**Deep Learning HomeWork 02**

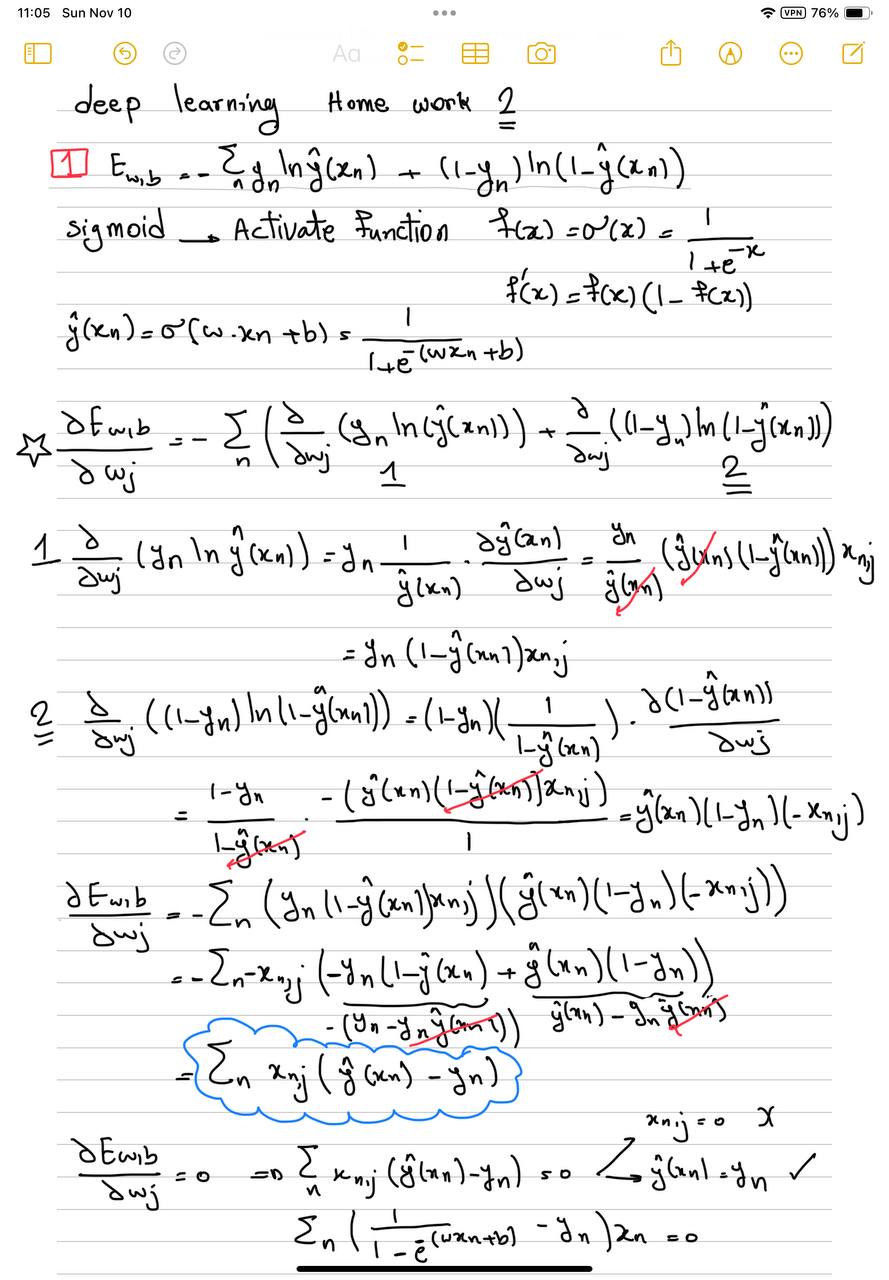
**Mahsa Naseri 402209015**

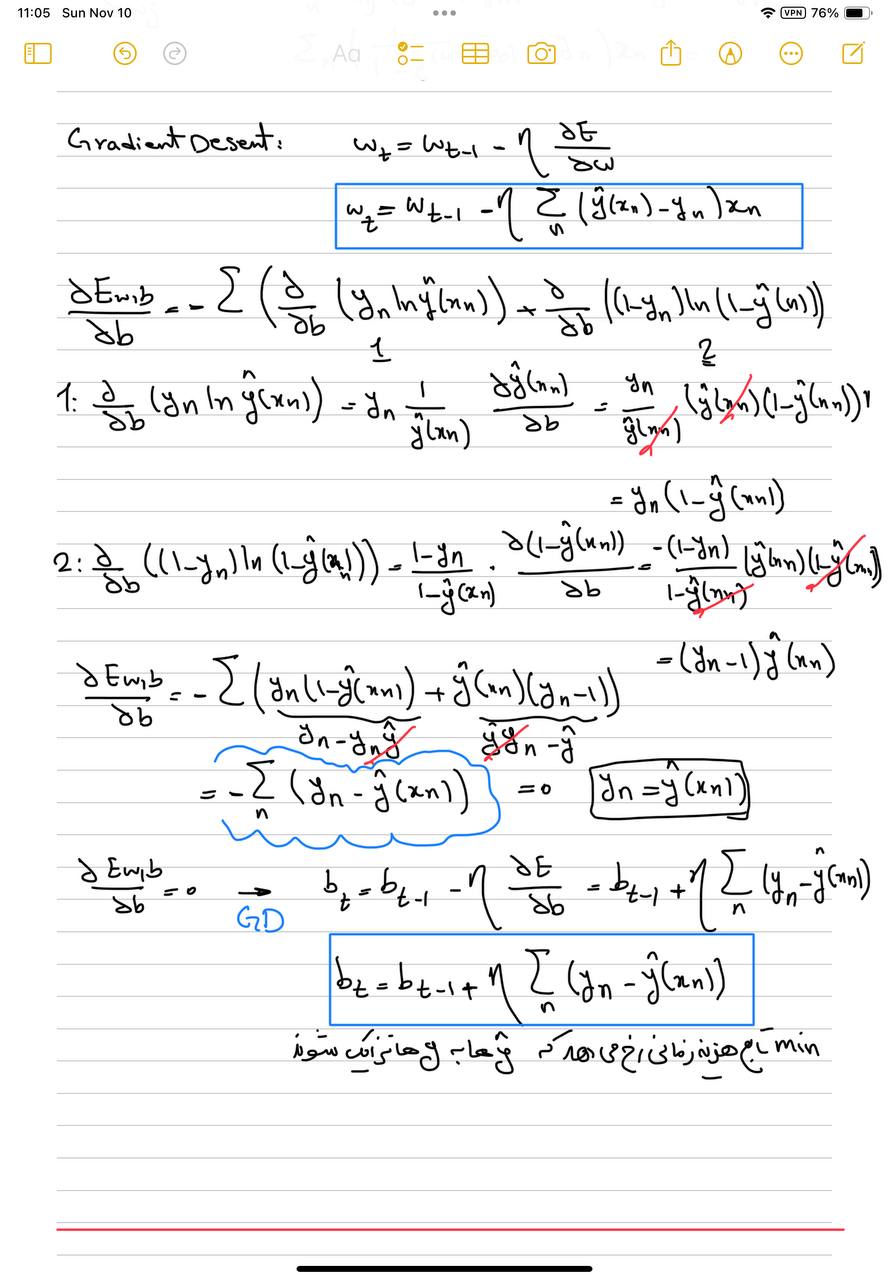
[Github link: https://github.com/mahsanaseri1374/DeepLearning\_HW2\_MahsaNaseri\_402209015](https://github.com/mahsanaseri1374/DeepLearning_HW2_MahsaNaseri_402209015)

