Deep Learning HomeWork 02

Mahsa Naseri 402209015

بخش تئوری

سوال ۱:

$\boxed{1}$ $E_{w,b} = -\sum_n y_n \ln \hat{y}(x_n) + (1-y_n)\ln(1-\hat{y}(x_n))$

sigmoid $\longrightarrow$ Activate Function $\quad f(x) = \sigma(x) = \dfrac{1}{1+e^{-x}}$

$$f'(x) = f(x)(1-f(x))$$

$$\hat{y}(x_n) = \sigma(w \cdot x_n + b) = \frac{1}{1+e^{-(wx_n+b)}}$$

$$\star \; \frac{\partial E_{w,b}}{\partial w_j} = -\sum_n \Big(\underbrace{\frac{\partial}{\partial w_j}(y_n \ln(\hat{y}(x_n)))}_{1} + \underbrace{\frac{\partial}{\partial w_j}((1-y_n)\ln(1-\hat{y}(x_n)))}_{2} \Big)$$

$$1 \quad \frac{\partial}{\partial w_j}(y_n \ln \hat{y}(x_n)) = y_n \frac{1}{\hat{y}(x_n)} \cdot \frac{\partial \hat{y}(x_n)}{\partial w_j} = \frac{y_n}{\cancel{\hat{y}(x_n)}}(\cancel{\hat{y}(x_n)}(1-\hat{y}(x_n)))\, x_{n,j}$$

$$= y_n(1-\hat{y}(x_n))x_{n,j}$$

$$2 \quad \frac{\partial}{\partial w_j}((1-y_n)\ln(1-\hat{y}(x_n))) = (1-y_n)\left(\frac{1}{1-\hat{y}(x_n)}\right) \cdot \frac{\partial(1-\hat{y}(x_n))}{\partial w_j}$$

$$= \frac{1-y_n}{\cancel{1-\hat{y}(x_n)}} \cdot \frac{-(\hat{y}(x_n)\cancel{(1-\hat{y}(x_n))}x_{n,j})}{1} = \hat{y}(x_n)(1-y_n)(-x_{n,j})$$

$$\frac{\partial E_{w,b}}{\partial w_j} = -\sum_n \left(y_n(1-\hat{y}(x_n))x_{n,j}\right)\left(\hat{y}(x_n)(1-y_n)(-x_{n,j})\right)$$

$$= -\sum_n -x_{n,j}\Big(\underbrace{-y_n(1-\hat{y}(x_n))}_{-(y_n - y_n\cancel{\hat{y}(x_n)})} + \underbrace{\hat{y}(x_n)(1-y_n)}_{\hat{y}(x_n) - y_n\cancel{\hat{y}(x_n)}}\Big)$$

$$= \sum_n x_{n,j}(\hat{y}(x_n) - y_n)$$

$$\frac{\partial E_{w,b}}{\partial w_j} = 0 \Rightarrow \sum_n x_{n,j}(\hat{y}(x_n) - y_n) = 0 \quad \begin{cases} x_{n,j} = 0 & \times \\ \hat{y}(x_n) = y_n & \checkmark \end{cases}$$

$$\sum_n \left(\frac{1}{1-e^{(wx_n+b)}} - y_n\right)x_n = 0$$

Gradient Desent: $$w_t = w_{t-1} - \eta \frac{\partial E}{\partial w}$$

$$\boxed{w_t = w_{t-1} - \eta \sum_n \left(\hat{y}(x_n) - y_n\right) x_n}$$

$$\frac{\partial E_{w,b}}{\partial b} = -\sum \Big(\underbrace{\frac{\partial}{\partial b}\left(y_n \ln \hat{y}(x_n)\right)}_{1} + \underbrace{\frac{\partial}{\partial b}\left((1-y_n)\ln(1-\hat{y}(x_n))\right)}_{2} \Big)$$

$$1: \frac{\partial}{\partial b}\left(y_n \ln \hat{y}(x_n)\right) = y_n \frac{1}{\hat{y}(x_n)} \frac{\partial \hat{y}(x_n)}{\partial b} = \frac{y_n}{\cancel{\hat{y}(x_n)}} \left(\cancel{\hat{y}(x_n)}\right)\left(1-\hat{y}(x_n)\right)$$

$$= y_n\left(1-\hat{y}(x_n)\right)$$

$$2: \frac{\partial}{\partial b}\left((1-y_n)\ln\left(1-\hat{y}(x_n)\right)\right) = \frac{1-y_n}{1-\hat{y}(x_n)} \cdot \frac{\partial\left(1-\hat{y}(x_n)\right)}{\partial b} = \frac{-(1-y_n)}{\cancel{1-\hat{y}(x_n)}} \left(\hat{y}(x_n)\right)\left(\cancel{1-\hat{y}(x_n)}\right)$$

$$= (y_n - 1)\hat{y}(x_n)$$

$$\frac{\partial E_{w,b}}{\partial b} = -\sum \Big(\underbrace{y_n\left(1-\hat{y}(x_n)\right)}_{y_n - \cancel{y_n \hat{y}}} + \underbrace{\hat{y}(x_n)(y_n - 1)}_{\cancel{\hat{y} y_n} - \hat{y}} \Big)$$

$$= -\sum_n \left(y_n - \hat{y}(x_n)\right) = 0 \qquad \boxed{y_n = \hat{y}(x_n)}$$

$$\frac{\partial E_{w,b}}{\partial b} = 0 \xrightarrow[GD]{} b_t = b_{t-1} - \eta \frac{\partial E}{\partial b} = b_{t-1} + \eta \sum_n \left(y_n - \hat{y}(x_n)\right)$$

$$\boxed{b_t = b_{t-1} + \eta \sum_n \left(y_n - \hat{y}(x_n)\right)}$$

min تابع هزینه زمانی رخ می‌دهد که $\hat{y}$ ها به y ها نزدیک شوند

سوال ۲:

2

آ)

covariate shift: این پدیده زمانی رخ می‌دهد که توزیع داده‌های ورودی در لایه‌های مختلف شبکه تغییر می‌کند: وقتی پارامترهای لایه‌ی قبلی در طول آموزش تغییر می‌کند در واقع توزیع داده‌های ورودی لایه‌های بعدی نیز تغییر می‌کند و باعث می‌شود هر لایه با توزیع‌های جدید به اجبار خودش را تطبیق دهد و این باعث کند شدن روند آموزش و همگرایی می‌شود.

BN: با نرمال‌سازی ورودی‌ها (میانگین را به صفر و واریانس را ۱ می‌کند و این کار را برای هر Batch انجام می‌دهد) باعث می‌شود توزیع داده‌ها پایدار نگه داشته شود و از تغییرات شدید در توزیع آنها جلوگیری می‌کند. باعث افزایش سرعت یادگیری و امکان استفاده از learning rate بالاتر می‌شود و همچنین از مشکل gradient vanishing جلوگیری می‌کند.

$$\mu = \frac{1}{n}\sum_{j=1}^{n} x_j \quad , \quad \sigma^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j-\mu)^2$$

x ↓ batch of input

normalize → $$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

ε ← برای پایداری عددی

ب) هر batch شامل نمونه‌های تصادفی از داده‌ها است، همچنین محاسبه‌ی میانگین و واریانس برای هر batch نیز نویز کوچکی به شبکه اضافه می‌کند و با اضافه کردن نویز به هر Batch به عمل نرمالایز کردن مانند یک regularizer عمل می‌کند و از overfitting جلوگیری می‌کند و به مدل اجازه می‌دهد که با یادگیری ویژگی‌های ثابت در mini-batch های مختلف تعمیم‌پذیری بهتری داشته باشد. پس جلوگیری از overfitting کرده و عملکرد مدل را روی داده‌های دیده نشده بهبود می‌بخشد

