Linear Models, MLPs, Deep Neural Networks

Cognitive Robotics

Marco Ciccone

Dipartimento di Informatica Elettronica e Bioingegneria
Politecnico di Milano

Outline

- Intro
  - Recap on Linear Models
  - Gradient Descents
  - Loss functions
  - Activations
  - Neural Networks
- Difficulties in training
  - Model capacity
  - Vanishing gradient
- Deep Learning
  - Motivations
  - Theoretical foundations
  - Dropout
  - Batch Normalization

Refresh I - Supervised Learning

- We have an annotated dataset $D =< X, Y >$

$$y_i = f(x_i), \quad \forall\, x_i \in X,\ y_i \in Y$$

- Regression $x_i \in \mathbb{R}^N, y_i \in \mathbb{R}$
- Classification $x_i \in \mathbb{R}^N, y_i \in \{\Omega_1, \Omega_2, \ldots, \Omega_k\}$

GOAL:

- Find the mapping between X and Y $y_i \approx \hat{y}_i = h(x_i; \theta)$ *(parametric model)*
- We are looking for an approximator of the function **f(x)** that generated our data

Pay attention to overfitting! Keep your model simple!

We want to be able to predict **y** for unseen inputs, NOT learning by heart the dataset!

Linear Classification

Refresh II - Binary Logistic Regression

Use a **Linear model** to find the mapping between the input and the classes

Reminder on **Logistic Regression:**

- **Problem**

  $\mathbf{x}_i$ input vectors, $\mathbf{y}_i$ binary variable. From x predict y.

  Here, our model predicts: $p(Y = 1|X = x) = sigmoid(Wx + b).$

- **Loss or Cost function**

$$J(\theta) = -\sum_i y_i log(p(Y = 1|X = x_i)) + (1 - y_i) \log(p(Y = 0|X = x_i))$$

Refresh III - Multinomial Logistic Regression

Softmax Regression is Logistic Regression generalization to multiple classes.

- **Problem**

  $\mathbf{x}_i$ input vectors, $\mathbf{y}_i$ discrete variable between 0 and K-1.

  Here, our model predicts: $p(Y = y|X = x) = (softmax(xW + b))_y$

- **Loss or Cost function**

$$J(\theta) = -\frac{1}{N}\sum_i f(x_i;\theta)\log(y_i).$$

Softmax function

We are going to use this function very much during the course

$$softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

It takes a vector of arbitrary real-valued scores (in z) and squashes it to a vector of values between zero and one.

You can interpret $softmax(z)_i$ as the probability of the input z of belonging to the class i (outcome of a Categorical Distribution).

Learning is cast as Optimization

Life lesson

As engineers you're going to do only one thing in your life

Optimize a Loss function

How do we minimize the Loss?

We need to find its minimum, setting its derivative to zero and solving it w.r.t. **w**:

$$\frac{\partial J(\theta)}{\partial \mathbf{w}} = 0$$

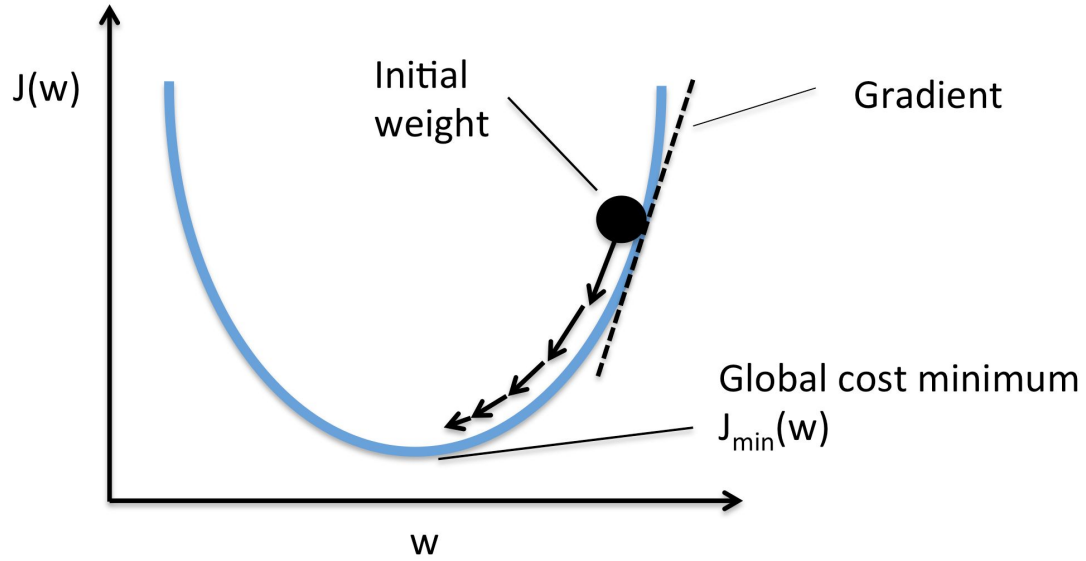
Life is hard: closed-form solutions are practically **never** available.

Use iterative techniques such as *Gradient Descent* or fancier algorithms:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial J(\theta)}{\partial \mathbf{w}}$$

Gradient Descent

- Gradient gives you the direction of the steepest ascent
- We want to find the parameters that minimize the Loss
- We step along the direction of the the *steepest descent* (negative gradient)



Gradient Descent variants

- **Batch Gradient Descent**

$$J(\theta) = -\frac{1}{N}\sum_i \log(p(Y = y_i|X = x_i))$$

(Use all the examples of the training set)

- **Stochastic Gradient Descent (SGD)**

$$\hat{J}_{SGD}(\theta) = -\log(p(Y = y_i|X = x_i))$$

(Use only one sample)

(Unbiased, but High Variance)

- **Minibatch Gradient Descent**

$$\hat{J}_{minibatch}(\theta) = -\frac{1}{n}\sum_i \log(p(Y = y_i|X = x_i))$$

(Use a subset of n samples)

(Good variance-computation tradeoff)

Gradient over entire dataset is impractical
Better to take quick, noisy steps!

Estimate gradient over a *mini-batch* of examples

Stochastic Gradient Descent (SGD)

Iterative algorithm that performs an update after each (subset of) example(s):

- Initialize the parameters $\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
- For **N** iterations
  - For each (subset of) training example $(x^{(t)}, y^{(t)})$
    - $\Delta = -\nabla_\theta J(f(x^t;\theta), y^{(t)}) - \lambda \nabla_\theta \Omega(\theta)$
    - $\theta \leftarrow \theta + \alpha\Delta$

Training epoch = Iterate **over all** examples

To apply this algorithm you need to choose:

- The loss function $J(f(x^t;\theta), y^{(t)})$
- The procedure to compute the parameter gradients $\nabla_\theta J(f(x^t;\theta), y^{(t)})$
- The regularizer term $\Omega(\theta)$

How do we choose the Loss function?

- Regression - *Mean Squared Error (MSE)*

$$J(\theta) = \frac{1}{N}\sum_i (y_i - f(x_i;\theta))^2.$$

- Classification - *Cross Entropy (xEntropy)*

$$J(\theta) = -\frac{1}{N}\sum_i f(x_i;\theta)\log(y_i).$$

Be creative!

The Loss depends on the task you want to solve, but it has one caveat:

Loss must be differentiable!

