CS6421: Deep Neural Networks

Gregory Provan

Spring 2020

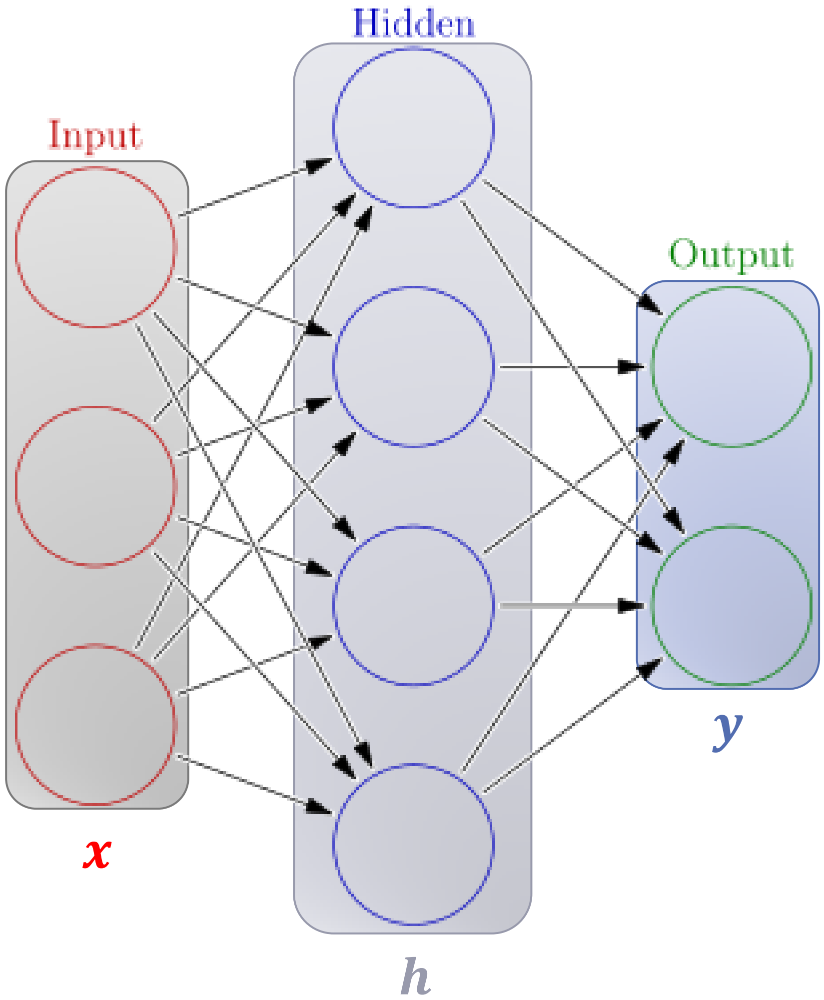
Lecture 3: Modularity and Network Architectures

Based on notes from E. Gavves

Overview

- Modularity in Deep Learning
- Popular Deep Learning modules
- Significance of Modularity

Neural Network: Definition



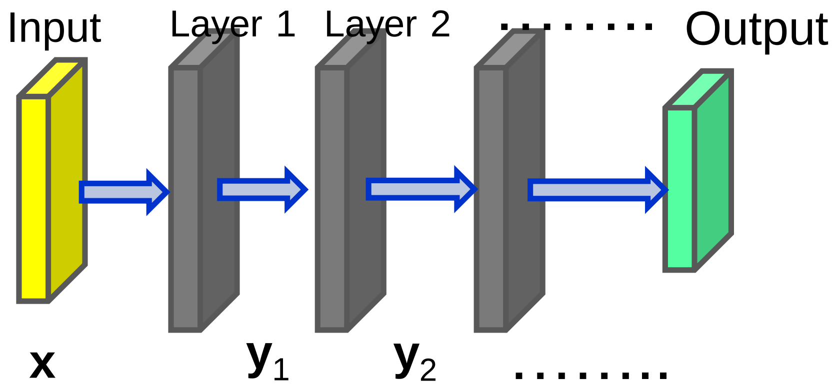
Weights

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

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

Activation functions

Deep Network Representation



$\mathbf{y}_1 = f_1(\mathbf{x}, \theta_1)$

$\mathbf{y}_2 = f_2(\mathbf{y}_1, \theta_2)$

$\mathbf{y}_2 = f_2(f_1(\mathbf{x}, \theta_1), \theta_1)$

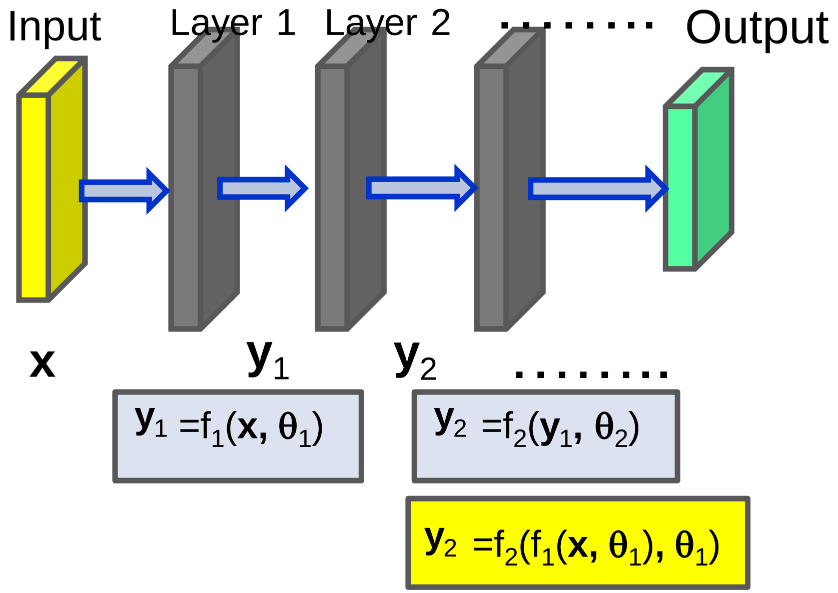
- Structure
- Functions
- Parameters

What do the symbols mean?

- f_i: arbitrary functions (activation functions)
- θ_i: parameters

We choose the f_i and learn the θ_i

Deep Network Structure

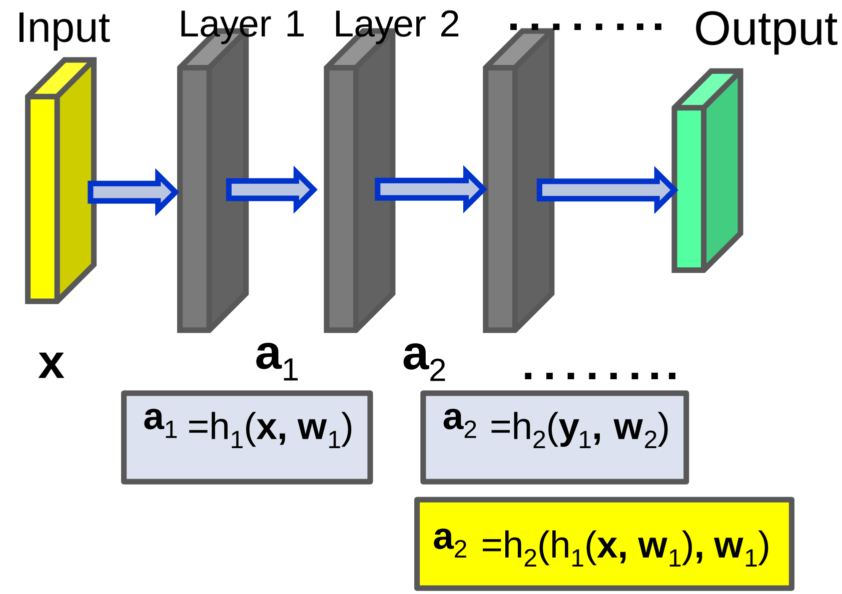


Each layer: matrix/vector

- $\mathbf{y}_i$: matrix/vector of outputs
- $\mathbf{x}$: input matrix/vector

Must use linear algebra for DL operations

Summary: Deep Network



Deep Network: family of parametric, non-linear, hierarchical representations

$$\mathbf{a}_N(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_2, \ldots \mathbf{w}_N) = h_N(h_{N-1}(\ldots(h_1(\mathbf{x}, \mathbf{w}_1), \mathbf{w}_{N-1}), \mathbf{w}_N)$$