$n \longrightarrow$ mini-batch

$$\begin{cases} \hat{x}_i = x_i - \mu & [x_1, x_2 \cdots x_n] \quad \mu = \frac{1}{n}\sum_{j=1} x_j \\ y_i = \gamma \hat{x}_i + \beta & [y_1, y_2 \cdots y_n] \quad \frac{\partial L}{\partial x_i} = ? \quad \text{cost } F = L \end{cases}$$

chain rule: $$\frac{\partial L}{\partial \hat{x}_i} = \sum_{j=1}^{n} \frac{\partial L}{\partial y_j} \cdot \frac{\partial y_j}{\partial \hat{x}_i} \overset{1}{\Longrightarrow} \frac{\partial L}{\partial \hat{x}_i} = \gamma \cdot \frac{\partial L}{\partial y_i} \quad \text{I}$$

$$\left.\begin{aligned} \frac{\partial y_j}{\partial \hat{x}_i} \Rightarrow \quad & \frac{\partial y_j}{\partial \hat{x}_i} = \gamma \text{ in the same index } \boxed{j = i} \\ & \frac{\partial y_j}{\partial \hat{x}_i} = 0 \quad j \neq i \end{aligned}\right] \ 1$$

$$\hat{x}_i = x_i - \mu \quad , \quad \mu = \frac{1}{n}\sum_{j=1}^{n} x_j$$

$$\hookrightarrow \frac{\partial \hat{x}_i}{\partial x_i} = 1 - \frac{1}{n} \quad \boxed{j = i} \qquad \frac{\partial \hat{x}_i}{\partial x_i} = -\frac{1}{n} \quad \boxed{j \neq i}$$

II

$$\text{I}, \text{II} : \frac{\partial L}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial L}{\partial \hat{x}_j} \cdot \frac{\partial \hat{x}_j}{\partial x_i}$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i}\left(1 - \frac{1}{n}\right) + \sum_{j \neq i} \frac{\partial L}{\partial \hat{x}_j}\left(-\frac{1}{n}\right)$$

$$\boxed{\frac{\partial L}{\partial x_i} = \gamma \frac{\partial L}{\partial y_i}\left(1 - \frac{1}{n}\right) - \frac{\gamma}{n} \sum \frac{\partial L}{\partial y_i}}$$

ث) $n \to \infty \,,\, n=1$

$$n \to \infty \quad \begin{cases} 1-\frac{1}{n} \simeq 1 \\ -\frac{1}{n} \simeq 0 \end{cases} \Rightarrow \frac{\partial L}{\partial x_i} \simeq \gamma \cdot \frac{\partial L}{\partial y_i}$$

$$n=1 \to 1-\frac{1}{n} = 0 \to \frac{\partial L}{\partial x_i} = \gamma \cdot \frac{\partial L}{\partial y_1}$$

$$\to -\frac{1}{n} = -1 \quad \times$$

$$\to n=1 \to \mu = x_1 \Rightarrow \hat{x}_1 = x_1 - \mu = 0$$

که در واقع به ۰ نرمالایز می‌شود گویا BN تاثیری ندارد و نرمال سازی برای $n \geq 1$ اعمال می‌شود.

$n=1 \to \hat{x}_i = 0 \to$ نیازی به محاسبه گرادیان نیست

سوال ۳:

$\boxed{3}$

آ)

$$\frac{\partial \hat{y}_K}{\partial z_i^{(2)}} = ? \qquad z^{(1)} = w^{(1)} x + b^{(1)} \;,\; a^{(1)} = \text{leakyRelu}\, z^{(1)}$$

$$z^{(2)} = w^{(2)} a^{(1)} + b^{(2)} \;,\; \hat{y} = \text{softmax}(z^{(2)})$$

$$\text{softmax}: \quad \hat{y}_K = \frac{e^{z_K^{(2)}}}{\sum_j e^{z_j^{(2)}}}$$

$$\frac{\partial \hat{y}_K}{\partial z_i^{(2)}} = \begin{cases} \xrightarrow{i=K} \dfrac{\partial \hat{y}_K}{\partial z_i^{(2)}} = \hat{y}_K (1 - \hat{y}_K) \\[2ex] \xrightarrow{i \neq K} \;\; '' \;\; = \dfrac{-e^{z_K^{(2)}} e^{z_i^{(2)}}}{\left(\sum_{j=1}^{K} e^{z_j^{(2)}}\right)^2} = -\hat{y}_K \hat{y}_i \end{cases}$$

حالت کلی

$$\frac{\partial \hat{y}_K}{\partial z_i^{(2)}} = \hat{y}_K (\delta_{Ki} - \hat{y}_i)$$

ب)

$$L = \sum_{i=1}^{K} -y_i \log \hat{y}_i \qquad \frac{\partial L}{\partial z^{(2)}} = ? \qquad y_K = 1 \quad y_j = 0 \quad \boxed{i \neq j}$$

$$\frac{\partial L}{\partial z_i^{(2)}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i^{(2)}}$$

$$\frac{\partial L}{\partial z_i^{(2)}} = \frac{\partial}{\partial \hat{y}_i} \underbrace{\left(\sum_{i=1}^{K} -y_i \log \hat{y}_i \right)}_{-\underbrace{y_K}_{1} \log \hat{y}_K = -\log \hat{y}_K} \cdot \left(-\hat{y}_K \hat{y}_i \right)$$

$$\frac{\partial L}{\partial z_i^{(2)}} = \frac{\partial(-\log \hat{y}_K)}{\partial \hat{y}_i} \cdot (-\hat{y}_K \hat{y}_i) = \frac{-1}{\cancel{\hat{y}_K}} \cdot (-\cancel{\hat{y}_K} \hat{y}_i) = \hat{y}_i$$

پ)

$$\frac{\partial L}{\partial w^{(1)}} = \underbrace{\overbrace{\frac{\partial L}{\partial z^{(2)}}}^{\hat{y}} \cdot \frac{\partial z^{(2)}}{\partial a^{(1)}}}_{*} \cdot \frac{\partial a^{(1)}}{\partial w^{(1)}} \qquad \mathrm{I}$$

$$\frac{\partial L}{\partial z^{(2)}} = \hat{y}$$ از پارت قبلی

$$\frac{\partial z^{(2)}}{\partial a^{(1)}} = W^{(2)} \qquad * = \frac{\partial L}{\partial a^{(1)}} = \hat{y}\, W^{(2)} \qquad \mathrm{II}$$

$$\hat{a}_j^{(1)} = \begin{cases} z_j^{(1)} & z_j^{(1)} > 0 \\ 0.01\, z_j^{(1)} & z_j^{(1)} < 0 \end{cases} \Rightarrow * \; \frac{\partial \hat{a}^{(1)}}{\partial z^{(1)}} = \begin{cases} 1 & z^{(1)} > 0 \\ 0.01 & z^{(1)} < 0 \end{cases}$$

$$\frac{\partial a^{(1)}}{\partial w^{(1)}} = \frac{\partial a^{(1)}}{\partial \hat{a}^{(1)}} \cdot \frac{\partial \hat{a}^{(1)}}{\partial z^{(1)}} \cdot \underbrace{\frac{\partial z^{(1)}}{\partial w^{(1)}}}_{x}$$

$$a^{(1)} = \underbrace{\mathrm{Dropout}(\hat{a}^{(1)}, p = 0.2)}_{1-p=0.8} \Rightarrow \boxed{\frac{\partial a^{(1)}}{\partial \hat{a}^{(1)}} = 0.8} \qquad \mathrm{III}$$

