ML and PyTorch Basics

Wei Zhang, Ulrich Finkler

CSCI-GA.3033-022 HPML

Piazza

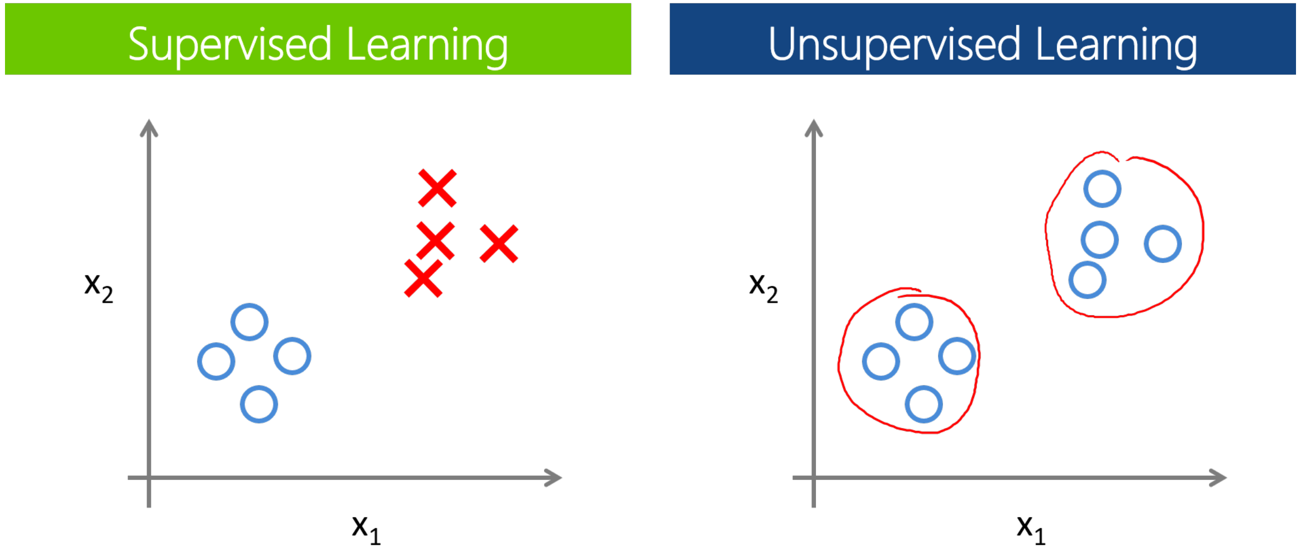
piazza.com/nyu/fall2019/csciga3033020

Summary

- Machine Learning definition
- Linear and Logistic Regression
- Feed forward, Loss function and Backpropagation
- Inference and Training
- PyTorch basics
  - tensors
  - graph
  - NN
  - training

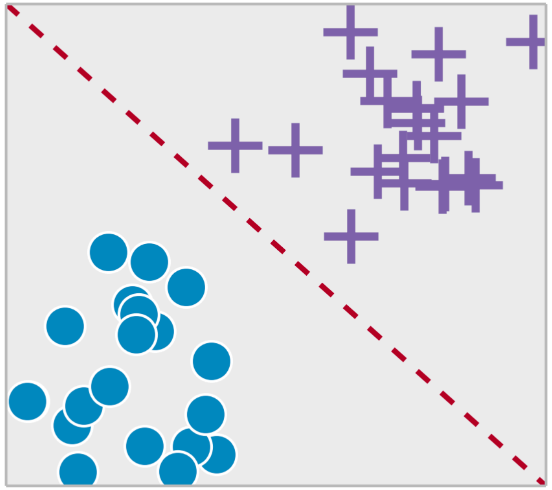
Machine Learning

- **Machine Learning:** algorithms that learn from the data to build a predictive analytical model
  - **Supervised Learning**: A labeled dataset (correct input and output) is available, so you can give it to the model to train it
  - **Unsupervised Learning**: A labeled dataset is not available (only input), the model has to learn from the data without knowing in advance what is the expected output

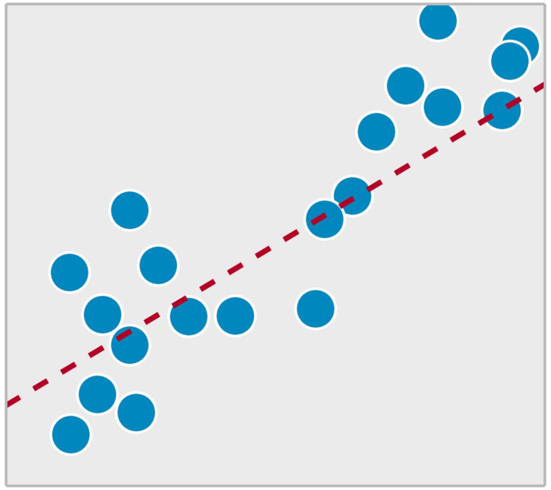


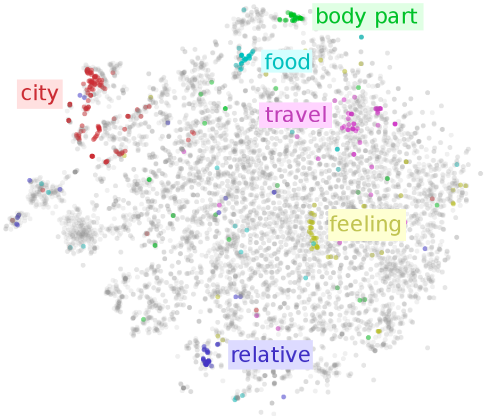
Supervised vs Unsupervised Learning Examples