[Google drive link: https://drive.google.com/drive/folders/1LhpeUO8tcGSwovNlqIx1e3SonjVImZYa?usp=sharing](https://drive.google.com/drive/folders/1LhpeUO8tcGSwovNlqIx1e3SonjVImZYa?usp=sharing)

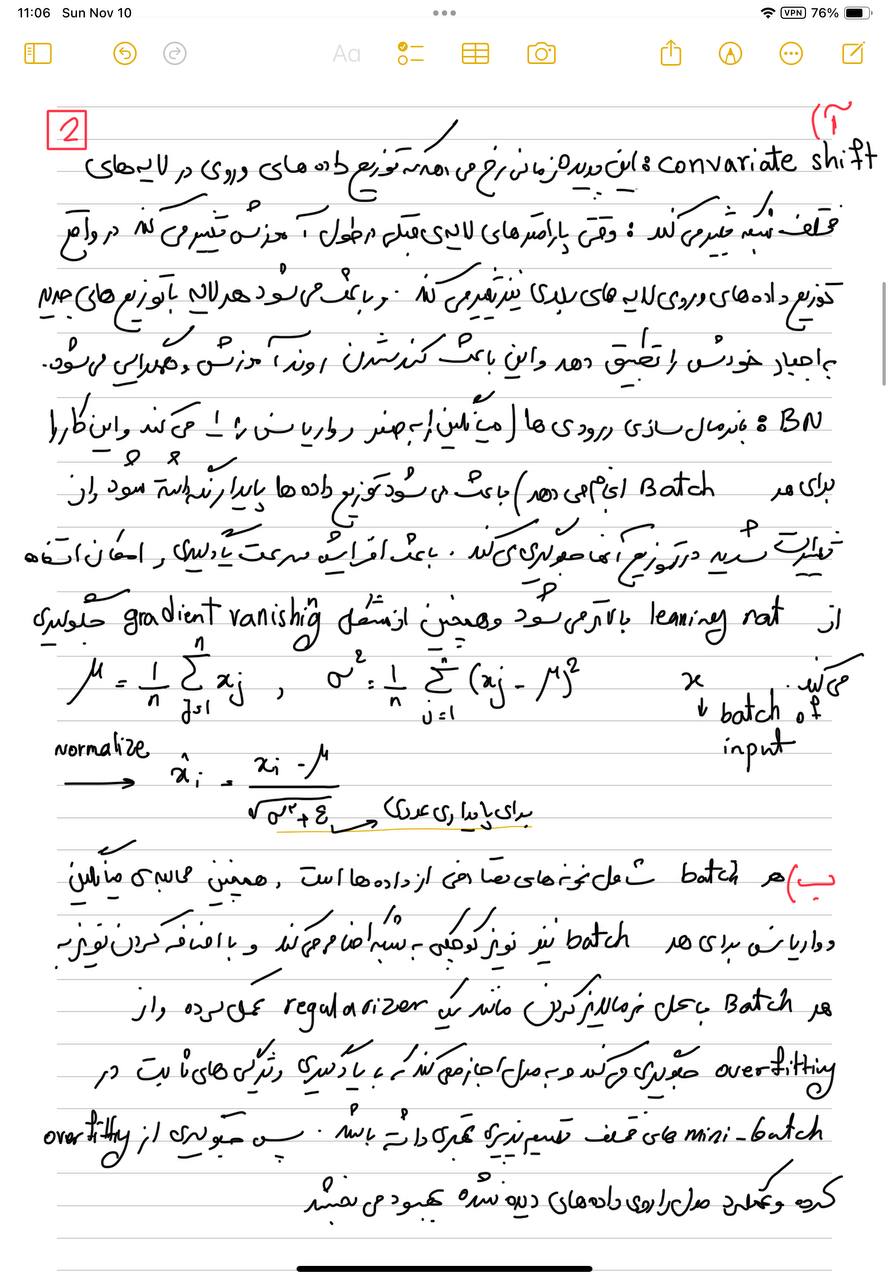
**بخش تئوری**

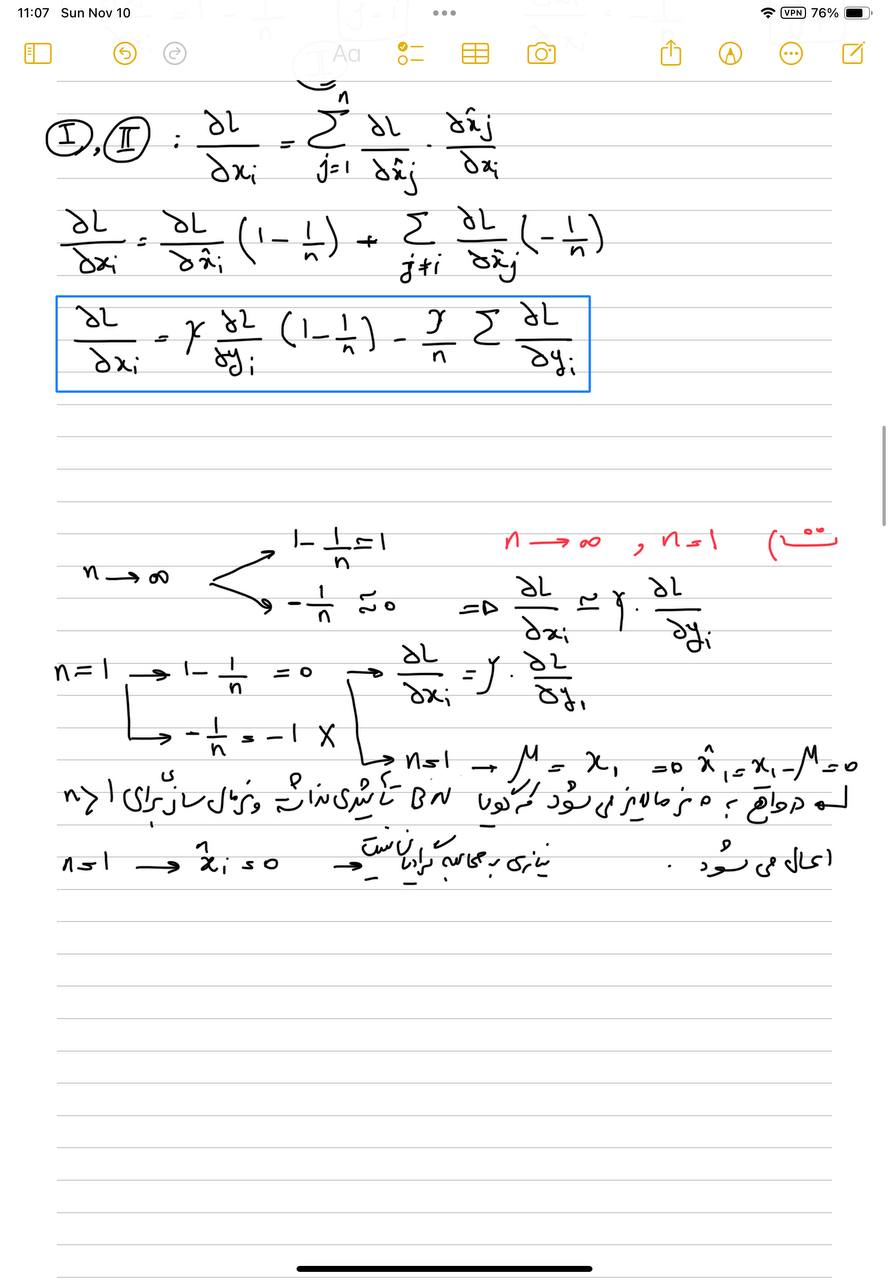
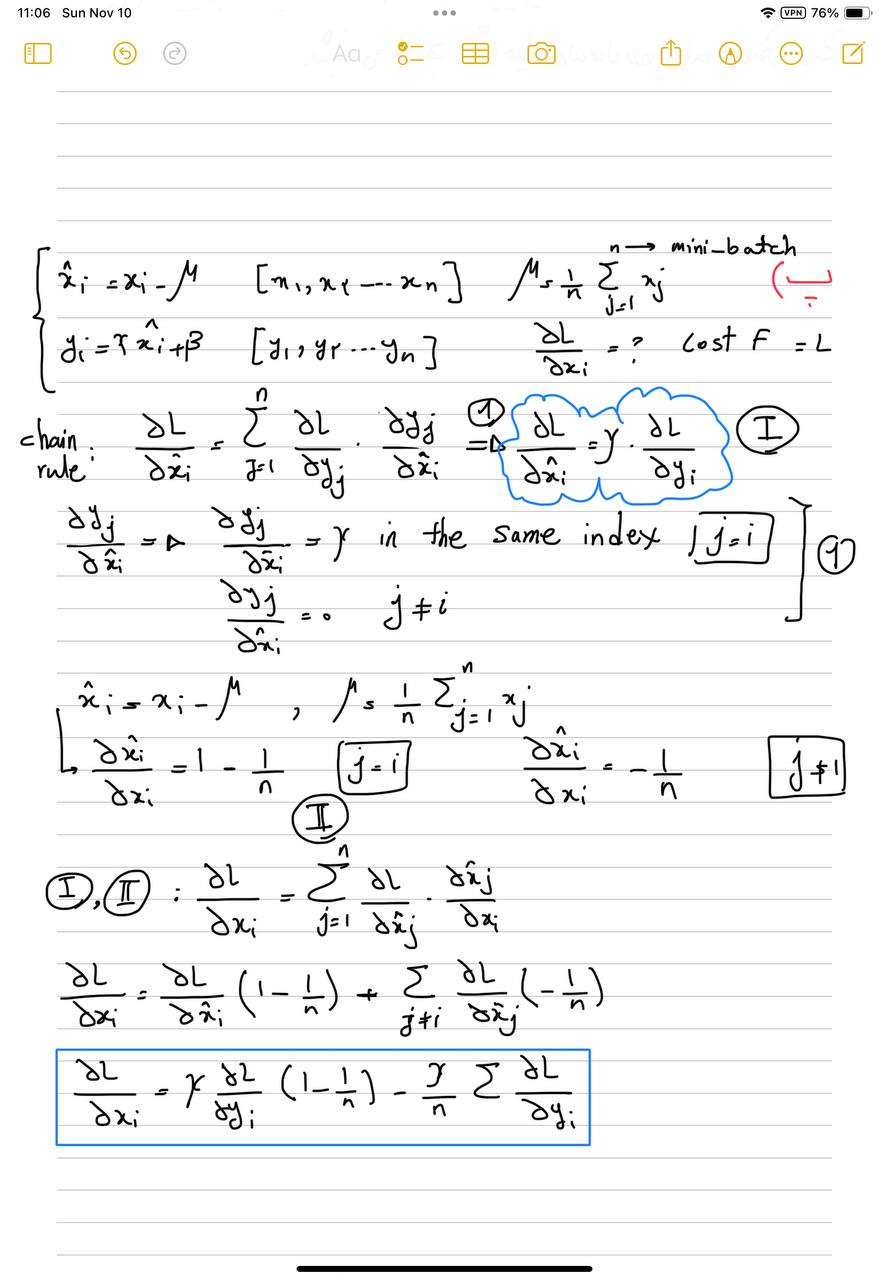
**سوال ۱:**



