به نام خدا

مهدی فقهی
401722136

سوال چهار)

قسمت الف)

یکی از علت‌هایی که ترجیح می‌دهیم تابع فعالسازی ما خطی نباشد این است که :
مشکل این فعال سازی این است که نمی توان آن را در یک محدوده خاص تعریف کرد. اعمال این تابع در تمام گره ها باعث می شود که تابع فعال سازی مانند رگرسیون خطی عمل کند. لایه نهایی شبکه عصبی به عنوان تابعی خطی از لایه اول کار خواهد کرد. موضوع دیگر نزول گرادیان است که وقتی تمایز انجام می شود، خروجی ثابتی دارد که خوب نیست زیرا در حین انتشار پس انتشار سرعت تغییر خطا ثابت است که می تواند خروجی و منطق پس انتشار را خراب کند.

هر تابع غیرخطی را که نه .
انتخاب تابع فعال سازی متناسب با مسئله و سرعت حرکت‌ای است که باعث حرکت ما به سوی جواب می‌شود است . در جایی ممکن است ترجیح ما به استفاده از تابع خطی نیز باشد اما به طور کلی سعی می کنیم تابع را غیرخطی را برای اینکار انتخاب کنیم که کراندار باشد و مقادیر را بین یک min , max قرار دهد .

قسمت ب)

در این حالت، معادلات الگوریتم یادگیری یا activation function هیچ گونه تغییری در وزن شبکه ایجاد نمی کند و مدل گیر می کند. توجه به این نکته مهم است که وزن بایاس در هر نورون به طور پیش فرض روی صفر تنظیم شده است، نه یک مقدار تصادفی کوچک.
به طور خاص، گره هایی که در کنار هم در یک لایه پنهان متصل به ورودی های یکسان هستند، باید وزنهای متفاوتی برای الگوریتم یادگیری برای به روز رسانی وزن ها داشته باشند.
از طرفی:

If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters . If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.

قسمت ت)

تا زمانی که minimum محلی خود برسند ما کاهش گرادیان را خواهیم داشت . البته برای یک ورودی خاص و تکرار آن به صورت مرتب . و هرچه ورودی پیچیده‌تری داشته باشیم و اگر تابع active function مناسب تعریف نکنیم احتمال اینکه در یک minimum محلی گیر کنیم و دیگر الگوریتم بهبود پیدا نکند بیشتر می‌شود ولی ممکن است به ازی یک ورودی خاص در حالت stotastice مرتب خطا بالا و پایین شود ولی در کل روند به سوی local minimum است در حالت generalization که ما میانگین خطاها را استفاده می‌کنیم به ازای تمامی نمونه‌ها و نه یک نمونه مشخص گرادیان ما به صورت پیوسته‌تری کاهش پیدا می‌کند و بزرگتر نخواهدشد هرچند که ممکن است در ثابت شود .

قسمت ث) در توضیحات هوم ورک قرار گرفته است .

توضیح هوم ورک دوم کد :

برای پیاده سازی تابع relu ابتدا بین بیشترین مقدار ورودی تابع relu و صفر یکی را انتخاب می‌کنیم .
در صورتی که قسمت مشتق فعال باشد خروجی برابر با یک می‌شود .

```
relu_out = np.maximum(x,0)
if  deriv:
  relu_out =  1 * (x > 0)

return relu_out
```

برای پیاده سازی تابع sigmoid ابتدا تابع مقدار تابع را برحسب حاصل ریاضی تابع sigmoid پیدا سازی می‌کنیم .در صورتی که قسمت مشتق فعال باشد خروجی با مشتق برابر می‌شود با حاصل بدست آمده ضرب در مقدار یک منهای حاصل بدست آمده .

```
    sig_out = 1/(1 + np.exp(-x))

    # print(sig_out)
    # print(1 - sig_out)
    if  deriv:
        sig_out = sig_out * (1 - sig_out)

    ########################################
    #     Put your implementation here     #
    ########################################
```

در کلاس MLP برای پیاده سازی feed_forward باید حاصل خروجی هر لایه به عنوان ورودی لایه بعد داده شود تا لایه آخر که حاصل خروجی را بیرون می‌دهد .