Refresh IV - Artificial Neuron

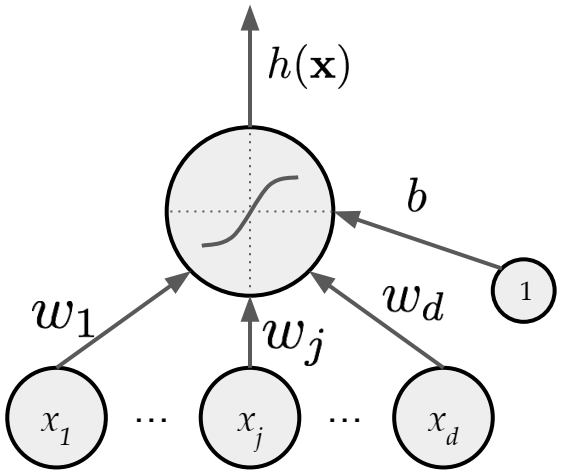
- Neuron pre-activation

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^T \mathbf{x}$$

- Neuron (output) activation

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

- $\mathbf{w}$ are the connection weights
- b is the neuron bias
- $g(\cdot)$ is the activation function



Refresh IV - Artificial Neuron

Images from Hugo Larochelle's DL Summer School Tutorial

It could do binary classification:

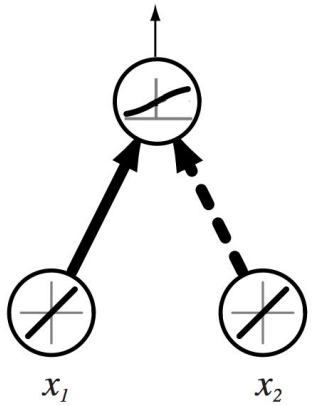
with *sigmoid*, can interpret neuron as estimating

$$p(Y = 1|X = x) = sigmoid(Wx + b).$$

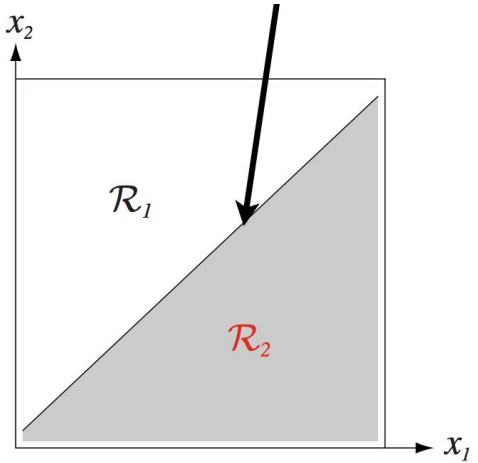
This is again **Logistic Regression**!

if greater than 0.5, predict class 1

otherwise, predict class 0

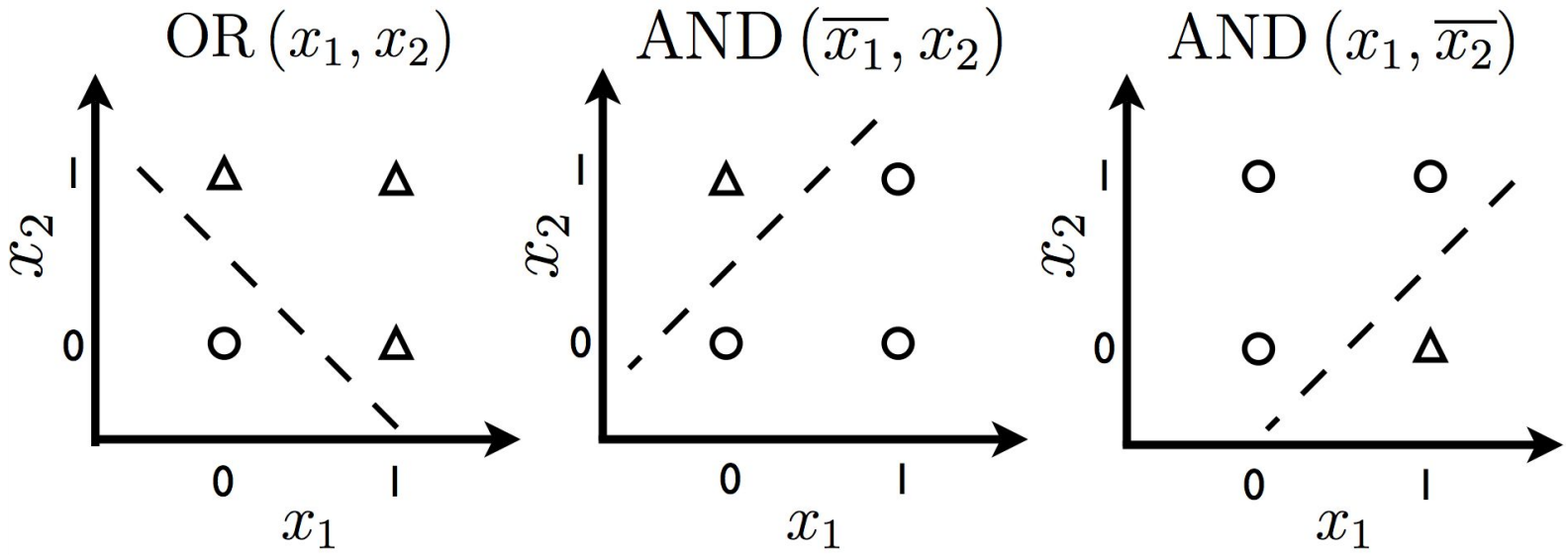


Decision boundary is Linear!



Refresh IV - Artificial Neuron

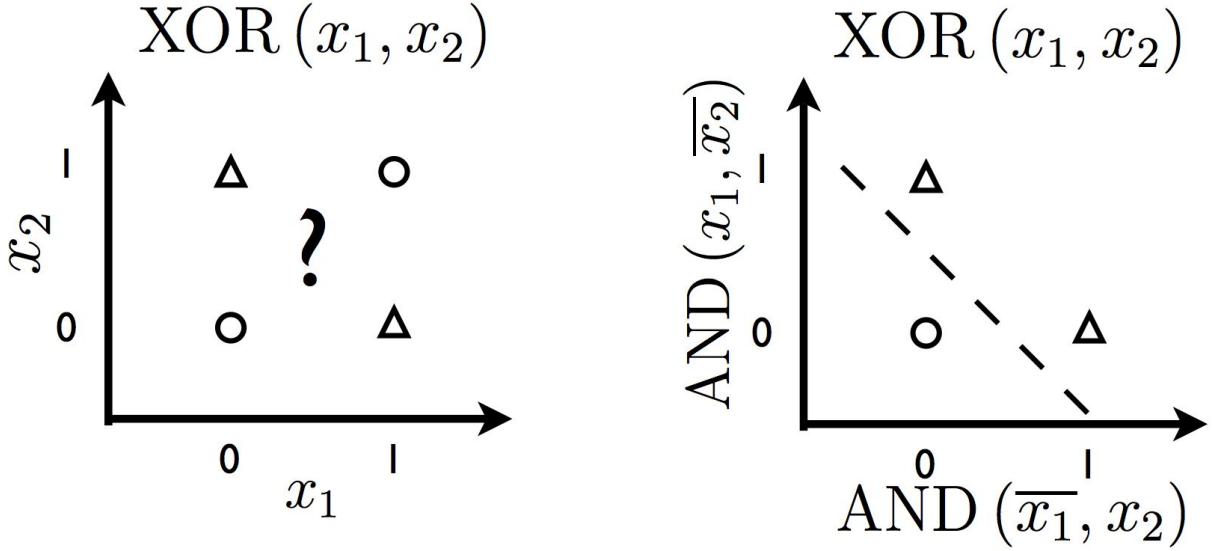
- Artificial Neuron can solve *linearly separable* problems…



Images from Hugo Larochelle's DL Summer School Tutorial

Refresh IV - Artificial Neuron

- But it **can't** solve nonlinearly separable problems…



- … unless the input is transformed in a better representation.

