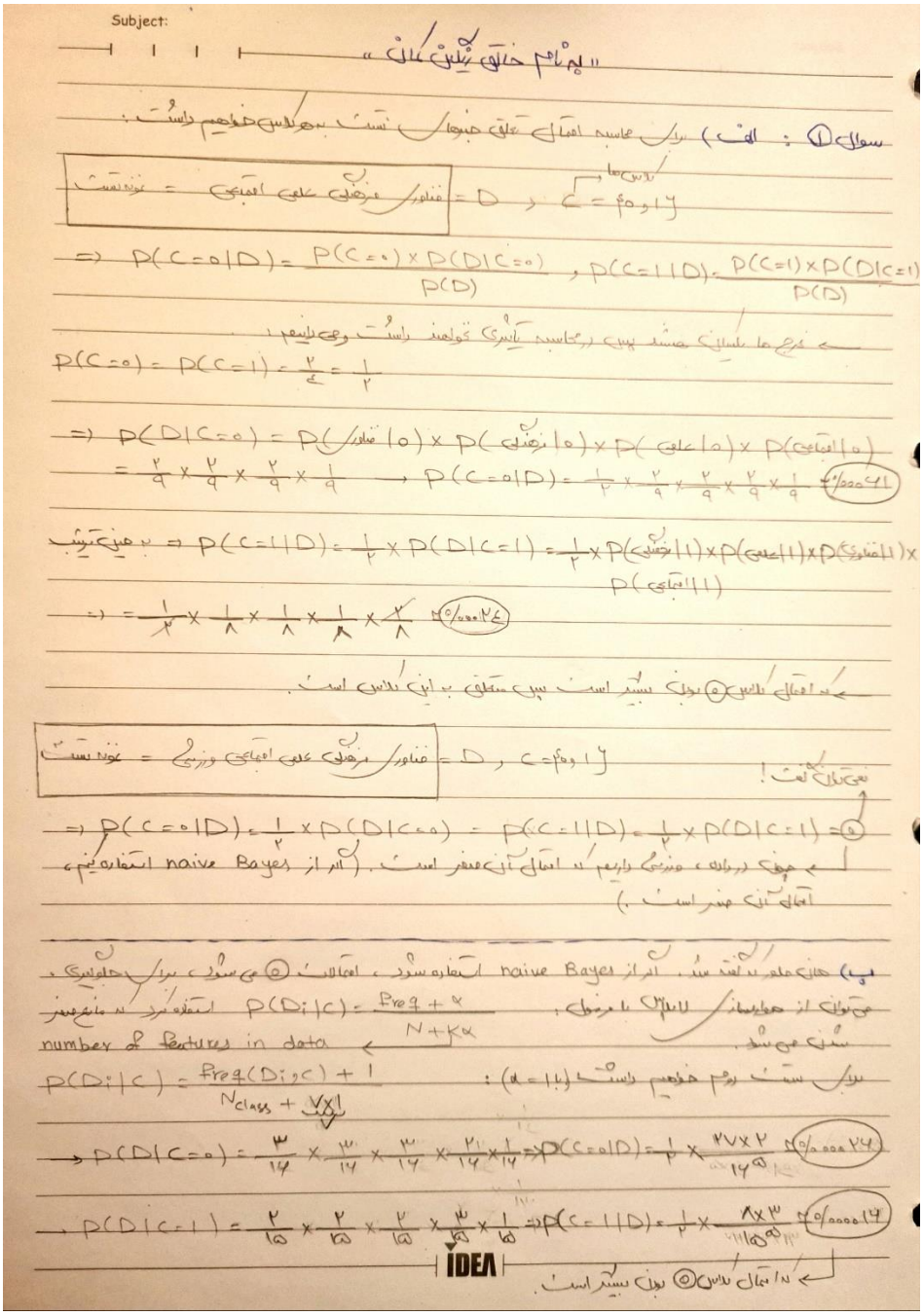
به نام خالق رنگین کمان

ستاره باباجانی 99521109- تمرین سری 1



مرجع:

https://towardsdatascience.com/laplace-smoothing-in-na%C3%AFve-bayes-algorithm-9c237a8bdece

سوال 2: -

سوال ③ : داریم :

$\Phi(a) = \int_{-\infty}^{a} N(\theta \mid 0,1)\, d\theta$ ، $N(\theta, 1)$ توزیع نرمال استاندارد

برای محاسبه احتمال خروجی y (که یک عدد باینری است) به شرط ورودی $\vec{x}$ و وزن‌های $\vec{w}$ و بایاس b ، خواهیم داشت :

$$P(y \mid \vec{x}, \vec{w}, b) = \begin{cases} \Phi(\vec{x}\cdot\vec{w} + b) & \text{if } y=1 \\ 1 - \Phi(\vec{x}\cdot\vec{w}+b) & \text{if } y=0 \end{cases}$$

که دراصطلاح این رابطه برای یک نقطه $(\vec{x}_i, y_i)$ به اصطلاح تخمینی است : likelihood

$$P(y_i \mid \vec{x}_i, \vec{w}, b) = \begin{cases} \Phi(\vec{x}_i\cdot\vec{w} + b) & \text{if } y_i=1 \\ 1 - \Phi(\vec{x}_i\cdot\vec{w} + b) & \text{if } y_i=0 \end{cases}$$

اگر این اسلایدها ، خروجی باینری نیم ، خواهیم داشت :

$$P(y_i \mid \vec{x}_i, \vec{w}, b) = y_i \cdot \Phi(\vec{x}_i\cdot\vec{w}+b) + (1-y_i)\cdot(1-\Phi(\vec{x}_i\cdot\vec{w}+b))$$

← برای محاسبه تابع هزینه هنر شرطی ، تابع منفی لگاریتم می‌گیریم . یعنی :

$$\Rightarrow L_i(\vec{w}, b) = -\log\left(P(y_i \mid \vec{x}_i, \vec{w}, b)\right)$$

یعنی $L_i(\vec{w}, b) = \begin{cases} -\log\Phi(\vec{x}_i\cdot\vec{w}+b) & \text{if } y_i=1 \\ -\log(1-\Phi(\vec{x}_i\cdot\vec{w}+b)) & \text{if } y_i=0 \end{cases}$

برابر است با $L_i(\vec{w}, b) = -y_i \cdot \log\Phi(\vec{x}_i\cdot\vec{w}+b) - (1-y_i)\log(1-\Phi(\vec{x}_i\cdot\vec{w}+b))$

← حال برای محاسبه خروجی ، میانگین ضررها را حساب می‌کنیم :

$$\Rightarrow L(\vec{w}, b) = \frac{1}{n}\sum_{i=1}^{n} L_i(\vec{w}, b) =$$

چون y_i داشتیم ، و n تعداد نمونه‌ها است .
ضرب به طرح تبدیل شده است (این اسلایدها)

یعنی $\boxed{L(\vec{w}, b) = \frac{-1}{n}\sum_{i=1}^{n} y_i \cdot \log(\Phi(\vec{x}_i\cdot\vec{w}+b)) + (1-y_i)\cdot\log(1-\Phi(\vec{x}_i\cdot\vec{w}+b))}$

IDEA

سوال4: الف) توابع فعال‌سازی یک جزء ضروری در پرسپترون‌های چندلایه (MLP) و دیگر معماری‌های شبکه عصبی هستند. آنها چندین هدف مهم را انجام می دهند، از جمله:

- معرفی غیر خطی بودن: توابع فعال سازی غیرخطی بودن را به مدل وارد می کند، که به شبکه اجازه می دهد تا روابط پیچیده و غیر خطی را در داده ها تقریب زده و یاد بگیرد. بدون توابع فعال‌سازی غیرخطی، کل شبکه عصبی مانند یک تبدیل خطی عمل می‌کند و باعث می‌شود که قادر به یادگیری و نمایش الگوهای پیچیده نباشد.

- توابع پیچیده یادگیری: توابع فعال سازی غیرخطی شبکه عصبی را قادر می سازد تا طیف وسیعی از توابع را تقریب و یاد بگیرد و آنها را برای کارهای مختلف از جمله تشخیص تصویر، پردازش زبان طبیعی و غیره مناسب می کند. ظرفیت مدل‌سازی توابع پیچیده برای حل مسائل دنیای واقعی بسیار مهم است.

- لایه های پشته ای: MLP ها از چندین لایه نورون تشکیل شده اند. بدون توابع فعال‌سازی، این لایه‌ها به طور موثر در یک تبدیل خطی منفرد فرو می‌روند و شبکه را قدرتمندتر از یک پرسپترون تک لایه نمی‌سازند. توابع فعال سازی به شما امکان می دهد چندین لایه را روی

هم قرار دهید و شبکه های عصبی عمیقی ایجاد کنید که قادر به یادگیری ویژگی های سلسله مراتبی هستند.

- نمایش پراکنده: برخی از توابع فعال سازی، مانند واحدهای خطی اصلاح شده (ReLU)، می توانند نمایش های پراکنده را در شبکه تشویق کنند. فعال‌سازی‌های پراکنده از نظر محاسباتی کارآمد هستند و می‌توانند به تمرکز شبکه بر روی مرتبط‌ترین ویژگی‌ها در داده‌ها کمک کنند.

