Tutorials for Colab and PyTorch

Jinwoo Kim

jinwoo-kim@kaist.ac.kr

Table of Contents [link to the materials]

PyTorch Tutorial

Basics components and operations

Importing PyTorch

- Colab supports PyTorch by default

```
[1]
import torch

torch.__version__
```

```
'1.4.0'
```

Tensors: basic computing unit of Pytorch

- Basically, tensors are for representing scalars, vectors, and matrices
- Similar to NumPy's ndarrays, but <u>supports **GPU** acceleration</u>

Matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Defining a tensor

```
[1]  import torch
     import numpy as np

[2]  # numpy ndarray
     np.array([[1, 2], [3, 4]])

     array([[1, 2],
            [3, 4]])

[3]  # pytorch tensor on cpu
     torch.tensor([[1, 2], [3, 4]])

     tensor([[1, 2],
             [3, 4]])
```

Defining a tensor on GPU

```
[4]  # pytorch tensor on gpu
     torch.tensor([[1, 2], [3, 4]], device='cuda')

     tensor([[1, 2],
             [3, 4]], device='cuda:0')
```

Tensor <-> ndarray

```
[5]  # numpy ndarray to pytorch tensor
     np_array = np.array([[1, 2], [3, 4]])
     torch.from_numpy(np_array)

     tensor([[1, 2],
             [3, 4]])

[6]  # pytorch tensor to numpy ndarray
     torch_tensor = torch.tensor([[1, 2], [3, 4]])
     torch_tensor.numpy()

     array([[1, 2],
            [3, 4]])
```

Basic arithmetic operations with tensors

Addition

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 6 \\ 6 & 6 \end{bmatrix}$$

Subtraction

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} - \begin{bmatrix} 5 & 4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} -4 & -2 \\ 0 & 2 \end{bmatrix}$$

element-wise multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 5 & 8 \\ 9 & 8 \end{bmatrix}$$

element-wise division

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \div \begin{bmatrix} 5 & 4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.5 \\ 1.0 & 2.0 \end{bmatrix}$$

```
[7]  a = torch.tensor([[1., 2.], [3., 4.]])
     b = torch.tensor([[5., 4.], [3., 2.]])
```

```
[8]  a+b
```

```
tensor([[6., 6.],
        [6., 6.]])
```

```
[9]  a-b
```

```
tensor([[-4., -2.],
        [ 0.,  2.]])
```

```
[10]  a*b
```

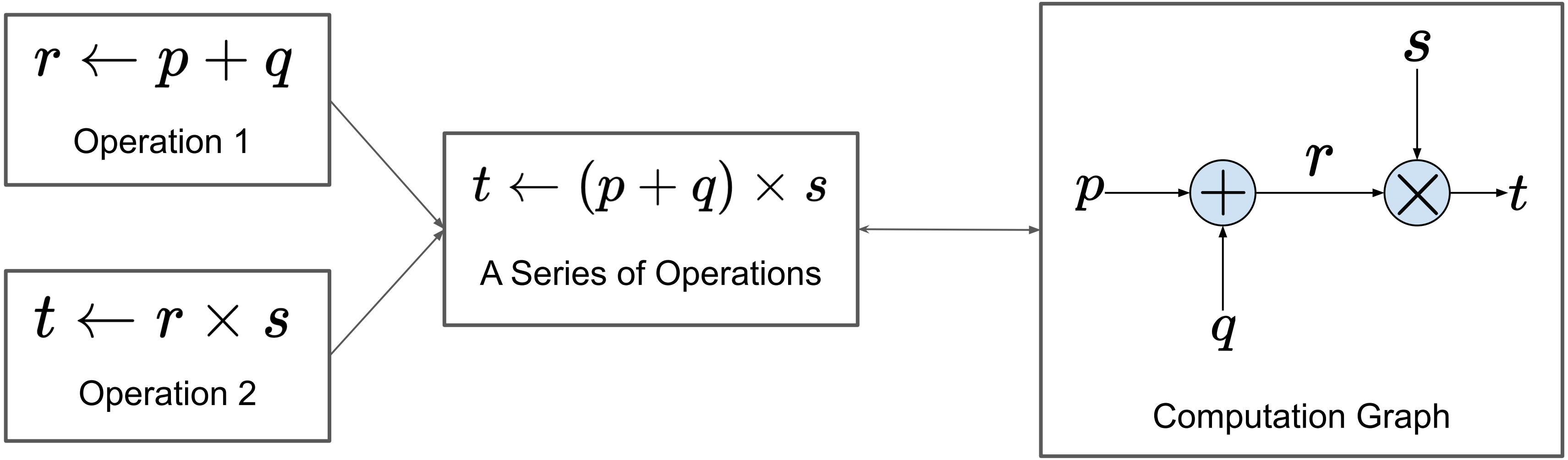
```
tensor([[5., 8.],
        [9., 8.]])
```

```
[11]  a/b
```

```
tensor([[0.2000, 0.5000],
        [1.0000, 2.0000]])
```

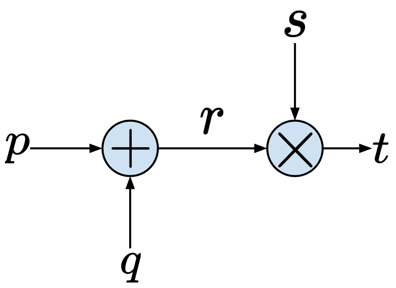
Computation graph

- A series of operations constructs a computation graph
- Any operation between tensors defines a node in the computation graph

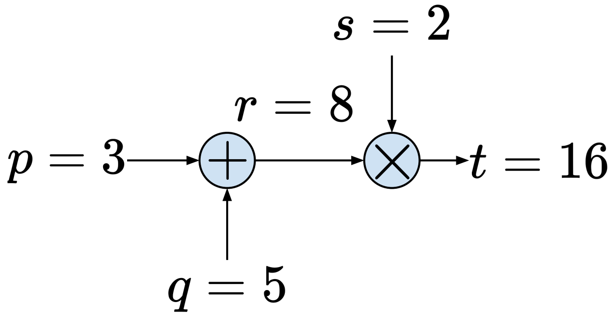


Computation graph and Forward function

- Consider below computation graph as our forward function



- If we assume *p=3*, *q=5*, and *s=2*, then we get *r=8* and *t=16*.