Linear models are not powerful enough!
We need nonlinear models:

Neural Networks

Neural Networks

Refresh V: Feedforward Neural Networks

Multilayer Perceptron - MLP - Fully-Connected

- Could have **L** hidden layers
  - pre-activation (for any k > 0)

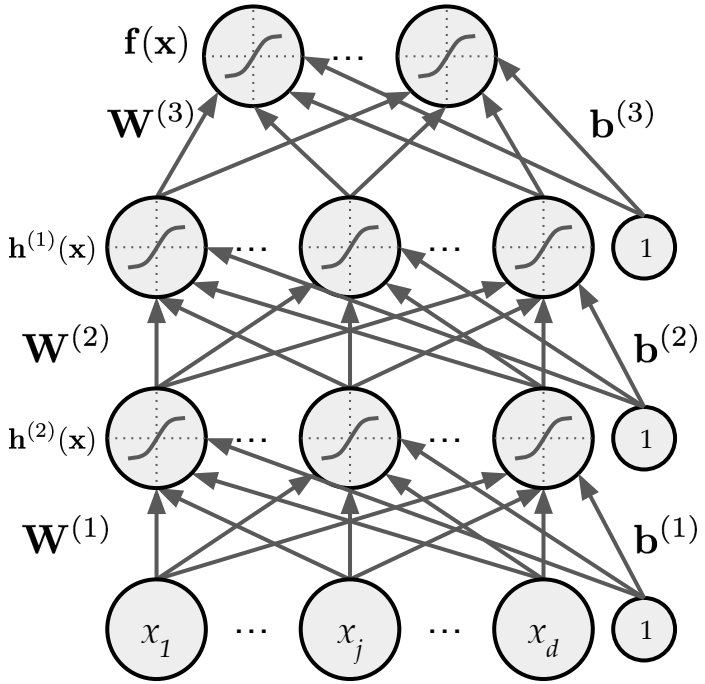
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}) + \mathbf{b}^{(k)}$$

  - hidden layer activation (k = 1 to L)

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

  - output activation (k = L + 1)

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Remember

The output of each layer of a NN is a (*nonlinear*) combination of its inputs

Refresh V - Chain rule, Backpropagation

Recall we want to compute the gradient of the loss w.r.t. the weights and update them using gradient descent.

Let **x** be a real number and two functions $f : \mathbb{R} \to \mathbb{R}$, $g : \mathbb{R} \to \mathbb{R}$

Now consider the composite function $z = f(g(x)) = f(y)$, where $y = g(x)$
Then the derivative of **f** w.r.t. **x** can be computed applying the ***chain rule***:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

Leibniz's notation

NN are complex composite functions

Backpropagation is a way of computing gradients of expressions through recursive application of chain rule.

