UNSW SYDNEY | Australia's Global University

COMP9321

Data Services Engineering

Term 1, 2019

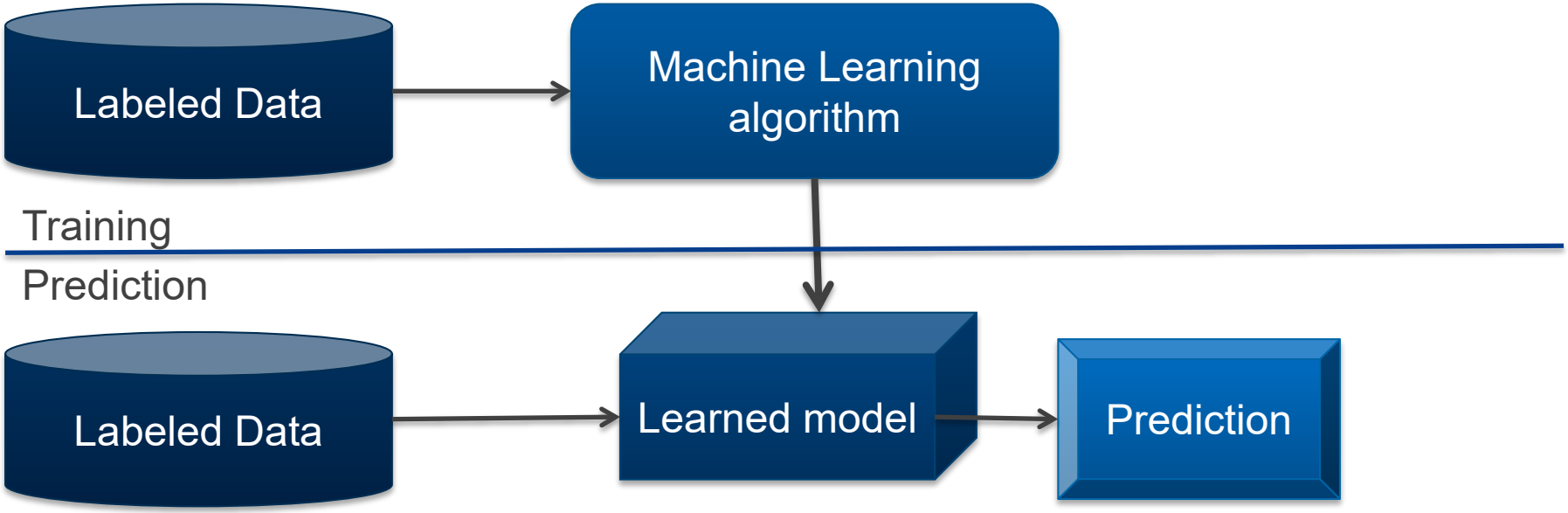
Week 7 Lecture 1

Introduction to Neural Networks

COMP9321 2019T1

Recall

Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed

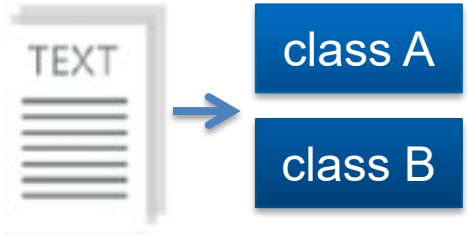


Methods that can learn from and make predictions on data

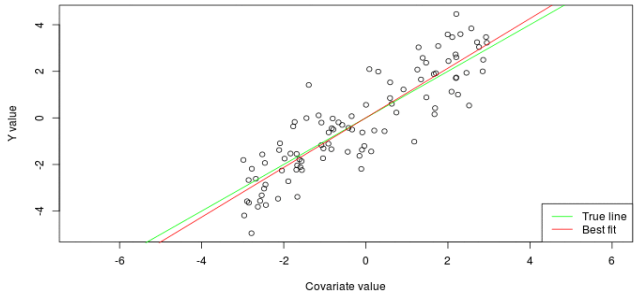
Recall

Supervised: Learning with a labeled training set
Example: email classification with already labeled emails

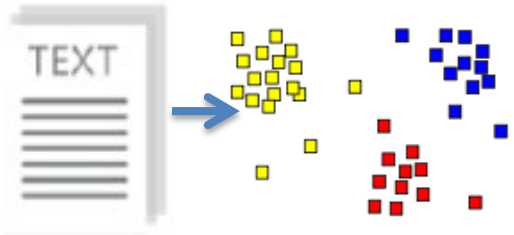
Unsupervised: Discover patterns in unlabeled data
Example: cluster similar documents based on text



Classification



Regression



Clustering

What is Deep Learning (DL) ?

Artificial Intelligence

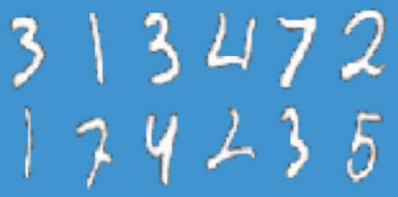
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicitly being programmed

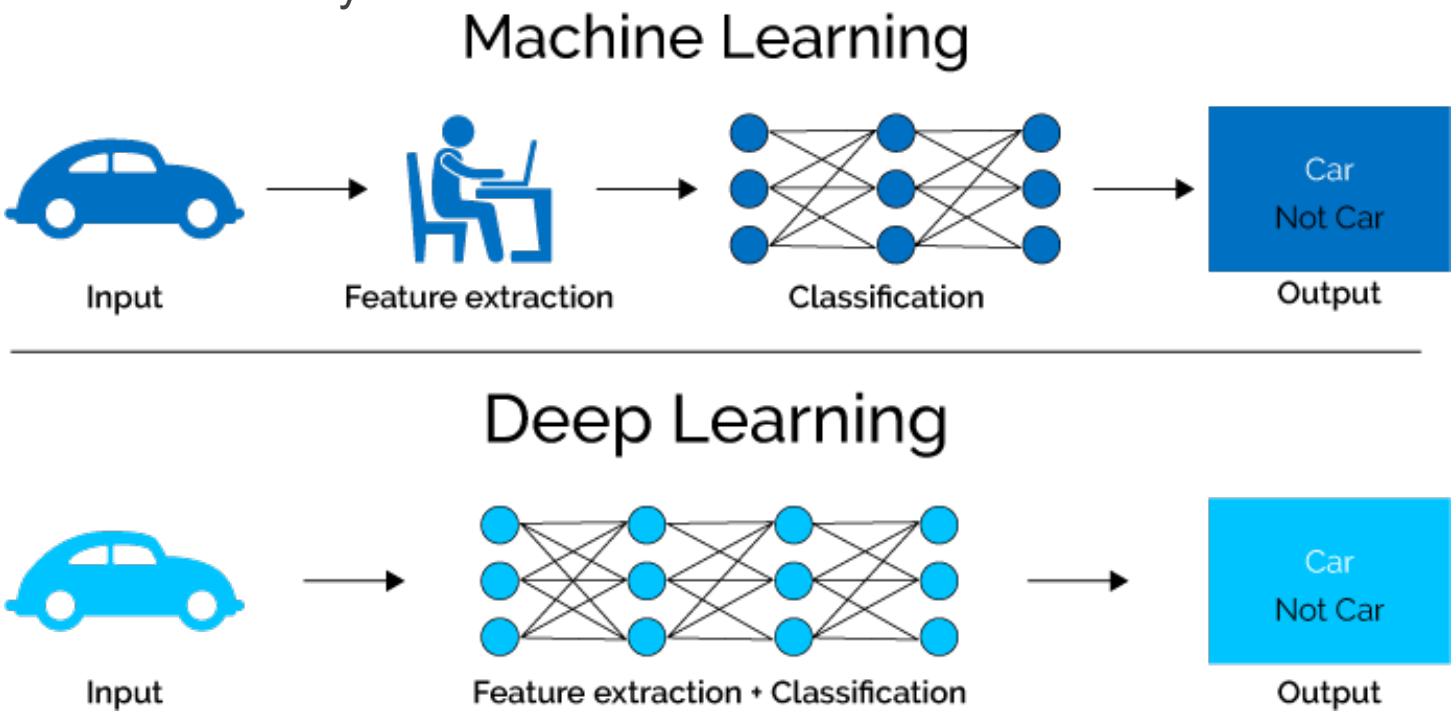
Deep Learning

Extract patterns from data using neural networks



What is Deep Learning (DL) ?

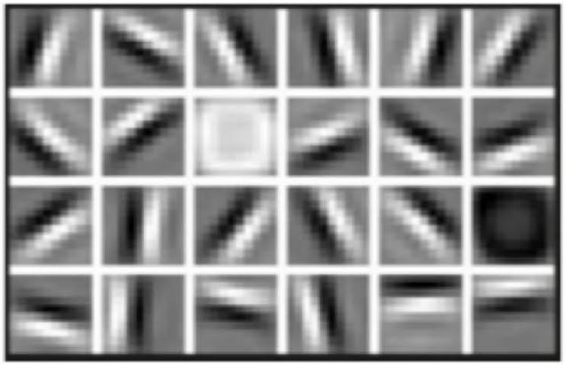
A machine learning subfield of learning representations of data. Exceptional effective at learning patterns.
Deep learning algorithms attempt to learn (multiple levels of) representation by using a hierarchy of multiple layers
If you provide the system tons of information, it begins to understand it and respond in useful ways.



https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

What is Deep Learning (DL) ?

Low Level Features



Lines & Edges

Mid Level Features

Eyes & Nose & Ears

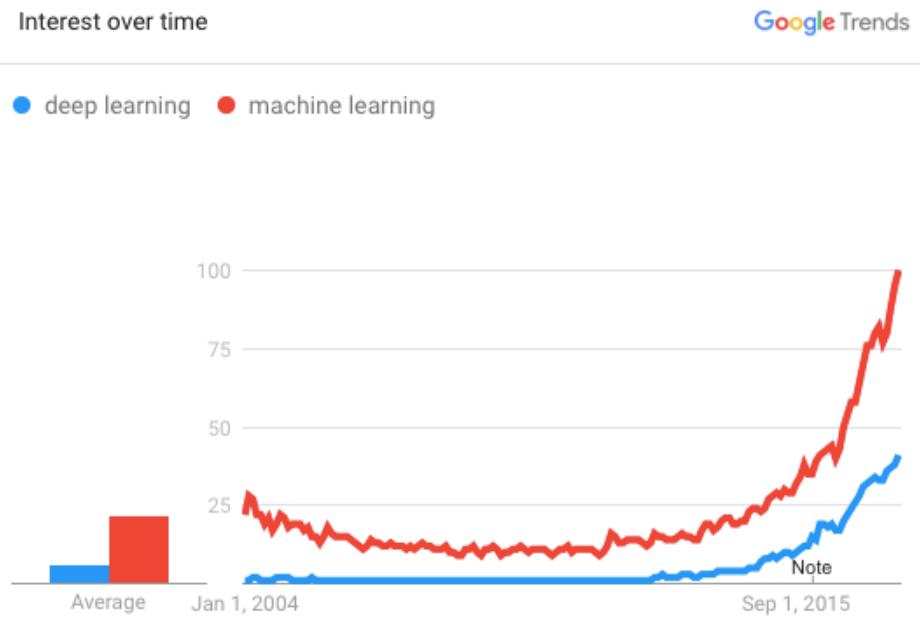
High Level Features

Facial Structure

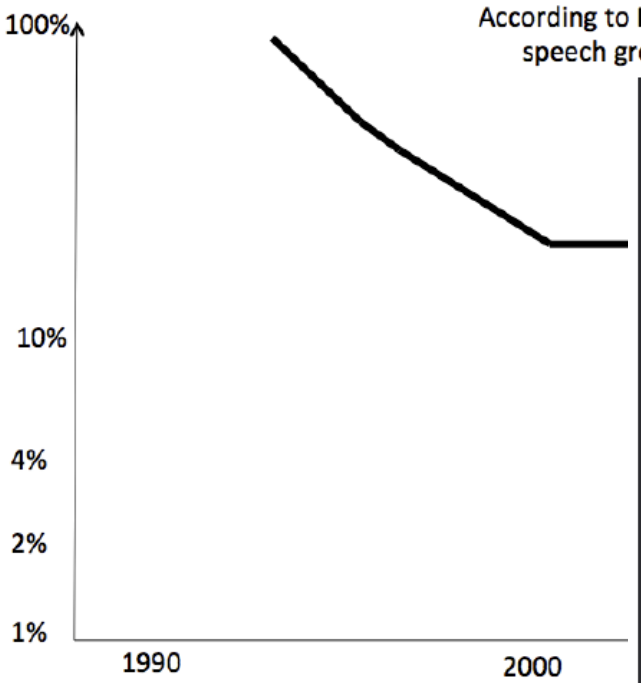
Why is DL useful?

- Manually designed features are often over-specified, incomplete and take a long time to design and validate
- Learned Features are easy to adapt, fast to learn
- Deep learning provides a very flexible, (almost?) universal, learnable framework for representing world, visual and linguistic information.
- Can learn both unsupervised and supervised
- Effective end-to-end joint system learning
- Utilize large amounts of training data

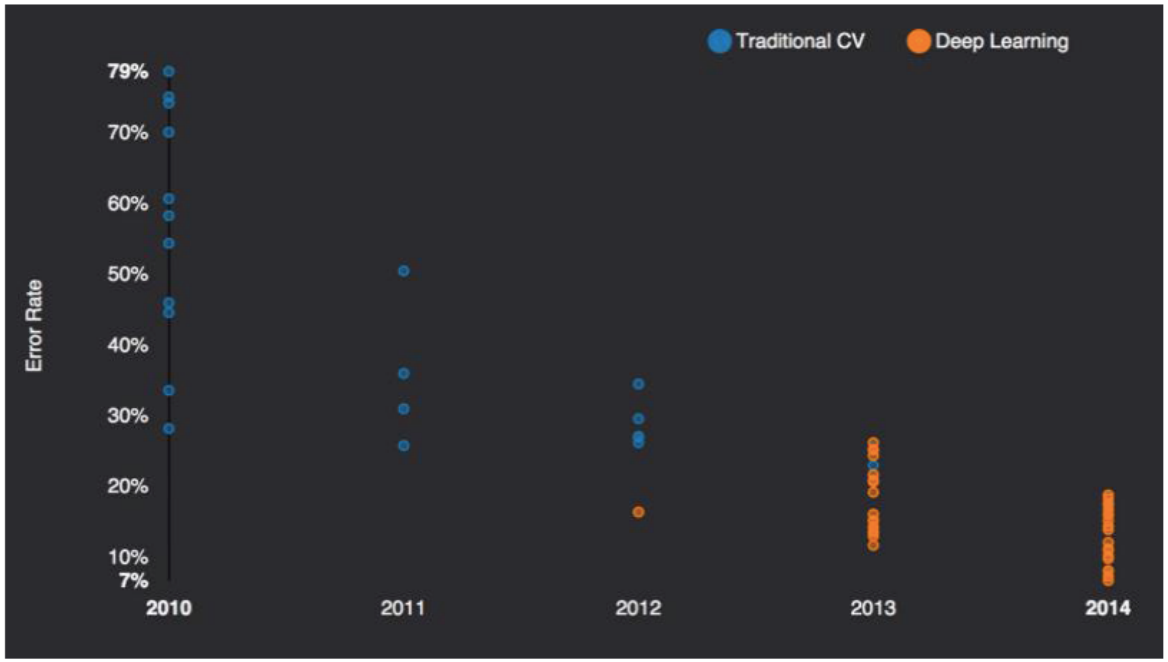
In ~2010 DL started outperforming other ML techniques
first in speech and vision, then NLP



State of the art in …



Deep Learning in Speech Recognition



ImageNet: The "computer vision World Cup"

Several big improvements in recent years in NLP

- ✓ Machine Translation
- ✓ Sentiment Analysis
- ✓ Dialogue Agents
- ✓ Question Answering
- ✓ Text Classification …

Leverage different levels of representation

- words & characters
- syntax & semantics

Why NOW…

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

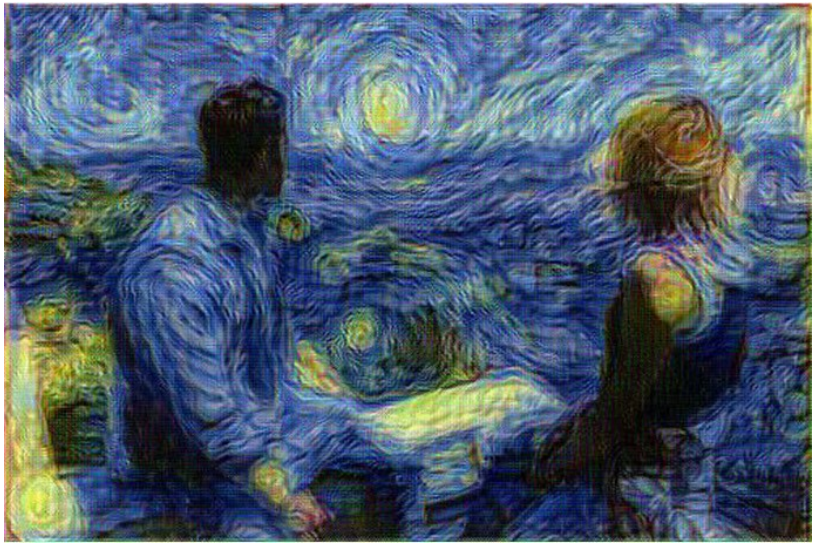
WIKIPEDIA The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

Cat Dog



2019 Turing Award

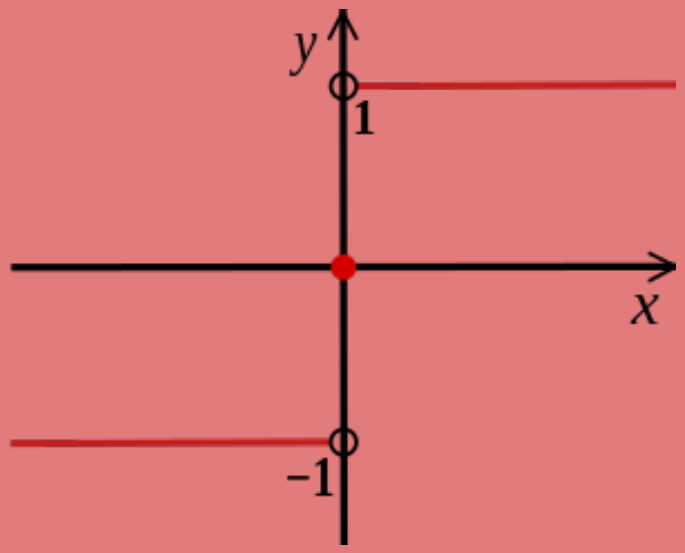

Deep Learning

Using gradient ascent for linear classifiers

Recall…

This decision function isn't differentiable:

