Deep Learning for Beginners

Weiguang Guan
guanw@sharcnet.ca

Code & data: guanw.sharcnet.ca/ss2017-deeplearning.tar.gz

SHARCNET™

Outline

- Part I: Introduction
  - Overview of machine learning and AI
  - Introduction to neural network and deep learning (DL)
- Part II: Case study – Recognition of handwritten digits
  - Write our own DL code
  - Use a DL library

Reference

- "Deep Learning Tutorial" by Yann LeCun, http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf
- "Deep Learning Tutorial" by Yoshua Bengio, http://deeplearning.net/tutorial/deeplearning.pdf
- "Neural Networks and Deep Learning" by Michael Nielsen, http://neuralnetworksanddeeplearning.com

Part I

Introduction to AI, Machine learning, and neural network

Overview

- What is AI?
- What/how can a machine learn?
- Machine learning methods with focus on deep learning
- Caveats and pitfalls of machine learning

Artificial Intelligence (AI)

- What is AI
  - Def 1: Computer systems able to perform tasks that normally require human intelligence.
  - Def 2: intelligent machines that work and react like humans
  - Def 3 … : more on the internet…

Artificial Intelligence (AI)

- Are these in the domain of AI?
  - Computing
  - Database
  - Logical operations
  - …

Artificial Intelligence (AI)

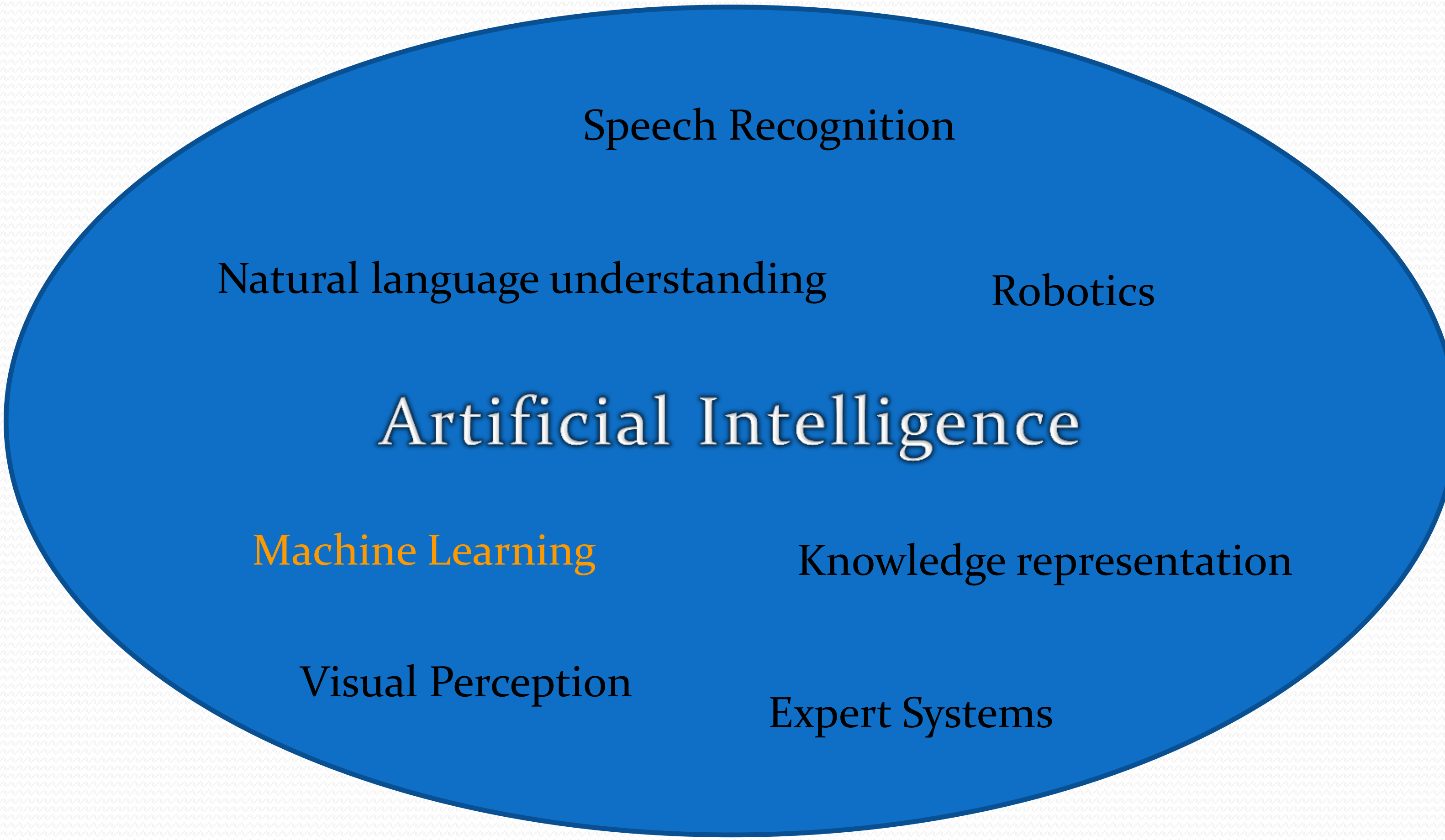
Intelligent robot made by Boston Dynamics

https://youtu.be/rVlhMGQgDkY

AI system (my definition)

- Is able to perform an intelligent task by learning from examples
- We humans don't know the explicit rules/instructions to perform the task

Artificial Intelligence (AI)

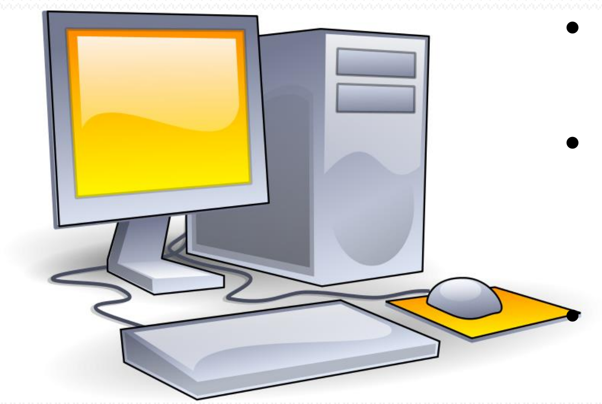


Artificial Intelligence (AI)

- History of AI
- Brain vs computer

Logic gates

Bio-chemical
operations
???

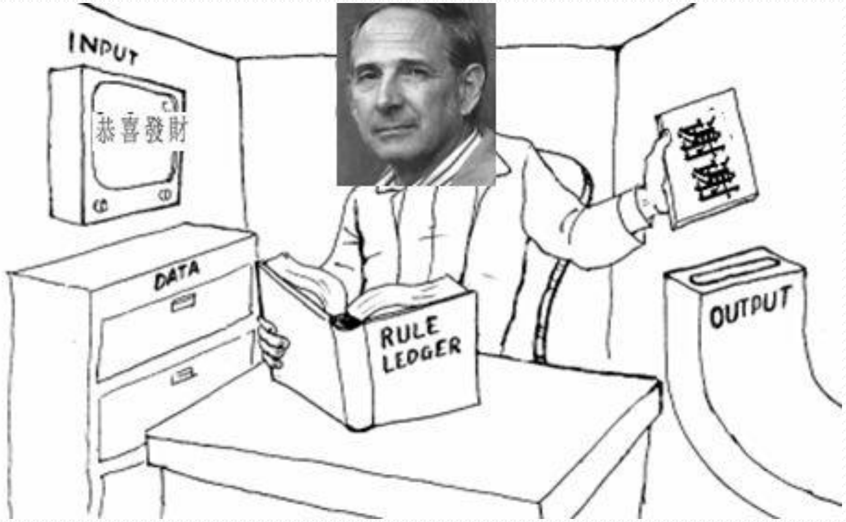


- Memory
- Information processing
  (Computing vs thinking)
- Sensing
  (camera and microphone vs eyes and ears)
- Responding

Artificial Intelligence (AI)

- What are minds?
- What is thinking?
- To what extent can computers have intelligence?

Strong AI vs weak AI



Chinese room

Artificial Intelligence (AI)

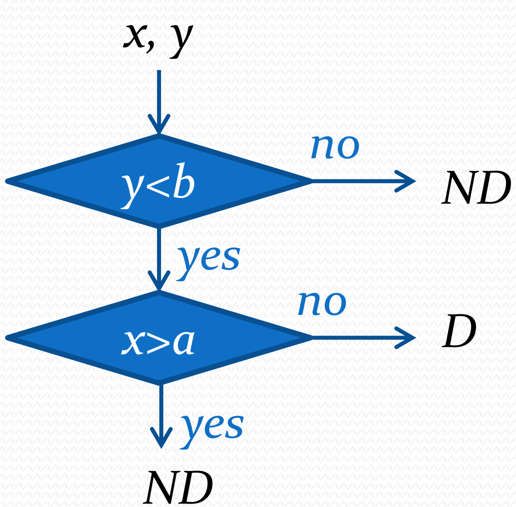
- Computers don’t and won’t have
  - Passion, feeling, consciousness…
  - Inherent common sense
  - Motivation
- Computers can be trained to do particular tasks (as good as humans or even better)
- “Thinking is computing”

How does Machine Learning work

- What is learned by computer?

  A parameterized model used to perform a particular task.

  Task: to predict diabetes based on sugar intake *x* and hours of exercise *y per day*.

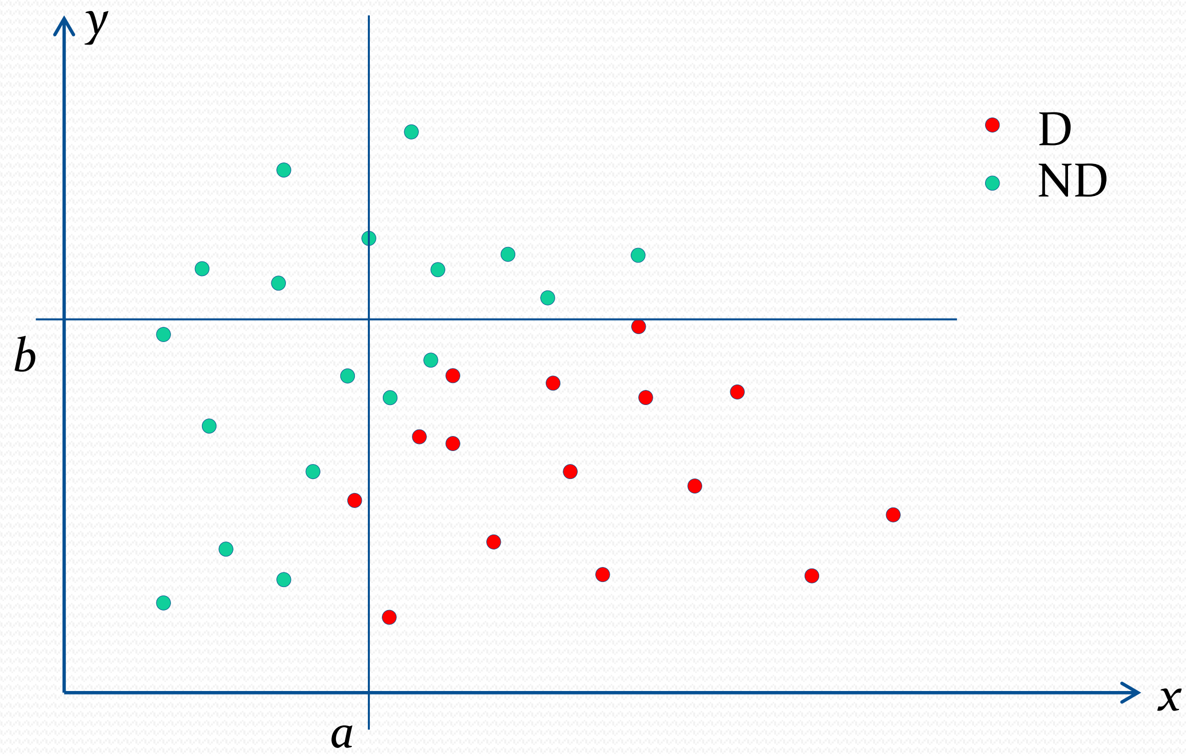


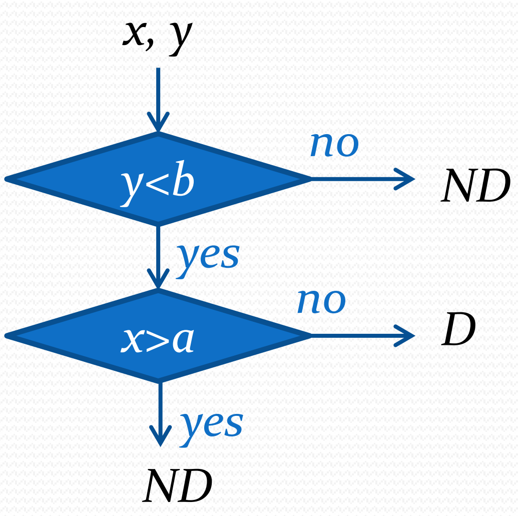
Input: *x*, *y*
Output: either D (Diabetes)
or ND (not Diabetes)
Parameters: *a*, *b*

Machine Learning

- How does a computer learn?

