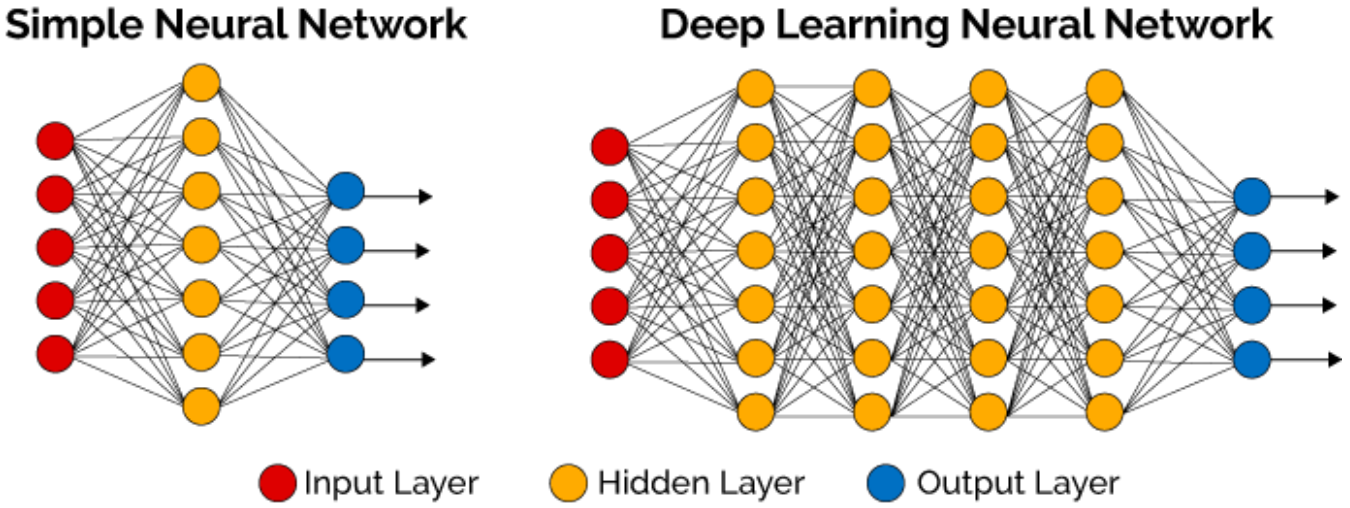
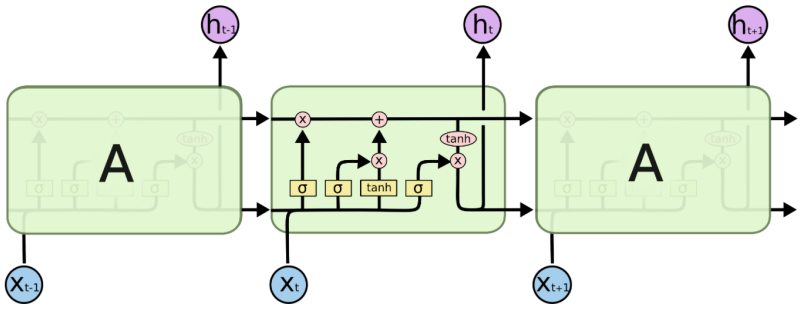
DSCI 552, Machine Learning for Data Science

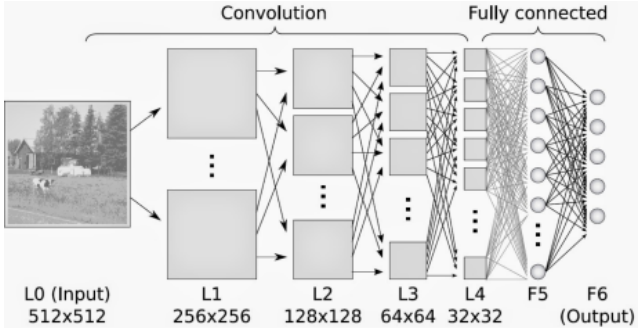
University of Southern California

M. R. Rajati, PhD

Lesson 10
Neural Networks and Deep Learning







Introductory Remark

In this lecture, we use matrix and vector notations. Matrices and vectors are shown by bold letters.

Vectors of functions are shown with bold letters (e.g. **f**), and unless otherwise stated, act on vectors element-wise.

Perceptron

Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0 \\ -1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0 \end{cases}$$

This is called a *step function*, which reads:

- the output is 1 if "$\boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0$" is true, and the output is -1 if instead "$\boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0$" is true

Perceptron

Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0 \\ -1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0 \end{cases}$$

By convention, the two classes are +1 or -1.

Perceptron

Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0 \\ -1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0 \end{cases}$$

- Often these parameters are called **weights**.
- $\boldsymbol{\beta}=(\beta_1, \beta_2, \ldots, \beta_p)$

Perceptron

Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, \boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0 \\ -1, \boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0 \end{cases}$$

By convention, ties are broken in favor of the positive class.

- If "$\boldsymbol{\beta}^T\mathbf{x} + \beta_0$ " is exactly 0, output +1 instead of -1.

Perceptron

The $\boldsymbol{\beta}$ parameters are unknown. This is what we have to learn.

$$f(\mathbf{x}) = \begin{cases} 1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0 \\ -1, & \boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0 \end{cases}$$

In the same way that linear regression learns the slope parameters to best fit the data points, perceptron learns the parameters to best separate the instances.

Learning the Weights

The perceptron algorithm learns the weights by:

1. Initialize all weights **β** to 0 or randomly.
2. Iterate through the training data. For each training instance, classify the instance.
    a) If the prediction (the output of the classifier) was correct, don't do anything. (It means the classifier is working, so leave it alone!)
    b) If the prediction was wrong, modify the weights by using the **update rule**.
3. Repeat step 2 some number of times (more on this later).

Learning the Weights

What does an **update rule** do?

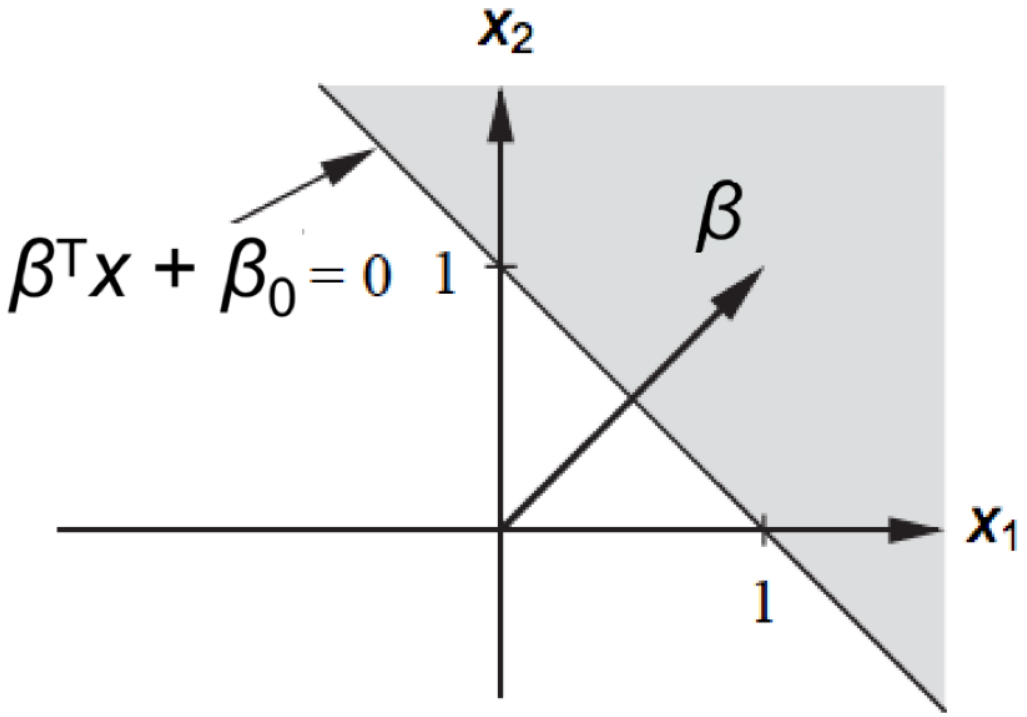
- If the classifier predicted an instance was negative but it should have been positive…

Currently: $\boldsymbol{\beta}^T\mathbf{x} + \beta_0 < 0$ Want: $\boldsymbol{\beta}^T\mathbf{x} + \beta_0 \geq 0$

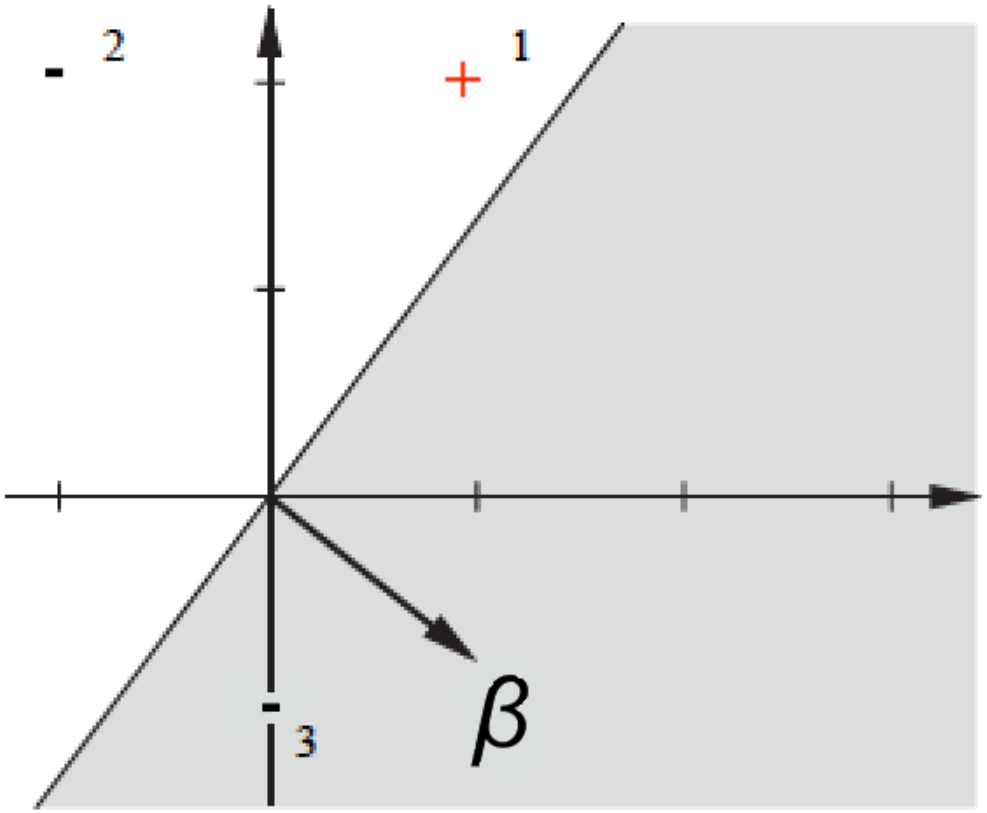
  - Adjust the weights $\boldsymbol{\beta}$ so that this function value moves toward positive
- If the classifier predicted positive but it should have been negative, shift the weights so that the value moves toward negative.

Structure of A Perceptron: Example

$$\beta_1 = 1 \quad \beta_2 = 1 \quad \beta_0 = -1$$



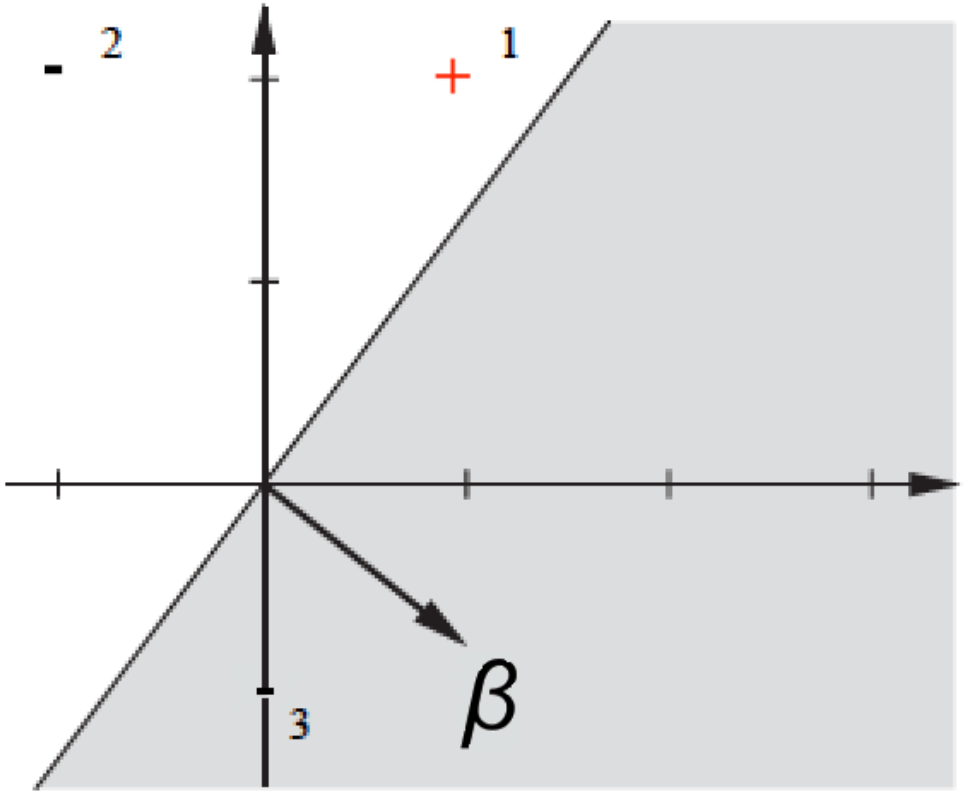
Learning the Weights



+ and - classes
Random initial weight:
$\boldsymbol{\beta}^T=[1, -0.8]$;

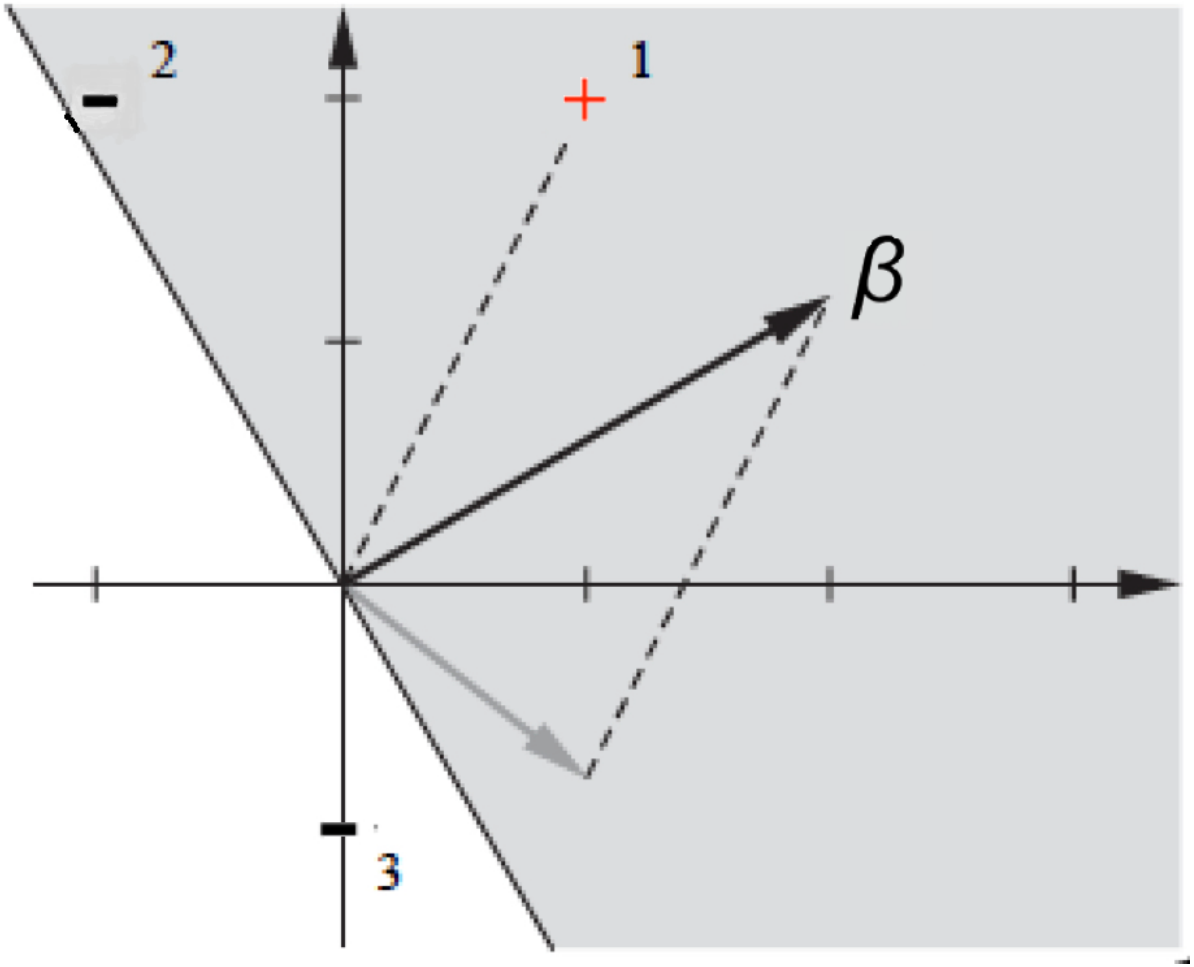
For now, assume that $\beta_0=0$ all the time.

Learning the Weights



The Perceptron classifies **x**(1) as - incorrectly, so the weights have to be changed.

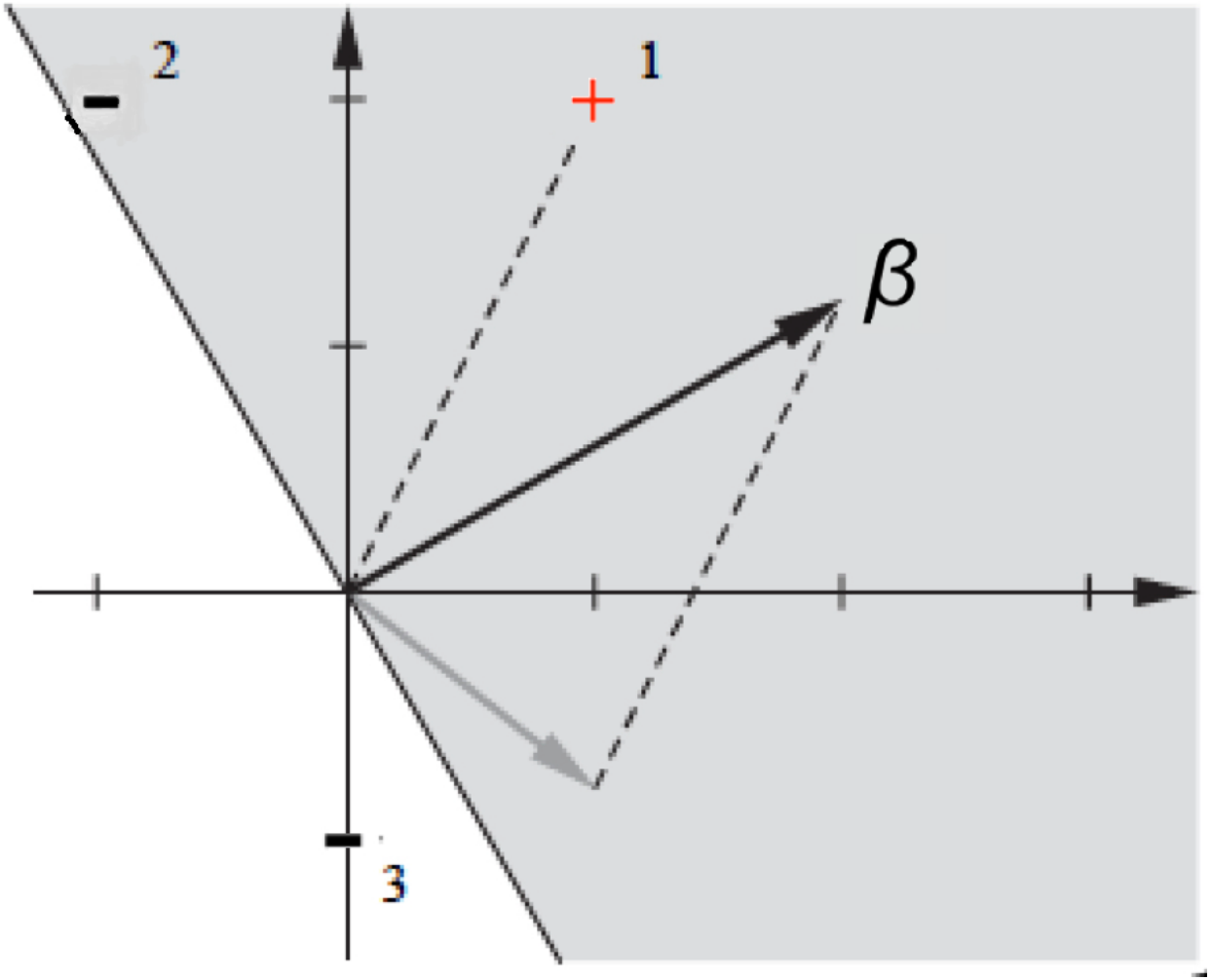
Learning the Weights



The weight vector has to move towards the misclassified vector.

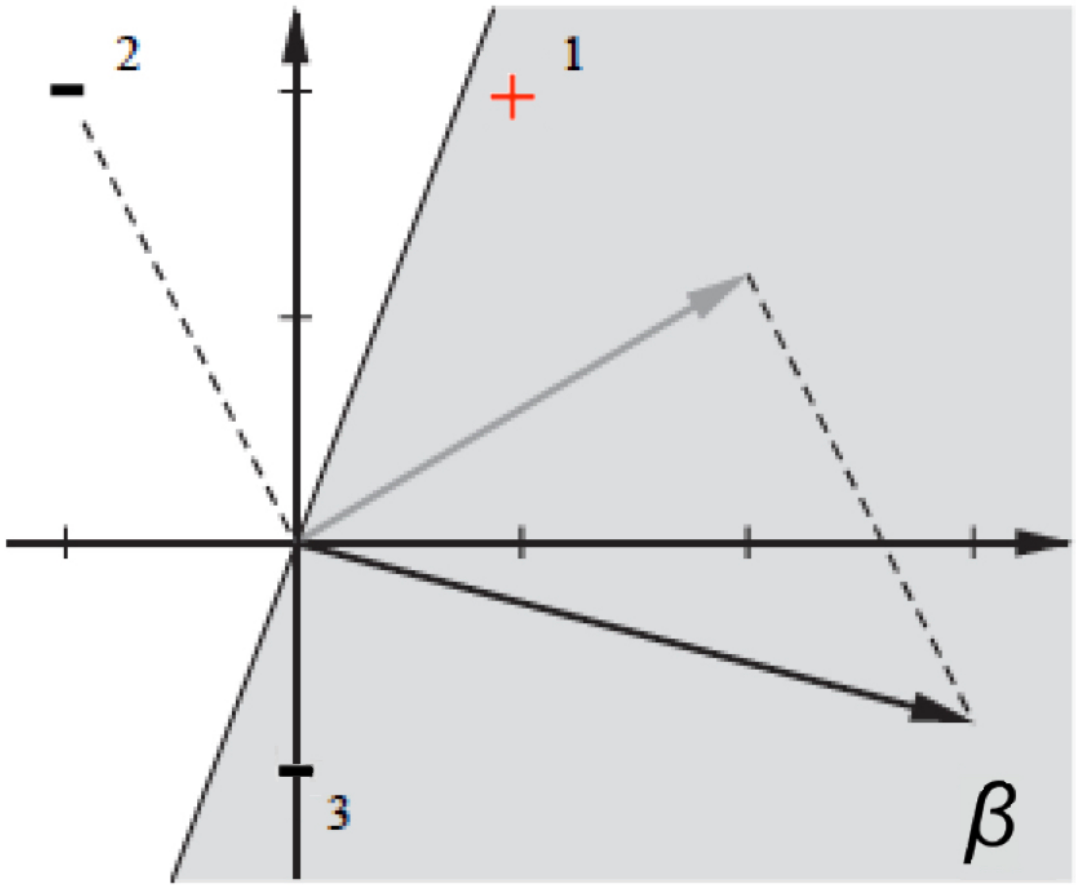
$\boldsymbol{\beta}^{new} = \boldsymbol{\beta}^{old} + \mathbf{x}(1)$

Learning the Weights



The Perceptron misclassifies **x**(2) as +.

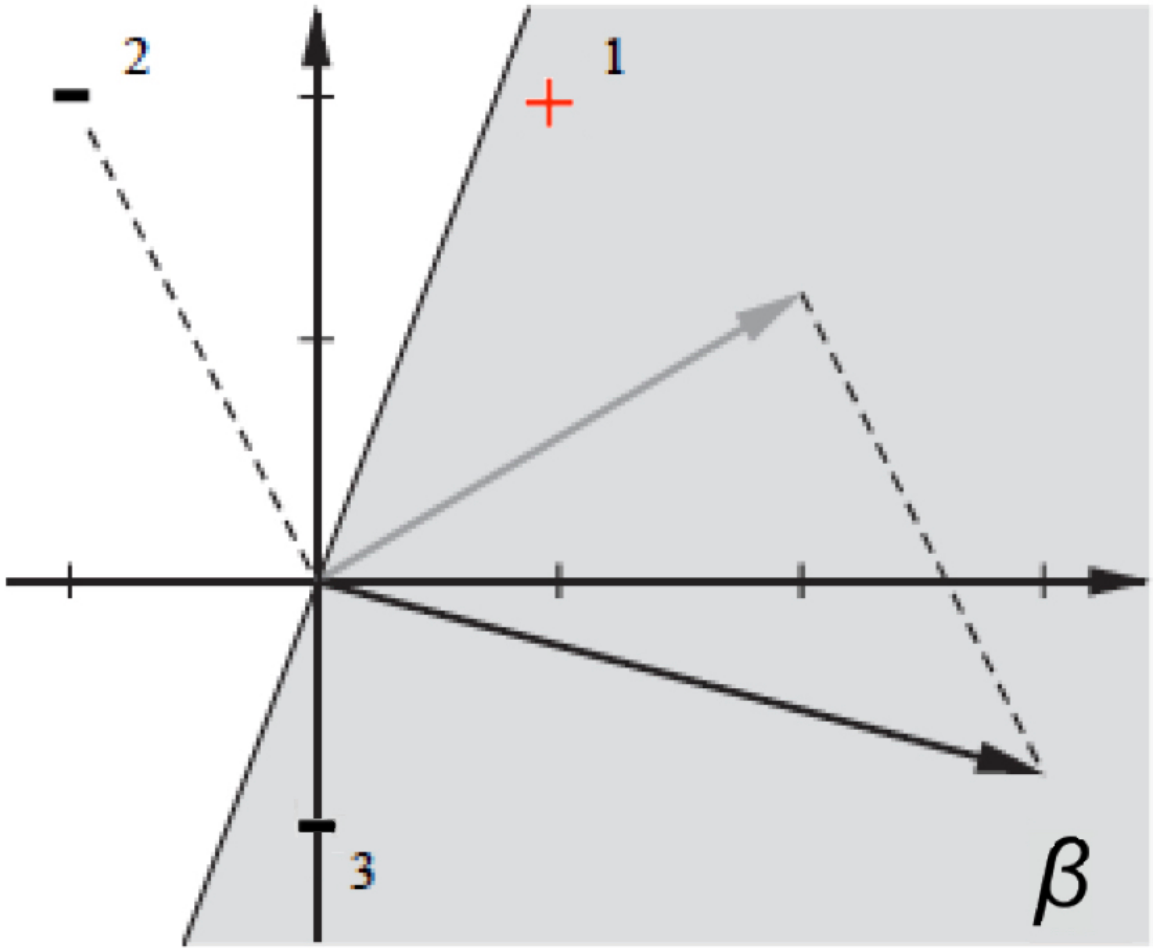
Learning the Weights



The weight vector has to move away from the misclassified vector.

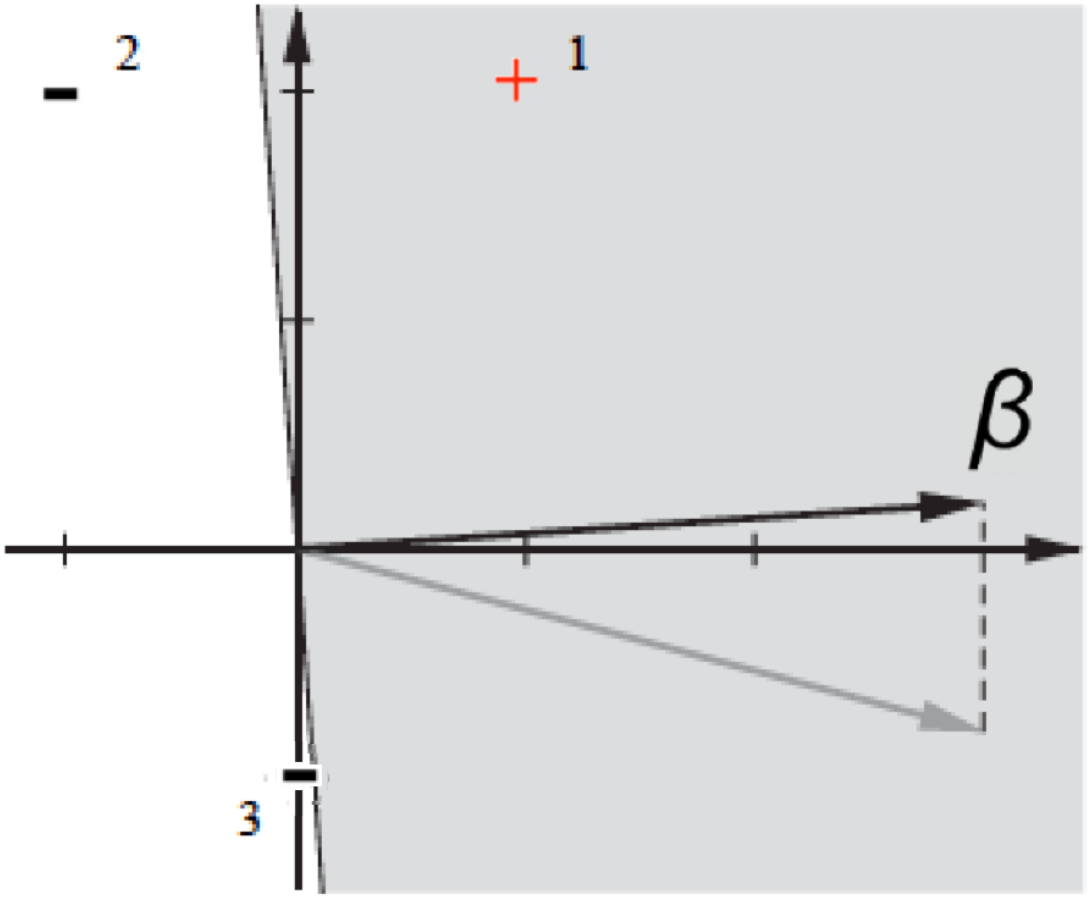
$\boldsymbol{\beta}^{new} = \boldsymbol{\beta}^{old} - \mathbf{x}(2)$

Learning the Weights



The Perceptron misclassifies **x**(3) as +.

Learning the Weights



The weight vector has to move away from **x**(3).

