

CS5242 : Neural Networks and Deep Learning

Lecture 5: Multi-Layer Perceptron Inference and Learning

Semester 1 2021/22

Xavier Bresson

<https://twitter.com/xbresson>

Department of Computer Science
National University of Singapore (NUS)



Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

Outline

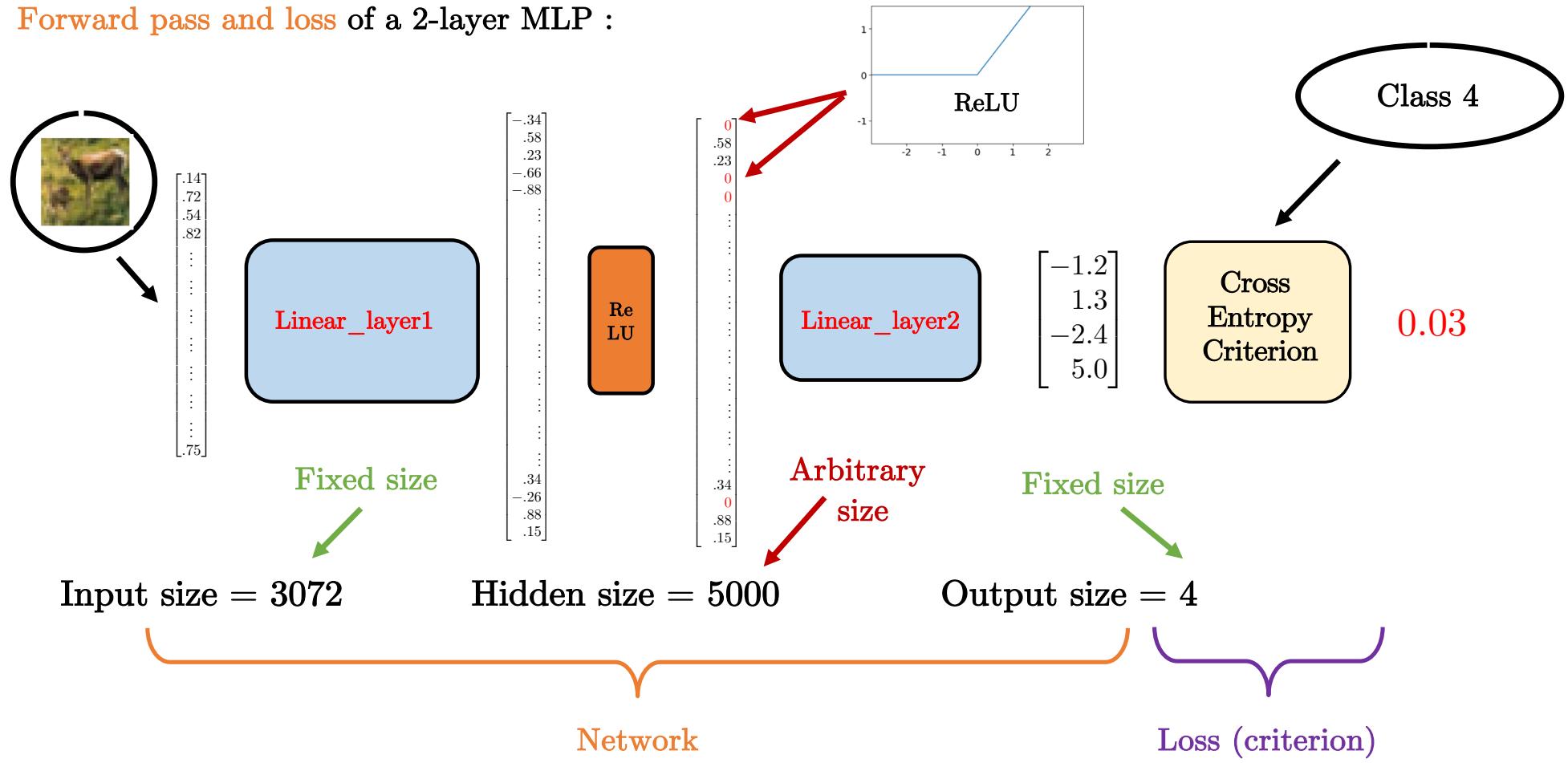
- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

MLP

- Introduced in Frank Rosenblatt in 1957.
- What is a Multi-Layer Perceptron (MLP)?
 - A MLP is the simplest feedforward neural networks (no cycle connection).
 - A MLP is composed by a series of linear layers + non-linear activations.
 - A MLP is a.k.a. Fully Connected (FC) neural networks.
- Example: Two-layer MLP
 - Consider a two-layer network with only 4 classes:
Cat, Dog, Horse, Deer

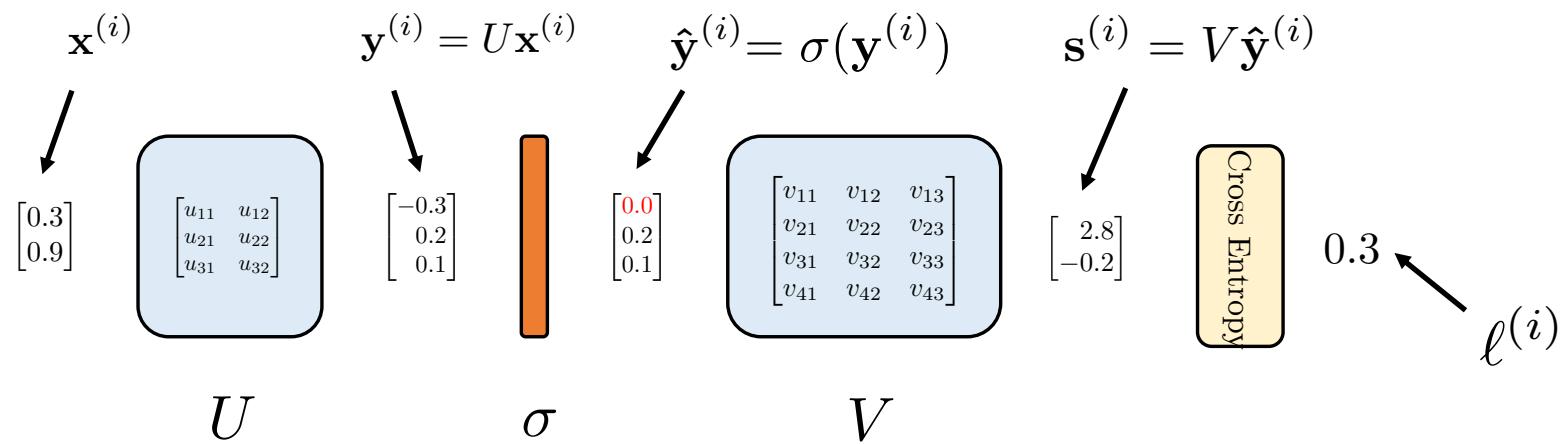
Two-layer MLP

- Forward pass and loss of a 2-layer MLP :



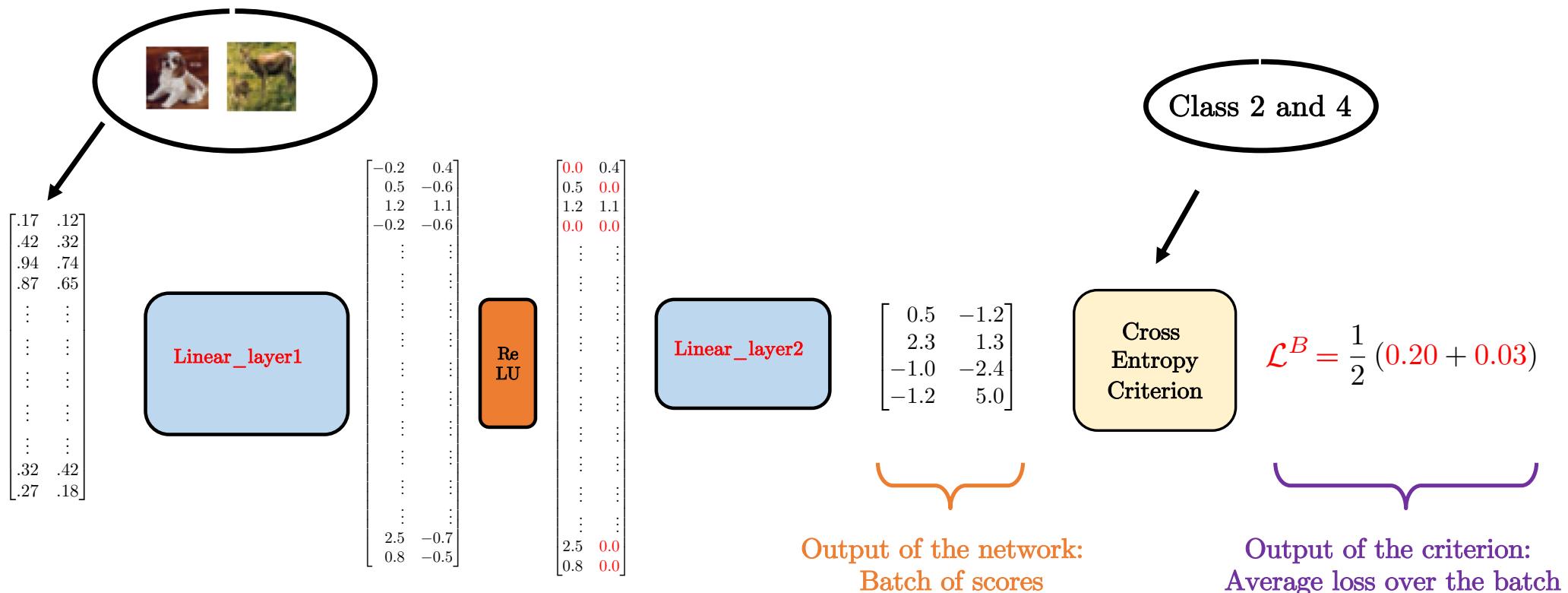
Two-layer MLP

- Forward pass and loss of a 2-layer MLP with mathematical variables :



