به نام خدا

تمرین سوم یادگیری عمیق

غزل زمانی‌نژاد

۹۷۵۲۲۱۶۶

.1

- **الف) در این نمودار عملکرد بهینه‌سازهای متفاوت بر روی دیتاست MNIST نشان داده شده‌است. طبق نمودار، عملکرد Adam Optimizer از عملکرد سایر بهینه‌سازها بهتر بوده‌است.**

در AdaGrad نرخ یادگیری بر پارامتر (تطبیقی) داریم که می‌تواند عملکرد را در مسائل با گرادیان‌های پراکنده (مثلا زبان‌های طبیعی و بینایی ماشین) بهبود ببخشد. نرخ یادگیری در طول زمان به سمت صفر میل می‌کند.

RMSProp تقریبا مشابه AdaGrad است با این تفاوت که میانگین اندازه مربع گرادیان‌ها را مورد توجه قرار می‌دهد و نه مجموع‌شان. می‌تواند عملکرد خوبی بر روی داده‌های نویزی داشته‌باشد. همانطور که در شکل مشاهده می‌شود عملکرد تابع سبز (RMSProp) مقداری بهتر از عملکرد تابع آبی (AdaGrad) بوده‌است و توانسته مقدار loss را بیشتر کاهش دهد.

بهینه‌ساز Adam ایده‌های اصلی AdaGrad و RMSProp را ترکیب می‌کند تا بتواند گرادیان‌های پراکنده بر روی داده‌های نویزی را هندل کند. در SGD نرخ‌یادگیری در تمام مدت آموزش ثابت است اما در Adam اینگونه نیست. در Adam هم از first moment (یعنی اگر dxها هم علامت هستند پس یکدیگر را تقویت کنند) و هم از second moment استفاده می‌کنیم (یعنی از exponential weighted average برای مربع گرادیان‌ها). همچنین برای مقابله با بایاس، از bias correction نیز استفاده می‌کنیم. به کمک هایپرپارامترهای beta1 و beta2 میتوانیم decay rate گرادیان‌ها و مربع گرادیان‌ها را تنظیم کنیم. در چند مقاله (که در سایت ذکر شده) این بهینه‌ساز به عنوان بهترین بهینه‌ساز در مسائل یادگیری عمیق ذکر شده‌است. مطابق نمودار تابع صورتی توانسته میزان loss را به کمترین مقدار ممکن برساند.

SGDNesterov هم در این نمودار مشاهده می‌شود. تقریبا مشابه momentum است اما میزان overshoot در آن از momentum کمتر است و با احتمال بیشتری از momentum در local minima گیر می‌کند و همگرایی آن سریعتر است. همچنین هزینه محاسباتی آن زیاد است. تابع قرمز هم توانسته عملکرد خوبی در بین سایر بهینه‌سازها داشته‌باشد.

AdaDelta هم تقریبا مشابه AdaGrad است. با این تفاوت که در AdaGrad میانگین مربع گرادیان‌ها را از لحظه اول تا همان لحظه در نظر می‌گیرد اما AdaDelta میانگین xتای گذشته را در نظر می‌گیرد و نه میانگین از ابتدا.

- ب) برای انتخاب بهینه‌ساز مناسب برای مسئله باید عوامل متعددی را در نظر بگیریم. بهینه‌سازها به دو دسته کلی تقسیم می‌شوند: بهینه‌سازهای Gradient Descent و بهینه‌سازهای adaptive. در دسته اول باید نرخ یادگیری را به صورت دستی tune کنیم اما در دسته دوم نرخ یادگیری به صورت خودکار tune می‌شود.

بهینه‌سازهای Gradient Descent:

Batch Gradient Descent: در یک گام بر روی تمامی داده‌ها آموزش را انجام می‌دهیم. می‌تواند بسیار آهسته عمل کند و برای دیتاست‌های بزرگ اصلا مناسب نیست.

Stochastic Gradient Descent: در یک گام تنها بر روی یک نمونه آموزش را انجام می‌دهیم. آپدیت وزن‌ها مرتبا انجام می‌شود و دارای واریانس بالایی است. همچنین objective function نوسان‌های زیادی می‌کند.

Minibatch Gradient Descent: در یک گام بر روی یک batch از داده‌ها آموزش را انجام می‌دهیم. ترکیبی از مزیت‌های Batch Gradient Descent و Stochastic Gradient Descent را دارد.

Minibatch Gradient Descent از میان ۳ دسته بالا بهترین بهینه‌ساز است. در صورتی که اندازه دیتاست کوچک باشد می‌توانیم از Batch Gradient Descent هم استفاده کنیم. برای همه آنها باید نرخ یادگیری را به صورت دستی tune کنیم. برای داده‌های پراکنده مناسب نیستند. با احتمال زیادی در local minima گیر می‌کنیم.

بهینه‌سازهای Adaptive:

Adagrad: به عنوان بهینه‌ساز تطبیقی به کار می‌رود. یعنی برای آپدیت کردن فیچرهای تکرارشونده از stepsize کوچک استفاده می‌کند و برای فیچرهای نادر آنها را بیشتر آپدیت می‌کند. ایراد این بهینه‌ساز آن است که برای تنظیم نرخ یادگیری از تمامی اطلاعات گذشته استفاده می‌کند و بعد از مدتی نرخ یادگیری بسیار کوچک می‌شود (و باعث می‌شود شبکه عملا چیز جدیدی یاد نگیرد).

Adadelta: مشابه روش قبل است اما از تمامی اطلاعات گذشته استفاده نمی‌کند و تنها به تعداد معلومی از گرادیان‌های گذشته استفاده می‌کند. در این روش مشکل vanishing نرخ یادگیری برطرف می‌شود.

RMSProp: مشابه Adagrad است اما به جای استفاده از مجموع مربع گرادیان‌های گذشته، از exponential weighted average استفاده می‌کند.

