به نام خدا

تمرین ششم یادگیری عمیق

غزل زمانی‌نژاد

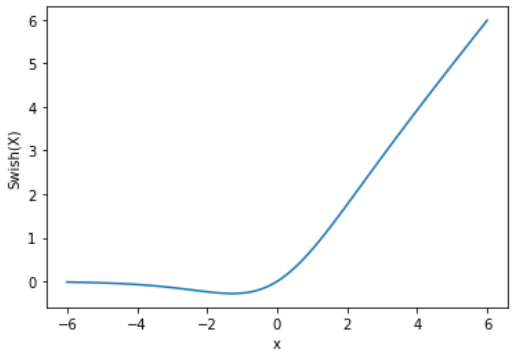
۹۷۵۲۲۱۶۶

1. الف) Swish:

f(x) = x * sigmoid(x)

تابع sigmoid و swish را تعریف میکنیم و بعد به کمک matplotlib آن را رسم میکنیم.

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

def sigmoid(x):
  return 1 / (1 + np.exp(-x))

def swish(x):
  return x * sigmoid(x)

x = np.linspace(-6, 6, 100)
y = swish(x)

plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("Swish(X)")

plt.show()
```

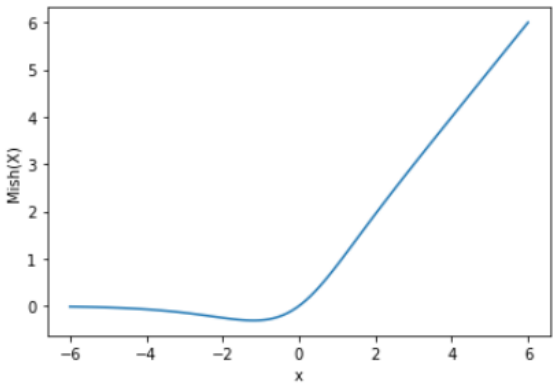


Mish:

f(x) = x * tanh(softplus(x)) = x * tanh(ln(1 + e^x))

تابع softplus و mish را تعریف میکنیم و بعد به کمک matplotlib آن را رسم میکنیم.

```python
def softplus(x):
    return np.log(1 + np.exp(x))

def mish(x):
    return x * np.tanh(softplus(x))

# x2 = np.linspace(-3, 3, 100)
y2 = mish(x)

plt.plot(x, y2)
plt.xlabel("x")
plt.ylabel("Mish(X)")

plt.show()
```



ب) ابتدا مشتق swish را محاسبه میکنیم:

$$swish(x) = x \times \sigma(x)$$

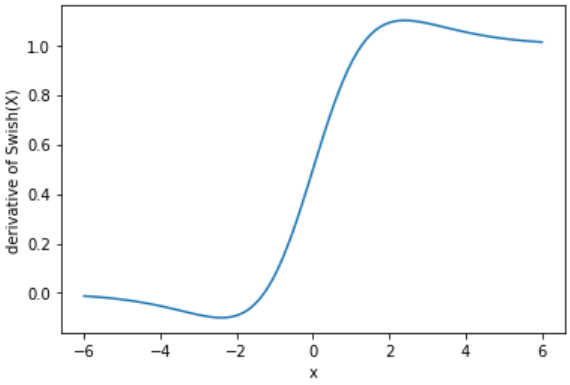
$$\xrightarrow{\text{derivative}} \quad 1\,\sigma(x) + x\left(\sigma(x)\,(1-\sigma(x))\right) = \sigma(x) + x\sigma(x) + x\sigma^2(x)$$

$$= \underbrace{x\sigma(x)}_{swish(x)} + \sigma(x)\left[1 - \underbrace{x\sigma(x)}_{swish(x)}\right]$$

$$= swish(x) + \sigma(x)\,(1 - swish(x))$$

سپس آن را رسم میکنیم:

```python
y2 = swish(x) + sigmoid(x) * (1 - swish(x))

plt.plot(x, y2)
plt.xlabel("x")
plt.ylabel("derivative of Swish(X)")

