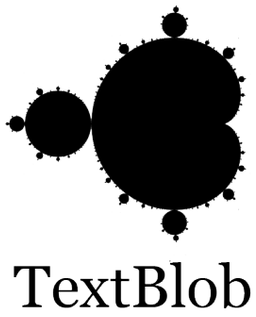
Deep Learning and Computer Vision

Adham Ehab
tw/gh: @adhaamehab

slides and code @ https://cutt.ly/cvsc19

About me

- Full Stack Data scientist at **ArqamFC** (StatsBomb company)
- Worked at 4 different startups in 5 years
- Graduated this year from FCIS
- Writer at Towards Data Science
- Maintainer of TextBlobAR
- Active open source contributor since 2018



Agenda

1. **Introduction**

2. **Classical Computer Vision algorithms** (25 Mins).

3. **Introduction to OpenCV** (25 mins).

4. **Computer vision as a platform for deep learning** (15 mins).

5. **Deep learning Frameworks** (10 mins)

6. **Learning to learn with Pytorch** (50 mins)

7. **What's Full stack data science? and Why you should care?** (20 mins)

8. **A full stack Scene parsing model** (45 mins)

9. **A deeper neural networks (state-of-the-art).** (15 mins)

10. **Discussions.**

Goals

Or what you will gain from spending 4 hours listening to me?

- **Learn the basic theories behind classical cv algorithms.**
- **Learn to use OpenCV to implement more complex algorithms.**
- **Learn about the different deep-learning frameworks.**
- **Learn about full stack data science.**
- **Learn to use Pytroch to train your own models.**
- **Learn to use pre-trained models with Pytroch.**
- **Build a Full stack scene segmentation app.**
- **Learn about current research state-of-the-art.**

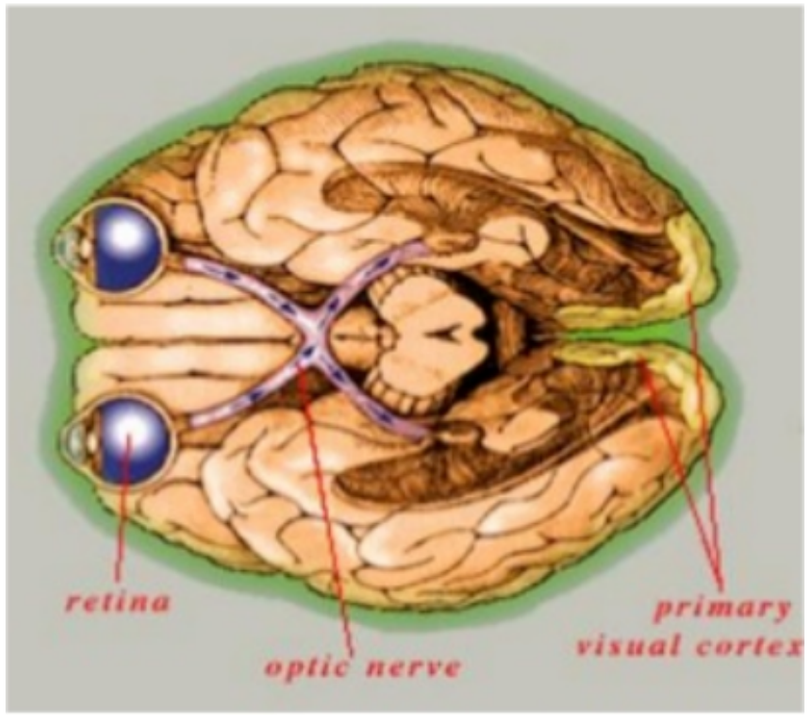
Introduction

The goal of computer vision

- To bridge the gap between pixels and “meaning”

0	3	2	5	4	7	6	9	8
3	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
5	2	3	0	1	2	3	4	5
4	3	2	1	0	3	2	5	4
7	4	5	2	3	0	1	2	3
6	5	4	3	2	1	0	3	2
9	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Introduction



25% of the whole brain is for vision.

80% of the brain is associated with vision in someway

Introduction

Computer vision

“acquiring”, “processing”, “analyzing” and “understanding”
digital images and extraction of high-dimensional data from
the real world in order to produce numerical or symbolic
, in the forms of decisions

Introduction

Image processing

A method to convert an image into digital form and perform some operations on it, in order to get and enhanced image or to extract some useful information from it

Introduction

Digital representation of an image

90	67	68	75	78	98	185	180	153	139	132	106	70	80	81	69	69	67	35	34
92	87	73	78	82	132	180	152	134	120	102	106	95	75	72	63	75	42	19	29
63	102	89	76	98	163	166	164	175	159	120	103	132	96	68	42	49	46	17	22
45	83	109	80	130	158	166	174	158	134	105	71	82	121	80	51	12	50	31	17
39	69	92	115	154	122	144	173	155	105	98	86	82	106	83	76	17	29	41	19
34	80	73	132	144	110	142	181	173	122	100	88	141	142	111	87	33	18	46	36
37	93	88	136	171	164	137	171	190	149	110	137	168	161	132	96	56	23	48	49
66	117	106	147	188	202	198	187	187	159	124	151	167	158	138	105	80	55	59	54
127	136	107	144	188	197	188	184	192	172	124	151	138	108	116	114	84	46	67	54
143	134	99	143	188	172	129	127	179	167	106	118	111	54	70	95	90	46	69	52
141	137	96	146	167	123	91	90	151	156	121	93	78	82	97	91	87	45	66	39
139	137	80	131	162	145	131	129	154	161	158	149	134	122	115	99	84	35	52	30
137	133	56	104	165	167	174	181	175	169	165	162	158	142	124	103	67	19	31	23
135	132	65	86	173	186	200	198	181	171	162	153	145	135	121	104	53	14	15	33
132	132	88	50	149	182	189	191	186	178	166	157	148	131	106	78	28	10	15	44

Classical Computer Vision

- Understanding Features.
- Introduction to Harris corner detection.
- Introduction to descriptors and SIFT.
- Feature Matching and Image alignment.

Features

If we aim to teach a computer how to solve a Jigsaw puzzle.

What are the projecting theory that a computer needs to follow in order to solve the puzzle?

Features

If the computer can play jigsaw puzzles, why can’t we give a lot of real-life images of a good natural scenery

to computer and tell it to stitch all those images to a big single image?

If the computer can stitch several natural images to one,

what about giving a lot of pictures of a building or any structure and tell computer to create a 3D model out of it?

Features

In order to solve this challenging problem we need to answer one basic questions.

Features

How do you solve a jigsaw puzzles?

Features

And the answer is . . .

Features & Patterns

Features

we are looking for specific patterns or different features which are unique, which can be easily tracked, and easily compared.

Features

We search for these features in an image, we find them, we find the same features in other images, we align them.
That's it.

That's how you solve a jigsaw puzzle

Features

Features are a unique areas in images which we can identify images with (Like Jigsaw corner peaces)

Features

so, How can I teach a computer to find those features in an image?

We will discover this in the next few slides

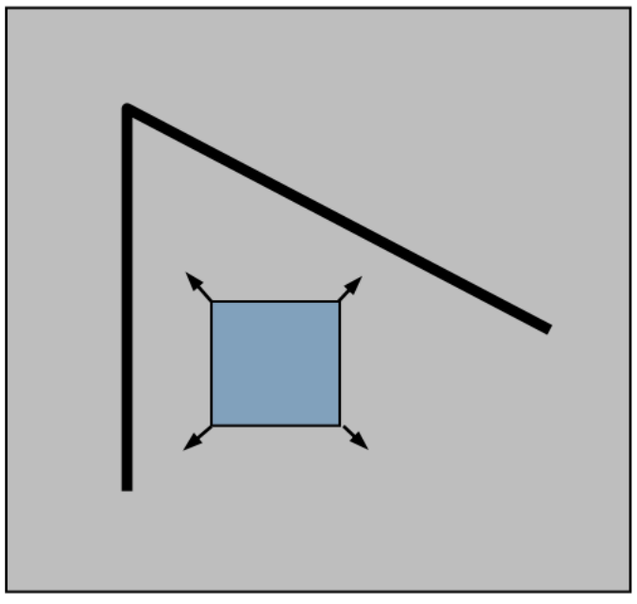
Features

How do we measure uniqueness of a feature?

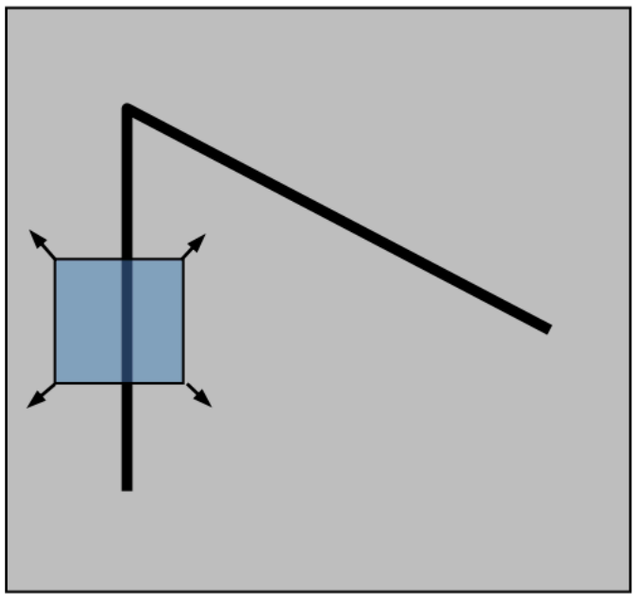
Assume we have a small window of pixels.

The uniqueness of the area of pixels of this window can be measure by *how much change happens when shifting the window*

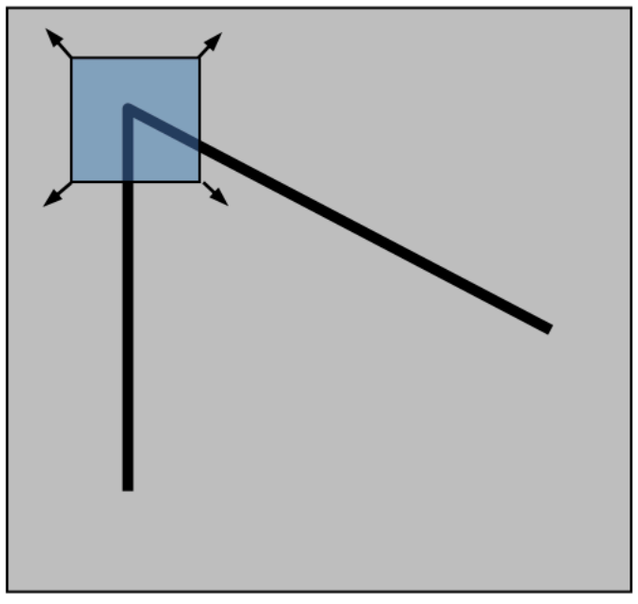
Features



“flat” region:
no change in all directions



“edge”:
no change along the edge direction

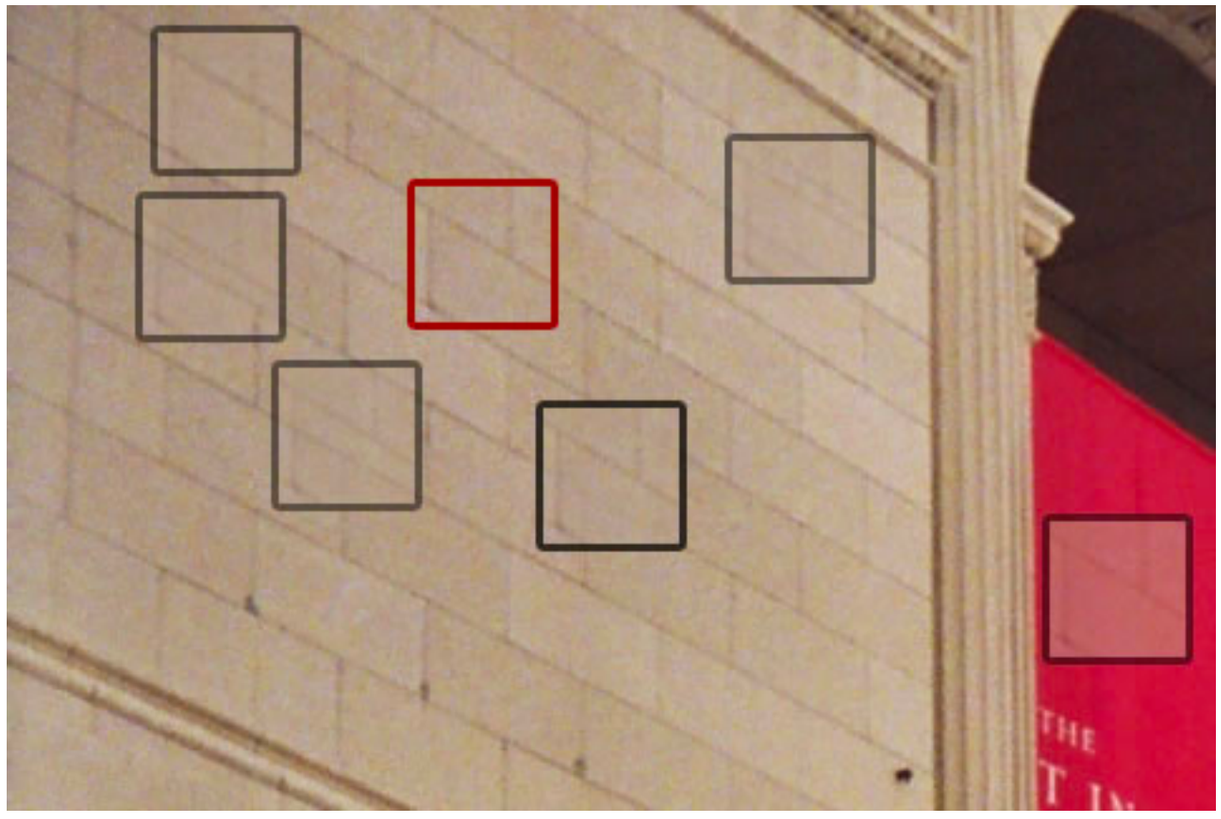


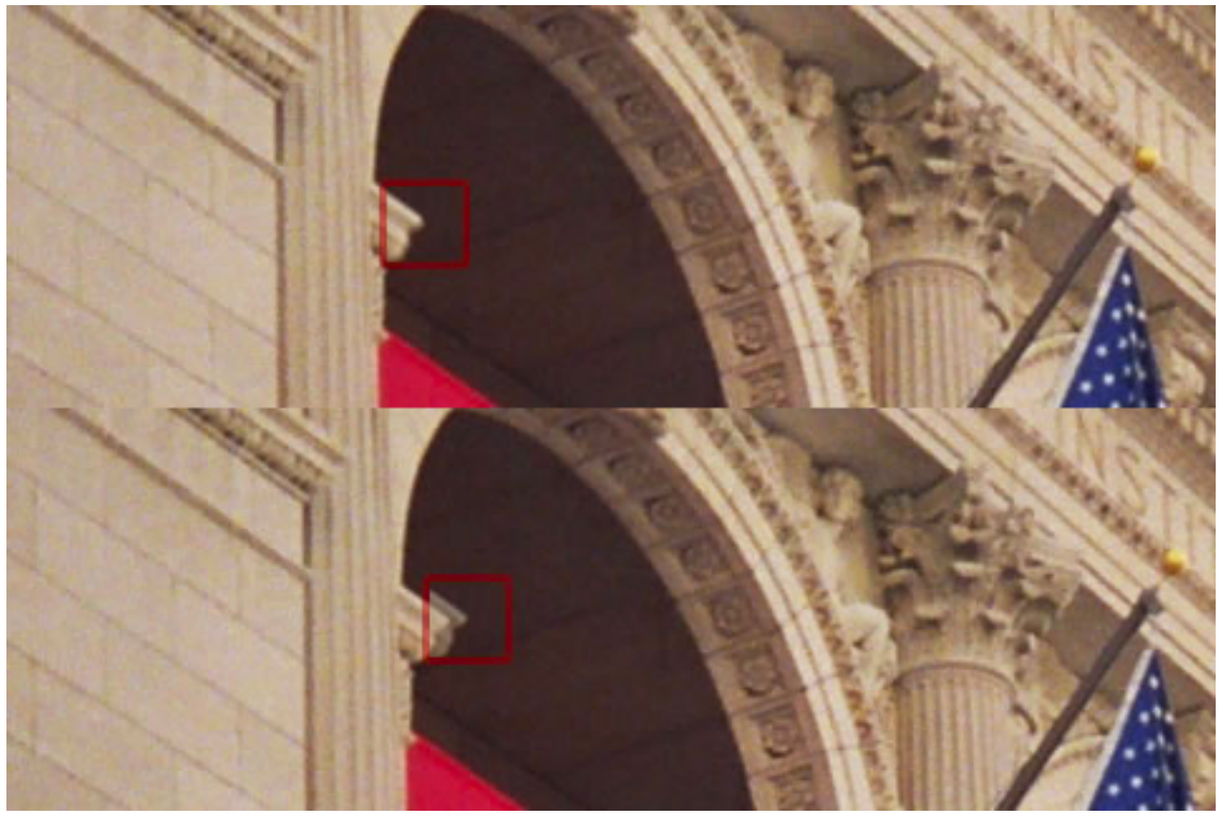
“corner”:
significant change in all directions

Features

So, to measure the most unique features we need to find which feature happens to locate on a corner.