Adam: علاوه بر داشتن مزیت‌های Adadelta و RMSProp، مزیت momentum یعنی استفاده از گرادیان‌های گذشته (به نوعی مفهوم سرعت و شتاب در فیزیک) را نیز دارد. همچنین در مقایسه با بعضی روش‌ها میزان حافظه کمتری برای انجام محاسبات نیاز دارد.

در میان ۴ روش بالا، Adam معمولا بهترین انتخاب است. و برای دیتاست‌های پراکنده نیز عملکرد خوبی دارد. همچنین نیازی نیست نرخ یادگیری به صورت دستی تنظیم شود.

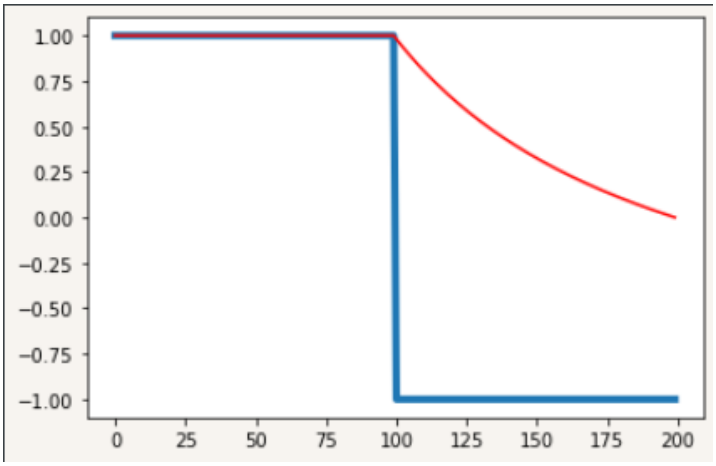
نتیجه نهایی: مطابق بسیاری از مقالات Adam بهترین بهینه‌ساز برای مسائل یادگیری عمیق است. پیشنهاد می‌شود در صورتی که در ابتدای کار خودمان نظری نداریم با Adam شروع کنیم و بعد سایر بهینه‌سازها را امتحان کنیم. اما در انتخاب بهینه‌ساز باید به عواملی از جمله اندازه دیتاست، مقدار پراکندگی داده‌ها، نوع داده‌ها و فیچرهای آنها، امکان تنظیم دستی نرخ یادگیری و ... توجه کنیم.

.2

الف) در قسمت اول میانگین نقاط را محاسبه و رسم می‌کنیم. ابتدا یک بردار دارای یک سطر و ۲۰۰ ستون میسازیم که ۱۰۰ المان ابتدایی ۱ و ۱۰۰ المان انتهایی ۱- هستند. با استفاده از np.cumsum مجموع المان‌ها تا لحظه t را محاسبه می‌کنیم. سپس بر تعداد المان‌ها تا لحظه t تقسیم می‌کنیم.

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

d1 = np.full((1, 100), 1)
d2 = np.full((1, 100), -1)
d = np.concatenate((d1, d2), axis=1)
print("vector d", d)

sum = np.cumsum(d)
avg = [sum[i]/(i+1) for i in range(sum.size)]
print("average of vector d", avg)

```

بردار d و همچنین میانگین آن را با کتابخانه matplotlib رسم می‌کنیم.

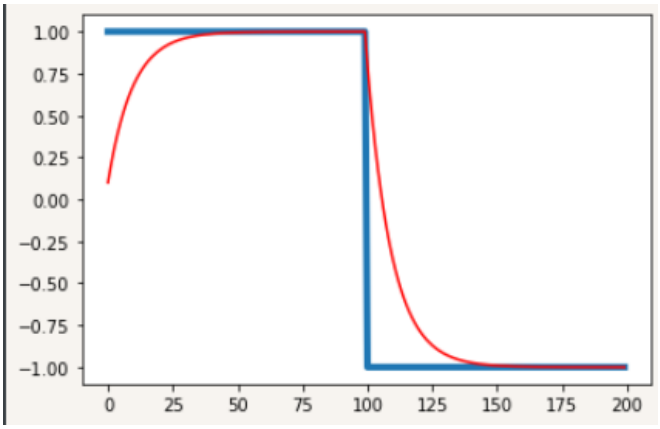


نقطه روی نمودار قرمز در لحظه t، میانگین نقاط نمودار آبی تا لحظه t است.

ب) در این قسمت، exponential moving average بردار d را در یک حلقه محاسبه می‌کنیم.