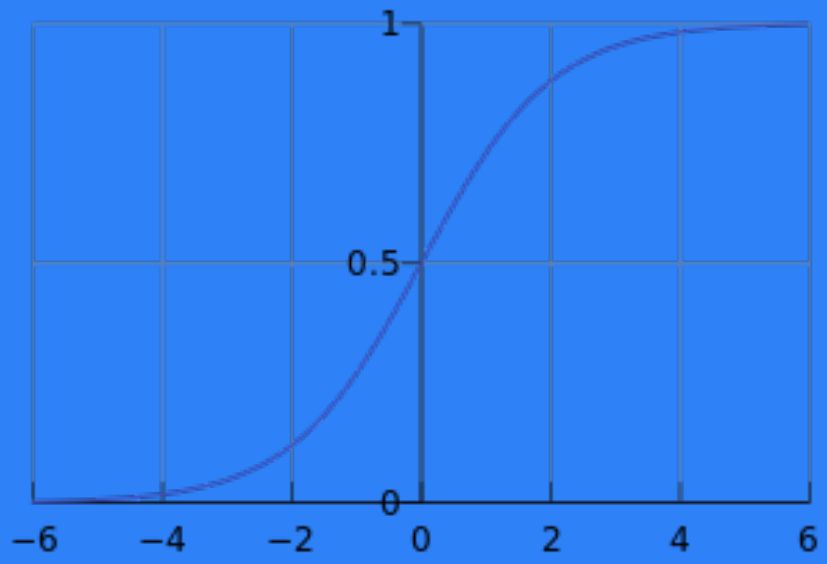
$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



sign(x)

Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$



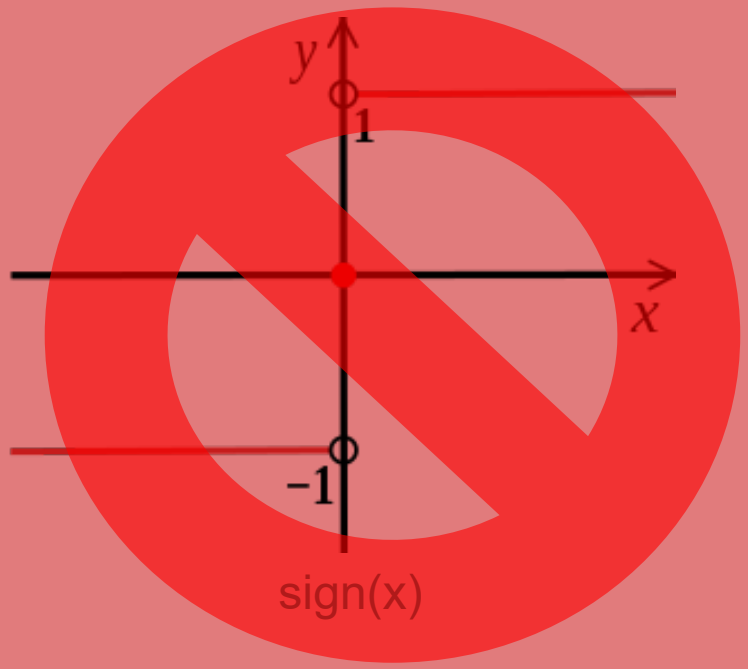
$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Using gradient ascent for linear classifiers

Recall…

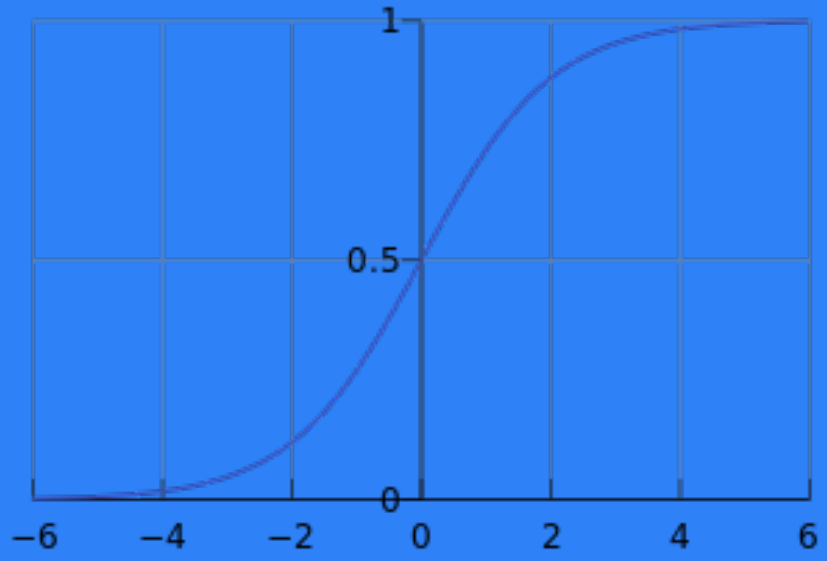
This decision function isn't differentiable:

$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$



$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Recall…

Data: Inputs are continuous vectors of length K. Outputs are discrete.

$$\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N} \text{ where } \mathbf{x} \in \mathbb{R}^K \text{ and } y \in \{0, 1\}$$

Model: Logistic function applied to dot product of parameters with input vector.

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

Learning: finds the parameters that minimize some objective function. $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$

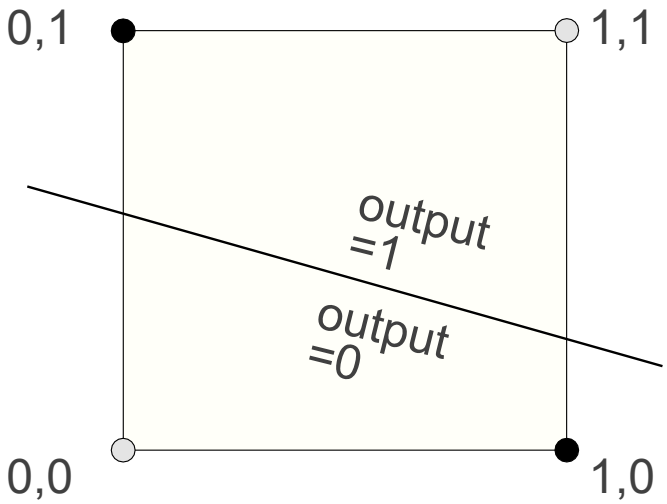
Prediction: Output is the most probable class.

$$\hat{y} = \underset{y \in \{0,1\}}{\operatorname{argmax}}\, p_{\boldsymbol{\theta}}(y|\mathbf{x})$$

Limitations of Linear Classifiers

Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features xi

Many decisions involve non-linear functions of the input Canon



The positive and negative cases cannot be separated by a plane

What can we do?

Limitations of Linear Classifiers

We would like to construct non-linear discriminative classifiers that utilize functions of input variables

Use a large number of simpler functions

If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs

Or we can make these functions depend on additional parameters → need an efficient method of training extra parameters

Inspiration: The Brain

Many machine learning methods inspired by biology, e.g., the (human) brain

Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^{4}$ other neurons

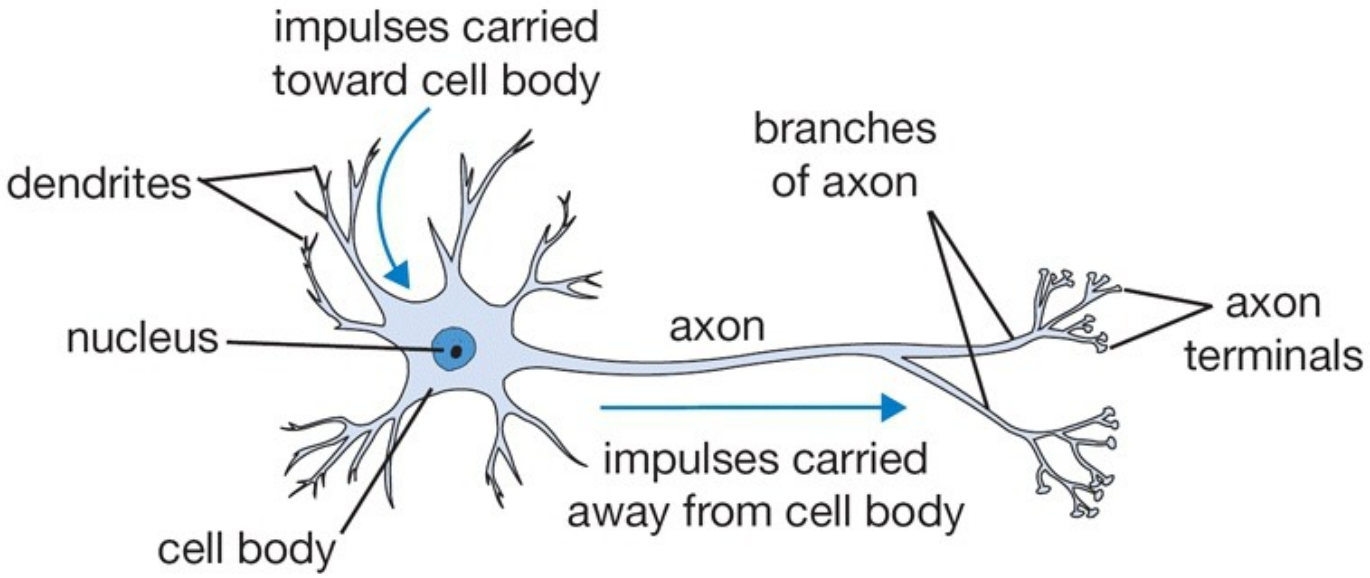


Figure : The basic computational unit of the brain: Neuron

[Pic credit: http://cs231n.github.io/neural-networks-1/]

Mathematical Model of a Neuron: Perceptron

Neural networks define functions of the inputs (hidden features), computed by neurons

Artificial neurons are called units

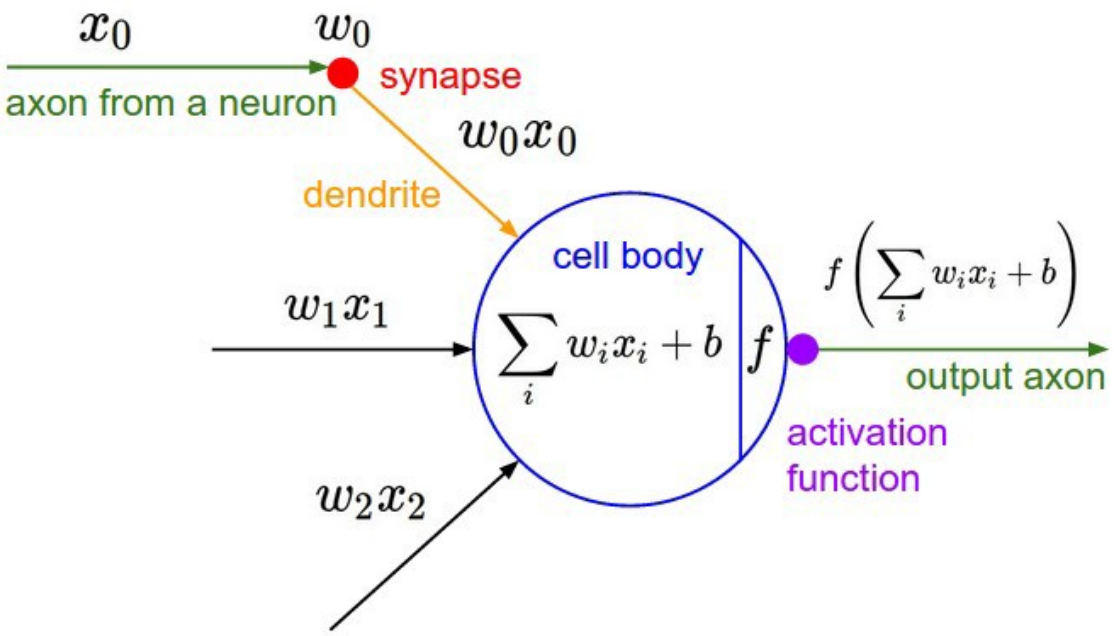


Figure : A mathematical model of the neuron in a neural network

[Pic credit: http://cs231n.github.io/neural-networks-1/]

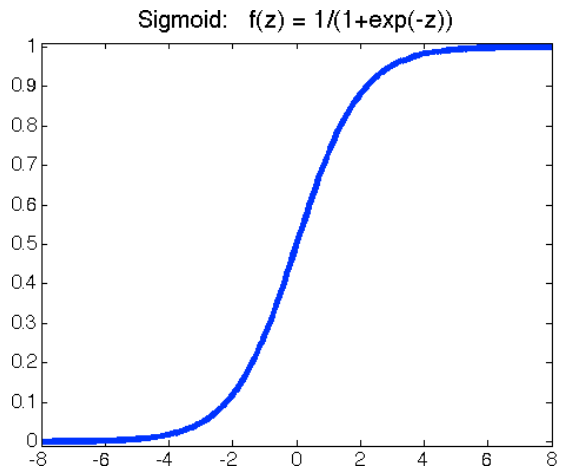
Activation Functions

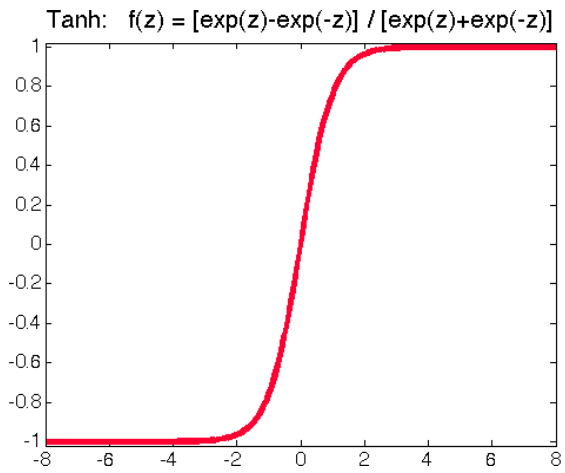
Most commonly used activation functions:

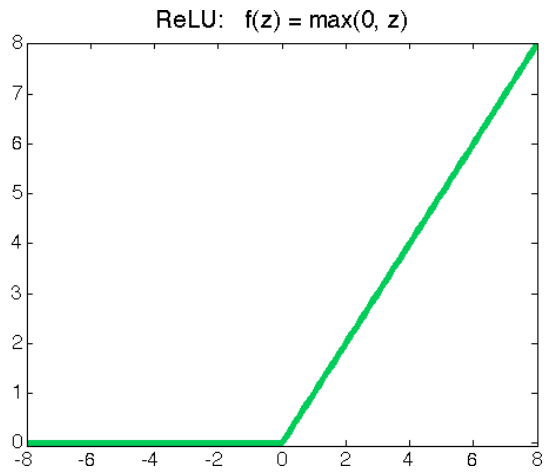
Sigmoid:

Tanh:

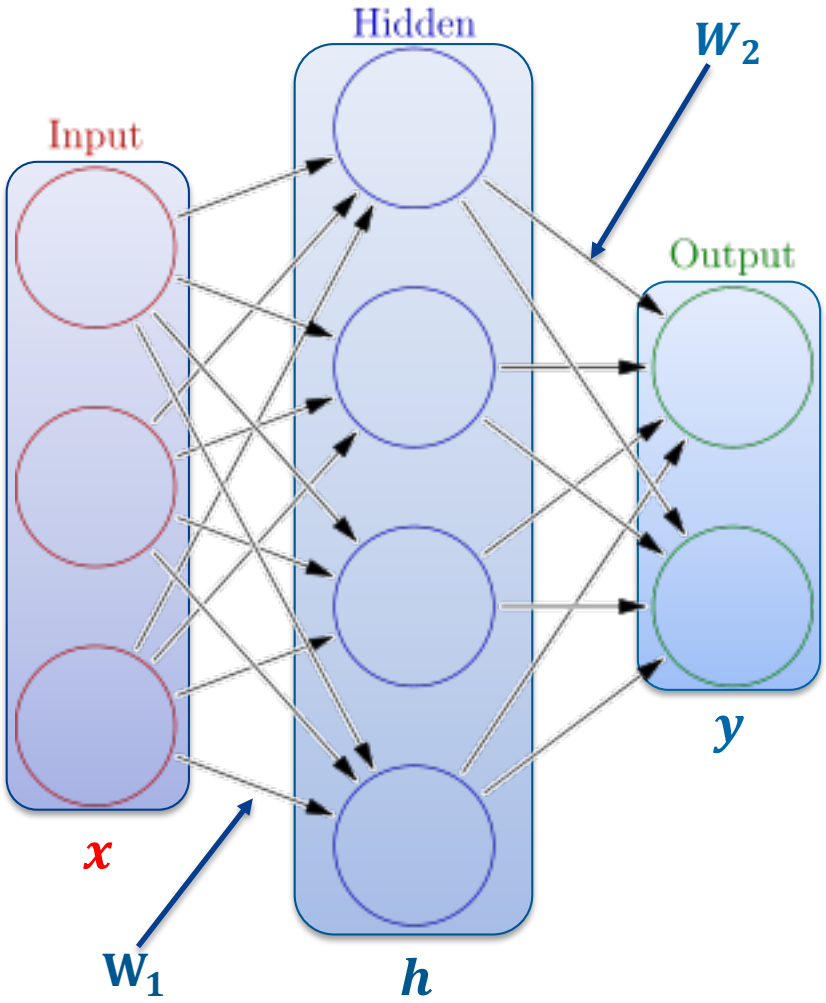
ReLU (Rectified Linear Unit): ReLU(z)







Single-Layer Neural Network



Weights

$$h = \sigma(W_1 x + b_1)$$

$$y = \sigma(W_2 h + b_2)$$

Activation functions

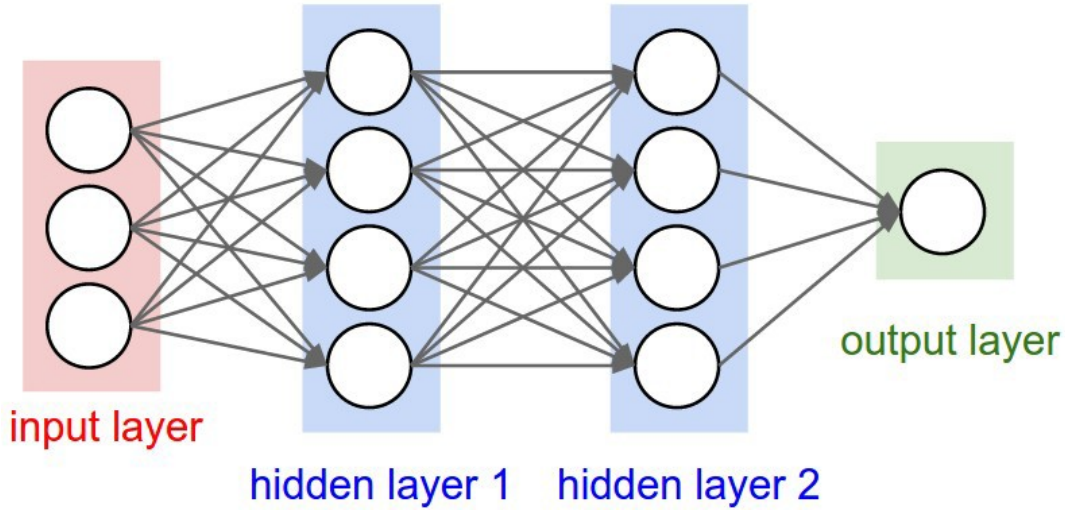
How do we train?