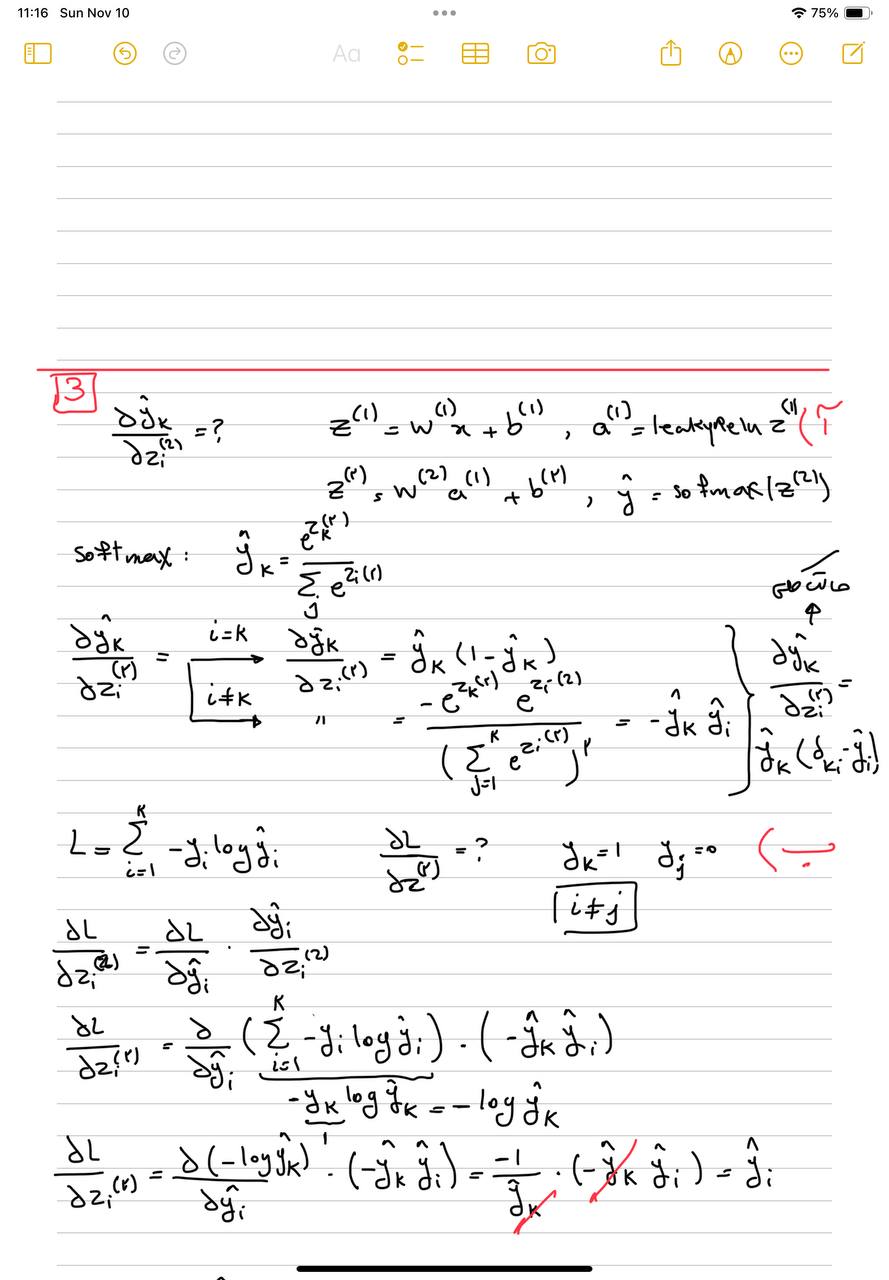


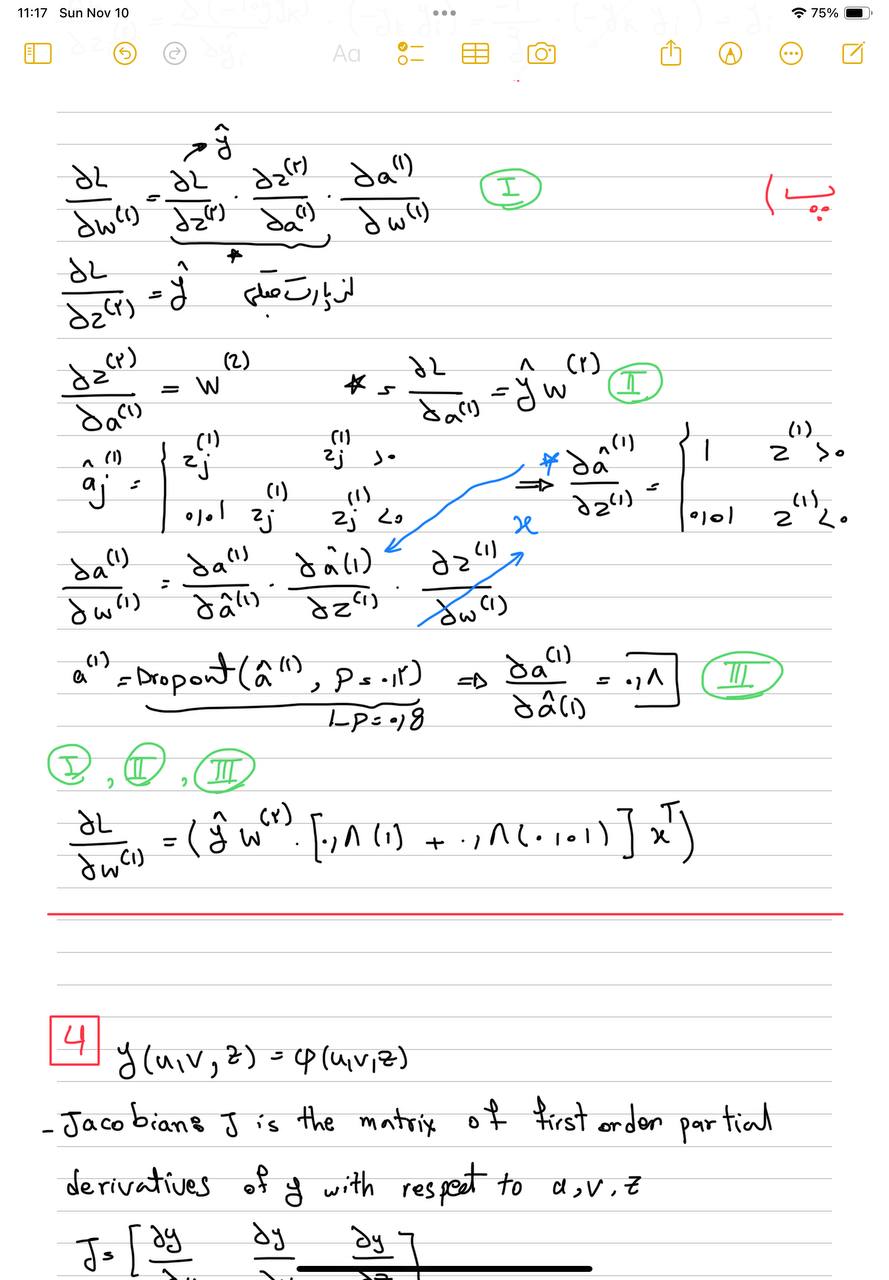
**سوال ۲:**



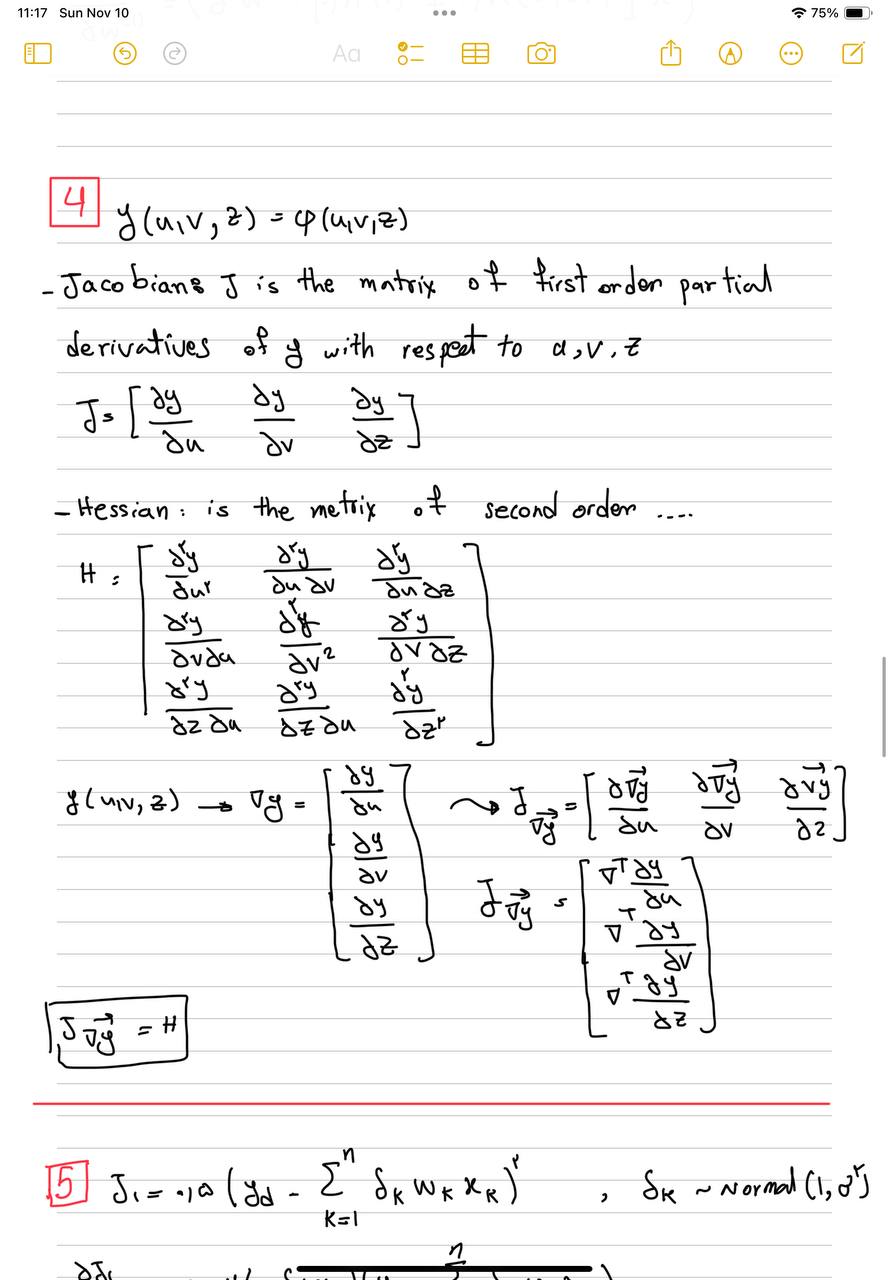


**سوال ۳:**

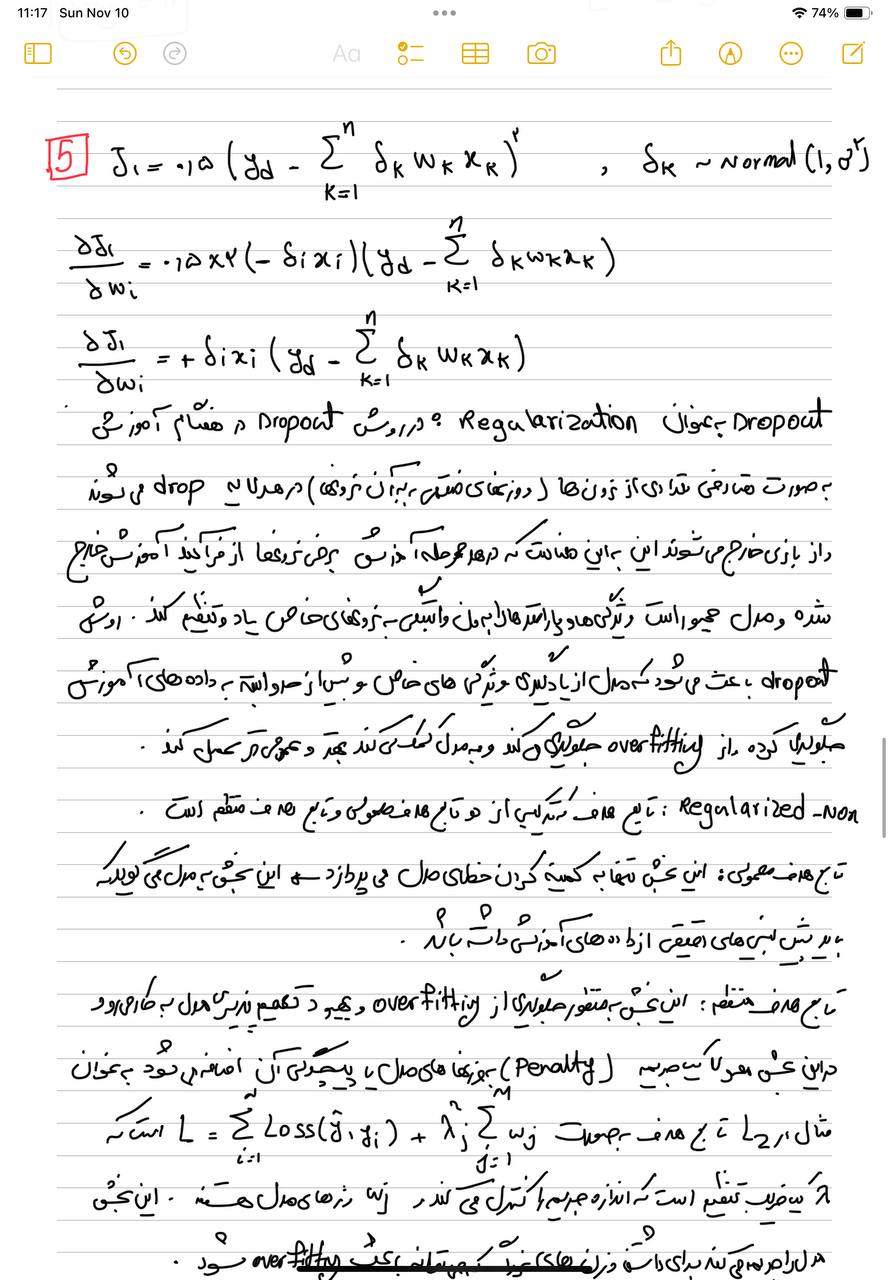


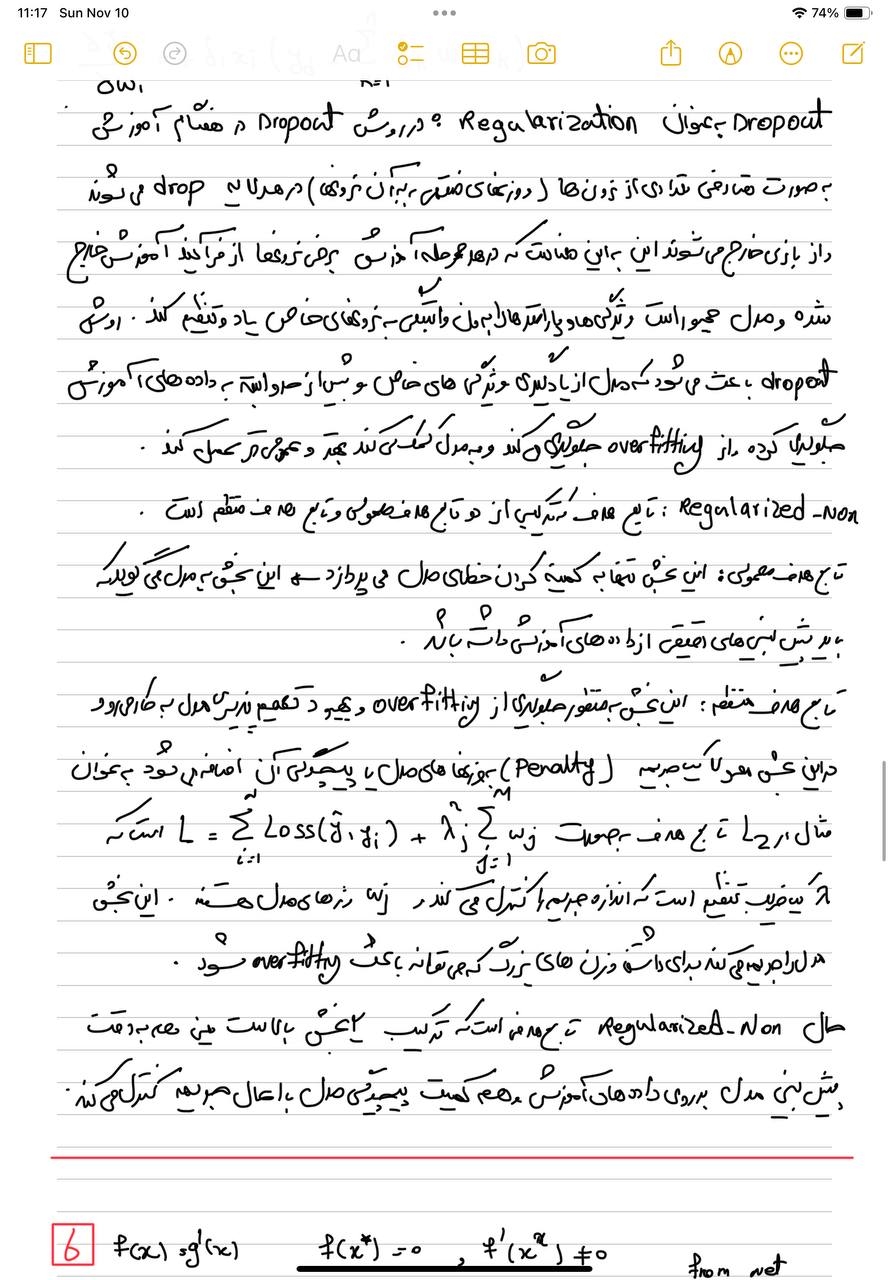


**سوال ۴:**

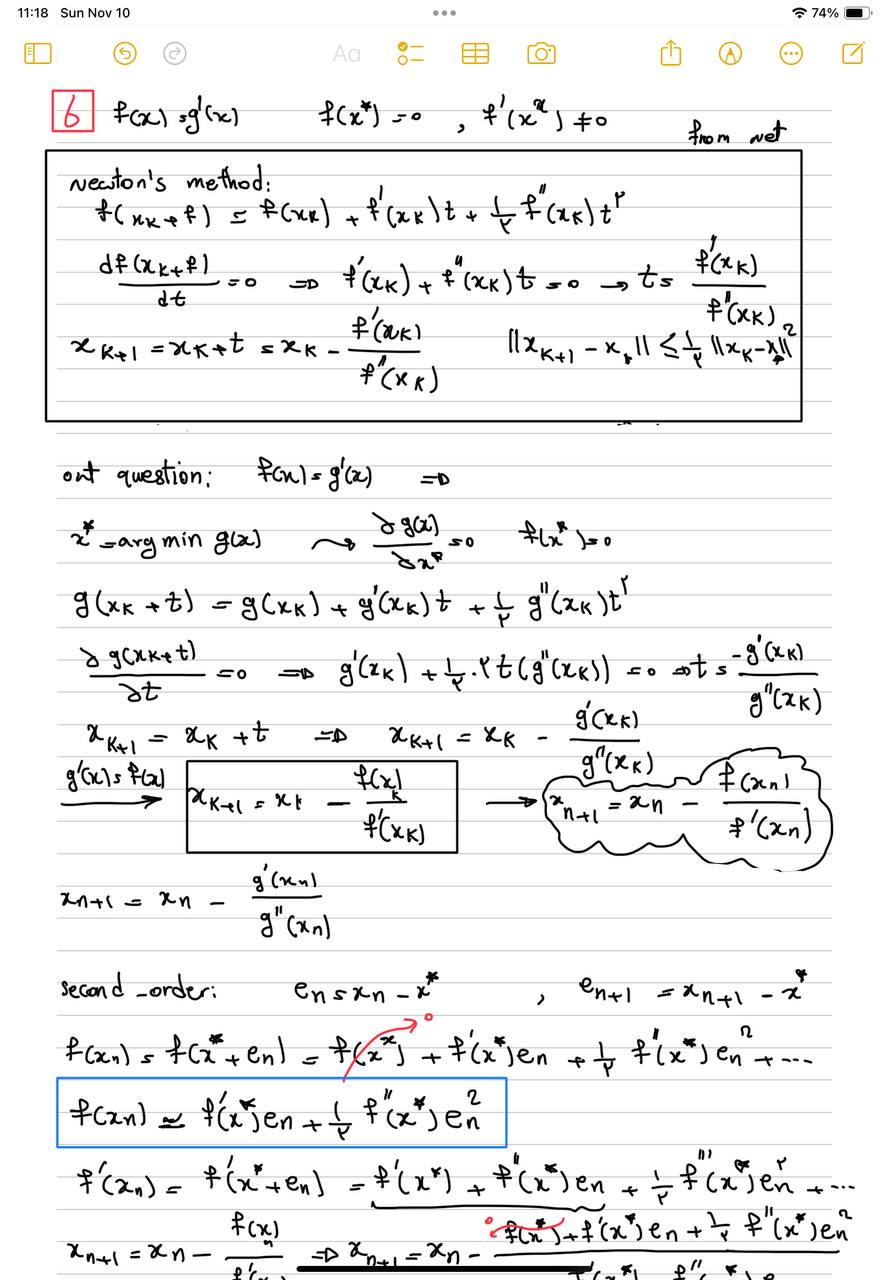


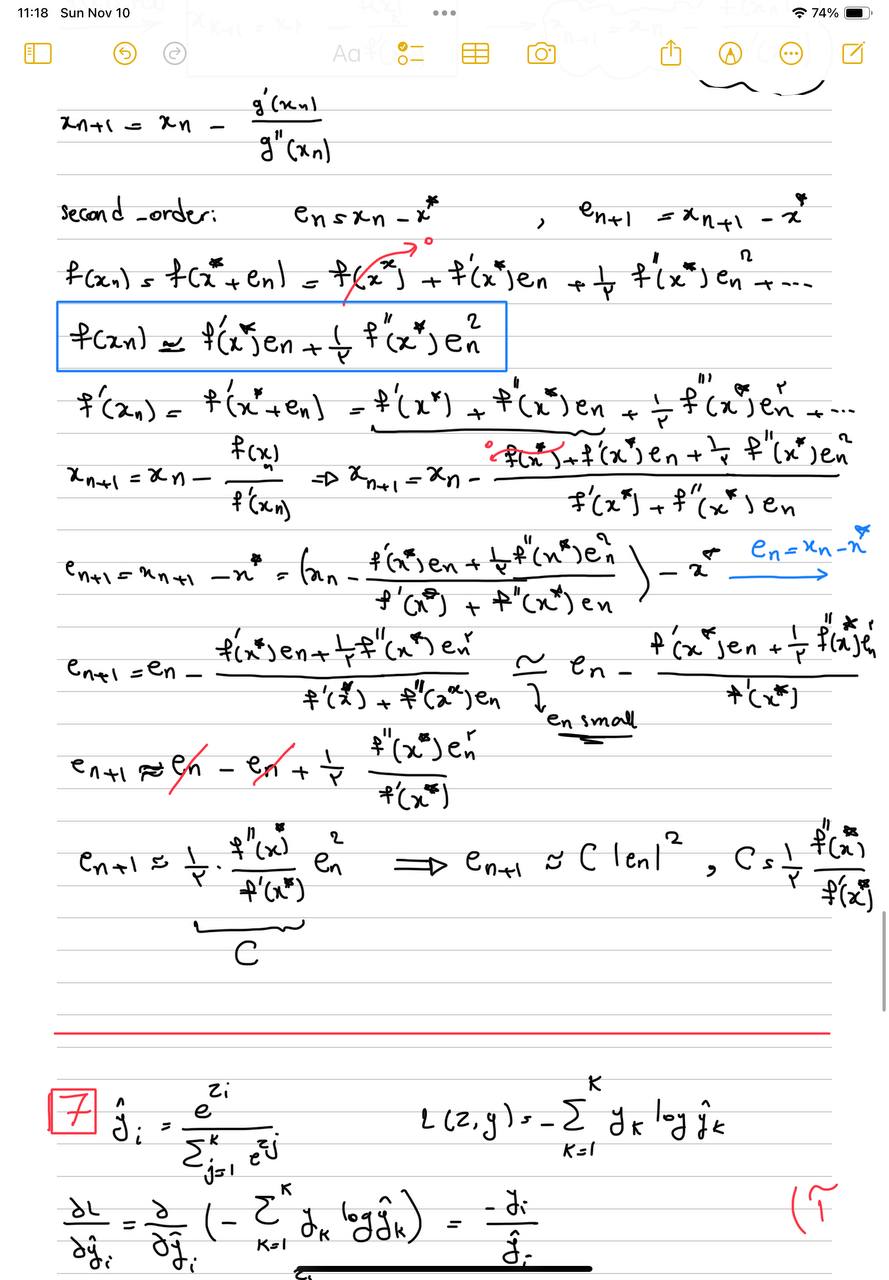
**سوال ۵:**



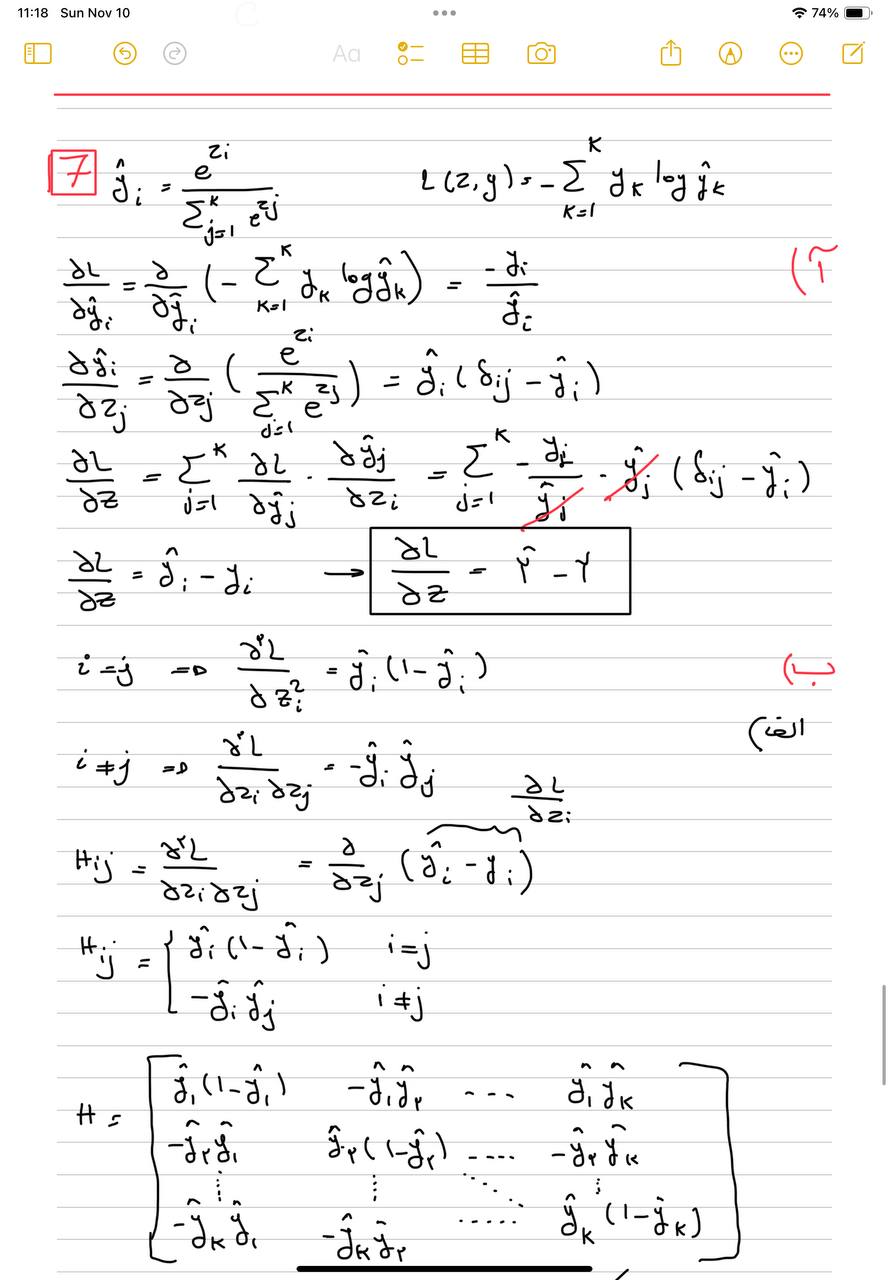