```
beta = 0.9
exp_avg = np.zeros((1, 200))
# let m1[0] = 0 + (1-beta) * d[0]
exp_avg[0][0] = (1-beta) * d[0][0]
for i in range(d.size-1):
  exp_avg[0][i+1] = beta * exp_avg[0][i] + (1-beta) * d[0][i+1]

```

سپس نتیجه را رسم می‌کنیم.



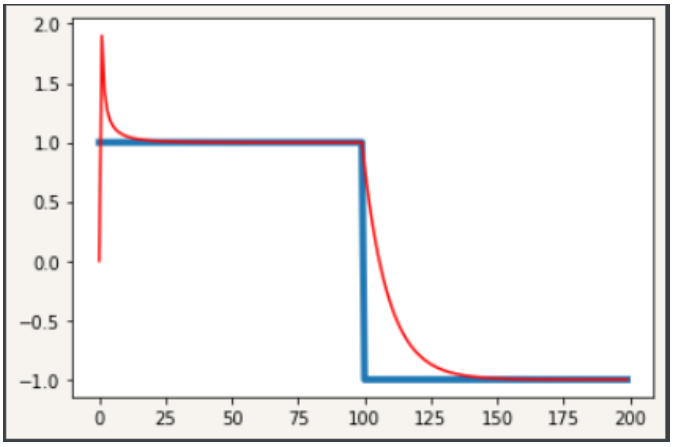
در این نوع میانگین، مطابق نمودار مشاهده می‌شود که مقادیر اخیر بیشتر مورد توجه قرار می‌گیرند (و مقادیر ابتدایی تاثیر چندانی در میانگین در لحظه ۲۰۰ ندارد). این نمودار شباهت بیشتری به بردار اصلی d دارد چون در محاسباتش از مقادیر اخیر بیشتر استفاده می‌کند.

ج) در این قسمت به محاسبه exponential moving average با استفاده از bias correction می‌پردازیم.

از بردار استفاده شده در قسمت قبل استفاده می‌کنیم و فقط در زمان t، مقدار را به 1 – beta^t تقسیم می‌کنیم.

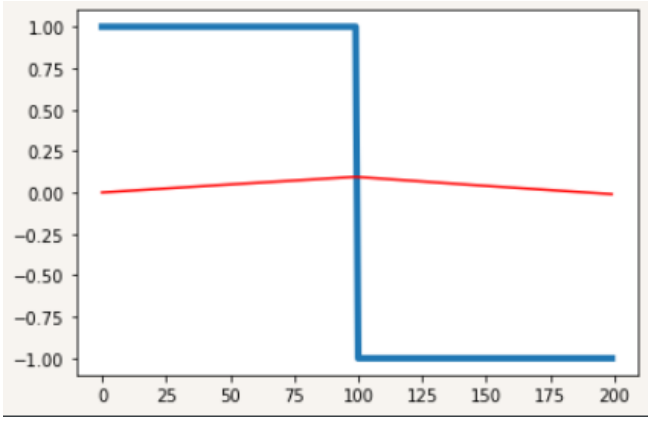
```
time = np.arange(0, d.size)
bias_correction = np.zeros((1, 200))
for i in range(1, d.size):
  bias_correction[0][i] = exp_avg[0][i] / (1 - beta**time[i])

```

سپس نتیجه را رسم می‌کنیم.

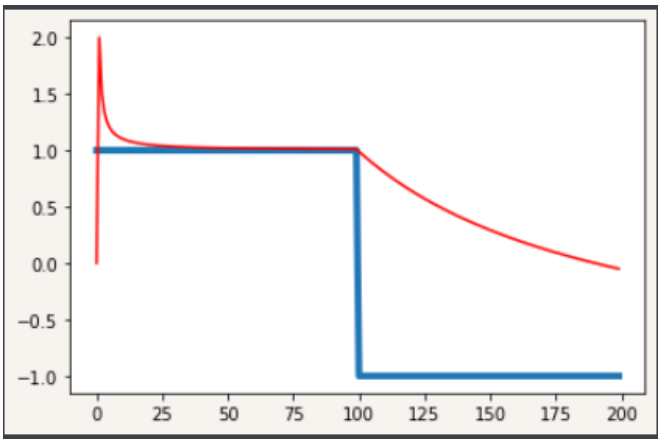


وقتی نمودار قسمت قبل را با بردار d مقایسه می‌کنیم، متوجه می‌شویم که میانگین نمایی در نقاط ابتدایی شباهت کمتری با بردار اصلی دارد. در این بخش از bias correction برای برطرف کردن این مشکل استفاده می‌کنیم. اعمال bias correction بیشترین تاثیر را در نقاط ابتدایی دارد و به مرور زمان، (0.9 ^t) به سمت صفر و مخرج به سمت ۱ میل می‌کند و تاثیر تقسیم از بین می‌رود. با اعمال bias correction میانگین نمایی بیشترین شباهت را به بردار d چه در نقاط ابتدایی و چه در نقاط انتهایی دارد.

ه) قسمت ج را با بتا ۰,۹۹۹ (مقدار بسیار نزدیک به ۱) تکرار می‌کنیم. نتیجه در شکل زیر مشاهده می‌شود.



قسمت د را با بتا ۰,۹۹۹ تکرار می‌کنیم. بعد از bias correction نمودار در نقاط ابتدایی نیز تخمین خوبی از میانگین نقاط است.



چون مقدار بتا بسیار به ۱ نزدیک است میانگین نمایی به مقادیر قبلی وابستگی بسیار زیادی دارد و تنها ۰,۰۰۱ از مقدار جدید را در محاسبات استفاده می‌کند.

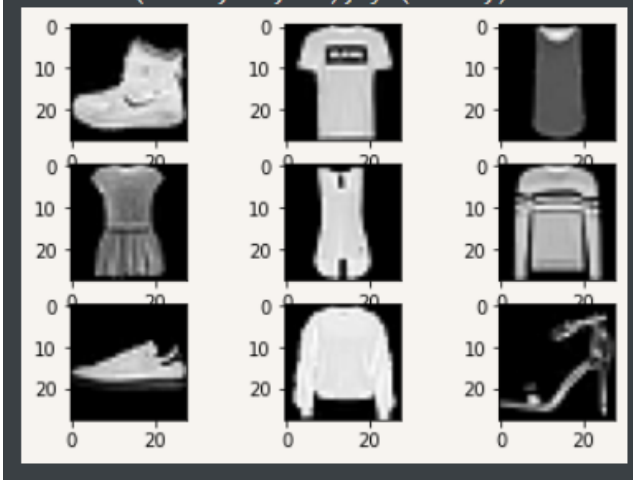
ثابت می‌شود که مقدار میانگین نمایی، تقریبا میانگینی از 1 / (1-beta) داده اخیر است. یعنی با بتا ۰,۹۹۹ میانگینی از ۱۰۰۰ داده اخیر است. در بردار d این میانگین نزدیک به صفر است زیرا نصف اعداد ۱+ و نصف دیگر اعداد ۱- هستند. اما با بتا ۰,۹ میانگینی از ۱۰ داده اخیر است.

3. **در این سوال ابتدا هسته رندوم‌های موجود در کد را تعیین می‌کنیم (تا بتوانیم نتایج قسمت‌های مختلف را با یکدیگر مقایسه کنیم). سپس دیتاست Fashion MNIST را به کمک کتابخانه keras لود می‌کنیم. بعد داده‌ها را نرمالایز می‌کنیم تا میزان پراکندگی داده‌ها زیاد نباشد. در یک حلقه چند نمونه از داده را چاپ می‌کنیم تا بهتر با دیتاست آشنا شویم. و در نهایت داده‌ها را reshape می‌کنیم به گونه‌ای که یک کانال داشته باشد.**