$\boldsymbol{\beta}^{new} = \boldsymbol{\beta}^{old} - \mathbf{x}(3)$

Learning the Weights

The perceptron update rule:

$$\boldsymbol{\beta}(i+1) = \boldsymbol{\beta}(i) + 0.5[y(i)-f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i))]\,\mathbf{x}(i)$$

Learning the Weights

The perceptron update rule:

$$\boldsymbol{\beta}(i+1) = \boldsymbol{\beta}(i) + 0.5[y(i) - f(\boldsymbol{\beta}^T(i)\mathbf{x}(i) + \beta_0(i))]\,\mathbf{x}(i)$$

If $y(i) = f(\boldsymbol{\beta}^T(i)\mathbf{x}(i) + \beta_0(i))$. i.e., if Perceptron correctly classifies the instance $\mathbf{x}(i)$, the weights will not be updated.

Learning the Weights

The perceptron update rule:

$\boldsymbol{\beta}(i+1) = \boldsymbol{\beta}(i) + 0.5[y(i)-f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i))]\,\mathbf{x}(i)$

If $y(i) = 1$ and $f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i)) = -1$, then $.5[y(i)-f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i))]=1$, so the weights will move towards $\mathbf{x}(i)$ to make $\boldsymbol{\beta}^T\mathbf{x}(i) + \beta_0$ more positive and $\mathbf{x}(i)$ is more likely to be classified correctly.

Learning the Weights

The perceptron update rule:

$\boldsymbol{\beta}(i+1) = \boldsymbol{\beta}(i) + 0.5[y(i)-f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i))]\,\mathbf{x}(i)$

If $y(i) =-1$ and $f(y(i)-f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i)))=+1$, then $.5[y(i)-f(\boldsymbol{\beta}^T(i)\mathbf{x}(i)+ \beta_0(i))]=-1$, so the weights will move away from $x(i)$ to make $\boldsymbol{\beta}^T\mathbf{x}(i) + \beta_0$ more negative and $\mathbf{x}(i)$ is more likely to be classified correctly.

Learning the Weights

The perceptron update rule:

$\boldsymbol{\beta}(i+1) = \boldsymbol{\beta}(i) + 0.5e(i)\,\mathbf{x}(i)$

where $e(i) = e(i) = y(i) - f(\boldsymbol{\beta}^T(i)\mathbf{x}(i) + \beta_0(i))$ is the *error* of the model in classifying $\mathbf{x}(i)$.

Learning the Weights

For the bias term we have:

$\beta_0(i+1) = \beta_0(i) + .5[e(i)]$

This means if $e(i) = y(i) - f(\boldsymbol{\beta}^T(i)\mathbf{x}(i) + \beta_0(i))$ is positive, increase the bias term and if it is negative, decrease it, i.e.:

Move the decision boundary up or down to correct the classifier

Learning the Weights

Move the decision boundary up or down to correct the classifier.

Example: $\beta_1 = 1$, $\beta_2 = -1$, $\beta_0 = 0$. $\mathbf{x}(1) = [0\ 1]^T$ and $y(1) = +1$.

Remark

So far, we have used **β** for showing the parameters of the decision hyperplane, which is a common notation for regression.

Using **w** is way more common for showing weights in neural networks. Let's switch to **w**.

The parameter β_0 is often called a bias, and is more commonly shown as b.

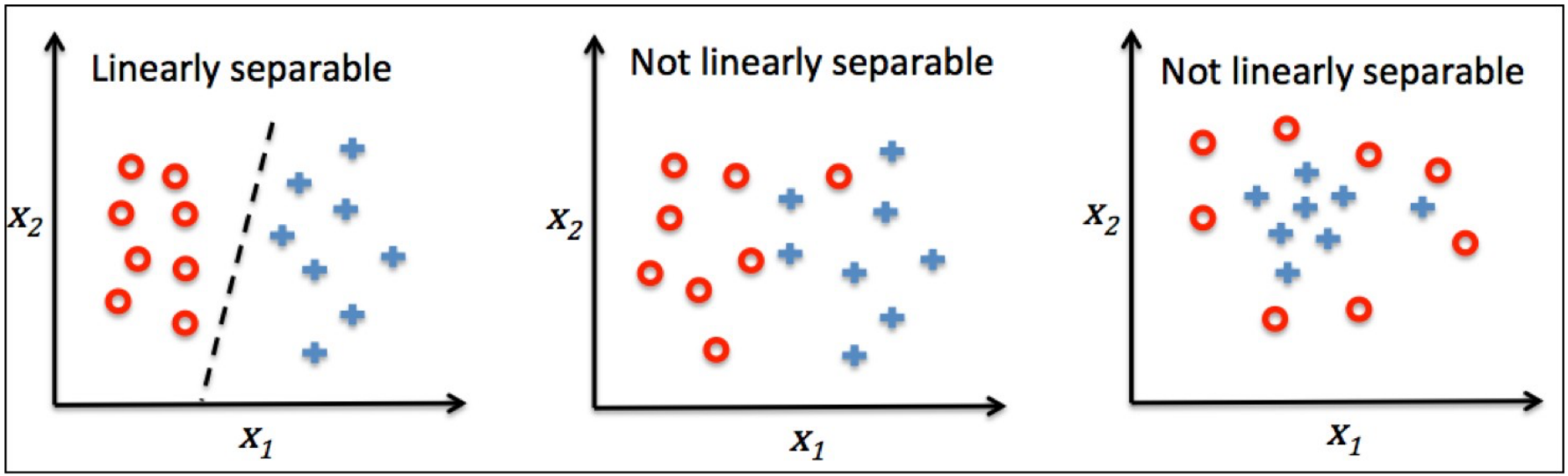
Perceptron vs. SVC

Note that Perceptron and SVC are both linear classifiers. However, SVC's optimization needs all data to yield a classifier, so it is a *batch* algorithm.

On the other hand, perceptron *adapts* its weights online, based on the data points that are presented to it. Thus it is an *online* algorithm.

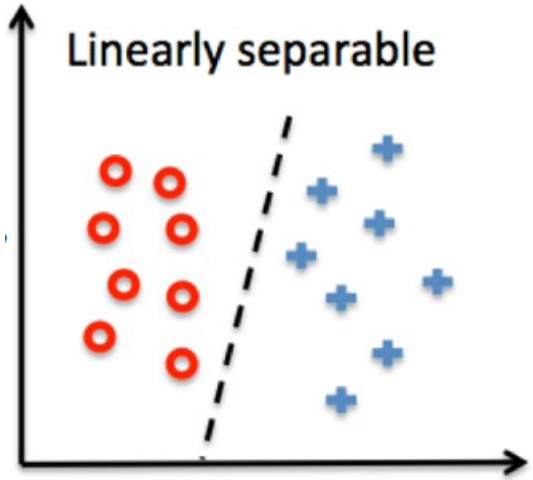
Linear Separability

RECALL: The training instances are **linearly separable** if there exists a hyperplane that will separate the two classes.



Linear Separability

If the training instances are linearly separable, the perceptron algorithm will eventually find weights **W** such that the classifier gets everything correct.



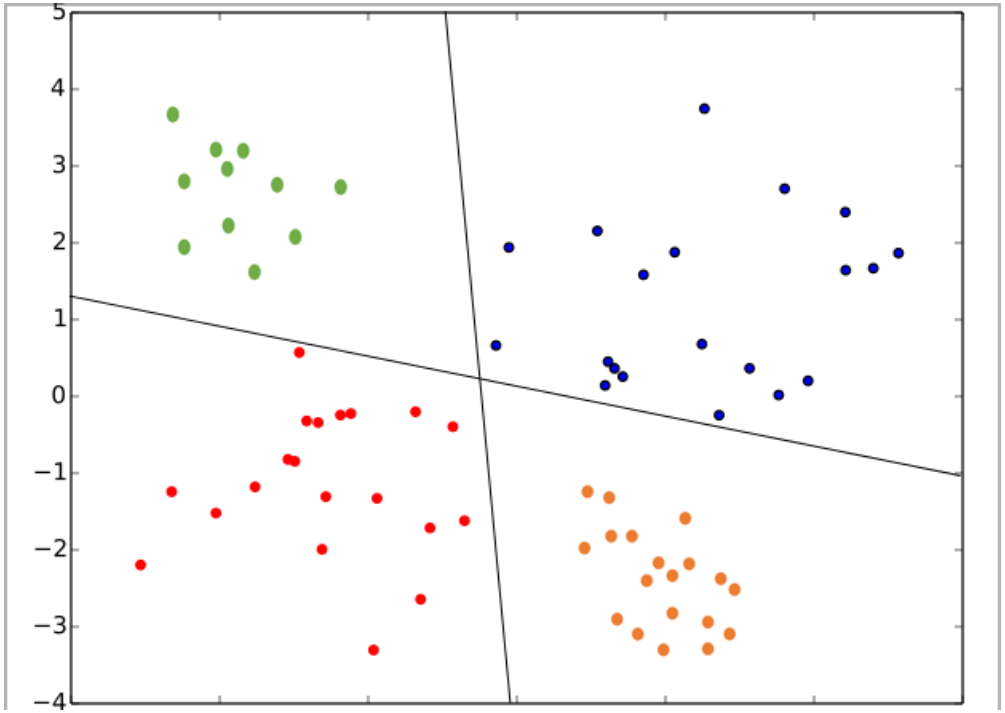
Linear Separability

If the training instances are not linearly separable, the classifier will always get some predictions wrong.

- You need to implement some type of **stopping criteria** for when the algorithm will stop making updates, or it will run forever.
- Usually this is specified by running the algorithm for a maximum number of **iterations** or **epochs**.

Multi-class Perceptron: Architecture

Using S hyperplanes, one can partition the space $\mathbf{R}^p$ to 2^S regions. Each region can represent a class. For $p = 2$ and $S = 2$:



Multi-class Perceptron

The weights associated with each hyperplane can be represented as the columns of a *weight matrix* **W**:

$\mathbf{W} = [\mathbf{w}_1|\mathbf{w}_2|\ldots|\mathbf{w}_S]$

The biases can be augmented in a vector **b**:
$\mathbf{b} = [b_1\, b_2 \ldots b_S]^T$

Multi-class Perceptron

A maximum of $K = 2^S$ classes can be encoded as vectors with S elements of the form:

$\mathbf{y} = [y_1\, y_2 \ldots y_S]^T \; y_k \in \{-1, 1\}$

In other words, each of the S perceptrons is in charge of one of the elements of $\mathbf{y}$.

Multi-class Perceptron: Learning Rule

The weights can be updated for multiclass perceptron. Assume that $\mathbf{e}(i)$ *is the vector of errors at epoch i*

$$\mathbf{W}(i+1) = \mathbf{W}(i) + 0.5\mathbf{x}(i)\mathbf{e}^T(i)$$

As well as the biases

$$\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + 0.5\mathbf{e}^T(i)$$

Multi-class Perceptron: Learning Rule

$$\mathbf{W}(i+1) = \mathbf{W}(i) + 0.5\mathbf{x}(i)\mathbf{e}^T(i)$$

$$\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + 0.5\mathbf{e}^T(i)$$

$$\mathbf{e}(i) = \mathbf{y}(i) - \mathbf{f}(\mathbf{W}^T(i)\mathbf{x}(i)+\mathbf{b}(i))$$

$\mathbf{y}(i)$ is the actual label vector for x(i), and $\mathbf{W}^T(i)\mathbf{x}(i)+\mathbf{b}(i)$ is a vector of S linear combinations of features of $\mathbf{x}(i)$, whose S elements are fed to S step functions, symbolically shown as **f**.

Multi-class Perceptron: Learning Rule

$$\mathbf{W}(i+1) = \mathbf{W}(i) + 0.5\mathbf{x}(i)\mathbf{e}^T(i)$$

$$\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + 0.5\mathbf{e}^T(i)$$

We wish to show that the above rule is nothing but application of the Perceptron Learning Rule to each of the S perceptrons:

Multi-class Perceptron: Learning Rule

$$\mathbf{W}(i+1) = \mathbf{W}(i) + 0.5\mathbf{x}(i)\mathbf{e}^T(i)$$

$$\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + 0.5\mathbf{e}^T(i)$$

Multi-class Perceptron: Learning Rule

Example: Assume that biases are kept zero and

$\mathbf{w}_1(0) = [1\ 1]^T$
$\mathbf{w}_2(0) = [1\ -1]^T$

Assume that $\mathbf{x}(1) = [1\ 0]^T$ is in a class encodes as $C_1 = [1\ -1]^T$ and $\mathbf{x}(2) = [0\ -1]^T$ is in a class encoded as $C_0 = [-1\ -1]^T$. Update the weights according to the perceptron rule.

Learning Rate

Let's make a modification to the update rule:

$$\mathbf{W}(i+1) = \mathbf{W}(i) + \alpha \mathbf{x}(i)\mathbf{e}^T(i)$$

$$\mathbf{b}^T(i+1) = \mathbf{b}^T(i) + \alpha \mathbf{e}^T(i)$$

where α is called the **learning rate** or **step size**.

- When you update w_j to be more positive or negative, this controls the size of the change you make (or, how large a "step" you take).
- If $\alpha = 0.5$ (a common value), then this is the same update rule from the earlier slide.

Learning Rate

How to choose the step size?

- If α is too small, the algorithm will be slow because the updates won't make much progress.
- If α is too large, the algorithm will be slow because the updates will "overshoot" and may cause previously correct classifications to become incorrect.
- One choice: in first iterations choose a large learning rate and then reduce it $\alpha(i) = \alpha_0/(i+c)$

Learning Rate

See:

https://www.jeremyjordan.me/nn-learning-rate/

A Little Bit of History

- McCulloch and Pitts in 1943 contended that neurons with a binary threshold activation function can implement logical functions.
- In 1949, Hebb observed a working principle in neurons that inspired the invention of Perceptron Learning Rule.

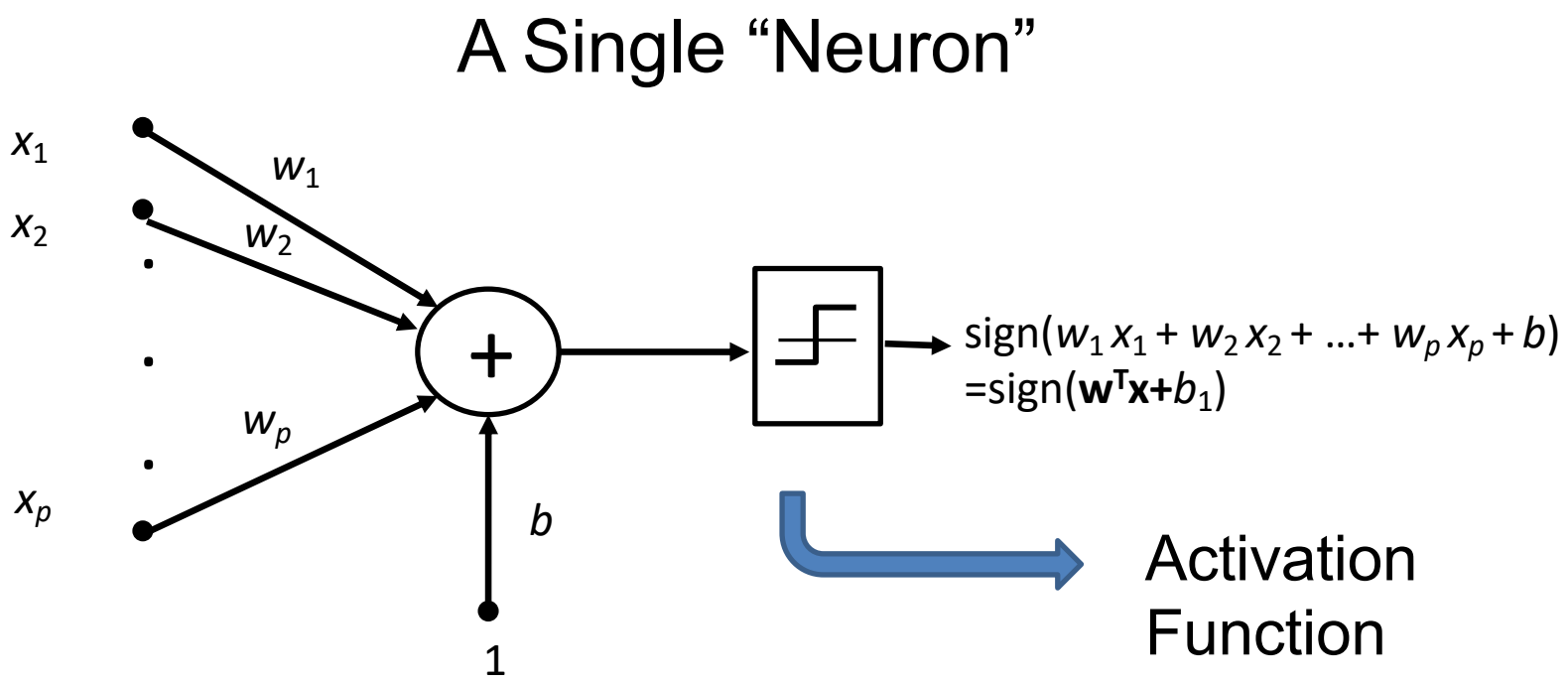
A Little Bit of History

- Hebb observed that, when two neurons fire together, the connection between the neurons is strengthened
- He concluded that this is one of the fundamental operations necessary for learning and memory.

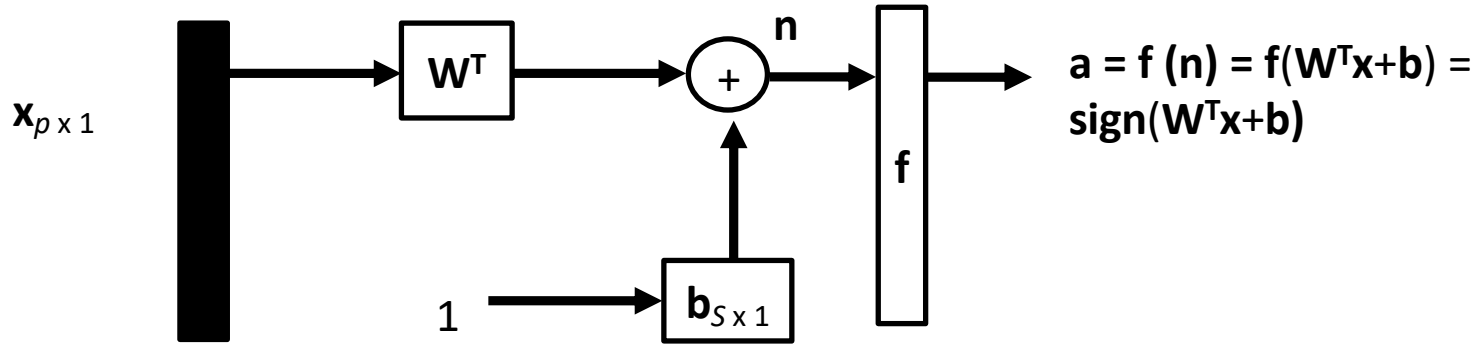
A Little Bit of History

- Rosenblatt, using the McCulloch-Pitts neuron and the findings of Hebb, went on to develop the first Perceptron Learning Rule.
- Perceptron could learn in the Hebbian sense through the weighting of inputs
- Perceptron was instrumental in the later formation of neural networks.

Perceptron as A Neural Architecture



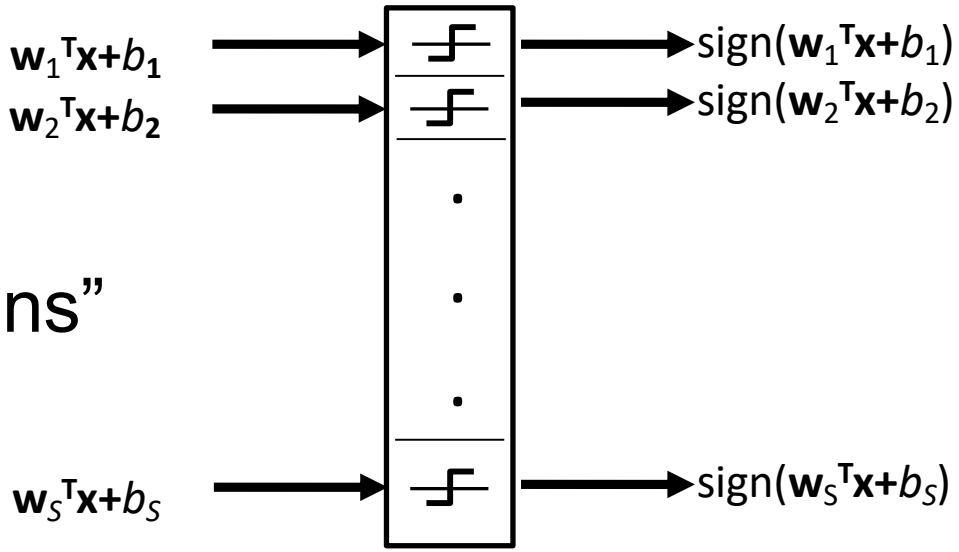
Perceptron as A Neural Architecture



Dimensions:

$\mathbf{W}_{p \times S}$

$\mathbf{W}^T{}_{S \times p}$



Multiple “Neurons”

Adaptation in Neural Systems

- The Perceptron Learning Rule acts like a simple neural system: it *adapts* its weights based on the *error* that is a result of *mismatch* between its output and the desired output
- This general process is the basis of supervised learning with a large class of *neural networks*.

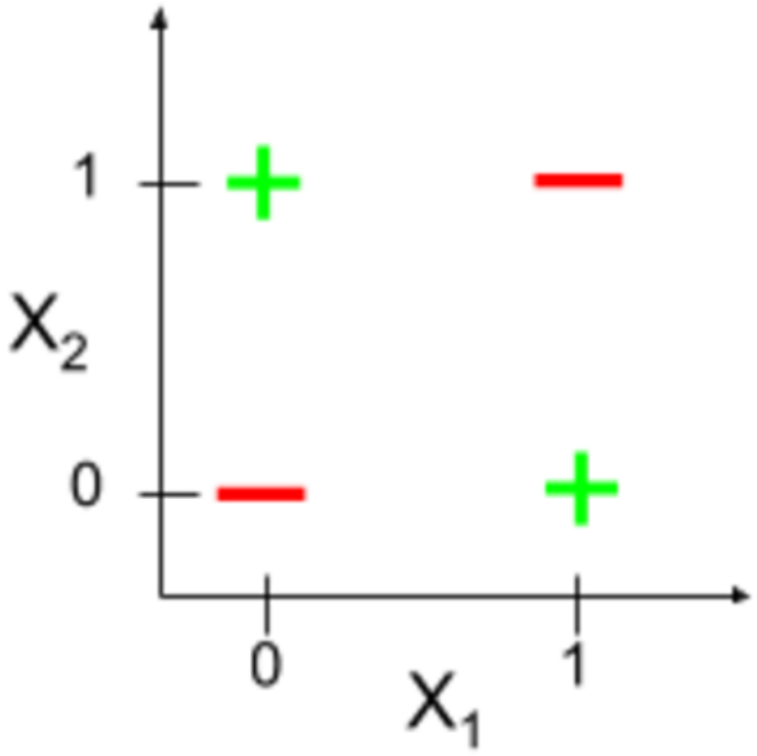
Caveat: Linear Separability

- If the training set is not separable, the Perceptron rule never converges.
- Naturally, it may have a large classification error.

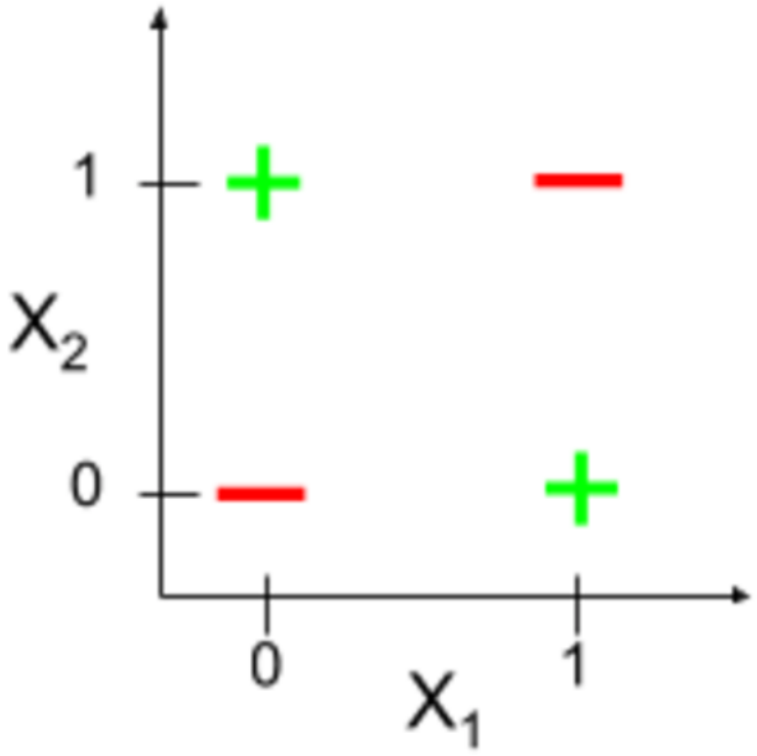
Solution?

- The general solution is to represent the data in a *feature space* where they are linearly separable.
- One idea is two use *layers* of perceptrons and adjust their weights to learn the feature representations

Example: XOR Problem

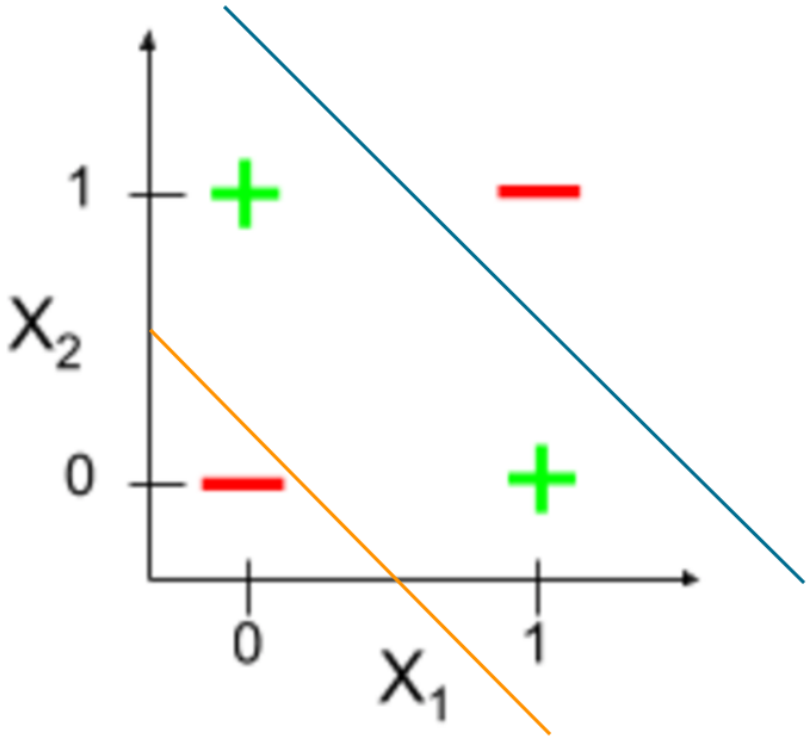


Example: XOR Problem

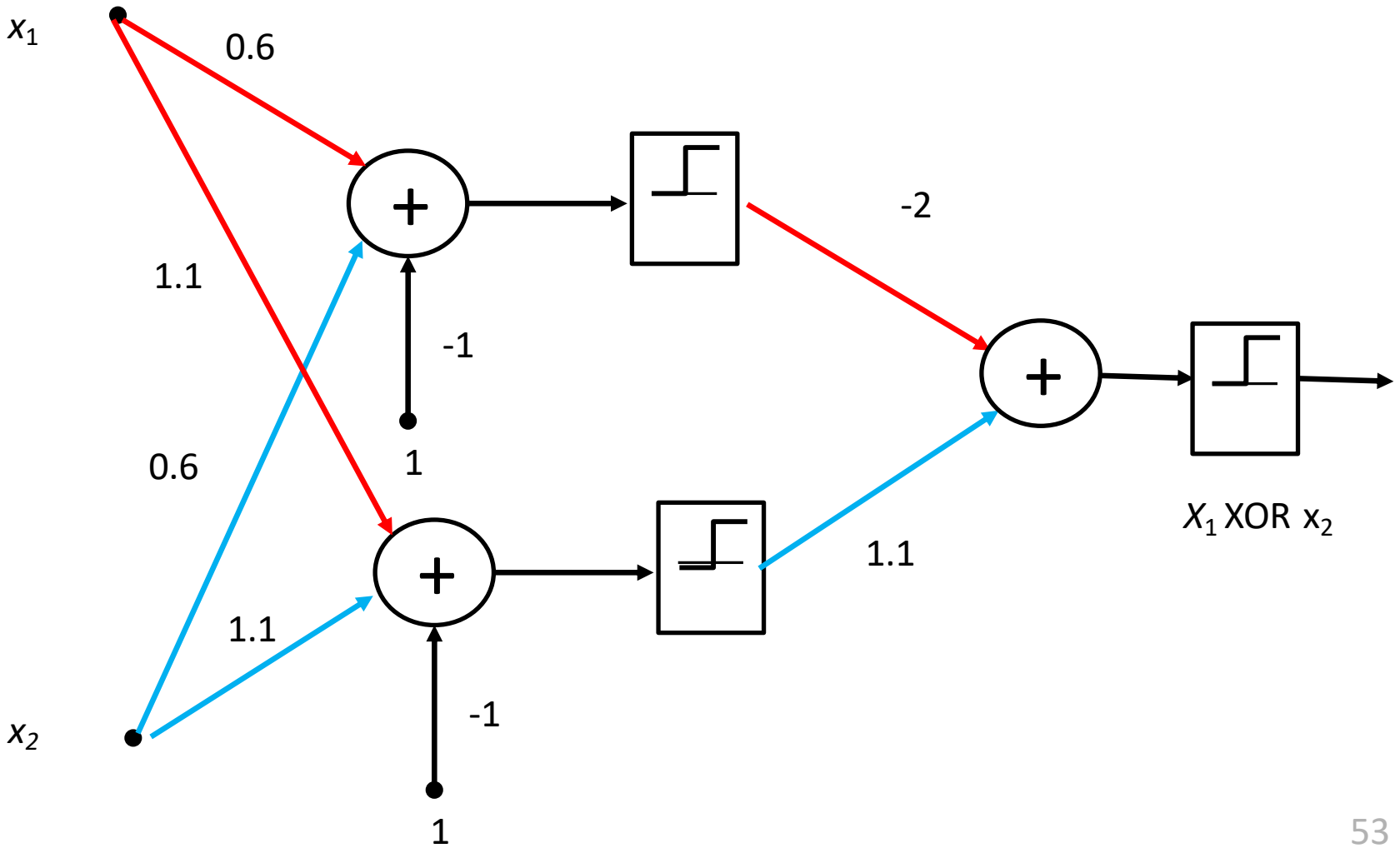


Inputs		Outputs
X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Example: XOR Problem



Example: XOR Problem



Caveat

- Finding the weights is ad-hoc and becomes complicated when implementing complex functions with multiple inputs.
- A principled way of representing and training multiple layers of Perceptrons is needed.