- اجتناب از مشکل گرادیان ناپدید شدن: توابع فعال‌سازی می‌توانند به کاهش مشکل گرادیان ناپدید شدن که در طول انتشار پس‌پخش در شبکه‌های عمیق رخ می‌دهد، کمک کنند. برخی از توابع فعال‌سازی، مانند ReLU، برای ورودی‌های مثبت اشباع نمی‌شوند و به شیب‌ها اجازه می‌دهند راحت‌تر جریان پیدا کنند و از ناپدید شدن آنها در طول تمرین جلوگیری می‌کنند.

- مدیریت ورودی های منفی: توابع فعال سازی، مانند توابع Sigmoid و Tanh، می توانند ورودی ها را به محدوده خروجی مورد نظر نگاشت (به عنوان مثال، [0، 1] برای Sigmoid و [-1، 1] برای Tanh)، که می

تواند برای Tanh مفید باشد. کاربردهای خاص مانند طبقه بندی باینری یا رگرسیون محدود.

- منظم‌سازی: برخی از توابع فعال‌سازی، مانند تابع Dropout، می‌توانند با غیرفعال کردن تصادفی بخشی از نورون‌ها در طول تمرین، به‌عنوان نوعی منظم‌سازی عمل کنند. این به جلوگیری از برازش بیش از حد کمک می کند و تعمیم مدل را بهبود می بخشد.

علاوه بر این، همان طور که در اسلاید درس آمده، توابع فعال سازی برای هر نورون مشخص میکنند که خروجی بخش خطی در چه شرایطی منجر به فعال شدن نورون باید بشود.

ب) در تئوری، می توان از هر تابع فعال سازی غیر خطی برای پرسپترون چند لایه (MLP) استفاده کرد. انتخاب تابع فعال سازی باید به نیازهای خاص شبکه عصبی و ماهیت مشکل بستگی داشته باشد. در اینجا برخی از ملاحظات برای انتخاب یک تابع فعال سازی وجود دارد:

- ReLU (واحد خطی اصلاح شده): ReLU به دلیل سادگی و کارایی در بسیاری از موارد یکی از پرکاربردترین توابع فعال سازی است. به خصوص

برای شبکه های عمیق مناسب است. با این حال، ممکن است از مشکل "ReLU در حال مرگ" رنج ببرد که در آن برخی از نورون ها در طول تمرین غیرفعال می شوند.

- Sigmoid و Tanh: توابع Sigmoid و Tanh برای کارهایی مانند طبقه بندی باینری مفید هستند، جایی که می خواهید خروجی را در محدوده خاصی محدود کنید (0 تا 1 برای Sigmoid و -1 تا 1 برای Tanh). آنها همچنین در شبکه های عصبی بازگشتی خاص استفاده می شوند.

- Leaky ReLU: Leaky ReLU مشکل "ReLU در حال مرگ" را با اجازه دادن یک گرادیان کوچک برای ورودی های منفی برطرف می کند. این می تواند ثبات تمرین را بهبود بخشد.

- PReLU: PReLU (ReLU پارامتریک) گونه‌ای از Leaky ReLU است که در آن شیب قسمت منفی را می‌توان در طول آموزش یاد گرفت و انعطاف‌پذیری بیشتری را فراهم می‌کند.

- واحد خطی نمایی ELU (ELU): گزینه دیگری است که می تواند مشکل در حال مرگ ReLU را کاهش دهد و نشان داده شده است که در برخی از سناریوها عملکرد خوبی دارد.

- واحد خطی نمایی مقیاس‌دار SELU (SELU): یک تابع فعال‌سازی خودکار است که می‌تواند به تثبیت فعال‌سازی‌ها و گرادیان‌ها در شبکه‌های عمیق تحت شرایط خاص کمک کند.

اما اگر بخواهیم به صورت کلی تر راجع به پارامتر هایی که هنگام توابع غیرخطی باید درنظر داشت صحبت کنیم، خواهیم داشت:

- Task Requirements: The nature of your problem and its requirements play a significant role in the choice of activation function. For example, for binary classification tasks, the Sigmoid function is commonly used, while for most other tasks, ReLU variants are popular.

- Gradient Behavior:Consider the behavior of the activation function's gradient. Some functions, like ReLU, can suffer from the "dying ReLU" problem,

where gradients can become zero for a large portion of the input space. This can affect training stability and slow convergence.

- Vanishing and Exploding Gradients:Certain activation functions, like Sigmoid and Tanh, are more prone to vanishing gradients, which can hinder training in deep networks. On the other hand, some functions can lead to exploding gradients, making training difficult. Choose an activation function that mitigates these issues in your specific architecture.

- Computation Efficiency: Some activation functions are computationally more efficient than others. Simple functions like ReLU and its variants are faster to compute compared to more complex functions like Sigmoid or Tanh.

- Training Stability:The choice of activation function can affect the stability and speed of training. Functions like Leaky ReLU or ELU may provide better

training stability compared to the standard ReLU, especially in deeper networks.

- Bias Handling:Consider how the activation function handles bias. Some functions have a natural bias term (e.g., Sigmoid and Tanh), while others do not (e.g., ReLU). This may affect the initial training dynamics and convergence of your model.

- Output Range: If your problem requires specific output ranges (e.g., [0, 1] for probabilities in binary classification), choose an activation function that naturally produces outputs in that range.

- Expressiveness: Different activation functions have different levels of expressiveness. Some functions, like Swish, have been shown to provide better modeling capabilities for certain tasks. Experiment with different functions to determine which one captures the data's patterns effectively.

- Architectural Compatibility: The choice of activation function should be compatible with the overall architecture of your neural network. For example, if you're using a gated architecture like LSTMs or GRUs in a recurrent neural network, the activation functions in those units are specialized for sequential data.

- Hyperparameter Tuning: Be prepared to conduct hyperparameter tuning to optimize the choice of activation function along with other hyperparameters. The ideal activation function may vary depending on the specific model architecture and dataset.

- Empirical Testing: Ultimately, it's often best to empirically test different activation functions on your specific problem and data. This involves training models with different activation functions and comparing their performance, training speed, and stability.

- Regularization: Some activation functions, like dropout, can also act as a form of regularization. If you need regularization, consider how the activation function may interact with other regularization techniques in your model.

منبع:

https://chat.openai.com/

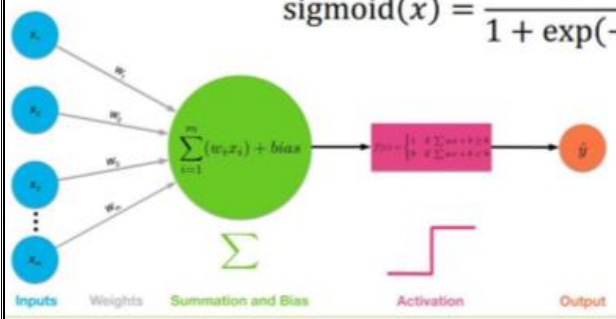
سوال5: الف)

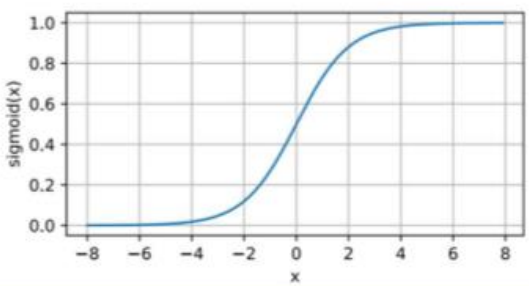
- تابع sigmoid:

Sigmoid

- در اولین شبکه‌های عصبی، دانشمندان علاقه‌مند به مدل‌سازی نورون‌های بیولوژیکی بودند که یا فعال می‌شوند یا نمی‌شوند
- با توسعه روش‌های یادگیری مبتنی بر گرادیان، نیاز بود تا تقریب مشتق‌پذیر استفاده شود

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

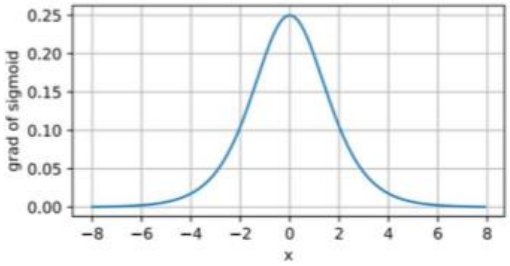




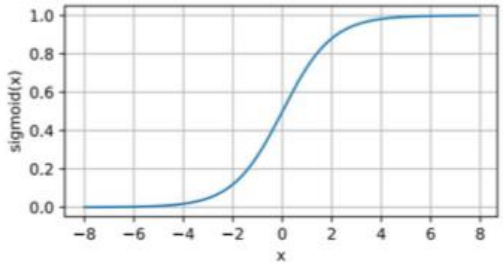
Sigmoid

- در بسیاری از مقادیر، گرادیان نزدیک به صفر است که بهینه‌سازی شبکه‌های عمیق را پیچیده می‌کند
- یکی از ایرادات sigmoid این است که خروجی آن همواره مثبت است

$$\frac{d}{dx}\text{sigmoid}(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$



$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$



Sigmoidیک مقدار واقعی را به عنوان ورودی می گیرد و مقدار دیگری را بین 0 و 1 خروجی می دهد. تابع فعال سازیsigmoid ، ورودی محدوده (−∞،∞) را به محدوده (0،1) ترجمه می کند.