Batch mode

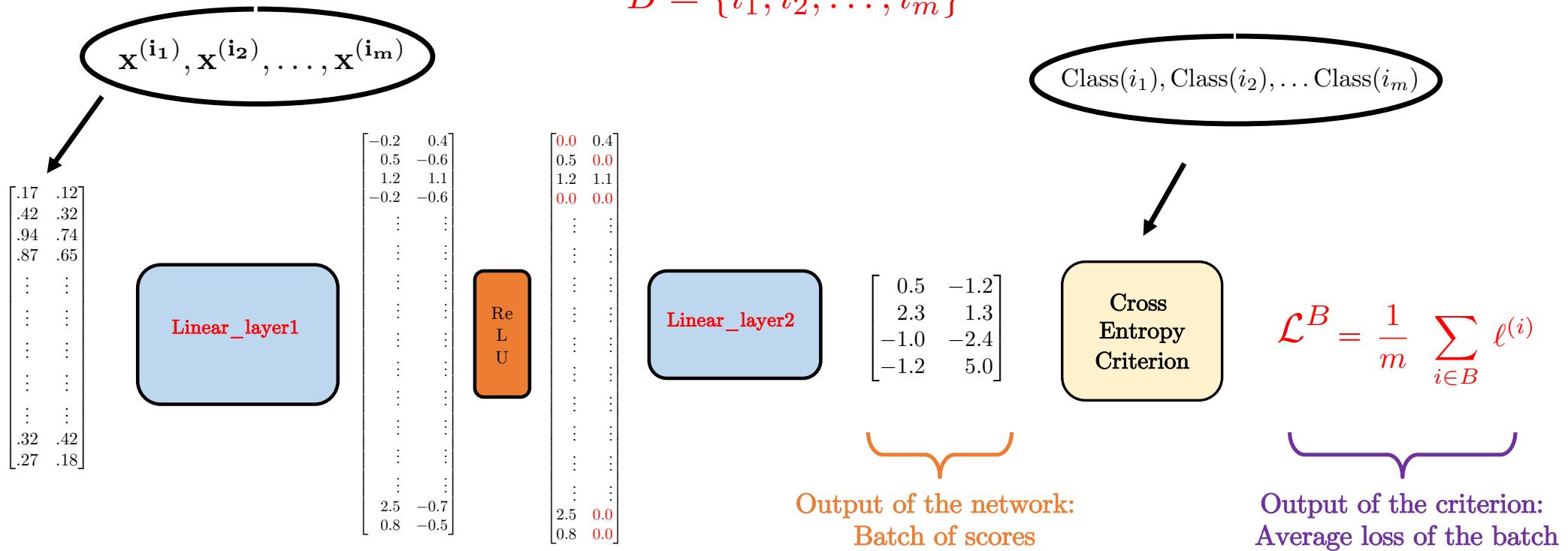
- Forward pass and loss of a 2-layer MLP for two input images :



Batch mode

- Batch size is arbitrary :
 - Sample randomly B indices from the training set :

$$B = \{i_1, i_2, \dots, i_m\}$$

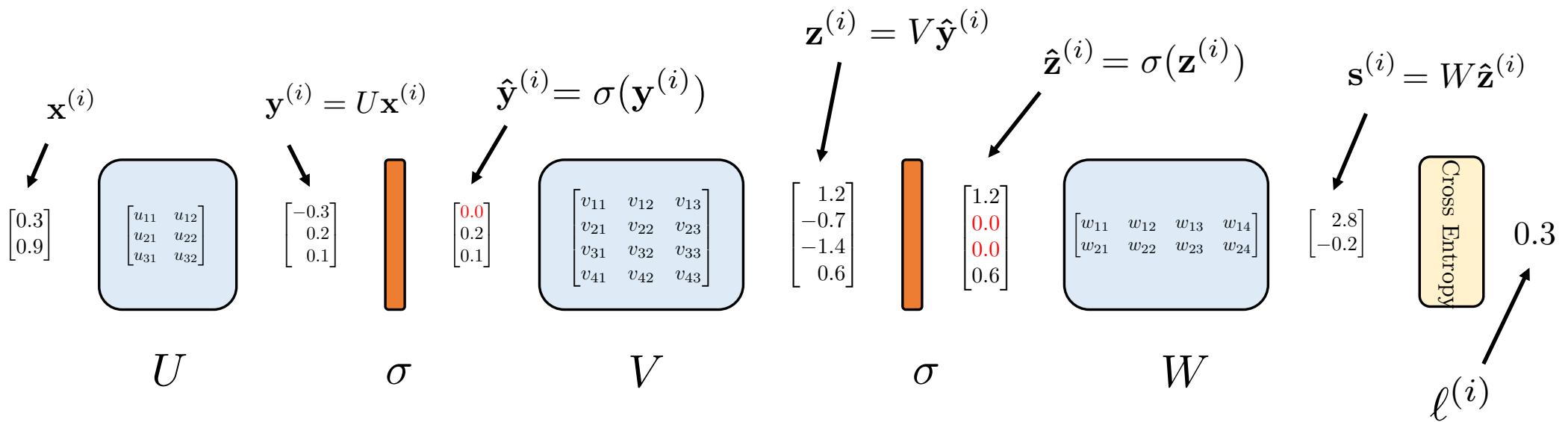


Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

Three-layer MLP

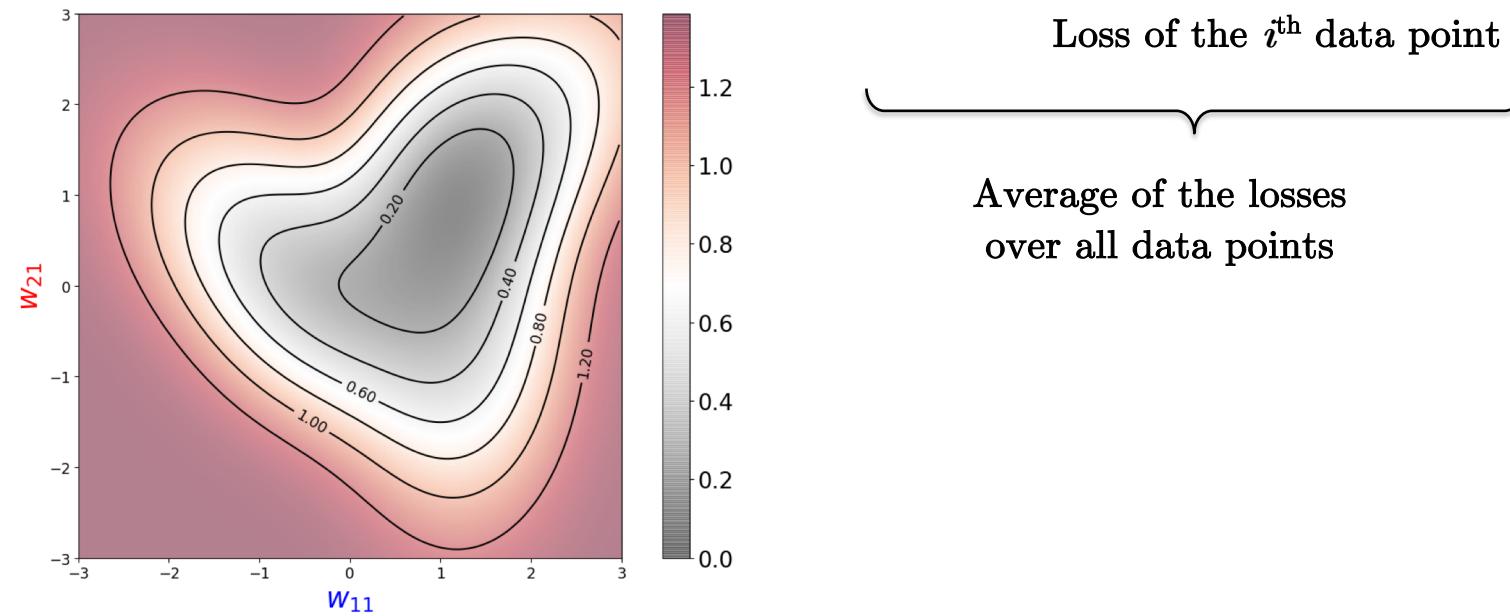
- We consider a **three-layer MLP** :



MLP loss

- We want to find the parameters that minimize the cross-entropy loss:

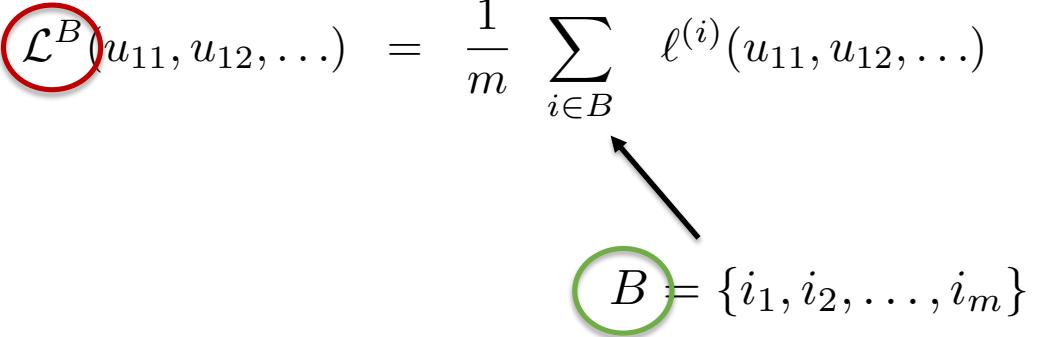
$$\mathcal{L}(u_{11}, u_{12}, \dots) = \frac{1}{N} \sum_{i=1}^N \ell^{(i)}(u_{11}, u_{12}, \dots)$$



Mini-batch MLP loss

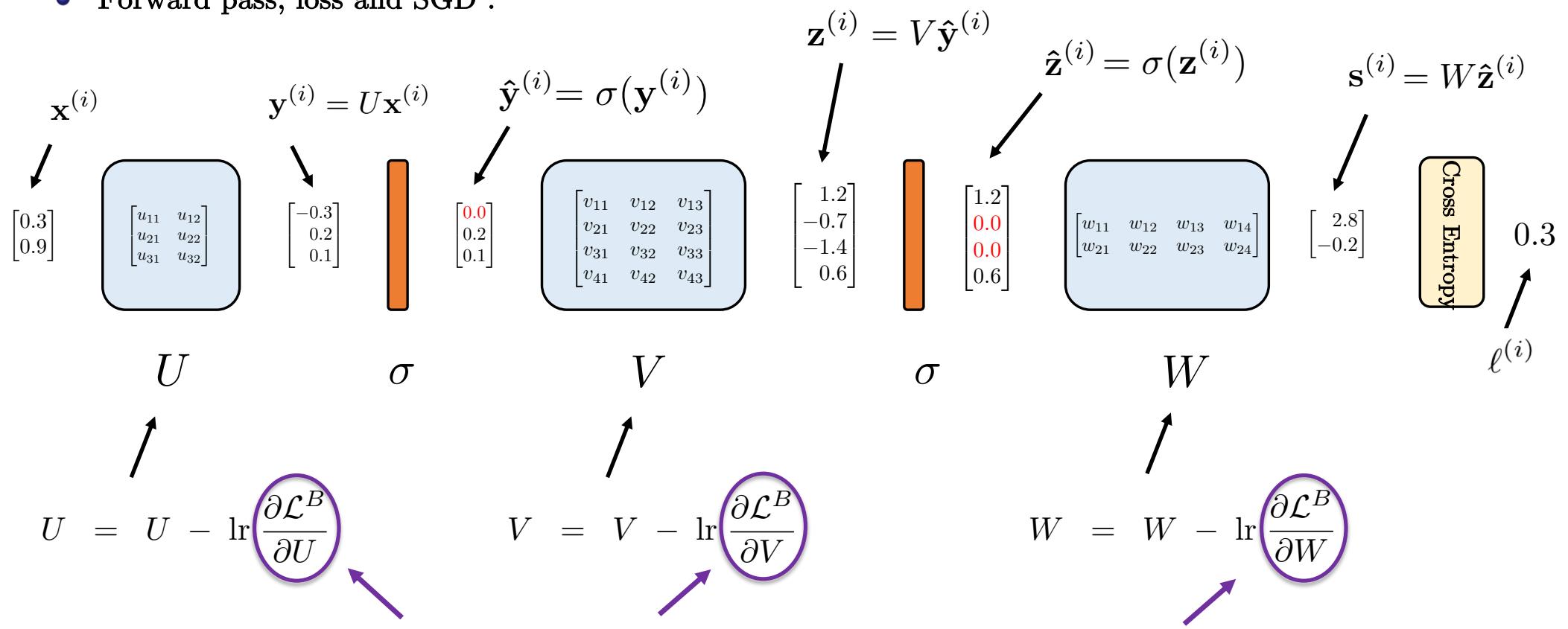
- Approximated mini-batch loss L^B :
 - Average of the losses of the data points in each batch B :

$$\mathcal{L}^B(u_{11}, u_{12}, \dots) = \frac{1}{m} \sum_{i \in B} \ell^{(i)}(u_{11}, u_{12}, \dots)$$


Mini-batch B

Stochastic Gradient Descent

- Forward pass, loss and SGD :



Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- **Backpropagation**
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

Backpropagation

- Backpropagation:

- Forward pass: Compute the average of the losses \mathcal{L}^B of the data points in the batch by forwarding the batch through the net.
- Backward pass: Get the derivatives of \mathcal{L}^B w.r.t. the net parameters by back-propagating the loss value (error of prediction) through the net.
- Update weight: Use the SGD formula to get the new weight values that make the network (a bit) better at prediction:

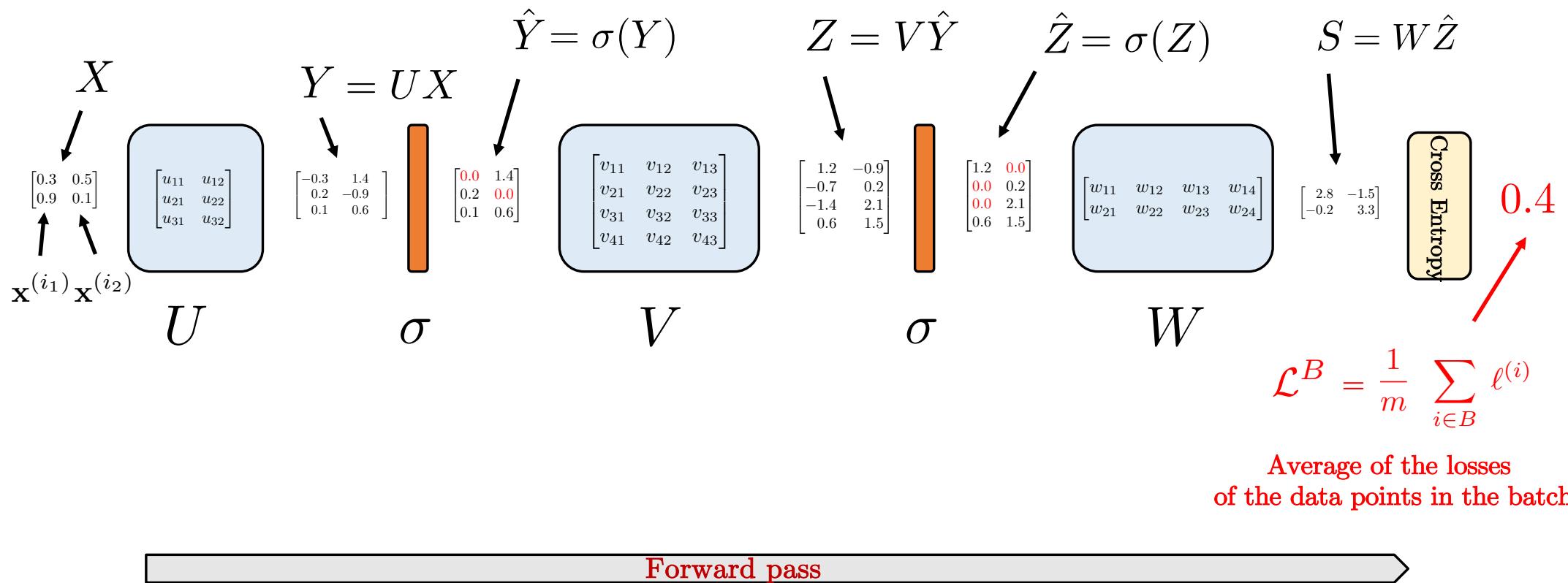
$$U = U - \text{lr} \frac{\partial \mathcal{L}^B}{\partial U}$$

$$V = V - \text{lr} \frac{\partial \mathcal{L}^B}{\partial V}$$

$$W = W - \text{lr} \frac{\partial \mathcal{L}^B}{\partial W}$$

Forward pass

- Forward pass and loss :



Backward pass

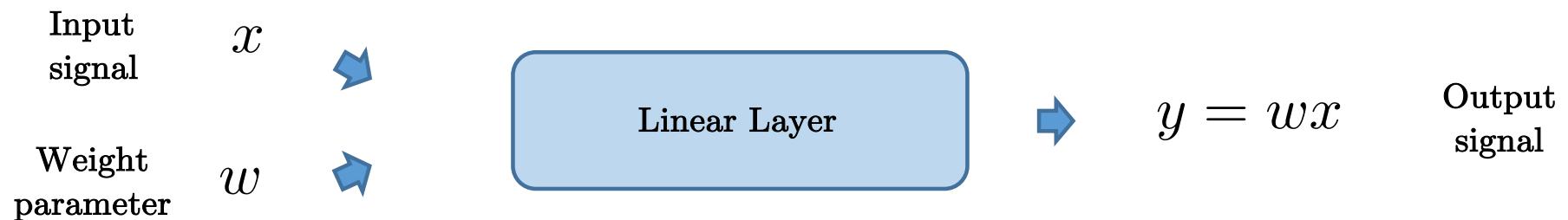
- How do we compute the gradients of the loss w.r.t. network parameters?

$$\frac{\partial \mathcal{L}^B}{\partial U} \quad \frac{\partial \mathcal{L}^B}{\partial V} \quad \text{and} \quad \frac{\partial \mathcal{L}^B}{\partial W}$$

- Let us introduce the backpropagation formula for a linear layer (that performs pattern matching).
- Understanding the backpropagation for a linear layer is enough to generalize the backpropagation process to any layer, and any network !