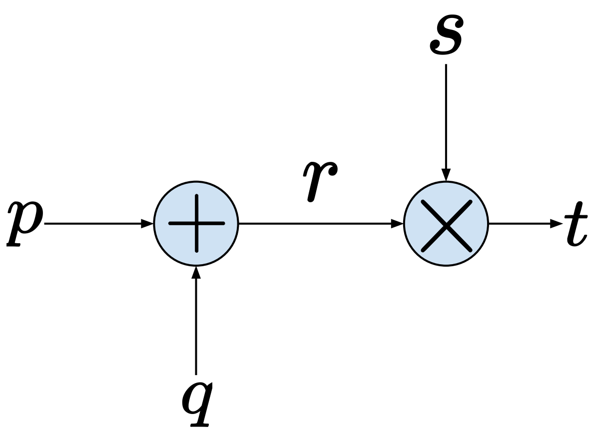



```
[14] p = torch.tensor(3.)
     q = torch.tensor(5.)
     s = torch.tensor(2.)
     r = p + q
     t = r * s
```

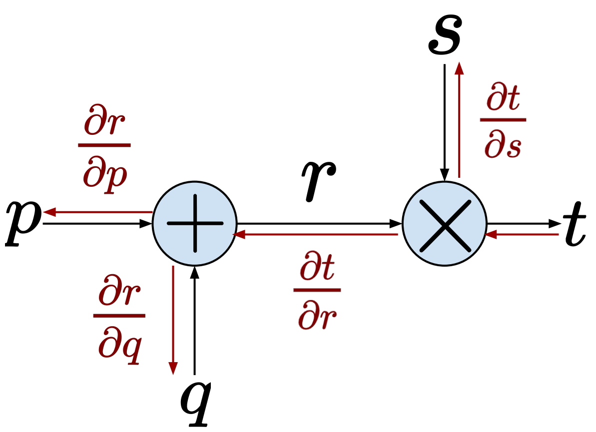
```
print('r :', r)
print('t :', t)
```

```
r : tensor(8.)
t : tensor(16.)
```

Forward & Backward functions

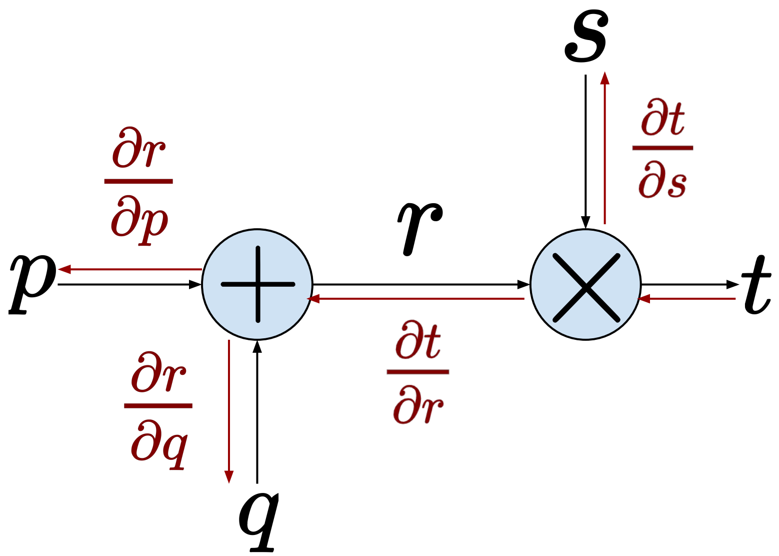


Forward Function



Backward Function

Backward function and Chain rule



$$\frac{\partial t}{\partial s} = \frac{\partial (r \times s)}{\partial s} = r \qquad \frac{\partial r}{\partial p} = \frac{\partial (p+q)}{\partial p} = 1$$

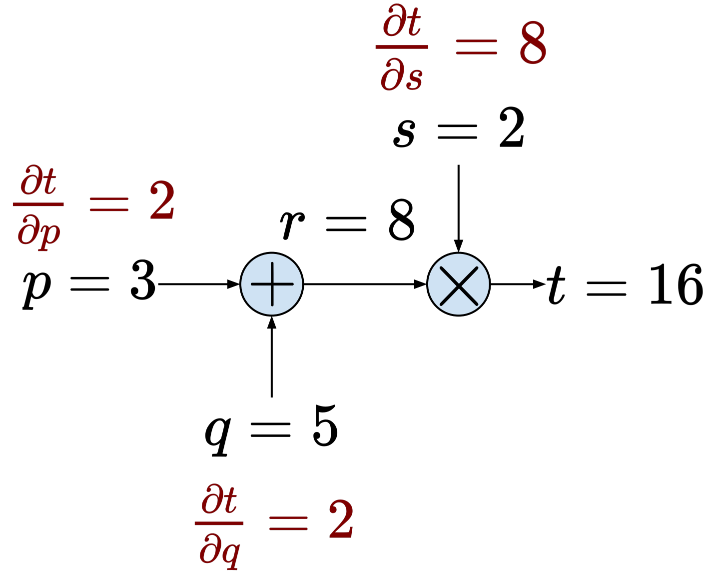
$$\frac{\partial t}{\partial r} = \frac{\partial (r \times s)}{\partial r} = s \qquad \frac{\partial r}{\partial q} = \frac{\partial (p+q)}{\partial q} = 1$$

$$\frac{\partial t}{\partial q} = \frac{\partial t}{\partial r} \times \frac{\partial r}{\partial q} = s \times 1 = s$$

$$\frac{\partial t}{\partial p} = \frac{\partial t}{\partial r} \times \frac{\partial r}{\partial p} = s \times 1 = s$$

Backward function: an example

- If we assume *p=3*, *q=5*, *s=2*, *r=8*, and *t=16*, then *dt/ds=8*, *dt/dp=2* and *dt/dq=2*



Automatic differentiation (AutoGrad)

- Wait, then do we have to calculate all the derivatives on our own ?
- What if the variables are vectors and matrices, but not scalars ?
- No worries ! AutoGrad Package in PyTorch will do that for us.

```python
import torch

# Forward Propagation
p = torch.tensor([3.], requires_grad=True)
q = torch.tensor([5.], requires_grad=True)
s = torch.tensor([2.], requires_grad=True)
r = p + q
t = r * s

print('p :', p.item())
print('q :', q.item())
print('s :', s.item())
print('r :', r.item())
print('t :', t.item())
```

```
p : 3.0
q : 5.0
s : 2.0
r : 8.0
t : 16.0
```

```python
# Backward Propagation
t.backward()

print('dt/dp :', p.grad.item())
print('dt/dq :', q.grad.item())
print('dt/ds :', s.grad.item())
```

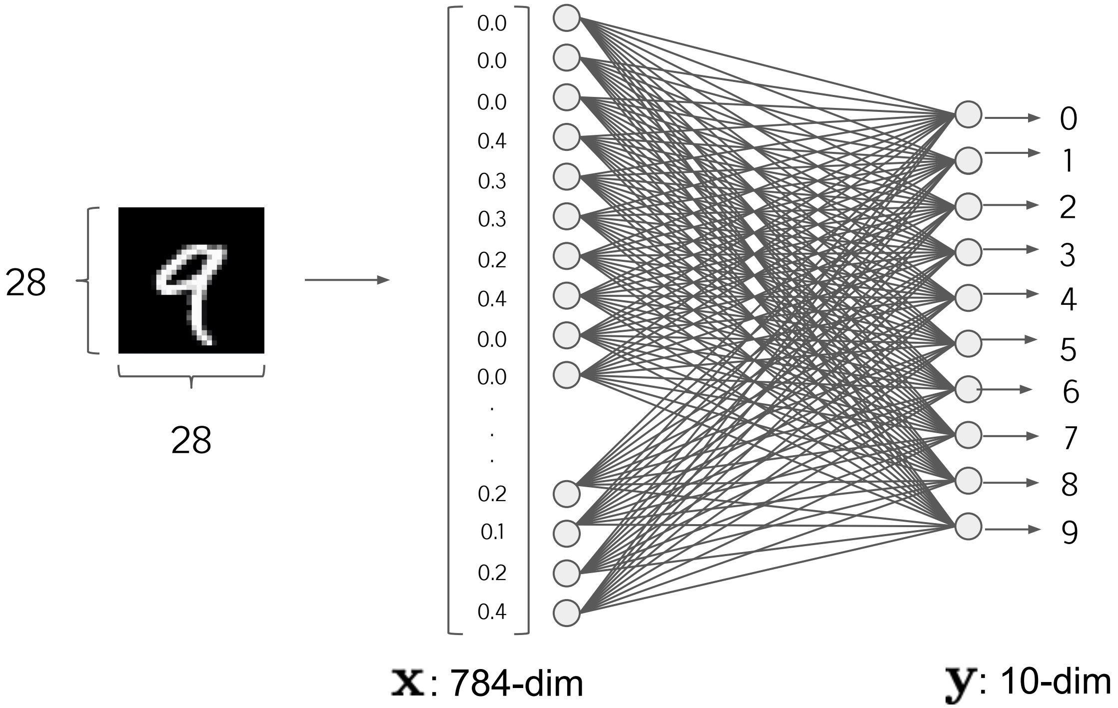
```
dt/dp : 2.0
dt/dq : 2.0
dt/ds : 8.0
```