I, II, III

$$\frac{\partial L}{\partial w^{(1)}} = \left(\hat{y}\, W^{(2)} \cdot \left[0.8\,(1) + 0.8\,(0.01)\right] x^T\right)$$

سوال ۴:

$\boxed{4}$ $y(u,v,z) = \varphi(u,v,z)$

- Jacobians J is the matrix of first order partial derivatives of y with respect to u, v, z

$$J = \begin{bmatrix} \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} & \frac{\partial y}{\partial z} \end{bmatrix}$$

- Hessian: is the metrix of second order ....

$$H = \begin{bmatrix} \frac{\partial^2 y}{\partial u^2} & \frac{\partial^2 y}{\partial u \partial v} & \frac{\partial^2 y}{\partial u \partial z} \\ \frac{\partial^2 y}{\partial v \partial u} & \frac{\partial^2 y}{\partial v^2} & \frac{\partial^2 y}{\partial v \partial z} \\ \frac{\partial^2 y}{\partial z \partial u} & \frac{\partial^2 y}{\partial z \partial u} & \frac{\partial^2 y}{\partial z^2} \end{bmatrix}$$

$$y(u,v,z) \rightarrow \nabla y = \begin{bmatrix} \frac{\partial y}{\partial u} \\ \frac{\partial y}{\partial v} \\ \frac{\partial y}{\partial z} \end{bmatrix} \rightsquigarrow J_{\vec{\nabla y}} = \begin{bmatrix} \frac{\partial \vec{\nabla y}}{\partial u} & \frac{\partial \vec{\nabla y}}{\partial v} & \frac{\partial \vec{\nabla y}}{\partial z} \end{bmatrix}$$

$$J_{\vec{\nabla y}} = \begin{bmatrix} \nabla^T \frac{\partial y}{\partial u} \\ \nabla^T \frac{\partial y}{\partial v} \\ \nabla^T \frac{\partial y}{\partial z} \end{bmatrix}$$

$$\boxed{J_{\vec{\nabla y}} = H}$$

سوال ۵:

$\boxed{5}$ $$J_1 = 0.5\left(y_d - \sum_{k=1}^{n} \delta_k w_k x_k\right)^2 \quad , \quad \delta_k \sim Normal(1, \sigma^2)$$

$$\frac{\partial J_1}{\partial w_i} = 0.5 \times 2(-\delta_i x_i)\left(y_d - \sum_{k=1}^{n} \delta_k w_k x_k\right)$$

$$\frac{\partial J_1}{\partial w_i} = +\delta_i x_i \left(y_d - \sum_{k=1}^{n} \delta_k w_k x_k\right)$$

Dropout به عنوان Regularization: در روش Dropout در هنگام آموزش به صورت تصادفی تعدادی از نورون ها (وزن‌های متصل به آن نورون‌ها) در هر لایه drop می‌شوند و از بازی خارج می‌شوند. این به این معناست که در هر مرحله از آموزش برخی نورون‌ها از فرآیند آموزش خارج شده و مدل مجبور است ویژگی‌ها و پارامترها را بدون وابستگی به نورون‌های خاص یاد و تعمیم کند. روش dropout باعث می‌شود که مدل از یادگیری ویژگی‌های خاص و بیش از حد وابسته به داده‌های آموزشی جلوگیری کرده و از overfitting جلوگیری می‌کند و به مدل کمک می‌کند بهتر و عمومی‌تر عمل کند.

Regularized-Non: تابع هدف ترکیبی از دو تابع هدف معمولی و تابع هدف منظم است.

تابع هدف معمولی: این بخش تنها به کمینه کردن خطای مدل می‌پردازد ← این بخش به مدل می‌گوید که باید پیش‌بینی‌های دقیق از داده‌های آموزشی داشته باشد.

تابع هدف منظم: این بخش به منظور جلوگیری از overfitting و بهبود تعمیم پذیری مدل به کار می‌رود. در این بخش معمولا یک جریمه (Penalty) به وزن‌ها های مدل یا پیچیدگی آن اضافه می‌شود به عنوان مثال در L_2 تابع هدف به صورت $L = \sum_{i=1}^{n} Loss(\hat{y}, y_i) + \lambda_j \sum_{j=1}^{M} w_j$ است که λ یک ضریب تنظیم است که اندازه جریمه را کنترل می‌کند و w_j وزن‌های مدل هستند. این بخش مدل را جریمه می‌کند برای داشتن وزن‌های بزرگ که می‌تواند باعث overfitting شود.

حال Regularized-Non تابع هدفی است که ترکیب ۲ بخش بالاست یعنی هم به دقت پیش‌بینی مدل بر روی داده‌های آموزشی و هم کمیت پیچیدگی مدل با اعمال جریمه کنترل می‌کند.

سوال ۶:

$\boxed{6}$ $f(x) = g'(x)$ $\qquad f(x^*) = 0 \quad , \; f'(x^*) \neq 0$

from net

$$\boxed{\begin{array}{l} \text{newton's method:} \\ f(x_k + t) \simeq f(x_k) + f'(x_k)t + \frac{1}{2} f''(x_k)t^2 \\ \frac{d f(x_k + t)}{dt} = 0 \Rightarrow f'(x_k) + f''(x_k)t = 0 \rightarrow t = \frac{f'(x_k)}{f''(x_k)} \\ x_{k+1} = x_k + t = x_k - \frac{f'(x_k)}{f''(x_k)} \qquad \|x_{k+1} - x_*\| \leq \frac{1}{2} \|x_k - x_*\|^2 \end{array}}$$

out question: $f(x) = g'(x) \Rightarrow$

$$x^* = \arg\min g(x) \rightsquigarrow \frac{\partial g(x)}{\partial x^*} = 0 \qquad f(x^*) = 0$$

$$g(x_k + t) = g(x_k) + g'(x_k)t + \frac{1}{2} g''(x_k)t^2$$

$$\frac{\partial g(x_k + t)}{\partial t} = 0 \Rightarrow g'(x_k) + \frac{1}{2} \cdot 2t(g''(x_k)) = 0 \Rightarrow t = \frac{-g'(x_k)}{g''(x_k)}$$

$$x_{k+1} = x_k + t \Rightarrow x_{k+1} = x_k - \frac{g'(x_k)}{g''(x_k)}$$

$$\xrightarrow{g'(x) = f(x)} \boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}} \longrightarrow x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$x_{n+1} = x_n - \frac{g'(x_n)}{g''(x_n)}$$

second order: $\quad e_n = x_n - x^* \quad , \quad e_{n+1} = x_{n+1} - x^*$

$$f(x_n) = f(x^* + e_n) = f(x^*) + f'(x^*)e_n + \frac{1}{2} f''(x^*)e_n^2 + \cdots$$

0

$$\boxed{f(x_n) \simeq f'(x^*)e_n + \frac{1}{2} f''(x^*)e_n^2}$$

$$f'(x_n) = f'(x^* + e_n) = \underbrace{f'(x^*) + f''(x^*)e_n} + \frac{1}{2} f'''(x^*)e_n^2 + \cdots$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \Rightarrow x_{n+1} = x_n - \frac{\overset{0}{\cancel{f(x^*)}} + f'(x^*)e_n + \frac{1}{2} f''(x^*)e_n^2}{f'(x^*) + f''(x^*)e_n}$$

$$e_{n+1} = x_{n+1} - x^* = \left(x_n - \frac{f'(x^*)e_n + \frac{1}{2} f''(x^*)e_n^2}{f'(x^*) + f''(x^*)e_n}\right) - x^* \xrightarrow{e_n = x_n - x^*}$$

$$e_{n+1} = e_n - \frac{f'(x^*)e_n + \frac{1}{2} f''(x^*)e_n^2}{f'(x^*) + f''(x^*)e_n} \underset{e_n \text{ small}}{\simeq} e_n - \frac{f'(x^*)e_n + \frac{1}{2} f''(x^*)e_n^2}{f'(x^*)}$$

$$e_{n+1} \approx \cancel{e_n} - \cancel{e_n} + \frac{1}{2} \frac{f''(x^*)e_n^2}{f'(x^*)}$$

$$e_{n+1} \simeq \underbrace{\frac{1}{2} \cdot \frac{f''(x^*)}{f'(x^*)}}_{C} e_n^2 \Longrightarrow e_{n+1} \simeq C\,|e_n|^2 , \quad C = \frac{1}{2} \frac{f''(x^*)}{f'(x^*)}$$