```
# set random seeds
np.random.seed(5)
random.seed(10)
tf.random.set_seed(2)

# load dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

print('Train: X=%s, y=%s' % (x_train.shape, y_train.shape))
print('Test: X=%s, y=%s' % (x_test.shape, y_test.shape))

# normalize dataset
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255


# plot first few images
for i in range(9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(x_train[i], cmap=pyplot.get_cmap('gray'))
pyplot.show()

# reshape dataset to have a single channel
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))
```

شکل داده‌های ورودی و آزمون و همچنین نمونه داده‌ها در زیر مشاهده می‌شود.

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```



برای سادگی کار تعریف شبکه عصبی همراه با هایپرپارامترها و همچنین آموزش و تست آن را با استفاده از یک کلاس انجام می‌دهیم. Constructor **این کلاس به عنوان ورودی نرخ یادگیری، نوع بهینه‌ساز، تعداد** epoch**ها، اندازه** minibatch**، تعداد نورون‌های لایه مخفی و درصد داده‌های اعتبارسنجی را می‌گیرد.**

```python
class Fashion_MNIST:
  def __init__(self, learning_rate, optimizer, epoch, batch, hidden_size, val_size):
    self.learning_rate = learning_rate
    self.optimizer = optimizer
    self.epoch = epoch
    self.batch = batch
    self.hidden_size = hidden_size
    self.val_size = val_size

```

به کمک تابع define_model، ابتدا یک مدل sequential (موجود در کتابخانه keras) می‌سازیم. اولین لایه این شبکه Flatten است. به کمک آن داده ۲بعدی به داده یک بعدی تبدیل می‌شود (استفاده از minibatch مانعی در استفاده از Flatten ندارد و با استفاده کردن از لایه Flatten هر نمونه به تنهایی به داده یک بعدی تبدیل می‌شود. لایه بعدی همان لایه مخفی است که دارای تعداد مشخصی نورون است. برای این لایه از Dense استفاده می‌کنیم. و تابع فعالسازی آن را Relu قرار می‌دهیم (این تابع در واقع ماکسیمم صفر و مقدار ورودی را برمی‌گرداند). در لایه آخر ۱۰ نورون موجود است چون این مسئله ۱۰ کلاسه است. برای این لایه از Dense با ۱۰ نورون استفاده می‌کنیم و تابع فعالسازی آن را softmax قرار می‌دهیم. خروجی این تابع فعالسازی نشان می‌دهد که هر ورودی با چه احتمالی می‌تواند مربوط به هر یک از ۱۰ کلاس باشد.

سپس نوع بهینه‌ساز را مشخص می‌کنیم. نرخ یادگیری را نیز به بهینه‌ساز می‌دهیم. به‌طور کلی در keras برای استفاده از بهینه‌سازها از tf.keras.optimizers استفاده می‌شود.

در نهایت مدل را با استفاده از بهینه‌ساز معلوم، تابع ضرر و متریک دقت کامپایل می‌کنیم. و مدل را به عنوان خروجی تابع بازمی‌گردانیم.

```python
  def define_model(self):
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten()) # data format is channels_last
    model.add(tf.keras.layers.Dense(self.hidden_size, activation='relu'))
    model.add(tf.keras.layers.Dense(10, activation='softmax'))
    # compile model
    if (self.optimizer == 'sgd'):
      opt = tf.keras.optimizers.SGD(learning_rate=self.learning_rate)
    elif self.optimizer == 'Adam':
      opt = tf.keras.optimizers.Adam(learning_rate=self.learning_rate)
    elif self.optimizer == 'RMSprop':
      opt = tf.keras.optimizers.RMSprop(learning_rate=self.learning_rate)
    else: opt = tf.keras.optimizers.Adagrad(learning_rate=self.learning_rate)

    model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

برای آموزش مدل از تابع train_model استفاده می‌کنیم. این تابع به عنوان ورودی مدل را می‌گیرد. با استفاده از دستور model.fit به آموزش می‌پردازیم. به عنوان ورودی x و y آموزش، اندازه minibatch، اندازه epoch و درصد داده‌های اعتبارسنجی را به fit می‌دهیم. خروجی را بازمی‌گردانیم.

```
27   def train_model(self, model):
28     history = model.fit(x_train,
29                         y_train,
30                         batch_size=self.batch,
31                         epochs=self.epoch,
32                         validation_split=self.val_size)
33     return history
```

برای محاسبه میزان دقت بر روی داده‌های آزمون از تابع test_model استفاده می‌کنیم. این تابع به عنوان ورودی مدل را می‌گیرد (به آن مدل بعد از training را پاس می‌دهیم). با استفاده از model.evaluate، میزان دقت روی داده‌های آزمون را بدست می‌آوریم.

```
35   def test_model(self, model):
36     score = model.evaluate(x_test, y_test)
37
```

- الف) این دیتاست ۱۰ کلاسه مربوط به لباس است. شامل ۶۰هزار داده آموزشی و ۱۰هزار داده آزمون است. هر یک از داده‌ها یک تصویر سیاه و سفید با ابعاد ۲۸ * ۲۸ می‌باشد (یعنی هر تصویر تنها ۱ کانال دارد و نه ۳ کانال). کلاس‌های این دیتاست عبارتند از:

0 T-shirt/top
1 Trouser
2 Pullover
3 Dress
4 Coat
5 Sandal
6 Shirt
7 Sneaker
8 Bag
9 Ankle boot

- ب) درصد داده‌های اعتبارسنجی را ۱۰ نظر می‌گیریم. چون در این مسئله ۶۰هزار داده آموزشی داریم و جدا کردن ۱۰درصد از آن (یعنی ۶هزار نمونه) به فرآیند آموزش آسیبی نمی‌رساند. از طرفی با ۶هزار داده اعتبارسنجی می‌توانیم شبکه مورد نظرمان را fine-tune کنیم.

