CS2109S Tutorial 6

Neural Networks and Back Propagation

(AY 25/26 Semester 2)

March 19, 2026

(Prepared by Benson)

Contents

Forward Propagation
- Recap
- Q1. Forward Propagation
- Q2. Function Approximation

Back Propagation
- Recap
- Q3. Back Propagation
- Q4. Gradient Flow in Deep Neural Networks

Bonus. Behind Pytorch Autograd (Practical)

Warmup

Warm-Up Exercise: Let $\underset{m\times 1}{\hat{\boldsymbol{y}}} = \underset{m\times n}{\boldsymbol{W}^\top} \underset{n\times 1}{\boldsymbol{x}}$.

Find $\dfrac{\partial \mathcal{E}}{\partial \boldsymbol{x}}$. $n\times 1$

$\boldsymbol{W}$: $n\times m$
$\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}$: $m\times 1$

A. $\boldsymbol{W} \cdot \dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}$ $(n\times m)\cdot(m\times 1)$

B. $\boldsymbol{W}^\top \cdot \dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}$

C. $\dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}} \cdot \boldsymbol{W}$

D. $\dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}} \cdot \boldsymbol{W}^\top$

Find $\dfrac{\partial \mathcal{E}}{\partial \boldsymbol{W}}$. $n\times m$

$\boldsymbol{x}$: $n\times 1$
$\frac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}$: $m\times 1$

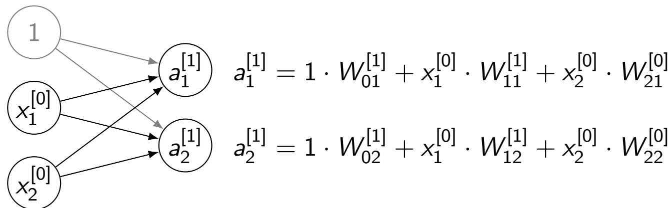
A. $\boldsymbol{x} \cdot \left(\dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}\right)^\top$ $(n\times 1)\cdot(1\times m)$

B. $\boldsymbol{x}^\top \cdot \dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}$

C. $\left(\dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}}\right)^\top \cdot \boldsymbol{x}$

D. $\dfrac{\partial \mathcal{E}}{\partial \hat{\boldsymbol{y}}} \cdot \boldsymbol{x}^\top$

Recap: Forward Propagation



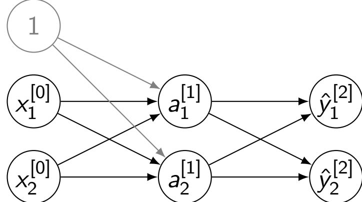
$$\boldsymbol{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \end{bmatrix} = \begin{bmatrix} W_{01}^{[1]} & W_{11}^{[1]} & W_{21}^{[1]} \\ W_{02}^{[1]} & W_{12}^{[1]} & W_{22}^{[1]} \end{bmatrix} \times \begin{bmatrix} 1 \\ x_1^{[0]} \\ x_2^{[0]} \end{bmatrix} = \boldsymbol{W}^\top \times \boldsymbol{x}^{[0]}$$

Q1. Forward Propagation

Suppose there is a data input $\boldsymbol{x} = (2,3)^\top$ and the actual output label is $\boldsymbol{y} = (0.7, 0.4)^\top$. The weights for the network are

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} 0.2 & 0.6 \\ 0.4 & -0.1 \\ -0.5 & 0.2 \end{bmatrix}, \boldsymbol{W}^{[2]} = \begin{bmatrix} 2 & 3 \\ 1 & -1 \end{bmatrix}$$

Calculate $\boldsymbol{a}^{[1]}$, $\hat{\boldsymbol{y}}^{[2]}$ and $L(\hat{\boldsymbol{y}}^{[2]}, \boldsymbol{y})$.



