PyTorch Introduction

2018. 5. 11.

Lee, Gyeongbok

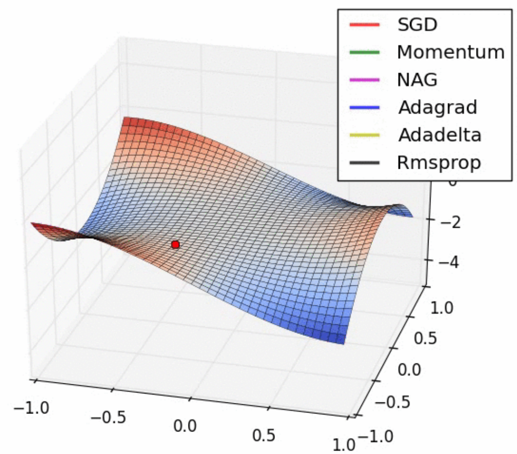
DI-LAB
Data Intelligence Laboratory

Contents

Recap: Training Process

1. Sample labeled data (batch input)
2. Forward it through the network, get predictions
3. Back-propagate the errors
4. Update the network weights

Optimize (min. or max.) objective/cost function $J(\theta)$
Generate error signal that measures difference between predictions and target values



https://medium.com/@ramrajchandradevan/the-evolution-of-gradient-descend-optimization-algorithm-4106a6702d39

Deep Learning Frameworks

- Early days:
  - Caffe, Torch, Theano
- Tensorflow (by Google)
- PyTorch (by Facebook Research)
- DyNet (by CMU)
- Keras (TF backend)

Why use framework?

```python
# -*- coding: utf-8 -*-
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Forward definition

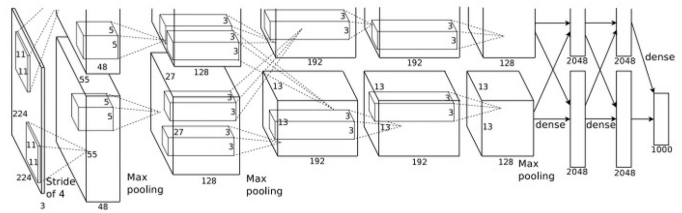
Loss definition

Backpropagate defi

Manual weight update

Without using framework…

We needs to define all of things



$$J(\theta) = -\sum_i ln(\hat{y}_i)$$

Negative Log Likelihood

$$\frac{\partial J}{\partial W_j} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W_j} = -\frac{1}{y}\frac{\partial y}{\partial W_j} = -\frac{1}{\frac{e^{W_y X}}{\sum e^{XW}}}\frac{\sum e^{XW} e^{W_y X} 0 - e^{W_y X} e^{W_j X} X}{(\sum e^{XW})^2} = \frac{X e^{W_j X}}{\sum e^{XW}} = XP$$

$$\frac{\partial J}{\partial W_y} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W_y} = -\frac{1}{y}\frac{\partial y}{\partial W_y} = -\frac{1}{\frac{e^{W_y X}}{\sum e^{XW}}}\frac{\sum e^{XW} e^{W_y X} X - e^{W_y X} e^{W_y X} X}{(\sum e^{XW})^2} = \frac{1}{P}(XP - XP^2) = X(P-1)$$

$$W_i = W_i - \alpha \frac{\partial J}{\partial W_i}$$

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ
Require: Initial parameter x_1
1: Initialize accumulation variables $E[g^2]_0 = 0, E[\Delta x^2]_0 = 0$
2: **for** $t = 1 : T$ **do** %% Loop over # of updates
3: Compute Gradient: g_t
4: Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$
5: Compute Update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t$
6: Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta$
7: Apply Update: $x_{t+1} = x_t + \Delta x_t$
8: **end for**

Why use framework?

```python
# -*- coding: utf-8 -*-
import torch

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

Forward definition
(Named!)

Loss definition
(Pre-defined!)

Backpropagate definition
(Done by machine! - with autograd)

Automatic weight update
(Predefined!)

