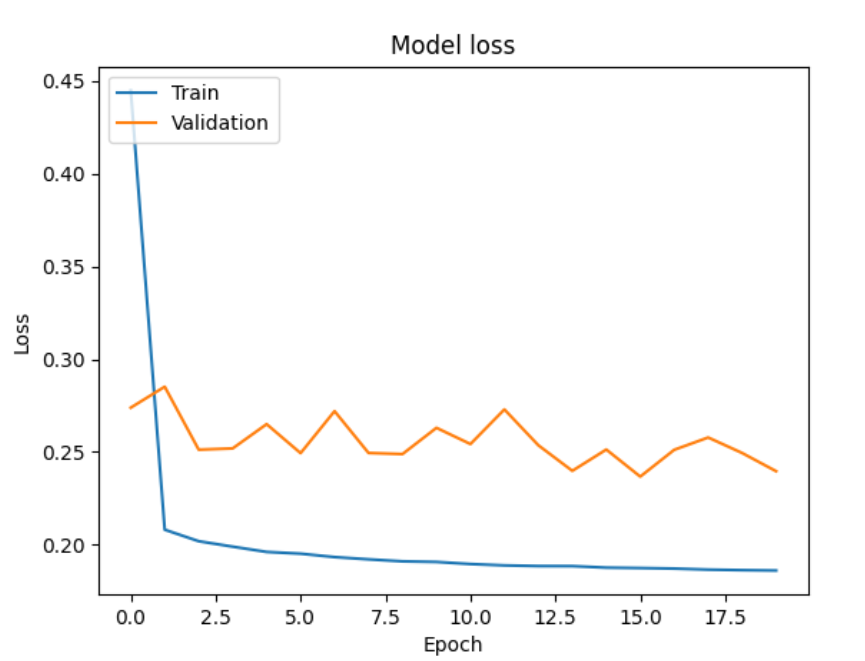




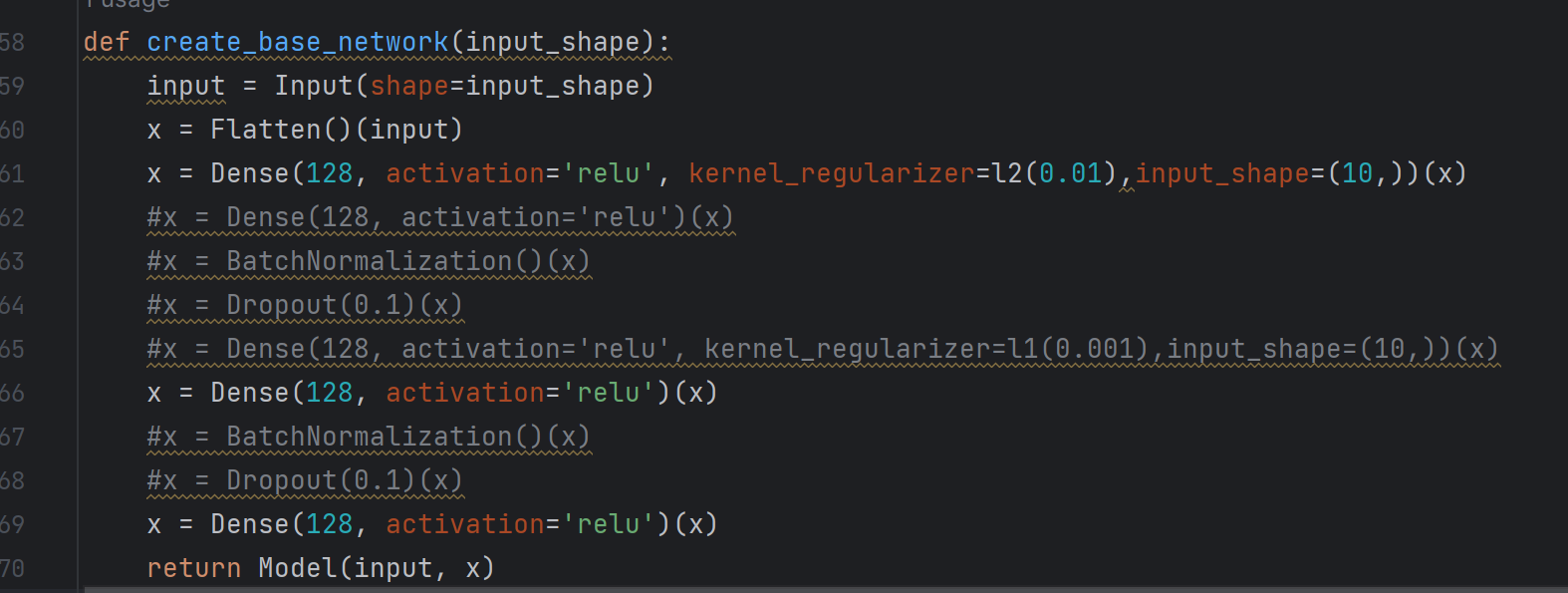
регуляризацію L1



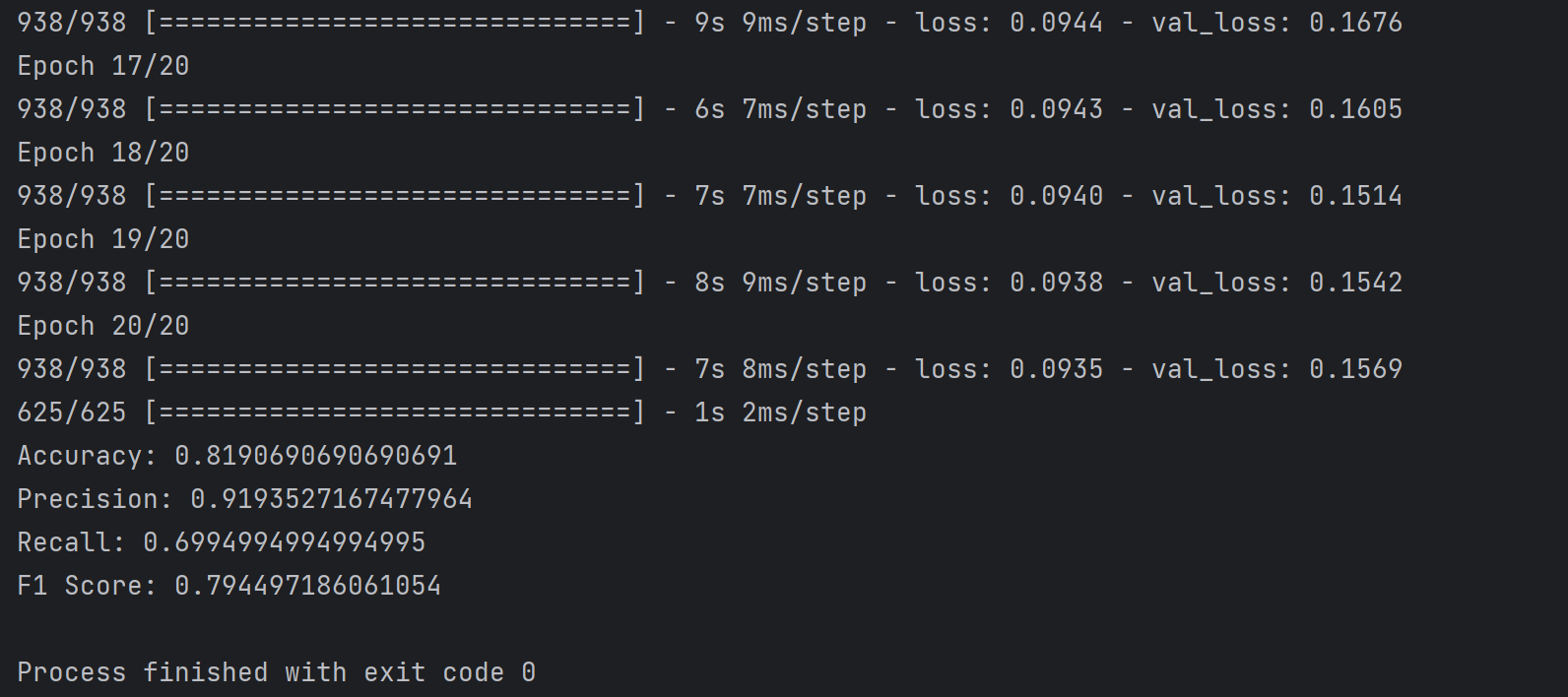
Результати

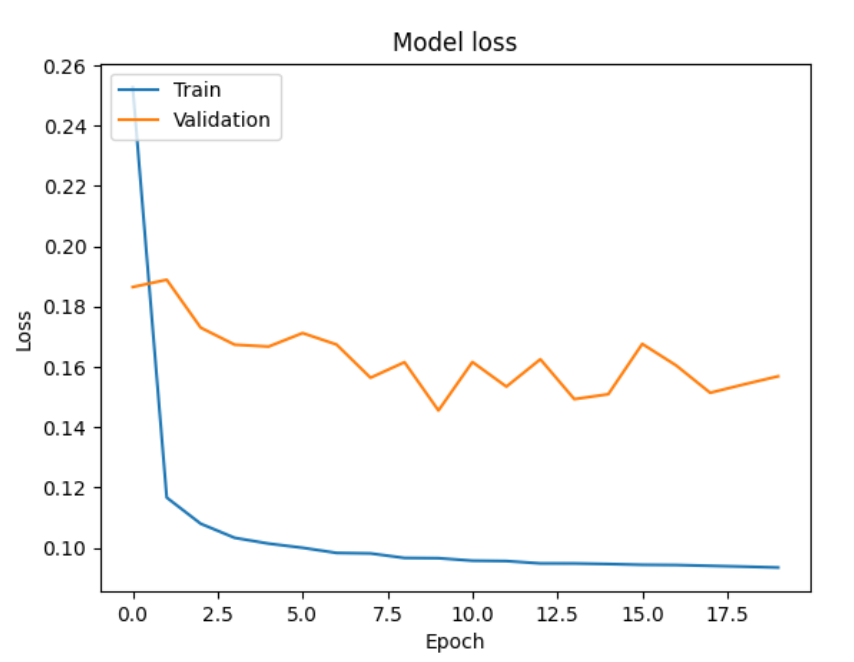
 

регуляризацію L2

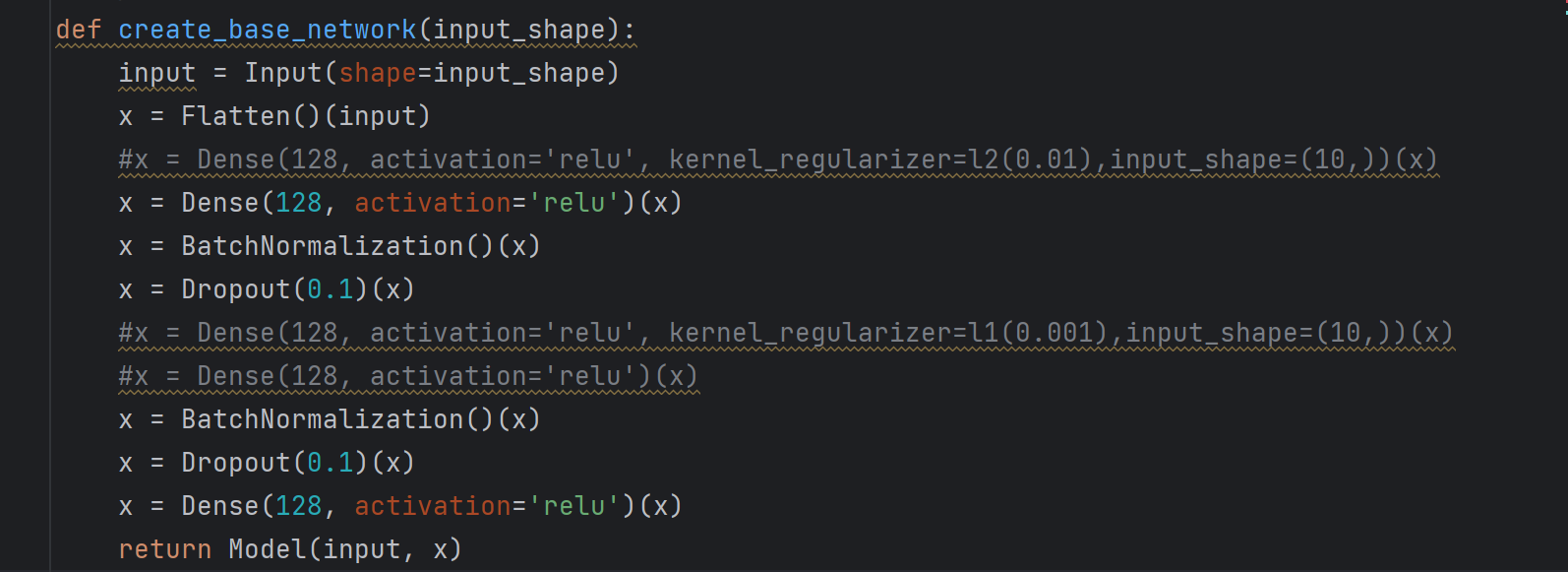


Результати

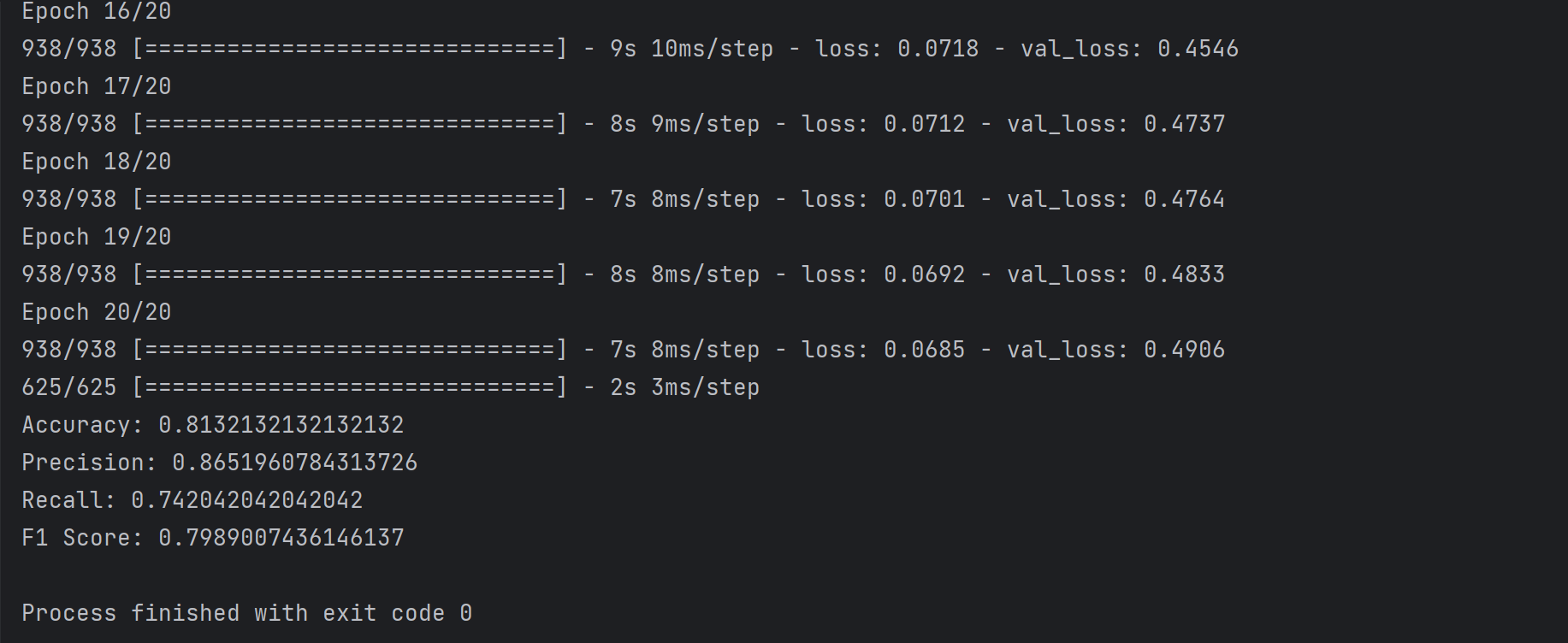


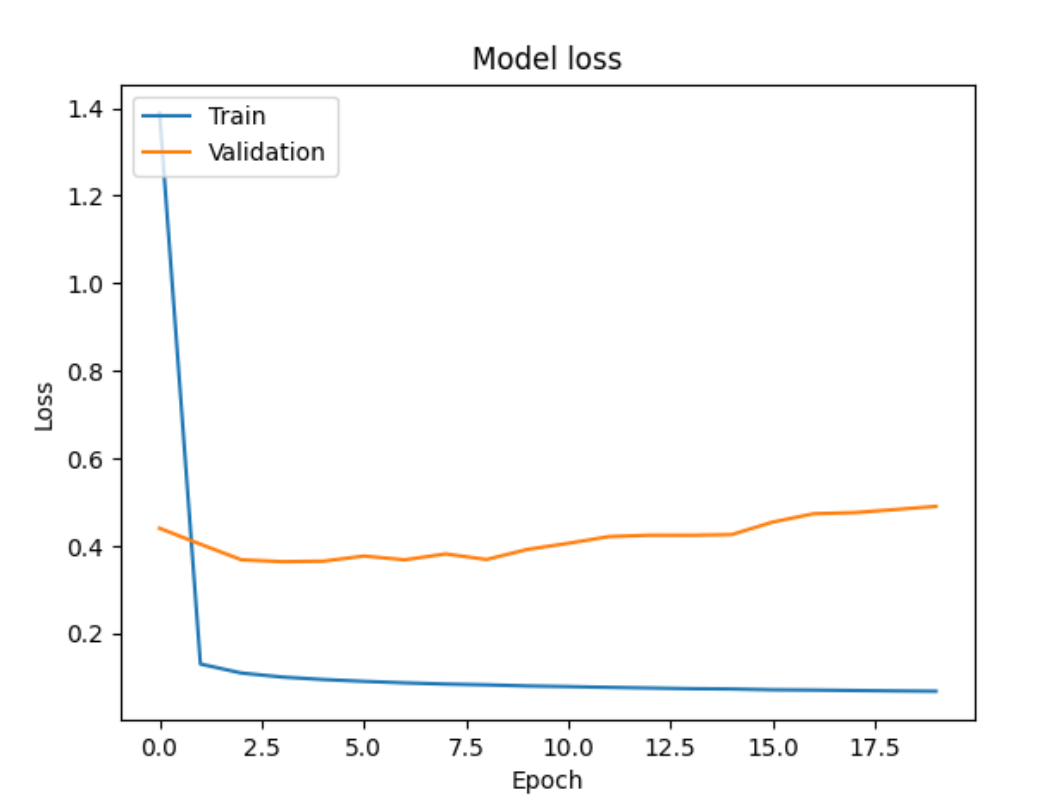


регуляризацію (dropout) та нормалізацію

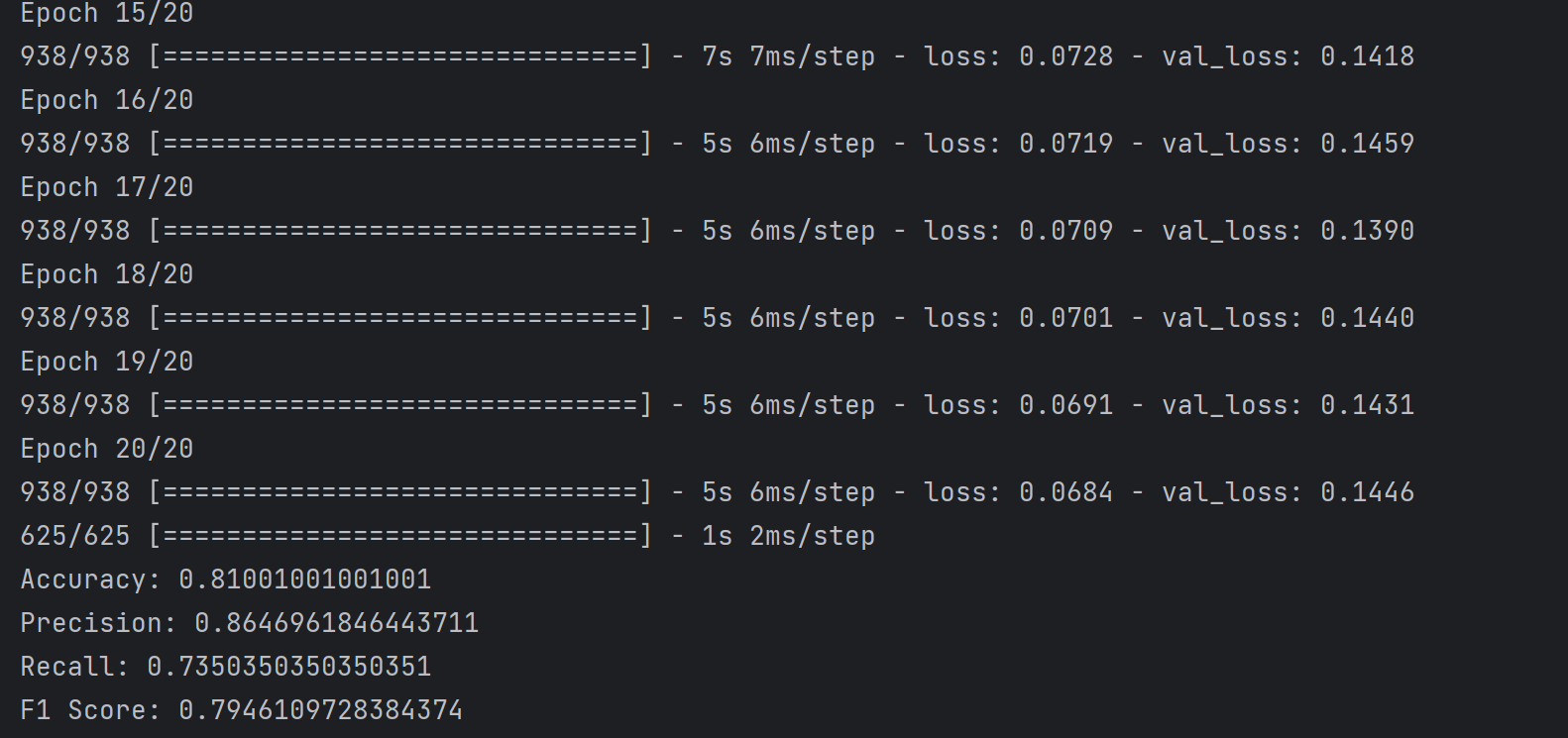