4 + 2 = 6 neurons (not counting inputs)
[3 x 4] + [4 x 2] = 20 weights
4 + 2 = 6 biases

26 learnable **parameters**

Neural Network Architecture (Multi-Layer Perceptron)

Going deeper: a 3-layer neural network with two layers of hidden units



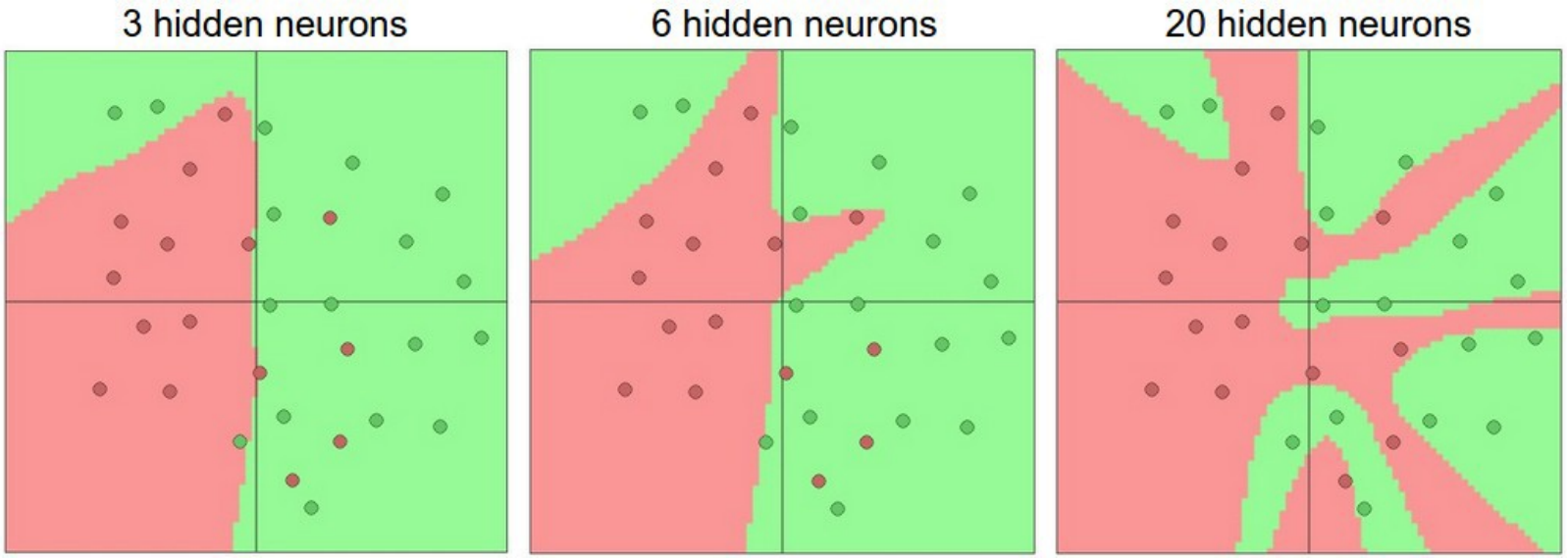
Naming conventions; a N-layer neural network:

- N − 1 layers of hidden units
- One output layer

Why Deeper? Representation Power

Neural network with at least one hidden layer is a universal approximator (can represent any function).

Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko



The capacity of the network increases with more hidden units and more hidden layers

Why go deeper? Read e.g.,: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana

DEMO

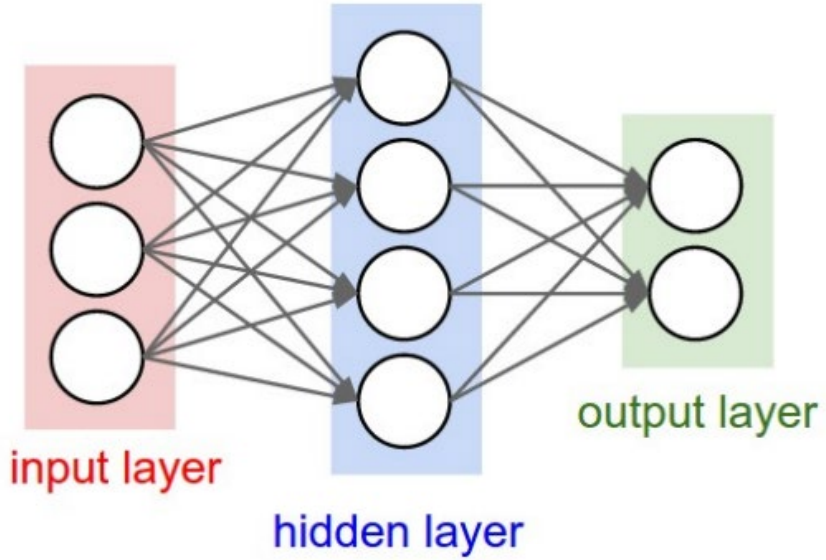
Demo

Neural Network

We only need to know two algorithms

- Forward pass: performs inference
- Backward pass: performs learning

Forward Pass



Output of the network can be written as:

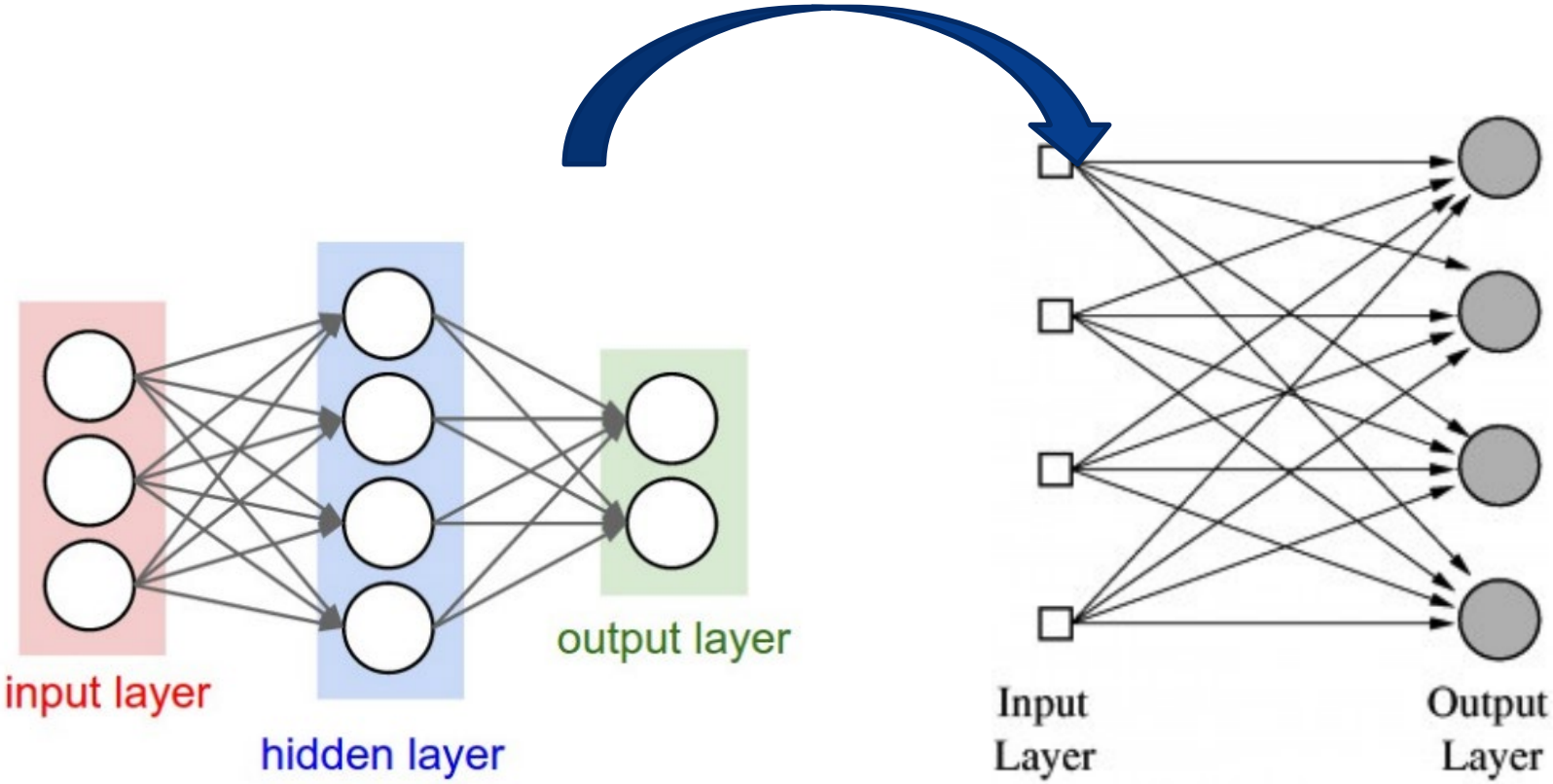
$$h_j(\mathbf{x}) = f\left(v_{j0} + \sum_{i=1}^{D} x_i v_{ji}\right)$$

$$o_k(\mathbf{x}) = g\left(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}\right)$$

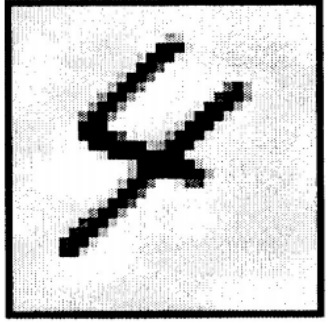
(j indexing hidden units, k indexing the output units D number of inputs)

Activation functions f , g:
sigmoid/logistic, tanh, or rectified linear (ReLU)

Forward Pass: special case



Network Neural Network: An Example



Classify image of handwritten digit (32x32 pixels): 4 vs non-4

How would you build your network? For example, use one hidden layer and the sigmoid activation function:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

How can we train the network, that is, adjust all the parameters **w**?

How to Train a Neural Network?

Find weights: $\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$

where o = f (x; w) is the output of a neural network

Define a loss function, eg:

Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$

Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$

Gradient descent: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$

where η is the learning rate (and E is error/loss)

Useful Derivatives

name	function	derivative
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

Training Neural Networks: Back-propagation

Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

Given any error function E, activation functions g() and f (), just need to derive gradients

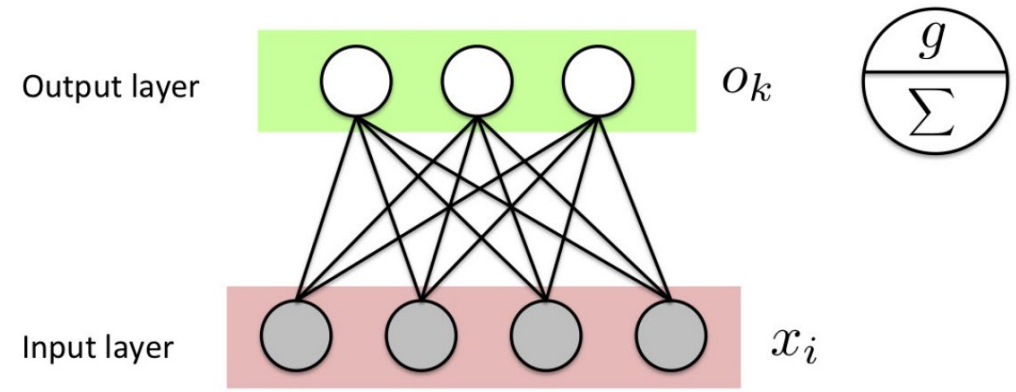
Key Idea behind Backpropagation

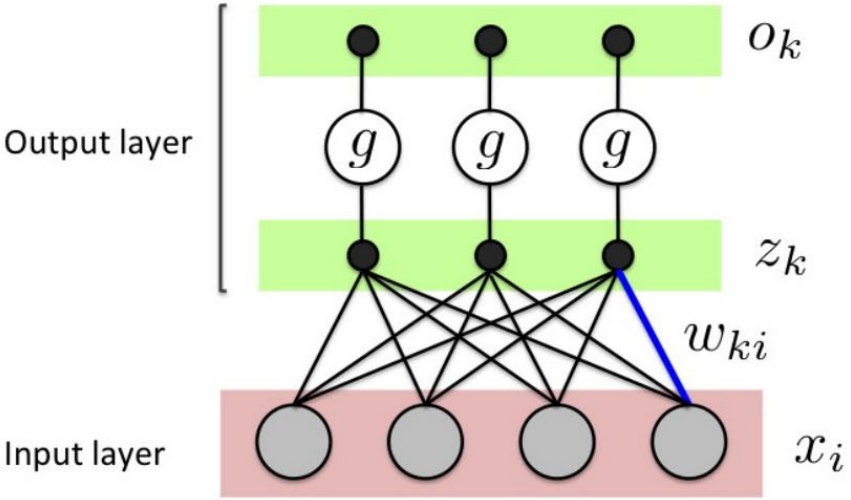
We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity

- Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
- Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
- We can compute error derivatives for all the hidden units efficiently
- Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit

This is just the chain rule!

Computing Gradients: Single Layer Network





o_k Output of unit k

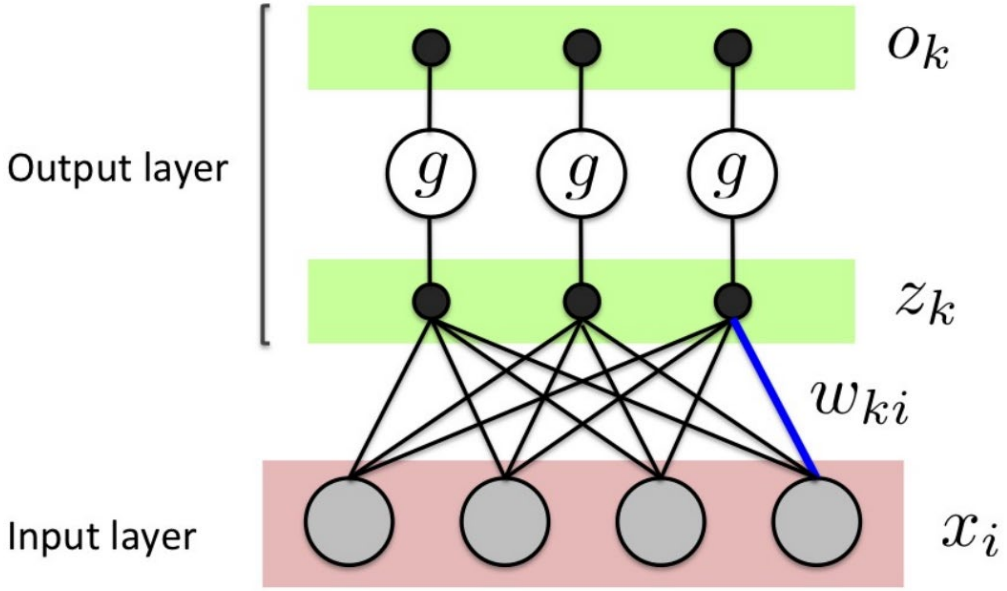
g Output layer activation function

z_k Net input to output unit k

w_{ki} Weight from input i to k

x_i Input unit i

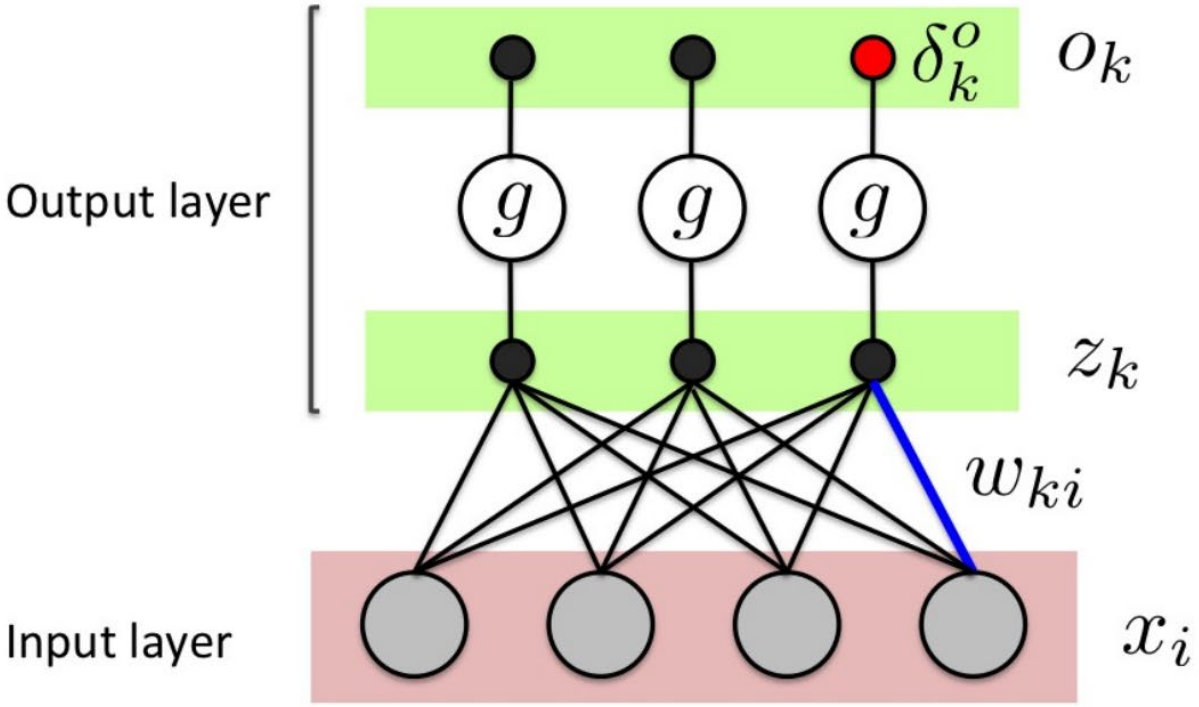
Computing Gradients: Single Layer Network



Error gradients for single layer network $\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$

Error gradient is computable for any continuous activation function g(), and any continuous error function