- ج) در این قسمت می‌خواهیم شبکه را با مقادیر مختلفی نورون در لایه مخفی و ثابت نگه داشتن سایر هایپرپارامترها امتحان کنیم.

شماره ۱: تمامی هایپرپارامترها را مطابق صورت سوال در نظر می‌گیریم. تعداد نورون‌های لایه مخفی برابر با ۱۶ است.

```
class1 = Fashion_MNIST(0.001, 'sgd', 50, 64, 16, 0.1)
model1 = class1.define_model()
print("training the model")
class1.train_model(model1)
print("testing the model")
class1.test_model(model1)
```

میزان دقت و خطای شبکه در آموزش، اعتبارسنجی و آزمون مطابق زیر است:

```
Epoch 50/50
844/844 [==============================] - 1s 2ms/step - loss: 0.5025 - accuracy: 0.8295 - val_loss: 0.5020 - val_accuracy: 0.8288
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5297 - accuracy: 0.8163
```

شماره۲: تمامی هایپرپارامترها را مطابق صورت سوال در نظر می‌گیریم. تعداد نورون‌های لایه مخفی برابر با ۳۲ است.

```
class2 = Fashion_MNIST(0.001, 'sgd', 50, 64, 32, 0.1)
model2 = class2.define_model()
print("training the model")
class2.train_model(model2)
print("testing the model")
class2.test_model(model2)
```

میزان دقت و خطای شبکه در آموزش، اعتبارسنجی و آزمون مطابق زیر است:

```
Epoch 50/50
844/844 [==============================] - 1s 2ms/step - loss: 0.4798 - accuracy: 0.8358 - val_loss: 0.4769 - val_accuracy: 0.8298
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5072 - accuracy: 0.8231
```

شماره ۳: تمامی هایپرپارامترها را مطابق صورت سوال در نظر می‌گیریم. تعداد نورون‌های لایه مخفی برابر با ۶۴ است.

```
class3 = Fashion_MNIST(0.001, 'sgd', 50, 64, 64, 0.1)
model3 = class3.define_model()
print("training the model")
class3.train_model(model3)
print("testing the model")
class3.test_model(model3)
```

میزان دقت و خطای شبکه در آموزش، اعتبارسنجی و آزمون مطابق زیر است:

```
Epoch 50/50
844/844 [==============================] - 2s 2ms/step - loss: 0.4657 - accuracy: 0.8413 - val_loss: 0.4630 - val_accuracy: 0.8388
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4949 - accuracy: 0.8282
```

شماره ۴: تمامی هایپرپارامترها را مطابق صورت سوال در نظر می‌گیریم. تعداد نورون‌های لایه مخفی برابر با ۱۲۸ است.

```
class4 = Fashion_MNIST(0.001, 'sgd', 50, 64, 128, 0.1)
model4 = class4.define_model()
print("training the model")
class4.train_model(model4)
print("testing the model")
class4.test_model(model4)
```

میزان دقت و خطای شبکه در آموزش، اعتبارسنجی و آزمون مطابق زیر است:

```
Epoch 50/50
844/844 [==============================] - 2s 2ms/step - loss: 0.4613 - accuracy: 0.8435 - val_loss: 0.4623 - val_accuracy: 0.8357
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4913 - accuracy: 0.8322
```

مطابق شبکه‌های بالا مشاهده می‌شود که مقدار دقت آموزش و اعتبارسنجی بسیار بهم نزدیک هستند یعنی مدل‌ها دچار overfit نشده و توانسته بر روی داده‌هایی که از قبل ندیده عملکرد خوبی داشته باشند. اما این مدل‌ها ممکن است دچار underfit شده باشند زیرا در بسیاری از مسائل با دیتاست بزرگ، دقت آموزش بالای ۹۰ درصد است اما در این مسئله دقت زیر ۸۵درصد است. استفاده از SGD نیز از عواملی است که باعث کمتر بودن مقدار دقت است زیرا SGD نمی‌تواند به خوبی الگوریتم‌های تطبیقی تابع ضرر را کمینه کند.

در آزمایش‌های بالا مشاهده می‌شود که مدل دارای ۱۲۸ نورون در لایه مخفی دارای بهترین عملکرد (کمترین میزان خطای اعتبارسنجی در میان ۳ مدل دیگر و بیشترین میزان دقت در داده‌های اعتبارسنجی و آزمون) است.

د) در این قسمت آزمایش‌ها را با ۳ درصد داده اعتبارسنجی مختلف تکرار می‌کنیم و نتیجه را چاپ می‌کنیم.

شماره یک: val_size = 0.15

یک – یک:

```
Epoch 50/50
797/797 [==============================] - 1s 2ms/step - loss: 0.5073 - accuracy: 0.8292 - val_loss: 0.5157 - val_accuracy: 0.8183
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5367 - accuracy: 0.8166
```

یک – دو:

```
Epoch 50/50
797/797 [==============================] - 1s 2ms/step - loss: 0.4814 - accuracy: 0.8352 - val_loss: 0.4910 - val_accuracy: 0.8279
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5123 - accuracy: 0.8212
```

یک – سه:

```
Epoch 50/50
797/797 [==============================] - 2s 2ms/step - loss: 0.4714 - accuracy: 0.8411 - val_loss: 0.4815 - val_accuracy: 0.8316
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5039 - accuracy: 0.8270
```

یک – چهار:

```
Epoch 50/50
797/797 [==============================] - 2s 2ms/step - loss: 0.4601 - accuracy: 0.8443 - val_loss: 0.4709 - val_accuracy: 0.8350
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4913 - accuracy: 0.8318
```

در قمست یک بهترین عملکرد مربوط به شبکه با ۱۲۸ نورون است.