plt.show()
```



ابتدا مشتق mish را محاسبه میکنیم:

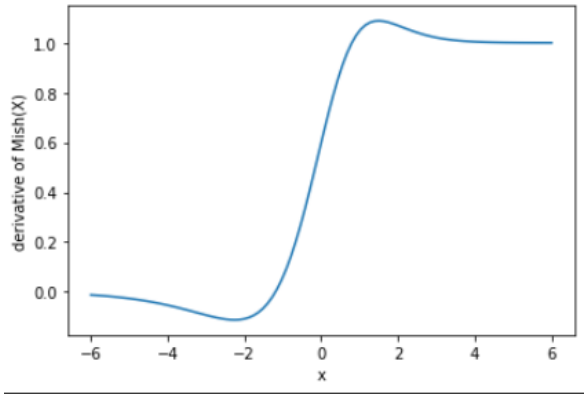
$$mish(x) = x \tanh(softplus(x))$$

$$\xrightarrow{\text{derivative}} \quad \tanh(softplus(x)) + x\underbrace{(1-\tanh^2(softplus(x)))}_{sech^2(softplus(x))}\frac{e^x}{1+e^x} =$$

$$= \frac{mish(x)}{x} + x \cdot sech^2(softplus(x))\, sigmoid(x)$$

سپس آن را رسم میکنیم:

```
y4 = mish(x) / x + x * (1 - np.tanh(softplus(x))**2) * sigmoid(x)

plt.plot(x, y4)
plt.xlabel("x")
plt.ylabel("derivative of Mish(X)")