Multi-Layer Perceptron (MLP)

- MLP consists of multiple layers of Perceptrons
- Each layer may have a different number of neurons
- Different types of activation functions can be used, not just the sign (threshold) function

Multi-Layer Perceptron (MLP)

- MLPs can essentially be represented as nested functions:

  $\mathbf{g}(\mathbf{x}) = \mathbf{g}^{(M)}(\mathbf{g}^{(M-1)}(\ldots(\mathbf{g}^{(3)}(\mathbf{g}^{(2)}(\mathbf{g}^{(1)}(\mathbf{x}))))\ldots))$

- For example, a three layer perceptron can be represented as:
- $\mathbf{g}(\mathbf{x}) = \mathbf{g}^{(3)}(\mathbf{g}^{(2)}(\mathbf{g}^{(1)}(\mathbf{x})))$

Multi-Layer Perceptron (MLP)

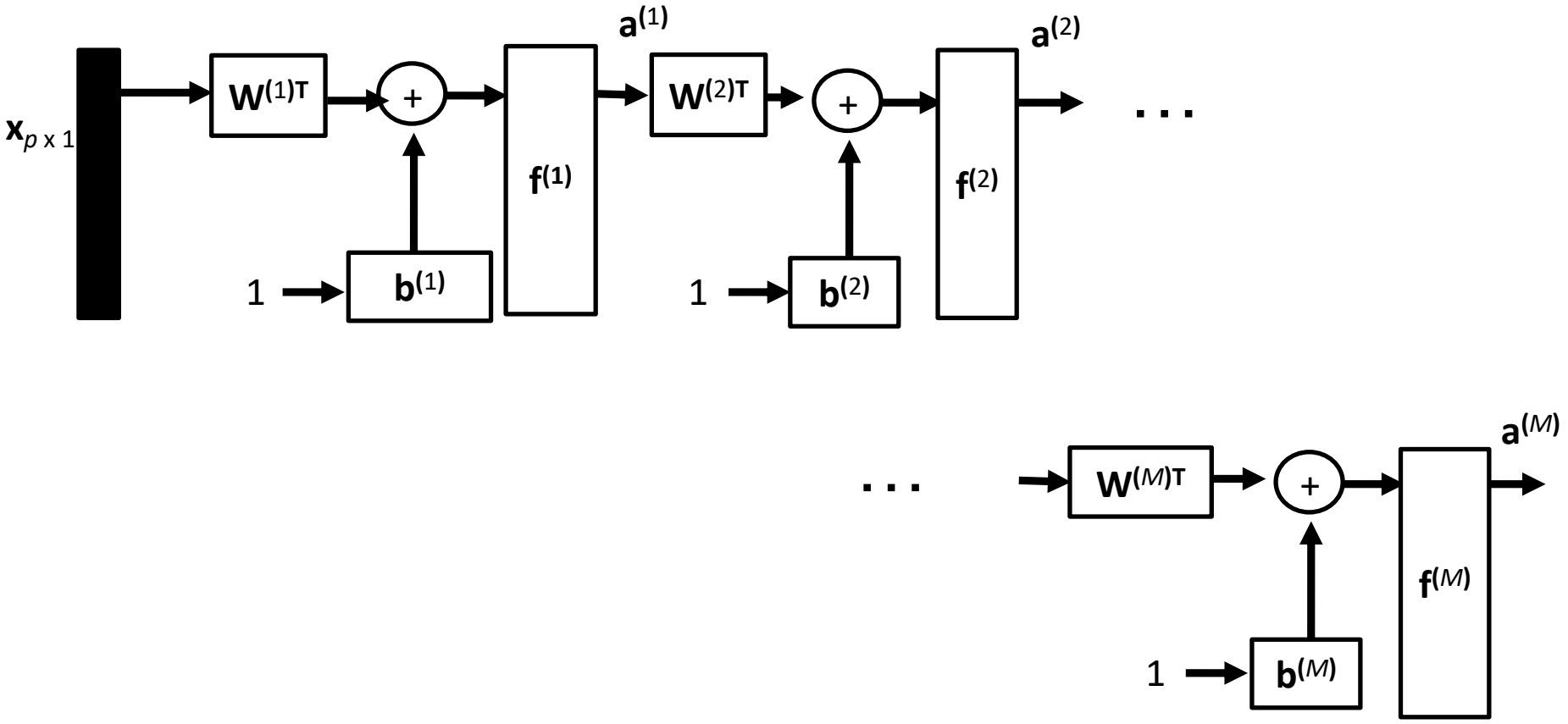
- Each $\mathbf{g}^{(i)}$ is a layer of neurons with its weight matrix $\mathbf{W}^{(i)}$ and bias vector $\mathbf{b}^{(i)}$ and activation function $\mathbf{f}^{(i)}$

Multi-Layer Perceptron (MLP)

- This means the output of the i^{th} layer, $\mathbf{a}^{(i)}$ is:

$$\mathbf{a}^{(i)} = \mathbf{f}^{(i)}(\mathbf{W}^{(i)\,T}\mathbf{a}^{(i-1)} + \mathbf{b}^{(i)}) = \mathbf{g}^{(i)}(\mathbf{a}^{(i-1)})$$

Multi-Layer Perceptron (MLP)



Multi-Layer Perceptron (MLP)

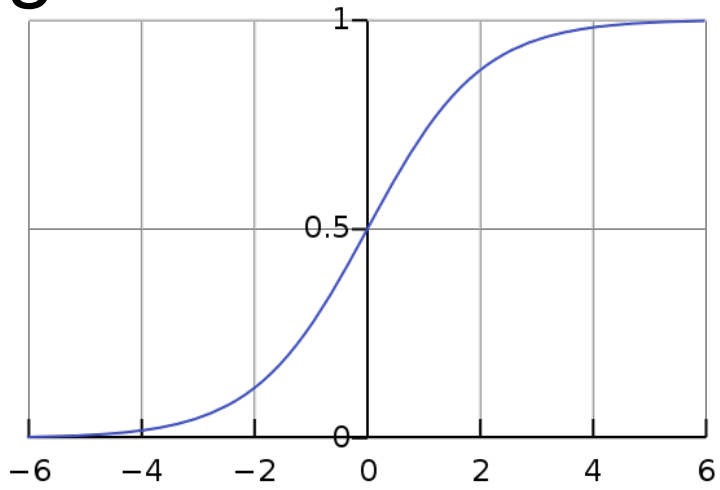
- MLPs are also called Feedforward Neural Networks
- Especially when the number of layers is large (e.g. M >10), they are called Deep Feedforward Neural Networks

Multi-Layer Perceptron (MLP)

•Weights in each layer can be learned so that the MLP is used for either classification or regression tasks.

Types of Activation Functions Used

- Traditionally, sigmoid functions were commonly used for both classification and regression tasks

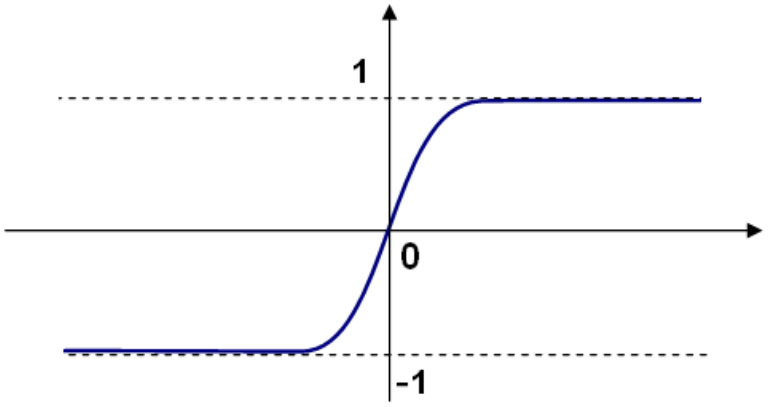


$$f(x) = \frac{1}{1 + e^{-x}}$$

$$= \frac{e^{x}}{e^{x} + 1}$$

Types of Activation Functions Used

- Traditionally, tanh sigmoid functions were commonly used for both classification and regression tasks

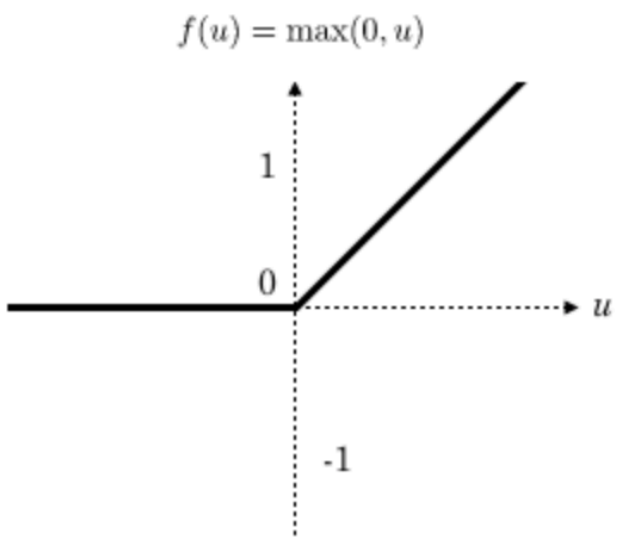


$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$= \frac{e^{2x} - 1}{e^{2x} + 1}$$

Types of Activation Functions Used

- In more modern practices for training deep neural networks, the Rectified Linear Unit (ReLu) is commonly used



MLPs are Powerful Tools for Regression

- It can be proven mathematically that any *smooth* function can be approximated with any precision using an MLP with only two layers:
  - A *hidden layer* of smooth nonlinear functions (e.g. sigmoids)
  - An *output layer* of smooth functions (even linear functions are adequate!)

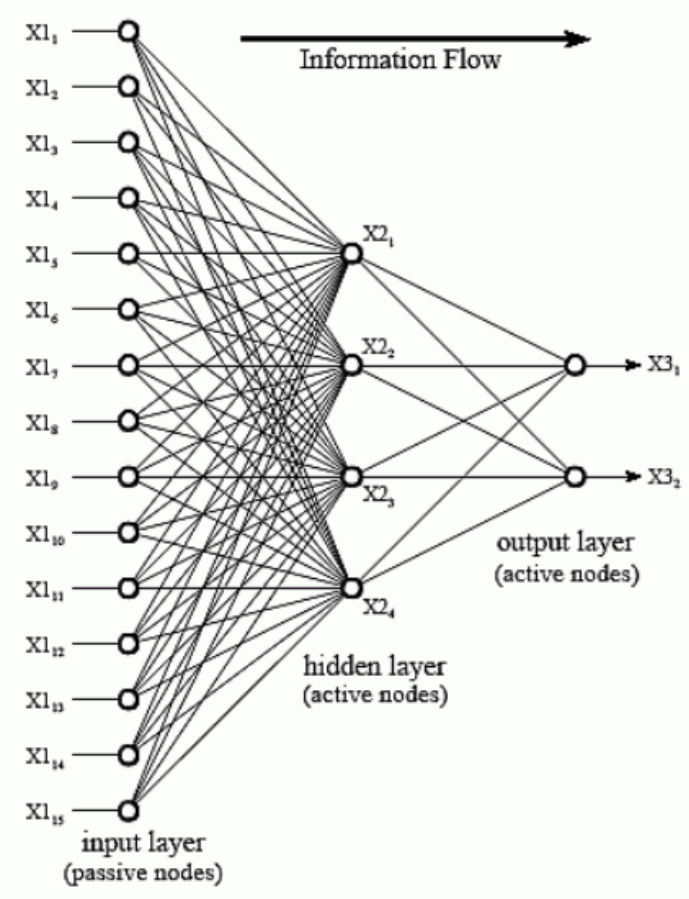
- This means that MLPs are *universal approximators!*

MLPs are Powerful Tools for Classification

- MLPs can be used for classification usually with at least:
  - A *hidden layer* of smooth nonlinear functions (e.g. sigmoids)
  - An *output layer* of linear functions followed by threshold functions.
  - Classes are usually encoded using
    - Binary encoding
    - One hot encoding
  - Therefore, desired outputs are vectors representing each class

Architectural Considerations

- Choice of depth (number of layers) of network
- Choice of width (number of neurons) of each layer



Architectural Considerations

- Deeper networks have
  - Far fewer units in each layer
  - Often generalize well to the test set
- They are often more difficult to train
  - Ideal network architecture must be found via experimentation guided by validation set error

There is No Free Lunch!

There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in training set.

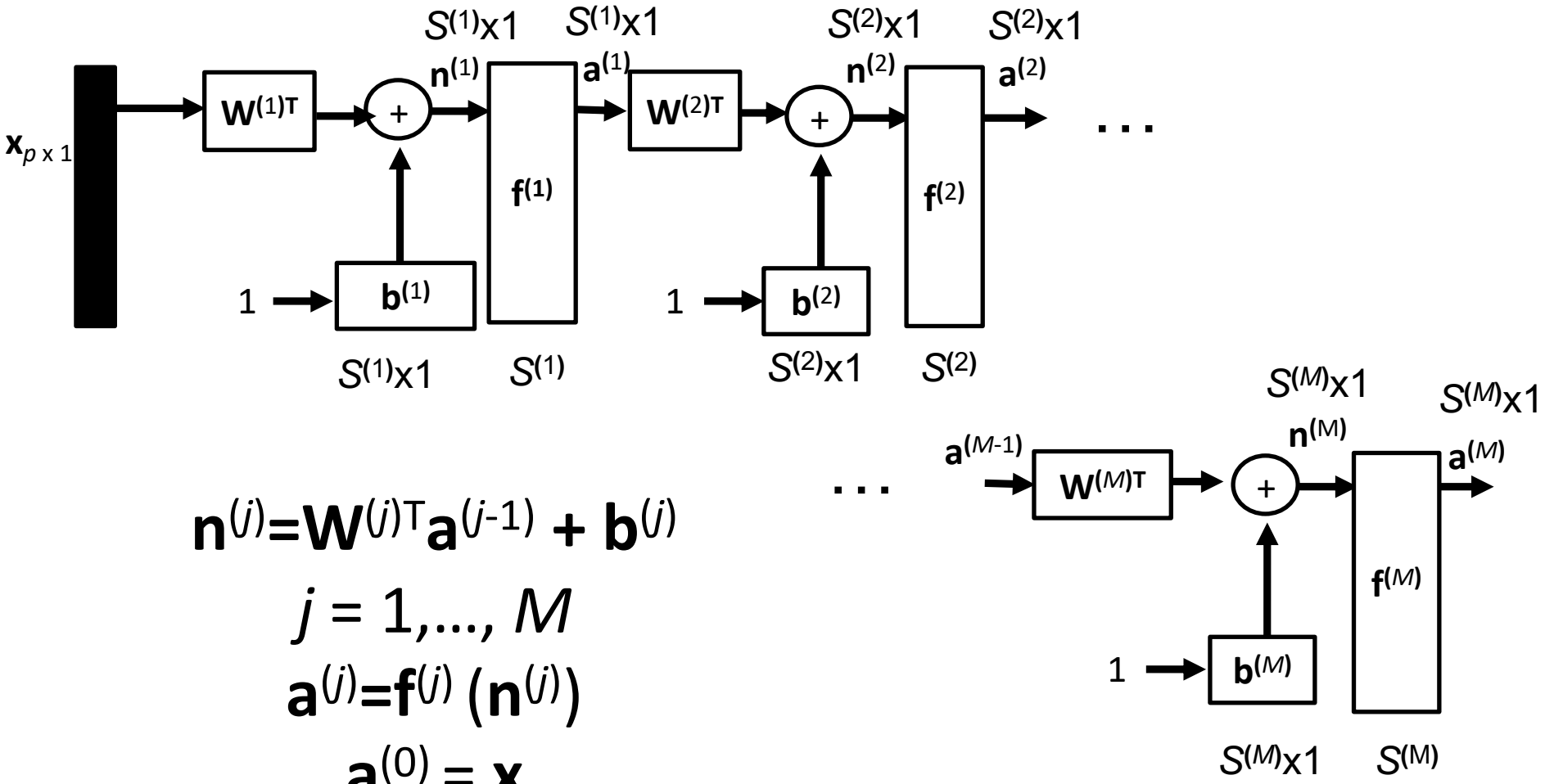
How to Train MLPs

- MLPs are trained using optimization algorithms.
- Optimization algorithms need calculation of Gradients to be numerically solved

How to Train MLPs

- *Backpropagation* (of errors) is a brilliant yet simple method to calculate gradients used to adjust the weights in MLPs to learn regression and classification tasks
- It was proposed by Rumelhart, Hinton, and Williams in 80's
- Still used in modern practices

Multi-Layer Perceptron (MLP)



$$\mathbf{n}^{(j)} = \mathbf{W}^{(j)T}\mathbf{a}^{(j-1)} + \mathbf{b}^{(j)}$$

$$j = 1, \ldots, M$$

$$\mathbf{a}^{(j)} = \mathbf{f}^{(j)}\left(\mathbf{n}^{(j)}\right)$$

$$\mathbf{a}^{(0)} = \mathbf{x}$$

Training Formulation for MLP

- Training Set
- {**x**(1), **y**(1)} , {**x**(2), **y**(2)} ,… , {**x**(N), **y**(N)}
- We would like to minimize some objective function *J* by finding "suitable" weight matrices for each layer.

Training Formulation for MLP

- A suitable objective function J is expected sum of square errors

$$J = E\left\{\sum_{i=1}^{S^M} e_i^2\right\} = E\left\{\sum_{i=1}^{S^M} (y_i - a_i^M)^2\right\}$$

$$= E\{\mathbf{e}^T\mathbf{e}\}$$

Training formulation for MLP

$$J = E\{\mathbf{e}^T\mathbf{e}\} \approx \frac{1}{N}\sum_{k=1}^{N}\mathbf{e}(k)^T\mathbf{e}(k)$$

$$\mathbf{e}(k) = \mathbf{y}(k) - \mathbf{a} = \mathbf{y}(k) - \mathbf{a}^{(M)}$$

$$= \mathbf{y}(k) - \mathbf{g}^{(M)}(\mathbf{g}^{(M-1)}(\ldots(\mathbf{g}^{(3)}(\mathbf{g}^{(2)}(\mathbf{g}^{(1)}(\mathbf{x}(k)))))\ldots))$$

Training formulation for MLP

$$J = E\{\mathbf{e}^T\mathbf{e}\} \approx \frac{1}{N}\sum_{k=1}^{N}\mathbf{e}(k)^T\mathbf{e}(k)$$

- We don't know *J* at each moment that some pair **x**(*k*) and **y**(*k*) becomes available.

Training formulation for MLP

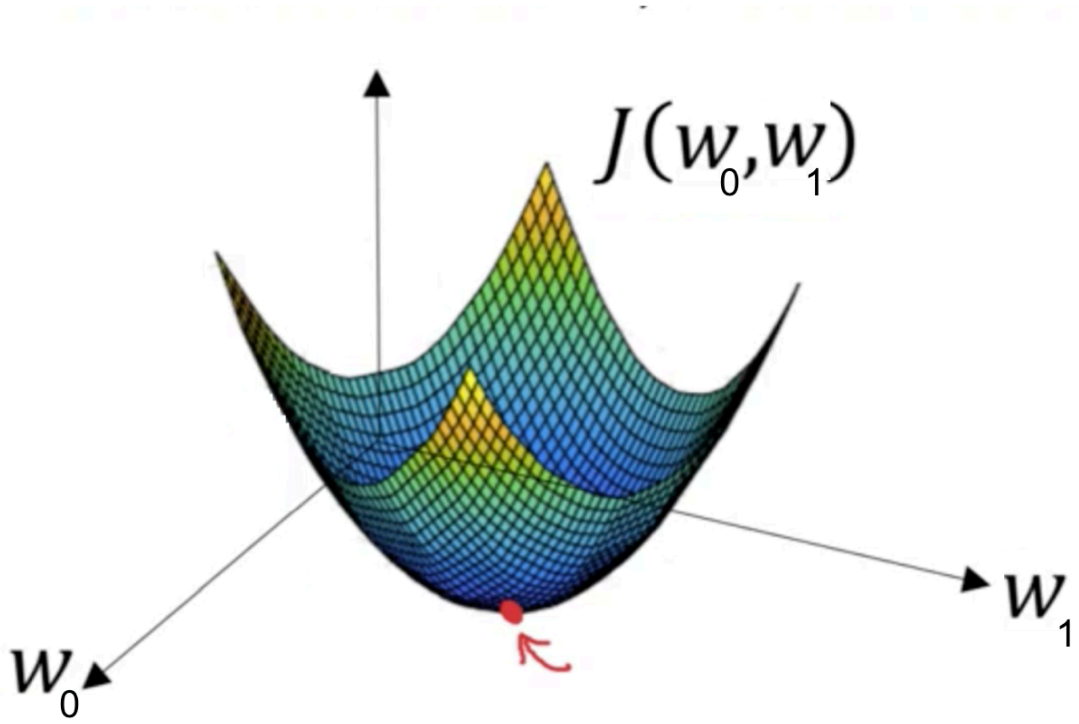
$$J = E\{\mathbf{e}^T\mathbf{e}\} \approx \frac{1}{N}\sum_{k=1}^{N}\mathbf{e}(k)^T\mathbf{e}(k)$$

- We approximate *J* based on the error of the network with respect to the pair **x**(*k*) and **y**(*k*), i. e. $J \approx \mathbf{e}(k)^T\mathbf{e}(k)$
- Alternatively, one can use a "mini-batch" of *L* pairs. For simplicity, we use only one pair.

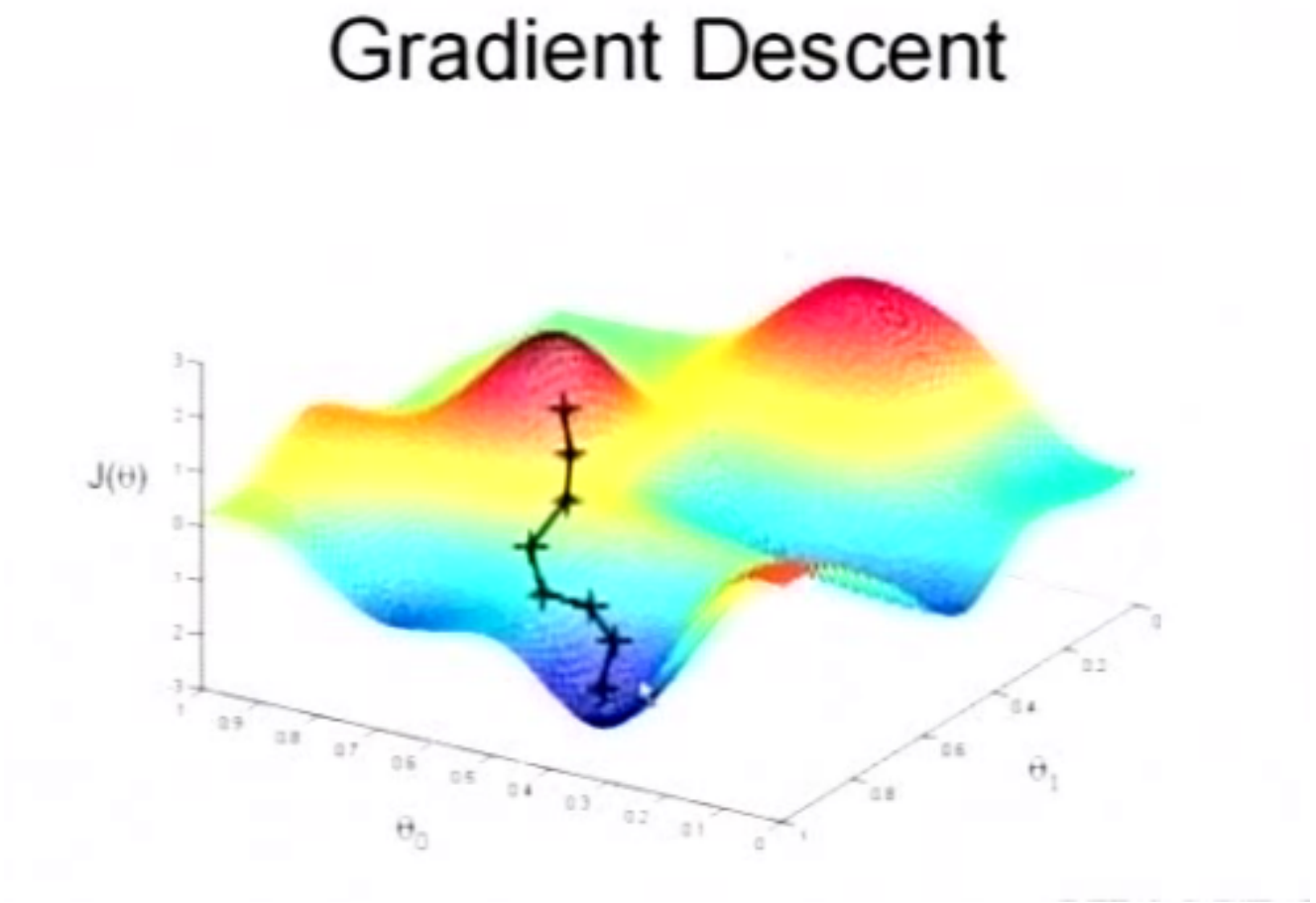
Approximate Gradient Descent

- $W_{ij}^{(m)}(k+1) = w_{ij}^{(m)}(k) - \alpha \partial J / \partial w_{ij}^{(m)}$
- $b_i^{(m)}(k+1) = b_i^{(m)}(k) - \alpha \partial J / \partial b_i^{(m)}$

Gradient Descent Formulation



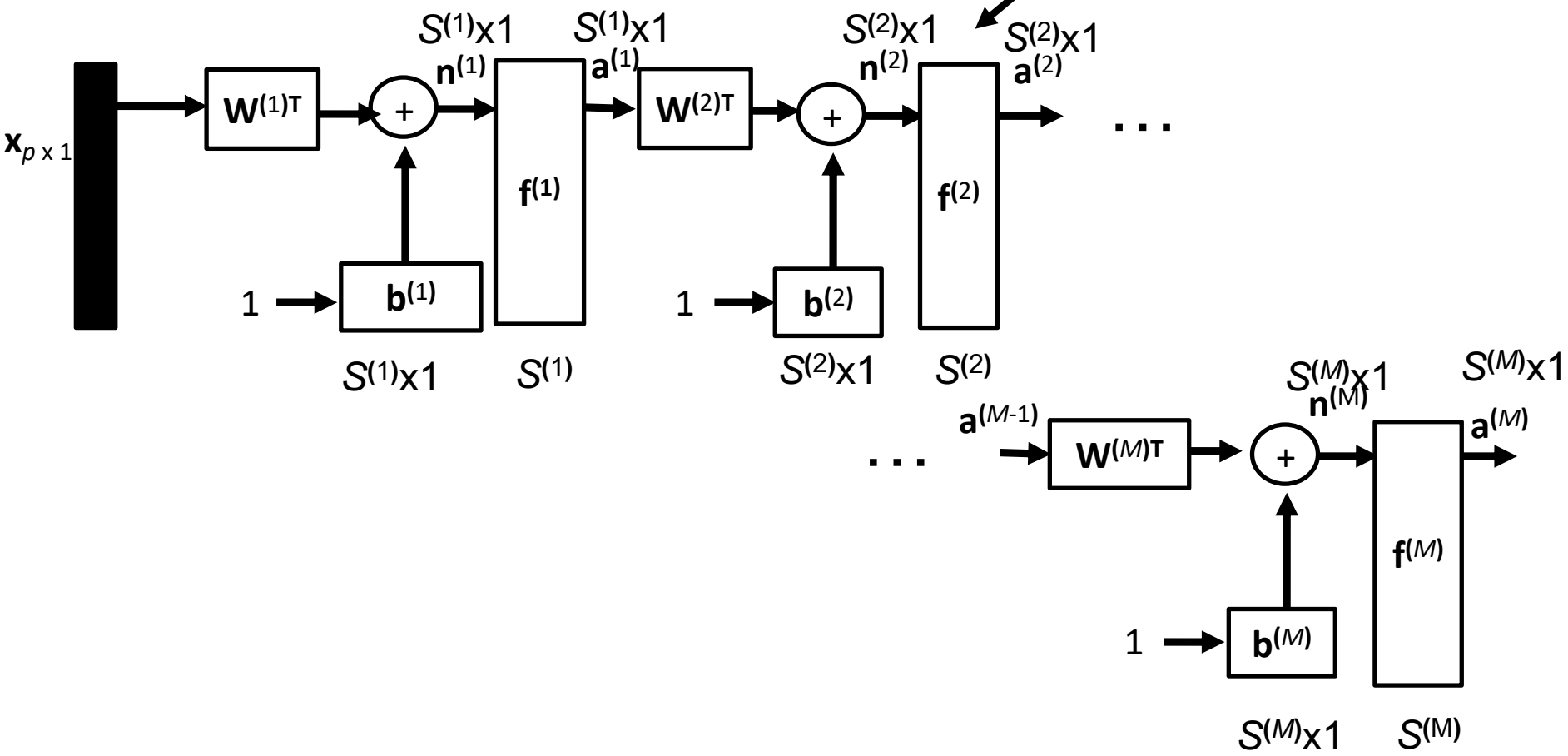
Gradient Descent Formulation



Forward Propagation

- $\mathbf{a}^{(0)} = \mathbf{x}(k)$
- $\mathbf{a}^{(m+1)} = \mathbf{f}^{(m+1)}(\mathbf{W}^{(m+1)\mathrm{T}}\mathbf{a}^{(m)} + \mathbf{b}^{(m+1)})$
- $m = 0, 1, 2, \ldots, M-1$
- $\mathbf{a}^{(M)} = \mathbf{a}$

Elementwise operation



Back Propagation

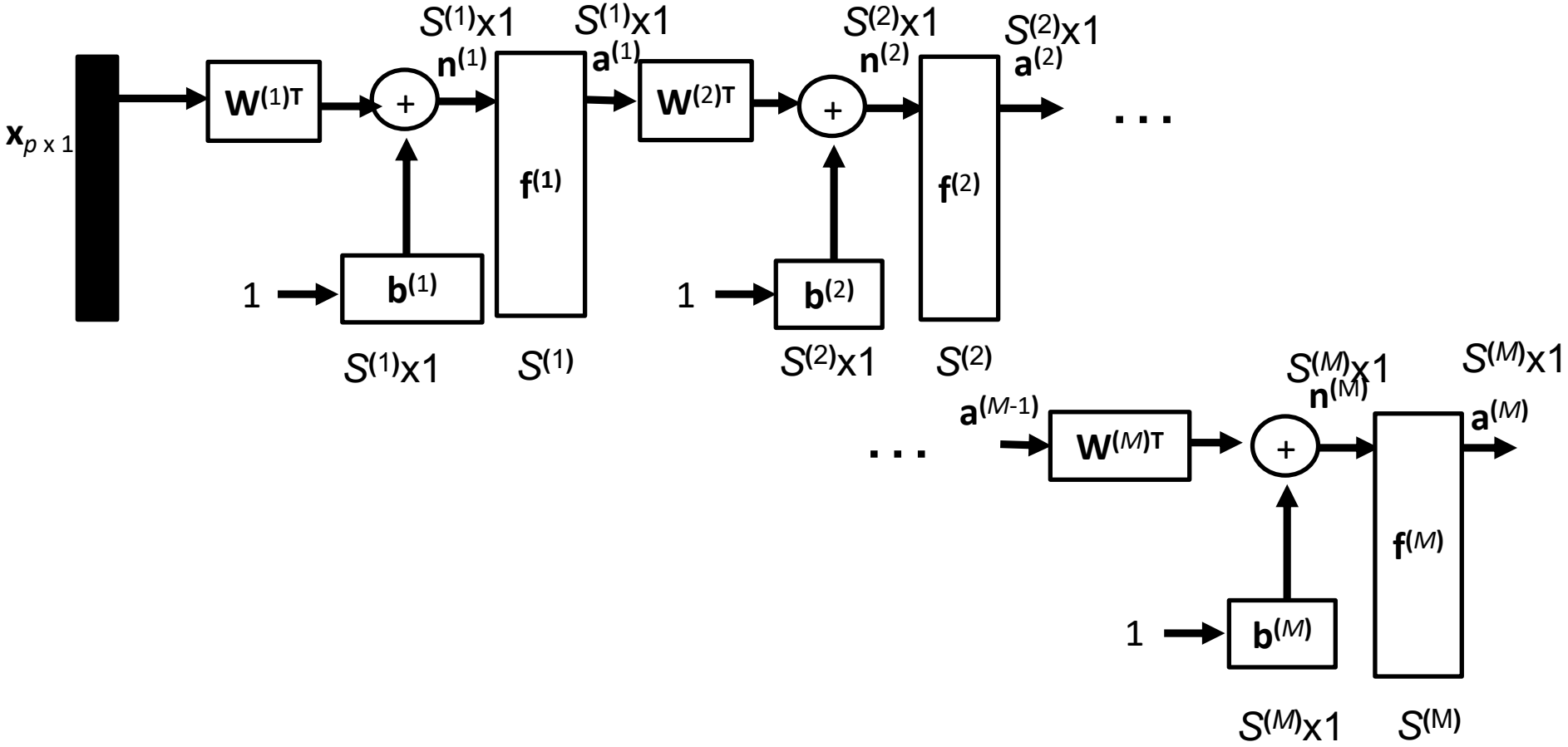
- Define $\mathbf{F}'^{(m)}(\mathbf{n}^{(m)})$ as:

$$\begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}^m(n_2^m) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}$$

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}$$

Back Propagation

- $\mathbf{s}^{(M)} = -2\mathbf{F}'^{(M)}(\mathbf{n}^{(M)})(\mathbf{y}-\mathbf{a})$
- $\mathbf{s}^{(m)} = \mathbf{F}'^{(m)}(\mathbf{n}^{(m)})\mathbf{W}^{(m+1)}\mathbf{s}^{(m+1)}$
- $m = M-1,\ldots,2,1$
- $\mathbf{a}^{(M)} = \mathbf{a}$



Back Propagation

- The sensitivities are computed by starting at the last layer, and then propagating backwards through the network to the first layer.

$$\mathbf{s}^{M} \rightarrow \mathbf{s}^{M-1} \rightarrow \ldots \rightarrow \mathbf{s}^{2} \rightarrow \mathbf{s}^{1}$$

Back Propagation

- For an objective function other than $J = \mathbf{e}^T\mathbf{e}$, the term $-2(\mathbf{y}-\mathbf{a})$ in sensitivity calculation
  $\mathbf{s}^{(M)} = -2\mathbf{F}'^{(M)}(\mathbf{n}^{(M)})(\mathbf{y}-\mathbf{a})$
  will be replaced with $\nabla_{\mathbf{a}} J$

Weight Update

- $\mathbf{W}^{(m)}(k+1) = \mathbf{W}^{(m)}(k) - \alpha \mathbf{a}^{(m-1)} \mathbf{s}^{(m)T}$
- $\mathbf{b}^{(m)T}(k+1) = \mathbf{b}^{(m)T}(k) - \alpha \mathbf{s}^{(m)T}$
- Compare these equations with the Perceptron update rule.