PyTorch Tutorial

Building a neural network

Let's build a one-layer baby network

- One-layer classifier for MNIST digit classification



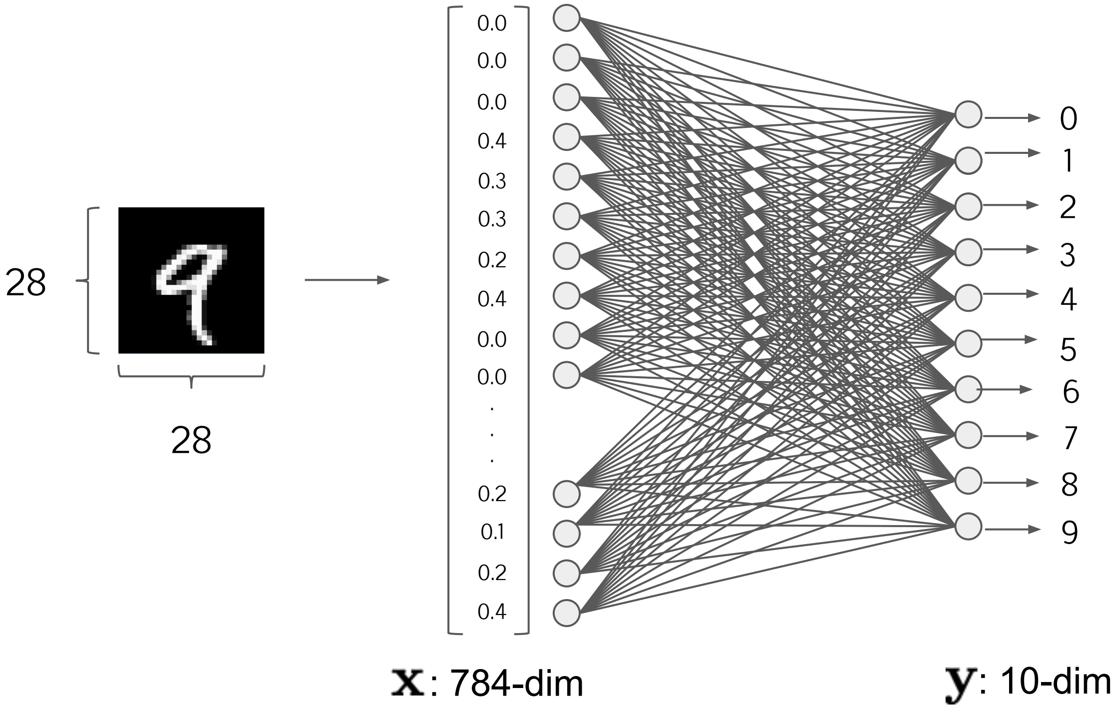
$$\mathbf{y} = \mathbf{x}\boxed{\mathbf{W}} + \boxed{\mathbf{b}}$$

Size of W? [784, 10]

Size of b? [1, 10]

Let's build a one-layer baby network

- One-layer classifier for MNIST digit classification



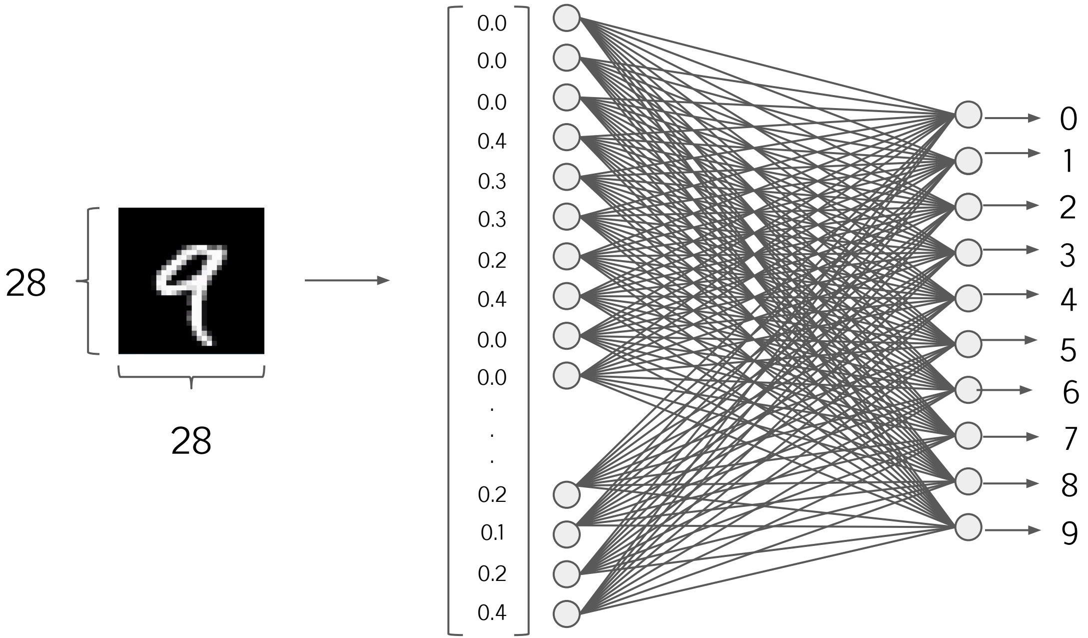
$$\mathbf{y} = \mathbf{x}\boxed{\mathbf{W}} + \boxed{\mathbf{b}}$$

```
import math

weights = torch.randn(784, 10) / math.sqrt(784)
weights.requires_grad_()
bias = torch.zeros(10, requires_grad=True)
```

Code credit: Jeremy Howard

Let's build a one-layer baby network

- One-layer classifier for MNIST digit classification



$\mathbf{x}$: 784-dim

$\mathbf{y}$: 10-dim

$$\boxed{\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{b}}$$

```
import math

weights = torch.randn(784, 10) / math.sqrt(784)
weights.requires_grad_()
bias = torch.zeros(10, requires_grad=True)
```

```
def log_softmax(x):
    return x - x.exp().sum(-1).log().unsqueeze(-1)

def model(xb):
    return log_softmax(xb @ weights + bias)
```

Code credit: Jeremy Howard

Baby network: forward propagation

```
bs = 64  # batch size

xb = x_train[0:bs]  # a mini-batch from x
preds = model(xb)  # predictions
preds[0], preds.shape
print(preds[0], preds.shape)
```

output:

```
tensor([-1.9759, -2.1991, -1.9989, -2.4762, -2.6573, -2.2036, -2.7582, -2.5692,
        -2.2971, -2.2089], grad_fn=<SelectBackward>) torch.Size([64, 10])
```

Code credit: Jeremy Howard

Baby network: loss function

- Binary cross-entropy loss

```
def nll(input, target):
    return -input[range(target.shape[0]), target].mean()

loss_func = nll
```

Code credit: Jeremy Howard

Baby network: a training loop

Sampling the minibatch (of the size bs)

Forward & loss computation

Gradient update step

```
from IPython.core.debugger import set_trace

lr = 0.5  # learning rate
epochs = 2  # how many epochs to train for

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        xb = x_train[start_i:end_i]
        yb = y_train[start_i:end_i]
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        with torch.no_grad():
            weights -= weights.grad * lr
            bias -= bias.grad * lr
            weights.grad.zero_()
            bias.grad.zero_()
```

Code credit: Jeremy Howard

What should we have for this single-layer network?

