ENGR-E 399/599 & CSCI-B 590
Deep Learning Architecture and Hardware Acceleration

Lab 1

Administrative Lab #1

Lab #1 is released today

- Run Pytorch In Google Colab With Free GPU
- Run Pytorch code following this tutorial

Lab 1 is due on 11:59 pm, 2/21/2023

Submit your code and report to *Canvas*

Overview

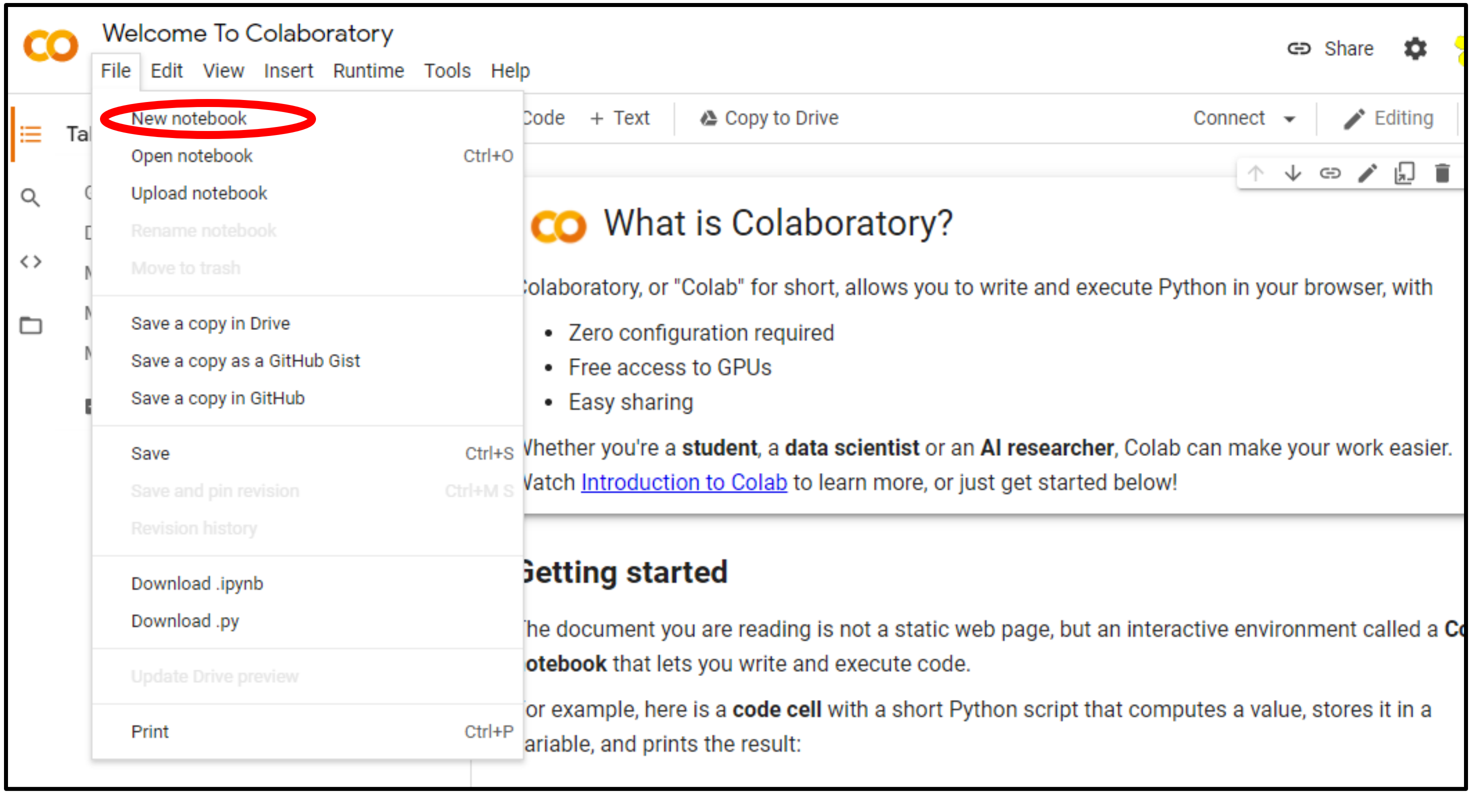
- What is Google Colab
- Getting Started with Colab
- Pytorch
  - ✓ Pytorch Tensors
  - ✓ Tensor operations
  - ✓ CUDA support
  - ✓ *autograd* in Pytorch
  - ✓ Backpropagation
- Lab 1