Classification

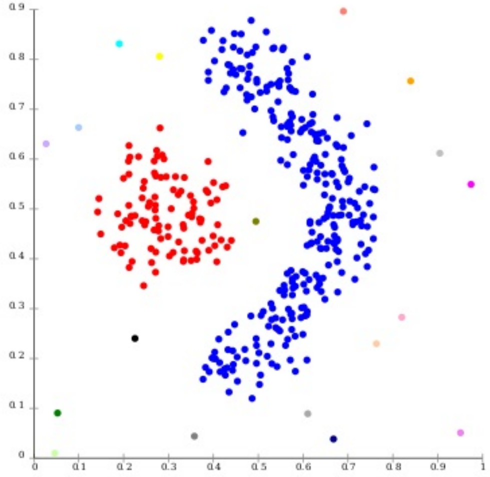


Regression



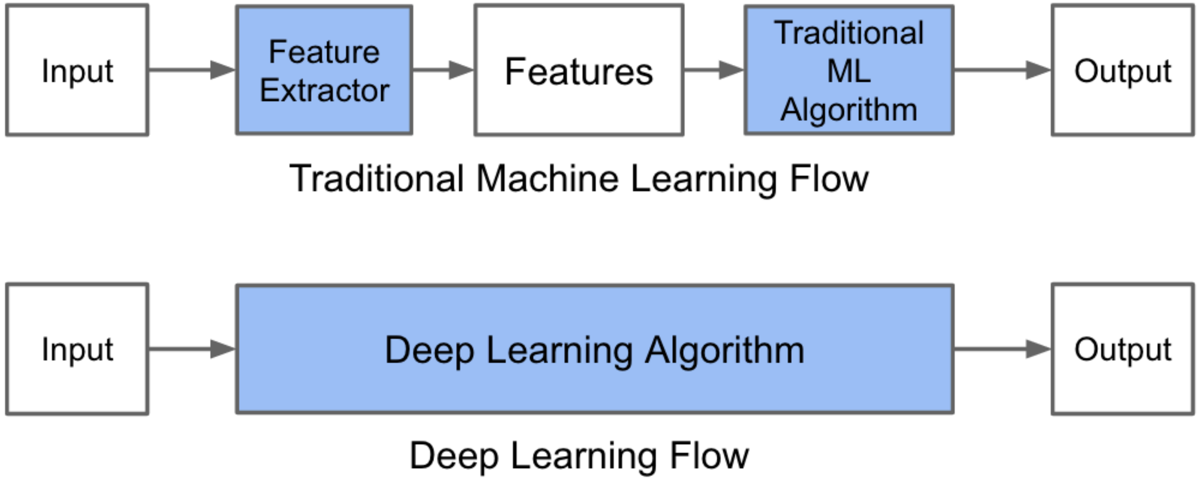


Word embeddings



Clustering

Traditional Machine Learning vs. Deep Learning



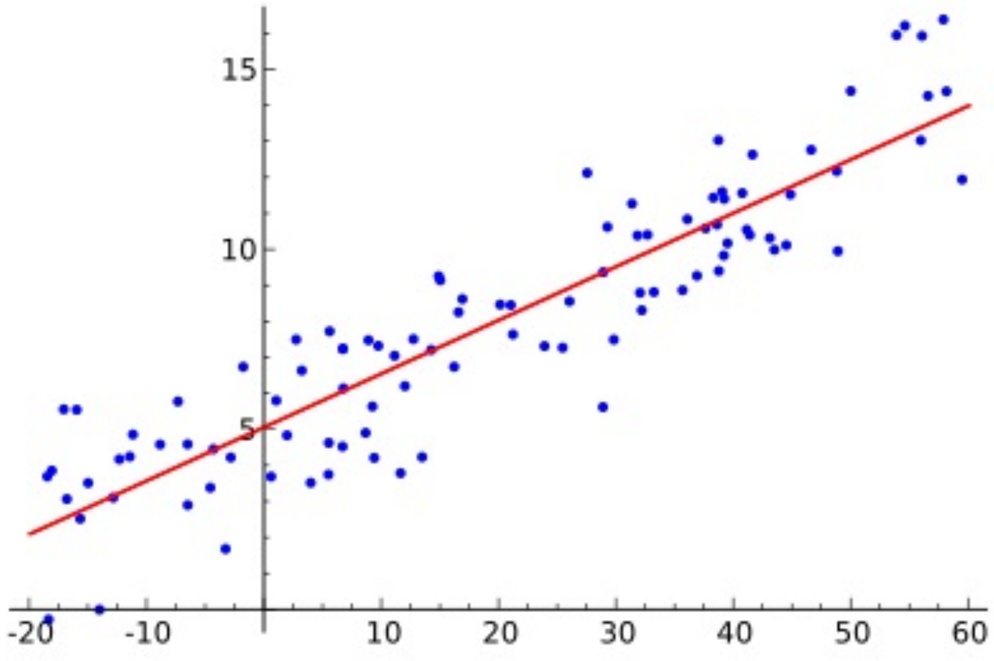
- In traditional Machine Learning an explicit feature extraction phase is needed
  - Feature engineering is difficult, time-consuming and requires domain expertise
- In Deep Learning its (typically) done by the algorithm

Linear Regression

- **Linear regression** finds the best-fitting straight line (regression line)
  - The distance between the points to the regression line represent the errors
- Given training data $\{(x_i , y_i): 1 \leq i \leq n\}$ i.i.d. from distribution D
- Find W, X and b such that:

$$\hat{Y} = \mathrm{f(x)} = W^T X + b$$

  - $\hat{Y}$ is the predicted value
  - W^T is the weights vector
  - X is the features vector
  - b is the bias



Loss Function

- **Loss function:** maps output values (predictions) to losses
  - Minimize loss function => Optimizing parameters for fitting
- Examples of **loss functions** for the linear regression:
  - **L1-norm:** It is basically minimizing the sum of the absolute differences **(S)** between the target value (**y_i**) and the estimated values **$f(x_i)$**:

$$L = \sum_{i=1}^{n} |y_i - f(x_i)|$$

  - **L2-norm:** It is basically minimizing the sum of the square of the differences **(S)** between the target value (**y_i**) and the estimated values **$f(x_i)$:**

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

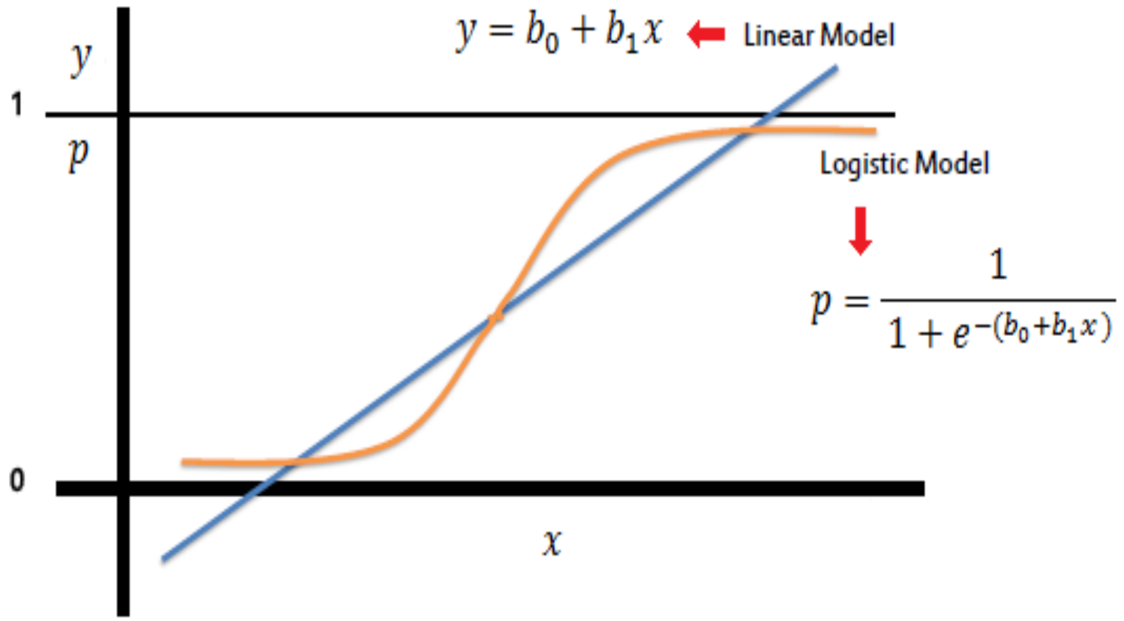
Logistic regression

- **Logistic Regression** is used to model the probability of a binary event: output is in [0,1]:

$$\hat{Y} = \mathrm{f}(z) = \frac{1}{1+e^{-z}}$$

$$z = W^T X + b$$

  - $\hat{Y}$ is the binary event probability (predicted value)
  - z is the scalar output of the linear combination
  - W^T is the vector of weights
  - X is the vector of inputs (features)
  - b is the bias scalar
- Linear regression output can assume all values while Logistic regression only [0,1]

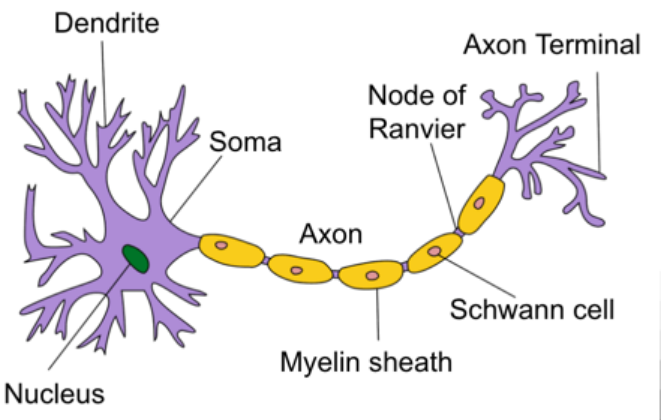


Artificial Neuron

- Artificial Neuron:

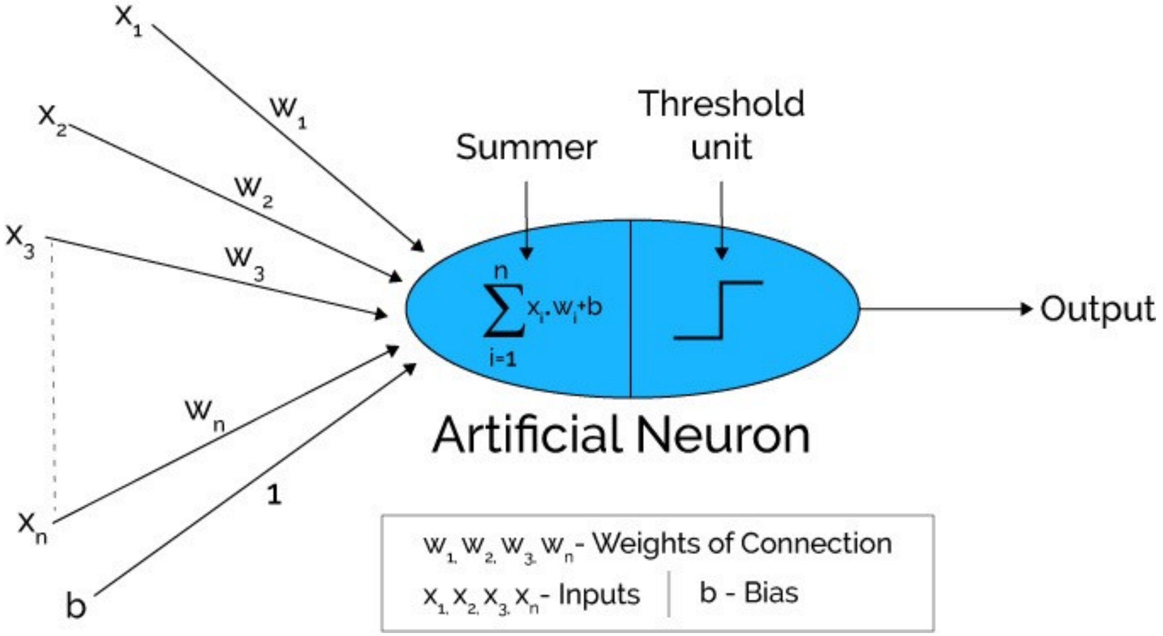
$$Z = W^T X + b$$

$$\hat{Y} = \mathrm{A(Z)}$$



Structure of a typical neuron
(source: Wikipedia)