سوال ۷:

7 $\hat{y}_i = \dfrac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ $\qquad L(z,y) = -\sum_{K=1}^{K} y_K \log \hat{y}_K$

آ)

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{\partial}{\partial \hat{y}_i}\left(-\sum_{K=1}^{K} y_K \log \hat{y}_K\right) = \frac{-y_i}{\hat{y}_i}$$

$$\frac{\partial \hat{y}_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) = \hat{y}_i(\delta_{ij} - \hat{y}_i)$$

$$\frac{\partial L}{\partial z} = \sum_{j=1}^{K} \frac{\partial L}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i} = \sum_{j=1}^{K} -\frac{y_j}{\cancel{\hat{y}_j}} \cdot \cancel{\hat{y}_j}(\delta_{ij} - \hat{y}_i)$$

$$\frac{\partial L}{\partial z} = \hat{y}_i - y_i \longrightarrow \boxed{\frac{\partial L}{\partial z} = \hat{Y} - Y}$$

ب)

$$i = j \Rightarrow \frac{\partial^2 L}{\partial z_i^2} = \hat{y}_i(1 - \hat{y}_i)$$

الف)

$$i \neq j \Rightarrow \frac{\partial^2 L}{\partial z_i \partial z_j} = -\hat{y}_i \hat{y}_j$$

$$H_{ij} = \frac{\partial^2 L}{\partial z_i \partial z_j} = \frac{\partial}{\partial z_j}\left(\overbrace{\hat{y}_i - y_i}^{\frac{\partial L}{\partial z_i}}\right)$$

$$H_{ij} = \begin{cases} \hat{y}_i(1 - \hat{y}_i) & i = j \\ -\hat{y}_i \hat{y}_j & i \neq j \end{cases}$$

$$H = \begin{bmatrix} \hat{y}_1(1-\hat{y}_1) & -\hat{y}_1\hat{y}_2 & \cdots & \hat{y}_1\hat{y}_K \\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1-\hat{y}_2) & \cdots & -\hat{y}_2\hat{y}_K \\ \vdots & \vdots & \ddots & \vdots \\ -\hat{y}_K\hat{y}_1 & -\hat{y}_K\hat{y}_2 & \cdots & \hat{y}_K(1-\hat{y}_K) \end{bmatrix}$$

ب) باید اثبات شود که $v^T H v \geq 0$

$$H_{ij} = \begin{cases} \hat{y}_i (1-\hat{y}_i) & i=j \\ -\hat{y}_i \hat{y}_j & i\neq j \end{cases}$$

$$v^T H v = \sum_{i=1}^{K} \sum_{j=1}^{K} v_i v_j H_{ij}$$

$$v^T H v = \underbrace{\sum_{i=1}^{K} v_i^2 \hat{y}_i (1-\hat{y}_i)}_{\text{sum of variance} \xrightarrow{so} >0} - \overbrace{\sum_{i=1}^{K} \sum_{j\neq i} v_i v_j \hat{y}_i \hat{y}_j}^{\text{covariances between classes}}$$

non_negetive

$$\boxed{v^T H v \geq 0}$$

عناصر قطری: واریانس احتمال هر کلاس را نشان می‌دهد.

عناصر غیرقطری: کوواریانس بین کلاس‌های مختلف را نشان می‌دهد.

☆ مثبت نیمه‌معین بودن: این ماتریس ساختاری شبیه به یک ماتریس کوواریانس دارد زیرا: عناصر احتمالات تابع softmax ساخته شده‌اند. مجموعشان ۱ و هر کدام بین صفر تا ۱ قرار دارند. ماتریس کوواریانس همواره مثبت و نیمه‌معین است به این معنا که هر مقدار ویژه آنها غیرمنفی است.

ج) مثبت نیمه‌معین بودن ماتریس هسین نشان می‌دهد که تابع آنتروپی متقاطع نسبت به ورودی‌های قبل از softmax محدب است. محدب بودن این تابع تضمین می‌کند که به روش گرادیان نزولی روی این تابع به سمت کمینه Global حرکت می‌کند و این ویژگی به بهبود پایداری و دقت در یادگیری شبکه کمک می‌کند.

بخش عملی

Github link:
Google drive link: https://drive.google.com/drive/folders/1LhpeUO8tcGSwovNlqIx1e3SonjVImZYa?usp=sharing

سوال ۱:

برای پیاده سازی rosenbrock کد زیر را اجرا میکنیم که در ان مشق اول نسبت به x[0], x[1] میگیریم

```
def rosenbrock(x):
    """Returns the value and gradient of Rosenbrock's function at x: 2d vector"""
    x1, x2 = x[0], x[1]
    val = 100*(x[1]-x[0]**2)**2+(x[0]-1)**2
    dv_dx0 = -400 * x[0] * (x[1] - x[0] ** 2) + 2 * (x[0] - 1)
    dv_dx1 = 200 * (x[1] - x[0] ** 2)
    grad = np.array([dv_dx0, dv_dx1])

    return val, grad
```

برای پیاده سازی rosenbrock_hessianکد زیر را اجرا میکنیم که در ان مشق دوم نسبت به x[0], x[1] میگیریم

```
def rosenbrock_hessian(x):
    """Returns the value, gradient and hessian of Rosenbrock's function at x: 2d
vector"""
    val, grad = rosenbrock(x)

    # Hessian matrix components
    d2v_dx0x0 = 1200 * x[0] ** 2 - 400 * x[1] + 2
    d2v_dx0x1 = -400 * x[0]
    d2v_dx1x0 = -400 * x[0]
    d2v_dx1x1 = 200

    # Hessian matrix
    hessian = np.array([[d2v_dx0x0, d2v_dx0x1],
                        [d2v_dx1x0, d2v_dx1x1]])

    return val, grad, hessian
```

و GD هم به صورت زیر پیاده سازی می شد

```
def GD(f, theta0, alpha, stop_tolerance=1e-10, max_steps=1000000):
    """Runs gradient descent algorithm on f.

    Args:
        f: function that when evaluated on a Theta of same dtype and shape as Theta0
            returns a tuple (value, dv_dtheta) with dValuedTheta of the same shape
            as Theta
        theta0: starting point
        alpha: step length
        stop_tolerance: stop iterations when improvement is below this threshold
        max_steps: maximum number of steps
    Returns:
```

```
        tuple:
        - theta: optimum theta found by the algorithm
        - history: list of length num_steps containing tuples (theta, (val,
dv_dtheta: np.array))

    """
    history = []

    theta = theta0

    step = 0

        # Initial function evaluation
    val, grad = f(theta)
    print(grad)
    history.append((theta.copy(), (val, grad)))

    while step < max_steps:
        # Gradient descent step: update theta
        theta = theta - alpha * grad

        # Evaluate function at new theta
        new_val, grad = f(theta)
        history.append((theta.copy(), (new_val, grad)))

        # Check for convergence
        improvement = abs(val - new_val)
        if improvement < stop_tolerance:
            break

        # Update the value for next iteration
        val = new_val
        step += 1

    history.append([theta, f(theta)])
    return theta, history
```

