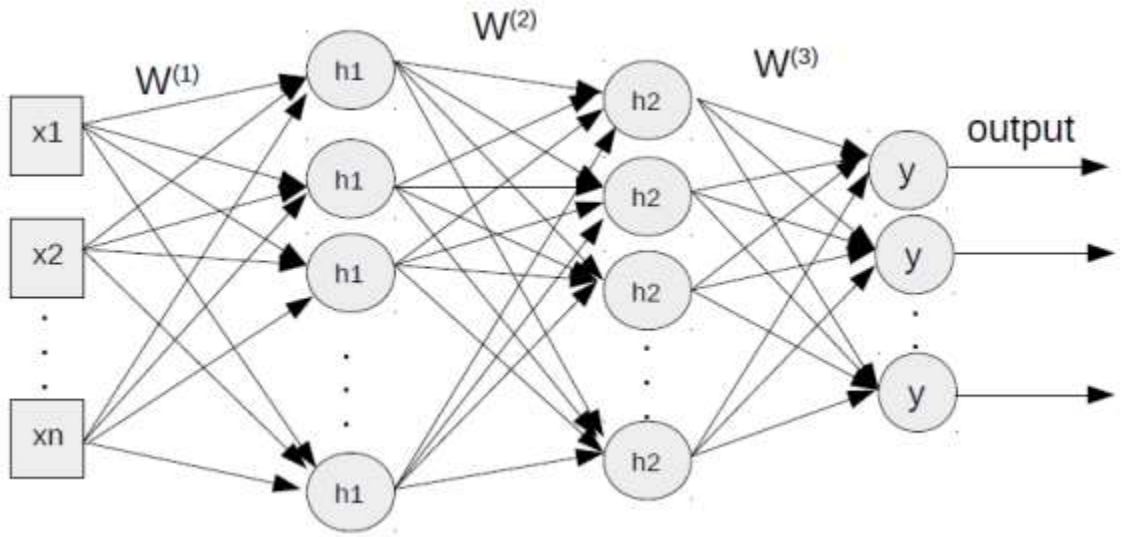
گزارش درس یادگیری ماشین

دکتر سیدین

زهرا عربی

۹۵۲۳۰۸۳

ساختن شبکه عصبی چندلایه با روش پس انتشار خطا



$$h1 = g(W^{(1)}x + b^{(1)})$$

الگوریتم کلی در این شبکه ها به این صورت است که در هر نورون ورودی ها در وزنها ضرب شده و مجموع تمام این حاصل ضرب ها از یک تابع فعالساز گذر میکند و نتیجه به عنوان خروجی نورودن در آن لایه مشخص میشود.

بردار خروجی کل آن لایه برابر خواهد بود با مجموعه خروجی های هر نورون در آن لایه.

نکته ی قابل توجه در این الگوریتم تعیین مناسب وزنها است ، به گونه ای که مقدار خطای بدست آمده از داده های تست از یک مقدار مشخصی کمتر بشود.

در واقع شبکه‌ی عصبی به دنبالِ یادگیری از طریقِ تغییر در **وزن‌ها (W)و انحراف‌ها (b)**است. از این رو یادگیری در شبکه های عصبی با **تکرار** انجام می‌شود. یعنی چندین مرتبه داده‌های یک مجموعه داده (Dataset)به الگوریتم تزریق می‌شود و این الگوریتم با کم و زیاد کردن وزن‌ها و انحراف‌ها، می‌تواند تفاوت‌ها را در داده‌های آموزشی تشخیص دهد.

این الگوریتم به روش پس انتشار خطا عمل میکند به گونه ای که در ابتدا به روش feedforwardیا حرکت رو به جلو شبکه با یک سری وزنها ساخته میشود و سپس با محاسبه ی مقدار خطا از ابتدا شروع به اصلاح وزنها میکنیم به گونه ای که خطا کاهش یابد.

داده های آموزشی ای در این پروژه استفاده شده است داده های MNISTهستند.