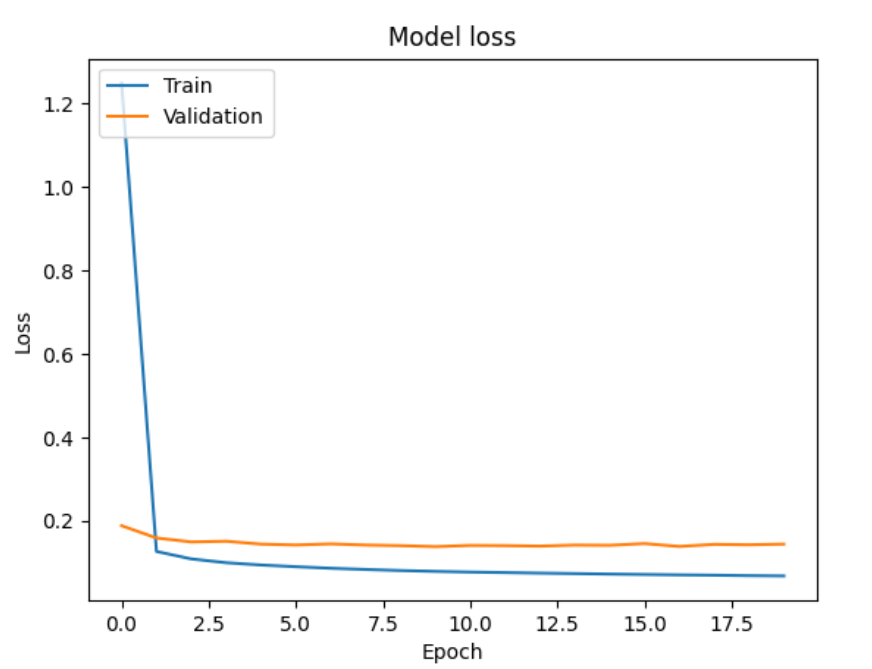
Результати



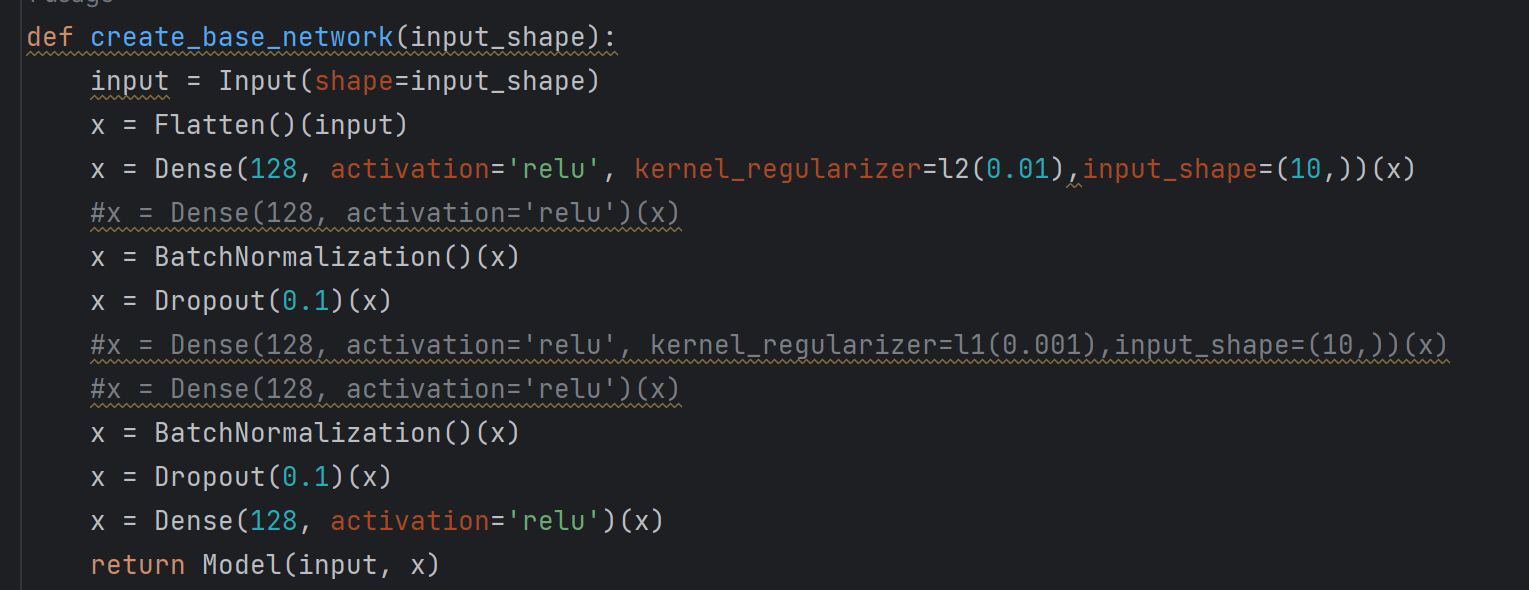


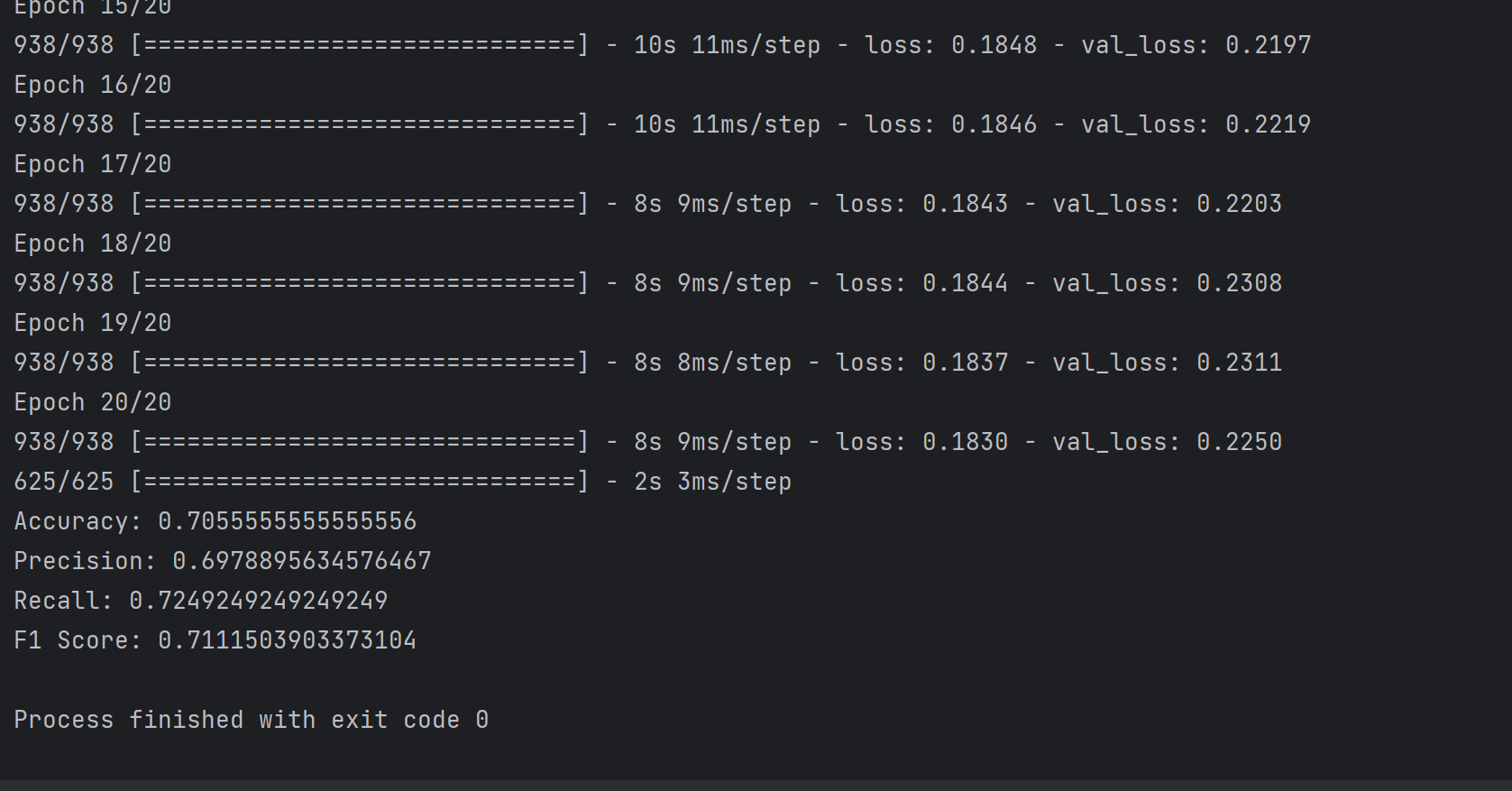
Приберемо одину нормалізацію

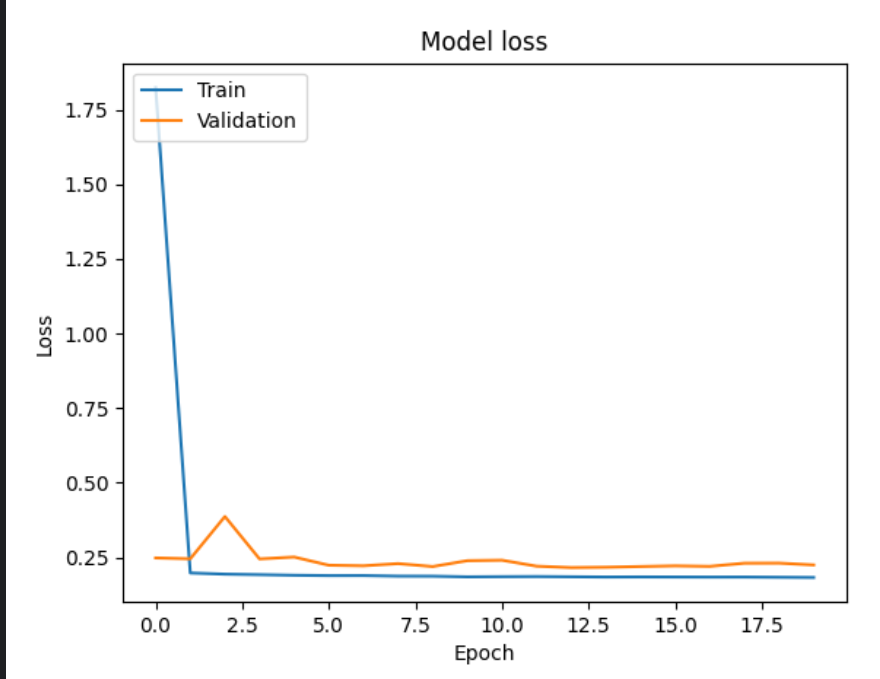




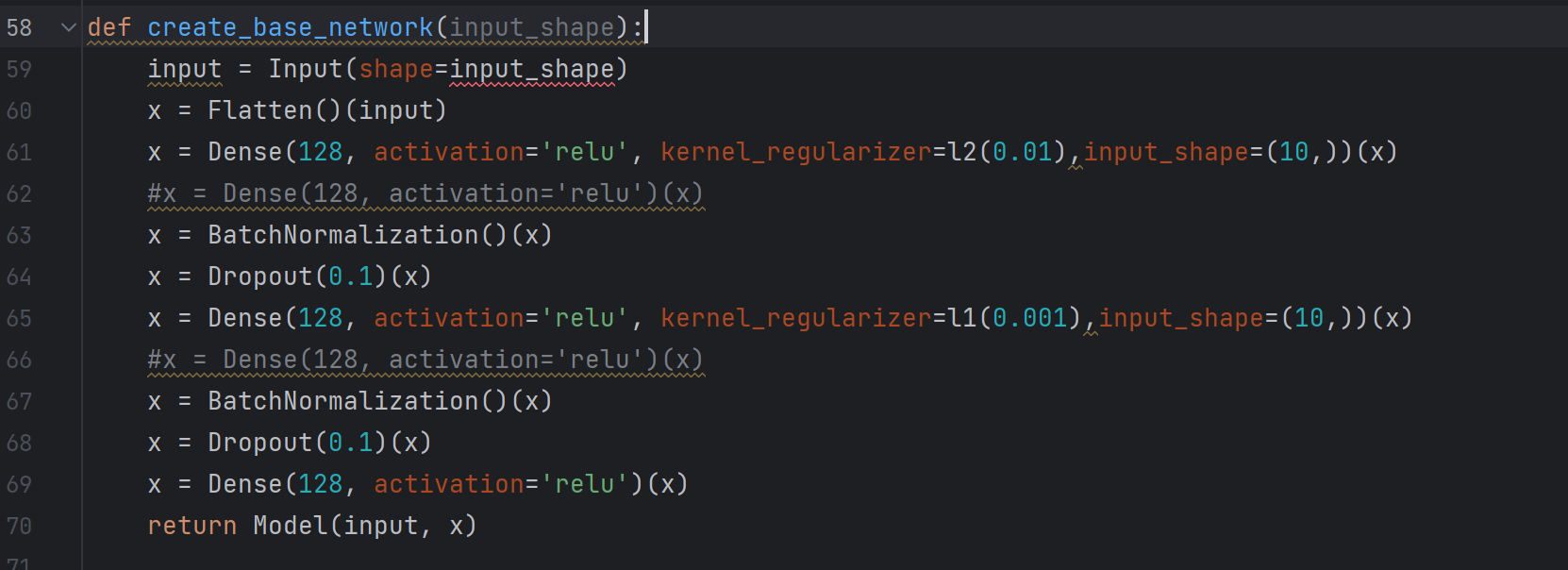
регуляризацію (dropout, L2) та нормалізацію



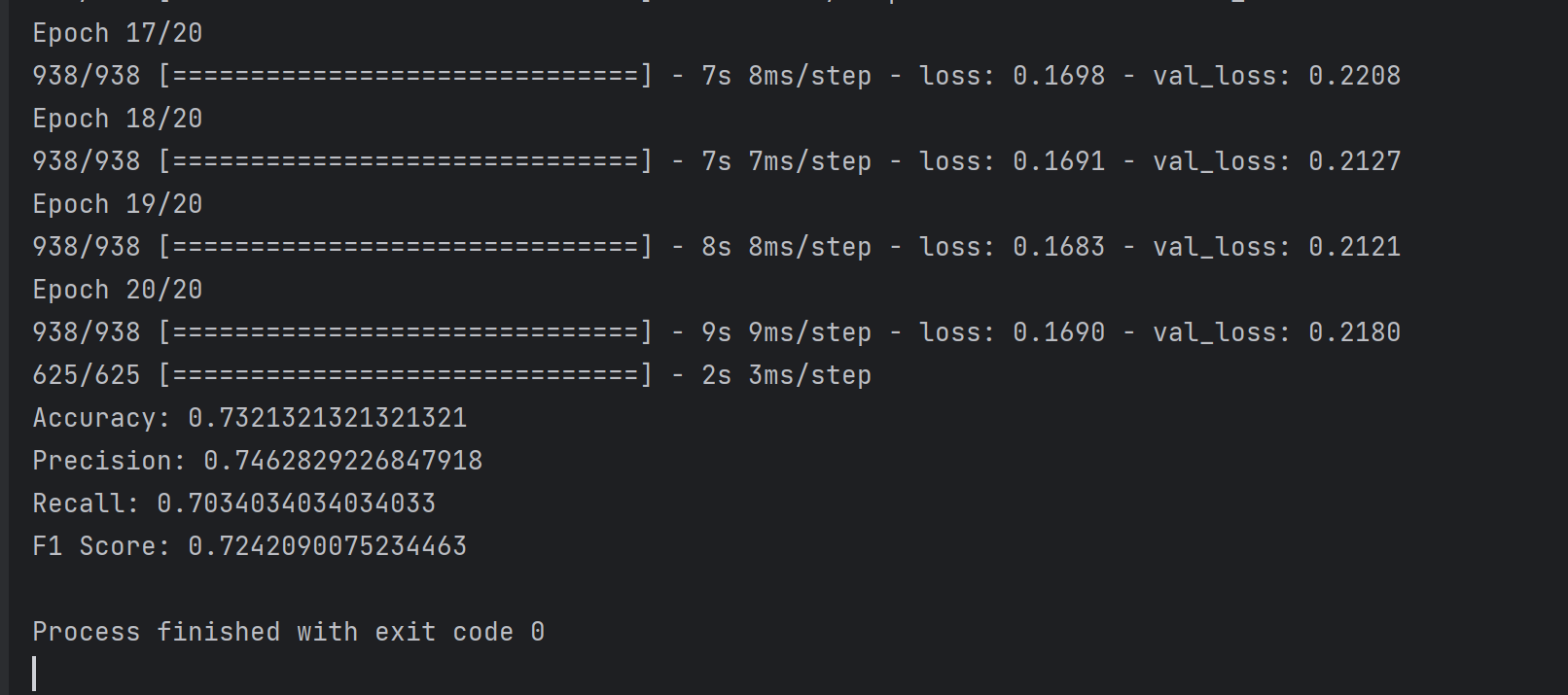
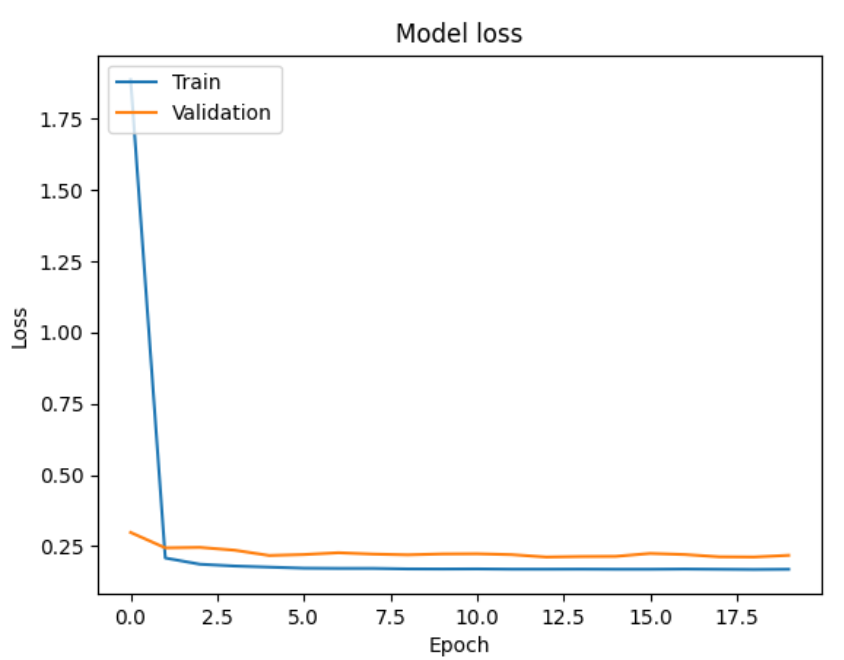
Результати



регуляризацію (dropout, L2, L1) та нормалізацію



Результати

