Machine Learning

Neural Network I - Lecture XI

Videos for This Lecture

- A Brief History of Neural Networks (Part 0)
- Perceptron (Part 1)
- Multi-Layer Perceptrons (Part 2)
- Obtaining the Gradients (Part 3)

Course Outline

Basic Concepts

- Parametric Method,
- Bayesian Learning and Nonparametrics Methods
- Clustering and Mixture of Gaussians

Classification Approaches

- Linear Discriminants
- Ensemble Methods and Boosting
- Randomized Trees, Forest

Reinforcement Learning

- Classical Reinforcement Learning
- Deep Reinforcement Learning

Deep Learning

- Foundations
- Optimization

Today's topics

A Brief History of Neural Networks

Perceptron

- Definition
- Loss functions
- Regularization
- Limits

Multi-Layer Perceptrons

- Definition
- Learning with hidden units

Obtaining the Gradients

- Naive analytical differentiation
- Numerical differentiation
- Backpropagation

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

- And a cool learning algorithm: “Perceptron Learning”
- Hardware implementation “Mark I Perceptron” for 20x20 pixel image analysis

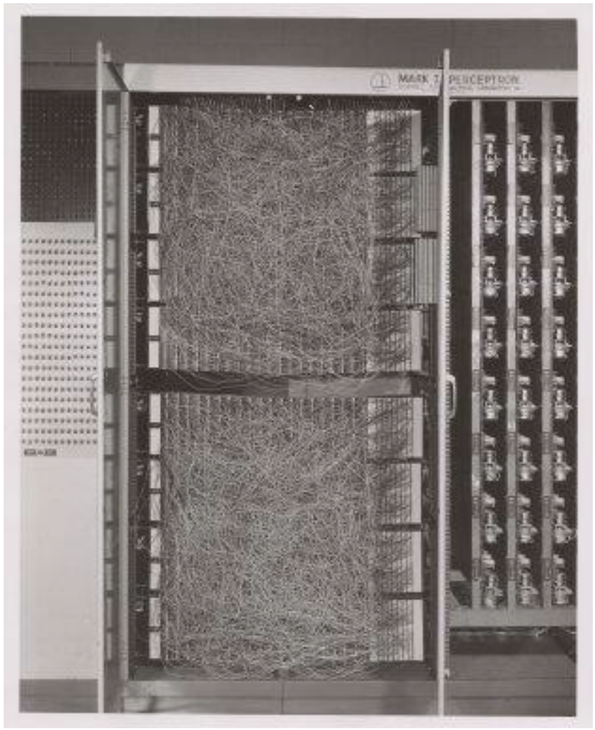


Image source: Wikipedia

A Brief History of
A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

- Showed that (single-layer) Perceptrons cannot solve all problems
- This was misunderstood by many that they were worthless

Image source: Wikipedia

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

- Some notable successes with multi-layer perceptrons.
- Backpropagation learning algorithm

Image source: Wikipedia

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

- Some notable successes with multi-layer perceptrons.
- Backpropagation learning algorithm
- But they are hard to train, tend to overfit, and have unintuitive parameters.
- So, the excitement fades again …

Image source: Wikipedia

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

- Notably Support Vector Machines
- Machine Learning becomes a discipline to its own.
- The general public and the press still love Neural Networks.

Image source: Wikipedia

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

2005+ Gradual progress

- Better understanding how to successfully train deep networks
- Availability of large datasets and powerful GPUs
- Still under radar for many disciplines apply ML

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

2005+ Gradual progress

2012 Breakthrough results

- ImageNet Large Scale Visual Recognition Challenge
- A ConvNet halves the error rate of dedicated vision Approaches.
- Deep Learning is widely adopted.

Part 1, Video NeuralNetworkI_p1

- Perceptron

Today's topics

A Brief History of Neural Networks

Perceptron

- Definition
- Loss functions
- Regularization
- Limits

Multi-Layer Perceptrons

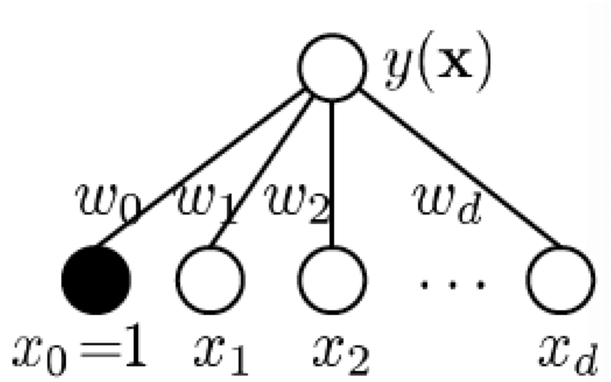
- Definition
- Learning with hidden units

Obtaining the Gradients

- Naive analytical differentiation
- Numerical differentiation
- Backpropagation

Perceptrons (Rosenblatt 1957)