Activations

Linear activation

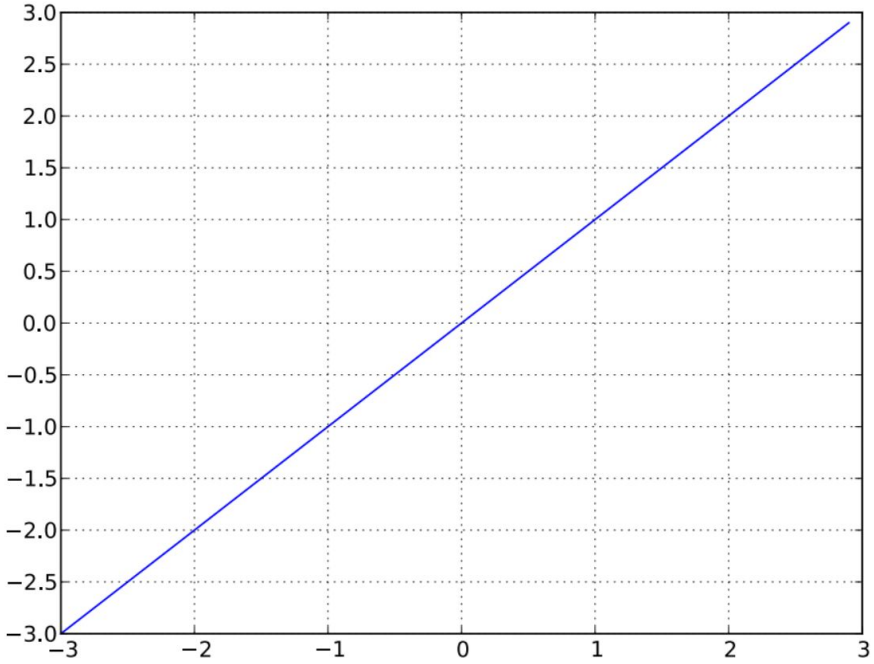
Linear activation function

$$g(a) = a$$

Partial derivative

$$g'(a) = 1$$

Not so interesting…



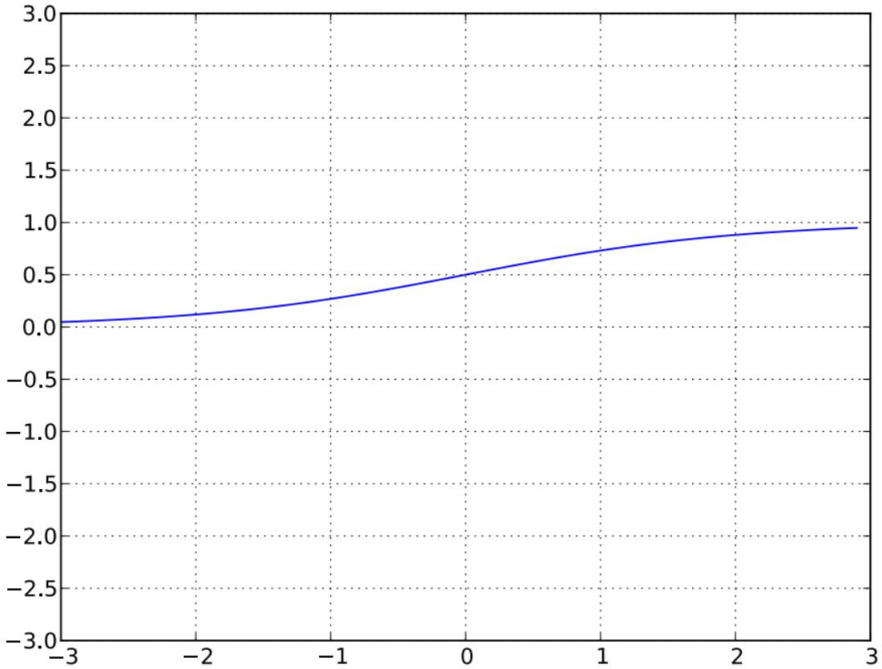
Sigmoid

Sigmoid activation function

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

Partial derivative

$$g'(a) = g(a)(1 - g(a))$$



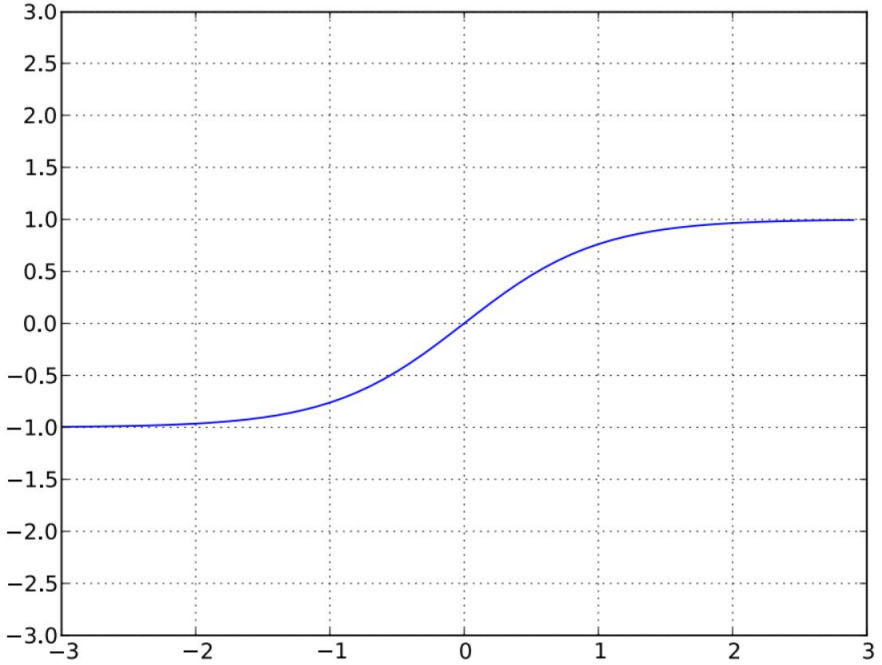
Hyperbolic Tangent

Tanh activation function

$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

Partial derivative

$$g'(a) = 1 - g(a)^2$$

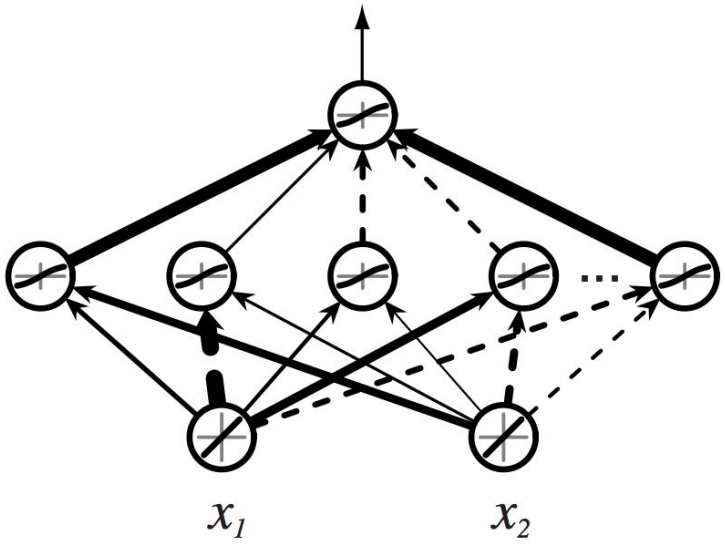


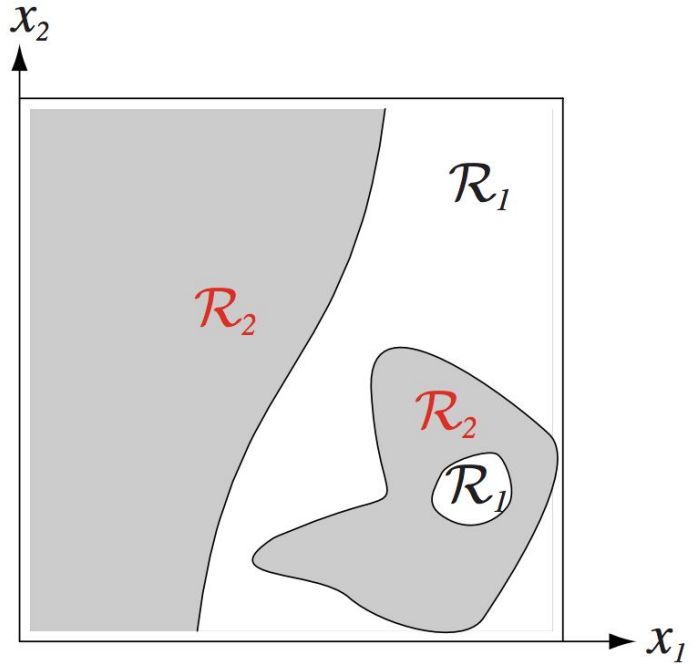
Model capacity

Universal approximation theorem (Hornik, 1991):

“ A single hidden layer feedforward neural network can approximate any measurable function to any desired degree of accuracy on a compact set ”

NNs as universal approximators





Images from Hugo Larochelle's DL Summer School Tutorial

NNs as universal approximators

What does it mean?

- Regardless of what function we are trying to learn, a large enough MLP will be able to represent it.
- The theorem holds for linear, sigmoid, tanh and many other hidden layer activation functions.

This is a good result, but it doesn't mean there is a learning algorithm that can find the necessary parameter values!

NNs as universal approximators

In the worse case, an *exponential number of hidden units* may be required.

In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may have to be *unfeasibly large* and may fail to learn and generalize correctly.

And Deep Learning save us all…

Deep Learning

- Deep learning is research on learning models with multilayer representations
  - Multilayer (feedforward) neural network
  - Multilayer graphical model (deep belief network, deep Boltzmann machine)

- Each layer corresponds to a ‘‘*distributed representation*’’
  - Units in layer are not mutually exclusive
    - each unit is a separate feature of the input
    - two units can be ‘‘active’’ at the same time
  - they do not correspond to a partitioning (clustering) of the inputs
    - in clustering, an input can only belong to a single cluster

Distributed Representation I

- It is possible to represent exponential number of regions with a linear number of parameters.
- It can learn a very complicated function (with many ups and downs) with a low number of examples (Not true in practice…)
- In non-distributed representations, the number of parameters are linear to the number of regions.
- Here, the number of regions potentially grow exponentially with the number of parameters and number of examples.

Deep Learning - Theoretical justification

A deep architecture can represent certain functions (exponentially) more compactly

Instead of growing our network wider, we grow it deeper

References

- "Learning Deep Architectures for AI", Yoshua Bengio, 2009
- "Exploring Strategies for Training Deep Neural Networks", Larochelle et Al, 2009
- "Shallow vs. Deep Sum-Product Networks", Delalleau & bengio, 2011
- "On the number of response regions of deep feed forward networks with piece-wise linear activations", Pascanu et Al, 2013

Distributed Representation II

- Features are individually meaningful. They remain meaningful despite the other features. There maybe some interactions but most features are learned independent of each other.
- We don’t need to see all configurations to make a meaningful statement.
- Non-mutually exclusive features create a combinatorially large set of distinguishable configurations.

Deep Learning - Theoretical justification II

- Using deep architectures expresses a useful prior over the space of functions the model learns.

- Encodes a very general belief that the function we want to learn should involve *composition of several simpler functions.*

- We can interpret the learning problem as discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation.

Deep Learning - Example

Boolean functions

- A Boolean circuit is a sort of feed-forward network where hidden units are logic gates (i.e. AND, OR or NOT functions of their arguments)

- Any Boolean function can be represented by a ‘‘single hidden layer’’ Boolean circuit
  - however, it might require an exponential number of hidden units

- It can be shown that there are Boolean functions which
  - require an exponential number of hidden units in the single layer case
  - require a polynomial number of hidden units if we can adapt the number of layers

If the function we are trying to learn has a particular characteristic obtained through composition of many operations,

then it is better to approximate these functions with a deep neural network.

Remark

A deeper network does not correspond to a higher capacity.

Deeper doesn’t mean we can represent more functions.