  Learns from (many) samples





Machine Learning

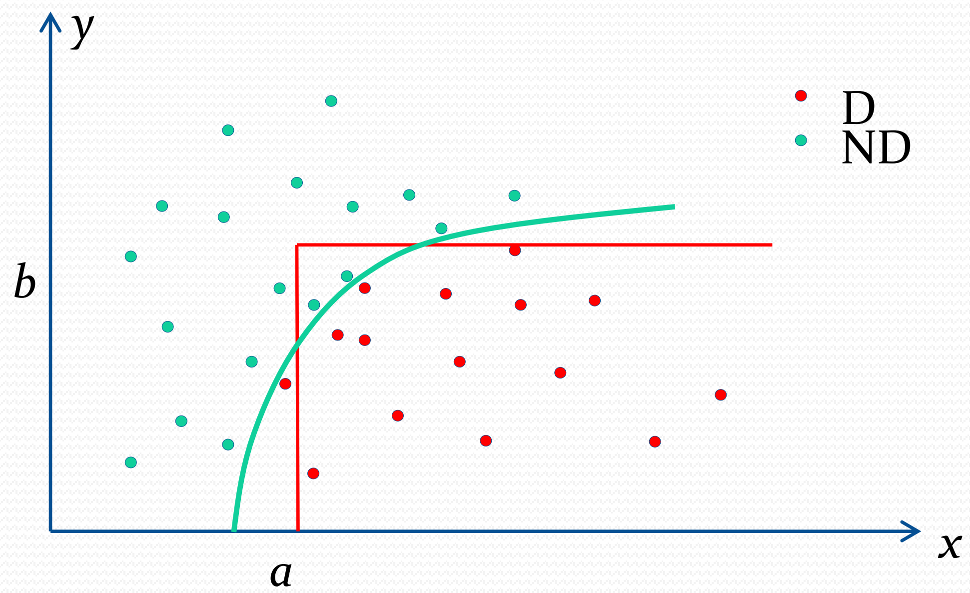
- Learning becomes an optimization problem:

  Determine parameters (a, b) so that a pre-defined cost function (e.g., misclassification error) is minimized.

- Training or learning is usually an iterative process where parameters are gradually changed to make the cost function smaller and smaller.

Machine Learning

- Basic concepts (cont'd)
  - Feature space
  - Decision boundary



Two categories of learning

- Supervised learning

  Learn from annotated samples

- Unsupervised learning

  Learn from samples without annotation

Machine learning methods

- Deep learning
- Boosting
- Support Vector Machine (SVM)
- Naïve Bayes
- Decision tree
- Linear or logistic regression
- K-nearest neighbours (KNN)

Sample data

- Basic concepts
  - Sample
    - Sample is defined as a vector of attributes (or features), each of which can be
      - Numerical
      - Categorical
        - Ordered
        - No order
      - Binary or Boolean
  - Label can be
    - Categorical (most often, binary) --- classification
    - Numerical --- regression

Sample

- Example of data sample

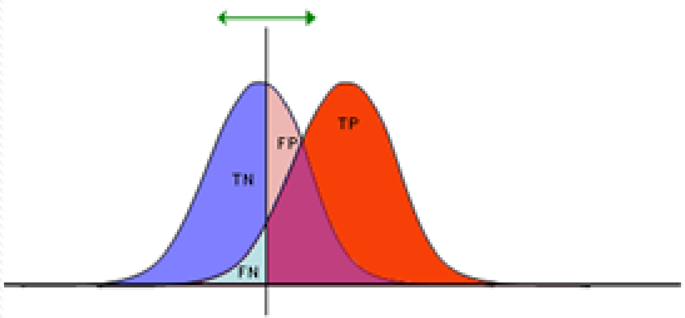
Name	Occupation	Smoking	Sugar intake	Hours of exercise	…	Glucose level	Diabetes
Peter	Driver	Yes	100.0	5.5	…	80	No
Nancy	Teacher	No	50.0	3	…	120	Yes
…	…	…	…	…	…	…	…

Feature vector

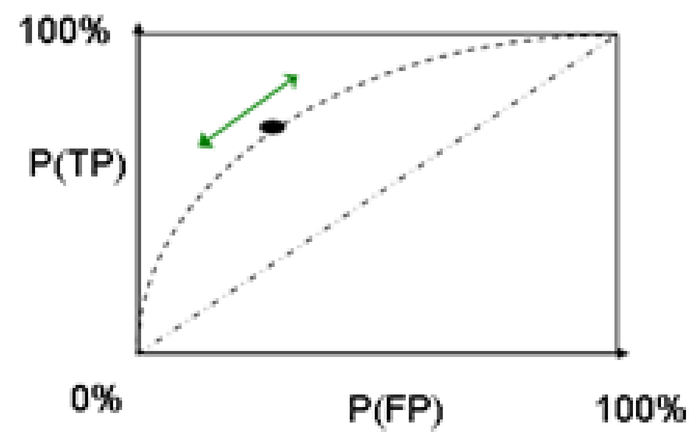
Annotation

Performance measures

- Basic concepts (cont'd)
  - Classification error
    - Binary classification
      - TP, TN, FP, FN
      - ROC curve
      - TPR = TP/(TP+FN), FPR =FP/(FP+TN)



TP	FP
FN	TN
1	1



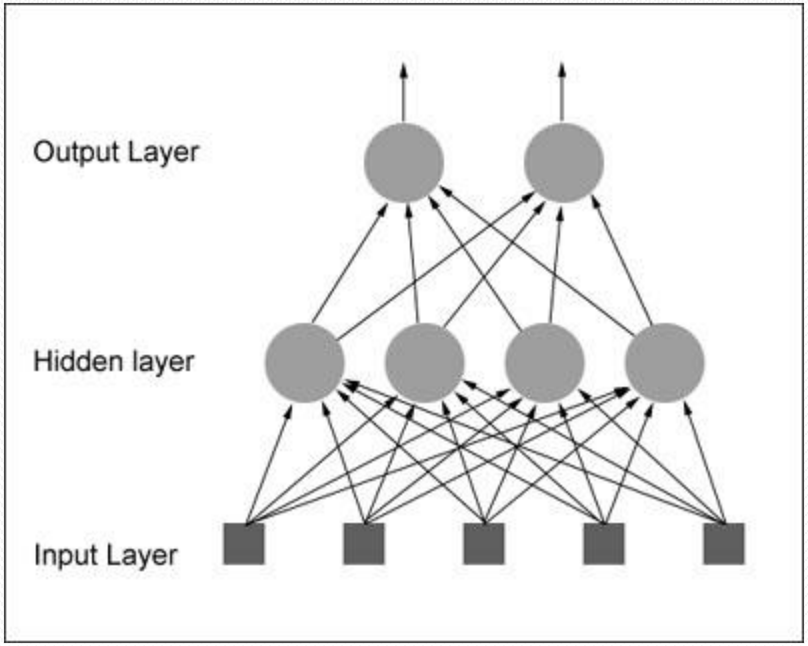
Performance measures

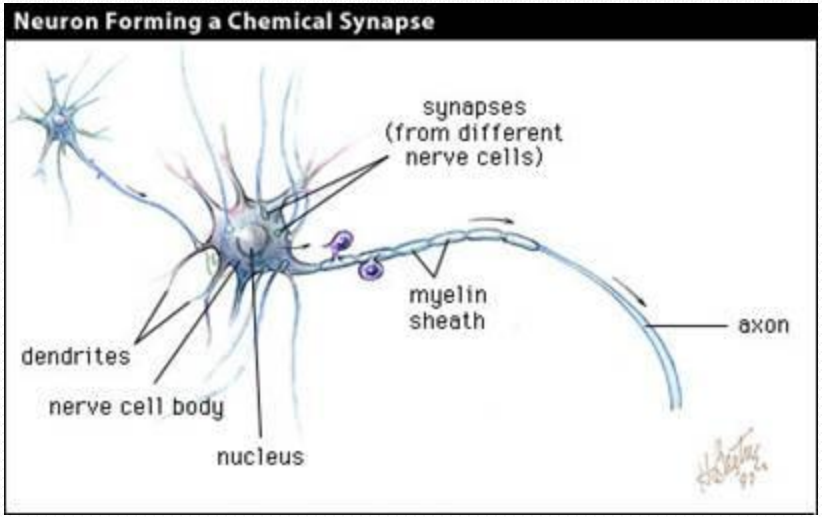
- Basic concepts (cont'd)
  - Classification error
    - Multiclass classification
      - Confusion matrix

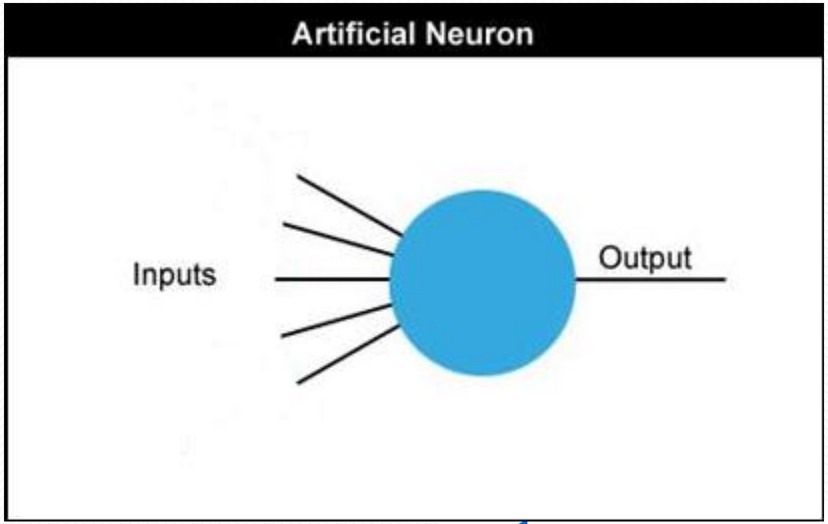
		Predicted		
		Cat	Dog	Rabbit
Actual class	Cat	5	3	0
	Dog	2	3	1
	Rabbit	0	2	11

Neural network and deep learning

- Directed graph of neurons





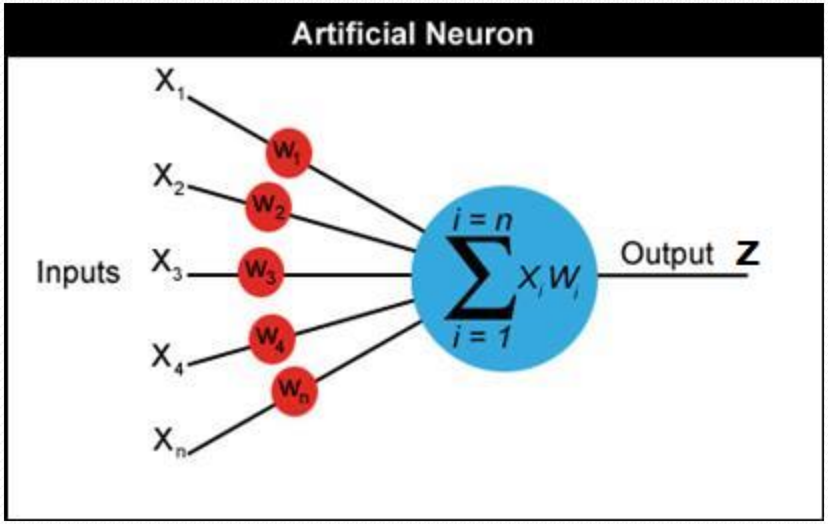


Neurons

- Neurons
  - Linear weighted sum
  - Perceptron
  - Sigmoid
  - Rectified Linear Units (ReLu)
  - …
- Layers:
  - Pooling
  - Convolution
  - Loss
  - …

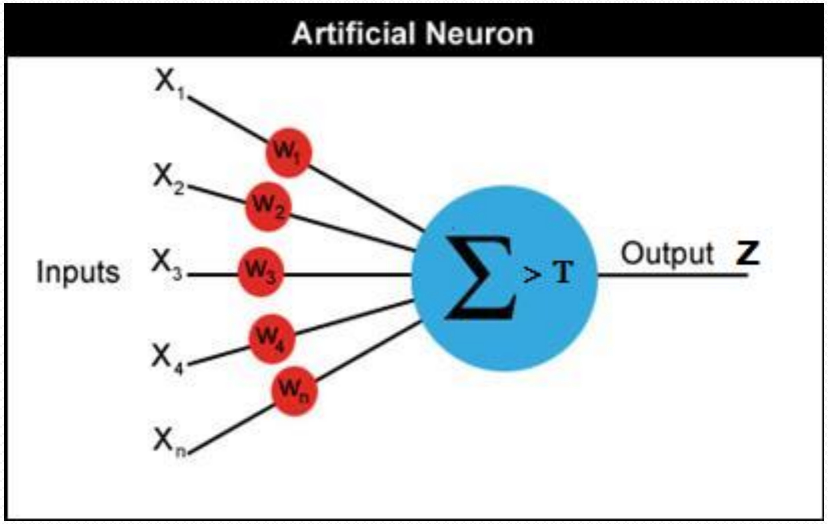
Linear weighted sum

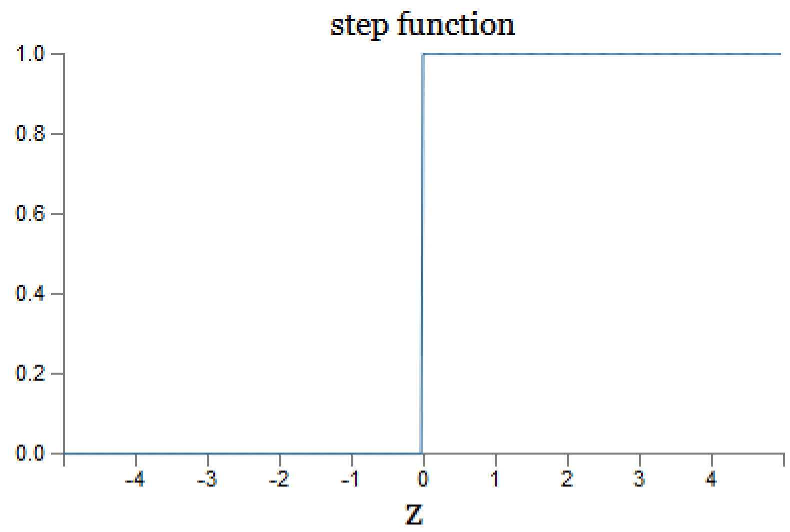
- $z = \sum_j w_j x_j = \boldsymbol{w}^t \, \boldsymbol{x}$



Perceptron

- $z = \begin{cases} 0, \text{if } \sum_j w_j x_j \leq T \\ 1, \quad \text{otherwise} \end{cases}$
- $z = \begin{cases} 0, \text{if } \boldsymbol{w}^t \boldsymbol{x} + b \leq 0 \\ 1, \quad \text{otherwise} \end{cases}$
  - weights: $\boldsymbol{w} = (w_1, w_2, \dots, w_n)$
  - bias: $b = -T$