What is Google Colab

- An online research tool provided by Google for machine learning education and research.
- A free cloud service that offers Jupyter Notebooks via remote servers.
- Users can use GPU and TPU resources from Google to run their arbitrary Python code using Google Colab
- Recommended read resources:
  - ✓ Google’s Colab intro notebook

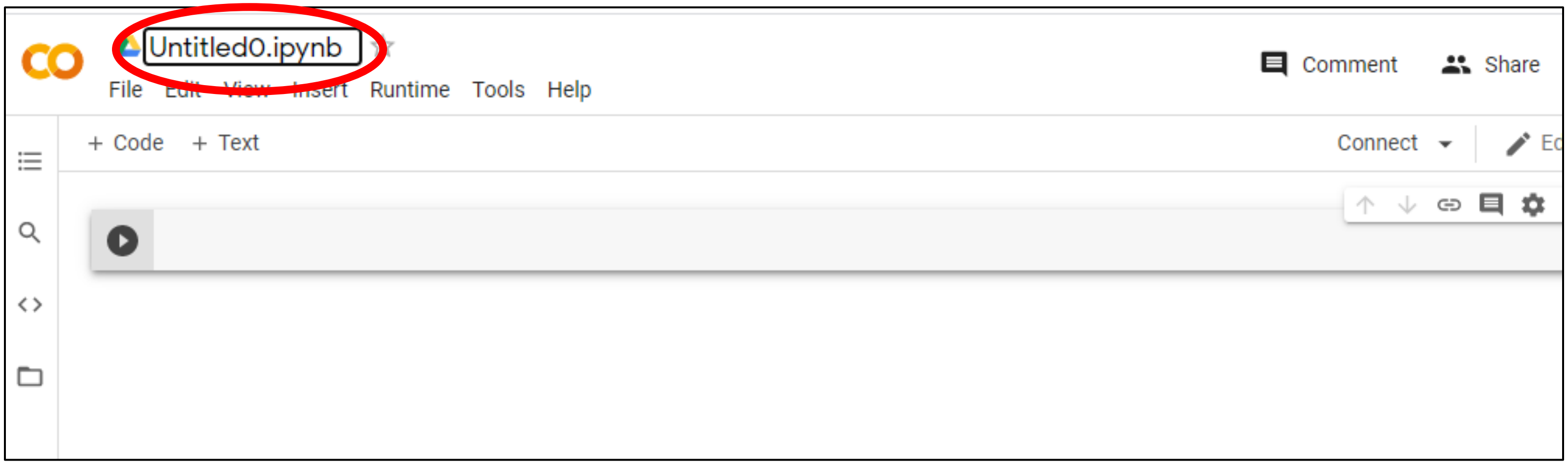
Getting Started with Colab

- Go to [Google Colab]
- Sign in with your Google Account
- Create a new notebook via *File -> New notebook*