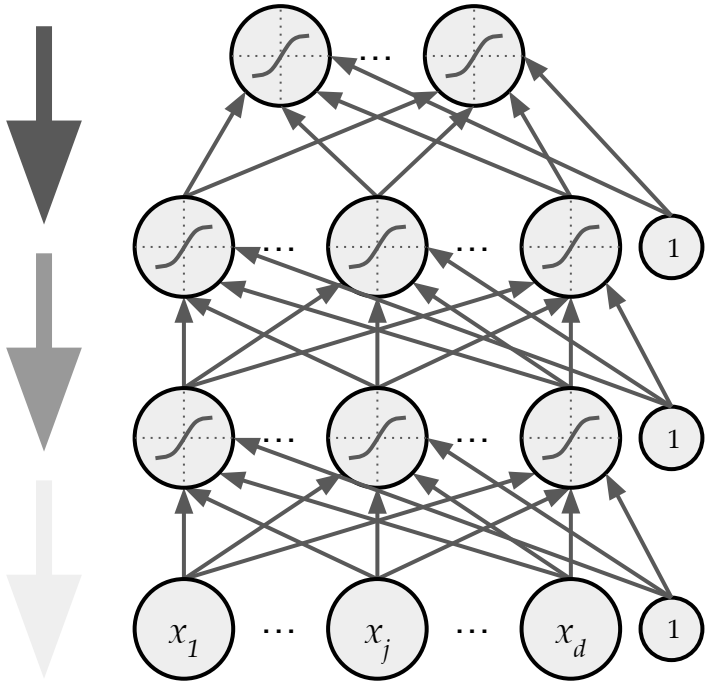
Training a Deep Neural Network is hard I

First hypothesis

Optimization is harder (underfitting)

- *Vanishing gradient problem*
- Saturated units block gradient propagation

This is a well known problem in *recurrent neural networks* (we'll see in a few lectures)

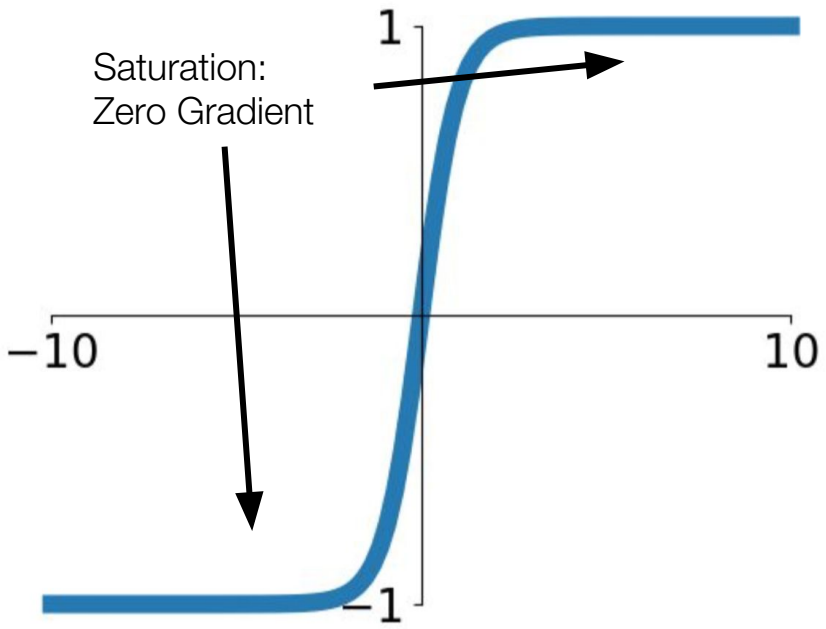


Vanishing gradient

Activation functions such as Sigmoid or Tanh, *saturates to 1*

=> Gradient is close to 0

=> No Gradient, No Learning



BackProp requires several gradient multiplications, so if the gradients are close to zero, it quickly vanishes.

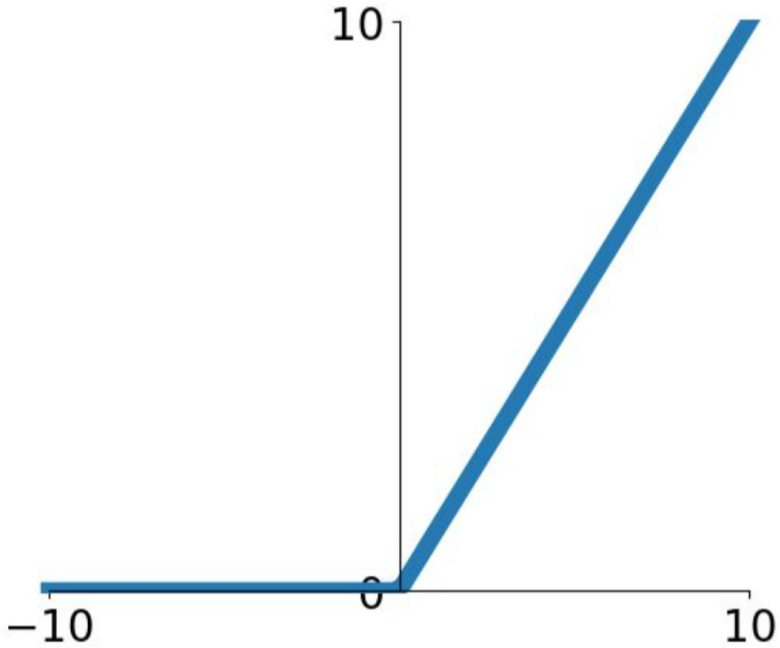
Rectified Linear Unit I

- ReLU activation function

$$g(a) = ReLU(a) = max(0, a)$$

Partial derivative

$$g'(a) = 1_{a>0}$$



Rectified Linear Unit II

Pros

- **Faster SGD Convergence:** compared to the sigmoid/tanh functions(6x faster). It is argued that this is due to its linear, non-saturating form (in +region).
- **Sparse activation:** For example, in a randomly initialized network, only about 50% of hidden units are activated (having a non-zero output).
- **Efficient gradient propagation:** No vanishing or exploding gradient problems.
- **Efficient computation:** Just thresholding at zero (No exponential funcions).
- **Scale-invariant:** $\max(0, ax) = a \max(0, x)$

Rectified Linear Unit II

Potential problems

- **Non-differentiable at zero:** however it is differentiable anywhere else, including points arbitrarily close to (but not equal to) zero.
- **Non-zero centered output**
- **Unbounded:** Could potentially blow up.
- **Dying Neurons:** ReLU neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. No gradients flow backward through the neuron, and so the neuron becomes stuck in a perpetually inactive state and "dies".

Large of dead numbers of neurons => decreasing the model capacity

(Typically arises when the learning rate is set too high)

Rectified Linear Unit III (variants)

- **Leaky ReLU**: attempt to fix the “dying ReLU” problem. Instead of being zero when x<0, a leaky ReLU will instead have a small negative slope (of 0.01, or so).

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ 0.01x & otherwise \end{cases}$$

- **pReLU**: The slope in the negative region (alpha) become a learned parameter.

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ \alpha x & otherwise \end{cases}$$

- **ELU**: try to make the mean activations closer to zero which speeds up learning.