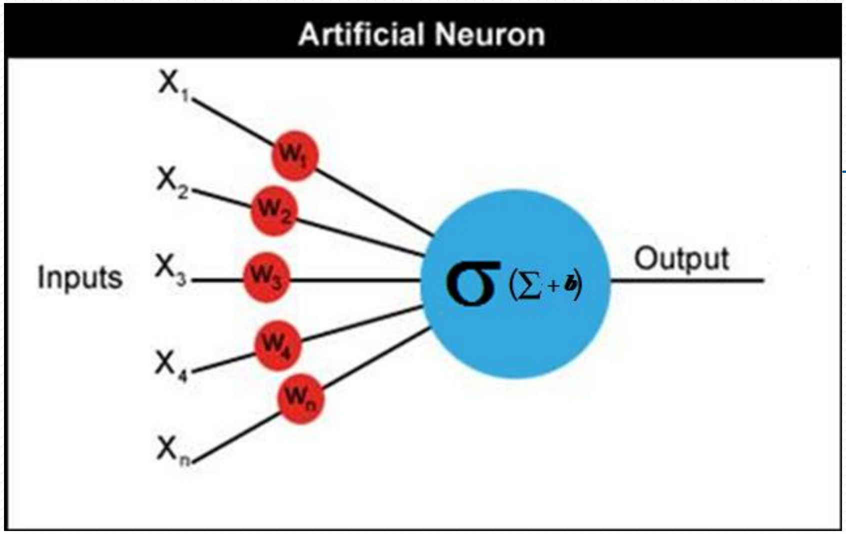


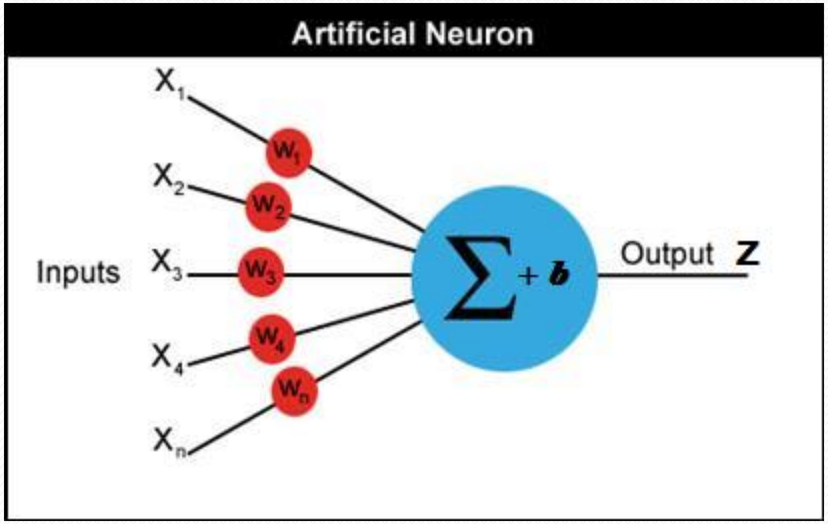


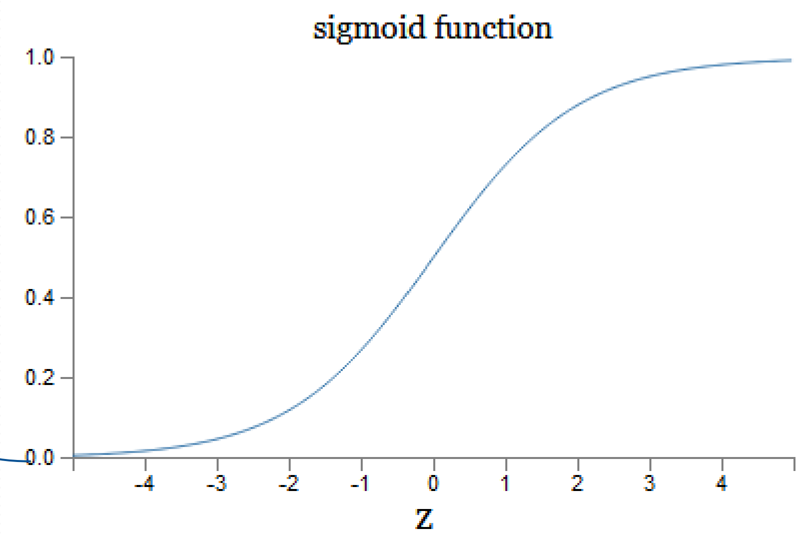
Sigmoid neuron

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

where $z = \boldsymbol{w}^t \boldsymbol{x} + b$



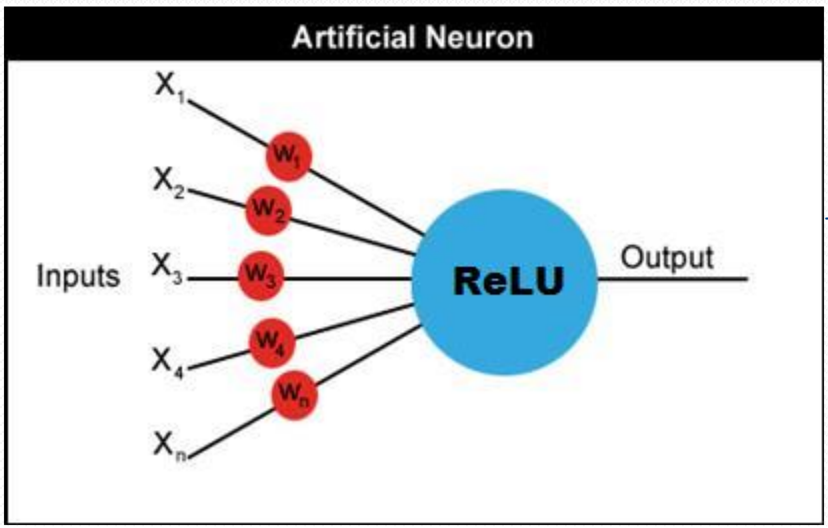


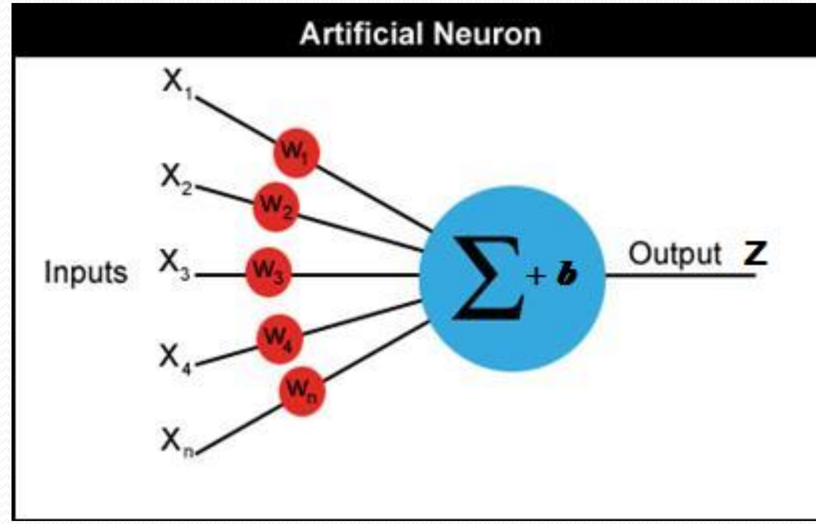


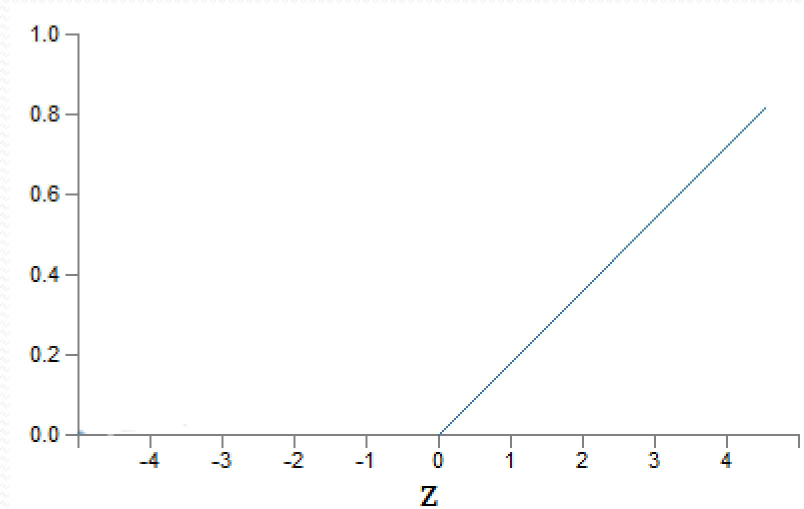
Rectified Linear Units

$$\sigma(z) = \max(0, z)$$

where $z = \boldsymbol{w}^t \boldsymbol{x} + b$

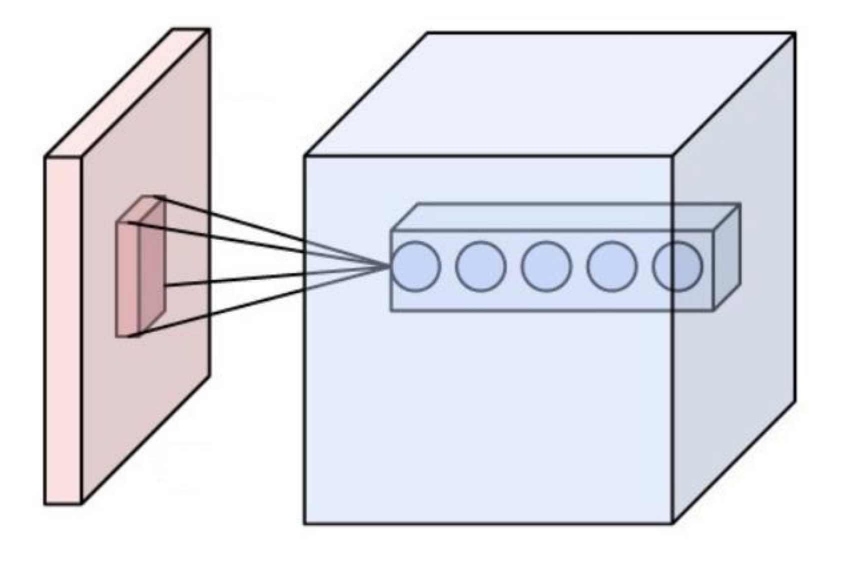






Convolution layer

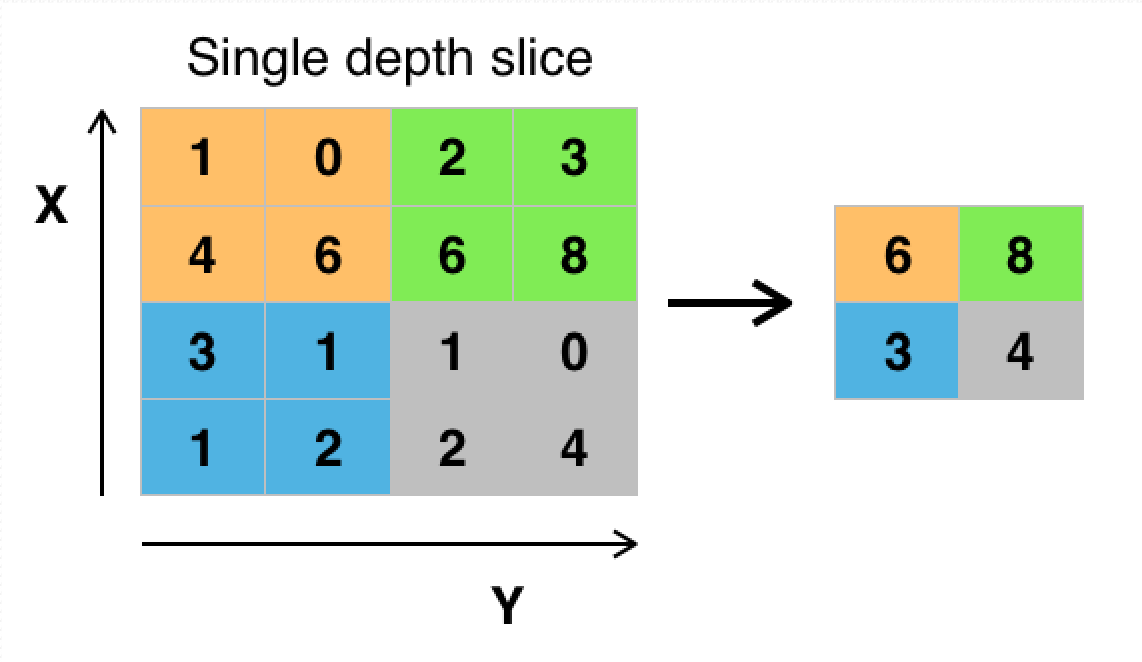
- A set of learnable filters (or kernels) --- 2D array of weights
- Each neuron (blue) of a filter has its receptive field (red) in the input image
- Dot product between receptive field and a filter