از این رو دو آرایه در نظر میگیریم weighted_ins_out_put و res_out_put در اولی حاصل مقدار وزن و بایاس را قرار میدهیم و در دومی مقدار خروجی حاصل از ورودی برحسب وزن و بایاس را پیدا می‌کنیم .

```
    for number_item in range(len(self.act_funcs)):
      weighted_ins_out_put = []
      res_out_put = []
      for input in mlp_out:
          item = self.parameters[number_item]
          function = self.act_funcs[number_item]

          new_input = input.T.dot(item['w']) +  item['b']
          weighted_ins_out_put.append(new_input)
          res_out_put.append(function(new_input))

      self.activations.append(np.array(res_out_put))
      self.weighted_ins.append(np.array(weighted_ins_out_put))
      mlp_out = np.array(res_out_put)
    # print(self.activations[0].shape)
    # print(self.activations[1].shape)
    return mlp_out
```

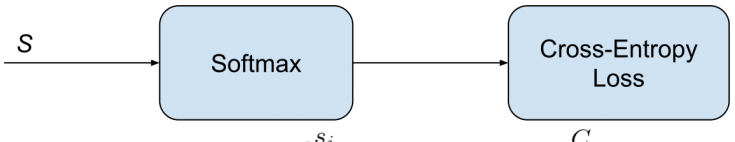
در قسمت بعد تابع softmax را برحسب فرمول ریاضی آن پیاده‌سازی می‌کنیم .

به ازای تمامی ورودی‌های آرایه که هر کدوم یک خود یک آرایه است فرمول را حساب میکنیم و در لیستی اضافه میکنیم و درنهایت لیست موجود را به یک numpy array تبدیل کرده و به عنوان خروجی برمی‌گردانیم .

```
soft_out = []
for item in y_hat:
  soft_out.append(np.exp(item) / np.sum(np.exp(item), axis=0))

return np.array(soft_out)
```

برای پیاده سازی قسمت categorical cross entropy که در آن می‌خواهیم مقدار loss را پیدا کنیم که برابر است با میانگین خطاهای هر آیتم در هر epoch برابر است که این خطا برای هر آیتم از فرمول زیر بدست می‌آید.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

```
loss = []
for number_item in range(len(y)):
  loss.append(- np.sum(y[number_item]*np.log(y_soft[number_item])))
return np.mean(np.array(loss))
```

برای پیاده سازی قسمت بعد که نوشتن قسمت back propagation است باید از آخرین لایه مقدار لایه مقدار خطا را حساب کنیم تا به اولین لایه برسیم .

- calculate gradient of the loss with respect to $\hat{y}$
  $g \longleftarrow \nabla_{\hat{y}} Loss$
- for each layer L starting from the ouput layer:
  $g \longleftarrow g \odot f'(weightedInput^{(L)})$ ($weightedInput^{(L)}$ is the weighted input of Lth layer and f is the activation function)
  $\nabla_{b^{(L)}} Loss \longleftarrow \sum_i^{batch} g_i$
  $\nabla_{w^{(L)}} Loss \longleftarrow output^{(L-1)T} g$ ($output^{(L-1)}$ is the output of $(L-1)$th layer)
  $g \longleftarrow g w^{(L)T}$

محاسبات ریاضی را به ترتیب گفته شده در اینجا حساب می‌کنیم مرحله اول که در کد پیاده سازی شده توسط شما موجود بود فقط قسمت دوم را پیاده سازی میکنیم یک حلقه ایجاد میکنیم و به ازای هر لایه مقدار میزانی که باید وزن‌ها و بایاس باید تغییر کند را حساب می‌نماییم برحسب فرمول ریاضی بالا .

```
for i in reversed(range(num_layers)):
  ########################################
  #     Put your implementation here     #
  ########################################
  g = g * mlp.act_funcs[i](mlp.weighted_ins[i],deriv=True)
  # # print("**********************")
  # # print(g.shape)
  # # print(mlp.activations[i-1].shape)
  # # print("**********************")
  db = np.sum(g,axis=0)
  dw = activations[i].T.dot(g)
  g = g.dot(mlp.parameters[i]['w'].T)
  gradients=[{'w':dw,'b':db}]+gradients
 #gradients.append({'w':dw,'b':db})
# pprint.pprint(gradients)



return gradients
```

در مرحله بعد به مرحله optimization می‌رسیم که در این مرحله باید سه مدل SGD را پیاده سازی کنیم .
در واقع از ما خواسته شده است که تابع gradient descent با مقدار momentum را پیاده‌سازی کنیم .

```
Updated_parameters = []
if len(self.velocity) == 0 :
    self.velocity = [{'dw':g['w'],'db':g['b']} for  g in grads]
    Updated_parameters = [{'w':p['w']-self.lr*v['dw'], 'b':p['b']-self.lr*v['db']} for v, p in zip(self.velocity, parameters)]
else:
    self.velocity = [{'dw':self.momentum*old_v['dw']+(1-self.momentum)*g['w'],'db':self.momentum*old_v['db']+(1-self.momentum)*g['b']}
                     for old_v, g in zip(self.velocity,grads)]
    Updated_parameters = [{'w':p['w']-self.lr*v['dw'], 'b':p['b']-self.lr*v['db']} for v, p in zip(self.velocity, parameters)]

#Updated_parameters = [{'w':p['w']-lr*g['w'], 'b':p['b']-lr*g['b']} for g, p in zip(grads, parameters)]

return Updated_parameters
```

در مرحله اول که مقدار velocity ما برابر با یک لیست خالی است مقدار update parameters را برابر با gradient descent بدون در نظر گرفتن velocity در نظر می‌گیریم در مرحله بعد به کمک فرمول زیر مقدار updated parameters را حساب می‌نماییم .