Thus, Locating corners will means locating the unique features.





Harris Corner Detector

Harris Corner Detector is a corner detection operator that is commonly used to extract corners and infer features of an image.

Harris Corner Detector

Main steps for Harris

- Preprocessing (Gaussian and Sobel)
- Find a suitable window size.
- Calculate the difference `E(u, v)` for each window and it's variation.
- Select most unique windows with highest `E` value using thresholding.
- Calculate the score `R` for every selected window using eigenvalues
- All windows that have a score R greater than a certain value are corners. They are good tracking points.

Harris Corner Detector

- Let's define our window with size 3 X 3.

- we take the sum squared difference (SSD) of the pixel values before and after the shift and identifying pixel windows where the SSD is large for shifts in all 8 directions.

- Let us define the change function `E(u, v)` as the **sum** of all the sum squared differences (SSD)

  `u, v` are the `x, y` coordinates of every pixel in our 3 x 3 window.

  `I` is the intensity value of the pixel.

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2$$

Harris Corner Detector

The difference equation

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$$

Using Taylor Expansion

$$E(u,v) \approx [u \quad v] \left(\sum \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

Rename the summed-matrix

$$M = \sum w(x,y) \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix}$$

Now the equation become

$$E(u,v) \approx [u \quad v]\, M \begin{bmatrix} u \\ v \end{bmatrix}$$

Harris Corner Detector

By solving for the eigenvectors of M to maximize change in all directions

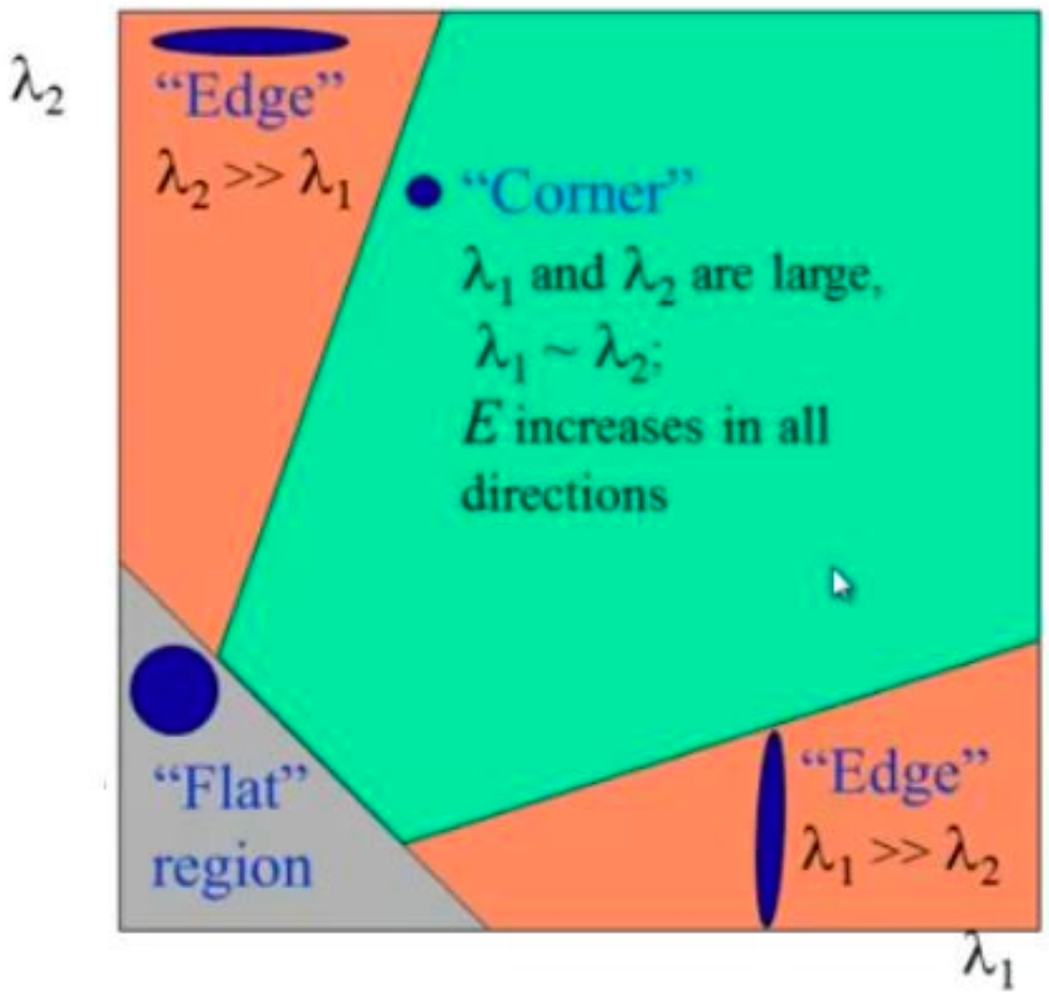
$$R = \det M - k(trace\, M)^2$$

$$\det M = \lambda_1 \lambda_2$$

$$trace\, M = \lambda_1 + \lambda_2$$

$\lambda 1$ and $\lambda 2$ are the eigenvalues of M.
So the values of these eigenvalues decide whether a region is a corner, edge or flat

Harris Corner Detector



Features

Now as we can find the most unique features in any image.

Is that all?

No, For sure :)

Image Descriptors

Image Descriptors

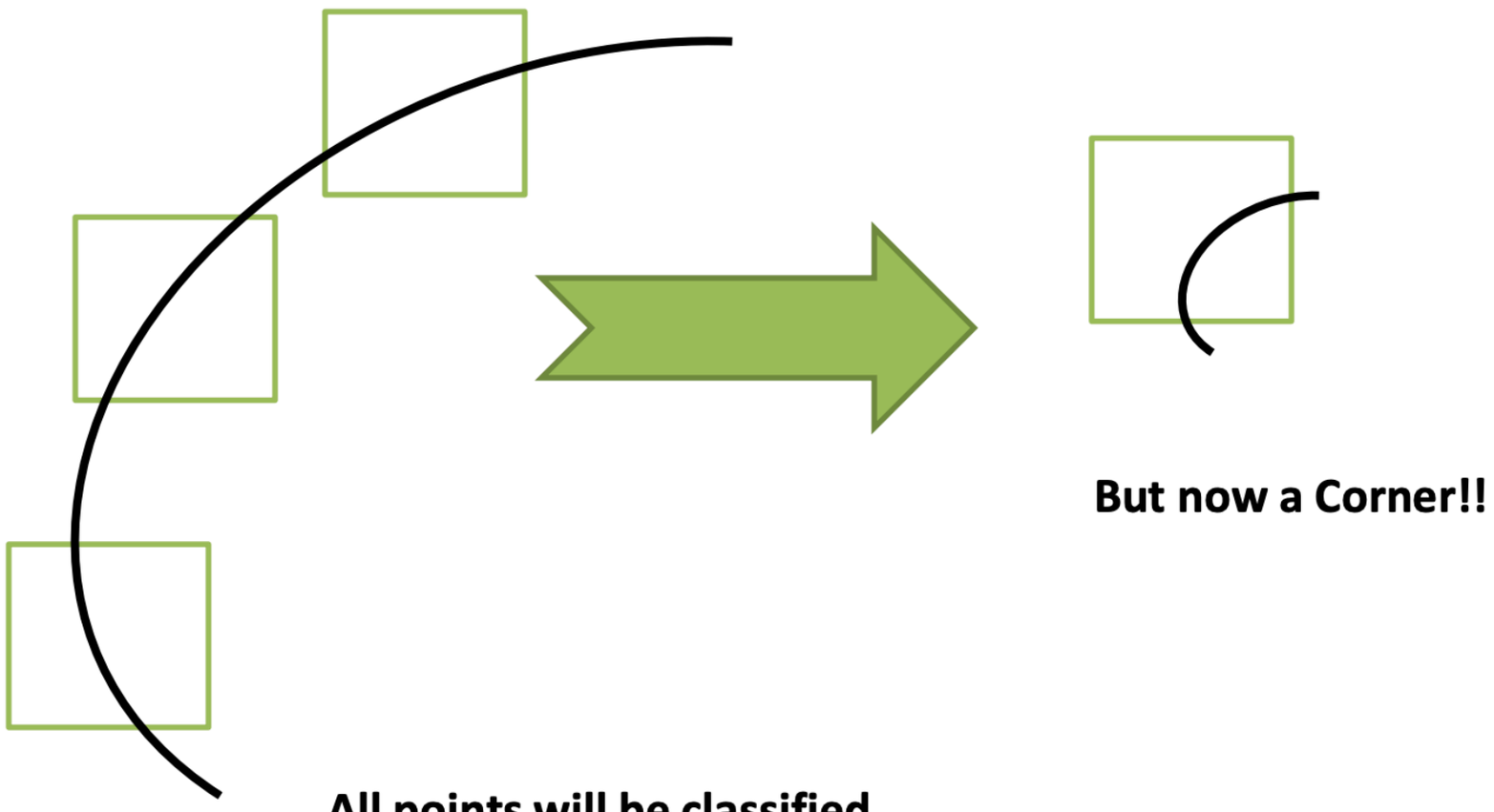
Image Descriptors are algorithms which extract a set of unique features for an image.

SIFT DESCRIPTOR

Scale-invariant feature transform (aka. SIFT) is a feature detection and description algorithm which tries to find key points in images which are not sensitive to changes in image resolution, scale, rotation, changes in illumination (eg, position of lights)

SIFT vs HARRIS

Harris corner detection is variant for scaling, rotation and intensity change.



SIFT

- The core idea of SIFT is to create a robust feature extractor which overcome Harris problems (Invariance for affine transformation). And not just find the features but to be able to describe them. *Hence the name*

SIFT

Steps for extracting Key points

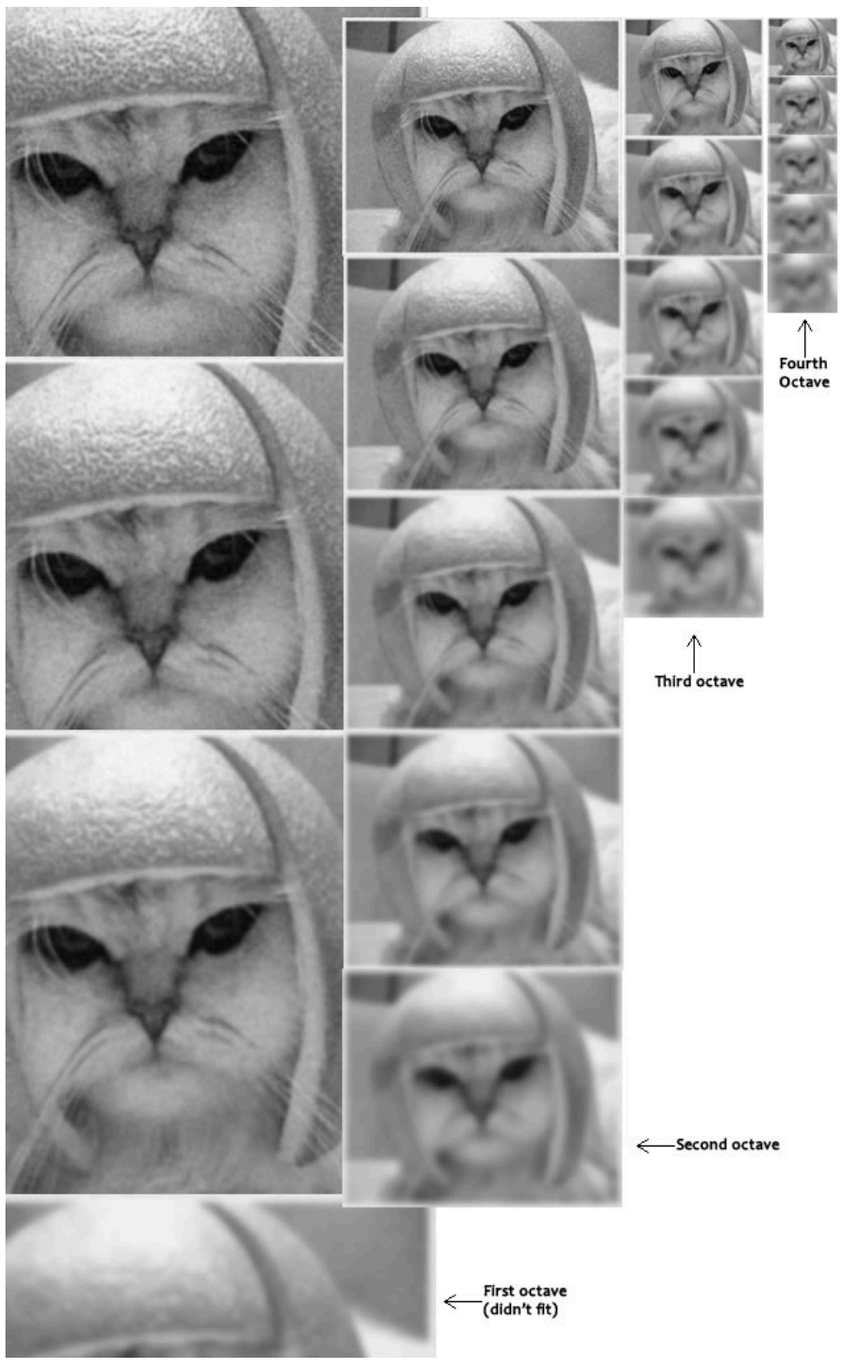
- Scale space peak selection
  - Potential locations for finding features
- Key point localization
  - Accurately locating the features key points.
- Orientation Assignment
  - Assigning orientation to the key point
- Key point descriptor
  - Describing the key point as a high dimensional vector

SIFT

1 - Scale space peak selection

For the input image, generate progressively blurred out versions using Gaussian filter with different scale level.

SIFT


Fourth Octave
Third octave
Second octave
First octave (didn't fit)

SIFT

1 - Scale space peak selection

Applying Laplacian of gaussian approximation (LoG) on each blurred blurred image in all scale levels to extract the key point of high construct (Potential features). *But log very expensive! :(*

So, In practice we use the difference of gaussian which gives almost the same result in much less time. (Why?)

- *Subtraction is much much faster than division.*

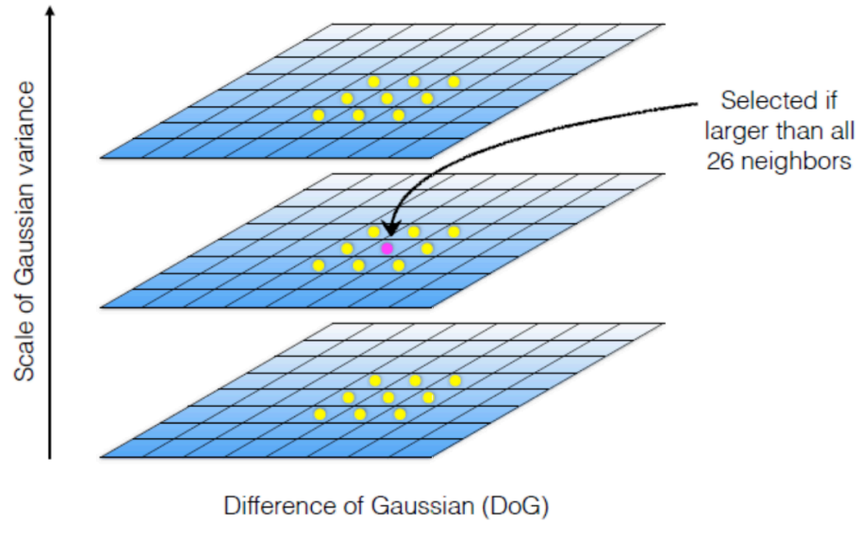
SIFT

2 - Key point localization

After we extract the potential key points we need to find the accurate locations of the key points. How?

By iterating over all neighbours point in each scale above and below and mark the point as a key point if and only if it’s greater or smaller from all neighbours.