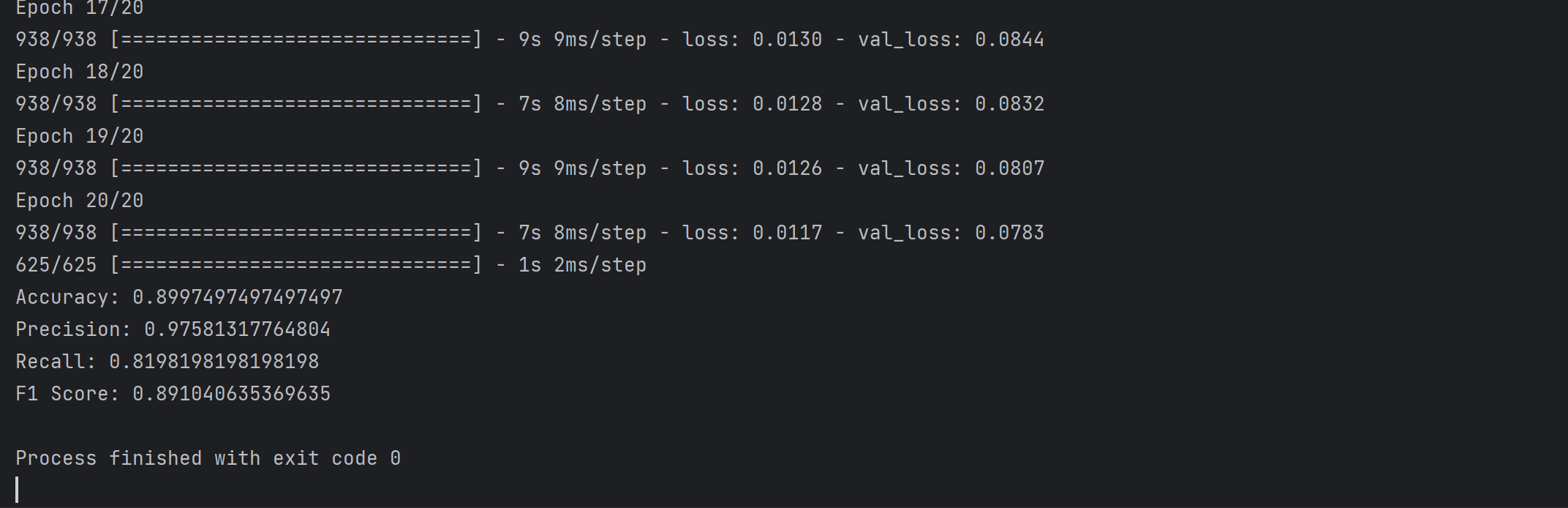
Порівнявши результати, що найкращий результат був отриманий під час регулізації Dropout (Accuracy = 0.8784, Precision = 0.9683, Recall = 0.7824, F1 Score = 0.8655), далі найкращий результат був отриманий після моделювання без нормалізації та без регуляції (Accuracy = 0.8608, Precision=0.9424, Recall = 0.7687 , F1 Score = 0.8467), тому можна зробити висновок, що найефективніше себе показала себе регуляція dropout