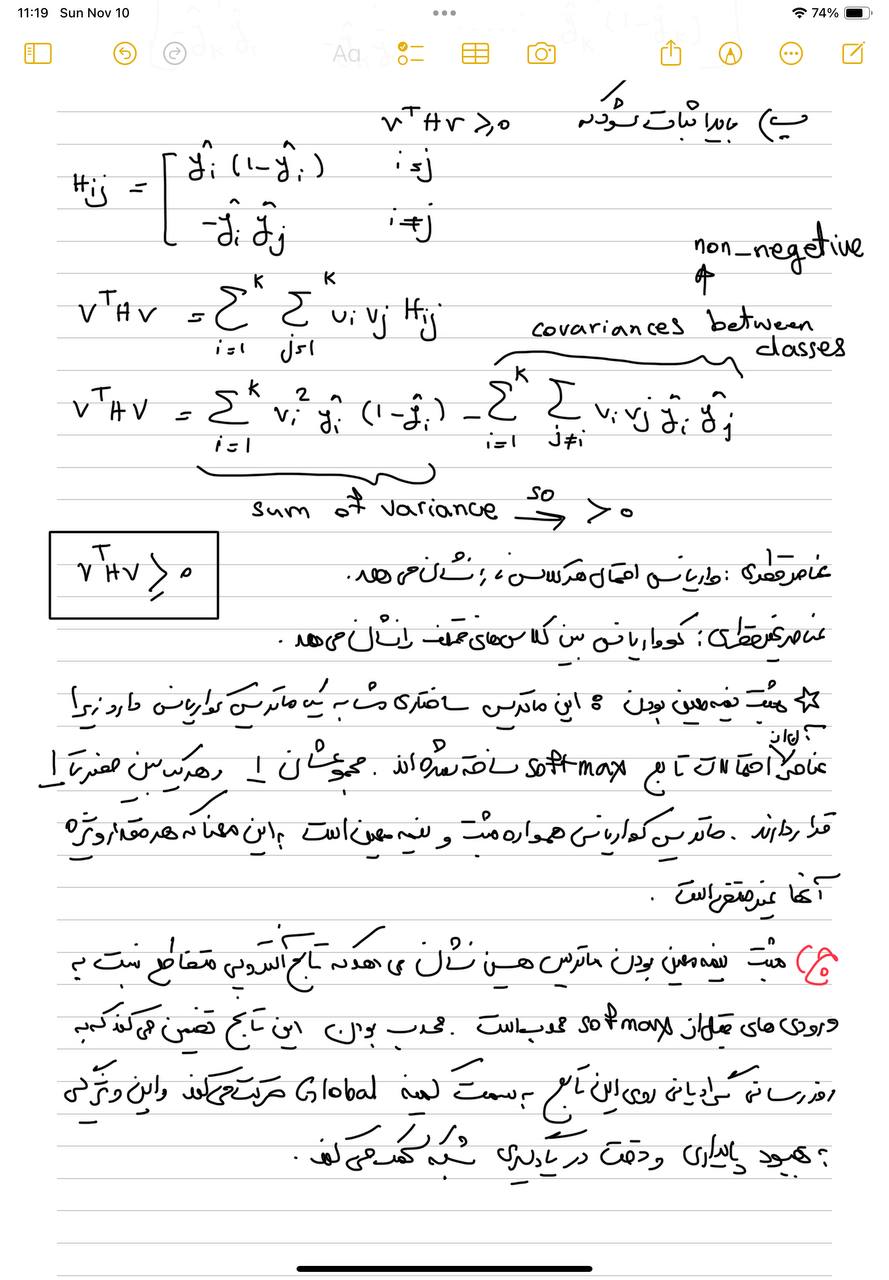
**سوال ۶:**





**سوال ۷:**





**بخش عملی**

[Github link: https://github.com/mahsanaseri1374/DeepLearning\_HW2\_MahsaNaseri\_402209015](https://github.com/mahsanaseri1374/DeepLearning_HW2_MahsaNaseri_402209015)

[Google drive link: https://drive.google.com/drive/folders/1LhpeUO8tcGSwovNlqIx1e3SonjVImZYa?usp=sharing](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")