plt.show()
```



ت) مزیت ReLU نسبت به sigmoid و tanh: در شبکه های عمیق، ReLU نسبت به sigmoid و tanh عملکرد بهتری دارد زیرا با مشکل vanishing gradients یعنی کوچک شدن بیش از حد مقادیر گرادیان ها رو به رو نمیشویم و برای مقادیر مثبت، گرادیان در شبکه جریان پیدا میکند. به دلیل اینکه ReLU از بالا بیکران است، از saturation برای مقادیر مثبت ورودی جلوگیری میکند. دلیل دیگر برتری ReLU آن است که تعمیم بهتری دارد و سرعت همگرایی آن بیشتر است. همچنین بسیاری از متخصصین سادگی و قابل اطمینان بودن ReLU را به سایر توابع فعالسازی ترجیح میدهند زیرا در بسیاری از موارد بهبود عملکرد سایر توابع به مدل و دیتاست وابسته است.

Swish کدام مشکلات ReLU را رفع کرده و چه شباهت هایی با آن دارد؟: این تابع همچون ReLU از بالا بیکران و از پایین کران دار است. اما برخلاف ReLU غیریکنواخت است. تمامی ویژگی های swish یعنی کران دار از پایین، بی کران از بالا، غیریکنواختی و هموار بودن آن مزیت محسوب میشوند. به بررسی آنها میپردازیم:

بیکران بودن از بالا یک ویژگی مهم محسوب میشود چون مشکل saturation آموزش کند برای گرادیان های نزدیک 0 را برطرف کرده.

کران دار بودن از پایین اثر قوی regularization دارد. توابعی که حد آنها 0 است تاثیر به سزای regularization دارند زیرا مقادیر منفی بزرگ در آنها فراموش میشود.

swish برخلاف ReLU برای ورودی های منفی کوچک، مقداری منفی است و باعث غیریکنواختی میشود. همین امر باعث تقویت جریان یافتن گرادیان ها میشود. در تابع ReLU، مقدار خروجی به ازای تمامی مقادیر منفی 0 است. اگر شبکه به جایی برسد که همیشه مقادیر منفی داشته باشد یعنی ReLU به طور مداوم مقدار 0 را برگرداند، باعث بروز پدیده dying ReLU میشود و شبکه قدرت طبقه بندی بین داده ها را از دست میدهد. با استفاده از swish از این پدیده جلوگیری میشود. همچنین باعث بالا بردن robustness در برابر مقداردهی های اولیه پارامترها و نرخ یادگیری های متفاوت میشود.

هموار بودن (smoothness) نقش مهمی در بهینه سازی و تعمیم دارد. همواری خروجی بر همواری میزان ضرر تاثیر مستقیم میگذارد. اگر ضرر هموار باشد در بهینه سازی راحت تر است و حساسیت مقداردهی اولیه پارامترها و نرخ یادگیری کاهش می یابد.

مقایسه در batch sizeهای مختلف: همانطور که انتظار میرود با افزایش اندازه batch، میزان عملکرد ReLU و swish کاهش می یابد. با این وجود برای batch sizeهای متفاوت (چه کوچک و چه بزرگ) swish عملکرد بهتری دارد.

Mish کدام مشکلات ReLU را رفع کرده و چه شباهت هایی با آن دارد؟:

تابع mish هموار، پیوسته، self-regularized و غیریکنواخت است. همچون swish از بالا بی کران و از پایین کران دار است (تمامی مزیت های swish را دارد).

Mish با داشتن یک پیش شرط یعنی همان Δ(x) از dying ReLU جلوگیری می کند. و باعث جریان یافتن بهتر گرادیان در شبکه می شود.

بی کران بودن از بالا از saturation جلوگیری می کند (وجود saturation موجب آهسته شدن فرآیند یادگیری در گرادیان های نزدیک 0 می شود).

با وجود این موارد، در صورت استفاده از تابع فعال سازی mish یک trade-off میان بالا رفتن دقت و بالا رفتن هزینه های محاسباتی وجود دارد.

ث) اگر پارامترهای swish را به شکل زیر تغییر دهیم به ReLU شباهت بیشتری پیدا میکند:

f(x, beta) = 2x * sigmoid(beta * x)

در این تابع اگر beta = 0 باشد، swish = x و خطی میشود.

اگر beta → inf، سیگموید به تابع 0–1 تبدیل میشود و swish شبیه ReLU میشود.

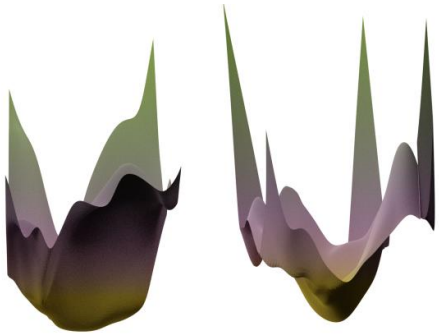
اگر beta به عنوان یک پارامتر قابل آموزش تعیین شود، میتوانیم درجه برون یابی مدل را با آن کنترل کنیم. به این گونه از تابع swish بدون ضریب 2، swish-beta میگوییم. مزیت تابع swish-beta این است که میتواند میان تابع خطی و ReLU تغییر کند.

ج) تابع mish بسیار به تابع swish شباهت دارد. در مشتق اول تابع mish، ضریبی از تابع swish وجود دارد:

$$f'(x) = sech^2(softplus(x))xsigmoid(x) + \frac{f(x)}{x}$$

$$= \Delta(x)swish(x) + \frac{f(x)}{x}$$

Δ(x) موجود در این رابطه باعث هموارتر شدن گرادیان ها نسبت به گرادیان های تابع swish می شود. این ضریب اثر regularization دارد و باعث میشود بهینه سازی کانتور بهتر انجام شود. و در نهایت موجب می شود تابع mish در شبکه های عمیق و پیچیده نسبت به swish برتری داشته باشد. شکل زیر به خوبی نشان دهنده این برتری است. در سمت راست تابع ضرر با تابع فعال سازی swish و در سمت چپ تابع ضرر با تابع فعال سازی mish دیده می شود. تصویر سمت چپ (mish) هموارتر است.



left: mish, right: swish

2. **الف) فرمول تابع ضرر MSE:**

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

فرمول تابع ضرر Binary Cross Entropy:

$$\mathrm{Loss} = -\frac{1}{\underset{\text{size}}{\text{output}}}\sum_{i=1}^{\underset{\text{size}}{\text{output}}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

همانطور که در فرمول های فوق مشاهده می شود، مقدار تابع MSE حداکثر می تواند 1 باشد و به دلیل کوچک بودن مقادیر گرادیان دیرتر بهینه می شود. اما Binary Cross Entropy نسبت به خطا بسیار حساس است و حتی مقدار loss می تواند به بی نهایت میل کند. به همین دلیل در epoch اول میان این دو مقدار اختلاف دیده می شود. در مسائل دسته بندی هدف بالا بردن مقدار دقت است و نه کم کردن میزان ضرر. یعنی با مقایسه نمودارهای loss نمی توانیم بگوییم کدام یک عملکرد بهتری داشته است.

در ابتدای کار که هنوز شبکه پارامترها را یاد نگرفته، خروجی 0.5 است (مستقل از داده شبکه احتمال 50 درصد را پیش بینی می کند). خطای MSE را محاسبه می کنیم (اختلاف بین پیش بینی شبکه و مقدار واقعی در هر صورت 0.5 است):

MSE = (1 / n) (0.5)^2 = 0.25

خطای Binary Cross Entropy را محاسبه می کنیم:

Y = 1: Binary Cross Entropy = -1 * ln (0.5) – 0 = ln(2) = 0.69

Y = 0: Binary Cross Entropy = 0 – (1 – 0) * ln (1 - 0.5) = ln(2) = 0.69

مشاهده می شود این مقادیر با مقادیر نمودارها در epoch نخست یکی است.

ب) همانطور که در نمودار دیده می شود، تابع MSE پس از 100 epoch همچنان آموزش می بیند و به همگرایی نرسیده به همین دلیل اختلاف میان نمودار اعتبارسنجی و آموزش کمتر است. اما در نمودار Binary Cross Entropy بعد از آموزش دیدن در تعدادی epoch، شبکه به سمت overfit شدن می رود. یعنی الگوها را بیشتر حفظ می کند و آنها را به درستی یاد نمی گیرد تا تعمیم دهد. به همین دلیل در نمودار سمت راست، بعد از مدت خطای آموزش کاهش می یابد اما خطای اعتبارسنجی تقریبا ثابت می ماند و میان این دو نمودار فاصله ایجاد می شود.

ج) Binary Cross Entropy: در این نمودار تقریبا بعد از 60 epoch مشاهده می شود که مقدار خطای اعتبارسنجی رو به افزایش می رود. یعنی در آن نقطه شبکه به سمت overfit شدن می رود. پس بهتر است آموزش را متوقف کنیم تا شبکه الگوها را حفظ نکند و بتواند تعمیم دهد.

MSE: در این نمودار هم خطای آموزش و هم خطای اعتبارسنجی همچنان در حال کاهش هستند. یعنی شبکه هنوز underfit است و مشکل بایاس داریم. پس بهتر است همچنان به آموزش ادامه دهیم تا زمانی که خطای اعتبارسنجی به کمترین مقدار خود برسد (سرعت یادگیری در MSE نسبتا کم است و به همین دلیل بعد از epoch100 شبکه هنوز در حال آموزش دیدن است).

3. سلول اول: دیتاست MNIST که شامل رقم های 0 تا 9 است از طریق کراس load می کنیم و شکل داده های آزمون و تست را چاپ می کنیم.

