شبکه‌های عصبی

دانشگاه فردوسی مشهد

تمرین دو

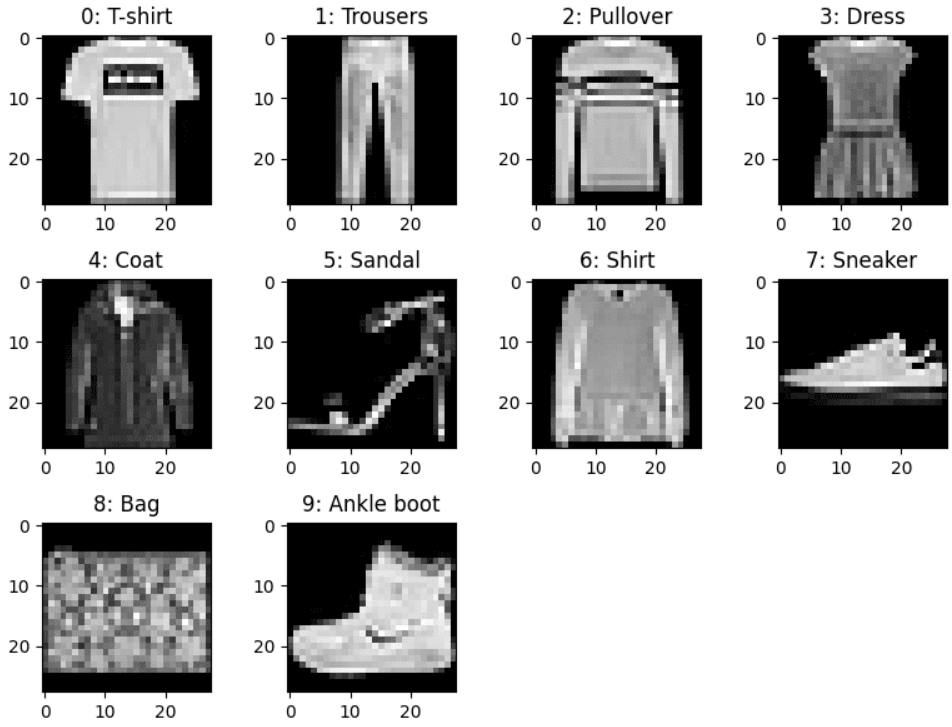
نیمسال دوم تحصیلی ۱۴۰۳-۱۴۰۲

گروه مهندسی کامپیوتر | مهلت ارسال: ۱۴۰۳/۱/۲۰ ۲۳:۵۹

در این تمرین قصد داریم از شبکه MLP که در تمرین قبل کد آن را آماده کرده اید برای طبقه بندی دیتاست Fashion MNIST استفاده کنیم و به بررسی تاثیرات مفاهیم مختلف شبکه های عصبی از جمله توابع متفاوت فعالیت، وزن‌دهی اولیه وزن‌ها، mini-batch gradient descent و بهینه‌سازهای متفاوت اسن. برای پیاده سازی از توابع استاندارد Python و Numpy استفاده کنید و از Toolbox های مختلف مربوط به دیپ لرنینگ استفاده نکنید. در ابتدا به توضیح دیتاست می‌پردازیم و سپس مواردی که باید پیاده شوند توضیح داده می‌شود.

دیتاست Fashion MNIST

دیتاست Fashion MNIST این دیتاست همانند دیتاست MNIST شامل ۶۰۰۰۰ عکس به عنوان داده آموزش و ۱۰۰۰۰ عکس به عنوان داده تست است و داده ها مروبط به ۱۰ نوع لباس مختلف همانند کفش، شلوار و ... می‌باشد. عکس ها به صورت grayscale هستند و ابعاد آنها ۲۸ در ۲۸ می‌باشد و مقادیر پیکسل ها از ۰ تا ۲۵۵ است.



شکل ۱: چند نمونه از داده های Fashion MNIST.

برای این تمرین، از آنجایی که این دیتاست شامل ۱۰ کلاس است، دو کلاس ابتدایی آن که پیراهن و شلوار است

را جدا شده اند و با فرمت npz در اختیار شما قرار داده شده است. برای لود کردن داده ها از دستورات زیر استفاده کنید.

```
x_train = np.load('x_train.npz')['arr_0']
x_test = np.load('x_test.npz')['arr_0']
y_train = np.load('y_train.npz')['arr_0']
y_test = np.load('y_test.npz')['arr_0']
```

توابع فعالیت

هدف در این قسمت بررسی دقیق تر توابع فعالیت و ویژگی‌های آن‌ها می‌باشد.