Activation Func:

- [1]: $\text{LeakyReLU}(x) = \max(0.2x, x)$.
- [2]: $\text{ReLU}(x) = \max(0, x)$.

$$\boldsymbol{a}^{[1]} = \text{LeakyReLU}((\boldsymbol{W}^{[1]})^\top \boldsymbol{X}) = \text{LeakyReLU}\left(\begin{bmatrix} 0.2 & 0.4 & -0.5 \\ 0.6 & -0.1 & 0.2 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \text{LeakyReLU}\left(\begin{bmatrix} -0.5 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} -0.1 \\ 1 \end{bmatrix}$$

$$\hat{\boldsymbol{y}}^{[2]} = \text{ReLU}((\boldsymbol{W}^{[2]})^\top \boldsymbol{a}^{[1]}) = \text{ReLU}\left(\begin{bmatrix} 2 & 1 \\ 3 & -1 \end{bmatrix}\begin{bmatrix} -0.1 \\ 1 \end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix} 0.8 \\ -1.3 \end{bmatrix}\right) = \begin{bmatrix} 0.8 \\ 0 \end{bmatrix}$$

$$L(\hat{\boldsymbol{y}}^{[2]}, \boldsymbol{y}) = \frac{1}{2}\left((\hat{y}_1^{[2]} - y_1)^2 + (\hat{y}_2^{[2]} - y_2)^2\right) = \frac{1}{2}\left((0.8 - 0.7)^2 + (0 - 0.4)^2\right) = 0.085$$

Q2. Function Approximation

Can we use a single neuron to model the relationship between x and $f(x)$?

(a) With feature transformation?
- Add a transformed feature $|x - 1|$.
- Then $y = |x - 1|$.

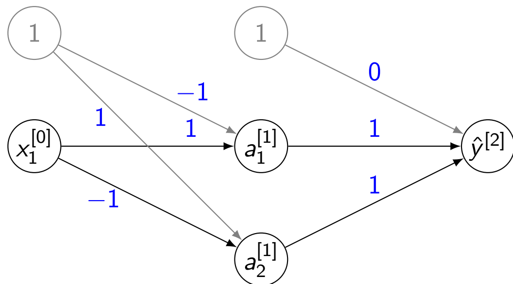
(b) Without feature transformation?
- No, a single neuron can only model linear relationships.

x	$f(x)$
-3	4
-2	3
-1	2
0	1
1	0
2	1
3	2

Q2. Function Approximation

(c) Determine the values of $\boldsymbol{W}^{[1]}$ and $\boldsymbol{W}^{[2]}$ to approximate the function $y = f(x)$.

- $y = |x - 1| = \max(0, x - 1) + \max(0, 1 - x)$



x	$f(x)$
-3	4
-2	3
-1	2
0	1
1	0
2	1
3	2

$$\therefore \boldsymbol{W}^{[1]} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \boldsymbol{W}^{[2]} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Q2. Function Approximation

Conclusion:

- Without non-linear activations, the entire network **collapses to a simple linear model**.

$$\hat{y} = (\boldsymbol{W}^{[L]})^\top \dots (\boldsymbol{W}^{[2]})^\top (\boldsymbol{W}^{[1]})^\top \boldsymbol{x}$$
$$= \left((\boldsymbol{W}^{[L]})^\top \dots (\boldsymbol{W}^{[2]})^\top (\boldsymbol{W}^{[1]})^\top \right) \boldsymbol{x}$$

- Non-linear activation functions let the network model non-linear relationships in the data. Increasing the depth of the network will help the model learn more complex relationships.