```
import numpy as np
import matplotlib.pyplot as plt
import random
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

print("train input shape", x_train.shape)
print("train label shape", y_train.shape)
print("test input shape", x_test.shape)
print("test label shape", y_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
train input shape (60000, 28, 28)
train label shape (60000,)
test input shape (10000, 28, 28)
test label shape (10000,)
```

سلول دوم: برای اینکه داده های آزمون بیشتری در اختیار داشته باشیم، از data augmentation استفاده می کنیم. در اینجا رنگ هر پیکسل را invert می کنیم. یعنی اگر سیاه باشد آن را سفید، و اگر سفید باشد آن را سیاه می کنیم. با استفاده از این کار تعداد داده ها دو برابر می شود. پس از تولید تصاویر جدید، آنها را به دیتاست اصلی اضافه می کنیم (concatenate).

```
# data augmentation
x_train_aug = np.subtract(255, x_train)
x_test_aug = np.subtract(255, x_test)

new_x_train = np.concatenate((x_train, x_train_aug), axis=0)
new_y_train = np.concatenate((y_train, y_train), axis=0)
new_x_test = np.concatenate((x_test, x_test_aug), axis=0)
new_y_test = np.concatenate((y_test, y_test), axis=0)
```

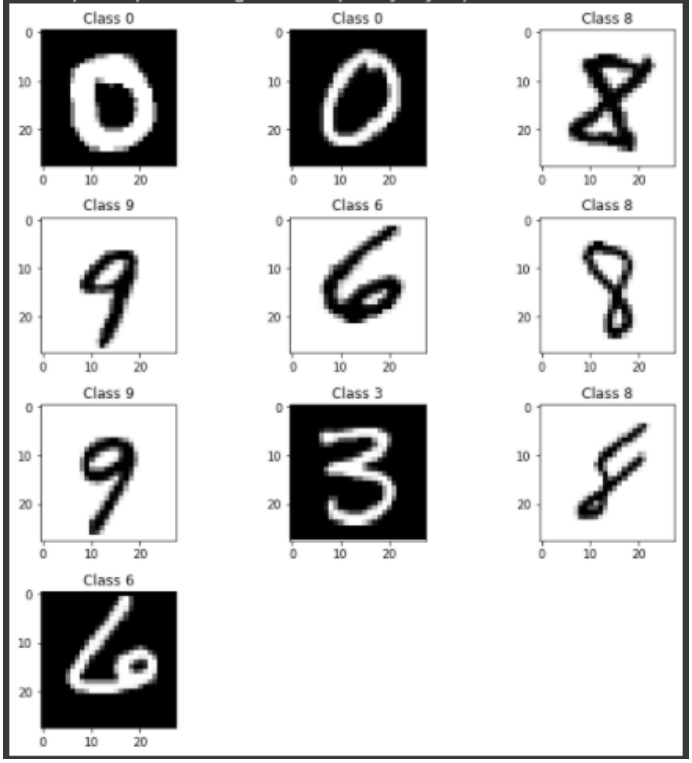
سلول سوم: داده های آموزشی و آزمون را shuffle می کنیم تا در هر مجموعه داده ها ترتیب خاصی نداشته باشند. در اینجا آرگومان random_state را عدد خاصی تنظیم می کنیم تا برچسب متناظر با هر داده اشتباه نشود و لیست برچسب ها هم به همان صورت ترکیب شوند.