- Network parameters
  - Tensors for weight and bias
- Forward and backward mechanisms
  - $y=xW + b$ in this case
- Gradient of parameters
  - All parameters in the network should hold the gradient of the loss w.r.t itself
- Loss function
  - A binary cross entropy loss in this case
- Optimizations
  - Weight initialization, a naive gradient update mechanism (SGD)

How about more complicated networks?

A single linear network

# of layers: 1

A matrix multiplication and addition

Inception (GoogleNet)

# of layers: a lot

Parallel convolution with different filter sizes, nonlinear functions, batch norm, average and max poolings, multi-head loss, ...

Solution: modularize the computations

- Modularize a layer using a **class**, which comes with handy utility functions
  (e.g. managing parameters/gradients, forward/backward, switching b/w training/evaluation modes, etc)

```
from torch import nn

class Mnist_Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(784, 10) / math.sqrt(784))
        self.bias = nn.Parameter(torch.zeros(10))

    def forward(self, xb):
        return xb @ self.weights + self.bias
```

It inherits other utility functions defined in torch.nn.Module

Solution: modularize the computations

- Modularize a layer using a **class**, which comes with handy utility functions (e.g. managing parameters/gradients, forward/backward, switching b/w training/evaluation modes, etc)

```python
class Linear(Module):
    __constants__ = ['in_features', 'out_features']

    def __init__(self, in_features, out_features, bias=True):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)

    def forward(self, input):
        return F.linear(input, self.weight, self.bias)
```

Example:
Actual definition of
fully-connected layer in PyTorch

Solution: modularize the computations

- Modularize a layer using a **class**, which comes with handy utility functions
  (e.g. managing parameters/gradients, forward/backward, switching b/w training/evaluation modes, etc)

- We can build a complicated neural network by simply composing these layers

Google Colaboratory

Before we start ...

1. Create your google account if you don’t have one

2. Go to this Link and open Introduction to Colab.ipynb

3. Click `File` tab -> Click `Save a copy in drive` button
   - This will save the notebook file in your google drive, which is necessary to follow the tutorial
   - Check `Colab Notebooks` directory in your google drive if the notebook is saved successfully

What is Colaboratory?

- Colaboratory is a **free Jupyter notebook** environment that requires no setup and runs entirely in the cloud.

- With Colaboratory you can write and execute code, save and share your analyses, and access powerful computing resources, all for free from your browser.

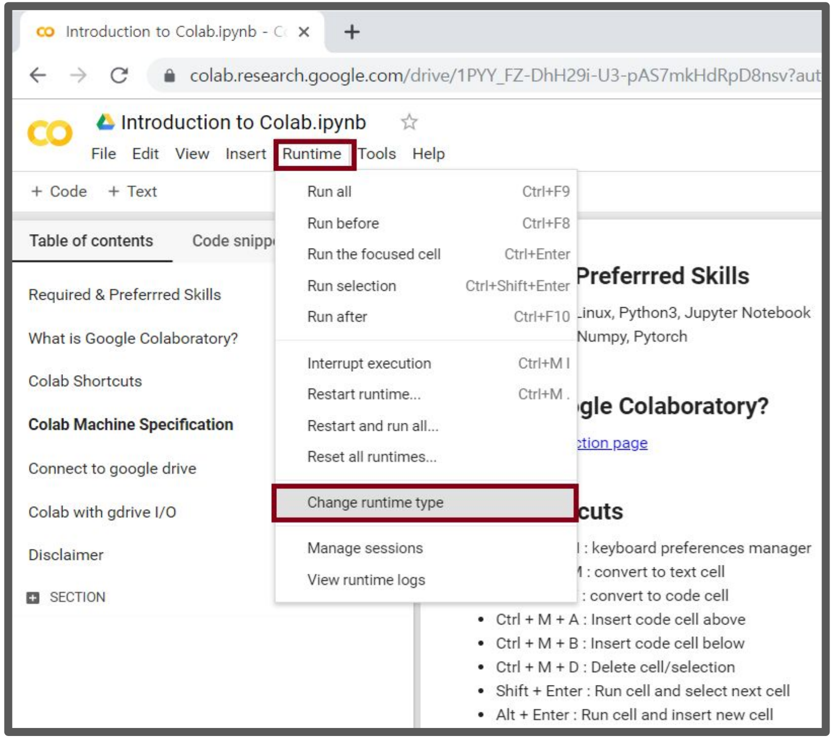
- Colaboratory is run on a Ubuntu 18.04 virtual machine equipped with 13GB RAM, ~310GB Storage limits, and GPUs (K80, TPU).

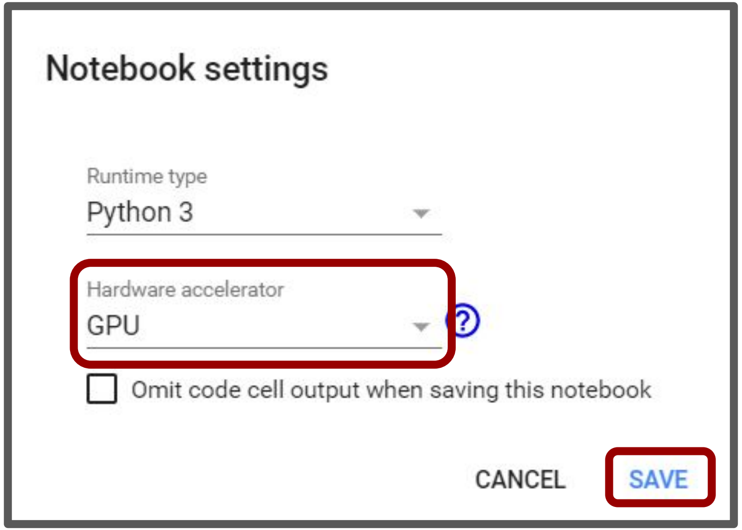
Handy shortcuts

- Ctrl + M + H : keyboard preferences manager
- Ctrl + M + M : convert to text cell
- Ctrl + M + Y : convert to code cell
- Ctrl + M + A : Insert code cell above
- Ctrl + M + B : Insert code cell below
- Ctrl + M + D : Delete cell/selection
- Shift + Enter : Run cell and select next cell
- Alt + Enter : Run cell and insert new cell
- Ctrl + M + I : Interrupt execution
- Ctrl + M + . : Restart Runtime
- Ctrl + / : comment/uncomment