مزایا:

1. خروجی در محدوده [0، 1] که برای طبقه بندی باینری مناسب است، جایی که می توان آن را به عنوان یک احتمال تفسیر کرد.
2. صاف و قابل تمایز است که باعث می شود در طول انتشار پس از انتشار به خوبی رفتار کند.
3. به طور تاریخی در شبکه های عصبی و رگرسیون لجستیک استفاده می شود.

معایب:

1. از مشکل شیب ناپدید شدن رنج می برد، به خصوص برای ورودی های بسیار مثبت یا بسیار منفی، که می تواند سرعت تمرین را کاهش دهد.
2. خروجی ها صفر محور نیستند، که می تواند بهینه سازی را در شبکه های عمیق چالش برانگیز کند.
3. محدود به وظایف طبقه بندی باینری.

- تابع softmax:

فعال‌سازی Softmax استفاده کنیم که تعمیم تابع Sigmoid است

$$\mathbf{o} = \mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{b}$$

$$\hat{y}_i = \mathrm{softmax}(\mathbf{o})_i = \frac{\exp(o_i)}{\sum_{k=1}^{q} \exp(o_k)}$$

- تابع ضرر متناسب با این تابع فعال‌سازی، categorical cross-entropy است

$$l(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{q} y_i \log \hat{y}_i$$

مشتق softmax

$$l(\mathbf{y},\hat{\mathbf{y}}) = -\sum_{i=1}^{q} y_i \log \frac{\exp(o_i)}{\sum_{k=1}^{q}\exp(o_k)} = \sum_{i=1}^{q} y_i \log \sum_{k=1}^{q}\exp(o_k) - \sum_{i=1}^{q} y_i o_i$$

$$= \log \sum_{k=1}^{q}\exp(o_k) \sum_{i=1}^{q} y_i - \sum_{i=1}^{q} y_i o_i = \log \sum_{k=1}^{q}\exp(o_k) - \sum_{i=1}^{q} y_i o_i$$

$$\partial_{o_i} l(\mathbf{y},\hat{\mathbf{y}}) = \frac{\exp(o_i)}{\sum_{k=1}^{q}\exp(o_k)} - y_i = \mathrm{softmax}(\mathbf{o})_i - y_i$$

تابع Softmax توزیع احتمالات رویداد را بر روی 'n' رویدادهای مختلف محاسبه می کند. به طور کلی، این تابع احتمالات هر کلاس هدف را بر روی تمام کلاس های هدف ممکن محاسبه می کند. بعداً احتمالات محاسبه شده به تعیین کلاس هدف برای ورودی های داده شده کمک می کند.

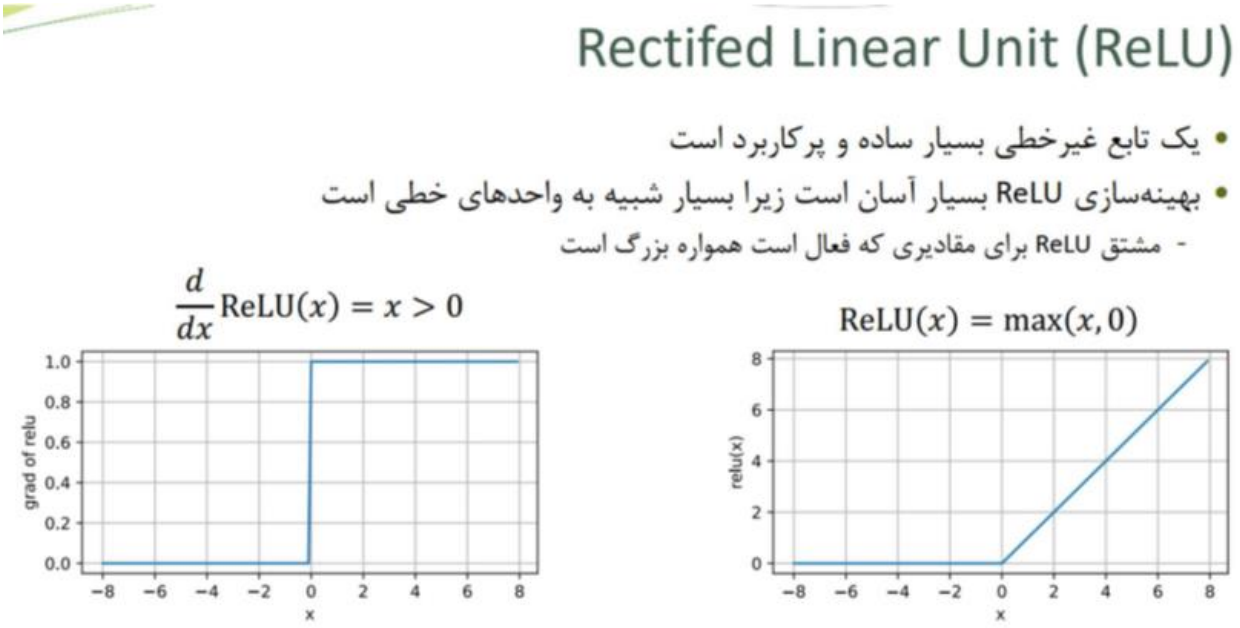
مزایا:

1. بردار اعداد واقعی را به توزیع احتمال در چندین کلاس تبدیل می کند و آن را برای کارهای طبقه بندی چند کلاسه مناسب می کند.
2. مجموع احتمالات کلاس برابر با 1 است.
3. در لایه خروجی شبکه های عصبی برای طبقه بندی چند کلاسه استفاده می شود.

معایب:

1. محاسبه نمایی در تابع softmax می‌تواند منجر به بی‌ثباتی عددی در هنگام برخورد با مقادیر ورودی بسیار بزرگ یا کوچک شود، که به طور بالقوه باعث مشکلات سرریز یا زیر جریان می‌شود.
2. خروجی softmax بسیار به مقادیر ورودی وابسته است و آن را به مقادیر پرت حساس می کند.

- تابع RelU:



فرمول ساده است: max(0,z). علیرغم نامش، Rectified Linear Units، خطی نیست و همان مزایای Sigmoid را دارد اما عملکرد بهتری دارد.

مزایا:

1. از نظر محاسباتی کارآمد است، زیرا شامل آستانه گذاری ساده است.
2. از مشکل ناپدید شدن گرادیان برای ورودی های مثبت رنج نمی برد، که می تواند آموزش در شبکه های عمیق را سرعت بخشد.
3. پراکندگی در فعال‌سازی را ترویج می‌کند و باعث می‌شود شبکه بر روی ویژگی‌های مرتبط تمرکز کند.
4. به طور گسترده مورد استفاده قرار می گیرد و عملکرد بسیار خوبی در کارهای مختلف به ویژه یادگیری عمیق نشان داده است.

معایب:

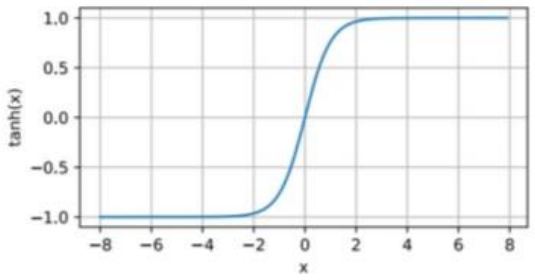
1. ممکن است از مشکل "ReLU در حال مرگ" رنج ببرد، جایی که برخی از نورون ها می توانند غیرفعال شوند (همیشه خروجی صفر دارند) و هرگز در طول تمرین برای ورودی های منفی بهبود نمی یابند.
2. خروجی ها محدود نیستند، که می تواند در موارد خاصی که خروجی های محدود مورد نیاز است، مشکلاتی ایجاد کند.
3. صفر محور نیست، که می تواند منجر به مشکلات همگرایی در برخی از معماری های شبکه شود.

- تابع Tanh:

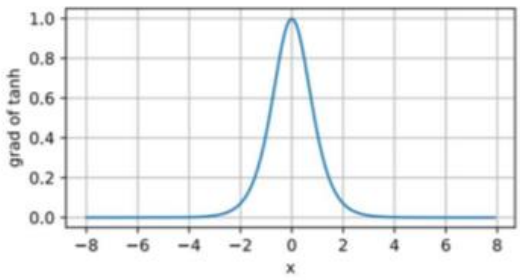
Tanh

- مشابه با تابع Sigmoid خروجی آن محدود است اما به بازه ۱- تا ۱+
- اشباع شدن تابع tanh، بهینه‌سازی را دشوار می‌کند و از لحاظ محاسباتی هم به دلیل exp پرهزینه است

$$\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)} = 2\sigma(2x) - 1$$



$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$



تابع tanh یکی دیگر از تابع های ممکن است که می تواند به عنوان یک تابع فعال سازی غیر خطی بین لایه های یک شبکه عصبی استفاده شود. چند چیز مشترک با عملکرد فعال سازی سیگموئید دارد. برخلاف تابع سیگموید که مقادیر ورودی را بین 0 و 1 ترسیم می کند، Tanh مقادیر بین 1- و 1 را ترسیم می کند. مشابه تابع سیگموید، یکی از ویژگی های جالب تابع tanh این است که مشتق tanh را می توان در بیان کرد.