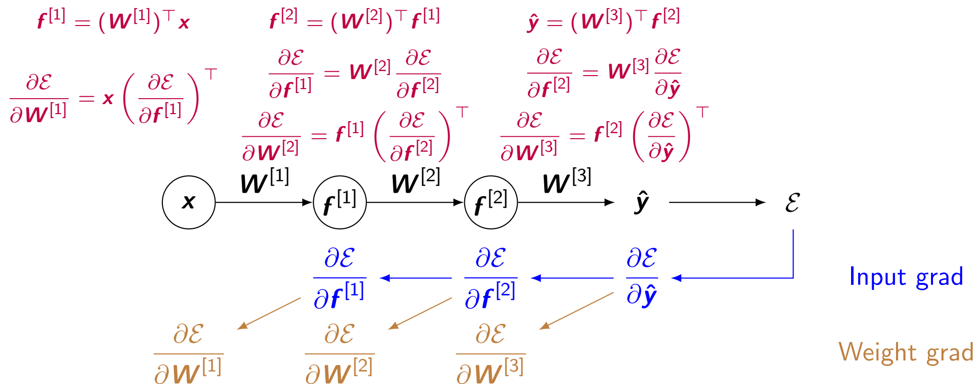
Discussion Question. Universal Approximation

Extra Slide

How "expressive" are neural network models? How many hidden layers are enough?

Recap. Back Propagation

Forward propagation helps us to evaluate $\hat{\boldsymbol{y}}$ in a neural network given $\boldsymbol{x}$. To perform gradient descent, we also need to evaluate $\dfrac{\partial \mathcal{E}}{\partial \boldsymbol{W}^{[i]}}$ for all layers (to update the weights).



Recap. Back Propagation

Suppose a neural network takes time T for inference. What's the closest estimate of the training time using the same data?

A. T

B. $2T$

C. $3T$

D. $4T$

E. None of the other options

Solution: Forward propagation takes time T. Backward propagation requires time $2T$ (2 matrix multiplications are involved for input gradient & weight gradient). Hence the total time is $3T$.

Q3. Back Propagation

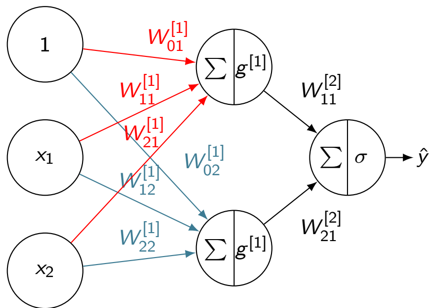
$$L(\hat{y}, y) = -y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

(a) **Show that** $\frac{\partial L(\hat{y},y)}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$.

$\log x \to \frac{1}{x}$ $\quad$ $\frac{\partial}{\partial \hat{y}}(1 - \hat{y})$

$$\frac{\partial \mathcal{E}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \times (-1)\right)$$

$$= -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$



$$\boldsymbol{z}^{[1]} = (\boldsymbol{W}^{[1]})^\top \boldsymbol{x}$$

$$\boldsymbol{a}^{[1]} = g^{[1]}(\boldsymbol{z}^{[1]}) = \text{ReLU}(\boldsymbol{z}^{[1]})$$

$$\boldsymbol{z}^{[2]} = (\boldsymbol{W}^{[2]})^\top \boldsymbol{a}^{[1]}$$

$$\hat{y} = g^{[2]}(\boldsymbol{z}^{[2]}) = \sigma(\boldsymbol{z}^{[2]})$$

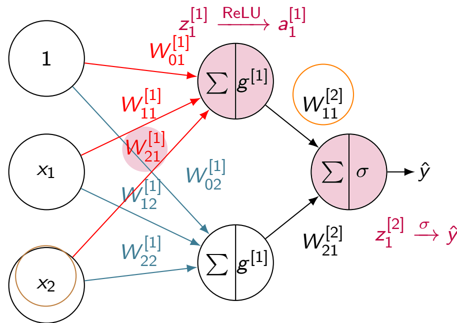
Q3. Back Propagation

Compute $\frac{\partial L(\hat{y},y)}{\partial W_{21}^{[1]}}$.