Getting Started with Colab

- Click the file name on the top to rename the notebook



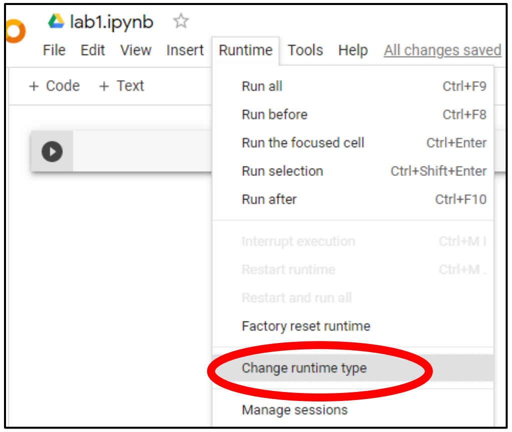
Getting Started with Colab

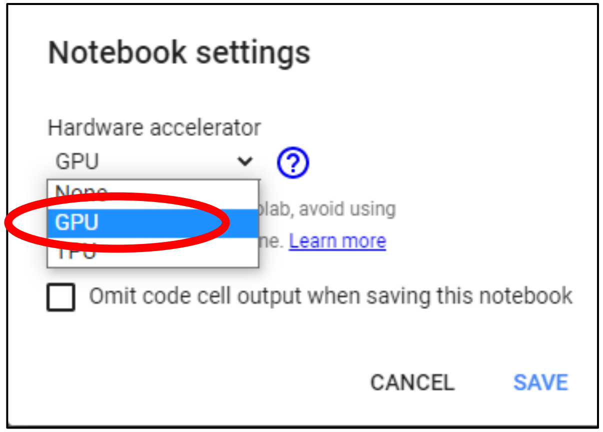
- Setting up GPU, note that

✓You will get 12 hours of execution time: Disk, RAM, CPU Cache, and the Data that is on the virtual machine will get *erased every 12 hours*

✓The session will be disconnected if you are idle for more than 60 minutes

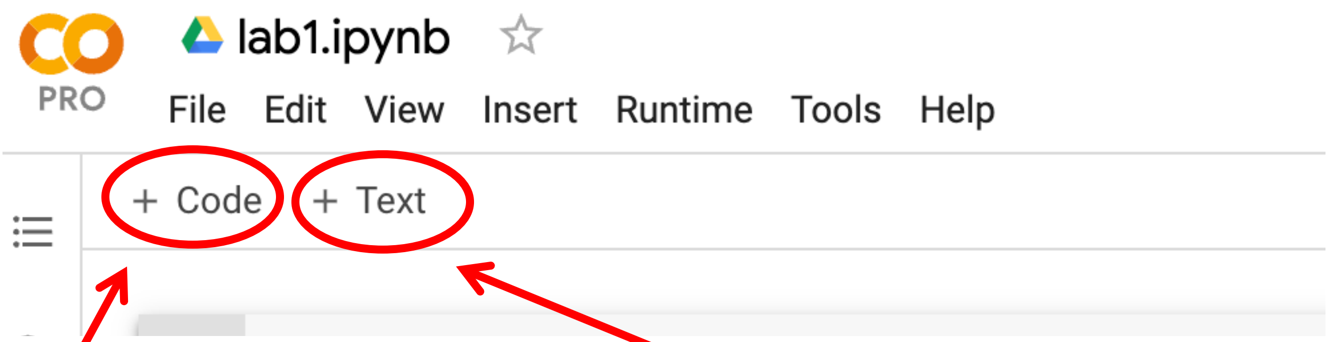
- Enable GPU by going to *Runtime -> Change runtime type -> Hardware accelerator -> GPU*





Getting Started with Colab

- Creating new code cell or text cell.
  - ✓ Code cell allow you to write and run python code.
  - ✓ Text cell allow you to write formatted text using *Markdown* syntax.