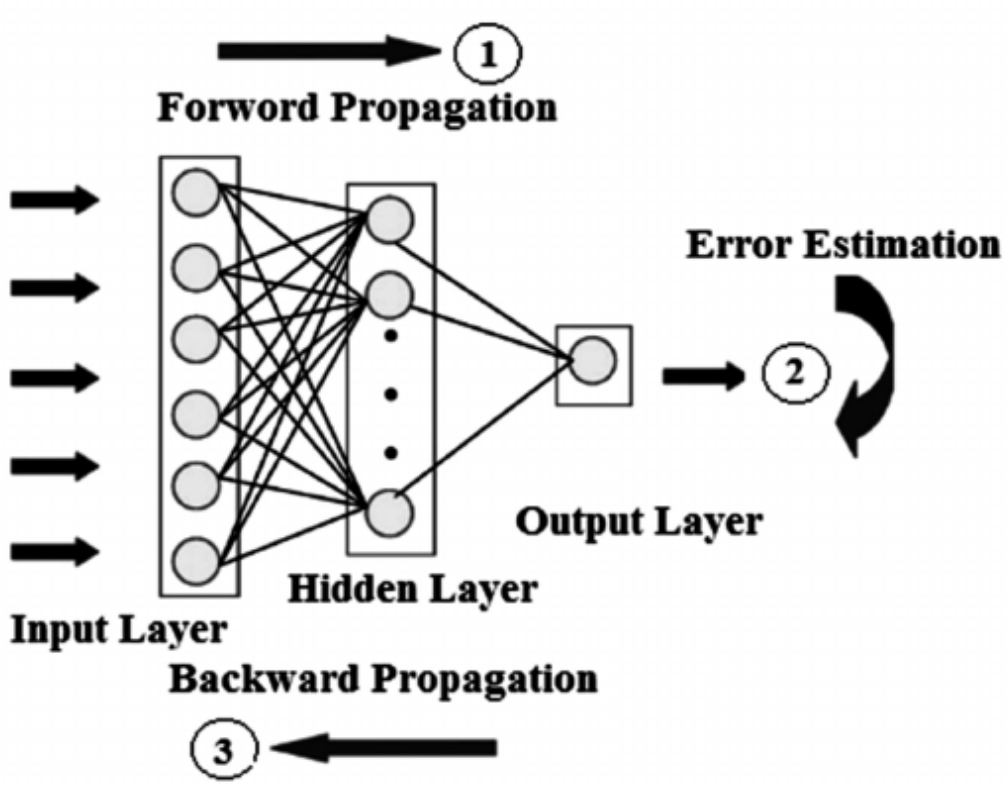
Weight Update

- $\mathbf{W}^{(m)}(k+1) = \mathbf{W}^{(m)}(k) - \alpha \mathbf{a}^{(m-1)} \mathbf{s}^{(m)T}$
- $\mathbf{b}^{(m)T}(k+1) = \mathbf{b}^{(m)T}(k) - \alpha \mathbf{s}^{(m)T}$
- We present each (randomly selected) training sample (or batch of samples) and use the forward and backward paths to calculate sensitivities and update the weights.
- This is called an iteration.

Weight Update

- $\mathbf{W}^{(m)}(k+1) = \mathbf{W}^{(m)}(k) - \alpha \mathbf{a}^{(m-1)} \mathbf{s}^{(m)T}$
- $\mathbf{b}^{(m)T}(k+1) = \mathbf{b}^{(m)T}(k) - \alpha \mathbf{s}^{(m)T}$
- Each time we finish presenting the whole training set to the network, we complete an epoch.
- An epoch includes multiple iterations.
- Training the network requires multiple epochs.

Backpropagation Schema



Regularization Methods

- In general, any method to prevent overfitting or help the optimization is considered regularization.
- One can add L1 and L2 regularizers to the objective function in backpropagation.
- However, empirical regularization is also very popular for Neural Networks.

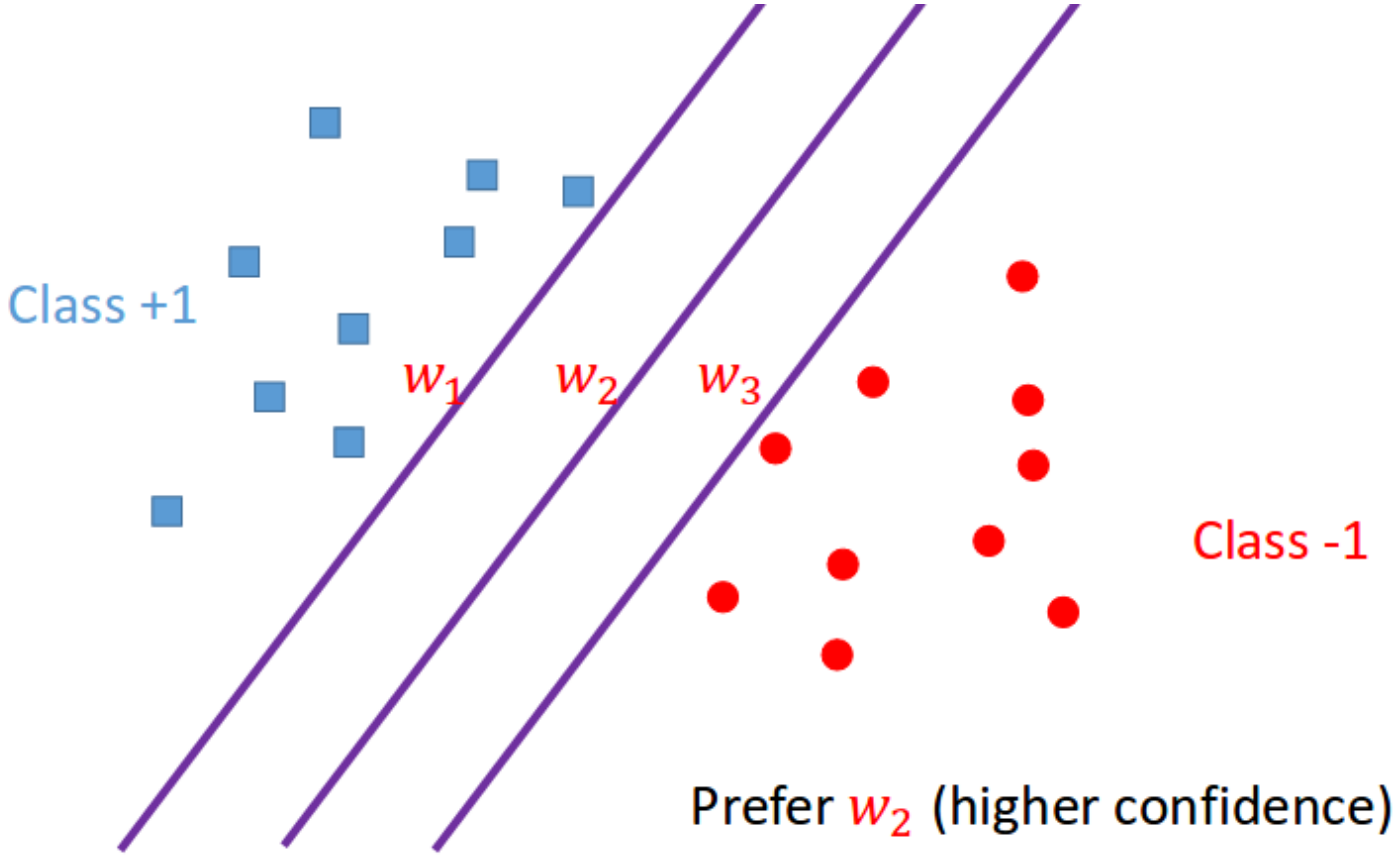
Regularization Methods

- One can show that L2 regularization is equivalent to *weight decay*
- $\mathbf{W}^{(m)}(k+1) = (1-\eta\alpha)\mathbf{W}^{(m)}(k) - \alpha\mathbf{a}^{(m-1)}\mathbf{s}^{(m)T}$
- $\mathbf{b}^{(m)T}(k+1) = (1-\eta\alpha)\mathbf{b}^{(m)T}(k) - \alpha\mathbf{s}^{(m)T}$
- *η is called the decay rate.*

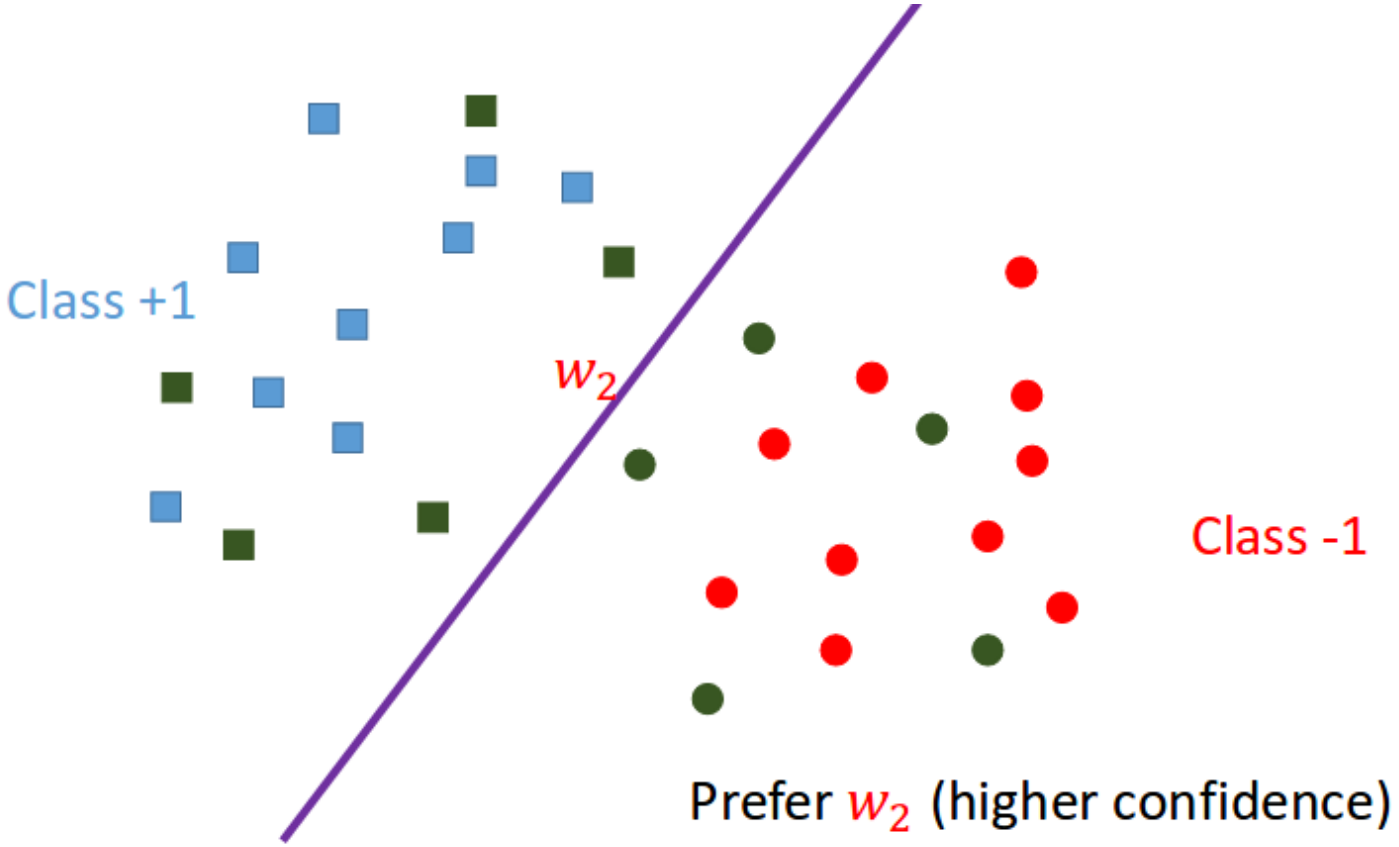
Regularization Methods

- Empirical regularization is also very popular for Neural Networks.

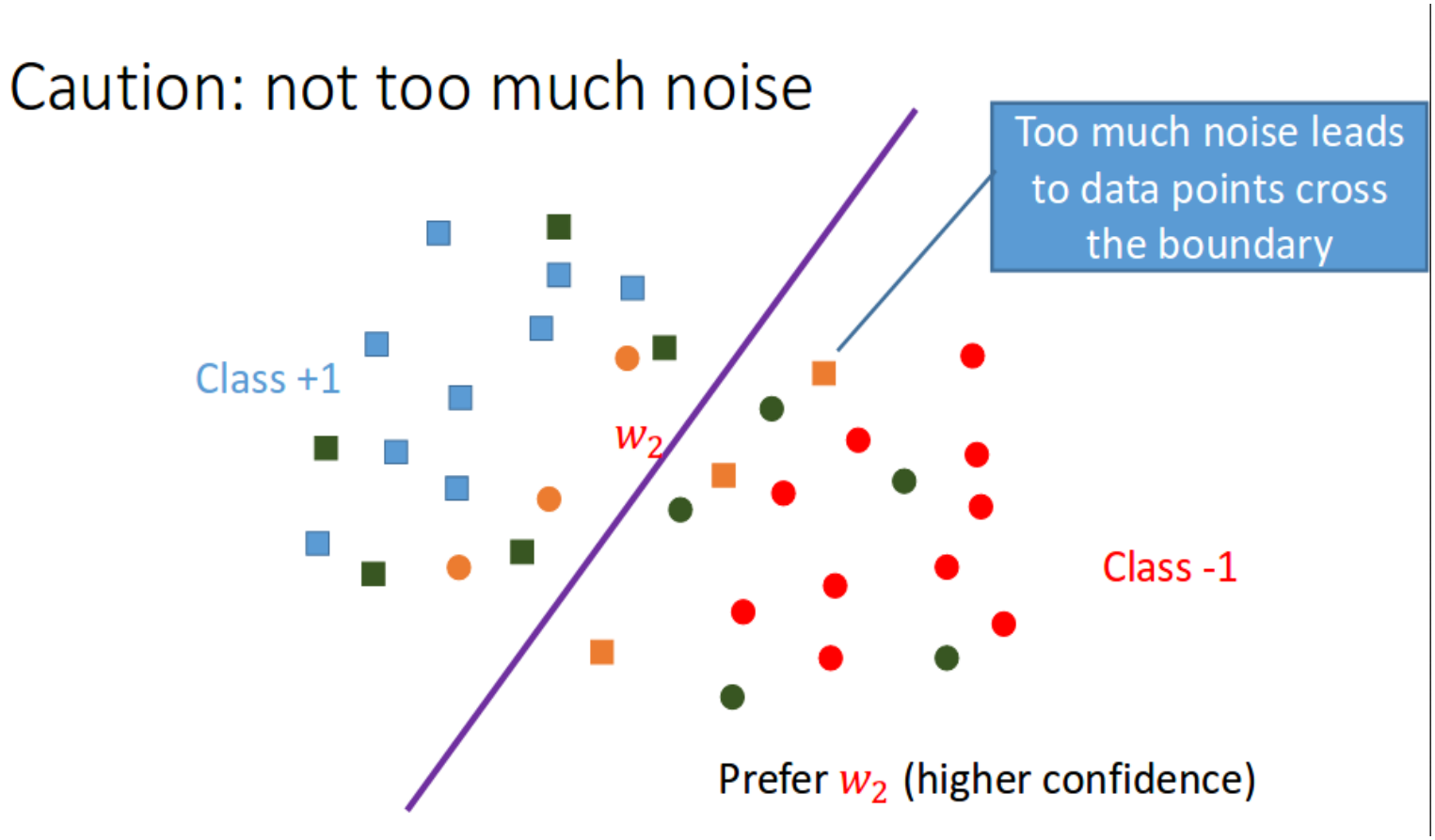
Adding Noise to The Input



Adding Noise to The Input



Drawback: Too much noise is harmful



Adding Noise to Weights

- Adding noise to the weights can be shown to be equivalent to adding a regularization term to the objective function\
- Noise has shown to have benefits when added to many learning algorithms.

Data Augmentation

- Adding noisy or transformed versions of training data to the training set

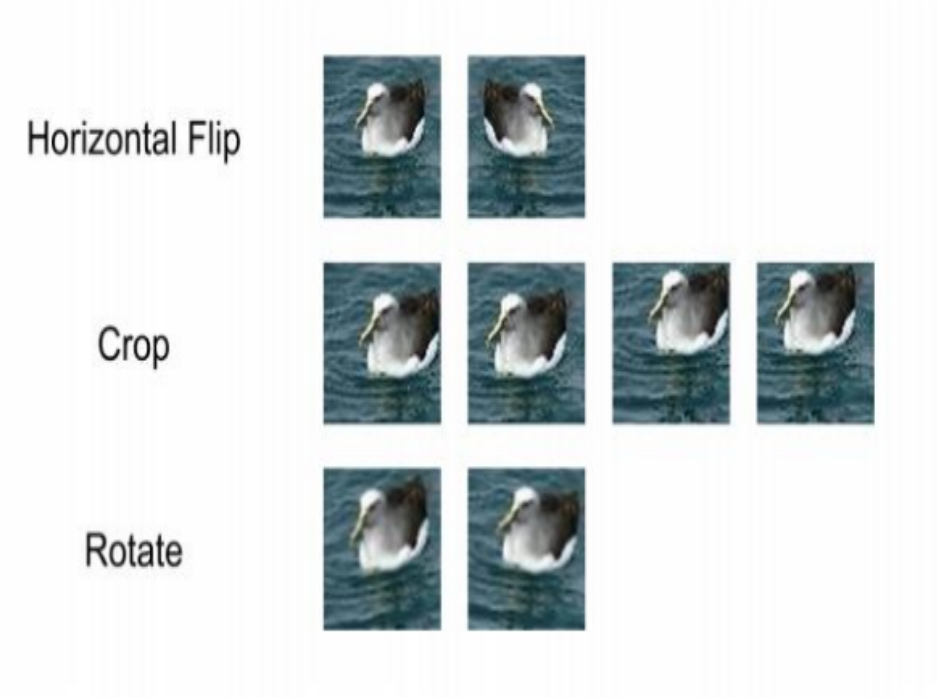


Figure from *Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7*, by Keven Wang

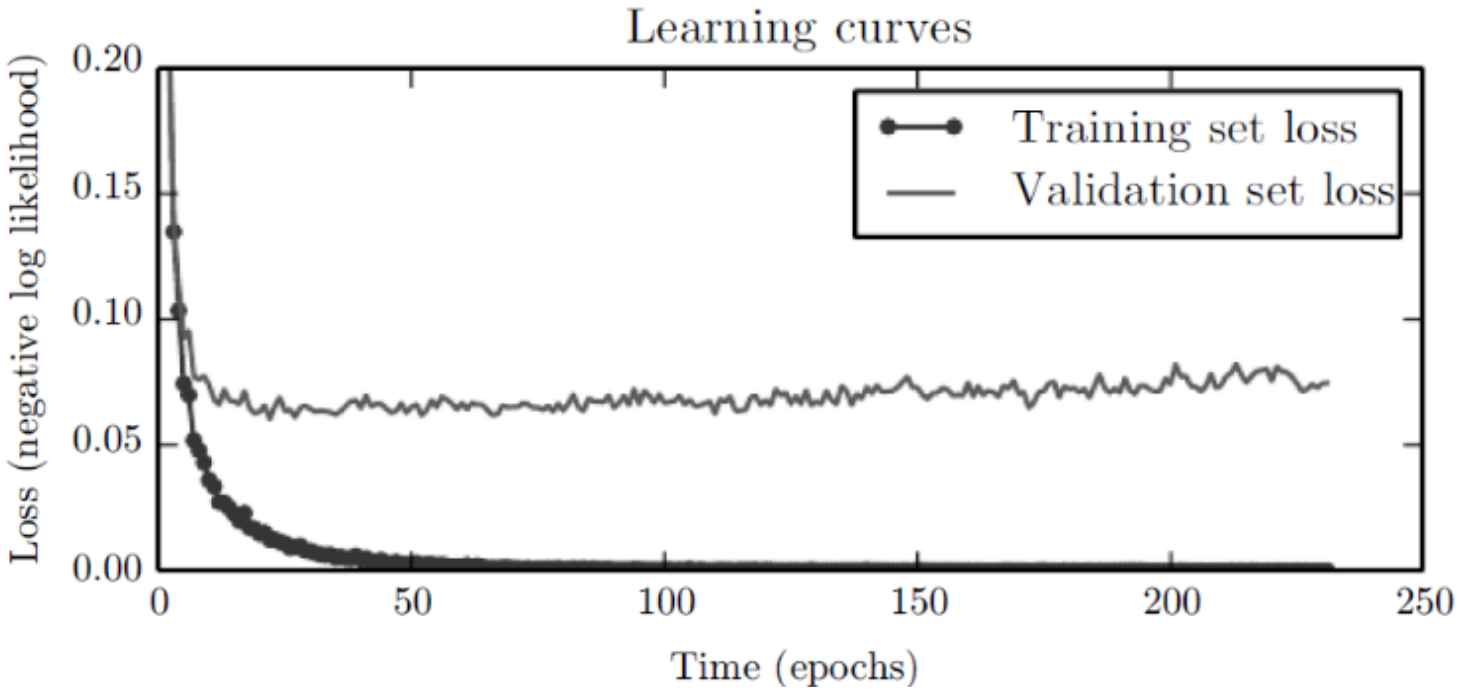
Be Careful!

- Be careful about the transformation applied:
- Example: classifying ‘b’ and ‘d’
- Example: classifying ‘6’ and ‘9’

Early Stopping

- Idea: don't train the network to have very small training error
- Prevent overfitting: use a validation set and validation error to decide when to stop the algorithm
- When the validation error starts increasing, overfitting is occurring.

Early Stopping



Another Version of Early Stopping

- When training, monitor validation error
- Every time validation error improved, store a copy of the weights
- When validation error not improved for some time, stop
- Return the copy of the weights stored

Early Stopping: Pros and Cons

- Advantage
  - Efficient: along with training; only store an extra copy of weights
  - Simple: no change to the model/algorithm
- Disadvantage:
  - Need validation data

Early Stopping: Pros and Cons

- How to remedy the disadvantage?
- After early stopping of the first run, train a second run and reuse validation data

Early Stopping: Pros and Cons

- How to reuse validation data?
  - 1.Start fresh, train with both training data and validation data up to the previous number of epochs
  - 2. Start from the weights in the first run, train with both training data and validation data until the training loss < the validation loss at the early stopping point

Dropout

- **Randomly** select weights to update
- More precisely, in each update step:
- Randomly sample a different binary mask to all the input and hidden units
- Multiple the mask bits with the units and do the update as usual
- Typical dropout probability: 0.2 for input and 0.5 for hidden units

What regularizations are frequently used?

- L2 regularization
- Early stopping
- Dropout
- Data augmentation if the transformations known/easy to implement

See Appendix

Unsupervised Learning/ Feature Learning Using NNs

&

Pre-Training

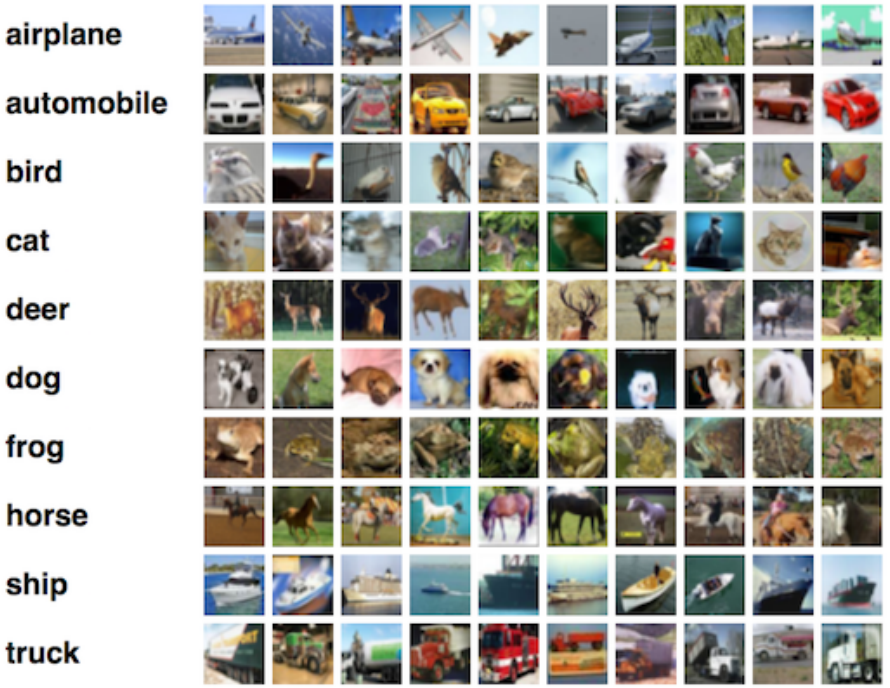
See Appendix

Adversarial Training

The Challenge of Dealing with Images

MNIST: Handwritten digit recognition
CIFAR-10: 10 distinct classes – airplane, automobile, bird, cat, deer, dog, frog, horse, ship & truck)

MNIST



CIFAR-10

Image Classification and MLPs

- A two-layer MLP can achieve an accuracy of 98.2%, which can be quite easily improved.
- Fully connected MLPs will usually not be the model of choice for image-related tasks
- It is far more typical to make advantage of a convolutional neural network (CNN) for images.

Image Classification and MLPs: Weight Proliferation

- The number of parameters (weights) of MLPs fed with raw image data is very large for images
- CIFAR-10: 32×32×3 colored images

Image Classification and MLPs: Weight Proliferation

- If we treat each channel of each pixel as an independent input to an MLP, each neuron of the first hidden layer adds ~3000 new parameters to the model!
- The number of weights quickly becomes unmanageable as image sizes grow larger

Image Classification and MLPs: Weight Proliferation

- If we treat each channel of each pixel as an independent input to an MLP, each neuron of the first hidden layer adds 3072 new parameters to the model!
- The number of weights quickly becomes unmanageable as image sizes grow larger
- This is sometimes called weight proliferation

Image Classification and MLPs: Downsampling

- Common solution: *down-sampling* the images to have fewer weights
- Drawback: direct down-sampling = loosing valuable information
- Key concept: images carry more information than their vectorized versions
- Adjacent pixels carry some information about the image

Exploiting The Structure

- The information carried by adjacent pixels can be summarized.
- Summarization is performed by the so-called convolution operator.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

A Simple “Image”

The Convolution Operator: Kernel/ Filter

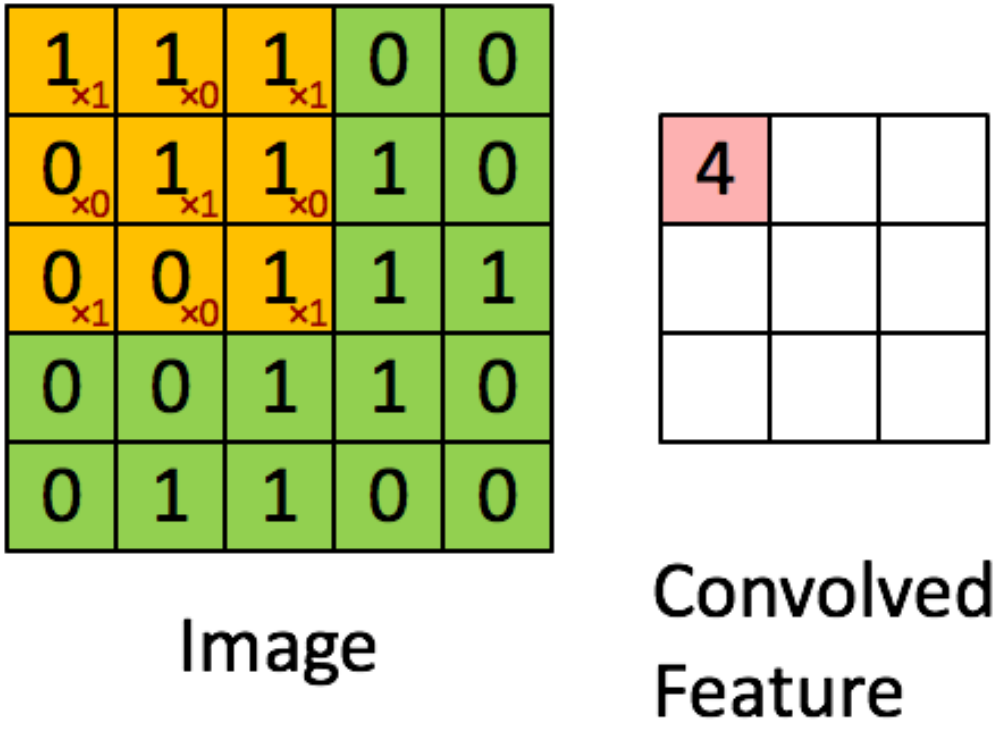
- The Kernel (or Filter) in the convolution operator is a small matrix which encodes a way of extracting an interesting feature of an image.

