به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین اول

پرسش‌های ۱ و ۲	نام و نام خانوادگی	محمدمهدی برقی
	رایانامه	mmbarghi@ut.ac.ir
پرسش‌های ۳ و ۴	نام و نام خانوادگی	علی خرم فر
	رایانامه	khoramfar@ut.ac.ir
	تاریخ ارسال پاسخ	۱۴۰۳/۰۱/۲۳

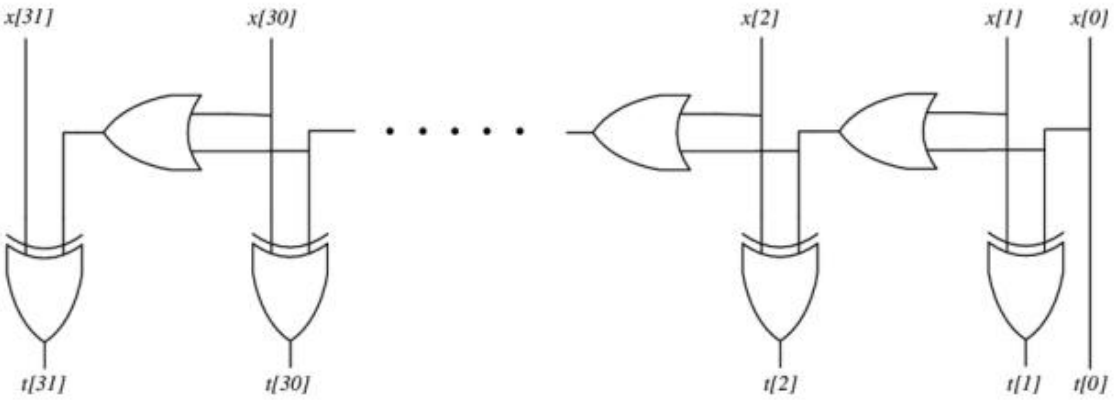
فهرست

فهرست تصاویر

پرسش ۱– Mcculloch Pitts

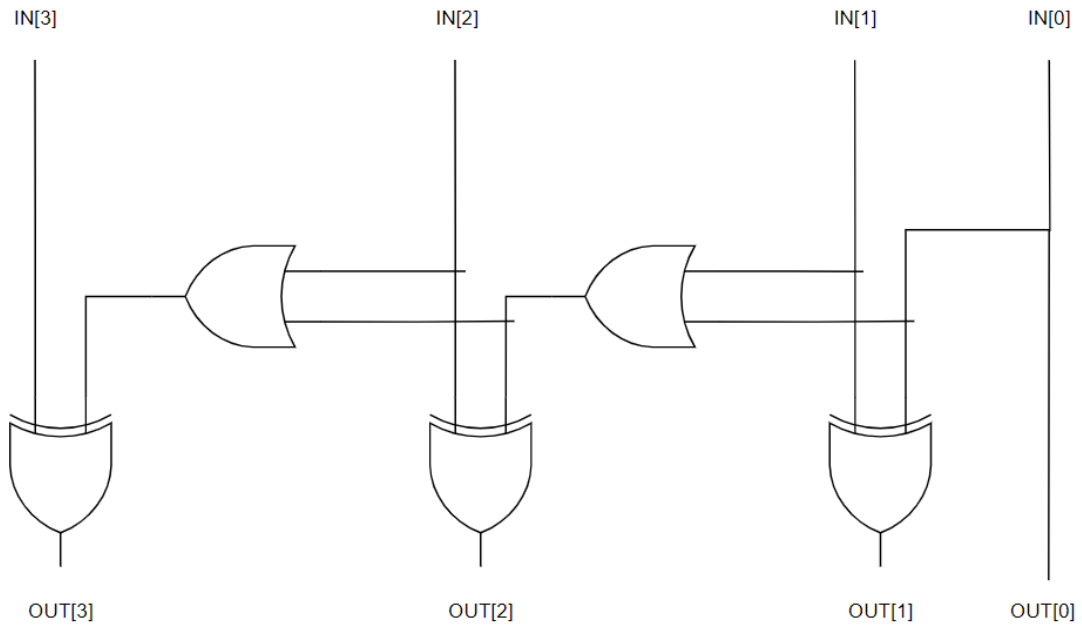
۲-۱- پیاده‌سازی تئوری شبکه

برای پیاده سازی این شبکه با توجه به مدار دیجیتال محاسبه کننده مکمل دو در صورت سوال داریم:



شکل ۱- مدار ۳۲ بیتی محاسبه کننده مکمل دو

که این مدار برای اعداد ۳۲ بیتی است و برای اعداد چهار بیتی که در این سوال مد نظر ما هست داریم:



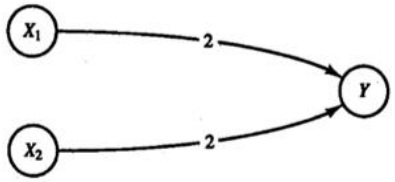
شکل ۲- مدار ۴ بیتی محاسبه کننده مکمل دو

از طرف دیگر با پیاده سازی شبکه‌های عصبی که مانند گیت‌های OR و XOR عمل می‌کنند و در آن‌ها از نورون‌های Mcculloch Pitts استفاده می‌شود، آشنا شدیم و داریم:

گیت OR:

OR

x_1	x_2	→	y
1	1		1
1	0		1
0	1		1
0	0		0

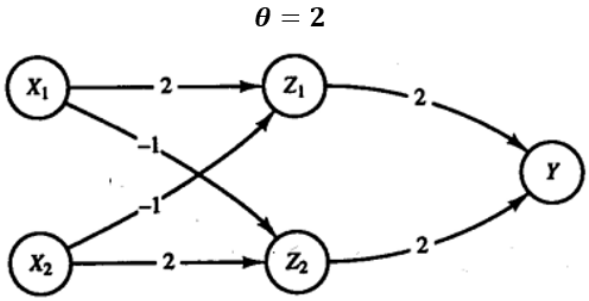


شکل ۳- گیت OR با استفاده از نورون های MccullochPitts

و برای گیت XOR:

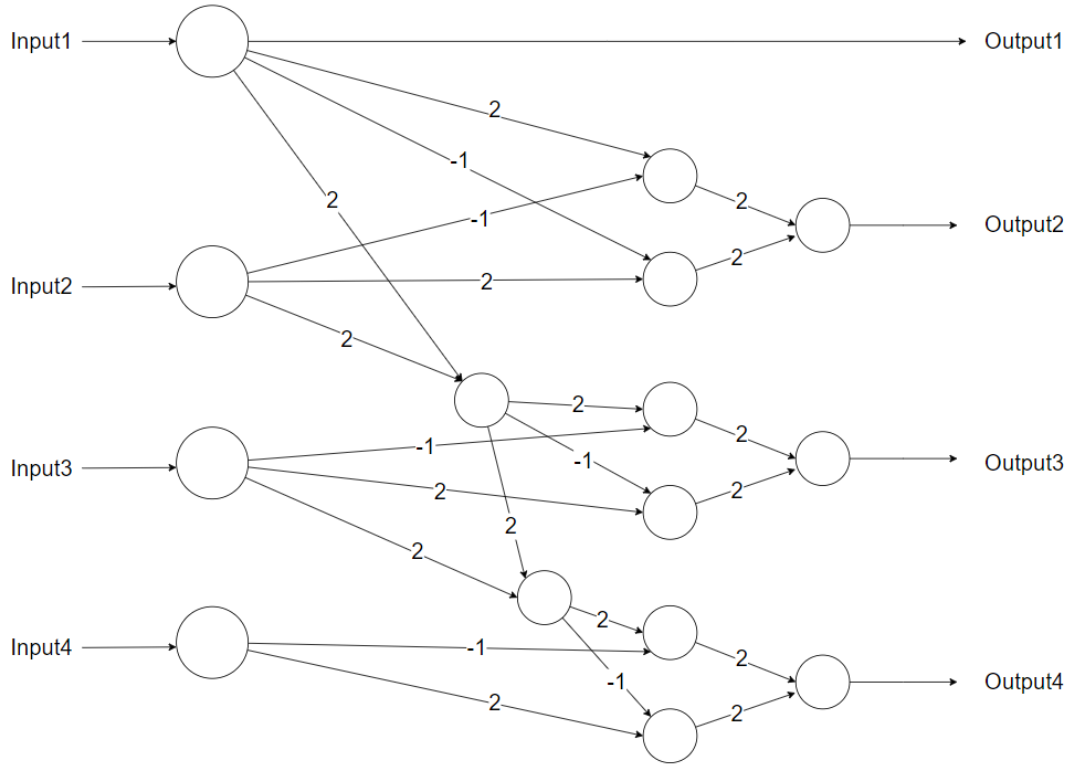
- XOR

x_1	x_2	→	y
1	1		0
1	0		1
0	1		1
0	0		0



شکل ۴- گیت XOR با استفاده از نورون های MccullochPitts

در مدار مورد بررسی نیز تنها از دو گیت OR و XOR استفاده شده است، پس کافی است به جای هر یک از این گیت‌ها شبکه‌ی عصبی مناسب آنها به همراه نورون‌ها و وزن‌ها مناسب را قرار دهیم که در نهایت خواهیم داشت:



شکل ۵- شبکه عصبی محاسبه کننده مکمل دو، ۴ بیتی

بدین ترتیب در شکل نمایش داده شده (شکل ۵) این شبکه به همراه نورون‌ها، ارتباطات بین آنها و وزن هر یال برروی آن نشان داده شده است.

۳-۱- پیاده سازی کد شبکه

برای پیاده سازی کد شبکه نیز به ترتیب توضیح داده شده در قسمت قبل عمل کردم بدین صورت که:

ابتدا کتابخانه‌های مورد نیز این سوال را فراخوانی کرده‌ام:

```
Q1

[2] import matplotlib.pyplot as plt
    import networkx as nx
    import numpy as np
```

شکل ۶- فراخوانی کتابخانه‌های مورد نیاز

سپس در ابتدا به تعریف یک نورون MccullochPitts به عنوان واحد سازنده کل شبکه‌ای که در ادامه بررسی خواهیم کرد، پرداخته ام:

```python
class MccullochPittsNeuron:
    def __init__(self, weights, threshold):
      self.weights = weights
      self.threshold = threshold

    def activate(self, inputs):
      value_checker = 0

      for i in range(len(self.weights)):
        neuron_value = self.weights[i] * inputs[i]
        value_checker += neuron_value

      if value_checker >= self.threshold:
        return 1
      else:
        return 0
```

شکل ۷- تعریف نورون MccullochPitts

در کلاسی که برای این نورون تعریف شده است، ابتدا در تابع سازنده آن وزن‌های ورودی‌ها و thresholdای که برای فعال سازی این نورون نیاز است را دریافت برای آن تنظیم کرده ام.

سپس یک تابع activate متطابق با آنچه در نورون‌های MccullochPitts داریم نوشته ام.

در این تابع ابتدا وزن‌های ورودی در تمامی ورودی ها ضرب شده و در متغیر value_checker باهم جمع می‌شوند و در آخر براساس activator تعریف شده با مقدار threshold مقایسه شده و اگر مقدار بزرگتر مساوی باشد، نورون فعال می‌شود (یک خروجی می‌دهد.) و در غیر این صورت غیرفعال می‌ماند و صفر خروجی می‌دهد.

در ادامه ابزارها و گیت‌هایی که برای استفاده در شبکه کلی ایجاد مکمل دو نیاز دارم (یعنی گیت‌های OR و XOR) را با استفاده از این نورون ایجاد کرده ام:

بدین صورت که برای گیت XOR ابتدا با توجه به شکل ۴ و ساختار این گیت با استفاده از نورون‌های MccullochPitts داریم:

```
def xor(bits):
  n1 = MccullochPittsNeuron([2,-1], 2)
  n2 = MccullochPittsNeuron([-1,2], 2)
  n3 = MccullochPittsNeuron([2,2], 2)

  n1_out = n1.activate([bits[0],bits[1]])
  n2_out = n2.activate([bits[0],bits[1]])
  n3_out = n3.activate([n1_out,n2_out])


  return n3_out
```

شکل ۸- کد تعریف گیت XOR با استفاده از نورون MccullochPitts

که در آن ابتدا سه نورون ایجاد کرده که threshold همگی برای فعال‌سازی ۲ است، سپس به طریقی که در شکل ۴ نشان داده شده است، اتصالات مناسب با وزن‌های مورد نیاز را بین نورون‌ها ایجاد می‌کنیم.