$$\frac{\partial L(\hat{y},y)}{\partial W_{21}^{[1]}} = \underbrace{\frac{\partial L(\hat{y},y)}{\partial \hat{y}}}_{\text{from (a)}} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z_0^{[2]}}}_{\sigma'(x)} \cdot \frac{\partial z_0^{[2]}}{\partial a_1^{[1]}} \cdot \underbrace{\frac{\partial a_1^{[1]}}{\partial z_1^{[1]}}}_{\text{ReLU}'(x)} \cdot \frac{\partial z_1^{[1]}}{\partial W_{21}^{[1]}}$$

$$= \left[-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right] \cdot \hat{y}(1-\hat{y}) \cdot W_{11}^{[2]} \cdot \left(\begin{cases} 1 & \text{if } z_1^{[1]} > 0 \\ 0 & \text{otherwise} \end{cases}\right) \cdot x_2$$

$$= \begin{cases} (\hat{y}-y)W_{11}^{[2]}x_2 & \text{if } z_1^{[1]} > 0 \\ 0 & \text{otherwise} \end{cases}$$



$$\boldsymbol{z}^{[1]} = (\boldsymbol{W}^{[1]})^\top \boldsymbol{x}$$
$$\boldsymbol{a}^{[1]} = g^{[1]}(\boldsymbol{z}^{[1]}) = \text{ReLU}(\boldsymbol{z}^{[1]})$$
$$\boldsymbol{z}^{[2]} = (\boldsymbol{W}^{[2]})^\top \boldsymbol{a}^{[1]}$$
$$\hat{y} = g^{[2]}(\boldsymbol{z}^{[2]}) = \sigma(\boldsymbol{z}^{[2]})$$

Q3. Back Propagation

(d) The 1100 samples consist of 100 samples from spam emails and 1000 samples from non-spam emails. To deal with the imbalanced in the dataset, the system introduces two hyperparameters, α and β, in the loss function:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\frac{1}{n}\sum_{i=1}^{n}\left[\alpha(y^{(i)} \cdot \log(\hat{y}^{(i)})) + \beta((1-y^{(i)})\log(1-\hat{y}^{(i)}))\right]$$

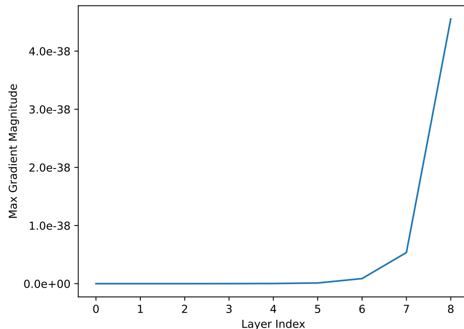
Why do you think that she introduced the hyper-parameters α and β? How should she set their values?

- We would like the weight of cultivar A $\approx$ the weight of cultivar B.
- $\alpha = 10\beta$ (since the dataset has 10 times more samples from cultivar B than those from cultivar A).

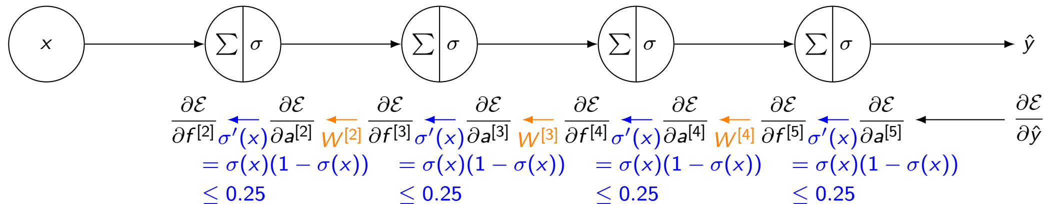
Q4. Gradient Flow in Deep Neural Networks

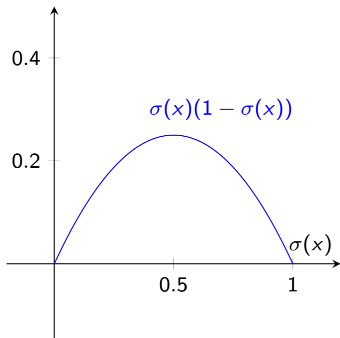
Suppose we use σ (the sigmoid function) as our activation function in a neural network with 50 hidden layers, as per the code in the accompanying Python notebook.