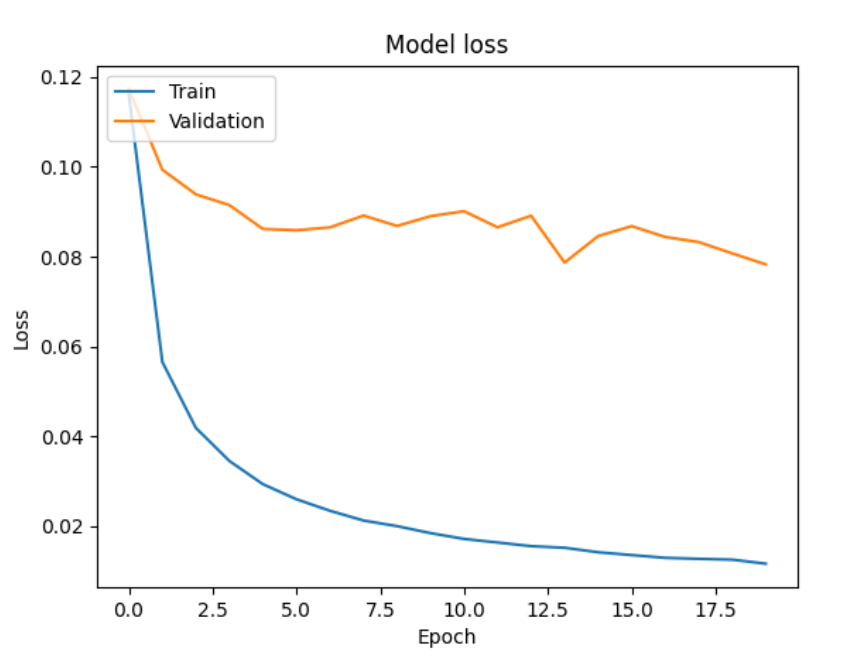
Змінимо структуру базової архітектури

збільшимо кількість шарів

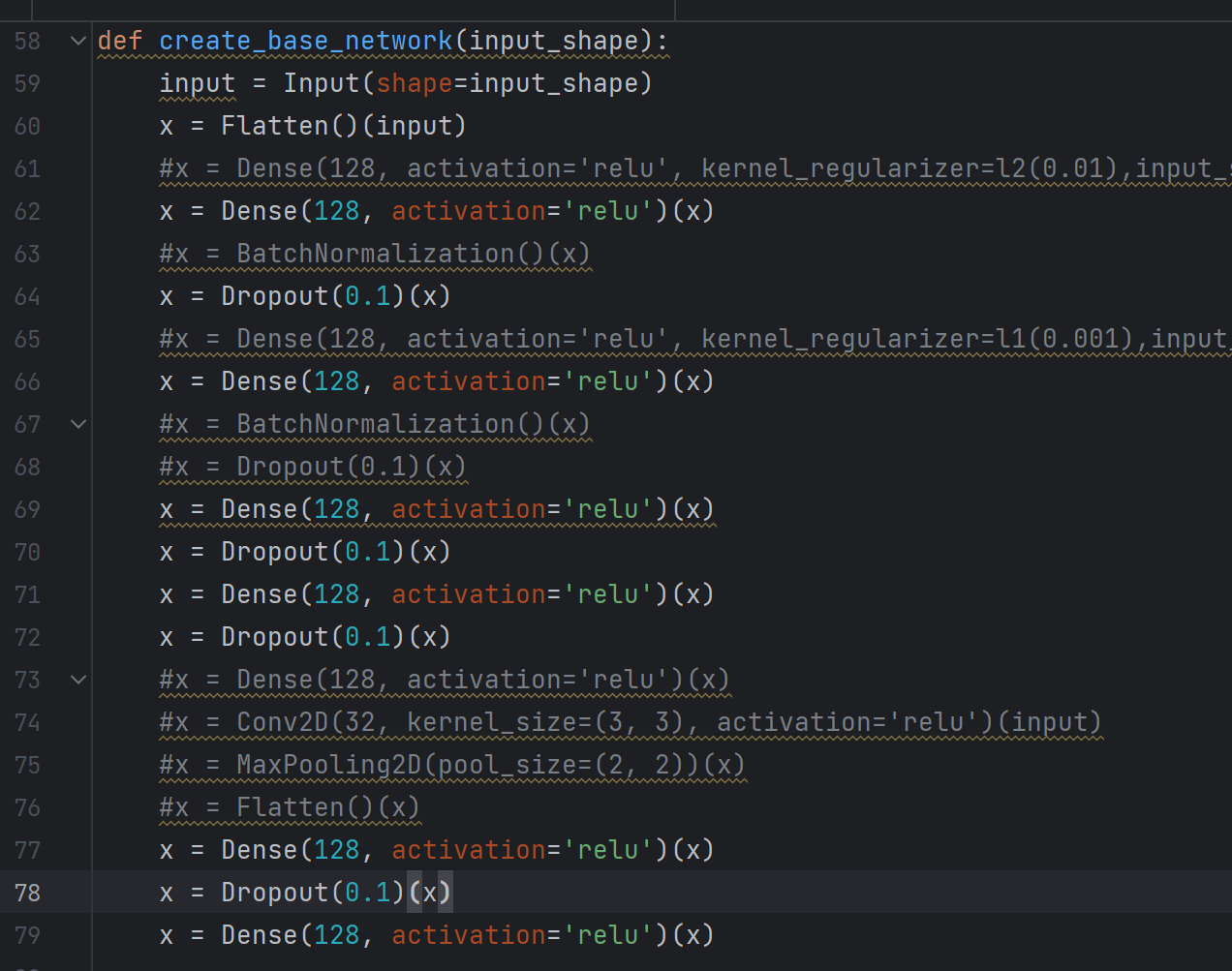


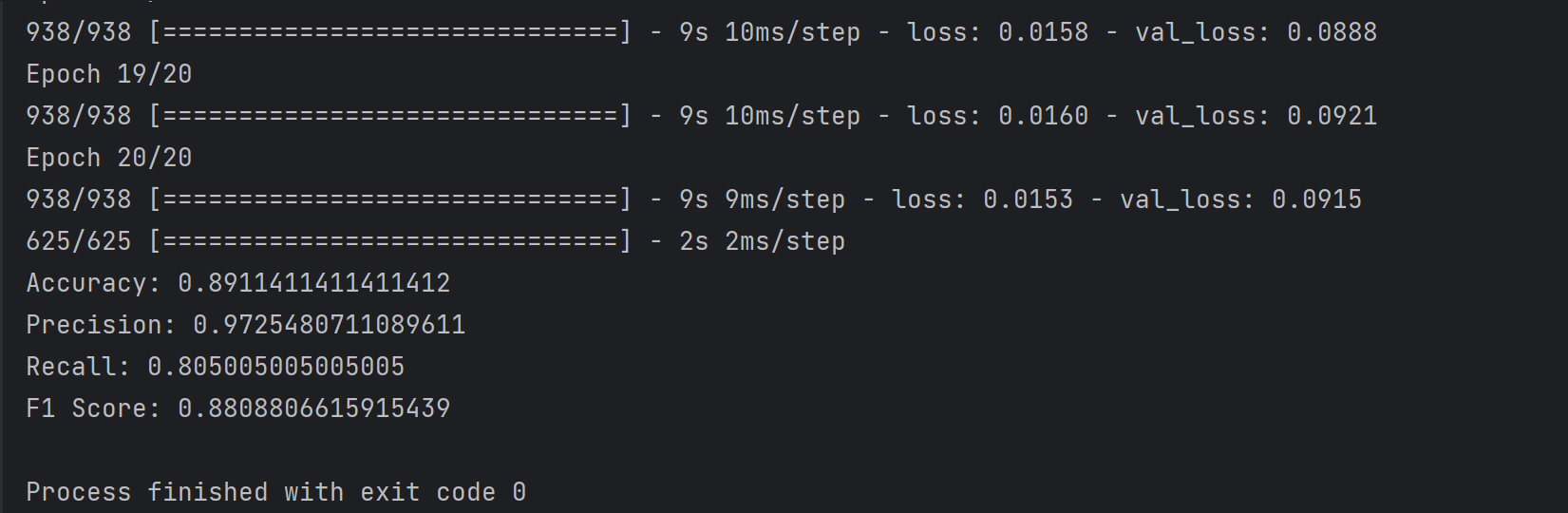
Результати

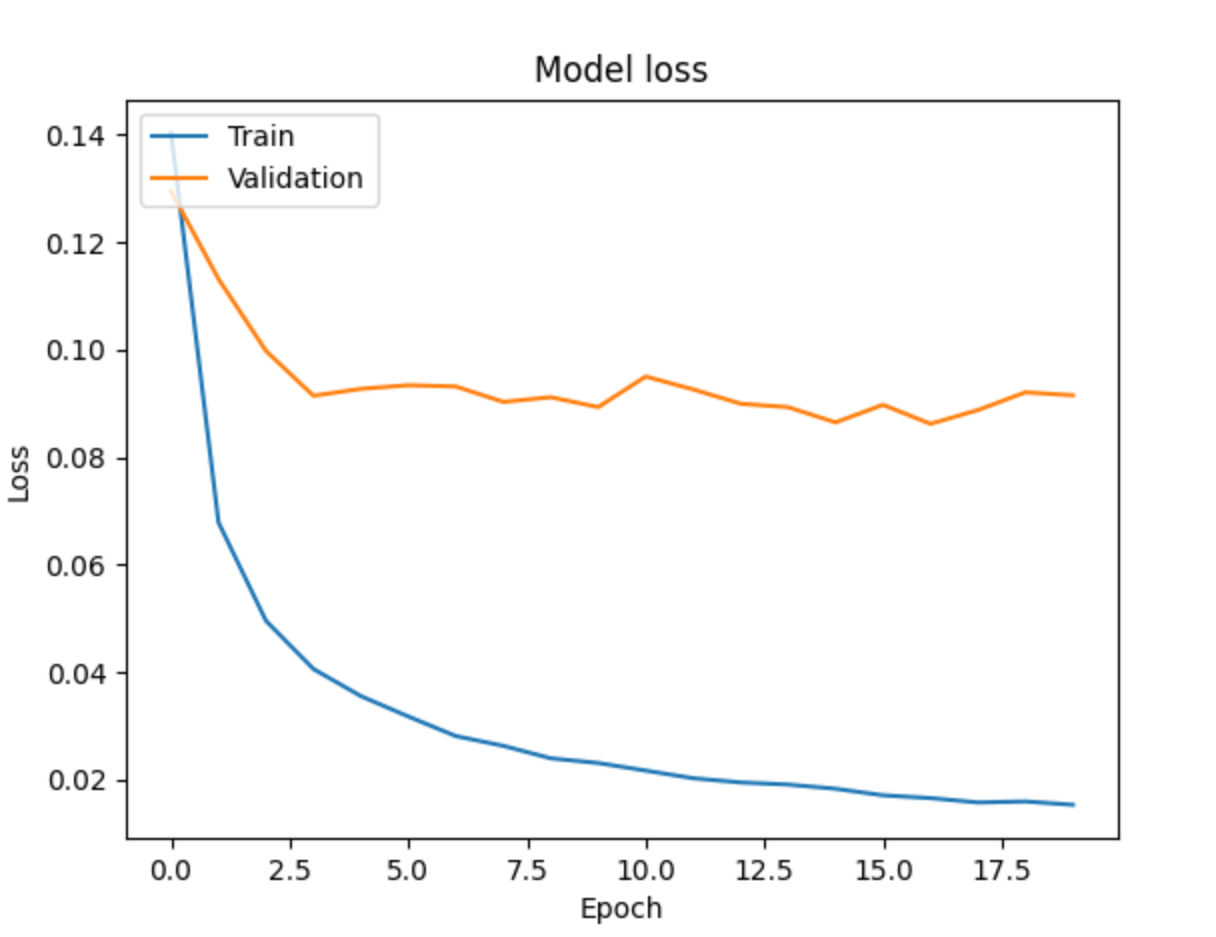




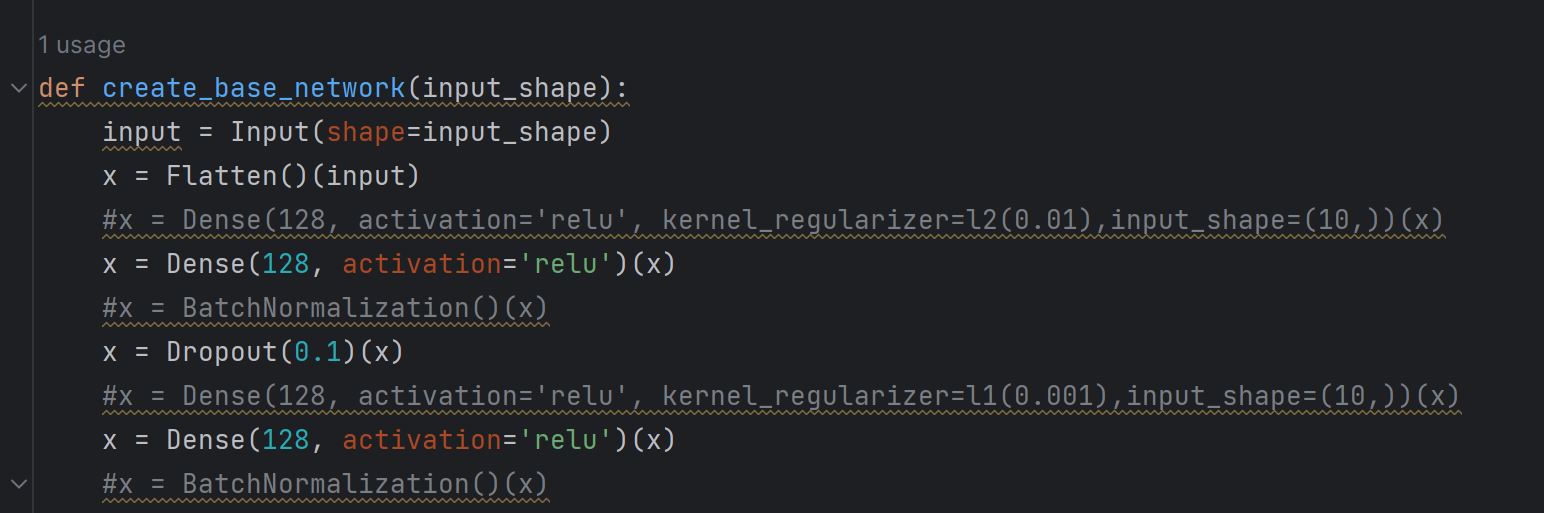
Додамо додаткову регуляцію dropout



Результати

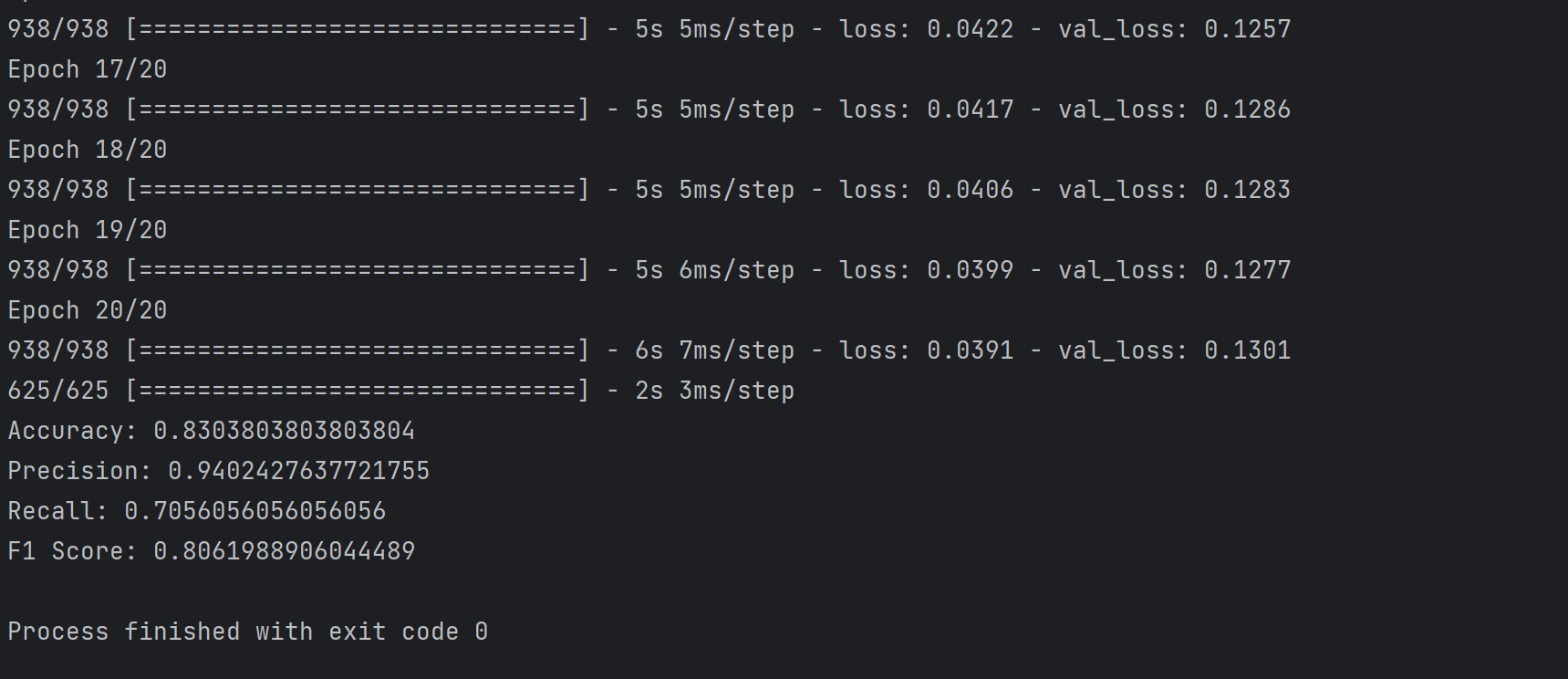


зменшимо кількість шарів



результат

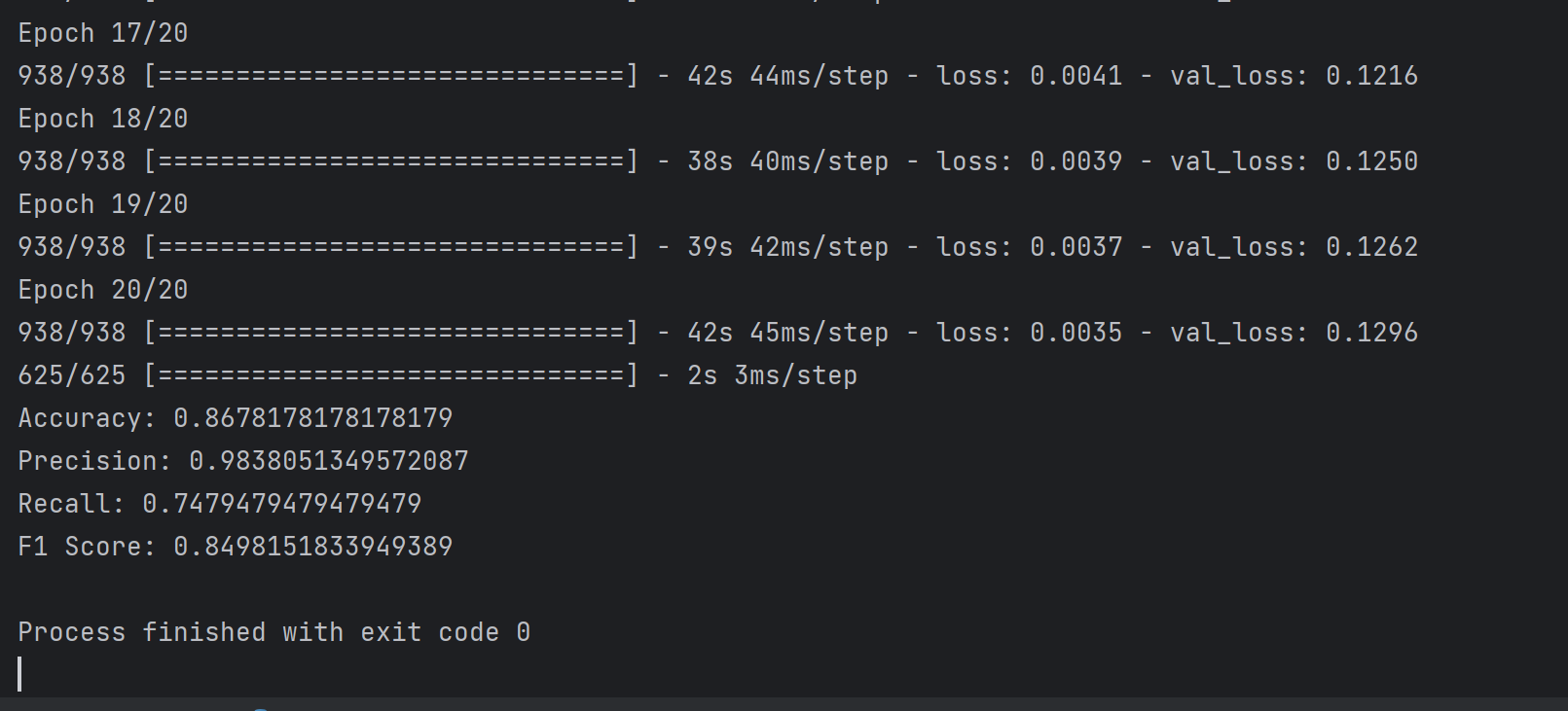


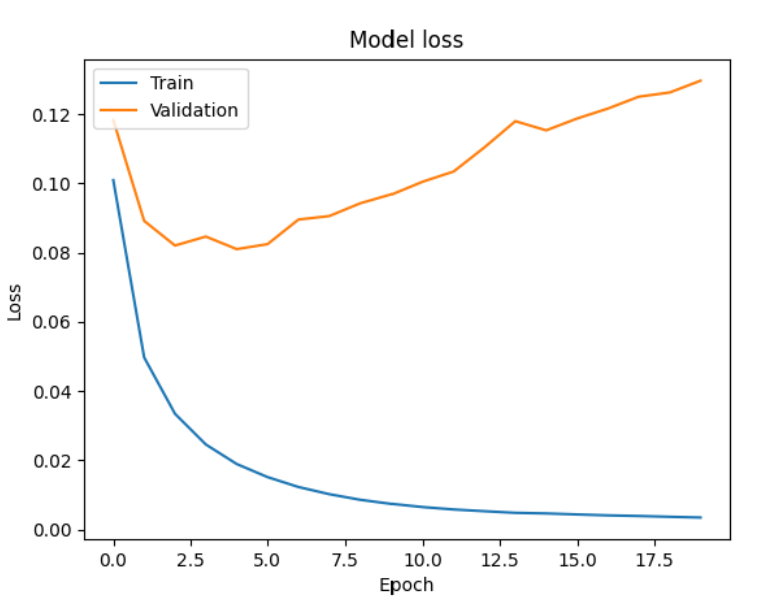


Спробуємо змінити базову архітектуру мережі, використовуючи шари згорткової нейронної мережі (Convolutional Neural Network, CNN):