حال برای صحت از اطمینان کارکرد این گیت نیز تمامی حالت های دو ورودی را تست و ارزیابی می‌کنیم:

```
print(xor([0,1]))
print(xor([0,0]))
print(xor([1,1]))
print(xor([1,0]))
```

```
1
0
0
1
```

شکل ۹- بررسی و تست گیت XOR پیاده سازی شده

سپس برای ایجاد گیت OR با نورون‌های MccullochPitts نیز به همین صورت عمل می‌کنیم که مطابق شکل۳ یک نورون برای اینکار در نظر گرفته که دو ورودی به آن با وزن‌های نشان داده شده ([۲و۲]) متصل شده‌اند:

```
def or_neuron (bits):
  n1 = MccullochPittsNeuron([2,2], 2)
  return n1.activate([bits[0],bits[1]])
```

شکل ۱۰- تعریف گیت OR با استفاده از نورون MccullochPitts

برای این گیت هم تمامی حالت‌های دو ورودی برای اطمینان از صحت، عملکرد بررسی شده اند:

```
print(or_neuron([1,1]))
print(or_neuron([1,0]))
print(or_neuron([0,1]))
print(or_neuron([0,0]))
```

```
1
1
1
0
```

شکل ۱۱- بررسی و تست کد گیت OR پیاده سازی شده

در نهایت، حالا با داشتن دو گیت مورد نیاز می‌توان به سادگی شبکه مد نظر برای محاسبه مکمل دو را ایجاد کرد، بدین صورت که:

```
def twos_complement(input_bit):
  out0 = input_bit[3]
  out1 = xor([input_bit[3],input_bit[2]])
  carry1 = or_neuron([input_bit[3],input_bit[2]])
  out2 = xor([carry1,input_bit[1]])
  carry2 = or_neuron([carry1,input_bit[1]])
  out3 = xor([carry2,input_bit[0]])

  return([out3,out2,out1,out0])
```

شکل ۱۲- پیاده سازی شبکه محاسبه مکمل دو با استفاده از گیت‌های طراحی شده

دقیقا مطابق شبکه در شکل ۵ و مدار دیجیتالی شکل ۲ برای ایجاد اتصالات بین گیت‌ها عمل می‌کنیم، فقط باید برای ایجاد این اتصالات توجه داشت که ابتدا بیت خروجی صفر، یعنی اولین بیت از سمت راست محاسبه می‌شوند به همین ترتیب بیت‌ها از راست محاسبه می‌شوند تا به آخرین بیت برسند. همچنین باید توجه داشت از آنجایی که ایندکس‌گذاری پایتون از سمت چپ لیست هست برای دریافت اولین بیت ورودی نیز از ۳ امین بیت لیست ورودی شروع و به همین ترتیب ذکر شده ادامه دادم.

در نهایت هم برای تست نهایی این شبکه پیدا سازی شده، تمامی ۱۶ حالت ممکن برای اعداد ۴ بیتی را برای اطمینان از صحت عملکرد این شبکه بررسی کردیم:

```
print(f"input: {[0, 1, 0, 1]} --> output: {twos_complement([0, 1, 0, 1])}")
print(f"input: {[0, 0, 1, 1]} --> output: {twos_complement([0, 0, 1, 1])}")
print(f"input: {[0, 1, 1, 1]} --> output: {twos_complement([0, 1, 1, 1])}")
print(f"input: {[1, 0, 0, 0]} --> output: {twos_complement([1, 0, 0, 0])}")
print(f"input: {[1, 1, 0, 0]} --> output: {twos_complement([1, 1, 0, 0])}")
print(f"input: {[1, 0, 1, 0]} --> output: {twos_complement([1, 0, 1, 0])}")
print(f"input: {[1, 1, 1, 0]} --> output: {twos_complement([1, 1, 1, 0])}")
print(f"input: {[1, 0, 0, 1]} --> output: {twos_complement([1, 0, 0, 1])}")
print(f"input: {[1, 1, 0, 1]} --> output: {twos_complement([1, 1, 0, 1])}")
print(f"input: {[1, 0, 1, 1]} --> output: {twos_complement([1, 0, 1, 1])}")
print(f"input: {[1, 1, 1, 1]} --> output: {twos_complement([1, 1, 1, 1])}")
```

```
input: [0, 0, 0, 0] --> output: [0, 0, 0, 0]
input: [0, 1, 0, 0] --> output: [1, 1, 0, 0]
input: [0, 0, 1, 0] --> output: [1, 1, 1, 0]
input: [0, 1, 1, 0] --> output: [1, 0, 1, 0]
input: [0, 0, 0, 1] --> output: [1, 1, 1, 1]
input: [0, 1, 0, 1] --> output: [1, 0, 1, 1]
input: [0, 0, 1, 1] --> output: [1, 1, 0, 1]
input: [0, 1, 1, 1] --> output: [1, 0, 0, 1]
input: [1, 0, 0, 0] --> output: [1, 0, 0, 0]
input: [1, 1, 0, 0] --> output: [0, 1, 0, 0]
input: [1, 0, 1, 0] --> output: [0, 1, 1, 0]
input: [1, 1, 1, 0] --> output: [0, 0, 1, 0]
input: [1, 0, 0, 1] --> output: [0, 1, 1, 1]
input: [1, 1, 0, 1] --> output: [0, 0, 1, 1]
input: [1, 0, 1, 1] --> output: [0, 1, 0, 1]
input: [1, 1, 1, 1] --> output: [0, 0, 0, 1]
```

شکل ۱۳- تست و بررسی شبکه محاسبه مکمل دو

پرسش۲– حملات خصمانه در شبکه‌های عصبی

ابتدا کتابخانه‌های مورد نیاز برای قسمت‌های مختلف سوال را فراخوانی می‌کنیم:

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.losses import CategoricalCrossentropy
from keras.utils import to_categorical
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import keras.backend as K
from keras.losses import categorical_crossentropy
```

۲-۲- آشنایی با مجموعه دادگان

- ابتدا طبق خواسته سوال این دیتاست (MNIST) لود و از داده های مربوط به ابعاد و تعداد نمونه در تصاویر دسته آموزش و تست مورد بررسی قرار گرفتند:

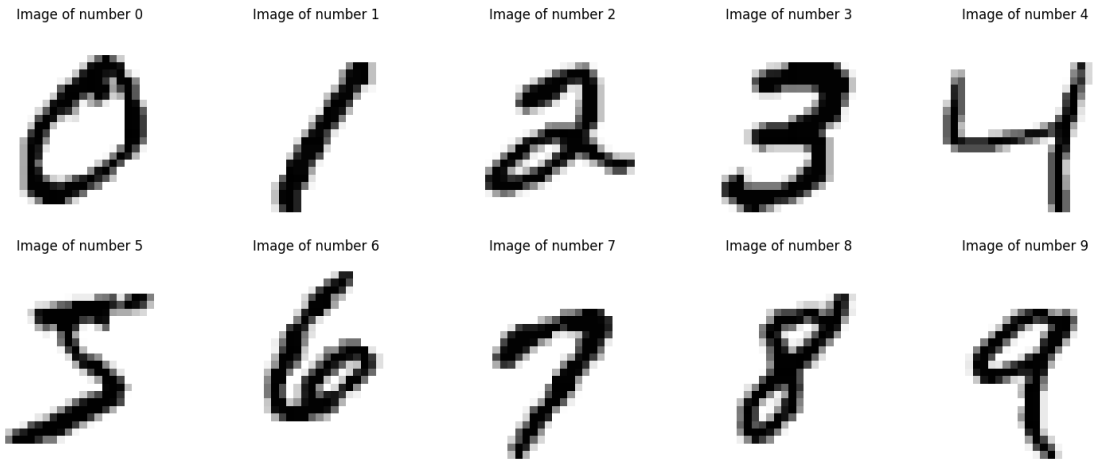
```
Q2-1 Load and check dataset

[ ] (train_images, train_labels), (test_images, test_labels) = mnist.load_data()

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

[ ] print(f"number of training samples: {train_images.shape[0]}")
    print(f"size of each training image: {train_images.shape[1:]}")
    print(f"number of test samples: {test_images.shape[0]}")
    print(f"size of each test image: {test_images.shape[1:]}")

    number of training samples: 60000
    size of each training image: (28, 28)
    number of test samples: 10000
    size of each test image: (28, 28)
```

از آنجایی که این یک دیتاست معروف و پرکاربرد است به صورت آماده در دیتاست‌های موجود در کتابخانه Keras وجود دارد به همین ترتیب به سادگی با فراخوانی آن ابتدا تصاویر تست و آموزش به همراه برچسب‌های هر یک از داده‌ها را فراخوانی و لود کرده ام و سپس به سادگی ابعاد متفاوت این دادگان مورد بررسی قرار دادم که نشان می‌دهد تمامی تصاویر سایز ۲۸ در ۲۸ پیکسل دارند و تعداد نمونه‌های آموزشی ۶۰۰۰۰ عدد و تعداد نمونه‌های تست ۱۰۰۰۰ عدد هستند.

- سپس برای نمایش یک نمونه از هر تصویر کد زیر را دارم:

```
fig, axs = plt.subplots(2, 5, figsize=(15, 6))

for i in range(10):
    image = train_images[train_labels == i][0]
    axs[i//5, i%5].imshow(image, cmap=plt.cm.binary)
    axs[i//5, i%5].set_title(f"Image of number {i}")
    axs[i//5, i%5].axis('off')

plt.tight_layout()
plt.show()
```

که در آن اولین نمونه از کلاس هر لیبل را از میان تصاویر برای دادگان آموزش برداشته و چون می‌خواستیم در یک تصویر تمامی موارد را داشته باشیم به صورت یک subplot که ۵ ستون و دو ردیف دارد در نظر گرفتیم و نتیجه بدین صورت شد:



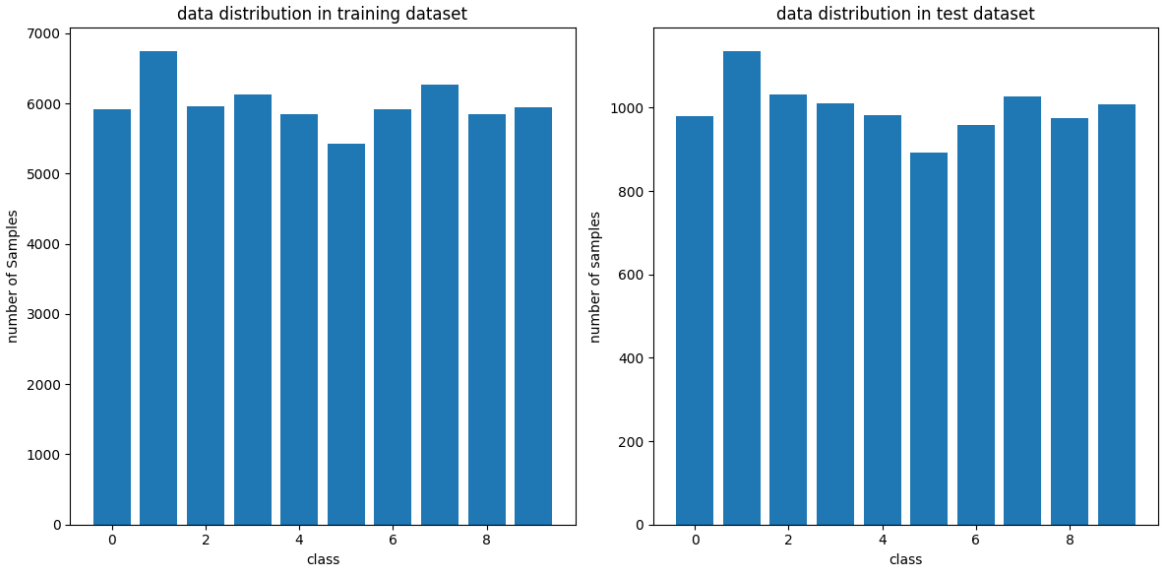
شکل ۱۴- تصاویر هر کلاس

- برای نمایش پراکندگی داده های به صورت histogram نیز بدین صورت عمل کردم:

```
train_class_distribution = np.bincount(train_labels)
test_class_distribution = np.bincount(test_labels)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.bar(range(10), train_class_distribution)
plt.title('data distribution in training dataset')
plt.xlabel('class')
plt.ylabel('number of Samples')

plt.subplot(1, 2, 2)
plt.bar(range(10), test_class_distribution)
plt.title('data distribution in test dataset')
plt.xlabel('class')
plt.ylabel('number of samples')

plt.tight_layout()
plt.show()
```

که در آن ابتدا برروی برچسب داده های تست و آموزش یک binning انجام دادم سپس مجدد در دو قسمت مختلف یک subplot (یکی برای هیستوگرام پراکندگی داده‌ها در نمونه آموزشی و یکی هم برای هیستوگرام پراکندگی داده ها در نمونه تست) نمودارهای لازم را با استفاده از تابع bar از کتابخانه

matplotlib.pyplot بر اساس ده کلاس مختلف داده و تعداد نمونه برای هر برچسب نمایش دادم که نتیجه بدین صورت شد:



شکل ۱۵- پراکندگی کلاس هر داده در نمونه های آموزشی و تست

با مقایسه نمودارها صورت حدودی و چشمی می‌توان یافت که تقریبا توزیع کلاس هم در داده های آموزشی و هم در داده های تست مناسب بوده و بالانس هستند، اما به صورت دقیق‌تر با یک محاسبه آماری ساده نیز می‌توان به این نتیجه رسید به طوری که اگر انحراف معیار استاندارد از میانگین داده به یک مقدار ثابتی (در اینجا بدلیل نزدیکی تعداد داده ها ۰.۱ در نظر گرفته شده است.) بزرگتر بود می توان نتیجه گرفت که توزیع داده ها نرمال نیست و نیاز به نرمالایز کردن است، اما در اینجا، این مقدار متناسب بوده و نیازی به عملیات دیگر جهت توزیع نرمال نیست.

```
if np.std(train_class_distribution) > 0.1 * np.mean(train_class_distribution) or np.std(test_class_distribution) > 0.1 * np.mean(test_class_distribution):
    print("unbalance")
else:
    print("balance")
```

```
balance
```

- در ادامه برای نرمال سازی داده ها با استفاده از روش Min-Max بزرگترین و کوچک‌ترین مقدار برای هر یک از پیکسال‌های داده ها را بدست می‌آوریم که برابر ۰ و ۲۵۵ است.