```
from sklearn.utils import shuffle
new_x_train, new_y_train = shuffle(new_x_train, new_y_train, random_state=10)
new_x_test, new_y_test = shuffle(new_x_test, new_y_test, random_state=8)
```

سلول چهارم: پس از مخلوط کردن داده ها شکل آنها و همچنین 10تا از داده های ورودی با برچسب اصلی شان را چاپ می کنیم.

```
print("train input shape after augmentation", new_x_train.shape)
print("test input shape after augmentation", new_x_test.shape)

plt.rcParams['figure.figsize'] = (9,9) # Make the figures a bit bigger

for i in range(10):
  plt.subplot(4,3,i+1)
  plt.imshow(new_x_train[i], cmap='gray', interpolation='none')
  plt.title("Class {}".format(new_y_train[i]))

plt.tight_layout()
```

```
train input shape after augmentation (120000, 28, 28)
test input shape after augmentation (20000, 28, 28)
```



سلول پنجم: برای اینکه داده های ورودی را از لایه کانولوشنی عبور دهیم، باید شکل آنها را تغییر دهیم. برای این کار از reshape استفاده می کنیم.

```
shape1 = new_x_train.shape
shape2 = new_x_test.shape
new_x_train = np.reshape(new_x_train, (shape1[0], shape1[1], shape1[2], 1))
new_x_test = np.reshape(new_x_test, (shape2[0], shape2[2], shape2[2], 1))

print("train input shape after reshape", new_x_train.shape)
print("test input shape after reshape", new_x_test.shape)
```

```
train input shape after reshape (120000, 28, 28, 1)
test input shape after reshape (20000, 28, 28, 1)
```

سلول ششم: برای اینکه پراکندگی داده های ورودی زیاد نباشد، آنها را normalize می کنیم. یعنی مقدار تمامی پیکسل های تمام تصاویر را بر 255 تقسیم می کنیم. سپس برچسب هر تصویر را به شکل one hot با استفاده از to_categorical انکود می کنیم. ورودی اول این تابع برداری است که میخواهیم آن را انکود کنیم و ورودی دوم تعداد کل کلاس هاست که در اینجا 10 رقم داریم پس تعداد کلاس ها نیز 10 است.

```
# normalized = (x - min) / (max - min)
# in this example = x / 255
import tensorflow as tf

new_x_train = new_x_train / 255
new_x_test = new_x_test / 255

new_y_train = tf.keras.utils.to_categorical(new_y_train, num_classes=10)
new_y_test = tf.keras.utils.to_categorical(new_y_test, num_classes=10)
```

سلول هفتم: تابع sequential_network را تعریف می کنیم. این تابع آلفا را به عنوان ورودی می گیرد و در خروجی مدل از نوع sequential را برمی گرداند. این مدل دارای لایه های زیر است:

لایه کانولوشن اول که دارای تابع فعال سازی Leaky ReLU با پارامتر آلفا است.

لایه کانولوشن دوم که مشابه لایه کانولوشن اول است و تنها kernel size آن با لایه اول متفاوت است.

لایه Flatten که خروجی های لایه قبل را flat می کند.

لایه Dense که خروجی نهایی شبکه (10 نورون) را تشکیل می دهد. دارای تابع فعال سازی softmax است.

```python
from keras.models import Sequential
from keras.layers import Dense, Conv2D, LeakyReLU, Flatten, PReLU

def sequential_network(alpha):
  leaky_relu = LeakyReLU(alpha=alpha)
  model = Sequential()
  model.add(Conv2D(8, 7, activation=leaky_relu))
  model.add(Conv2D(8, 5, activation=leaky_relu))
  model.add(Flatten())
  model.add(Dense(10, activation='softmax'))

  return model
```

برای اینکه از درست بودن لایه های شبکه مطمئن شویم یک نمونه از آن را می سازیم و آن را build می کنیم. سپس خلاصه شبکه را چاپ می کنیم.