۱.۱) تابع فعالیت همانی

تابع فعالیت همانی، ورودی نورون را به همان مقدار برای خروجی نورون نسبت می‌دهد.

$$f(x) = x$$

استفاده از این تابع فعالیت برای داده‌های غیر جداپذیر خطی ممکن نیست. علت این امر این است که شبکه ای که نورون های لایه‌های مخفی آن از این تابع استفاده کنند، صرف نظر از تعداد لایه ها و تعداد نورون‌های هر لایه همواره تابعی خطی را یاد می‌گیرد.

۲.۱) Sigmoid (Logistic Activation Function)

$$f(x) = \frac{1}{1 + e^{-x}}$$

این تابع به ازای ورودی‌های منفی تر و مثبت از مقداری، شیب کمی دارد. این امر باعث می‌شود که در فرآیندهای اپدیت گرادیانی که نیاز به مشتق گرفتن از توابع فعالیت است،مشتق‌ها کوچک باشند و فرآیند همگرایی به کندی برای رنج زیادی از مقادیر انجام شود. استفاده معمول تر از تابع در لایه آخر برای طبقه بندی دو کلاسه است.

۳.۱) Tanh or hyperbolic tangent Activation Function

$$f(x) = 2 \times sigmoid(2x) - 1$$

این تابع فعالیت نیز همانند سیگموید s شکل است. از آنجایی که خروجی این تابه بین -۱ و ۱ قرار دارد و یک تابع فعالیت zero-centered است، مقادیر خروجی نورون ها را اطراف ۰ نگاه می‌دارند و به عبارتی از بزرگ شدن یا کوچک شدن گرادیان‌ها جلوگیری می‌کند. به همین علت از این تابع بیشتر در لایه‌های مخفی و از سیگموید در لایه آخر استفاده می‌شود.

این تابع همچنان مشکل تابع سیگموید در کند کردن فرآیند آموزش و vanishing gradient به خصوص در شبکه‌های عمیق را دارد.

۴.(۱ ReLU

$$f(x) = \max(0, x)$$

این تابع سرعت خوبی برای رسیدن به همگرایی دارد و از مشکل اشباع گرادیان برای ورودی های مثبت جلوگیری می‌کندو به مشکل vanishing gradients را نیز کمک می‌کند. این تابع مشکلی دارد که ان را dying ReLU problem می‌نامند. زمانی که اکثر داده ورودی در رنج منفی قرار بگیرد چون این تابع خروجی ۰ می‌دهد و گرادیان ۰ نیز ۰ است، در فرآیند BP گرادیان‌ها نمیتوانند به عقب بازگردند و وزنها آپدیت نمی‌شوند.

۵.(۱ Leaky ReLU

این تابع شرایط ReLU را دارد و برای همگرایی سریع است و دچار dying ReLU problem نمی‌شود چون برای مقادیر منفی نیز مقداری شیب منفی و درنتیجه گرادیان منفی دارد.

وزن‌دهی اولیه پارمترهای شبکه

آموزش شبکه عصبی به میزان قابل توجهی به وزن‌دهی اولیه پارمترها بستگی دارد. این نقطه اولیه میتواند حتی در اینکه شبکه همگرا بشود یا نه هم تاثیر داشته باشد. به عنوان مثال مقدار دهی اولیه می‌تواند به گونه‌ای unstable انجام شود که مشکلات عددی به‌وجود بیاید. حتی در زمانی که یادگیری همگرا می‌شود، مقدار دهی اولیه مناسب میتواند در اینکه با چه سرعتی همگرایی به‌دست می‌آید یا به نقطه ای با چه هزینه‌ای همگرا می‌شود تاثیر داشته باشد.علاوه بر این، وزن‌دهی اولیه بر تعمیم پذیری نیز اثر دارد.

۱.(۲ وزن دهی تمام صفر

در این روش تمام پارامترهای اولیه با صفر مقداردهی اولیه می‌شوند.