دیتاست mnist

این مجموعه داده شامل تعداد زیادی عکس های دست نوشته از اعداد به همراه برچسب های متناظر است.

هر عکس از ۷۸۴ پیکسل تشکیل شده است در نتیجه هر بردار ورودی شامل ۷۸۴ المان است(که همان ویژگی های هر نمونه ی آموشی است.)

از آنجایی که ده رقم از ۰تا۹ در عکسها موجود است خروجی ۱۰ کلاسه است. در کد مربوط به الگوریتم نحوه ی استخراج داده ها از پکیج مربوط به تنسورفلو در پایتون با جزییات بیشتر ی بررسی میکنیم.

گام اول:

پس از import کردن پکیج tensorflow به داده های MNIST را توسط دستورات زیر دسترسی پیدا میکنیم.

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

#%% data

# read mnist data. If the data is there, it will not download again.
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('/home/nasir/dataRsrch/imageDBMain/MNIST/', one_hot=True)
```

گام ۲:تعریف تابع multi_layer_perceptron_mnist

ورودی های تابع مقدار ورودی شبکه یا همان بردار x ،بردار وزن w و بردار بایاس b است.

۱. حال نوبت به ساختن شبکه میرسد.با توجه به الگوریتم گفته شده در قسمت قبل در هر لایه باید ضرب ماتریسی بردار ورودی لایه در بردار وزن ضرب شده و با مقدار بایاس جمع گردد.
دستور متناظر با این ضرب در تنسورفلو tf.matmul بوده و برای جمع از tf.add استفاده میکنیم.

۲. مقدار حاصل این ترکیب خطی از یک تابع فعالساز عبور میکند. با توجه به خواسته های مساله در این پروژه یک بار از فعالساز همانی یا unity و بار دیگر از فعالساز simoidاستفاده میکنیم.

برای لایه ی اول مقدار ورودی برابر با x و بردار وزن مربوط به همان لایه است.در لایه های بعد بردار خروجی لایه قبل(خروجی پس از عبور از تابع فعالساز) به عنوان ورودی جدید بوده و در وزن همان لایه ضرب شده با بایاس متناظر با لایه ی جدید جمع میشود .

به همین ترتیب خروجی هر لایه به لایه بعد مرتبط شده و در آخر به یک بردار ۱۰ المانه از خروجی های میرسیم.

در خروجی تابع این بردار بازگردانده میشود.

```
def multi_layer_perceptron_mnist(x, weights, biases):

    hidden_layer1 = tf.add(tf.matmul(x, weights['w_h1']), biases['b_h1'])
    hidden_layer1 = tf.nn.relu(hidden_layer1)    # apply ReLU non-linearity
    hidden_layer2 = tf.add(tf.matmul(hidden_layer1, weights['w_h2']), biases['b_h2'])
    hidden_layer2 = tf.nn.relu(hidden_layer2)

    out_layer = tf.add(tf.matmul(hidden_layer2, weights['w_out']), biases['b_out'])

    return out_layer
```

گام ۳: تعیین پارامترها و متغیرهای مدل

متغیر learning_rate یا همان num_iter، برابر با تعداد epoch یا تعداد دفعاتی که نیاز است یک دور به شکل کامل از داده های آموزش برای trainکردن شبکه استفاده کنیم. batch_size سایز را مشخص میکند (هر چند داده را با هم استفاده کنیم و خطا را محاسبه کنیم) ،display_step برای تعیین تعداد خطاهاست که بعد از هر display_step خطای متوالی از آنها میانگین گیری میکنیم و به صورت نمودار تغییرات میانگین خطا را با پیش روی یادگیری شبکه بررسی میکنیم.

متغیر های num_input، num_hidden1 ، num_hidden2 ، num_output به ترتیب تعداد المان های بردار ورودی ،تعداد نورونها در لایه پنهان اول ، لایه پنهان دوم و تعداد المانهای بردار خروجی است.

حال متغیرهای وردودی و خورجی را با tf. Placehoderتعریف میکنیم.از این دستور برای مشخص کردن ساختا و قالب یک متغیر استفاده میکنیم .

تفاوت آن با tf.Variable در این است که دستور tf.Variable برای متغیرهای شبکه مثل وزنها و بایاسها و متغیر های قابل آموزش کاربرد دارد.