How to setup GPU

- Runtime -> Change runtime type -> Set Hardware accelerator to GPU -> Save





Mount your google drive - 1

Run the script below, and follow the instruction.
Namely,

1. Go to the given URL in a browser
2. Select your cs470 account and log-in
3. Allow access to the google account
4. Copy the given authorization code, and paste it into the blank below

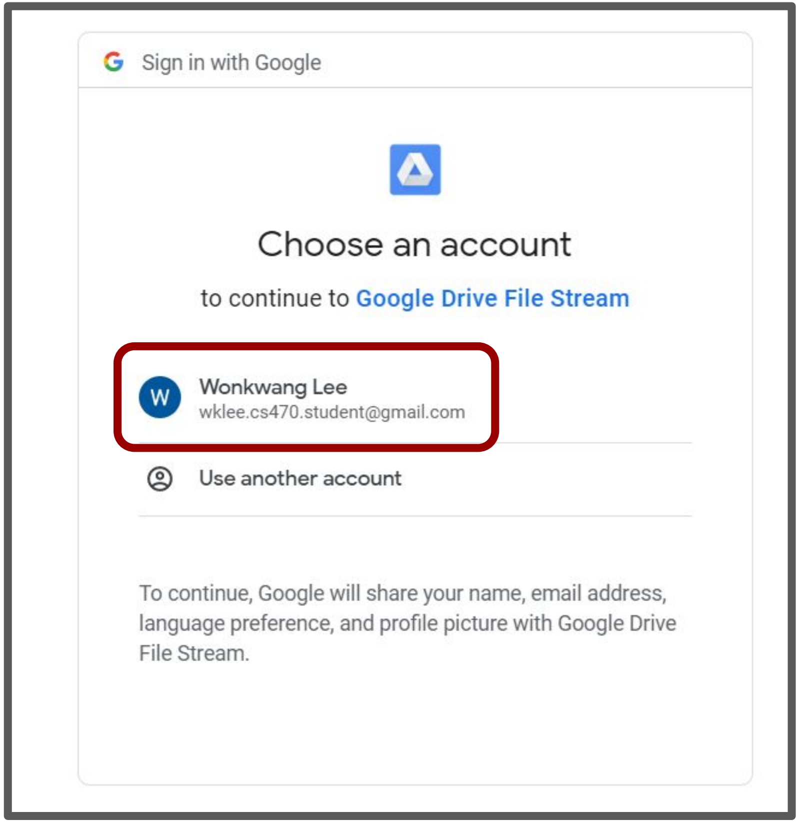
If you succeed, then you'll see "Mounted at /gdrive"

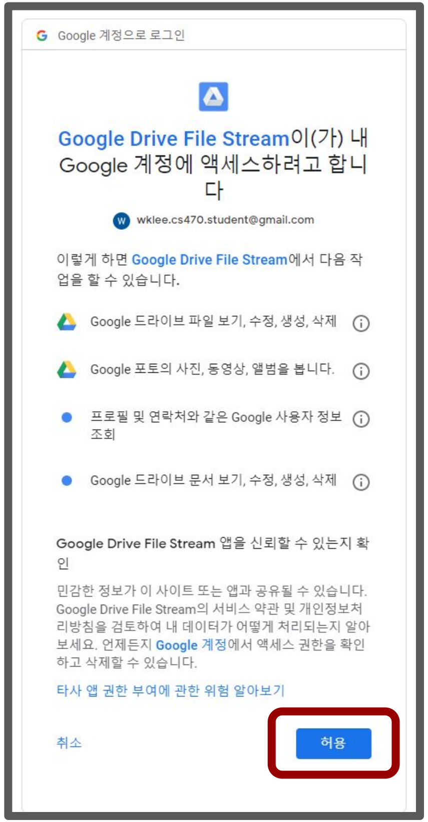
Note : This step should be repeated everytime you initialize the runtime sesison

```
from google.colab import drive

drive.mount('/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n

Enter your authorization code:
```

Mount your google drive - 2





Mount your google drive - 3

Run the script below, and follow the instruction.

Namely,

1. Go to the given URL in a browser
2. Select your cs470 account and log-in
3. Allow access to the google account
4. Copy the given authorization code, and paste it into the blank below

If you succeed, then you'll see "Mounted at /gdrive"

Note : This step should be repeated everytime you initialize the runtime sesison

```
from google.colab import drive

drive.mount('/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n

Enter your authorization code:
```

Mount your google drive - 4

```
from google.colab import drive
drive.mount('/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6

Enter your authorization code:
..................................................................
```

```
from google.colab import drive
drive.mount('/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.g

Enter your authorization code:
··········
Mounted at /gdrive
```

Access to your google drive in Colab

```python
# check what's in the mounted gdrive using Colab
import os


gdrive_root = '/gdrive/My Drive'
print('In gdrive :', os.listdir(gdrive_root))

notebook_dir = os.path.join(gdrive_root, 'Colab Notebooks')
print('In Colab Notebooks :', os.listdir(notebook_dir))
```

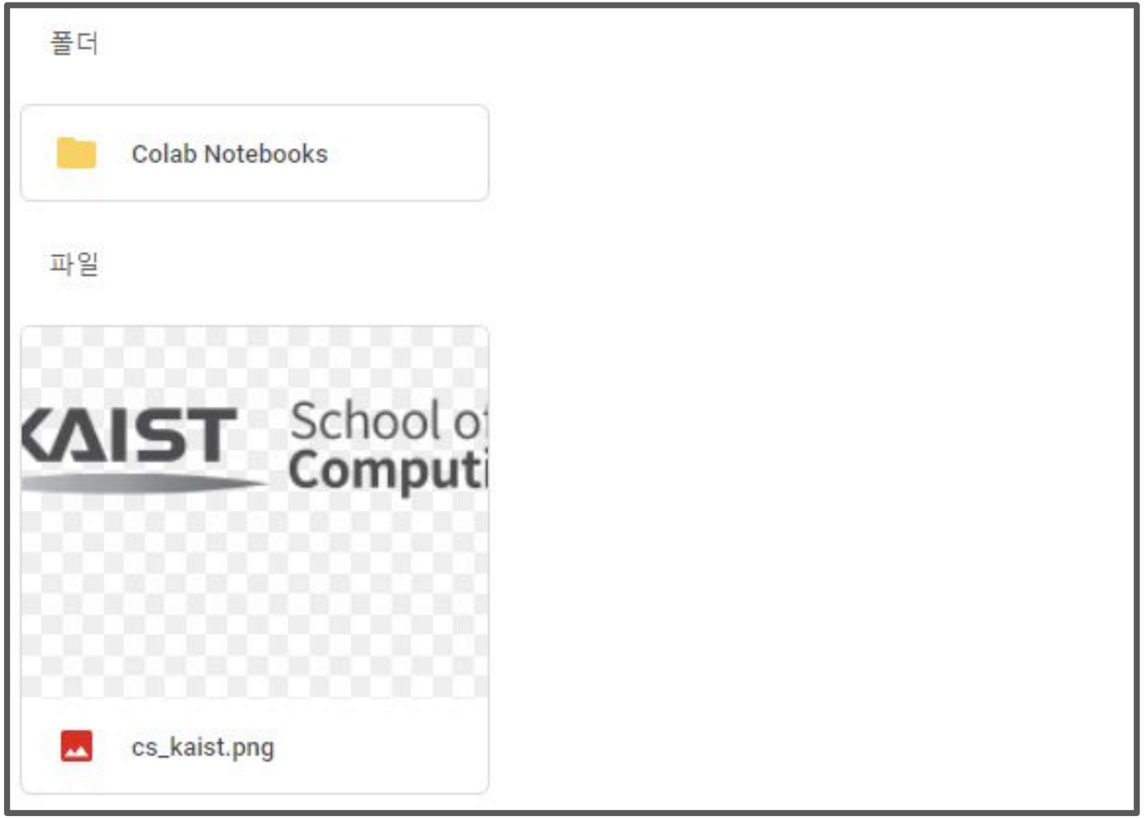
```
In gdrive : ['Colab Notebooks']
In Colab Notebooks : ['Copy of Introduction to Colab.ipynb']
```