Training: optimize network parameters to minimise loss over training set

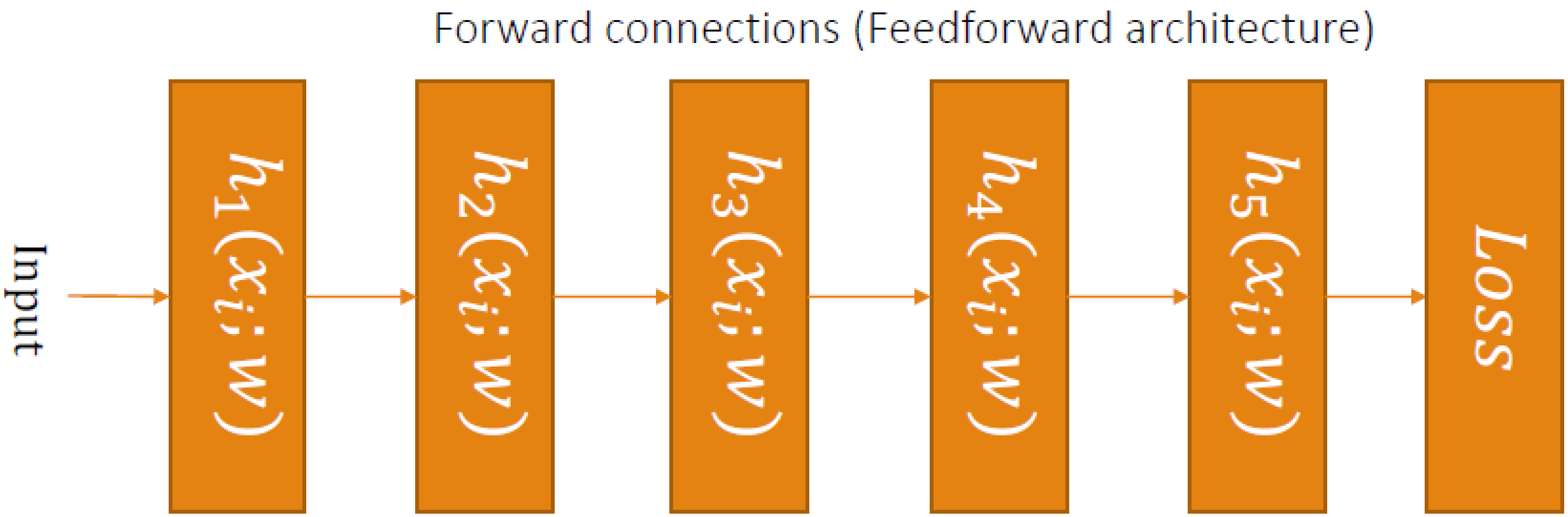
$$w^* = \arg\min_w \sum_{(x,y)\subseteq(X,Y)} J[a, h_L(x, \mathrm{w}_{1,\ldots L})]$$

Neural Network: Definition

- A family of parametric, non-linear and hierarchical representation learning functions
  - massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- $a^L(x; w^1, \ldots, w^L) = h^L(h^{L-1} \ldots h^1(x, w^1), w^{L-1}), w^L)$
  - x:input,
  - w^l: parameters for layer l,
  - $a^l = h^l(x, w^l)$: (non-) linear function
- Given training corpus $\{X, Y\}$ find optimal parameters
  - $w^* \leftarrow \mathrm{argmin}_w \sum_{(x,y)\subseteq(X,Y)} L(y, aL(x))$