```
# hyper-parameters
learning_rate = 0.01
num_iter = 30
batch_size = 100
display_step = 10        # display the avg cost af

# variables
num_input = 784          # units in the input laye
num_hidden1 = 128        # units in the first hidc
num_hidden2 = 256
num_output = 10          # units in the output lay

# trainum_inputg data and labels
x = tf.placeholder('float', [None, num_input])
y = tf.placeholder('float', [None, num_output])
```

گام ۴: تعریف و مقداردهی اولیه ی وزنها و بایاسها

از دیشکنری پایتون برای تعریف این متغیرها استفاده شده است ، به گونه ای که مقادیر کلیدهای دیکشنری w1,w2,w3 و محتوا یا آنها یک ماتریس با ایعاد مقدار ورودی در بعد خوروجی همان لایه ، با مقادیر رندوم با توزیع نرمال گوسی است.
دیکشنری مربوط به بایاس نیز به همین صورت تعریف شده و بعد ماتریس متناظر با هر لایه برابر تعداد نورونهای لایه است .

گام ۵: فراخوانی تابع multi_layer_perceptron_mnist برای ساختن شبکه با ورودیهای تعریف شده ،تعریف تابع ضرر و اپتیمایزر

پس از فراخوانی و ساخت شبکه ، به سراغ تابع هزینه میرویم.

تابع هزینه ی مورد نظر برای این مسئله تابع Root Mean Square Error در نظر گرفته شده است که از ضابطه ی زیر برای محاسبه ی ضرر استفاده میکند.

Root Mean Square Error (RMSE)

$$RMSE = \sqrt{\frac{1}{Q}\sum_{q=1}^{Q}\left(y^q - \hat{y}^q\right)^2}$$

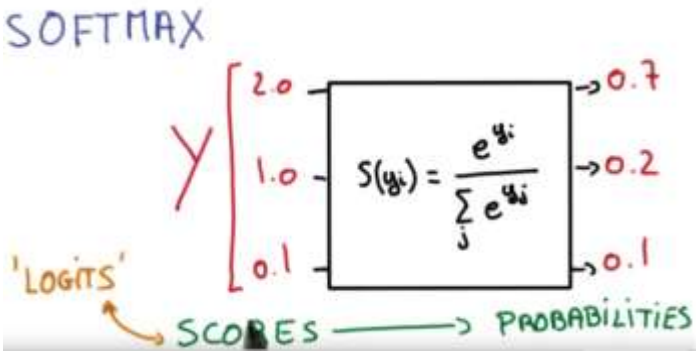
برای محاسبه ی مربع خطا از دستور tf.nn.softmax_cross_entropy_with_logits استفاده شده است.

آرگمان های ورودی این متود مقدار خروجی و مقدار برچسب صحیح آن است .در واقع این دستور مقدار شباهت دو آرگمان ورودی را برمیگرداند.

منطق کد بر مبنای مفهوم cross validityاستوار است .

$$H(p,q) = -\sum_{x} p(x)\ \log q(x).$$

با محاسبه ی آنتروپی مربوط به احتمال pو q را محاسبه میکنیم. در واقع میزان شباهت این دو را بررسی میکنیم.
از طرفی مقادیر خروجی ها و مقدار برچسب ها لزوما اعداد کوچکتر از یک نیستند. از این رو از دستور softmaxبرای تغییر مقادیر آنها استفاده میشود به این صورت که با اعمال این دستور مقادیر مربوطه با استفاده از رابطه ی زیر به فرم احتمال در می آیند.
احتمال هر کدام از ۱۰ مقدار yنسبت به کل المانهای y.



در واقع الگوریتم بیان شده معادل با این کد است :