(a) Play around with the code. Notice that when performing back propagation, the gradient magnitudes of the first few layers are extremely small. What do you think causes this problem?



Q4. Gradient Flow in Deep Neural Networks





Vanishing gradient!

Issue: Convergence will be slow.

Q4. Gradient Flow in Deep Neural Networks

Suppose we use σ (the sigmoid function) as our activation function in a neural network with 50 hidden layers, as per the code in the accompanying Python notebook.

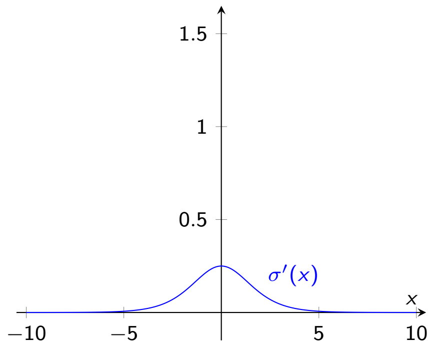
(b) Based on what we have learnt thus far, how can we **mitigate** this problem? Test out your solution by modifying the code and checking the gradient magnitudes.

Q4. Gradient Flow in Deep Neural Networks

Sigmoid Function:

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

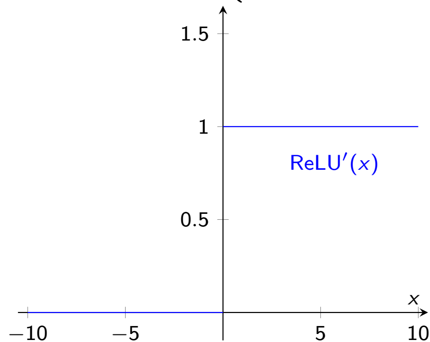
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



ReLU Function:

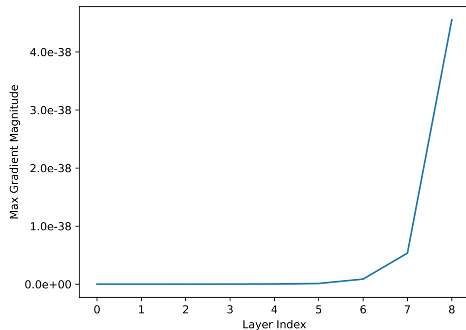
$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

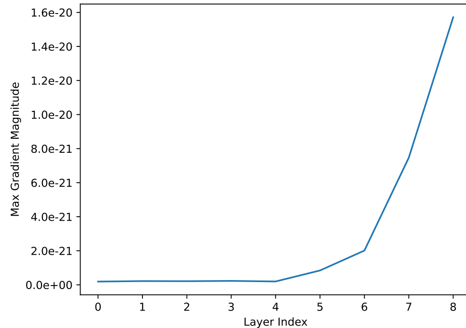


Q4. Gradient Flow in Deep Neural Networks

Sigmoid Function:

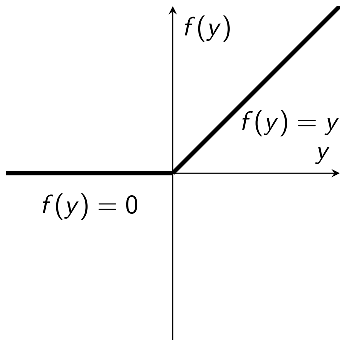


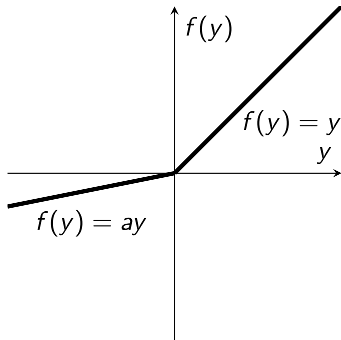
ReLU Function:



Q4. Gradient Flow in Deep Neural Networks

This problem occurs when majority of the activations are 0 (meaning the underlying pre-activations are mostly non-positive), resulting in the network dying midway. The gradients passed back are also 0 which leads to poor gradient descent performance and hence poor learning. Refer to the figure below on the ReLU and Leaky ReLU activation functions.