Download an image into your google drive

```
# download and save an image
!wget https://cs.kaist.ac.kr/common/images/header/logo_top.png -O '/gdrive/My Drive/cs_kaist.png'
print('In gdrive :', os.listdir(gdrive_root))

# Go to the google drive hompage(https://drive.google.com/drive/my-drive),
# log-in using your CS470 account,
# and browse your gdrive directory to check if the image is downloaded successfully
```

```
--2019-09-08 12:12:39--  https://cs.kaist.ac.kr/common/images/header/logo_top.png
Resolving cs.kaist.ac.kr (cs.kaist.ac.kr)... 192.249.19.36
Connecting to cs.kaist.ac.kr (cs.kaist.ac.kr)|192.249.19.36|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6313 (6.2K) [image/png]
Saving to: ‘/gdrive/My Drive/cs_kaist.png’

/gdrive/My Drive/cs 100%[===================>]   6.17K  --.-KB/s    in 0s

2019-09-08 12:12:41 (13.3 MB/s) - ‘/gdrive/My Drive/cs_kaist.png’ saved [6313/6313]

In gdrive : ['Colab Notebooks', 'cs_kaist.png']
```

Download an image into your google drive



Load the image from your google drive

```python
# load the saved image
from PIL import Image

image_path = os.path.join(gdrive_root, 'cs_kaist.png')
img = Image.open(image_path)
img
```

KAIST School of Computing

Disclaimer

- Runtime session will last **at most** 12 hours, regardless of devices (e.g. CPU, GPU, and TPU)
  - if you left the browser opened, probably the session will last at most 12 hours
  - if you closed the browser, probably the session will last at most 90 minutes

- Therefore, it is **highly recommended** that you periodically **back-up your data/outputs to your gdrive** and resume your training by re-loading your saved data. Otherwise, you'll lose everything you've trained as soon as the session is recycled.

PyTorch + Colab

Train and test a simple MLP-based MNIST classifier

Again,

1. Create your google account if you don’t have one

2. Go to this Link and open
   5. Train and Test a simple MLP-based MNIST classifier.ipynb

3. Click `File` tab -> Click `Save a copy in drive` button
   - This will save the notebook file in your google drive, which is necessary to follow the tutorial
   - Check `Colab Notebooks` directory in your google drive if the notebook is saved successfully

Common steps for training a neural network in Colab

1. Connect to your google drive
2. Import modules
3. Configure the experiments (e.g. hyper-parameters)
4. Construct data pipeline
5. Construct a neural network builder
6. Initialize the network and optimizer
7. Load pre-trained weight if exists
8. Train the network
9. Visualize and analyze the results

1. Connect to your google drive

- This step is **required** if you want to save checkpoints into your drive and load them later on

```
from google.colab import drive

drive.mount('/gdrive')
gdrive_root = '/gdrive/My Drive'
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?

Enter your authorization code:
··········
Mounted at /gdrive
```

2. Import modules

```
import os

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST
```

3. Configure the experiments

```
# training & optimization hyper-parameters
max_epoch = 10
learning_rate = 0.0001
batch_size = 200
device = 'cuda'

# model hyper-parameters
input_dim = 784 # 28x28=784
hidden_dim = 512
output_dim = 10
```

4. Construct data pipeline

- **torchvision.datasets.MNIST** will automatically construct **MNIST** dataset.
- **torch.utils.data.DataLoader** receives MNIST dataset and does followings
  - parse data using multi-processing
  - make mini-batches of data
  - shuffle data when make a mini-batch

```
data_dir = os.path.join(gdrive_root, 'my_data')

transform = transforms.ToTensor()

train_dataset = MNIST(data_dir, train=True, download=True, transform=transform)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=True)

test_dataset = MNIST(data_dir, train=False, download=True, transform=transform)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, drop_last=False)
```

5. Construct a neural network builder

- Here we're going to train a simple MLP-based neural network with ReLU non-linear activation functions.

```
class MyClassifier(nn.Module):
  def __init__(self, input_dim=784, hidden_dim=512, output_dim=10):
    super(MyClassifier, self).__init__()
    self.layers = nn.Sequential(
      nn.Linear(input_dim, hidden_dim),
      nn.ReLU(),
      nn.Linear(hidden_dim, hidden_dim),
      nn.ReLU(),
      nn.Linear(hidden_dim, hidden_dim),
      nn.ReLU(),
      nn.Linear(hidden_dim, output_dim),
    )

  def forward(self, x):
    batch_size = x.size(0)
    x = x.view(batch_size, -1)
    outputs = self.layers(x)
    return outputs
```

6. Initialize the network and optimizer

- Initialize the network and pass its parameters to the optimizer

```
my_classifier = MyClassifier(input_dim, hidden_dim, output_dim)
my_classifier = my_classifier.to(device)

optimizer = optim.Adam(my_classifier.parameters(), lr=learning_rate)
```

7. Load pre-trained weight if exists

- Later when we train a neural network, we're going to save checkpoints periodically into the location 'gdrive/My Drive/checkpoints'.
- And if you have a saved checkpoint there, this block will load it and resume the training.

```
ckpt_dir = os.path.join(gdrive_root, 'checkpoints')
if not os.path.exists(ckpt_dir):
  os.makedirs(ckpt_dir)

ckpt_path = os.path.join(ckpt_dir, 'lastest.pt')
if os.path.exists(ckpt_path):
  ckpt = torch.load(ckpt_path)
  best_acc = ckpt['best_acc']
  my_classifier.load_state_dict(ckpt['my_classifier'])
  optimizer.load_state_dict(ckpt['optimizer'])
  print('checkpoint is loaded !')
  print('current best accuracy : %.2f' % best_acc)
else:
  best_acc = 0
```

8. Now train the network

- Training session consists of mainly three parts:
  - train phase
  - test phase
  - backup phase

```
it = 0
train_losses = []
test_losses = []
for epoch in range(max_epoch):
  # train phase
  # ...

  # test phase
  # ...

  # save checkpoint whenever there is improvement in performance
  # ...
```