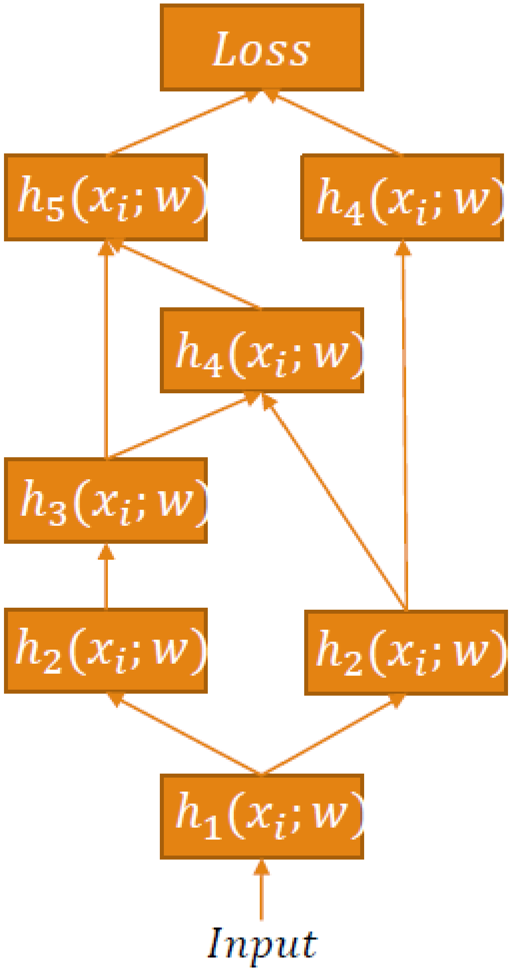
Architectural View of Deep Networks

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very complex



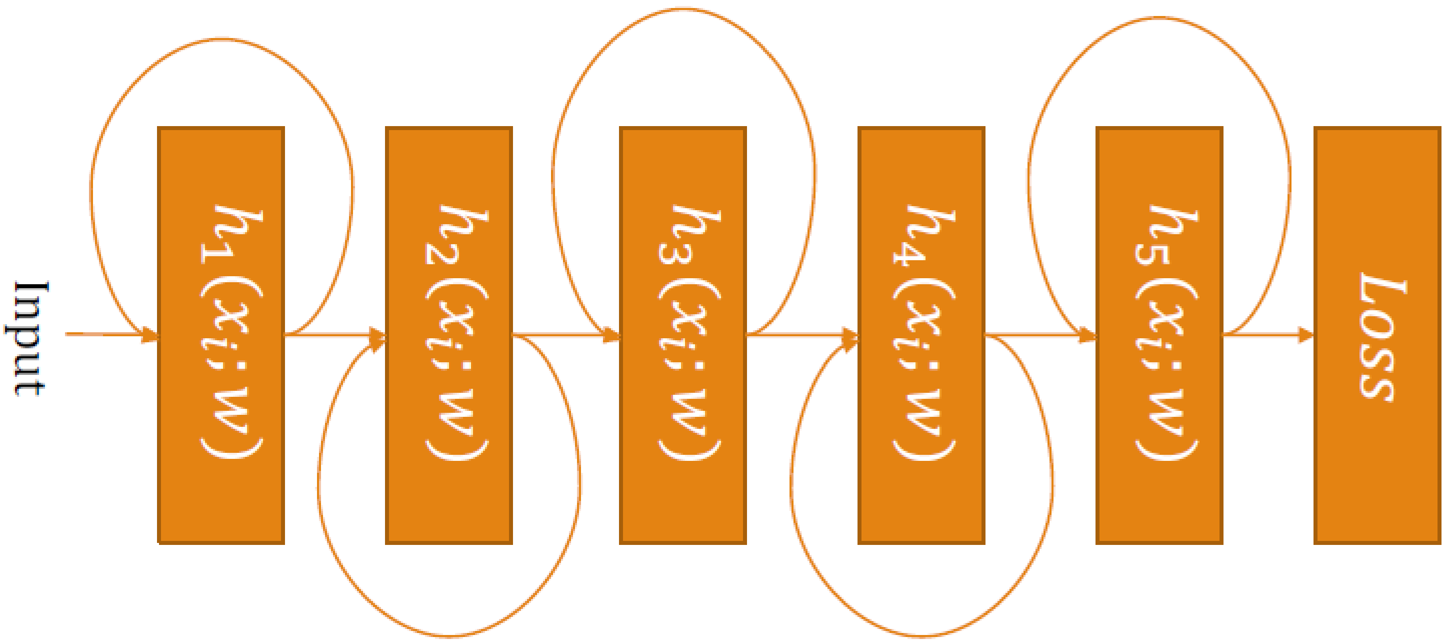
Abstract Architecture

- DAG structure
  - Directed acyclic graph



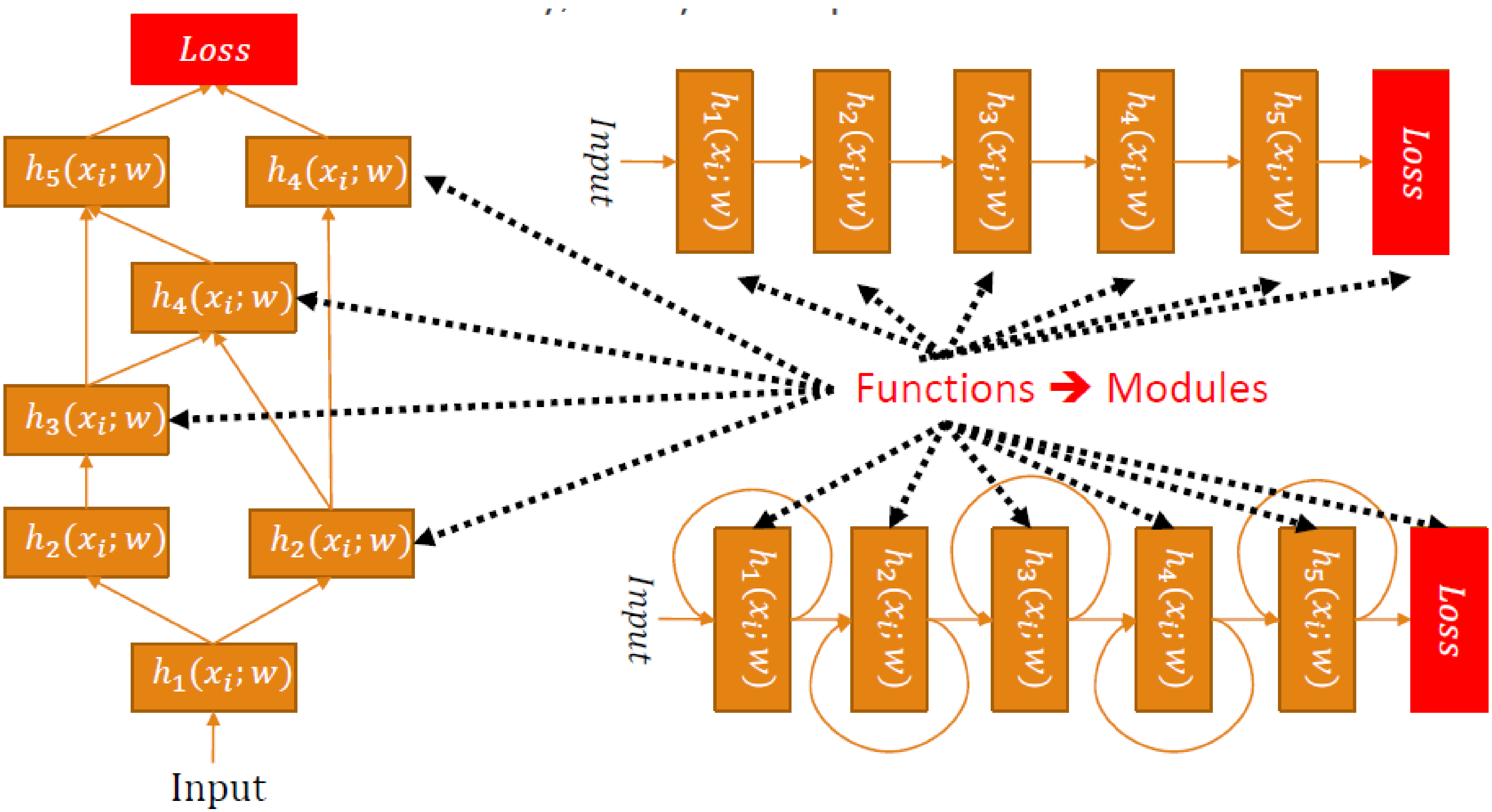
Example: RNN Architecture

- Want to capture temporal dependencies



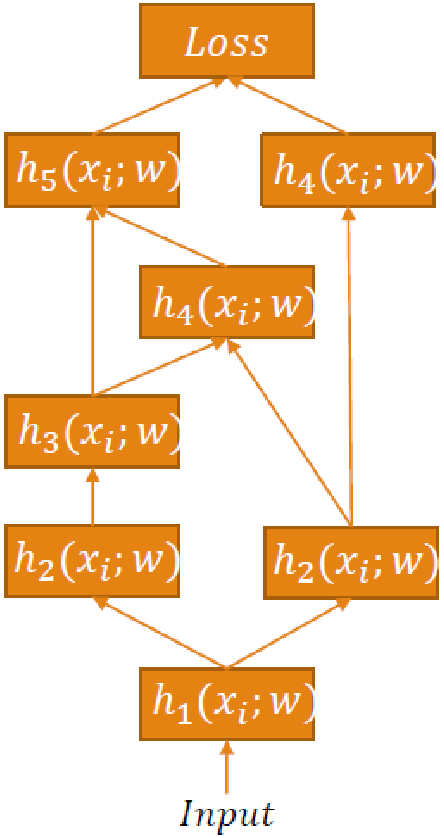
Loopy connections (Recurrent architecture, special care needed)

Modular Structure Determines Function



Module

- A module is a building block for our network
- Each module is an object/function $a = h(x; w)$ that
  - Contains trainable parameters w
  - Receives as an argument an input x
  - Returns an output a based on the activation function $h(\dots)$
- The activation function should be (at least) first-order differentiable (almost) everywhere
  - Required for BackPropagation
- For easier/more efficient backpropagation $\rightarrow$ store module input
  - easy to get module output fast
  - easy to compute derivatives

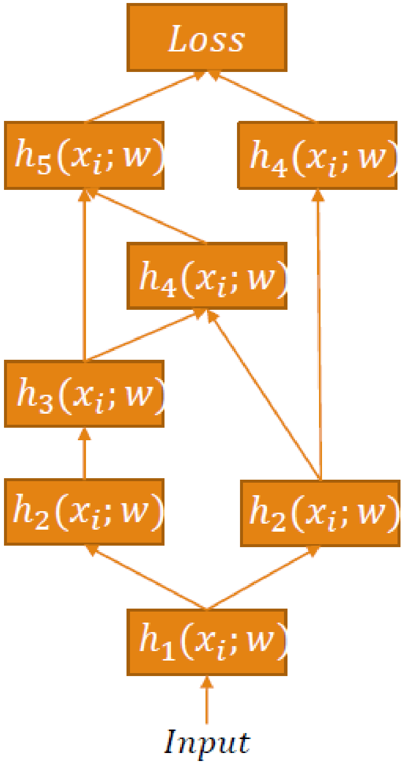


Modular Composition

- A neural network is a composition of modules (building blocks)
- Any architecture works (in theory)
- If the architecture is a feedforward cascade, no special care needed
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form recurrent connections (studied later)

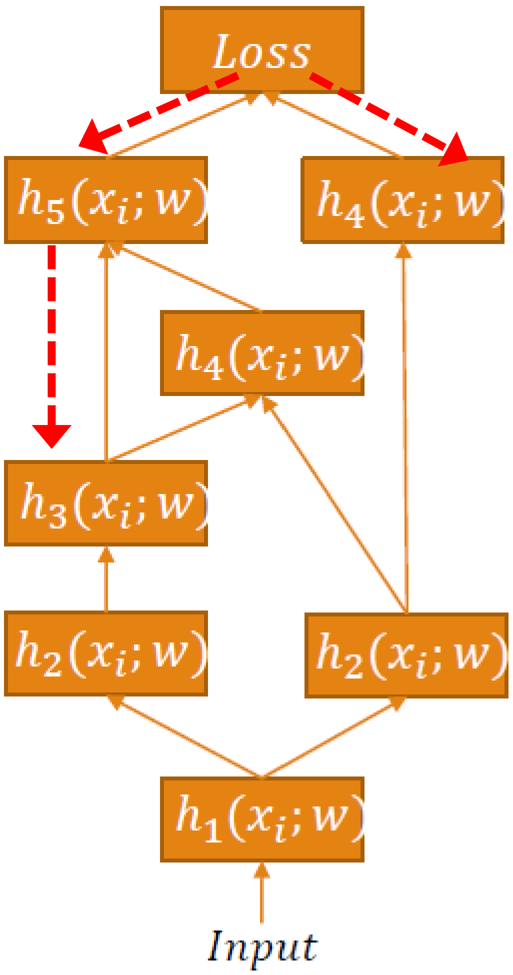
Forward Computations

- Simply compute the activation of each module in the network
- $a^l = h^l(x^l; w)$, where $a^l = x^{l+1}$
- Must know the precise function behind each module $h^l(\ldots)$
- Recursive operations
  - One module's output is another's input
- Steps
  - Visit modules one by one starting from the data input
  - Some modules might have several inputs from multiple modules
- Compute modules activations with the right order
  - Make sure all the inputs computed at the right time



Why is Differentiability Important?

- Modules must work with Backpropagation
- Use partial derivatives to backpropagate errors



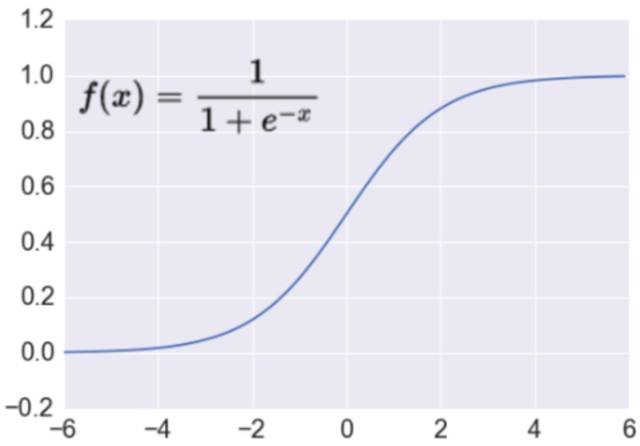
Must use “Good” Functions in Modules

- Some functions perform better than others in particular roles
  - E.g., sigmoid vs. ReLU as activation
  - Loss: squared-error vs. cross-entropy
- Must understand functional properties to build high-performance Deep-Networks

Examples of Functional Modules

- Sigmoid
- tanh
- ReLU

Sigmoid Module



http://adilmoujahid.com/images/activation.png

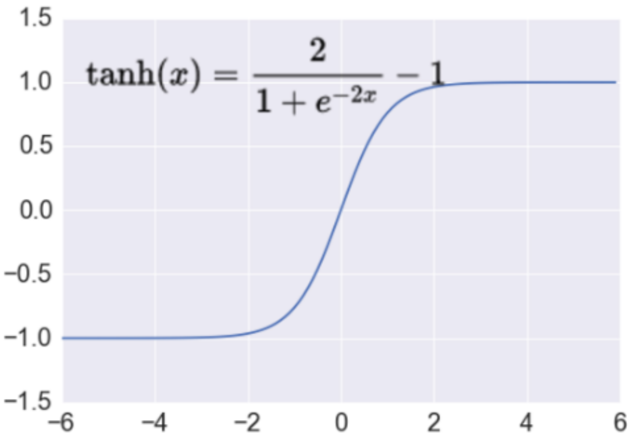
Takes a real-valued number and "squashes" it into range between 0 and 1.