مزایا:

1. خروجی در محدوده [1و1-]، که به کاهش مشکل گرادیان ناپدید شدن در مقایسه با Sigmoid کمک می کند.
2. صفر محور، آن را برای بهینه سازی و همگرایی در شبکه های عمیق مناسب تر می کند.

3. صاف و قابل تمایز، تسهیل انتشار پس زمینه.

معایب:

1. هنوز مستعد مشکل گرادیان ناپدید شدن است، به ویژه برای ورودی های بسیار مثبت یا بسیار منفی.
2. عملیات نمایی آن را از نظر محاسباتی گرانتر از ReLU می کند.
3. خروجی ها ممکن است همیشه برای همه انواع وظایف مناسب نباشند، زیرا محدود هستند.

حال به مقایسه این توابع میپردازیم:

Activation Function	Output Range	Advantages	Disadvantages
Sigmoid	[0, 1]	- Suitable for binary classification. - Smooth and differentiable. - Historically used in neural networks.	- Suffers from vanishing gradient, especially for extreme inputs. - Outputs not zero-centered. - Limited to binary classification.
Softmax	[0, 1] (sums to 1)	- Converts input into a probability distribution for multi-class classification. - Enforces sum of probabilities to be 1.	- Exponential computation can lead to numerical instability. - Sensitive to outliers in input values.
ReLU (Rectified Linear Unit)	[0, ∞)	- Computationally efficient and fast due to simple thresholding. - Mitigates vanishing gradient problem for positive inputs. - Promotes sparse activations. - Widely used with good performance in deep learning.	- "Dying ReLU" problem for negative inputs (some neurons become inactive). - Outputs are unbounded. - Not zero-centered.
Tanh (Hyperbolic Tangent)	[-1, 1]	- Outputs in [-1, 1], reducing vanishing gradient issues compared to Sigmoid. - Zero-centered, aiding convergence. - Smooth and differentiable.	- Still susceptible to vanishing gradient, especially for extreme inputs. - More computationally expensive compared to ReLU. - Outputs may not always be suitable for all tasks.

منابع:

https://chat.openai.com/

https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-/guide

ب) حال در این بخش میخواهیم این توابع را پیاده سازی کنیم. در 3 سلول اول، کتابخانه ها مورد نیاز ایمپورت شد و دیتا ساخته شد و یک تابع برای نمایش توابع طراحی شد:

```
[1]  1 # import libraries
     2 import numpy as np
     3 import torch
     4 import torch.nn as nn
     5 import matplotlib.pyplot as plt
```

```
[2]  1 # create some dummy data
     2 test_data_np = np.linspace(-10.0, 10.0, num=500)
     3 test_data_torch = torch.tensor(test_data_np, dtype=torch.float64)
```

```
[3]  1 # define a function for plotting
     2 def plot_function(x, y):
     3     plt.plot(x, y)
```

این تابع نیز برای تست توابع نوشته شده استفاده میشود:

```
     1 # define a function to test the two activation functions
     2 def test(x, function1, function2):
     3     output_fn1 = function1(x)
     4     output_fn2 = function2(x)
     5     return torch.allclose(output_fn1,output_fn2), output_fn1
```

حال به بررسی کد های جدید زده شده میپردازیم:

1. تابع sigmoid: طبق فرمول داده شده و با استفاده از تابع exp داریم:

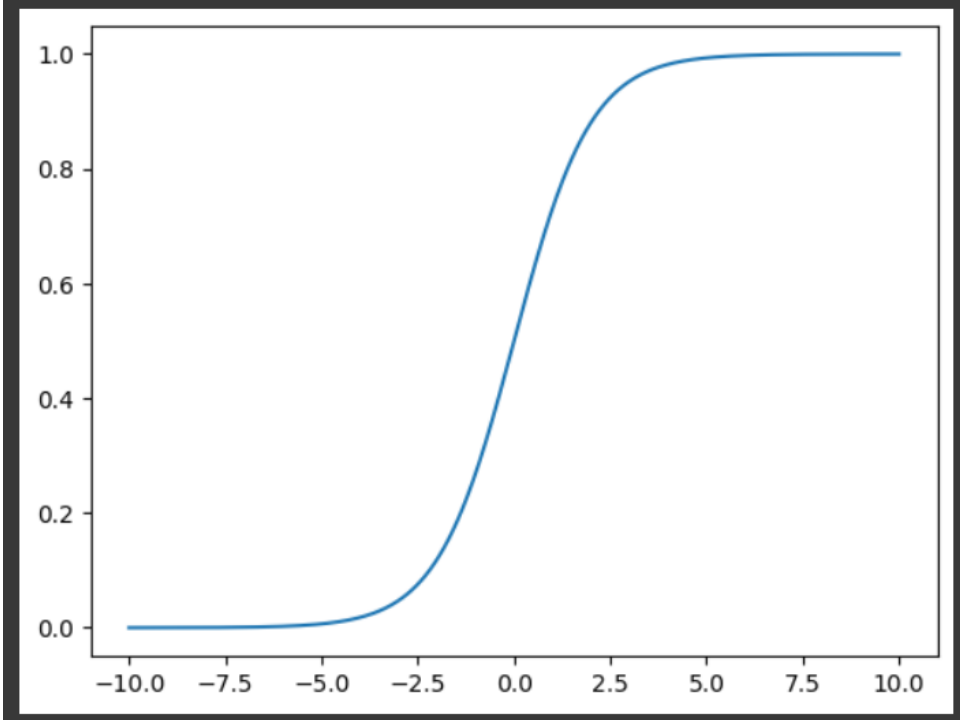
```
[6]  1 # implement the sigmoid activation function
     2 ##### TO DO #####
     3 def my_sigmoid(x):
     4     return 1 / (1 + torch.exp(-x))
```

حال خروجی تست این تابع و سپس شکل آن را میبینیم:

```
1 # test your function against the sigmoid activation function from torch
2 result_sigmoid, output_sigmoid = test(test_data_torch, my_sigmoid, nn.Sigmoid())
3 print(result_sigmoid)
```

```
True
```

```
1 # plot the sigmoid activation function
2 plot_function(test_data_np, output_sigmoid)
```



2. تابع softmax: طبق فرمول و با استفاده از توابع آماده exp, sum داریم: (تابع sum جمع را حساب میکند)

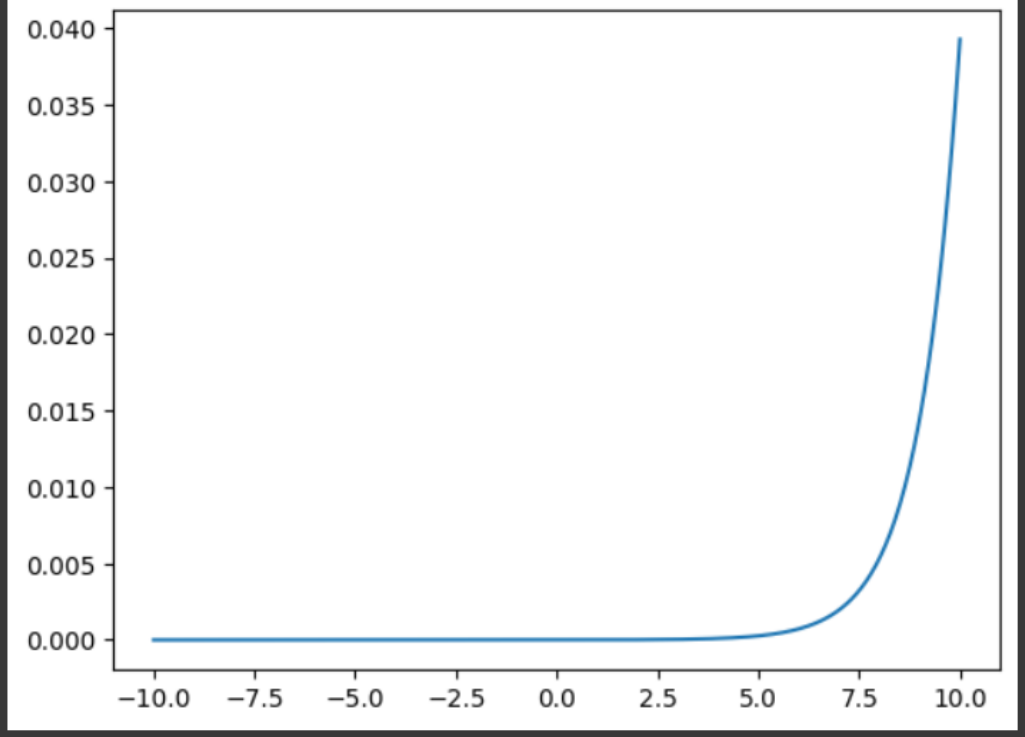
```python
# implement the softmax activation function
##### TO DO #####
def my_softmax(x):
  return torch.exp(x) / torch.sum(torch.exp(x))
```

حال خروجی تست این تابع و سپس شکل آن را میبینیم:

```python
# test your function against the softmax activation function from torch
result_softmax, output_softmax = test(test_data_torch, my_softmax, nn.Softmax())
print(result_softmax)
```

```
True
```

```
# plot the softmax activation function
plot_function(test_data_np, output_softmax)
```



