به نام خدا

درس مبانی هوش محاسباتی

تمرین سری اول

مدرس درس:
جناب آقای دکتر مزینی

تهیه شده توسط:
الناز رضایی ۹۸۴۱۱۳۸۷

تاریخ ارسال: ۱۴۰۱/۰۸/۱۷

سوال ۱:

در مورد تفاوت stochastic، batch و mini-batch تحقیق کنید. سپس، روش SGD را با GD مقایسه کنید و مشکلات SGD را ذکر کنید و بیان کنید چگونه با استفاده از Momentum می‌توان این روش را بهبود بخشید.

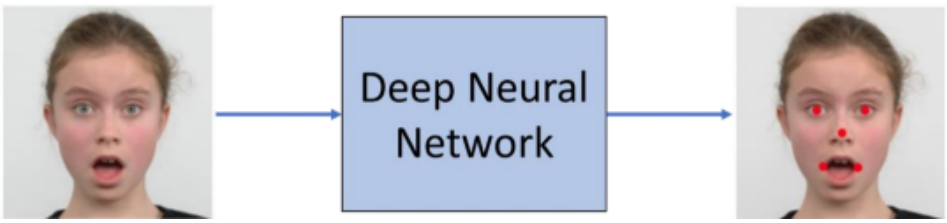
پاسخ ۱:

در روش mini-batch، ما دیتاست خود را به چند دسته تقسیم می‌کنیم و وزن‌ها و بایاس، در پایان هر دسته آپدیت می‌شوند. بنابراین، با استفاده از این روش، سریع‌تر به سمت global minimum حرکت می‌کنیم. در روش batch، هر دسته معادل کل دیتاست است. از این روش معمولا برای دیتاست‌هایی با تعداد داده کمتر از ۲۰۰۰ استفاده می‌شود. در روش stochastic، هر دسته برابر با یک نمونه از دیتاست است. مشکل این روش، از دست دادن مزیت به‌دست آمده از بردارسازی است ولی از طرفی سریع‌تر همگرا می‌شود. به طور خلاصه، با وجود دقیق‌تر بودن روش batch، روش stochastic سریع‌تر می‌باشد. قابل ذکر است تنها کد روش mini-batch را می‌توان برای هر سه روش استفاده کرد و فقط سایز آن را برای روش stochastic برابر ۱، و برای batch، برابر با تعداد کل نمونه‌ها قرار دهیم. با توجه به توضیحات داده شده، تفاوت اصلی بین این روش‌ها، تعداد نمونه‌های استفاده شده برای هر epoch، زمان و مقدار تلاش لازم برای رسیدن به global minimum تابع هزینه می‌باشد.

در روش GD، پارامترها را فقط بعد از هر epoch به روزرسانی می‌کند. یعنی پس از محاسبه مشتقات برای همه مشاهدات، پارامترها آپدیت می‌شوند؛ اما در روش SGD، پارامترها را به‌ازای هر مشاهده به‌روزرسانی می‌کند که منجر به تعداد بیشتری آپدیت می‌شود. بنابراین این روش سریع‌تر بوده و به تصمیم‌گیری سریع‌تر کمک می‌کند. از معایب این روش، می‌توان به تقریبی بودن گرادیان، حرکت نویزی، گیر افتادن در نقاط local minimum و متوقف شدن در جاهایی که مشتق صفر است (مانند نقاط زینی). یکی از روش‌های معرفی شده برای بهبود SGD، روش SGD + Momentum می‌باشد. برای حل مشکل گیر کردن در local minimum و متوقف شدن در نقاط زینی، سرعت را هم در محاسبات تاثیر می‌دهیم. در واقع در این روش، ابتدا گرادیان را حساب کرده و سپس سرعت را محاسبه می‌کنیم که می‌شود ضریبی از سرعت قبلی که همان اصطکاک است، به‌علاوه شیب در نقطه‌ای که هستیم. سپس فرآیند بهینه‌سازی و آپدیت قبلی را انجام می‌دهیم. بنابراین در اینجا سرعت را به‌نوعی تخمین زدیم که می‌شود میانگین گرادیان‌های بقیه. با توجه به توضیحات داده شده، در این روش هنگام رسیدن به مینیمم محلی، یا سرعتمان زیاد است و از روی تپه بعدی می‌پرد، یا اگر سرعتش کم باشد، به تدریج در همان نقطه همگرا می‌شود و بنابراین از local minimaهای کوچک، به‌آسانی رد می‌شود. همچنین از نقاط زینی هم به سادگی عبور می‌کند. همچنین این روش باعث کاهش حرکت نویزی هم می‌شود (به دلیل شتاب گرفتن). نکته دیگری که وجود دارد، خیلی وقت‌ها SGD نمی‌تواند همگرا شود، اما SGD + Momentum حتما همگرا می‌شود.