Backpropagation for linear layer – scalar case

- Forward pass:



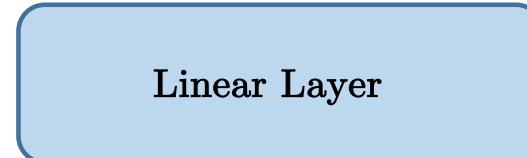
- Backward pass:

$$\frac{\partial \mathcal{L}^B}{\partial x} = \frac{\partial \mathcal{L}^B}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Chain rule on $\mathcal{L}^B(y(w, x))$!

Output gradients

$$= e.w$$



$$\frac{\partial \mathcal{L}^B}{\partial w} = \frac{\partial \mathcal{L}^B}{\partial y} \cdot \frac{\partial y}{\partial w}$$

$$= e.x$$

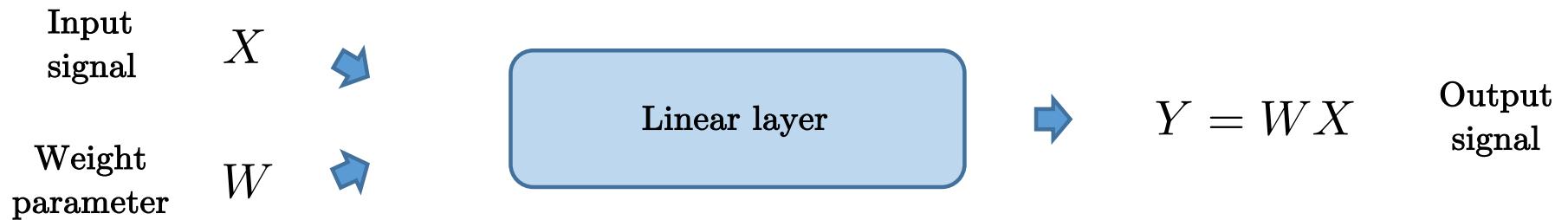
$$e = \frac{\partial \mathcal{L}^B}{\partial y}$$

Input
gradient

This is the scalar case.

Backpropagation for linear layer - matrix case

- Forward pass:



- Backward pass:

$$\frac{\partial \mathcal{L}^B}{\partial X} = \left(\frac{\partial Y}{\partial X} \right)^T \cdot \frac{\partial \mathcal{L}^B}{\partial Y}$$

Chain rule on $\mathcal{L}^B(Y(W, X))!$

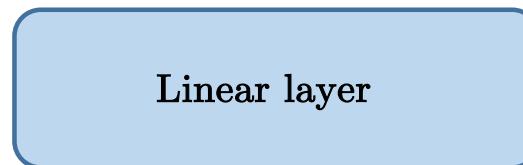
$$= W^T E$$

Matrix dimension consistency

Output gradients

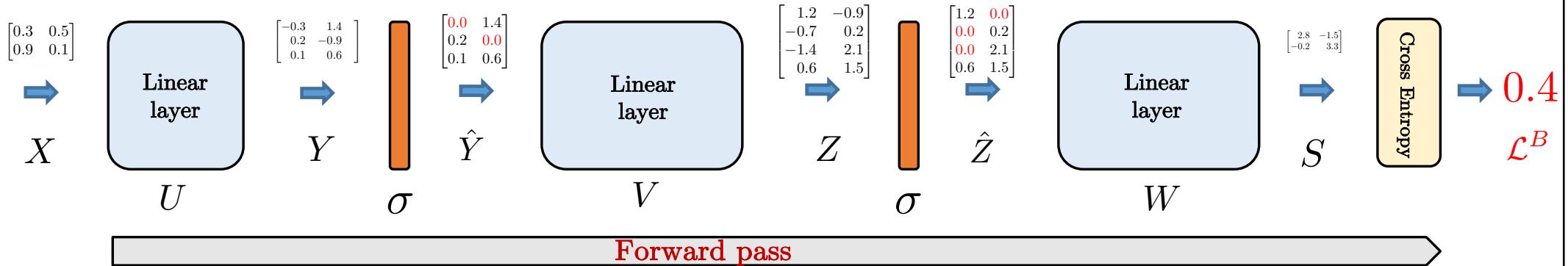
$$\frac{\partial \mathcal{L}^B}{\partial W} = \frac{\partial \mathcal{L}^B}{\partial Y} \cdot \left(\frac{\partial Y}{\partial W} \right)^T$$

$$= EX^T$$

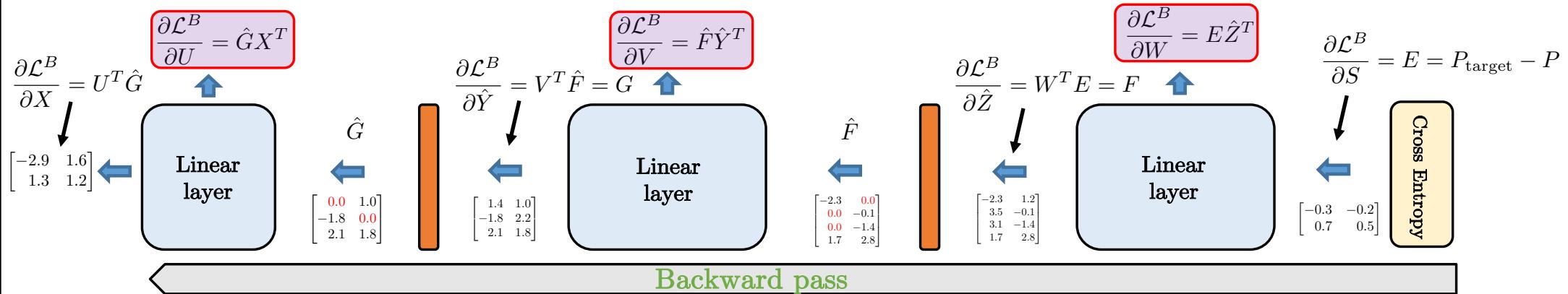


Backpropagation for MLP

- Compute all outputs of the forward pass :

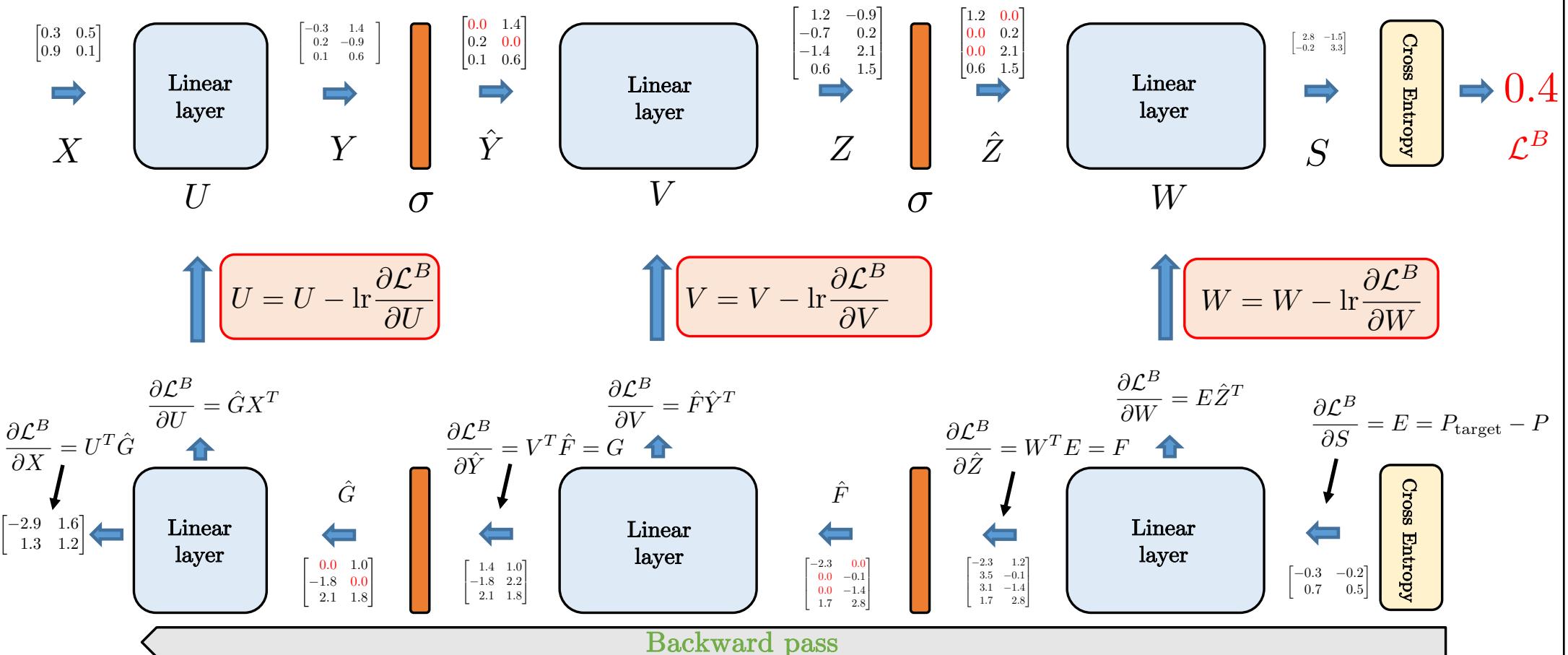


- Compute all gradients of network parameters sequentially in the backward direction :