$$R^n \rightarrow [0,1]$$

+ Nice interpretation as the **firing rate** of a neuron
  - 0 = not firing at all
  - 1 = fully firing

- Sigmoid neurons **saturate** and **kill gradients**, thus NN will barely learn
  - when the neuron's activation are 0 or 1 (saturate)
    - gradient at these regions almost zero
    - almost no signal will flow to its weights
    - if initial weights are too large then most neurons would saturate

Tanh Module



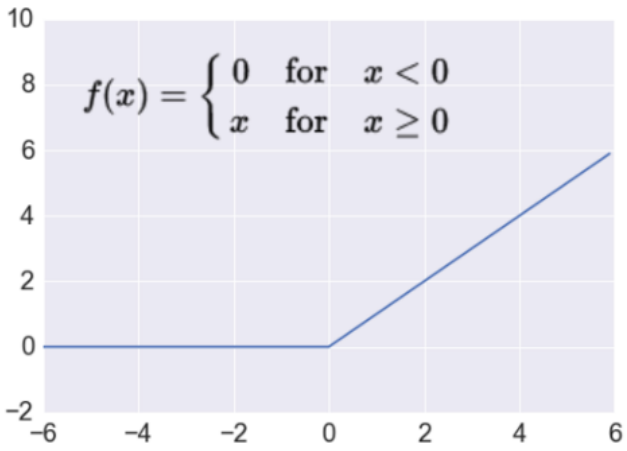
http://adilmoujahid.com/images/activation.png

Takes a real-valued number and "squashes" it into range between -1 and 1.

$$R^n \rightarrow [-1,1]$$

- Like sigmoid, tanh neurons **saturate**
- Unlike sigmoid, output is **zero-centered**
- Tanh is a **scaled sigmoid**: $\tanh(x) = 2sigma(2x) - 1$

ReLU Module



http://adilmoujahid.com/images/activation.png

Takes a real-valued number and thresholds it at zero $\mathrm{f}(x) = \max(0, x)$

$$R^n \rightarrow R^n_+$$

Most Deep Networks use ReLU nowadays

- Trains much **faster**
  - accelerates the convergence of SGD
  - due to linear, non-saturating form
- Less expensive operations
  - compared to sigmoid/tanh (exponentials etc.)
  - implemented by simply thresholding a matrix at zero
- More **expressive**
- Prevents the **gradient vanishing problem**

Centered Non-Linearities

- Remember: a deep network is a hierarchy of similar modules
  - One ReLU is the input to the next ReLU
- Consistent behaviour $\rightarrow$ input/output distributions must match
  - Otherwise, you will soon have inconsistent behaviour
  - If ReLU-1 returns always highly positive numbers, e.g. ~10,000
    - the next ReLU-2 biased towards highly positive or highly negative values (depending on the weight w)
    - ReLU (2) essentially becomes a linear unit.
- We want our non-linearities to be mostly activated around the origin (centred activations)
  - the only way to encourage consistent behaviour without constraining the architecture

New Modules

- Everything can be a module, given some ground rules
- How to make our own module?
  - Write a function that follows the ground rules
    - Needs to be (at least) first-order differentiable (almost) everywhere
- Hence, we need to be able to compute the partial derivatives
  - $\partial a(x;\theta)/\partial x$ and $\partial a(x;\theta)/\partial \theta$

Module of Modules

- As everything can be a module, a module of modules could also be a module
- Can make new building blocks as we please, if we expect them to be used frequently
- The same rules for eligibility of modules still apply

1 sigmoid ≈ 2 modules

- Assume the sigmoid $\sigma(\ldots)$operating on top of wx
  - $a=\sigma(wx)$
- Directly computing it
  - $\rightarrow$ complicated backpropagation equations
- Since everything is a module, we can decompose this to 2 modules
  - $\boxed{a_1 = wx} \rightarrow \boxed{a_2 = \sigma(a_1)}$

1 sigmoid ≈ 2 modules

- Two backpropagation steps instead of one
- Gradients are simpler
  - Algorithmic way of computing gradients
  - Avoid taking more gradients than needed in a (complex) non-linearity

$$\boxed{a_1 = wx} \rightarrow \boxed{a_2 = \sigma(a_1)}$$

Many Modules are Possible

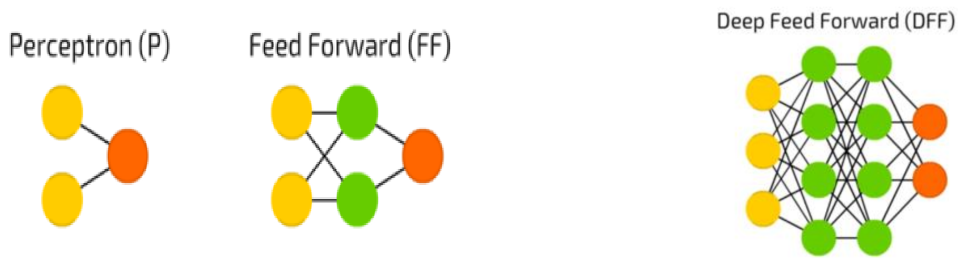
- Many will work comparably to existing ones
  - Not interesting, unless they work consistently better and there is a reason
- Regularization modules
  - Dropout
- Normalization modules
  - ℓ_2-normalization, ℓ_1-normalization
- Loss modules
  - Hinge loss
- Most concepts discussed in the course can be defined as modules

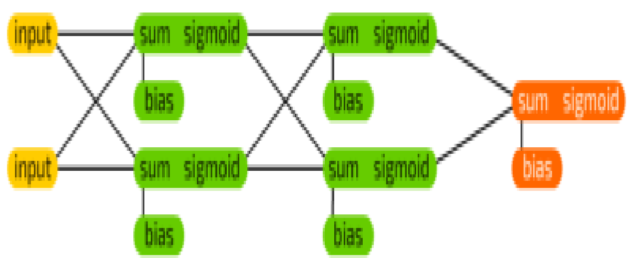
Deep Network Architectures

- Architecture
  - Modular structure of a deep network
- Architecture is critical to good performance
  - Arbitrary structure may work, but be inefficient
- Architecture is application-specific
  - Network classes have particular “base” architectures