```
loss_per_instance_1 = -tf.reduce_sum(y_true * tf.log(y_hat_softmax), reduction_indices=[1])
```

که به صورت معادل به جای این کد میتوانیم از tf.nn.softmax_cross_entropy_with_logits استفاده کنیم.

حال برای میانگین گیری دستور tf. reduce_mean را به کار میگیریم.(توضیحات بیشتر درباره ی tf. reduce_mean در پیوست قرار دارد)

در نهایت از یک آپتیمایزر استفاده میکنیم به گونه ای که با یک مقدار مشخص سرعت یادگیری در جهت کاهش مقدار خطا یا خروجیloss function پیش رود.

```
model = multi_layer_perceptron_mnist(x, weights, biases)

# cost function and optimization
loss_func = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss_func)
```

گام۶: graf و session

Session وGraph دو ساختار مهم در کدهای یادگیری ماشین و با استفاده از تنسورفلو به چشم میخورد. گراف در واقع ساختار محاسباتی را تعریف می کند نه عملیات محاسباتی انجام می‌دهد و نه شامل داده ای هست تنها عملیاتی که می بایست روی ورودی انجام شود را نگهداری می کند .در بخش های قبل با معرفی شبکه در واقع گراف مربوطه را تشکیل دادیم.
هدف Session اجرای کل یا بخشی از گراف است بطوریکه حافظه مورد نیاز جهت اجرا اشغال میشود و در واقع ثابت ها و متغیرهای تعریف شده توسط این نگهداری می شود.

در این بخش باید یک Session تعذیف کنیم و آنرا اجرا کنیم.

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
```

گام۷: train کردن مدل

```
# Train the model
for iter in range(num_iter):
    avg_cost = 0.0
    num_batch = int(mnist.train.num_examples / batch_size)   # total number of batches
    for nB in range(num_batch):
        trainData, trainLabels = mnist.train.next_batch(batch_size=batch_size)
        tmp_cost, _ = sess.run([loss_func, optimizer], feed_dict={x: trainData, y: trainLabels})

        avg_cost = avg_cost + tmp_cost / num_batch

    correct_pred = tf.equal(tf.arg_max(model, 1), tf.arg_max(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_pred, 'float'))
    acc = accuracy.eval(session=sess, feed_dict={x: mnist.test.images, y: mnist.test.labels})

    if iter % display_step == 0:
        print('Epoch: %04d' %(iter+1), 'cost= ' + "{:.5f}".format(avg_cost), 'accuracy: ' + "{:.5
    cost_all = np.append(cost_all, avg_cost)
    acc_all = np.append(acc_all, acc)
```

برای آموزش شبکه از دو حلقه ی تودرتو استفاده میکنیم. حلقه ی بیرونی به اندازه ی تعداد epochها پیشرفته و دومی به اندازه ی تعداد داده ها تقسیم بر batch_numپیش میرود.

در حلقه ی درونی به ازای تعداد batchها ، به اندازه ی سایز batch داده از برمیدارد .خروجی دستور دو متغیر است که متناظر با train Data و trainLableاست.

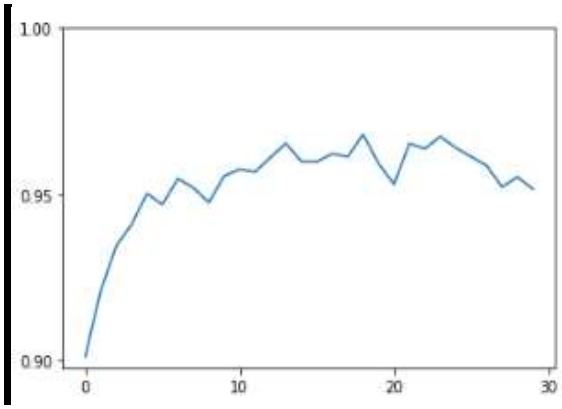
تابع هزینه و اپتمایزر را با این مقادیر ورودی بدست می آوریم.سپس مقدار خطا یا خروجی تابع هزینه را برای محاسبه ی میانگین و رسم نمودار استفاده میکنیم.