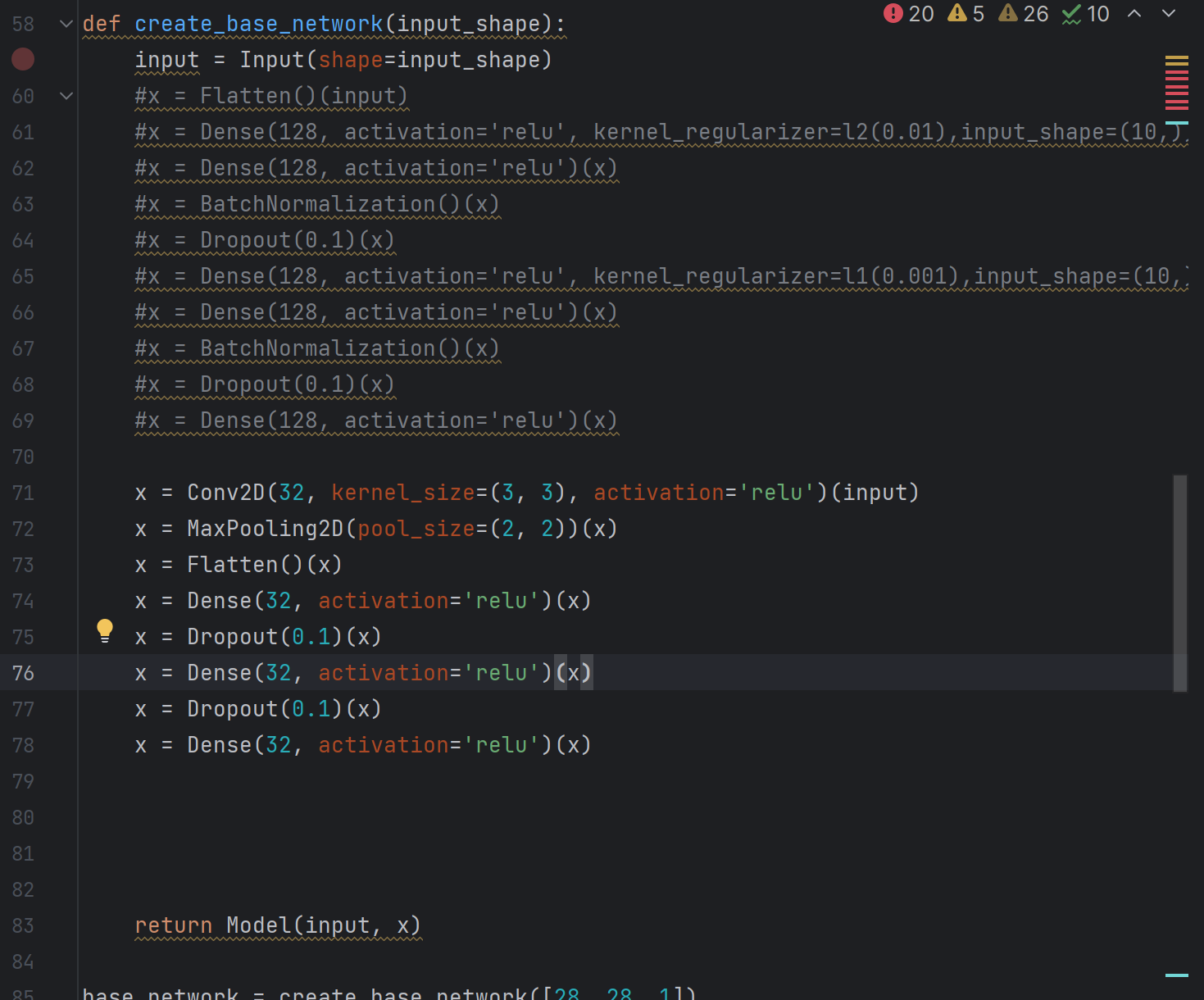


Результат

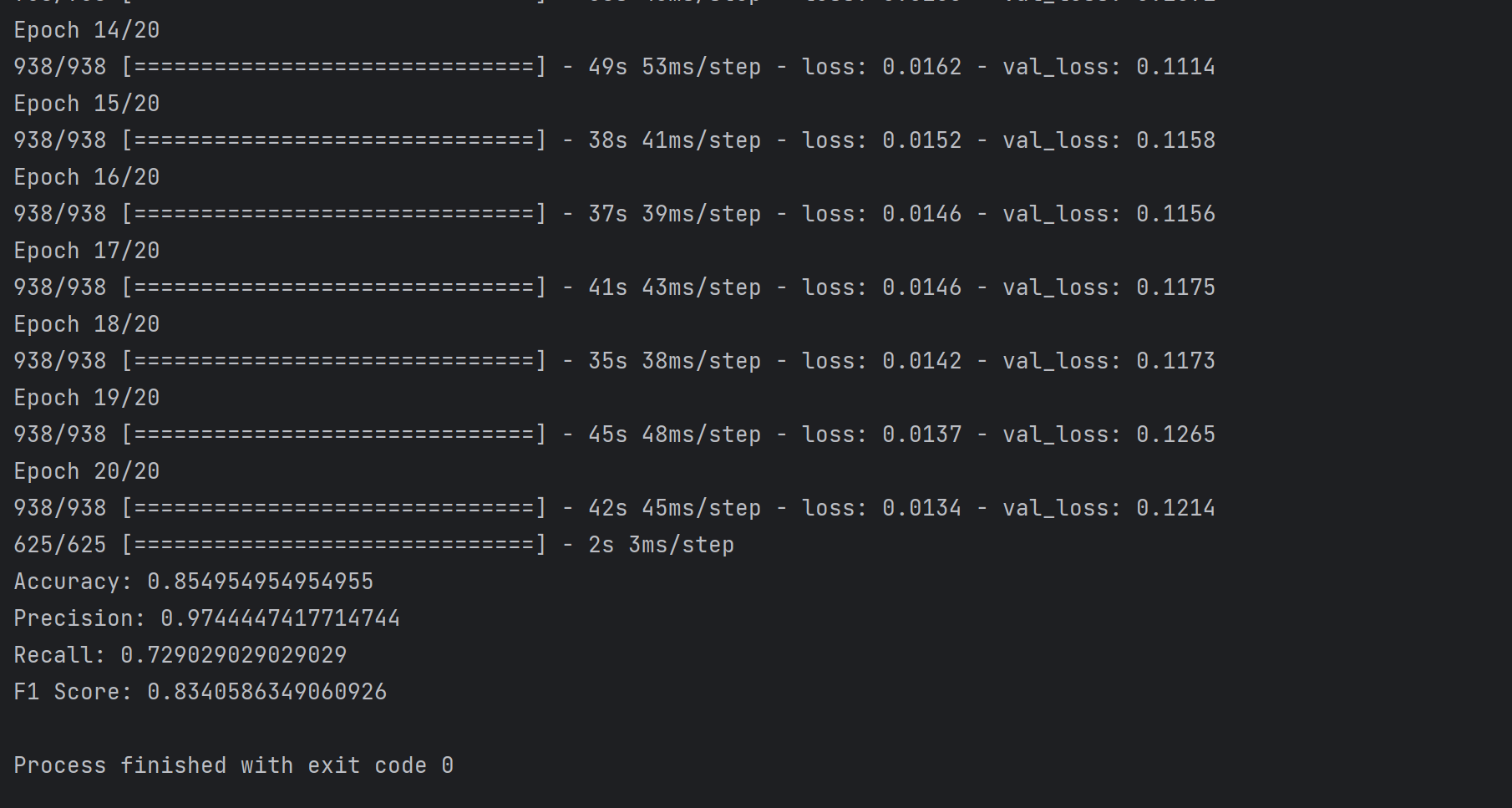


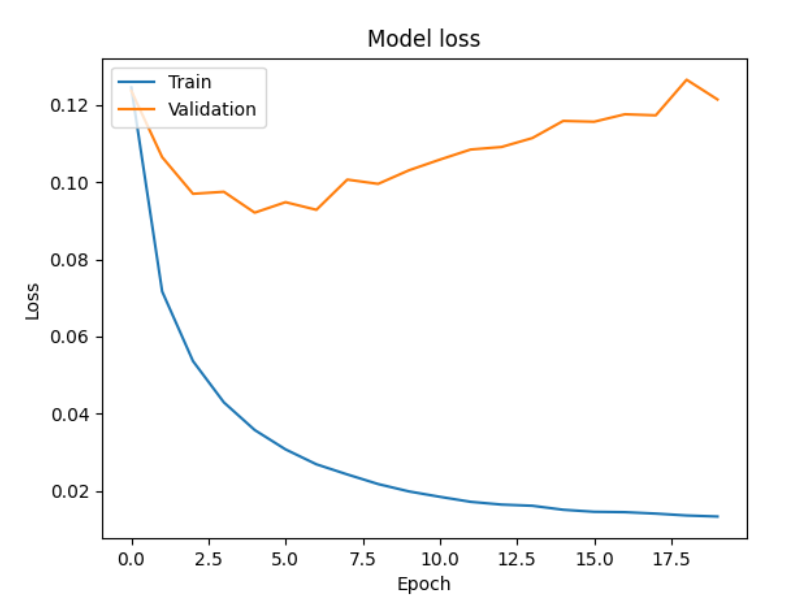


Додамо регуляцію dropout

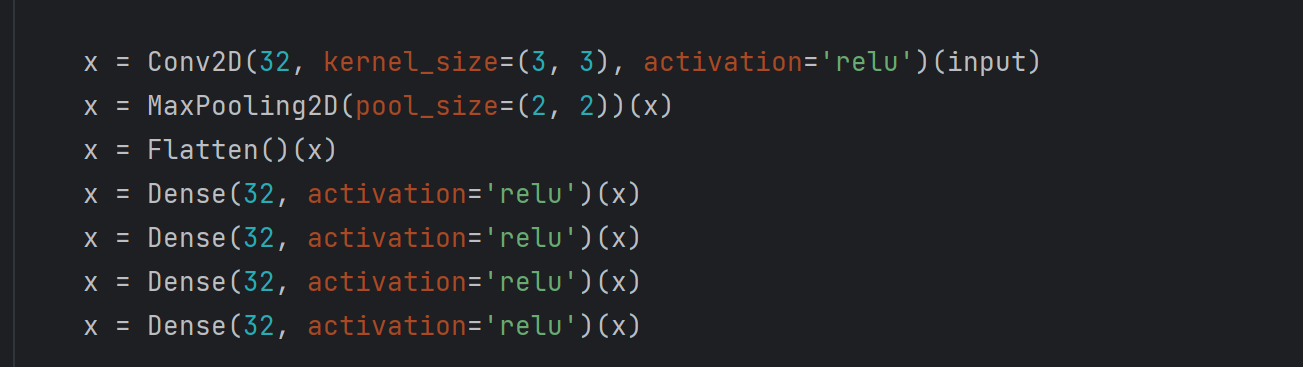


Результат

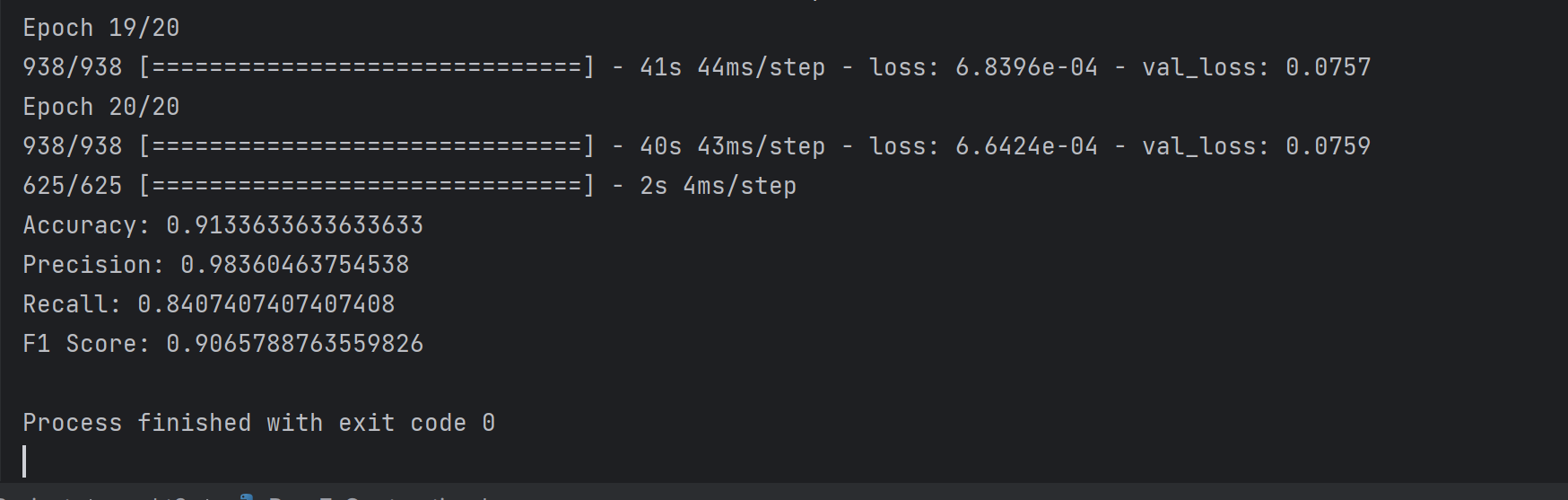


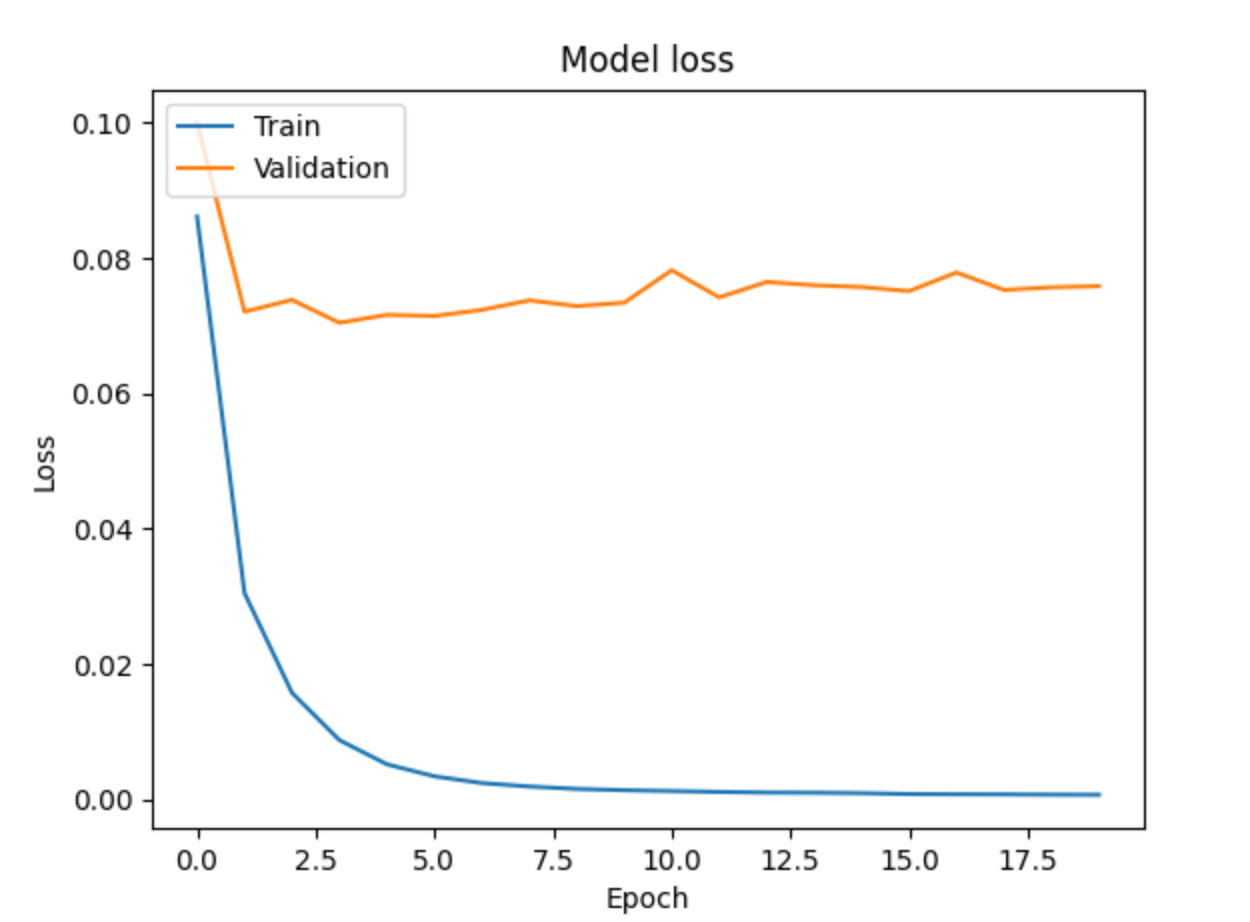