Computing Gradients: Single Layer Network

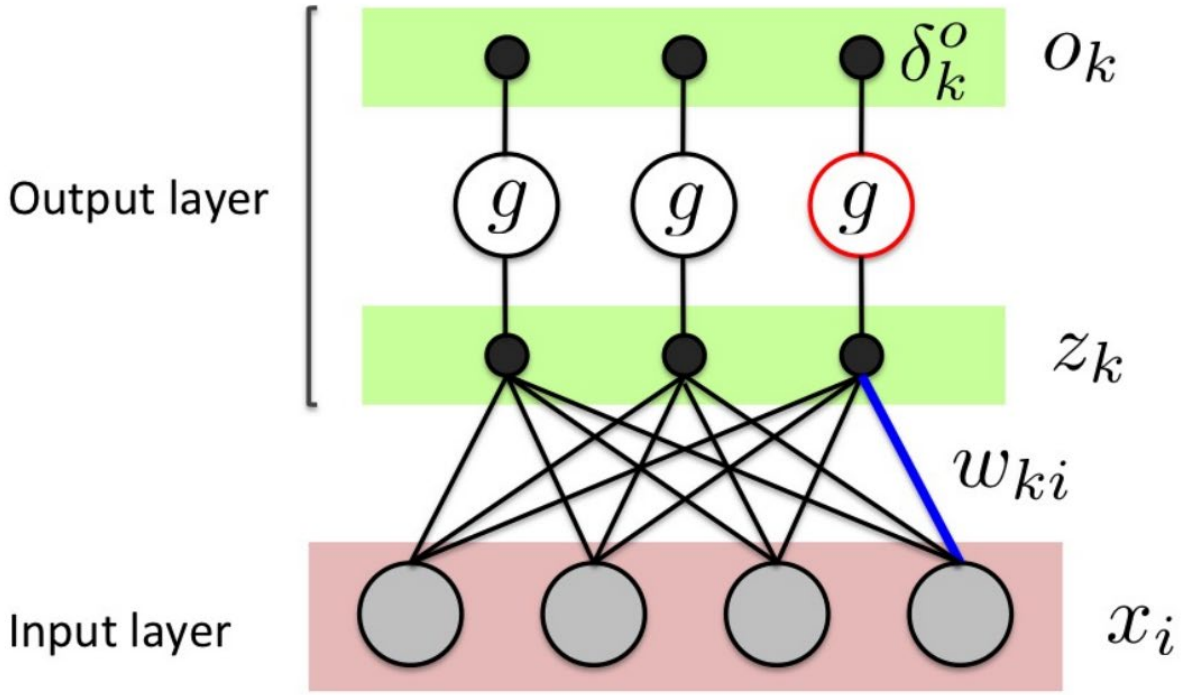


$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \underbrace{\frac{\partial E}{\partial o_k}}_{\delta_k^o} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network

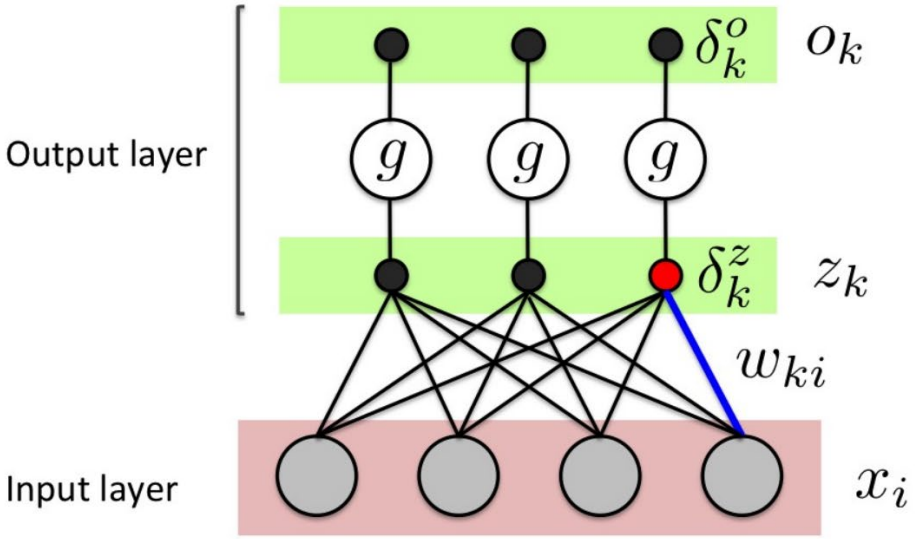


$$\frac{\partial o_k}{\partial z_k}$$

Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^o \frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network

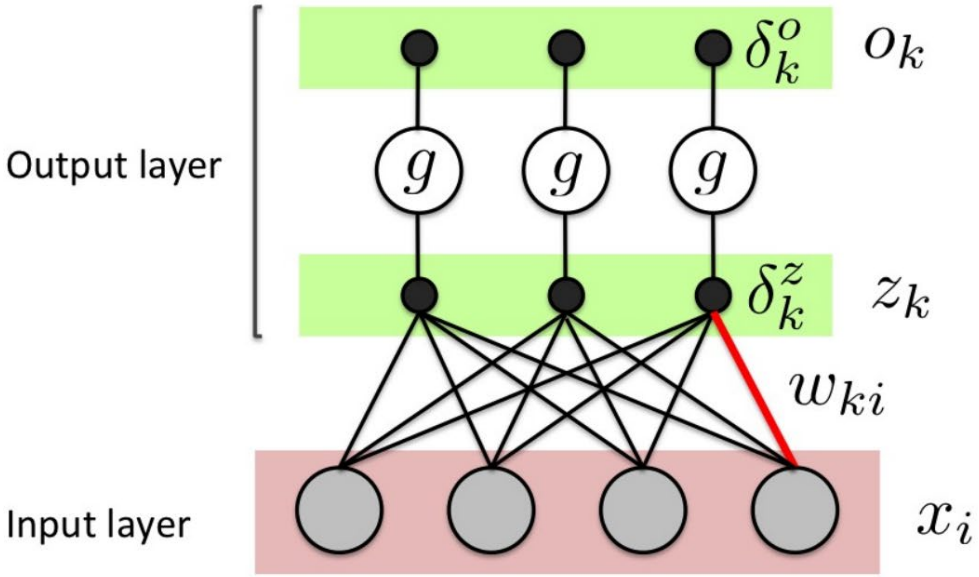


$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \underbrace{\delta_k^o \cdot \frac{\partial o_k}{\partial z_k}}_{\delta_k^z}\frac{\partial z_k}{\partial w_{ki}}$$

Computing Gradients: Single Layer Network



$$\frac{\partial z_k}{\partial w_{ki}}$$

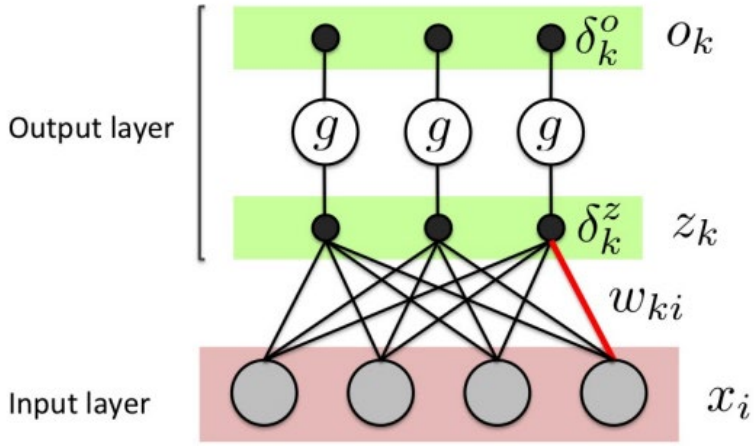
Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \cdot x_i$$

Gradient Descent for Single Layer Network

Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)} := \delta_k^o$$



Using logistic activation functions:

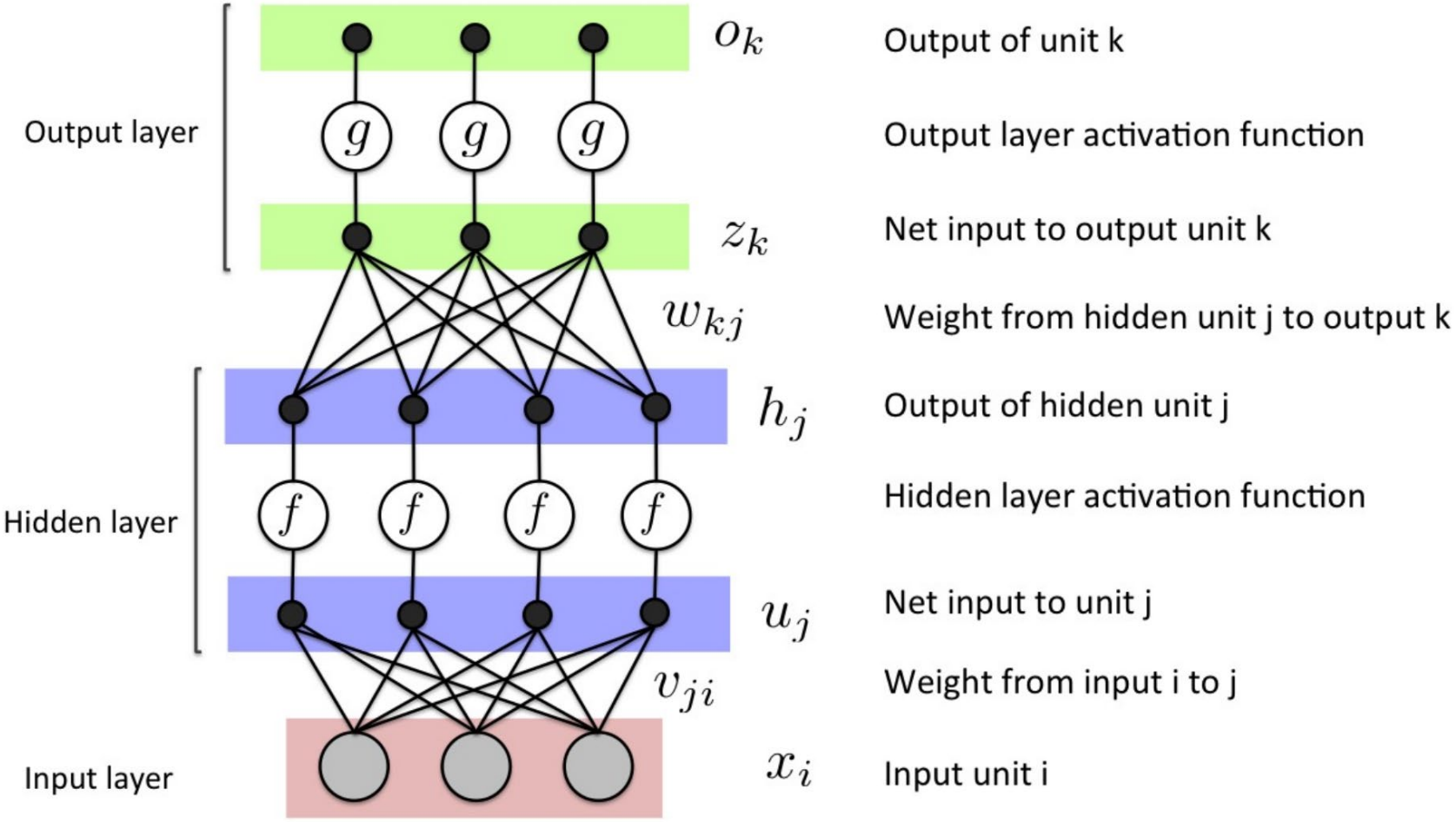
$$o_k^{(n)} = g(z_k^{(n)}) = (1 + \exp(-z_k^{(n)}))^{-1}$$
$$\frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} = o_k^{(n)}(1 - o_k^{(n)})$$

The error gradient is then: $\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{ki}} = \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$

The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

Multi-Layer Neural Network

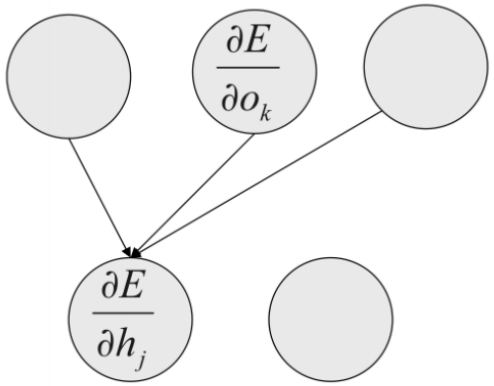


Back-propagation: Sketch on One Training Case

Convert discrepancy between each output and its target value into an error derivative

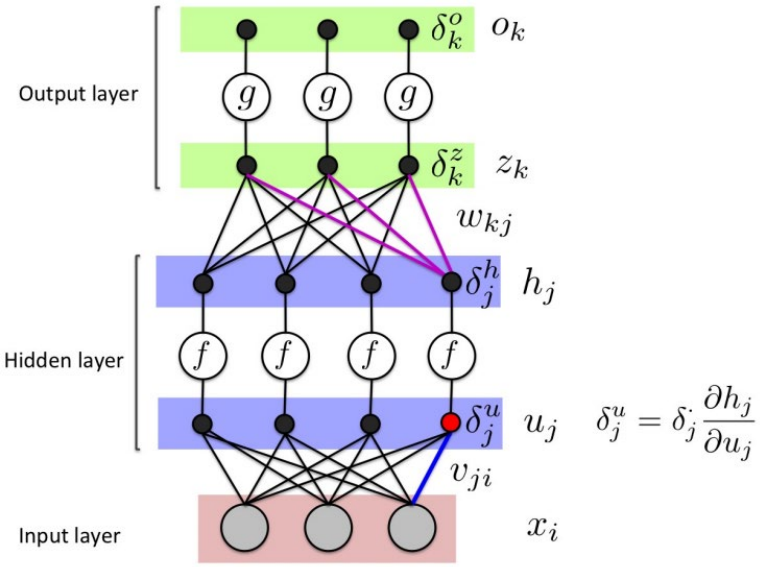
$$E = \frac{1}{2}\sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at k to each unit j according to its influence on k (depends on wkj)]



Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

Gradient Descent for Multi-layer Network



The output weight gradients for a multi-layer network are the same as for a single layer network

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δk is the error w.r.t. the net input for unit k

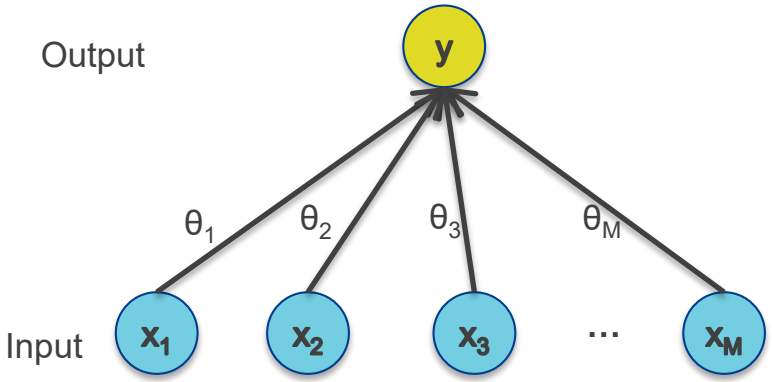
Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{u,(n)} x_i^{(n)}$$

Backpropagation
Training

Case 1: Logistic Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

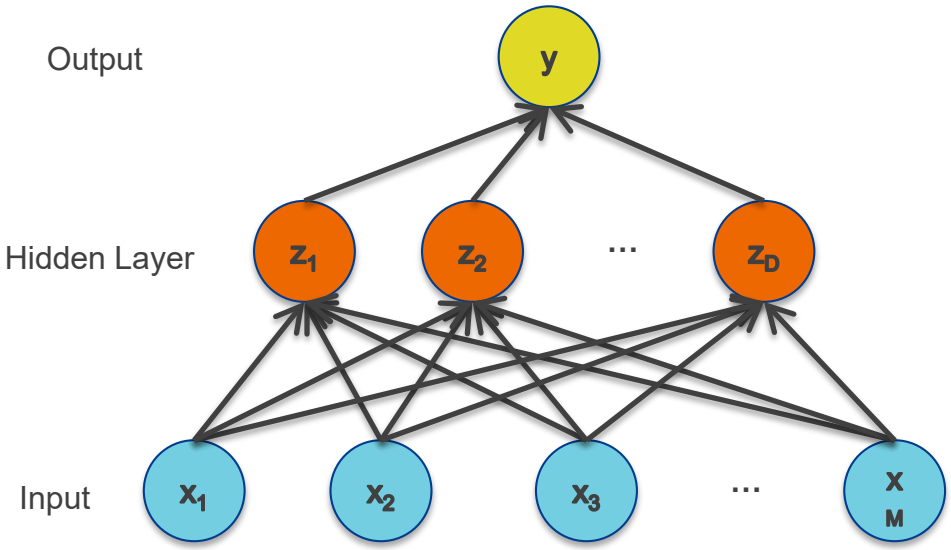
$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

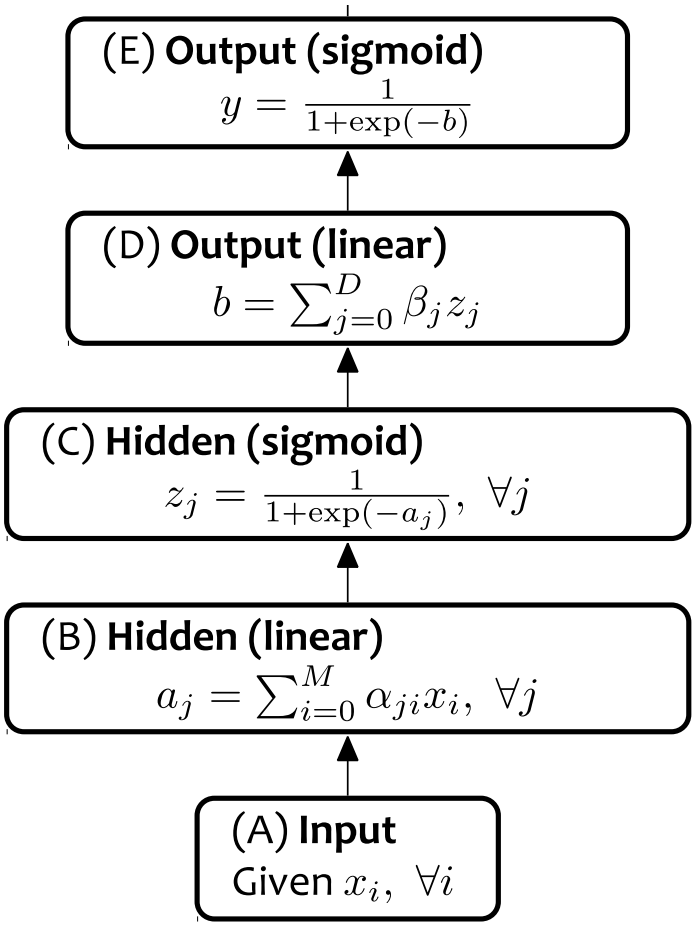
$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \frac{da}{d\theta_j} = x_j$$