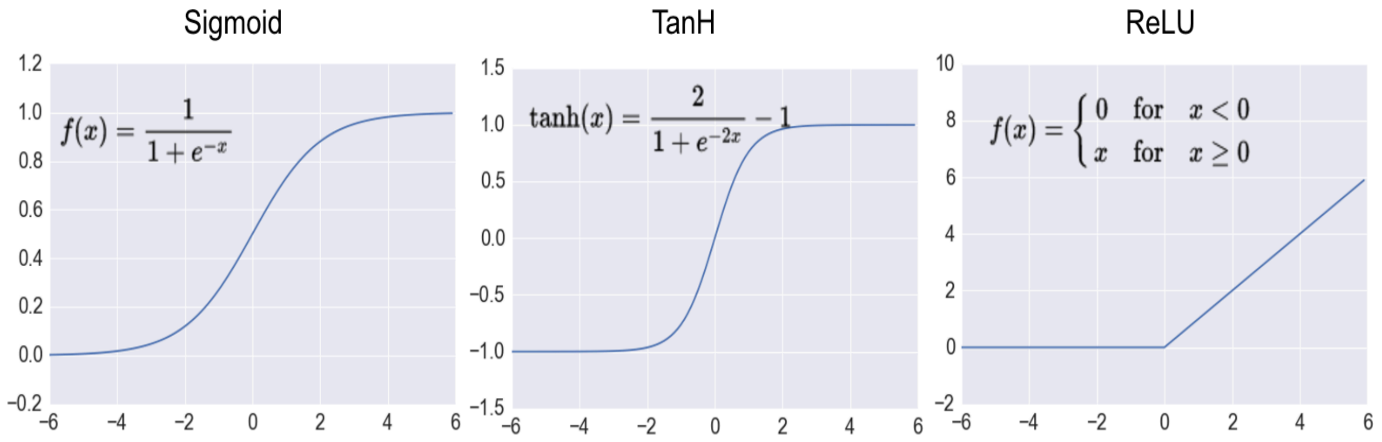
- W^T : weights vector
- X : input features vector:
  - **1 sample has multiple features**
- $\hat{Y}$: prediction scalar
- b: bias scalar
- $A(Z)$: activation function (threshold scalar)



From: https://medium.com/@xenonstack/overview-of-artificial-neural-networks-and-its-applications-2525c1addff7

Neuron’s Activation Functions

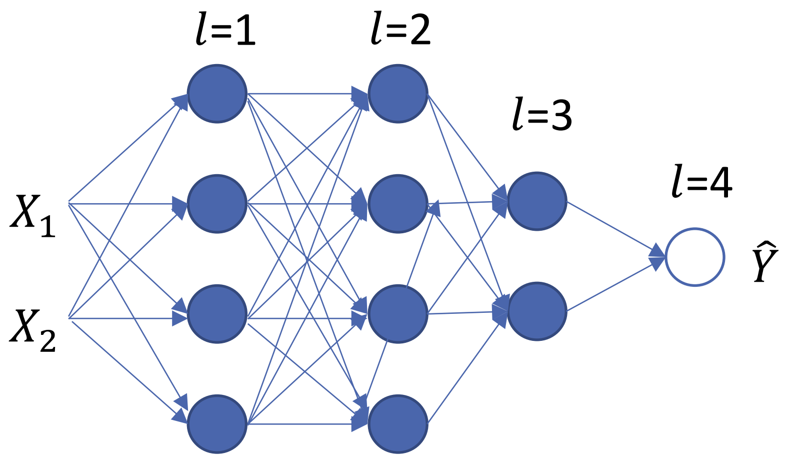
- Threshold (Activation) Functions:
  - Sigmoid: $A(Z) = \frac{1}{1+e^{-z}}$
    - Used for binary classification (0,1)
  - Tanh: $A(Z) = \frac{2}{1+e^{-2z}}$ -1
    - Used for generic classification (-1,+1)
  - RELU: $A(Z) = \begin{cases} if\ Z < 0\ then\ 0 \\ if\ Z \geq 0\ then\ Z \end{cases}$
    - Faster than Sigmoid or Tanh



From: http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/

Neural Networks

- Able to model non-linear functions
- Each neuron computes its value based on linear combination of values of neurons that point into it
- Can add more layers of hidden units: deeper hidden unit response depends on earlier hidden layers

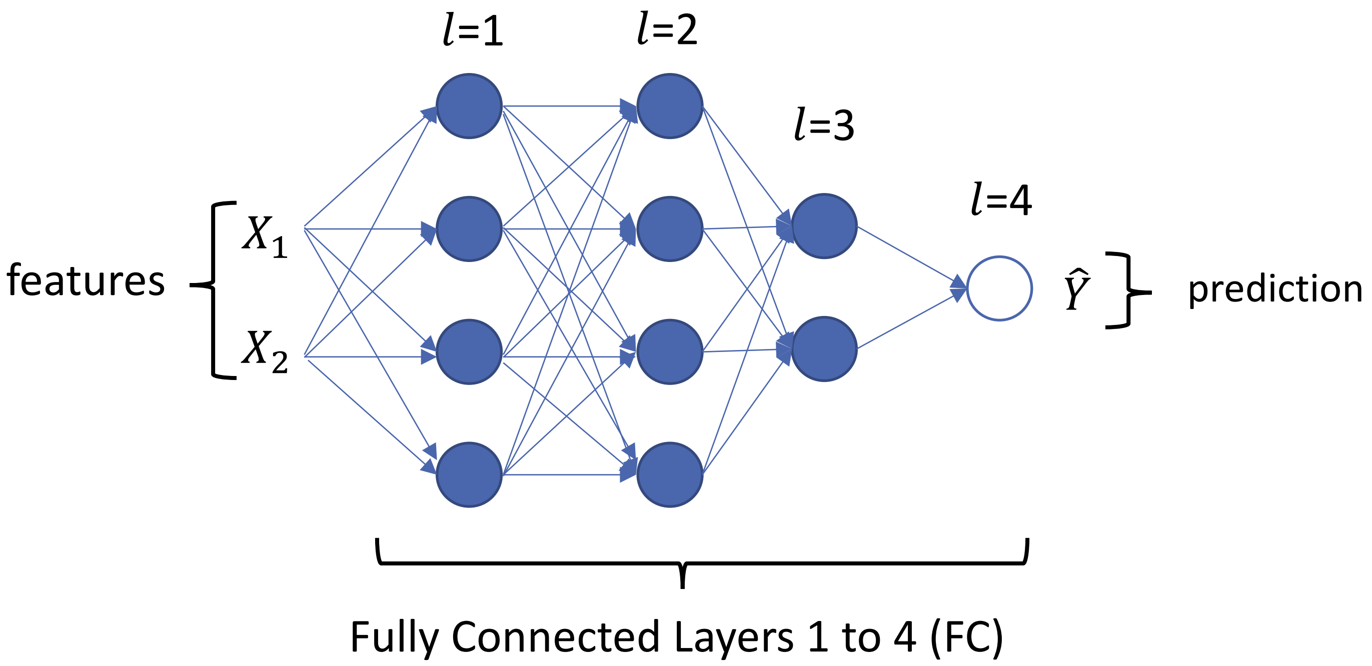


Neural Networks Lifecycle

1. **Definition phase:**
    - Define Number, Structure and Type of Layers
    - Define other algorithmic and structural parameters (Hyperparameters)
2. **Training phase:** discover neuron’s weights and biases
3. **Inference phase:** use model to make predictions (or classify)

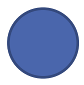
Neural Network Definition

- Feed-Forward and Fully Connected NN:

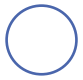


- Each output of the layer l is connected to all inputs of layer $l+1$

- Two types of nodes:



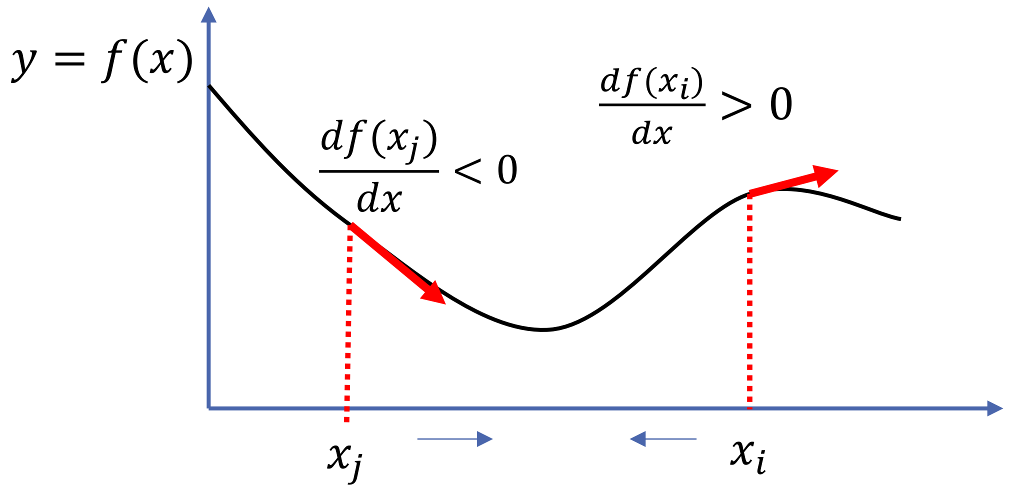
$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = RELU(Z^{[l]})$$



$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
$$\hat{Y} = Sigmoid(Z^{[l]})$$

Training - Gradient Descent

- Find x_k such that $y_k = f(x_k)$ is a (local) minima
  - f is defined and differentiable
- Gradient Descent:
  - Start with random x_o
  - Repeat:
    - x_{n+1} <- $x_n - \alpha \frac{df(x_n)}{dx}$
    - Until you don't see any improvement



- α is the **Learning Rate:** determines how fast we descend the curve
  - α is usually very small: 0.01 or less

Training - Gradient Descent

- Used to minimize the **Cost Function** (loss/error across all samples)
- Gradient descent is that it will more often than not get stuck into the first local minimum that it encounters
  - There is no guarantee that this local minimum it finds is the best (global) one (bottom figure)

Algorithm 1 Gradient Descent

Input: Differentiable function $\mathbf{f(x)}$ where $\mathbf{f(x)} : \mathbf{R^n} \rightarrow \mathbf{R}$

Start point $\mathbf{x_{old}}$

Output: The local minima $\mathbf{x^*}$ that minimize $\mathbf{f(x)}$

while TRUE **do**

  $\mathbf{tmpDelta} \leftarrow \mathbf{x_{old}} - \alpha.(\nabla \mathbf{f}(\mathbf{x_{old}}))$

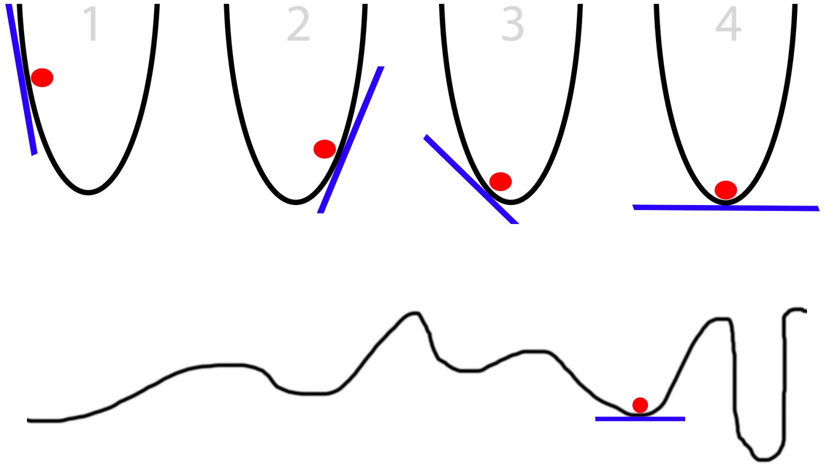
  if $\mathbf{abs(tmpDelta - x_{old}) < CRITERIA}$ **then**

    break

  end if

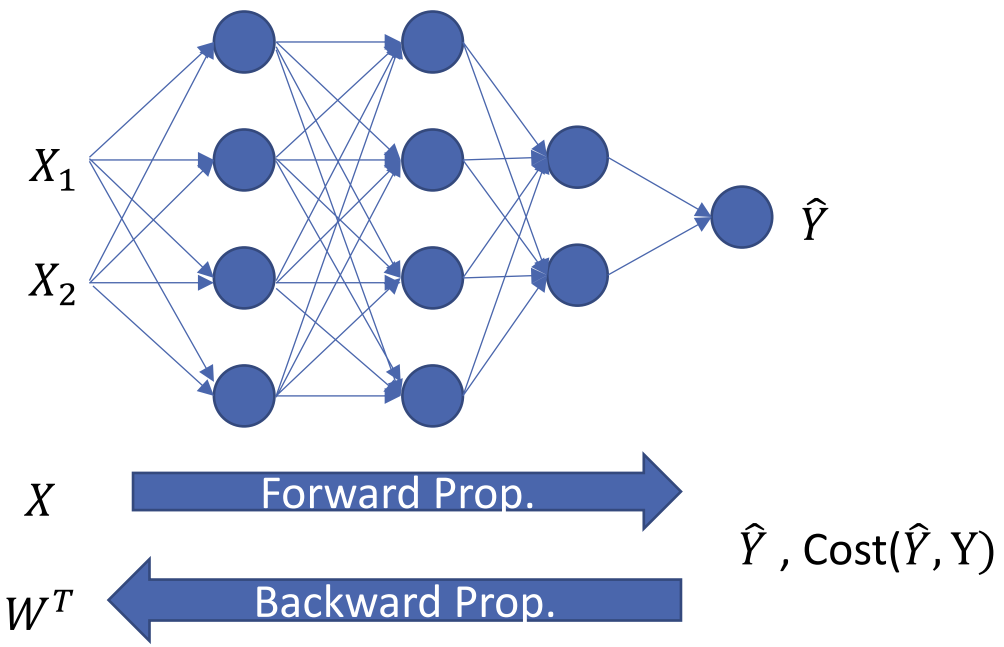
  $\mathbf{x_{old}} \leftarrow \mathbf{tmpDelta}$