Додамо додаткові шари

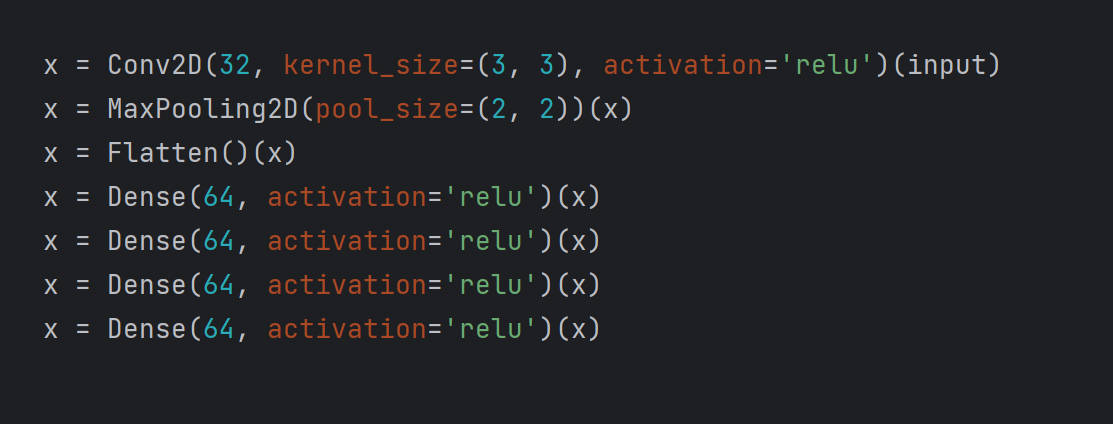


Результат

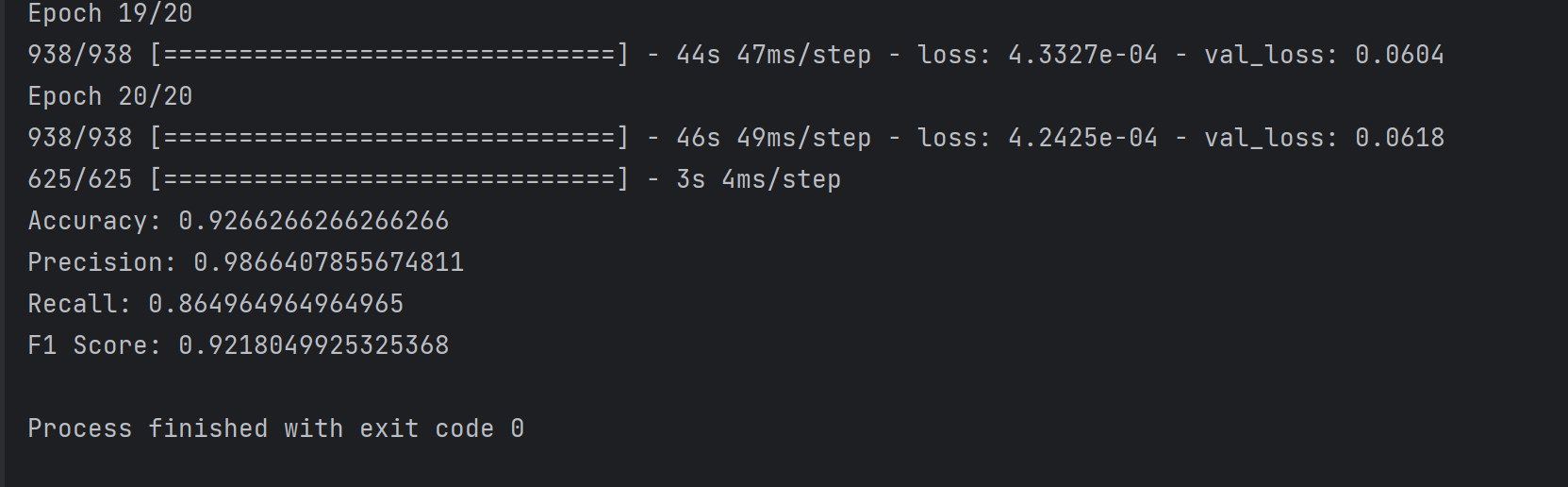


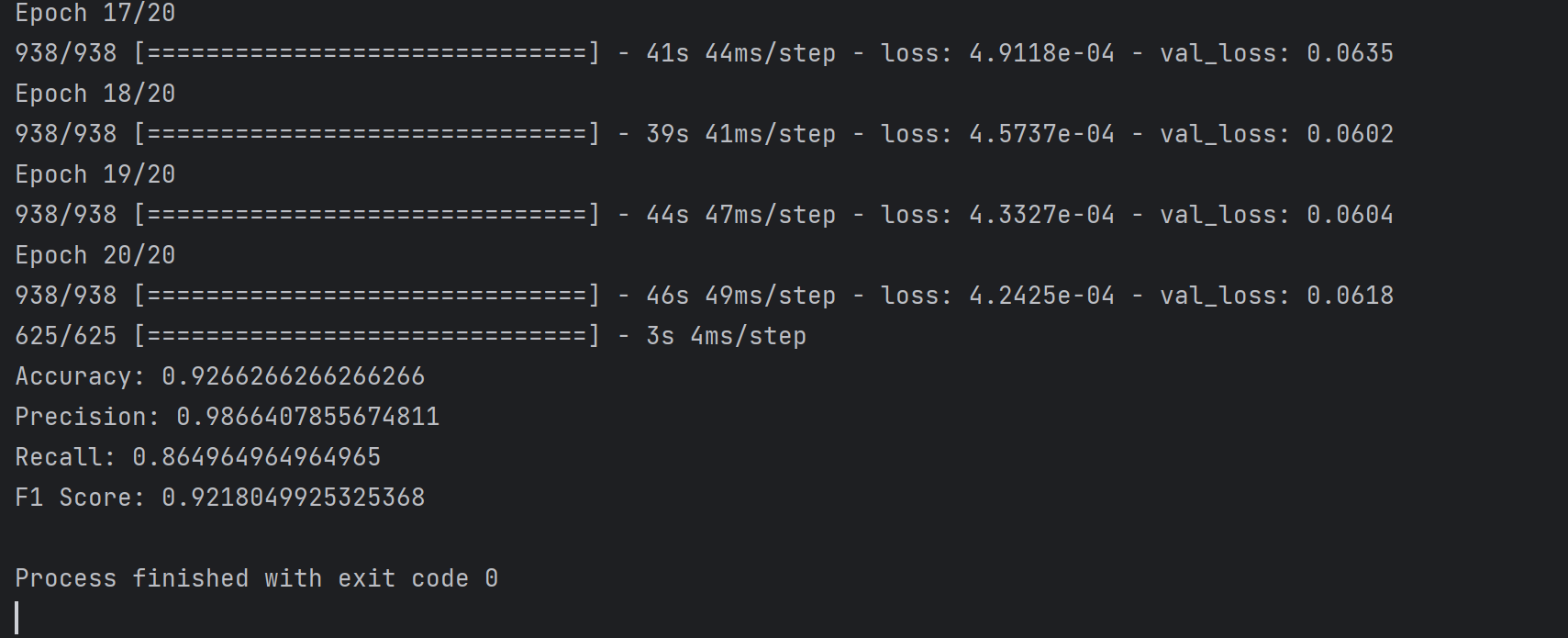


Збільшимо розміри шарів



Результат

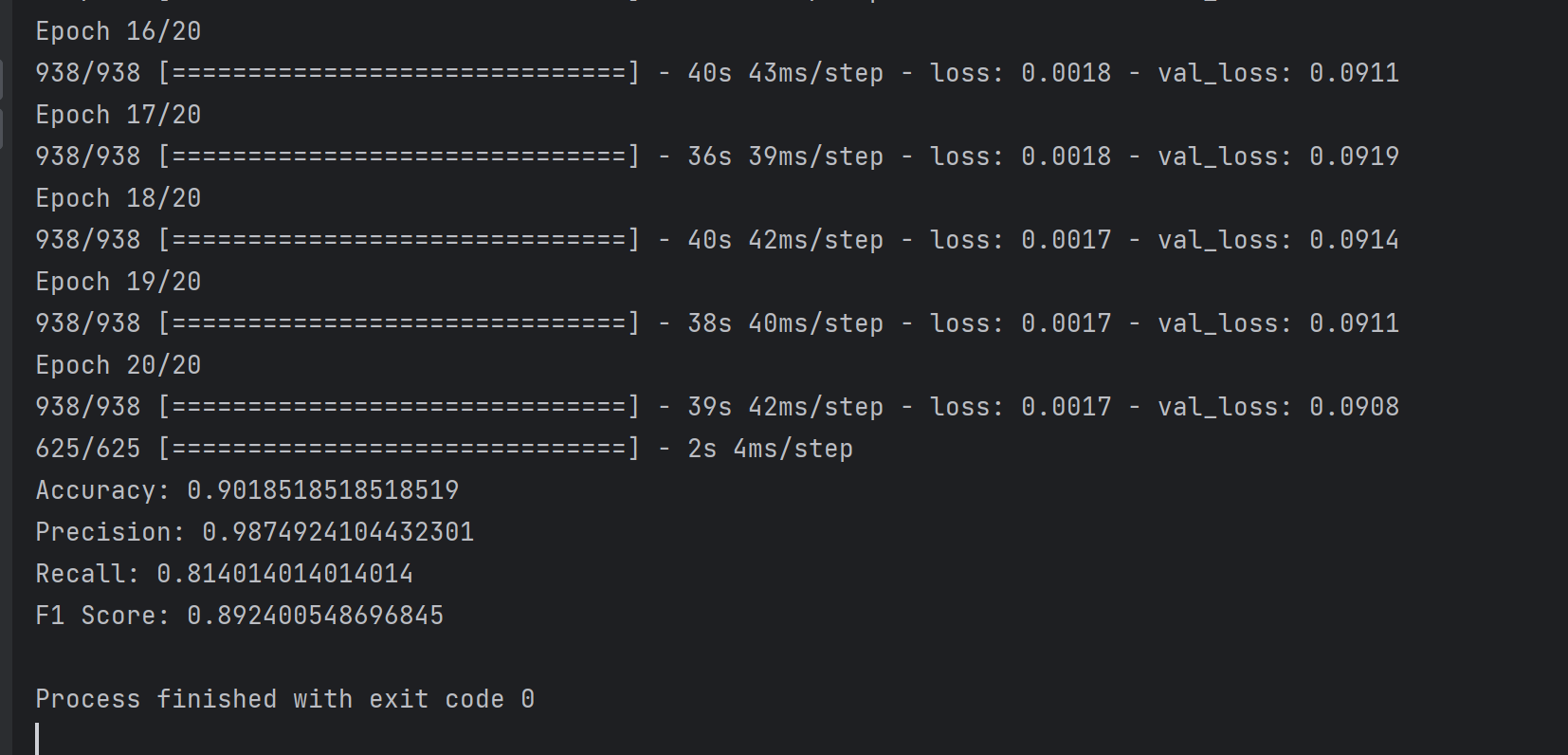
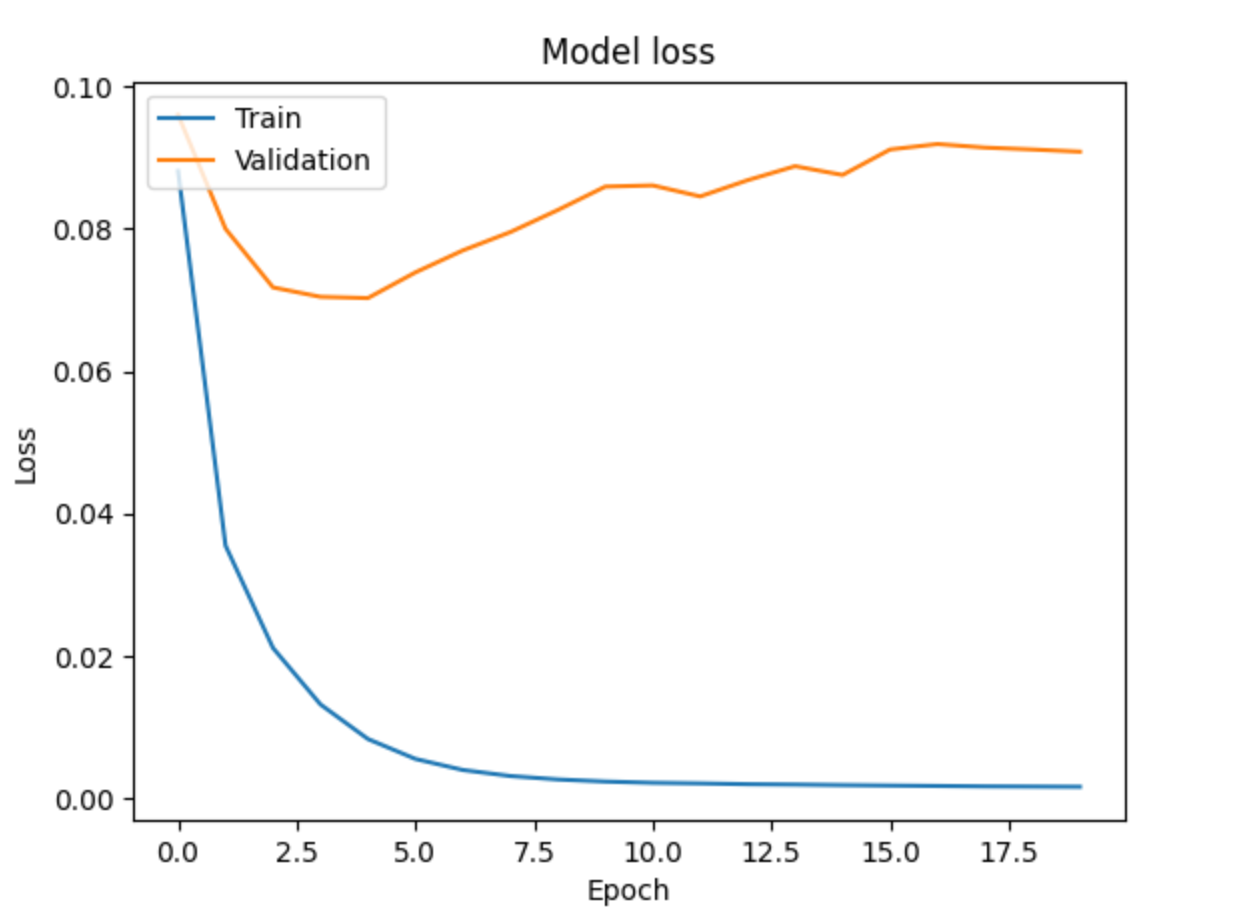