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$
$$W = W - \alpha V_t$$

در شکل بالا الفا برابر با learning rate و بتا برابر با momentum است .

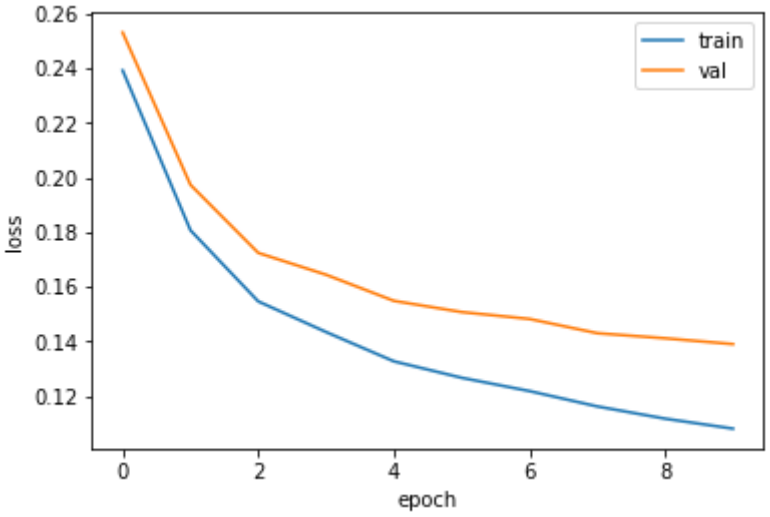
در مرحله بعد شروع به ساخت یک شبکه عصبی به کمک مشخصات زیر کردم .

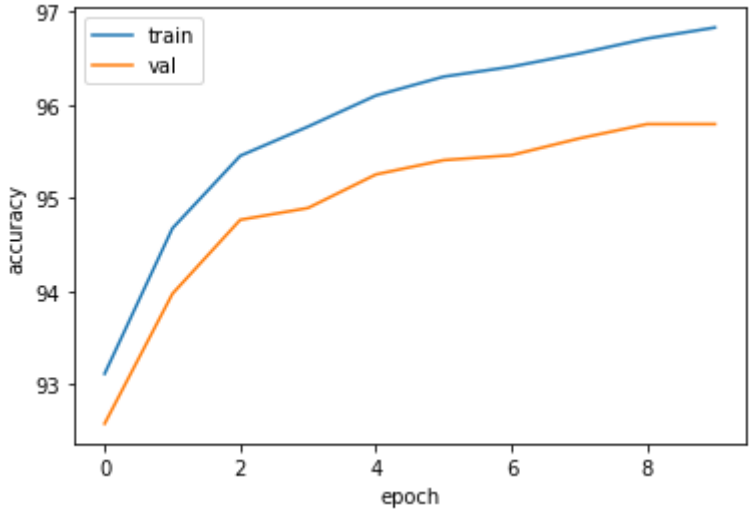
ابتدا با تابع sigmoid در activation function شروع کردم که result نسبت به حاصل با relu مقدار کم دقت تر بود و باز relu نسبت به حالت linear دقت کمتری داشت .

پس از استفاده از linear بعد از چند تست درحالت زیر متوقف شدم زیرا مدل را بهتر یاد می گرفت اما دقت روی train set افزایشی نداشت پس با مقدار لایه‌های زیر متوقف شدم و به حاصل های زیر رسیدم .

```
[ ]  mlp = MLP(x_train.shape[-1])
     mlp.add_layer(10)
     mlp.add_layer(512)
     mlp.add_layer(10)
```

```
epoch 2: 44/? [00:04<00:00, 7.53it/s]
(15000, 10)
(45000, 10)
training acc: 94.68 %
test acc: 93.97 %
epoch 3: 44/? [00:06<00:00, 13.80it/s]
(15000, 10)
(45000, 10)
training acc: 95.45 %
test acc: 94.77 %
epoch 4: 44/? [00:03<00:00, 14.60it/s]
(15000, 10)
(45000, 10)
training acc: 95.77 %
test acc: 94.89 %
epoch 5: 44/? [00:03<00:00, 6.75it/s]
(15000, 10)
(45000, 10)
training acc: 96.10 %
test acc: 95.25 %
epoch 6: 44/? [00:03<00:00, 14.56it/s]
(15000, 10)
(45000, 10)
training acc: 96.30 %
test acc: 95.41 %
epoch 7: 44/? [00:03<00:00, 14.42it/s]
(15000, 10)
(45000, 10)
training acc: 96.41 %
test acc: 95.46 %
epoch 8: 44/? [00:05<00:00, 6.58it/s]
(15000, 10)
(45000, 10)
training acc: 96.55 %
test acc: 95.64 %
epoch 9: 44/? [00:03<00:00, 14.87it/s]
(15000, 10)
(45000, 10)
training acc: 96.71 %
test acc: 95.79 %
epoch 10: 44/? [00:03<00:00, 14.35it/s]
(15000, 10)
(45000, 10)
training acc: 96.83 %
test acc: 95.79 %
```