سوال ۲:

یافتن مکان نقاط مهم در چهره یکی از مراحل بسیار مهم در الگوریتم‌های تحلیل چهره است. همانطور که در شکل زیر نشان داده شده است، ورودی چنین شبکه‌ای یک تصویر برش خورده از چهره است و خروجی آن تخمینی از مختصات نقاط مورد نظر است. در این مثال، خروجی مختصات ۵ نقطه شامل مرکز دو چشم، مرکز بینی و گوشه‌های دهان بوده است که برای نمایش بهتر، بر روی تصاویر اصلی رسم شده‌اند (خروجی شبکه یک تصویر نیست، بلکه خروجی مختصات نقاط است و برای نمایش بهتر بر روی تصویر رسم شده‌اند) .



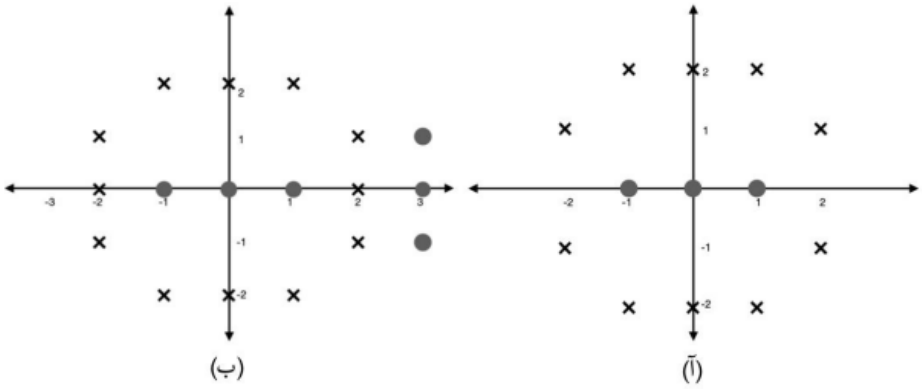
- الف) اگر بخواهیم چنین شبکه‌ای طراحی کنیم، در لایه آخر شبکه چند نورون باید داشته باشیم؟ به نظر شما بهتر است از چه تابع فعال‌سازی در لایه آخر استفاده کنیم؟ تابع ضرر مناسب برای حل این مسئله به نظر شما چیست؟ لطفا پاسخ‌های خود را به جزئیات توضیح دهید.
- ب) این کد را بررسی کنید و مشخص کنید در آن برای حل مسئله بالا از چه تابع فعال‌سازی و از چه تابع ضرری استفاده کرده است.