With using framework…

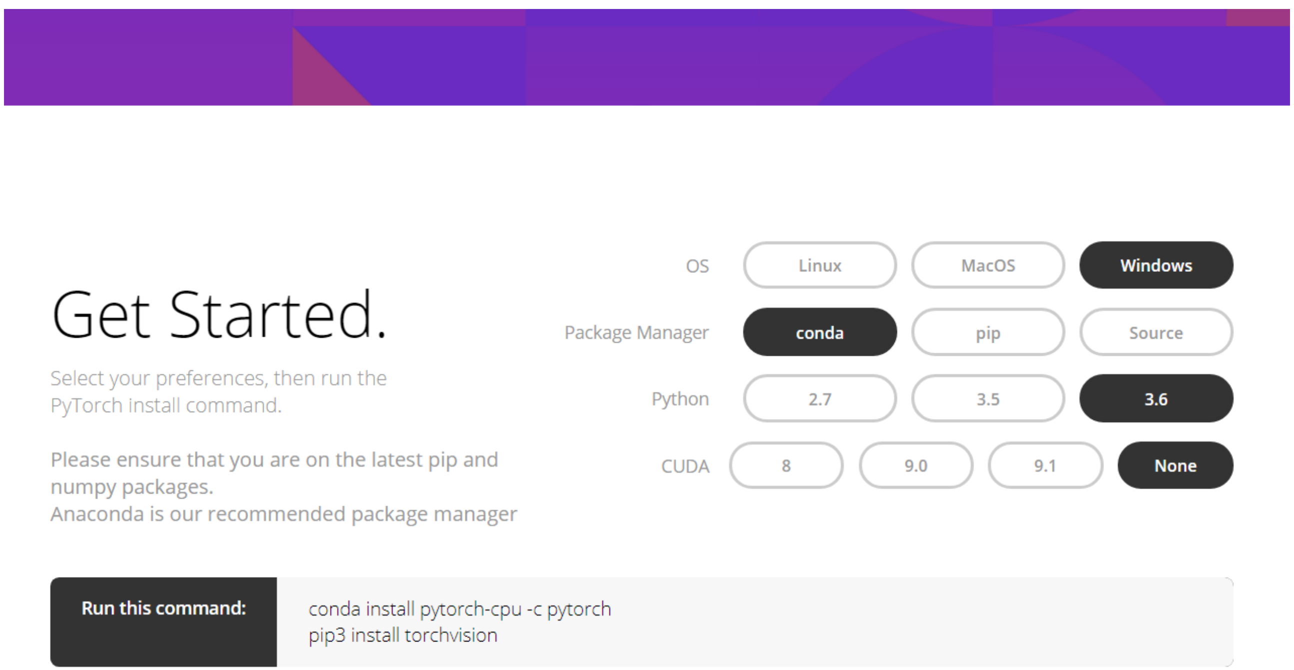
You can focus on your model!

Many optimizers & loss functions provided by the framework

(of course, you can manually define your own function)

PyTorch Installation

- Follow instruction in the website
  - current version: 0.4.0
  - Set cuda if you have Nvidia GPU and CUDA installed
  - Strongly recommend to use Anaconda for Windows



Programming Pattern

- Data Provider
  - Pre-processing data
- Design model (network)
  - Define as class
- Construct loss / optimizer
  - Using provided APIs
- Training cycle
  - Implement for-loop of forward, backward, update

How to provide data?

- All data type used in PyTorch is tensor
  - Similar with numpy array, but can be used in GPU

```
>>> torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])
```