end while

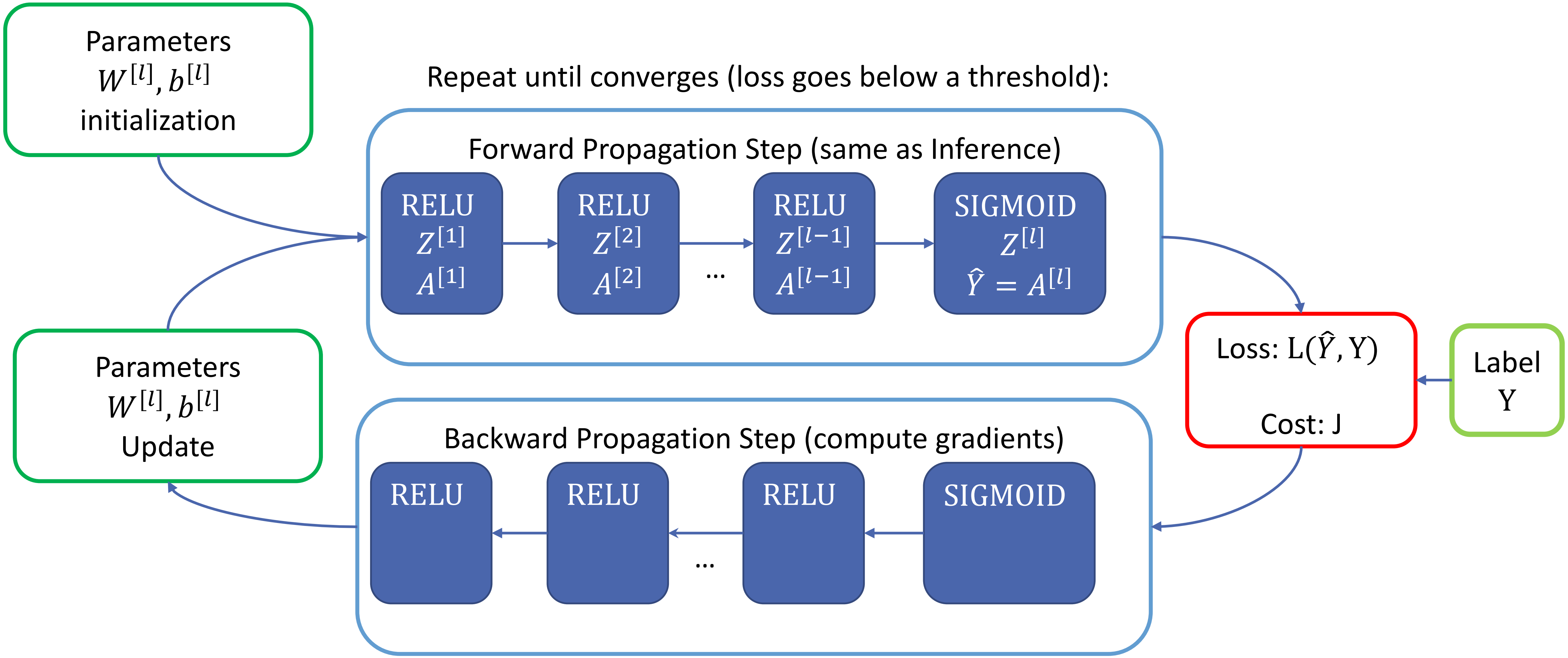


Training - Gradient Descent

- Prerequisite: Network structure is already defined (hyperparameters)
  - Type and number of layers (FC or other types)
  - Number of neurons on each layer
  - Activation functions of each layer
- **Batch Gradient Descent: use all samples**
- Gradient Discover all weights and biases:
  - While (COST < threshold)
    - **Forward Propagation**:
      - compute prediction
      - Same formulas as Inference
    - **Backward Propagation**
      - compute gradient
      - adjust weights

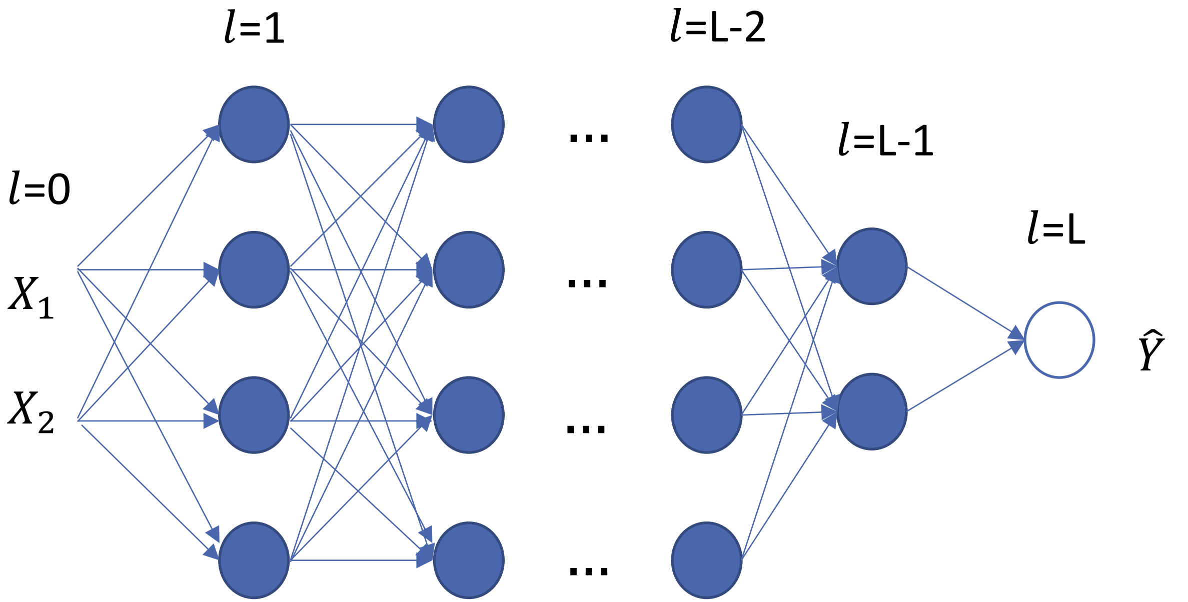


Training - Gradient Descent Algorithm



Forward Propagation (and Inference as well)

- m samples
- L layers
- $n^{[l]}$ is the number of neurons for layer l (where $l \in [0, L]$)
- $n^{[0]}$ is the number of features of each sample (layer 0)
- For each layer l compute:
  - $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$
  - $A^{[l]} = g^{[l]}(Z^{[l]})$
- Where
  - $A^{[0]} = X$
  - $g^{[L]}$ is a Sigmoid function
  - $g^{[l]}$ for $l < L$ is a RELU function
- Matrices shapes
  - $W^{[l]}$: $(n^{[l]}, n^{[l-1]})$
  - $b^{[l]}$: $(n^{[l]}, 1)$
  - $A^{[l]}$: $(n^{[l]}, m)$
  - $Z^{[l]}$: $(n^{[l]}, m)$



Forward Propagation – Cost Function

- Cost function (cross-entropy):

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1 - y^{(i)})\log\left(1 - a^{[L](i)}\right)\right)$$

  - $y^{(i)}$ is the target (label) value from the dataset of the *i*-th sample
  - $a^{[L](i)}$ is the output of the forward propagation (prediction) of the *i*-th sample
  - m is the number of samples used (dataset size)
  - **Cost function: a function of individual samples loss functions (J)**

Backward Propagation – Parameter Updates

- For each layer, we want to update the parameters with the gradients

$$W^{[l]} \Leftarrow W^{[l]} - \alpha \frac{dJ}{dW^{[l]}}$$

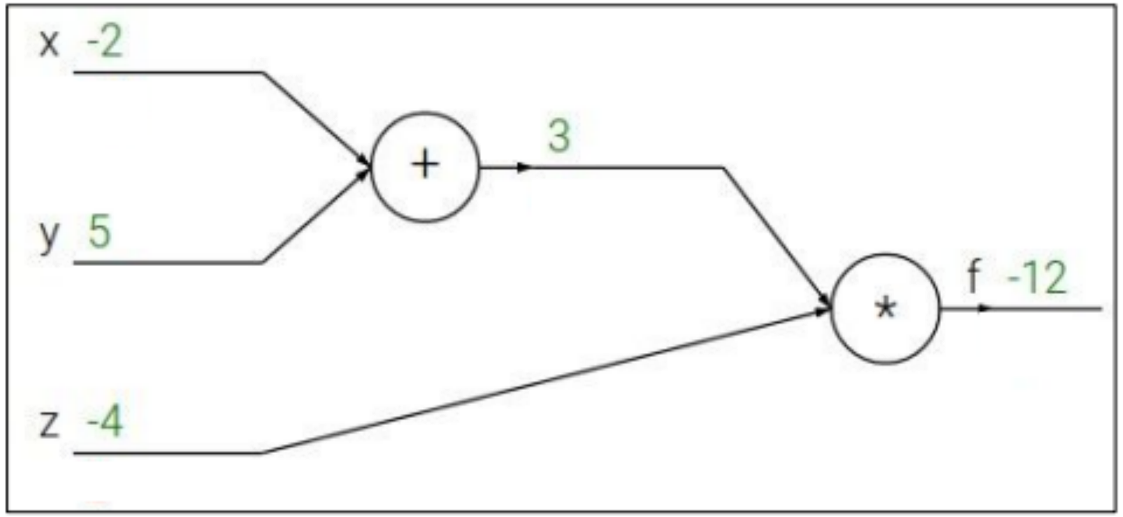
$$b^{[l]} \Leftarrow b^{[l]} - \alpha \frac{dJ}{db^{[l]}}$$

- How do we compute $\frac{dJ}{dW^{[l]}}$ and $\frac{dJ}{db^{[l]}}$?
- Go backward from the cost function J: backward propagation

Backward Propagation Example

- Initial Function $\mathrm{f(x,y,z)} = (x + y) * z$
- Computation Graph Functions:
  - $q(x,y) = x + y$
  - $f(q,z) = qz$
- Inputs: $\mathrm{x = \text{-}2, y = 5,\ z = \text{-}4}$
- Want to obtain:
  - $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$