پاسخ ۲:

- الف) تعداد نورون‌ها در لایه آخر، برابر با ۱۰ می‌باشد. چرا که هر نقطه کلیدی، یک x و یک y دارد و برای اینکه ما مختصات این نقاط را به دست بیاوریم، به ۱۰ نورون نیاز داریم تا x و y این نقاط را به ما بدهند. activation function مورد استفاده در این سوال نیز بهتر است sigmoid باشد تا عددی بین ۰ و ۱ که همان احتمال کلیدی بودن یا نبودن یک پیکسل است را به ما بدهد. همچنین بهتر است از mse برای تابع ضرر استفاده شود تا پیش‌بینی پرت نداشته باشیم.
- ب) در این کد، از تابع فعال‌سازی sigmoid و از تابع ضرر mse with don't care استفاده می‌کنیم. این تابع ضرر میانگین فاصله اقلیدسی بین نقاط حقیقی و نقاط پیش‌بینی‌شده را محاسبه می‌کند. همچنین نقاطی که توسط انسان قابل یافتن نیستند با مختصات (۰، ۰) در مجموعه داده نشان داده می شوند. این نقاط در محاسبه تابع ضرر حذف می شوند.

سوال ۳:

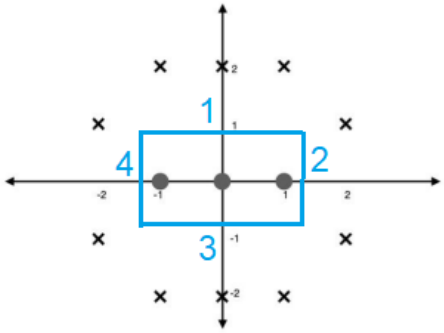
یک پرسپترون تنها می‌تواند داده‌هایی که به صورت خطی قابل جدا کردن هستند را دسته بندی کند. توضیح دهید که Madaline چگونه مسئله‌ی دسته‌بندی را برای داده‌هایی که غیر خطی هستند حل می‌کند. آیا می‌توان دیاگرام‌های زیر را توسط Madaline دسته‌بندی کرد؟ توضیحات لازم را ارائه دهید و در صورت امکان پذیری دسته بندی، معماری شبکه عصبی خود را شرح دهید.



پاسخ ۳:

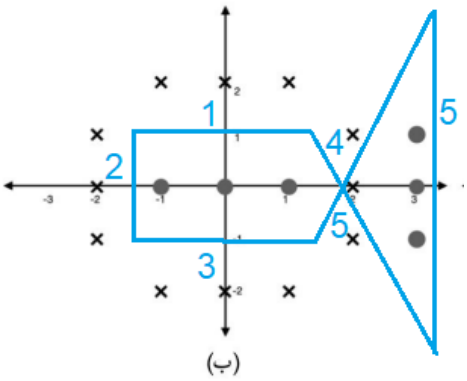
در مدل Madaline، با استفاده از چند Adaline به صورت Parallel می‌توان مسائل غیر خطی را نیز تفکیک کرد. به این‌صورت که هر Adaline، یک خط تولید می‌کند و با تقاطع و اشتراک‌گیری این خطوط، می‌توان یک ناحیه غیر خطی را یاد گرفته و دسته‌بندی غیر خطی داشته باشیم؛ البته شرط لازم در این روش، separable بودن نواحی است.

دیاگرام آ را می‌توان با استفاده از ۴ خط ۱، ۲، ۳ و ۴ که هر کدام از خطوط بیانگر یک Adaline می‌باشد، تفکیک کرد. یعنی در واقع شبکه Madaline ما، از ۴ Adaline که به صورت Parallel هستند و یک واحد And کننده برای پیدا کردن ناحیه درونی، تشکیل شده است. نتیجه دسته‌بندی دیاگرام آ با استفاده از Madaline، در تصویر زیر آورده شده است.



(آ)

در مورد دیاگرام ب، امکان تفکیک پذیری آن وجود ندارد؛ زیرا نمی‌توانیم convexای پیدا کنیم که بتواند دو ناحیه را کامل از هم جدا کند. برای مثال، در شکل زیر، یک مدل پیشنهاد داده شده است که همانطور که مشخص است، باز هم نتوانستیم آن را به طور کامل تفکیک کنیم و یک ضربدر، درون داده‌های دایره‌ای قرار گرفته است.



(ب)

سوال ۴:

به سوالات زیر پاسخ دهید.