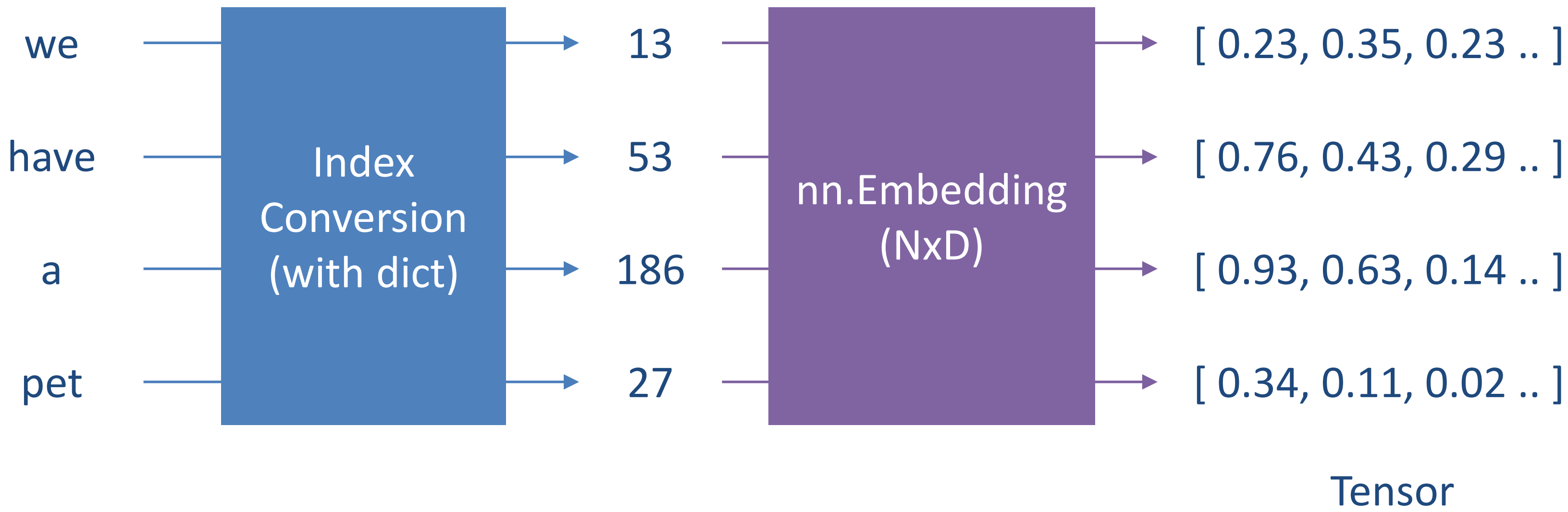
  - Provides memory sharing (numpy)

```
>>> a = numpy.array([1, 2, 3])
>>> t = torch.from_numpy(a)
>>> t
tensor([ 1,  2,  3])
>>> t[0] = -1
>>> a
array([-1,  2,  3])
```

  - Other creation ops (fill with zero, one, random, etc)
    - https://pytorch.org/docs/stable/torch.html#creation-ops

How to provide data?

- How about text data?
  - Treat as symbol: vocabulary
    - You may use some dictionary for conversion
  - Using word embedding
    - https://pytorch.org/docs/master/nn.html#sparse-layers



PyTorch Network

What we have to consider:
Structure and forward process of neural network (in common case)

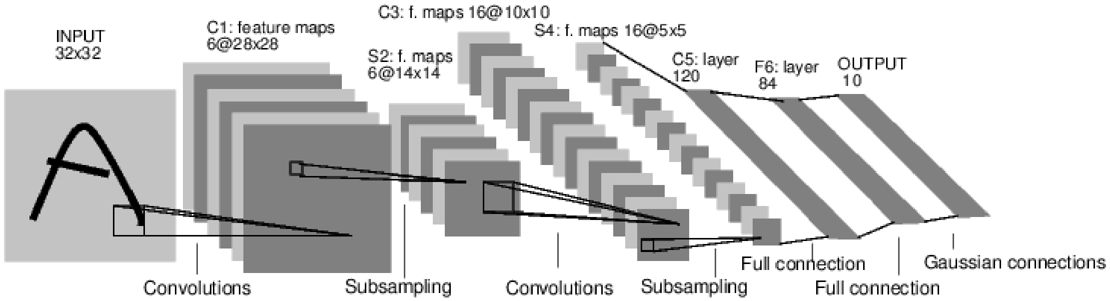
```
import torch.optim as optim

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

- `net = Net()` — NN is modularized as a class
- `criterion = nn.CrossEntropyLoss()` — Loss function (pre-defined)
- `optimizer = optim.SGD(...)` — Optimizing Algorithm (Stochastic Gradient Descent, pre-defined)

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

- Load the train data (with iteration)
- Forward NN
- Backpropagate + Update weights

Neural Network in PyTorch