For each max or min found, output is the **location** and the **scale**.



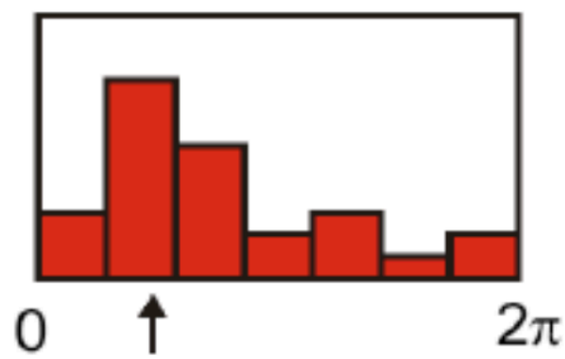
SIFT

3- Orientation Assignment

To give SIFT the ability of scale-transform invariance. We calculate an orientation assignment for each key point.

Take a window (16x16) around each key point and calculate the gradient magnitude and orientation of each pixel and assign the most dominant orientation(s) in this region to the key point

We create histogram of local gradient directions at selected scale – 36 bins (each 10 degrees) and select the highest peak



SIFT

4 - Key point descriptor

To make sure that each key point is unique SIFT describe each feature with the following:

- Location in pixels `X, Y`

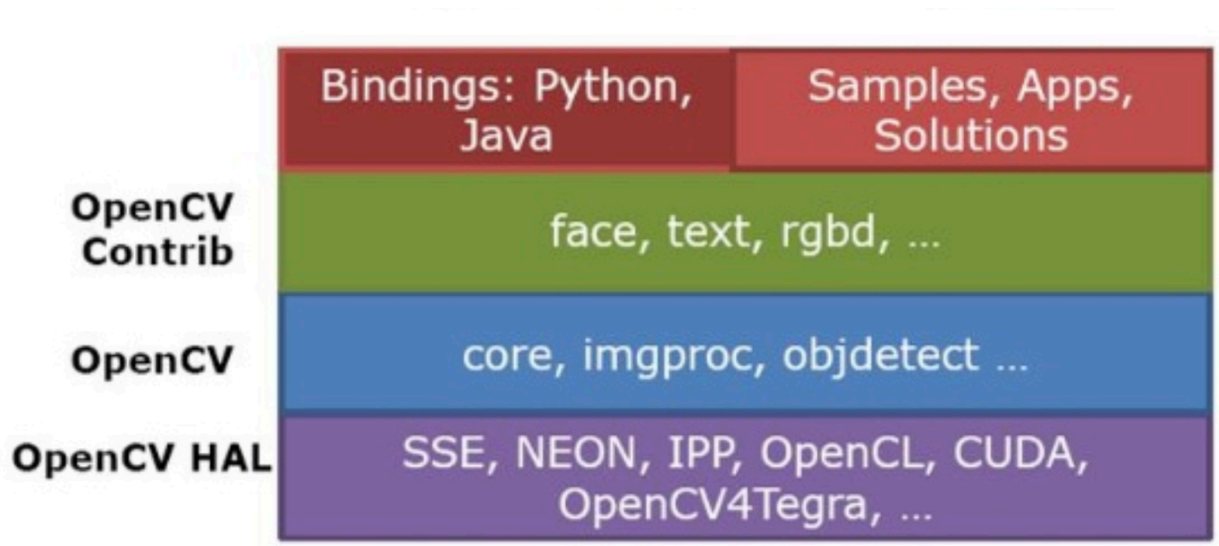
- Scale σ

- Gradient magnitude `M`

- Orientation θ

Open CV

- BSD license, ***12M*** downloads, ***700k+*** lines of code.
- Huge community, Open source.
- Runs on every platform (OSX, *nix, windows, mobile OS, etc)



- Find more at http://opencv.org (user)
- Or http://code.opencv.org (developer)

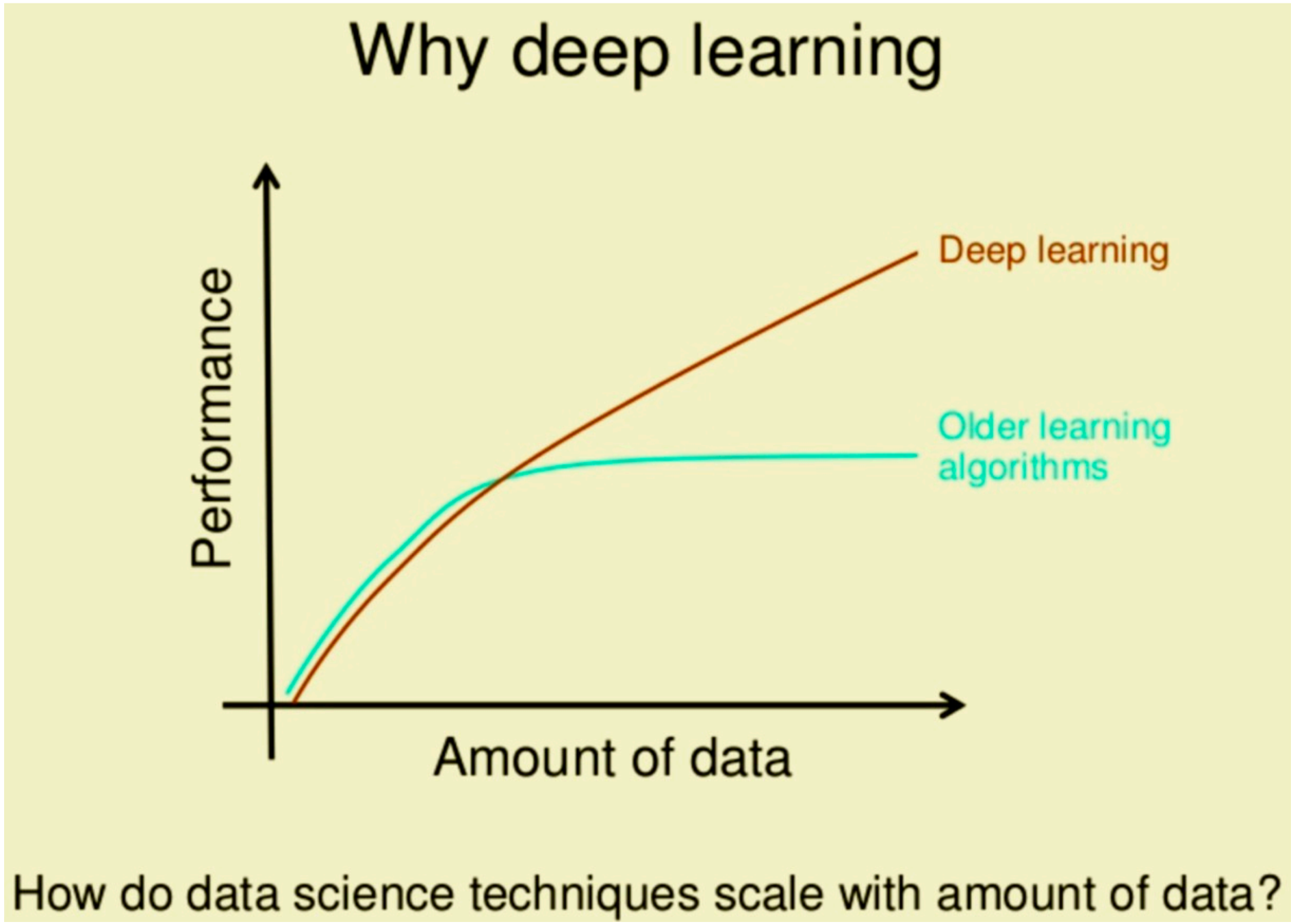
Talk is cheap.

Show me the Code

Deep Learning

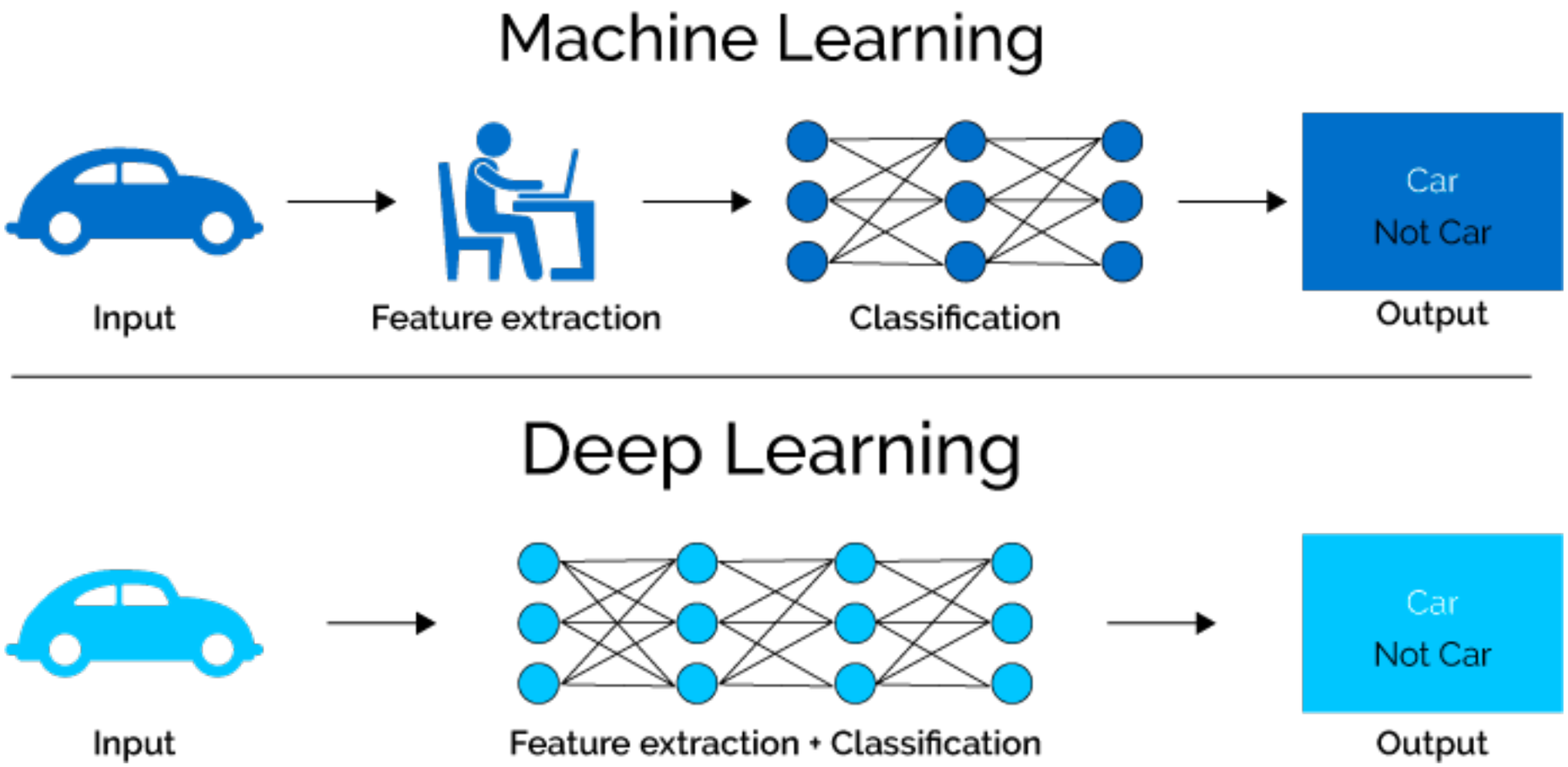
Why going Deep?

- **Less Data**



Why going Deep?

- **More automated, less hyperparams**



Why going Deep?

- **Going too far than traditional vision algorithms**

Browse state-of-the-art

1028 leaderboards • 1187 tasks • 1041 datasets • 13894 papers with code

Follow on Twitter for updates

paperswithcode.com

Computer Vision

Semantic Segmentation

19 leaderboards

487 papers with code

Image Classification

46 leaderboards

422 papers with code

Object Detection

36 leaderboards

354 papers with code

Image Generation

37 leaderboards

176 papers with code

Denoising

13 leaderboards

170 papers with code

See all 637 tasks

Hundreds of possible tasks with deep learning in vision

Why going Deep?

The five promises of deep learning for computer vision are as follows:

- ***The Promise of Automatic Feature Extraction***. Features can be automatically learned and extracted from raw image data.
- ***The Promise of End-to-End Models.*** Single end-to-end models can replace pipelines of specialized models.
- ***The Promise of Model Reuse***. Learned features and even entire models can be reused across related tasks.
- ***The Promise of Superior Performance.*** Techniques demonstrate better skill than classical methods on challenging tasks.
- ***The Promise of General Method.*** A single general method (e.g. convolutional neural networks) can be used on a range of related tasks.

Deep learning Frameworks

Deep learning Frameworks

Deep learning Frameworks

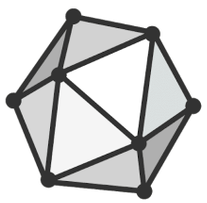
TensorFlow

Keras

Deep learning Frameworks

TensorFlow

Keras



ONNX

theano

mxnet

Deep learning Frameworks

PYTORCH

Introduction to Pytorch

What is PyTorch?

- Developed by Facebook
  - Python first
  - Dynamic Neural Network

- Endorsed by Director of AI at Tesla

Also, this guy created SKLearn

Packages of PyTorch

This Tutorial

Package	Description
torch	a Tensor library like Numpy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience
torch.legacy(.nn/.optim)	legacy code that has been ported over from torch for backward compatibility reasons