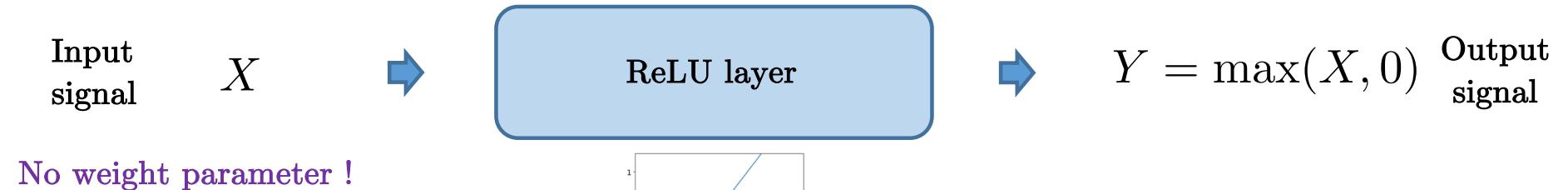
Backpropagation for MLP

- Update all network parameters with gradient descent sequentially in the backward direction :



Backpropagation for ReLU layer

- Forward pass:



- Backward pass:

$$\frac{\partial Y}{\partial X} = \frac{\partial}{\partial X} \max(X, 0) = \begin{cases} 1 & \text{if } X \text{ value is positive} \\ 0 & \text{otherwise} \end{cases}$$

$= \text{step}(X)$

Output gradient $\frac{\partial \mathcal{L}^B}{\partial X} = \left(\frac{\partial Y}{\partial X} \right)^T \cdot \frac{\partial \mathcal{L}^B}{\partial Y} = \text{step}(X).E$

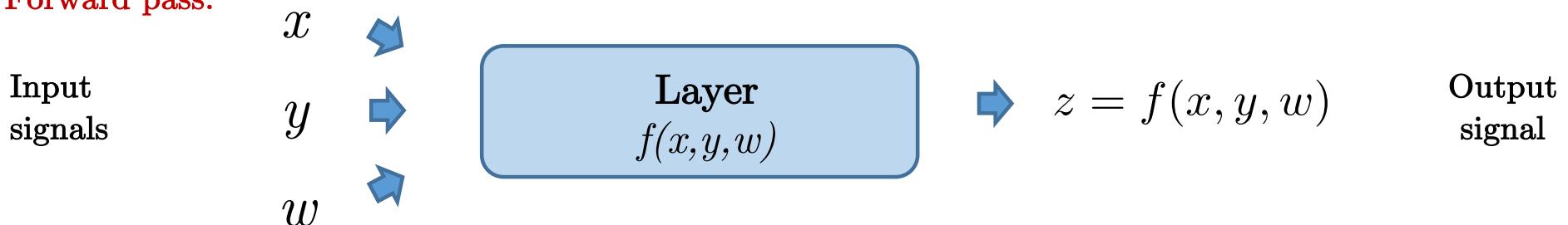
$E = \frac{\partial \mathcal{L}^B}{\partial Y}$ Input gradient

$$\frac{\partial Y}{\partial X} = \begin{cases} 1 & \text{if } X > 0 \\ 0 & \text{otherwise} \end{cases}$$

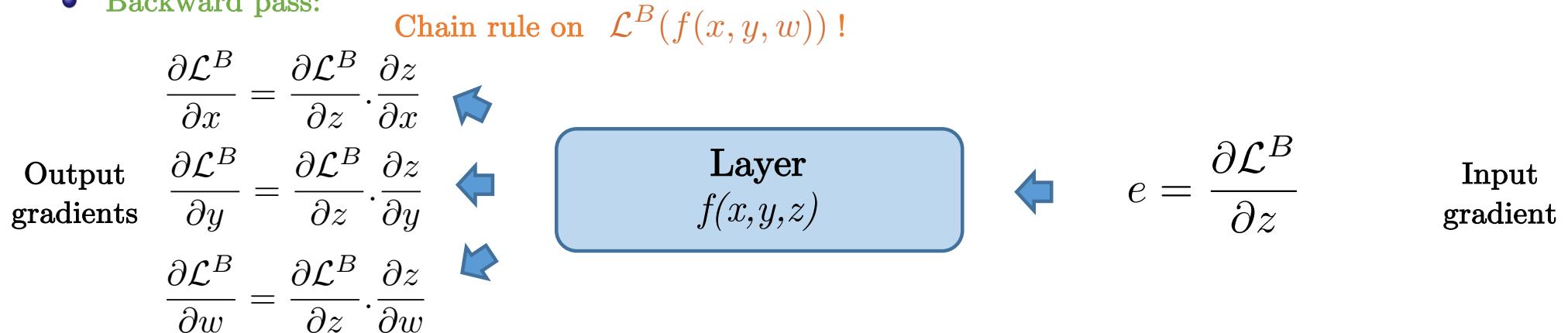
Chain rule on $\mathcal{L}^B(Y(W, X))!$

Generalizing backpropagation for any layer

- Forward pass:



- Backward pass:



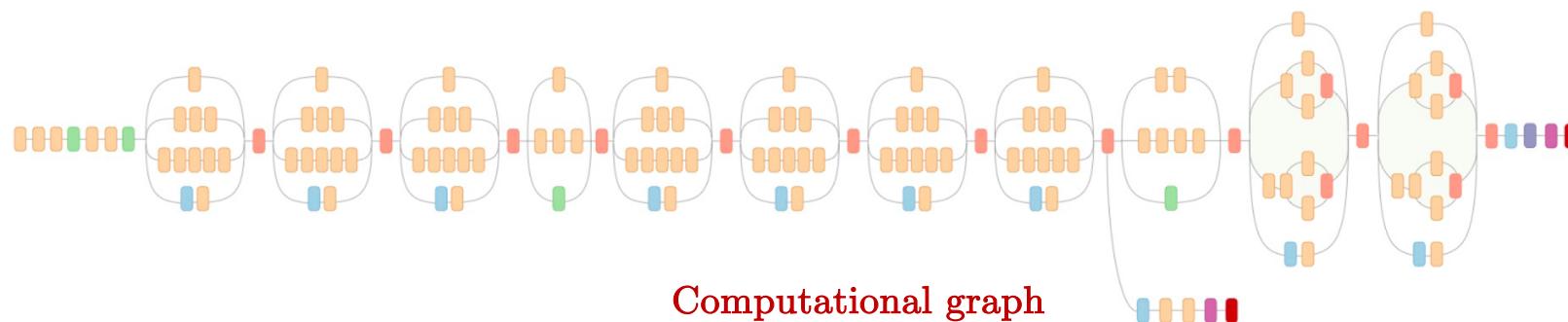
Backpropagation formula: Output gradient = Input gradient . Local gradient !

Backpropagation for any (huge) network

- Backpropagation is a local rule:

$$\text{Output gradient} = \text{Input gradient} \cdot \text{Local gradient}$$

- As such, it can scale up to any number of layers!



- Convolution
- AvgPool
- MaxPool
- Concat
- Dropout
- Fully connected
- Softmax



Design your own layer



Xavier Bresson @xbresson · 1 Oct 2017

@nitish_gup - I defined a new function torch.mm for sparse pytorch variables



Sparse x Dense -> Dense matrix multiplication

Hi everyone, I am trying to implement graph convolutional layer (as described in Semi-Supervised Classification with Graph Convolutional Networks) in... discuss.pytorch.org