```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

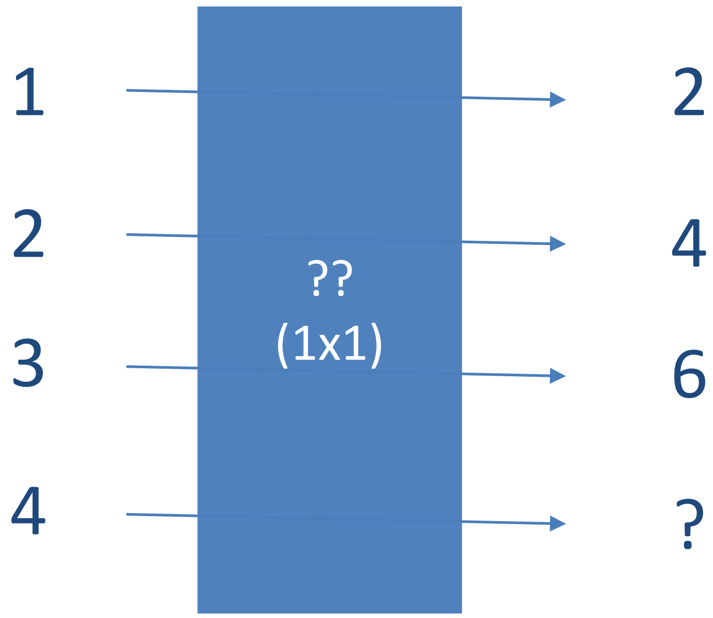
Define each Neural Network Components

Define what to do with Neural Network

https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py

Try with very simple example

- 1x1 Linear



Data

- Let's make tensor

```
>>> import torch
>>> x_data = torch.tensor([[1.0],[2.0],[3.0]])
>>> y_data = torch.tensor([[2.0],[4.0],[6.0]])
>>> print(x_data)
tensor([[ 1.],
        [ 2.],
        [ 3.]])
>>> print(y_data)
tensor([[ 2.],
        [ 4.],
        [ 6.]])
```

Network

- Let's make network (1x1 Linear)

```
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        y_pred = self.linear(x)
        return y_pred

model = Model()
```

Inheritance from class nn.Module

Linear Matrix

Define forward action

Loss/Optimizer

- Let's use PyTorch API

MSE: Mean Squared Error

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

```
import torch.optim as optim

criterion = nn.MSELoss(size_average=False)
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Backpropagation for all parameters in model

List of loss functions
- L1 Loss
- MSE Loss
- CrossEntropy Loss
- NLL Loss
...

https://pytorch.org/docs/master/nn.html#loss-functions

List of optimizer
- SGD
- Adam
- Adagrad
- RMSProp
...

https://pytorch.org/docs/master/optim.html

Training Loop

```
N = 500
for epoch in range(N):
    y_pred = model(x_data)

    loss = criterion(y_pred, y_data)
    print(epoch, loss.item())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Train 500 times

(try yourself: How does N affects the result?)

convert scalar tensor to python scalar

“clean” gradients

Training result

```
antest1@ubuntu:~/story/model$ python3 t.py
0 52.7883186340332
1 23.500415802001953
2 10.462265014648438
3 4.658045291900635
4 2.074162006378174
5 0.9238845705986023
6 0.4118058681488037
7 0.18383538722991943
8 0.08234208822250366
9 0.03715283423662186
10 0.017028830945491791
11 0.008063172921538353
12 0.004064972512423992

          ....

491 1.5964578778948635e-05
492 1.5734132830402814e-05
493 1.550719389342703e-05
494 1.528497159597464e-05
495 1.5065820662130136e-05
496 1.4849933904770296e-05
497 1.4634493709309027e-05
498 1.4426146663026884e-05
499 1.4218197065929417e-05
```

Error goes smaller!