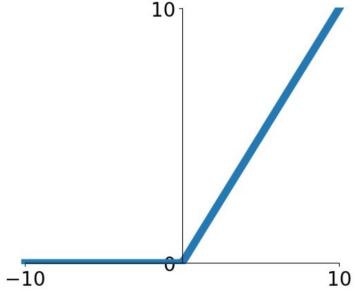
alpha tuned by hand

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ \alpha(e^x - 1) & otherwise \end{cases}$$

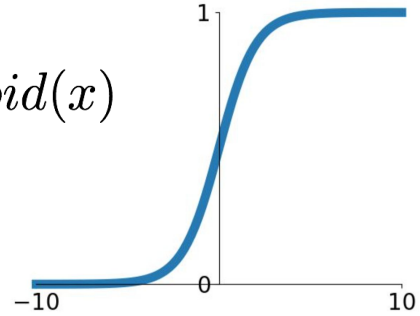
Activations Recap

ReLU

$$f(x) = max(0, x)$$

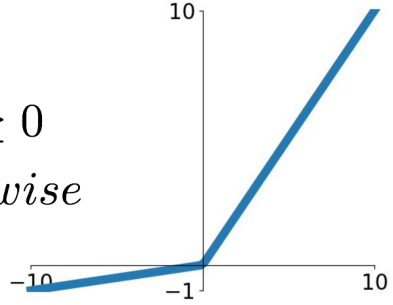


$$f(x) = sigmoid(x)$$

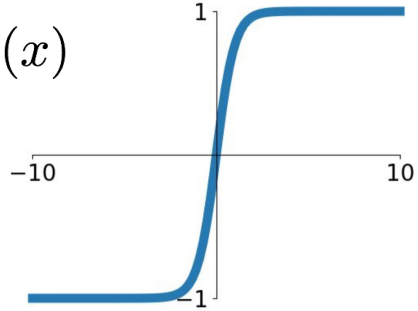


Leaky ReLU

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ 0.01x & otherwise \end{cases}$$

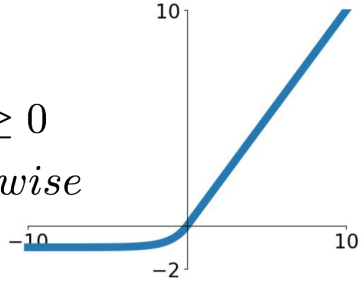


$$f(x) = tanh(x)$$



ELU

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ \alpha(e^x - 1) & otherwise \end{cases}$$



TLDR: What neuron type should I use?

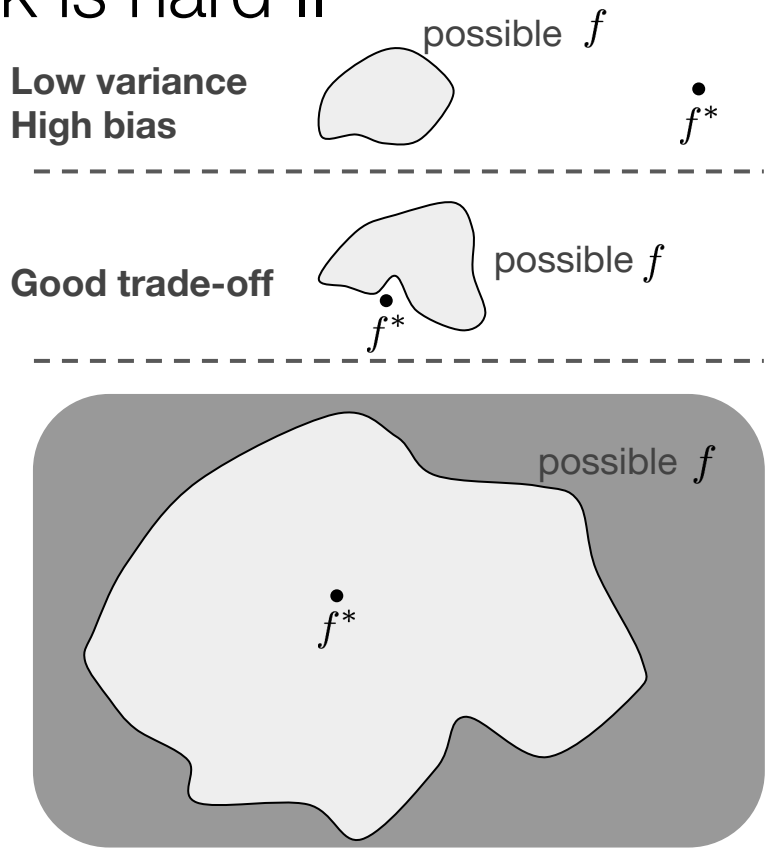
- **Use the ReLU nonlinearity**
  - Be careful with your learning rates and possibly monitor the fraction of “dead” units in a network.
  - If this concerns you, give Leaky ReLU a try.
- **Never use sigmoid.**
- You can try *tanh*, but expect it to work worse than *ReLU/LeakyReLU/ELU/Maxout.*

Training a Deep Neural Network is hard II

Second hypothesis (overfitting)

- we are exploring a space of complex functions
- deep nets usually have lots of parameters

Might be in a high variance / low bias situation



Training a Deep Neural Network is hard II

Depending on the problem, one or the other situation will tend to dominate

- If first hypothesis (underfitting) => **need to better optimize**
  - Better optimization methods (SGD + Momentum, RMSProp, Adam, Adadelta …)
  - Better parameters initialization
  - Better nonlinearities (ReLU …)
  - Batch Normalization
  - Use GPUs (if you increase your model then you need more power)
- If second hypothesis (overfitting) => **use better regularization**
  - Unsupervised learning (Not so much nowadays)
  - Stochastic «dropout» training

Reference:

Understanding the difficulty of training Deep Feedforward Neural Networks

Stochastic Regularization: Dropout

Problem of feature co-adaptation: a feature detector is only helpful in the context of several other specific feature detectors.

This is bad and it leads to overfitting!

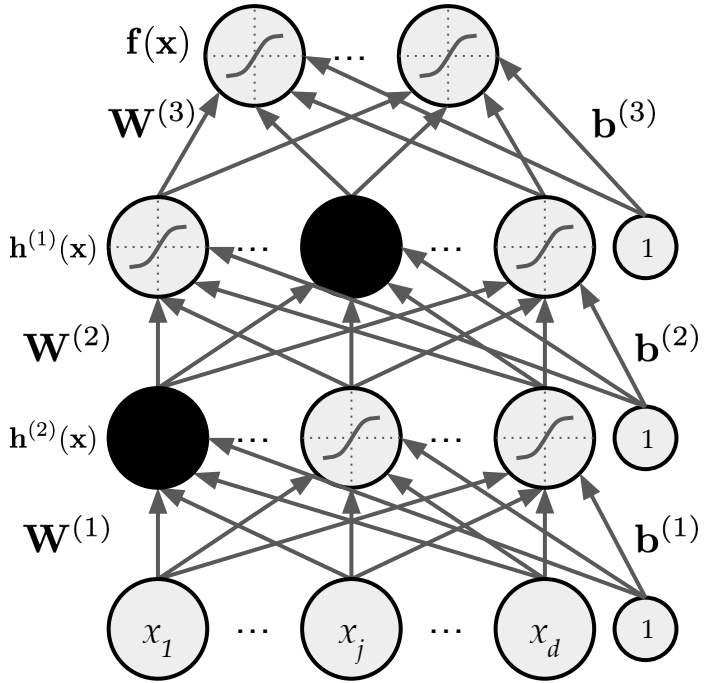
We want that each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate.

Stochastic Regularization: Dropout

Idea: Randomly turn off some neurons of the network

- Each hidden unit is set to zero with **p** probability
- Each layer can have a different $\mathbf{p_i}$ prob
- Usually set **p = 0.5** (It depends on the task)

By randomly omitting neurons we force them to learn an independent feature preventing hidden units to rely on other units (co-adaptation).