3. تابع RelU: با استفاده از فرمول و تابع آماده max(برای ماکسیمم گرفتن) داریم: (تابع tensor مقدار تنسور 0 است)

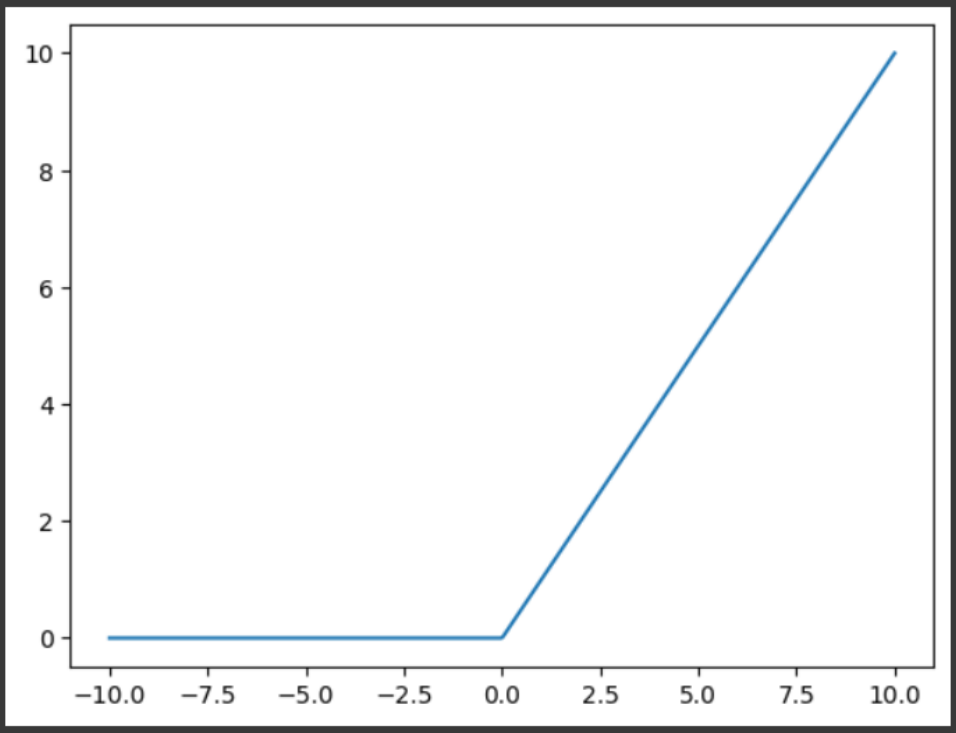
```
# implement the ReLU activation function
##### TO DO #####
def my_relu(x):
    return torch.max(torch.tensor(0), x)
```

حال خروجی تست این تابع و سپس شکل آن را میبینیم:

```
1 # test your function against the ReLU activation function from torch
2 result_relu, output_relu = test(test_data_torch, my_relu, nn.ReLU())
3 print(result_relu)
```

```
True
```

```
1 # plot the ReLU activation function
2 plot_function(test_data_np, output_relu)
```



4. تابع Leaky RelU: طبق فرمول و با درنظر گرفتن مقدار 0.05 برای آلفا(این مقدار قابل تغییر است) خواهیم داشت:

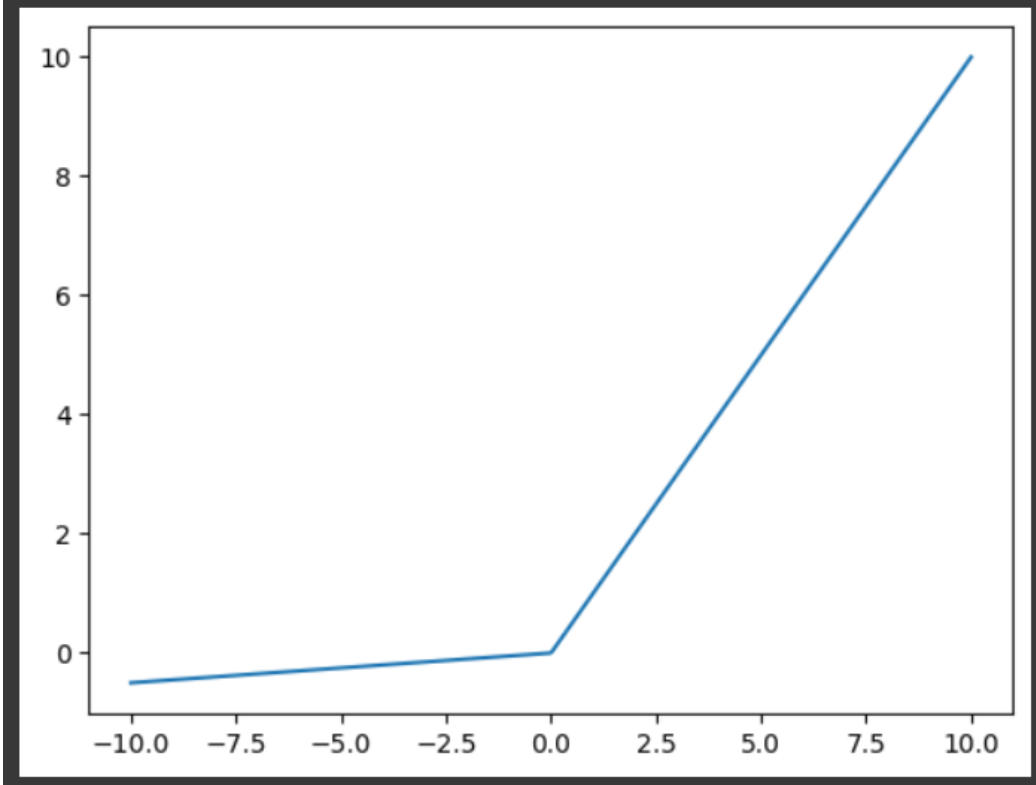
```python
# implement the Leaky ReLU activation function
##### TO DO #####
def my_leaky_relu(x, alpha=0.05):  # adjust the value of alpha
    return torch.max(alpha * x, x)
```

حال خروجی تست این تابع و سپس شکل آن را میبینیم:

```python
# test your function against the Leaky ReLU activation function from torch
result_leaky_relu, output_leaky_relu = test(test_data_torch, my_leaky_relu, nn.LeakyReLU())
print(result_leaky_relu)
```

```
False
```

```python
# plot the Leaky ReLU activation function
plot_function(test_data_np, output_leaky_relu)
```



5. تابع Tanh: طبق فرمول و استفاده از تابع exp خواهیم داشت:

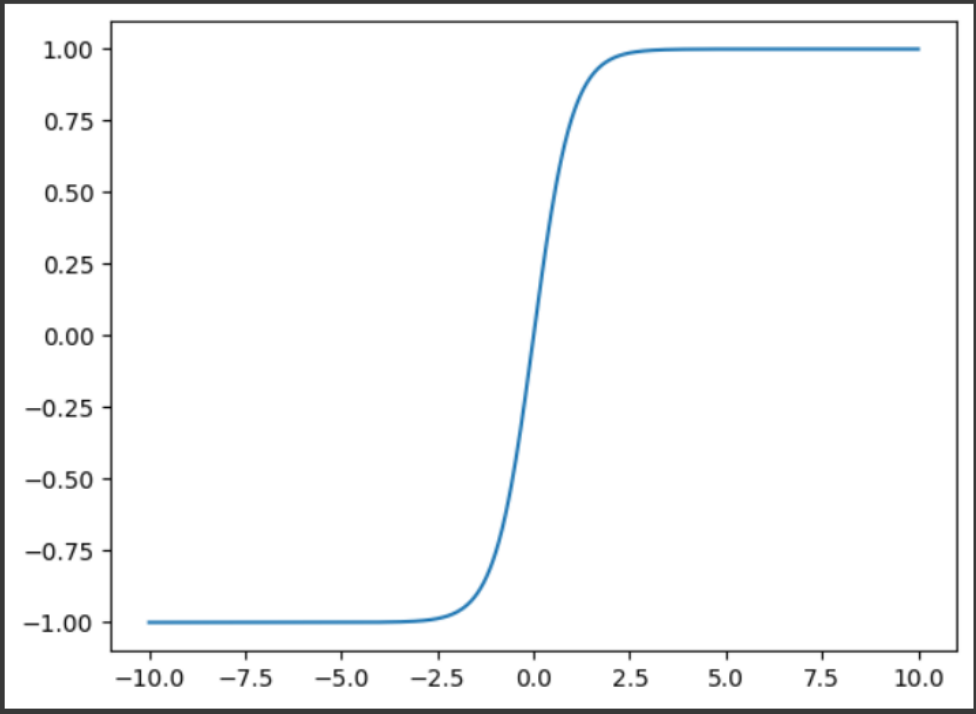
```python
# implement the tanh activation function
##### TO DO #####
def my_tanh(x):
    return (1 - torch.exp(-2 * x)) / (1 + torch.exp(-2 * x))
```

حال خروجی تست این تابع و سپس شکل آن را میبینیم:

```python
# test your function against the tanh activation function from torch
result_tanh, output_tanh = test(test_data_torch, my_tanh, nn.Tanh())
print(result_tanh)
```

```
True
```

```python
# plot the tanh activation function
plot_function(test_data_np, output_tanh)
```



همان طور که مشاهده میشود، تمامی توابع درست پیاده سازی شده اند.