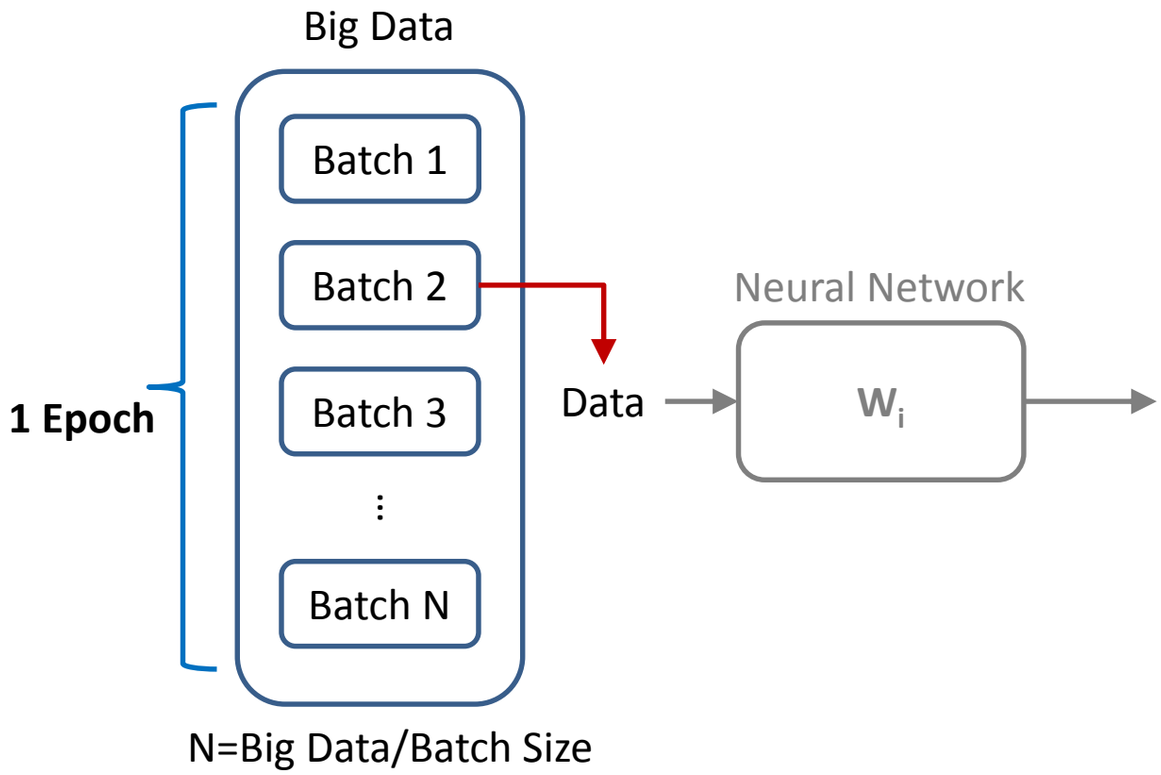
Neural Network in Brief

- Supervised Learning
  - Learning a function f, that f(x)=y

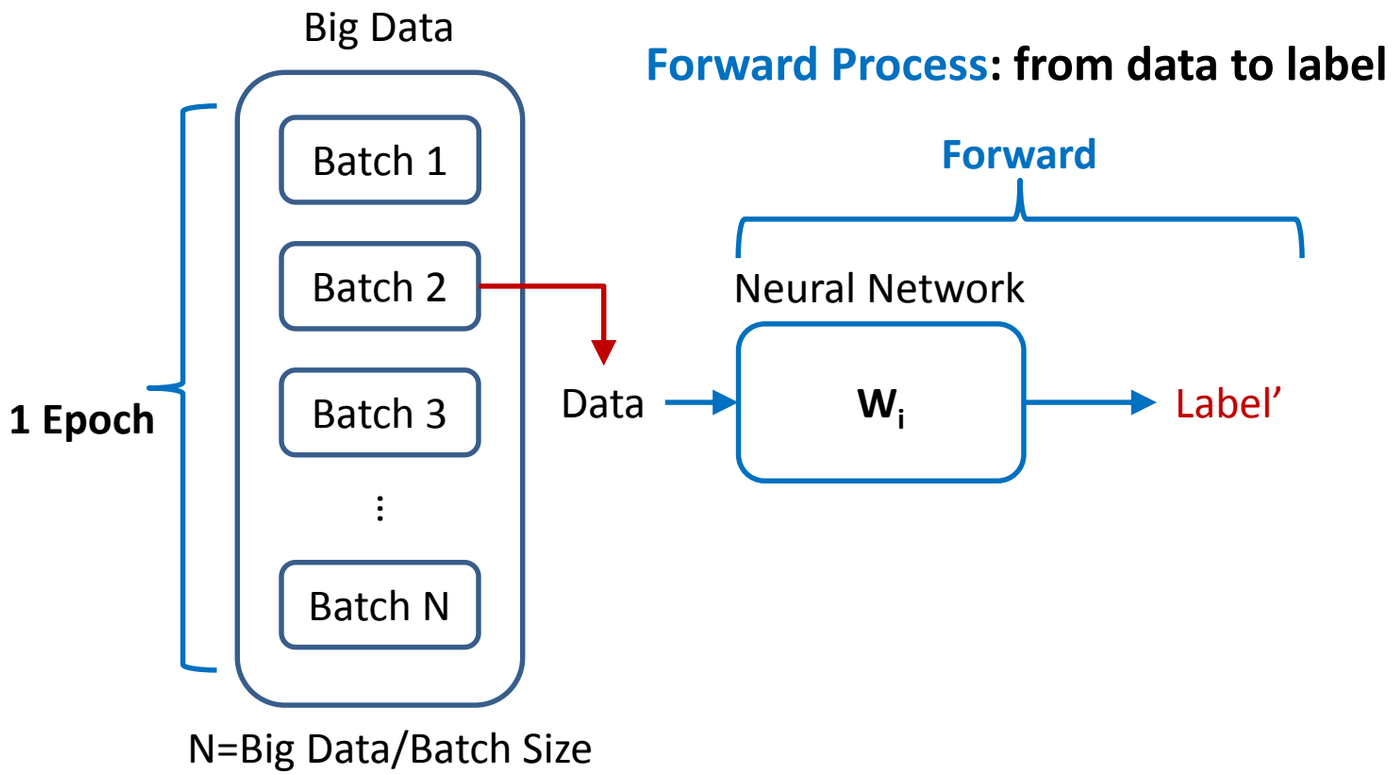
Trying to learn f(.), that f(x)=y

Data	Label
X1	Y1
X2	Y2
…	…

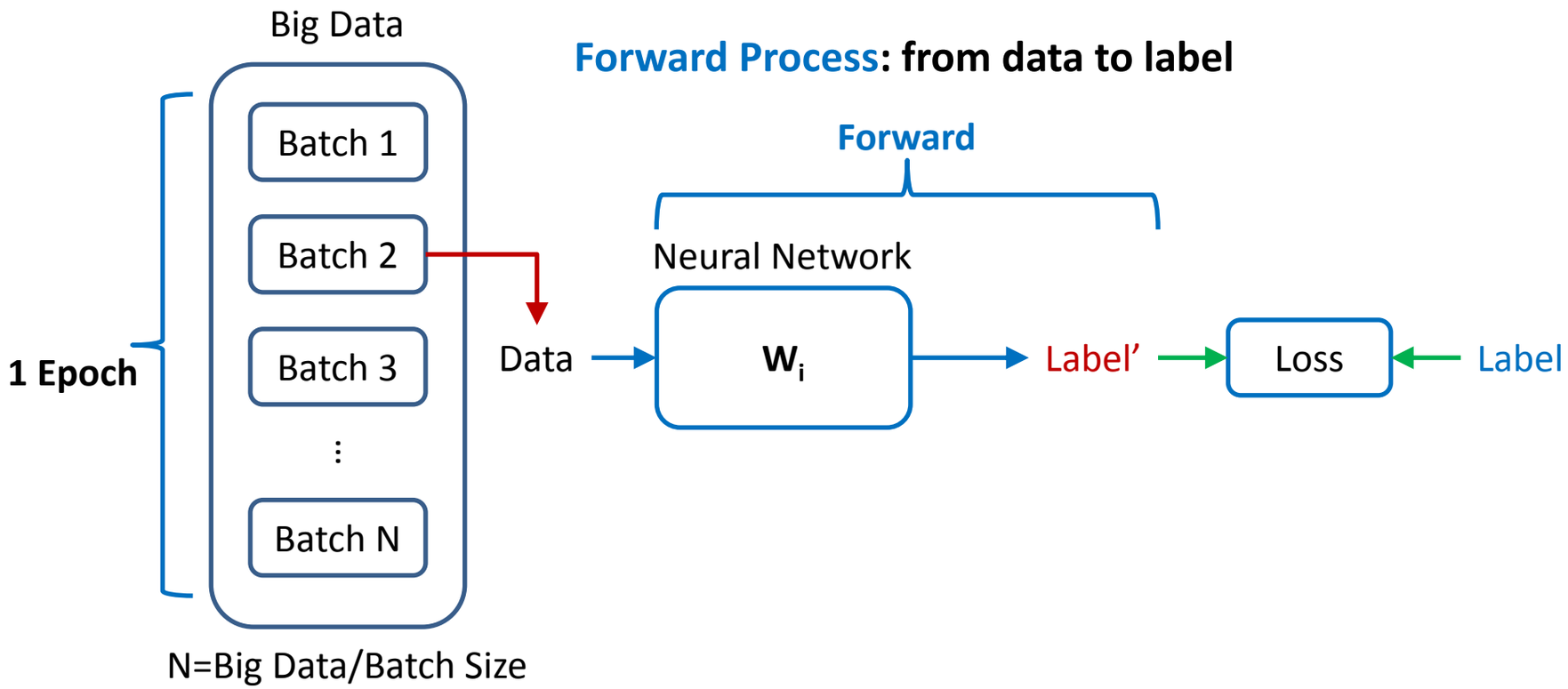
Neural Network in Brief



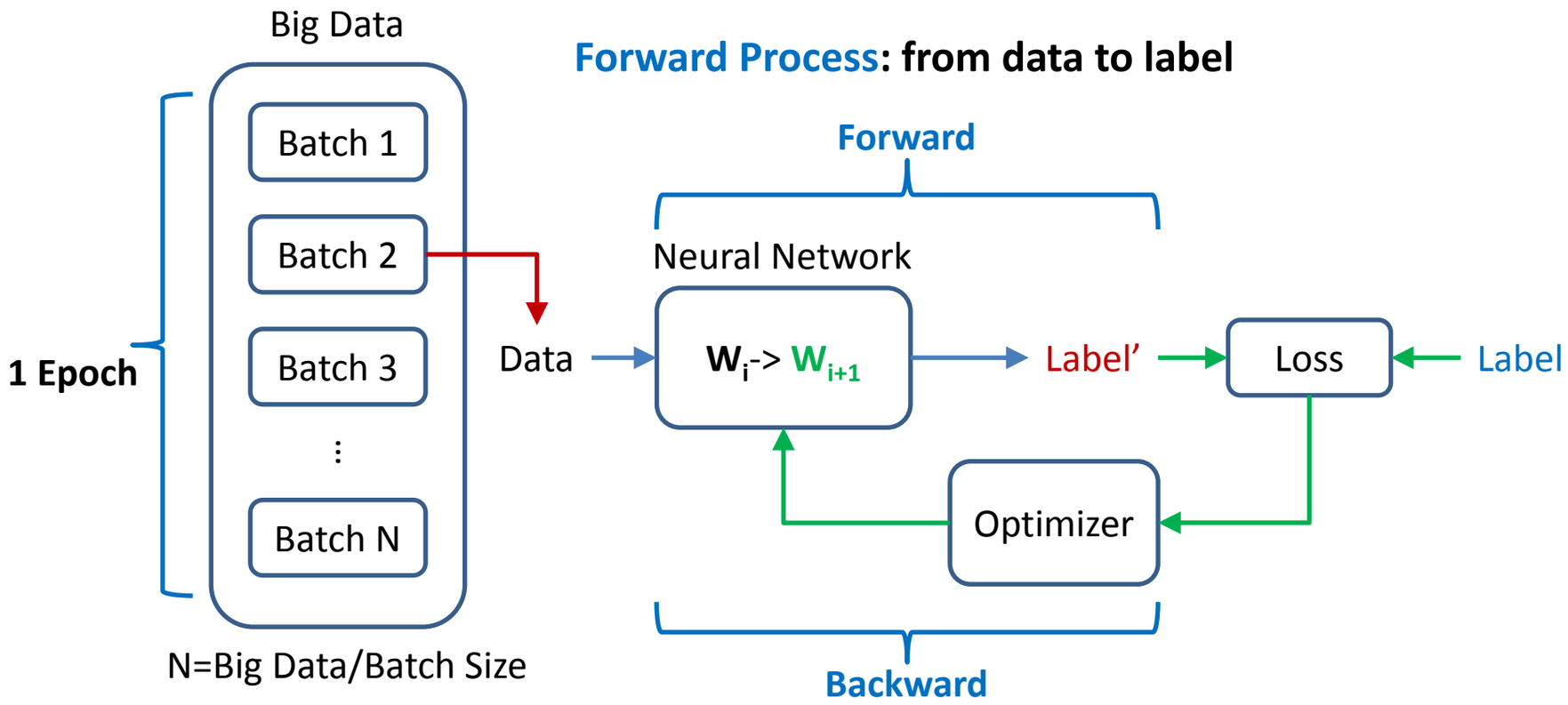
Neural Network in Brief



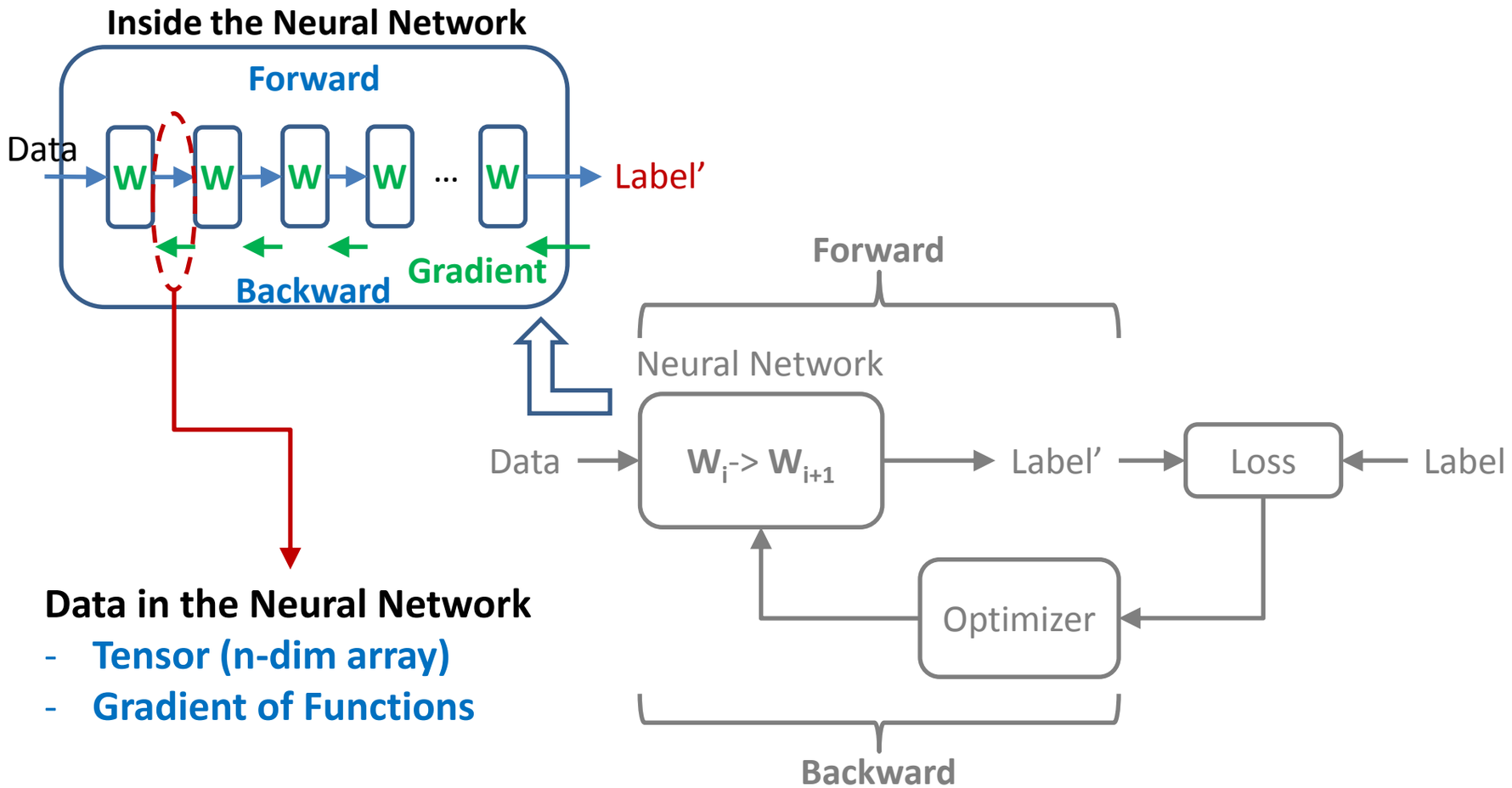
Neural Network in Brief



Neural Network in Brief



Neural Network in Brief

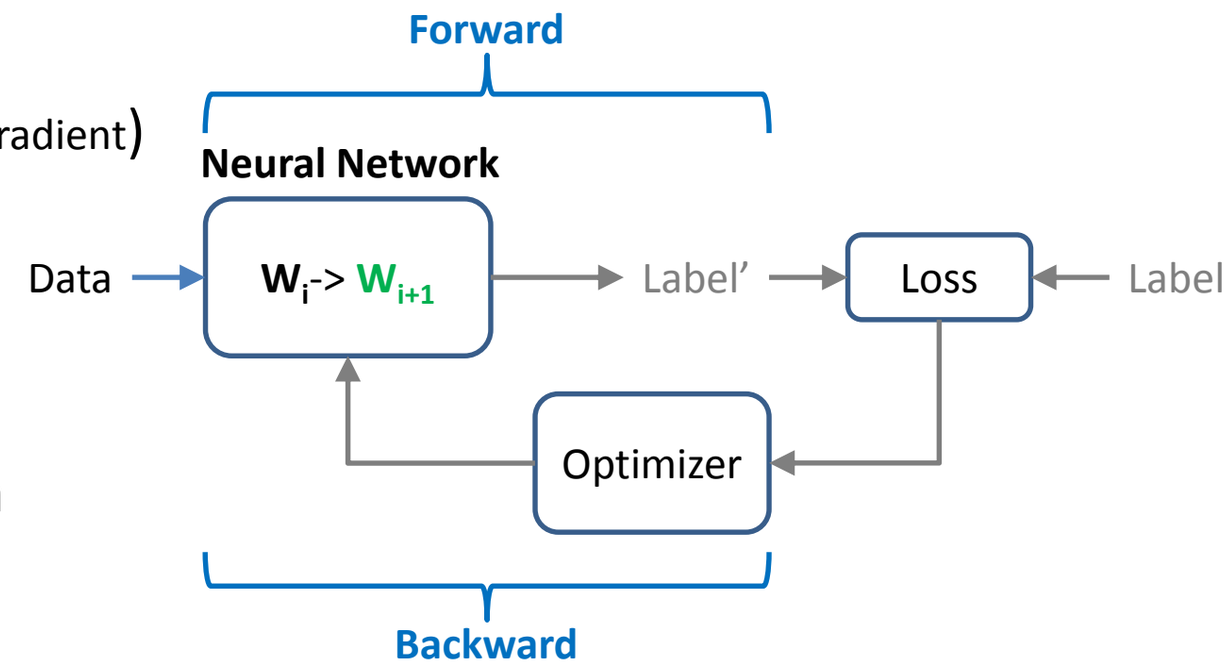


Concepts of PyTorch

- Modules of PyTorch

Data:
- Tensor
- Variable (for Gradient)