1	0	1
0	1	0
1	0	1

A Simple Kernel

The Convolution Operator

- The result is called a feature map, convolved feature, or activation map.



Different Kernels?

- It is evident that different Kernels will produce different Feature Maps for the same input image.
- Consider the following image:

Sharpen

0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0

Blur

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

Edge enhance

	0	0	0	
	-1	1	0	
	0	0	0	

Edge detect

	0	1	0	
	1	-4	1	
	0	1	0	

Emboss

	-2	-1	0	
	-1	1	1	
	0	1	2	

The Convolution Operator: A More Practical Example

Which Kernel to Use?

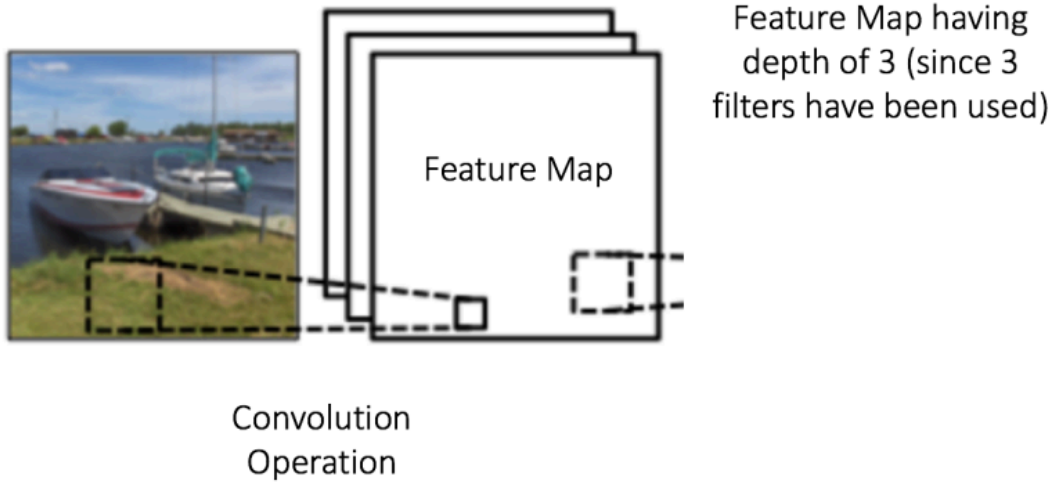
- Modern Machine Learning is about using machines to decide which features are the best
- A structure called a Convolutional Neural Network *learns* the values of these filters

Structure of a CNN

- The size of the Feature Map (Convolved Feature) is controlled by three parameters:
  - Depth
  - Stride
  - Zero-padding

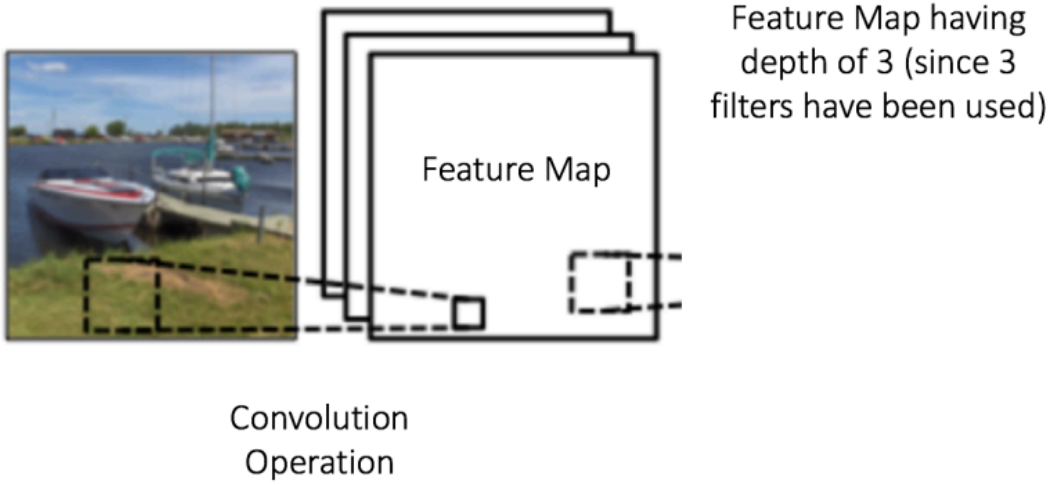
Structure of a CNN: Depth

- Depth is the number of filters used for convolution.
- In the following network, three distinct filters are used, thus three different feature maps



Structure of a CNN: Depth

- These three feature maps can be viewed as three stacked matrices, so, the “depth” of the feature map would be three.



Structure of a CNN: Stride

- Stride: the number of pixels by which we slide our filter matrix over the input matrix.
- Stride = 1: the filters move one pixel at a time.
- Stride = 2: the filters jump 2 pixels at a time .
- Larger stride yield smaller feature maps.

Structure of a CNN: Zero-Padding

- Sometimes, we pad the input matrix with zeros around the border to apply the filter to bordering elements of the input image matrix.

Structure of a CNN: Zero-Padding

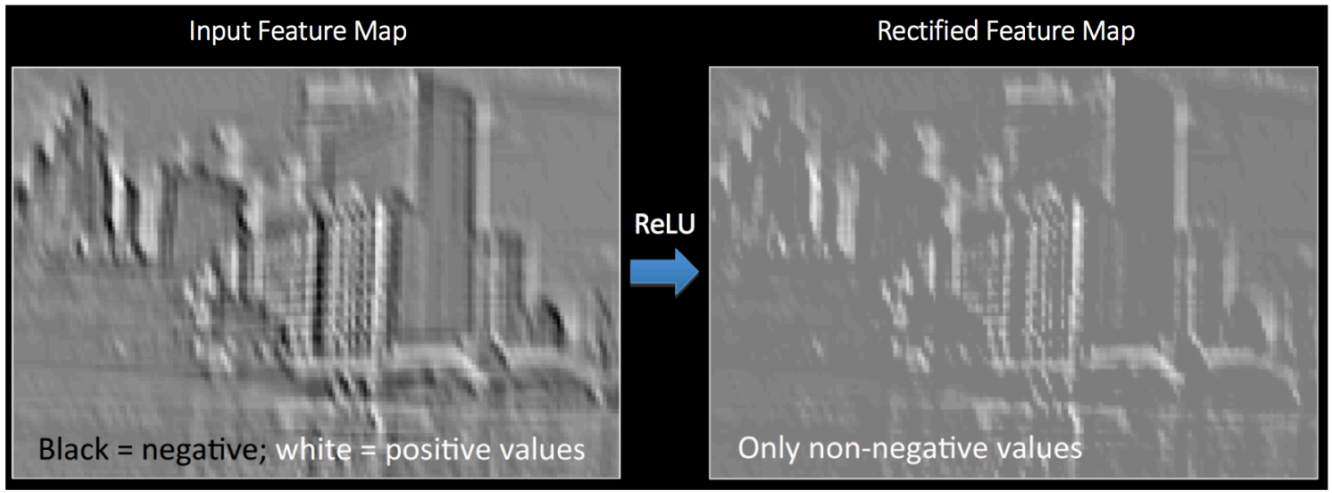
- Zero padding controls the size of the feature maps.
- Adding zero-padding: *wide convolution*
- Not using zero-padding : *narrow convolution*.

Introducing Nonlinearity

- ReLU is applied element-wise (per pixel) and replaces all negative pixel values in the feature map by zero.
- ReLU introduces nonlinearity in CNN to deal with inherent nonlinearity of classification problems

Introducing Nonlinearity

- ReLU is applied to feature maps yielding Rectified Feature Maps
- *Tanh* and *sigmoids* have also been used
- ReLU has shown better performance.



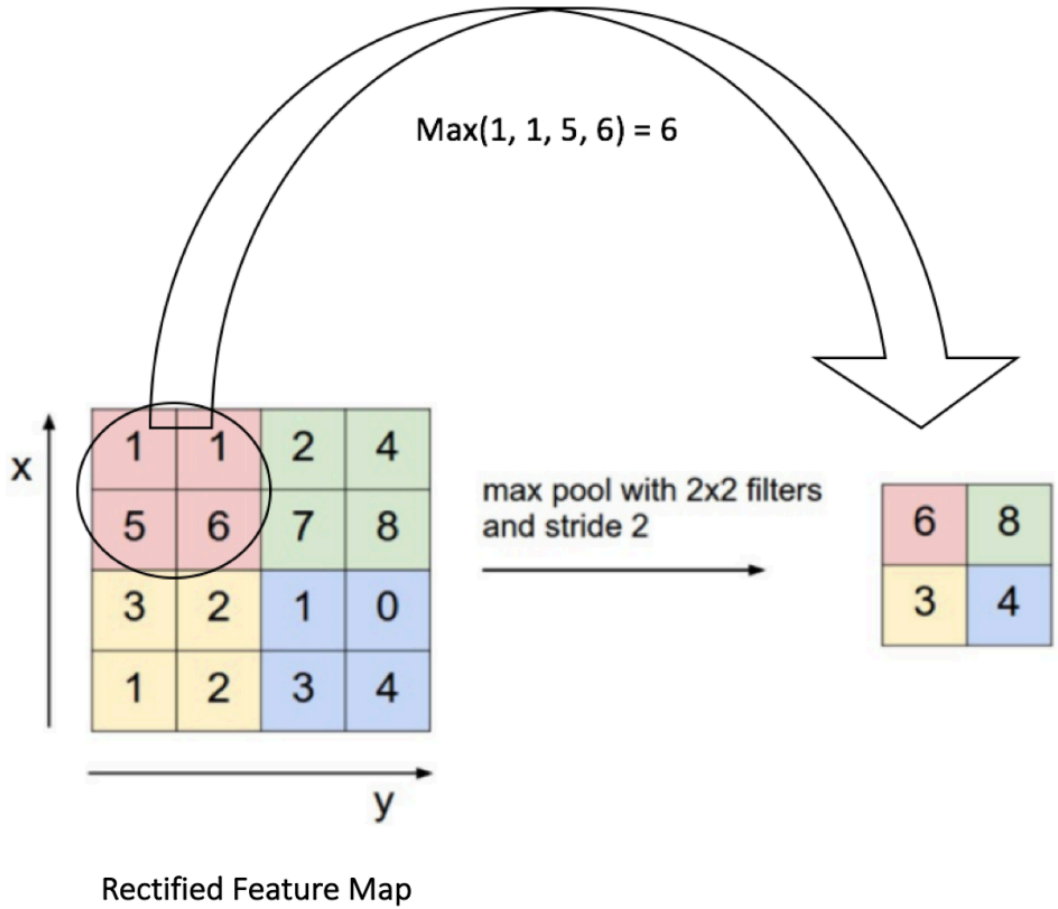
Pooling = Downsampling

- Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information.
- Spatial Pooling can be of different types: Max, Average, Sum etc.

Pooling = Downsampling

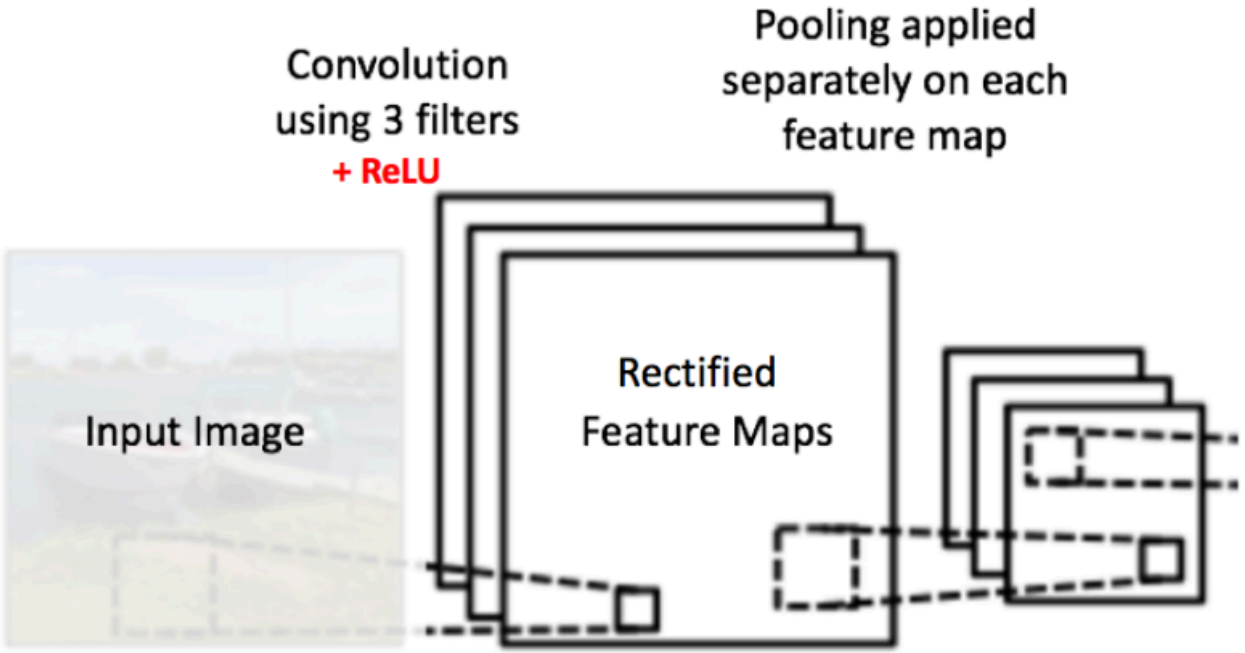
- Max Pooling: define a spatial neighborhood (e.g., a 2×2 window) and take the largest element within that window.
- Instead the largest element the average (Average Pooling) or sum of all elements could be taken.
- In practice, Max Pooling has been shown to work better.

Max Pooling Example



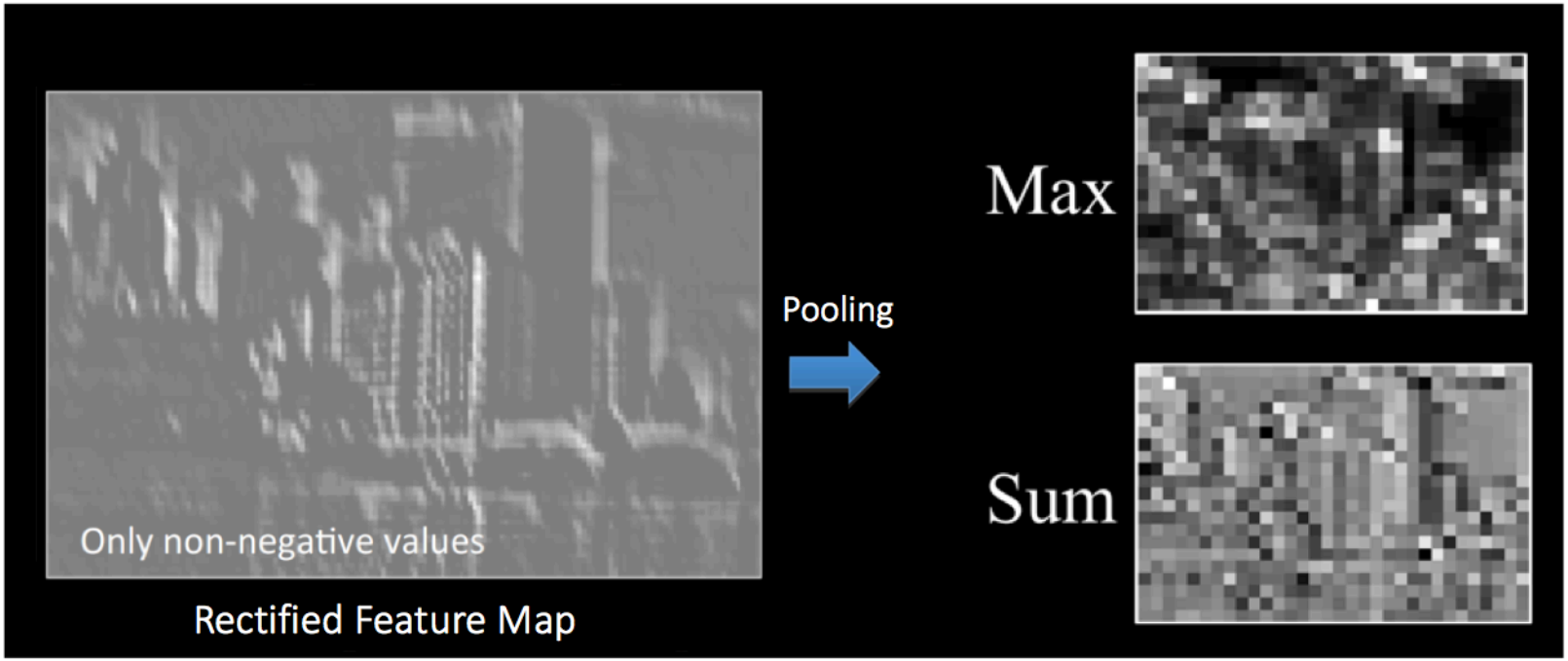
Max Pooling Strides

- Slide the 2 x 2 window by 2 cells (also called “stride”) and take the maximum value in each region to reduces the dimensionality of the feature map.



Max Pooling Strides

- Comparing Max Pooling and Sum Pooling on the Rectified Feature Map of The Buildings Image



Properties of Pooling

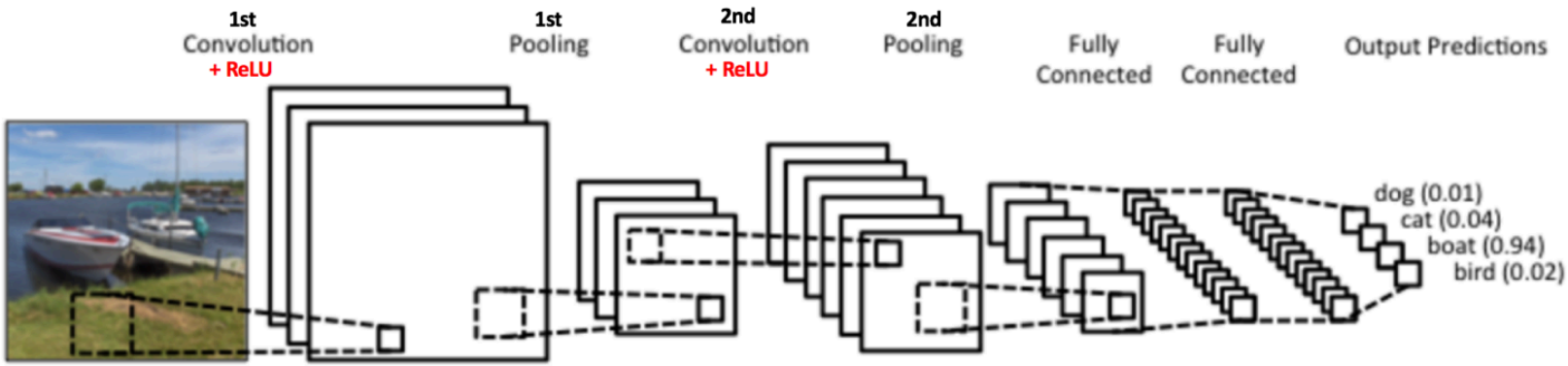
- Reduces the dimensions of the input
- Reduces the number of parameters and, therefore, controls overfitting
- Is invariant to **small** transformations, distortions and translations in the input image
  - A small distortion in input will not change the output of Pooling too much – since we take the maximum / average value in a local neighborhood

Properties of Pooling

- Yields an almost scale invariant representation of our image.
- This is very powerful since we can detect objects in an image no matter where they are located
- In essence, convolutions extract features from the image and pooling makes the features invariant to transformations

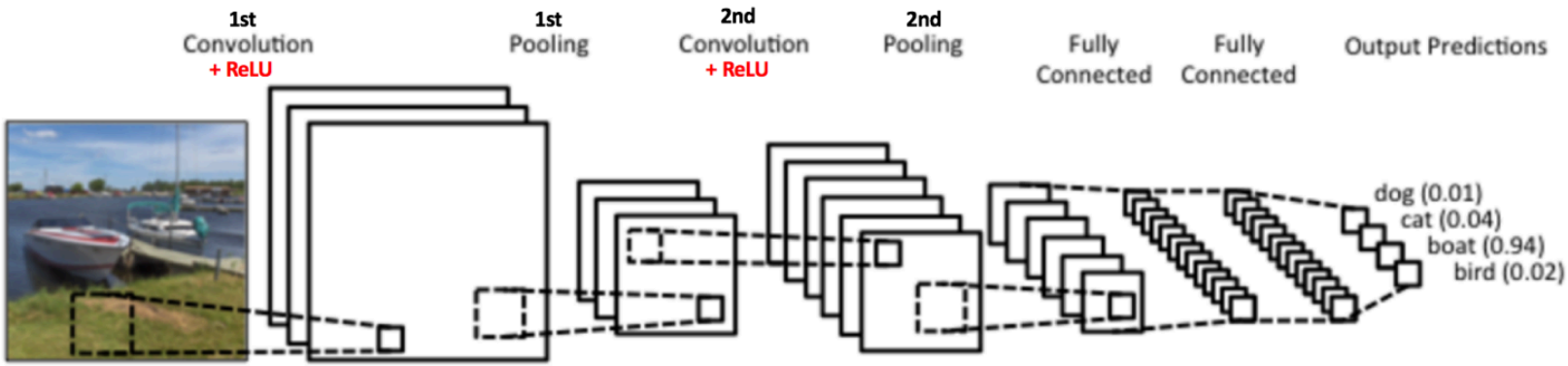
Convolution-ReLU-Pooling Repeated

- The convolution-ReLU-Pooling operation can be repeated many times in many convolutional "layers" with different depths and strides



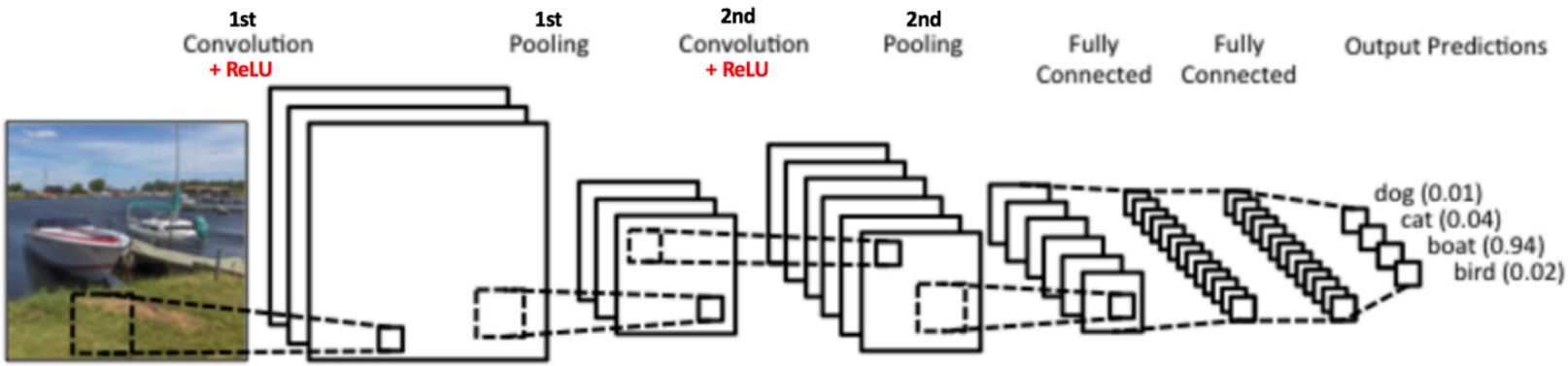
Fully Connected Layer

- Convolutional Layers play the role of feature extractors for a Deep Neural Network
- The output layer of the MLP must be softmax



Other Classification Techniques?

- Instead of MLP, other classifiers such as SVM can be used

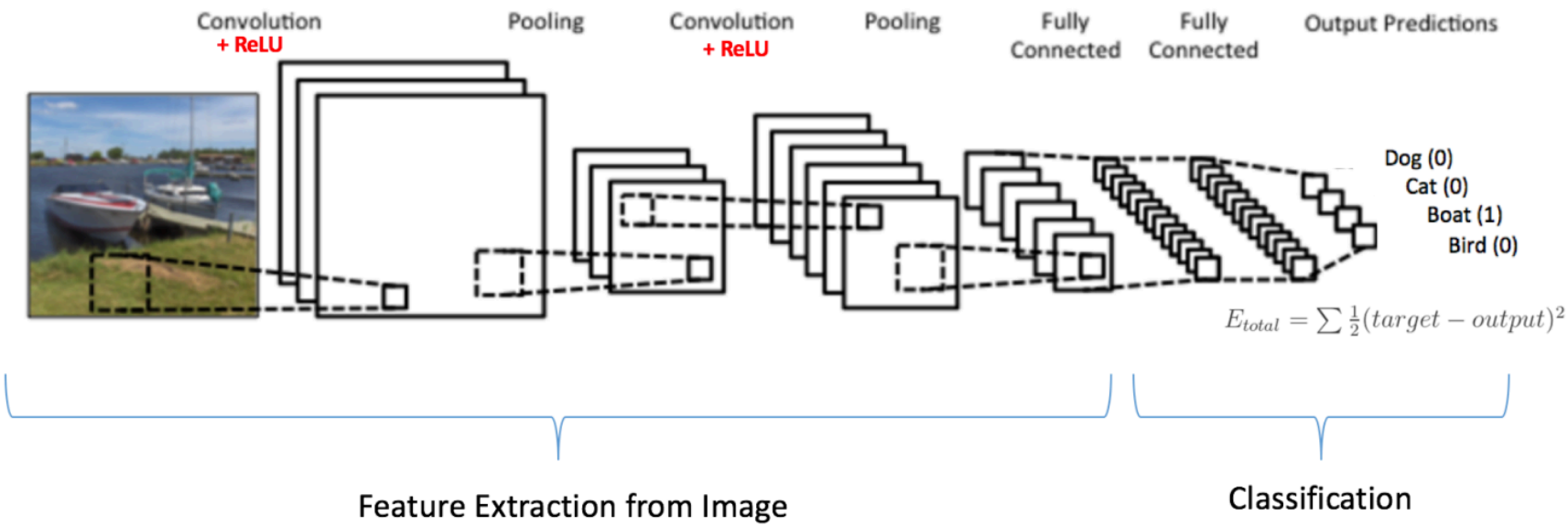


MLP: Even Better Feature Learning!

- Adding a fully-connected layer is also a (usually) cheap way of learning non-linear combinations of these features.
- Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better

Putting It Altogether: Train Using Backpropagation

- Input Image = Boat
- Target Vector = [0, 0, 1, 0]



Putting It Altogether: Train Using Backpropagation

- Initialize weights and filters randomly and repeat the following steps for the training set many times
- Pass each image through the network (forward pass) and read the output
- Calculate the total error
  - **Total Error = $\sum$ (target probability – output probability) ^{2}**
- Target Vector = [0, 0, 1, 0]
- Output Vector =[0.2, 0.4, 0.3, 0.1]
- Error =0.7

Putting It Altogether: Train Using Backpropagation

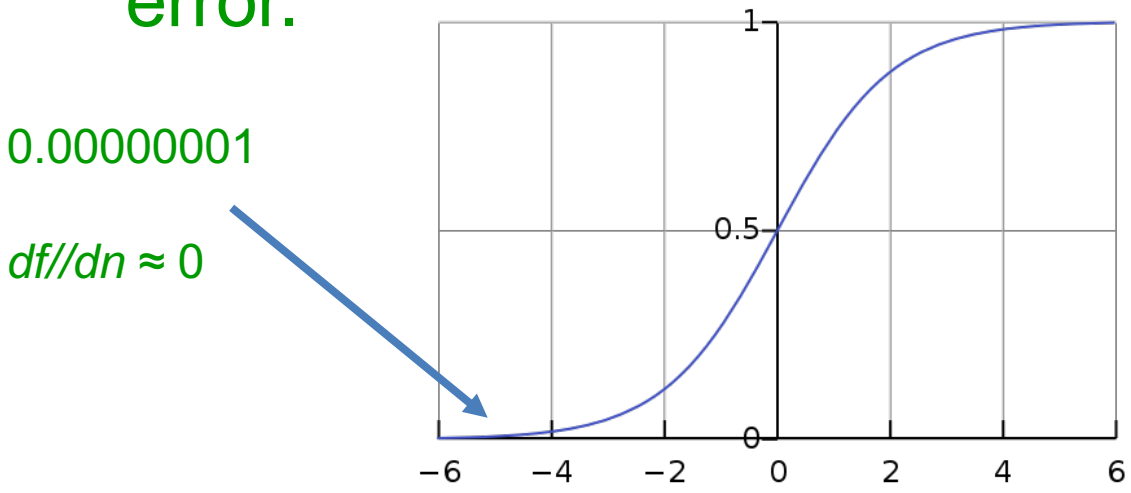
- Use Backpropagation to calculate the *gradients* of the error with respect to all weights in the network and use *gradient descent* to update all filter values / weights and to minimize the output error.
  - The weights are adjusted in proportion to their contribution to the total error.

Putting It Altogether: Train Using Backpropagation

- Number of filters, filter sizes, architecture of the network etc are fixed and do not change during training process
- Only the values of the filter matrix and connection weights are updated.

Problems with squared error

- The squared error measure has some drawbacks:
  - If the desired output is 1 and the actual output is 0.00000001 there is almost no gradient for a sigmoid unit to fix up the error.



Problems with squared error

- The squared error measure has some drawbacks:
  - If we are trying to assign probabilities to mutually exclusive class labels, we know that the outputs should sum to 1, but we are depriving the network of this knowledge.

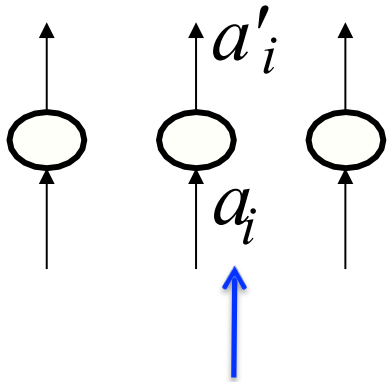
Problems with squared error

- Is there a different cost function that works better?
  - Yes: Force the outputs to represent a probability distribution across discrete alternatives.

Softmax

The output units in a softmax group use a non-local non-linearity:

$$a_i' = \frac{e^{a_i}}{\sum_{j \in group} e^{a_j}}$$

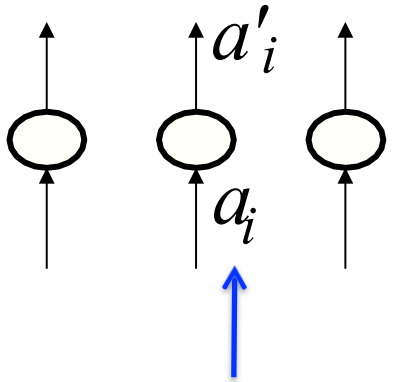


softmax group

$$\frac{\partial a_i'}{\partial a_i} = a_i'(1 - a_i')$$

this is called the "logit"

Relationship with Logistic Regression?



softmax group

this is called the "logit"

$$a'_i = \frac{e^{a_i}}{\sum_{j \in group} e^{a_j}}$$

$$\frac{\partial a'_i}{\partial a_i} = a'_i(1 - a'_i)$$

In logistic regression, logits are *linear functions* of the features. In neural networks, they are features learned by multiple layers of the network!

Relationship with Logistic Regression?