ج) معماری که اینجانب معرفی میکنم به این شرح است:

1. تعداد لایه ها و دلیل:

- لایه ورودی: 64 نورون (یکی برای هر پیکسل). لایه ورودی باید به اندازه تعداد پیکسل در تصاویر نورون داشته باشد.
- لایه پنهان: 128 نورون (یا هر مقدار معقول دیگر). یک لایه پنهان برای این کار ساده کافی است. 128 نورون به عنوان یک نقطه شروع معقول انتخاب می شوند و می توان آن را بر اساس عملکرد تنظیم کرد.
- لایه خروجی: 3 نورون (یکی برای هر کلاس). از آنجایی که سه کاراکتر مختلف برای طبقه بندی داریم، لایه خروجی باید سه نورون داشته باشد، یکی برای هر کلاس.

2. تعداد نورون ها و دلیل:

تعداد نورون ها در لایه های ورودی و خروجی بر اساس ماهیت داده ها و وظیفه طبقه بندی تعیین می شود.

تعداد نورون ها در لایه پنهان تا حدودی دلخواه است و می توان آن را در طول آزمایش تنظیم کرد. 128 نورون به عنوان نقطه شروع انتخاب می شوند تا به شبکه اجازه دهند الگوهای نسبتاً پیچیده ای را بیاموزند.

3. تابع و دلیل فعال سازی:

از فعال سازی ReLU (واحد خطی اصلاح شده) برای لایه پنهان استفاده میکنیم. ReLU یک انتخاب رایج است زیرا به جریان گرادیان کمک می کند و می تواند غیرخطی ها را به خوبی مدیریت کند.

از یک تابع فعال سازی Softmax برای لایه خروجی برای تولید احتمالات برای هر کلاس استفاده میکنیم. Softmax معمولا برای مسائل طبقه بندی چند کلاسه استفاده می شود.

4. تابع ضرر و علت آن:

از Cross-Entropy Loss برای مسائل طبقه بندی با کلاس های متعدد استفاده میکنیم. از دست دادن متقابل آنتروپی یک انتخاب مناسب برای طبقه بندی چند کلاسه است و شبکه را تشویق می کند تا احتمالاتی نزدیک به 0 یا 1 برای کلاس صحیح تولید کند.

د) حال میخواهیم کد معماری تعریف شده را بزنیم.

1. ابتدا تمامی کتابخانه هایی که به آنها نیاز خواهیم داشت را ایمپورت میکنیم:

```python
import os
from PIL import Image
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

2. حال مدلی که ذکر کردیم را تعریف میکنیم: این مدل شامل یک لایه پنهان و یک لایه خروجی است که ورودی و خروجی آنها در شکل قابل رویت است. ورودی لایه میانی تعداد پیکسل ها یعنی 256 و خروجی آن که ورودی لایه بعدی است، 128 است. لایه پایانی نیز خروجی 3 به علت سه کلاسه بودن مسئله دارد.

تابع forward نیز forward passing را مشخص میکند و تابع فعال سازی هر لایه را تعریف میکند.

```python
# Define the MLP model
# The super() function should be called to initialize the parent class, and it should not be used to call the __init__ method.
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.fc1 = nn.Linear(256, 128) # Hidden Layer with 128 neurons
        self.fc2 = nn.Linear(128, 3) # Output Layer with 3 neurons (3 classes)

    def forward(self, x):
        x = torch.relu(self.fc1(x)) # ReLU activation for the hidden layer
        x = torch.softmax(self.fc2(x), dim=1) # Softmax activation for the output layer
        return x
```

3. حال باید تصویر ها را به نحوی reshape کرد که برای مدل و تنسور قابل فهم باشد. سپس دیتای آموزش و ارزیابی تعریف میشوند.(کامنت هر خط در شکل قابل رویت است):

```
# with the help of chat GPT:
def load_image(image_file):
    img = Image.open(image_file)
    # Resize the image to 8x8
    img = img.resize((8, 8))
    # Convert the image to a tensor and flatten it into a 1D array
    img_tensor = transforms.ToTensor()(img).view(256)
    return img_tensor

image_files = ["Q5_1.png", "Q5_2.png", "Q5_3.png"]

# Load and preprocess the images
image_data = [load_image(image_file) for image_file in image_files]
labels = [0, 1, 2]

train_data = image_data
train_labels = labels
test_data = image_data
test_labels = labels

```

لیبل های کلاس به ترتیب ورودی ها 1و2و3 گذاشته شد و چون دیتای کمی داریم، دیتای آموزش و ارزیابی یکی شد که همان طور که انتظار میرود accuracy برای داده test، 100 خواهد شد.

4. حال مدل را صدا زده و training را آغاز میکنیم:

```python
model = SimpleMLP()

criterion = nn.CrossEntropyLoss()  # Cross-Entropy Loss for multi-class classification
optimizer = optim.SGD(model.parameters(), lr=0.01)  # Stochastic Gradient Descent optimizer

#these numbers are custom
num_epochs = 100
batch_size = 3  # Since we have three images
train_loss_values = []

#Training
for epoch in range(num_epochs):
    epoch_loss = 0.0  # Initialize loss for this epoch
    for i in range(0, len(train_data), batch_size):
        inputs = torch.stack(train_data[i:i+batch_size])
        labels = torch.tensor(train_labels[i:i+batch_size])

        optimizer.zero_grad()  # Zero the gradients
        outputs = model(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Calculate the loss

        loss.backward()  # Backpropagation for optimizing
        optimizer.step()  # Update the model's parameters
        epoch_loss += loss.item()

    # Calculate the average loss for this epoch
    avg_epoch_loss = epoch_loss / len(train_data)
    train_loss_values.append(avg_epoch_loss)  # Append to the list
    print(f"Epoch [{epoch+1}/{num_epochs}] - Loss: {loss.item()}")

```

همان طور که مشاهده میشود ابتدا مدل صدا زده شد و سپس تابع ضرر و تابع optimizer تعریف شد. تعداد ایپوک ها بصورت دلخواه 100 گذاشته شد و سایز هر batch هم به دلیل سه عکس، 3 گذاشته شد.

در کد طبق تعداد ایپوک ها حلقه داریم و سپس هر عکس به مدل داده میشود و فرآیند forward passing انجام میشود. در نهایت برای اپدیت کردن وزن ها، از optimizer استفاده میشود.(از back propagation استفاده شده

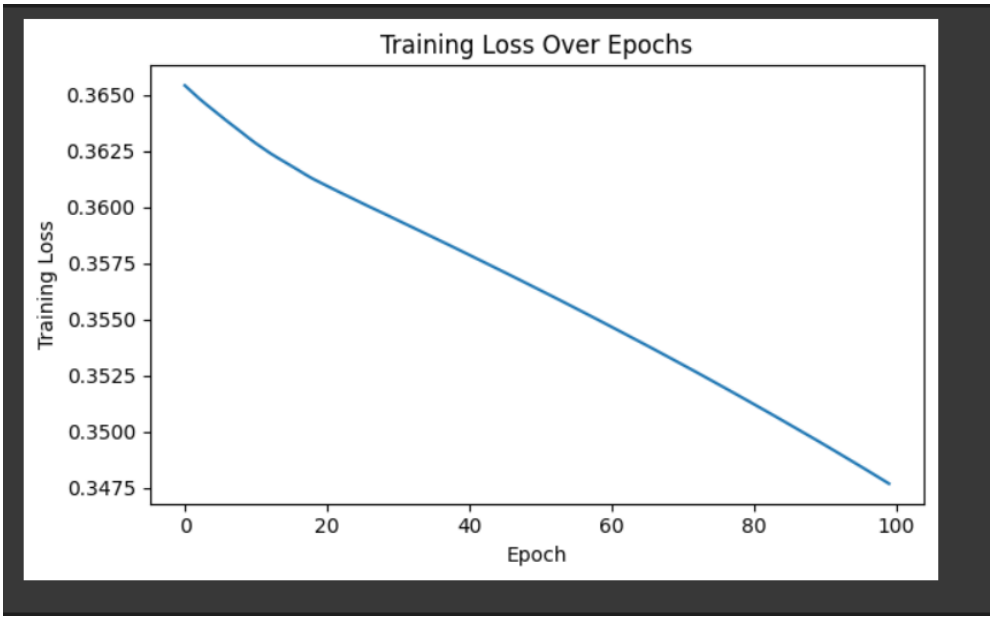
است) سپس برای اینکه خروجی بهتری داشته باشیم در هر ایپوک مقدار loss چاپ شد:

```
Epoch [1/100] - Loss: 1.10541570186615
Epoch [2/100] - Loss: 1.1038767099380493
Epoch [3/100] - Loss: 1.1025326251983643
Epoch [4/100] - Loss: 1.101553201675415
Epoch [5/100] - Loss: 1.100767970085144
Epoch [6/100] - Loss: 1.100049614906311
Epoch [7/100] - Loss: 1.0993293523788452
Epoch [8/100] - Loss: 1.0986448526382446
Epoch [9/100] - Loss: 1.0979958772659302
Epoch [10/100] - Loss: 1.0974016189575195
Epoch [11/100] - Loss: 1.0968527793884277
Epoch [12/100] - Loss: 1.0962334871292114
Epoch [13/100] - Loss: 1.095621109008789
Epoch [14/100] - Loss: 1.095030426979065
Epoch [15/100] - Loss: 1.0944331884384155
Epoch [16/100] - Loss: 1.093700647354126
Epoch [17/100] - Loss: 1.092957615852356
Epoch [18/100] - Loss: 1.0922226905822754
Epoch [19/100] - Loss: 1.0915024280548096
Epoch [20/100] - Loss: 1.090773105621338
Epoch [21/100] - Loss: 1.0900579690933228
Epoch [22/100] - Loss: 1.0893398523330688
```

برای اینکه درک بهتری از خروجی داشته باشیم، از کتابخانه matplotlib استفاده کردم. کد زده شده و خروجی آن به شرح زیر است:

```python
# Plot the training loss
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss_values)
plt.xlabel("Epoch")
plt.ylabel("Training Loss")
plt.title("Training Loss Over Epochs")

plt.tight_layout()
plt.show()
```



5. در نهایت مدل آموزش دیده را ارزیابی میکنیم. چون داده آموزش و تست یکی است، accuracy نهایی 100 میشود:

```
31
32 # Testing
33 correct = 0
34 total = 0
35 with torch.no_grad():
36     for i in range(len(test_data)):
37         inputs = test_data[i].unsqueeze(0)  # Convert a single image to a 1-sample batch
38         labels = test_labels[i]
39
40         outputs = model(inputs)
41         _, predicted = torch.max(outputs, 1)
42         total += 1
43         if predicted == labels:
44             correct += 1
45
46 print("----------------------------------------")
47 print(f"Accuracy on test data: {100 * correct / total}%")
```