8. Now train the network - train phase

```
it = 0
train_losses = []
test_losses = []
for epoch in range(max_epoch):
  # train phase
  for inputs, labels in train_dataloader:
    it += 1

    # load data to the GPU.
    inputs = inputs.to(device)
    labels = labels.to(device)

    # feed data into the network and get outputs.
    logits = my_classifier(inputs)

    # calculate loss
    # Note: `F.cross_entropy` function receives logits, or pre-softmax outputs, rather than final probability scores.
    loss = F.cross_entropy(logits, labels)

    # Note: You should flush out gradients computed at the previous step before computing gradients at the current step.
    #       Otherwise, gradients will accumulate.
    optimizer.zero_grad()

    # backprogate loss.
    loss.backward()

    # update the weights in the network.
    optimizer.step()

    # calculate accuracy.
    acc = (logits.argmax(dim=1) == labels).float().mean()

    if it % 200 == 0:
      print('[epoch:{}, iteration:{}] train loss : {:.4f} train accuracy : {:.4f}'.format(epoch, it, loss.item(), acc.item()))

  # save losses in a list so that we can visualize them later.
  train_losses.append(loss.item())
```

8. Now train the network - test phase

- test phase follows the same steps as the train phase, except that:
  - use test data instead of train data
  - do not back-propagate loss and update weights

```
# test phase
n = 0.
test_loss = 0.
test_acc = 0.
for test_inputs, test_labels in test_dataloader:
  test_inputs = test_inputs.to(device)
  test_labels = test_labels.to(device)

  logits = my_classifier(test_inputs)
  test_loss += F.cross_entropy(logits, test_labels, reduction='sum')
  test_acc += (logits.argmax(dim=1) == test_labels).float().sum()
  n += inputs.size(0)

test_loss /= n
test_acc /= n
test_losses.append(test_loss.item())
print('[epoch:{}, iteration:{}] test_loss : {:.4f} test accuracy : {:.4f}'.format(epoch, it, test_loss.item(), test_acc.item()))
```

8. Now train the network - checkpointing

- It is always a good idea to save the checkpoints periodically, otherwise you'll lose everything you've trained if the session is expired.

```
# save checkpoint whenever there is improvement in performance
if test_acc > best_acc:
  best_acc = test_acc
  # Note: optimizer also has states ! don't forget to save them as well.
  ckpt = {'my_classifier':my_classifier.state_dict(),
          'optimizer':optimizer.state_dict(),
          'best_acc':best_acc}
  torch.save(ckpt, ckpt_path)
  print('checkpoint is saved !')
```

8. Now train the network - results

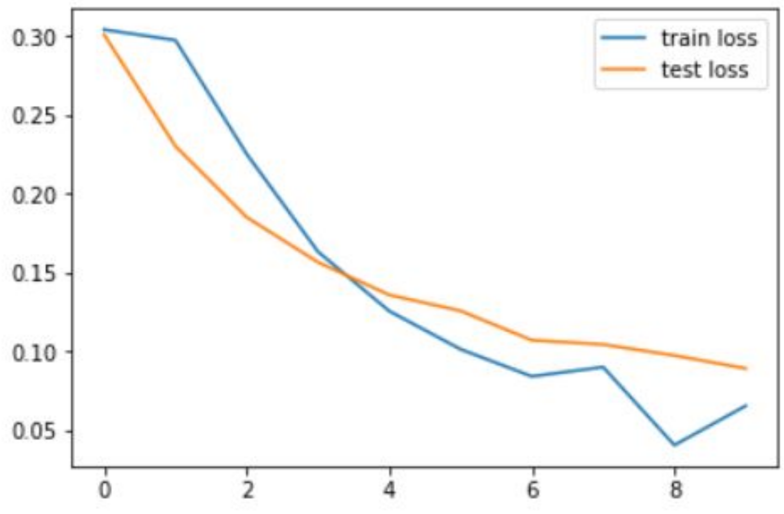
- After 10 epochs,
  you'll get near 98% accuracy on MNIST dataset

```
[epoch:5, iteration:1600] train loss : 0.1125 train accuracy : 0.9700
[epoch:5, iteration:1800] train loss : 0.1012 train accuracy : 0.9700
[epoch:5, iteration:1800] test_loss : 0.1256 test accuracy : 0.9606
checkpoint is saved !
[epoch:6, iteration:2000] train loss : 0.0906 train accuracy : 0.9750
[epoch:6, iteration:2100] test_loss : 0.1069 test accuracy : 0.9663
checkpoint is saved !
[epoch:7, iteration:2200] train loss : 0.0742 train accuracy : 0.9750
[epoch:7, iteration:2400] train loss : 0.0898 train accuracy : 0.9750
[epoch:7, iteration:2400] test_loss : 0.1042 test accuracy : 0.9671
checkpoint is saved !
[epoch:8, iteration:2600] train loss : 0.0897 train accuracy : 0.9800
[epoch:8, iteration:2700] test_loss : 0.0972 test accuracy : 0.9698
checkpoint is saved !
[epoch:9, iteration:2800] train loss : 0.0885 train accuracy : 0.9700
[epoch:9, iteration:3000] train loss : 0.0652 train accuracy : 0.9750
[epoch:9, iteration:3000] test_loss : 0.0889 test accuracy : 0.9720
checkpoint is saved !
```

9. Visualize and analyze the results - 1

```
import matplotlib.pyplot as plt

plt.plot(train_losses, label='train loss')
plt.plot(test_losses, label='test loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f0183010be0>
```



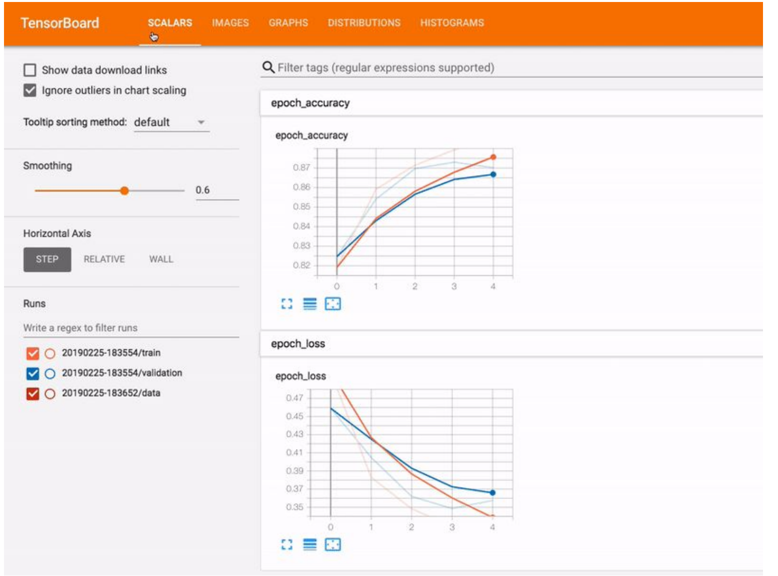
9. Visualize and analyze the results - 2

```
import random
from PIL import Image

num_test_samples = len(test_dataset)
random_idx = random.randint(0, num_test_samples)

topil = transforms.transforms.ToPILImage()
test_input, test_label = test_dataset.__getitem__(random_idx)
test_prediction = F.softmax(my_classifier(test_input.unsqueeze(0).to(device)), dim=1).argmax().item()
print('label : %i' % test_label)
print('prediction : %i' % test_prediction)

test_image = topil(test_input)
test_image.resize((128, 128))
```

```
label : 0
prediction : 0
```

Monitoring the training: Tensorboard

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases, or other tensors as they change over time
- Projecting embeddings to a lower dimensional space
- Displaying images, text, and audio data
- Profiling TensorFlow programs
- And much more