Function:
- NN Modules
- Optimizer
- Loss Function



Concepts of PyTorch

- Modules of PyTorch

Data:

- **Tensor**
- Variable (for Gradient)

Function:

- NN Modules
- Optimizer
- Loss Function

- Similar to Numpy

```
x = torch.Tensor(5, 3)
print(x)
```

Out:

```
1.00000e-36 *
  0.0228  0.0000  1.3490
  0.0000  0.0958  0.0000
  0.0958  0.0000  0.0958
  0.0000  0.0958  0.0000
  0.0958  0.0000  0.0958
[torch.FloatTensor of size 5x3]
```

```
x = torch.rand(5, 3)
print(x)
```

Out:

```
0.2285  0.2843  0.1978
 0.0092  0.8238  0.2703
 0.1266  0.9613  0.2472
 0.0918  0.2827  0.9803
 0.9237  0.1946  0.0104
[torch.FloatTensor of size 5x3]
```

Concepts of PyTorch

- Modules of PyTorch

Data:

- **Tensor**
- Variable (for Gradient)

Function:

- NN Modules
- Optimizer
- Loss Function

- Operations
  - z=x+y
  - torch.add(x,y, out=z)
  - y.add_(x) # in-place

Concepts of PyTorch

- Modules of PyTorch

Data:
- **Tensor**
- Variable (for Gradient)

Function:
- NN Modules
- Optimizer
- Loss Function

- Numpy Bridge

- To Numpy
  - a = torch.ones(5)
  - b = a.numpy()

- To Tensor
  - a = numpy.ones(5)
  - b = torch.from_numpy(a)

Concepts of PyTorch

- Modules of PyTorch

Data:
- **Tensor**
- Variable (for Gradient)

Function:
- NN Modules
- Optimizer
- Loss Function

- CUDA Tensors

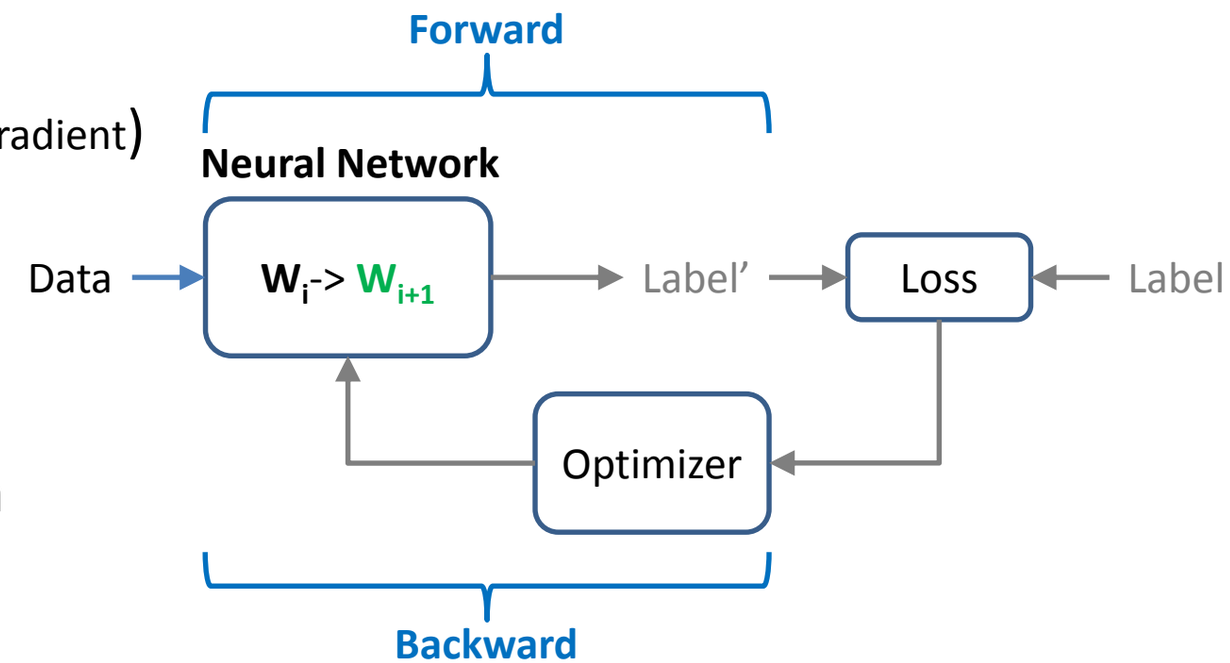
- Move to GPU
  - x = x.cuda()
  - y = y.cuda()
  - x+y

Concepts of PyTorch

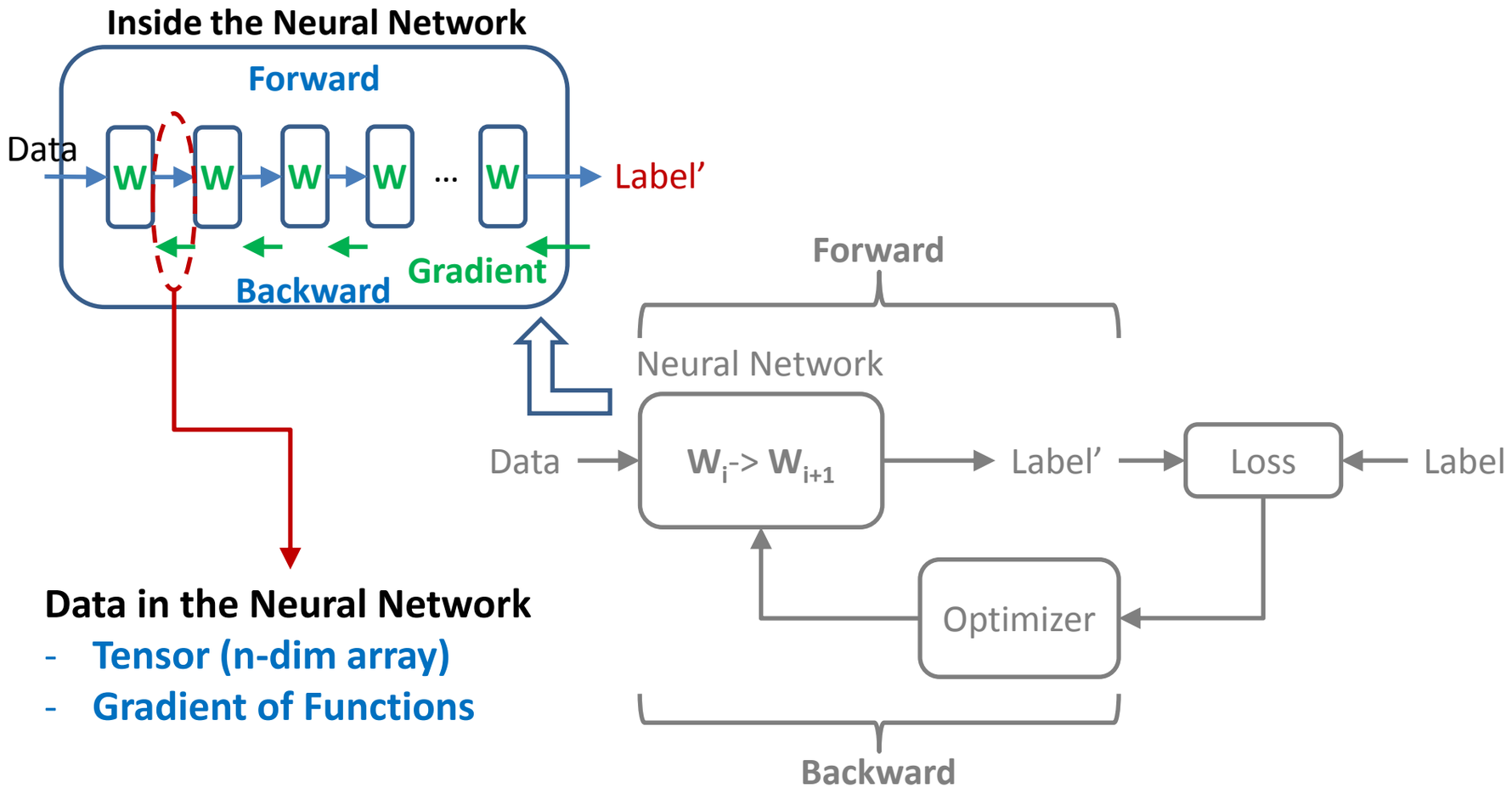
- Modules of PyTorch

Data:
- Tensor
- Variable (for Gradient)

Function:
- NN Modules
- Optimizer
- Loss Function



Neural Network in Brief



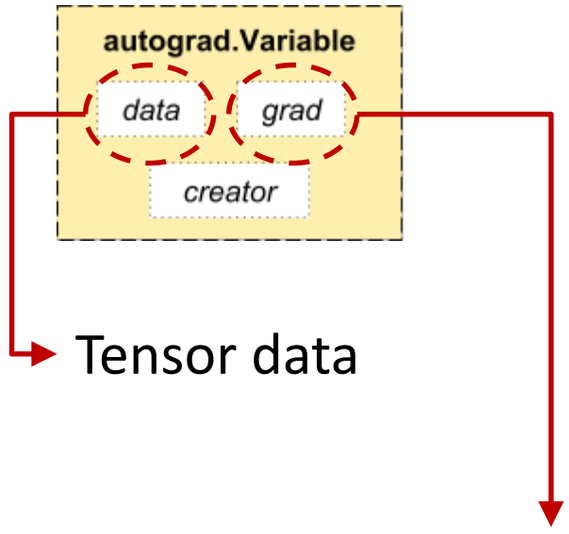
Concepts of PyTorch

- Modules of PyTorch

Data:
- Tensor
- **Variable (for Gradient)**

Function:
- NN Modules
- Optimizer
- Loss Function

- Variable



Concepts of PyTorch

- Modules of PyTorch

Data:
- Tensor
- **Variable (for Gradient)**

Function:
- NN Modules
- Optimizer
- Loss Function

- Variable
- x = Variable(torch.ones(2, 2), requires_grad=True)
- print(x)