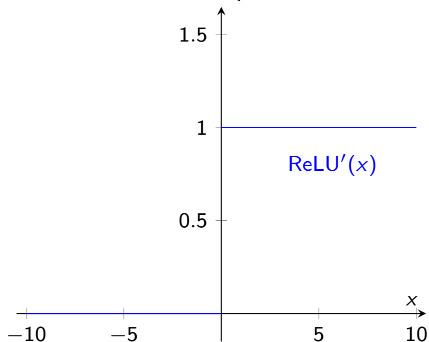
How does `Leaky ReLU` fix this? What happens if we set $a = 1$ in `Leaky ReLU`?

Q4. Gradient Flow in Deep Neural Networks

ReLU Function:

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

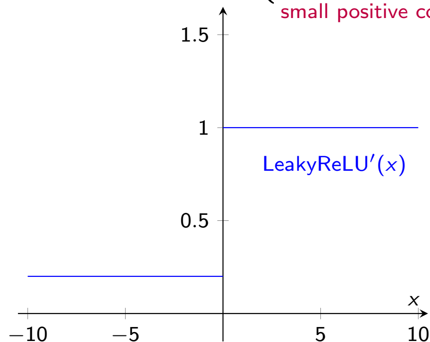


Leaky ReLU Function:

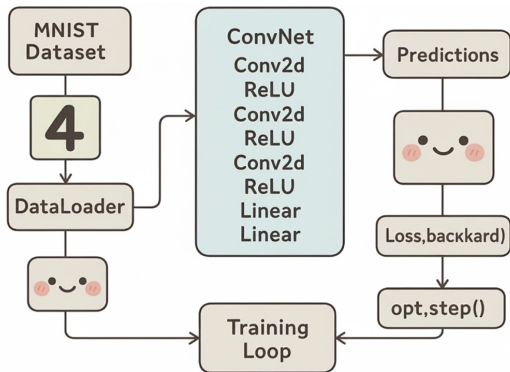
$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ a & \text{otherwise} \end{cases}$$

small positive constant



PyTorch: Combining Everything




```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.c1 = nn.Conv2d(1, 16, 3, 1)
        self.c2 = nn.Conv2d(16, 32, 3, 1)
        self.c3 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(64*2*2, 128)
        self.fc2 = nn.Linear(128, 10)
        self.pool = nn.MaxPool2d(2)

    def forward(self, x):
        x = self.pool(F.relu(self.c1(x)))
        x = self.pool(F.relu(self.c2(x)))
        x = self.pool(F.relu(self.c3(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        return self.fc2(x)

# Training
opt = torch.optim.Adam(model.parameters())
loss_fn = nn.CrossEntropyLoss()

for epoch in range(1):
    for xb, yb in train:
        pred = model(xb)
        loss = loss_fn(pred, yb)
        opt.zero_grad()
        loss.backward()
        opt.step()
    print(f"Loss: {loss.item():.4f}")
```

Bonus. Behind Pytorch Autograd (Practical)

Extra Slide

Copy the template code from the website.

Complete the function `visualize_autograd_graph(loss)` to print out the Pytorch autograd graph. Get started by reading the provided starter code and the comments.

```python
# Sample neural network with 2 linear layers
class TwoLayerNet(nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = nn.Linear(D_in, H)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(H, D_out)

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x

model = TwoLayerNet(1000, 100, 10)
loss = nn.MSELoss()
```