Збільшимо розміри шарів та зменшимо кількість шарів



Результати

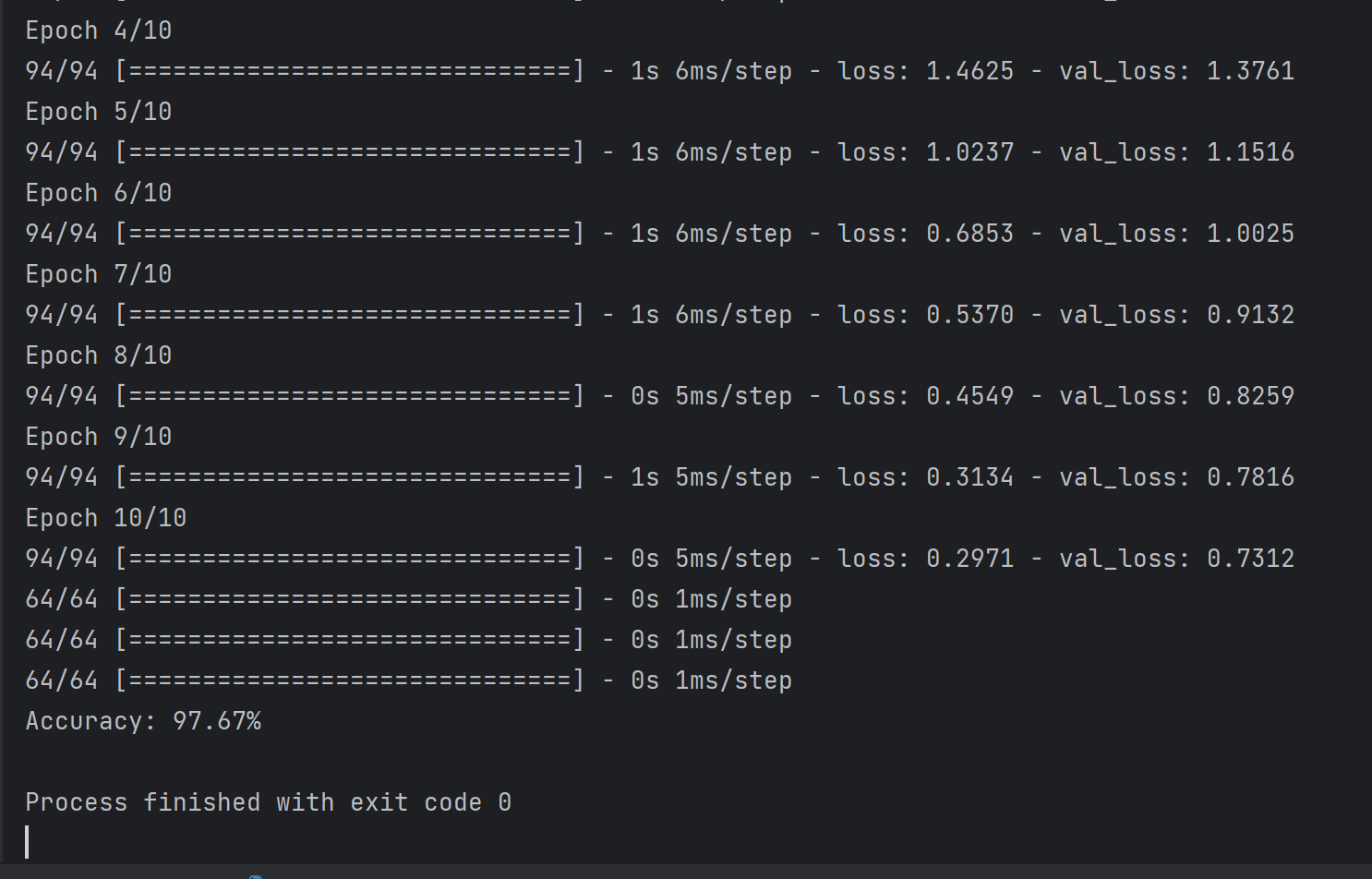
 

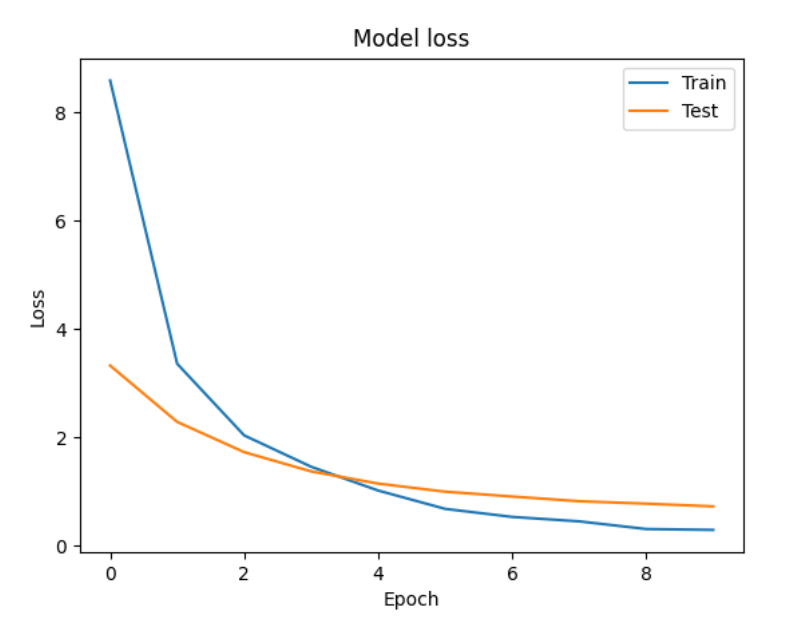
У результаті змін архітектури нашої сіамської нейроної мережі можна зробити висновок, що згорткова нейронна мережа, збільшення кількості та розмірів шарів покращала результат, збільшивши точність нашої мережі при роботі з нашими даними

Реалізуемо використання triplet loss замість contrastive loss.

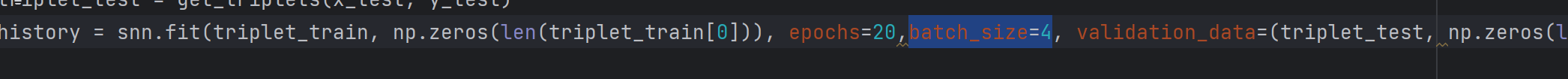
Код

import gzip  
import idx2numpy  
import numpy as np  
import tensorflow as tf  
from tensorflow.keras.datasets import mnist  
from tensorflow.keras.models import Model  
from tensorflow.keras.layers import Input, Flatten, Dense, Dropout, Lambda  
from tensorflow.keras.optimizers import Adam  
import matplotlib.pyplot as plt  
  
def load\_kmnist():  
 with gzip.open('train-images-idx3-ubyte.gz', 'r') as f:  
 x\_train = idx2numpy.convert\_from\_file(f)  
 with gzip.open('train-labels-idx1-ubyte.gz', 'r') as f:  
 y\_train = idx2numpy.convert\_from\_file(f)  
 with gzip.open('t10k-images-idx3-ubyte.gz', 'r') as f:  
 x\_test = idx2numpy.convert\_from\_file(f)  
 with gzip.open('t10k-labels-idx1-ubyte.gz', 'r') as f:  
 y\_test = idx2numpy.convert\_from\_file(f)  
 return (x\_train, y\_train), (x\_test, y\_test)  
(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()  
  
x\_train = x\_train.astype('float32')  
x\_test = x\_test.astype('float32')  
x\_train = x\_train[:15000, :, :]  
x\_test = x\_test[:10000, :, :]  
y\_train = y\_train[:15000]  
y\_test = y\_test[:10000]  
x\_train /= 255.  
x\_test /= 255.  
  
def create\_base\_model(input\_shape):  
 input = Input(shape=input\_shape)  
 x = Flatten()(input)  
 x = Dense(128, activation='relu')(x)  
 x = Dropout(0.1)(x)  
 x = Dense(128, activation='relu')(x)  
 x = Dropout(0.1)(x)  
 x = Dense(128, activation='relu')(x)  
 return Model(inputs=input, outputs=x)  
  
  
base\_model = create\_base\_model((28, 28))  
  
  
  
def triplet\_loss(y\_true, y\_pred, alpha = 0.4):  
 total\_lenght = y\_pred.shape.as\_list()[-1]  
 anchor, positive, negative = y\_pred[:,:int(total\_lenght\*1/3)], y\_pred[:,int(total\_lenght\*1/3):int(total\_lenght\*2/3)], y\_pred[:,int(total\_lenght\*2/3):]  
 pos\_dist = tf.reduce\_sum(tf.square(anchor - positive), axis=-1)  
 neg\_dist = tf.reduce\_sum(tf.square(anchor - negative), axis=-1)  
 basic\_loss = pos\_dist - neg\_dist + alpha  
 loss = tf.reduce\_sum(tf.maximum(basic\_loss, 0.0))  
 return loss  
  
anchor\_input = Input((28,28), name='anchor\_input')  
positive\_input = Input((28,28), name='positive\_input')  
negative\_input = Input((28,28), name='negative\_input')  
anchor\_embedding = base\_model(anchor\_input)  
positive\_embedding = base\_model(positive\_input)  
negative\_embedding = base\_model(negative\_input)  
merged\_vector = tf.keras.layers.concatenate([anchor\_embedding,  
positive\_embedding, negative\_embedding], axis=-1)  
snn = Model(inputs=[anchor\_input,positive\_input, negative\_input],  
outputs=merged\_vector)  
snn.compile(optimizer=Adam(0.0001), loss=triplet\_loss)  
  
def create\_base\_model(input\_shape):  
 input = Input(shape=input\_shape)  
 x = Flatten()(input)  
 x = Dense(128, activation='relu')(x)  
 x = Dropout(0.1)(x)  
 x = Dense(128, activation='relu')(x)  
 x = Dropout(0.1)(x)  
 x = Dense(128, activation='relu')(x)  
 return Model(inputs=input, outputs=x)  
base\_model = create\_base\_model((28,28))  
  
def triplet\_loss(y\_true, y\_pred, alpha = 0.4):  
 total\_lenght = y\_pred.shape.as\_list()[-1]  
 anchor, positive, negative = y\_pred[:,:int(total\_lenght\*1/3)],y\_pred[:,int(total\_lenght\*1/3):int(total\_lenght\*2/3)], y\_pred[:,int(total\_lenght\*2/3):]  
 pos\_dist = tf.reduce\_sum(tf.square(anchor - positive), axis=-1)  
 neg\_dist = tf.reduce\_sum(tf.square(anchor - negative), axis=-1)  
 basic\_loss = pos\_dist - neg\_dist + alpha  
 loss = tf.reduce\_sum(tf.maximum(basic\_loss, 0.0))  
 return loss  
  
anchor\_input = Input((28,28), name='anchor\_input')  
positive\_input = Input((28,28), name='positive\_input')  
negative\_input = Input((28,28), name='negative\_input')  
anchor\_embedding = base\_model(anchor\_input)  
positive\_embedding = base\_model(positive\_input)  
negative\_embedding = base\_model(negative\_input)  
merged\_vector = tf.keras.layers.concatenate([anchor\_embedding,  
positive\_embedding, negative\_embedding], axis=-1)  
snn = Model(inputs=[anchor\_input,positive\_input, negative\_input],  
outputs=merged\_vector)  
snn.compile(optimizer=Adam(0.0001), loss=triplet\_loss)  
  
  
def get\_triplets(data, labels):  
 anchor\_images = np.array([]).reshape((-1, 28, 28))  
 positive\_images = np.array([]).reshape((-1, 28, 28))  
 negative\_images = np.array([]).reshape((-1, 28, 28))  
  
 unique\_labels = np.unique(labels)  
 for label in unique\_labels:  
 same\_class = data[labels == label]  
 diff\_class = data[labels != label]  
  
 for anchor\_img in same\_class:  
 for \_ in range(2): # Just take two triplets per anchor image for simplicity  
 positive\_img = same\_class[np.random.randint(len(same\_class))]  
 negative\_img = diff\_class[np.random.randint(len(diff\_class))]  
 anchor\_images = np.vstack([anchor\_images, np.expand\_dims(anchor\_img, axis=0)])  
 positive\_images = np.vstack([positive\_images, np.expand\_dims(positive\_img, axis=0)])  
 negative\_images = np.vstack([negative\_images, np.expand\_dims(negative\_img, axis=0)])  
 return [anchor\_images, positive\_images, negative\_images]  
  
  
triplet\_train = get\_triplets(x\_train, y\_train)  
triplet\_test = get\_triplets(x\_test, y\_test)  
history = snn.fit(triplet\_train, np.zeros(len(triplet\_train[0])), epochs=10,batch\_size=32, validation\_data=(triplet\_test, np.zeros(len(triplet\_test[0]))))  
  
plt.plot(history.history['loss'])  
plt.plot(history.history['val\_loss'])  
plt.title('Model loss')  
plt.ylabel('Loss')  
plt.xlabel('Epoch')  
plt.legend(['Train', 'Test'], loc='upper right')  
plt.show()  
anchor\_embeddings = base\_model.predict(triplet\_test[0])  
positive\_embeddings = base\_model.predict(triplet\_test[1])  
negative\_embeddings = base\_model.predict(triplet\_test[2])  
  
# Computing the distances  
pos\_dist = np.sum(np.square(anchor\_embeddings - positive\_embeddings), axis=1)  
neg\_dist = np.sum(np.square(anchor\_embeddings - negative\_embeddings), axis=1)  
  
# Accuracy computation  
accuracy = np.sum(pos\_dist < neg\_dist) / len(pos\_dist)  
print(f"Accuracy: {accuracy \* 100:.2f}%")