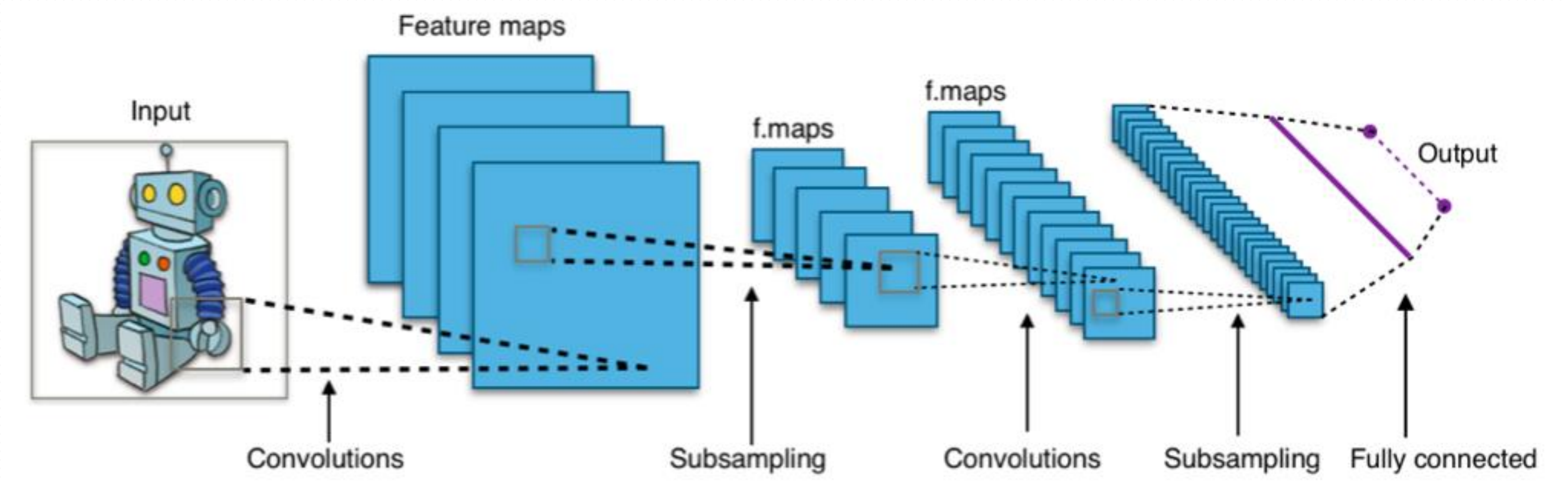
Pooling layer

- Sub-sampling

Example of max pooling

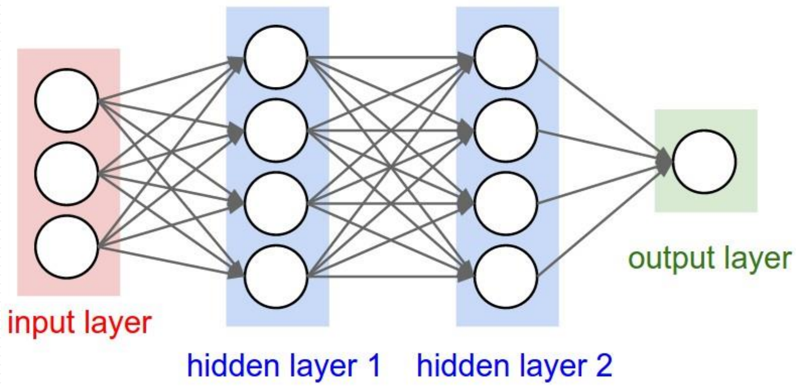


Architecture of neural network

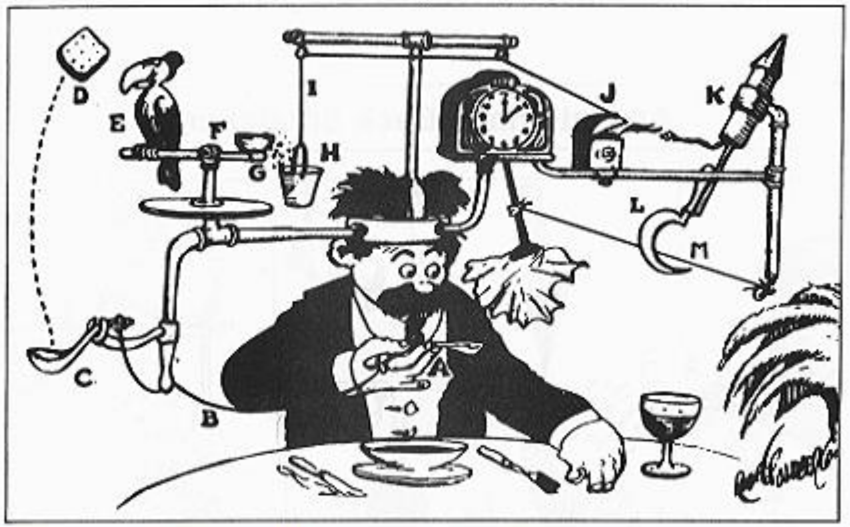


Train a neural network

- Purpose: To determine the parameters of a given network so that it will produce desired outputs.



Self-Operating Napkin



Professor Butts and the Self-Operating Napkin (1931)

SHARCNET™

How to achieve the training goal

- Define a cost (objective) function to measure the performance of a NN, such as

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - g(\boldsymbol{x}_i))^2$$

where $\boldsymbol{x}_i$ and y_i are the feature vector and label of the i-th sample. $g(\boldsymbol{x}_i)$ is the output of the NN.

- $g(\boldsymbol{x}_i)$ depends on the parameters of the NN.
- Gradient descent is used to find the values of these parameters that minimize the cost function.

Gradient descent (1D function)

$y = f(x) = x^2 - 6x + 10$

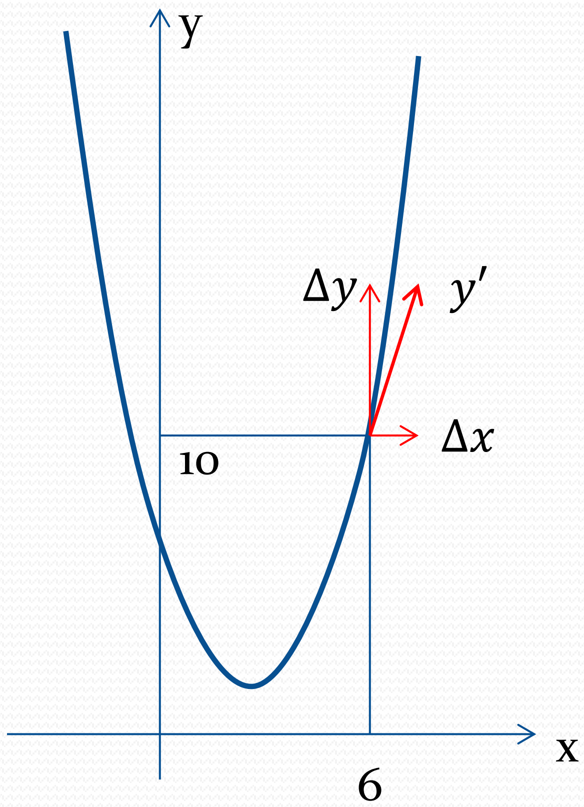
- Starting $x = 6$, we get $y = 10$
- $x = x \pm \Delta x$, where $\Delta x = 0.01$, we get

  $y = 9.9401$ for $x = 5.99$

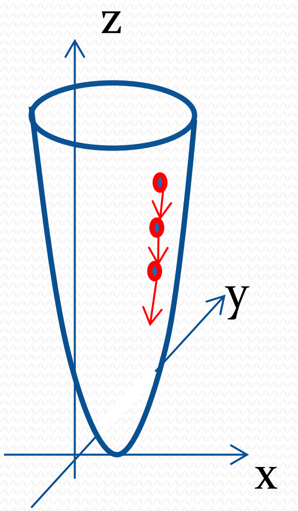
  and

  $y = 10.0601$ for $x = 6.01$

- $y' = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \approx \frac{0.0601}{0.01} = 6.01$ where $\Delta y = f(x + \Delta x) - f(x)$
- Analytic $y' = 2x - 6$



Gradient descent (2D function)

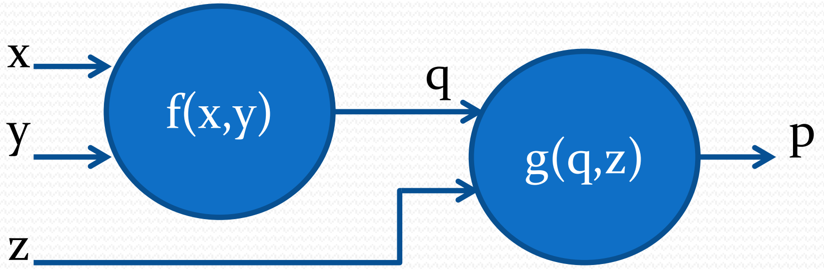


$z = f(x, y) = x^2 + y^2 - 6x + 10$

- Starting $x = 6, y = -2$, we get $z = 14$
- Analytic: $\frac{\partial z}{\partial x} = 2x - 6, \quad \frac{\partial z}{\partial y} = 2y$
- Gradient $\nabla z = (\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y})$ is the fastest ascending direction
- Iteratively approaching with learning rate $\eta = 0.01$
  - Round #1: $(x, y) = (6, -2)$, we get $\nabla z = (6, -4)$
  - Round #2: $(x, y) = (x, y) - \eta \nabla z = (5.94, -1.96)$, we get $\nabla z = (5.88, -3.92)$
  - Round #3: ……

Train a neural network

- Initialize the state of a neural network (by randomizing all the parameters)
- Iterative process
  - Feed forward
  - Backpropagation (chain rule)



$$\frac{\partial p}{\partial x} = \frac{\partial p}{\partial q}\frac{\partial q}{\partial x}$$

$$\frac{\partial p}{\partial y} = \frac{\partial p}{\partial q}\frac{\partial q}{\partial y}$$

$$\frac{\partial p}{\partial z} = \frac{\partial p}{\partial z}$$

Chain rule of derivatives

$$p = f(x, y, z)$$
$$x = x(u, v, w)$$
$$y = y(u, v, w)$$
$$z = z(u, v, w)$$

$$\frac{\partial p}{\partial u} = \frac{\partial p}{\partial x} * \frac{\partial x}{\partial u} + \frac{\partial p}{\partial y} * \frac{\partial y}{\partial u} + \frac{\partial p}{\partial z} * \frac{\partial z}{\partial u}$$

$$\frac{\partial p}{\partial v} = \frac{\partial p}{\partial x} * \frac{\partial x}{\partial v} + \frac{\partial p}{\partial y} * \frac{\partial y}{\partial v} + \frac{\partial p}{\partial z} * \frac{\partial z}{\partial v}$$

$$\frac{\partial p}{\partial w} = \frac{\partial p}{\partial x} * \frac{\partial x}{\partial w} + \frac{\partial p}{\partial y} * \frac{\partial y}{\partial w} + \frac{\partial p}{\partial z} * \frac{\partial z}{\partial w}$$

Wait a minute ...

- Questions
  - We want to minimize a cost function, not the output of NN
  - We want to tweak the parameters of NN, not the input data (x, y, z ...) to do the minimization
- Let change the roles ...
  - Consider $a, b, \ldots, f$ as variables in $z = f(x, y) = ax^2 + by^2 + cxy + dx + ey + f$
  - The cost function C is a function of the output and ground truth so that we can compute $\frac{\partial C}{\partial a}, \frac{\partial C}{\partial b}, \ldots, \frac{\partial C}{\partial f},$
  - Apply gradient descent on $a, b, \ldots, f$ the same way

Training a neural network

- Cost function $C(\boldsymbol{w}, \boldsymbol{b}) = \frac{1}{n}\sum_{i=1}^{n} C_i = (y_i - g(\boldsymbol{x}_i))^2$, where $\boldsymbol{w}$, $\boldsymbol{b}$ are the weights and biases of NN.
- $\nabla C = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i$, n is the number of training samples
- $\nabla C \approx \frac{1}{m}\sum_{i=1}^{m} \nabla C_i$, randomly divide training set into small subsets (mini-batches), each of which contains $m \ll n$ samples. An epoch is one complete pass going through all the mini-batches.
- Training with mini-batches is called stochastic gradient descent (SGD)

Caveats and pitfalls

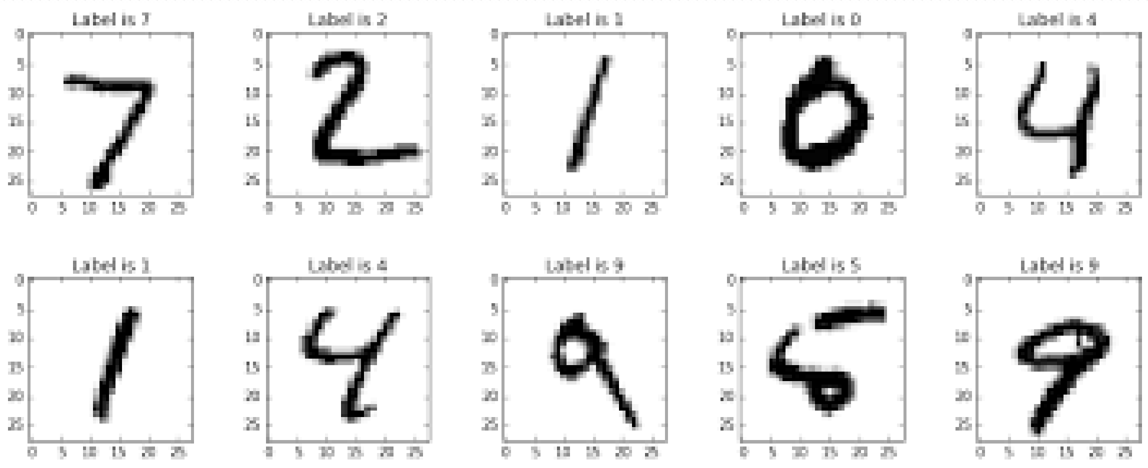
- Feature selection
  - Relevance
  - Redundancy
- Sample data
  - Mislabeling
  - Outliers
- Overfitting
  - Small training set
  - Low-quality training data
  - Over-training
- Confidence level of classification
- Re-tune a trained model to operate on a different position on the ROC curve

Part II

Case Study:
Recognition of hand-written digits

Tasks

- Write our own NN code
- Use DL libraries (Tensorflow)



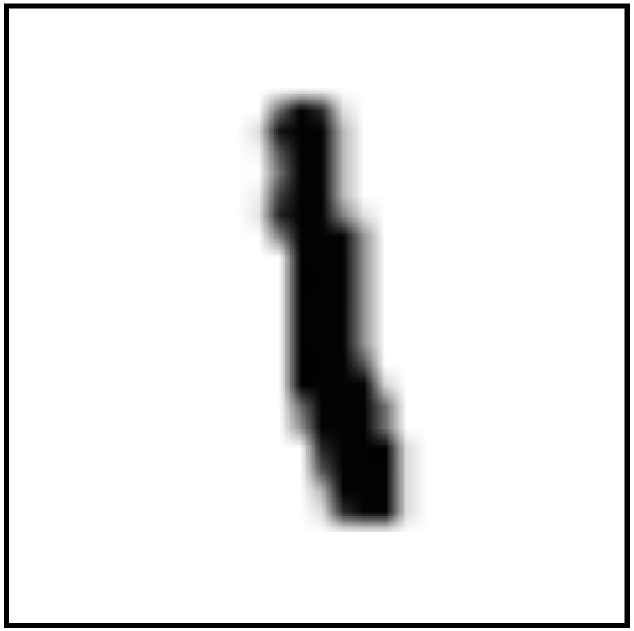
Four elements in deep learning

- Datasets (for training, testing …)
- Design of a neural network
- Cost function that training tries to minimize
- Training method(solver or optimizer, such as SGD, AdaDelta, Adaptive Gradient, etc)