Computation Graph for (x+y)* z



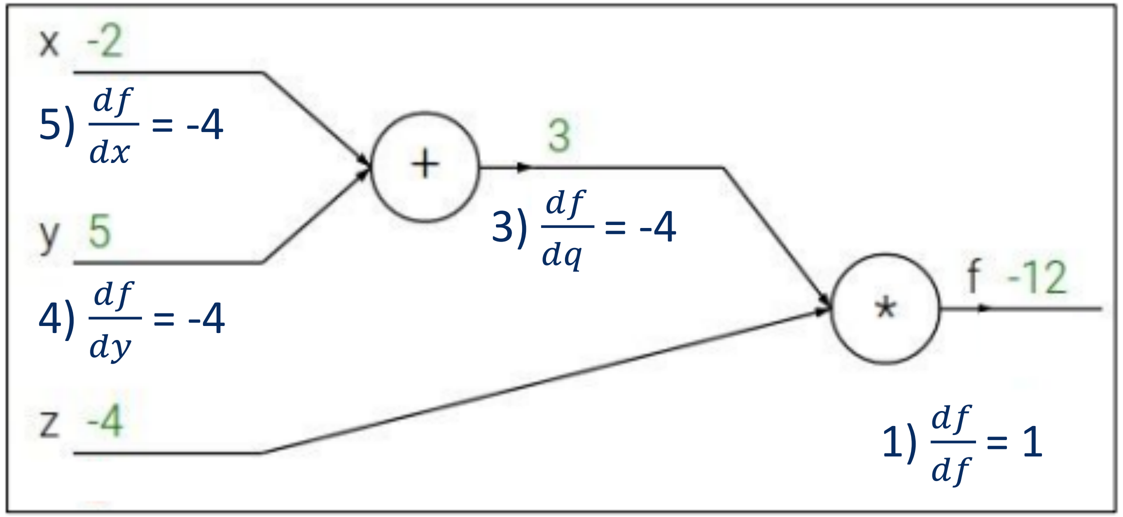
From http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

(Don’t confuse **Computation Graph** with actual **Neural Network**!)

Backward Propagation Example

- Computation Graph Functions:
  - $q(x, y) = x + y$
  - $f(q, z) = qz$
- Basic gradients :
  - $\dfrac{dq(x,y)}{dx} = 1, \ \dfrac{dq(x,y)}{dy} = 1$
  - $\dfrac{df(q,z)}{dq} = z, \ \dfrac{df(q,z)}{dz} = q$
- Compute gradients with chain rule:
  - $\dfrac{df}{dz} = q = 3$
  - $\dfrac{df}{dx} = \dfrac{df}{dq} * \dfrac{dq}{dx} = z * 1 = -4$
  - $\dfrac{df}{dy} = \dfrac{df}{dq} * \dfrac{dq}{dy} = z * 1 = -4$

Computation Graph for (x+y)* z



2) $\frac{df}{dz}$ = 3

From http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

Stochastic Gradient Descent with Mini-batch

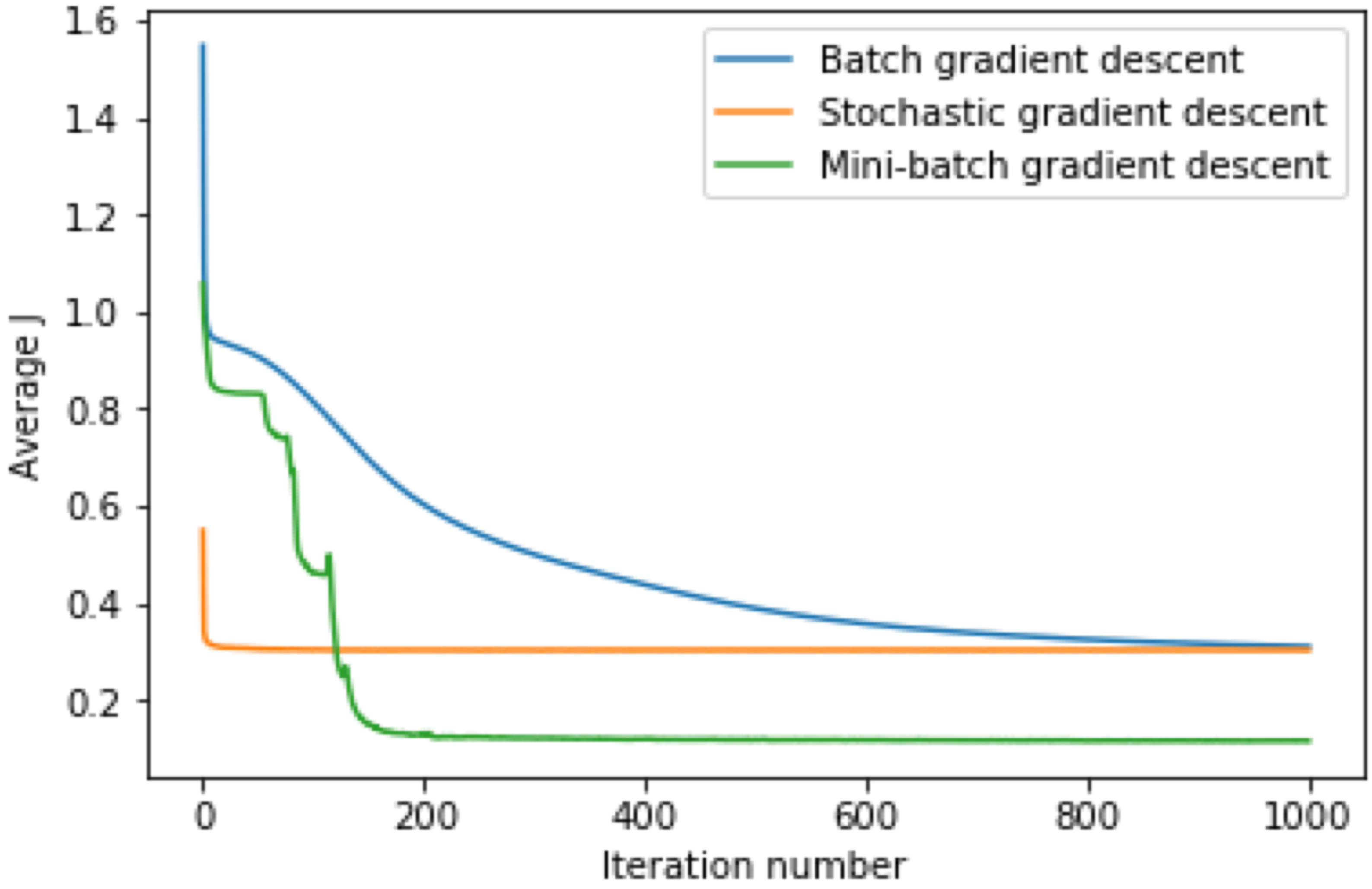
- (Batch) Gradient descent: J is computed for **all samples**
  - J is computed as the mean of loss functions for **all** samples:
    - $J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1 - y^{(i)})\log\left(1 - a^{[L](i)}\right)\right)$
  - Single update of weights with all samples at each iteration
  - Very smooth => Local Minima
  - Too much memory (all dataset)
- Stochastic Gradient Descent:
  - 1 Sample: J is computed with 1 sample and the weights updated
    - Too slow!
  - **Mini-batch SGD** (most used): J is computed with a **batch** (10-500) of samples

SGD Mini batch size implications

- Step: update the model with 1 mini-batch
- Epoch: composed of all steps (complete training-set)
- Small mini-batch
  - More computation per Epoch
  - Converges faster
  - Can achieve a local minima (not general – overfitting)

- Larger mini-batch
  - Converges slower
  - Can achieve a better minima

Batch size comparison



From: http://adventuresinmachinelearning.com/stochastic-gradient-descent/

PyTorch

[...] a Python based scientific computing package targeted at two sets of audiences:

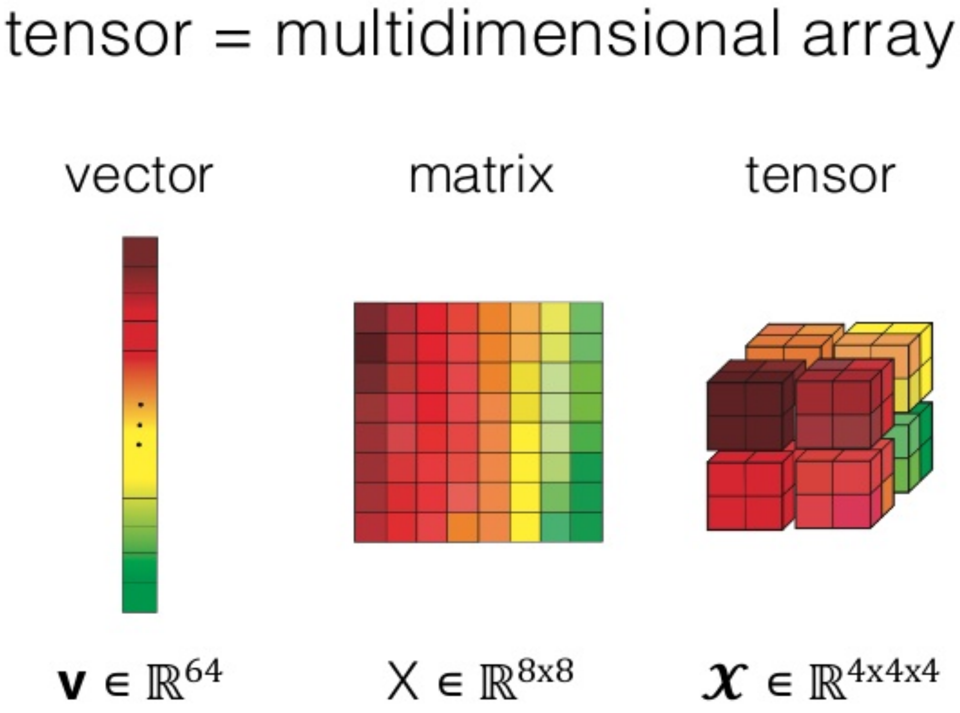
- *A replacement for numpy to use the power of GPUs*
- *a deep learning research platform that provides maximum flexibility and speed*

[This lesson uses material from http://pytorch.org/tutorials/ throughout.]