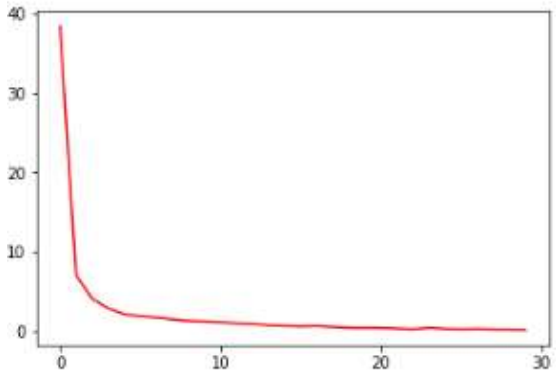
مقدار avg_cost مقدار میانگین خطا پس از هر batchاست.برای بررسی صحت نتیجه ی حاصل متغیرcorrect_pred را برابر خروجی مقایسه برچسب و خروجی مدل قرار میدهیم.برای بدست آوردن میزان صحت را accuracy برابر با میانگین این متغیر قرار میدهیم.

برای محاسبه ی مقدار accuracy باید با داده ها آنرا بررسی کنیم. از اینرو مقدارش را به ازای تمام داده های mnist بررسی میکنیم.

بررسی نمودارها و مقادیر کمی مربوط به خطا و صحت

با استفاده از مقادیر بدست آمده میتوان نمودار تغییر خطای کل با افزایش داده ها و مقدار accuracy یا صحت رسم کرد که به صورت روبرو در می آید.





میزان صحت

مقدار هزینه

همانطور که از شکل مشخص است مقدار هزینه با یادگیری شبکه کاهش می یابد (خطا کم میشود)

و میزان دقت و صحت پاسخ ها نیز بالاتر میرود.

```
Epoch: 0001 cost= 38.38015 accuracy: 0.90120
Epoch: 0011 cost= 1.13958 accuracy: 0.95750
Epoch: 0021 cost= 0.42379 accuracy: 0.95300
Optimization done...
```

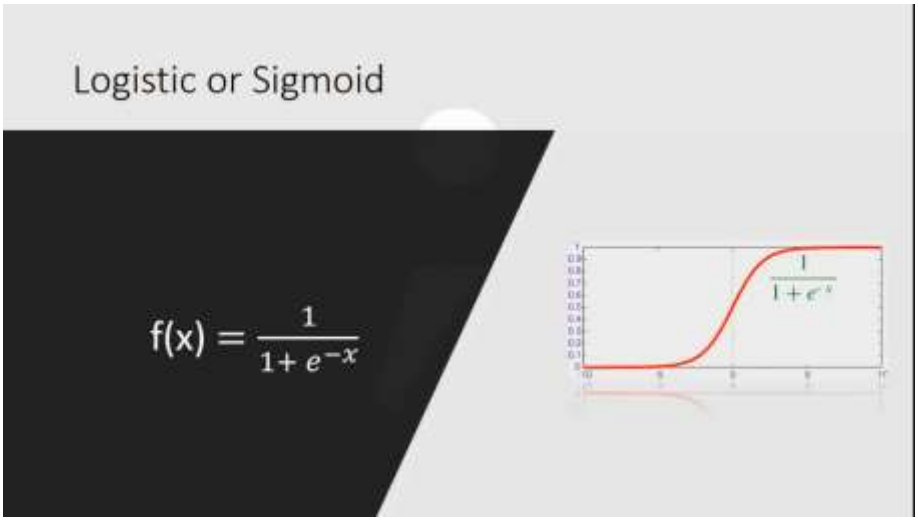
مقدار accuracy و cost به ازای هر display_step محاسبه شده است.با افزایش مقدار iter(شماره ی نمونه) دقت بالا رفته و costکاهش پیدا کرده است. به ازای iter های ۱۱ به بالا مقدار accuracy حول۰,۹۵ قرار دارد.