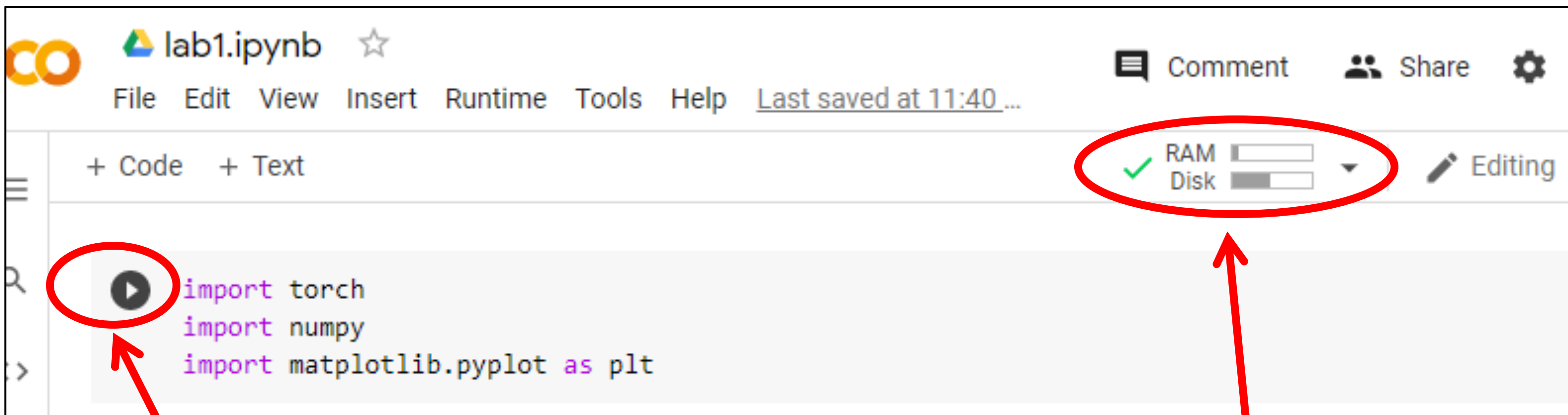


Click it to create a new code cell under the cell chosen

Click it to create a new text cell under the cell chosen

Getting Started with Colab

- Running python code and import required libraries



Click it to run the code

You will see this after importing libs

Overview

- What is Google Colab
- Getting Started with Colab
- Pytorch
  - ✓ Pytorch Tensors
  - ✓ Tensor operations
  - ✓ CUDA support
  - ✓ *autograd* in Pytorch
  - ✓ Backpropagation

Pytorch Tensors

- Tensor: n-dimensional numpy array

  Tensor can be simply regarded as a multidimensional array mathematically. It is the base elements for computing in neural networks.