- To install: https://github.com/pytorch/pytorch#installation
- For this course use the version 1.0

Tensors

- Tensors are matrix-like data structures which are essential components in deep learning libraries and efficient computation.
- GPUs are especially effective at calculating operations between tensors



From: https://www.slideshare.net/BertonEarnshaw/a-brief-survey-of-tensors

- Tensor operations:
  - ones, zeros, add, dot, etc.
- PyTorch tensors can live on
  - CPU
  - GPU (speedup!)

PyTorch Tensors

- Import Torch:

```
from __future__ import print_function
import torch
```

- Construct a 2x3 matrix, uninitialized:

```
x = torch.Tensor(2, 3)
print(x)
 0.00e+00  0.00e+00  1.15e-24
 -1.58e+29  1.67e-37  2.97e-41
[torch.FloatTensor of size 2x3]
```

- Use tensor in CUDA

```
device = torch.device("cuda")
y = torch.ones_like(x, device=device) # directly create a tensor on GPU
x = x.to(device) # or just use .to("cuda")
```

- Initialize zeros or ones tensors

```
x = torch.zeros(2,3)
x = torch.ones(2,3)
```

- Convert a numpy array

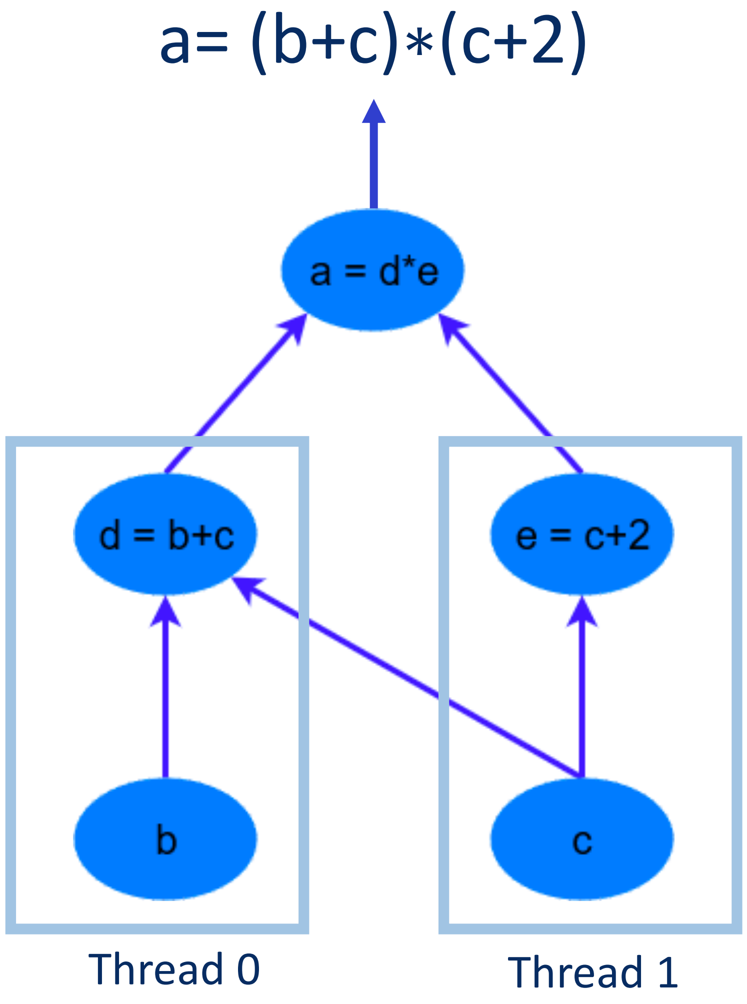
```
b = torch.from_numpy(a)
```

- the slice functionality is available like in numpy

```
x = torch.rand(2,3) # Initialize a tensor randomly
print x[:,1] #second column
0.6297 0.1196
[torch.FloatTensor of size 2]
print x[0,:] #first row
0.9749 0.6297 0.3045
[torch.FloatTensor of size 3]
```

Computation Graphs

- A computational graph represents a function in a directed acyclic graph of its component functions
- **Performance optimizations:** Computation Graph exposes parallelism!
- In PyTorch the graph construction is **dynamic:** the graph is built at run-time
  - Easier debugging
  - Better for some algorithms (RNNs)
- In TensorFlow is graph construction is **static:** meaning the graph is "compiled" and then run
  - Compiler adds latency but can also apply optimizations



Autograd in PyTorch

- **Autograd** builds the Computation Graph **Dynamically**
- The **Tensor** class is the main component of this autograd system in PyTorch (from PyTorch 0.4 version, the *Variable* class is deprecated)
- If you set its attribute *.requires_grad* as *True*, it starts to track all operations on it
- The gradient for this tensor will be accumulated into *.grad* attribute
- Tensors allow automatic gradient computation when the *.backward()* function is called
- Based on the graph, *<variable>.backward()* computes the **gradient** and writes it in ***grad***
  - Example: **b.backward()** computes $\frac{d(y)}{dx}$

```
x = torch.randn(5, 5) # requires_grad=False by default
y = torch.randn(5, 5) # requires_grad=False by default
z = torch.randn((5, 5), requires_grad=True)
a = x + y
a.requires_grad
            False
b = a + z
b.requires_grad
            True
b.backward
```

PyTorch Tensors, Functions and Gradients

- Create a tensor

```
x = torch.tensor(torch.ones(2, 2) * 2,
requires_grad=True)
```

- Do a simple math equation:

```
z = 2 * (x * x) + 5 * x
```

- To get the gradient of this operation with respect to *x* i.e. *dz/dx* we can analytically calculate this.

- If all elements of *x* are 2, then we should expect the gradient *dz/dx* to be a (2, 2) shaped tensor with 13-values.
- However, first we have to run the .backwards() operation to compute these gradients.
- To compute gradients, we need to compute them with respect to something.
- In this case, we can supply a (2,2) tensor of 1-values to be what we compute the gradients against – so the calculation simply becomes *d/dx*:

```
z.backward(torch.ones(2, 2))
print(x.grad)
        tensor([[ 13.,  13.],
                [ 13.,  13.]])
```

PyTorch Neural Network

- *torch.nn.module* is used to define a neural network
- Example: NN with 3 fully connected layers
  - Using RELU activation for 1st and 2nd layer
  - Input to 1st FC layer: 256 features
  - Input to 2nd FC layer: 120 values
  - Input to 3rd FC layer: 10 values
- Define only forward prop. : backward prop is automatically derived from it

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

#Inherit from class nn.Module
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # y = Wx + b
        self.fc1 = nn.Linear(256, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

PyTorch Loss Function

- Loss function:
  - How "far" from the **target** is the **output** of forward propagation (prediction)
- NN provides various loss functions with syntax:
  - *loss = <loss-function>(output, target)*
  - *loss, output* and *target* are Tensors

```
#net is the network previously defined
#input is the input data of the network
net = Net()
input = torch.randn(256) # a dummy input
output = net(input)
#criterion is a Mean-Squared Error loss function
criterion = nn.MSELoss()

target = torch.arange(1, 11)  # a dummy target
#target comes from the labelled dataset
loss = criterion(output, target)

print(loss)
```

PyTorch Backpropagation and weights update

- First reset gradients of the network
- Compute backward prop.
  - It uses *autograd* insides
- Update the weights with STG:

  *weight = weight - learning_rate * gradient*
- Different optimization algorithms are in *torch.optim*

```
# Zeroes the gradient buffer of all parameters
net.zero_grad()

#Backpropagation step
loss.backward()

#Stocastic Gradient Descent weights update
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

PyTorch Examples

Autograd Example (1)

```
Import torch

dtype = torch.float
device = torch.device("cpu") # Use the CPU as device
# Alternatively, use device = torch.device("cuda:0") to run on GPU

