Programming Tools For Image Processing

Traditional Image Processing

Neural Networks

Håkon Hukkelås | hakon.hukkelas@ntnu.no

Todays Lecture

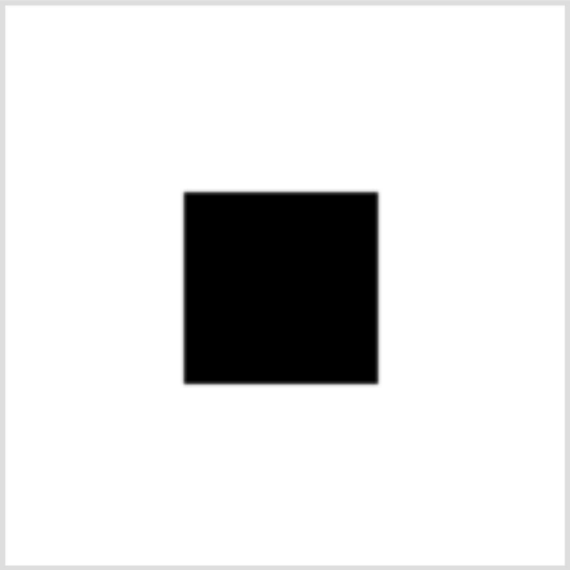
- Oriented towards the programming assignments
  - Image processing with numpy
  - Convolutional operations
  - Backpropagation
  - Neural Networks in Pytorch

Representation of Images

```python
im = np.array([
        [255, 255, 255],
        [255, 0,   255],
        [255, 255, 255]
])
plt.imshow(im, cmap="gray")
```

Can be represented in the range 0-255 (8bit unsigned int)
Shape: (Image Height, Image Width, #color channels)

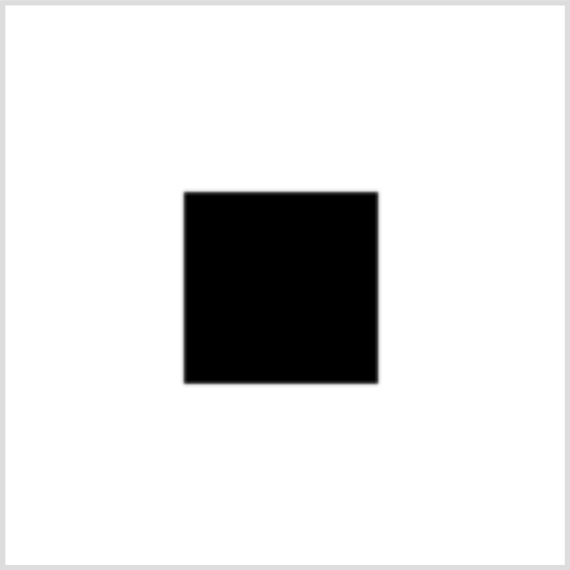
Representation of Images



```python
im = np.array([
      [255, 255, 255],
      [255, 0,   255],
      [255, 255, 255]
])
plt.imshow(im, cmap="gray")
```

Can be represented in the range 0-255 (8bit unsigned int)
Shape: (Image Height, Image Width, #color channels)

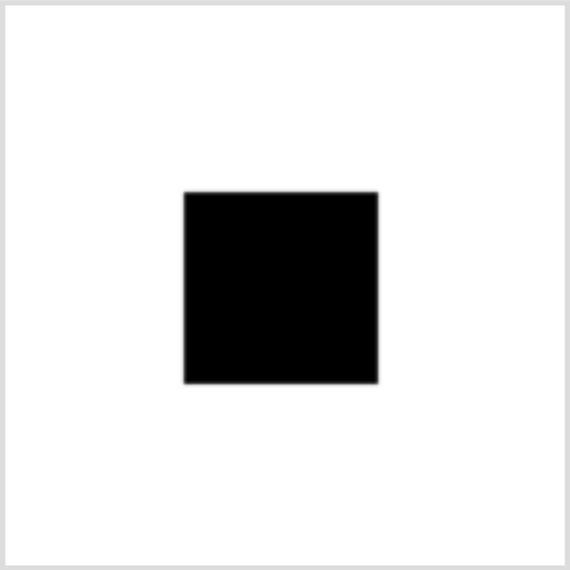
Representation of Images



```
im = np.array([
      [1., 1., 1.],
      [1., 0., 1.],
      [1., 1., 1.]
])
plt.imshow(im, cmap="gray")
```

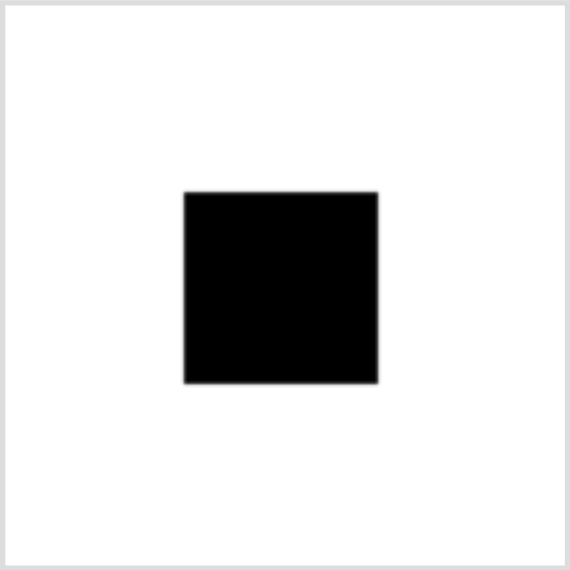
Can be represented in the range 0-1 (float)
Shape: (Image Height, Image Width, #color channels)

Working with Numpy



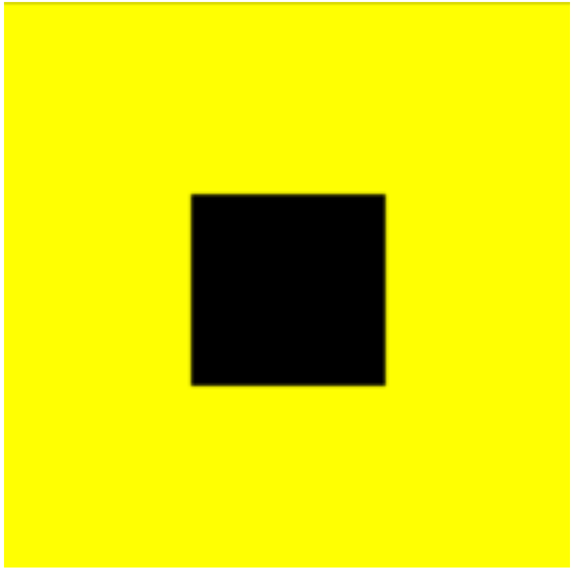
```python
# We can make this image
# into RGB (3 dimensions)
im = im.reshape(3, 3, 1)
im = np.tile(im, [1, 1, 3])
print(im.shape)
```

Working with Numpy



```python
# We can set the color
# "blue" to 0
im[:, :, 2] = 0
plt.imshow(im)
```

Working with Numpy



```
# We can set the color
# "blue" to 0
im[:, :, 2] = 0
plt.imshow(im)
```

Working with Numpy

```python
import skimage
chelsea = skimage.data.chelsea()
plt.imshow(chelsea)
print(f"Image has shape: {chelsea.shape},",
      f"with dtype={chelsea.dtype},",
      f"min value={chelsea.min()},",
      f"max value={chelsea.max()}")
```

Image has shape: (300, 451, 3),
with dtype=uint8,
min value=0, max value=231

Working with Numpy

```python
import skimage
chelsea = skimage.data.chelsea()
plt.imshow(chelsea)
print(f"Image has shape: {chelsea.shape},",
      f"with dtype={chelsea.dtype},",
      f"min value={chelsea.min()},",
      f"max value={chelsea.max()}")
```

Image has shape: (300, 451, 3),
with dtype=uint8,
min value=0, max value=231

Working with Numpy

```python
# Lets put a green box in the middle
chelsea_l = chelsea.copy()
chelsea_l[100:150, 200:300, :] = 0
chelsea_l[100:150, 200:300, 1] = 255
plt.imshow(chelsea_l)
```

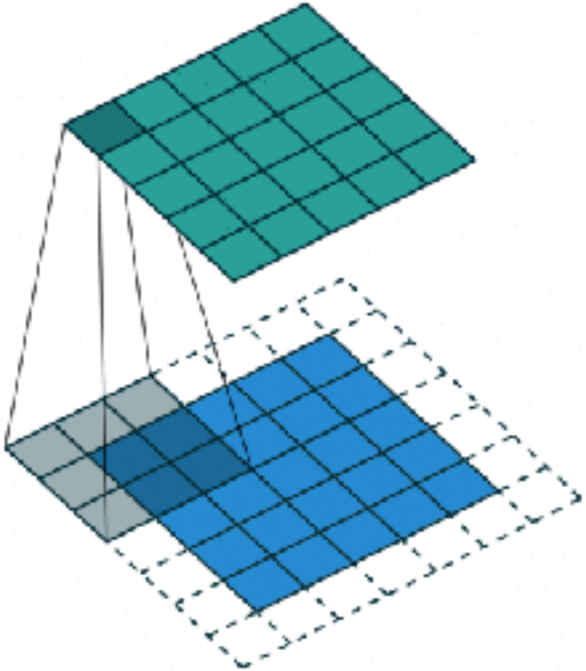
Row, Column, Color

Working with Numpy

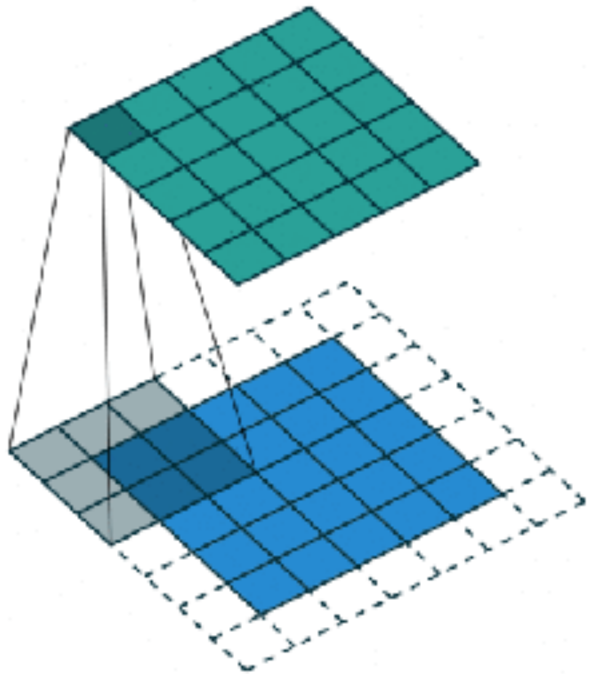
```python
# Lets put a green box in the middle
chelsea_l = chelsea.copy()
chelsea_l[100:150, 200:300, :] = 0
chelsea_l[100:150, 200:300, 1] = 255
plt.imshow(chelsea_l)
```

Row, Column, Color

Convolutional operation

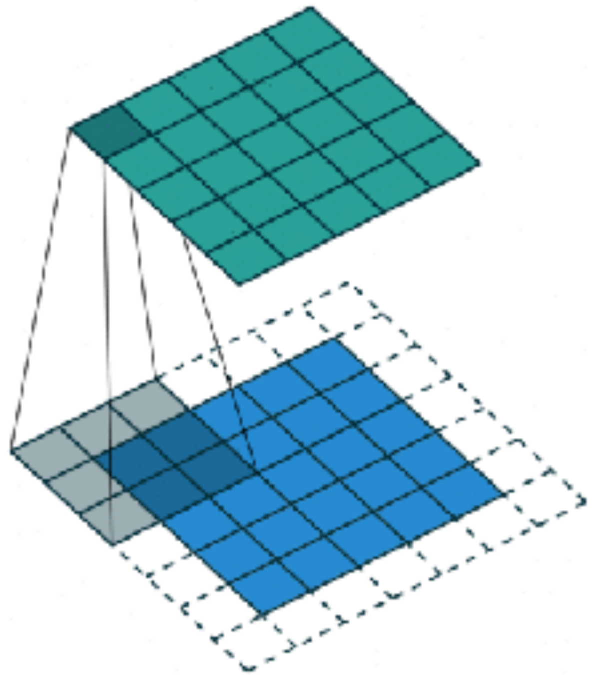


Convolutional operation



Convolutional operation

1. Place the kernel in the top left corner
2. Apply the kernel and compute the result (single number)
3. Slide the kernel to the right by 1 pixel