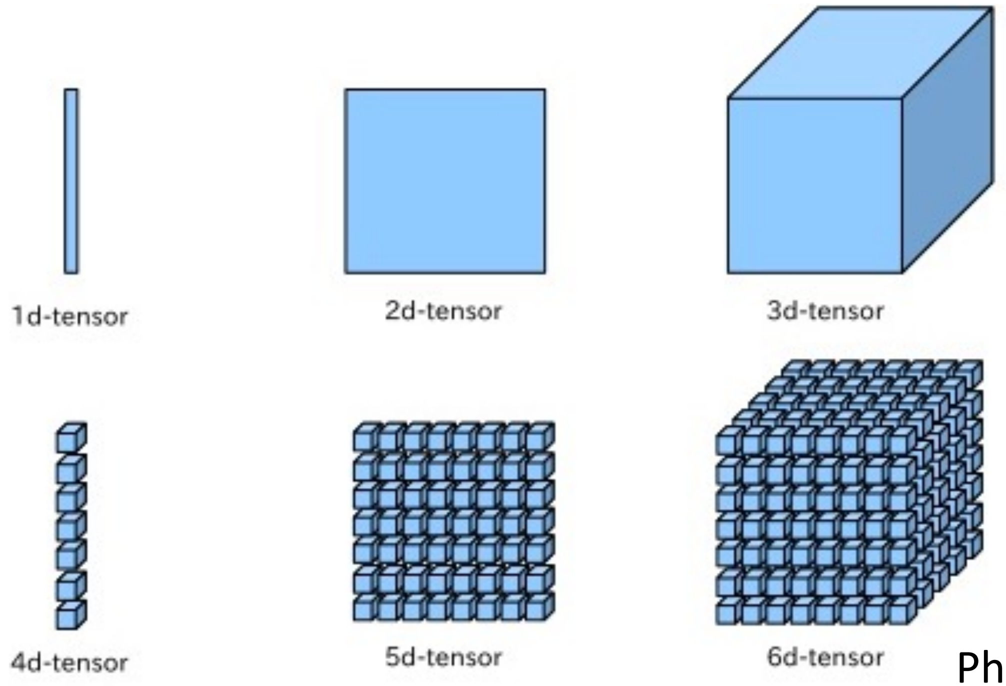


Photo credit to *knoldus*

Pytorch Tensors

- Pytroch Tensors can use GPUs to accelerate their numeric computation compared to numpy tensors.

```
## import required libraries
import time, torch
import numpy as np
## create numpy tensor with size of (10000, 10000)
x = np.random.rand(10000, 10000)
## set start time
start1 = time.time()
## execute multiplication of numpy tensors on CPU
z1 = np.dot(x.T, x)
## print the time cost of the multiplication
print("{} seconds passed ". format(time.time() - start1))
```

```
17.18445920944214 seconds passed
```

Pytorch Tensors

- Pytroch Tensors can use GPUs to accelerate their numeric computation compared to numpy tensors.

```
## create torch tensor on GPU
y = torch.from_numpy(x).cuda()
## set start time
start2 = time.time()
## execute multiplication of torch tensors on GPU
z2 = torch.mm(y.T, y)
## print the time cost of the multiplication
print("{} seconds passed ". format(time.time() - start2))
```

```
0.00026917457580566406 seconds passed
```

```
z2 = z2.cpu().numpy()
print(np.allclose(z1, z2))
```

```
True
```

Tensors Operations

- The following problems will be covered by several helpful tensor operations to be introduced in this tutorial:
  - ✓ How to manually create a tensor with scalars?
  - ✓ How to manually create a tensor with random numbers?
  - ✓ How to index and slice a tensor?
  - ✓ How to reshape a tensor?
  - ✓ How to execute mathematic operations over tensors?
  - ✓ How to enable CUDA support for a tensor?

Tensors Operations

- Creating tensors with all ones/zeros.

  torch.ones() and torch.zeros() accept the size of the tensor as input parameters.

```
## creating a tensor of 4 rows and 3 columns consisting of ones
x = torch.ones(4,3)
print(x)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

```
## creating a tensor of 4 rows and 3 columns consisting of ones
x = torch.zeros(4,3)
print(x)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

Tensors Operations

- Creating tensor with random values

  torch.manual_seed() can set the seed of generating random number, which is commonly used in reproducing results. torch.rand() accept the size of the tensor as input parameters.

```
## We often set a specific value as random seed to increase the reproducibility
torch.manual_seed(3)
## generating a random tensor
x = torch.rand(4, 3)
print(x)
```

```
tensor([[0.0043, 0.1056, 0.2858],
        [0.0270, 0.4716, 0.0601],
        [0.7719, 0.7437, 0.5944],
        [0.8879, 0.4510, 0.7995]])
```

Tensors Operations

- Creating tensor with random values

  torch.randn() generates tensor with random numbers sampled from a standard normal distribution.

```
## generating a random tensor from normal distribution
x = torch.randn(4,3)
print(x)
```

```
tensor([[ 0.3375,  1.0111, -1.4352],
        [ 0.9774,  0.5220,  1.2379],
        [-0.8646,  0.2990,  0.4192],
        [-0.0799,  0.9264,  0.8157]])
```

Tensors Operations

- Indexing of tensors

  Accessing the elements present in the tensor by specifying the index of the element. The indexing method of Pytorch tensor is similar to that of Python list.

```
## create a tensor with certain values
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
## get the third row (the index start with 0)
print(x[2])
## get the element in row 1, and col 2
print(x[0][1])
print(x[0, 1])
## get the value of one element tensor by using item()
print(x[0][1].item())
## access a range of elements
print(x[0:3, 0:3]) # start index is 0, end index is 3
                   # the range will include the start index
                   # and exclude the end index
```

```
tensor([7, 8, 9])
tensor(2)
tensor(2)
2
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

Tensors Operations

- Slicing of tensors

  Selecting the elements present in the tensor by using ":" operator.