در این روش یکی از اصول مهم در وزن‌دهی اولی که نیاز به از بین بردن تقارن یا break symmetry است انجام نمی‌شود. دو نورون مخفی که با تابع فعالیت یکسان که به ورودی‌های مشابهی متصل هستند، باید پارامتر های اولیه متفاوتی داشته باشند و اگر اینگونه نباشد یک الگوریتم یادگیری قطعی (deterministic) که بر روی یک تابع هزینه قطعی اعمال شده است، به صورت دائم هر دو نورون را به یک صورت آپدیت خواهد کرد.

حتی در صورت آموزش مدل با الگوریتمی که که میتواند از نایقینی (stochasticity) برای محاسبه آپدیت‌های نورون‌ها استفاده کند(به طور مثال با استفاده از dropout) باز هم بهتر است وزن ها هر نورون به گونه‌ای تعیین شوند که هر نورون تابعی متفاوت از نورون‌های دیگر محاسبه کند.

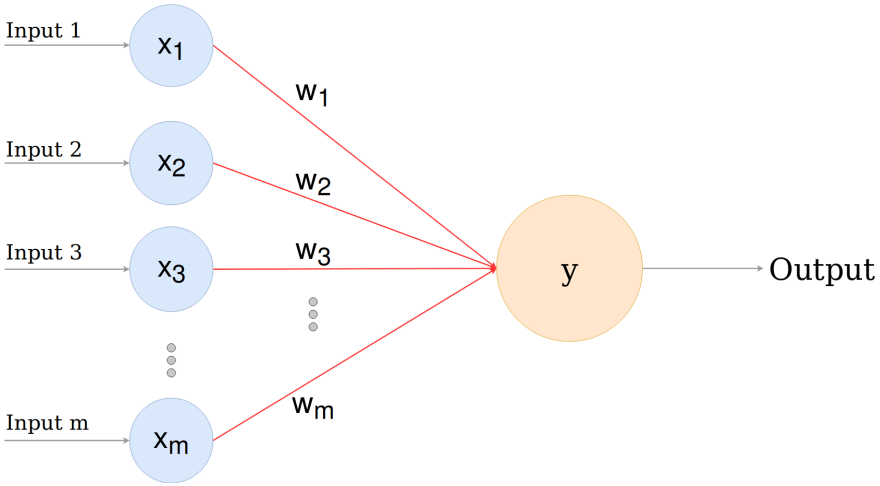
۲.(۲ وزن‌دهی رندم

روش دیگر وزن‌دهی اولیه بامقادیر رندوم از توزیع یکنواخت یا گوسی می‌باشد. انتخاب بین این دو نوع توزیع اهمیت چندانی ندارد و به طور جدی نیز بررسی نشده‌است. هر بار که وزن‌های شبکه مقداردهی اولیه می‌شوند باعث می‌شود یک مجموعه متفاوت از وزن‌ها به معنای یک نقطه شروع جدید در فرآیند بهینه‌سازی و احتمالا مقادیر نهایی متفاوت وزن‌ها(با ویژگی‌ها و عملکرد متفاوت) ایجاد می‌شود.

روش‌های همخوان‌شده تری با انواع مختلف توابع فعالیت برای مقداردهی اولیه وزن‌ها وجود دارد که در ادامه به بررسی برخی خواهیم پرداخت. همچنین در مقداردهی باید به این نکته توجه داشت که مقادیر اولیه متفاوت برای این وزن‌ها می‌تواند به این کمک کند که هیچ الگو ورودی (Input pattern) در فضای پوچ پیمایش رو به جلو (forward propagation) گم نشود و همچنین هیچ الگو گرادیانی در فضای پوچ (back-propagation) از دست نرود.

۳.(۲ مقداردهی دهی اولیه وزن‌ها برای توابع فعالیت Sigmoid و Tanh

یک نورون پرسپترون را در نظر بگیرید. اگر داشته باشیم:



شکل ۲: یک نورون perceptron

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n,\ y = g(z)$$

هر چه قدر تعداد ورودی‌ها بیشتر باشد، برای اینکه z بزرگ نشود(به منظور جلوگیری از اشباع تابع فعالیت)، باید w کوچکتر باشد. اگر W ماتریس وزن این نورون باشد، به دلیل نگاه داشتن این رابطه معکوس می‌توان برای واریانس این ماتریس نوشت:

$$Var(W) = \frac{1}{n}$$

بر همین اساس برای مقداردهی اولیه وزن ها در شبکه ای با توابع فعالیت Sigmoid و Tanh
از روشی به نام Xavier استفاده می‌شود که پیاده سازی آن در پایتون می‌تواند به صورت زیر انجام بگیرد:

```python
import numpy as np

def initialize_weights(shape):
    n = shape[0]  # Number of input units
    return np.random.randn(*shape) * np.sqrt(1/n)

# Example usage:
shape = (10, 5)  # Example shape for the weight matrix (input_size, output_size)
W1 = initialize_weights(shape)
```