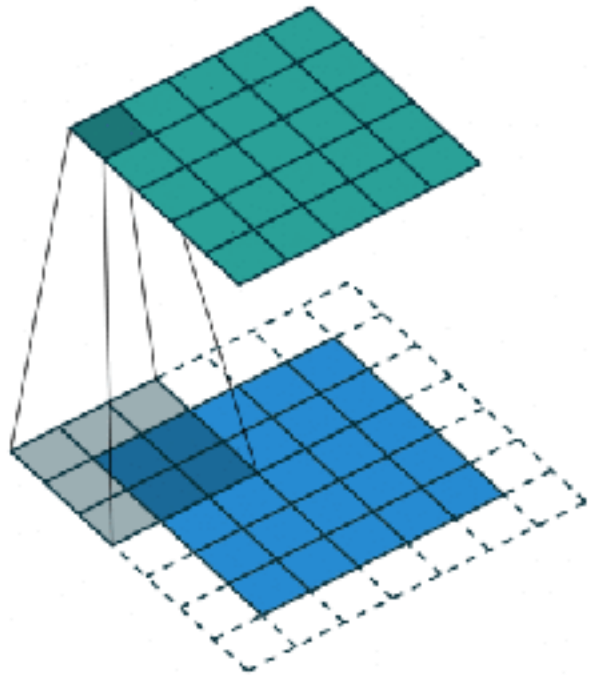
f= image, h = convolutional kernel

$$(f * h)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j)(h(x - i, y - j)$$

NTNU

Convolutional operation

1. Place the kernel in the top left corner
2. Apply the kernel and compute the result (single number)
3. Slide the kernel to the right by 1 pixel



f= image, h = convolutional kernel

$$(f * h)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j)(h(x - i, y - j)$$

Convolution Operation Usage

We can find edges by using Sobel:

Convolution Operation Usage

We can find edges by using Sobel (Vertical edges):

1	0	-1
2	0	-2
1	0	-1

Convolution Operation Usage

We can find edges by using Sobel (Vertical edges):

1	0	-1
2	0	-2
1	0	-1

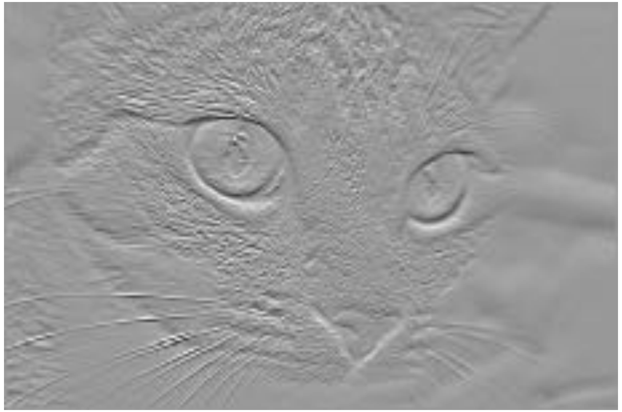
Convolution Operation Usage

We can find edges by using Sobel (horizontal):

1	2	1
0	0	0
-1	-2	-1

Convolution Operation Usage

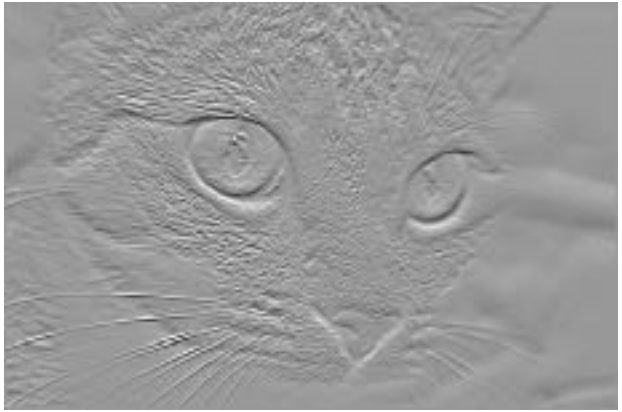
We can find edges by using Sobel (horizontal):



1	2	1
0	0	0
-1	-2	-1

Convolution Operation Usage

Can combine the horizontal and vertical edges



$$G = \sqrt{G_x^2 + G_y^2}$$

Convolution Operation Usage

Can combine the vertical and horizontal edges

$$G = \sqrt{G_x^2 + G_y^2}$$

Convolution Example

Convolve the 3x5 image with the 3x3 Kernel

1	2	3	4	5
7	8	9	10	11
12	13	14	15	16

0	1	0
1	0	1
0	2	0

$$(f * h)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j)(h(x - i, y - j)$$

Convolution Example

Convolve the 3x5 image with the 3x3 Kernel

Instead, we can rotate the kernel and perform correlation.

1	2	3	4	5
7	8	9	10	11
12	13	14	15	16

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

Convolution Example

Convolve the 3x5 image with the 3x3 Kernel

Instead, we can rotate the kernel and perform correlation.

1	2	3	4	5
7	$\boxed{8}$	9	10	11
12	13	14	15	16

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

Start in the top left (legal) corner

Convolution Example

Convolve the 3x5 image with the 3x3 Kernel

Instead, we can rotate the kernel and perform correlation.

1	2	3	4	5
7	8	9	10	11
12	13	14	15	16

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

33				

2*2 + 7*1 + 1*9 + 1*13 = 33

Start in the top left (legal) corner

Convolution Example

Convolve the 3x5 image with the 3x3 Kernel

Instead, we can rotate the kernel and perform correlation.

1	2	3	4	5
7	8	$\boxed{9}$	10	11
12	13	14	15	16

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

33	38	

Stride along the image (horizontally and vertically)

Convolution Example

Convolve the 3x5 image with the 3x3 Kernel

Instead, we can rotate the kernel and perform correlation.

1	2	3	4	5
7	8	9	$\boxed{10}$	11
12	13	14	15	16

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

33	38	43

Stride along the image (horizontally and vertically)

Convolution Example - With Padding

Convolve the 3x5 image with the 3x3 Kernel

0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	7	8	9	10	11	0
0	12	13	14	15	16	0
0	0	0	0	0	0	0

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

Convolution Example - With Padding

Convolve the 3x5 image with the 3x3 Kernel

0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	7	8	9	10	11	0
0	12	13	14	15	16	0
0	0	0	0	0	0	0

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

To keep the original shape, we pad with some value
Most common:

- Reflection Padding
- Zero Padding

Convolution Example - With Padding

Convolve the 3x5 image with the 3x3 Kernel

0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	7	8	9	10	11	0
0	12	13	14	15	16	0
0	0	0	0	0	0	0

0	2	0
1	0	1
0	1	0

(Kernel Rotated 180°)

9	12	15	18	15
22	33	38	43	36
27	42	46	50	37

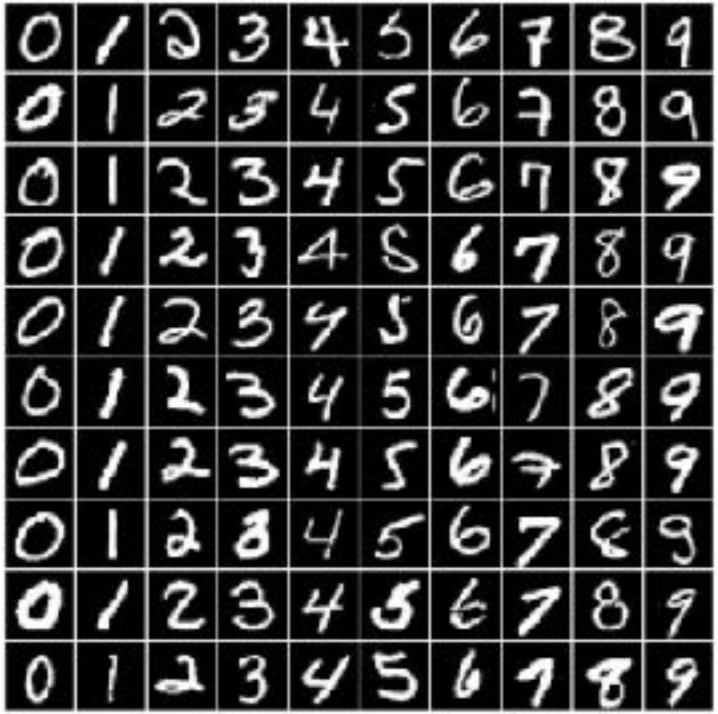
To keep the original shape, we pad with some value

Most common:

- Reflection Padding
- Zero Padding

Classification with Neural Networks

We are going to build a NN to classify digits (0-10)



We will use the MNIST database:

- 60,000 images in training set
- 10,000 images in test set

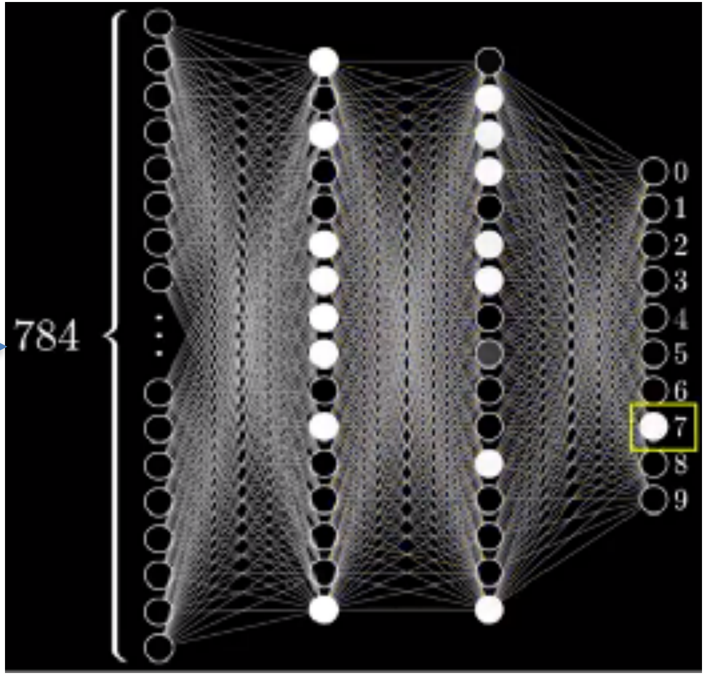
MNIST Neural Network

Each image has shape 28x28x1

Want to predict the confidence (0-1) for each digit

MNIST Neural Network

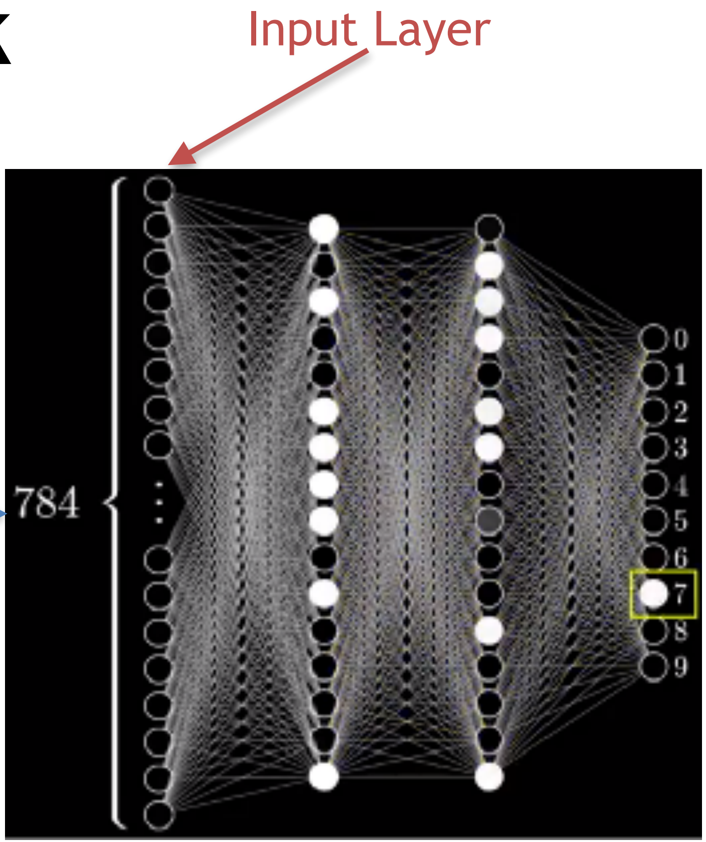
Each image has shape 28x28x1



Want to predict the confidence (0-1) for each digit

MNIST Neural Network

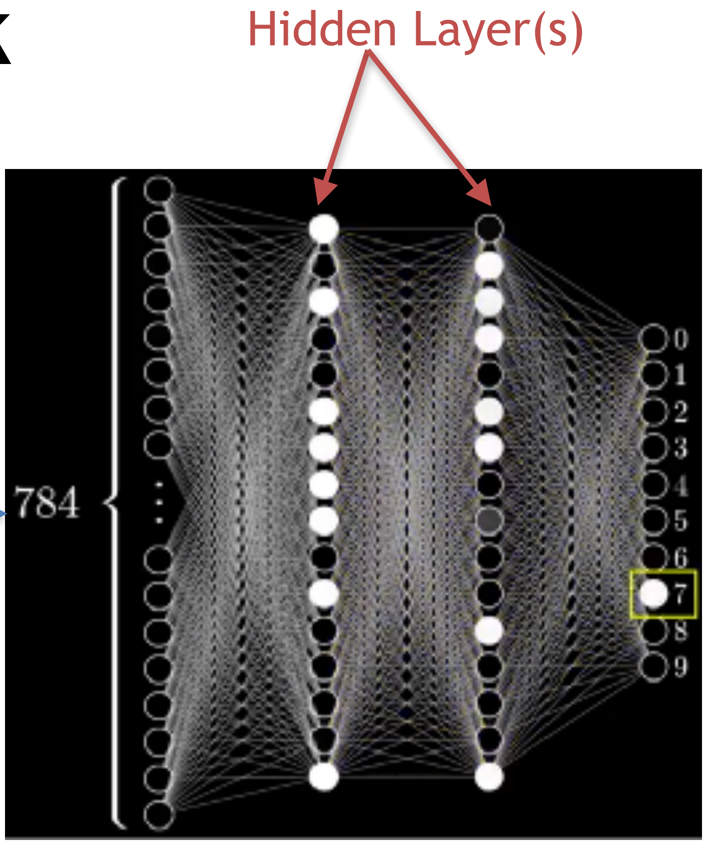
Each image has shape 28x28x1



Want to predict the confidence (0-1) for each digit

MNIST Neural Network

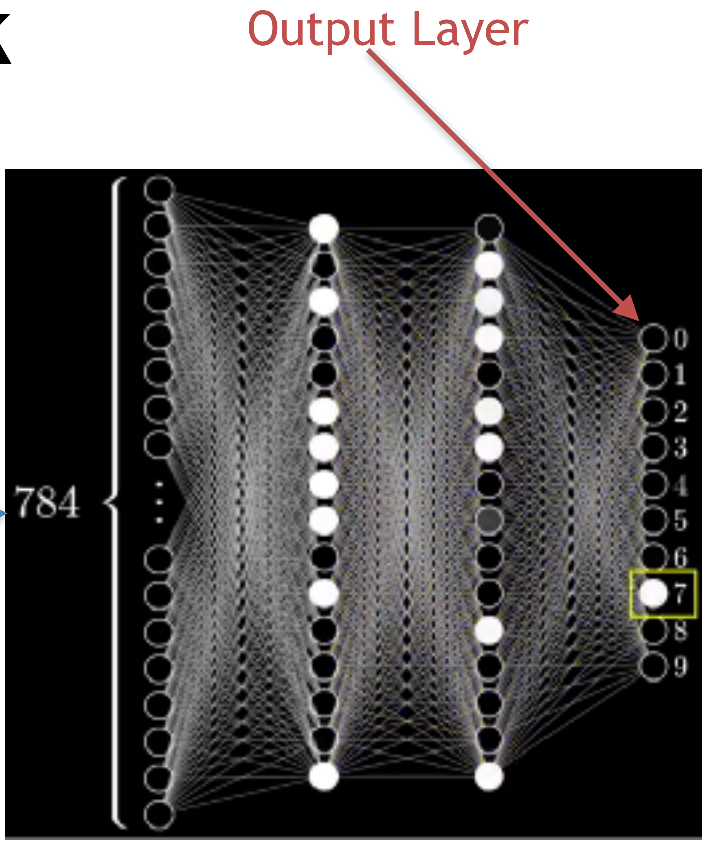
Each image has shape 28x28x1



Want to predict the confidence (0-1) for each digit

MNIST Neural Network

Each image has shape 28x28x1

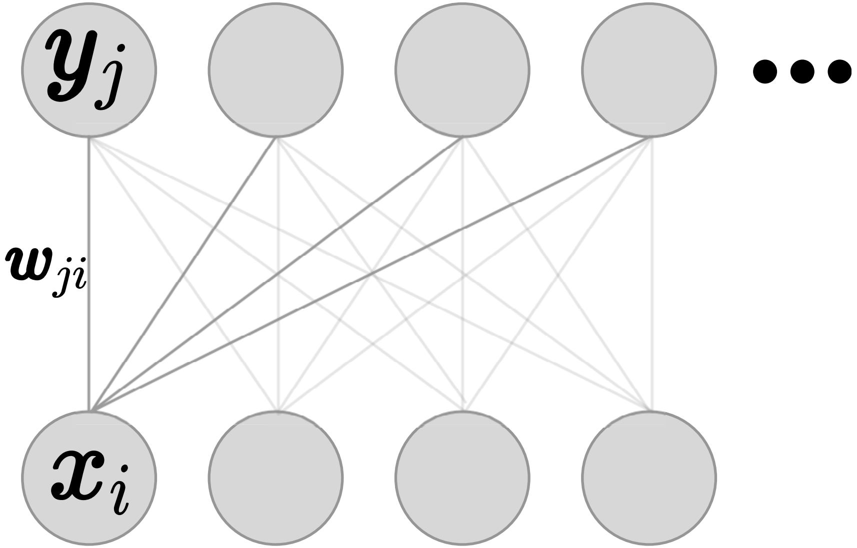


Want to predict the confidence (0-1) for each digit

MNIST Neural Network

We will have:

- y: 10 outputs
- x: 784 inputs

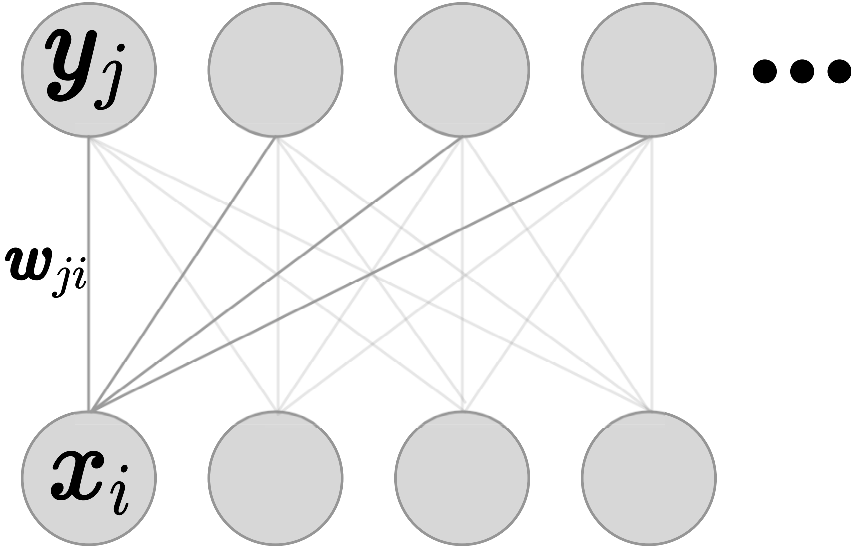


NTNU

MNIST Neural Network

We will have:

- y: 10 outputs
- x: 784 inputs
- w: [784, 10]
- bias: 10 (per output)



MNIST Neural Network - In Numpy

We will have:

- y: 10 outputs
- x: 784 inputs
- w: [784, 10]
- bias: 10 (per output)

```python
# Simple neural network
def forward(x, w):
    z = x.dot(w)
    a = softmax(z)
    return a
```

MNIST Neural Network - In Numpy

Our neural networks outputs confidence scores

[0.9, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01]

Digit “0”

Digit “7”

Training a Neural Network

We have to define a cost function:

$$C = -\sum_{n}^{N}\sum_{k=1}^{C}\hat{y}_k^n \ln(y_k^n)$$

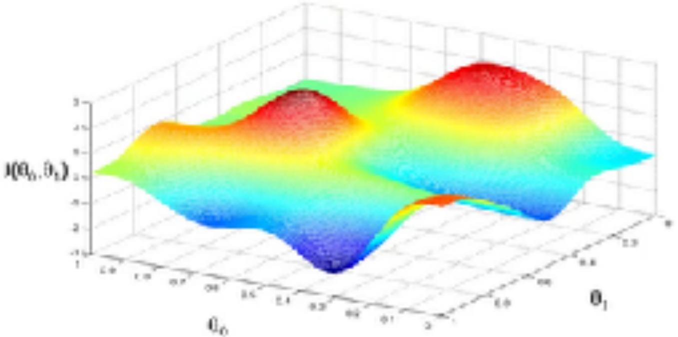
Cross entropy loss

- Measures “how good” our classification is over N training examples

Training a Neural Network

We have to define a cost function:

$$E = -\sum_{n}^{N} \sum_{k=1}^{C} \hat{y}_k^n \ln(y_k^n)$$

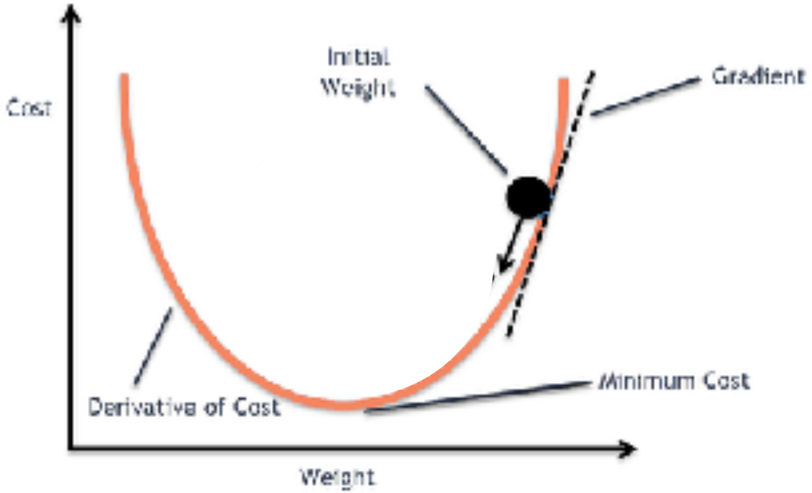


Cross entropy loss

- Measures “how good” our classification is over N training examples

Gradient descent

- The building block of all neural networks

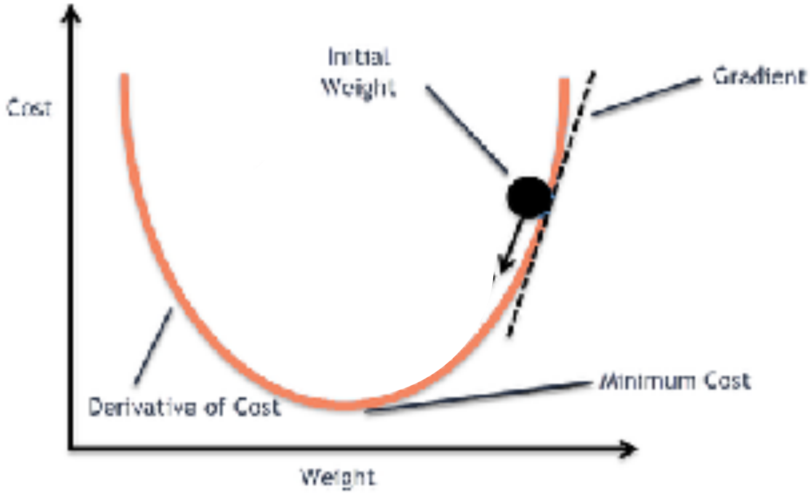


Gradient descent

- The building block of all neural networks
- Minimize the objective function

$$w_{t+1} = w_t - \alpha \frac{\partial C^n(w)}{\partial \theta}$$

- alpha: learning rate

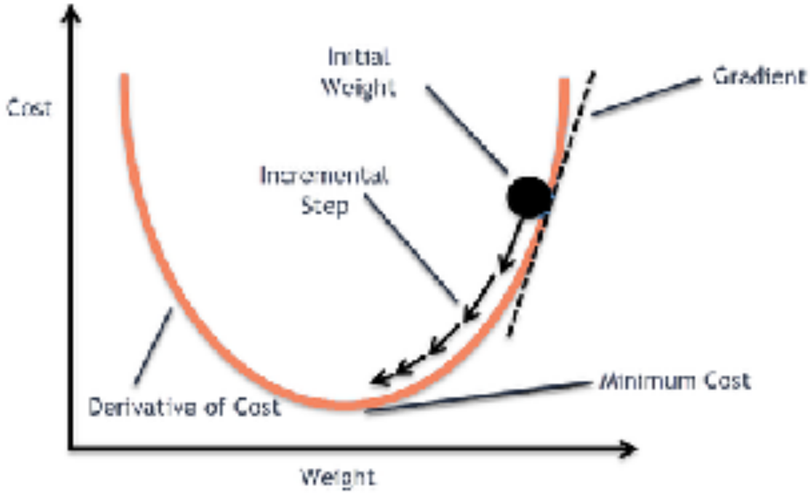


Gradient descent

- The building block of all neural networks
- Minimize the objective function

$$w_{t+1} = w_t - \alpha \frac{\partial C^n(w)}{\partial \theta}$$

- alpha: learning rate



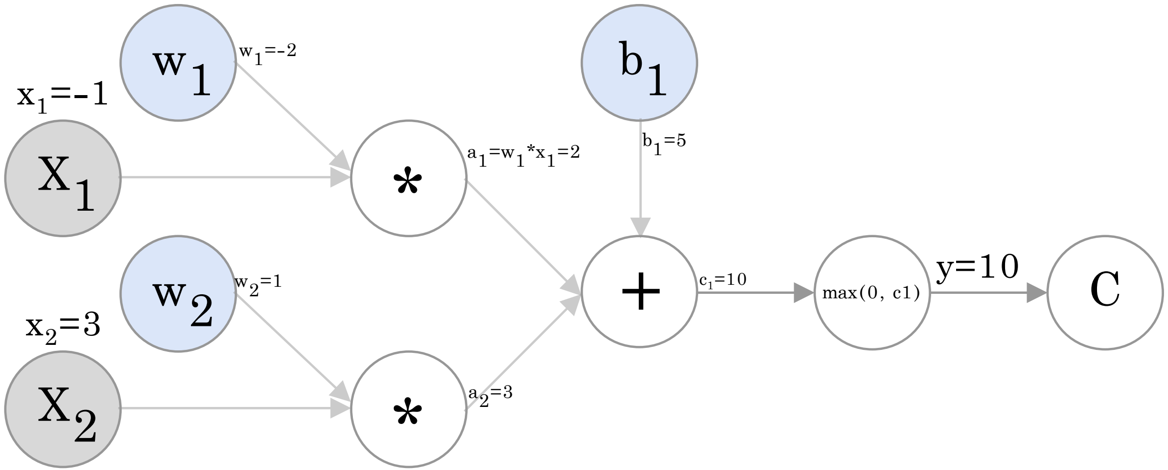
Backpropagation example

Let's say:

$$C = (y - \hat{y})^3$$

And:

$$\hat{y} = 5$$



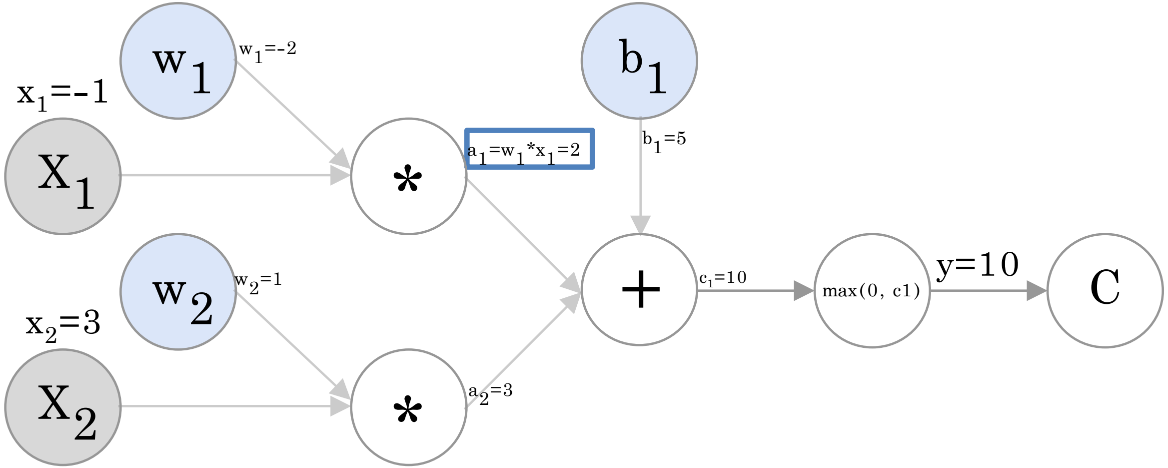
Backpropagation example

Let's say:

$$C = (y - \hat{y})^3$$

And:

$$\hat{y} = 5$$



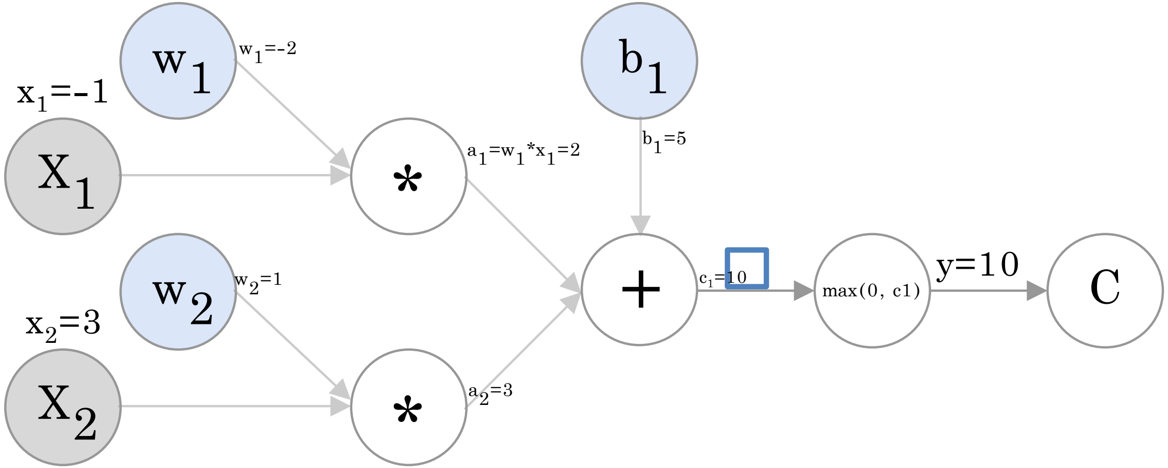
Backpropagation example

Let's say:

$$C = (y - \hat{y})^3$$

And:

$$\hat{y} = 5$$



NTNU

Backpropagation example

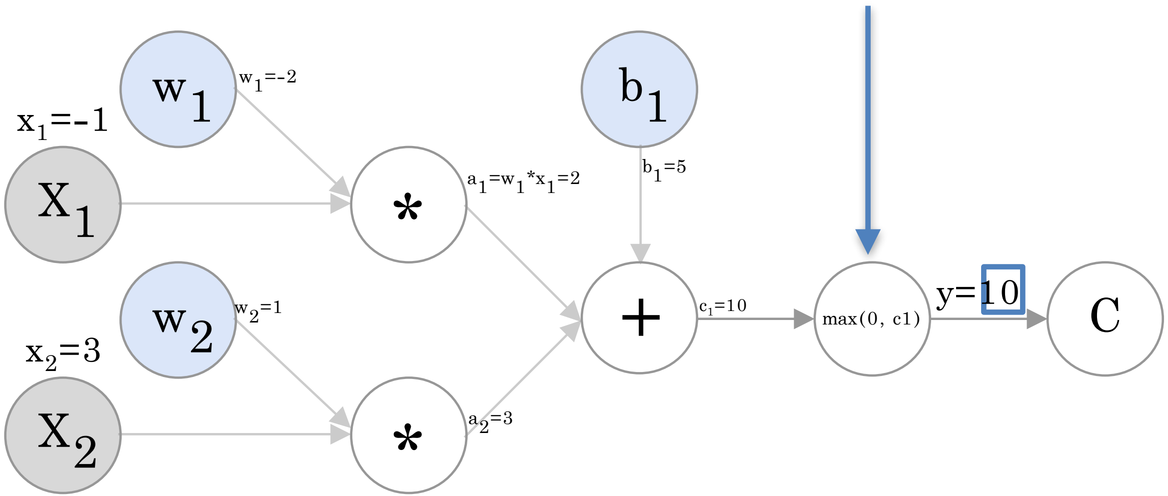
Let's say:

$$C = (y - \hat{y})^3$$

And:

$$\hat{y} = 5$$

ReLU activation function



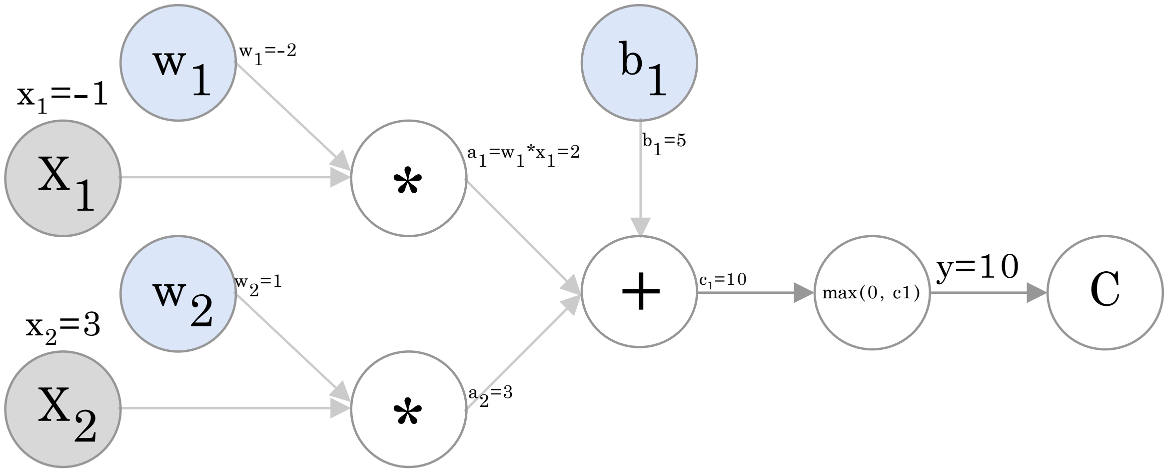
Backpropagation example

Let's say:

$$C = (y - \hat{y})^3$$

And:

$$\hat{y} = 5$$

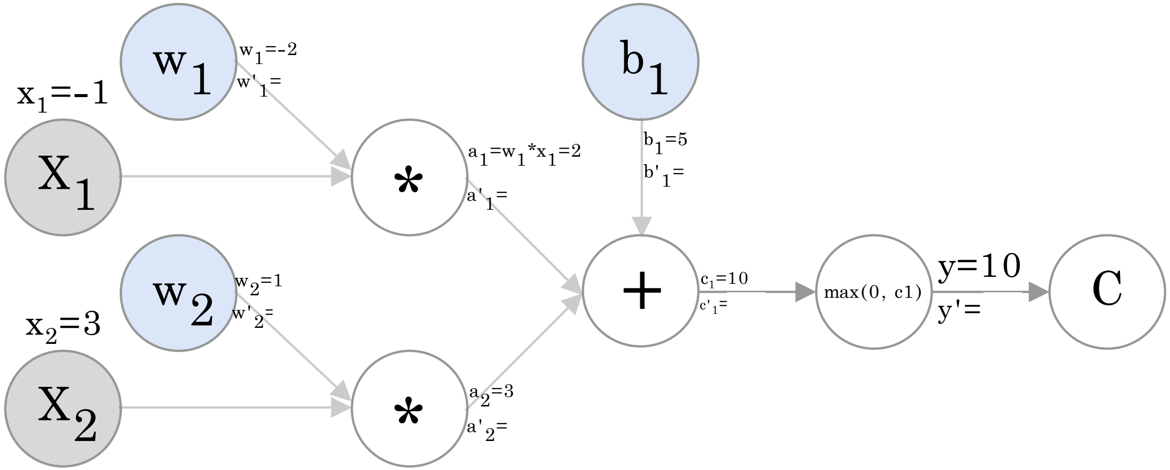


C = 125

Backpropagation example

We know that

1. $\frac{\partial C}{\partial y} = 3 * (y - \hat{y})^2$

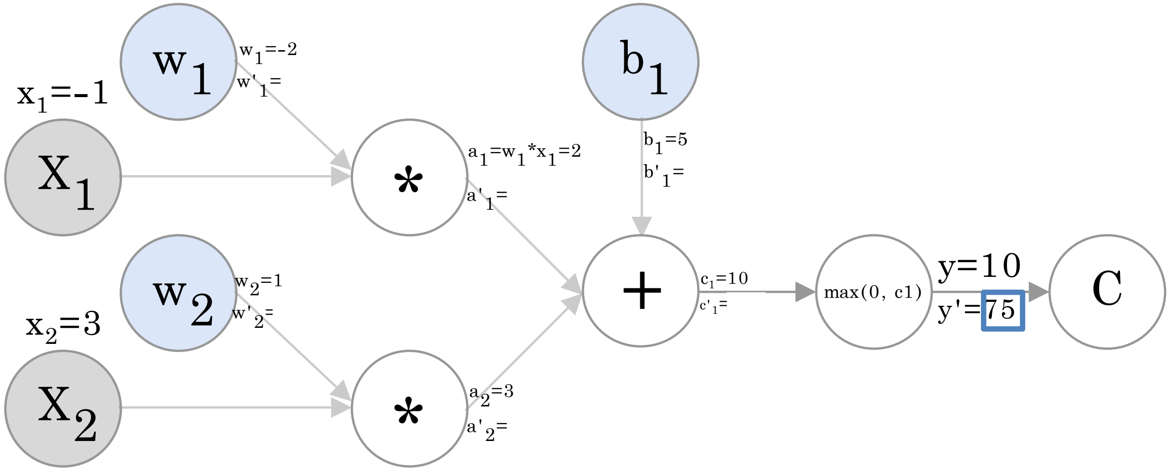


$C = (y - \hat{y})^3, \hat{y} = 5$

Backpropagation example

We know that

1. $\dfrac{\partial C}{\partial y} = 3 * (y - \hat{y})^2 = 75$



$C = (y - \hat{y})^3, \hat{y} = 5$

NTNU

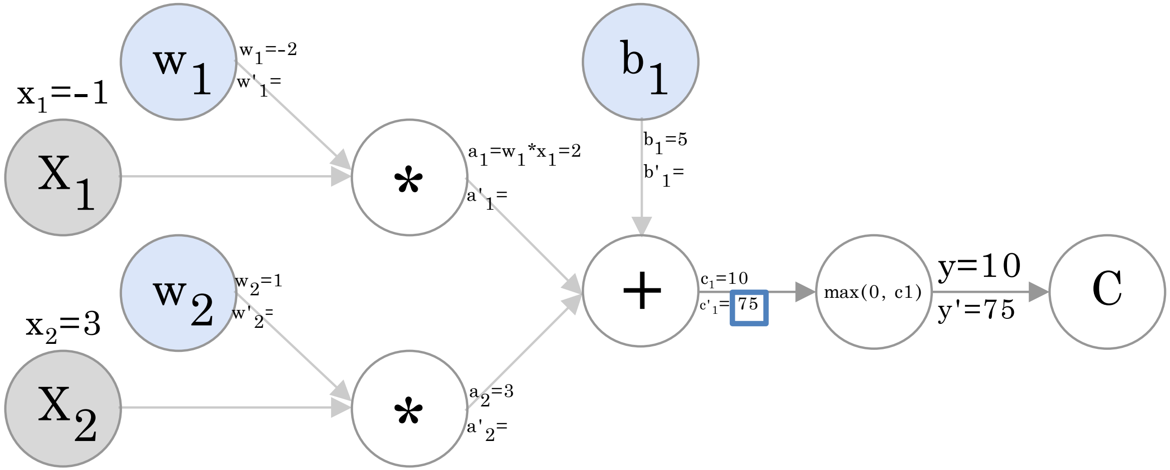
Backpropagation example

We know that

1. $\dfrac{\partial C}{\partial y} = 3 * (y - \hat{y})^2 = 75$

Then by using chain rule:

2. $\dfrac{\partial C}{\partial c_1} = \dfrac{\partial C}{\partial y}\dfrac{\partial y}{\partial c_1} = 75 * 1 = 75$



$C = (y - \hat{y})^3, \hat{y} = 5$

Backpropagation example

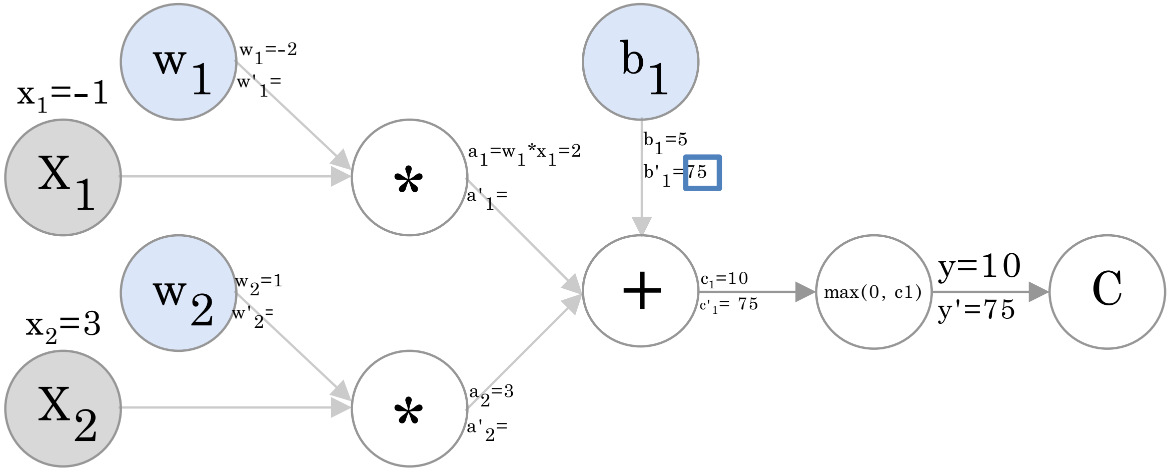
We know that

1. $\dfrac{\partial C}{\partial y} = 3 * (y - \hat{y})^2 = 75$

Then by using chain rule:

2. $\dfrac{\partial C}{\partial c_1} = \dfrac{\partial C}{\partial y}\dfrac{\partial y}{\partial c_1} = 75 * 1 = 75$

3. $\dfrac{\partial C}{\partial b_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial b_1} = 75 * 1 = 75$



$C = (y - \hat{y})^3, \hat{y} = 5$

NTNU

Backpropagation example

We know that

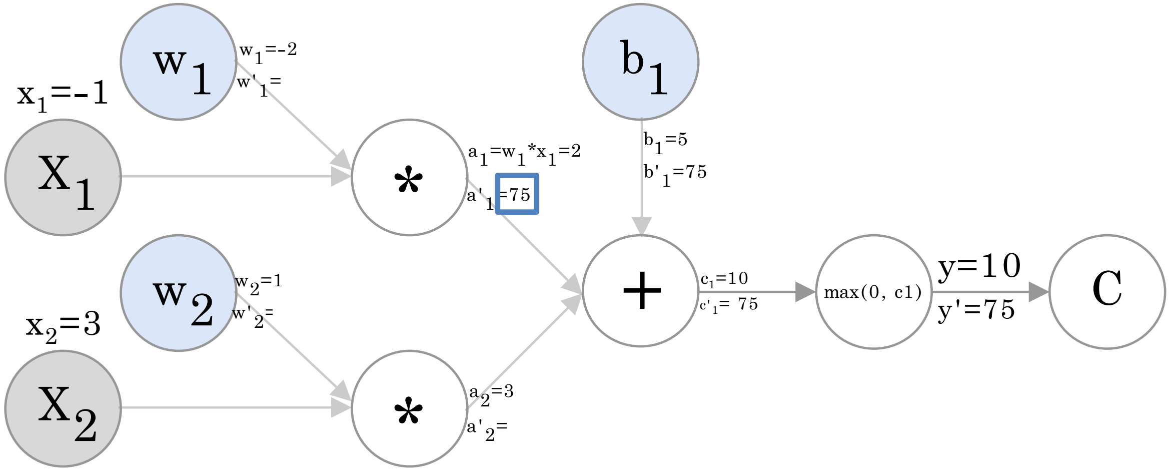
1. $\dfrac{\partial C}{\partial y} = 3 * (y - \hat{y})^2 = 75$

Then by using chain rule:

2. $\dfrac{\partial C}{\partial c_1} = \dfrac{\partial C}{\partial y}\dfrac{\partial y}{\partial c_1} = 75 * 1 = 75$

3. $\dfrac{\partial C}{\partial b_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial b_1} = 75 * 1 = 75$

4. $\dfrac{\partial C}{\partial a_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial a_1} = 75 * 1 = 75$



$C = (y - \hat{y})^3, \hat{y} = 5$

Backpropagation example

We know that

1. $\dfrac{\partial C}{\partial y} = 3 * (y - \hat{y})^2 = 75$

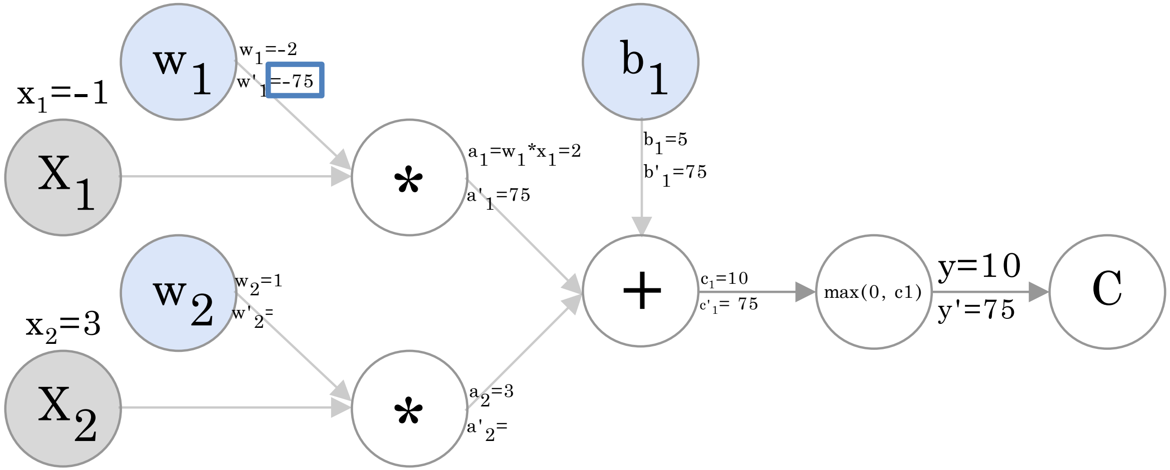
Then by using chain rule:

2. $\dfrac{\partial C}{\partial c_1} = \dfrac{\partial C}{\partial y}\dfrac{\partial y}{\partial c_1} = 75 * 1 = 75$

3. $\dfrac{\partial C}{\partial b_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial b_1} = 75 * 1 = 75$

4. $\dfrac{\partial C}{\partial a_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial a_1} = 75 * 1 = 75$

5. $\dfrac{\partial C}{\partial w_1} = \dfrac{\partial C}{\partial a_1}\dfrac{\partial w_1}{\partial a_1} = 75x_1 = -75$