```
## create a tensor with certain values
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
## try the slicing commands
print(x[:, 2]) # Every row, only the last column
print(x[0, :]) # Every column in first row
y = x[:, 1] # take the elements in the second columns and create a another tensor
print(y)
```

Tensors Operations

- Reshape a tensor

  Tensor.view() changes the shape of a tensor without changing the content.

```python
## creat a tensor with 3 rows and 2 columns
x = torch.tensor([[1, 2],
                  [3 ,4],
                  [5, 6]])
## reshaping to 2 rows and 3 columns
y = x.view(2, 3)
print (y)
```

```python
## creat a tensor with 3 rows and 2 columns
x = torch.tensor([[1, 2],
                  [3 ,4],
                  [5, 6]])
## use of -1 to reshaping a tensor
y = x.view(6, -1)
print (y)
```

Tensors Operations

- Mathematematic operations

Pytorch supports mathematic operations by using Pytorch arithmetic operation functions or python operands in an element-wise way.

```python
# create two tensors
x = torch.ones([3, 2])
y = torch.ones([3, 2])

# adding two tensors
z1 = x + y #method 1
z2 = torch.add(x, y) #method 2
print(torch.allclose(z1, z2))

# substracting two tensors
z1 = x - y #method 1
z2 = torch.sub(x,y) #method 2
print(torch.allclose(z1, z2))
```

```
True
True
```

```python
## create two tensors
x = torch.ones([3, 2])
y = torch.ones([3, 2])

## multiplying two tensors
z1 = x * y # method 1
z2 = torch.mul(x, y) # method 2

print(z1)
## check whether the two methods are equivalent
print(torch.allclose(z1, z2))
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
True
```

Tensors Operations

- In-place operations

The in-place operation changes the content directly without making a copy. Typically, a function postfixed with “_” denotes a in-place operation. “+=”and “*=” are also in-place operations.

```
#Create two tensors
x = torch.ones([3, 2])
y = torch.ones([3, 2])

#inplace operation
z = y.add_(x)
print(z)
```

CUDA Support

- Check CUDA support and transfer data to CPU

```
## check the number of CUDA supported GPU that are connected to the machine
print(torch.cuda.device_count())
## get the name of the GPU Card
print(torch.cuda.get_device_name(0))

## assign cuda GPU located at location '0' to a variable
cuda0 = torch.device('cuda:0')

## performing operations on GPU
a = torch.ones(3, 2, device=cuda0)
b = torch.ones(3, 2, device=cuda0)
c = a + b
print(c)

## move the result to CPU
c = c.cpu()
print(c)
```

```
1
Tesla T4
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]], device='cuda:0')
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
```

autograd in Pytorch

- Automatic differentiation package: *autograd*
- perform automatic gradient computation for all operations on tensors.

```
## create a tensor with requires_grad = True
## this will track all the operations performing on that tensor
x = torch.ones([3,3], requires_grad = True)
print(x)