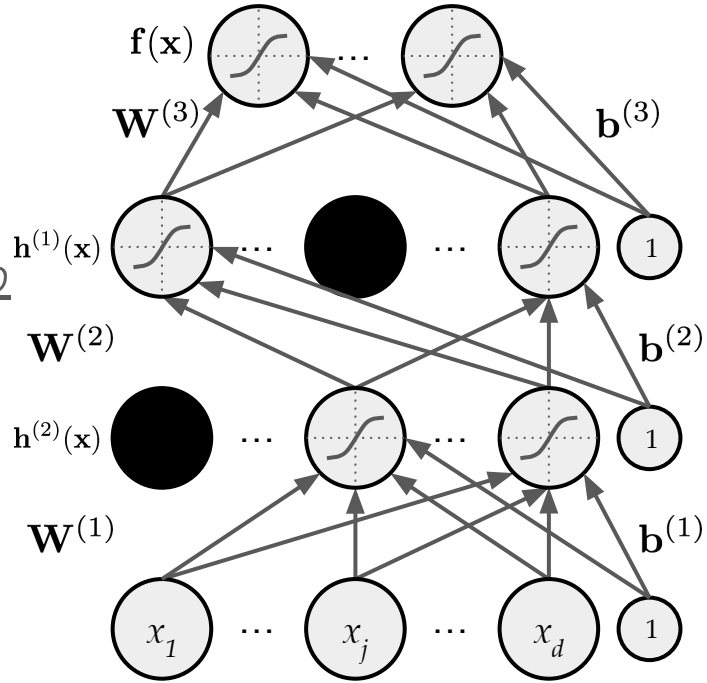
Stochastic Regularization: Dropout

- Use binary masks $\mathbf{m}^{(k)}$
- Masks are sampled from *Bernoulli* distributions with probability $p^{(k)}$, that means:

(1-p) proportion of the layer units are set to zero

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x} \odot \mathbf{m}^{(k)}) + \mathbf{b}^{(k)}$$

This is equivalent to multiplying the weights matrix by the binary vector to zero out entire rows.



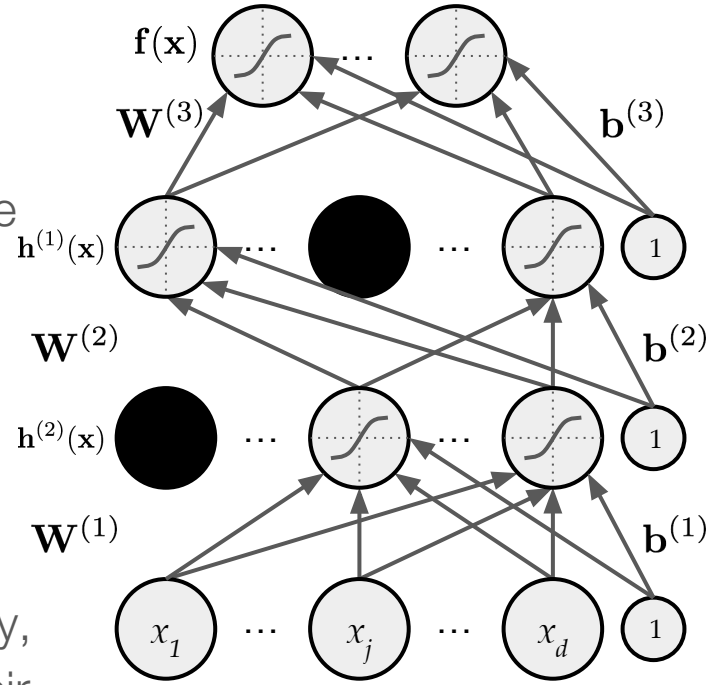
Stochastic Regularization: Dropout

Dropout can be seen as an “extreme” ensemble method.

We are averaging over different models, because we are removing different neurons at each minibatch

Idea: we train a number of *weaker classifiers*, and then at test time we use them by averaging the responses of all ensemble members.

Since each sub-network has been trained separately, it has learned different “*aspects*” of the data and their mistakes are different.



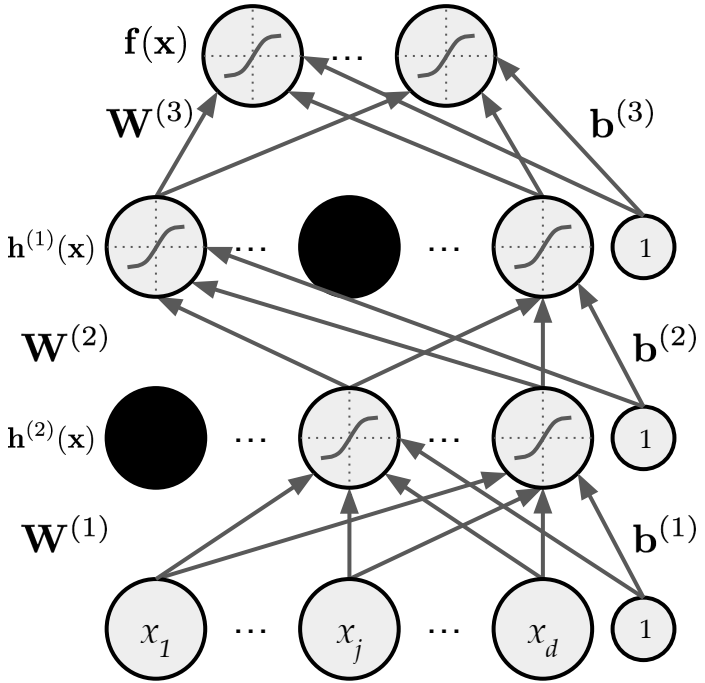
Stochastic Regularization: Dropout

Inference (Testing) time

Weight scaling (Approximated inference)

We remove the sampling mask and the weights are scaled by a factor of **p**, in order to maintain constant the output magnitude of the network.

This is equivalent to scale the input by 1/p at training time with no further scale at test time (much simpler)



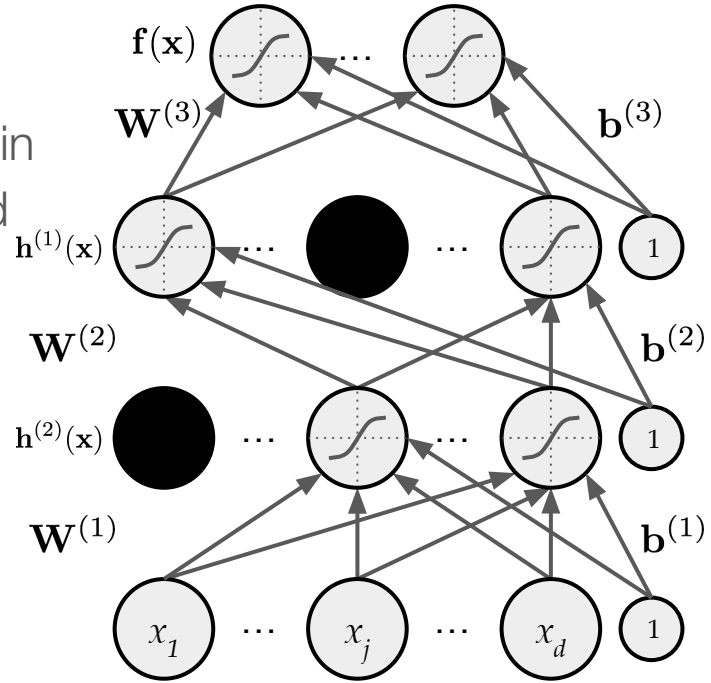
Stochastic Regularization: Dropout

Remark: We have a *stochastic model*

We are imposing a distribution over the weights, so in theory we should sample several model outputs and average them to get an estimate of the expected value

MC Dropout: sample several models at test time and average them.