In fact,

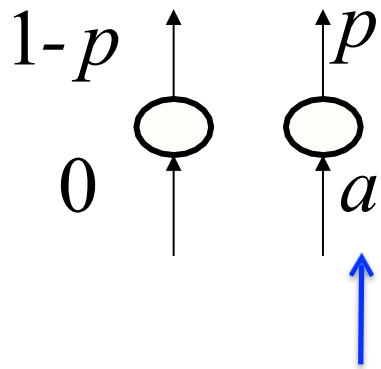
$a = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$

$= \beta^T \mathbf{x}(j) + \beta_0$

and

0

are the logits for two classes, where $\mathbf{x}(j)$ is the j^{th} data point whose label is $z(j) \in \{0,1\}$.



softmax group

Linear Function of Features

Relationship with Logistic Regression?

Consequently, for the positive class:

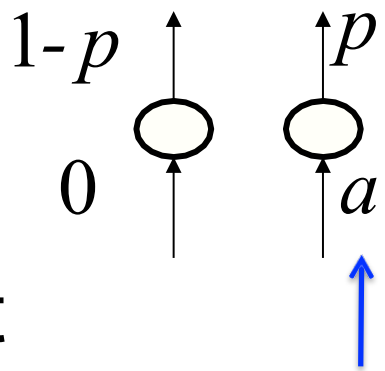
$p = e^a/(e^0+e^a) = e^a/(1+e^a)$

And obviously the output for the negative class is:

$1-p = 1/(1+e^a)$

The negative log-likelihood loss for this single data point **x**(*j*) is:

$-z(j)\log p-(1-z(j))\log(1-p)$

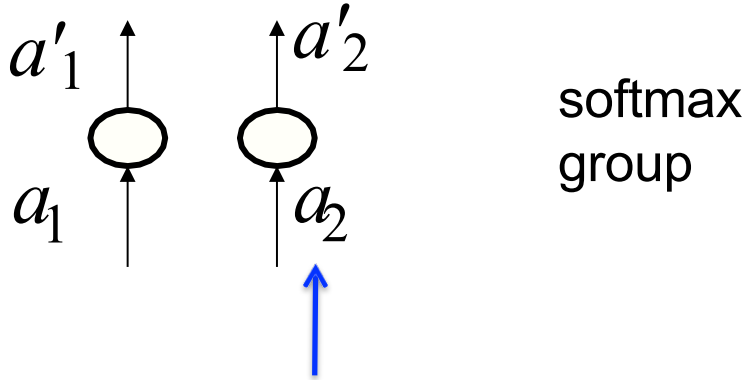


Relationship with Logistic Regression?

The log-likelihood loss for this single data point **x**(*i*) is:

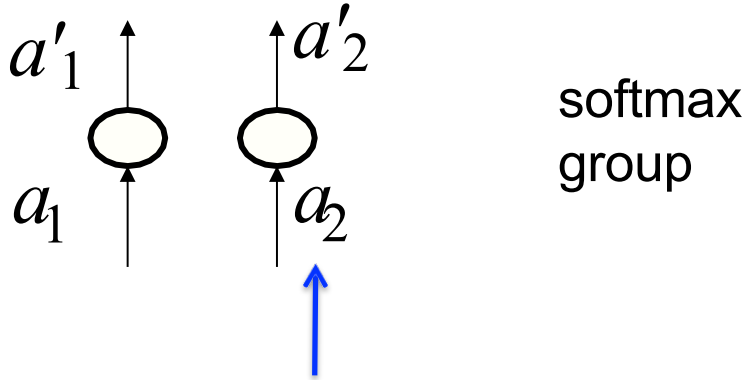
$$-z(j)\log p-(1-z(j))\log(1-p)$$

If we rename the desired outputs 1-*z*(*j*) and *z*(*j*) to y_1 and y_2, and the probability outputs of softmax as 1-*p*=a'_1 and *p*=a'_2, the negative log-likelihood becomes the cross-entropy function:



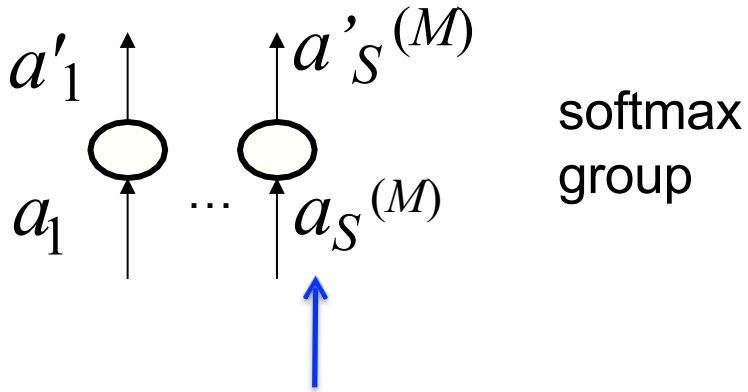
Relationship with Logistic Regression?

If we rename the desired outputs 1-z(*j*) and z(*j*) to y_1 and y_2, and the probability outputs of softmax as 1-*p*=a'_1 and *p*=a'_2, the negative log-likelihood becomes the cross-entropy function:



Relationship with Logistic Regression?

In this case, we only have two neurons representing two classes. If we have more than two classes, we have *M* neurons, and the cross-entropy loss for one data point is:



Cross-entropy: the right cost function to use with softmax

- This cost function is the right cost function.

$$J' = -\sum_{i=1}^{S^{(M)}} y_i \log a'_i$$

$$\frac{\partial J'}{\partial a_i} = \sum_j \frac{\partial J'}{\partial a'_j} \frac{\partial a'_j}{\partial a_i}$$

$$= (a'_i - y_i)$$

Cross-entropy: the right cost function to use with softmax

- J'_2 has a very big gradient when the target value is 1 and the output is almost zero.

$$J' = -\sum_i^{S^{(M)}} y_i \log a'_i$$

$$\frac{\partial J'}{\partial a_i} = \sum_j \frac{\partial J'}{\partial a'_j} \frac{\partial a'_j}{\partial a_i}$$

$$= (a'_i - y_i)$$

Cross-entropy: the right cost function to use with softmax

- This basically makes FFNNs with a softmax output layer a nonlinear version of logistic regression, where logits are highly nonlinear functions of the input that are learned from data.

$$J' = -\sum_{i}^{S^{(M)}} y_i \log a'_i$$

$$\frac{\partial J'}{\partial a_i} = \sum_j \frac{\partial J'}{\partial a'_j} \frac{\partial a'_j}{\partial a_i}$$

$$= (a'_i - y_i)$$

Sigmoids in hidden layers

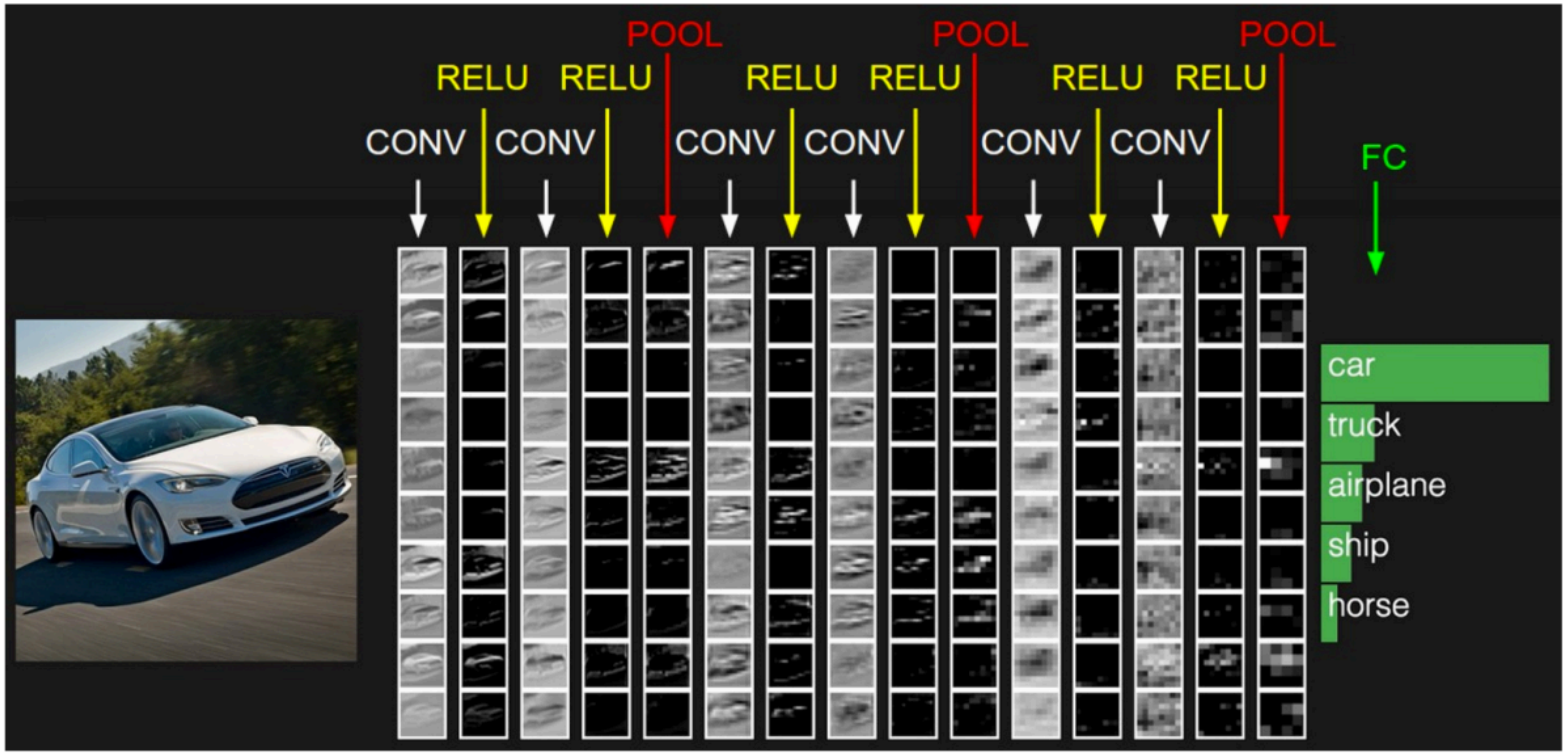
- Using cross entropy and softmax remedies the problem of sigmoids in the output layer being in their saturation region.
- In hidden layers, this cannot be done and sigmoids may cause problems, especially if there are many hidden layers.
- To solve this problem, ReLUs are used way more often than sigmoids in hidden layers in modern Deep Learning.

Note!

- Some of the best performing CNNs have tens of Convolution and Pooling layers
- Not necessary to have a Pooling layer after every Convolutional Layer.

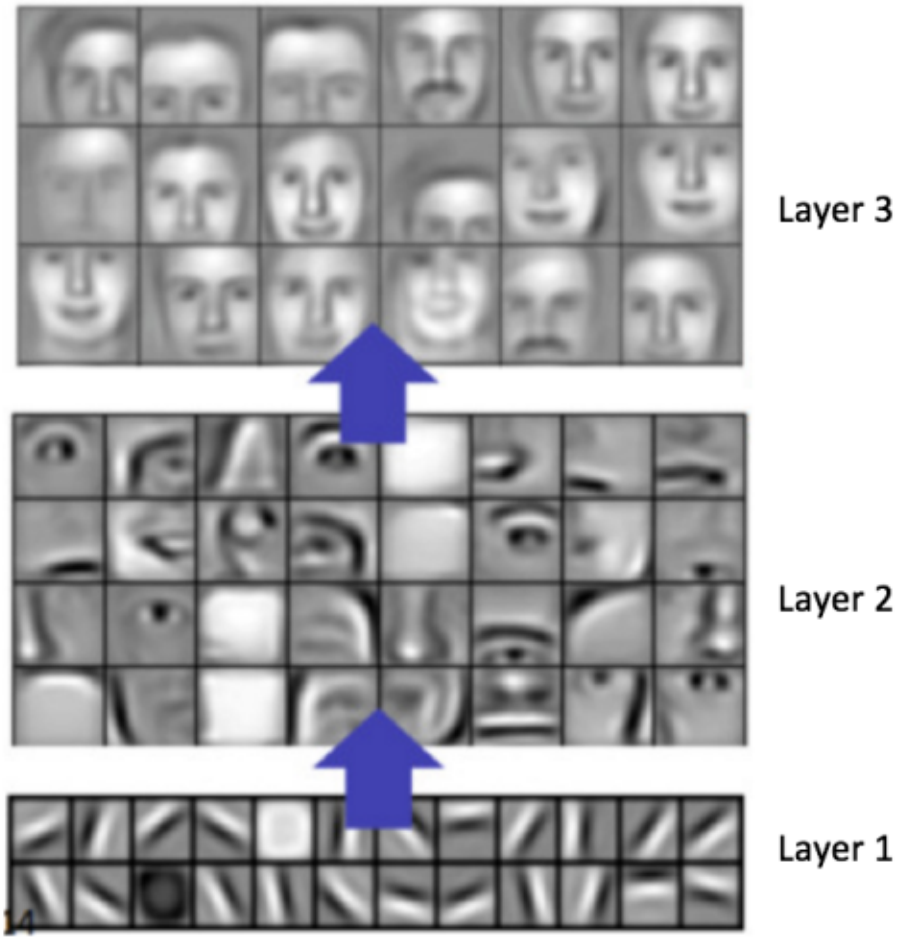
Note!

- Multiple Convolution + ReLU operations in succession before having a Pooling operation:



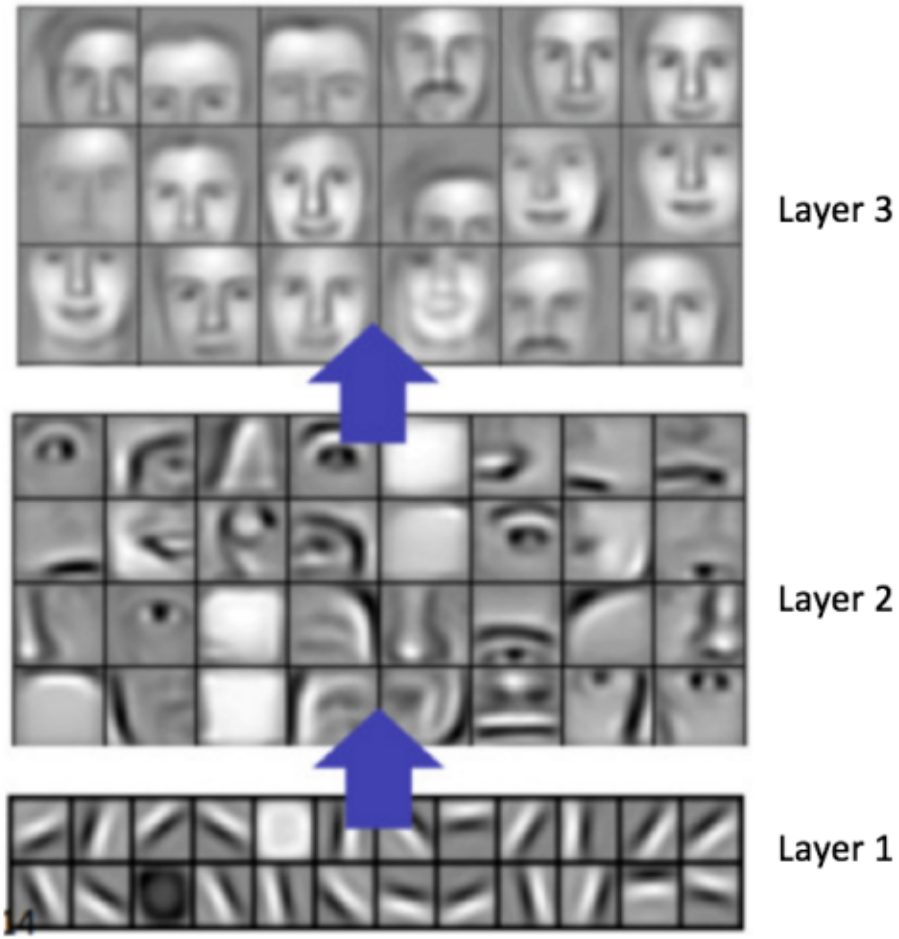
Visualizing CNNs

- Learning High-Level Features

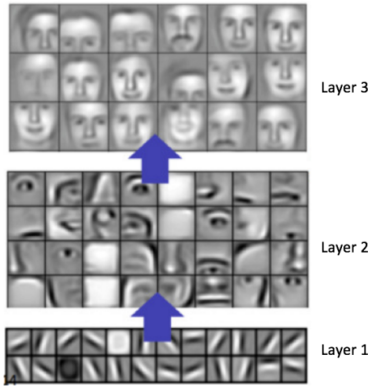


Visualizing CNNs: Example

- Learning High-Level Features

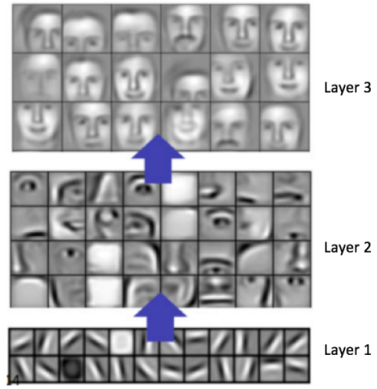


Visualizing CNNs: Example



- Detect edges from raw pixels in the first layer
- Uses the edges to detect simple shapes in the second layer,
- Use these shapes to deter higher-level features, such as facial shapes, in higher layers

Visualizing CNNs: Example



- This is only an example: real life convolution filters may detect objects that have no meaning to humans.

See Appendix

- Capsule Networks

Recurrent neural networks

- Dates back to (Rumelhart et al., 1986)
- A family of neural networks for handling sequential data, which involves variable length inputs or outputs

- Especially, for natural language processing (NLP)

Sequential data

- Each data point: A sequence of vectors $x^{(t)}$, for $1 \leq t \leq \tau$
- Batch data: many sequences with different lengths τ
- Label: can be a scalar, a vector, or even a sequence

- Example
  - Sentiment analysis
  - Machine translation

Example: Machine Translation

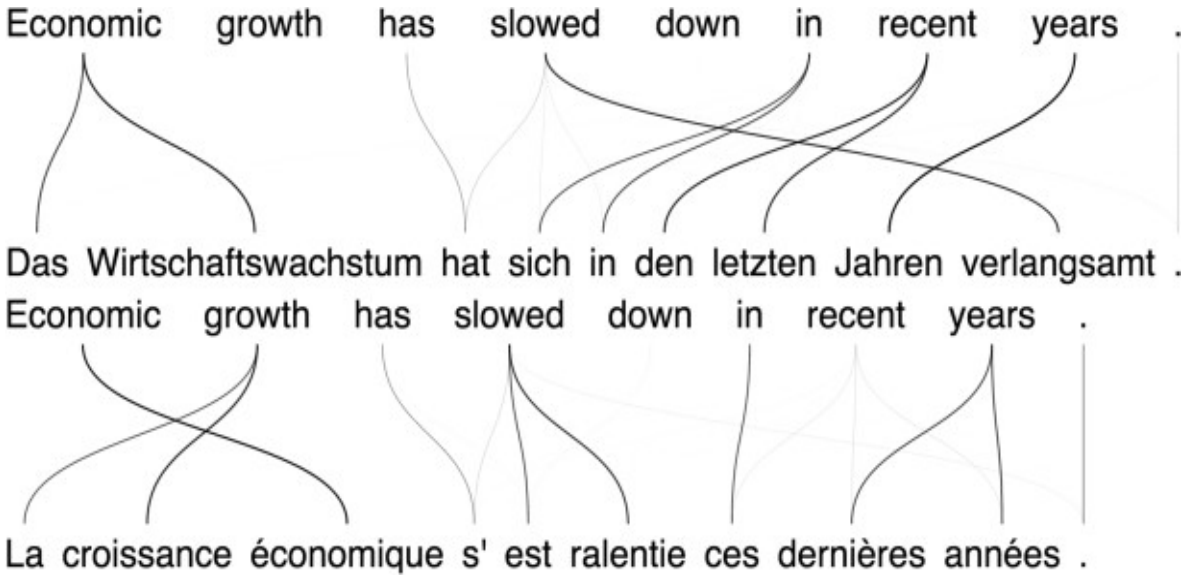


Figure from: devblogs.nvidia.com

More complicated sequential data

- Data point: two dimensional sequences like images
- Label: different type of sequences like text sentences
- Example: image captioning

Image captioning

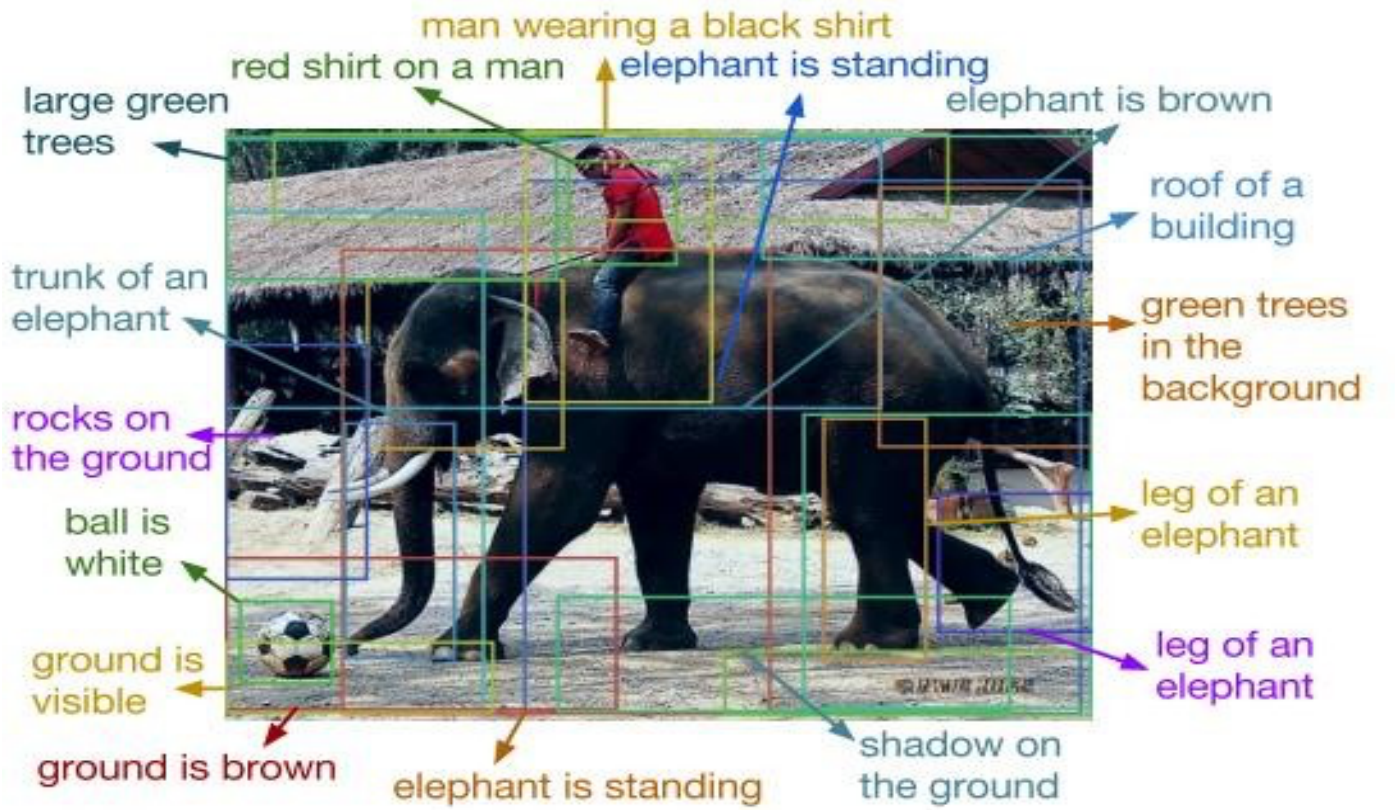
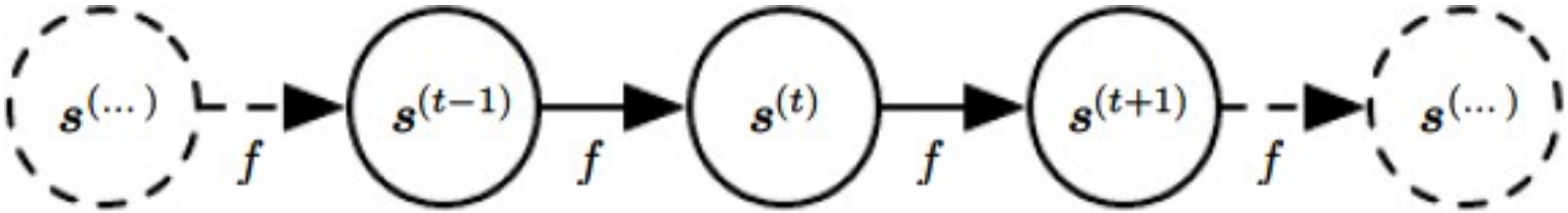


Figure from the paper “DenseCap: Fully Convolutional Localization Networks for Dense Captioning”, by Justin Johnson, Andrej Karpathy, Li Fei-Fei

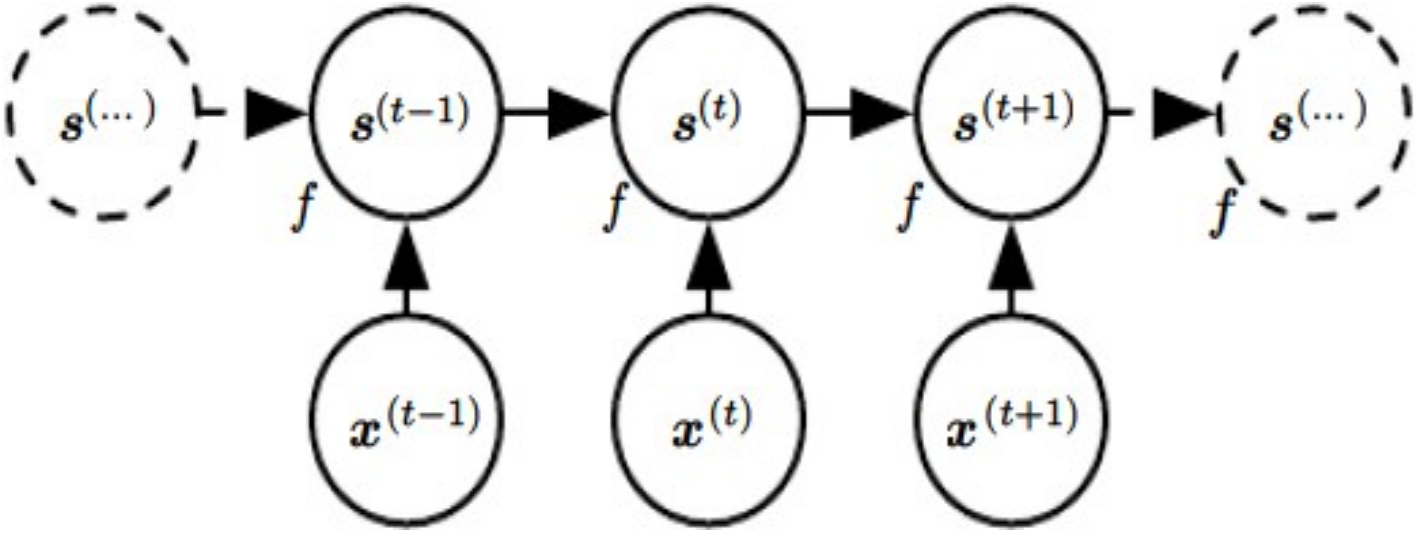
A typical dynamical system: computational graph



$$s^{(t+1)} = f(s^{(t)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

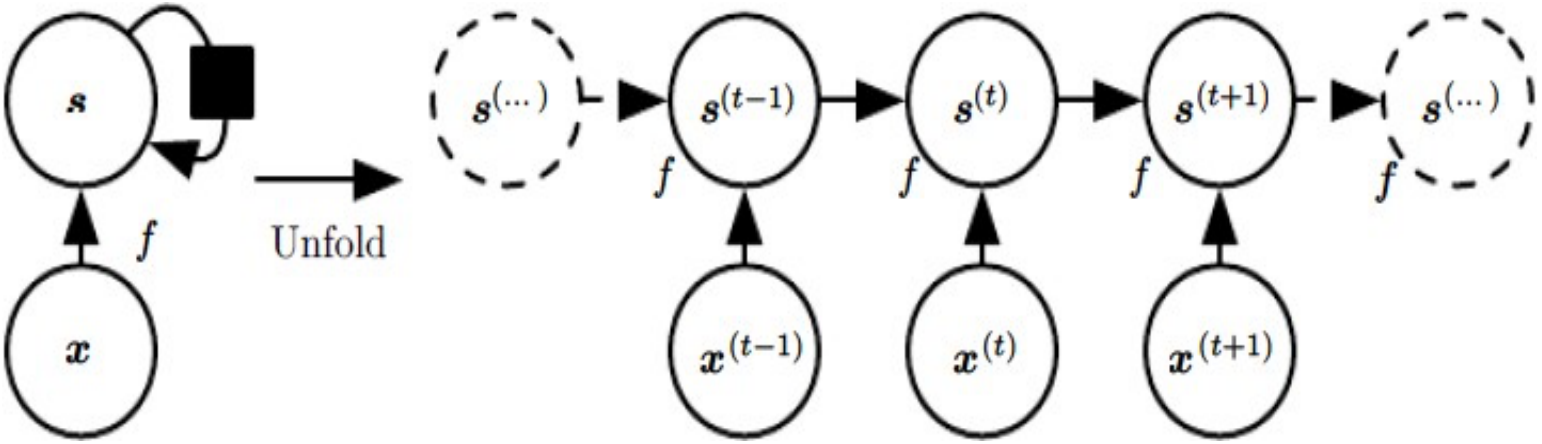
A system driven by external input



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Compact view

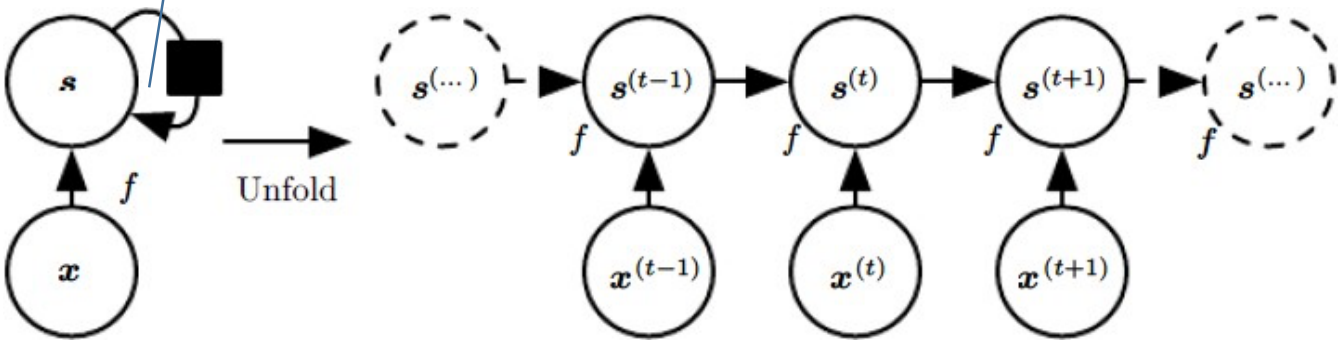


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Compact view

square: one step time delay



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Key: the same f and θ for all time steps