```python
test_model = sequential_network(0.3)
test_model.build(new_x_train.shape)
print(test_model.summary())
```

```
Model: "sequential_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_52 (Conv2D)           (120000, 22, 22, 8)       400
_________________________________________________________________
conv2d_53 (Conv2D)           (120000, 18, 18, 8)       1608
_________________________________________________________________
flatten_26 (Flatten)         (120000, 2592)            0
_________________________________________________________________
dense_26 (Dense)             (120000, 10)              25930
=================================================================
Total params: 27,938
Trainable params: 27,938
Non-trainable params: 0
_________________________________________________________________
None
```

<u>سلول هشتم</u>: در آن 3 متد تعریف می کنیم:

Train: **به عنوان ورودی مدل را دریافت می کند. آن را با شرایط خواسته شده یعنی تابع ضرر** Categorical Cross Entropy **و بهینه ساز** Adam **کامپایل می کنیم. سپس آن را در پنج** epoch **و با اندازه** 128 batch **با استفاده از تابع** fit **آموزش می دهیم.**

```
epoch_size = 5
batch_size = 128


def train(model):
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  history = model.fit(new_x_train, new_y_train, epochs=epoch_size, batch_size=batch_size)
  return history
```

Test: مقدار دقت و خطای روی داده های آزمون را محاسبه کرده و باز می گرداند.

```
def test(model):
  loss, acc = model.evaluate(new_x_test, new_y_test, verbose = 0)
  return (loss, acc)
```

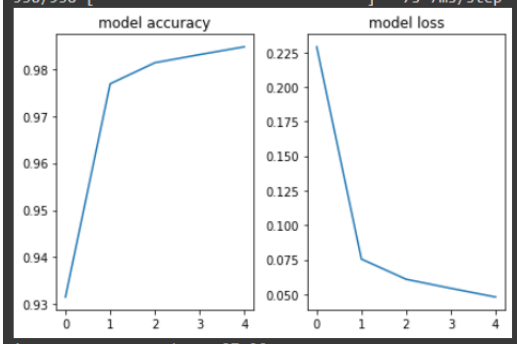
Plot: به عنوان ورودی تاریخچه ای از مدل را دریافت می کند تا بتواند نمودار میزان دقت و خطای مدل را در 2 subplot رسم کند.

```
def plot(history):
  figure, (ax1, ax2) = plt.subplots(1, 2)

  ax1.plot(history.history['accuracy'])
  ax1.set_title('model accuracy')

  ax2.plot(history.history['loss'])
  ax2.set_title('model loss')
  figure.tight_layout()
  plt.show()
```

سلول نهم: مقادیر آلفا داده شده را در یک لیست میریزیم تا به کمک یک حلقه مدل ها با آلفای متفاوت را آموزش دهیم و نتیجه را چاپ کنیم. در این حلقه: ابتدا با استفاده از تابع sequential_network ابتدا یک مدل با آلفای مشخص می سازیم. سپس با استفاده از تابع train آن را آموزش می دهیم. بعد از تابع test استفاده کرده و مقدار دقت و خطای مدل روی داده آزمون را می یابیم. بعد با استفاده از plot نمودارها را رسم می کنیم. نتایج در تصاویر زیر دیده می شود:

مدل اول با alpha = -0.5

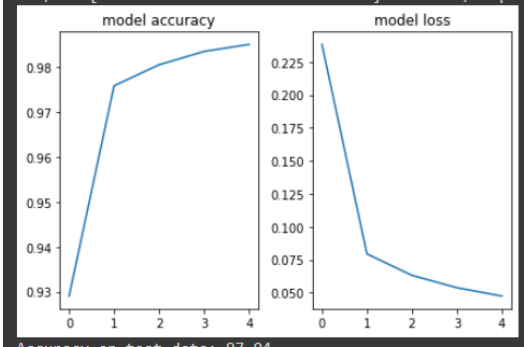
```
model with alpha = -1
Epoch 1/5
938/938 [==============================] - 7s 7ms/step - loss: 0.2291 - accuracy: 0.9316
Epoch 2/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0755 - accuracy: 0.9769
Epoch 3/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0609 - accuracy: 0.9814
Epoch 4/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0543 - accuracy: 0.9831
Epoch 5/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0480 - accuracy: 0.9847
```




```
Accuracy on test data: 97.96
Loss on test data: 0.07
**************************************************
```

مدل دوم با alpha = -1

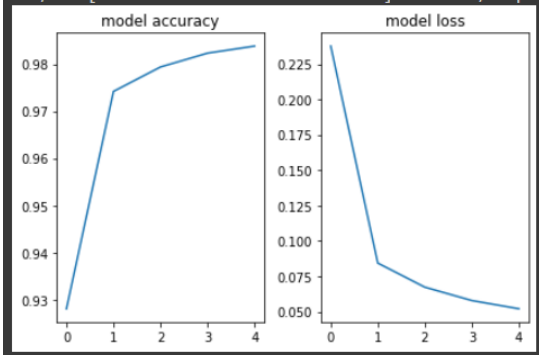
```
model with alpha = -0.5
Epoch 1/5
938/938 [==============================] - 7s 7ms/step - loss: 0.2384 - accuracy: 0.9291
Epoch 2/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0795 - accuracy: 0.9758
Epoch 3/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0632 - accuracy: 0.9805
Epoch 4/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0538 - accuracy: 0.9835
Epoch 5/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0475 - accuracy: 0.9850
```