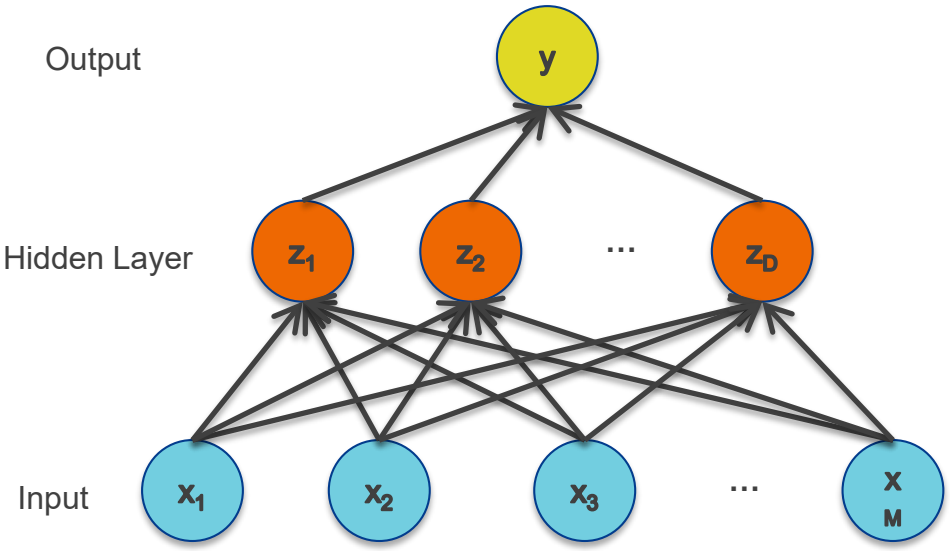
$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \frac{da}{dx_j} = \theta_j$$

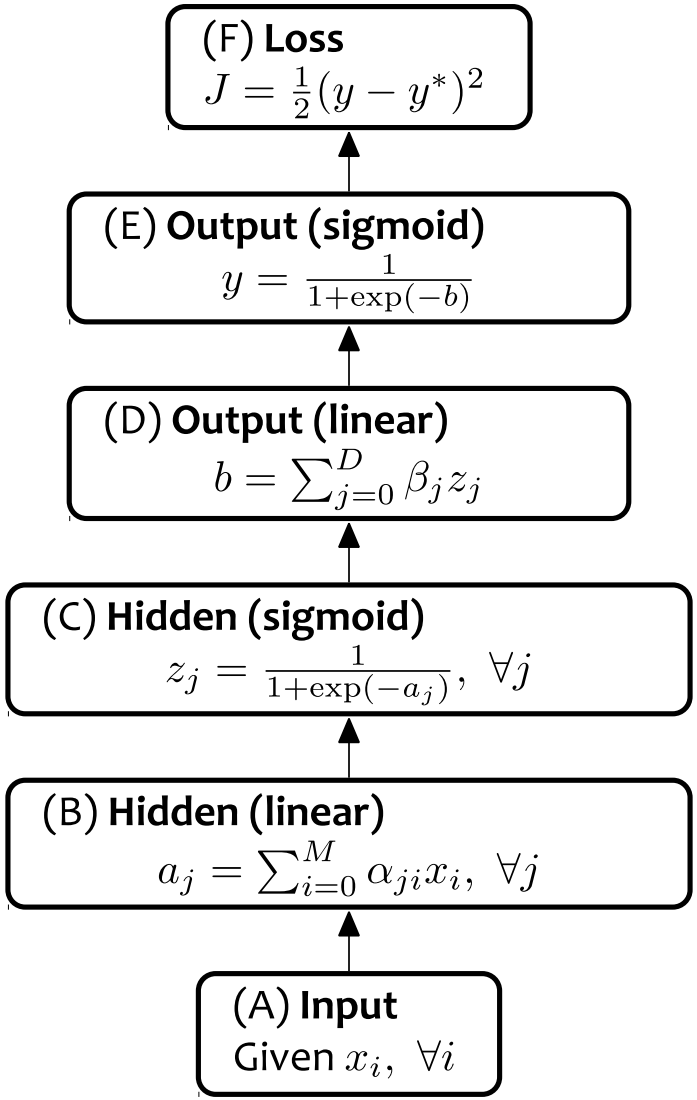
Backpropagation





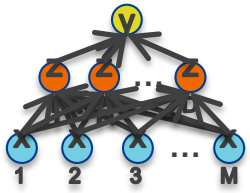
Backpropagation





Backpropagation Training

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$$

Backpropagation

Case 2:

	Forward	Backward
Module 5	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
Module 4	$y = \frac{1}{1 + \exp(-b)}$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$
Module 3	$b = \sum_{j=0}^{D} \beta_j z_j$	$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$ $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$
Module 2	$z_j = \frac{1}{1 + \exp(-a_j)}$	$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$
Module 1	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$ $\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D} \alpha_{ji}$

Choosing Activation and Loss Functions

When using a neural network for regression, sigmoid activation and MSE as the loss function work well

For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression)

$$E = -\sum_{n=1}^{N} t^{(n)} \log o^{(n)} + (1 - t^{(n)}) \log(1 - o^{(n)})$$

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

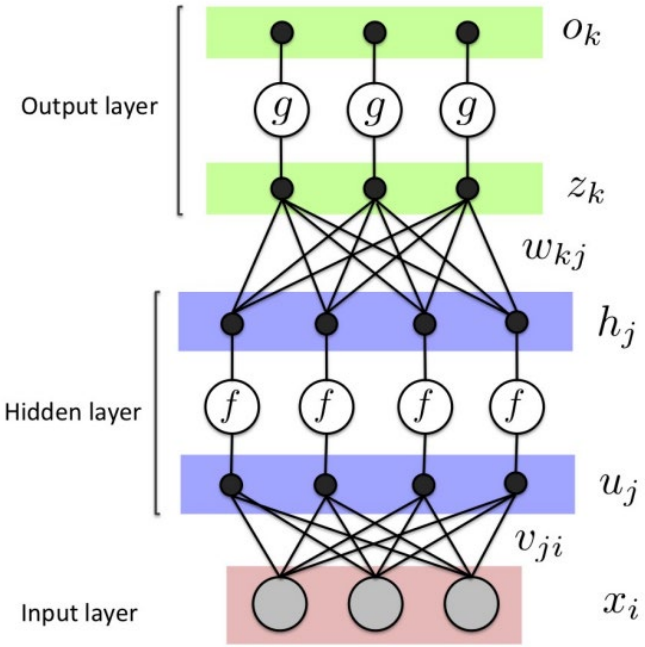
We can then compute via the chain rule

$$\frac{\partial E}{\partial o} = (o - t)/(o(1 - o))$$

$$\frac{\partial o}{\partial z} = o(1 - o)$$

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z} = (o - t)$$

Multi-class Classification



For multi-class classification problems, use cross-entropy as loss and the softmax activation function

$$E = -\sum_n \sum_k t_k^{(n)} \log o_k^{(n)}$$

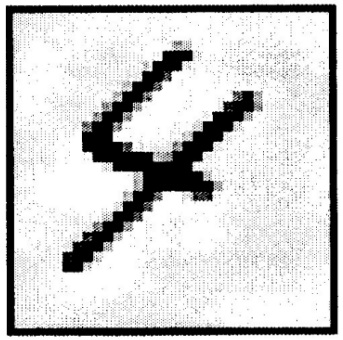
$$o_k^{(n)} = \frac{\exp(z_k^{(n)})}{\sum_j \exp(z_j^{(n)})}$$

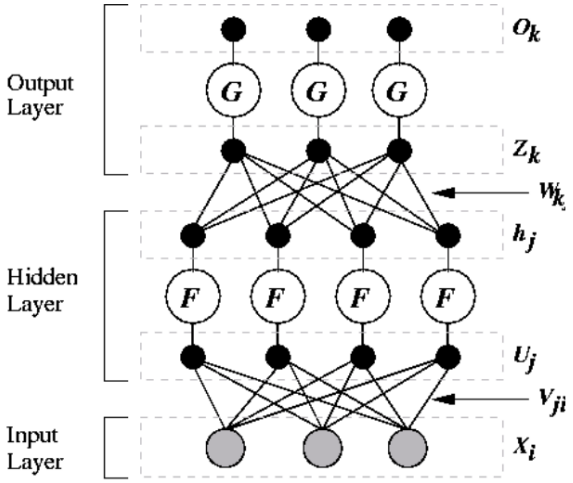
And the derivatives become

$$\frac{\partial o_k}{\partial z_k} = o_k(1 - o_k)$$

$$\frac{\partial E}{\partial z_k} = \sum_j \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial z_k} = (o_k - t_k) o_k (1 - o_k)$$

Ways to Use Weight Derivatives





Now trying to classify image of handwritten digit:
32x32 pixels
10 output units, 1 per digit
Use the softmax function:

$$o_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

Ways to Use Weight Derivatives

How often to update

- after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

- after each training case (stochastic gradient descent)
- after a mini-batch of training cases

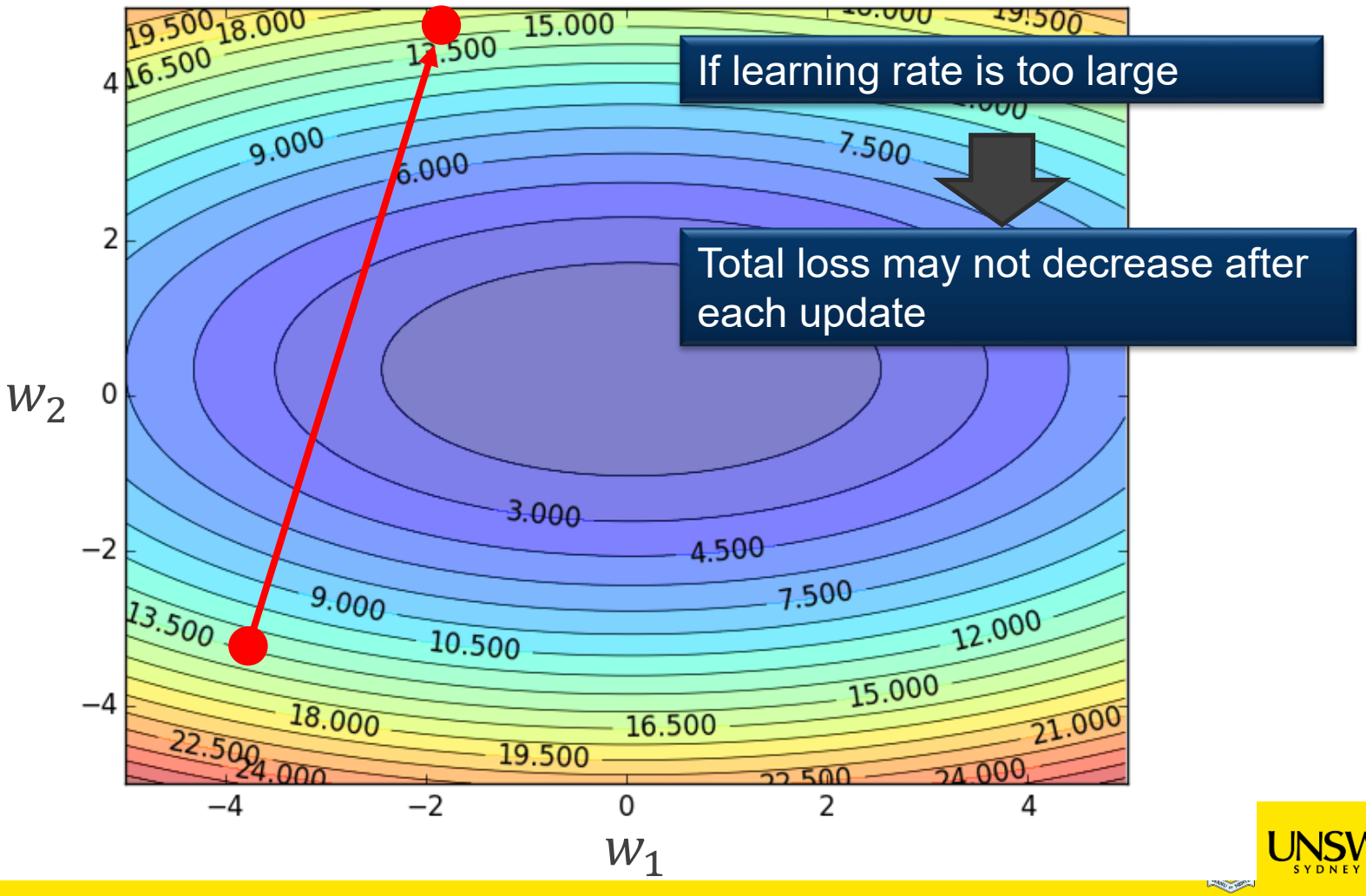
How much to update

- Use a fixed learning rate
- Adapt the learning rate
- Add momentum

$$\begin{aligned} w_{ki} &\leftarrow w_{ki} - v \\ v &\leftarrow \gamma v + \eta \frac{\partial E}{\partial w_{ki}} \end{aligned}$$

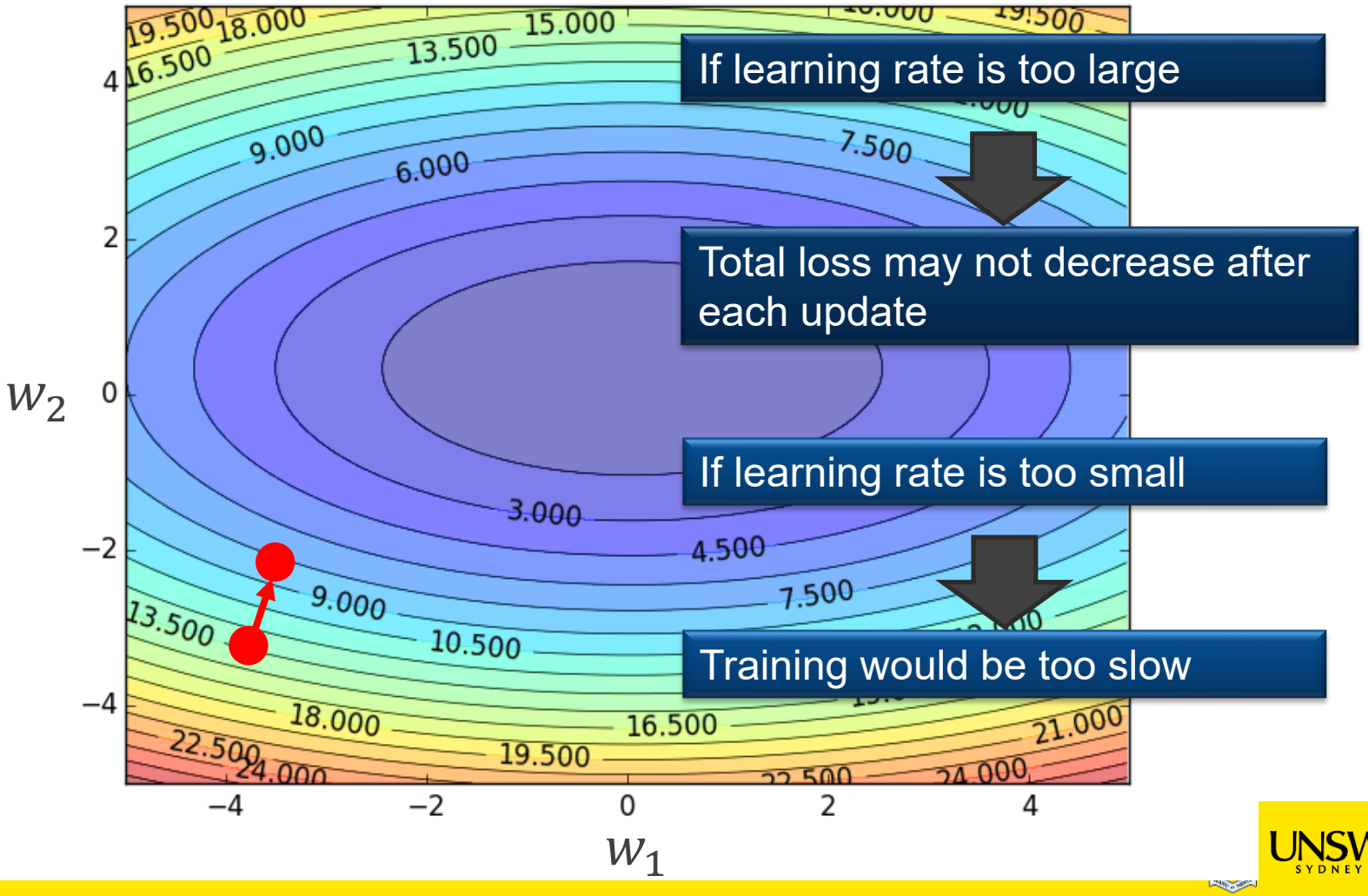
Learning Rates

Set the learning rate η carefully



Learning Rates

Set the learning rate η carefully



Learning Rates

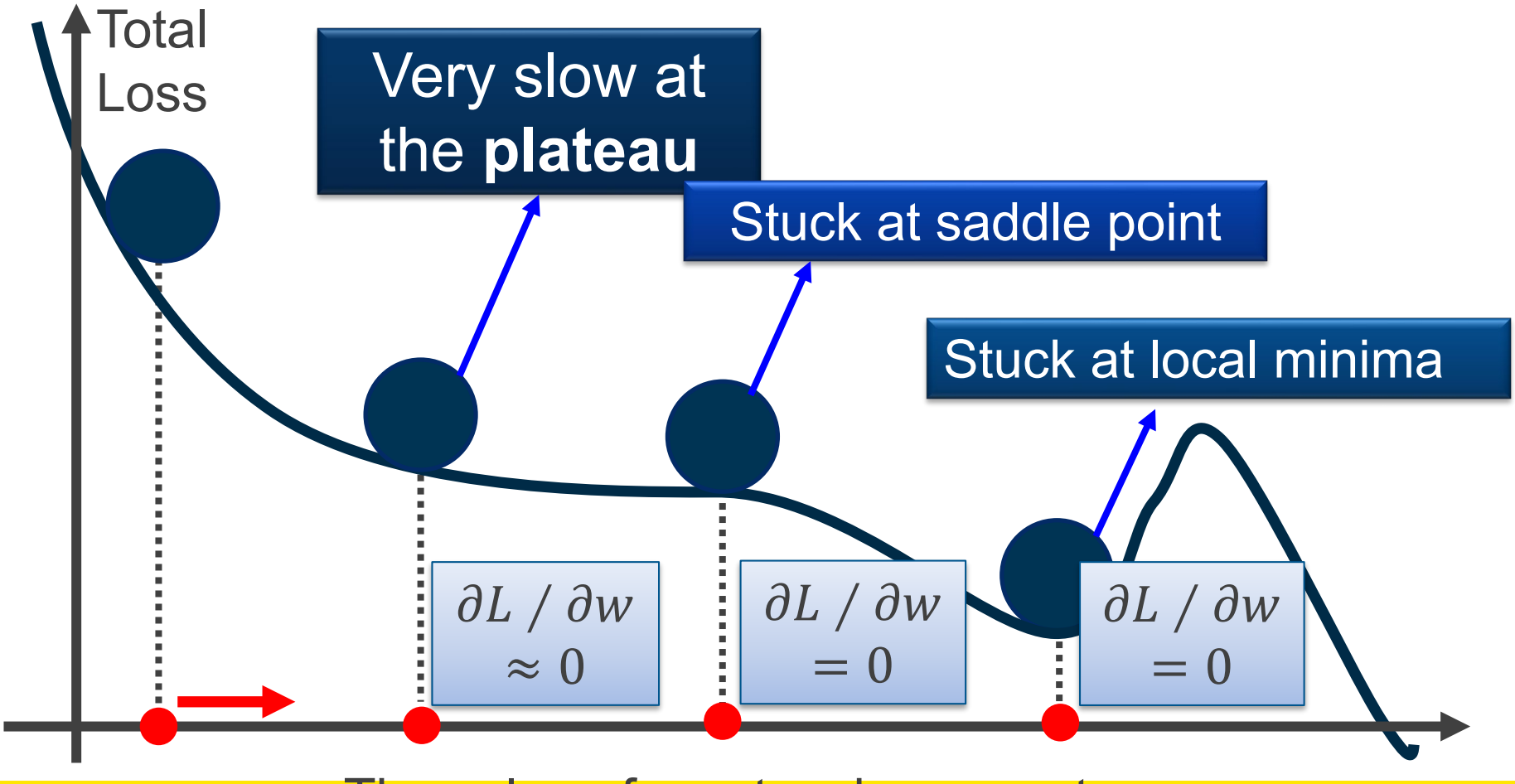
- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.