- الف) به نظر شما قابلیت تعمیم در کدام یک از شبکه های عصبیAdaline، Perceptron، Madaline و MLP بیشتر و در کدام‌یک کمتر است؟ توضیح دهید.
- ب) چه زمانی می‌گوییم شبکه دچار Overfit شده است؟ دلایل مختلف آن را توضیح دهید.
- ج) چه روش‌هایی برای جلوگیری و حل مشکل Overfit در شبکه‌های پرسپترون چند لایه وجود دارد؟

پاسخ ۴:

- الف) تفاوت اصلی بین Perceptrone و Adaline، این است که Perceptrone، یک binary response گرفته و سپس error را محاسبه کرده و از آن برای آپدیت کردن وزن‌ها، استفاده می‌کند؛ اما Adaline، از continous response استفاده کرده و وزن‌ها را آپدیت می‌کند. این ویژگی Adaline، باعث می‌شود آپدیت های آن قبل از تعیین شدن threshold، بیشتر شبیه ارور واقعی باشد؛ در نتیجه به مدل ما اجازه می‌دهد سریع‌تر همگرا شود و در نتیجه قابلیت generalization آن بیشتر باشد. همچنین Madaline، از AND چند Adaline به دست می‌آید و قابلیت حل مسائل غیر خطی را نیز دارد، بنابراین قابلیت تعمیم Madaline هم از Adaline بیشتر است؛ زیرا Adaline فقط برای مسائل تفکیک پذیر خطی قابل کاربرد هست، به علاوه تعداد لایه‌ها در Madaline از Adaline بیشتر می‌باشد. در MLP، به دلیل توانایی نگاشت غیر خطی و در نتیجه قابل استفاده بودن آن در مسائل classification پیچیده‌تر، قابلیت تعمیم آن، از همه بیشتر است. بنابراین اگر بخواهیم این مدل‌ها را براساس قابلیت تعمیم‌پذیری‌شان طبقه‌بندی کنیم، به شرح زیر می‌باشد:

  $\text{MLP} > \text{Madaline} > \text{Adaline} > \text{Perceptrone}$

  نکته: در این بخش از سوال، از لینک‌های زیر استفاده شده است.

  `https://datascience.stackexchange.com/questions/36368/what\`
  `-is-the-difference-between-perceptron-and-adaline#:~:text=`
  `The%20main%20difference%20between%20the,the%20binarized%20\`
  `output%20is%20produced).`
  `https://experts.umn.edu/en/publications/factors-controlling\`
  `-generalization-ability-of-mlp-networks`

- ب) زمانی که مدل ما روی داده‌های train خوب عمل کند اما بر داده‌های validation عملکرد خوبی نداشته باشد، اصطلاحا می‌گوییم مدل ما دچار overfitting شده است. در این حالت، شبکه یک نمونه را خوب یاد می‌گیرد (با خطای خیلی کم) و روی چند نقطه fit می‌شود و قابل تعمیم نیست. یعنی در واقع مدل روی چند ویژگی داده‌های train که معرف خوبی از کل داده‌ها نیستند، fit می‌شود. از علل رخ دادن overfitting، می‌توان به تعداد

بالای نورون، کم بودن data، واریانس بالای مدل، پیچیده بودن مدل و cleaned نبودن و نویزی بودن dataی مورد استفاده برای train اشاره کرد.

- ج) برای حل این مشکل، می‌توانیم از روش‌های کاهش تعداد نورون، داشتن testset (validation)، خراب کردن وزن‌ها بعد از هر iteration (باعث تغییر fit شدن می‌شود)، چک کردن overfitting (چک کنیم ببینیم چه زمانی overfitting رخ می‌دهد، اگر دیدیم training خوب پیش می‌رود و test بد پیش می‌رود، یعنی overfitting رخ داده است)، افزایش data (اگر data زیاد و معنی‌دار باشد، خوب است اما تعداد کم data، قطعا احتمال overfitting را افزایش می‌دهد)، اضافه کردن نویز به داده ورودی، regularization، متوقف کردن training در زمان کوتاه‌تر و ساده کردن مدل اشاره کرد.