Dataset of handwritten digits

- Dataset (http://yann.lecun.com/exdb/mnist/)
  - 60,000 training samples
  - 10,000 testing samples
  - Each sample is 28x28 gray scale image with a label telling what digit it is.
  - The label (0, 1, …, 9) is vectorized as a "one-hot" vector, e.g., [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] represents 3.
- checkdata.py

Dataset of handwritten digits

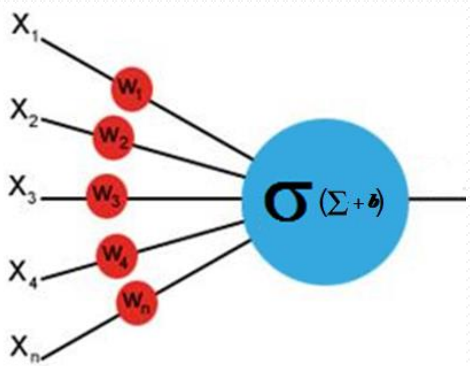


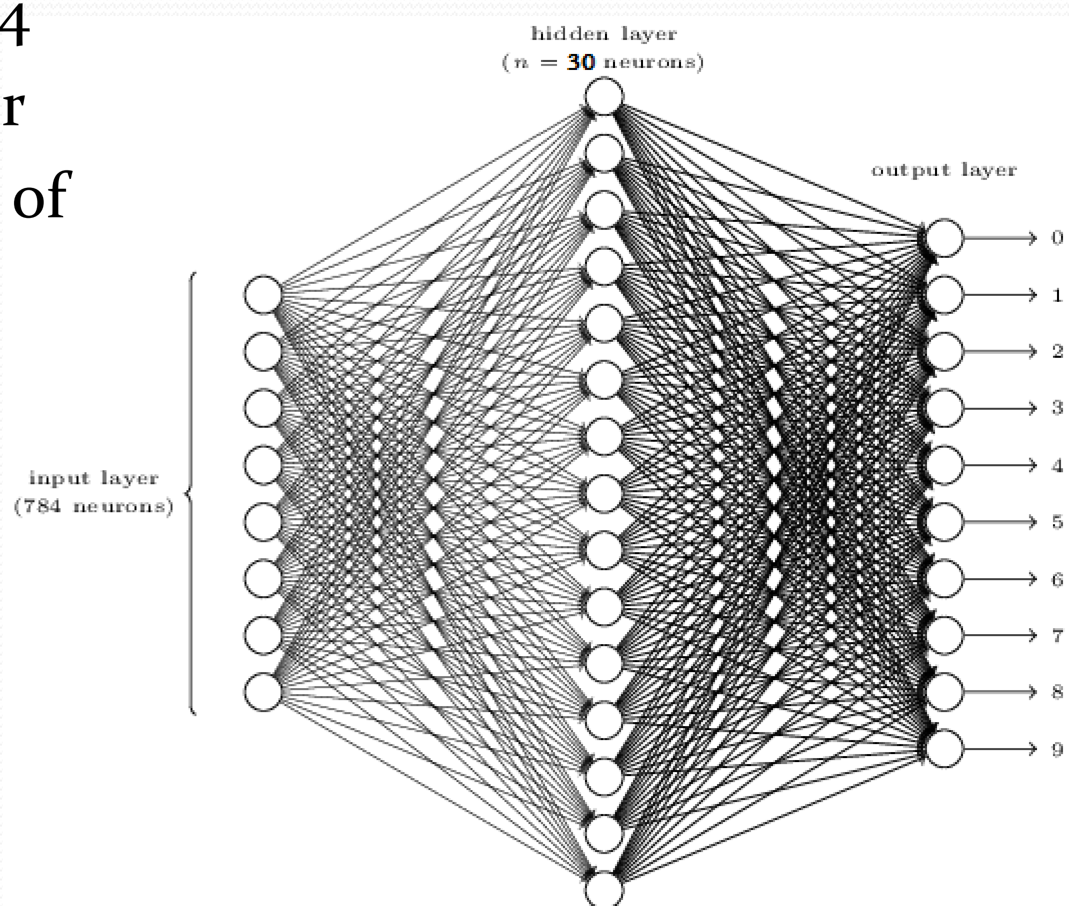
$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

One-hot vectorized label: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

Architecture of neural network

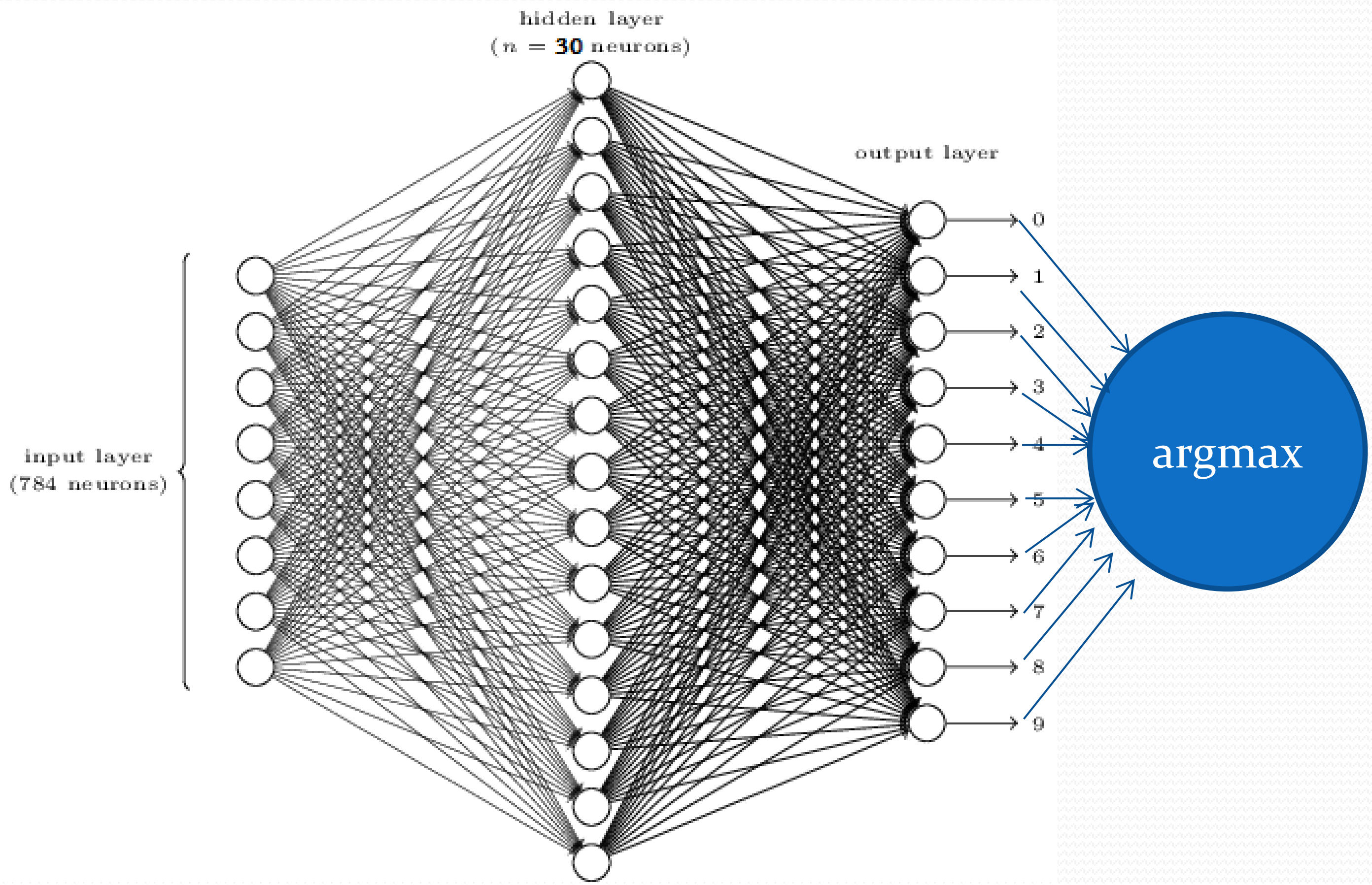
- Multilayer Perceptron (MLP) or fully connected NN
  - Input: 28x28=784
  - One hidden layer
  - Output: a vector of 10 elements





Architecture of neural network

- Multilayer perceptron (MLP)

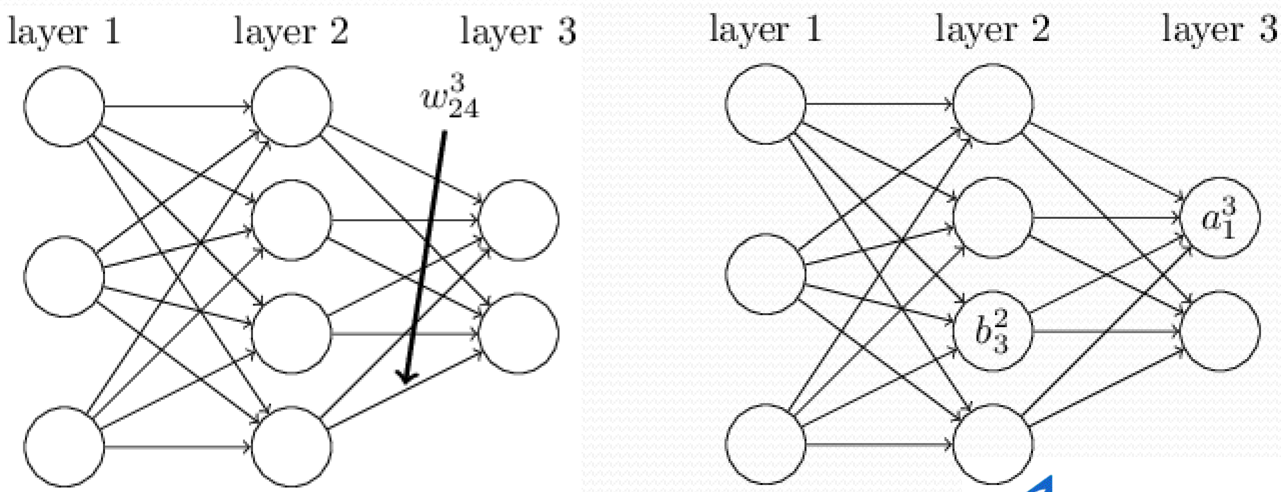


SHARCNET™

Cost function

- Some notations

$w_{j,k}^{(l)}$ denotes the weight of the connection between the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. Similarly, we define bias $b_j^{(l)}$, weighted sum+bias $z_j^{(l)}$, and activation $a_j^{(l)} = \sigma\left(z_j^{(l)}\right)$ at the j^{th} neuron in the l^{th} layer.



Cost function

- Given a input $(\boldsymbol{x}, \boldsymbol{y})$
  - Feedforward calculation

    $z_j^{(l)} = \sum_k w_{j,k}^{(l)} a_k^{(l-1)} + b_j^{(l)}$ or $\boldsymbol{z}^{(l)} = \mathbf{w}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$

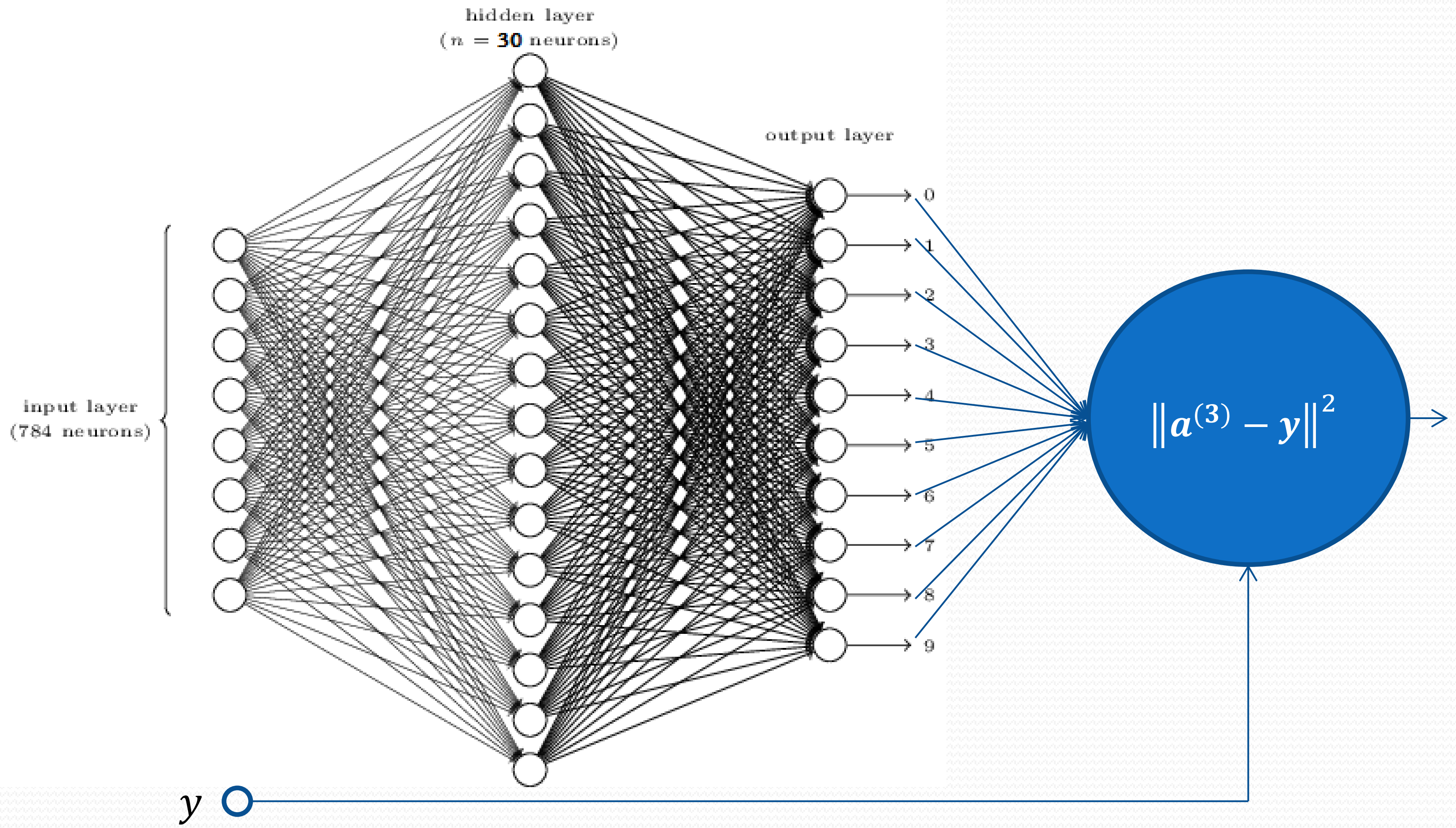
    $a_j^{(l)} = \sigma(z_j^{(l)})$ or $\boldsymbol{a}^{(l)} = \sigma\left(\boldsymbol{z}^{(l)}\right)$

    When $l = 1$, $a_j^{(1)} = x_j$
  - Cost function (quadratic function, MSE)

    $C = \frac{1}{m} \sum_x C_x$, where

    $C_x = \left\|\boldsymbol{a}^{(L)}(\boldsymbol{x}) - \boldsymbol{y}\right\|^2 = \sum_{j=1}^{10} \left(a_j^{(L)} - y_j\right)^2$, L is the number of layers of NN ($L = 3$ in this case).