- At the beginning, we are far from the destination, so we use larger learning rate
- After several epochs, we are close to the destination, so we reduce the learning rate
- E.g. 1/t decay: $\eta^t = \eta/\sqrt{t+1}$

Learning rate cannot be one-size-fits-all

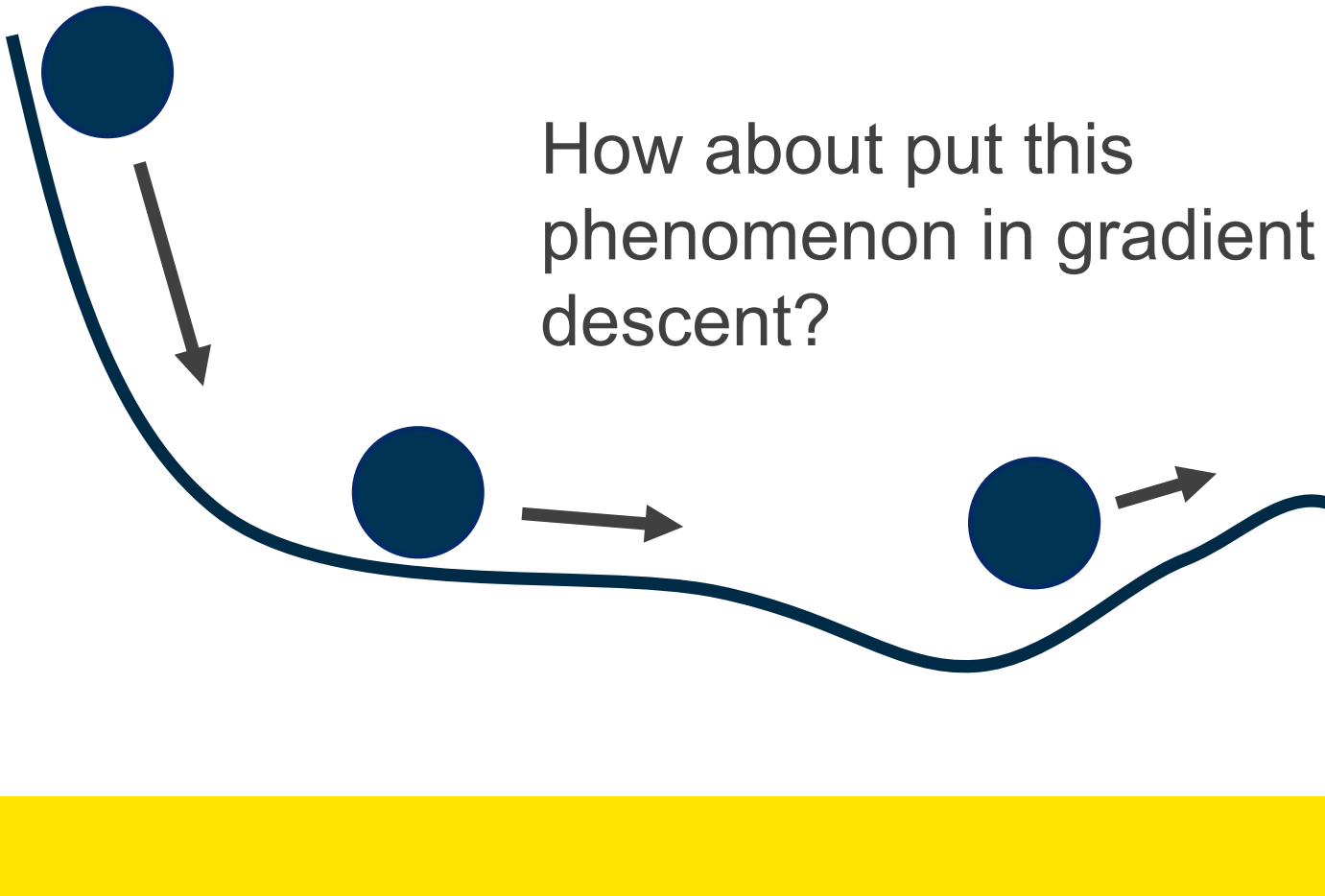
- Giving different parameters different learning rates