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Recurrent neural networks

- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed every time step

Recurrent neural networks

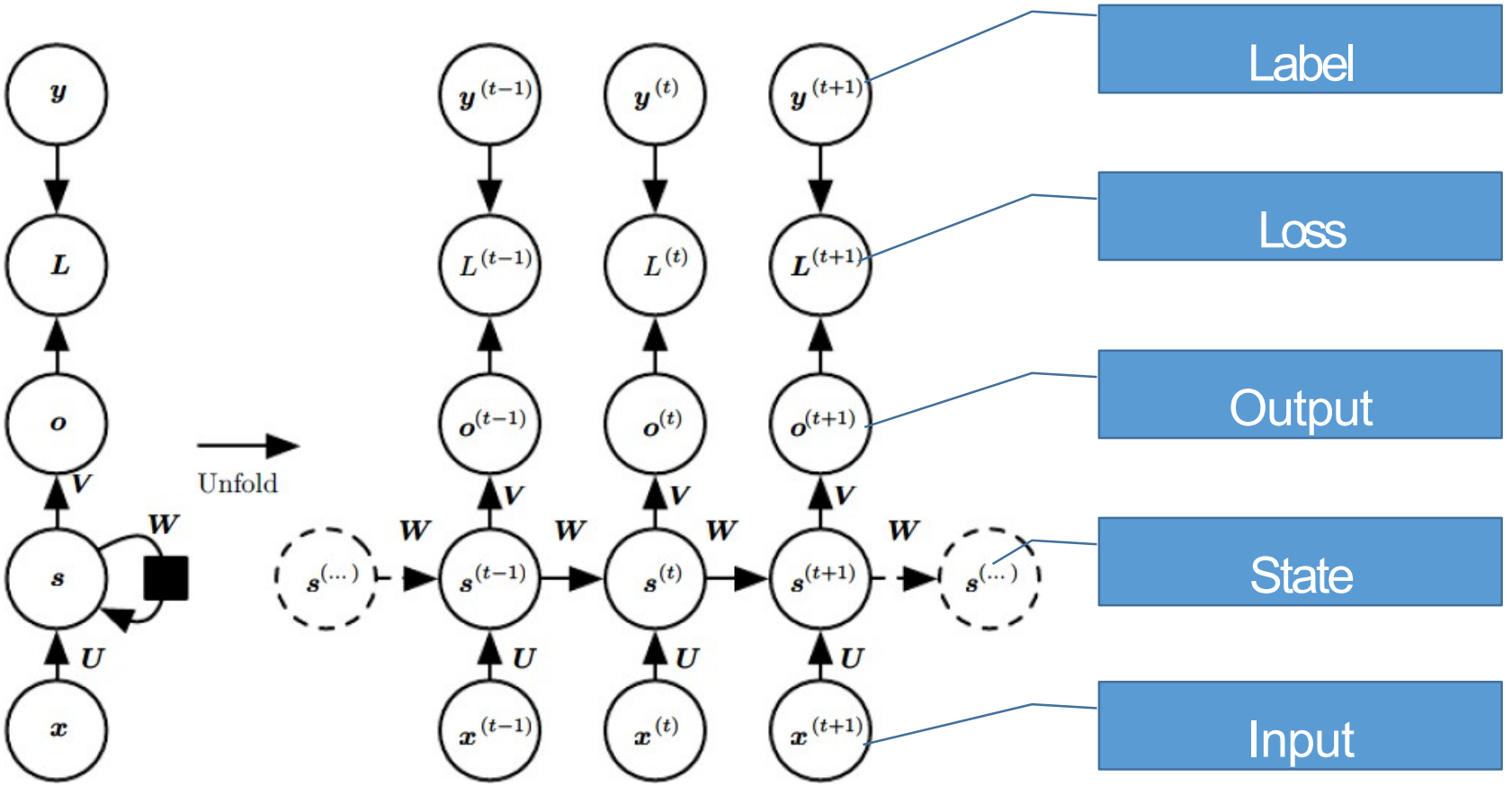
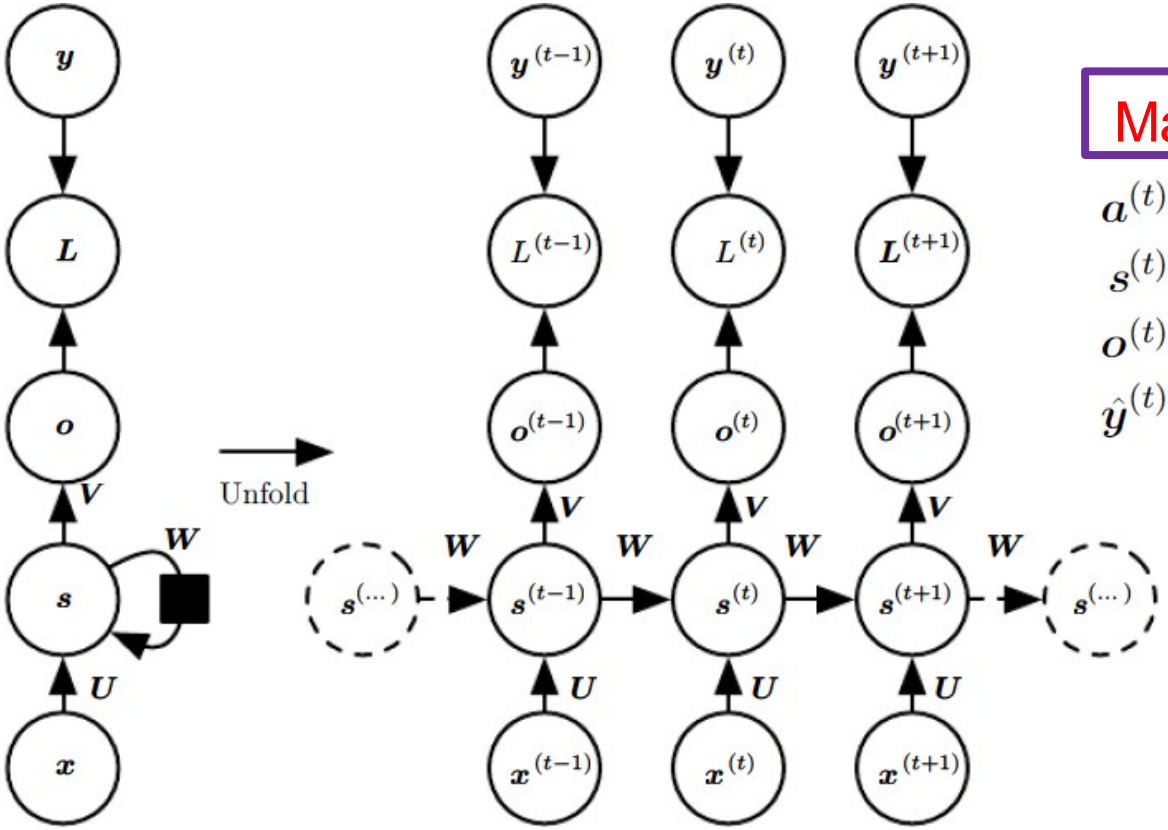


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville

Recurrent neural networks



Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

Advantage

- Yet still powerful (actually universal): any function computable by a Turing machine can be computed by such a recurrent network of a finite size (see, e.g., Siegelmann and Sontag (1995))

Training RNN

- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques
- Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters

RNN

- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed every time step

- **Many variants**
  - Information about the past can be in many other forms
  - Only output at the end of the sequence

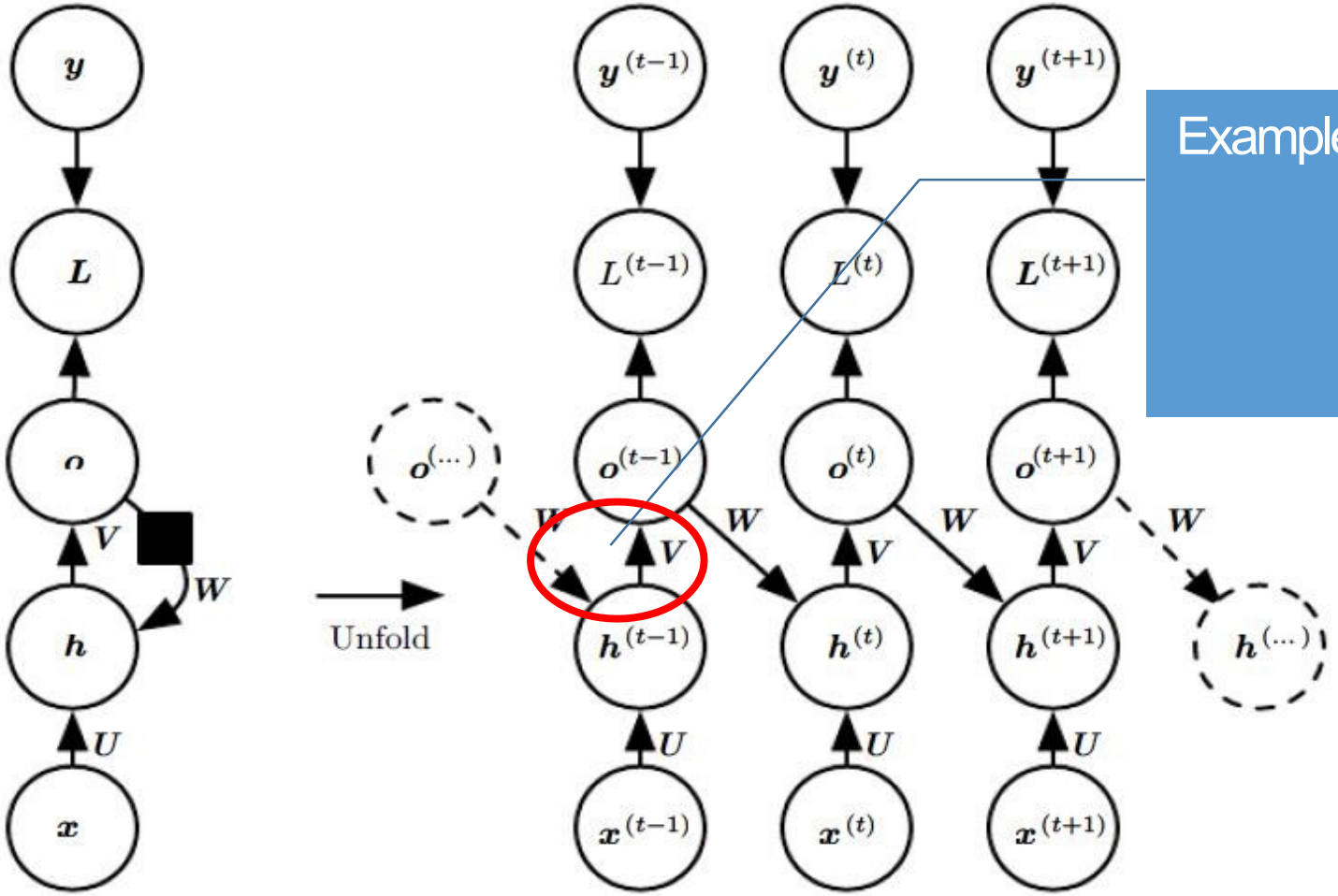


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Example: only output at the end

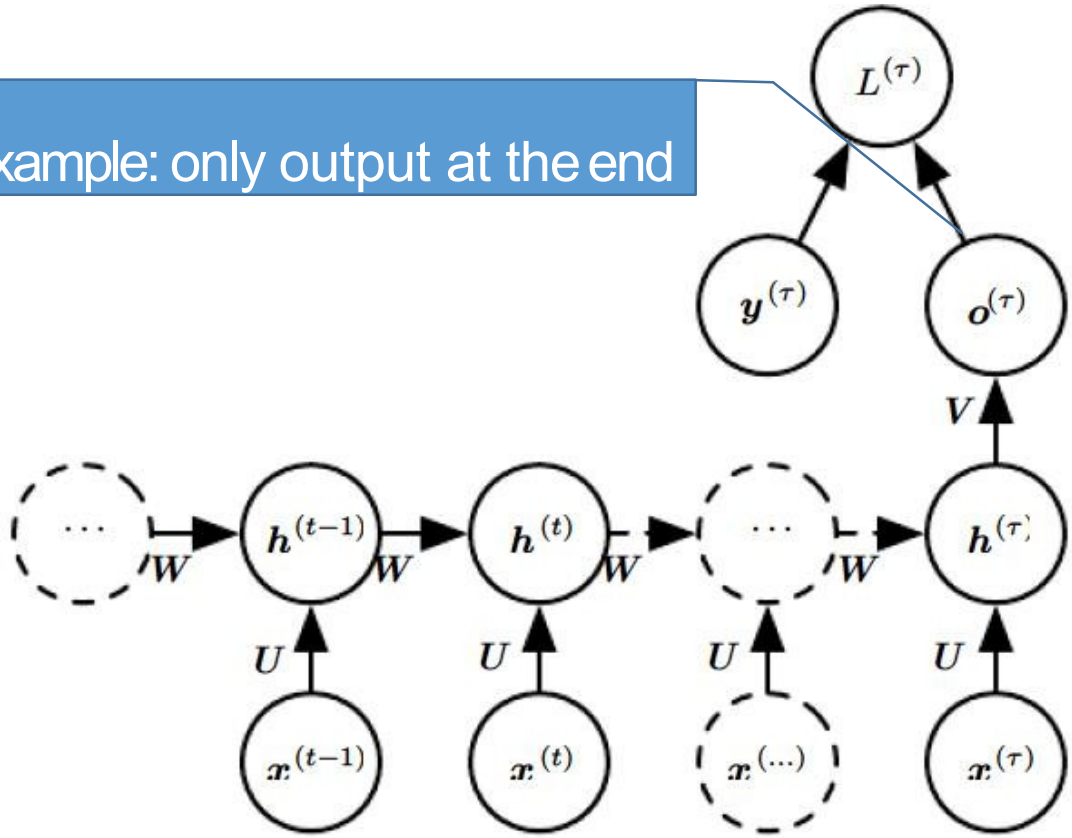


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Bidirectional RNNs

- Many applications: output at time t may depend on the whole input sequence
- Example in speech recognition: correct interpretation of the current sound may depend on the next few phonemes, potentially even the next few words
- Bidirectional RNNs are introduced to address this

BiRNNs

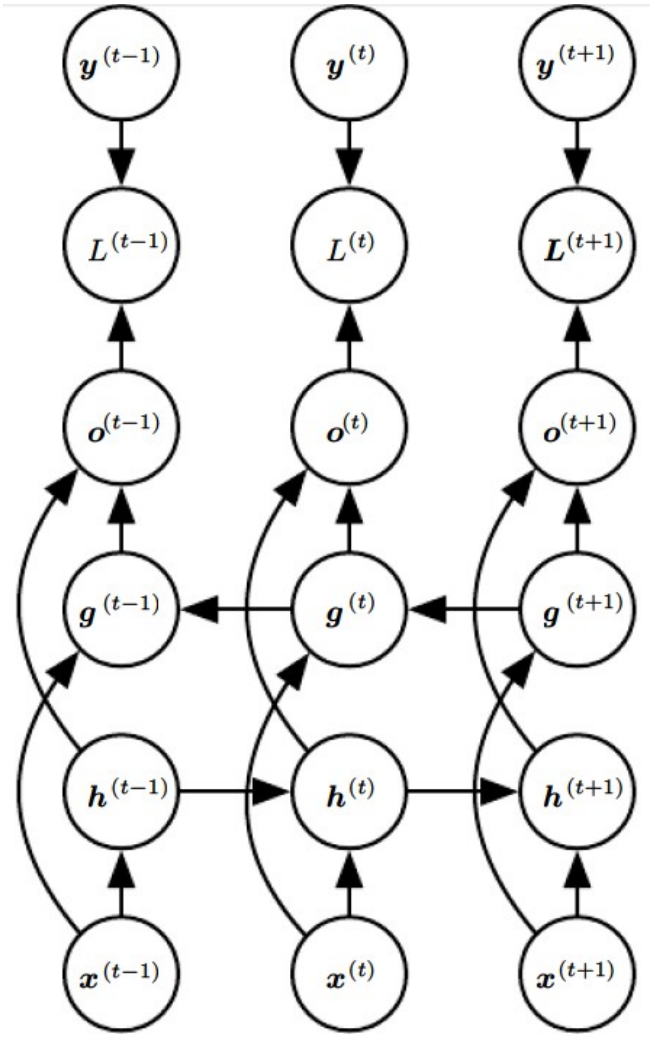


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Encoder-decoder RNNs

- RNNs: can map sequence to one vector; or to sequence of same length
- What about mapping sequence to sequence of different length?
- Example: speech recognition, machine translation, question answering, etc

Encoder-decoder RNNs

Learning to generate an output sequence $\mathbf{(y^{(1)}; \ldots; y^{(n_y)})}$ given an input sequence $\mathbf{(x^{(1)}; x^{(2)}; \ldots; x^{(n_x)})}$.

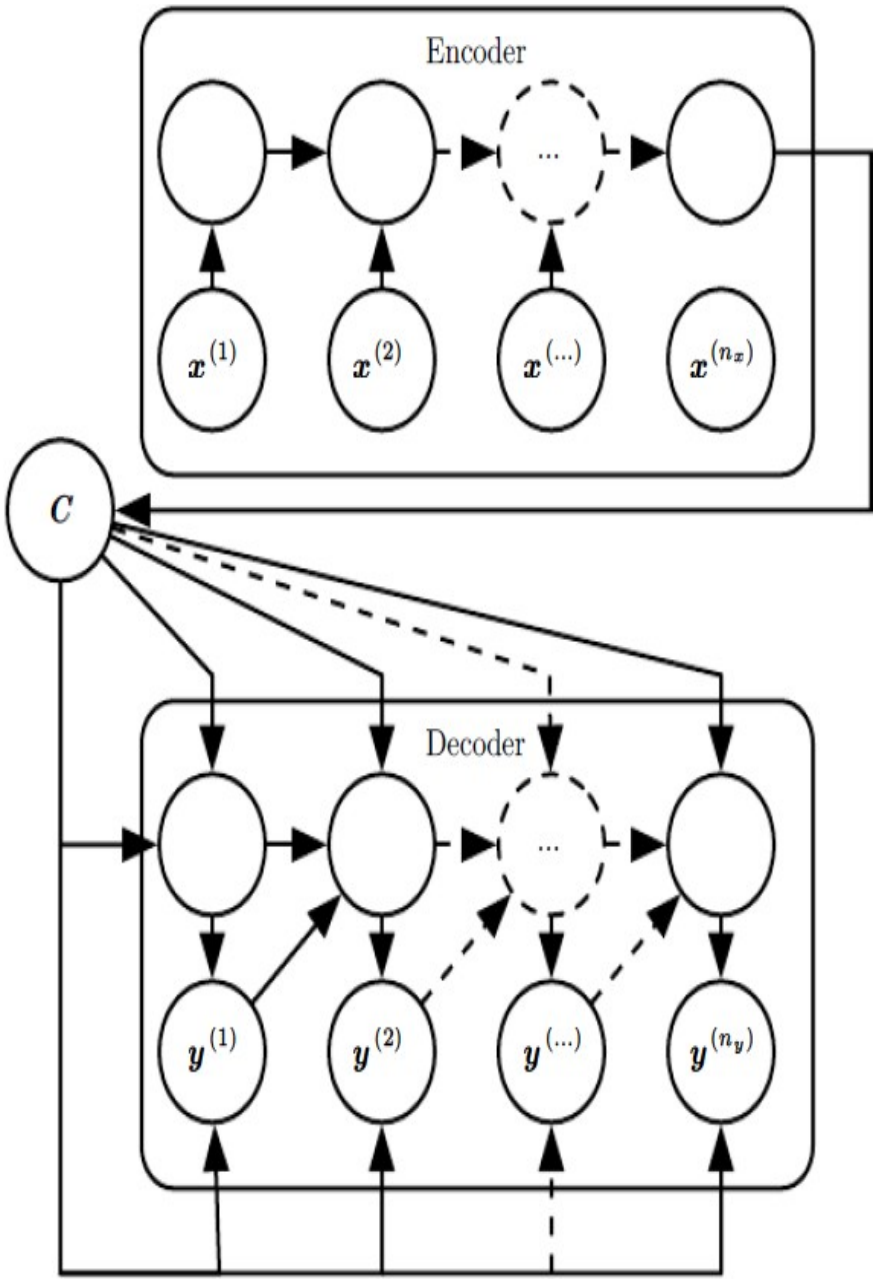


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Encoder-decoder RNNs

- Composed of an encoder RNN that reads the input sequence as well as a decoder RNN that generates the output sequence (or computes the probability of a given output sequence).

Encoder-decoder RNNs

- The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C , which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

The Challenge of Long-Term Dependencies

- Gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization).

The LSTM

- The most effective sequence models used in practical applications are called gated RNNs.
- These include the Long Short-Term Memory (LSTM) and networks based on the Gated Recurrent Unit (GRU).

A lot of material of this lesson on LSTM were adopted from:
A Critical Review of Recurrent Neural Networks for Sequence Learning
https://arxiv.org/abs/1506.00019
and
Understanding LSTM Networks
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

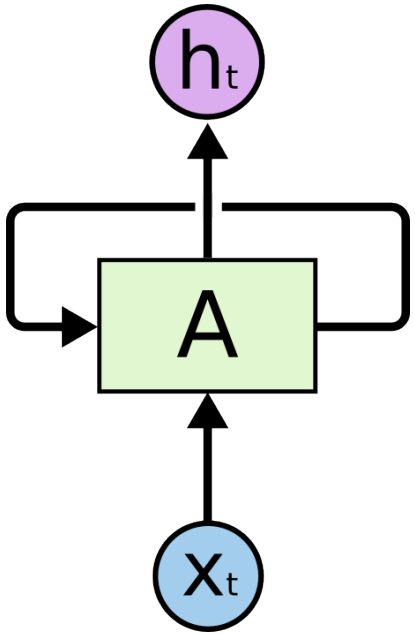
The LSTM

Applications:

- unconstrained handwriting recognition speech recognition
- handwriting generation
- machine translation
- image captioning parsing

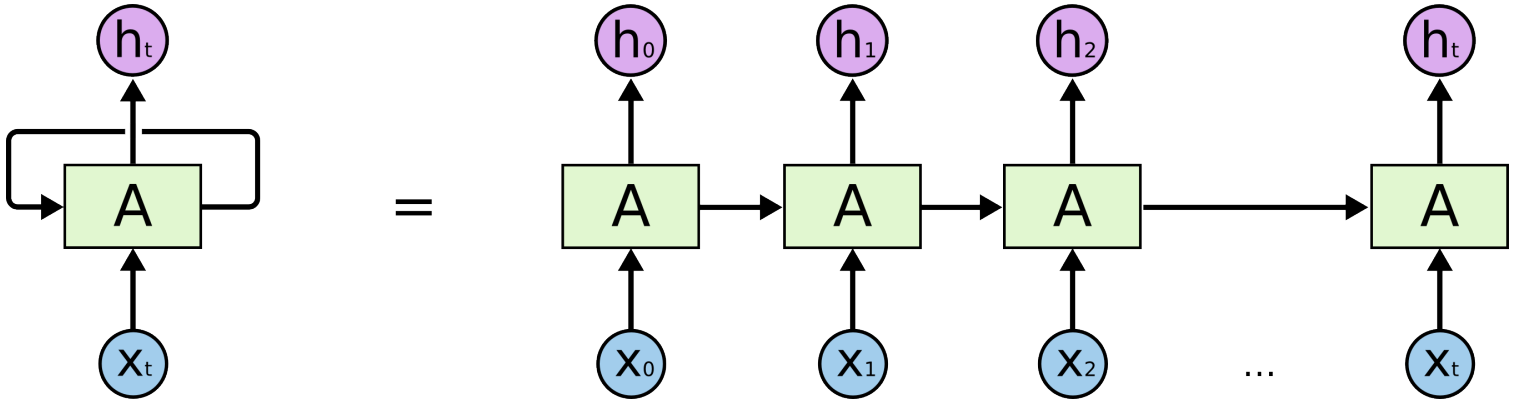
Traditional RNNs

A chunk of neural network, **A**, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.



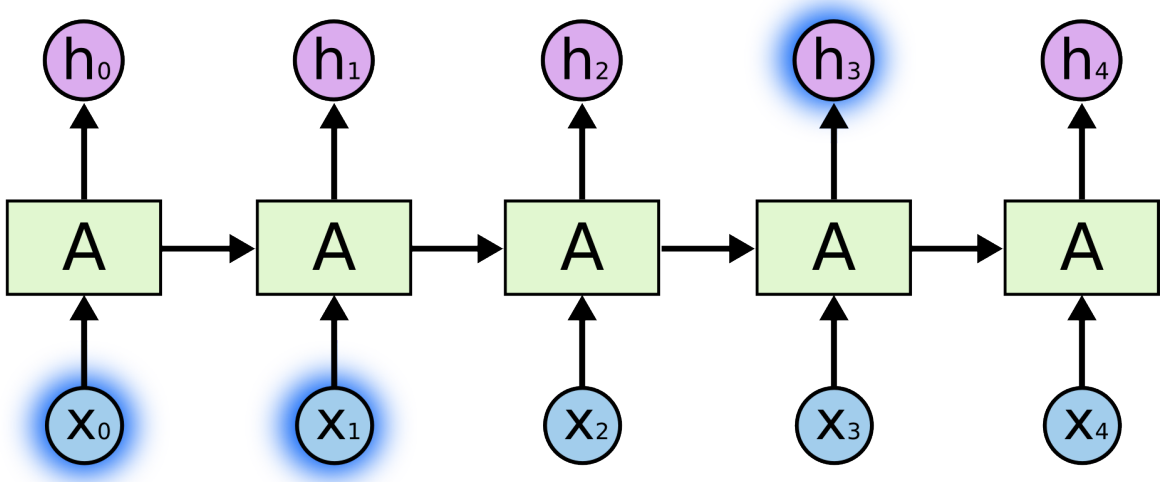
Traditional RNNs

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



The Problem of Long-Term Dependencies

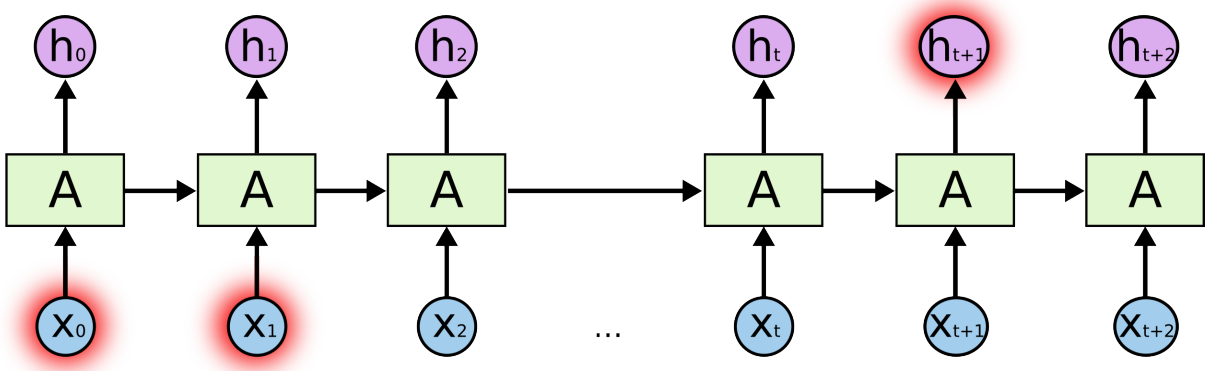
In cases where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



The Problem of Long-Term Dependencies

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.:

"*I grew up in Iran and studied there and in the United States, so I can speak…(Persian and English)*"

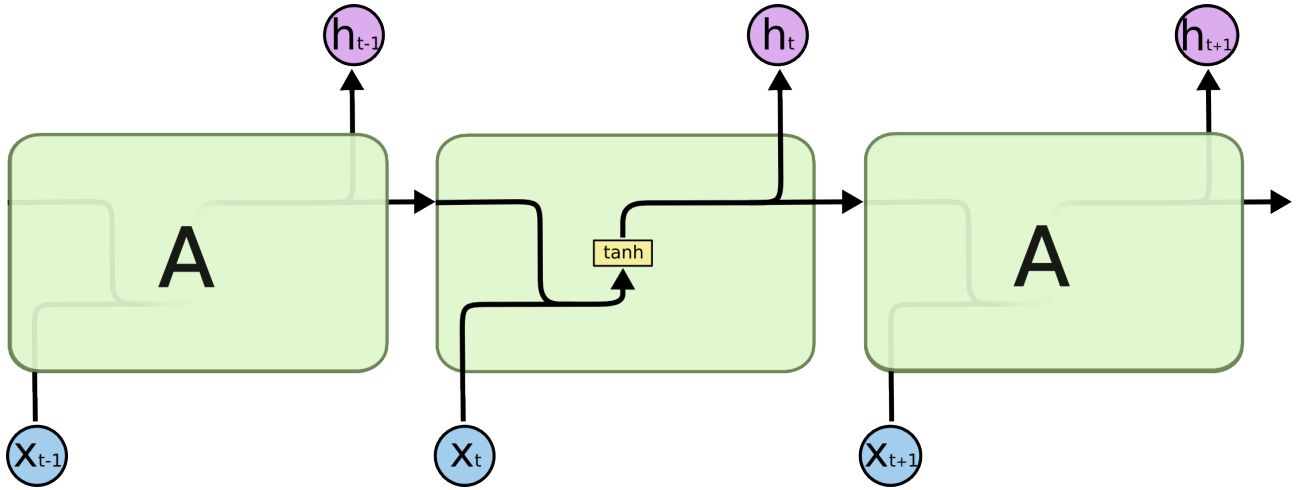


The LSTM

- Long Short Term Memory networks – usually just called “LSTMs” – are RNNs capable of learning long-term dependencies.
- Introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.[1]
- They work tremendously well on a large variety of problems, and are now widely used.

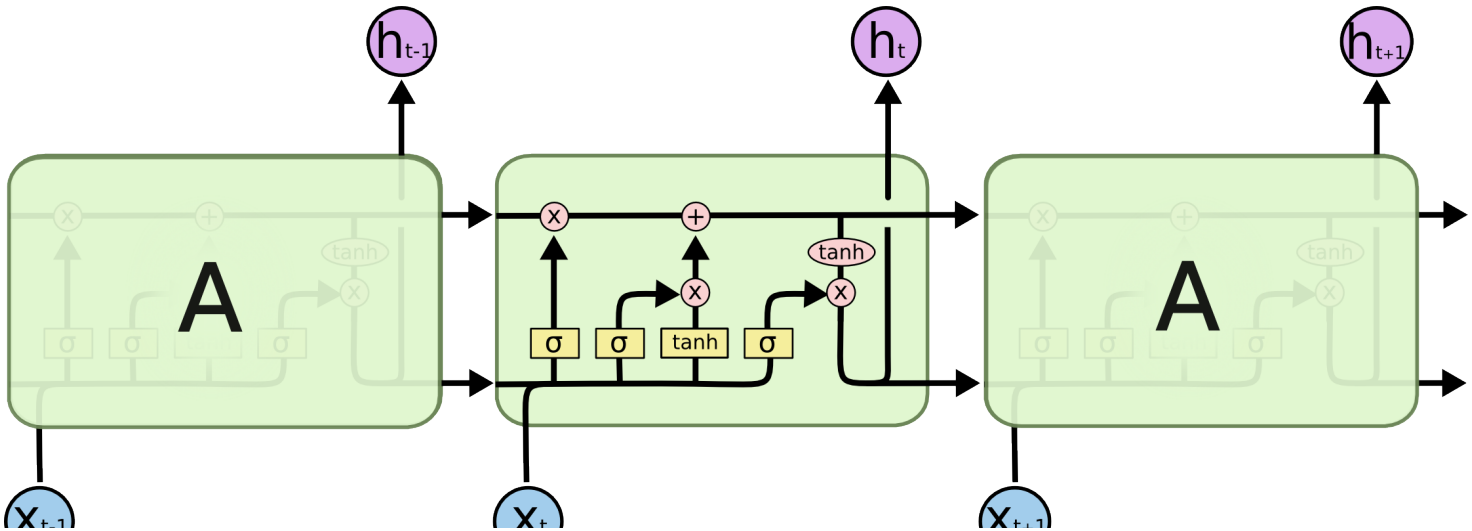
The LSTM

- LSTMs are explicitly designed to avoid the long-term dependency problem.
- RNNs: a chain of repeating modules of neural network that has a very simple structure, such as a single tanh layer.
-



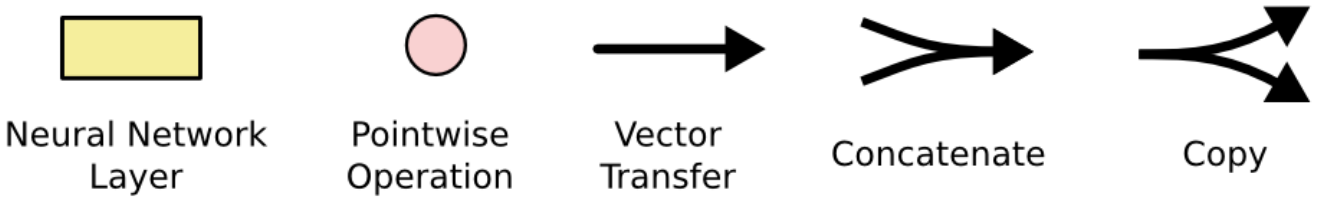
The LSTM

- LSTMs also have this chain like structure, but the repeating module has a different structure.
- Instead of a single neural network layer, there are four, interacting in a very special way. It is called a *memory cell*.