```
----------------------------------------
Accuracy on test data: 100.0%
```

سوال6: ترکیب آستانه 0.5 با تابع فعال‌سازی ReLU و تابع فعال‌سازی سیگموئید نهایی در یک شبکه عصبی برای طبقه‌بندی باینری می‌تواند برخی از چالش‌ها و مشکلات بالقوه را معرفی کند:

1. غیر یکنواختی ReLU:

تابع فعال‌سازی ReLU (واحد خطی اصلاح‌شده) در 0 قابل تمایز نیست. هنگام استفاده از ReLU، ممکن است تغییرات ناگهانی در خروجی شبکه در حدود آستانه 0.5 ایجاد شود که می‌تواند فرآیند آموزش را چالش‌برانگیزتر کند. به طور خاص، مشتق ReLU در 0 تعریف نشده است، و این می تواند بر الگوریتم های بهینه سازی مبتنی بر گرادیان تأثیر بگذارد.

2. چالش های آموزشی:

ترکیب ReLU با یک تابع فعال سازی سیگموئید نهایی، عدم یکنواختی در خروجی شبکه را معرفی می کند. آموزش یک شبکه عصبی با توابع فعال سازی غیر هموار می تواند منجر به مشکلات همگرایی، تمرین کند یا حتی گیر کردن در حداقل های محلی شود. ممکن است برای مدیریت این چالش ها به تکنیک های بهینه سازی پیشرفته تری نیاز باشد.

3. ناپدید شدن و انفجار گرادیان:

استفاده از ReLU به عنوان یک تابع فعال‌سازی میانی می‌تواند منجر به مشکلات ناپدید شدن گرادیان یا انفجار در حین انتشار پس‌انداز شود. تابع فعال‌سازی سیگموئید نهایی به له کردن خروجی شبکه تا محدوده [0، 1] کمک می‌کند، اما لایه‌های میانی ممکن است همچنان با گرادیان‌ها مشکل داشته باشند که تنظیم اولیه وزن مناسب و نرمال سازی دسته ای می تواند به کاهش این مشکلات کمک کند.

4. مدیریت داده های نامتعادل:

هنگام برخورد با مجموعه داده های نامتعادل، آستانه ثابت 0.5 ممکن است منجر به پیش بینی های مغرضانه شود که به نفع طبقه اکثریت است. مدیریت موثر عدم تعادل کلاس با استفاده از یک آستانه ثابت می تواند چالش برانگیز باشد. تکنیک هایی مانند یادگیری حساس به هزینه یا استفاده از مقادیر آستانه متفاوت برای کلاس های مختلف ممکن است ضروری باشد.

5. تفسیرپذیری مدل:

یک آستانه ثابت می تواند مدل را کمتر قابل تفسیر کند، زیرا ممکن است مشخص نباشد که چرا یک آستانه خاص انتخاب شده است و چگونه بر تصمیمات مدل تأثیر می گذارد. تنظیم آستانه برای موارد استفاده مختلف یا تفسیر خروجی مدل به روشی معنادار می تواند چالش برانگیز باشد.

برای پرداختن به این چالش‌ها، ممکن است لازم باشد که معماری‌های شبکه، توابع فعال‌سازی و مقادیر آستانه‌ای مختلف را آزمایش کنیم تا بهترین پیکربندی را برای کار طبقه‌بندی خاص پیدا کنیم. علاوه بر این، برای ارزیابی عملکرد مدل باید از معیارهای ارزیابی استفاده کرد که برای انتخاب آستانه قوی‌تر هستند، مانند منحنی‌های فراخوان دقیق یا منحنی‌های ROC.

سوال7: الف) مهمترین تفاوت بین یادگیری ماشین و یادگیری عمیق در معماری و عمق شبکه های عصبی مورد استفاده در این رویکردها نهفته است. در اینجا یک تمایز اساسی وجود دارد:

1. یادگیری ماشینی (ML):

- یادگیری ماشینی حوزه وسیع تری است که تکنیک ها و الگوریتم های مختلفی برای تحلیل و مدل سازی داده ها را در بر می گیرد.

- الگوریتم‌های ML شامل رگرسیون خطی، درخت‌های تصمیم، ماشین‌های بردار پشتیبان، k-نزدیک‌ترین همسایه‌ها، جنگل‌های تصادفی و غیره هستند.

- این الگوریتم‌ها اغلب شامل مهندسی ویژگی می‌شوند، جایی که دانش خاص دامنه برای انتخاب و تبدیل ویژگی‌های ورودی مربوطه استفاده می‌شود.

- مدل های ML معمولاً کم عمق هستند، به این معنی که دارای تعداد کمی لایه یا پارامتر هستند.

- انتخاب ویژگی و مهندسی مراحل بسیار مهمی در یادگیری ماشینی سنتی هستند تا داده ها را برای مدل قابل درک کنند.

- مدل های ML به طور کلی برای طیف گسترده ای از کاربردها، از جمله طبقه بندی، رگرسیون، خوشه بندی و سیستم های توصیه مناسب هستند.

2. یادگیری عمیق (DL):

- یادگیری عمیق زیرمجموعه ای از یادگیری ماشینی است که بر روی شبکه های عصبی با لایه های متعدد (شبکه های عصبی عمیق) تمرکز می کند.

- الگوریتم‌های DL شامل شبکه‌های عصبی پیش‌خور، شبکه‌های عصبی کانولوشنال (CNN)، شبکه‌های عصبی بازگشتی (RNN) و غیره هستند.

- مدل های یادگیری عمیق به طور خودکار ویژگی های سلسله مراتبی را از داده های خام یاد می گیرند و نیاز به مهندسی ویژگی های گسترده را از بین می برند.

- مدل‌های DL با عمقشان مشخص می‌شوند، با لایه‌های مخفی متعدد که می‌توانند الگوهای پیچیده و سلسله مراتبی را در داده‌ها یاد بگیرند و نشان دهند.

- DL در کارهایی مانند تشخیص تصویر، پردازش زبان طبیعی، تشخیص گفتار و سایر زمینه هایی که داده های خام را می توان مستقیماً پردازش کرد، بسیار موفق بوده است.

- یادگیری عمیق اغلب به حجم زیادی از داده ها و منابع محاسباتی قابل توجهی مانند GPU ها برای آموزش موثر شبکه های عصبی عمیق نیاز دارد.

به طور خلاصه، تفاوت اصلی در عمق شبکه های عصبی و سطح مهندسی ویژگی مورد نیاز است. یادگیری ماشین سنتی بر مهندسی ویژگی های دستی تکیه دارد و معمولاً از مدل های کم عمق تر استفاده می کند. از سوی دیگر، یادگیری عمیق شامل شبکه‌های عصبی عمیقی است که به طور خودکار ویژگی‌ها را از داده‌های خام یاد می‌گیرند و آن را به ویژه برای کارهایی با مجموعه داده‌های بزرگ و الگوهای پیچیده، مانند تشخیص تصویر و گفتار، قدرتمند می‌کنند. با این حال، یادگیری عمیق همچنین به داده ها و منابع محاسباتی قابل توجهی نیاز دارد که می تواند در برخی از برنامه ها محدودیت باشد.

ب) در یک شبکه عصبی عمیق با 16 لایه، معمولاً لایه‌های نهایی، از جمله لایه یازدهم، برای گرفتن نتیجه نهایی راحت‌تر هستند. در اینجا دلیل آن است:

1. ناپدید شدن گرادیان: یکی از چالش های آموزش شبکه های عصبی بسیار عمیق، مشکل گرادیان ناپدید شدن است. همانطور که به عمق شبکه می روید، گرادیان ها می توانند بسیار کوچک شوند و به روز رسانی پارامترهای لایه های قبلی را دشوار می کند. این بدان معنی است که لایه های اولیه (آنهایی که نزدیک به ورودی هستند) ممکن است ویژگی های پیچیده و سطح بالا را به طور موثر یاد نگیرند.

2. یادگیری ویژگی سلسله مراتبی: شبکه های عصبی عمیق برای یادگیری ویژگی های سلسله مراتبی طراحی شده اند. در این سلسله مراتب، لایه‌های اولیه اغلب ویژگی‌های سطح پایین، مانند لبه‌ها و الگوهای ساده را به تصویر می‌کشند، در حالی که لایه‌های بعدی ویژگی‌های سطح بالاتر و انتزاعی‌تری را به تصویر می‌کشند. لایه های نهایی مسئول ترکیب این ویژگی های سطح بالا برای تصمیم گیری یا پیش بینی نهایی هستند.

3. قدرت بازنمایی: لایه های عمیق تر در یک شبکه عصبی قدرت نمایش بالاتری دارند، زیرا می توانند الگوهای پیچیده و انتزاعی فزاینده ای را به تصویر بکشند. به همین دلیل است که لایه‌های نهایی اغلب برای گرفتن نتیجه نهایی مناسب هستند، زیرا ظرفیت بیان بیشتری دارند.

4. ویژگی های Task-Specific: در بسیاری از موارد، لایه های نهایی مختص کار هستند. به عنوان مثال، در یک شبکه عصبی کانولوشن (CNN) برای طبقه‌بندی تصویر، لایه‌های نهایی از لایه‌های کاملاً متصل تشکیل شده‌اند که ویژگی‌های آموخته‌شده در لایه‌های قبلی را برای پیش‌بینی کلاس ترکیب می‌کنند.

در یک شبکه عصبی عمیق معمولی، لایه‌های عمیق‌تر، از جمله لایه یازدهم، برای گرفتن اطلاعات سطح بالا و مختص کار و در نتیجه نتیجه نهایی راحت‌تر هستند.

ج) انتخاب بین شبکه های عمیق تر و گسترده تر به مشکل خاص، داده های موجود و منابع محاسباتی بستگی دارد. در اینجا یک دستورالعمل کلی وجود دارد:

اگر داده‌های موجود محدود باشد، معمولاً گرایش به شبکه‌های گسترده‌تر کارآمدتر است. شبکه‌های گسترده‌تر می‌توانند انواع ویژگی‌های ساده را به خود اختصاص دهند و در زمانی که داده‌ها کمیاب هستند، کمتر مستعد بیش از حد برازش هستند. آنها در مقایسه با شبکه های بسیار عمیق از نظر داده کارآمدتر هستند و به پارامترهای کمتری نیاز دارند.

با این حال، اگر به مقدار قابل توجهی از داده ها دسترسی داشته باشیم، به طور کلی انتخاب شبکه های عمیق تر کارآمدتر است. شبکه‌های عمیق‌تر قادر به یادگیری ویژگی‌های سلسله مراتبی و پیچیده‌تر هستند، و آنها را برای کارهایی که شامل الگوها و روابط پیچیده هستند، مناسب می‌سازد. در حالی که آموزش آنها می تواند چالش برانگیز باشد، اما می توانند تقریب های دقیق تری را برای توابع با داده های فراوان ارائه دهند.

در عمل، ما همچنین می‌توانیم معماری‌های ترکیبی را در نظر بگیریم که عمق و عرض را ترکیب می‌کنند، مانند شبکه‌های "عمیق و گسترده" تا از مزایای هر دو رویکرد استفاده کنند.

در عمل، انتخاب بین شبکه های عمیق تر و گسترده تر به مشکل خاص و منابع موجود بستگی دارد. در اینجا چند دستورالعمل وجود دارد:

1. توابع پیچیده: اگر تابع برای تقریب بسیار پیچیده و سلسله مراتبی باشد، شبکه های عمیق تر اغلب کارآمدتر هستند. آنها می توانند به طور خودکار روابط پیچیده را یاد بگیرند و به پارامترهای کمتری نیاز دارند.

2. داده های محدود: اگر مقدار داده محدودی دارید، شبکه های گسترده تر می توانند از نظر داده کارآمدتر باشند. آنها می توانند انواع مختلفی از ویژگی های ساده را به تصویر بکشند و از نصب بیش از حد جلوگیری کنند.

3. منابع محاسباتی: منابع محاسباتی موجود را در نظر بگیرید. آموزش شبکه‌های عمیق‌تر از نظر محاسباتی پرهزینه است، در حالی که شبکه‌های گسترده‌تر می‌توانند حافظه فشرده‌ای داشته باشند.

4. منظم سازی: از تکنیک های منظم سازی مناسب مانند ترک تحصیل، کاهش وزن و توقف زودهنگام برای جلوگیری از تطبیق بیش از حد در شبکه های وسیع تر استفاده کنید.

5. آزمایش تجربی: با معماری های مختلف، از جمله تغییرات در عمق و عرض، آزمایش کنید و از تکنیک هایی مانند اعتبار سنجی متقاطع استفاده کنید تا مشخص کنید کدام معماری برای مشکل خاص شما بهتر عمل می کند.

د) مزایا:

- نمایش بهبود یافته: مدل های عمیق تر می توانند ویژگی های پیچیده و سلسله مراتبی را در داده ها ثبت کنند. آنها می توانند یاد بگیرند که الگوها و روابط پیچیده را نشان دهند، که برای کارهایی مانند تشخیص تصویر و گفتار، پردازش زبان طبیعی و موارد دیگر بسیار مهم است.

- تعمیم بهتر: شبکه های عمیق اغلب در تعمیم به داده های جدید و دیده نشده بهتر هستند. آن‌ها می‌توانند ویژگی‌های انتزاعی و قوی‌تری را استخراج کنند، که بیش از حد برازش را کاهش داده و توانایی مدل را برای پیش‌بینی دقیق در طیف وسیعی از ورودی‌ها افزایش می‌دهد.

- بیان بالاتر: معماری های عمیق درجه بالاتری از بیان را ارائه می دهند و به آنها اجازه می دهد تا دسته وسیع تری از توابع را تقریب بزنند. این

بدان معنی است که آنها می توانند با طیف وسیع تری از وظایف مقابله کنند و روابط پیچیده تری را مدل کنند.

- یادگیری ویژگی کارآمد: شبکه‌های عمیق می‌توانند به طور خودکار ویژگی‌ها را از داده‌های خام بیاموزند و نیاز به مهندسی ویژگی‌های دستی گسترده را از بین ببرند. این امر به ویژه هنگام برخورد با داده های با ابعاد بالا یا بدون ساختار مفید است.

معایب:

- پیچیدگی آموزش: آموزش مدل های عمیق پیچیده تر است. شبکه‌های عمیق‌تر ممکن است از مسائلی مانند ناپدید شدن گرادیان‌ها (شیب‌ها بسیار کوچک می‌شوند) یا شیب‌های انفجاری (شیب‌ها بسیار بزرگ می‌شوند) رنج ببرند، که می‌تواند آموزش را چالش‌برانگیز کند.

- منابع محاسباتی: آموزش شبکه های عمیق اغلب به منابع محاسباتی قابل توجهی از جمله GPU یا TPU های قدرتمند نیاز دارد. یادگیری عمیق می تواند از نظر محاسباتی گران باشد، که ممکن است برای همه پروژه ها امکان پذیر نباشد.

- الزامات داده: مدل‌های عمیق معمولاً به داده‌های بیشتری برای تعمیم مؤثر نیاز دارند. اگر داده‌های محدودی دارید، شبکه‌های عمیق‌تر ممکن است مستعد بیش از حد برازش شوند. مدل های کم عمق ممکن است در چنین مواردی عملکرد بهتری داشته باشند.

- تنظیم Hyperparameter: شبکه های عمیق تر دارای هایپرپارامترهای بیشتری برای تنظیم هستند، مانند نرخ یادگیری، اندازه دسته و روش های اولیه. یافتن هایپرپارامترهای مناسب می تواند وقت گیرتر باشد.

- افزایش خطر بیش‌برازش: در حالی که شبکه‌های عمیق می‌توانند تعمیم‌سازی را بهبود بخشند، مدل‌های بسیار عمیق نیز در صورت عدم تنظیم مناسب در معرض خطر بیش از حد برازش هستند. تکنیک هایی مانند ترک تحصیل، عادی سازی دسته ای و توقف زودهنگام اغلب مورد نیاز است.

به طور خلاصه، افزودن لایه‌های بیشتر به یک شبکه عصبی می‌تواند به بهبود نمایش و تعمیم ویژگی منجر شود و یادگیری عمیق را برای کارهای پیچیده مناسب کند. با این حال، چالش‌های مربوط به پیچیدگی آموزش، منابع

محاسباتی، نیازهای داده، و نیاز به تنظیم و منظم‌سازی موثر فراپارامتر را نیز معرفی می‌کند.

پایان