شکل ۳: Xavier Initialization

این مقدار دهی وزن‌ها را بر اساس یک توزیع گوسی با واریانس $\frac{1}{n}$ مقدار می‌دهد. (از توزیع یکنواخت نیز می‌توان استفاده کرد)

۴.(۲ مقداردهی دهی اولیه وزن‌ها برای تابع فعالیت ReLU

روش Xavier برای تابع فعالیت ReLU مشکلاتی داردو به همین منظور از روش دیگری به نام He Initialization استفاده می‌شود.این مقدار دهی وزن‌ها را بر اساس یک توزیع گوسی با واریانس $\frac{2}{n}$ مقدار می‌دهد. (از توزیع یکنواخت نیز می‌توان استفاده کرد)

MiniBatch Gradient descent

الگوریتم Gradient descent که به آن Batch Gradient descent هم گفته می‌شود، برای هر گام آپدیت گرادیانی یک بار همه داده‌های دیتاست آموزشی را طی می‌کند و بعد وزن‌هارا یک بار آپدیت می‌کند. برای این الگوریتم اگر نمودار تابع هزینه بر حسب شماره پیمایش دیتاست آموزشی رسم کنیم مشاهده میشود که همواره نمودار کاهشی است و نویزی نمی‌باشد. در این الگوریتم می‌توان نرخ یادگیری را بزرگ‌تر اختیار کرد و گام‌های آپدیت بلند با نوبز کم برمی‌دارد.

الگوریتم MiniBatch Gradient descent داده را به قسمت‌هایی (mini-batch) تقسیم می‌کند به این صورت که برای داده‌های هر قسمت از داده‌های آموزش سمپل گرفته می‌شود و اپدیت وزن ها بعد از پیمایش هر mini-batch

صورت می‌گیرد. این فرآیند باعث ایجاد stochasticity در در فرآیند بهینه‌سازی و ایجاد نویز در تخمین گرادیان‌ها می‌شود به علت استفاده از زیرمجموعه‌هایی از داده‌های آموزش به جای کل دیتاست آموزشی. هر چه قدر اندازه mini-batch کوچکتر باشد نویز هم بیشتر می‌شود. برای این الگوریتم اگر نمودار تابع هزینه بر حسب شماره پیمایش دیتاست آموزشی رسم کنیم مشاهده می‌شود که نمودار نویزی است اما روند کلی کاهشی دارد.

از آنجایی که در مسائل امروزه شبکه عصبی سایز دیتاست های آموزشی بسیار بالا می‌باشد اسفاده از روش full batch که با پیمایش کل دیتاست تنها یک‌بار وزن‌ها را آپدیت می‌کند در مسائل با سایز زیاد داده ممکن نیست و به همین دلیل از روش mini-batch استفاده می‌شود. معمولا سایز mini-batch را توانی از ۲ استفاده می‌کنند و استفاده از سایزهای ۶۴، ۱۲۸، ۲۵۶ و ۵۱۲ مرسوم است.

Batch Gradient descent

for epoch in number of epochs:
- for all the training instances in the dataset compute the derivative of the cost function
- update the weights

Mini-batch Gradient descent

for epoch in number of epochs:
for batch in num of batches:
- for all the training instances in the batch sample compute the derivative of the cost function
- update the weights

Stochastic Gradient descent (SGD)

در الگوریتم MiniBatch Gradient descent یک حالت extreme این است که سایز mini-batch برابر با سایز دیتاست آموزشی باشد که در این حالت به همان الگوریتم گرادیان کاهشی فول بچ می‌رسیم. حالت دیگر extreme این است که سایز هر mini-batch برابر ۱ باشد و به عبارتی هر داده خود یک mini-batch باشد. در این حالت به الگوریتم Stochastic Gradient descent (SGD) می‌رسیم. این الگوریتم در هر Iteration تنها یک sample از داده را می‌بینند. در برخی از اپدیت ها گام‌ها در مسیر global optimum پیش می‌روند اما در برخی از آپدیت ها نیز دور می‌شوند و به همین علت SGD بسیار نویزی است. نکته دیگر این است که SGD هیچ گاه به خود مینیمم نمیرسد و در اطراف ان نوسان می‌کند. نویزی بودن این الگوریتم را می‌توان با انتخاب نرخ یادگیری مناسب(کوچک) حل کرد.