Out:
```
Variable containing:
 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

- y = x + 2
- z = y * y * 3
- out = z.mean()
- out.backward()
- print(x.grad)

Out:
```
Variable containing:
 4.5000  4.5000
 4.5000  4.5000
[torch.FloatTensor of size 2x2]
```

$$\text{out} = \frac{1}{4}\sum z_i$$

$$z_i = 3y_i^2 = 3(x_i + 2)^2$$

$$\frac{\partial \text{out}}{\partial x_i} = \frac{3}{2}(x_i + 2) = \frac{9}{2}$$

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

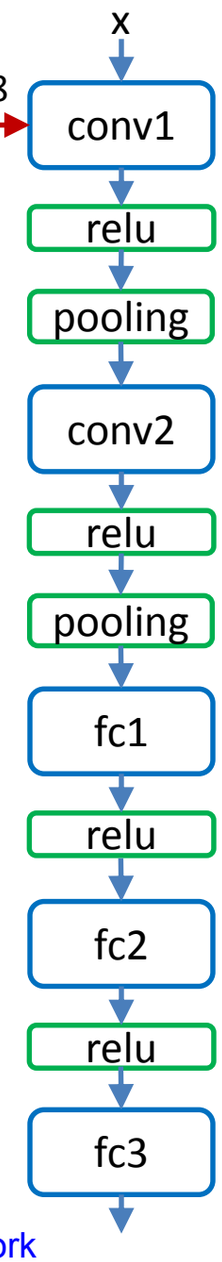
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 1x32x32->6x28x28

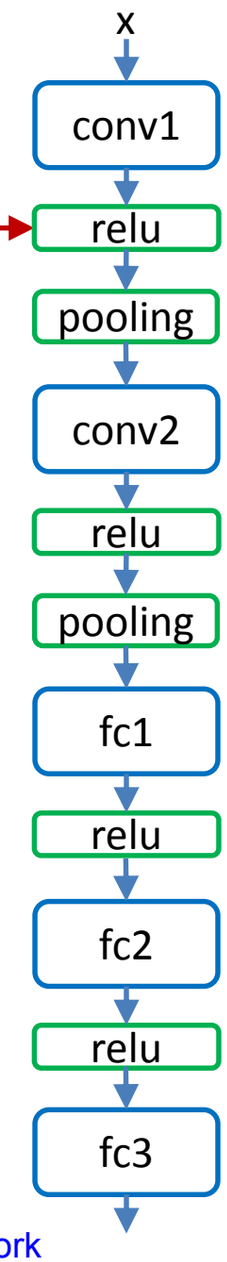


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules
(must have)

Build network
(must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```



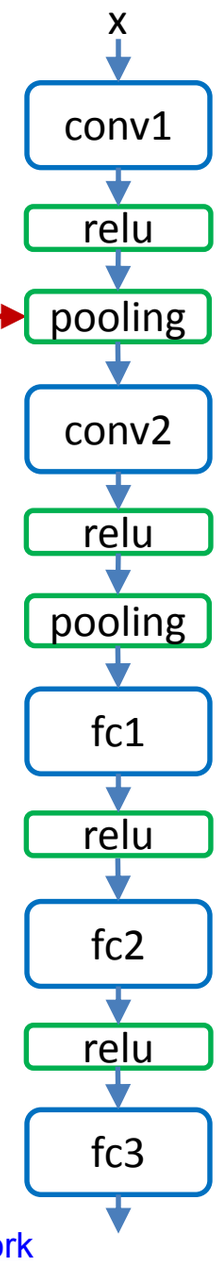
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 6x28x28 -> 6x14x14



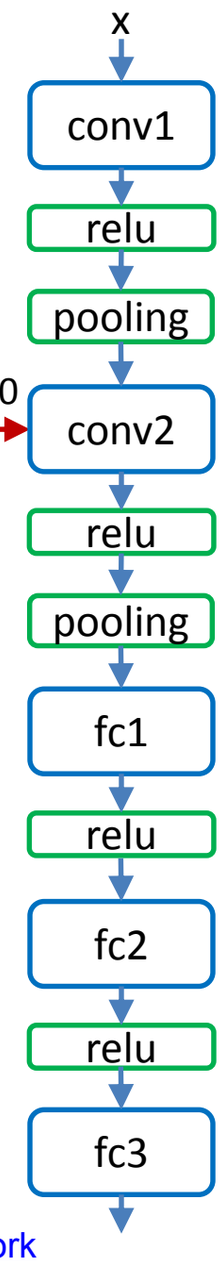
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 6x14x14 -> 16x10x10



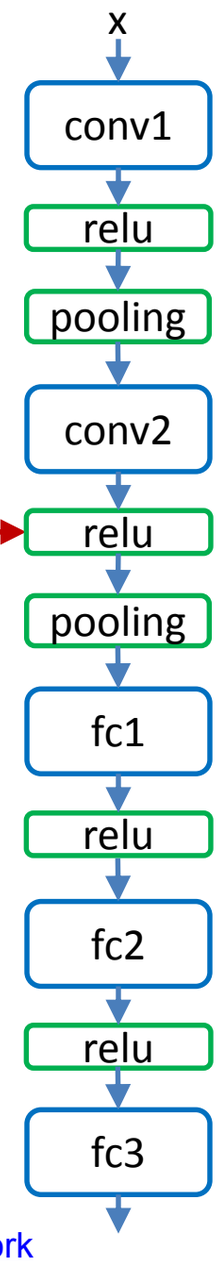
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 16x10x10



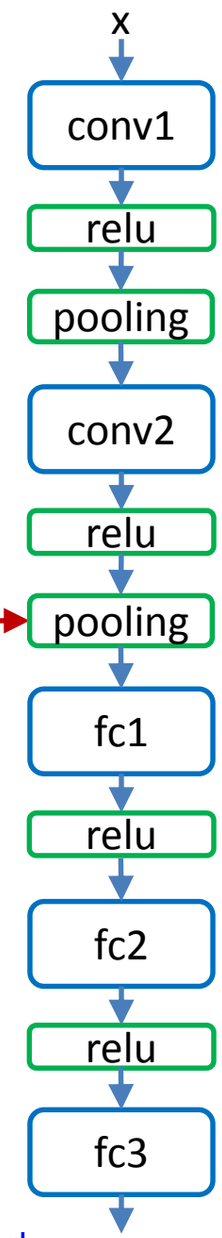
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 16x10x10 -> 16x5x5



- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

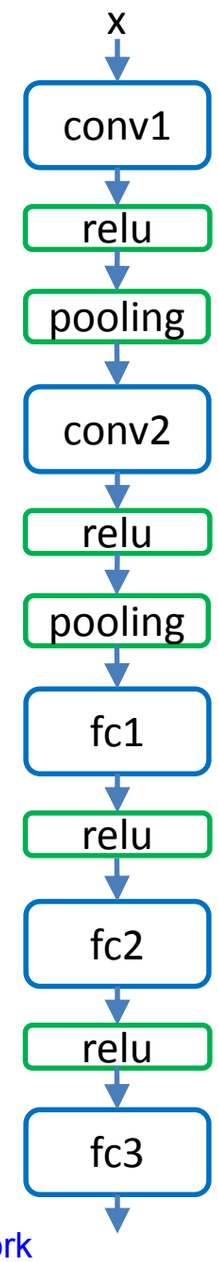
Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Flatten the Tensor
16x5x5

Tensor: [Batch N, Channel, H, W]

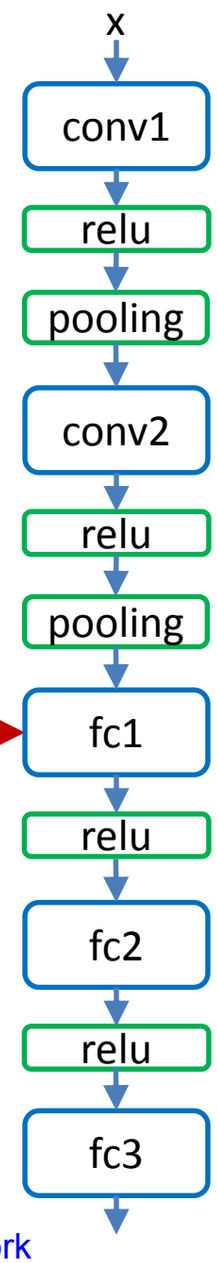


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

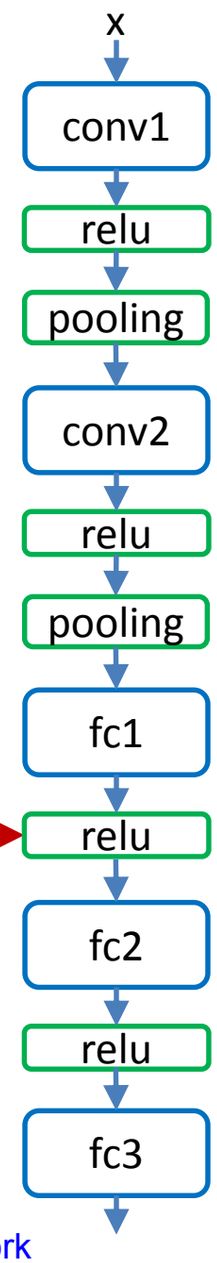


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules
(must have)

Build network
(must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

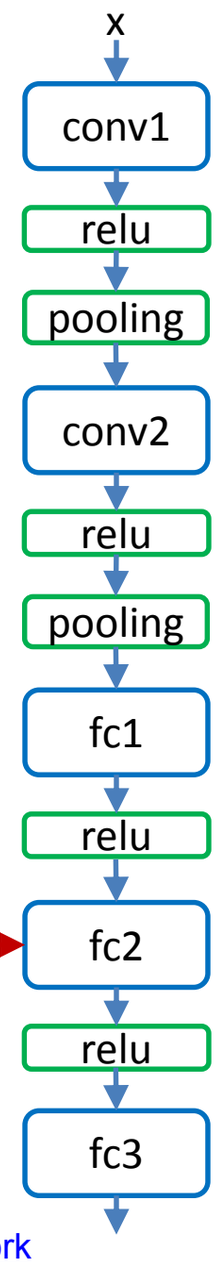


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

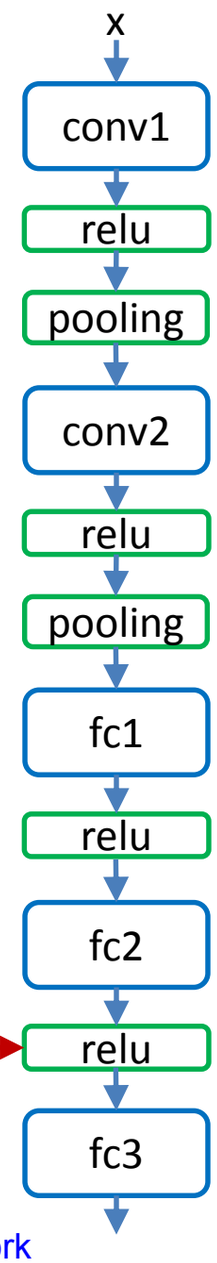


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

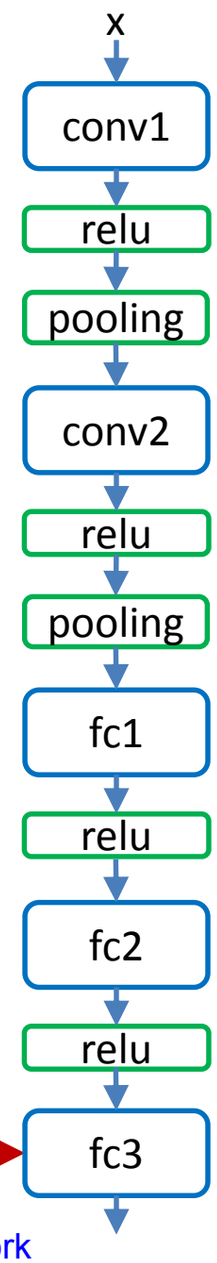


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```



- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - **NN Modules**
  - Optimizer
  - Loss Function

- NN Modules (torch.nn)
  - Modules built on Variable
  - Gradient handled by PyTorch
  - Common Modules
    - Convolution layers
    - Linear layers
    - Pooling layers
    - Dropout layers
    - Etc…

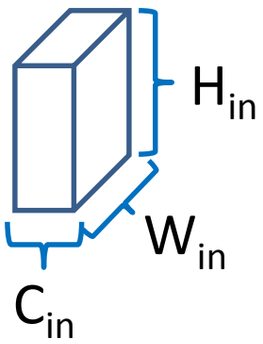
NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D
  - Example:
  - torch.nn.conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

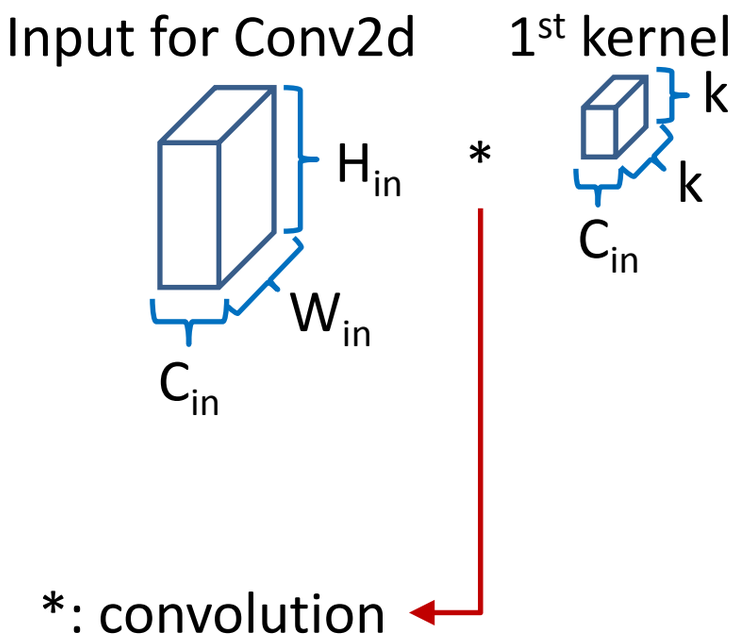
Input for Conv2d



*: convolution

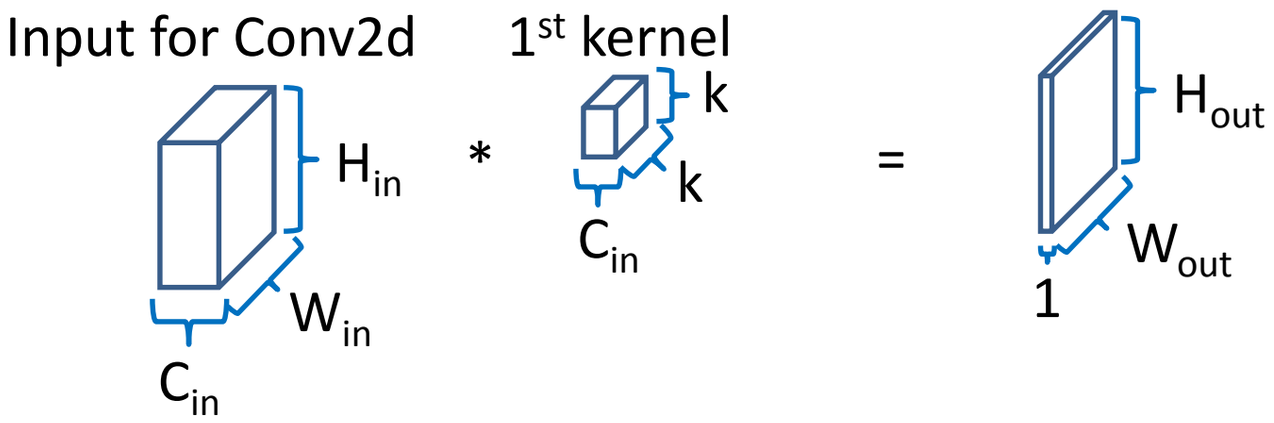
NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D



NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

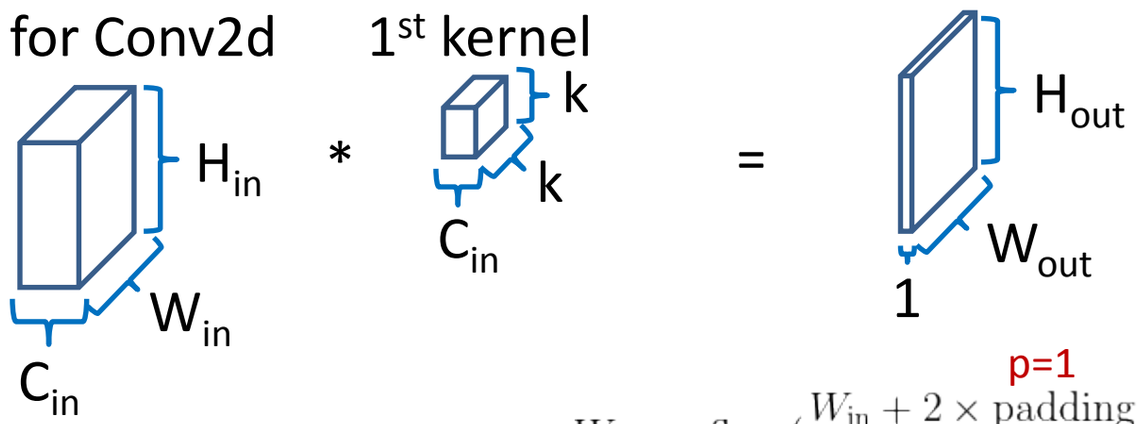


*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

Input for Conv2d 1st kernel



*: convolution

p=1 d=1 k=3

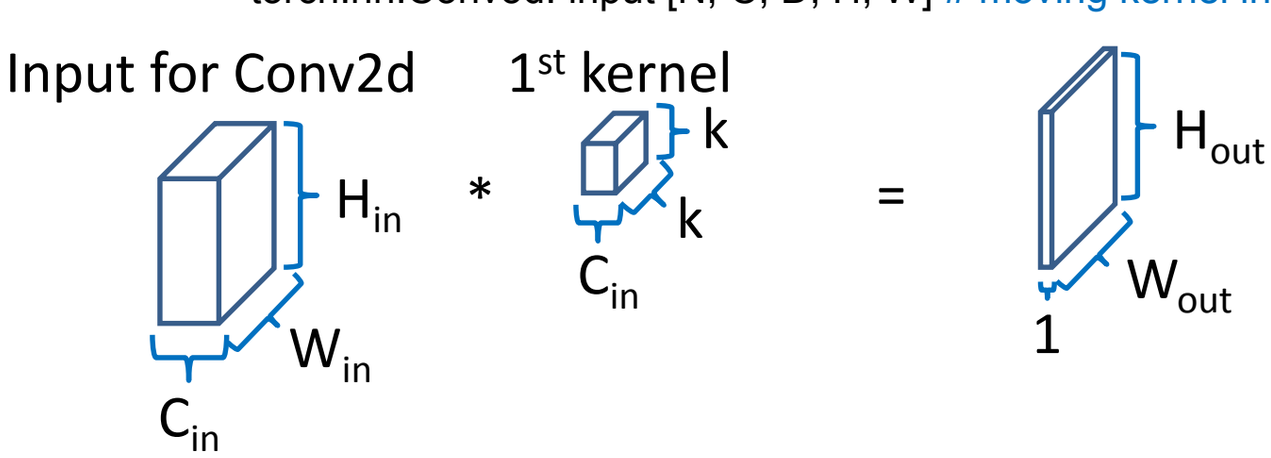
$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times \text{padding} - \text{dilation} \times (k-1) - 1}{\text{stride}} + 1\right)$$

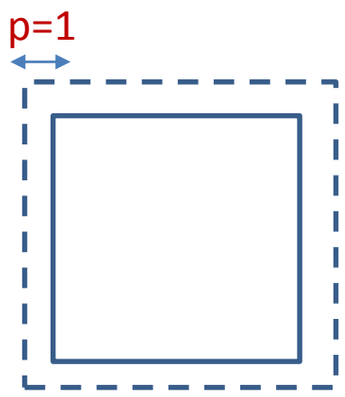
s=1, moving step size

$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times 1 - 1 \times (3-1) - 1}{1} + 1\right) = W_{\text{in}}$$