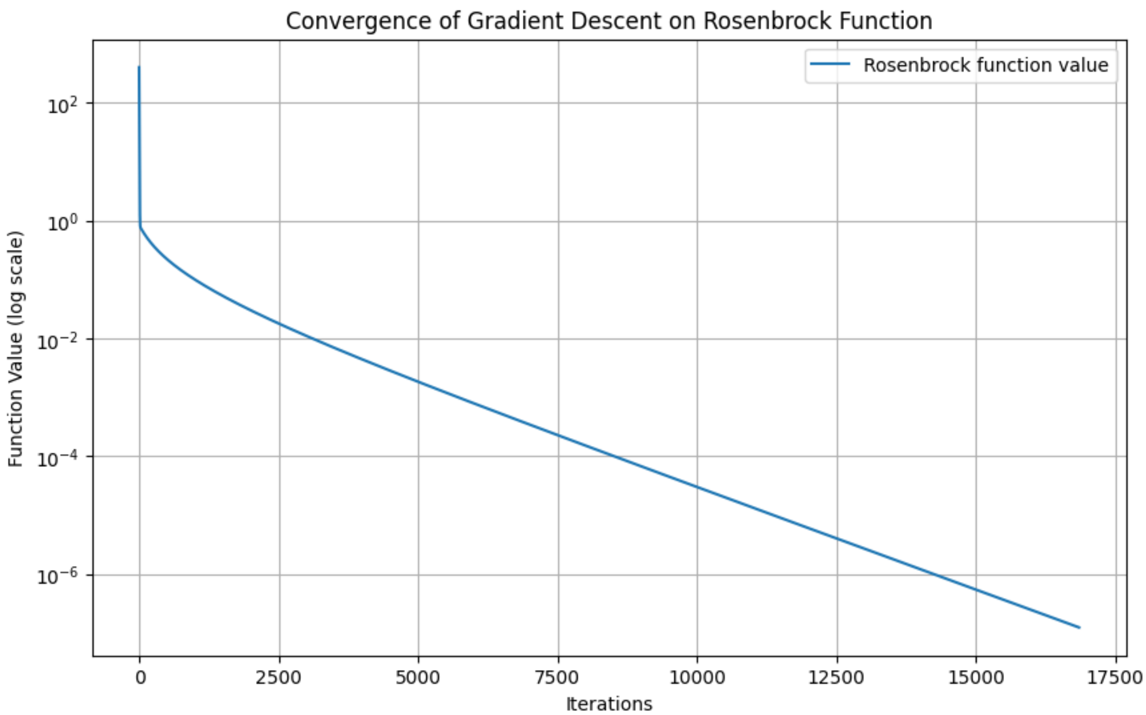
برای پیدا کردن optimum تابع rosenbrock داریم:

```
X0 = [0.,2.]
Xopt, Xhist = GD(rosenbrock, X0, alpha=1e-3, stop_tolerance=1e-10, max_steps=1e6)

print ("Found optimum at %s in %d steps (true minimum is at [1,1])" % (Xopt,
len(Xhist)))

# Plot how the value changes over iterations
#TODO
# Extract function values over iterations
values = [entry[1][0] for entry in Xhist]  # Get function values from history

# Plot the convergence of function value
```

```
plt.figure(figsize=(10, 6))
plt.plot(values, label="Rosenbrock function value")
plt.yscale("log")  # Use log scale to see the improvement more clearly
plt.xlabel("Iterations")
plt.ylabel("Function Value (log scale)")
plt.title("Convergence of Gradient Descent on Rosenbrock Function")
plt.legend()
plt.grid(True)
plt.show()
```

مقدار اپتیموم [0.99964674 0.99929219] است که به مقدار ۱ و ۱ نزدیک می باشدو نمودار ان هم به شکل زیر است.

```
Found optimum at [0.99964674 0.99929219] in 16855 steps (true minimum is at [1,1])
```



برای پیاده سازی متد Newton به اپتیموم [1,1] میشود

```
Found optimum at [1. 1.] (true minimum is at [1,1])
```

```
# Newton's Method
def Newton(f, theta0, alpha=1, stop_tolerance=1e-10, max_steps=1000000):
    """Performs Newton's optimization method with a simple line search.

    Args:
        f: function that when evaluated on a Theta of same dtype and shape as Theta0
            returns a tuple (value, gradient, hessian), where gradient and Hessian
            have the same shape as Theta.
        theta0: starting point.
        alpha: step length for backtracking line search (default is 1).
        stop_tolerance: stop iterations when the norm of the gradient is below this
threshold.
        max_steps: maximum number of iterations.
```

```
    Returns:
        tuple:
        - theta: optimal Theta after convergence or maximum steps.
        - history: list of tuples (theta, value, gradient) containing the
optimization path.
    """
    theta = theta0
    history = []
    # TODO
    for step in range(max_steps):
        # Evaluate the function, gradient, and Hessian
        val, grad, hessian = f(theta)

        # Store the current theta and function value
        history.append((theta.copy(), val))

        # Check if gradient norm is below the tolerance
        if np.linalg.norm(grad) < stop_tolerance:
            break

        # Update rule using inverse of the Hessian
        try:
            theta -= np.linalg.inv(hessian).dot(grad)
        except np.linalg.LinAlgError:
            print("Hessian is singular at step", step)
            break  # Stop if the Hessian is singular

    return theta, history

# Test Newton's method on the Rosenbrock function
X0 = [0., 2.]  # Initial guess
Xopt, Xhist = Newton(rosenbrock_hessian, X0)

print("Found optimum at %s (true minimum is at [1,1])" % Xopt)
```

Part two: MLP for MNIST Classification

برای این بخش ۳ فایل مد نظر را کامل کرده و داریم: با بچ سایز ۳۲ و تعداد epoch =20

نتایج فاز ترین:

و با مدل mlp با لایه های FCLayer and SigmoidLayer:

```
sigmoidMLP = nn.Sequential(
    FCLayer(784, 128),
    SigmoidLayer(),
    FCLayer(128, 10)
)
```

```
Epoch [1] Average training loss: 0.0771, Average training accuracy: 0.4956

Epoch [2] Average training loss: 0.0556, Average training accuracy: 0.7378

Epoch [3] Average training loss: 0.0506, Average training accuracy: 0.7836

Epoch [4] Average training loss: 0.0482, Average training accuracy: 0.8046
```

```
Epoch [5] Average training loss: 0.0466, Average training accuracy: 0.8173

Epoch [6] Average training loss: 0.0456, Average training accuracy: 0.8254

Epoch [7] Average training loss: 0.0448, Average training accuracy: 0.8307

Epoch [8] Average training loss: 0.0442, Average training accuracy: 0.8344

Epoch [9] Average training loss: 0.0437, Average training accuracy: 0.8384

Epoch [10] Average training loss: 0.0432, Average training accuracy: 0.8408

Epoch [11] Average training loss: 0.0429, Average training accuracy: 0.8430

Epoch [12] Average training loss: 0.0426, Average training accuracy: 0.8455

Epoch [13] Average training loss: 0.0423, Average training accuracy: 0.8465

Epoch [14] Average training loss: 0.0420, Average training accuracy: 0.8474

Epoch [15] Average training loss: 0.0418, Average training accuracy: 0.8489

Epoch [16] Average training loss: 0.0415, Average training accuracy: 0.8504

Epoch [17] Average training loss: 0.0413, Average training accuracy: 0.8517

Epoch [18] Average training loss: 0.0411, Average training accuracy: 0.8514

Epoch [19] Average training loss: 0.0409, Average training accuracy: 0.8529

Epoch [20] Average training loss: 0.0407, Average training accuracy: 0.8533
```