1

2

6

|||

Show this thread

```
class my_sparse_mm(torch.autograd.Function):
    """
    Implementation of a new autograd function for sparse variables,
    called "my_sparse_mm", by subclassing torch.autograd.Function
    and implementing the forward and backward passes.
    """

    def forward(self, W, x): # W is SPARSE
        self.save_for_backward(W, x)
        y = torch.mm(W, x)
        return y

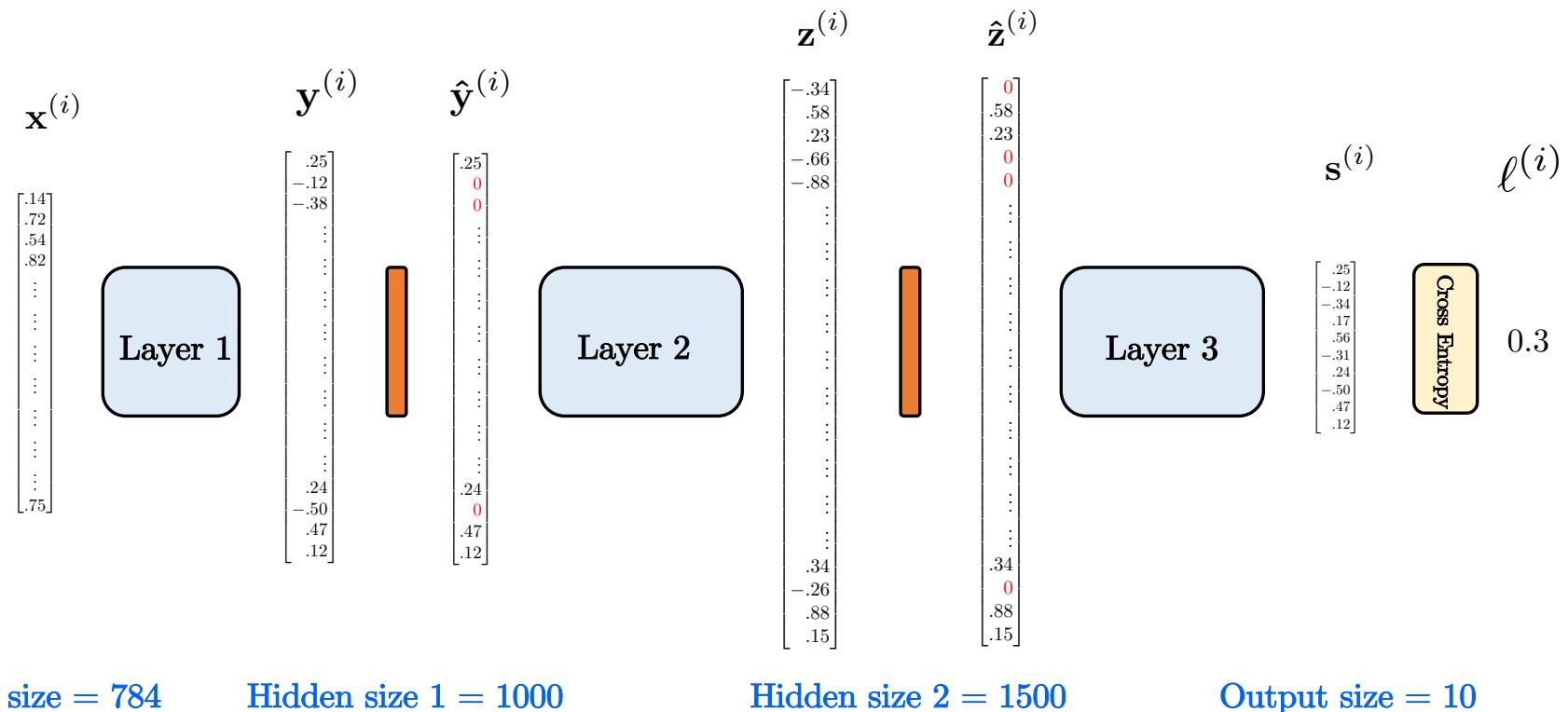
    def backward(self, grad_output):
        W, x = self.saved_tensors
        grad_input = grad_output.clone()
        grad_input_dL_dW = torch.mm(grad_input, x.t())
        grad_input_dL_dx = torch.mm(W.t(), grad_input )
        return grad_input_dL_dW, grad_input_dL_dx
```

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- **PyTorch implementation**
- Training with epochs
- Monitoring the loss
- Test set evaluation
- Final code

PyTorch implementation

- Let us implement a 3-layer network for MNIST and the cross-entropy loss :



Network and loss

- Define the network, instantiate it and choose the loss :

Make a class of
3-layer networks

Create the
network

Hyper-parameters:

- Input dim: 784
- 1st hidden layer: 1000
- 2nd hidden layer: 1500
- Output: 10

```
class three_layer_net(nn.Module):  
  
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):  
        super(three_layer_net , self).__init__()  
  
        self.layer1 = nn.Linear( input_size, hidden_size1 , bias=False)  
        self.layer2 = nn.Linear( hidden_size1, hidden_size2 , bias=False)  
        self.layer3 = nn.Linear( hidden_size2, output_size , bias=False)  
  
    def forward(self, x):  
  
        y      = self.layer1(x)  
        y_hat = F.relu(y)  
        z      = self.layer2(y_hat)  
        z_hat = F.relu(z)  
        scores = self.layer3(z_hat)  
  
        return scores  
  
net=three_layer_net(784, 1000, 1500, 10)  
criterion = nn.CrossEntropyLoss()
```

Define the
loss/criterion

Stochastic gradient descent

- Create an optimization function :
 - We create an optimizer object (from the `torch.optim` package):

```
optimizer = torch.optim.SGD( net.parameters() , lr=0.01 )
```

We choose our optimizer
to be an SGD
(there are other SGD-like
optimizers)

The optimizer is given access to
the parameters of the 3-layer network

We choose the
learning rate to be
 $lr = 0.01$

Training loop and backpropagation

Create the minibatch

Forward pass:
compute the scores

Compute the cross-
entropy criterion:

$$\mathcal{L}^B = \frac{1}{m} \sum_{i \in B} \ell^{(i)}$$

Backward pass:
compute the gradients

$$\frac{\partial \mathcal{L}^B}{\partial U} \quad \frac{\partial \mathcal{L}^B}{\partial V} \quad \frac{\partial \mathcal{L}^B}{\partial W}$$

```
for iter in range(5000):
    optimizer.zero_grad() ← Discussed later.

    indices = torch.LongTensor(bs).random_(0, 60000)
    minibatch_data = train_data[indices]
    minibatch_label = train_label[indices]

    inputs = minibatch_data.view(bs, 784)

    inputs.requires_grad_() ← Start recording all operations that will be done to the input tensor.

    scores = net(inputs)
    loss = criterion(scores, minibatch_label)

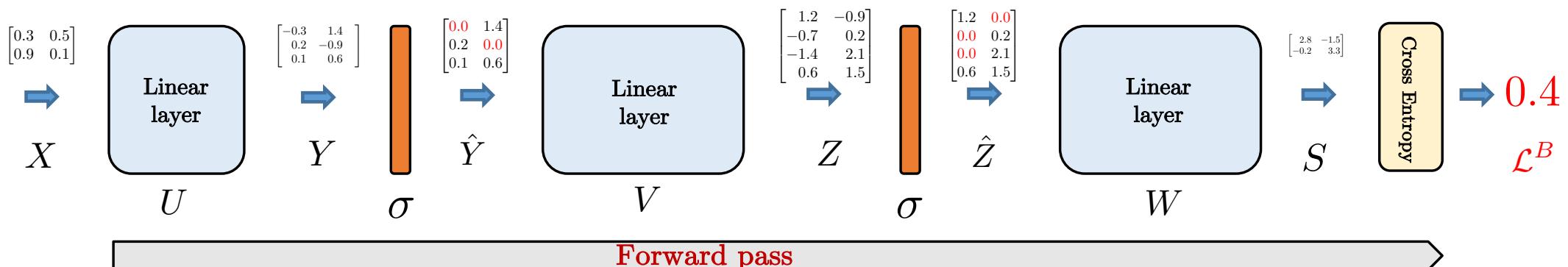
    loss.backward()
    optimizer.step() ← Do one step of SGD to update the weights
```

Start recording all operations that will be done to the input tensor.

Discussed later.

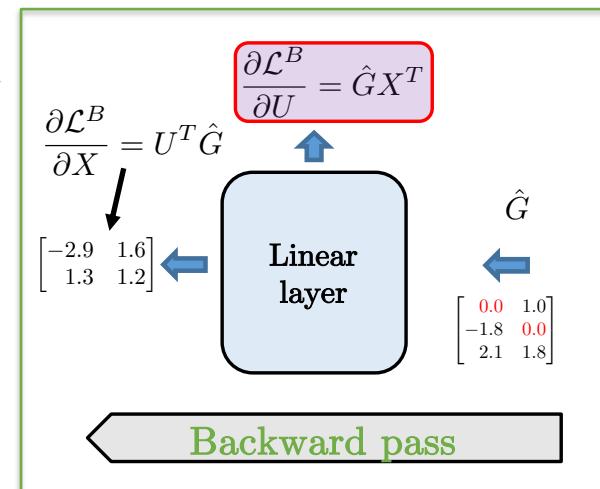
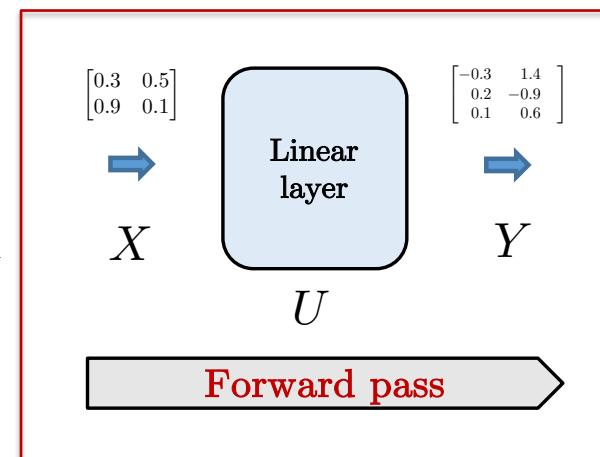
Automatic backpropagation

- Function `requires_grad_0` :
 - It tells Pytorch to start recording all operations that will be done to the input tensor X .
 - PyTorch will create the computational graph by stacking the layers and recording the outputs of all layers in sequential order.
 - It will then compute all gradients automatically (autograd) with the backward pass.