Hard to find optimal network parameters



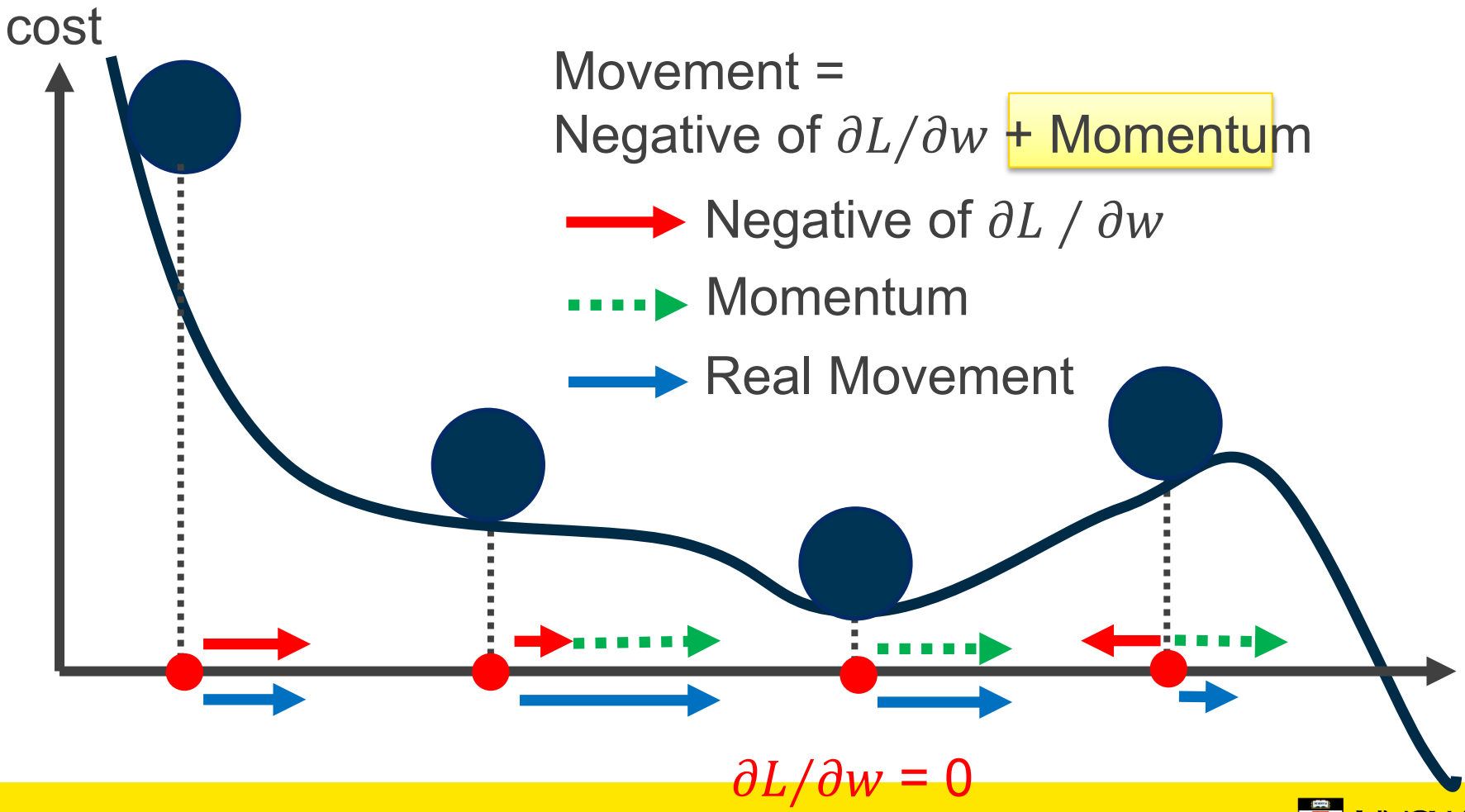
Add Momentum

Momentum



Add Momentum

Still not guarantee reaching global minima, but give some hope ……



Adam

RMSProp (Advanced Adagrad) + Momentum

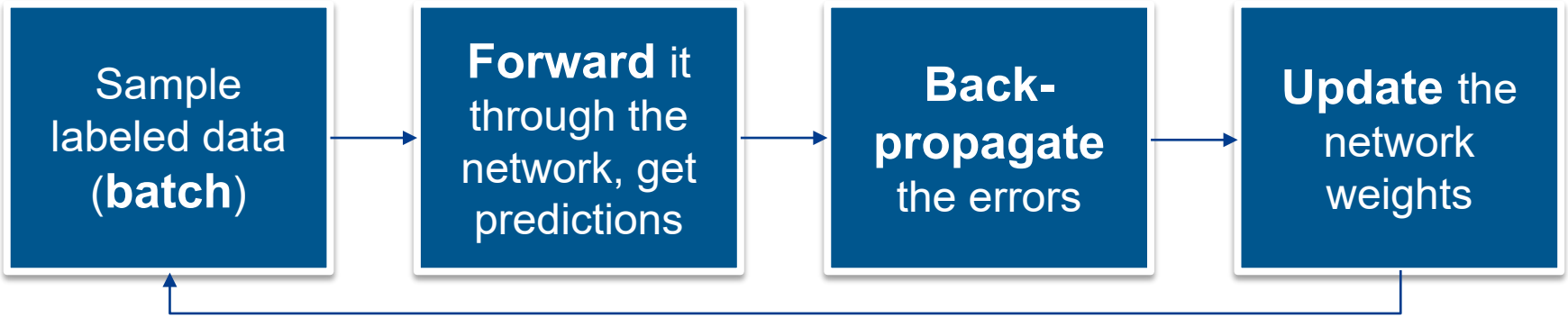
```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

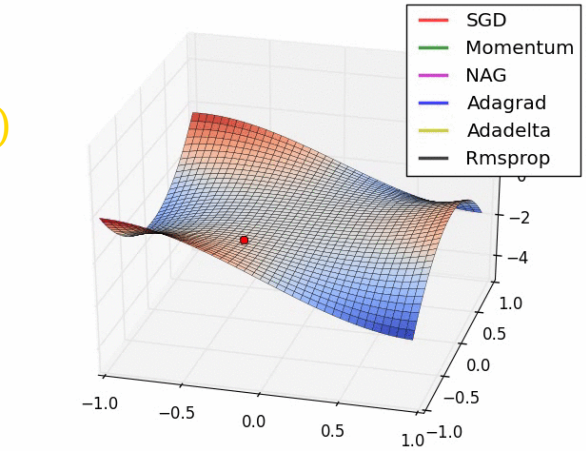
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

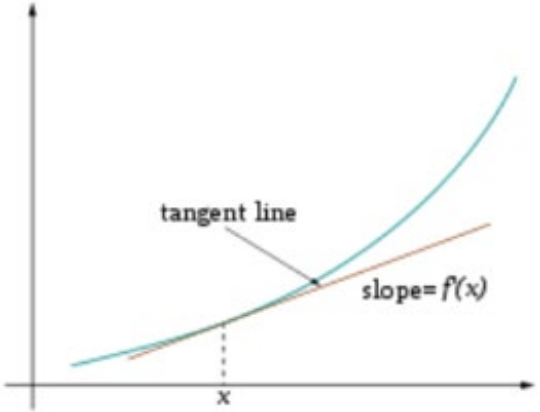
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

How to Train a Neural Network?



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





Use error signal to change the **weights** and get more accurate predictions
Subtracting a fraction of the **gradient** moves you towards the **(local) minimum of the cost function**

https://medium.com/@ramrajchandradevan/the-evolution-of-gradient-descend-optimization-algorithm-41

Overfitting

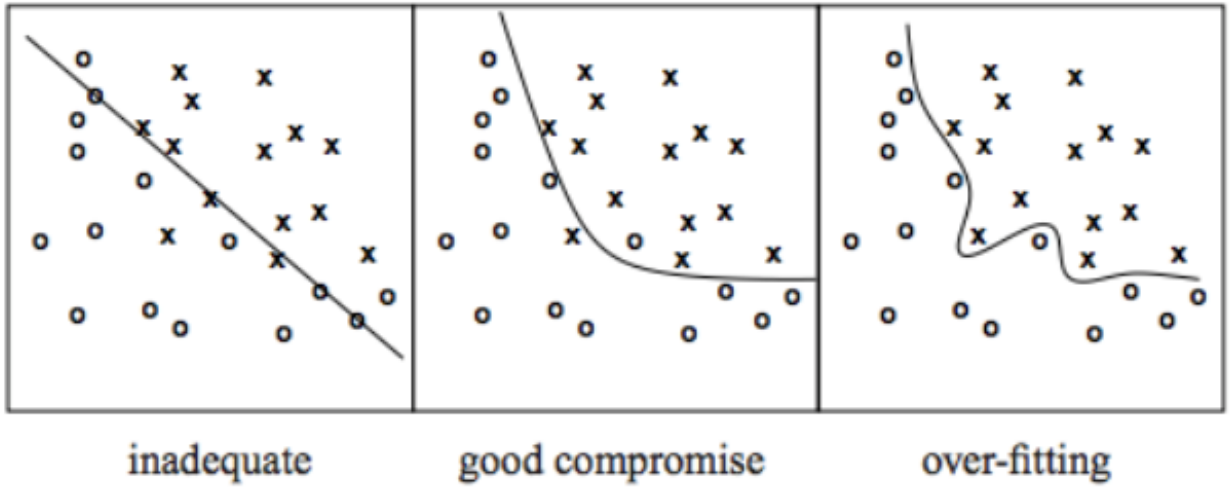
The training data contains information about the regularities in the mapping from input to output. But it also contains noise

- The target values may be unreliable.
- There is sampling error: There will be accidental regularities just because of the particular training cases that were chosen

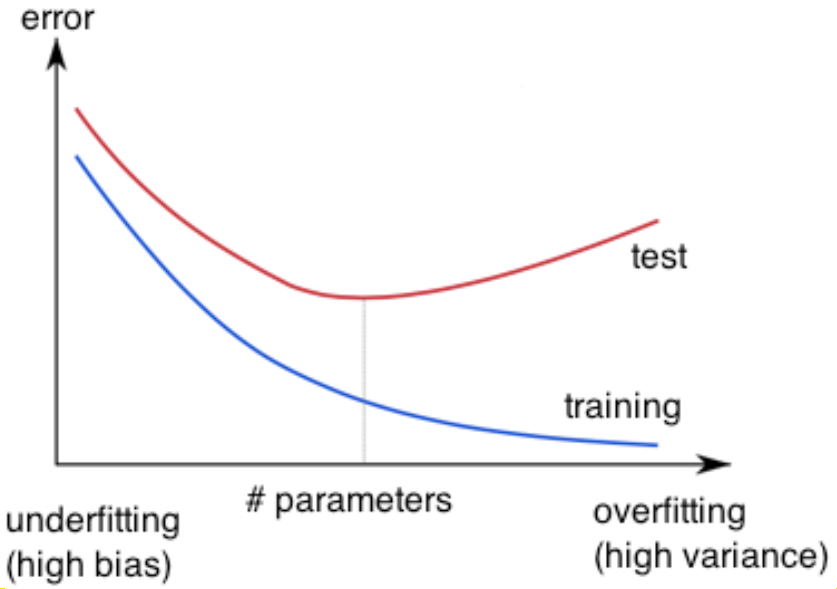
When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.

- So it fits both kinds of regularity.
- If the model is very flexible it can model the sampling error really well. This is a disaster

Overfitting



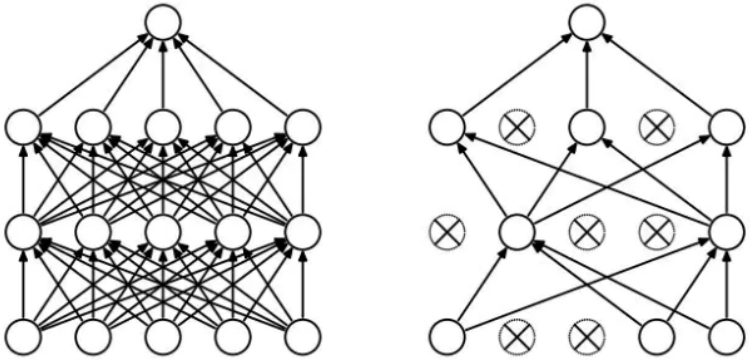
http://wiki.bethanycrane.com/overfitting-of-data



Learned hypothesis may fit the training data very well, even outliers (noise) but fail to generalize to new examples (test data)

https://www.neuraldesigner.com/images/learning/selection_error.svg

Regularization



Dropout

- Randomly drop units (along with their connections) during training
- Each unit retained with fixed probability p, independent of other units
- Hyper-parameter p to be chosen (tuned)

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." Journal of machine learning research (2014)

L2 = weight decay

- Regularization term that penalizes big weights, added to the objective
- Weight decay value determines how dominant regularization is during gradient computation
- Big weight decay coefficient → big penalty for big weights

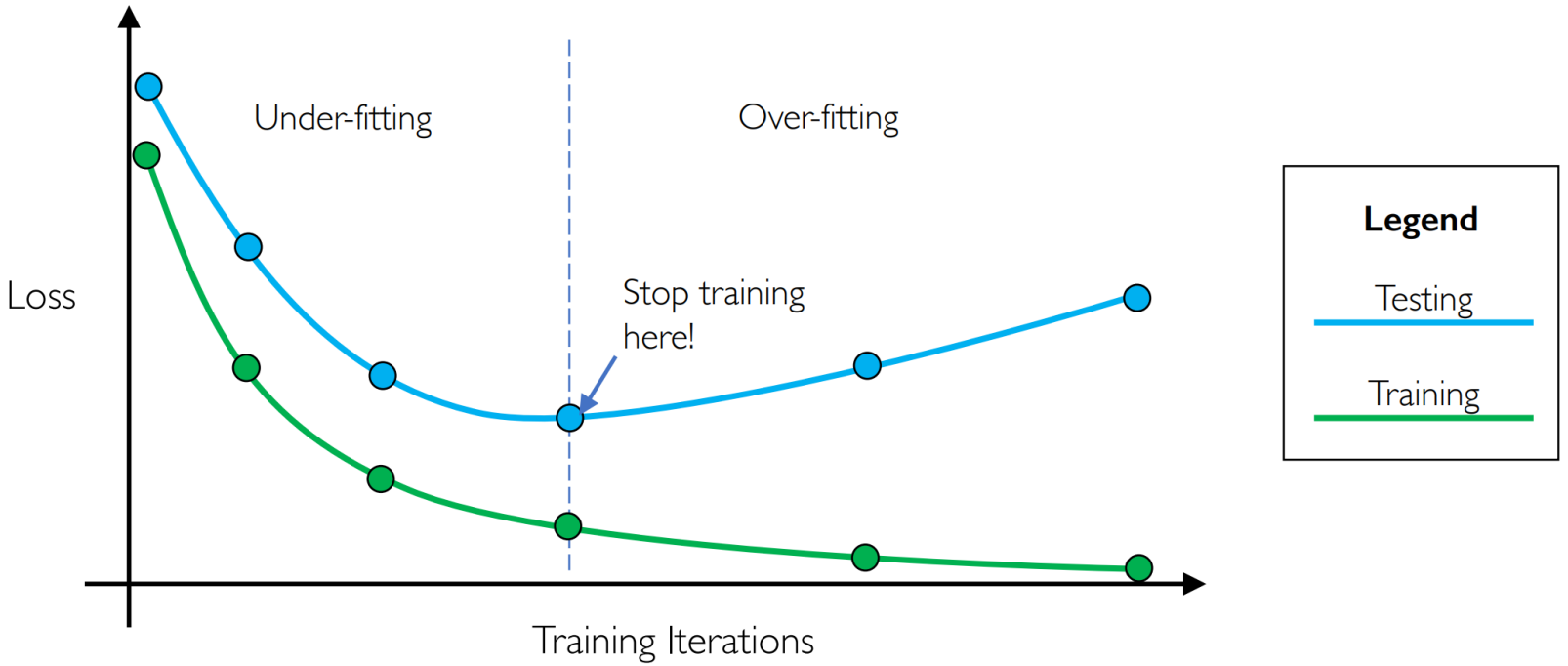
$$J_{reg}(\theta) = J(\theta) + \lambda \sum_{k} \theta_k^2$$

Early-stopping

- Use validation error to decide when to stop training
- Stop when monitored quantity has not improved after n subsequent epochs
- n is called patience

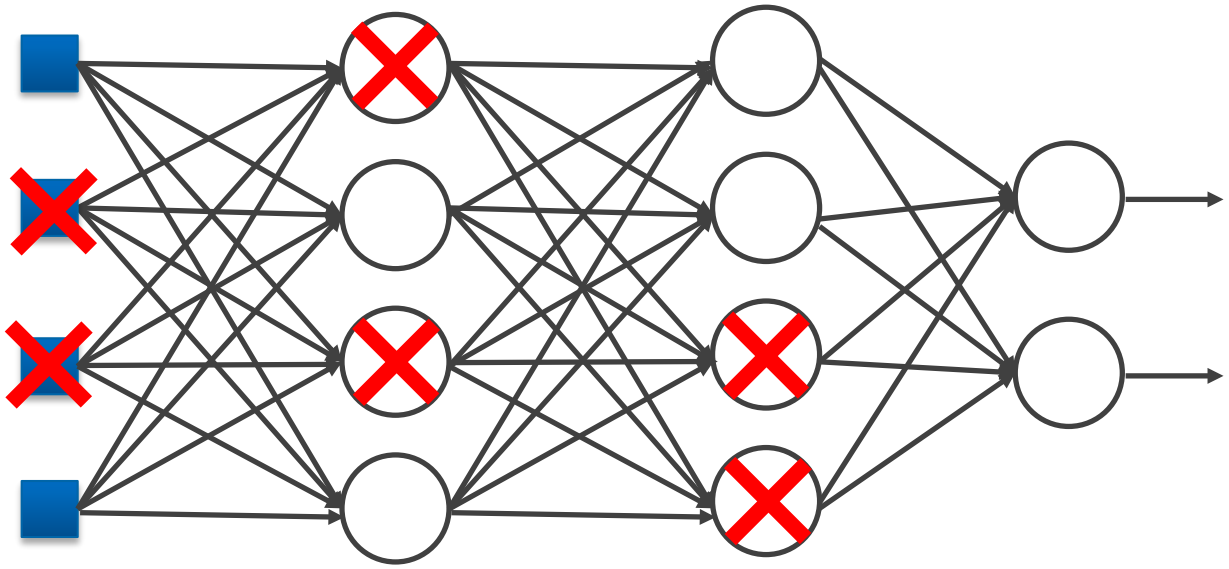
Early Stopping

- Stop training before we have a chance to overfit



Dropout

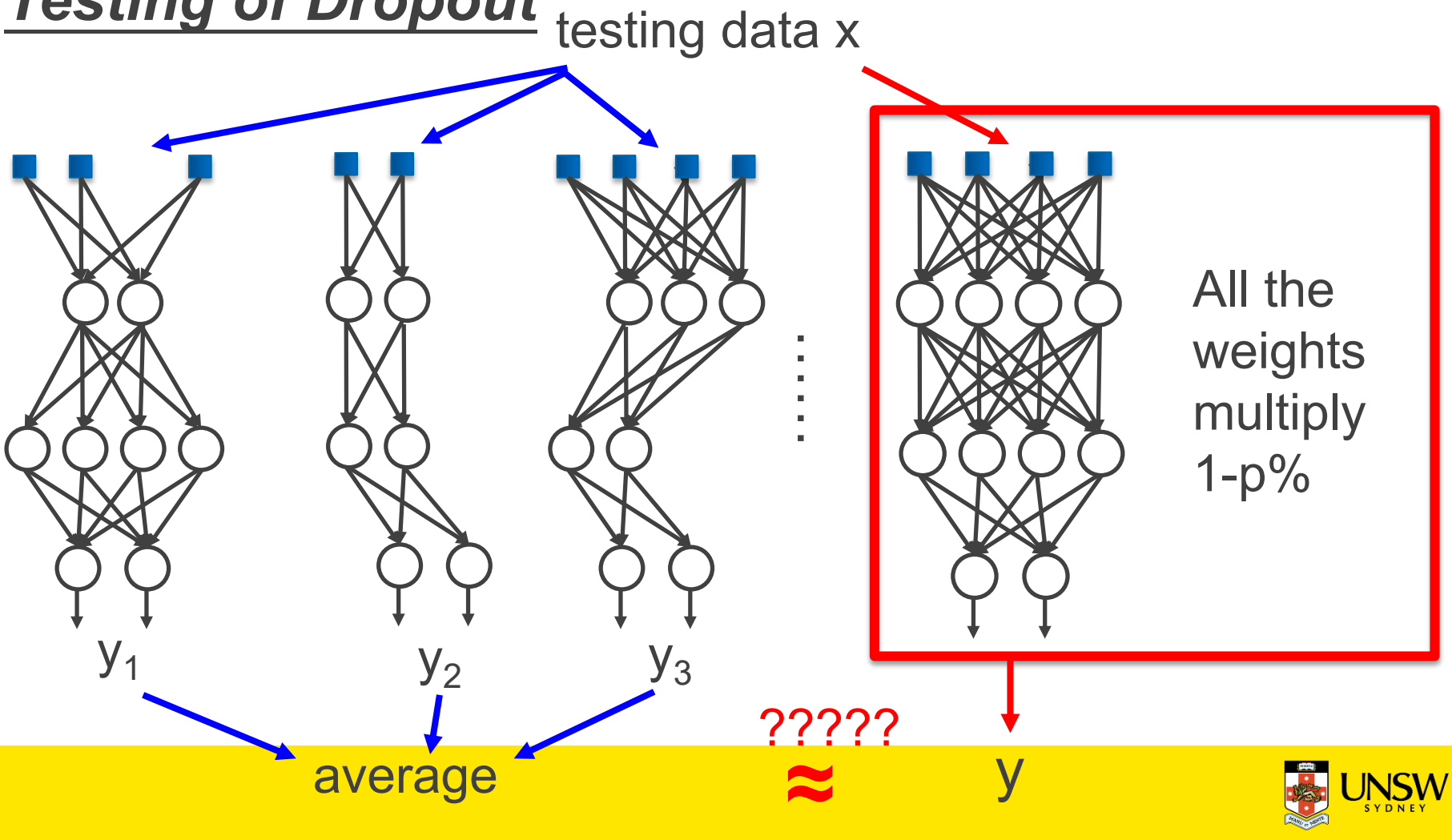
Training:



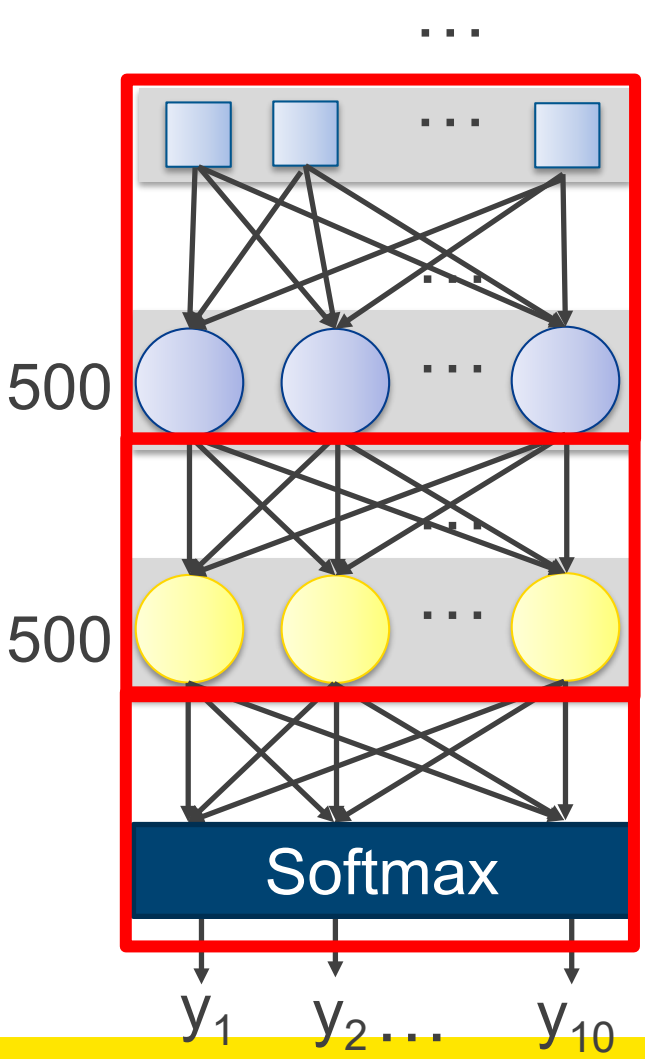
- **Each time before updating the parameters**
  - Each neuron has p% to dropout

Dropout is a kind of ensemble.

Testing of Dropout



Dropout




```
model = Sequential()
```

```
model.add( Dense( input_dim=28*28,
                  output_dim=500 ))
model.add( Activation('sigmoid') )
```

model.add(dropout(0.8)

```
model.add( Dense( output_dim=500 ) )
model.add( Activation('sigmoid') )
```

model.add(dropout(0.8)

```
model.add( Dense(output_dim=10 ) )
model.add( Activation('softmax') )
```

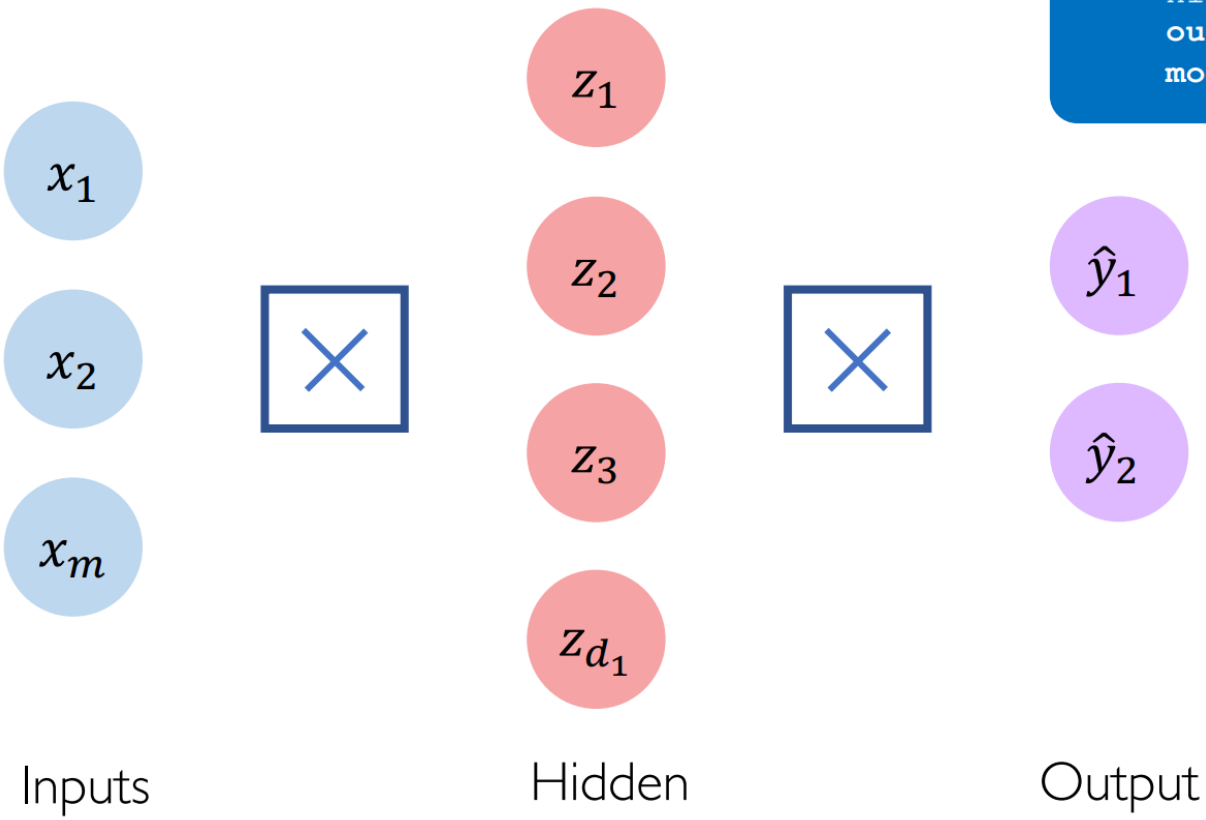
DL Frameworks

theano

Caffe

Microsoft
CNTK

Let's Play --- Practice

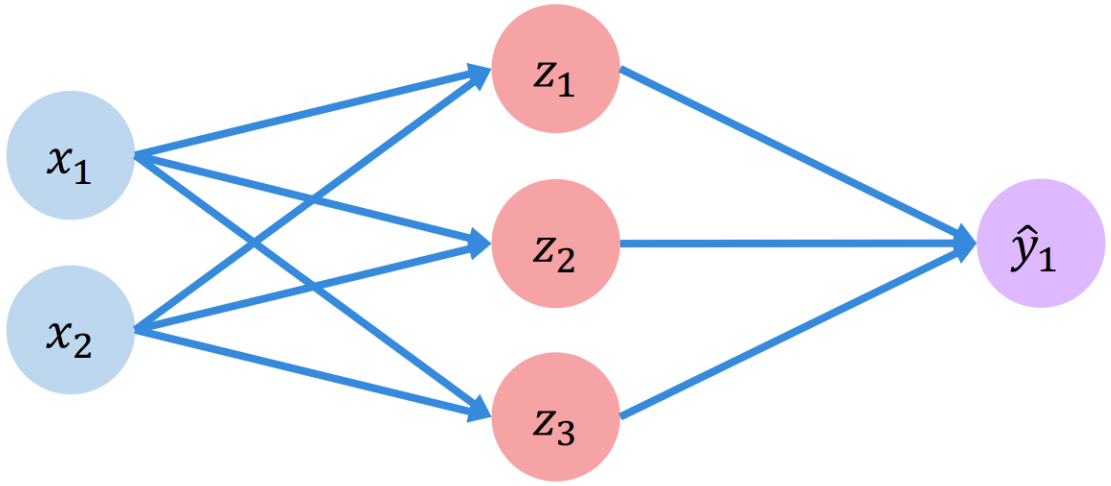



```
from tf.keras.layers import *

inputs = Inputs(m)
hidden = Dense(d1)(inputs)
outputs = Dense(2)(hidden)
model = Model(inputs, outputs)
```

Let's Play --- Practice

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



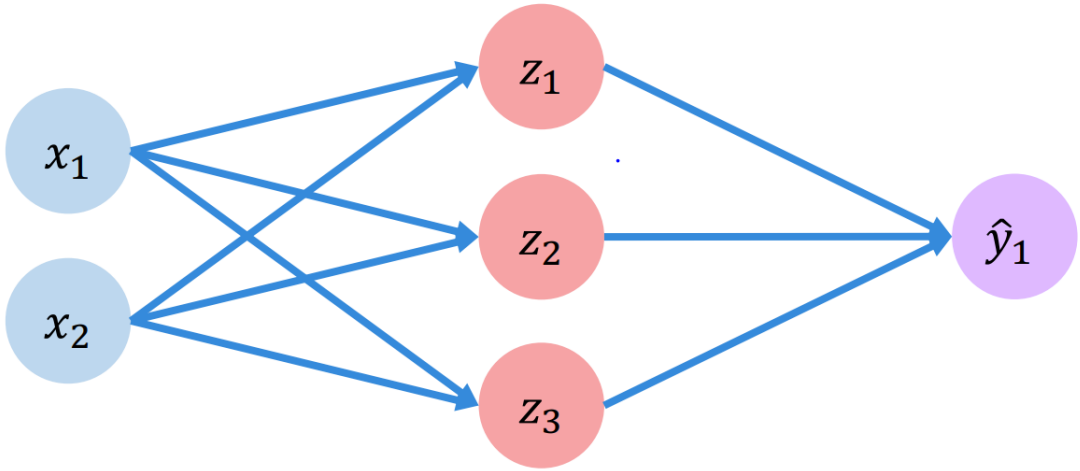
$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Let's Play --- Practice

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$	y
30	90
80	20
85	95
⋮	⋮

Final Grades (percentage)

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

Let's Play --- Practice

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

Let's Play --- Practice

Adaptive Learning Rate

- Momentum — `tf.train.MomentumOptimizer`
- Adagrad — `tf.train.AdagradOptimizer`
- Adadelta — `tf.train.AdadeltaOptimizer`
- Adam — `tf.train.AdamOptimizer`
- RMSProp — `tf.train.RMSPropOptimizer`