As if there is an extra node



Backpropagation

- C depends (indirectly) on $\left\{w_{j,k}^{(l)}, b_j^{(l)}\right\}$ where

$$l \in \{2,3\}$$

$$j,k \in \begin{cases} \{[1{:}30],[1{:}784]\}, & \text{if } l=2 \\ \{[1{:}10],[1{:}30]\}, & \text{if } l=3 \end{cases}$$

- $\Delta C \approx \sum \frac{\partial C}{\partial w_{j,k}^{(l)}} \Delta w_{j,k}^{(l)} + \sum \frac{\partial C}{\partial b_j^{(l)}} \Delta b_j^{(l)}$
- If we can calculate $\frac{\partial C}{\partial w_{j,k}^{(l)}}$, $\frac{\partial C}{\partial b_j^{(l)}}$, then we will know how to change each of these parameters $\left\{w_{j,k}^{(l)}, b_j^{(l)}\right\}$ to make C smaller.

Backpropagation

- Loop over m samples, then calculate the average:
$$\frac{\partial C}{\partial w_{j,k}^{(l)}} = \frac{1}{m}\sum_x \frac{\partial C_x}{\partial w_{j,k}^{(l)}}, \quad \frac{\partial C}{\partial b_j^{(l)}} = \frac{1}{m}\sum_x \frac{\partial C_x}{\partial b_j^{(l)}}$$
- Let's define $\delta_j^{(l)} \equiv \frac{\partial C_x}{\partial z_j^{(l)}}$ be the error of neuron j on layer l, or
$\boldsymbol{\delta}^{(l)} = \left(\delta_1^{(l)}, \delta_2^{(l)}, \dots\right)^t$ be the error of layer l.
- Since $\boldsymbol{z}^{(l+1)} = \mathbf{w}^{(l+1)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l+1)} = \mathbf{w}^{(l+1)}\sigma\left(\boldsymbol{z}^{(l)}\right) + \boldsymbol{b}^{(l+1)}$, we can get
$$\boldsymbol{\delta}^{(l)} = \left(\left(\mathbf{w}^{(l+1)}\right)^t \boldsymbol{\delta}^{(l+1)}\right) \odot \sigma'\left(\boldsymbol{z}^{(l)}\right)$$

where $\odot$ is Hadamard product operator (element-wise multiplication)

- This means we can pass the error backward
$$\boldsymbol{\delta}^{(L)} \to \boldsymbol{\delta}^{(L-1)} \to \cdots \to \boldsymbol{\delta}^{(2)}$$

SHARCNET™

Backpropagation

- Then what if we know $\boldsymbol{\delta}^{(l)}$ …

$$\frac{\partial C_{\boldsymbol{x}}}{\partial b_j^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

$$\frac{\partial C_{\boldsymbol{x}}}{\partial w_{j,k}^{(l)}} = \delta_j^{(l)} \frac{\partial z_j^{(l)}}{\partial w_{j,k}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

or

$$\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$$

$$\frac{\partial C_{\boldsymbol{x}}}{\partial \mathbf{w}^{(l)}} = \boldsymbol{\delta}^{(l)} \left(\boldsymbol{a}^{(l-1)}\right)^t$$

Backpropagation

Backpropagation starts at layer $L = 3$

$$\delta_j^{(L)} = \frac{\partial C_x}{\partial a_j^{(L)}} \sigma'\left(z_j^{(L)}\right)$$

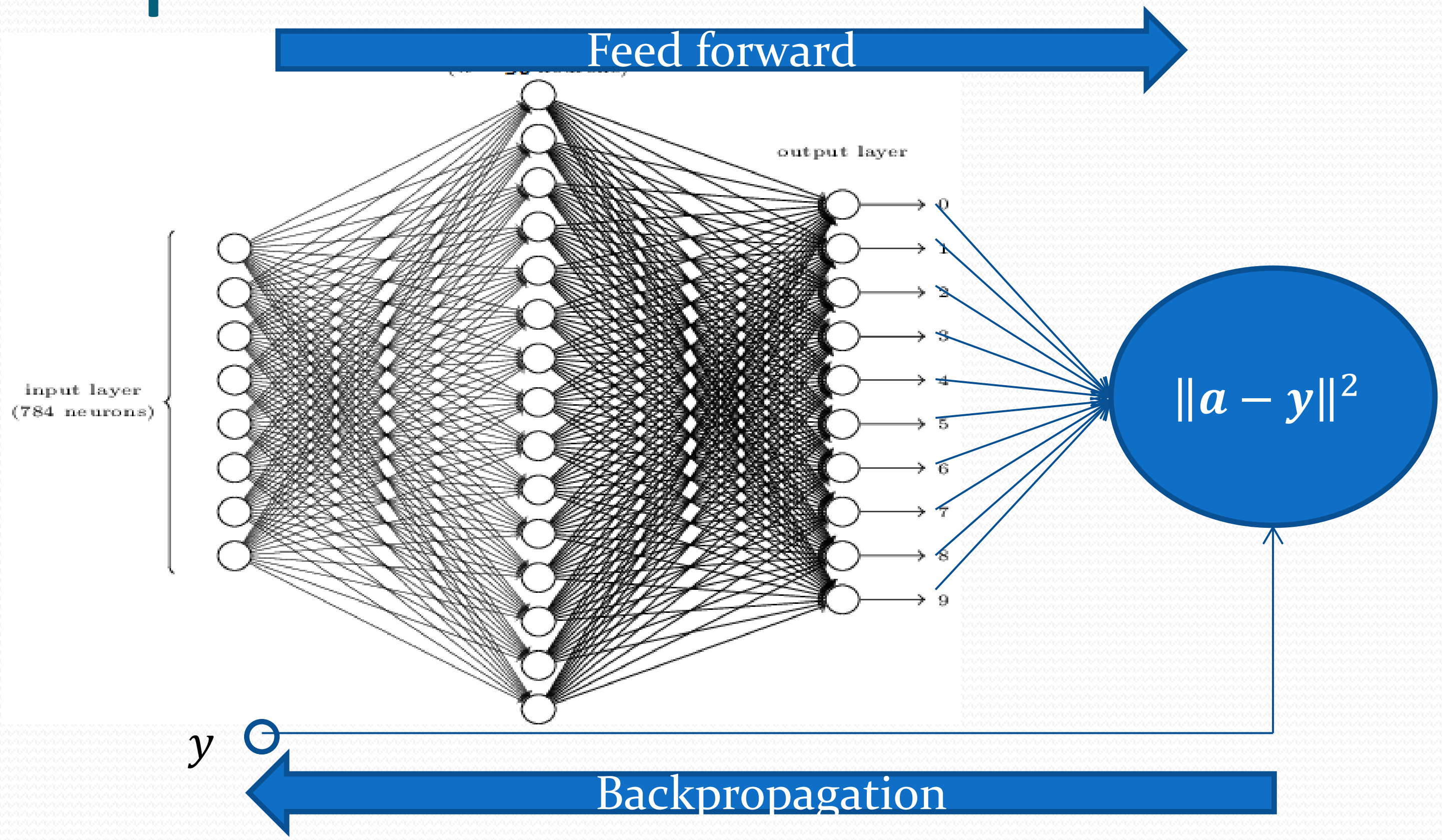
where

$$\frac{\partial C_x}{\partial a_j^{(L)}} = 2\left(a_j^{(L)} - y_j\right)$$

Or

$$\delta^{(L)} = \left(a_j^{(L)} - y_j\right) \odot \sigma'\left(z^{(L)}\right)$$

Two passes



$$\boldsymbol{\delta}^{(2)} \leftarrow \boldsymbol{\delta}^{(3)}$$

$$\downarrow \qquad\qquad \downarrow$$

$$\frac{\partial C_{\boldsymbol{x}}}{\partial \mathbf{w}^{(2)}}, \frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{b}^{(2)}} \qquad \frac{\partial C_{\boldsymbol{x}}}{\partial \mathbf{w}^{(3)}}, \frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{b}^{(3)}}$$

Write our own NN code

- Assuming 30 neurons in the hidden layer

Architecture:

```
self.num_layers = 3
self.sizes = [784, 30, 10]
```

Parameters:

```
self.weights = [[30x784], [10x30]]
self.biases = [30, 10]
```

We need to determine the values of these

30x784 + 10x30 + 30 + 10 = 23860 parameters

through a training process.

How?

In each iteration of the training process

- Feedforward
- Backpropagation

Training

- train.py

```
import mnist_loader
import network

# load reformatted data.
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()

# define the NN
net = network.Network([784, 30, 10])

# train with Stochastic Gradient Descent
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

NOTE:
training_data is a list of 50000 tuples, each of which is ([784], [10])

Class “Network” (network.py)

- __init__(self, sizes) initialize parameters with random numbers
- feedforward(self, a) takes ‘a’ as input and return the output of the NN.
- evaluate(self, test_data) evaluates the performance of the NN.
- SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None)
  - update_mini_batch(self, mini_batch, eta)
    - twopasses(self, x, y)

SHARCNET™

Two helper functions

```
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))
```

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

```
def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Functions of Network

```
def feedforward(self, x):
        for b, w in zip(self.biases, self.weights):
            x = sigmoid(np.dot(w, x)+b)
        return x

def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y) \
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

NOTE:
self.weights = [[30,784], [10, 30]]
self.biases = [30, 10]
$\boldsymbol{x}$=[784] is passed in as argument

In loop #1:

- $\boldsymbol{w}^t = [30, 784]$, $\boldsymbol{b} = [30]$
- $\boldsymbol{x} = \sigma(\boldsymbol{w}^t\boldsymbol{x} + \boldsymbol{b}) \rightarrow \boldsymbol{x}$=[30]

In loop #2:

- $\boldsymbol{w}^t = [10, 30]$, $\boldsymbol{b} = [10]$
- $\boldsymbol{x} = \sigma(\boldsymbol{w}^t\boldsymbol{x} + \boldsymbol{b}) \rightarrow \boldsymbol{x}$=[10]

Functions of Network

```
def SGD(self, training_data, epochs, mini_batch_size,
        eta, test_data=None):
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

NOTE:
epochs=30, mini_batch_size = 10
eta (or η)=3.0 learning rate

Generate 5000 randomized mini-batches

Update weights and biases by learning from each batch

If test data is provided, then evaluate the performance of the current NN

Functions of Network

```
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.twopasses(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

∇C_i

$\sum_{i=1}^{m} \nabla C_i$

$(\mathbf{w}, \boldsymbol{b}) = (\mathbf{w}, \boldsymbol{b}) + \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_i$

$$\text{nabla\_w}=\left(\begin{bmatrix} \frac{\partial C}{\partial w^{(2)}_{1,1}} & \cdots & \frac{\partial C}{\partial w^{(2)}_{1,784}} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w^{(2)}_{30,1}} & \cdots & \frac{\partial C}{\partial w^{(2)}_{30,784}} \end{bmatrix}, \begin{bmatrix} \frac{\partial C}{\partial w^{(3)}_{1,1}} & \cdots & \frac{\partial C}{\partial w^{(3)}_{1,30}} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w^{(3)}_{10,1}} & \cdots & \frac{\partial C}{\partial w^{(3)}_{10,30}} \end{bmatrix}\right)$$

$$\text{nabla\_b}=\left(\left[\frac{\partial C}{\partial b^{(2)}_{1}}, \dots, \frac{\partial C}{\partial b^{(2)}_{30}}\right], \left[\frac{\partial C}{\partial b^{(3)}_{1}}, \dots, \frac{\partial C}{\partial b^{(3)}_{10}}\right]\right)$$

Functions of Network

```
def twopasses(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = (activations[-1] - y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in xrange(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

∇C_i that depends on a single sample (x, y)

Feedforward pass:

$z = \boldsymbol{w}^t \boldsymbol{x} + b \rightarrow$ zs[]

$\sigma(z) = \frac{1}{1+e^{-z}} \rightarrow$ activations[]

Backpropagation pass:

delta= $\delta^{(L)} = \frac{\partial C_x}{\partial \boldsymbol{a}^{(L)}} \odot \sigma'(z^{(L)})$

Nabla_b[-1] $= \frac{\partial C_x}{\partial \boldsymbol{b}^{(L)}} = \delta^{(L)}$

Nabla_w[-1]$= \frac{\partial C_x}{\partial \mathbf{w}^{(L)}} = \boldsymbol{\delta}^{(L)}\left(\boldsymbol{a}^{(L-1)}\right)^t$

delta = $\boldsymbol{\delta}^{(l)} = \left(\left(\mathbf{w}^{(l+1)}\right)^t \boldsymbol{\delta}^{(l+1)}\right) \odot \sigma'\left(\boldsymbol{z}^{(l)}\right)$

Calculate nabla_w, nabla_b for layer 2 from $\boldsymbol{\delta}^{(l)}$

Tweak around

- Learning rate: 0.001, 1.0, 100.0, …
- Size of mini-batches: 10, 50, 100, …
- Number of neurons in the hidden layer: 15, 30, 100, …
- Number of hidden layers: 1, 2, 5, …

Use Tensorflow in recognition of handwritten digits

- Introduction to Tensorflow
- A warm-up
- A 2-layer NN (~92% recognition rate)
- A 3-layer NN (~94% recognition rate)
- A much better NN (~99% recognition rate)

Introduction to Tensorflow

- Tensorflow APIs
  - Low-level APIs --- Tensorflow Core that gives you a fine control
  - High-level APIs --- built upon the Core, which are more convenient and efficient to program with
- Tensor --- a multi-dimensional array which is the central unit of data structure, e.g., [batch, height, width, channel] for image data.
- Session --- encapsulation of the control and the state of Tensorflow runtime.