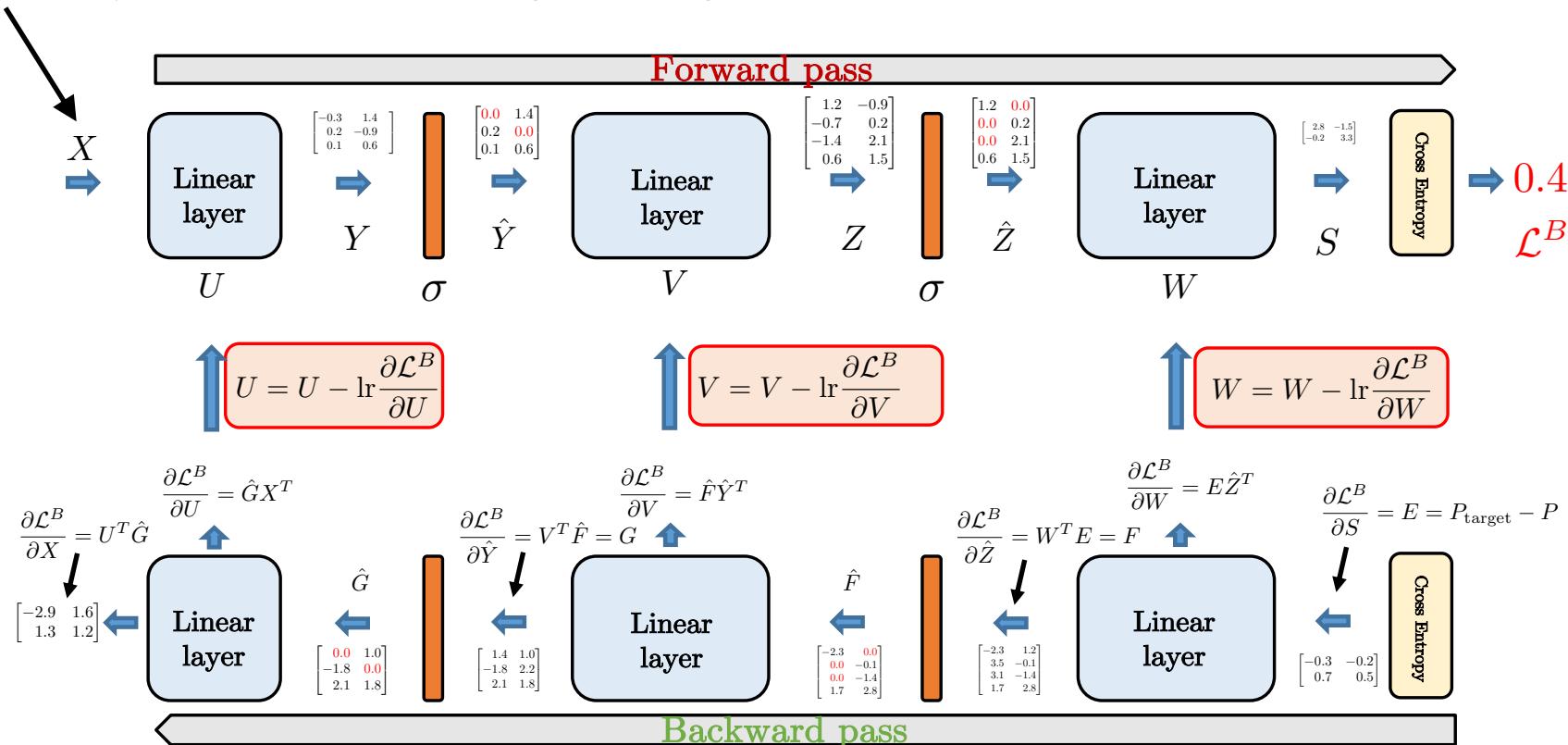
Automatic backpropagation

- Function `requires_grad_0` :
 - In Pytorch, each **forward operation** that can be performed on a tensor has a **backward equivalent** (already defined or that can be implemented).
 - In deep learning, **most layers** are composed of simple operations (sum, multiplication, `exp()`, convolution, etc) that are **efficiently implemented** with e.g. MKL on CPU or CUDA on GPU. Python functions are just wrappers.



Automatic backpropagation

Put a `requires_grad()` flag on this tensor and it will record the history of all the operations it goes through.



Then all these operations are replayed in reverse order with `loss.backward()` (the backward version of these operations)

Gradient accumulation

- If `optimizer.zero_grad()` is NOT used then Pytorch adds the gradient computed on this batch to the total gradient:

$$\frac{\partial \mathcal{L}}{\partial U} = \frac{\partial \mathcal{L}}{\partial U} + \frac{\partial \mathcal{L}^B}{\partial U} \Big|_{\text{computed on this batch}}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial W} + \frac{\partial \mathcal{L}^B}{\partial W} \Big|_{\text{computed on this batch}}$$

$$\frac{\partial \mathcal{L}}{\partial V} = \frac{\partial \mathcal{L}}{\partial V} + \frac{\partial \mathcal{L}^B}{\partial V} \Big|_{\text{computed on this batch}}$$

- This is only convenient if you want to do a true gradient where you use all the data points to compute the gradient.

```
for iter in range(5000):  
    optimizer.zero_grad()  
  
    indices = torch.LongTensor(bs).random_(0,60000)  
    minibatch_data = train_data[indices]  
    minibatch_label = train_label[indices]  
  
    inputs = minibatch_data.view(bs,784)  
  
    inputs.requires_grad_()  
  
    scores = net( inputs )  
  
    loss = criterion( scores , minibatch_label )  
  
    loss.backward()  
  
    optimizer.step()
```

Mini-batch gradient

- But if we want to do a stochastic gradient descent, we need function **optimizer.zero_grad()** :

$$\frac{\partial \mathcal{L}}{\partial U} = 0$$
$$\frac{\partial \mathcal{L}}{\partial V} = 0$$
$$\frac{\partial \mathcal{L}}{\partial W} = 0$$

$$\frac{\partial \mathcal{L}}{\partial U} = 0 + \frac{\partial \mathcal{L}^B}{\partial U} \Big|_{\text{computed on this batch}}$$
$$\frac{\partial \mathcal{L}}{\partial W} = 0 + \frac{\partial \mathcal{L}^B}{\partial W} \Big|_{\text{computed on this batch}}$$
$$\frac{\partial \mathcal{L}}{\partial V} = 0 + \frac{\partial \mathcal{L}^B}{\partial V} \Big|_{\text{computed on this batch}}$$

```
for iter in range(5000):  
  
    optimizer.zero_grad()  
  
    indices = torch.LongTensor(bs).random_(0,60000)  
    minibatch_data = train_data[indices]  
    minibatch_label = train_label[indices]  
  
    inputs = minibatch_data.view(bs,784)  
  
    inputs.requires_grad_()  
  
    scores = net( inputs )  
  
    loss = criterion( scores , minibatch_label )  
  
    loss.backward()  
  
    optimizer.step()
```

Lab 01

- Understanding the training loop

The image shows two Jupyter Notebook interfaces side-by-side, both titled "Lab 01 : MLP -- demo".

Left Notebook (demo):

- Cell 1:** Imports torch, torch.nn, torch.nn.functional, torch.optim, random, and utils.
- Cell 2:** Downloads MNIST data and loads it into train_data, train_label, and test_data tensors.
- Cell 3:** Defines a three-layer net class named `three_layer_net` with three linear layers.

Right Notebook (exercise):

- Cell 1:** Imports torch, torch.nn, torch.nn.functional, torch.optim, random, and utils.
- Cell 2:** Downloads CIFAR-10 data and prints its size.
- Cell 3:** Loads CIFAR-10 train labels and prints their size.
- Cell 4:** Loads CIFAR-10 test data and prints its size.

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- **Training with epochs**
- Monitoring the loss
- Test set evaluation
- Final code

Training with epochs

- Up to now we have been training as follows:
 - Feed 200 randomly selected images to the net (note that the same image can be drawn multiple times).
- An epoch is a full pass through the randomly ordered training set.
 - Training with epochs prevents sampling the same image before seeing all other training images.
 - At the beginning of the epoch, the 60,000 images from the training set are randomly ordered.
 - Then:
 - Feed image 0 - 199 to the net and train.
 - Feed image 200 - 399 to the net and train.
 - ...
 - Feed image 59,800 - 59,999 to the net and train.

Training with epochs

- We have been training as follows:

```
for iter in range(5000):  
    optimizer.zero_grad()  
  
    indices = torch.LongTensor(bs).random_(0,60000)  
    minibatch_data = train_data[indices]  
    minibatch_label = train_label[indices]  
  
    inputs = minibatch_data.view(bs,784)  
    inputs.requires_grad_()  
  
    scores = net( inputs )  
  
    loss = criterion( scores , minibatch_label )  
  
    loss.backward()  
  
    optimizer.step()
```

- Training with epochs :

- Here we do 8 complete passes through the training set.

```
for epoch in range(8):  
    shuffled_indices=torch.randperm(60000)  
  
    for count in range(0,60000,bs):  
        optimizer.zero_grad()  
  
        indices=shuffled_indices[count:count+bs]  
        minibatch_data = train_data[indices]  
        minibatch_label= train_label[indices]  
  
        inputs = minibatch_data.view(bs,784)  
        inputs.requires_grad_()  
  
        scores=net( inputs )  
  
        loss = criterion( scores , minibatch_label )  
  
        loss.backward()  
  
        optimizer.step()
```

Lab 02

- Training with epochs

The image shows two Jupyter Notebook interfaces side-by-side, both titled "Lab 02".