$C = (y - \hat{y})^3, \hat{y} = 5$

Backpropagation example

We know that

1. $\dfrac{\partial C}{\partial y} = 3 * (y - \hat{y})^2 = 75$

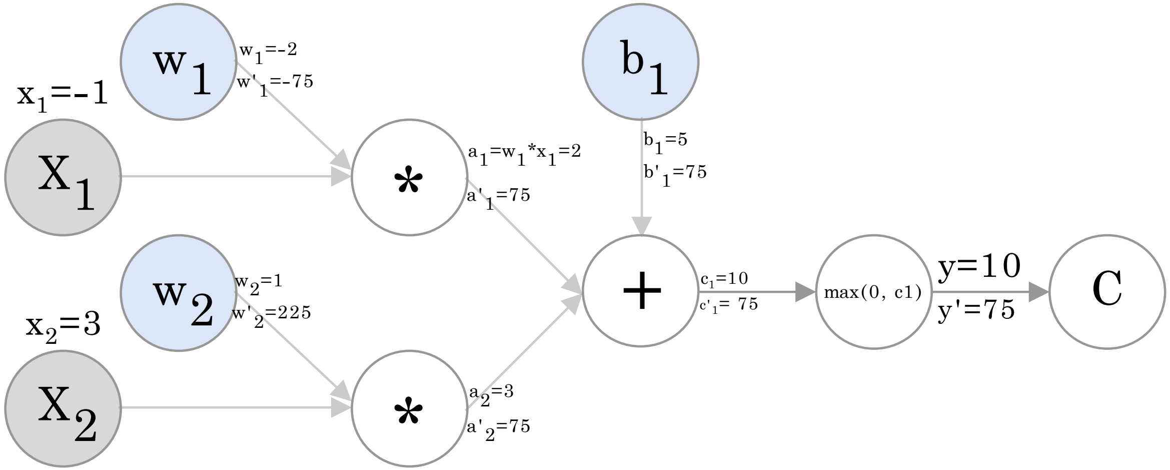
Then by using chain rule:

2. $\dfrac{\partial C}{\partial c_1} = \dfrac{\partial C}{\partial y}\dfrac{\partial y}{\partial c_1} = 75 * 1 = 75$

3. $\dfrac{\partial C}{\partial b_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial b_1} = 75 * 1 = 75$

4. $\dfrac{\partial C}{\partial a_1} = \dfrac{\partial C}{\partial c_1}\dfrac{\partial c_1}{\partial a_1} = 75 * 1 = 75$

5. $\dfrac{\partial C}{\partial w_1} = \dfrac{\partial C}{\partial a_1}\dfrac{\partial w_1}{\partial a_1} = 75x_1 = -75$

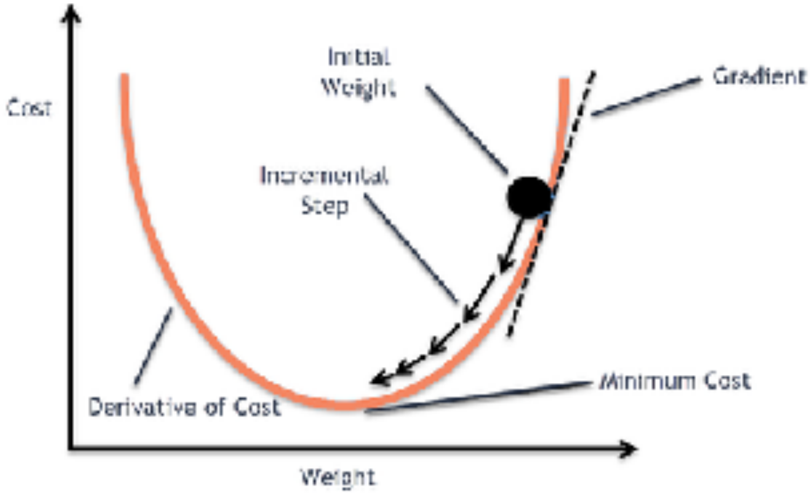


$C = (y - \hat{y})^3, \hat{y} = 5$

Gradient descent

Our update rule ($\alpha = .01$):

$$w_1 = w_1 - \alpha \frac{\partial C}{\partial w_1}$$



NTNU

Gradient descent

Our update rule ($\alpha = .01$):

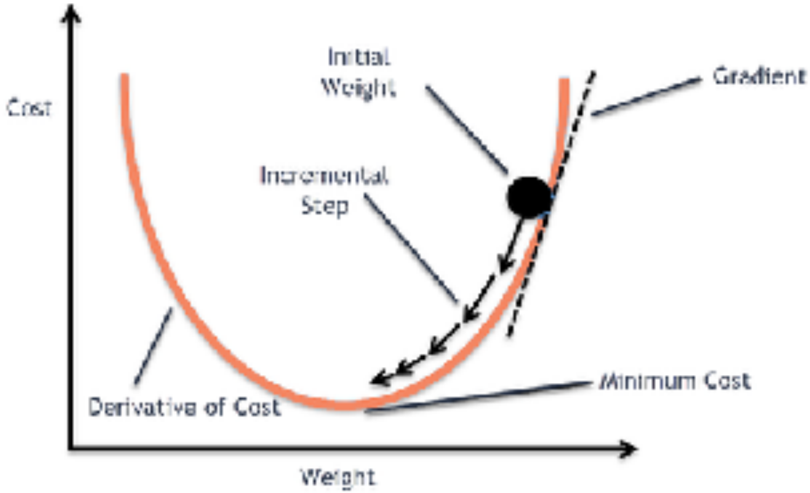
$$w_1 = w_1 - \alpha \frac{\partial C}{\partial w_1}$$

We know:

$$\frac{\partial C}{\partial w_1} = -75, \text{ and } w_1 = -2$$

Then,

$$w_1 = -2 - 0.01 \cdot (-75) = -1.25$$

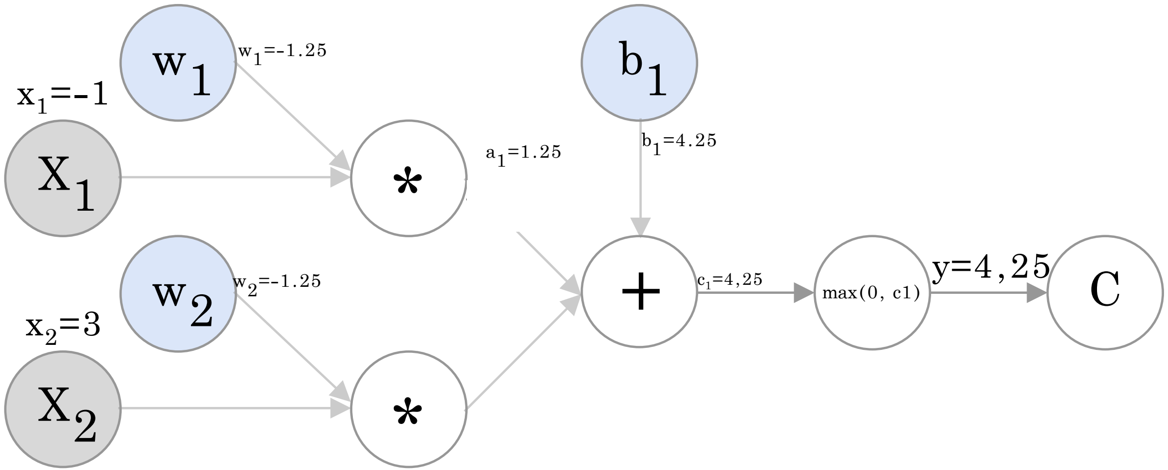


Gradient descent - Did we get closer?

Yes!

New prediction:

$$y = 4.25$$



NTNU

Neural Network/Machine Learning Concepts

Hyperparameters:

- Parameters we set before training.
  - Learning rate
  - Batch size

Neural Network/Machine Learning Concepts

Hyperparameters:

- Parameters we set before training.
  - Learning rate
  - Batch size

Minibatch:

- Instead of updating weights on a single training example, we take the average over a minibatch

Neural Network/Machine Learning Concepts

Hyperparameters:

- Parameters we set before training.
  - Learning rate
  - Batch size

Minibatch:

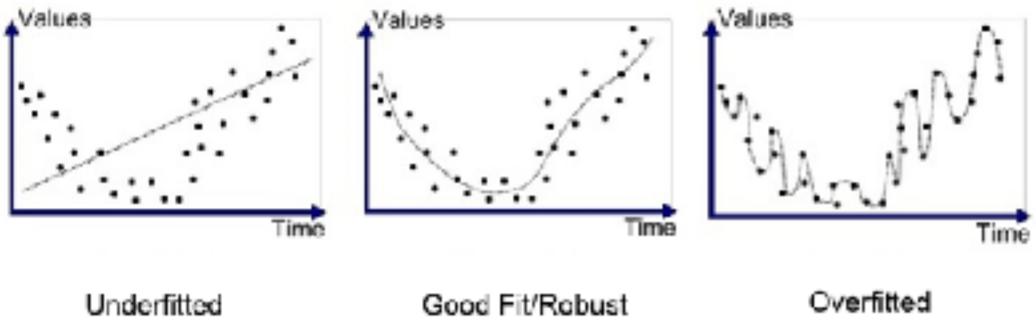
- Instead of updating weights on a single training example, we take the average over a minibatch

Datasets:

- Train: Used only for training
- Validation: Used to validate model / tune hyperparameters
- Test: Final evaluation of model (should not be used frequently)

Overfitting

- Model memorise training points
- Does not generalise to the underlying function



Neural Network/Machine Learning Concepts

Hyperparameters:

- Parameters we set before training.
    - Learning rate
    - Batch size

Minibatch:

- Instead of updating weights on a single training example, we take the average over a minibatch

Datasets:

- Train: Used only for training
- Validation: Used to validate model / tune hyperparameters
- Test: Final evaluation of model (should not be used frequently)

Gradient Descent - In Numpy

```
def gradient_decent(X, outputs, targets, weights):
  N = X.shape[0]
  for i in range(weights.shape[0]):
    dw_i = - 2 * (targets - outputs) * X[:, i:i+1]

    dw_i = dw_i.mean(axis=0)
    weights[i] = weights[i] - learning_rate * dw_i
  return weights

```

Gradient Descent - In Numpy

For every weight

```
def gradient_decent(X, outputs, targets, weights):
  N = X.shape[0]
  for i in range(weights.shape[0]):
    dw_i = - 2 * (targets - outputs) * X[:, i:i+1]

    dw_i = dw_i.mean(axis=0)
    weights[i] = weights[i] - learning_rate * dw_i
  return weights
```

Gradient Descent - In Numpy

Compute gradient

```
def gradient_decent(X, outputs, targets, weights):
  N = X.shape[0]
  for i in range(weights.shape[0]):
    dw_i = - 2 * (targets - outputs) * X[:, i:i+1]

    dw_i = dw_i.mean(axis=0)
    weights[i] = weights[i] - learning_rate * dw_i
  return weights

```

Gradient Descent - In Numpy

Find mean over all training examples

```python
def gradient_decent(X, outputs, targets, weights):
  N = X.shape[0]
  for i in range(weights.shape[0]):
    dw_i = - 2 * (targets - outputs) * X[:, i:i+1]

    dw_i = dw_i.mean(axis=0)
    weights[i] = weights[i] - learning_rate * dw_i
  return weights

```

Gradient Descent - In Numpy

Perform update

```
def gradient_decent(X, outputs, targets, weights):
  N = X.shape[0]
  for i in range(weights.shape[0]):
    dw_i = - 2 * (targets - outputs) * X[:, i:i+1]

    dw_i = dw_i.mean(axis=0)
    weights[i] = weights[i] - learning_rate * dw_i
  return weights

```

Gradient Descent - In Numpy

This is a lot of code for a single layer!

And it's <u>extremely</u> slow

```python
def gradient_decent(X, outputs, targets, weights):
  N = X.shape[0]
  for i in range(weights.shape[0]):
    dw_i = - 2 * (targets - outputs) * X[:, i:i+1]

    dw_i = dw_i.mean(axis=0)
    weights[i] = weights[i] - learning_rate * dw_i
  return weights

```

Instead, use a framework

Why?