- Expensive, but more accurate.
- Not so used unless you want to compute the confidence of the model (the variance).



Stochastic Regularization: Dropout

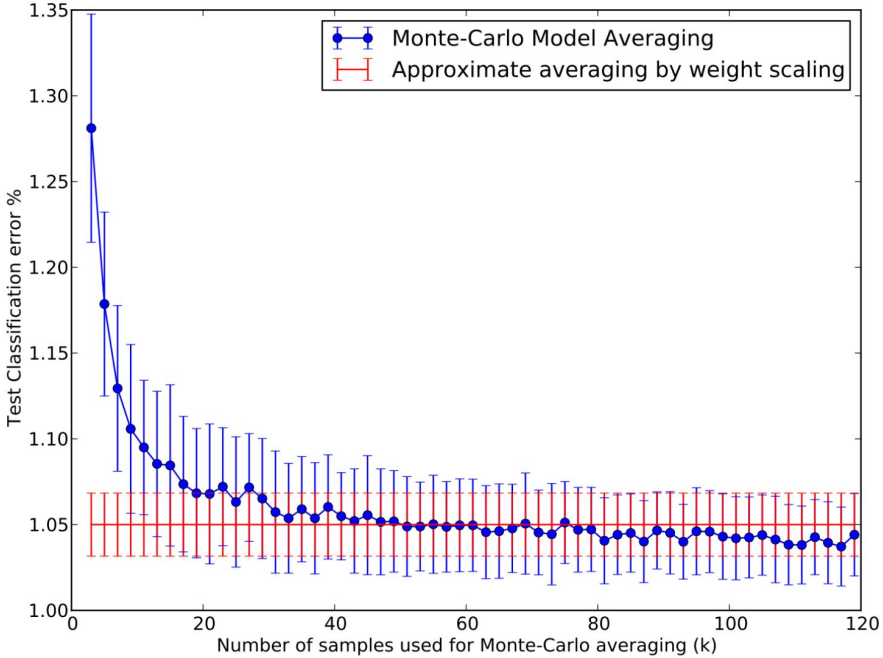


Figure 11: Monte-Carlo model averaging vs. weight scaling.

Stochastic Regularization: Dropout

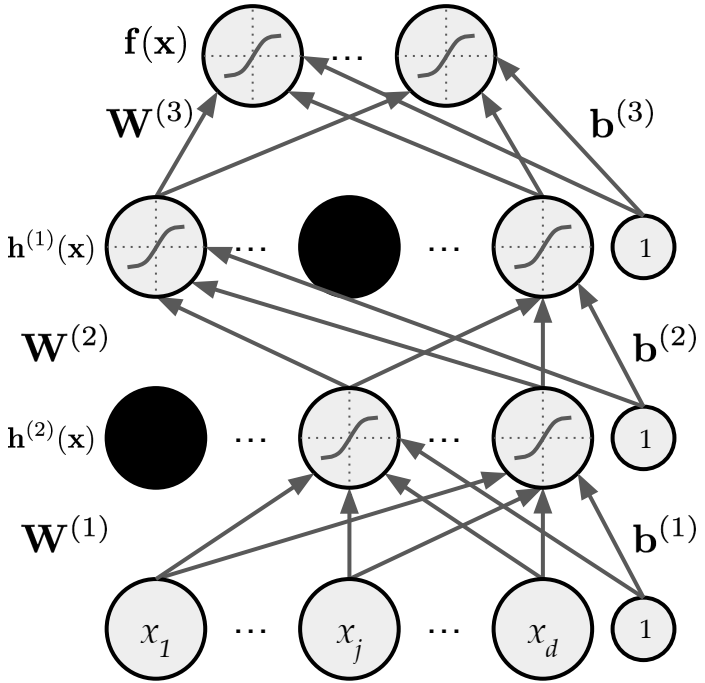
Reference

[Improving neural networks by preventing co-adaptation of feature detectors]

Hinton, Srivastava, Krizhevsky, Sutskever and Salakhutdinov, 2012.

[Dropout: A Simple Way to Prevent Neural Networks from Overfitting]

Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov



How to initialize the weights?

Zero or constant initialization

Don’t do it!

- If every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates.

In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

- By the way… You can initialize the **biases** to zero if you break the symmetry with the weights

Small random number initialization

Weights sampled from a Gaussian distribution with :

- zero mean
- 1e-2 standard deviation

Works ~*okay* for small networks, but problems with deeper networks!

Small random number initialization

As always the problem is in the gradient…

If the NN has very small weights also its gradients will be small!

This could greatly diminish the *"gradient signal"* flowing backward through a network, and could become a problem for deep networks.

Smarter initializations

- *“Xavier initialization”* Glorot et Al 2010

  A simple explanation from Andy's blog

But this mathematical derivation assumes linear activations, and ReLu nonlinearity breaks it. We can do better!

- *“He initialization”* He et Al 2015

Here, the mathematical derivation assumes ReLU activations.

Weights initialization

Proper parameters initialization in Neural Networks is an active area of research...

- Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010
- Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013
- Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014
- Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015
- Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015
- All you need is a good init, Mishkin and Matas, 2015

Internal Covariance Shift

Definition: Change in the input distribution to a learning system.

In the case of deep networks, the input to each layer is affected by parameters in all the input layers.

Remember: we have a highly nonlinear function, so even small changes to the network get amplified down the network.

This leads to change in the input distribution to internal layers of the deep network and is known as internal covariate shift.

Normalization

Normalizing the inputs will speed up training (Lecun et al. 1998)

It is well established that networks converge faster if the inputs have been whitened (ie zero mean, unit variances) and are uncorrelated so internal covariate shift leads to just the opposite.

Could normalization also be useful at the level of the hidden layers?

Yes, do Batch Normalization

Batch Normalization

Elegant technique proposed by Ioffe & Szegedy, 2015

- Based on the fact that normalization is a *simple differentiable operation.*
- Alleviates a lot of headaches with properly initializing neural networks by explicitly forcing the activations throughout the network to take on a *unit gaussian distribution* at the beginning of the training.
- Consists in putting the *BatchNorm* layer immediately after fully connected layers (or convolutional layers), and before nonlinearities.
- Can be interpreted as *doing preprocessing at every layer* of the network, but integrated into the network itself in a differentiable way.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Simple Linear operation!
So it can be back-propagated

Apply a linear transformation, to squash the range, so that the network can decide (learn) how much normalization needs.

Can also learn to recover the Identity mapping

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$
$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

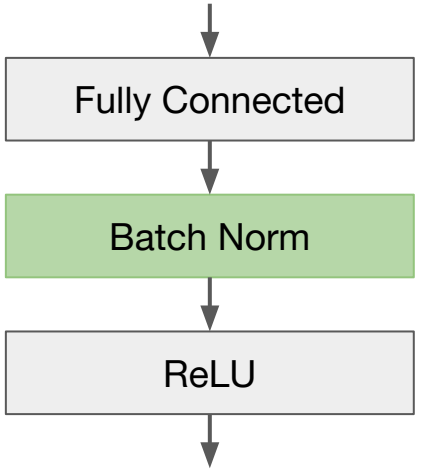
Batch Normalization

- Each unit's pre-activation is normalized (mean subtraction, stddev division)
- During training, mean and stddev is computed for each minibatch
- Backpropagation takes into account the normalization
- **Note:** *at test time*, the **global mean / stddev is used**

(The global statistics are estimated using running averages during the training)

Batch Normalization

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization
- slightly reduces the need for dropout



Acknowledgements

This slides are highly based on material taken from:

- Hugo Larochelle
- Andrej Karpathy
- Laurent Dinh