Stochastic Gradient descent (SGD)

for epoch in number of epochs:

for instance in total dataset:

- for the current instance compute the derivative of the cost function
- update the weights

SGD with Momentum

به دلیل نوسانات آپدیت ها در الگوریتم SGD باید از نرخ یادگیری کوچکی استفاده کرد تا واگرایی رخ ندهد و این موضوع باعث کند شدن فرآیند بهینه‌سازی می‌شود.

Algorithm 1 SGD with Momentum

Require: α and β
On iteration t:

Compute gradients d_w and d_b for the sample
$V_{d_w} = \beta V_{d_w} + (1-\beta) d_w$
$V_{d_b} = \beta V_{d_b} + (1-\beta) d_b$

$W = W - \alpha V_{d_w}$
$b = b - \alpha V_{d_b}$

این الگوریتم moving average برای گرادیان‌هارا محاسبه می‌کند و با استفاده از آن آپدیت وزن‌هارا انجام می‌دهد. این الگوریتم گام های الگوریتم گرادیان کاهشی را ملایم تر می‌کند. این میانگین گیری باعث می‌شود که در بعد نوسانات، میانگین این نوسانات به حدودا صفر برسد و مسیر رسیدن به نقطه بهینه هموار تر شود. برای توضیح استفاده از جمله مومنتوم از analogy پرت کردن یک توپ در یک دره استفاده می‌شود که در باره آن می‌توانید از اینجا بخوانید.

Root Mean Squared prop(RMSprop)

Algorithm 2 Root Mean Squared prop(RMSprop)

Require: α and β
On iteration t:

Compute gradients d_w and d_b for the mini-batch
$S_{d_w} = \beta V_{d_w} + (1-\beta) d_w^2$
$S_{d_b} = \beta V_{d_b} + (1-\beta) d_b^2$

$W = W - \alpha \frac{d_w}{\sqrt{S_{d_w}}}$
$b = b - \alpha \frac{d_b}{\sqrt{S_{d_b}}}$

در این الگوریتم برای بعدی که در آن سرعت بهینه سازی بیشتری میخواهیم مقدار میانگین کمتر خواهد بود به این صورت که در بعدی که نوسان بیشتری دارد چون مقدار گرادیان ها و درنتیجه توان ۲ آن ها بیشتر می‌شود مقدار میانگین گرادیانی انها بیشتر شده و در نتیجه در آپدیت مقدار کمتری آپدیت می‌شوند(چون جذر میانگین در مخرج قرار گرفته است در فرمول‌های آپدیت) و سرعت آپدیت در بعد با نوسان زیاد کمتر میٰشود. ااثر این الگوریتم این است که می‌توان بدون واگرایی از نرخ یادگیری بزرگتری استفاده کرد که این به معنای افزایش سرعت یادگیری است. (منظور از ابعاد پارمترهایی هستند که در حال آپدیت آن ها هستیم)

Adaptive Moment Estimation(Adam)

Algorithm 3 Adaptive Moment Estimation(Adam)

Require: α and β_1 and β_2
On iteration t:

Compute gradients d_w and d_b for the mini-batch
$V_{d_w} = \beta_1 V_{d_w} + (1-\beta_1) d_w$ (Update biased first moment estimate)
$V_{d_b} = \beta_1 V_{d_b} + (1-\beta_1) d_b$ (Update biased first moment estimate)

$S_{d_w} = \beta_2 V_{d_w} + (1-\beta_2) d_w^2$ (Update biased second moment estimate)
$S_{d_b} = \beta_2 V_{d_b} + (1-\beta_2) d_b^2$ (Update biased second moment estimate)