سوال ۵:

با استفاده از کتابخانه NumPy یک پرسپترون طراحی کنید تا تابع NOR را با استفاده از روش Descent Gradient یاد بگیرد. در نهایت برای اطمینان از درستی آموزش پرسپترون، نموداری شامل خط متمایز کننده ای که توسط پرسپترون آموخته شده است و نقاط نشان دهنده تابع فوق به همراه برچسپ‌هایشان رسم نمایید.

پاسخ ۵:

در ابتدا توابعی پایه‌ای که برای حل این سوال نیاز داریم را تعریف می‌کنیم. تابع فعال‌سازی استفاده شده در این مثال، sigmoid بوده که یک تابع با همان نام در کدمان تعریف می‌کنیم. همچنین تابع محاسبه ضررمان، cross entropy error می‌باشد که در کدمان با نام CEE تعریف شده است.
سپس چون برای فاز back-propagation به مشتق تابع ارور و سیگموید نیاز داریم، تابعی برای محاسبه این دو نیز می‌نویسیم که با نام dCEE و dsigmoid در کدمان وجود دارند. توابعی که تا اینجا توضیح داده شد، در قطعه کد زیر آورده شده‌اند.

Define basic functions

```
import numpy as np
import matplotlib.pyplot as plt

# sigmoid(z) = 1 \ 1 + e^-z
def sigmoid(z):
  return 1 / (1 + np.exp(-z))

# E (y,  y_hat) = -y ln(y_hat) - (1-y) ln(1-y_hat)
def CEE(y_predicted, y):
  if y == 1:
    return -1 * np.log(y_predicted)
  else:
    return -1 * np.log(1 - y_predicted)

# dsig(z) / dz = sig(z)(1 - sig(z))
def dsigmoid(z):
  return z * (1 - z)

# dE / dy_predicted = -(y / y_predictes) + ((1-y) / (1 - y_predicted))
def dCEE(y_predicted, y):
  if y == 1:
    return -1 / y_predicted
  else:
    return 1 / (1 - y_predicted)
```

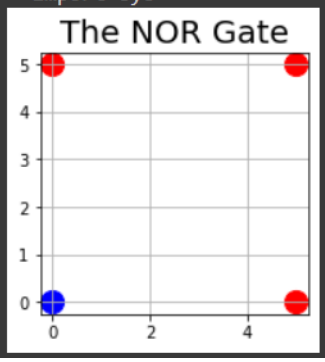
توابع اولیه‌ی پیاده‌سازی شده

سپس در این بخش، یک دیتاست تعریف کرده و آن را plot می‌کنیم. می‌دانیم که گیت NOR در صورت مثبت بودن همه ورودی‌ها، ۰ و در غیر این‌صورت ۱ می‌شود. کد زده شده در این بخش به همراه نتیجه حاصله در شکل زیر آورده شده است.

Plot NOR

```
# dataset
x = np.array([[0, 0], [0, 5], [5, 0], [5, 5]])
y = np.array([1, 0, 0, 0])
# Plot
fig = plt.figure(figsize=(3, 3))
plt.title('The NOR Gate', fontsize=20)
ax = fig.add_subplot(111)
# red: class 0, blue: class 1.
ax.scatter(0, 0, s=200, c='b', label="Class 1")
ax.scatter(0, 5, s=200, c='r', label="Class 0")
ax.scatter(5, 0, s=200, c='r', label="Class 0")
ax.scatter(5, 5, s=200, c='r', label="Class 0")
plt.grid()
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:7: Matplotl
  import sys
```