# 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 input and outputs.
# Setting requires_grad=False indicates that we do not need to compute gradients
# with respect to these Tensors during the backward pass.
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Create random Tensors for weights.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)
```

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

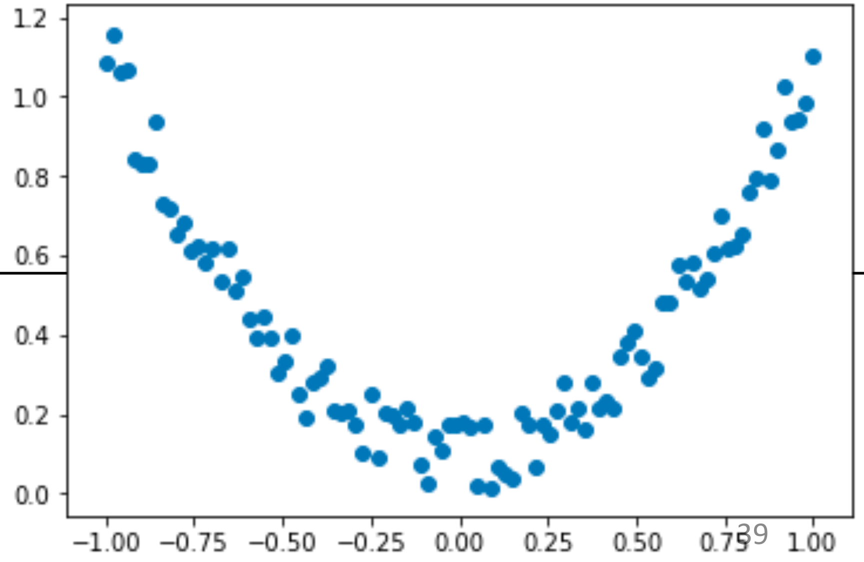
Autograd Example (2)

```
learning_rate = 1e-6
for t in range(500):
  # Forward pass: compute predicted y using operations on Variables;
  # mm: matrix multiply; clamp: clamp in [min;max]
  y_pred = x.mm(w1).clamp(min=0).mm(w2)
  # Compute and print loss. Now loss is a Tensor of shape (1,)
  # loss.item() is a scalar value holding the loss.
  loss = (y_pred - y).pow(2).sum()
  print(t, loss.item())
  # Use autograd to compute the backward pass.
  loss.backward()
  # Update weights using gradient descent; w1.data and w2.data are Tensors,
  # w1.grad and w2.grad are Variables and w1.grad.data and w2.grad.data are Tensors.
  with torch.no_grad():
    w1 -= learning_rate * w1.grad
    w2 -= learning_rate * w2.grad
    # Manually zero the gradients after updating weights
    w1.grad.zero_()
    w2.grad.zero_()
```

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Linear Regression Example (1)

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)  # x data (tensor), shape=(100, 1)
#unsqueeze: reshape tensor
#linspace: return a one-dimensional vector of 100 points between -1 and 1
y = x.pow(2) + 0.2*torch.rand(x.size())             # noisy y data (tensor), shape=(100, 1)

plt.scatter(x.numpy(), y.numpy())
plt.show()
```



https://github.com/MorvanZhou/PyTorch-Tutorial

Linear Regression Example (2)

```
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden)   # hidden layer
        self.predict = torch.nn.Linear(n_hidden, n_output)   # output layer

    # For the forward() method, we supply the input data x as the primary argument.
    # The we pass through the two layers of our simple network
    def forward(self, x):
        x = F.relu(self.hidden(x))      # activation function for hidden layer
        x = self.predict(x)             # linear output
        return x
```

https://github.com/MorvanZhou/PyTorch-Tutorial

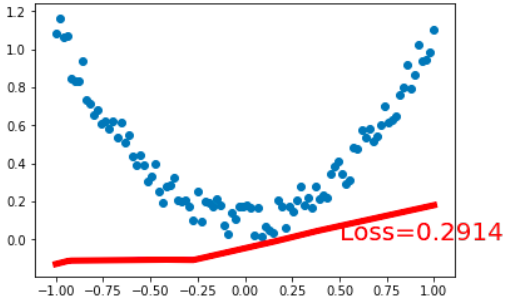
Linear Regression Example (3)

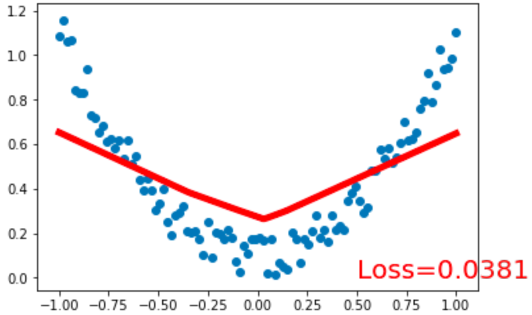
```
# create the network
net = Net(n_feature=1, n_hidden=10, n_output=1)     # define the network
print(net)  # net architecture

# create a stochastic gradient descent optimizer
# the method net.parameters() (from the base nn.Module class that we inherit in the Net
class), contains all the parameters of our network
optimizer = torch.optim.SGD(net.parameters(), lr=0.2)
# define the loss, we use the regression mean squared loss
loss_func = torch.nn.MSELoss()

plt.ion()   # something about plotting
for t in range(200):
    prediction = net(x)     # input x and predict based on x
    loss = loss_func(prediction, y)     # must be (1. nn output, 2. target)
    # With optimizer.zero_grad() we resets all the gradients in the model
    # so that it is ready to go for the next back propagation pass.
    optimizer.zero_grad()   # clear gradients for next train
    loss.backward()         # backpropagation, compute gradients
    # runs a back-propagation operation from the loss Tensor backwards
    # through the network and execute a gradient descent step based on
    # the gradients calculated during the .backward() operation.
    optimizer.step()        # apply gradients
```

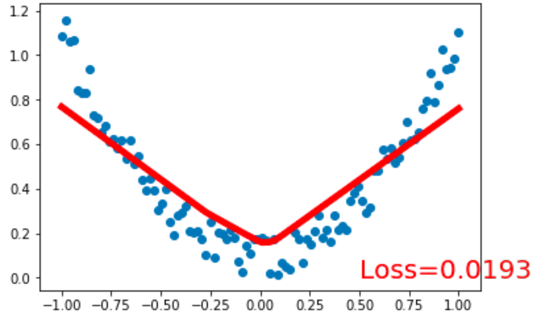
https://github.com/MorvanZhou/PyTorch-Tutorial

Linear Regression Example (4)

```
        # plot every 5 epochs the learning process
        if t % 5 == 0:
            # plot and show learning process
            plt.cla()
            plt.scatter(x.numpy(), y.numpy())
            plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=5)
            plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 20, 'color':  'red‘})
            plt.pause(0.1)
plt.ioff()
plt.show()
```

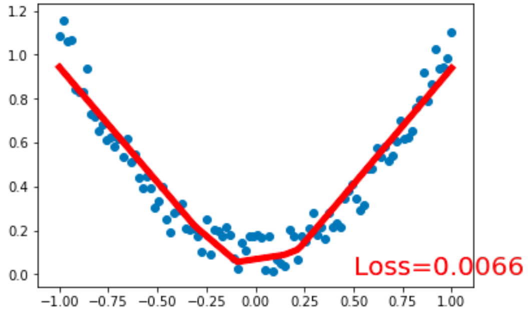


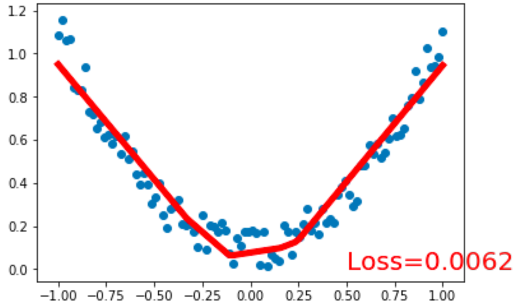




https://github.com/MorvanZhou/PyTorch-Tutorial

....





NN Example (1)

```
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 as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(reduction='sum')
```

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

NN Example (2)

From: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

```
learning_rate = 1e-4
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(x)
    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.item())
    # Zero the gradients before running the backward pass.
    model.zero_grad()
    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Tensors with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()
    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

Lesson Key Points

- ML Basics:
  - Linear Regression, Logistic Regression, Gradient Descent
  - Neural Networks (Inference/Training, STG, Batch Size)
- PyTorch:
  - Tensors, Variables, Autograd
  - Examples: Autograd, Linear Regression, NN

AI services in real life

- Speech recognition
  - https://www.ibm.com/watson/services/speech-to-text/
- Natural language classifier
  - https://www.ibm.com/watson/services/natural-language-classifier/