```
min_pixel_value = np.min(train_images)
max_pixel_value = np.max(train_images)

print('min pixel:', min_pixel_value)
print('max pixel:', max_pixel_value)
```

```
min pixel: 0
max pixel: 255
```

حال برای نرمال سازی Min-Max با توجه به فرمول داریم:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

که به توجه به صفر بودن کوچک ترین مقدار ممکن تنها با تقسیم مقدار هر پیکسل بر ۲۵۵ می توان نرمال سازی را بدین گونه انجام داد:

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

با استفاده از این نرمال ساز MinMax داده های مورد نظر بین ۰و۱ اسکیل می شوند و علت این کار این است که با انجام این کار با توجه فرمول گرادیان، کم شدن مقدار آن برای حالتی که مقادیر داده کوچک‌تر باشد بسیار سریع تر عمل می کند و زودتر همگرا می‌شود. به عبارت دقیق تر هر چه مقادیر مورد بررسی در مقیاس مشابه تری باشند منحنی‌های تابع هزینه نرم‌تر و یکنواخت تر بوده و در زمان کمتری می توان مینیمم آنها را یافت.

۳-۲- ایجاد و آموزش مدل

برای این قسمت ابتدا با توجه به ویژگی‌های ذکر شده باید مدل مد نظر را ایجاد، کامپایل و آموزش دهیم که به ترتیب داریم:

```
Q2-3

train_images = train_images.reshape((-1, 784))
test_images = test_images.reshape((-1, 784))
```

ابتدا ورودی دو بعدی تصاویر ۲۸*۲۸ را به یک بردار خطی یک بعدی ۲۸*۲۸ = ۷۸۴ خانه ای تبدیل می‌کنیم. این کار برای این است که تعداد نورون های لایه ورودی ۷۸۴ تا یعنی متناسب با تعداد پیکسل‌ها هستند پس باید در یک بردار ارائه بدهیم.

در ادامه برچسب داده ها را نیز با توجه به اینکه در ده دسته بندی مختلف قرار دارند به صورت One-Hot و categorical تبدیل می‌کنیم:

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

سپس مدل را با توجه به ویژگی‌هایی که در هر لایه ذکر شده است، ایجاد کرده:

```
model = Sequential([
    Dense(512, input_dim=784, activation='relu'),
    Dense(128, activation='relu'),
    Dense(32, activation='relu'),
    Dense(10, activation='softmax')
])
```

طبق پارامترهای خواسته شده کامپایل و ایجاد می‌کنیم:

```
model.compile(optimizer=Adam(learning_rate=6e-5), loss=CategoricalCrossentropy(), metrics=['accuracy'])
```

و در ادامه با توجه به داده های آموزشی، در ۲۵ ایپاک آموزش می‌دهیم:

```
model.fit(train_images, train_labels, epochs=25)
```

```
Epoch 1/25
1875/1875 [==============================] - 23s 12ms/step - loss: 0.5066 - accuracy: 0.8629
Epoch 2/25
1875/1875 [==============================] - 17s 9ms/step - loss: 0.2006 - accuracy: 0.9440
Epoch 3/25
1875/1875 [==============================] - 17s 9ms/step - loss: 0.1477 - accuracy: 0.9582
Epoch 4/25
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1170 - accuracy: 0.9667
Epoch 5/25
1875/1875 [==============================] - 17s 9ms/step - loss: 0.0953 - accuracy: 0.9725
Epoch 6/25
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0800 - accuracy: 0.9768
Epoch 7/25
1875/1875 [==============================] - 18s 10ms/step - loss: 0.0671 - accuracy: 0.9811
Epoch 8/25
1875/1875 [==============================] - 16s 8ms/step - loss: 0.0573 - accuracy: 0.9835
Epoch 9/25
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0493 - accuracy: 0.9858
Epoch 10/25
1875/1875 [==============================] - 17s 9ms/step - loss: 0.0422 - accuracy: 0.9880
Epoch 11/25
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0359 - accuracy: 0.9903
```

در آخر هم برای بررسی دقت و میزان loss مدل آموزش دیده با استفاده از تابع evaluate ارزیابی مدل را انجام می‌دهیم که با توجه سادگی این مسئله از دقت بسیار بالای ۹۸ درصد برخوردار هستیم:

```
loss, accuracy = model.evaluate(test_images, test_labels)
print(f"loss: {loss}")
print(f"accuracy: {accuracy}")
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.0801 - accuracy: 0.9804
loss: 0.0801054835319519
accuracy: 0.980400025844574
```

در نهایت برای سرعت بیشتر و استفاده در مراحل بعدی مدل ذخیره هم شده است که مجدد بدون آموزش شبکه قابلیت بارگذاری مجدد آن را داشته باشیم:

```
model.save('model.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning: You are saving your model a
  saving_api.save_model(
```

- همانطور که گفته شد لایه ورودی این شبکه دارای ۷۸۴ نورون هست که در واقع به ازای هر پیکسل از تصویر ۲۸*۲۸ = ۷۸۴ یک نورون برای ورودی آن در نظر گرفته شده است.
- تابع‌های فعال‌سازی (active) در اکثر شبکه‌های عصبی معمولا متفاوت با سایر لایه‌شبکه هستند، زیرا از مهم‌ترین وظایف لایه آخر و لایه خروجی این است که با توجه به خروجی مورد نیاز و انتظار از شبکه خروجی خودش را ارائه دهد یعنی به طور دقیق‌تر در این سوال یک مسئله classification چندتایی داریم که از میان ده کلاس باید تشخیص دهیم که هر تصویر مربوط به کدام کلاس است. به همین دلیل در لایه آخر از Softmax استفاده می‌کنیم زیرا خروجی نیاز است تا نشان‌دهنده رقم تشخیص داده شده باشد. نه مانند ReLU مجموعه ای از نورون‌های فعال شده یا غیرفعال، به همین ترتیب Softmax به عنوان خروجی به طور خاص یک بردار از احتمال حضور نمونه در هر یک از کلاس‌ها را می دهد.

۴-۲- پیاده سازی حمله FGSM

در گام اول ابتدا به مراحل پیاده‌سازی حمله می‌پردازیم:

طبق نمونه کد موجود در سوال ما در ابتدا یک تابع fgsm داریم که به عنوان ورودی مدل آموزش دیده، داده‌هایی که می‌خواهیم حمله را از طریق آنان پیاده‌سازی و اجرا کنیم (X)، برچسب داده‌هایی که حمله از طریق آن‌ها صورت می گیرد و پارامتر اپسیلون را به تابع می‌دهیم و تابع مقدار مناسب و مهندسی شده‌ای از نویز (delta) را به ازای هر نمونه داده شده در x ارائه می‌کند. که با افزودن آن نویز به تصویر می توان دقت مدل بطور فاجعه آمیزی کاهش داد:

```
def fgsm(model, x, y, epsilon):
    x_tensor_format = tf.convert_to_tensor(x, dtype=tf.float32)

    with tf.GradientTape() as tape:
        tape.watch(x_tensor_format)
        predicted_result = model(x_tensor_format)
        loss = tf.keras.losses.categorical_crossentropy(y, predicted_result)

    gradients = tape.gradient(loss, x_tensor_format)
    sing_of_grad = tf.sign(gradients)
    delta = epsilon * sing_of_grad

    return delta
```

در این تابع با توجه به پیاده سازی در لایه پایین تر از keras با استفاده از tensorflow ابتدا داده ورودی را به نوعی تنسور تبدیل کرده و تغییر گرادیان در هر مرحله با استفاده از مدل نتیجه را تشخیص داده و مقدار loss بدست آمده را بر اساس داده های کتگوریکال بررسی می کنیم، سپس مقدار گرادیان و علامت آن را تشخیص می‌دهیم و با ضرب کردن مقدار eps در این علامت مقدار نویز مورد نیاز را پیدا می‌کنیم.

حالا مقدار delta را برای داده آموزشی مد نظر یافته:

```
delta = fgsm(model, train_images, train_labels, 0.1)
```

و در نهایت این مقدار را با تصویر اصلی جمع می‌کنیم و به مدل آموزش دیده می‌دهیم برای تشخیص و ارزیابی مجدد برروی داده‌های خراب شده:

```
bad_train_images = train_images + delta
predict_attacked_train = model.predict(bad_train_images)
```

```
1875/1875 [==============================] - 8s 4ms/step
```

```
loss, accuracy = model.evaluate(bad_train_images, train_labels)

print('loss:', loss)
print('accuracy:', accuracy)
```

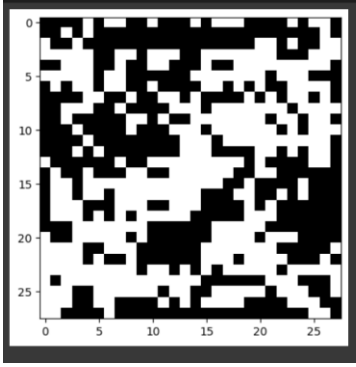
```
1875/1875 [==============================] - 7s 4ms/step - loss: 26.7210 - accuracy: 0.0439
loss: 26.720977783203125
accuracy: 0.04388333484530449
```

مشاهده می‌شود که دقت از ۹۸ درصد به طور زیادی به حدود ۰.۰۴ درصد رسیده است، همچنین مقدار loss مدل نیز از مقدار ۰.۰۸ که مقدار ناچیزی بوده است به ۲۶.۷ رسیده است!

به طور عملی می‌توان اینطور یک نمونه را بررسی کرد:

نویز تولید شده برای نمونه ۵۰۰ ام مجموعه آموزشی بدین صورت بوده است:

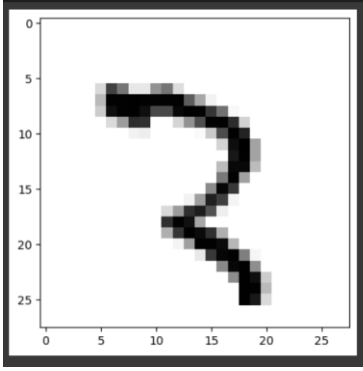
```
noise = delta[500].numpy()
noise = noise.reshape((28, 28))
plt.imshow(noise, cmap=plt.cm.binary)
plt.show()
```



شکل ۱۶- نویز تولید شده توسط **FGSM**

همچنین خود نمونه قبل از اعمال نویز بدین شکل بوده است:

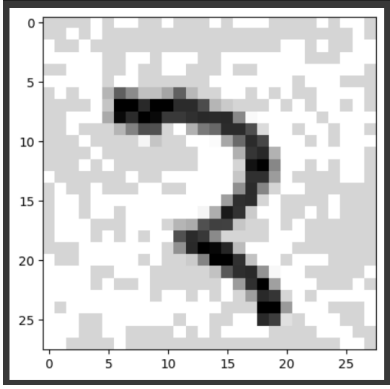
```
good_sample_img = train_images[500]
good_sample_img = good_sample_img.reshape((28, 28))
plt.imshow(good_sample_img, cmap=plt.cm.binary)
plt.show()
```



شکل ۱۷- تصویر نمونه سالم

و تصویر پس از اعمال نویز بدین صورت در آمده است:

```
noisy_sample_img = bad_train_images[500].numpy()
noisy_sample_img = noisy_sample_img.reshape((28, 28))
plt.imshow(noisy_sample_img, cmap=plt.cm.binary)
plt.show()
```



شکل ۱۸- تصویر نمونه حمله شده

که در نهایت منجر به تشخیص کاملا اشتباه شده است:

```
sample_label = np.argmax(train_labels[500])
print(f"label: {sample_label}")
predictions_class = np.argmax(predict_attacked_train[500])
print(f"predict: {predictions_class}")
```

```
label: 3
predict: 7
```

همچنین من عملیات قبل را برای داده های تست نیز تکرار کردم که نتیجه تقریبا مشابهی داشت:

```
test_delta = fgsm(model, test_images, test_labels, 0.1)
test_images_adv = test_images + test_delta
predict_attacked_test = model.predict(test_images_adv)
```

```
313/313 [==============================] - 2s 5ms/step
```

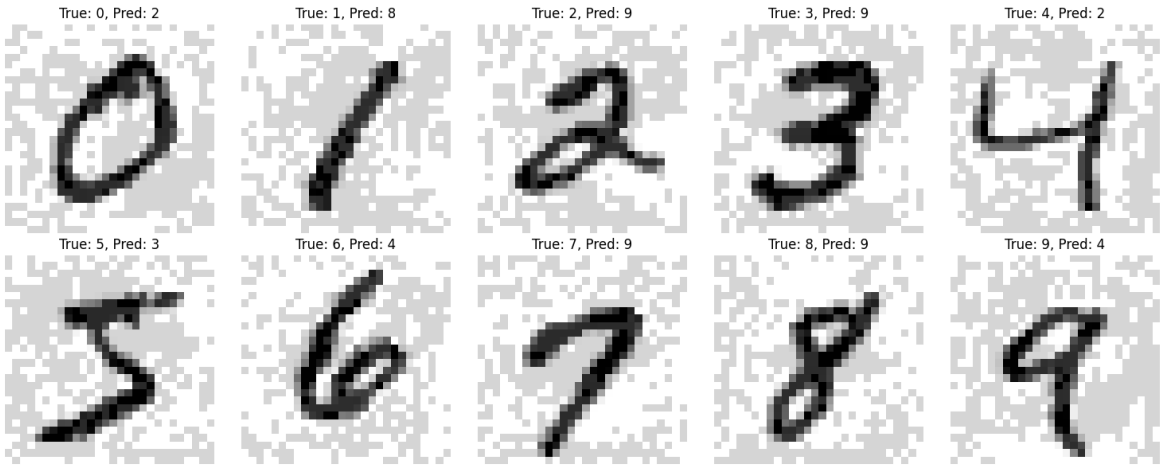
```
loss, accuracy = model.evaluate(test_images_adv, test_labels)

print('loss:', loss)
print('accuracy:', accuracy)
```

```
313/313 [==============================] - 1s 5ms/step - loss: 26.8735 - accuracy: 0.0471
loss: 26.873531341552734
accuracy: 0.0471000000834465
```

- برای نمایش یک نمونه از کلاس به همراه برچسب آن نمونه و تشخیص مدل، پس از ایجاد حمله در کد آن داریم:

```
train_label_class = np.argmax(train_labels, axis=1)
classes, counts = np.unique(train_label_class, return_counts=True)

examples = []

for c in classes:
    andis = np.where(train_label_class == c)[0]
    example = bad_train_images[andis[0]]
    examples.append(example)

examples = np.array(examples)
predictions = model.predict(examples)
predictions_class = np.argmax(predictions, axis=1)

fig, axs = plt.subplots(2, 5, figsize=(15, 6))

for i in range(len(examples)):
    axs[i//5, i%5].imshow(examples[i].reshape((28, 28)), cmap=plt.cm.binary)
    axs[i//5, i%5].set_title(f"True: {classes[i]}, Pred: {predictions_class[i]}")
    axs[i//5, i%5].axis('off')

plt.tight_layout()
plt.show()
```

که در آن ابتدا یک مجموعه از کلاس‌های موجود در برچسب‌های مورد استفاده را نگه داشته سپس به ازای هر کلاس آن نمونه را داخل لیست examples نگه می داریم در ادامه نمونه را به مدل می دهیم تا تشخیص دهد و در نهایت در subplot ها مجزا تصویر، برچسب و مقدار تشخیص داده شده به ازای هر نمونه را نمایش می‌دهیم که نتیجه بدین صورت است:



شکل ۱۹- نمونه های حمله شده از هر کلاس

۵-۲- پیاده سازی حمله PGD

ابتدا برای پیاده سازی این حمله مانند قسمت قبل یک تابع PGD در نظر گرفته شده است:

```
def pdg(model, images, labels, epsilon, alpha, num_iter):
    attack_mask = tf.zeros_like(images)

    for i in range(num_iter):
        with tf.GradientTape() as tape:
            tape.watch(attack_mask)
            bad_sample = images + attack_mask
            predictions = model(bad_sample)
            loss = tf.keras.losses.categorical_crossentropy(labels, predictions)

        gradients = tape.gradient(loss, attack_mask)
        sign_of_gradient = tf.sign(gradients)
        attack_mask += alpha * sign_of_gradient
        attack_mask = tf.clip_by_value(attack_mask, -epsilon, epsilon)

    return attack_mask
```

که در آن علاوه بر مدل، نمونه‌ها، برچسب‌های هر یک از نمونه‌ها و سه پارامتر مربوط به این حمله یعنی epsilon, alpha, num_iter دریافت می‌شوند.

در این تابع ابتدا یک ماسک خالی ایجاد شده کرده و سپس به ازای تعداد دفعات اجرای حمله (num_iter) در هر بار اجرا الگوریتم تا حدودی مانند FGSM عمل می‌گردد البته به همراه تغییراتی که در این حمله تعداد دفعات تکرار حمله افزایش یافته است و در مرحله محاسبه گرادیان یک مقدار گرادیان نیز به حمله افزوده گردید است که در دور این تغییرات ضریب خورده به ماسک نهایی که باید به تصویر افزوده گردد، اضافه می‌شوند و در نهایت با تابع clip_by_value یک نرمال سازی براساس میزان مثبت و منفی eps که رنج تغییرات را مشخص می‌کنند انجام شده است و نهایتا ماسکی که با اضافه کردن آن به داده اصلی می‌توان حمله اعمال کرد، را به عنوان خروجی تابع خواهیم داشت.

سپس در ادامه delta هم برای داده های آموزشی و تست را ایجاد کرده و به هر یک از آن‌ها افزودیم و به مدل داده و دقت و loss آن را ارزیابی کردم، ابتدا برای نمونه‌های train_images داریم:

```
delta = pdg(model, train_images, train_labels, epsilon=0.1, alpha=0.01, num_iter=40)
bad_train_images = train_images + delta
predict_attacked_train = model.predict(bad_train_images)
```

```
1875/1875 [==============================] - 7s 4ms/step
```

و دقت و loss را برای آن ارزیابی کردیم:

```
loss, accuracy = model.evaluate(bad_train_images, train_labels)

print('loss:', loss)
print('accuracy:', accuracy)
```

```
1875/1875 [==============================] - 7s 4ms/step - loss: 36.9725 - accuracy: 1.8333e-04
loss: 36.97246170043945
accuracy: 0.000183333337190561
```

که به دقت بسیار بسیار پایین و نزدیک به صفر ۰.۰۰۰۱ رسیدیم که دقتی پایین تر از FGSM است.

همچنین برای loss نیز با اینکه eps مانند حمله FGSM است اما loss نیز به طور قابل توجهی افزوده شده است و به ۳۶.۹۷ رسیده است.

برای داده های تست نیز به همین ترتیب داریم:

```
delta = pdg(model, test_images, test_labels, epsilon=0.1, alpha=0.01, num_iter=40)
bad_test_images = test_images + delta
predict_attacked_test = model.predict(bad_test_images)
```

```
313/313 [==============================] - 1s 3ms/step
```

```
loss, accuracy = model.evaluate(bad_test_images, test_labels)

print('loss:', loss)
print('accuracy:', accuracy)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 37.2519 - accuracy: 1.0000e-04
loss: 37.25192642211914
accuracy: 9.999999747378752e-05
```

که مجدد با توجه کم بودن تعداد نمونه و تغییرات زیاد شاهد دقت پایین تر و loss بیشتر هستیم.

- در رابطه با تفاوت FGSM و PGD می‌توان به موارد زیر اشاره کرد:

اولین موردی که در این دو حمله تفاوت دیده میشد مدت زمان اجرا و منابع محاسباتی مورد استفاده بود، بدین صورت که FGSM سریع‌تر اجرا شده و در حین اجرا از RAM و CPU کمتری استفاده می‌کرد، دلیل این مسئله هم مشخص است، زیرا همانطور که در قسمت پیاده سازی ها گفته شد، FGSM یک روش یک مرحله ای است و تنها یک بار گرادیان را محاسبه می کند، اما PGD گسترش یافته FGSM است که همان فرایند را در چند مرحله تکرار می کند و این امر باعث می‌شود با اینکه PGD منابع بیشتری مصرف کند و برای ایجاد ماسکی که نمونه افزوده می‌گردد، زمان بیشتری بخواهد

اما PGD به مدل بهتری برای حملات خصمانه به شبکه‌های عصبی تبدیل شود و طور کلی خروجی بهتری را به ما ارائه می‌دهد.

- مزایای PGD نسبت FGSM همانطور در قسمت قبل گفته شد این است که می‌تواند نمونه بهتری برای حمله کارامدتر تولید کند و نسبت به FGSM با eps ثابت همانطور که دیده شد دقت را پایین تر و loss را فزایش دهد.

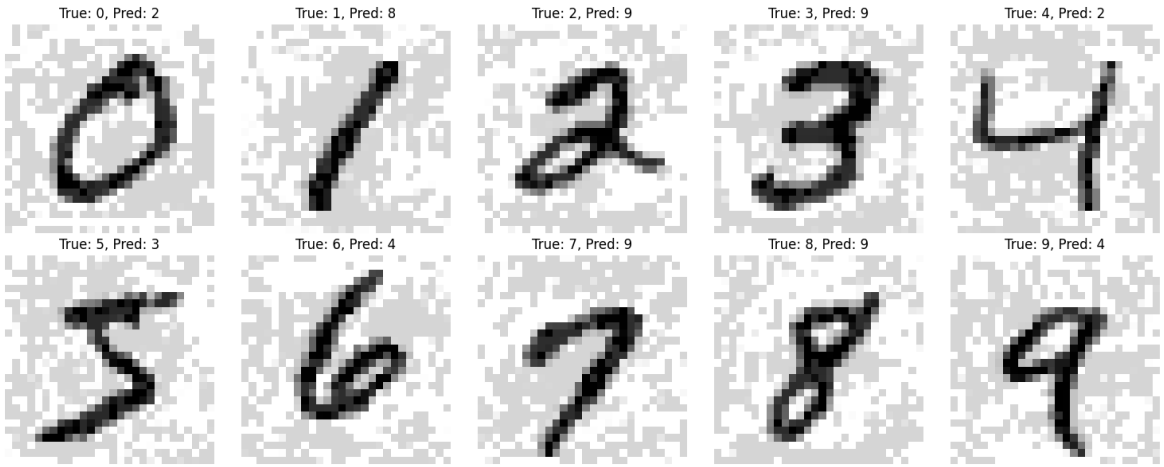
همچنین یک نکته جالب دیگر در PGD نسبت FGSM این بود که بین نمونه های مشابه PGD تغییرات و خرابکاری کمتری در نمونه ایجاد می کند اما تاثیر آن بیشتر است!!

همچنین نکته دیگر که در بررسی ها مورد توجه بود، این بود که پارامترهای ثابت در FGSM یعنی eps تاثیر بیشتری داشتند و تغییر پارامترهای ثابت در PGD تاثیر کمتر و منجر به تغییرات کمتری میشدند.

- برای بررسی یک نمونه از هر کلاس نیز داریم:

```
train_label_class = np.argmax(train_labels, axis=1)
classes, counts = np.unique(train_label_class, return_counts=True)

examples = []

for c in classes:
    andis = np.where(train_label_class == c)[0]
    example = bad_train_images[andis[0]]
    examples.append(example)

examples = np.array(examples)
predictions = model.predict(examples)
predictions_class = np.argmax(predictions, axis=1)

fig, axs = plt.subplots(2, 5, figsize=(15, 6))

for i in range(len(examples)):
    axs[i//5, i%5].imshow(examples[i].reshape((28, 28)), cmap=plt.cm.binary)
    axs[i//5, i%5].set_title(f"True: {classes[i]}, Pred: {predictions_class[i]}")
    axs[i//5, i%5].axis('off')