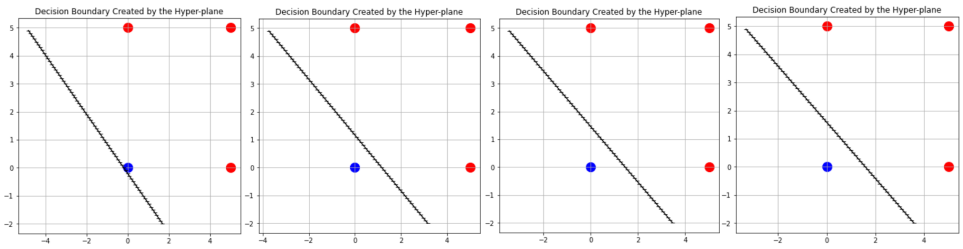
معرفی dataset و plot کردن آن

در ادامه، وزن‌ها را مقدار دهی اولیه می‌کنیم و فاز training را انجام می‌دهیم. در این مثال، learning rate برابر با 0.01 و تعداد epochها، 2000 فرض شده است.

```
w0 = np.random.uniform(low=-0.01, high=0.01, size=(1,))
w1 = np.random.uniform(low=-0.01, high=0.01, size=(1,))
w2 = np.random.uniform(low=-0.01, high=0.01, size=(1,))
average_CEE = []
for epoch in range(2000):
  errors = []
  seeds = np.arange(x.shape[0])
  np.random.shuffle(seeds)
  for seed in seeds:
    input = x[seed]
    # z = sigma(wi * xi) + w0
    z = w1 * input[0] + w2 * input[1] + w0
    y_predicted = sigmoid(z)
    errors.append(CEE(y_predicted, y[seed]))
    dE_dw0 = dCEE(y_predicted, y[seed]) * dsigmoid(y_predicted)
    dE_dw1 = dCEE(y_predicted, y[seed]) * dsigmoid(y_predicted) * input[0]
    dE_dw2 = dCEE(y_predicted, y[seed]) * dsigmoid(y_predicted) * input[1]
    w0 = w0 - 0.01 * dE_dw0
    w1 = w1 - 0.01 * dE_dw1
    w2 = w2 - 0.01 * dE_dw2
  if epoch % 500 == 0:
    fig = plt.figure(figsize=(6, 6))
    plt.title('The NOR Gate', fontsize=20)
    ax = fig.add_subplot(111)
    ax.scatter(0, 0, s=200, c='b', label="Class 1")
    ax.scatter(0, 5, s=200, c='r', label="Class 0")
    ax.scatter(5, 0, s=200, c='r', label="Class 0")
    ax.scatter(5, 5, s=200, c='r', label="Class 0")
    plt.title('Decision Boundary Created by the Hyper-plane')
    x1 = np.arange(-2, 5, 0.1)
    w1 * x[0] + w2 * x[1] + w0
    x2 = (-w1 / w2) * x1 - (w0 / w2)
    plt.grid()
    plt.plot(x2, x1, '-k', marker='_', label="DB")
    plt.xlabel('x1', fontsize=20)
    plt.ylabel('x2', fontsize=20)
```

فاز trainig

همچنین نتایج این بخش را بعد از هر ۵۰۰ epoch، نمایش می‌دهیم. نتایج این بخش، در تصاویر زیر به ترتیب از چپ به راست آورده شده‌اند.



همانطور که در تصویر بالا نیز مشخص است، پس از ۱۰۰۰ epoch در واقع perceptrone ما به خوبی عمل می‌کند و مرز را به درستی تشخیص می‌دهد. در تصاویر بعد، فقط دقت و میزان بهینه بودن مرزمان بیشتر می‌شود.
نکته: در حل این سوال، از لینک زیر کمک گرفته شد.

https://www.mldawn.com/train-a-perceptron-to-learn-the-and-\
gate-from-scratch-in-python/

سوال ۶:

با استفاده از کتابخانه Keras یک شبکه‌ی پرسپترون چند لایه طراحی کنید تا عملیات دسته‌بندی را برروی دیتاست MNIST ، که شامل تصاویر دست نویس اعداد ۰ تا ۹ است، انجام دهد. نمودارهای دقت و خطا را برای هر iteration رسم کنید.