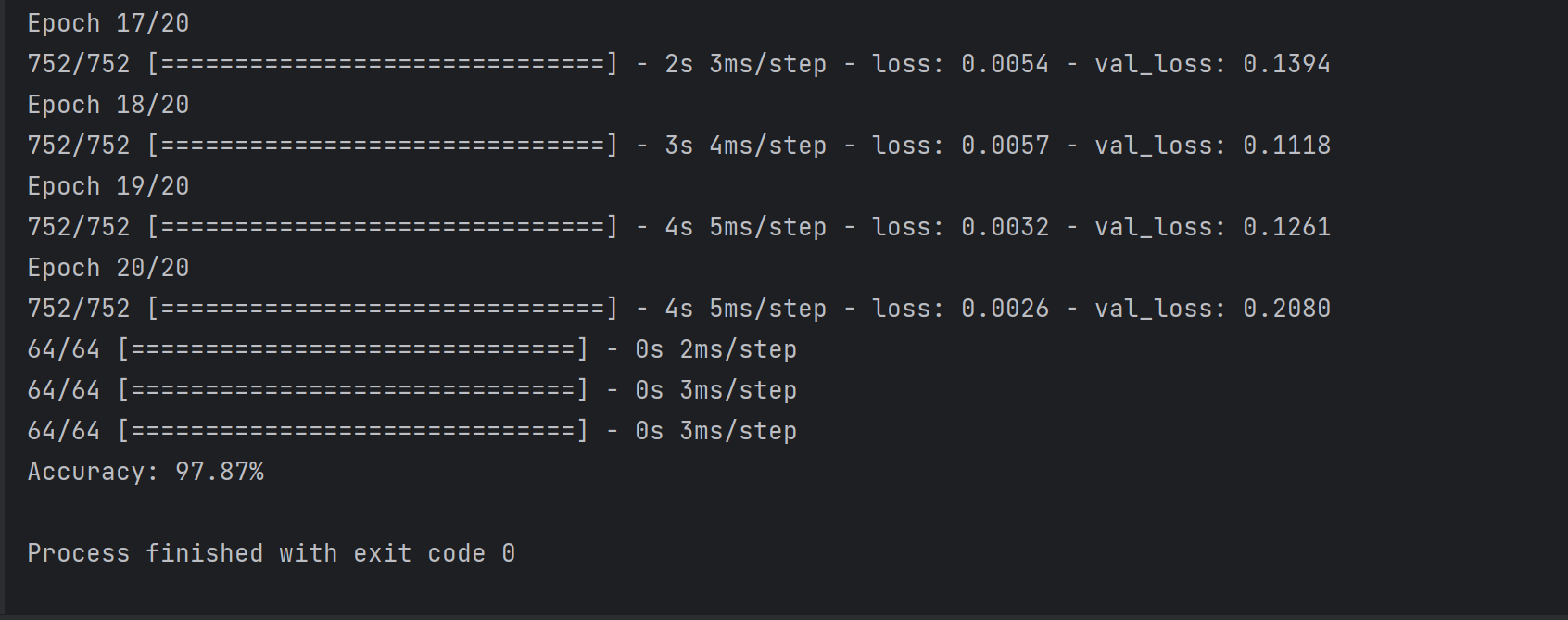


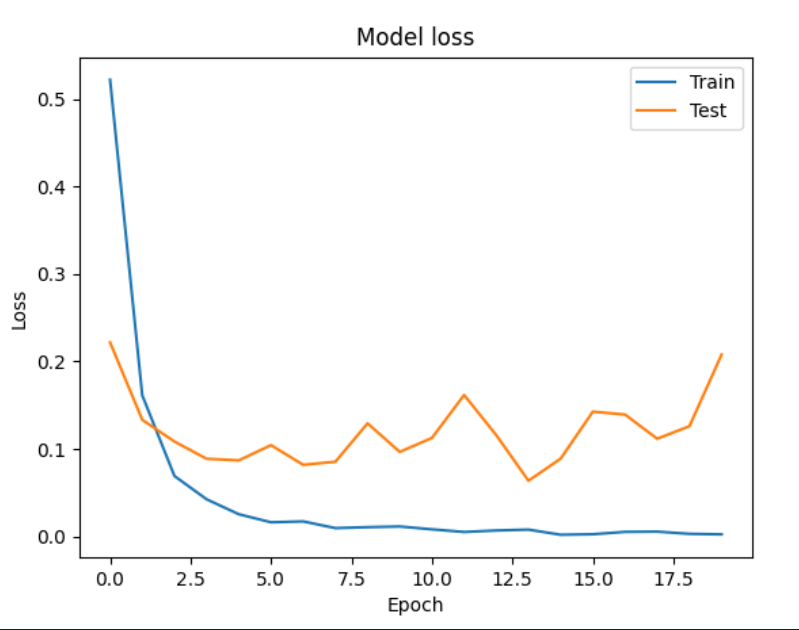


Зменшимо кількість batch\_size

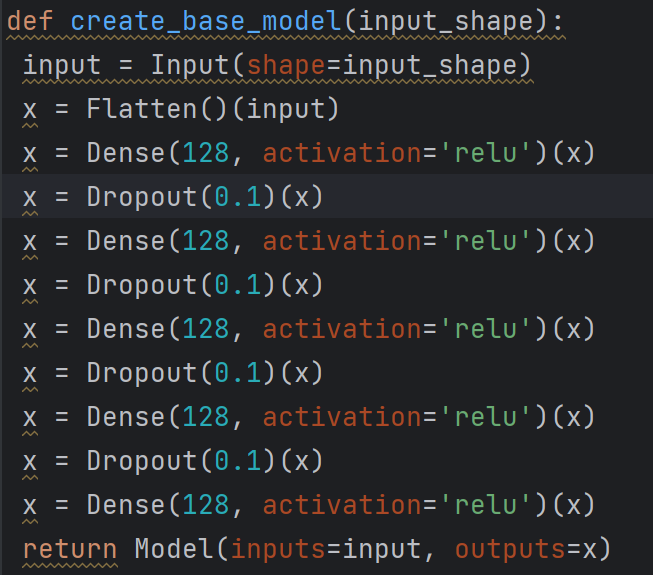


Результат



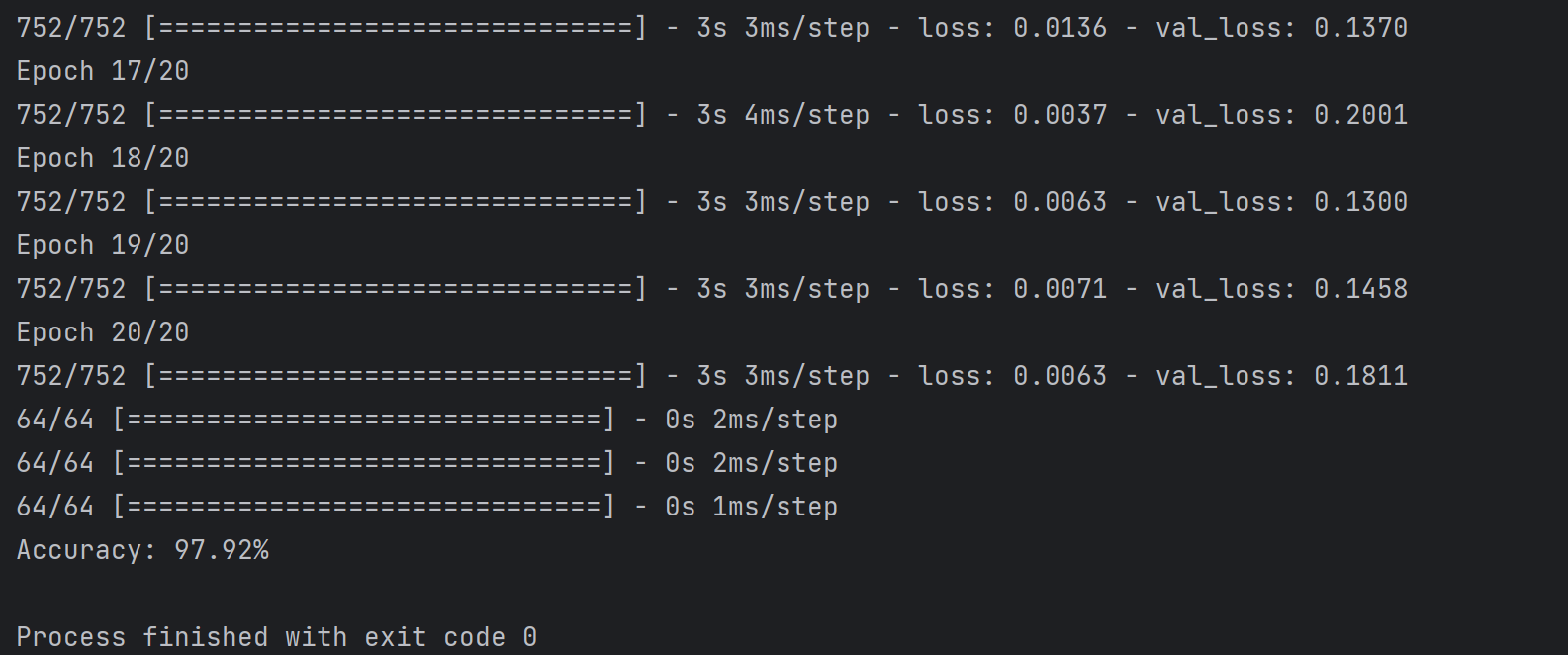


Збільшимо кількість шарів та регуляції dropout



результати





Розглянувши результати сіамської на основі triplet loss та сіамська мережана основі contrastive loss, можна помітити, що зі зміної в структурі архітектури сіамські мережі покращуються або погіршуються результати**.** Сontrastive loss штрафує мережу, якщо відстань між двома вхідними даними велика, але не штрафує мережу, якщо відстань між двома вхідними даними мала. Це означає, що мережа може навчитися просто мінімізувати відстань між двома вхідними даними, навіть якщо ці вхідні дані не є схожими. Triplet loss штрафує мережу, якщо відстань між позитивним прикладом і негативним прикладом велика. Це означає, що мережа повинна навчитися збільшувати відстань між позитивними прикладами і негативними прикладами, навіть якщо відстань між позитивними прикладами вже невелика. Це допомагає мережі навчитися створювати уявлення, які краще відображають подібність між двома вхідними даними.Для покращення результатів рекомендую: Розглянути використання глибших архітектур або архітектур для вилучення більшеї кількості інформації з зображення, змінити гіперпараметри, такі як розмір пакета (batch\_size), швидкість навчання (learning\_rate) і кількість епох, для пошуку оптимальних значень, додати додатковішарів агрегації або шарів взвешування для об'єднання ознак, витягнутих з кількох гілок сіамської мережі. Додайти шари нормалізації ( Batch Normalization ) щоб поліпшити стабільність навчання та збільшити швидкість збіжності. Використати методи регуляризації, такі як Dropout або L2 регуляризація, для попередження перенавчання.

**Контрольні питання**

1. **Що таке сіамська мережа та які її основні застосування?**

Сіамська мережа – це архітектура нейронних мереж, яка використовується, головним чином, для вирішення задачі порівняння або визначення схожості між двома входами. Вона складається з двох ідентичних підмереж, які розділяють ті ж самі параметри та ваги. Обидві підмережі приймають різні вхідні дані та виводять ознаки, які потім порівнюються.

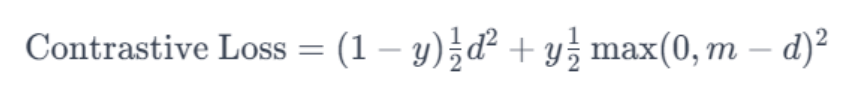
Основне застосування сіамських мереж - у завданнях впізнавання обличчя та об'єктів, а також у сферах, де потрібно визначити схожість між двома об'єктами або даними. Наприклад, їх можна використовувати для розпізнавання осіб, аутентифікації, відстеження об'єктів на відео, розпізнавання підписів тощо. Сіамські мережі дозволяють вчитися представленню об'єктів, яке є робочим для завдань порівняння та розпізнавання.

1. **Яка основна відмінність архітектури сіамської мережі від традиційних нейронних мереж?**

Основна відмінність архітектури сіамської мережі від традиційних нейронних мереж полягає в тому, що вона має два входи, а не один. Ці два входи можуть бути зображеннями, відео, звуком або будь-якими іншими даними, які можна порівняти. Сіамська мережа навчається знаходити подібність або різницю між двома входами. У традиційній нейронній мережі є один вхід, а вихід мережі є функцією від цього входу. Наприклад, нейронна мережа може бути навчена класифікувати зображення. Вихід мережі буде вказувати на те, до якої категорії належить зображення.

1. **Опишіть принцип роботи contrastive loss.**

Визначення: Contrastive loss використовується для навчання сіамської мережі розрізнювати позитивні та негативні пари прикладів. Математичний опис:

****

Де: y – мітка подібності (1, якщо пара є позитивною і 0, якщо вона негативна).

d – евклідова відстань між ознаками виведеними з двох підмереж.

m – порогове значення, яке визначає, якою повинна бути максимальна відстань між позитивними парами.

Принцип роботи contrastive loss можна описати наступним чином:

1. Введення пар об'єктів: Для навчання моделі за допомогою contrastive loss вводяться пари об'єктів. Кожна пара складається з позитивного (схожого) та негативного (відмінного) прикладу.
2. Формування репрезентацій: Модель обчислює репрезентації для об'єктів у кожній парі. Ці репрезентації можуть бути отримані за допомогою одного зі шарів мережі, наприклад, після підсумовування гілок сіамської мережі.
3. Визначення втрати: Для кожної пари об'єктів визначається втрата contrastive loss. Зазвичай вона включає дві складові: одна для позитивного прикладу, яка вказує, що репрезентації схожих об'єктів повинні бути близькими, і інша для негативного прикладу, яка вказує, що репрезентації відмінних об'єктів повинні бути віддаленими. Типовою функцією втрати є Euclidean distance або cosine similarity між репрезентаціями.
4. Мінімізація втрати: Модель намагається мінімізувати втрату contrastive loss під час навчання. Це сприяє вивченню репрезентацій так, щоб об'єкти однієї класифікаційної категорії або схожі об'єкти мали близькі репрезентації, тоді як об'єкти різних категорій були віддаленими в просторі репрезентацій.
5. **У чому полягає принцип роботи triplet loss та як він відрізняється від contrastive loss?**

Кількість зразків:

Triplet loss працює з трьома зразками: anchor, positive та negative.

Contrastive loss працює з парами зразків, визначаючи їх як positive чи negative.

Мета:

Triplet loss спрямований на максимізацію відстані між anchor та negative та мінімізацію відстані між anchor та positive.

Contrastive loss спрямований на мінімізацію відстані між positive та максимізацію відстані між negative.

Обчислення:

Triplet loss використовує трійки зразків і порівнює відстані між ними.

Contrastive loss використовує парами зразків та враховує відстані між ними залежно від того, чи є вони positive чи negative.

1. **Які основні етапи підготовки даних для навчання сіамської мережі?**

Підготовка даних для навчання сіамської мережі включає кілька основних етапів:

1. Збір даних: Зібрати дані, які будуть використовуватися для тренування. Це можуть бути зображення, текстові дані або інші типи інформації, залежно від призначення мережі.
2. Попередня обробка даних: Цей етап включає очистку даних, видалення шуму, масштабування або нормалізацію даних для забезпечення стабільності та оптимальності під час тренування.
3. Розбиття на пари даних: У випадку сіамської мережі потрібно розбити дані на пари, які будуть подаватися на вхід мережі. Це може бути пара зображень для порівняння або пара текстових рядків для визначення схожості.
4. Створення меток: Для кожної пари даних потрібно мати відповідні мітки, які вказують на їхню схожість або відмінність. Наприклад, якщо ми тренуємо сіамську мережу для розпізнавання облич, ми можемо мати пари зображень з мітками "схожі" або "різні".
5. Створення датасету: На цьому етапі формується датасет, де кожна пара даних має свою мітку. Датасет використовується для навчання моделі.
6. Аугментація даних (опціонально): Іноді використовують методи аугментації, які дозволяють розширити набір даних шляхом зміни наявних зображень (обертання, зміна розміру, зміщення тощо) для покращення роботи моделі та запобігання перенавчанню.
7. Перетворення даних: Підготовка даних у відповідний формат для подачі їх до сіамської мережі, що може включати векторизацію тексту, попередню обробку зображень, або інші операції, необхідні для конкретного типу даних.
8. Навчання: Навчання моделі на підготованих даних, після чого використовується валідаційний набір для оцінки продуктивності моделі та виявлення перенавчання.