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

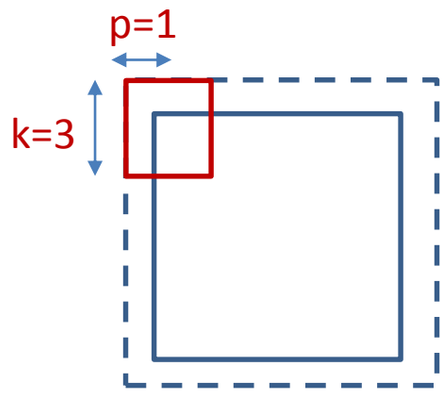


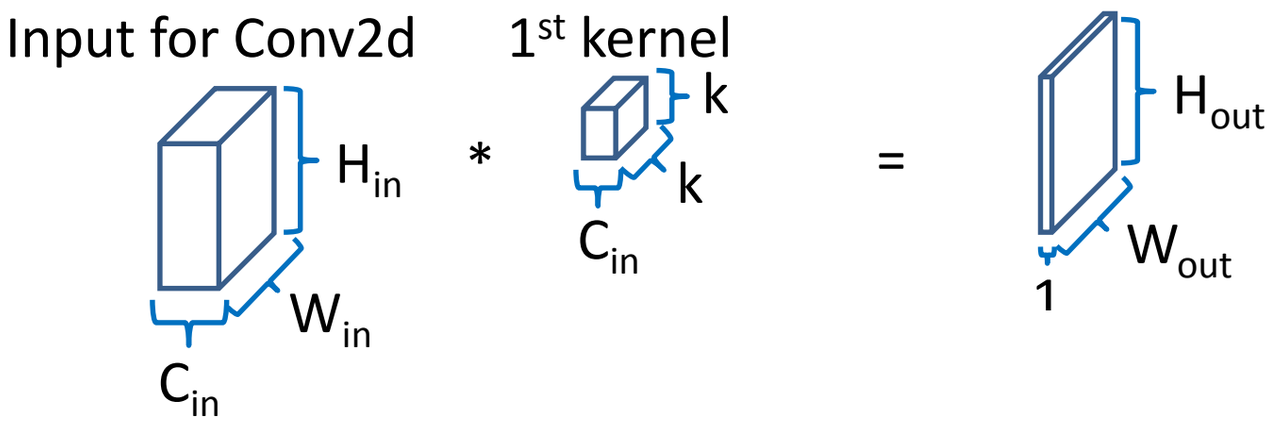


*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

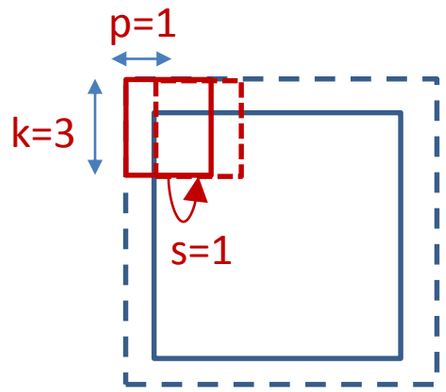


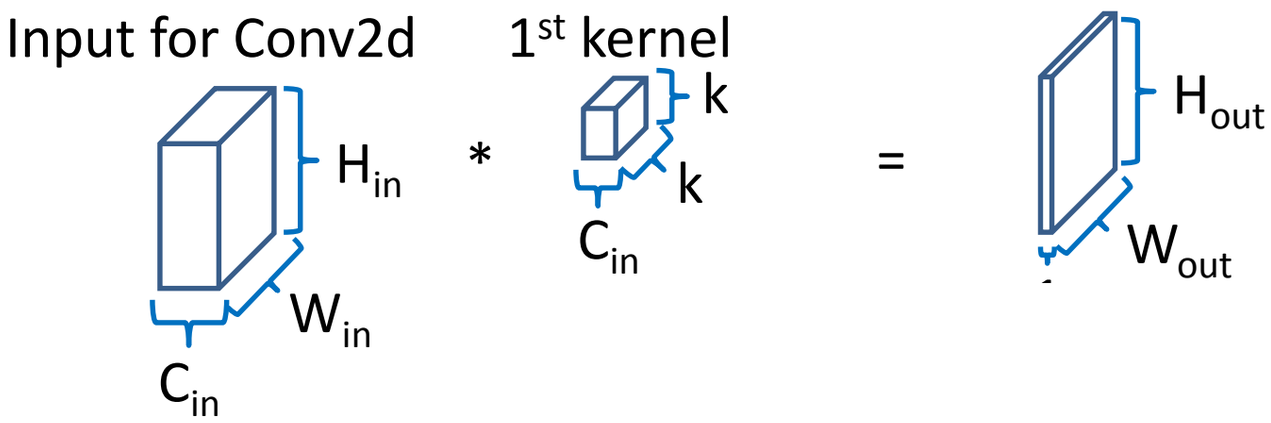


*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

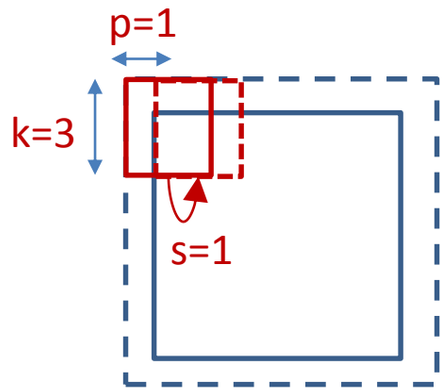


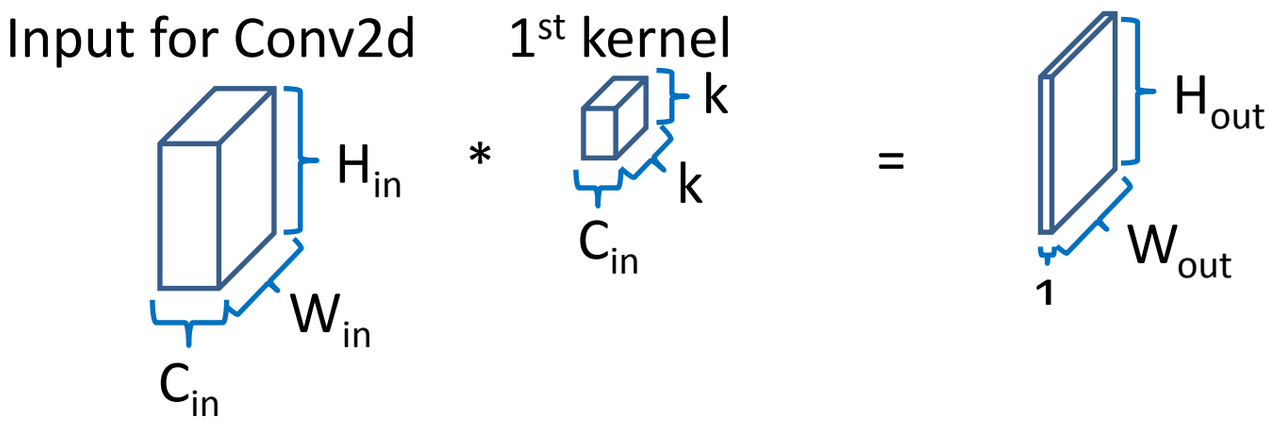


*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

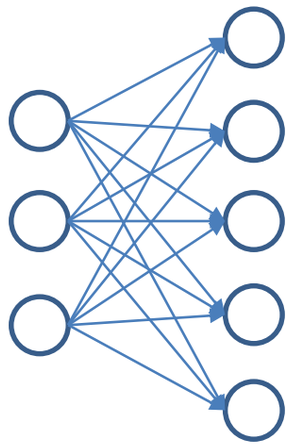




*: convolution

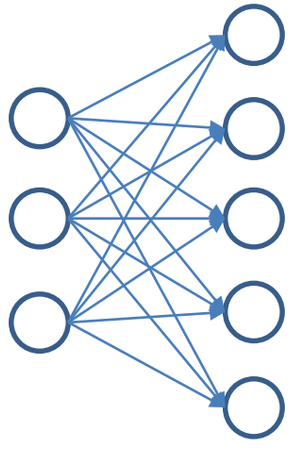
NN Modules

- Linear Layer
  - torch.nn.Linear(in_features=3, out_features=5)
  - y=Ax+b



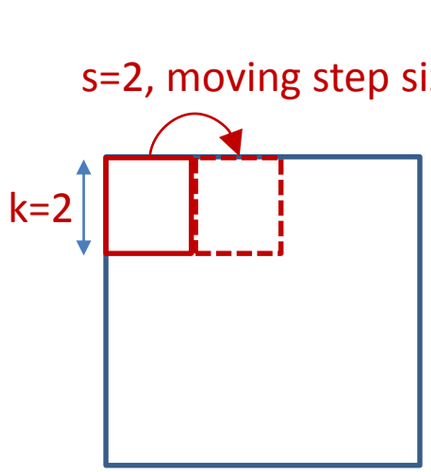
NN Modules

- Linear Layer
  - torch.nn.Linear(in_features=3, out_features=5)
  - y=Ax+b



NN Modules

- Pooling Layer
  - torch.nn.AvgPool2d(kernel_size=2, stride=2, padding=0)
  - torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)



Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - **NN Modules**
  - Optimizer
  - Loss Function

- NN Modules (torch.nn)
  - Modules built on Variable
  - Gradient handled by PyTorch
- Common Modules
  - Convolution layers
  - Linear layers
  - Pooling layers
  - Dropout layers
  - Etc…

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - NN Modules
  - **Optimizer**
  - **Loss Function**

- Optimizer (torch.optim)
  - SGD
  - Adagrad
  - Adam
  - RMSprop
  - …
  - 9 Optimizers (PyTorch 0.2)

- Loss (torch.nn)
  - L1Loss
  - MSELoss
  - CrossEntropy
  - …
  - 18 Loss Functions (PyTorch 0.2)

What We Build?

Define modules (must have)

Build network (must have)

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# 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, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
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 = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

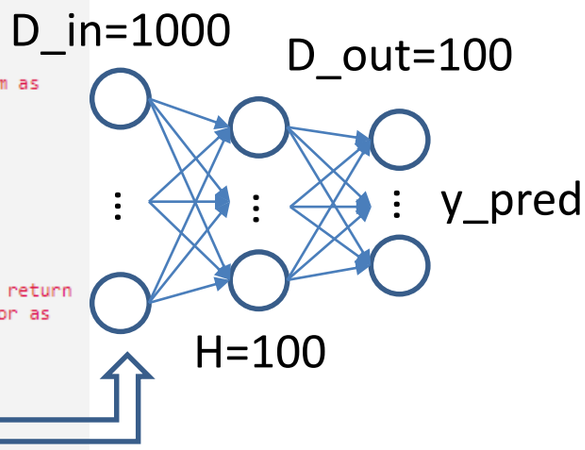
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

Define modules (must have)

Build network (must have)

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# 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, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
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 = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

What We Build?



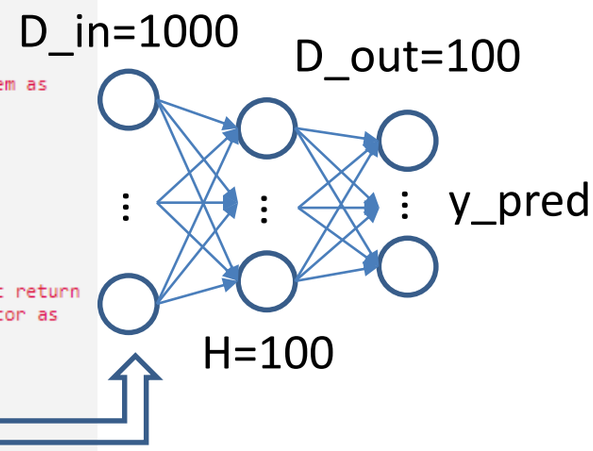
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

Define modules (must have)

Build network (must have)

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# 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, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
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 = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

What We Build?



Don’t Update y (y are labels here)

Construct Our Model

Optimizer and Loss Function

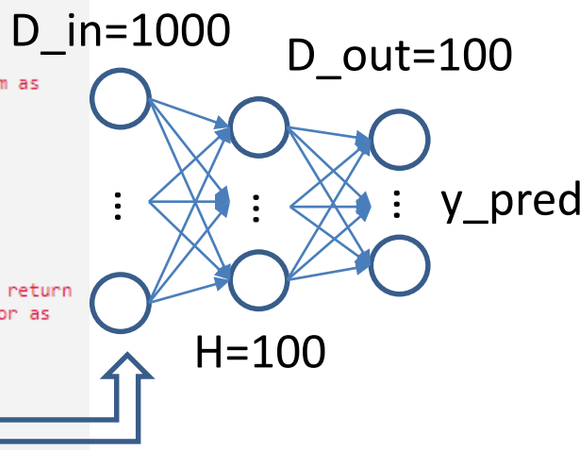
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

Define modules (must have)

Build network (must have)

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# 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, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