شماره دو: val_size = 0.2

دو – یک:

```
Epoch 50/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5054 - accuracy: 0.8291 - val_loss: 0.5100 - val_accuracy: 0.8257
testing the model
313/313 [==============================] - 0s 997us/step - loss: 0.5354 - accuracy: 0.8170
```

دو – دو:

```
Epoch 50/50
750/750 [==============================] - 1s 2ms/step - loss: 0.4851 - accuracy: 0.8371 - val_loss: 0.4926 - val_accuracy: 0.8296
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5156 - accuracy: 0.8221
```

دو – سه:

```
Epoch 50/50
750/750 [==============================] - 2s 2ms/step - loss: 0.4785 - accuracy: 0.8381 - val_loss: 0.4850 - val_accuracy: 0.8311
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5077 - accuracy: 0.8267
```

دو – چهار:

```
Epoch 50/50
750/750 [==============================] - 2s 2ms/step - loss: 0.4676 - accuracy: 0.8429 - val_loss: 0.4751 - val_accuracy: 0.8348
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4963 - accuracy: 0.8287
```

در قمست دو بهترین عملکرد مربوط به شبکه با ۱۲۸ نورون است.

شماره سه: val_size = 0.25

سه – یک:

```
Epoch 50/50
704/704 [==============================] - 1s 2ms/step - loss: 0.5168 - accuracy: 0.8277 - val_loss: 0.5215 - val_accuracy: 0.8213
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5469 - accuracy: 0.8143
```

سه – دو:

```
Epoch 50/50
704/704 [==============================] - 1s 2ms/step - loss: 0.4906 - accuracy: 0.8343 - val_loss: 0.4991 - val_accuracy: 0.8280
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5213 - accuracy: 0.8231
```

سه – سه:

```
Epoch 50/50
704/704 [==============================] - 2s 2ms/step - loss: 0.4870 - accuracy: 0.8358 - val_loss: 0.4946 - val_accuracy: 0.8290
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5161 - accuracy: 0.8231
```

سه – چهار:

```
Epoch 50/50
704/704 [==============================] - 2s 3ms/step - loss: 0.4742 - accuracy: 0.8402 - val_loss: 0.4809 - val_accuracy: 0.8335
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5040 - accuracy: 0.8246
```

در قمست سه بهترین عملکرد مربوط به شبکه با ۱۲۸ نورون است.

پس از انجام ۱۲ آزمایش مشاهده می‌شود که بهترین عملکرد مربوط به شبکه دارای ۱۲۸ نورون و درصد داده‌های اعتبارسنجی ۰,۱ است.

بهترین تعداد نورون‌های لایه مخفی نسبت به حالت قبل تغییری نکرد. زیرا این مسئله در لایه ورودی دارای ۲۸ * ۲۸ نورون است و با زیادتر کردن تعداد نورون‌های لایه مخفی می‌توانیم عملکرد بهتری از شبکه داشته‌باشیم.

دقت فاز آزمون نیز تغییر چشمگیری نداشت (تنها در حد چند هزارم تغییر کرد) زیرا میزان تغییر در درصد داده‌های اعتبارسنجی زیاد نبود.

ه) در این قسمت، می‌خواهیم بهینه‌ساز مناسب برای حل این مسئله را بیابیم. برای این کار با بهترین شبکه‌ای که تاکنون پیدا کرده‌ایم (یعنی دارای ۱۲۸ نورون در لایه مخفی و ۰,۱ درصد داده‌های اعتبارسنجی) ادامه می‌دهیم.

شماره یک: اولین بهینه‌ساز مورد آزمایش Adam **است.**

```
Epoch 50/50
844/844 [==============================] - 2s 3ms/step - loss: 0.0952 - accuracy: 0.9647 - val_loss: 0.4549 - val_accuracy: 0.8862
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4877 - accuracy: 0.8814
```

در اینجا میزان دقت داده آموزشی به طرز چشمگیری افزایش پیدا کرده است. همچنین میزان دقت داده اعتبارسنجی و آزمون نیز بیشتر شده‌است. اما بین دقت داده آموزشی و دقت داده اعتبارسنجی فاصله زیادی مشاهده می‌شود. این بدین معناست که احتمالا مدل دچار overfit شده است.

شماره دو: دومین بهینه‌ساز مورد آزمایش RMSProp است.

```
Epoch 50/50
844/844 [==============================] - 3s 3ms/step - loss: 0.1178 - accuracy: 0.9583 - val_loss: 0.6104 - val_accuracy: 0.8875
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5986 - accuracy: 0.8854
```

در اینجا نیز میزان دقت داده آموزشی به طرز چشمگیری افزایش پیدا کرده است. همچنین میزان دقت داده اعتبارسنجی و آزمون نیز بیشتر شده‌است. دقت این مدل از دقت مدل با بهینه‌ساز Adam مقداری کمتر است و در مدل Adam میزان خطای اعتبارسنجی بهتر کمینه شده‌است. در این قسمت نیز مدل دچار overfit شده است.

شماره سه: سومین بهینه‌ساز مورد آزمایش Adagrad است.

```
Epoch 50/50
844/844 [==============================] - 2s 2ms/step - loss: 0.4433 - accuracy: 0.8518 - val_loss: 0.4476 - val_accuracy: 0.8422
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4747 - accuracy: 0.8375
```

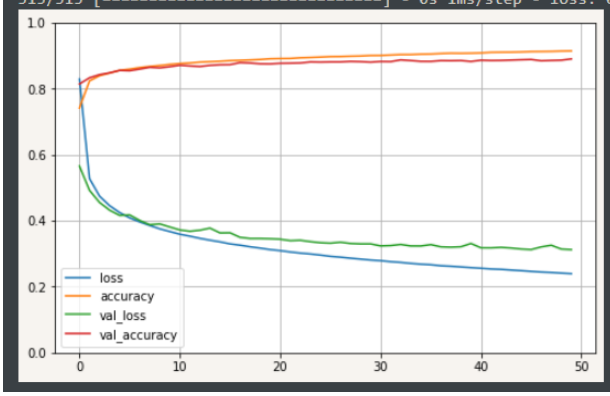
این مدل نسبت به مدل SGD اندکی بهتر است اما تغییر چندانی نکرده‌است. احتمالا مدل دچار underfit شده‌است چون دقت داده آموزشی مقدار بسیار زیادی ندارد. اما دقت اعتبارسنجی و آزمون اندکی بیشتر شده است چون Adagrad در محاسباتش از مربع گرادیان‌های گذشته استفاده می‌کند و در نتیجه می‌تواند با توجه به اطلاعات گذشته مقدار ضرر را بیشتر کمینه کند.