برای فاز تست داریم :

```
The test accuracy is 0.8628.
```

برای MLPبا لایه های FCLayer and ReLULayer:

```
reluMLP = nn.Sequential(
    FCLayer(784, 128),
    ReLULayer(),
    FCLayer(128, 10)
)
```

```
Epoch [1] Average training loss: 0.0712, Average training accuracy: 0.6373

Epoch [2] Average training loss: 0.0467, Average training accuracy: 0.8231

Epoch [3] Average training loss: 0.0399, Average training accuracy: 0.8589

Epoch [4] Average training loss: 0.0357, Average training accuracy: 0.8765

Epoch [5] Average training loss: 0.0327, Average training accuracy: 0.8874

Epoch [6] Average training loss: 0.0305, Average training accuracy: 0.8953

Epoch [7] Average training loss: 0.0288, Average training accuracy: 0.9011

Epoch [8] Average training loss: 0.0274, Average training accuracy: 0.9060
```

```
Epoch [9] Average training loss: 0.0262, Average training accuracy: 0.9103
Epoch [10] Average training loss: 0.0253, Average training accuracy: 0.9131
Epoch [11] Average training loss: 0.0244, Average training accuracy: 0.9162
Epoch [12] Average training loss: 0.0237, Average training accuracy: 0.9185
Epoch [13] Average training loss: 0.0231, Average training accuracy: 0.9205
Epoch [14] Average training loss: 0.0225, Average training accuracy: 0.9227
Epoch [15] Average training loss: 0.0220, Average training accuracy: 0.9248
Epoch [16] Average training loss: 0.0215, Average training accuracy: 0.9264
Epoch [17] Average training loss: 0.0211, Average training accuracy: 0.9279
Epoch [18] Average training loss: 0.0207, Average training accuracy: 0.9289
Epoch [19] Average training loss: 0.0204, Average training accuracy: 0.9297
Epoch [20] Average training loss: 0.0200, Average training accuracy: 0.9309
```

برای فاز تست داریم :

```
The test accuracy is 0.9312.
```

در این تغییرات، مدل شبکه عصبی را طوری تغییر دادم که احتمال **اورفیت** (Overfitting)بیشتری داشته باشد. اورفیت زمانی اتفاق می‌افتد که مدل بیش از حد پیچیده می‌شود و بیشتر به حفظ جزئیات داده‌های آموزشی پرداخته و قادر به تعمیم خوب روی داده‌های جدید نخواهد بود.

تغییرات اصلی که اعمال کردم:

- **افزایش پیچیدگی شبکه (لایه‌های بیشتر و تعداد نورون‌های بیشتر:**
- من تعداد نورون‌ها در لایه‌های مخفی را افزایش دادم (مثلاً از ۵۱۲ به ۱۰۲۴ و از ۲۵۶ به ۵۱۲) که باعث پیچیده‌تر شدن مدل می‌شود.
- همچنین تعداد لایه‌ها را از ۳ لایه به ۵ لایه افزایش دادم. این باعث می‌شود که مدل ظرفیت بیشتری برای یادگیری پارامترها داشته باشد.

۲. **حذف لایه‌های:Dropout**
- در مدل اصلی، از لایه‌های **Dropout**استفاده می‌شود که یک روش منظم‌کننده است و به مدل کمک می‌کند از اورفیت جلوگیری کند.
- من این لایه‌های Dropout را حذف کردم تا مدل بدون هیچ نوع محدودیتی بتواند پارامترها را یاد بگیرد و بیشتر به حفظ جزئیات داده‌های آموزشی بپردازد. این تغییر باعث می‌شود مدل به راحتی روی داده‌های آموزشی اورفیت کند.

۳. **افزایش ظرفیت مدل:**
- با افزایش تعداد لایه‌ها و نورون‌ها، مدل به طور کلی پیچیده‌تر و بزرگ‌تر شده است. این امر باعث می‌شود که مدل توانایی یادگیری بیشتر و به تبع آن احتمال اورفیت شدن نیز افزایش یابد، به ویژه زمانی که داده‌ها کافی نباشند یا مدل برای تعداد زیادی از دوره‌های آموزشی آموزش ببیند.

چرا این تغییرات باعث اورفیت می‌شوند؟

- **افزایش تعداد پارامترها** :مدل حالا تعداد بیشتری پارامتر برای یادگیری دارد. این باعث می‌شود مدل قادر به یادگیری جزییات بیشتری از داده‌های آموزشی باشد و به راحتی روی داده‌های آموزشی اورفیت کند.
- **حذف Dropout**: Dropout یک تکنیک است که برای جلوگیری از اورفیت استفاده می‌شود. حذف این تکنیک باعث می‌شود که مدل به صورت کامل به یادگیری داده‌ها پرداخته و احتمال اورفیت شدن بیشتر می‌شود.
- **شبکه عمیق‌تر** :با اضافه کردن لایه‌های بیشتر، مدل پیچیده‌تر شده و ظرفیت یادگیری آن افزایش می‌یابد که می‌تواند باعث اورفیت روی داده‌های آموزشی شود.

نتیجه:

این تغییرات باعث می‌شود که مدل توانایی یادگیری جزئیات زیادی از داده‌های آموزشی داشته باشد و احتمالاً نتواند به خوبی روی داده‌های جدید عمل کند (یعنی **اورفیت** می‌کند). این مدل برای آزمایش اورفیت مناسب است، به خصوص اگر داده‌های آموزشی کوچک یا تعداد دوره‌های آموزش زیاد باشد.

لایه با اور فیت :

```
#TODO: overfit the reluMLP model
num_epoch = 50

reluMLP = nn.Sequential(
    FCLayer(784, 1024),
    ReLULayer(),
    FCLayer(1024, 512),
    ReLULayer(),
    FCLayer(512, 256),
    ReLULayer(),
    FCLayer(256, 128),
    ReLULayer(),
    FCLayer(128, 64),
    ReLULayer(),
    FCLayer(64, 10)
)
criterion = nn.MSELoss()

# Initialize optimizer
sgd = SGD(reluMLP.parameters(), learning_rate=0.5)

# Train the model
reluMLP = train(reluMLP, criterion, sgd, train_dataloader, num_epoch, device=device)

test(reluMLP, test_dataloader, device)
```

```
Epoch [1] Average training loss: 0.0154, Average training accuracy: 0.9286

Epoch [2] Average training loss: 0.0061, Average training accuracy: 0.9726

Epoch [3] Average training loss: 0.0041, Average training accuracy: 0.9831

Epoch [4] Average training loss: 0.0029, Average training accuracy: 0.9884

Epoch [5] Average training loss: 0.0022, Average training accuracy: 0.9921

Epoch [6] Average training loss: 0.0016, Average training accuracy: 0.9946

Epoch [7] Average training loss: 0.0012, Average training accuracy: 0.9966

Epoch [8] Average training loss: 0.0009, Average training accuracy: 0.9974

Epoch [9] Average training loss: 0.0007, Average training accuracy: 0.9981

Epoch [10] Average training loss: 0.0006, Average training accuracy: 0.9989
```

```
Epoch [11] Average training loss: 0.0005, Average training accuracy: 0.9991
Epoch [12] Average training loss: 0.0004, Average training accuracy: 0.9993
Epoch [13] Average training loss: 0.0003, Average training accuracy: 0.9995
Epoch [14] Average training loss: 0.0003, Average training accuracy: 0.9996
Epoch [15] Average training loss: 0.0002, Average training accuracy: 0.9997
Epoch [16] Average training loss: 0.0002, Average training accuracy: 0.9998
Epoch [17] Average training loss: 0.0002, Average training accuracy: 0.9998
Epoch [18] Average training loss: 0.0002, Average training accuracy: 0.9998
Epoch [19] Average training loss: 0.0001, Average training accuracy: 0.9998
Epoch [20] Average training loss: 0.0001, Average training accuracy: 0.9998
Epoch [21] Average training loss: 0.0001, Average training accuracy: 0.9998
Epoch [22] Average training loss: 0.0001, Average training accuracy: 0.9998
Epoch [23] Average training loss: 0.0001, Average training accuracy: 0.9998
Epoch [24] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [25] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [26] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [27] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [28] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [29] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [30] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [31] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [32] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [33] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [34] Average training loss: 0.0001, Average training accuracy: 1.0000
Epoch [35] Average training loss: 0.0001, Average training accuracy: 0.9999
Epoch [36] Average training loss: 0.0001, Average training accuracy: 1.0000
Epoch [37] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [38] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [39] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [40] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [41] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [42] Average training loss: 0.0000, Average training accuracy: 1.0000
```

```
Epoch [43] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [44] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [45] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [46] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [47] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [48] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [49] Average training loss: 0.0000, Average training accuracy: 1.0000
Epoch [50] Average training loss: 0.0000, Average training accuracy: 1.0000
```