Introduction to Tensorflow

- To perform a specific computational task, one needs to
  - Build a computational graph
  - Run the computational graph
- Computational graph is a directed graph with edges connecting nodes specifying the data flow. A node could be
  - A constant (no input)
  - A variable
  - A placeholder (reserved for input data)

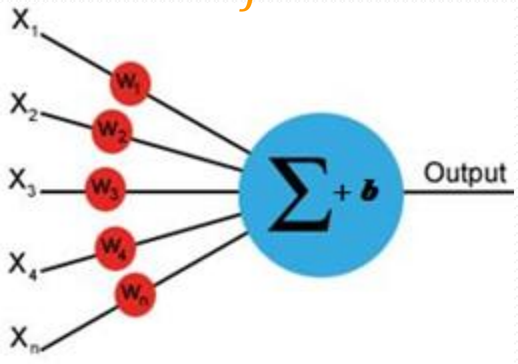
A warm-up exercise (tf_warmup.py)

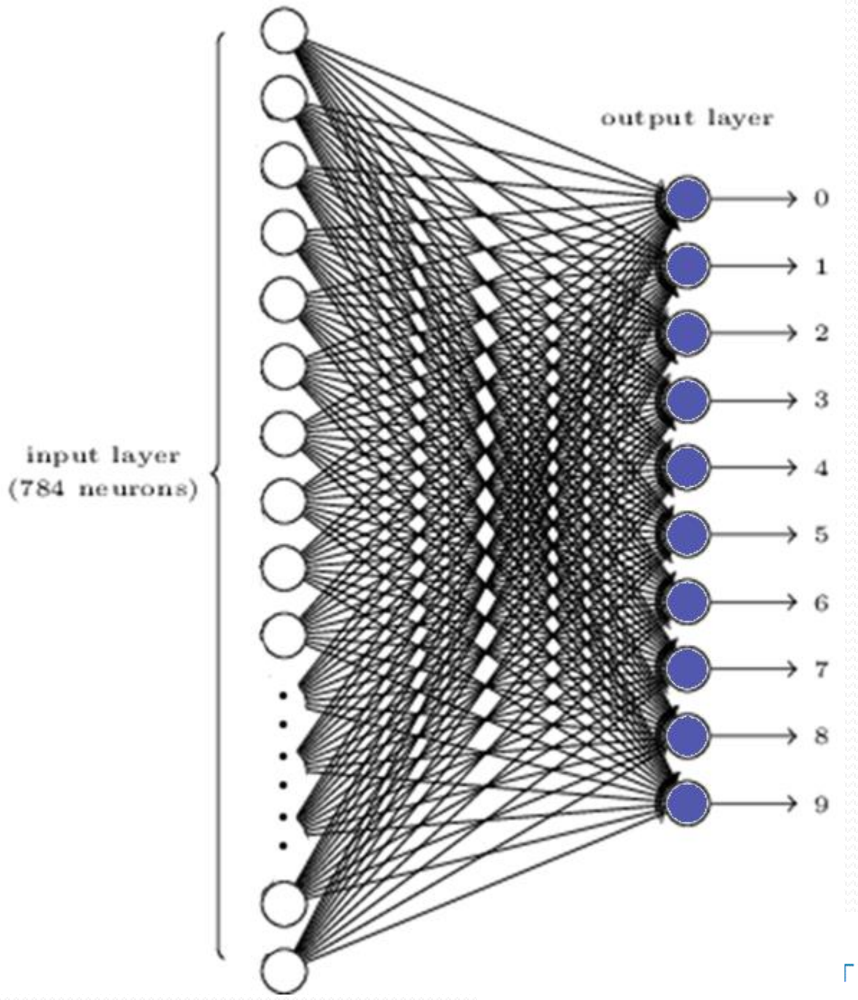
- Problem:
  - There is a linear model $y = w \cdot x + b$, where w, b are parameters of the model
  - Given a set of training data $\{(x_i, y_i)\}_i = \{(1,0), (2,-1), (3,-2), \dots\}$, we need to find the values of w, b so that the model "best" fits into the data
- The criteria of "best fit" is minimization of a loss function $C = \frac{1}{n}\sum_i (wx_i + b - y_i)^2$

A simple method (tf_mnist_2layers.py)

- A two-layer fully-connected network
  - Input: 28x28=784
  - Output: a vector of 10 elements

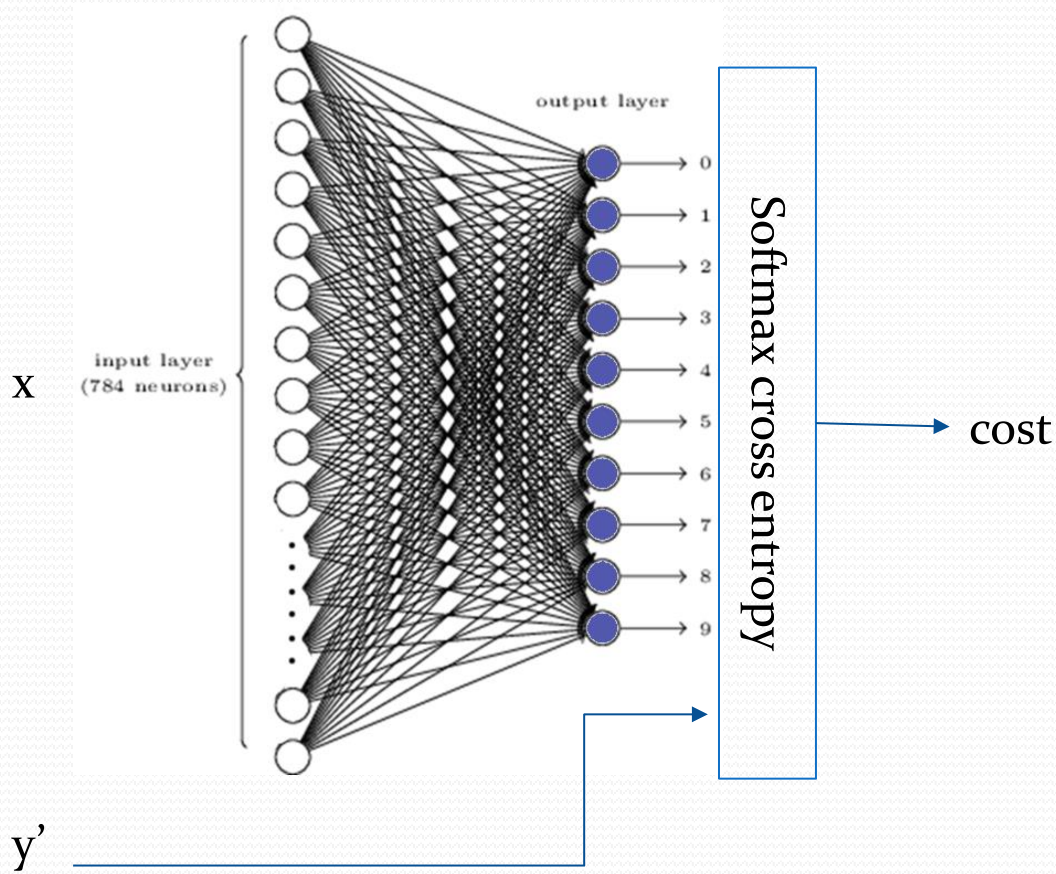
$$y_i = \sum_j w_{ij} x_j + b_i$$





A simple method (cont.)

- Training



Softmax cross entropy

- Softmax$(y_1, y_2, \ldots, y_n)$= $\frac{1}{\sum_i e^{y_i}}(e^{y_1}, e^{y_2}, \ldots, e^{y_n})$ turns a vector of outputs into a probability distribution
- Cross entropy between a prediction probability distribution $\boldsymbol{y}$ and a true distribution $\boldsymbol{y}'$ is defined as

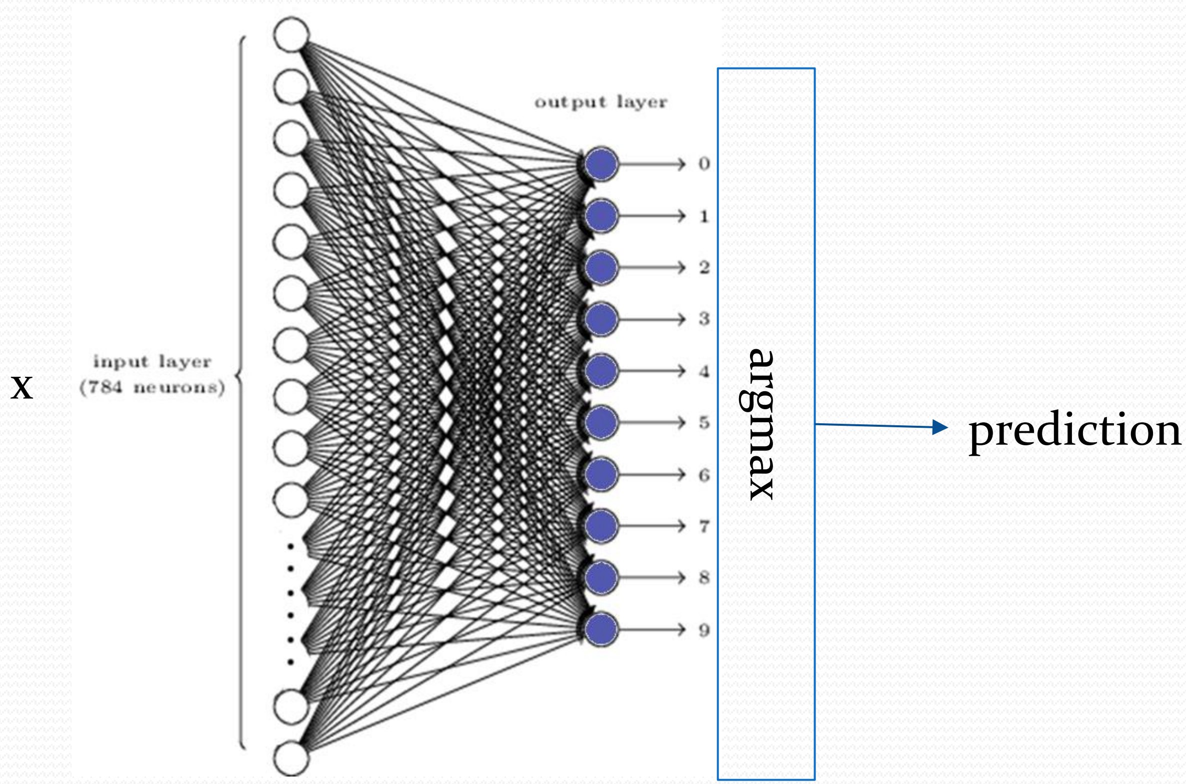
$$H_{y'}(\boldsymbol{y}) = -\sum_i y_i' log(y_i)$$

Since $0 \leq y_i, y_i' \leq 1$, so $H_{y'}(y) \geq 0$

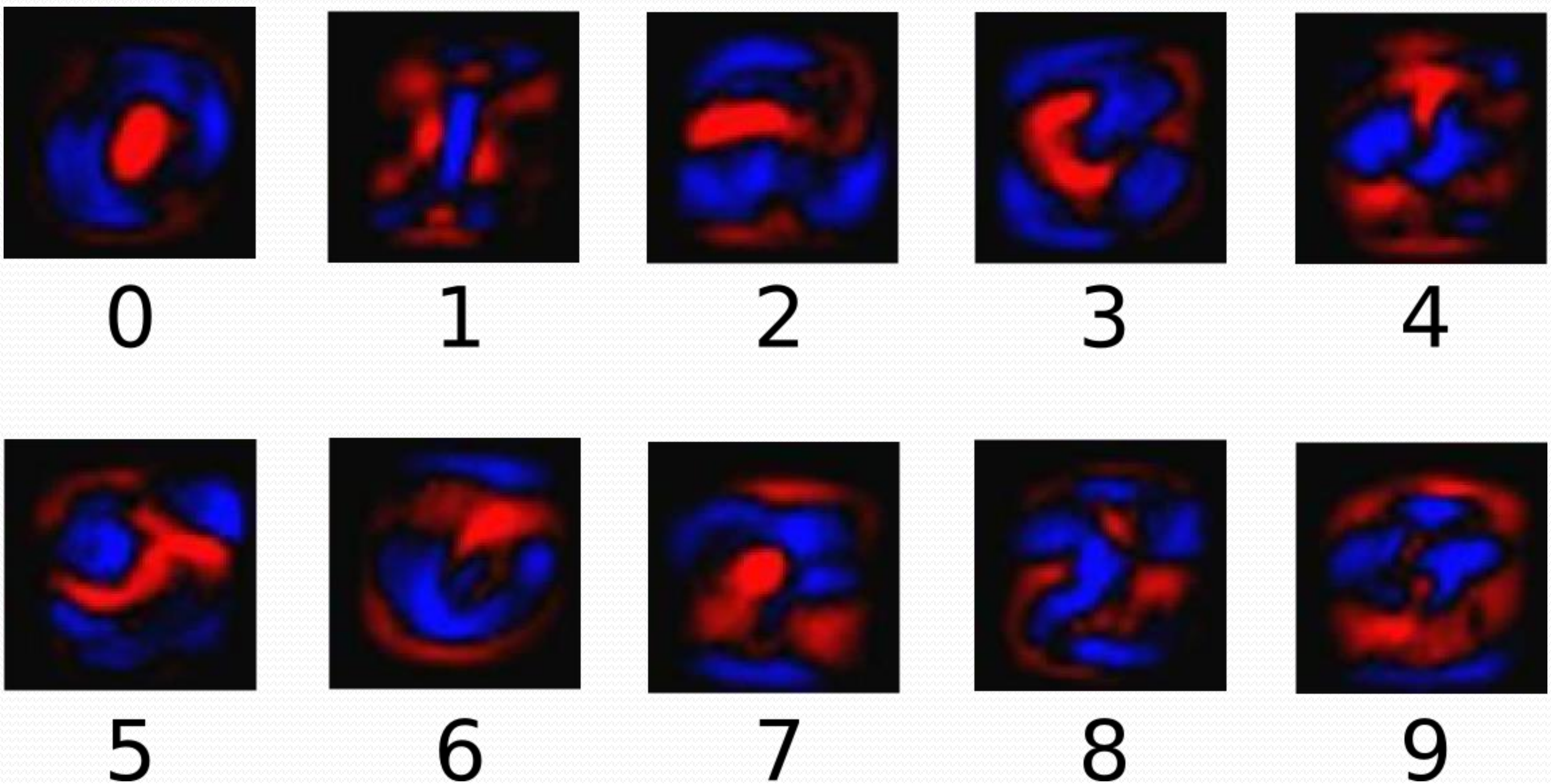
- **Tf.nn.softmax_cross_entropy_with_logits**
  - $\boldsymbol{y}\_$ =Softmax$(\boldsymbol{y})$
  - $H_{y'}(\boldsymbol{y}\_)$

A simple method (cont.)