نتیجه انجام این آزمایش: همانطور که انتظار میرفت Adam از سایر الگوریتم‌های تطبیقی بهتر عمل کرد و توانست به میزان دقت بیشتر و خطای کمتری دست یابد. دلیل آن این است که این بهینه‌ساز ترکیبی از ایده‌های خوب سایر بهینه‌سازها است.

و) در این بخش نرخ‌های یادگیری مختلف را بر روی بهترین شبکه‌ای که تاکنون پیدا کرده‌ایم (دارای ۱۲۸ نورون در لایه مخفی و ۰٫۱ درصد داده‌های اعتبارسنجی و بهینه‌ساز Adam) آزمایش می‌کنیم.

شماره یک: learning rate = 0.0001

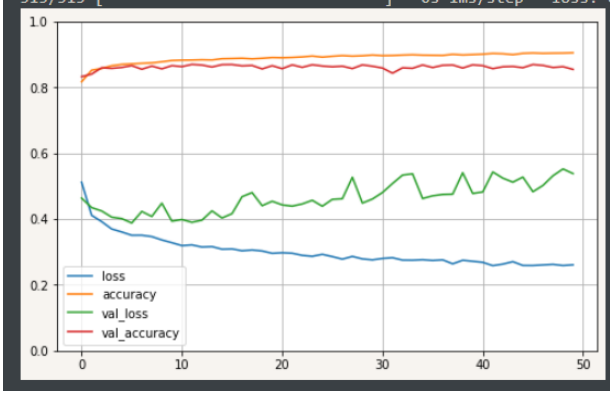
```
Epoch 50/50
844/844 [==============================] - 2s 3ms/step - loss: 0.2396 - accuracy: 0.9146 - val_loss: 0.3124 - val_accuracy: 0.8905
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.3402 - accuracy: 0.8826
```



با کم کردن نرخ یادگیری، سرعت یادگیری کم می‌شود زیرا در هر گام step کوچکتری برمی‌داریم در نتیجه پارامترها به میزان کمتری آپدیت می‌شوند.

شماره دو: learning rate = 0.01

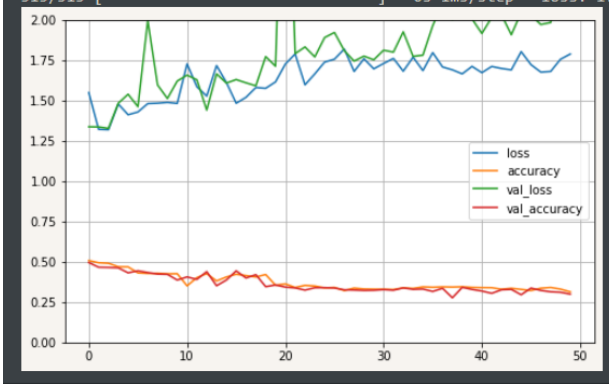
```
Epoch 50/50
844/844 [==============================] - 2s 3ms/step - loss: 0.2610 - accuracy: 0.9055 - val_loss: 0.5382 - val_accuracy: 0.8548
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.5746 - accuracy: 0.8407
```



با زیاد کردن نرخ یادگیری، سرعت یادگیری زیاد می‌شود زیرا در هر گام step بزرگتری برمی‌داریم در نتیجه پارامترها به میزان بیشتری آپدیت می‌شوند.

شماره سه: learning rate = 0. 1

```
Epoch 50/50
844/844 [==============================] - 2s 3ms/step - loss: 1.7891 - accuracy: 0.3133 - val_loss: 2.1526 - val_accuracy: 0.2988
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 1.9006 - accuracy: 0.3060
```



اگر نرخ یادگیری را بیش از حد زیاد کنیم، نتیجه عکس خواهد داشت چون نوسان پارامترها بسیار زیاد می‌شود و عملا شبکه نمی‌تواند مقادیر مناسبی برای برای کم کردن تابع ضرر یاد بگیرد. با رساندن نرخ یادگیری به ۰,۱ دقت شبکه نسبت به حالات قبل تقریبا نصف شده‌است.

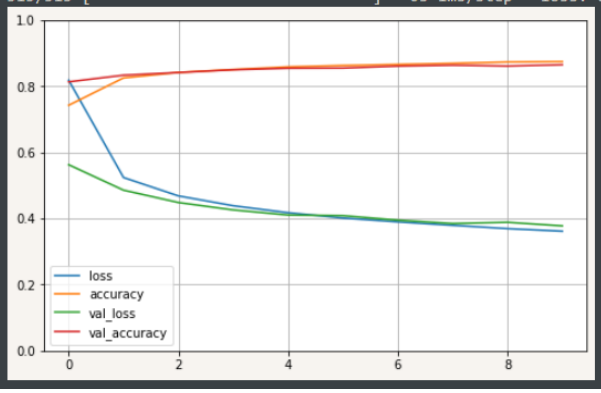
پس باید در انتخاب نرخ یادگیری دقت کافی به خرج دهیم تا مقدار آن برای شبکه نه خیلی زیاد و نه خیلی کم باشد.

در این بخش بهترین مدل (مدل دارای دقت بالا و ضرر کم) دارای نرخ یادگیری ۰,۰۰۰۱ است.

ز) طبق تعریف یک مدل زمانی همگرا می‌شود که در صورت ادامه دادن فرآیند آموزش، مدل بهبود نیابد.