Some Symbols

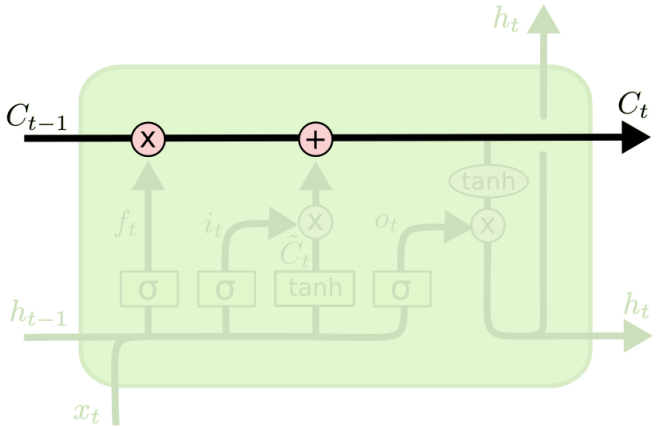
- Some symbols for representing NNs briefly



Pink circles: pointwise operations, e.g. vector addition
Yellow boxes: learned neural network layers

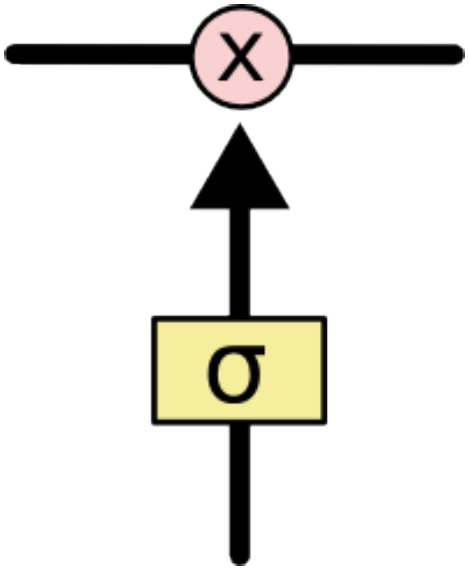
Core Idea: The Cell State

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
- The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



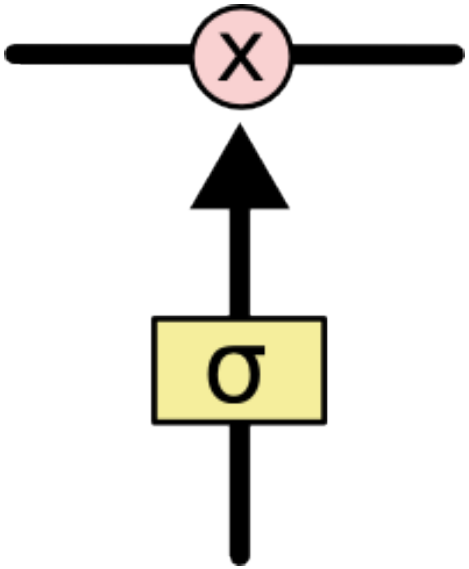
Core Idea: The Cell State

- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called *gates*.
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



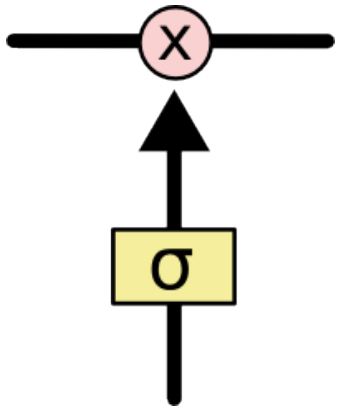
Core Idea: The Cell State

- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called *gates*.
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



Core Idea: The Cell State

- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.
- A value of zero means "let nothing through," while a value of one means "let everything through!"
- An LSTM has three of these gates, to protect and control the cell state.

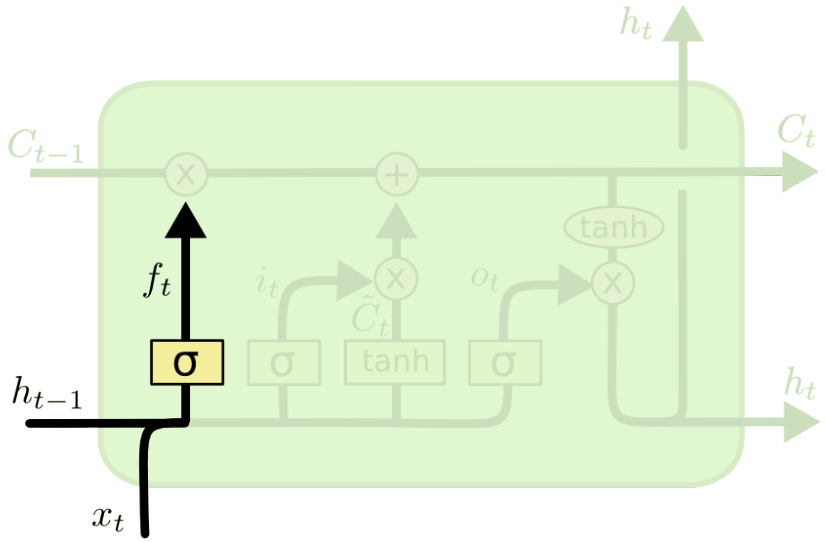


The Memory Cell: The First Step

- The first step for LSTM: decide what information is *thrown away* from the cell state. This decision is made by a sigmoid layer called the "forget gate layer."
- It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.
- A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

The Memory Cell: The First Step

- Example of a language model trying to predict the next word based on all the previous ones:
- The cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



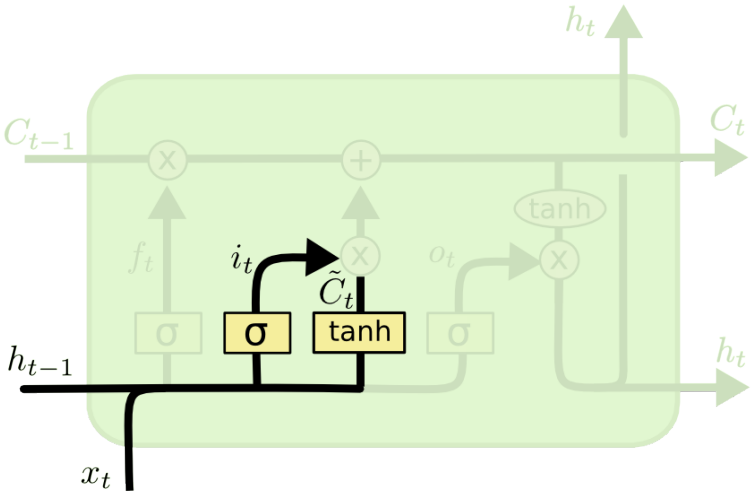
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The Memory Cell: The Second Step

- The next step is to decide what new information is store in the cell state. This has two parts:
  - First, a sigmoid layer called the “input gate layer” decides which values are updated.
  - Next, a *tanh* layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state.
- Then, these two are combined to create an update to the state.

The Memory Cell: The Second Step

- In the language model example, the gender of the new subject needs to be added to the cell state, to replace the old one being forgotten.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

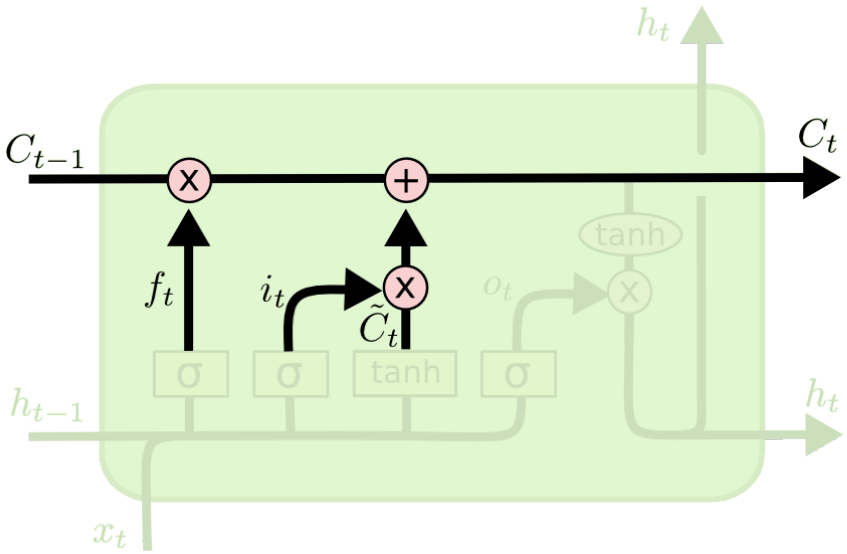
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Updating The Cell State

- It is now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.
- We multiply the old state by f_t, forgetting the things we decided to forget earlier.
- Then we add $i_t \odot \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.
- $\odot$ is the pointwise multiplication operation.

Updating The Cell State

In the case of the language model, this is where the information about the old subject's gender is removed and the new information, as decided in the previous steps, is added.



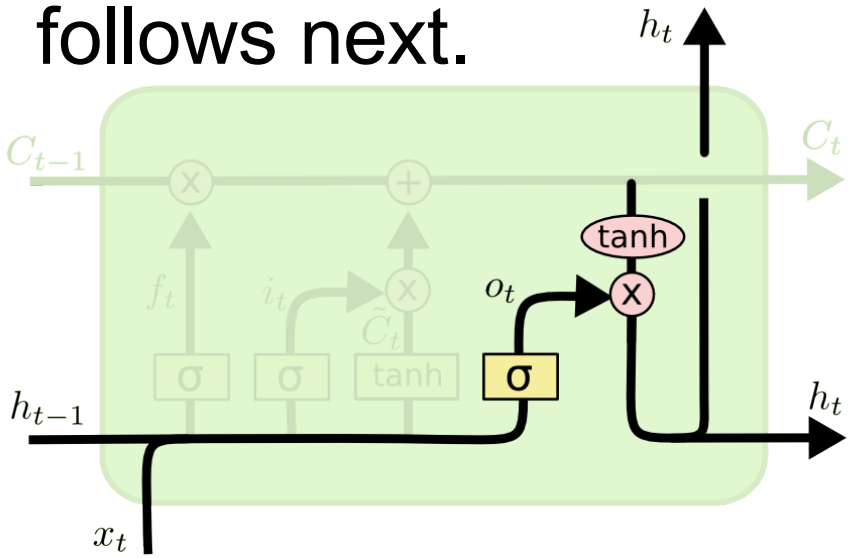
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Output of The Memory Cell

- The output is *filtered version* of the cell state.
- A sigmoid layer decides what parts of the cell state are to be output.
- Then, the cell state passes through through *tanh* (to force the values to be between −1 and 1) and is multiplied by the output of the sigmoid gate, so that it decides the parts to be output.

Output of The Memory Cell

- In language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next.
- For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

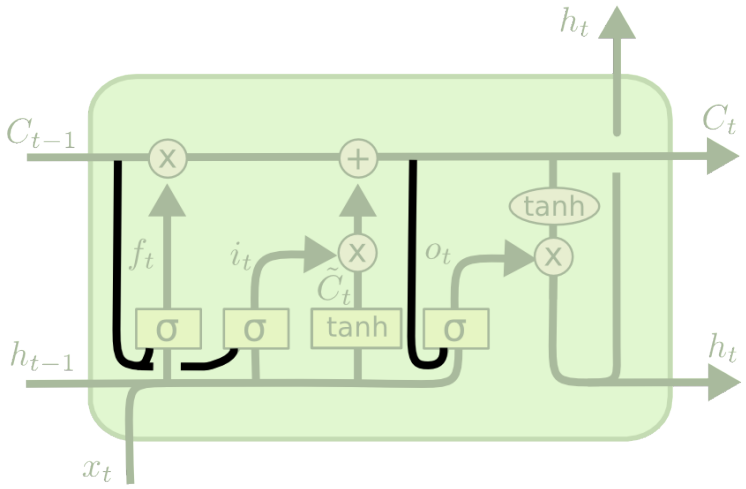


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t \odot \tanh\left(C_t\right)$$

Variants on Long Short Term Memory

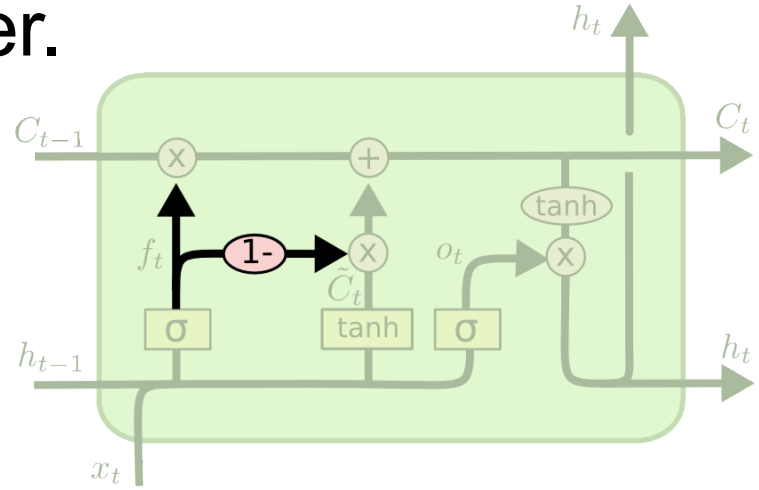
- One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding "peephole connections."
- This means that we let the gate layers look at the cell state.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

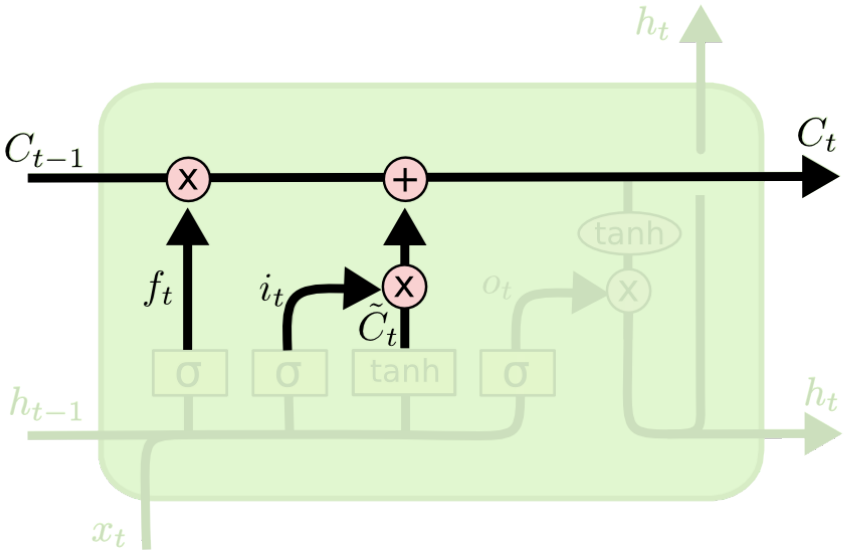
Variants on Long Short Term Memory

- Another variation is to use coupled forget and input gates.
- Instead of separately deciding what to forget and to what new information has to be added, those decisions are made together.
- We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.

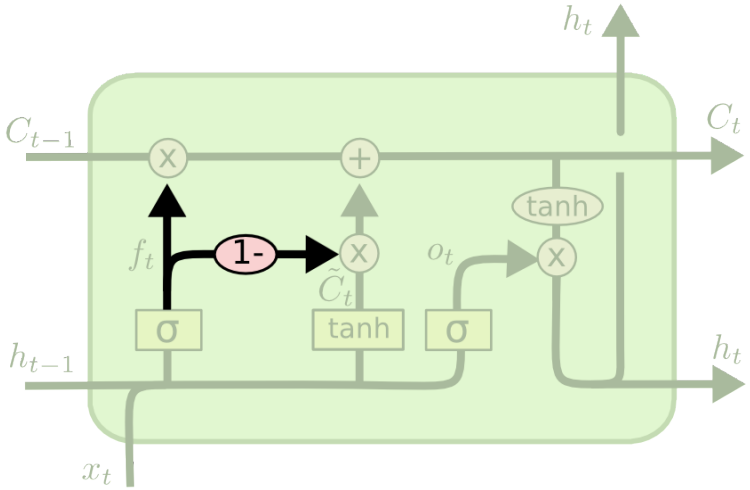


$$C_t = f_t \odot C_{t-1} + (1 - \boldsymbol{f_t}) \odot \tilde{C}_t$$

Variants on Long Short Term Memory



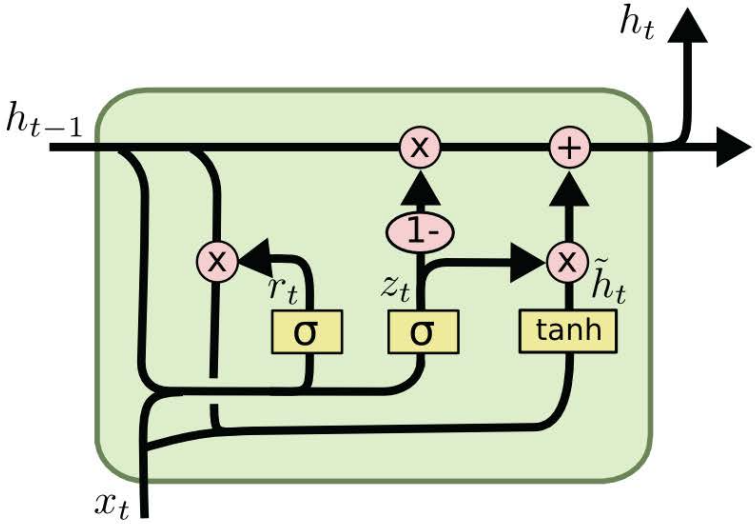
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$



$$C_t = f_t \odot C_{t-1} + (1 - \boldsymbol{f_t}) \odot \tilde{C}_t$$

Variants on Long Short Term Memory

- A slightly more dramatic is the Gated Recurrent Unit, or GRU (Cho, et al. (2014)).
- It combines the forget and input gates into a single "update gate."
- It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t \odot h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Long-Term Dependencies: Clockwork RNNs

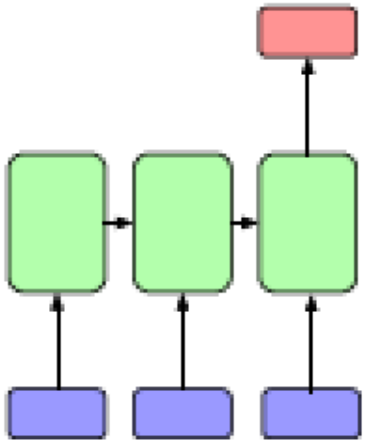
- In Clockwork Recurrent Neural Network, the long-term dependency problem is solved by having different parts (modules) of the RNN hidden layer running at different clock speeds, timing their computation with different, discrete clock periods, hence the name CW-RNN.

Long-Term Dependencies: Clockwork RNNs

- CW-RNNs train and evaluate faster since not all modules are executed at every time step, and have a smaller number of weights compared to RNNs, because slower modules are not connected to faster ones.

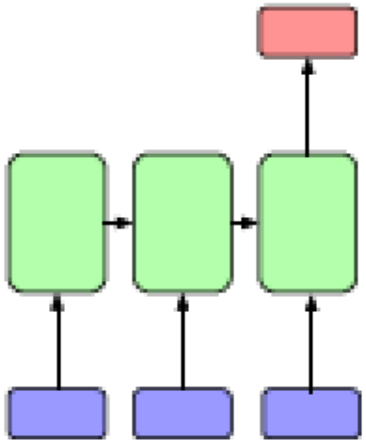
Architectures Constructed Using RNNs

Blue rectangles correspond to inputs, red rectangles to outputs and green rectangles to the entire hidden state of the neural network.



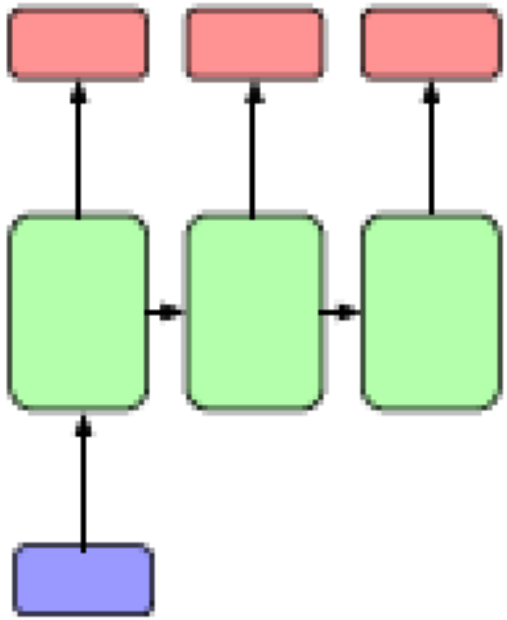
Architectures Constructed Using RNNs

Text and video classification are tasks in which a sequence is mapped to one fixed length vector.



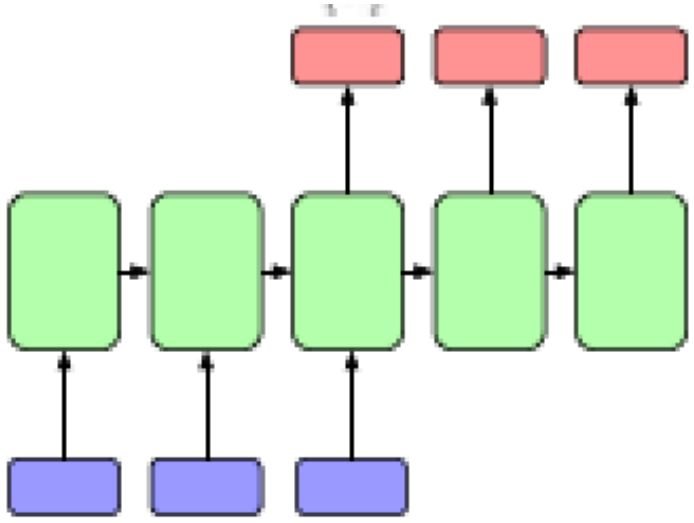
Architectures Constructed Using RNNs

Image captioning presents the converse case, where the input image is a single non-sequential data point.



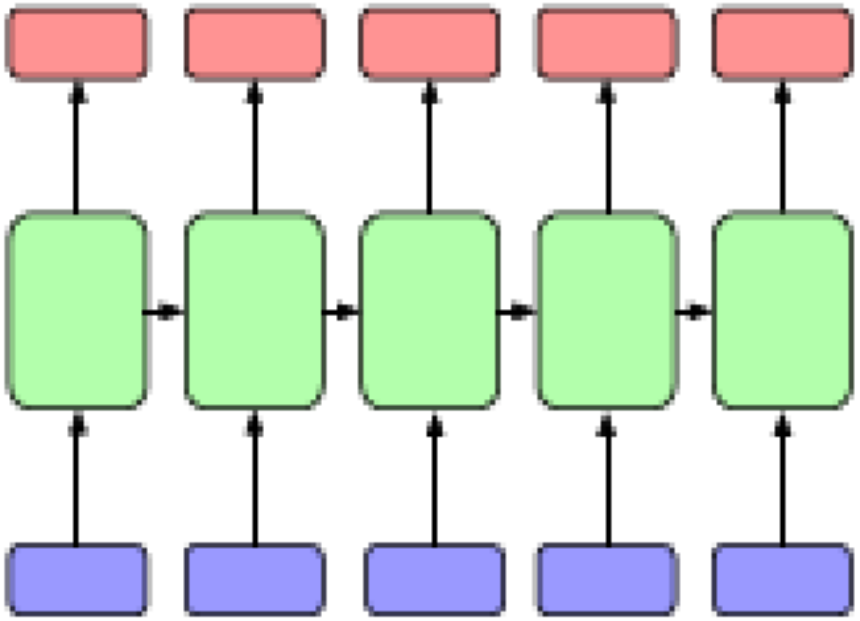
Architectures Constructed Using RNNs

This architecture has been used for natural language translation, a sequence-to-sequence task in which the two sequences may have varying and different lengths



Architectures Constructed Using RNNs

This architecture has been used to learn a generative model for text, predicting at each step the following character.

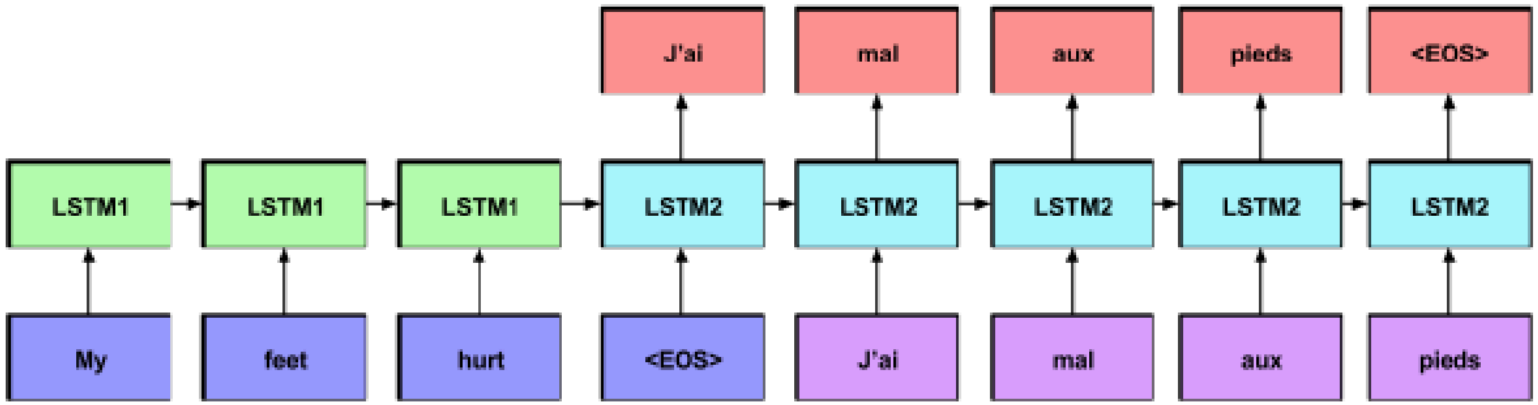


See Appendix

- Representations of Natural Language Inputs and Outputs

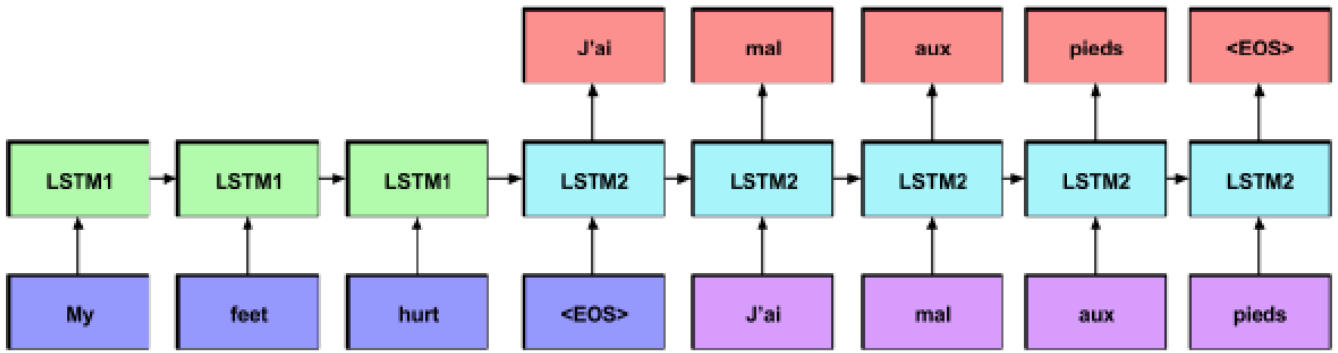
Sequence-to-Sequence LSTM Model

Sutskever et al. [2014] consists:
An encoding model (first LSTM) and a decoding model (second LSTM).



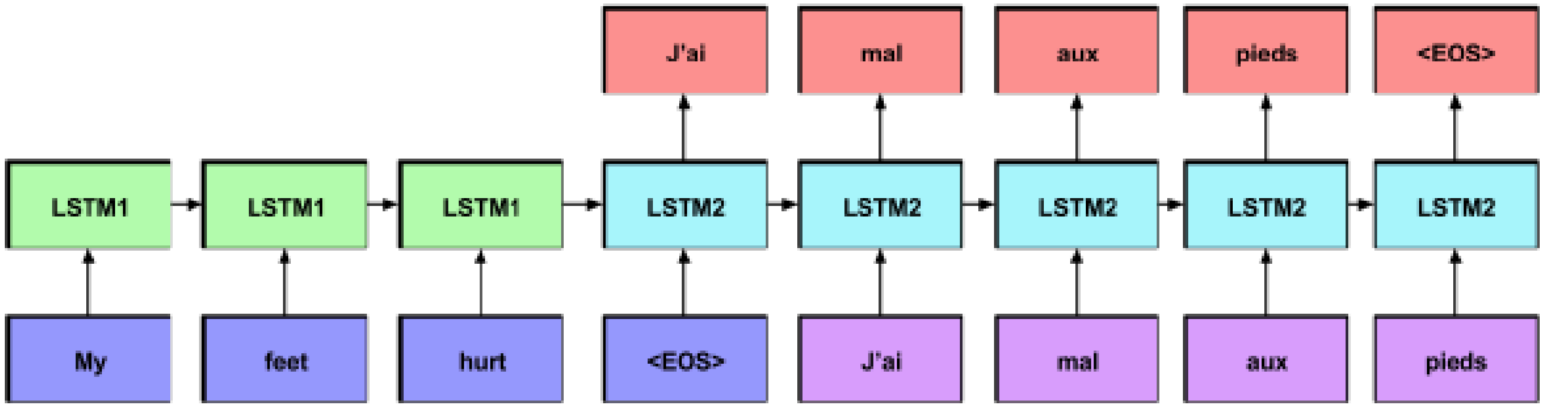
Sequence-to-Sequence LSTM Model

- The input blocks (blue and purple) correspond to word vectors, which are fully connected to the corresponding hidden state.
- Red nodes are softmax outputs.
- Weights are tied among all encoding steps and among all decoding time steps.



Sequence-to-Sequence LSTM Model

For training, the true inputs are fed to the encoder, the true translation
is fed to the decoder, and loss is propagated back from the outputs of the decoder across the entire sequence to sequence model.



See Appendix: Another Representation of LSTM