$V_{d_w}^{corrected} = \frac{V_{d_w}}{1-{\beta_1}^t}$ (Bias-corrected)
$V_{d_b}^{corrected} = \frac{V_{d_b}}{1-{\beta_1}^t}$ (Bias-corrected)
$S_{d_w}^{corrected} = \frac{S_{d_w}}{1-{\beta_2}^t}$ (Bias-corrected)
$S_{d_b}^{corrected} = \frac{S_{d_b}}{1-{\beta_2}^t}$ (Bias-corrected)

$W = W - \alpha \frac{V_{d_w}^{corrected}}{\sqrt{S_{d_w}^{corrected}}+\epsilon}$(Update parameters)
$b = b - \alpha \frac{V_{d_b}^{corrected}}{\sqrt{S_{d_b}^{corrected}}+\epsilon}$ (Update parameters)

این الگوریتم ترکیب دو الگوریتم قبلی می‌باشد. مشاهده می‌شود که چهار هایپرپارامتر دارد که نرخ یادگیری باید توسط تست مقدار مناسبی برایش اختیار شود. مقدار β_1 را معمولا ۹.۰ و مقدار β_2 ۹۹۹.۰ استفاده می‌شود. مقدار ϵ نیر برای جلوگیرذی از مشکلات عددی مانند مخرج صفر است و مقدار ان اهمیت چندانی ندارد و تنها باید مقدار کوچکی باشد.

L2 Regularization

در درس با مشکل overfitting آشنا شدید و مشاهده کردید که یک روش برای جلوگیری از آن، استفاده از روش های regularization است. در این بخش، شما باید L2 regularization که یکی از روش های regularization است را پیاده سازی کنید. در تمرین شماره ۳، با روش های regularization بیشتری آشنا خواهید شد.

در L2 regularization یک ترم اضافی مربعات وزن به تابع هزینه اضافه می‌شود. این کار باعث می‌شود تا وزن‌های شبکه خیلی بزرگ نشوند و درنتیجه مدل خیلی پیچیده نمی‌شود. این کار باعث جلوگیری از overfitting می‌شود.

فرمول تابع هزینه بعد از اضافه کردن ترم L2 regularization به صورت زیر است:

$$loss(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \gamma \sum_{j=1}^{n}\theta_j^2\right]$$

که در آن h_θ خروجی شبکه، و θ وزن های شبکه می‌باشد.

آزمایشات

شما باید ابتدا تمامی توابع فعالیت و بهینه ساز های معرفی شده به همراه L2 Regularization را پیاده سازی کنید. در این تمرین برای انجام آزمایشات باید از شبکه که در تمرین ۱ پیاده سازی کردید و دیتاست Fashion MNIST استفاده کنید. و فایل‌های کد و گزارش خود را در سامانه vu اپلود کنید.

- توابع فعالیت داکیومنت را بر روی شبکه MLP با تعداد لایه ها و نورون‌های متفاوت تست کنید. (دقت کنید که حتما توابع فعالیت مختلف را بر روی شبکه‌ای با ۴ یا ۵ لایه تست کنید تا تاثیر متفاوت آن ها بر Vanishing Gradient رامشاهده کنید.) برای هر تایع فعالیت بر روی مدل های متفاوتی که تست می‌کنید نمودار تابع هزینه و اکیورسی بر حسب ایپاک ر روی داده آموزش و تست را رسم کنید و در گزارشات خود درمورد سرعت یادگیری و مقدار بهینه و نتیجه که مدل های متفاوت با هم داشته اند بررسی انجام دهید.

- وزن‌دهی اولیه تمام صفر، رندوم اعداد کوچک، رندوم اعداد بزرگ، Xavier و He را با بکدیگر مقایسه کنید. بر روی یک مدل ۳ لایه تست کنید و نتایج هر روش را با رسم توابع هزینه و اکیورسی روی داده های تست و آموزش مقایسه کنید.

- شبکه ۴ لایه‌ای را با بهینه سازهای متفاوت آموزش دهید و learning curve را برای هر کدام رسم کنید و نتایج را بررسی کنید.

- این بار به این شبکه ۴ لایه L2 Regularization نیز اضافه کنید و دوباره این شبکه را به یک بهینه ساز دلخواه آموزش دهید و تابع هزینه بر روی داده آموزش و تست بر حسب ایپاک را با هم پلات کنید.

  و با حالت قبلی (شبکه ۴ لایه با همان بهینه ساز اما بدون L2 Regularization) مقایسه کنید.