پاسخ ۶:

در ابتدا کتابخانه‌های مورد نظر خود را import کرده و سپس دیتاست موردنظر را load می‌کنیم. در ادامه داده‌ها را normalized می‌کنیم (بخشی که بر ۲۵۵ تقسیم کردیم).

```
from tensorflow import keras
import numpy as np
from matplotlib import pyplot as plt
from keras.utils import np_utils
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
```

Load dataset

```
[5] (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras
11490434/11490434 [==============================] - 1s 0us/step
```

```
[6] x_train = x_train.astype(float) / 255.
    x_test  = x_test.astype(float) / 255.
```

همچنین با استفاده از to_categorical، ورودی y را به صورت یک vector ۰ و ۱ای در می‌آوریم تا بتوانیم در شبکه چند کلاسه استفاده کنیم.

```
x_train_flatten = x_train.reshape(x_train.shape[0], -1)
y_train_flatten = y_train.reshape(x_train.shape[0], -1)
x_test_flatten = x_test.reshape(x_test.shape[0], -1)
y_test_flatten = y_test.reshape(x_test.shape[0], -1)

y_train_categorical = np_utils.to_categorical(y_train_flatten, 10)
y_test_categorical = np_utils.to_categorical(y_test_flatten, 10)
```

مدل خود را به صورت sequential تعریف می‌کنیم.

Sequential model

```
model_temp = Sequential()

# Input Layer
model_temp.add(layers.Dense(128, input_shape=(x_train.shape[1] * x_train.shape[2],)))
model_temp.add(layers.Activation('relu'))

# Hidden Layer
model_temp.add(layers.Dense(128))
model_temp.add(layers.Activation('relu'))

# Output Layer
model_temp.add(layers.Dense(10))
model_temp.add(layers.Activation('softmax'))
```

یک summary از مدل خود می‌گیریم.

```
model_temp.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               100480

 activation (Activation)     (None, 128)               0

 dense_1 (Dense)             (None, 128)               16512

 activation_1 (Activation)   (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

 activation_2 (Activation)   (None, 10)                0

=================================================================
Total params: 118,282
Trainable params: 118,282
Non-trainable params: 0
_________________________________________________________________
```

در این مثال، چون تصاویر دیتاست 28 * 28 می‌باشد، لایه ورودی به تعداد featureها (28 * 28) می‌باشد. لایه آخر نیز چون تعداد کلاس‌هایمان ۱۰ تا می‌باشد (اعداد ۰ تا ۹)، دارای ۱۰ نورون می‌باشد. قابل ذکر است تمام لایه‌ها در این مثال Dense و Fully connected هستند. activation function نیز در لایه ورودی و لایه‌های پنهانی، Relu، و در لایه خروجی softmax می‌باشد. به دلیل چند کلاسه بودن سوال نیز از تابع خطا cross entropy استفاده می‌کنیم.

Compile model

```
[14] model_temp.compile(optimizer = SGD(), loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

چون تعداد داده‌ها در این دیتاست زیاد است، از mini-batch استفاده کردیم و اندازه داده در هر batch را برابر با ۱۲۸ قرار دادیم تا زمان هر epoch طولانی نشود. همچنین برای اینکه overfitting رخ ندهد، fitvalidation$_{s}plit$ کردن مدل برابر با 0.2 قرار می‌دهیم تا بخشی از داده train را برای hold out کردن استفاده کرده باشیم.

Train model