- Quickly implement and test ideas
- Automatically compute gradients
- Run it efficient

Frameworks

Caffe2
(Facebook/
UC Berkeley)

Torch
(NYU / Facebook)

Theano
(U Montreal)

Frameworks

Caffe2
(Facebook/
UC Berkeley)

PyTorch
(Facebook)

Torch
(NYU / Facebook)

Theano
(U Montreal)

Tensorflow
(Google)

Frameworks

Caffe2
(Facebook/
UC Berkeley)

PyTorch
(Facebook)

Torch
(NYU / Facebook)

Theano
(U Montreal)

Tensorflow
(Google)

Keras
(TF wrapper)

Less popular stuff

PaddlePaddle
(Baidu)

Chainer

MXNet
(Amazon)

CNTK
(Microsoft)

Others

Frameworks

Caffe2
(Facebook/
UC Berkeley)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

Tensorflow
(Google)

These are the most popular

Less popular stuff

PaddlePaddle
(Baidu)

Chainer

MXNet
(Amazon)

CNTK
(Microsoft)

Others

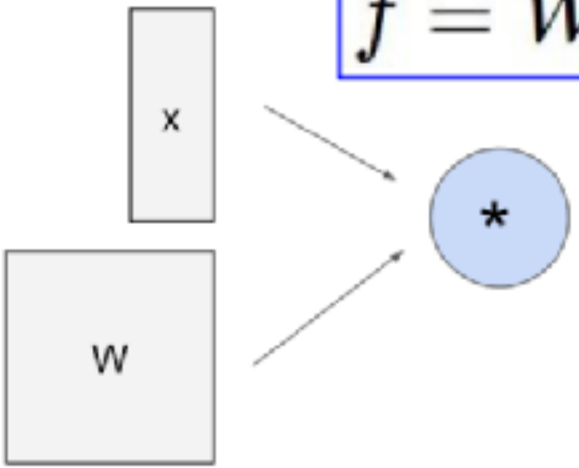
Neural Networks = Directed Acyclic Graphs

$$\boxed{f = Wx} \qquad \boxed{L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}$$

Neural Networks = Directed Acyclic Graphs

$$\boxed{f = Wx}$$

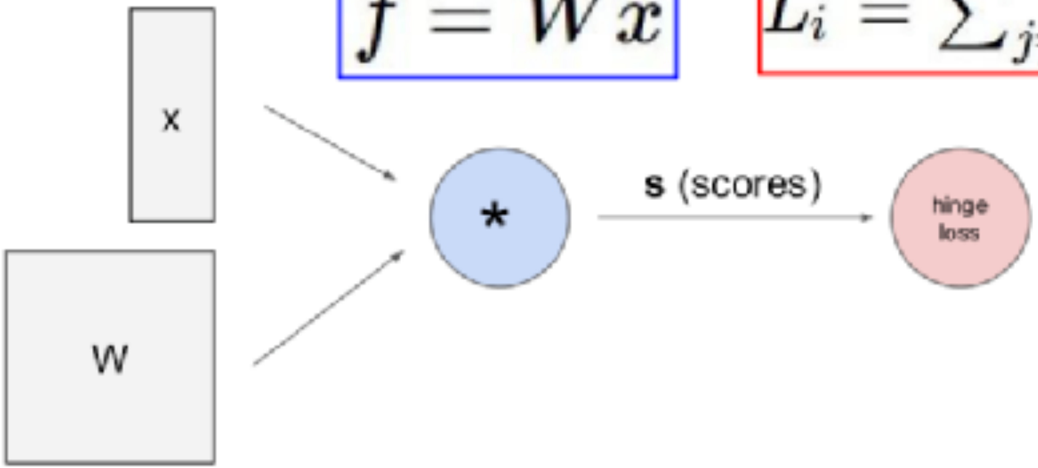
$$\boxed{L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}$$



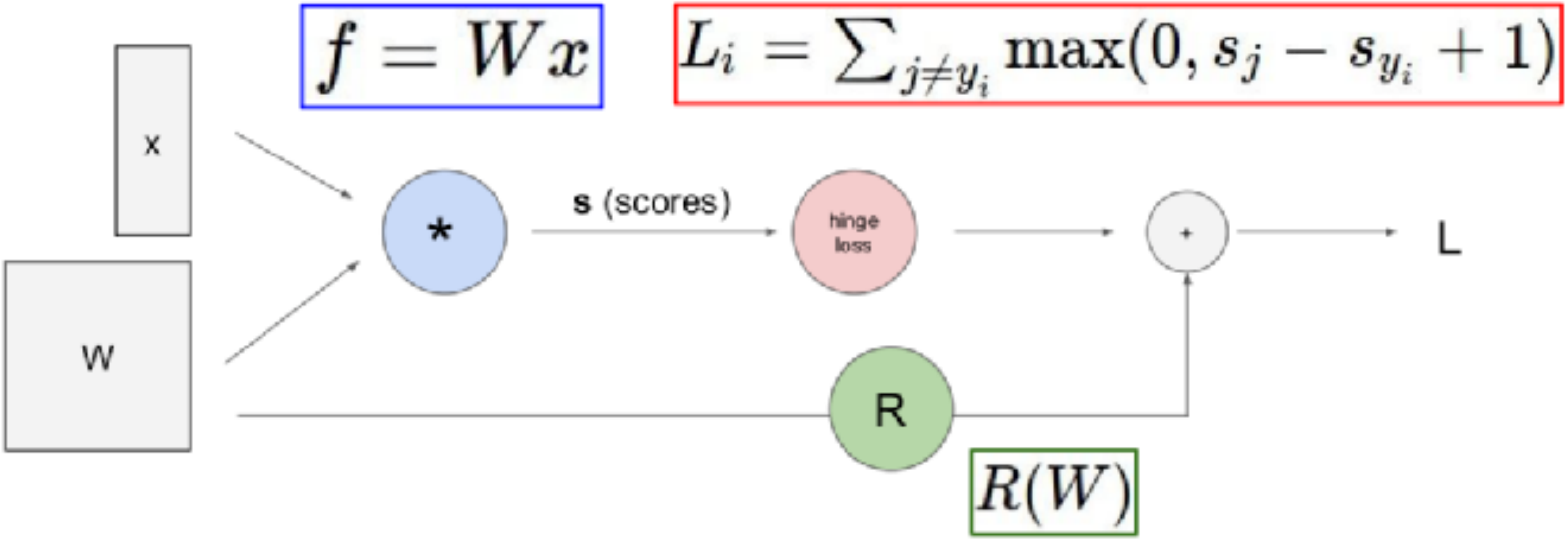
Neural Networks = Directed Acyclic Graphs

$$\boxed{f = Wx}$$

$$\boxed{L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}$$

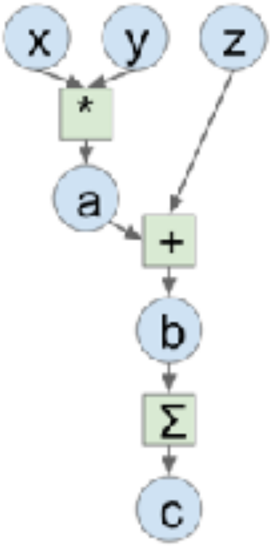


Neural Networks = Directed Acyclic Graphs



 Slide source: CS231n Lecture 8 NTNU

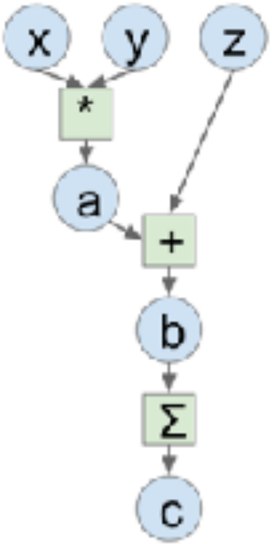
Computational Graph



Computational Graph

Numpy

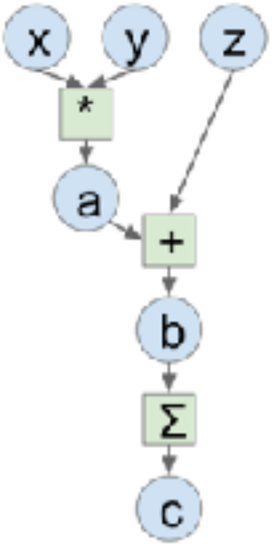
```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



Computational Graph

Numpy

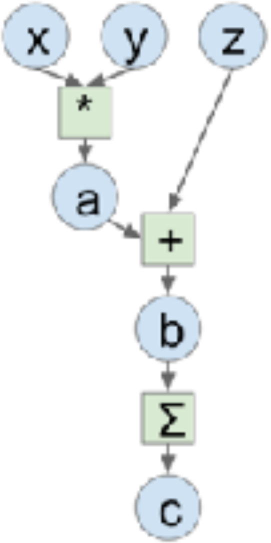
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Computational Graph

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



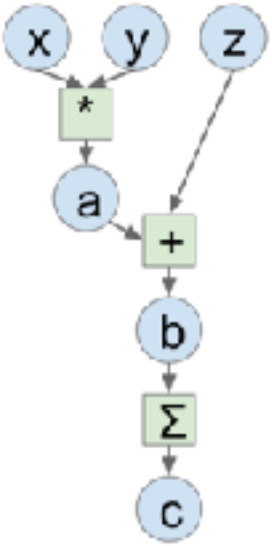
Good:
- Simple, clean API

Bad:
- Have to compute gradients ourself
- Can’t run on GPU

Computational Graph

Numpy

```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```python
import torch
N, D = 3, 4
x = torch.randn(N, D,
y = torch.randn(N, D)
z = torch.randn(N, D)
a = x*y
b = a + z
c = torch.sum(b)
```

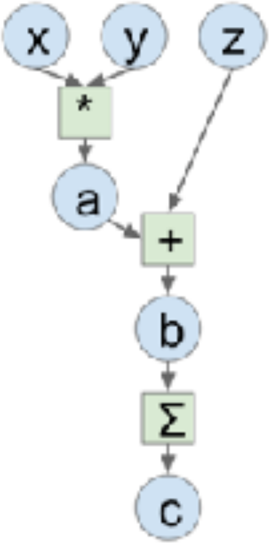
Forward Pass

 Slide source: [CS231n Lecture 8]

Computational Graph

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



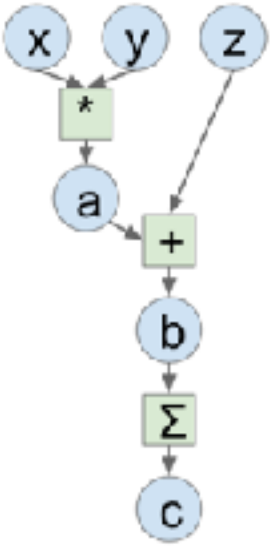
PyTorch

```
import torch
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)
a = x*y
b = a + z
c = torch.sum(b)
print(x.grad)
c.backward()
print(x.grad)
```

Computational Graph

Numpy

```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```python
import torch
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)
a = x*y
b = a + z
c = torch.sum(b)
print(x.grad)
c.backward()
print(x.grad)
```

```
None
tensor([[ 1.9707,  1.5089,  1.5041, -0.5507],
        [-0.1295, -0.6893, -0.2087, -0.0419],
        [ 0.0277,  1.4500,  1.3814,  0.3493]])
```

PyTorch
(in-depth)

Håkon Hukkelås | hakon.hukkelas@ntnu.no

Pytorch: Fundamental Concepts

- Tensor: Like a numpy array, but can run on GPUs
- Module: A neural network layer; stores states and learnable weights

Images in Pytorch

In numpy, an image has shape:

- [Number of images, height, width, #colors]

In Pytorch, an image has shape:

- [Number of images, #colors, height, width]

Pytorch: Tensors

Example: A 2-layer neural network

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Initialize the weights randomly

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Do want to save gradients w.r.t weights

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
    for (X,Y) in dataloader:
        X = pre_process_image(X)
        # forward pass
        z_j = X.mm(w1)
        a_j = torch.sigmoid(z_j)
        z_k = a_j.mm(w2)
        y_k = torch.softmax(z_k, dim=1)
        # Compute loss
        loss = loss_function(y_k, Y)
        losses.append(loss)
        # Backpropagation
        loss.backward()
        with torch.no_grad():
            w1 -= learning_rate * w1.grad
            w2 -= learning_rate * w2.grad
            w1.grad.zero_()
            w2.grad.zero_()
```

Pytorch: Tensors

Define our loss function:
Cross Entropy Loss

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```

Pytorch: Tensors

Pytorch uses “dataloaders” to efficiently load datasets

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
    for (X,Y) in dataloader:
        X = pre_process_image(X)
        # forward pass
        z_j = X.mm(w1)
        a_j = torch.sigmoid(z_j)
        z_k = a_j.mm(w2)
        y_k = torch.softmax(z_k, dim=1)
        # Compute loss
        loss = loss_function(y_k, Y)
        losses.append(loss)
        # Backpropagation
        loss.backward()
        with torch.no_grad():
            w1 -= learning_rate * w1.grad
            w2 -= learning_rate * w2.grad
            w1.grad.zero_()
            w2.grad.zero_()
```

Pytorch: Tensors

Bias trick and normalization

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```

Pytorch: Tensors

Define the forward pass

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
    for (X,Y) in dataloader:
        X = pre_process_image(X)
        # forward pass
        z_j = X.mm(w1)
        a_j = torch.sigmoid(z_j)
        z_k = a_j.mm(w2)
        y_k = torch.softmax(z_k, dim=1)
        # Compute loss
        loss = loss_function(y_k, Y)
        losses.append(loss)
        # Backpropagation
        loss.backward()
        with torch.no_grad():
            w1 -= learning_rate * w1.grad
            w2 -= learning_rate * w2.grad
            w1.grad.zero_()
            w2.grad.zero_()
```

Pytorch: Tensors

Compute the loss

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```

Pytorch: Tensors

Backpropagate the loss

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```

Pytorch: Tensors

Make gradient step on weights

torch.no_grad() means “don’t build a computational graph here

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```

Pytorch: torch.nn

Higher lever wrapper for defining neural networks

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```


This is a running example! Do not implement your network like this!