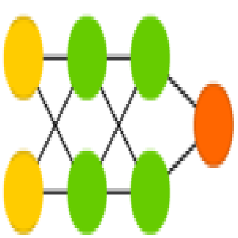
Zoo of Architectures

Asimovinstitute.org

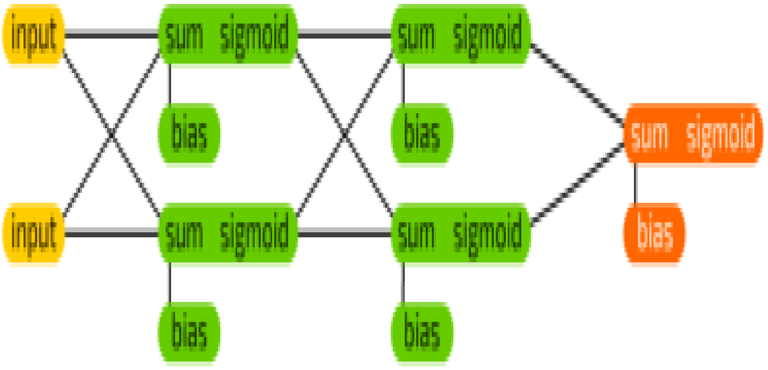




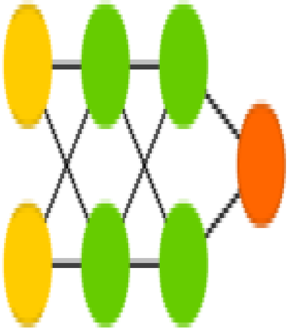
Deep Feed Forward Example



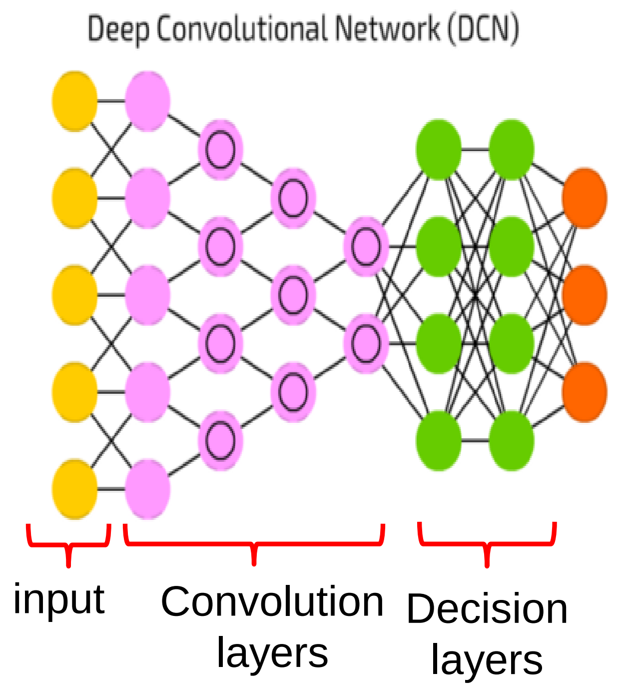
Deep FeedForward Architecture



Deep Feed Forward Example



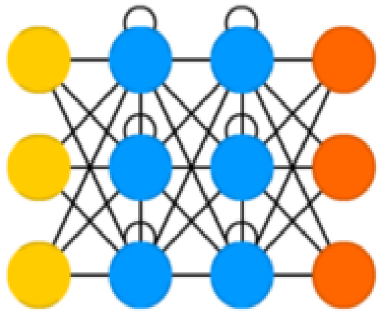
Convolution Network Architecture



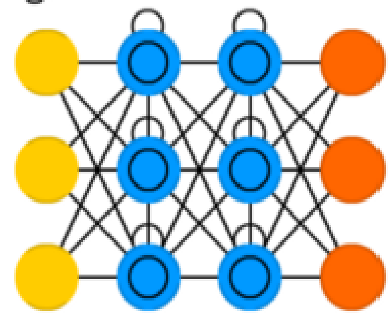
- Architecture designed for structured inputs
- Two main sub-structures
  - Convolution
  - Decision

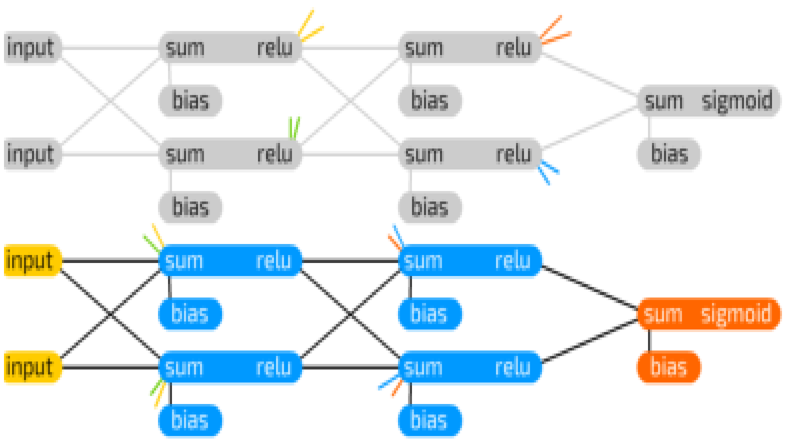
Temporal Architectures

Recurrent Neural Network (RNN)



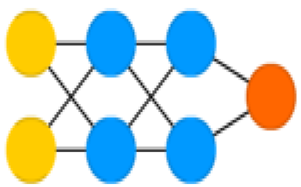
Long / Short Term Memory (LSTM)



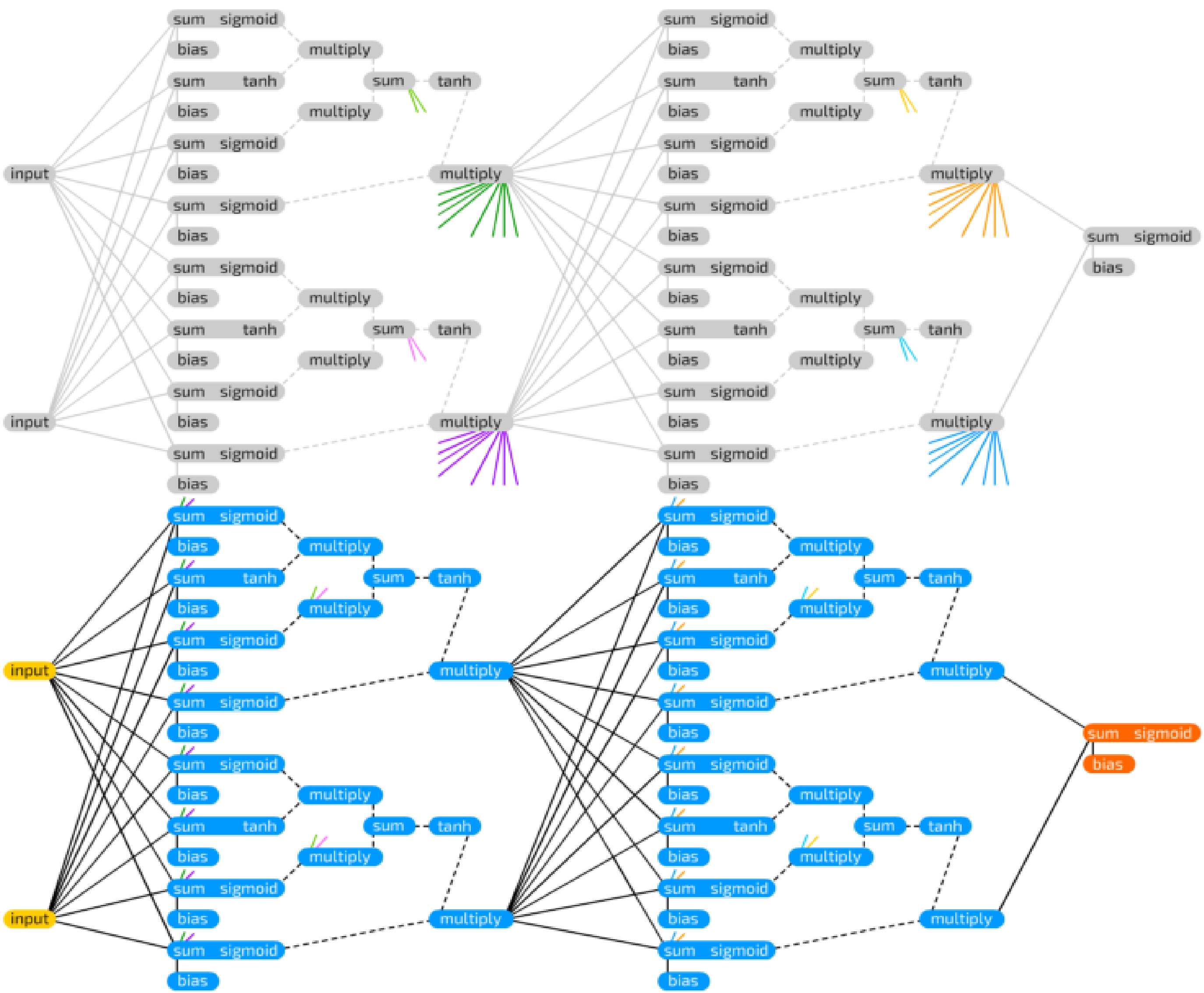


Deep Recurrent Example (previous iteration)

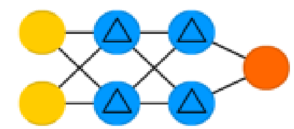
Deep Recurrent Example



LSTM Structure (Temporal)



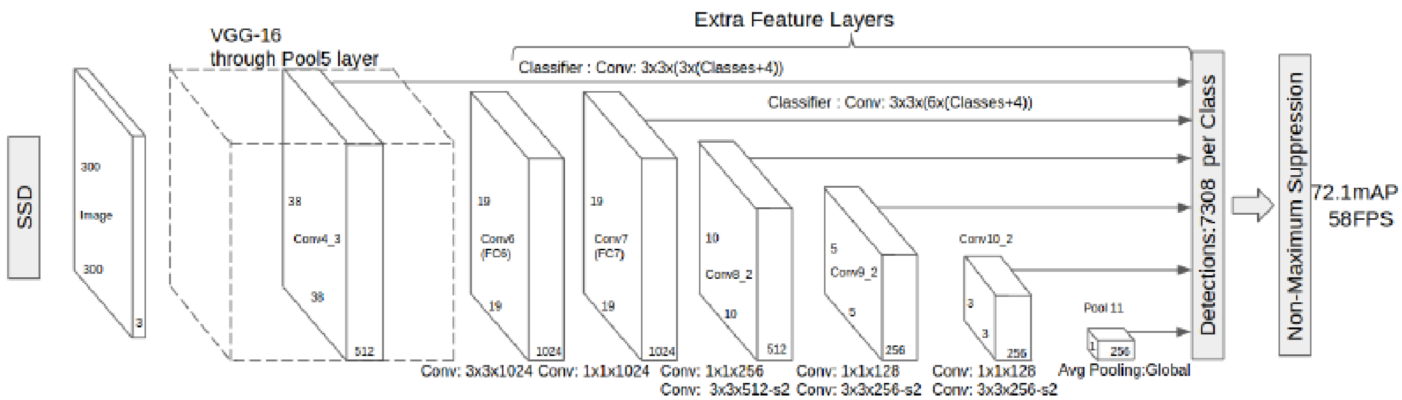
Deep LSTM Example
(previous iteration)