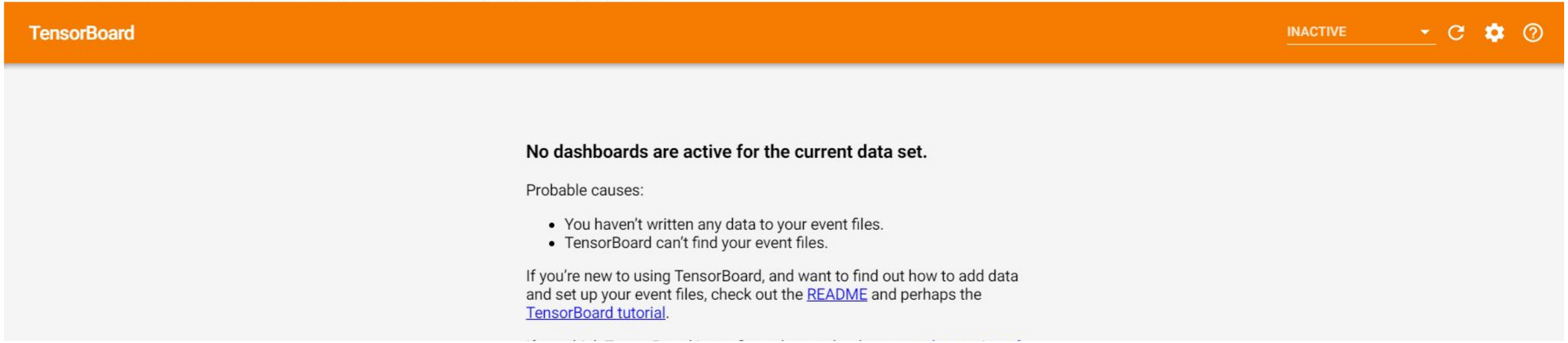
https://www.tensorflow.org/tensorboard

Monitoring the training: Tensorboard

- Colab supports tensorboard by default
- At first, it doesn't show anything since no logs are saved in the directory

[1]
```
from torch.utils.tensorboard import SummaryWriter

# specify a directory in `log_dir` for saving the logs
log_dir = 'logs'
writer = SummaryWriter(log_dir)

# initialize tensorboard
%load_ext tensorboard
%tensorboard --logdir $log_dir
```

```
Reusing TensorBoard on port 6006 (pid 490), started 0:09:33 ago. (Use '!kill 490' to kill it.)
```



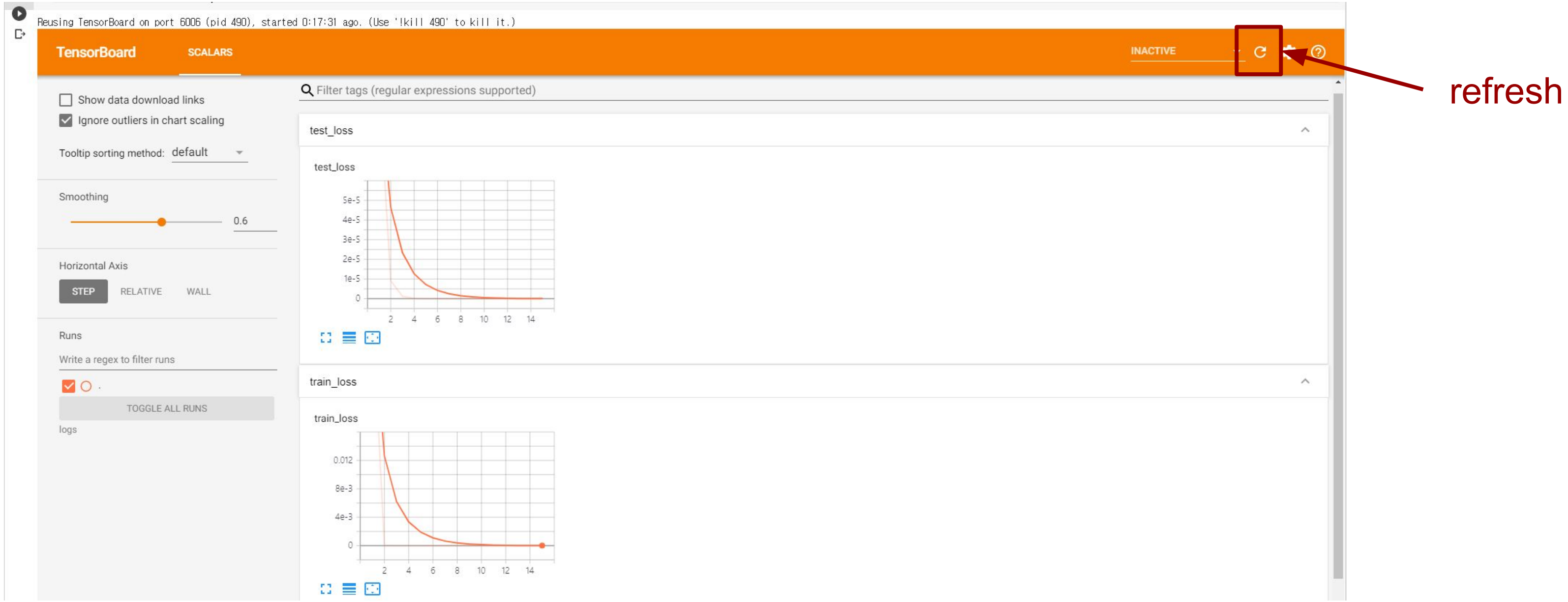
Monitoring the training: T

- Just add <u>two more lines</u>
  into the loop
  to save and visualize the losses

```
# train & test the model
max_epoch = 50
best_test_loss = 999999
for epoch in range(max_epoch):
  train_loss = 0.
  for train_iteration, (x, y) in enumerate(train_dataloader):
    # train_dataloader gives us minibatch data (x, y) at each iteration

    # forward x through the network
    y_pred = regressor(x)
    # compute loss
    loss = compute_loss(y_pred, y)
    train_loss += loss.item()/num_train_data

    # flush previously computed gradients
    optimizer.zero_grad()
    # backward the loss to compute gradients
    loss.backward()
    # update weights using the gradients
    optimizer.step()

  print('epoch : %02d | train loss : %.10f' % (epoch+1, train_loss))
  # log train_loss
  writer.add_scalar('train_loss', train_loss, epoch+1)

  num_test_sample = 0.
  test_loss = 0.
  for x, y in test_dataloader:
    # test_dataloader gives us minibatch data (x, y) at each iteration

    # forward x through the network
    y_pred = regressor(x)
    # compute loss
    loss = compute_loss(y_pred, y)
    test_loss += loss.item()/num_test_data

  print('epoch : %02d |  test loss : %.10f' % (epoch+1, test_loss))
  # log test_loss
  writer.add_scalar('test_loss', test_loss, epoch+1)
  # checkpoint the model weights if necessary (e.g. when the model achieved the best test loss.)
  torch.save(regressor.state_dict(), 'latest_weight.pt')
  if best_test_loss > test_loss:
    best_test_loss = test_loss
    torch.save(regressor.state_dict(), 'best_weight.pt')
```

Monitoring the training: Tensorboard

- Go back to the tensorboard, and press refresh button



More resources

- Official PyTorch tutorials
  - Highly recommend you to go through all four tutorials in it