```
history = model_temp.fit(x_train_flatten, y_train_categorical, batch_size=128,
                         epochs=20, verbose=1, validation_split =0.2)
```

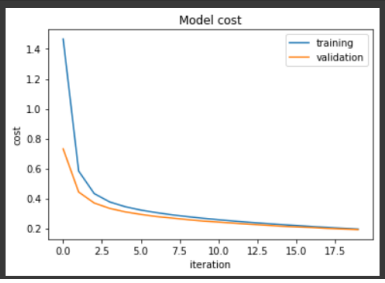
```
Epoch 1/20
375/375 [==============================] - 3s 7ms/step - loss: 1.4661 - accuracy: 0.6259 - val_loss: 0.7311 - val_accuracy: 0.8462
Epoch 2/20
375/375 [==============================] - 3s 7ms/step - loss: 0.5829 - accuracy: 0.8578 - val_loss: 0.4425 - val_accuracy: 0.8892
Epoch 3/20
375/375 [==============================] - 2s 6ms/step - loss: 0.4324 - accuracy: 0.8838 - val_loss: 0.3692 - val_accuracy: 0.9015
Epoch 4/20
375/375 [==============================] - 2s 7ms/step - loss: 0.3762 - accuracy: 0.8956 - val_loss: 0.3332 - val_accuracy: 0.9066
Epoch 5/20
375/375 [==============================] - 2s 6ms/step - loss: 0.3444 - accuracy: 0.9033 - val_loss: 0.3098 - val_accuracy: 0.9121
Epoch 6/20
375/375 [==============================] - 2s 6ms/step - loss: 0.3220 - accuracy: 0.9096 - val_loss: 0.2930 - val_accuracy: 0.9162
Epoch 7/20
375/375 [==============================] - 2s 6ms/step - loss: 0.3049 - accuracy: 0.9135 - val_loss: 0.2787 - val_accuracy: 0.9221
Epoch 8/20
375/375 [==============================] - 2s 7ms/step - loss: 0.2903 - accuracy: 0.9179 - val_loss: 0.2685 - val_accuracy: 0.9227
```

```
Epoch 9/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2781 - accuracy: 0.9209 - val_loss: 0.2580 - val_accuracy: 0.9266
Epoch 10/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2672 - accuracy: 0.9239 - val_loss: 0.2484 - val_accuracy: 0.9292
Epoch 11/20
375/375 [==============================] - 2s 7ms/step - loss: 0.2573 - accuracy: 0.9270 - val_loss: 0.2415 - val_accuracy: 0.9317
Epoch 12/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2484 - accuracy: 0.9297 - val_loss: 0.2342 - val_accuracy: 0.9333
Epoch 13/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2401 - accuracy: 0.9320 - val_loss: 0.2270 - val_accuracy: 0.9348
Epoch 14/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2325 - accuracy: 0.9341 - val_loss: 0.2212 - val_accuracy: 0.9373
Epoch 15/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2251 - accuracy: 0.9359 - val_loss: 0.2142 - val_accuracy: 0.9396
Epoch 16/20
375/375 [==============================] - 4s 10ms/step - loss: 0.2188 - accuracy: 0.9382 - val_loss: 0.2095 - val_accuracy: 0.9405
Epoch 17/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2120 - accuracy: 0.9406 - val_loss: 0.2053 - val_accuracy: 0.9417
Epoch 18/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2061 - accuracy: 0.9412 - val_loss: 0.1993 - val_accuracy: 0.9444
Epoch 19/20
375/375 [==============================] - 2s 6ms/step - loss: 0.2002 - accuracy: 0.9433 - val_loss: 0.1955 - val_accuracy: 0.9457
Epoch 20/20
375/375 [==============================] - 2s 6ms/step - loss: 0.1948 - accuracy: 0.9451 - val_loss: 0.1915 - val_accuracy: 0.9451
```

سپس نمودار دقت و خطا را مطابق شکل زیر رسم می‌کنیم.

Plot loss

```
[17] plt.plot(history.history['loss'])
     plt.plot(history.history['val_loss'])
     plt.title('Model cost')
     plt.ylabel('cost')
     plt.xlabel('iteration')
     plt.legend(['training', 'validation'], loc='upper right')
     plt.show()
```



Plot accuracy

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('accuracy')
plt.xlabel('iteration')
plt.legend(['training', 'validation'], loc='lower right')
plt.show()
```