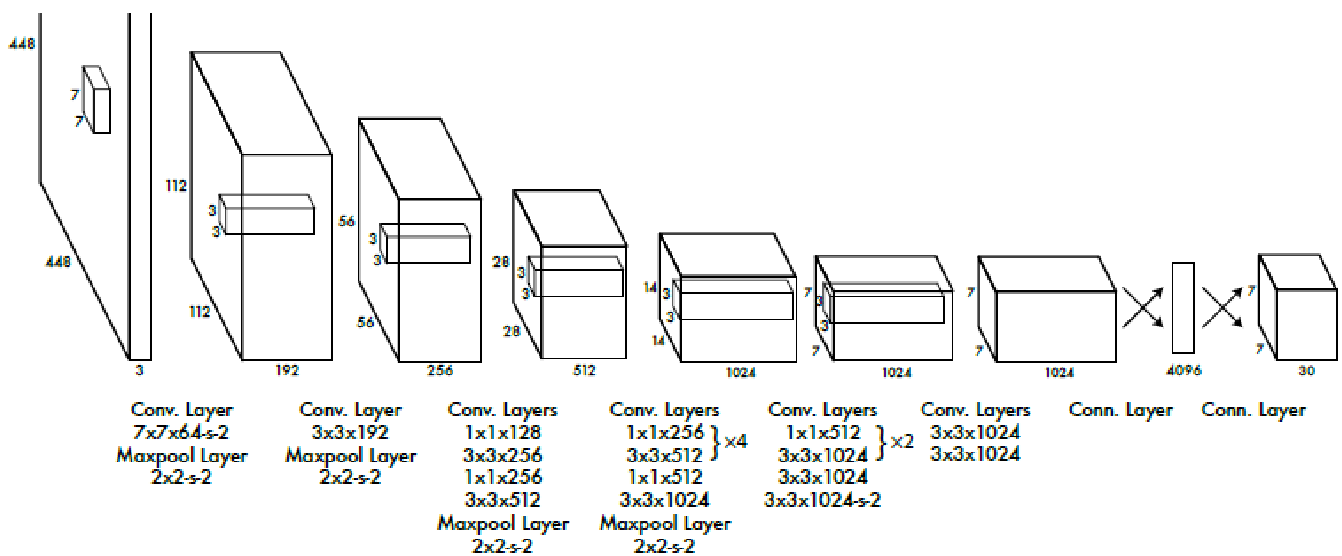
Deep LSTM Example

Many Application-Specific Architectures

- SSD



- YOLO



Summary

- **Module**
  - Captures functionality in a deep network
- **Key modules exist**
  - Sigmoid, tanh, convolution, etc.
- **Architecture**
  - Structural principle for modular composition
- **Must match architecture to application**

More Formally: Empirical Risk Minimization

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) + \lambda \Omega(\boldsymbol{\theta})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function

(loss function also called "cost function" denoted $J(\theta)$)

  - $\Omega(\boldsymbol{\theta})$ is a regularizer (penalizes certain values of $\boldsymbol{\theta}$)
- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

Any interesting cost function is complicated and non-convex

Solving the Risk (Cost) Minimization Problem
Gradient Descent – Basic Idea

Have some function $J(\theta_0, \theta_1)$

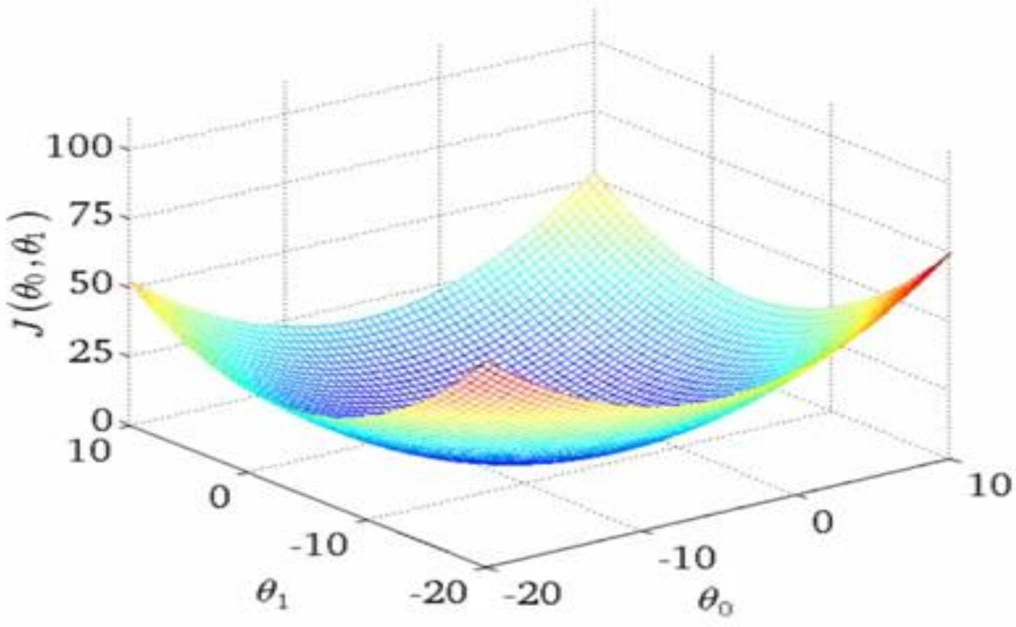
Want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

Outline:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$

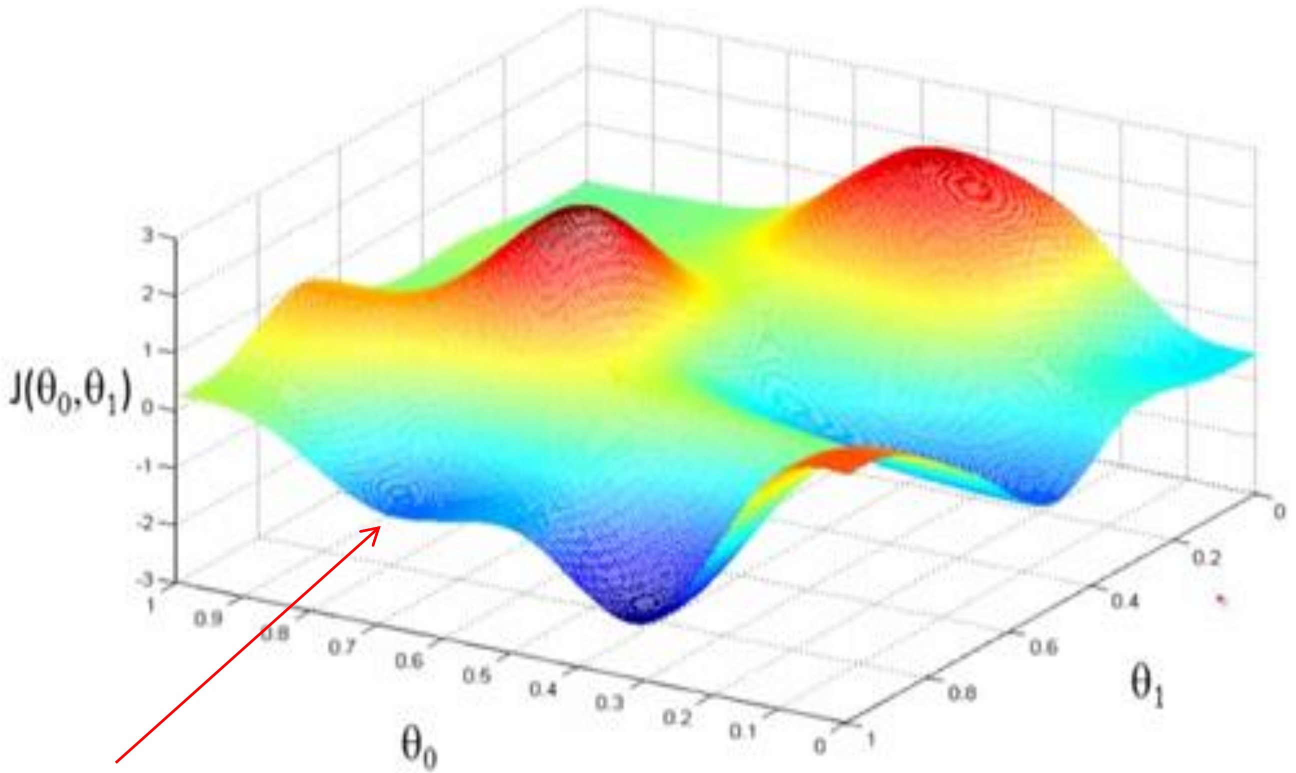
  until we hopefully end up at a minimum

Gradient Descent Intuition 1: Convex Cost Function



One of the many nice properties of convexity is that any local minimum is also a global minimum