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 = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

What We Build?



Don’t Update y (y are labels here)

Construct Our Model

Optimizer and Loss Function

Reset Gradient
Backward
Update Step

http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

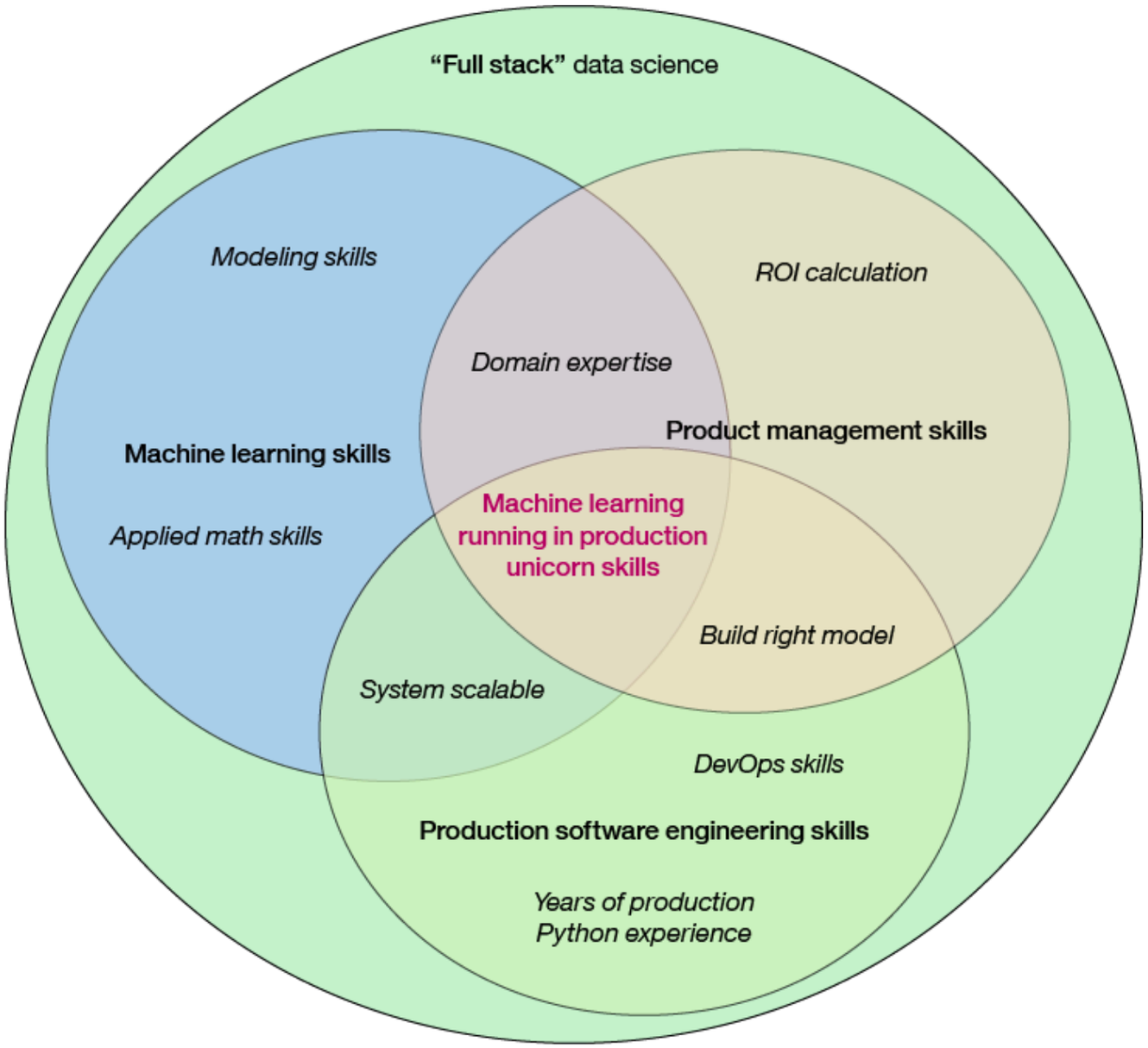
Full Stack - Data Science

Full Stack Data Scienc

From Data to Deployment

Data Scientist + Software Engineer + DevOps = FS Data Scientist

Full Stack Data Science



Full Stack Data Science

Full-stack data scientists

1

Prevent handoff mistakes

2

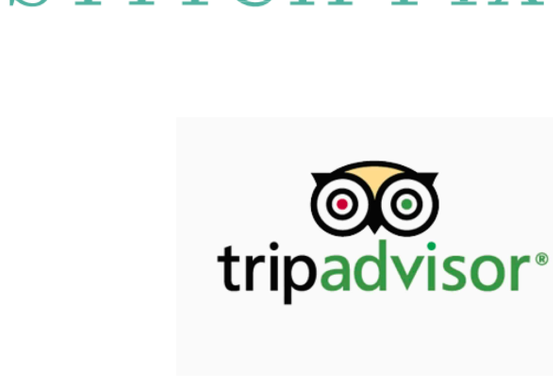
Can contribute on any team

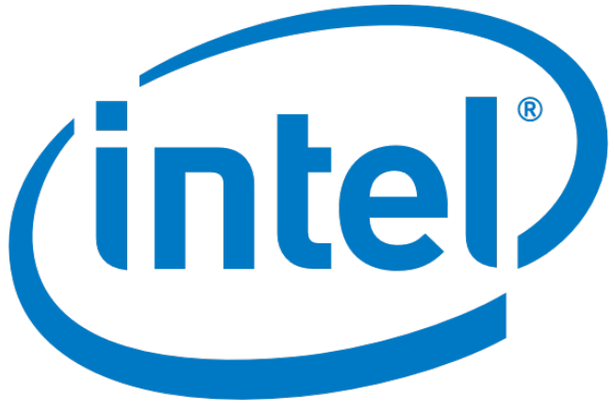
3

Have big picture in mind

But, Why?

Full Stack Data Science





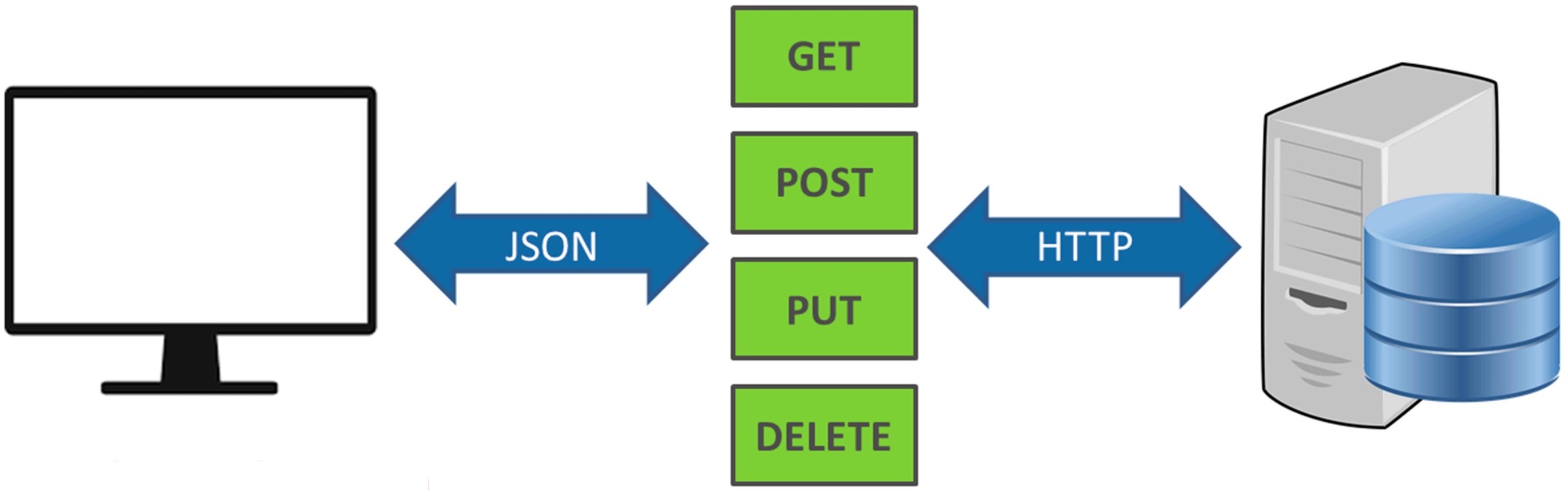
Full Stack Data Science

Cool, right?

Full Stack Data Science

Going Full Stack

Engineering of science

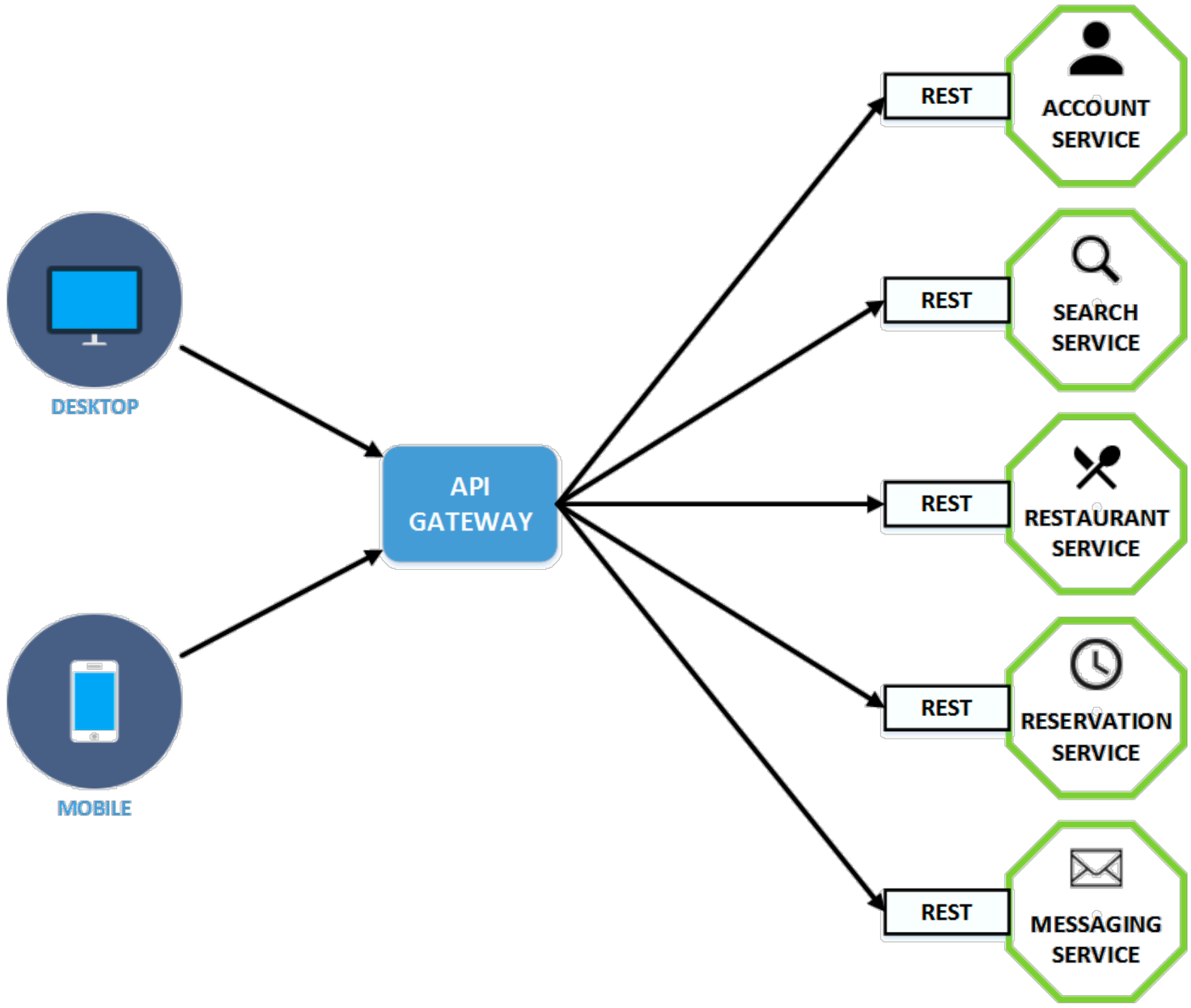


Client sends a **request**

HTTP methods

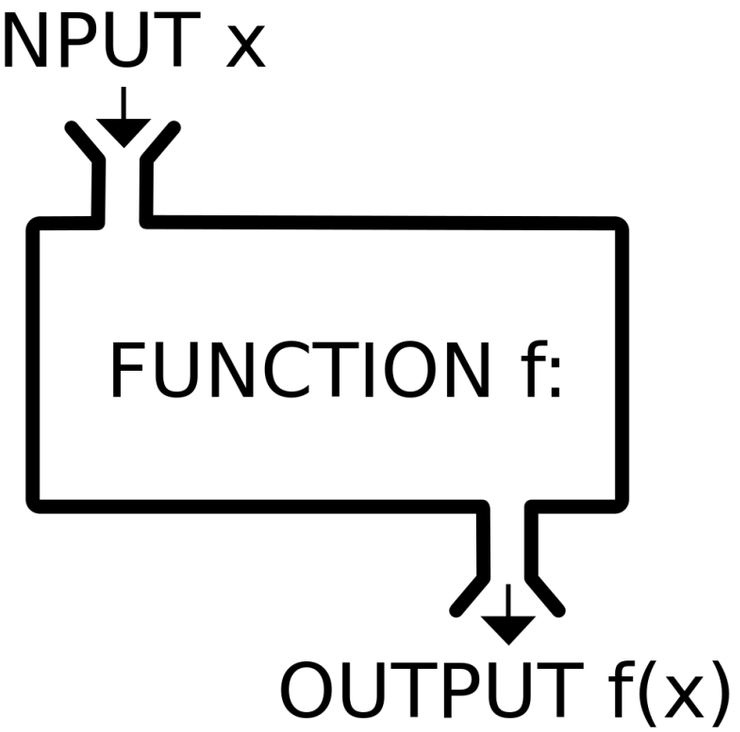
Server sends a **request**

Engineering of science



Engineering of science

We can consider an API as a project which acts as a function



Engineering of science

In production, Deep learning Models are wrapped in an RESTful API

So other teams and products can make use of it.

Full stack

Let's build something!

Faster R-CNN Architecture

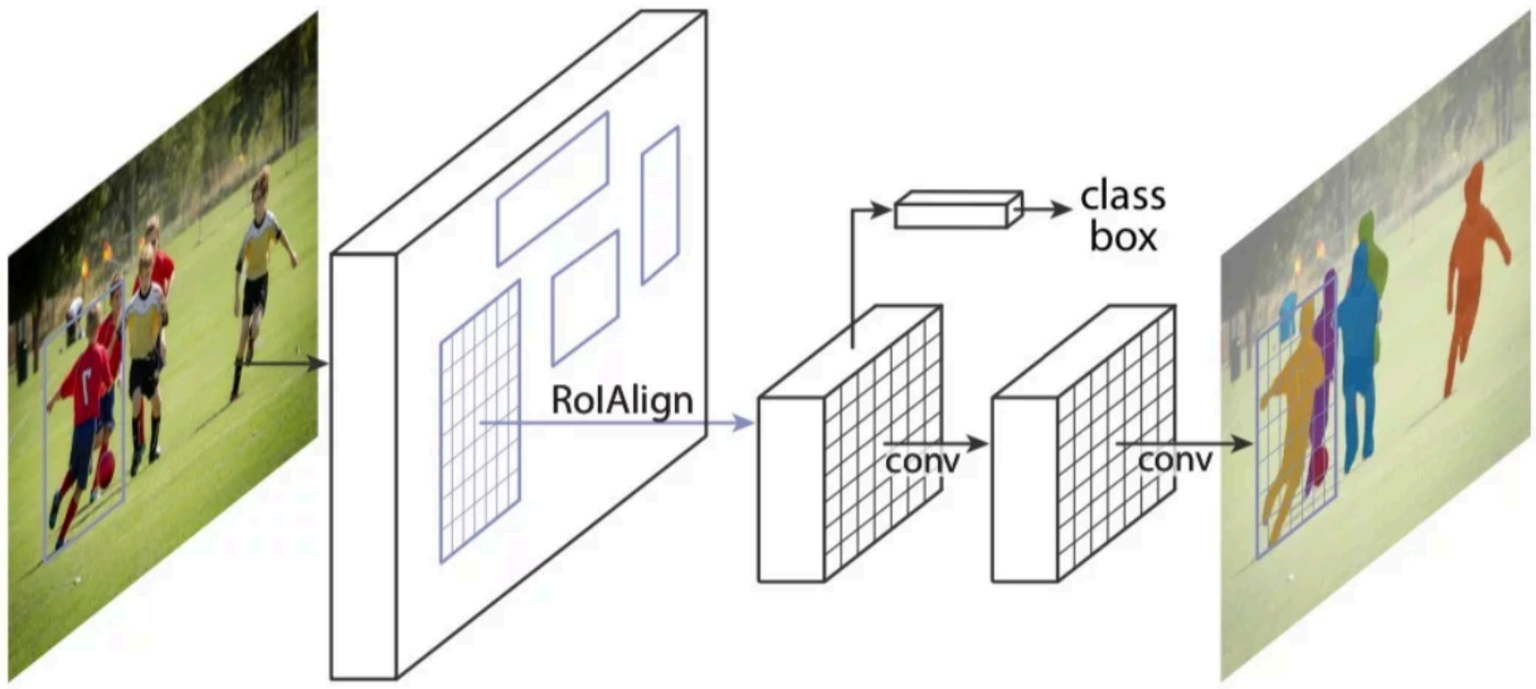
1. **Convolutional Layers**: The input image is passed through several convolutional layers to create a feature map. Think of the convolutional layers as a black box that takes in a 3-channel input image, and outputs an “image” with a much smaller spatial dimension (7×7), but a large number of channels (512).
2. **Region Proposal Network (RPN).** The output of the convolutional layers is used to train a network that proposes regions that enclose objects.
3. **Classifier:** The same feature map is also used to train a classifier that assigns a label to the object inside the box

Mask R-CNN Architecture

Mask R-CNN takes the idea one step further. In addition to feeding the feature map to the RPN and the classifier, it uses it to predict a binary mask for the object inside the bounding box.

One way of looking at the mask prediction part of Mask R-CNN is that it is a **Fully Convolutional Network (FCN)** used for semantic segmentation. The only difference is that the FCN is applied to bounding boxes, and it shares the convolutional layer with the RPN and the classifier.

Mask R-CNN Architecture



That’s It?

Full Stack Data science

What we build is the first iteration of the app.

Formally called an MVP.

The next iteration is Caching, Logging, Dockerizing, etc.

You now know the way :)

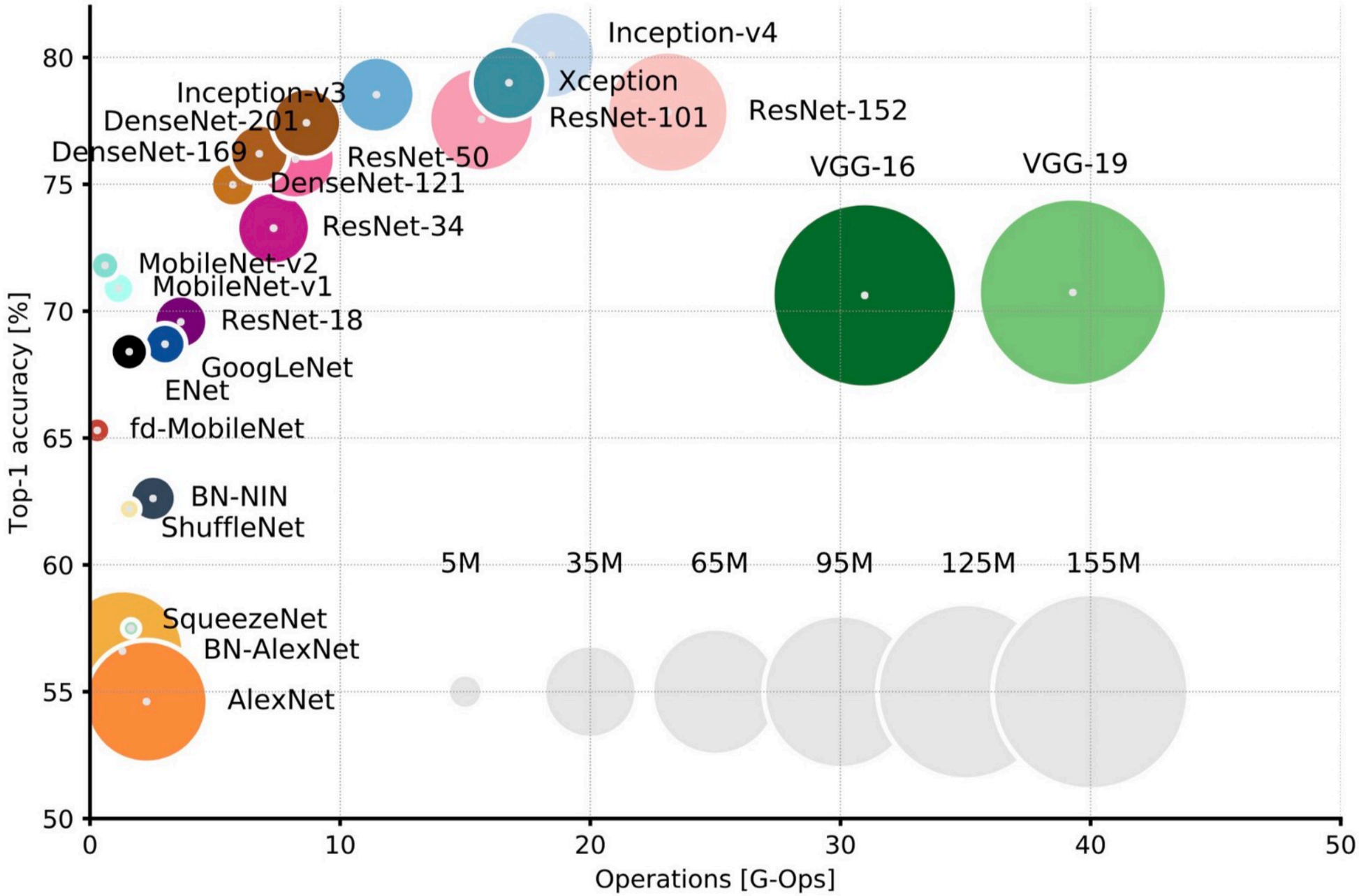
State of the art

SOTA

Usually, Data scientist doesn’t build their models from scratch.

Because that’s what’s deep learning about. ***Generalization***

SOTA


Inception-v4
Xception
ResNet-101
ResNet-152
Inception-v3
DenseNet-201
DenseNet-169
ResNet-50
DenseNet-121
VGG-16
VGG-19
ResNet-34
MobileNet-v2
MobileNet-v1
ResNet-18
GoogLeNet
ENet
fd-MobileNet
BN-NIN
ShuffleNet
SqueezeNet
BN-AlexNet
AlexNet
5M
35M
65M
95M
125M
155M
80
75
70
65
60
55
50
Top-1 accuracy [%]
0
10
20
30
40
50
Operations [G-Ops]

Questions ?