Prediction

- We use forward only for testing

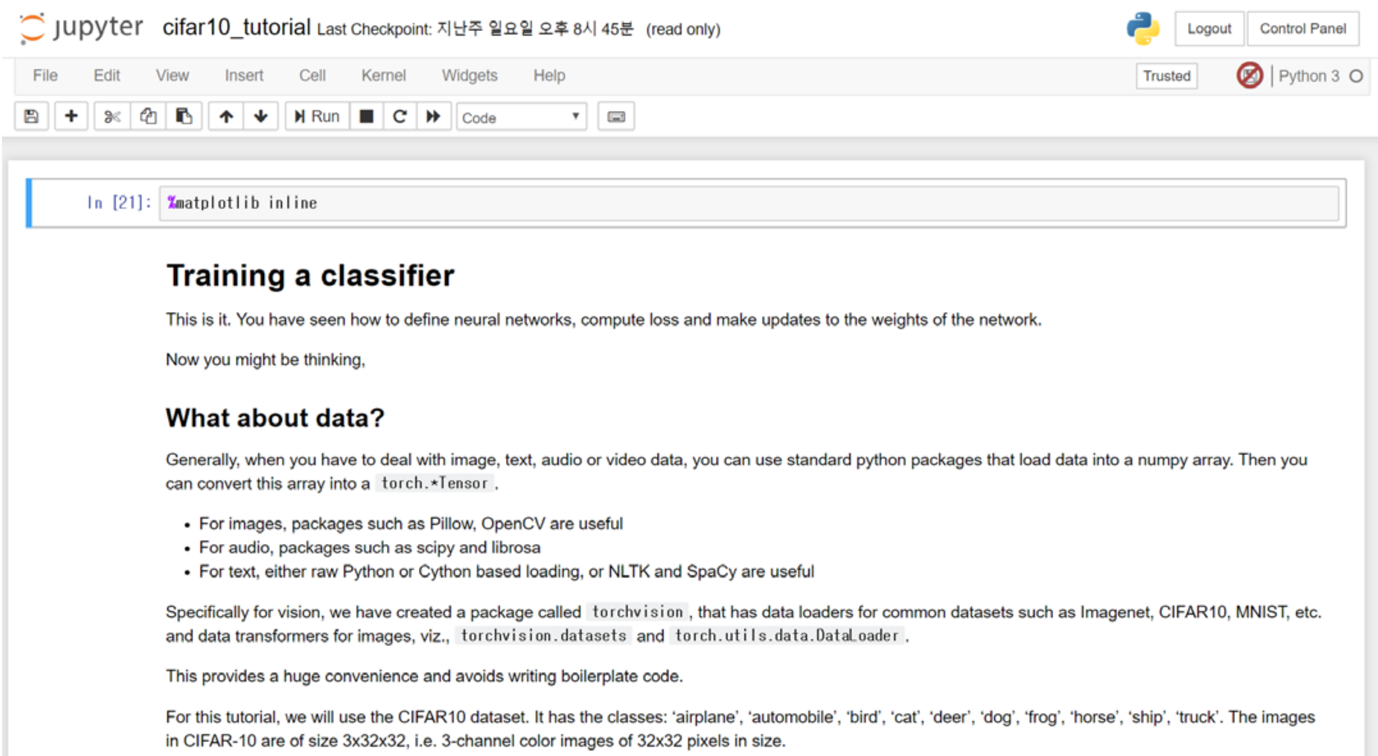
```
test_val = torch.tensor([4.0])
print("prediction result for {}: {}".format(
    4.0, model.forward(test_val).detach().numpy()[0]))
```

Why detach?
- detach from graph / not used for gradient calculation

```
prediction result for 4.0: 8.00433449768066
```

(exact number can be different)

CNN Example


jupyter cifar10_tutorial Last Checkpoint: 지난주 일요일 오후 8시 45분 (read only)

Logout Control Panel

File Edit View Insert Cell Kernel Widgets Help

Trusted Python 3

Run Code

In [21]:
```
%matplotlib inline
```

Training a classifier

This is it. You have seen how to define neural networks, compute loss and make updates to the weights of the network.

Now you might be thinking,

What about data?

Generally, when you have to deal with image, text, audio or video data, you can use standard python packages that load data into a numpy array. Then you can convert this array into a `torch.*Tensor`.

- For images, packages such as Pillow, OpenCV are useful
- For audio, packages such as scipy and librosa
- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful

Specifically for vision, we have created a package called `torchvision`, that has data loaders for common datasets such as Imagenet, CIFAR10, MNIST, etc. and data transformers for images, viz., `torchvision.datasets` and `torch.utils.data.DataLoader`.

This provides a huge convenience and avoids writing boilerplate code.

For this tutorial, we will use the CIFAR10 dataset. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.

Download Source Code Here:

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Other References

- https://pytorch.org/docs/stable/
  - Reference is our good friend

- Tutorials
  - https://github.com/jcjohnson/pytorch-examples
  - Below resources can be out-dated: needs code changes
    - Check https://pytorch.org/2018/04/22/0_4_0-migration-guide.html
  - https://github.com/hunkim/PyTorchZeroToAll
    - PyTorch Materials used in HKUST
  - https://cs230-stanford.github.io/pytorch-getting-started.html

NLP Implementations in PyTorch

- https://github.com/DSKSD/DeepNLP-models-Pytorch
  - ※ may not work in 0.4.0

Model	Links
01. Skip-gram-Naive-Softmax	[notebook / data / paper]
02. Skip-gram-Negative-Sampling	[notebook / data / paper]
03. GloVe	[notebook / data / paper]
04. Window-Classifier-for-NER	[notebook / data / paper]
05. Neural-Dependancy-Parser	[notebook / data / paper]
06. RNN-Language-Model	[notebook / data / paper]
07. Neural-Machine-Translation-with-Attention	[notebook / data / paper]
08. CNN-for-Text-Classification	[notebook / data / paper]
09. Recursive-NN-for-Sentiment-Classification	[notebook / data / paper]
10. Dynamic-Memory-Network-for-Question-Answering	[notebook / data / paper]