Gradient Decent Intuition 2



Can get stuck here if unlucky/start at the wrong place

Unfortunately, any interesting cost function is likely non-convex

Solving the Optimization Problem
Gradient Descent for Linear Regression

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

}

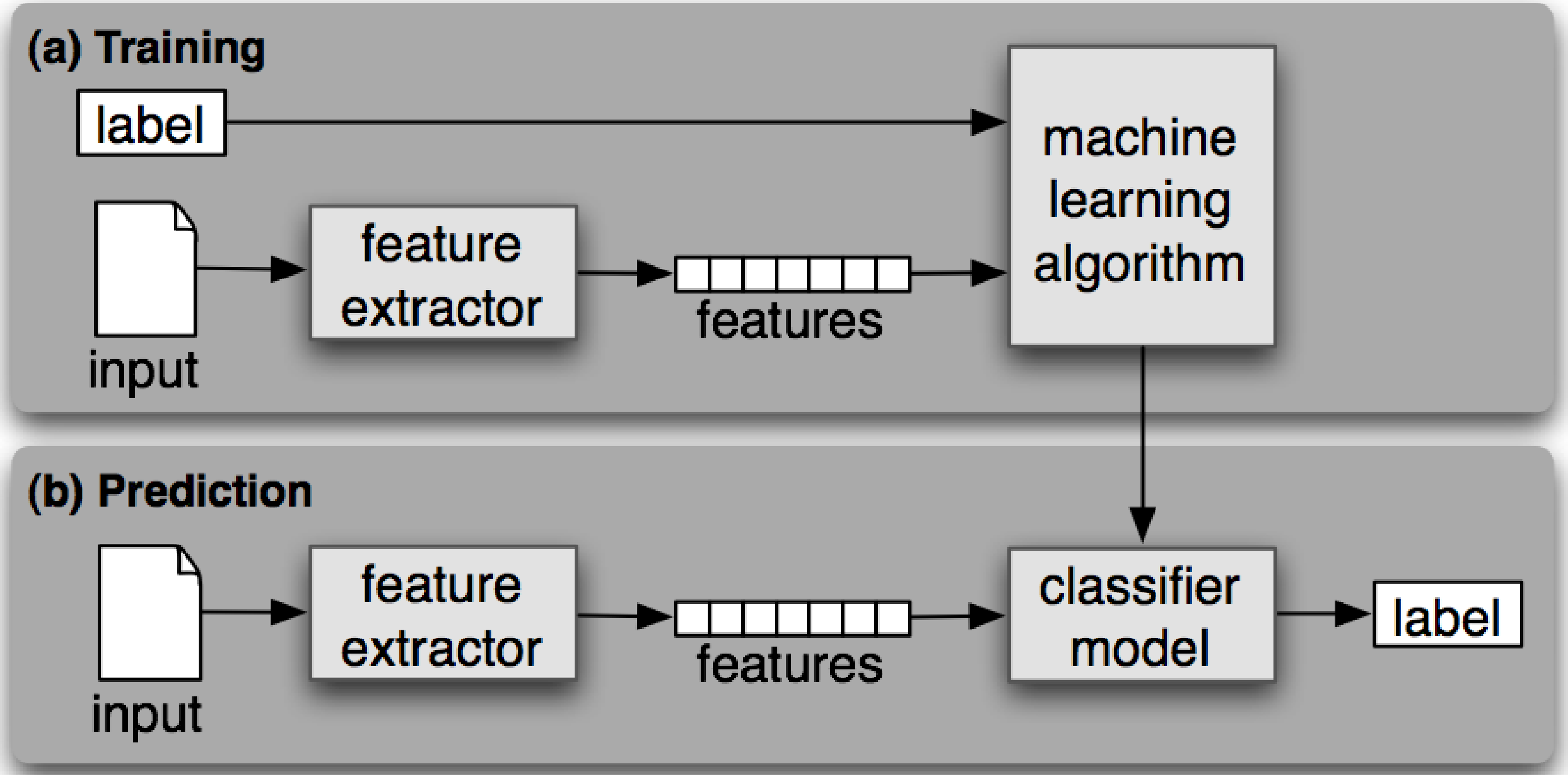
Linear Regression Model

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

The big breakthrough came from the Hinton lab at UToronto in the mid 80's where the back propagation algorithm was discovered (or perhaps re-discovered). "Backprop" is a simple way of computing the gradient of the loss function with respect to the model parameters θ

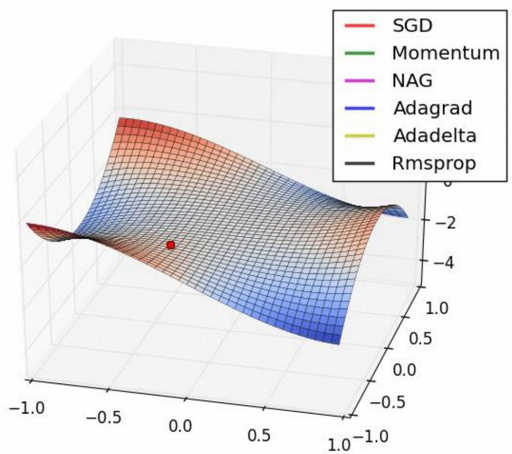
Summary: Supervised Learning Process

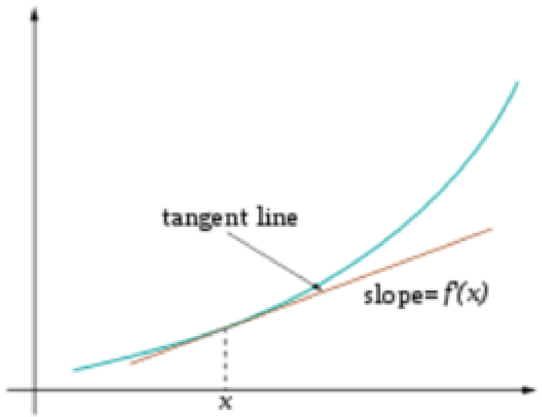


Training

Sample labeled data (**batch**) → **Forward** it through the network, get predictions → **Back-propagate** the errors → **Update** the network weights

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-4106a6702d39

Gradient Descent

objective/cost function $J(\theta)$

Review of backpropagation

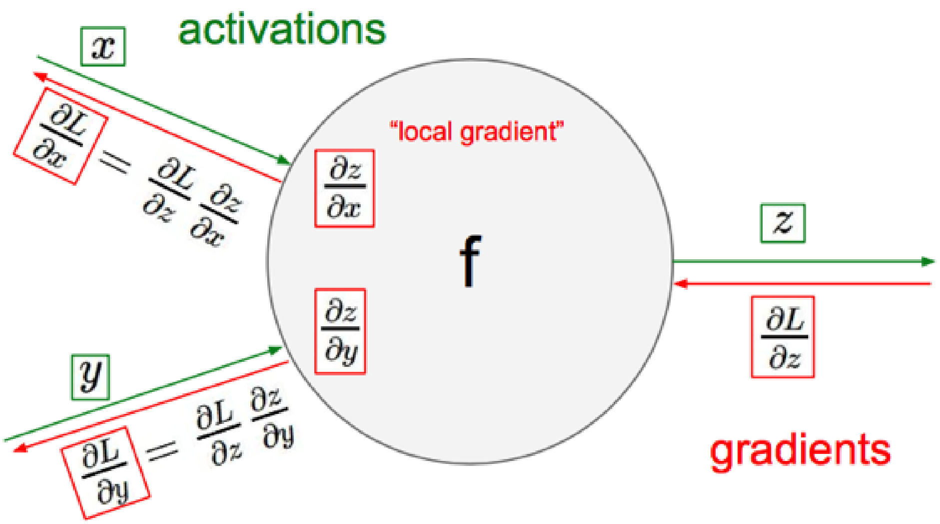
$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{d}{d\theta_j^{old}} J(\theta)$$

Update each element of θ

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

learning rate

Matrix notation for all parameters



Recursively apply **chain rule** though each node

One forward pass

Text (input) representation

TFIDF

Word embeddings

….

$\mathbf{W}$

0.2	-0.5	0.1
2.0	1.5	1.3
0.5	0.0	0.25
-0.3	2.0	0.0

$\boldsymbol{x_i}$

0.1
0.2
0.3

+

$\boldsymbol{b}$

1.0
3.0
0.025
0.0

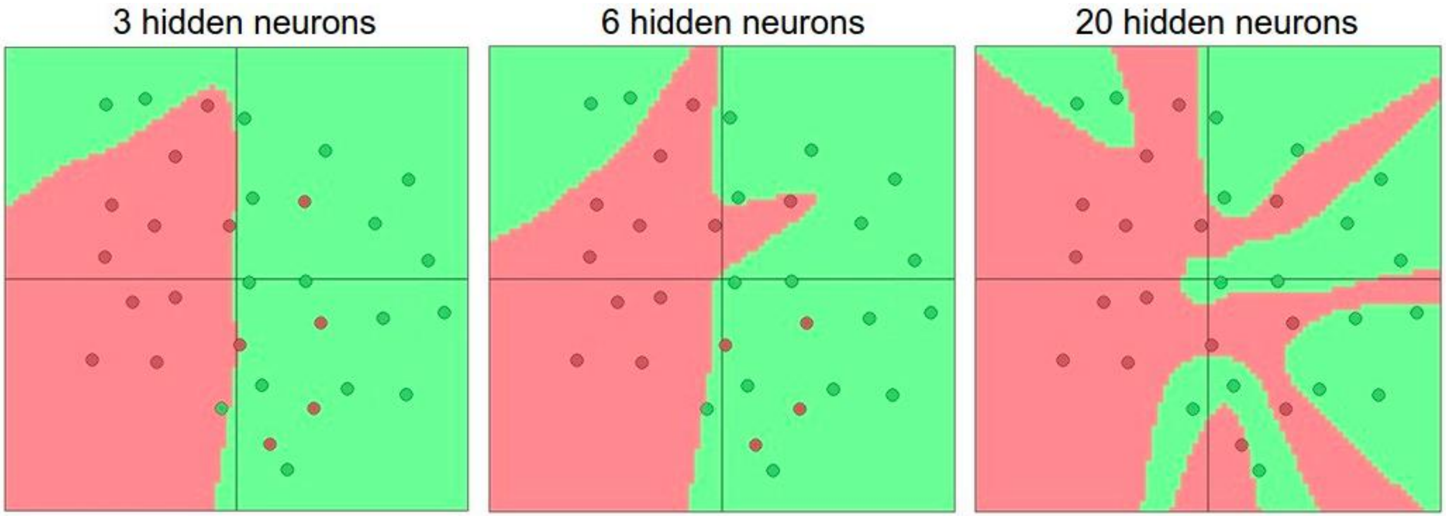
=

$\boldsymbol{\sigma(x_i; W, b)}$

0.95	very positive
3.89	$\boxed{\text{positive}}$
0.15	negative
0.37	very negative