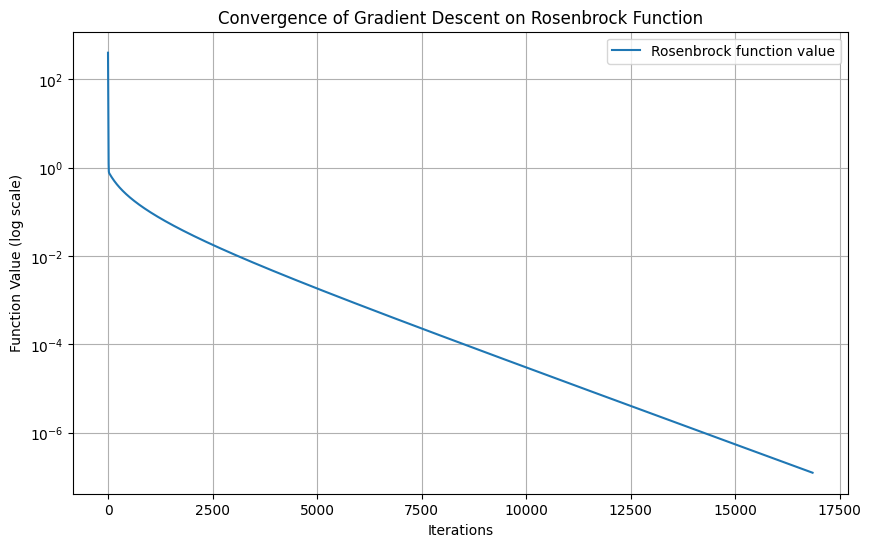
plt.legend()

plt.grid(True)

plt.show()

مقدار اپتیموم [0.99964674 0.99929219] است که به مقدار ۱ و۱ 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 MLP با 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.

1. **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.
2. **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 epoch حدودا 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

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)

# 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

from sklearn.neural\_network import MLPClassifier

MLPClassifier را فراخوانی میکنیم

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

# 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\_boundaryو create\_decision\_boundary\_gif را کامل کرده و gif را در پوشه ذخیره کردم**

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,

save\_all=True,

append\_images=images[1:],

duration=500,

loop=0

)

print(f"GIF saved as '{gif\_filename}'")

# return the gif

return gif\_filename

# 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'