به ازای iter های نزدیک ۰ مقدار خطا بسار بالا است اما با افزایش iter خطا ناگهانی کاهش پیدا میکند و حول ۰,۴ قرار میگیرد. در واقع در نمودار شاهد نقطه ی زانو هستیم.

این نقطه که در آن تغییرات خطا بسیار شدید است نقطه ی مناسبی برای سنجش تعداد iter های لازم برای اطمینان از کوچک بودن خطا است .

فعالساز سیگموید(sigmoid)

این فعالساز با ضابطه ی زیر عمل میکند:



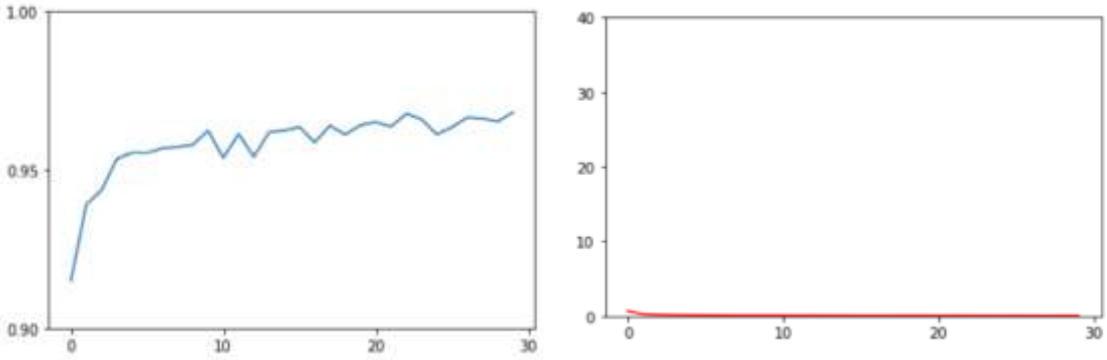
ساختن شبکه با استفاده از فعالساز sigmoid

در بخش قبل از فعالساز reLu استفاده شد و با تعلیم شبکه با این فعالساز نتایج حاصله پس از تست شبکه با داده های تست مقدار صحت و خطا را بررسی کردیم.

در این بخش فعالساز سیگموید را مورد بررسی قرار میدهیم.

```
hidden_layer1 = tf.add(tf.matmul(x, weights['w_h1']), biases['b_h1'])
hidden_layer1 = tf.nn.sigmoid(hidden_layer1)    # apply ReLU non-linearity
hidden_layer2 = tf.add(tf.matmul(hidden_layer1, weights['w_h2']), biases['b_h2'])
hidden_layer2 = tf.nn.sigmoid(hidden_layer2)
```

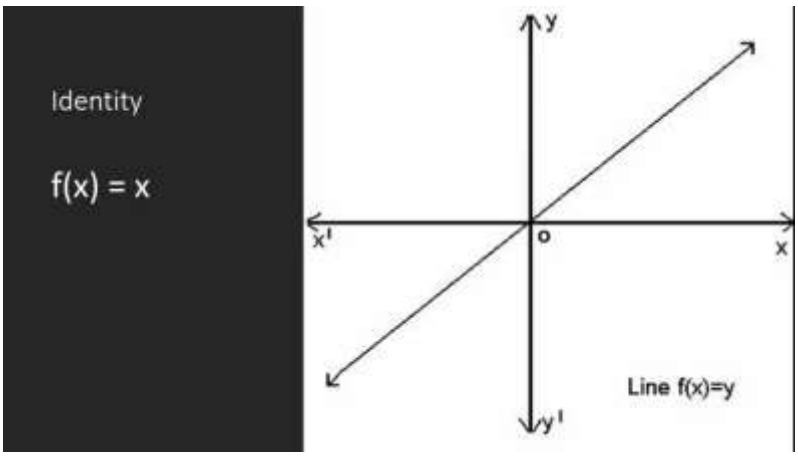
در کد بخش قبل تنها کافیست این بخش از کد را تغییر دهیم و بجای tf.nn.reLu از tf.nn.sigmoid استفاده کنیم.




```
Epoch: 0001 cost= 0.67607 accuracy: 0.91500
Epoch: 0011 cost= 0.08065 accuracy: 0.95380
Epoch: 0021 cost= 0.05839 accuracy: 0.96510
Optimization done...
```