```
tf.keras.layers.Dropout(p=0.5)
```

Keras

or theano

Very flexible

Need some effort to learn

Interface of TensorFlow or Theano

keras

Easy to learn and use

You can modify it if you can write TensorFlow or Theano

(still have some flexibility)

Keras

François Chollet is the author of Keras.

- He currently works for Google as a deep learning engineer and researcher.

Keras means *horn* in Greek

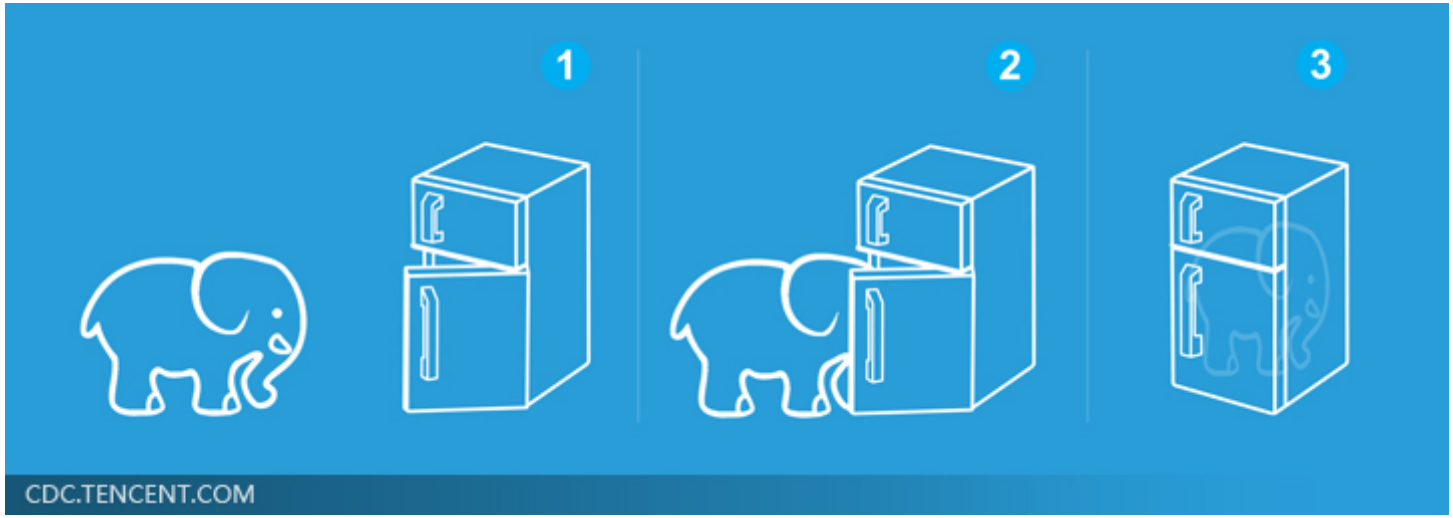
Documentation: http://keras.io/

Example: https://github.com/fchollet/keras/tree/master/examples

Three Steps for Deep Learning

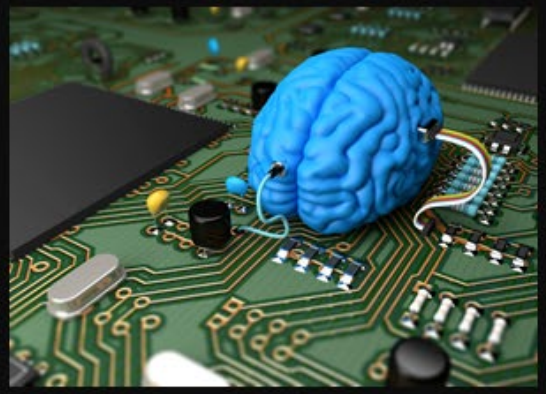
Step 1: define a set of function → Step 2: goodness of function → Step 3: pick the best function

Using Deep Learning is so simple ……



Deep Learning

What society thinks I do



What my friends think I do

What other computer scientists think I do

What mathematicians think I do

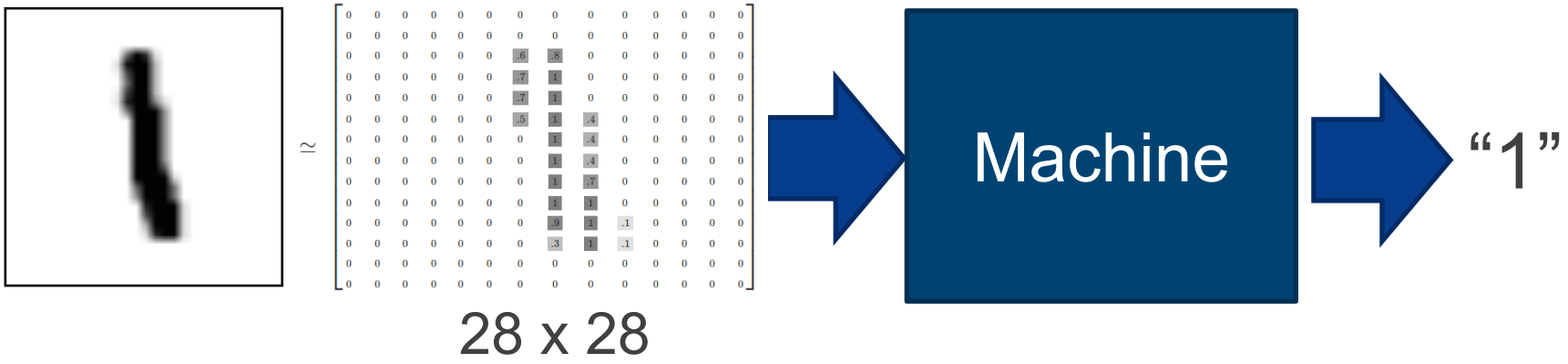
What I think I do

```
from theano import *
```

What I actually do

Example Application

Handwriting Digit Recognition

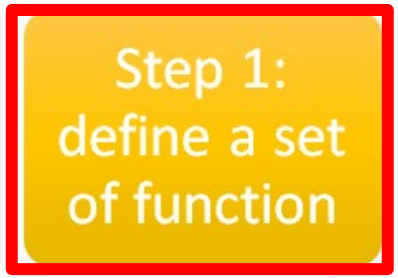


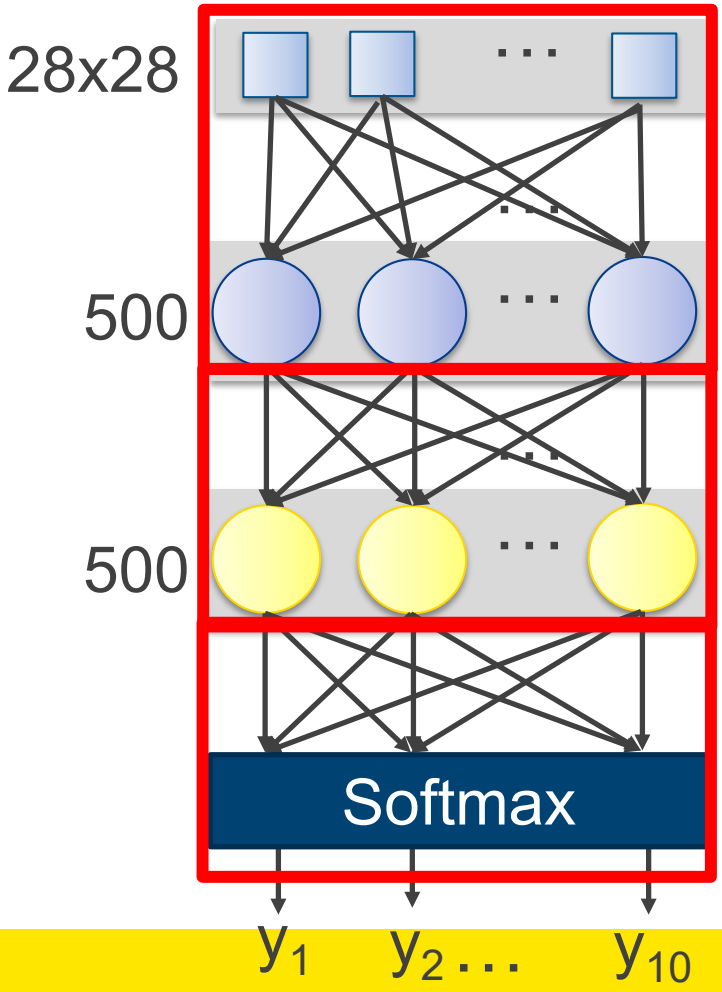
MNIST Data: http://yann.lecun.com/exdb/mnist/

“Hello world” for deep learning

Keras provides data sets loading function: http://keras.io/datasets/

Keras





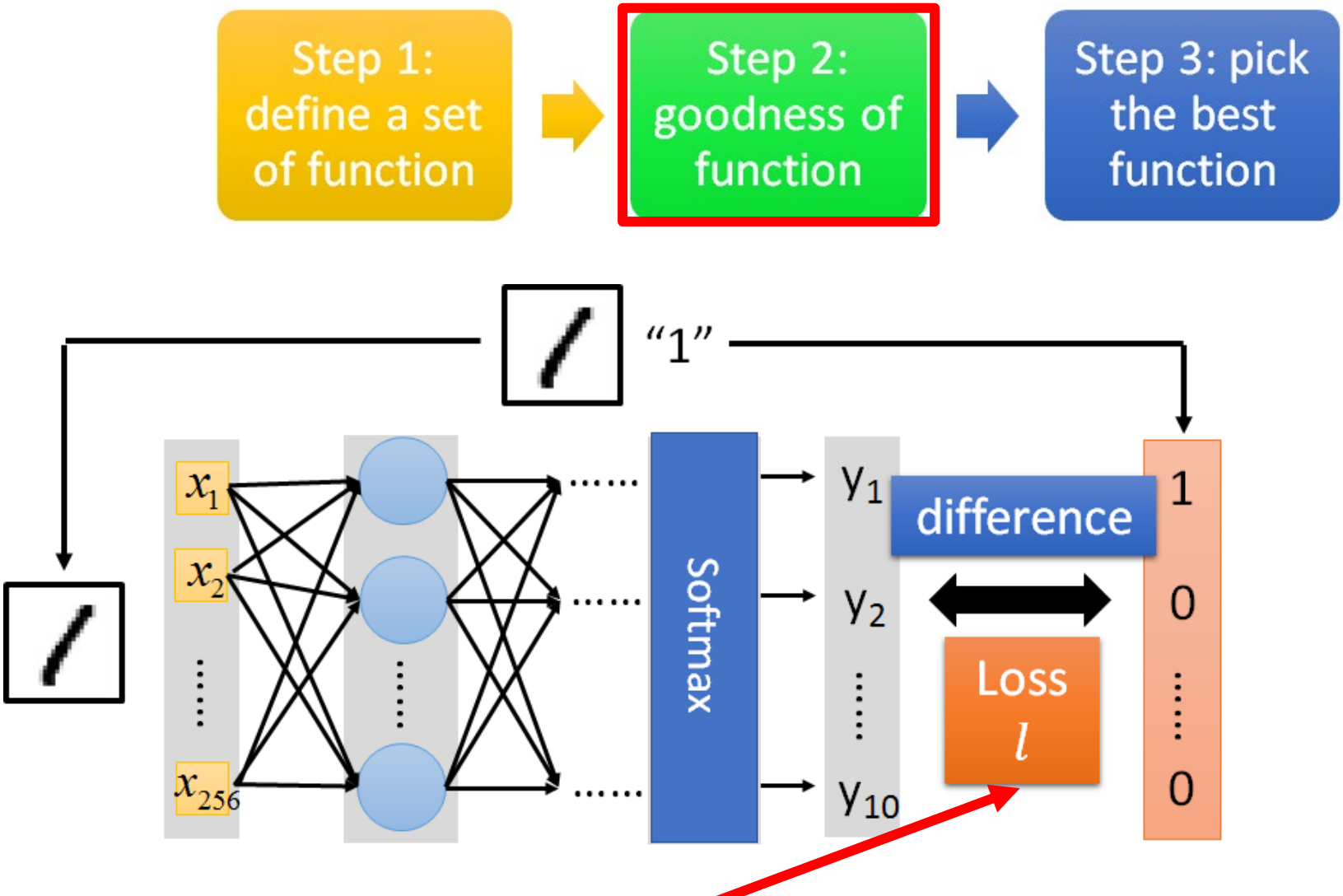

```
model = Sequential()
```

```
model.add( Dense( input_dim=28*28,
                  output_dim=500 ))
model.add( Activation('sigmoid') )
```

```
model.add( Dense( output_dim=500 ) )
model.add( Activation('sigmoid') )
```

```
model.add( Dense(output_dim=10 ) )
model.add( Activation('softmax') )
```

Keras




```
model.compile(loss='mse',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

Keras

Step 3.1: Configuration

```
model.compile(loss='mse',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

$$w \leftarrow w - \eta \partial L / \partial w$$

0.1

Step 3.2: Find the optimal network parameters

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```

Training data (Images)

Labels (digits)

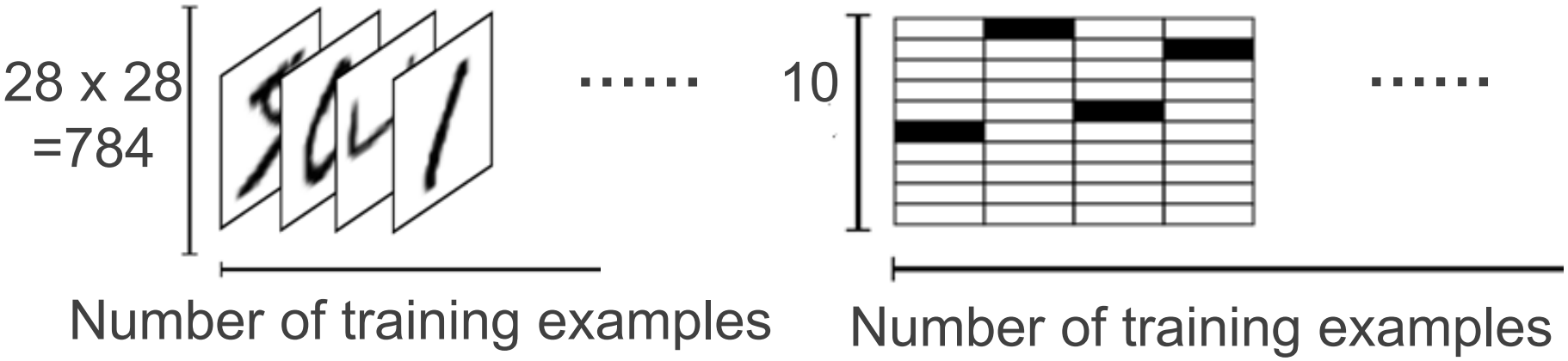
Keras

Step 3.2: Find the optimal network parameters

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```

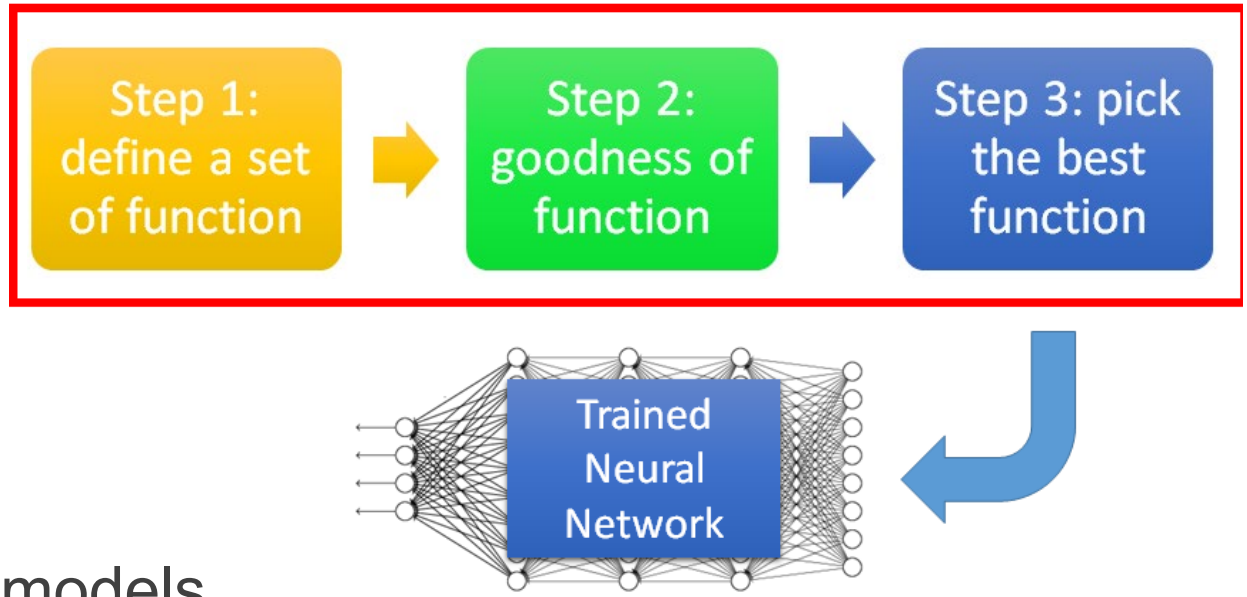
numpy array

numpy array



https://www.tensorflow.org/versions/r0.8/tutorials/mnist/beginners/index.html

Keras



Save and load models

http://keras.io/getting-started/faq/#how-can-i-save-a-keras-model

How to use the neural network (testing):

case 1:

```
score = model.evaluate(x_test,y_test)
print('Total loss on Testing Set:', score[0])
print('Accuracy of Testing Set:', score[1])
```

case 2:

```
result = model.predict(x_test)
```