Activation functions

Non-linearities needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function $W_1 W_2 x = Wx$

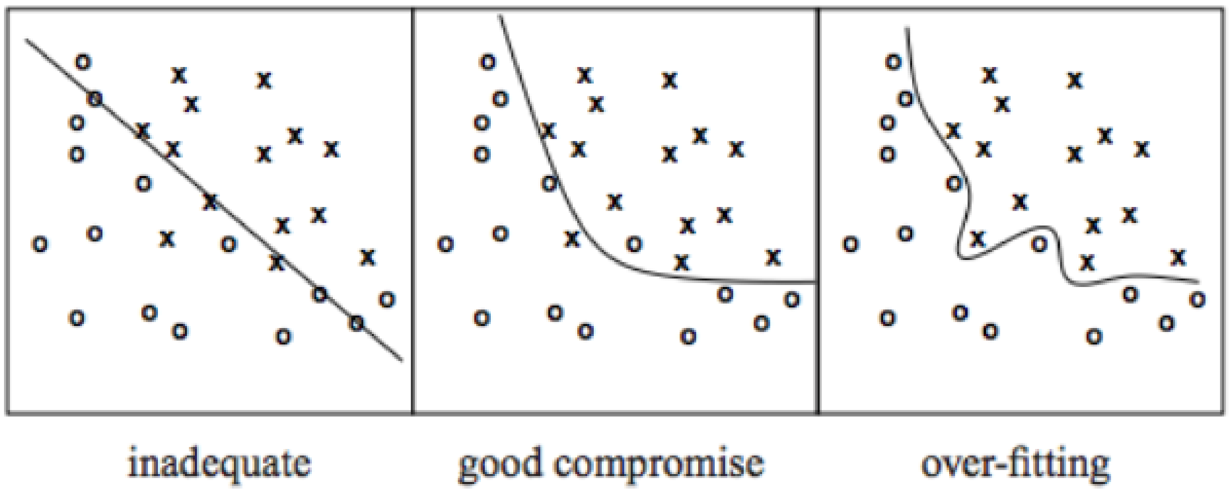


http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

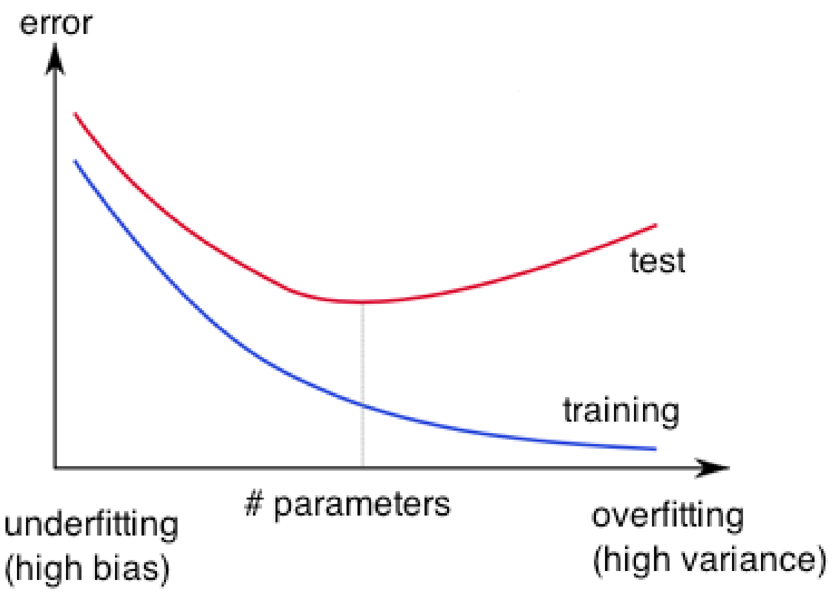
More layers and neurons can approximate more complex functions

Full list: https://en.wikipedia.org/wiki/Activation_function

Overfitting



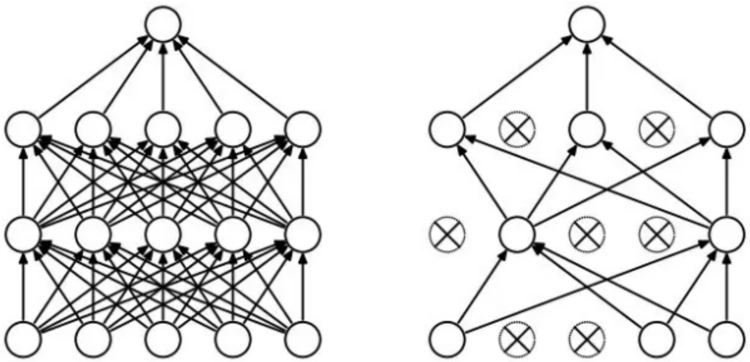
http://wiki.bethanycrane.com/overfitting-of-data



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

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

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

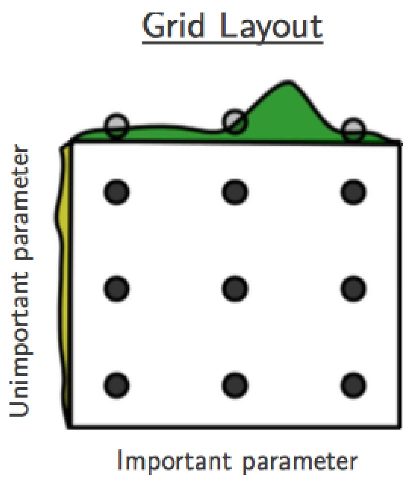
- Regularization term that penalizes big weights, added to the objective
- Weight decay value determines how dominant regularization is during gradient computation
- Big weight decay coefficient → big penalty for big weights

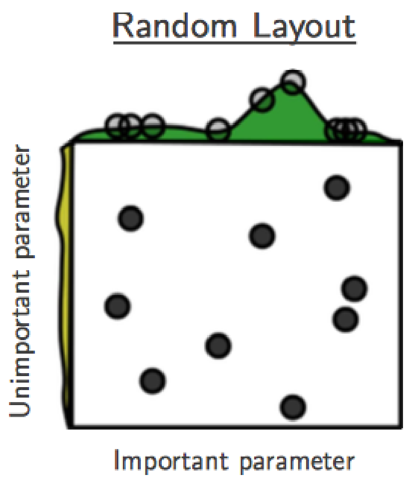
$$J_{reg}(\theta) = J(\theta) + \lambda \sum_k \theta_k^2$$

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

Tuning hyper-parameters





$g(x) \approx g(x) + h(y)$

g(x) shown in green
h(y) is shown in yellow

Bergstra, James, and Yoshua Bengio. "[Random search for hyper-parameter optimization.]" Journal of Machine Learning Research, Feb (2012)

"Grid and random search of 9 trials for optimizing function $g(x) \approx g(x) + h(y)$
With grid search, nine trials only test g(x) in three distinct places.
With random search, all nine trials explore distinct values of g. "

Both try configurations randomly and **blindly**
Next trial is independent to all the trials done before

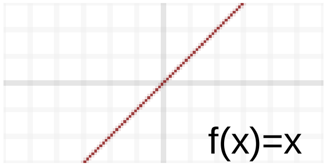
Bayesian optimization for hyper-parameter tuning:

Library available!

Make smarter choice for the next trial, minimize the number of trials

1. Collect the performance at several configurations
2. Make inference and decide what configuration to try next

Loss functions and output

	Classification	Regression
Training examples	R^n x {class_1, ..., class_n} (one-hot encoding)	R^n x R^m
Output Layer	Soft-max [map R^n to a probability distribution] $P(y=j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$	Linear (Identity) or Sigmoid 
Cost (loss) function	Cross-entropy	Mean Squared Error $J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$ Mean Absolute Error $J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left\lvert y^{(i)} - \hat{y}^{(i)}\right\rvert$

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right)\right]$$

List of loss functions