Left Notebook (demo):

- Cell 1:** `In [1]: import torch`
- Text:** Lets make an artificial training set of 10 images (28x28 pixels)
- Cell 2:** `In [2]: train_data = torch.rand(10,28,28)
print(train_data.size())
torch.Size([10, 28, 28])`
- Text:** Lets define a the random order in which we are going to visit these images:
- Cell 3:** `In [3]: shuffled_indices = torch.randperm(10)
print(shuffled_indices)
tensor([3, 8, 0, 9, 7, 5, 1, 4, 2, 6])`
- Text:** Visit the training set in this random order and do minibatch of size 2
- Cell 4:** `In [4]: bs=2
for count in range(0,10,bs):
 batch_of_indices = shuffled_indices[count:count+bs]
 print(batch_of_indices)
 batch_of_images = train_data[batch_of_indices]`

Right Notebook (exercise):

- Cell 1:** `In [1]: import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils`
- Text:** Download the data and print the sizes
- Cell 2:** `In [2]: from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()
train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')`
- Text:** Make a ONE layer net class.
- Cell 3:** `In [3]: class one_layer_net(nn.Module):
 def __init__(self, input_size, output_size):
 super(one_layer_net , self).__init__()
 self.linear_layer = nn.Linear(input_size, output_size , bias=False)
 def forward(self, x):
 scores = self.linear_layer(x)
 return scores`

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- **Monitoring the loss**
- Test set evaluation
- Final code

Monitoring the loss

- Monitoring the loss and the classification error :

Reset to zero at the beginning
of a new epoch

running loss +=
average loss of current mini-batch

running error +=
average error on current mini-batch

```
for epoch in range(30):
    running_loss=0
    running_error=0
    num_batches=0

    shuffled_indices=torch.randperm(60000)

    for count in range(0,60000,bs):
        optimizer.zero_grad()

        indices=shuffled_indices[count:count+bs]
        minibatch_data = train_data[indices]
        minibatch_label= train_label[indices]

        inputs = minibatch_data.view(bs,784)
        inputs.requires_grad_()
        scores=net( inputs )
        loss = criterion( scores , minibatch_label)
        loss.backward()
        optimizer.step()

        # compute and accumulate stats
        running_loss += loss.detach().item()
        error = utils.get_error( scores.detach() , minibatch_label)
        running_error += error.item()

        num_batches+=1

    # compute stats for the full training set
    total_loss = running_loss/num_batches
    total_error = running_error/num_batches

    print('epoch=',epoch, '\t loss=', total_loss , '\t error=', total_error*100 , 'percent')
```

Monitoring loss and error

- Running error = 10% + 15% + 12% + + 15%
- 
- Error rate on the 1st batch of 200 images
- Error rate on the last batch of 200 images

$$\text{Error rate on the full training set} = \frac{\text{running error}}{\text{number of batches}}$$

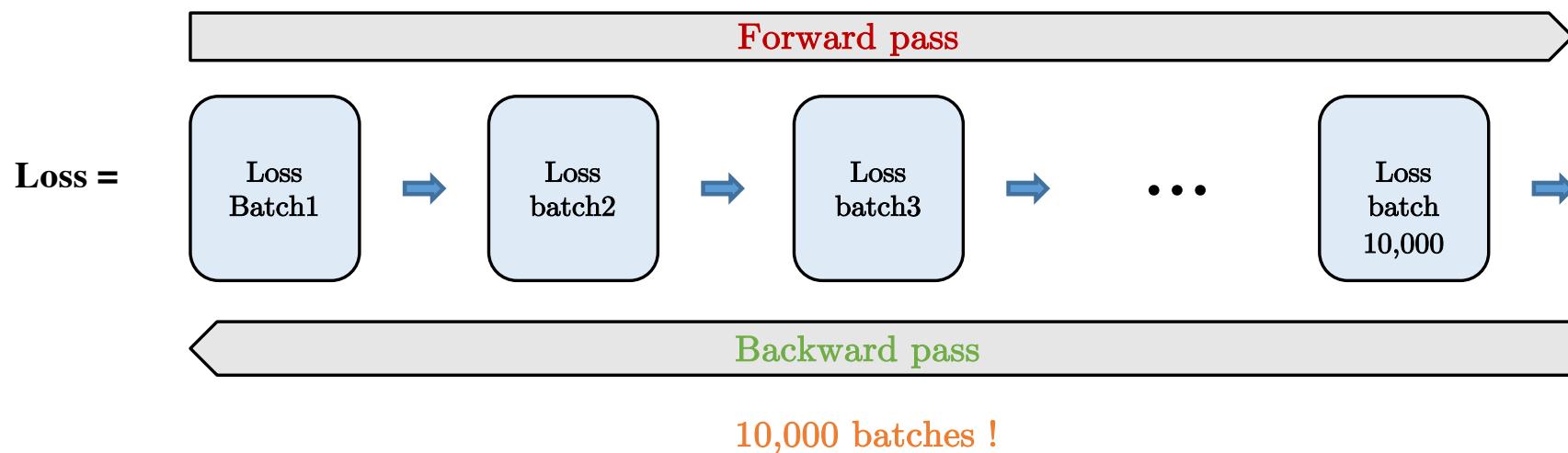
- Same for the running loss

PyTorch

- Understand the code line:
 - `running_loss += loss.detach().item()`
- Function `item()` :
 - Get a Python number from a tensor containing a single value.
- Function `detach()` :
 - This function extracts the value of the node in the computational graph.
 - This function is essential to avoid building a big computational graph for the `running_loss`.
 - If it is not used, then the graph will be big and the backward pass (autograd) will take a long time, but for nothing as the `running_loss` value does not need the gradient!

PyTorch

- Without `detach()` :
 - `running_loss += loss`
 - `loss.backward()`



Lab 03

- Computing the loss and the error rate

The image shows two Jupyter notebook interfaces side-by-side, both titled "Lab 03 : Loss and error rate -- demo" and "Lab 03 : Loss and error rate -- exercise".

Left Notebook (demo):

- In [1]:** Python code to import torch and its modules.
- In [2]:** Python code to download MNIST data and print its sizes.
- In [3]:** Python code to define a simple one-layer neural network class.

Right Notebook (exercise):

- In [1]:** Python code to import torch and its modules.
- In [2]:** Python code to download CIFAR-10 data and print its size.
- In [3]:** Python code to define a simple one-layer neural network class.

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- **Test set evaluation**
- Final code

Generalization performance

- Evaluation on the test set :
 - No need to **shuffle** the test set.
 - Just **make a pass** through the full test set in the original order.

```
running_error=0
num_batches=0

for i in range(0,10000,bs):

    minibatch_data = test_data[i:i+bs]
    minibatch_label= test_label[i:i+bs]

    inputs = minibatch_data.view(bs,784)

    scores = net( inputs )

    error = utils.get_error( scores , minibatch_label)

    running_error += error.item()

    num_batches+=1

total_error = running_error/num_batches

print( 'error rate on test set =', total_error*100 , 'percent' )
```

Lab 04

- Test set evaluation

Jupyter test_set_demo Last Checkpoint: 3 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O

In [1]: `import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils`

Lab 04 : Test set evaluation -- demo

Download the data and print the sizes

In [2]: `from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
test_label=torch.load(data_path+'mnist/test_label.pt')`

Make a ONE layer net class.

In [3]: `class one_layer_net(nn.Module):

 def __init__(self, input_size, output_size):
 super(one_layer_net, self).__init__()
 self.linear_layer = nn.Linear(input_size, output_size, bias=False)

 def forward(self, x):
 scores = self.linear_layer(x)
 return scores`

Jupyter test_set_exercise Last Checkpoint: 3 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 O

In [1]: `import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import utils`

Lab 04 : Test set evaluation -- exercise

Download the CIFAR dataset -- check the size carefully!

In [2]: `from utils import check_cifar_dataset_exists
data_path=check_cifar_dataset_exists()

train_data=torch.load(data_path+'cifar/train_data.pt')
train_label=torch.load(data_path+'cifar/train_label.pt')
test_data=torch.load(data_path+'cifar/test_data.pt')
test_label=torch.load(data_path+'cifar/test_label.pt')

print(train_data.size())`

torch.Size([50000, 3, 32, 32])

Make a ONE layer net class.

In []: `class one_layer_net(nn.Module):

 def __init__(self, input_size, output_size):
 super(one_layer_net, self).__init__()
 self.linear_layer = nn.Linear(input_size, output_size, bias=True)`

Outline

- Multi-Layer Perceptron
- Loss and gradient descent
- Backpropagation
- PyTorch implementation
- Training with epochs
- Monitoring the loss
- Test set evaluation
- **Final code**

Final code

- We just make the following improvements to our code:
 - Add a learning rate schedule.
 - Evaluate on test set every 10 epochs (rather than waiting for the end of the training).
 - Provide some timing information.
 - Count number of parameters in network.
- This final code is a prototype for any deep learning project:
 - What changes for a new project:
 - New data (collect, prepare, access to batch of training data)
 - New architecture
 - New learning rate schedule

Lab 05

- Demo of the final code :

jupyter final_demo Last Checkpoint: 3 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [1]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from random import randint
import time
import utils
```

In [2]:

```
from utils import check_mnist_dataset_exists
data_path=check_mnist_dataset_exists()

train_data=torch.load(data_path+'mnist/train_data.pt')
train_label=torch.load(data_path+'mnist/train_label.pt')
test_data=torch.load(data_path+'mnist/test_data.pt')
test_label=torch.load(data_path+'mnist/test_label.pt')
```

In [3]:

```
class one_layer_net(nn.Module):
    def __init__(self, input_size, output_size):
        super(one_layer_net, self).__init__()
        self.linear_layer = nn.Linear(input_size, output_size, bias=False)

    def forward(self, x):
        scores = self.linear_layer(x)
        return scores
```



Questions?