با توجه به مقادیر بدست آمده همانطور که مشاهده میشود در اطراف صفر مقدار خطا تقریبا به ۰٫۶۷نزدیک است . با افزایش این مقدار همچنان کاهش می یابد و به نزدیکی صفر میرسد . به گونه ای که در نمونه ی ۲۱۰ این مقدار به ۰٫۰۵۸ میرسد .در فعالساز reLu. در شماره نمونه ی مشابه خطا بیشتر از این مقدار بود که این موارد عملکرد بهتر سیگموید را نشان میدهد . مقدار صحت نیز در شماره نمونه های مشابه برای فعالساز سیگموید بیشتر از Reluاست.

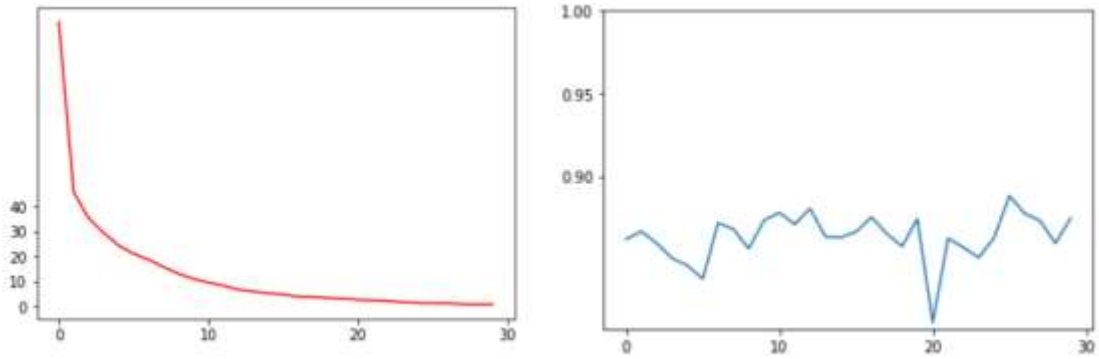
استفاده از فعالساز همانی



این فعالساز تغییری برای مقدار ورودی اش انجام نمیدهد ، در واقع مثل این است که اصن از فعالسازی استفاده نکرده ایم.

در کد دو خط مربوط به فعالساز را حذف میکنیم.

بررسی خطا و صحت




```
Epoch: 0001 cost= 113.67635 accuracy: 0.86200
Epoch: 0011 cost= 9.60862 accuracy: 0.87810
Epoch: 0021 cost= 2.70788 accuracy: 0.81170
Optimization done...
```

همانطور که از نتایج برمی آید هم مقدار خطا بسیار افزایش پیدا کرده و هم کاهش داشته است. در واقع استفاده از تابع همانی نسبت به دو فعالساز قبلی اصلا بهینه و دقیق نیست.

تغییر سایز batch

سایز مشخص میکند که مقدار خطا را روی چند داده ی آموزشی متوالی حساب کنیم. و متناسب با میزان خطا براداروزن را به گونه ای تغییر دهیم که خطا کاهش یابد .

در کد های قبل داده ها را به صورت دسته ای مورد بررسی قرار دادیم و فرضا بعد از هر ۱۰ داده وزنها را آپدیت کند.