plt.tight_layout()
plt.show()
```

که مانند قسمت قبل ابتدا کلیه کلاس‌های موجود در نمونه ها را پیدا کرده و از حالت category خارج می‌کنیم، سپس در میان داده ها، اولین عضو از هر کلاس را انتخاب می کنیم و در لیست examples نگهداری می‌کنیم. در ادامه برای هر یک از نمونه های داخل examples تشخیص مورد نیاز را انجام می‌دهیم و در نهایت در subplot های یک plot به صورت ۵ ستون، ۲ ردیفه نمایش دادیم که نتیجه آن بدین گونه است:



شکل ۲۰- نمونه های حمله شده از هر کلاس

پرسش ۳. Madaline و Adaline

۱-۱. Adaline

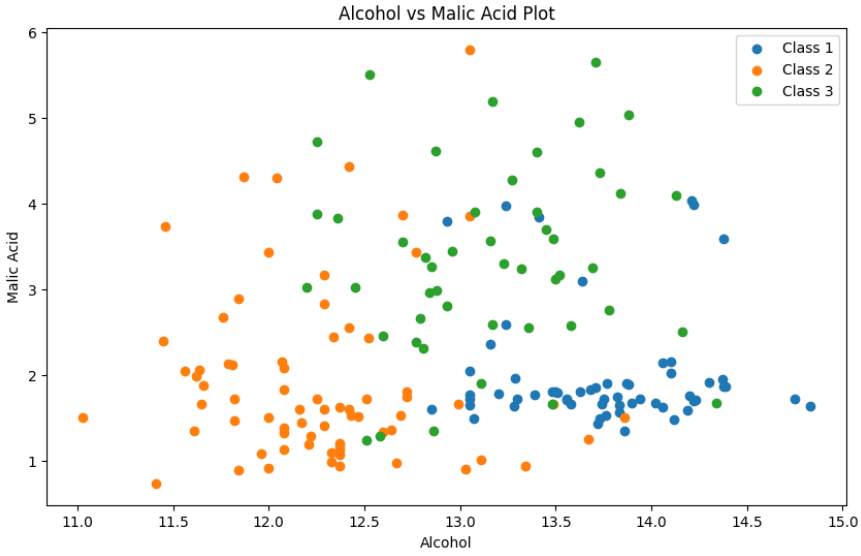
در این پرسش قصد داریم به کمک روش Adaline یک شبکه عصبی آموزش داده و یک مسئله طبقه‌بندی را حل کنیم. همانطور که در درس مشخص شد، این نوع شبکه تک‌لایه، تابع Activision بر اساس حد خاصی کار نمی‌کند و بر اساس مثبت یا منفی بودن نتیجه کار می‌کند. حال به صورت مرحله به مرحله خواسته‌های سوال را بررسی می‌کنیم.

پیش از انجام قسمت الف، دیتاست مورد نظر را ایمپورت کرده و نگاهی به ویژگی‌های آن داریم. این دیتاست ویژگی‌های مختلفی دارد و ما در این سوال قصد داریم که با کمک این ویژگی‌های بتوانیم عمل طبقه‌بندی را انجام دهیم.

	Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

پاسخ قسمت الف)

ابتدا برای داده‌های هر کلاس را با یک حلقه جداسازی می‌کنیم و برای داده‌های مربوط به هر کلاس با توجه به ویژگی Alchogol وMalic acid یک Scatter plot رسم می‌کنیم. نتیجه به صورت زیر است :



شکل ۲۱ نمودار پراکندگی داده‌ها باتوجه به ویژگی‌های **Alcohol** و **Malic acid**

با یک نگاه کلی به نمودار بالا می‌توان دریافت که میزان Alcohol کلاس ۲ کمتر از دیکر کلاس‌هاست و این مورد درباره کلاس ۱ برعکس است. همچنین کلاس ۳ Malic acid بیشتری نبست به دو کلاس دیگر دارد و Alcohol آن نیز به طور متوسط از کلاس۲ بیشتر و از کلاس ۱ کمتر است. این موارد کاملا شهودی هستند و نمی‌توان بر اساس آن‌ها تصمیم بگیریم ولی این مورد به ما کمک می‌کند که برای طبقه‌بندی از کدام ویژگی‌ها استفاده کرده و کدام یک تاثیر بیشتری در طبقه‌بندی دارند. با توجه به این نتیجه می‌توان گفت که باتوجه به شباهت مقادیر این ویژگی‌ها در کلاس‌های ۱ و ۳ ممکن است طبقه‌بندی کلاس ۱ دچار خطای زیادی شود.

سپس کلاس Adaline پیاده‌سازی می‌شود. به طور پیشفرض مقدار learning rate که بر اساس آن وزن‌ها بروزرسانی می‌شود برابر ۰.۰۱ و تعداد ایپاک‌ها که تعداد تکرار بروزرسانی شبکه در مرحله آموزش است نیز برابر ۵۰ است. تابع fit نیز برای آموزش مدل است و به صورت پیشفرض وزن صفر برای مدل در نظر گرفته می‌شود. متغیر cost نیز خطای هر مرحله را ذخیره می‌کند که میزان خطا همان تفاوت بین خروجی شبکه و لیبل واقعی است. در آخر نیز تابع net_input خروجی شبکه را محاسبه می‌کند. تابع activation نیز باتوجه اینکه مدل از نوع Adaline است با threshold برابر صفر مقایسه می‌شود.

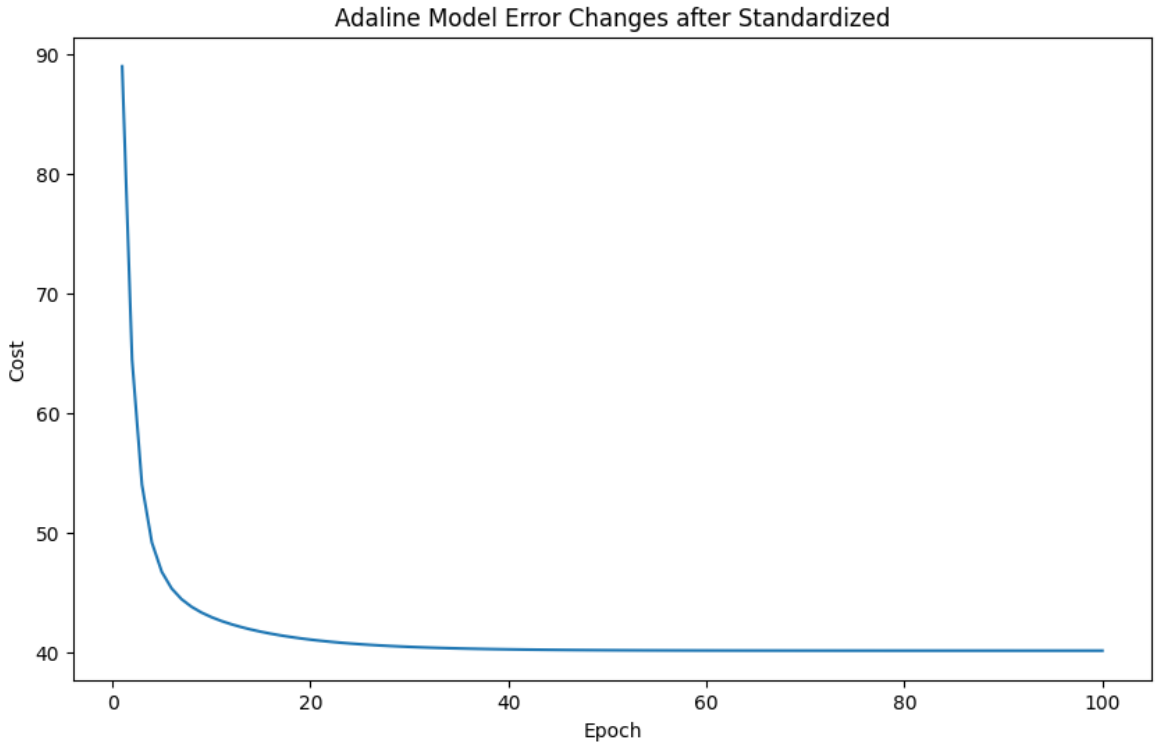
برای آموزش این مدل ابتدا داده‌ها باید به نحوی تبدیل شوند که طبقه‌بند، باینری شود. پس مقادیر کلاس ۱ برابر ۱ و دیگر کلاس‌ها برابر ۱- شوند. در برخی الگوریتم‌ها نظیر Adaline می‌توان برای افزایش کارایی و محاسبات بجای صفر از -۱ استفاده کرد.

اولین بار که کد اجرا شد با خطای زیر مواجه شدیم:

```
RuntimeWarning: overflow encountered in square
  cost = (errors**2).sum() / 2.0
```

برای حل این مشکل ویژگی‌های مورد بررسی یعنی Alcohol و Malic acid را نرمال‌سازی کرده و آن‌ها را به فرم نرمال استاندارد تبدیل می‌کنیم. برای اینکار میانگین آن‌ها از مقادیر کم شده و تقسیم بر انحراف معیار می‌شوند.

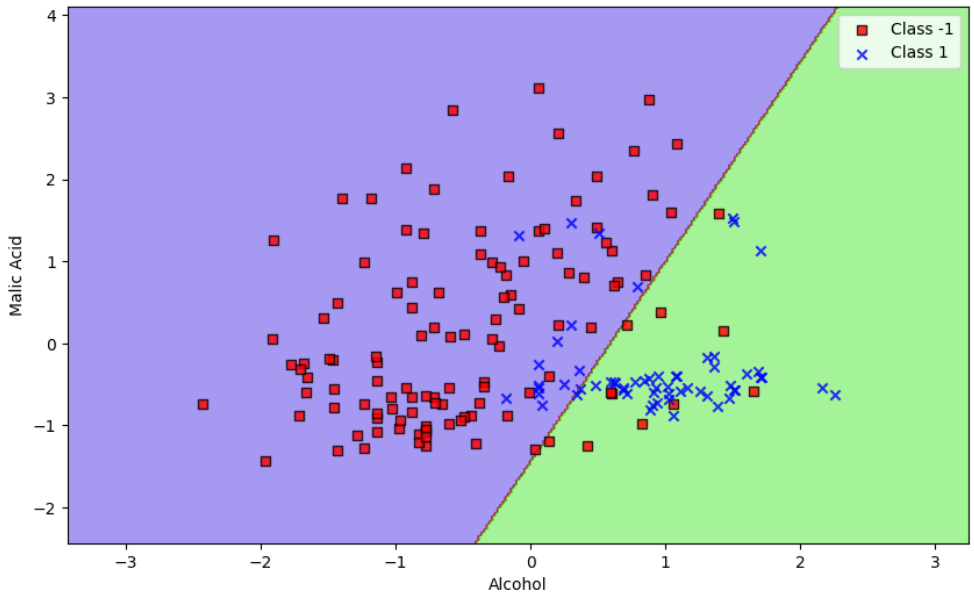
پس از این تغییر مجدد مدل با تعداد epoch ۱۰۰ و learning rate ۰.۰۱ آموزش داده‌شد. نمودار تغییرات خطا به صورت زیر است :



شکل ۲۲ نمودار تغییرات خطا نسبت به **Epoch** در مدل **Adaline**

همانطور که در شکل بالا مشخص است ۳۰ تکرار برای این مدل کافی است تا خطای آن به مقدار نسبتا ثابتی برسد و مقدار تغییرات خطا حدود ۴۰ باقی می‌ماند.

پس از این مرحله برای درک بهتر ناحیه تصمیم نیز رسم می‌کنیم تا متوجه شویم Adaline چه مناطقی را به عنوان کلاس ۱ جداسازی کرده است.



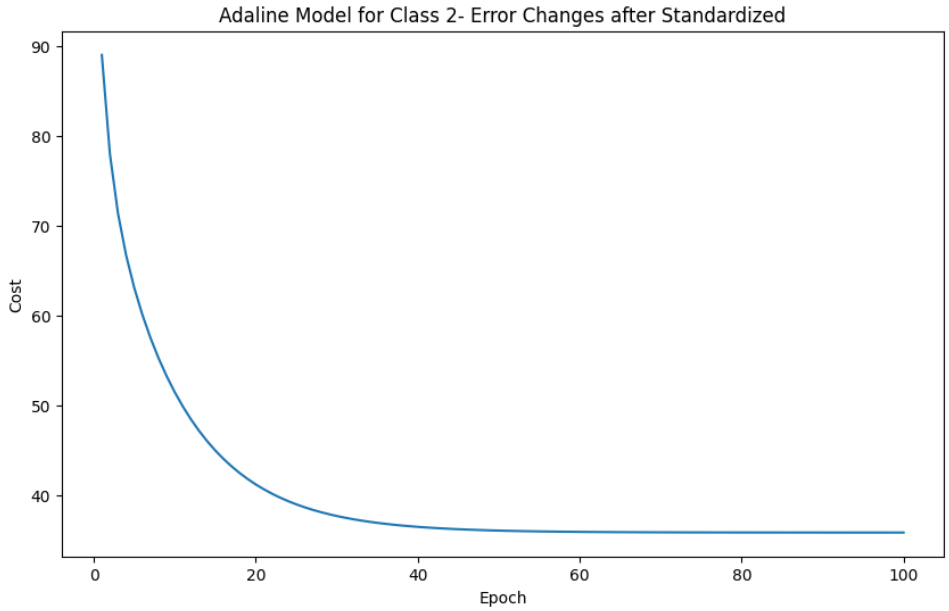
شکل ۲۳ مرز تصمیم‌گیری برای کلاس ۱

باتوجه به شکل بالا مشخص است که دلیل اینکه خطا از مراحلی به بعد ثابت می‌ماند، محدودیت‌های مربوط به خطی بودن طبقه‌بندی است.

پاسخ قسمت ب)

در این مرحله، همانند مرحله الف، طبقه‌بندی را روی داده‌های مربوط به کلاس ۲ اعمال می‌کنیم.

نتیجه تغییرات خطا به صورت زیر است :

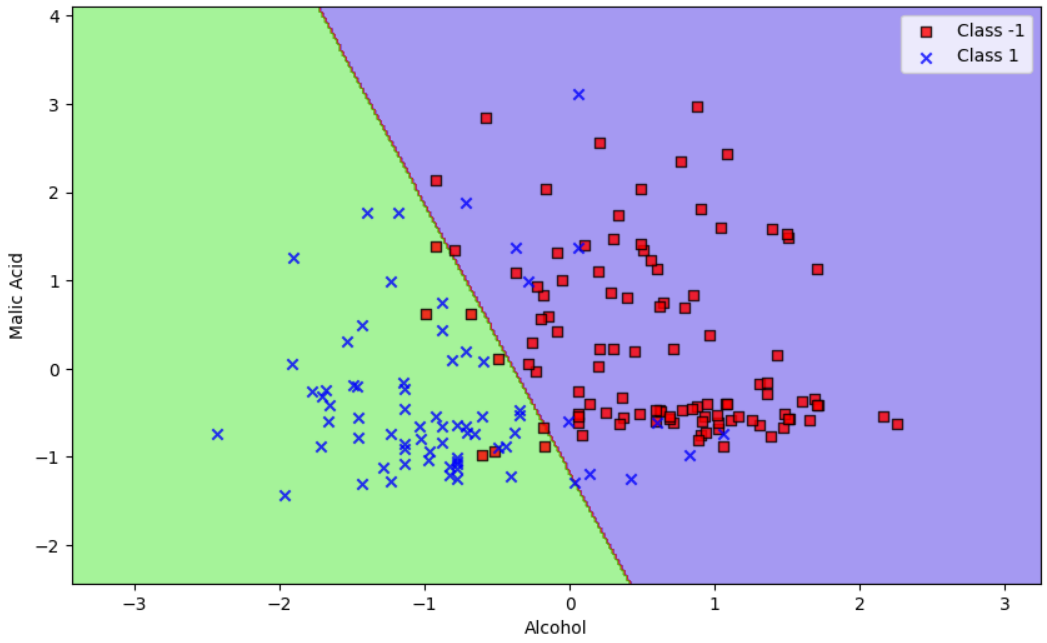


شکل ۲۴ نمودار تغییرات خطا نسبت به **Epoch** برای کلاس ۲

بامقایسه تغییرات خطا با قسمت الف متوجه می‌شویم که در این مدل، تغییرات خطا دیرتر کاهش می‌یابند ولی در نهایت به مقدار خطای کمتری می‌رسیم:

```
Final Cost of Part a:  40.16904379710083
Final Cost of Part b:  35.89945115912139
```

برای مقایسه این دو مدل مرز تصمیم مدل دوم را نیز رسم می‌کنیم.



شکل ۲۵مرز تصمیم‌گیری برای کلاس ۲

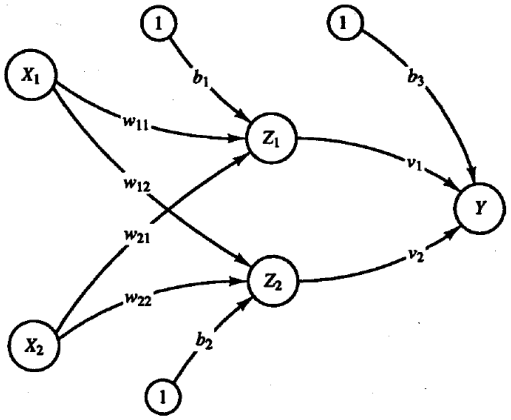
همانطور که مشخص است، در این مدل نسبت به مدل اول طبقه‌بندی بهتری صورت گرفته است. باتوجه به اینکه Adaline به صورت خطی عمل می‌کند، پراکندگی داده‌های کلاس ۲ باتوجه به ویژگی‌های ارائه‌شده به نحوی است که داده‌های بیشتری را می‌توان با یک خط از دیگر کلاس‌ها جداسازی کرد. این مورد به دلیل ذات Adaline است. در کلاس ۱ که مرز تصمیم آن نیز در شکل قبلی مشخص است، داده‌هایی که در فضای ویژگی در مجاورت کلاس‌های دیگر قرار دارند بیشتر است و نمی‌توانیم به صورت خطی آن‌هارا جدا کنیم و در صورتی که خط تغییر کند، خطا افزایش خواهدیافت.

۱-۱. Madaline

همانطور که در قسمت قبل هم اشاره شد برای حل مسائلی که با کمک جداسازی خطی امکان طبقه‌بندی وجود ندارد نیاز به روش‌های پیشرفته‌تر است. در این پرسش به بررسی Madaline می‌پردازیم که در اصل یک شبکه است که از تعدادی Adaline در لایه مخفی تشکیل شده که در نهایت با کمک یک نورون M&P AND یا OR ترکیب می‌شوند تا بتواند مشکل مربوط به خطی بودن Adalineرا حل کند. برای این روش ۲ الگوریتم در کتاب ارائه شده است.

در الگوریتم MRI نورون خروجی به صورت OR در نظر گرفته شده و به طور کلی فقط وزن‌ها و بایاس نورون‌های لایه مخفی در مرحله آموزش تغییر می‌کنند. یعنی وزن نورون خروجی در این الگوریتم ثابت است. ولی در الگوریتم MRII وزن و بایاس تمامی نورون‌ها در مرحله آموزش تغییر می‌کنند. اضافه شدن لایه مخفی قدرت مدل را افزایش می‌دهد ولی از طرفی پیچیدگی محاسبات در مرحله آموزش نیز افزایش خواهدیافت.

شکل زیر یک نمونه از این نوع شبکه را نشان می‌دهد:

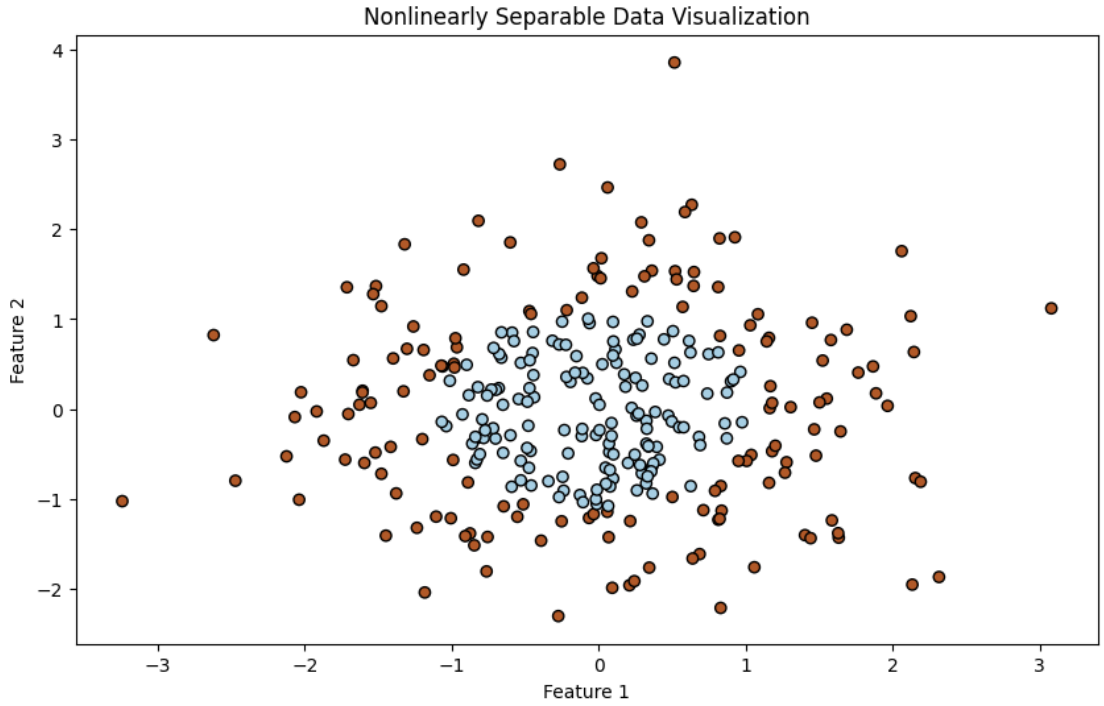


Figu
den

شکل ۲۶ ساختار شبکه **Madaline**

پاسخ قسمت ب)

برای حل این قسمت از پرسش الگوریتم MRII به صورت یک کلاس پیاده‌سازی شد. سپس طبق تصویر ارائه شده در متن سوال داده‌های مصنوعی تولید شدند. تصویر زیر پراکندگی داده‌ها را با توجه به ویژگی‌های ۱ و ۲ نشان می‌دهد:



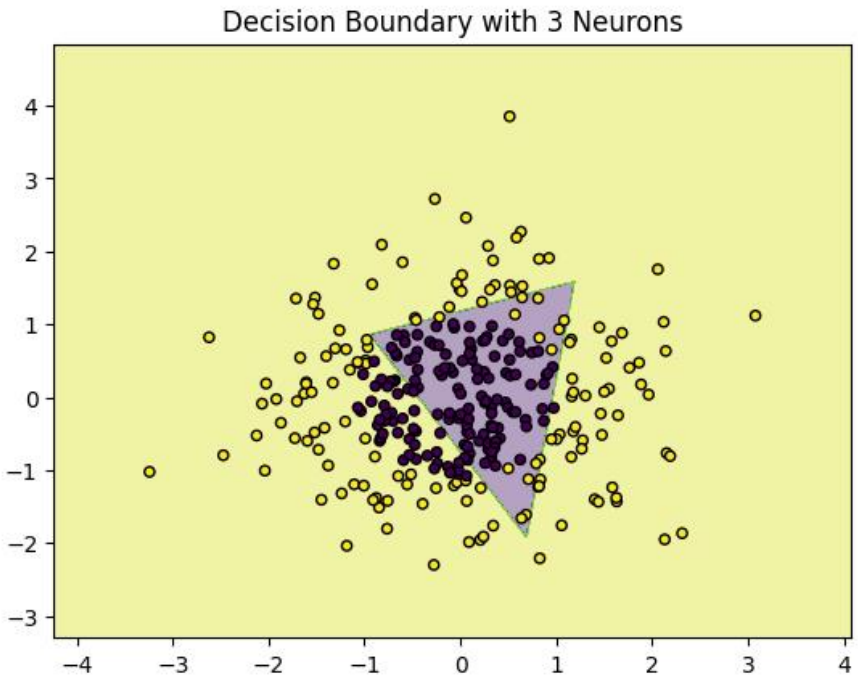
شکل ۲۷ نمودار پراکندگی داده‌های تولید شده بر اساس ویژگی ۱ و ۲

باتوجه به اینکه داده‌های آبی در درون داده‌های قهوه‌ای هستند، با کمک یک خط امکان طبقه‌بندی این داده‌ها وجود نداشته و باید چند خط استفاده شود تا بتوان با یک چندضعلی عمل طبقه‌بندی داده‌ها انجام شود که این هدف پرسش است که از Madaline بجای Adaline استفاده شود. در اصل به چند Adaline نیاز است.

کلاس مورد نظر پیاده‌سازی شد. پس از اجراهای مختلف به این نتیجه رسیدیم که برای این قسمت از مسئله برخلاف قسمتی قبلی، نرخ یادگیری ۰.۰۱ پایین است. پس این مدل را با نرخ یادگیری ۰.۱ آموزش دادیم. در نهایت نتایج زیر حاصل شد:

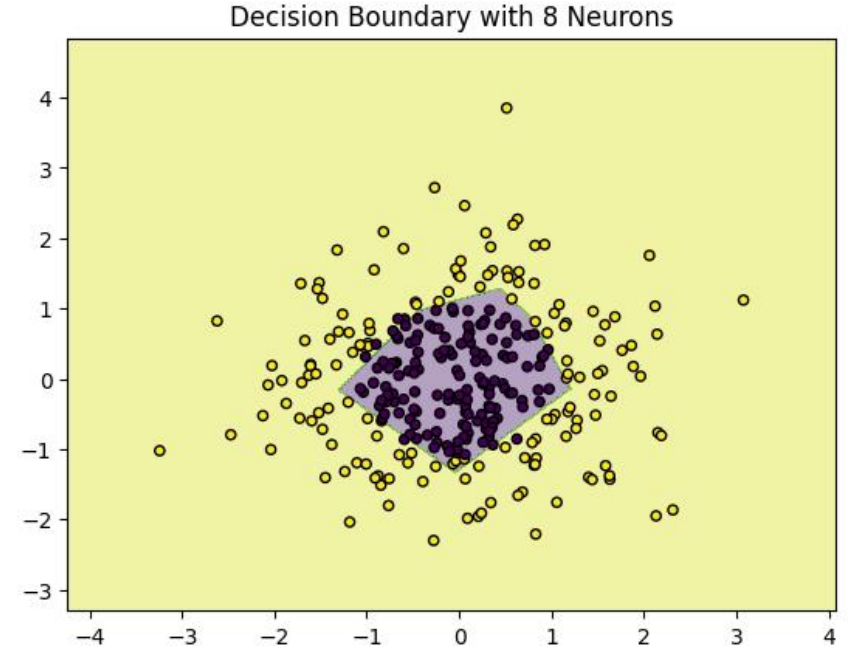
```
Accuracy for 3 neurons: 78.33%
Accuracy for 5 neurons: 95.00%
Accuracy for 8 neurons: 96.67%
```

مشاهده می‌شود که با افزایش تعداد نورون‌ها، دقت مدل افزایش پیدا می‌کند. دلیل این مورد این است که هر چه تعداد نورون‌ها بیشتر شود، با خط‌های بیشتری می‌توان ناحیه تصمیم را تنظیم کرد به نحوی که نقاط مربوطه را در بر بگیرد. وقتی از ۳ نورون استفاده می‌کنیم یعنی ۳ خط برای جداسازی استفاده می‌شود که به شکل یک مثلث خواهد بود:



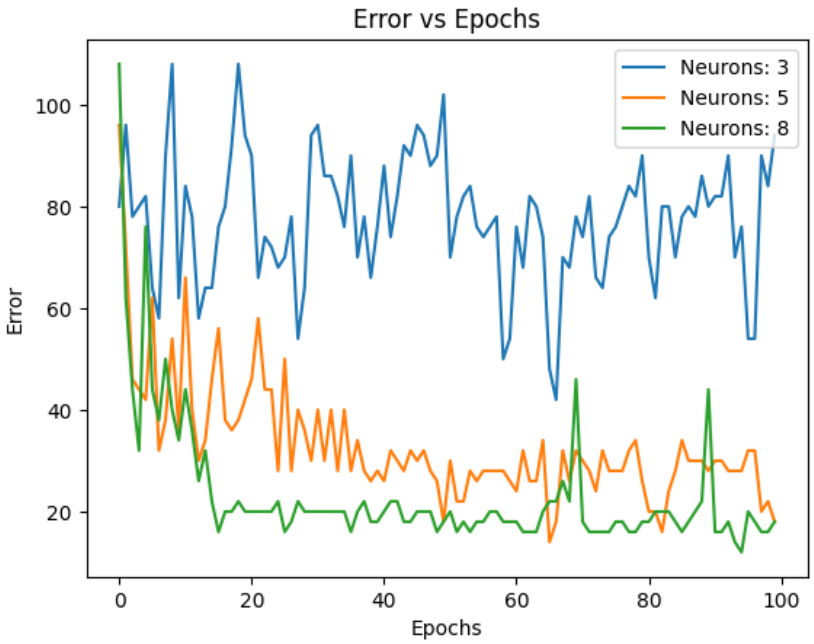
شکل ۲۸ ناحیه تصمیم با استفاده از ۳ نورون

ولی وقتی تعداد نورون‌ها افزایش پیدا کند، می‌توان با خطوط بیشتری مدل را آموزش داد و دقت نیز افزایش پیدا خواهدکرد که در شکل زیر برای تعداد نورون برابر ۸ این مورد مشاهده می‌شود:



شکل ۲۹ناحیه تصمیم با استفاده از ۵ نورون

در نهایت به بررسی تغییرات خطا می‌پردازیم:



شکل ۳۰ تغییرات خطا استفاده از تعداد نورون های ۳ و ۵ و۸ به نسبت **Epoch**

باتوجه به شکل بالا، تغییرات خطا در برای زمانی که از ۸ نورون استفاده می‌کنیم، وضعیت بهتری دارد و با تعداد ایپاک کمتری می‌توان به سطح بهتری از خطا دست یافت.

این مورد برای learning rate برابر ۰.۰۱ نیز بررسی شد ولی در این حالت، مسئله دچار بیش‌برازش می‌شد و نورون برابر ۸ دقت پایین تری نسبت به نورون برابر ۵ داشت. بنابراین جواب مسئله نهایی با نرخ یادگیری ۰.۱ بررسی شد.

پرسش ۴. شبکه عصبی بهینه

۴-۱. رگرشن

پاسخ قسمت الف)

بیش‌برازش زمانی اتفاق می‌افتد که مدل در فرایند یادگیری رفتار داده‌های آموزشی را بیش از حد حفظ می‌کند و برای داده‌های جدید قابلیت تعمیم نداشته باشد.یک دلیل آن می‌تواند این باشد که مدل نویز داده‌های آموزشی نیز به عنوان رفتار داده‌ها حفظ می‌کند. این مورد ممکن است زمانی پیش بیاید که پیچیدگی مدل نسبت به داده‌ها خیلی زیاد باشد. یکی دیگر از دلایل استفاده از داده‌های آموزشی نامناسب است. ممکن است یک سری داده آموزشی ارتباط خوبی با داده‌های جدید نداشته باشند.

پاسخ قسمت ب)

دربرابر دلایلی که گفته شد راه‌حل‌های متفاوتی وجود دارد که به حل مشکل بیش‌برازش کمک می‌کند. یکی از این روش‌ها Cross-Validation است که در آن داده‌های آموزشی به n تقسیم می‌شوند و هربار فرایند آموزش بر روی n-1 قسمت و یک قسمت تست متفاوت با قبلی انجام می‌شود. یکی دیگر از این موارد تعادل بین بایاس و واریانس برای تنظیم پیچیدگی مدل است. همچنین کاهش تعداد پارامترها بر این مورد اثرگذار است. در شبکه‌های عصبی برای کاهش پیچیدگی می‌توان از Dropout هم استفاده کرد که با غیرفعال‌سازی برخی نورون‌ها به صورت تصادفی در مرحله آموزش، باعث انتخاب ویژگی‌های بهتری می‌شود که به کاهش بیش‌برازش کمک می‌کنند.

یکی دیگر از موارد نیز که مربوط به تعداد و کیفیت داده‌های آموزشی است می‌توان با افزایش داده‌ها و روش Bootstrapping بهبود داد.

پاسخ قسمت پ)

هایپرپارامترها در شبکه عصبی مقادیری هستند که ساختار و رفتار کلی شبکه را تنظیم کرده و بر کارایی شبکه تاثیر بسیار زیادی دارند. این مقادیر قبل از شروع مرحله آموزش تنظیم می‌شوند و معمولا در اجرای هربار مرحله آموزش ثابت هستند. برای نمونه برخی از این هایپرپارامترها به صورت زیر هستند:

تعداد لایه‌های مخفی شبکه – تعداد نورون در هر لایه – نرخ یادگیری یا Learning rate – تعداد epoch یا تکرار مرحله آموزش

برای مقداردهی بهینه به آن‌ها یا اصطلاحا Tuning هایپرپارامترها، از روش‌های مختلفی می‌توان استفاده کرد. برای نمونه در روش انتخاب تصادفی یا Random search، ترکیب‌های مختلفی از مقادیر انتخاب شده و این مقادیر آزمایش می‌شوند. روش دیگر Bayesian optimization است. در این روش که بر اساس تئوری بیز عمل می‌کند، با کمک دانش موجود یک مدل احتمالاتی ارائه می‌دهد که یک مجموعه از هایپرپارامترها به نحوی تنظیم شوند که یک مقدار خاص بهینه شود. روش دیگر Grid Search است که در این روش یک لیست از هایپرپارامترها و میزان کارایی آن‌ها ارائه شده و الگوریتم با آزمایش تمامی ترکیب‌های ممکن بهترین مجموعه از هایپرپارامترها را انتخاب می‌کند.

از طرف دیگر پارامترها مقادیر داخلی شبکه هستند که در مرحله آموزش سعی در بهینه‌کردن آن‌ها داریم. برای نمونه بردار وزن‌ها و بایاس در شبکه جز پارامترها محسوب می‌شوند. تفاوت اساسی آن‌ها با هایپرپارامترها این است که هایپرپارامترها قبل از اجرای مرحله آموزش تعیین می‌شوند و به کمک آن‌ها سعی در تنظیم فرایند آموزش داریم در حالی که پارامترها با کمک داده‌ها تنظیم می‌شوند.

پاسخ قسمت ت)

افزایش داده‌ها

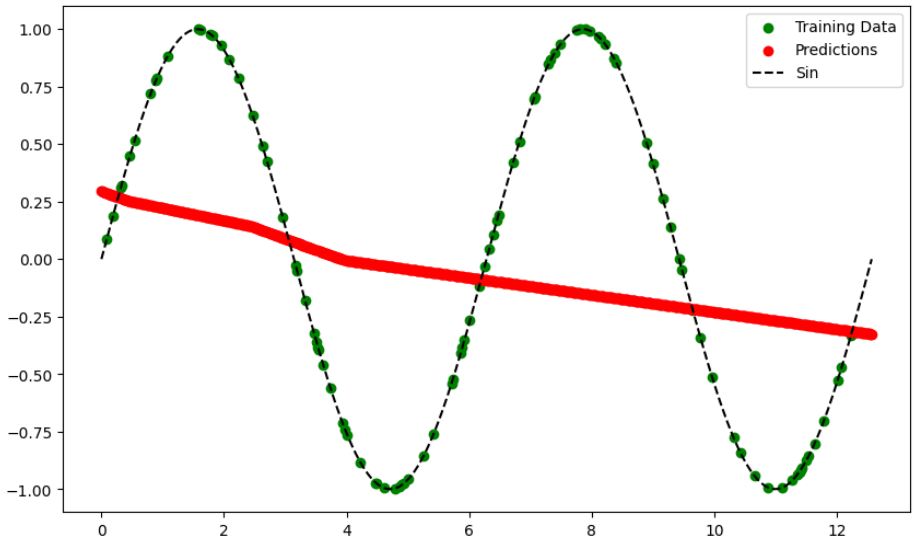
در مرحله اول ۱۰ درصد داده‌ها یا ۱۰۰ عدد از آن‌ها به عنوان داده آموزشی انتخاب شد. پس از آموزش مدل رگرشن نتیجه زیر حاصل شد:

برای این سوال و بررسی نتیجه رگرشن ۲ مورد بررسی می‌شود. یکی MSE که همان mean square error است و برابر با میانگین مربعات خطا است. و مورد دیگر R2 است که نشان می‌دهد در مدل چقدر از تغییرات متغیر وابسته تحت تاثیر متغیر مستقل است.

خروجی با ۱۰ درصد داده آموزشی به صورت زیر است:

```
MSE : 41.47% R2 : 17.06%
```

شکل زیر نتیجه اولین اجرا را نشان می‌دهد. نقطه‌های قرمز پیش بینی مدل بر اساس داده‌های تست هستند.



شکل ۳۱مدل رگرشن با کمک ۲ لایه و ۱۰ درصد داده‌ها

همانطور که در شکل مشخص است، این مدل رگرشن نتیجه خوبی ندارد و داده‌های پیش‌بینی شده تفاوت زیادی با داده‌های اصلی دارند. در این شبکه از ۲ لایه مخفی استفاده شده است.

حال در ادامه تعداد داده‌های آموزش افزایش را افزایش داده و مشاهده می‌شود که MSE کاهش یافته و R2 افزایش می‌یابد. ولی این مورد تا ۶۰ درصد داده‌ها باعث افزایش می‌شود و پس از آن R2 کاهش می‌یابد. این نتیجه قابل پیش‌بینی نبود. مجددا همین مدل با ۱ لایه مخفی نیز بررسی شد که تا ۹۰ درصد داده‌ها R2 افزایش می‌یافت. ولی با ۲ لایه مخفی در ۲ افزایش داده خطوط به نمودار سینوس نزدیک شده و نتیجه به صورت زیر حاصل شد:

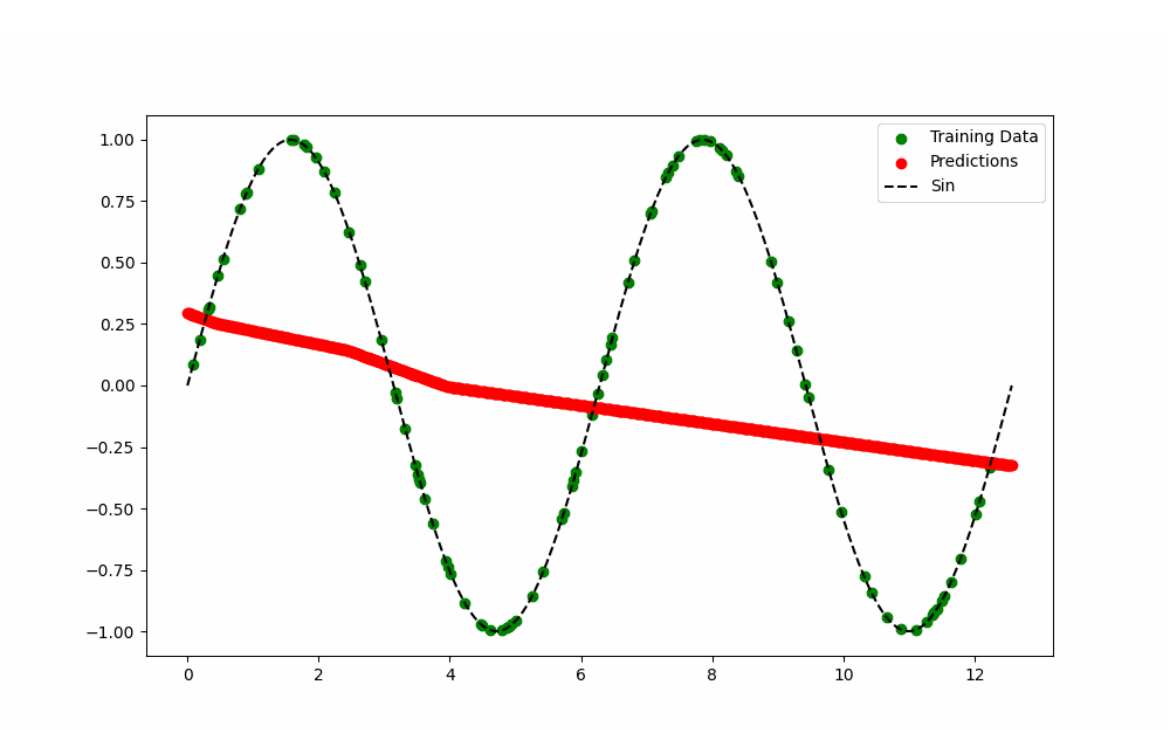
```
MSE : 8.75% R2 : 82.26%
```

ولی با افزایش داده بیشتر این مورد کاهش یافت:

```
MSE : 35.65% R2 : 23.78%
```

پس می‌توان نتیجه گرفت افزایش بیش از حد داده‌ها همیشه باعث افزایش کارایی مدل نمی‌شود و ممکن است بیش‌برازش اتفاق بیفتد.

نمودار تغییرات با افزایش داده‌ها به صورت Gif پیوست شد:



شکل ۳۲تغییرات مدل رگرسیون با افزایش داده‌ها به صورت **Gif**

افزایش لایه‌ها

در این قسمت از مسئله تمام مراحل قبلی مدل رگرسیون تکرار می‌شود ولی اینبار به جای افزایش تعداد داده‌ها، تعداد لایه‌ها افزایش پیدا می‌کند. هر لایه در این مسئله ۱۰ نورون دارد. در ابتدا و استفاده از یک لایه نتیجه حاصل به صورت زیر است:

```
MSE : 36.30% R2 : 21.67%
```

پس از افزایش تعداد لایه‌ها کارایی مدل افزایش می‌یابد به صورتی که در صورت استفاده از ۸ لایه نتیجه زیر حاصل شد:

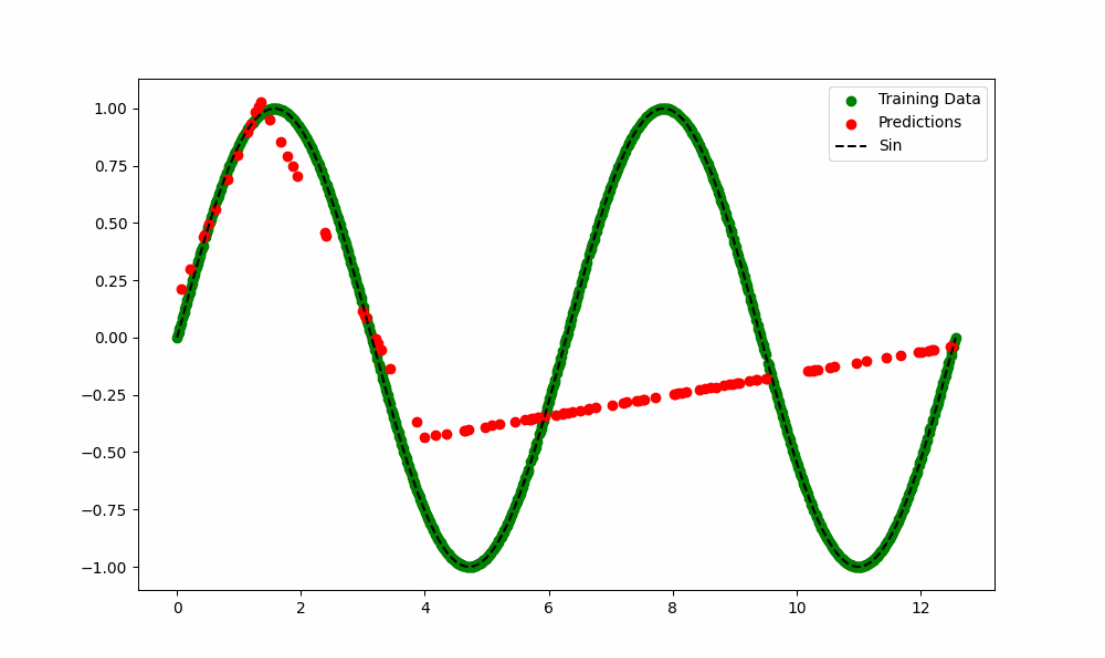
```
MSE : 0.34% R2 : 99.26%
```

ولی پس از آن یک کاهش ۲-۳ درصدی در استفاده از ۱۰ لایه حاصل شد. این افزایش و کاهش همینطور ادامه پیدا کرد به صورتی که در صورت استفاده از ۲۰ لایه R2 کاهش محسوسی داشت:

```
MSE : 2.31% R2 : 95.02%
```

می‌توان نتیجه گرفت که افزایش بیش از حد لایه‌ها همیشه باعث افزایش کارایی مدل نمی‌شود و تعداد لایه‌ها باید به اندازه‌ای باشد که هم دقت مدل خوب باشد و هم پیچیدگی آن زیاد نباشد. هرچه که تعداد لایه‌ها بیشتر باشد، محاسبات هم بیشتر خواهدشد که در این آزمایش مشخص شد این افزایش محاسبات نه تنها به بهبود مدل کمک نمی‌کند، بلکه باعث کاهش R2 و افزایش MSE شد.

نمودار تغییرات با افزایش داده‌ها به صورت Gif پیوست شد:



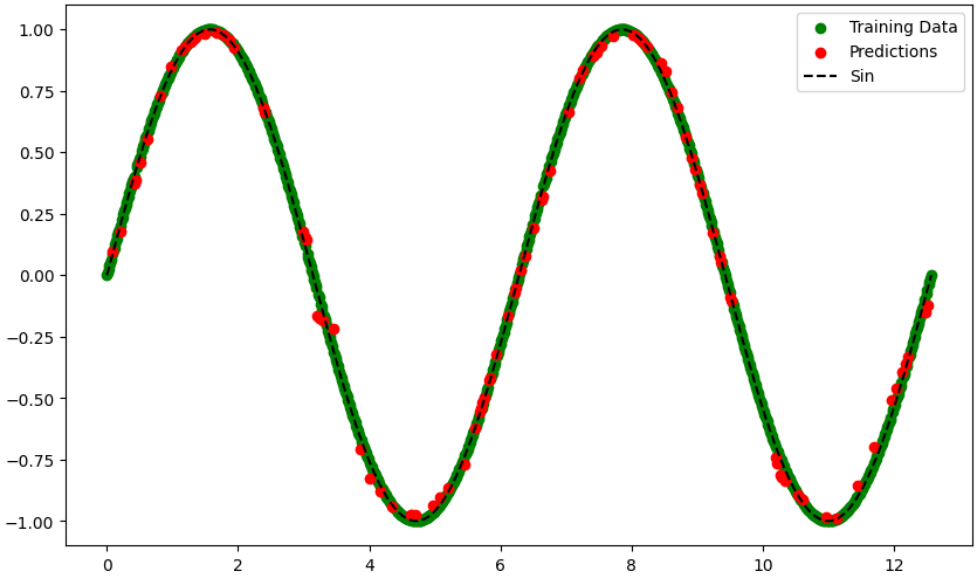
شکل ۳۳تغییرات مدل رگرسیون با افزایش لایه‌ها به صورت **Gif**

پاسخ قسمت ث)

در قسمت آخر این پرسش از GridSearchCV کتابخانه Sklearn برای یافتن بهترین تعداد لایه‌ها برای مسئله بالا استفاده می‌کنیم. خروجی به صورت زیر حاصل شد:

```
Layers =  19
MSE : 0.10% R2 : 99.78%
```

همانطور که در قسمت قبل هم اشاره شد، افزایش ۲۰ لایه لزوما بهترین جواب را نمی‌دهد و ۱۹ لایه نتیجه بهتری خواهد داشت:



شکل ۳۴ استفاده از ۱۹ لایه برای مسئله رگرسیون

این روش زمانی که تعداد هایپرپارامترها زیاد باشد، محاسبات زیاد و سنگینی خواهدداشت. در این صورت بهتر است روش Random Search هم بررسی شود. همچنین اگر Grid بزرگ باشد و داده‌های Validation کم باشند، امکان بیش‌برازش افزایش خواهدیافت. در این موارد این تکنیک ممکن است مناسب نباشد.

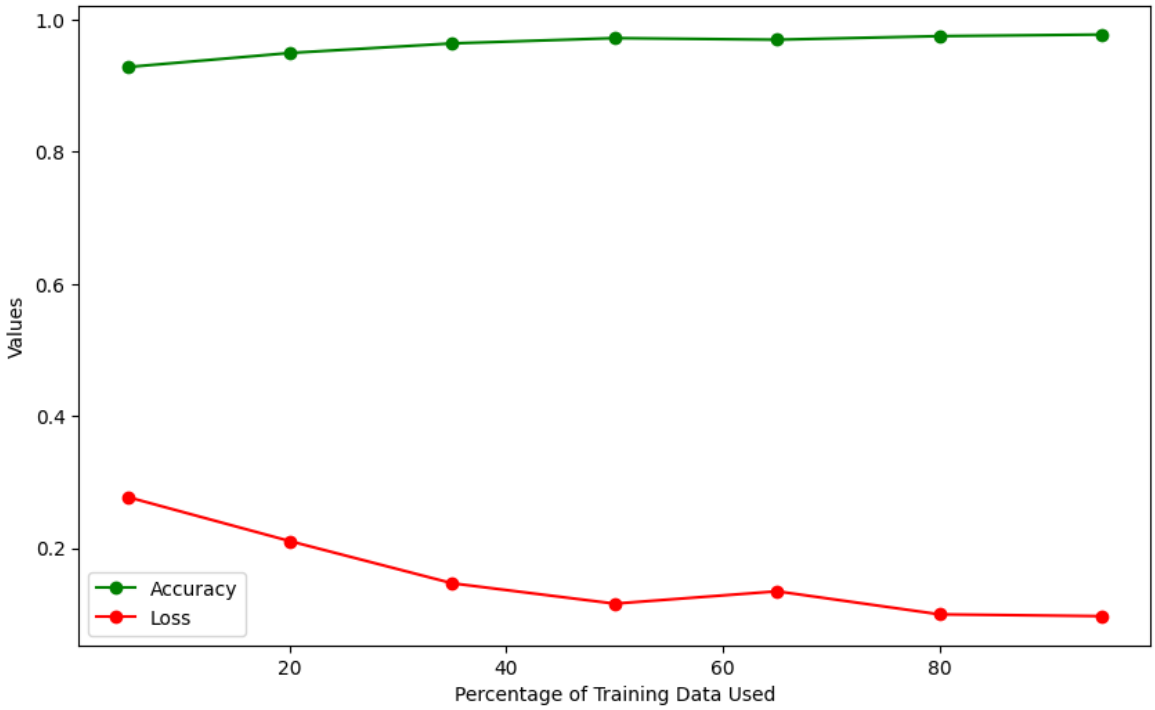
۴-۱. طبقه‌بندی

در قسمت قبلی مشکل بیش‌برازش در رگرسیون بررسی شد. در این مسئله نتیجه افزایش داده و افزایش لایه در مسئله طبقه‌بندی داده‌های MNIST را بررسی می‌کنیم.

افزایش داده‌ها

ابتدا دیتاست مورد نظر بارگذاری شد و پیش‌پردازش‌های لازم انجام شد.

در مرحله اول از این مسئله هربار ۱۵ درصد به داده‌های آموزشی اضافه می‌شود که این مورد در یک حلقه هربار انجام شده و خروجی مدل گزارش می‌شود. نتیجه به صورت زیر است:



شکل ۳۵تغییرات دقت و **loss** نسبت به درصد داده‌های آموزشی

مشاهده می‌شود که پس از استفاده از درصد خاصی از داده‌ها بهبود قابل توجهی حاصل نمی‌شود ولی به طور کلی روند افزایشی است ولی یک استثنا وجود دارد و در ۶۵ درصد است. در این نقطه کارایی کاهش یافته است. پس می‌توان نتیجه گرفت همیشه با افزایش داده‌ها دقت افزایش پیدا نمی‌کند و ممکن است این مورد کاهش یافته و بیش‌برازش رخ دهد.

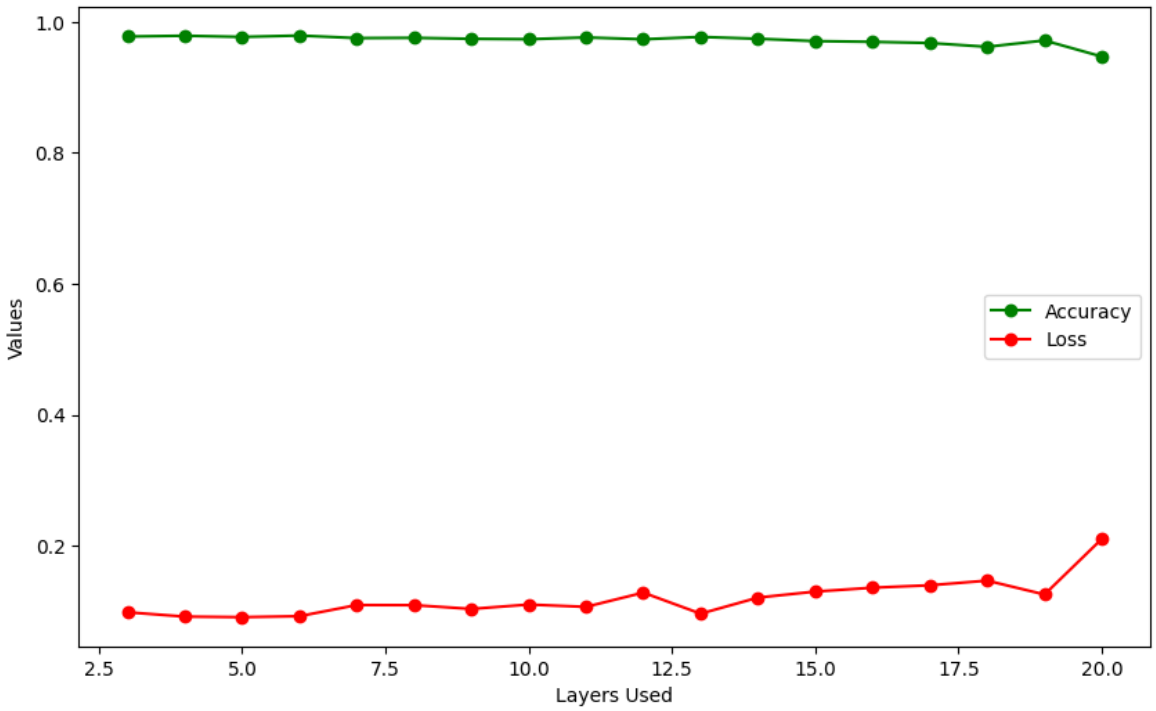
خروجی هر مرحله در نوت‌بوک قرار دارد.

افزایش لایه‌ها

در این قسمت از مسئله همان مدل قبلی ولی اینبار با تمام داده‌ها را آموزش داده و یکی یکی لایه‌ها را افزایش می‌دهیم.

پس از افزایش لایه‌ها مشاهده می‌شود که تغییر محسوسی در افزایش دقت حاصل نمی‌شود و دقت در حدود ۹۷ درصد باقی می‌ماند. از ۶ لایه به بعد دقت روند کاهشی هم دارد که این مورد احتمالا به دلیل افزایش پیچیدگی است. این مورد در لایه ۱۶م محسوس می‌شود و دقت حدود ۱ درصد نسبت به مراحل

اول با لایه کمتر کاهش دقت داشته است. در ۲۰ لایه کمترین دقت را داریم که حدود ۹۴ درصد می‌شود. پس نتیجه می‌گیریم که همیشه افزایش لایه‌ها به افزایش دقت کمک نمی‌کند. این مورد در هر مسئله متفاوت است و بستگی به مسائل دیگر هم دارد. ولی به طور کلی باید آزمایش شود چه تعداد لایه برای مسئله مناسب بوده که علاوه بر افزایش دقت موجب افزایش پیچیدگی بیش از حد و بیش‌برازش نشود.



شکل ۳۶ تغییرات دقت و **loss** نسبت به افزایش لایه‌ها

پاسخ قسمت ب)

برای پیاده‌سازی این مدل پس از جستجو یک مدل بر اساس CNN یافت شد که مربوط به فصل بعدی است ولی از آنجا که محدودیتی در این مورد در صورت سوال ذکر نشده بود از آن استفاده شد. لینک گیت‌هاب و مقاله مربوط به این پیاده‌سازی در زیر آورده شده است. دقت این مدل ۹۹.۹ درصد گزارش شده است.

https://paperswithcode.com/paper/an-ensemble-of-simple-convolutional-neural

https://github.com/ansh941/MnistSimpleCNN

پس از پیاده‌سازی آن و اجرای فقط ۱ ایپاک دقت زیر حاصل شد. به دلیل زمان زیاد برای اجرا ۱ ایپاک اجرا شد:

Test accuracy: 0.9908999800682068, Test loss: 0.03042251616716385