Pytorch: torch.nn

Higher lever wrapper for defining neural networks

Define each layer in model.

Each layer is a nn.Module() object, containing learnable weights.

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Pytorch: torch.nn

Changed loss function to nn.CrossEntropyLoss.

This includes the softmax!

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Pytorch: torch.nn

Simplifies our forward pass!

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Pytorch: torch.nn

Compute loss and perform backward pass

Each weight in model has requires_grad=True by default

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Pytorch: torch.nn

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Perform our gradient step
(and disable gradients)

Pytorch: torch.optim

Final piece you need to know

Implements **S**tochastic **G**radient **D**escent

Input: our learnable parameters (weights + biases) + learning rate

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

This is how its supposed to be!

Pytorch: torch.optim

Perform gradient step and reset the gradients

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: nn.Module

A PyTorch **Module** is a neural network layer; it inputs and outputs tensors

Can contain weights or other modules

Required for more complex layers

Easily customizable layers

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: nn.Module

Start with defining the model

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: nn.Module

Called when we initialize our model

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: nn.Module

Called when we perform forward pass

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                             lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: DataLoaders

A **DataLoader** wraps a dataset and provides features such as:

- Data augmentation
- Data pre-processing
- mini-batch shuffling and splitting

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: DataLoaders

Iterates over each batch in a epoch

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Deep Learning Hardware

CPU, GPU, TPU

We have two hardware choices:

- NVIDIA GPU
- Google Tensor Processing Unit (TPU)

- AMD? Really not used.

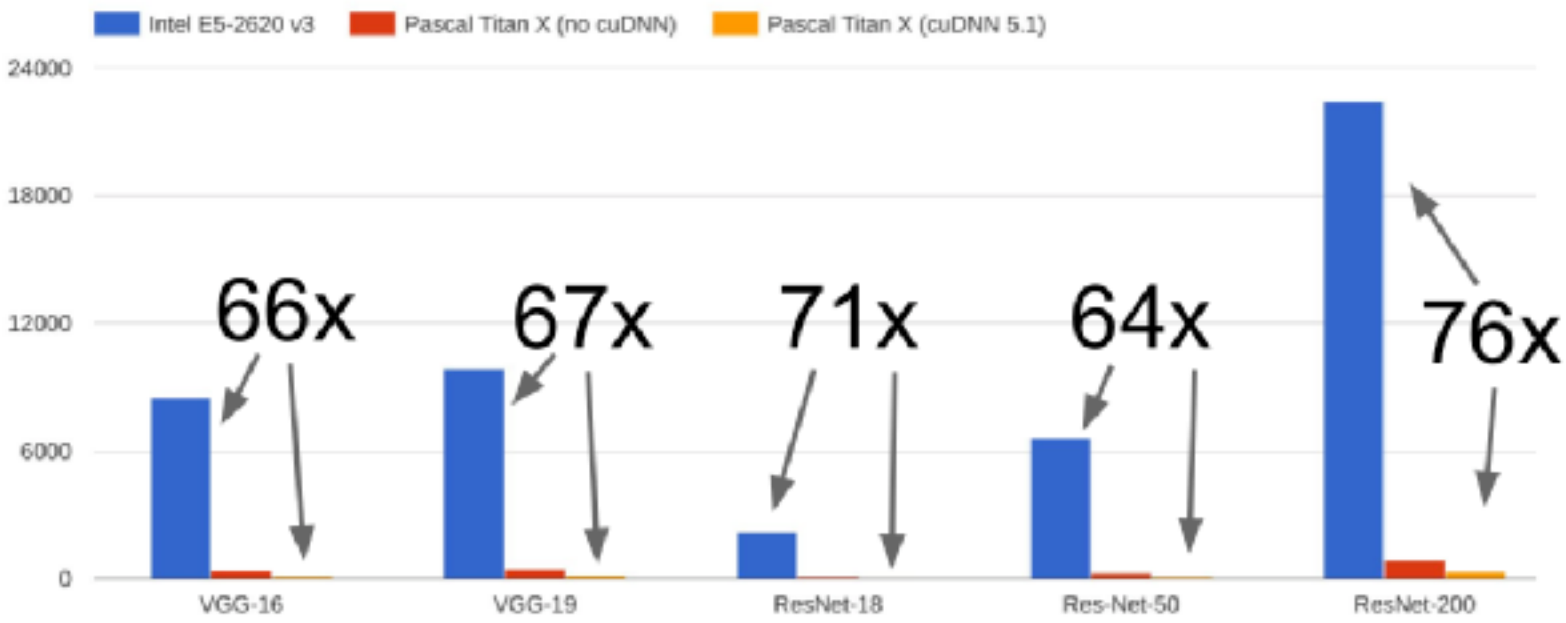
GPU vs CPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32

GPU: thousands of "dumb" cores: Great for parallel tasks

Neural Networks: "Only" matrix multiplication, easy in parallel

GPU vs CPU for CNNs



 Slide source: CS231n Lecture 8

GPU vs CPU: ResNet-200

- Forward pass:
  - Pascal Titan X: 104ms
  - CPU: Dual Xeon E5-2630 v3: 8,666ms (83x slower)
- Backward pass:
  - Pascal Titan X: 191 ms
  - CPU: Dual Xeon E5-2630 v3: 13,758 ms (72x slower)

Pytorch: On GPU

Utilizing GPU resources is simple!

Note, it is not required for assignment 1.
Recommended for assignment 2!

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

.cuda() transfers weights/tensors to GPU VRAM

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

.cuda() transfers weights/tensors to GPU VRAM

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

.cuda() transfers weights/tensors to GPU VRAM

CAREFUL: Calling .cuda() without a NVIDIA GPU available will cause error!

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

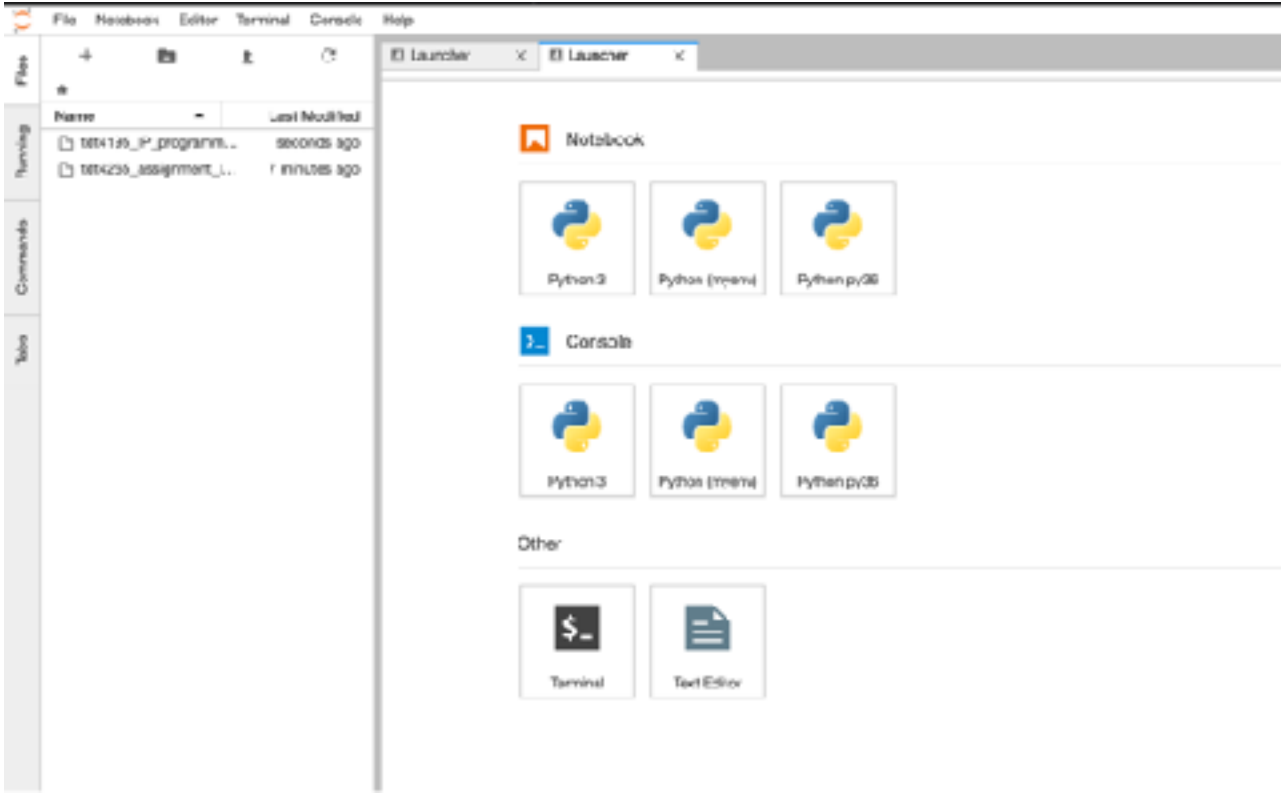
Instead: Implement a to_cuda() function

```python
def to_cuda(elements):
    if torch.cuda.is_available():
        if type(elements) == tuple or type(elements) == list:
            return [x.cuda() for x in elements]
        return elements.cuda()
    return elements
```

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = to_cuda(TwoLayerNet())

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = to_cuda([X_batch, Y_batch])
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Cool features & resources

Jupyter Notebook/Lab



Multi-layer neural networks

- What can a 1-layer network predict?
  https://bit.ly/2HO6iln

Multi-layer neural networks

- What can a 1-layer network predict?
  https://bit.ly/2HO6iln
- Answer: Only linearly separable functions

Multi-layer neural networks

- What can a 2-layer network predict? (1 hidden layer) https://bit.ly/2SndBEQ

Multi-layer neural networks

- What can a 2-layer network predict? (1 hidden layer) https://bit.ly/2SndBEQ
- Answer: Theoretically, everything possible.

Pytorch model zoo

Pytorch comes with a large amount of state-of-the-art models

Classification:

- https://pytorch.org/docs/stable/torchvision/models.html

Detection:

- https://pytorch.org/blog/torchvision03/

Google Colab

Google Colab is a jupyter notebook like system

With **FREE** GPU resources!

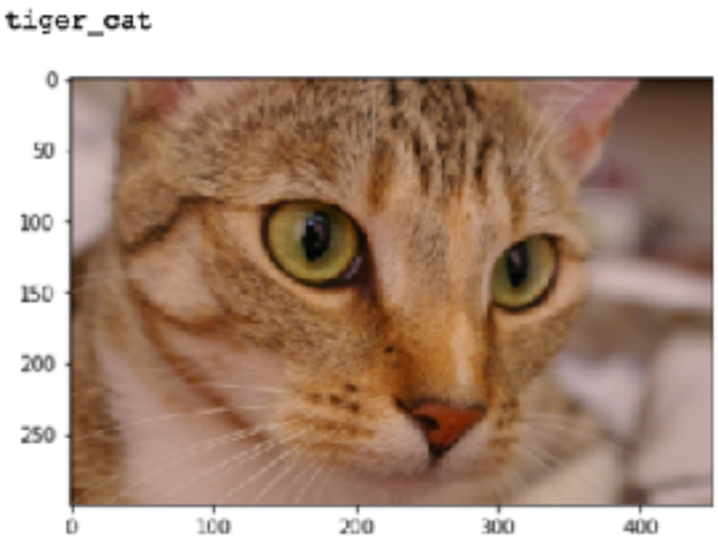
Some cool colabs:

- Detectron2
- DeepPrivay (Shameless plug)
- BigGAN

Pytorch model zoo

```
# Define model
model = torchvision.models.resnet152(pretrained=True)
model = model.eval()

im = skimage.data.chelsea()
im = Image.fromarray(im)
plt.imshow(im)

im = transform(im)
preds = model(im[None])
print(get_imagenet_class(preds))
```

Pytorch model zoo



NTNU