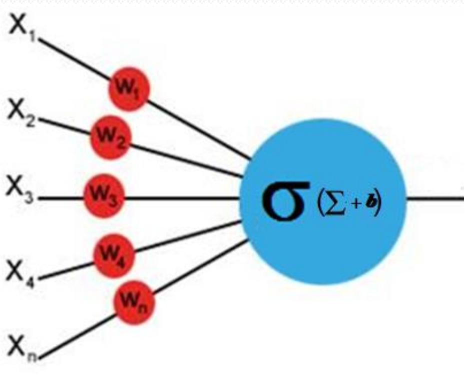
- Use trained model

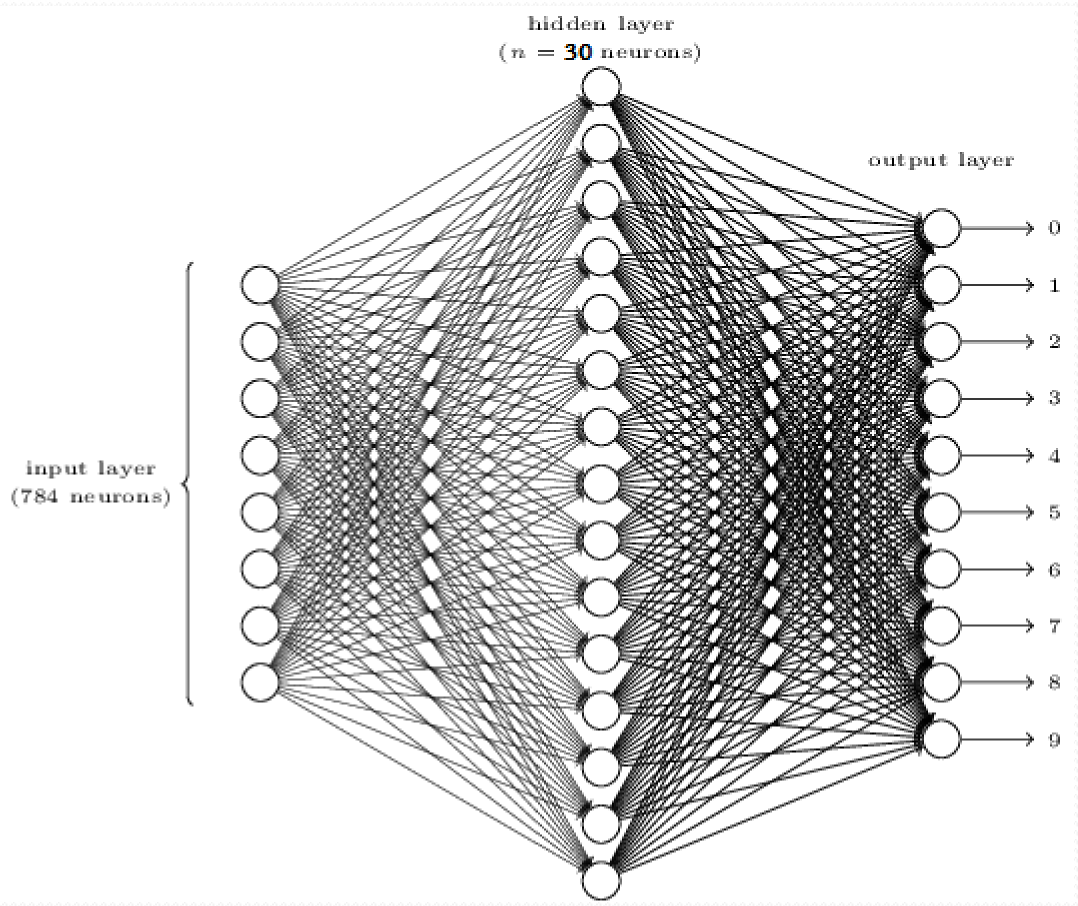


What’s learned



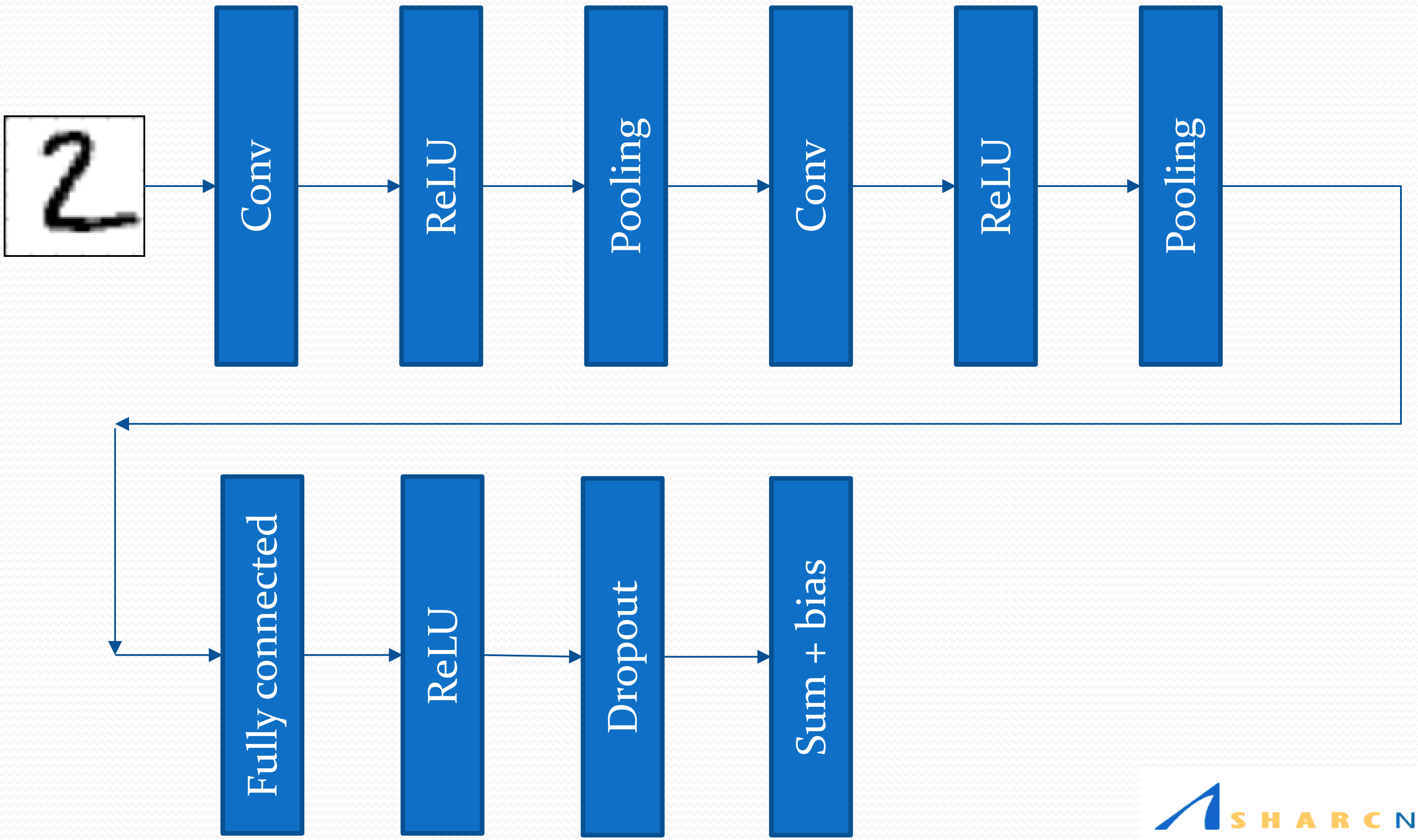
Add a hidden layer (tf_mnist_3layers_*.py)





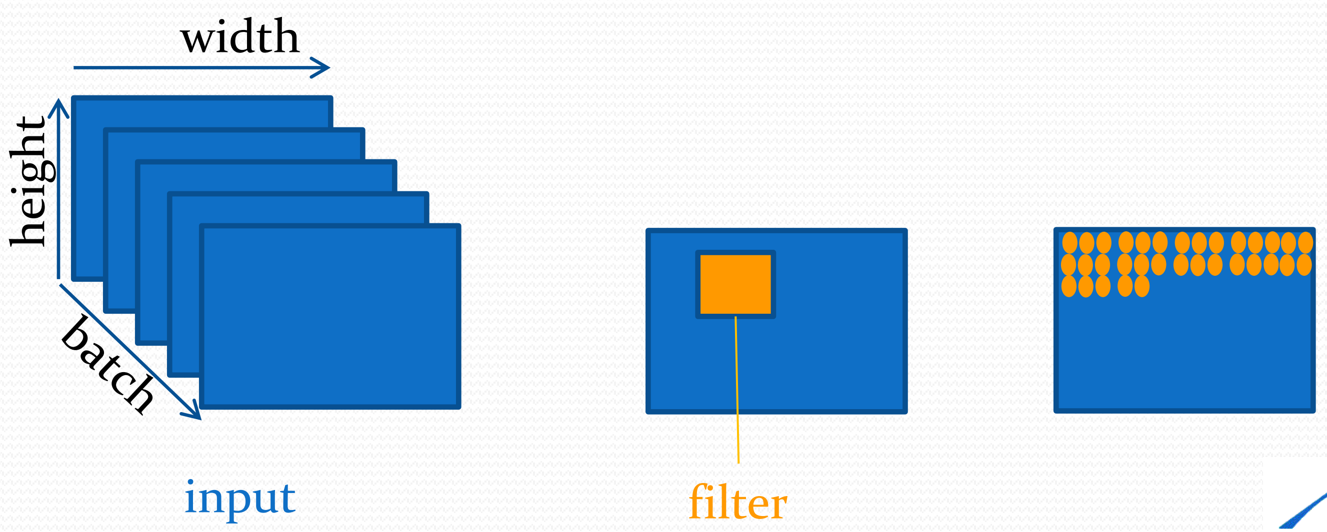
A much better NN

- Multilayer convolutional network



Some functions in TF

- tf.nn.conv2d(input, filter, strides, padding, ...)
  - input: 4D tensor [batch, height, width, channels]
  - filter: 4D tensor [f_height, f_width, in_channels, out_channels]
  - strides: 4D tensor
  - padding: “SAME’ or ‘VALID”



Discussions

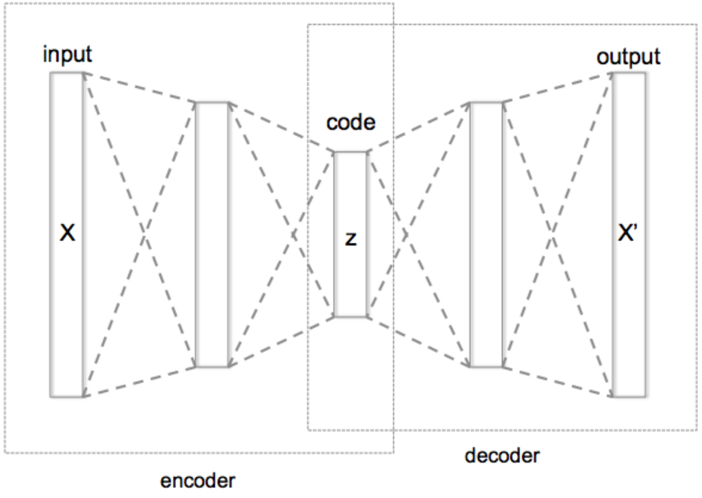
- Generalization, overfitting, regularization
- Difference between deep NN and traditional NN
- Why deep NN works much better

Generalization, overfitting, regularization

- Goal of machine learning: Generalization, i.e., learning general rules/patterns from training data)
- Pitfall: overfitting, i.e., a learned model performs much worse on unseen data
- Mechanism to prevent overfitting: regularization
  - Dropout layer
  - Monitoring performance with evaluation data during training process

Difference between deep NN and traditional NN

- Deeper: more hidden layers
- Combination of unsupervised and supervised learning (autoencoder, a generative NN)
- Better generalization and regularization mechanisms
- More advanced layers/neurons: convolutional, pooling, …



Why is deep NN so successful?

- It can approximate arbitrary functions well
- Features are extracted in a hierarchical way
  - Features extracted in lower layers are more concrete and local
  - Features extracted in higher layers are more abstract and global
- Deep NN and cheap learning