## perform a tensor addition and check the result
y = x+1
print (y)

## perform a tensor multiplication and check the result
z = y*y+1
print(z)

## adding all the values in z and check the result
t = torch.sum(z)
print(t)

## peform backpropagation (partial derivate of t with respect to x) and check the result
t.backward()
print(x.grad)
```

Reference

- NumPy tutorial

http://cs231n.github.io/python-numpy-tutorial/

- PyTorch master documentation

https://pytorch.org/docs/stable/index.html

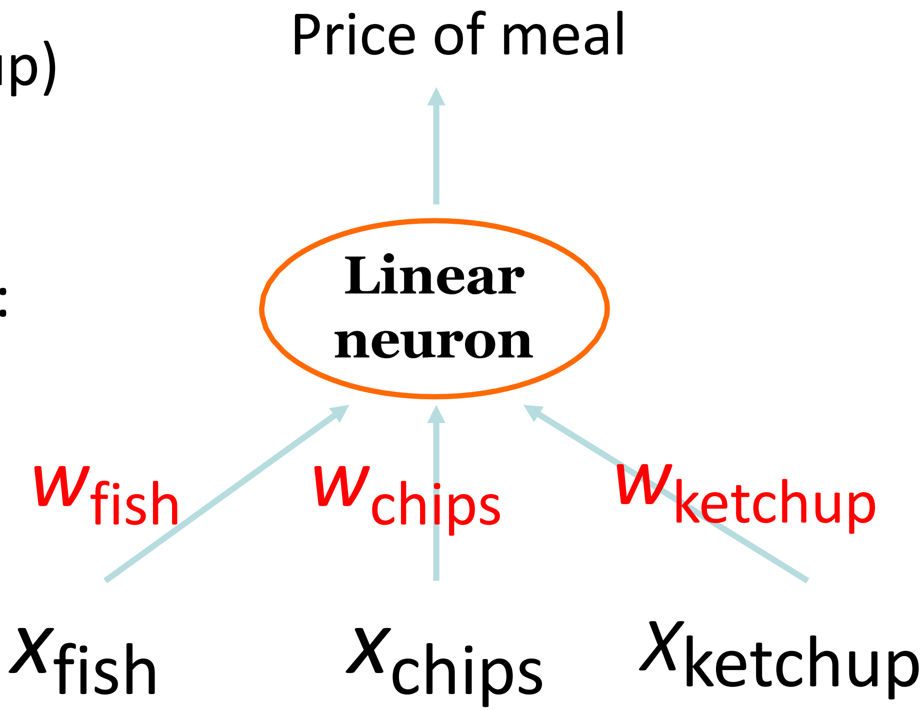
Lab 1

- Go through the tutorial in this slides
- Create a Colab notebook and show how you train a linear model in four iterations
  - ✓ Dataset and learning goal are illustrated in next pages of this slides. You can review the toy example in *Lecture 5 - Backpropagation* for more details

- Submit to canvas by **11:59 pm, 2/21/2023,** including:
  - ✓ Your code
  - ✓ A report with code output screenshot and the loss figures that shows how you update your model in the iterative way

Lab 1

- We have a dataset:
  - ✓ Input: (# fish, # chips, # ketchup)
  - ✓ Target: Total money
- Forward pass of the linear model:
  - ✓ Price = $x_{\text{fish}}\, w_{\text{fish}} + x_{\text{chips}}\, w_{\text{chips}} + x_{\text{ketchup}}\, w_{\text{ketchup}}$
- Learning goal:
  - ✓ $w = (w_{\text{fish}}, w_{\text{chips}}, w_{\text{ketchup}})$
- Initial value: $w_0 = (50, 50, 50)$



Lab 1

- Task 1: Implementing two weights updating methods:
  - ✓ "delta rule". This algorithm is elaborated in Lecture 5. Only one sample is used in each iteration. 4 iterations are required. Four training samples for "delta rule" are as follow:

    I: (5, 2, 4), T: 1250, $\varepsilon = 1/70$; I: (3, 3, 3), T: 900, $\varepsilon = 1/12$;
    I: (0, 5, 1), T: 350, $\varepsilon = 1/27$; I: (2, 1, 2), T: 550, $\varepsilon = 2/20$.

  - ✓ "batch delta rule". Batching samples and iterating all batches in each epochs. The weight changes are summed over training cases. 10 epochs and a batch size of 3 are required. Using the following data samples instead and the learning rate ε is 1/100:

    I: (5, 2, 4), T: 1250; I: (3, 3, 3), T: 900; I: (0, 5, 1), T: 350;
    I: (2, 1, 2), T: 550; I: (1, 0, 5), T: 650; I: (4, 2, 1), T: 800;
    I: (6, 1, 1), T: 1050; I: (2, 3, 4), T: 850; I: (7, 3, 0), T: 1200;
    I: (4, 4, 2), T: 1000; I: (1, 5, 7), T: 1100; I: (5, 1, 3), T: 1100.

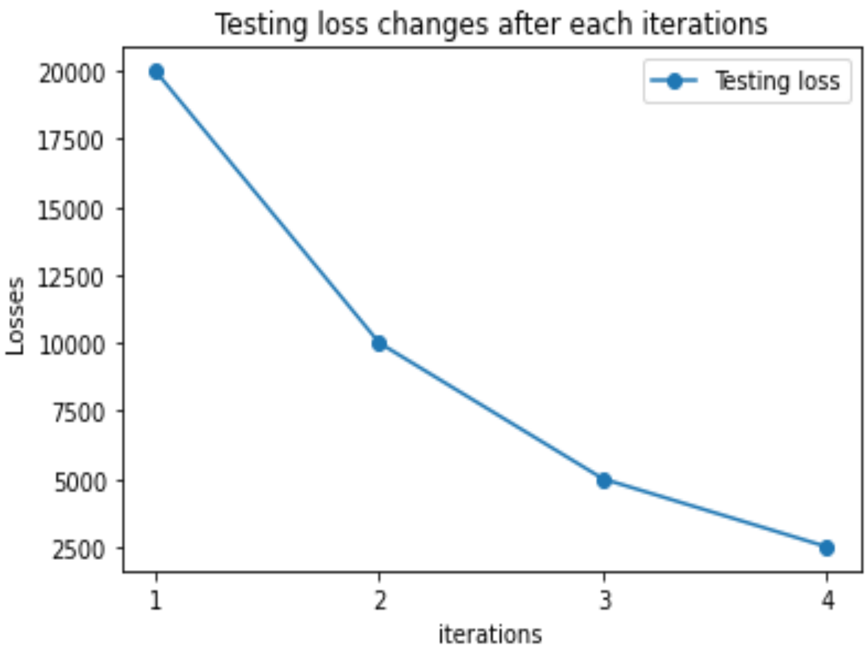
Lab 1

- Task 2: Validation of the linear model:
  - ✓ Collecting the following samples as testing dataset. After each iteration or epoch of training, the testing loss is collected, and testing loss changes should be plotted after training. The Mean Squared Error is used for calculate the loss.
  - ✓ Testing samples:

    I: (6, 1, 3), T: 1250

    I: (2, 2, 1), T: 500

    I: (4, 5, 0), T: 850



- Note:

  The loss figure should be similar to the one on the right side. You should obtain two figures for two weights updating methods.

Lab 1

```python
import torch
## It is recommended that using a class to customize the Dataset
## so that you can create an iterable loader by DataLoader.
from torch.utils.data import Dataset, DataLoader
## using matplotlib to plot your losses
import matplotlib.pyplot as plt

class train_data(Dataset):
   ## your code is here
   ## you need to define __init__, __len__, __getitem__ at least

class test_data(Dataset):
   ## your code is here

## You can implement the delta_rule and batch_delta_rule with
## functions or classes. The following code is just for reference.
def delta_rule(your_model, your_dataloader):
   ## your code is here
   ## plot your figure of loss changes after training
   ## or return the loss array of each iterations/epochs

def batch_delta_rule(your_model, your_dataloader):
   ## your code is here

## In this case you should run delta_rule(model, train_loader) and
## batch_delta_rule(model, train_loader) respectively.
## Applying the above weights updating methods to train
## your model instead of using the training API of pytorch.
## Again, you can complete it in your own way
```