این آزمایش را بر روی مدل دارای نرخ یادگیری ۰,۰۰۰۱ انجام می‌دهیم.

```
Epoch 10/10
844/844 [==============================] - 2s 2ms/step - loss: 0.3616 - accuracy: 0.8747 - val_loss: 0.3776 - val_accuracy: 0.8653
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4041 - accuracy: 0.8596
```

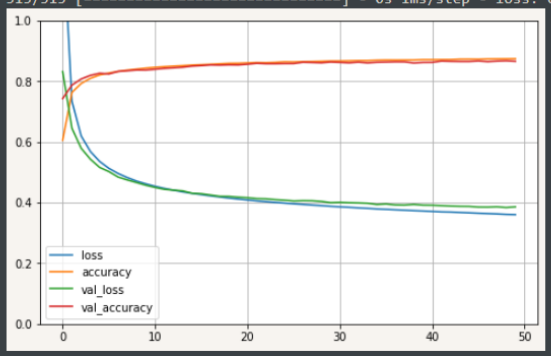


میزان یادگیری مدل در ۱۰ epoch کمتر از ۵۰ epoch است. به همین خاطر میزان دقت و عملکرد کلی مدل ضعیف‌تر از حالت قبل است. می‌دانیم که در صورت ادامه دادن فرآیند آموزش به دقت بالاتری دست می‌یابیم (چون یک بار این آزمایش را در ۵۰ epoch انجام داده‌ایم). پس این مدل همگرا نیست.

ح) برای مقایسه نمودارهای این سوال، آزمایش را بر روی بهترین مدل در ۵۰ epoch انجام می‌دهیم.

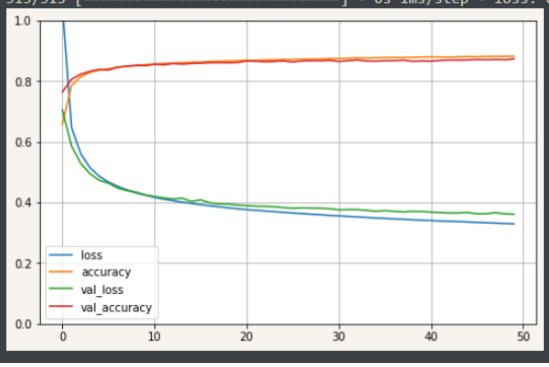
شماره یک: hidden_size = 16

```
Epoch 50/50
844/844 [==============================] - 1s 2ms/step - loss: 0.3603 - accuracy: 0.8743 - val_loss: 0.3860 - val_accuracy: 0.8658
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.4135 - accuracy: 0.8554
```



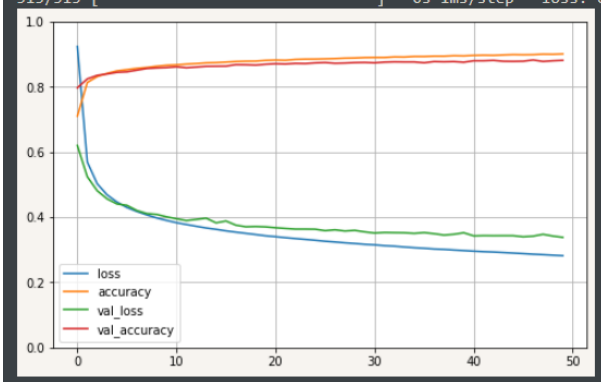
شماره دو: hidden_size = 32

```
Epoch 50/50
844/844 [==============================] - 1s 2ms/step - loss: 0.3296 - accuracy: 0.8822 - val_loss: 0.3613 - val_accuracy: 0.8737
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.3901 - accuracy: 0.8637
```



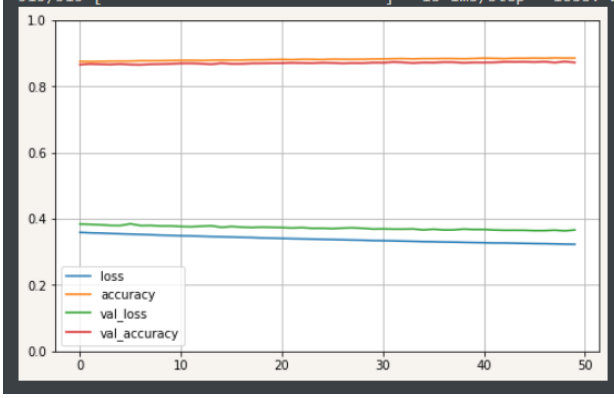
شماره سه: hidden_size = 64

```
Epoch 50/50
844/844 [==============================] - 2s 2ms/step - loss: 0.2819 - accuracy: 0.9012 - val_loss: 0.3382 - val_accuracy: 0.8815
testing the model
313/313 [==============================] - 0s 1ms/step - loss: 0.3573 - accuracy: 0.8733
```



شماره چهار: hidden_size = 128

```
Epoch 50/50
844/844 [==============================] - 1s 2ms/step - loss: 0.3231 - accuracy: 0.8858 - val_loss: 0.3664 - val_accuracy: 0.8723
testing the model
313/313 [==============================] - 1s 1ms/step - loss: 2.4808 - accuracy: 0.0999
```



بررسی: از مقایسه نمودار آبی و سبز متوجه می‌شویم که میزان ضرر در داده آموزشی به شکل بهتری کمینه شده‌است. و این در هر ۴ نمودار صادق است. اما هرچه تعداد نورون‌های لایه مخفی بیشتر می‌شود فاصله میان تابع ضرر val و تابع ضرر train بیشتر می‌شود (به جز حالت ۶۴ نورون که تابع ضرر از ۴ حالت دیگر هم در اعتبارسنجی و هم در آموزش بهتر کمینه شده است).

در این مسئله با افزایش تعداد نورون‌های لایه مخفی، عملکرد کلی شبکه بهبود می‌یابد اما فاصله میان اعتبارسنجی و آموزش نیز تقریبا بیشتر می‌شود.

منابع استفاده‌شده:

https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e

https://blog.tensorflow.org/2018/04/fashion-mnist-with-tfkeras.html

https://hanifi.medium.com/sequential-api-vs-functional-api-model-in-keras-266823d7cd5e