همانطور که در شکل می‌بینید مدل ما در حال overfit شدن بر روی داده‌های ورودی است .

قسمت ث)

البته اگر تعداد epoch را بالاتر می‌بردم این اتفاق به طور کامل روی میداد ولی همین الان هم از نمودار مشخص است هرچند که با سرعت خوبی مدل train در حال پیش‌روی به سوی خطای صفر است اما مدل validation ما درحال رسیدن به یک حالت است که بیشتر یاد نمیگیرد و شیب نمودار آن درحال صفر شدن و ثابت شدن و ممکن هست به سمت کم شدن نیز پیش رود .

در قسمت استفاده از کتابخانه kersa با sequential مدل مربوط را معرفی می‌کنیم و سپس با add کردن لایه مربوط را اضافه کردم و ترتیب اضافه کردن را نیز مشابه مقاله سایت تعداد نرون‌های هر لایه را مشخص و سپس activation را مشخص کردم .

```
from keras.models import Sequential
from keras.layers import Dense, Dropout,Flatten, Activation

model_dropout = Sequential()
model_dropout.add(Flatten(input_shape=(28, 28)))
model_dropout.add(Dense(128))
model_dropout.add(Dropout(0.5))
model_dropout.add(Activation('relu'))
model_dropout.add(Dense(128))
model_dropout.add(Dropout(0.5))
model_dropout.add(Activation('relu'))
model_dropout.add(Dense(10))
model_dropout.add(Activation('softmax'))

model_dropout.summary()
```

سپس مدل را compile کرده و سپس مدل را با داده مربوط learn می‌کنیم .

```
from keras.optimizers import SGD
sgd = SGD(learning_rate=0.01, momentum=0.8)
model_dropout.compile(loss='sparse_categorical_crossentropy',
              optimizer = sgd,
              metrics=['accuracy'])

history = model_dropout.fit(x_train[:1000].reshape(-1, 28 * 28) , y_train[:1000].reshape(-1, 28 * 28),
                            epochs=30, validation_data=(x_val[:1000].reshape(-1, 28 * 28), y_val[:1000].reshape(-1, 28 * 28)))
```

بعد ۳۰ epoch حاصل مقابل را خواهیم داشت .

```
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0533 - accuracy: 0.9143 - val_loss: 0.0238 - val_accuracy: 0.9633
Epoch 20/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0513 - accuracy: 0.9194 - val_loss: 0.0230 - val_accuracy: 0.9637
Epoch 21/30
1407/1407 [==============================] - 6s 5ms/step - loss: 0.0513 - accuracy: 0.9183 - val_loss: 0.0229 - val_accuracy: 0.9649
Epoch 22/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0507 - accuracy: 0.9173 - val_loss: 0.0223 - val_accuracy: 0.9653
Epoch 23/30
1407/1407 [==============================] - 6s 5ms/step - loss: 0.0491 - accuracy: 0.9213 - val_loss: 0.0219 - val_accuracy: 0.9662
Epoch 24/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0494 - accuracy: 0.9213 - val_loss: 0.0218 - val_accuracy: 0.9671
Epoch 25/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0483 - accuracy: 0.9244 - val_loss: 0.0213 - val_accuracy: 0.9673
Epoch 26/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0479 - accuracy: 0.9231 - val_loss: 0.0209 - val_accuracy: 0.9675
Epoch 27/30
1407/1407 [==============================] - 6s 5ms/step - loss: 0.0467 - accuracy: 0.9254 - val_loss: 0.0206 - val_accuracy: 0.9686
Epoch 28/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0467 - accuracy: 0.9250 - val_loss: 0.0205 - val_accuracy: 0.9686
Epoch 29/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0454 - accuracy: 0.9276 - val_loss: 0.0201 - val_accuracy: 0.9695
Epoch 30/30
1407/1407 [==============================] - 7s 5ms/step - loss: 0.0448 - accuracy: 0.9296 - val_loss: 0.0199 - val_accuracy: 0.9698
```