از epochحدودا ۱۰ به بعد داده های آموزش را حفظ کرده است

```
The test accuracy is 0.9836.
```

بعد از اضافه کردن دراپ اوت:

```
from layers import DropoutLayer
#TODO: add DropoutLayer to your model

#TODO: overfit the reluMLP model
num_epoch = 30

reluMLP = nn.Sequential(
    FCLayer(784, 1024),
    ReLULayer(),
    DropoutLayer(0.5),  # Dropout layer with a rate of 0.5

    FCLayer(1024, 512),
    ReLULayer(),
    DropoutLayer(0.5),  # Dropout layer with a rate of 0.5

    FCLayer(512, 256),
    ReLULayer(),
    DropoutLayer(0.5),  # Dropout layer with a rate of 0.5
    FCLayer(256, 128),
    ReLULayer(),
    DropoutLayer(0.5),  # Dropout layer with a rate of 0.5

    FCLayer(128, 64),
    ReLULayer(),
    DropoutLayer(0.5),  # Dropout layer with a rate of 0.5

    FCLayer(64, 10)
)
criterion = nn.MSELoss()

# Initialize optimizer
```

```python
sgd = SGD(reluMLP.parameters(), learning_rate=0.5)

# Train the model
reluMLP = train(reluMLP, criterion, sgd, train_dataloader, num_epoch, device=device)

test(reluMLP, test_dataloader, device)
```

```
Epoch [1] Average training loss: 0.0875, Average training accuracy: 0.1939
Epoch [2] Average training loss: 0.0793, Average training accuracy: 0.2974
Epoch [3] Average training loss: 0.0737, Average training accuracy: 0.3886
Epoch [4] Average training loss: 0.0673, Average training accuracy: 0.4494
Epoch [5] Average training loss: 0.0636, Average training accuracy: 0.4680
Epoch [6] Average training loss: 0.0613, Average training accuracy: 0.4785
Epoch [7] Average training loss: 0.0601, Average training accuracy: 0.4808
Epoch [8] Average training loss: 0.0592, Average training accuracy: 0.4874
Epoch [9] Average training loss: 0.0589, Average training accuracy: 0.4880
Epoch [10] Average training loss: 0.0585, Average training accuracy: 0.4889
Epoch [11] Average training loss: 0.0582, Average training accuracy: 0.4918
Epoch [12] Average training loss: 0.0580, Average training accuracy: 0.4896
Epoch [13] Average training loss: 0.0580, Average training accuracy: 0.4914
Epoch [14] Average training loss: 0.0576, Average training accuracy: 0.4942
Epoch [15] Average training loss: 0.0577, Average training accuracy: 0.4949
Epoch [16] Average training loss: 0.0576, Average training accuracy: 0.4940
Epoch [17] Average training loss: 0.0575, Average training accuracy: 0.4929
Epoch [18] Average training loss: 0.0573, Average training accuracy: 0.4942
Epoch [19] Average training loss: 0.0572, Average training accuracy: 0.4963
Epoch [20] Average training loss: 0.0572, Average training accuracy: 0.4947
Epoch [21] Average training loss: 0.0571, Average training accuracy: 0.4967
Epoch [22] Average training loss: 0.0570, Average training accuracy: 0.4949
Epoch [23] Average training loss: 0.0568, Average training accuracy: 0.4970
Epoch [24] Average training loss: 0.0567, Average training accuracy: 0.4974
Epoch [25] Average training loss: 0.0568, Average training accuracy: 0.4980
Epoch [26] Average training loss: 0.0566, Average training accuracy: 0.4986
Epoch [27] Average training loss: 0.0566, Average training accuracy: 0.4994
```

```
Epoch [28] Average training loss: 0.0566, Average training accuracy: 0.4994

Epoch [29] Average training loss: 0.0566, Average training accuracy: 0.4999

Epoch [30] Average training loss: 0.0564, Average training accuracy: 0.4998

The test accuracy is 0.5041.
```

سوال ۲:

Introduction to Loss Functions

```
class SimpleMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_hidden_layers=1,
last_layer_activation_fn=None):
        super(SimpleMLP, self).__init__()
        # TODO: Define the layers of the MLP
        layers = []

        # Input layer
        layers.append(nn.Linear(input_dim, hidden_dim))
        layers.append(nn.ReLU())

        # Hidden layers
        for _ in range(num_hidden_layers - 1):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.ReLU())

        # Output layer
        layers.append(nn.Linear(hidden_dim, output_dim))

        # Add the last layer activation function only if it's provided
        if last_layer_activation_fn is not None:
            layers.append(last_layer_activation_fn())

        # Combine layers into a Sequential module
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
class SimpleMLPTrainer:
    def __init__(self, model, criterion, optimizer):
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer

    def train(self, train_loader, num_epochs):
        #TODO: Implement the training loop
        #Note: You should also print the training loss at each epoch, use tqdm for
progress bar
        #Note: You should return the training loss at each epoch
        self.model.train()
        epoch_losses = []

        for epoch in range(num_epochs):
            total_loss = 0.0
            for inputs, targets in tqdm(train_loader, desc=f"Epoch
{epoch+1}/{num_epochs}"):
                # Ensure targets are 1D (class indices)
```

```
            targets = targets.view(-1)  # Convert to 1D tensor if needed

            # Forward pass with log_softmax for NLLLoss
            outputs = self.model(inputs)
            loss = self.criterion(outputs, targets)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            total_loss += loss.item()

        average_loss = total_loss / len(train_loader)
        epoch_losses.append(average_loss)
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {average_loss:.4f}")

    return epoch_losses
    pass

def evaluate(self, val_loader):
    #TODO: Implement the evaluation loop
    #Note: You should return the validation loss and accuracy
    self.model.eval()
    total_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            # Ensure targets are 1D (class indices)
            targets = targets.view(-1)

            # Forward pass with log_softmax for NLLLoss
            outputs = F.log_softmax(self.model(inputs), dim=1)
            loss = self.criterion(outputs, targets)
            total_loss += loss.item()

            # Calculate accuracy
            _, predicted_classes = torch.max(outputs, 1)
            correct += (predicted_classes == targets).sum().item()
            total += targets.size(0)

    average_loss = total_loss / len(val_loader)
    accuracy = (correct / total) * 100 if total > 0 else 0
    return average_loss, accuracy

    pass
```

```
# Load dataset
train_url =
"https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
data = pd.read_csv(train_url)

# Preprocessing (simple example)
data = data[['Pclass', 'Sex', 'Age', 'Fare', 'Survived']].dropna()
data['Sex'] = data['Sex'].map({'male': 0, 'female': 1})

# TODO: Convert the data to PyTorch tensors and create a DataLoader
X = data[['Pclass', 'Sex', 'Age', 'Fare']].values
y = data['Survived'].values

scaler = StandardScaler()
X = scaler.fit_transform(X)
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32).view(-1, 1)  # Reshape for
compatibility

# TODO: Split the data into training and validation sets
dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32)

# TODO: Define the model, criterion, and optimizer
input_dim = X.shape[1]    # Number of features
hidden_dim = 16           # Adjust based on experimentation
output_dim = 1            # Binary classification (Survived or not)

# Model
model = SimpleMLP(input_dim, hidden_dim, output_dim, num_hidden_layers=2,
last_layer_activation_fn=None)

# Criterion (loss function) and optimizer
criterion = nn.BCEWithLogitsLoss()  # Use BCEWithLogitsLoss for binary classification
without sigmoid activation in the model
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Print dataset and model information
print(f"Training samples: {len(train_dataset)}, Validation samples:
{len(val_dataset)}")
print(model)
```

```
 Training samples: 571, Validation samples: 143
SimpleMLP(
  (model): Sequential(
    (0): Linear(in_features=4, out_features=16, bias=True)
    (1): ReLU()
```

```
    (2): Linear(in_features=16, out_features=16, bias=True)
    (3): ReLU()
    (4): Linear(in_features=16, out_features=1, bias=True)
  )
)
```