```
Accuracy on test data: 97.94
Loss on test data: 0.06
**************************************************
```

مدل سوم با alpha = 0

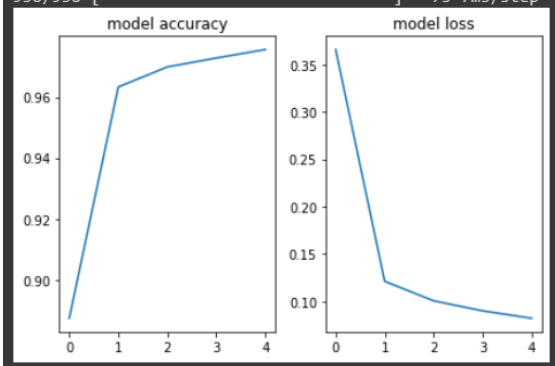
```
model with alpha = 0
Epoch 1/5
938/938 [==============================] - 7s 7ms/step - loss: 0.2377 - accuracy: 0.9281
Epoch 2/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0843 - accuracy: 0.9742
Epoch 3/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0673 - accuracy: 0.9794
Epoch 4/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0579 - accuracy: 0.9823
Epoch 5/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0520 - accuracy: 0.9838
```




```
Accuracy on test data: 98.22
Loss on test data: 0.06
**************************************************
```

مدل چهارم با alpha = 0.5

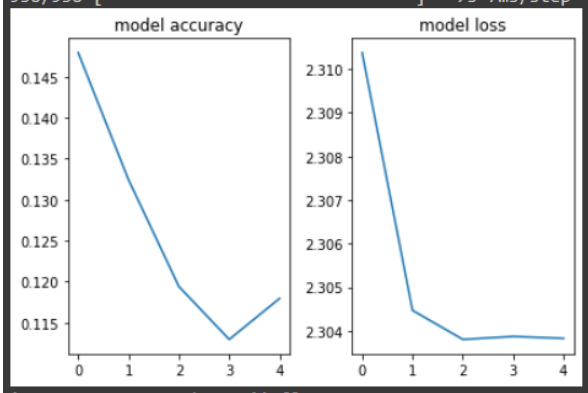
```
model with alpha = 0.5
Epoch 1/5
938/938 [==============================] - 8s 7ms/step - loss: 0.3658 - accuracy: 0.8875
Epoch 2/5
938/938 [==============================] - 7s 7ms/step - loss: 0.1212 - accuracy: 0.9633
Epoch 3/5
938/938 [==============================] - 7s 8ms/step - loss: 0.1005 - accuracy: 0.9699
Epoch 4/5
938/938 [==============================] - 7s 8ms/step - loss: 0.0899 - accuracy: 0.9728
Epoch 5/5
938/938 [==============================] - 7s 7ms/step - loss: 0.0821 - accuracy: 0.9756
```




```
Accuracy on test data: 97.29
Loss on test data: 0.08
**************************************************
```

مدل پنجم با alpha = 1

```
model with alpha = 1
Epoch 1/5
938/938 [==============================] - 7s 7ms/step - loss: 2.3104 - accuracy: 0.1479
Epoch 2/5
938/938 [==============================] - 7s 7ms/step - loss: 2.3045 - accuracy: 0.1324
Epoch 3/5
938/938 [==============================] - 7s 7ms/step - loss: 2.3038 - accuracy: 0.1194
Epoch 4/5
938/938 [==============================] - 7s 7ms/step - loss: 2.3039 - accuracy: 0.1129
Epoch 5/5
938/938 [==============================] - 7s 7ms/step - loss: 2.3038 - accuracy: 0.1179
```




```
Accuracy on test data: 11.69
Loss on test data: 2.30
****************************************************
```

برای پیدا کردن بهترین مقدار آلفا در میان مقادیری داده شده، دقت مدل ها را روی داده آزمون مقایسه می کنیم. مدل سوم با alpha = 0 بهترین دقت آزمون را دارد.

در مدل پنجم که alpha = 1 است، مشاهده می شود که دقت مدل از epoch اول بسیار کم و تقریبا نزدیک به حالت پیش بینی تصادفی است. و در طی epochهای بعدی در حال کاهش است. بررسی دلیل:

می دانیم فرمول تابع فعال سازی Leaky ReLU به صورت زیر است:

```
f(x) = alpha * x if x < 0
f(x) = x if x >= 0
```

اگر در این فرمول مقدار آلفا را برابر 1 قرار دهیم به شکل زیر در می آید:

f(x) = x if x < 0

f(x) = x if x >= 0

➔ f(x) = x

که این فرمول عملا یک تابع فعال سازی خطی است و هیچ تغییری در خروجی نورون ها حاصل نمی شود. یعنی نورون های خارج شده از لایه کانولوشن به همان صورت به لایه بعدی منتقل می شوند. به همین دلیل شبکه عملکرد بسیار ضعیفی داشته است.

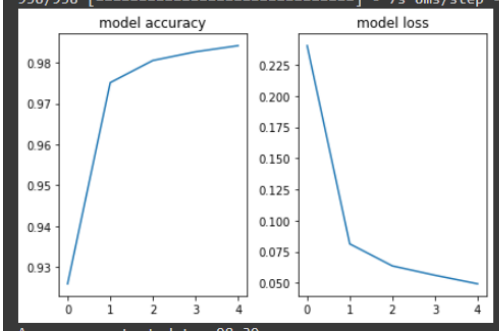
سلول دهم: در این سلول تابعی تعریف می کنیم که به عنوان خروجی یک مدل برمی گرداند. مشابه مدل های قبل است با این تفاوت که برای لایه های کانولوشنی آن از تابع فعال سازی PReLU استفاده می کنیم. برای اینکه هر فیلتر فقط یک مجموعه از پارامترها را داشته باشد مقدار shared_axes را در آن به [1,2] ست می کنیم.

```
def sequential_network_Prelu():
  prelu = PReLU(shared_axes=[1,2])
  model = Sequential()
  model.add(Conv2D(8, 7, activation=prelu))
  model.add(Conv2D(8, 5, activation=prelu))
  model.add(Flatten())
  model.add(Dense(10, activation='softmax'))

  return model
```

سلول یازدهم: مشابه مراحل، ابتدا یک مدل با تابع فعال سازی PReLU می سازیم. سپس آن را آموزش می دهیم. بعد مقدار دقت و خطا را روی داده آزمون محاسبه می کنیم. و در نهایت نمودارها را رسم می کنیم.

```
print("PReLU model")
model_prelu = sequential_network_Prelu()
hist = train(model_prelu)
test_loss, test_acc = test(model_prelu)
plot(hist)
print('Accuracy on test data: %.2f' % (test_acc*100))
print('Loss on test data: %.2f' % (test_loss))
```

```
PReLU model
Epoch 1/5
938/938 [==============================] - 8s 8ms/step - loss: 0.2403 - accuracy: 0.9259
Epoch 2/5
938/938 [==============================] - 7s 8ms/step - loss: 0.0813 - accuracy: 0.9751
Epoch 3/5
938/938 [==============================] - 7s 8ms/step - loss: 0.0637 - accuracy: 0.9806
Epoch 4/5
938/938 [==============================] - 7s 8ms/step - loss: 0.0562 - accuracy: 0.9827
Epoch 5/5
938/938 [==============================] - 7s 8ms/step - loss: 0.0493 - accuracy: 0.9842
```




```
Accuracy on test data: 98.39
Loss on test data: 0.05
```

سلول دوازدهم: می دانیم تابع PReLU تقریبا مشابه Leaky ReLU است. با این تفاوت که در آن پارامتر آلفا از مقادیر trainable شبکه است و شبکه خودش آن را آموزش می بیند. اما در Leaky ReLU مقدار آلفا به صورت دستی tune می شود.

پارامترهای آلفایی که شبکه پس از آموزش به آنها رسیده را چاپ می کنیم.

```
1 layers = model_prelu.layers
2
3 W1 = layers[0].get_weights()
4 print(W1[2])
5
6 W2 = layers[1].get_weights()
7 print(W2[2])
```

```
[[[-0.09735187 -0.2429723  -0.36114842 -0.15082116 -0.23535672
   -0.02661474 -0.2351685   0.391844  ]]]
[[[-0.09735187 -0.2429723  -0.36114842 -0.15082116 -0.23535672
   -0.02661474 -0.2351685   0.391844  ]]]
```

در حالت قبل، بهترین آلفا 0 بود. در اینجا نیز اکثر مقادیر آلفا (به جز یکی) که برای هر فیلتر یاد گرفته، کمترین فاصله را با 0 دارند (نسبت به فاصله با آلفاهای دیگر). پس بهترین مقدار آلفا نتیجه این آزمایش را تایید می کند. اگر بخواهیم مقدار آلفا در تابع فعال سازی را Leaky ReLU را به صورت دستی tune کنیم، بهتر است از مقدار منفی کوچک تا 0 را امتحان کنیم.