L1Loss

```
from torch.nn import L1Loss

# TODO: Train the model

model = SimpleMLP(input_dim=X.shape[1], hidden_dim=16, output_dim=1)
criterion = nn.L1Loss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

trainer = SimpleMLPTrainer(model, criterion, optimizer)
train_losses = trainer.train(train_loader, num_epochs=20)

# TODO: Evaluate the model
validation_loss, validation_accuracy = trainer.evaluate(val_loader)
print(f'Validation Loss: {validation_loss:.4f}, Accuracy:
{validation_accuracy:.2f}%')
```

```
Validation Loss: 0.4058, Accuracy: 60.14%
```

MSELoss

```
from torch.nn import MSELoss

# TODO: Train the model
criterion = nn.MSELoss()
optimizer = Adam(model.parameters(), lr=0.01)
trainer = SimpleMLPTrainer(model, criterion, optimizer)
train_losses = trainer.train(train_loader, num_epochs=20)


# TODO: Evaluate the model

print("\nEvaluating the model on the validation set:")
validation_loss, validation_accuracy = trainer.evaluate(val_loader)

print(f"\nValidation Loss: {validation_loss:.4f}")
print(f"Validation Accuracy: { validation_accuracy:.2f}%")
```

```
Validation Loss: 0.4058
Validation Accuracy: 60.14%
```

NLLLoss

```
# Run with relu activation function
from torch.nn import NLLLoss

criterion = nn.NLLLoss()
optimizer = Adam(model.parameters(), lr=0.01)
trainer = SimpleMLPTrainer(model, criterion, optimizer)
```

```python
# Train the model
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)  # Replace with your dataset
train_losses = trainer.train(train_loader, num_epochs=20)

# Evaluate the model
print("\nEvaluating the model on the validation set:")
validation_loss, validation_accuracy = trainer.evaluate(val_loader)

print(f"\nValidation Loss: {validation_loss:.4f}")
print(f"Validation Accuracy: {validation_accuracy:.2f}%")
```

بخش Regularization in Machine Learning

```python
from sklearn.neural_network import MLPClassifier
```

MLPClassifier را فراخوانی میکنیم
دیتا را لود کرده و به داده تست و ترین جدا میکنیم:

```python
# 1. Load and Prepare the Iris Dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels

# Select only two classes for binary classification (Setosa and Versicolor)
binary_mask = y < 2
X, y = X[binary_mask], y[binary_mask]
# Select two features for 2D visualization (Sepal Length and Petal Length)
X = X[:, [0, 2]]
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

توابع plot_decision_boundary و create_decision_boundary_gif را کامل کرده و gif را در پوشه ذخیره کردم

```python
def plot_decision_boundary(model, X, y, alpha):
    # Define the grid (use meshgrid)
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))
    # Predict over the grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Create a figure
```

```
    fig, ax = plt.subplots(figsize=(6, 5))

    # Plot the decision boundary
    ax.contourf(xx, yy, Z, alpha=0.3, levels=[-0.1, 0.1, 1.1], colors=['blue',
'red'])

    # Scatter plot of the training data
    scatter = ax.scatter(
        X[:, 0], X[:, 1], c=y, cmap='bwr', edgecolor='k', s=50
    )

    # Title and labels
    ax.set_title(f'MLP Decision Boundary (alpha={alpha})')
    ax.set_xlabel('Sepal Length (standardized)')
    ax.set_ylabel('Petal Length (standardized)')

    # Remove axes for clarity
    ax.set_xticks([])
    ax.set_yticks([])

    # Tight layout
    plt.tight_layout()

    # Save the plot to a BytesIO object
    buf = BytesIO()
    plt.savefig(buf, format='png')
    plt.close(fig)
    buf.seek(0)
    return Image.open(buf)
def create_decision_boundary_gif(alpha_values, X_train, y_train, n_neurons):

    # List to store images
    images = []

    for idx, alpha in enumerate(alpha_values):
        print(f"Processing alpha={alpha:.4f} ({idx + 1}/{len(alpha_values)})")

        # Create and train the MLP
        mlp = MLPClassifier(hidden_layer_sizes=(n_neurons,), alpha=alpha,
max_iter=1000, random_state=42)
        mlp.fit(X_train, y_train)

        # Plot decision boundary and get the image
        img = plot_decision_boundary(mlp, X_train, y_train, alpha)
        images.append(img)

    # Save the images as a GIF
    gif_filename = 'mlp_classification_boundaries.gif'
    images[0].save(
        gif_filename,
```

```python
        save_all=True,
        append_images=images[1:],
        duration=500,
        loop=0
    )

    print(f"GIF saved as '{gif_filename}'")

    # return the gif
    return gif_filename
```

```python
# Use np.logspace to generate alpha values, with at least 20 values
alpha_values = np.logspace(-3, 3, 20)  # Range from 0.001 to 1000, with 20 steps
# Define the number of neurons in the hidden layer
n_neurons = 10  # This can be adjusted based on the desired model complexity

# Create the decision boundary GIF
gif_dir = create_decision_boundary_gif(alpha_values, X_train, y_train, n_neurons)
```

```
Processing alpha=0.0010 (1/20)
Processing alpha=0.0021 (2/20)
Processing alpha=0.0043 (3/20)
Processing alpha=0.0089 (4/20)
Processing alpha=0.0183 (5/20)
Processing alpha=0.0379 (6/20)
Processing alpha=0.0785 (7/20)
Processing alpha=0.1624 (8/20)
Processing alpha=0.3360 (9/20)
Processing alpha=0.6952 (10/20)
Processing alpha=1.4384 (11/20)
Processing alpha=2.9764 (12/20)
Processing alpha=6.1585 (13/20)
Processing alpha=12.7427 (14/20)
Processing alpha=26.3665 (15/20)
Processing alpha=54.5559 (16/20)
Processing alpha=112.8838 (17/20)
Processing alpha=233.5721 (18/20)
Processing alpha=483.2930 (19/20)
Processing alpha=1000.0000 (20/20)
GIF saved as 'mlp_classification_boundaries.gif'
```