Standard Perceptron



- output layer
- weights
- inputs layer

Input Layer

- Hand-designed features based on common sense

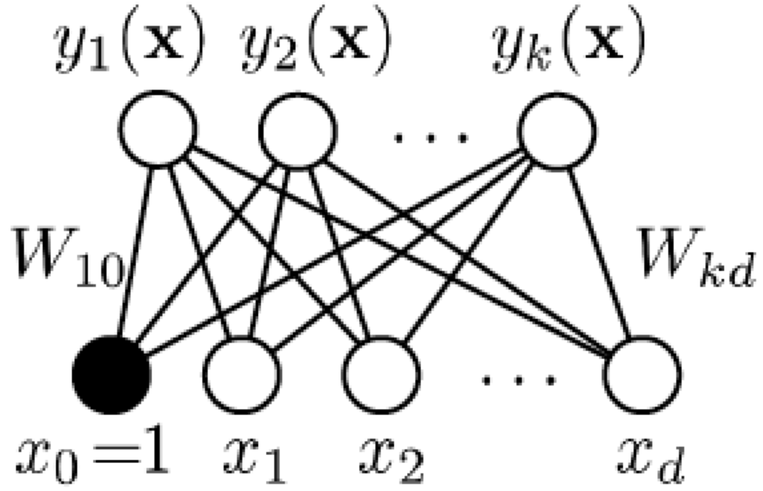
Outputs

- Linear output
  $y(\mathrm{x}) = \mathrm{w}^T \mathrm{x} + w_0$
- Logistic output
  $y(\mathrm{x}) = \sigma(\mathrm{w}^T \mathrm{x} + w_0)$

Learning = Determining the weights w

Extension: Multi-Class Networks

One output node per class



- output layer
- weights
- inputs layer

Outputs

- Linear output

$$y_k(\mathrm{x}) = \sum_{i=0}^{d} W_{ki} x_i$$

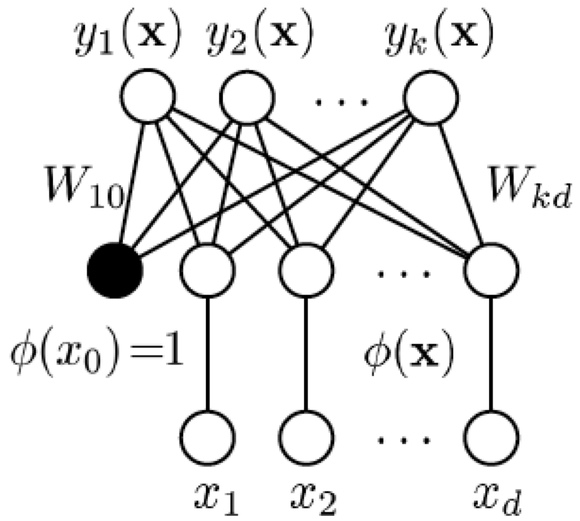
- Logistic output

$$y_k(\mathrm{x}) = \sigma(\sum_{i=0}^{d} W_{ki} x_i)$$

=> Can be used to do multidimensional linear regression or multi-class classification

Extension: Non-Linear Basis Functions

Straightforward generalisation



- output layer
- weights
- feature layer
- mapping
- inputs layer

Outputs

- Linear output

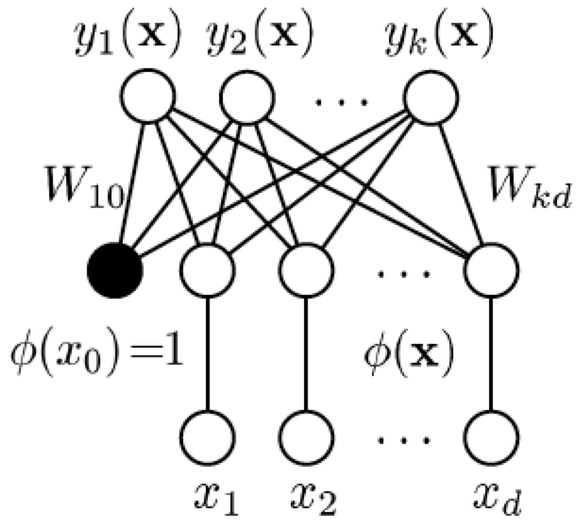
$$y_k(\mathrm{x}) = \sum_{i=0}^{d} W_{ki}\phi(x_i)$$

- Logistic output

$$y_k(\mathrm{x}) = \sigma(\sum_{i=0}^{d} W_{ki}\phi(x_i))$$

Extension: Non-Linear Basis Functions

Straightforward generalisation



- output layer
- weights
- feature Layer
- mapping
- inputs layer

Remarks

- Perceptrons are generalised linear discriminants!
- Everything we know about the latter can also be applied here
- Note: feature functions $\phi(\mathrm{x})$ are kept fixed, not learned!

Perceptron Learning

Very simple algorithm

Process the training case in some permutation

- If the output unit is correct, leave the weights along.
- If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
- If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.

This is guaranteed to converge to a correct solution if such a solution exists.

Perceptron Learning

Let' analyse this algorithm …

Process the training case in some permutation

- If the output unit is correct, leave the weights along.
- If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
- If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.

Translation

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)}$$

Perceptron Learning

Let' analyse this algorithm …

Process the training case in some permutation

- If the output unit is correct, leave the weights along.
- If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
- If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.

Translation

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta(y_k(\mathrm{x}_n; \mathrm{w}) - t_{kn})\phi_j(\mathrm{x}_n)$$

- This is the Delta rule a.k.a. LMS rule!

=> Perceptron Learning corresponds to 1st-order (stochastic) Gradient Descent of a quadratic error function!.

Loss Functions

We can now also apply other loss functions

- L2 loss — Least-squares regression

$$L(t, y(\mathrm{x})) = \sum_n (y(\mathrm{x}_n) - t_n)^2$$

- L1 loss — Median regression

$$L(t, y(\mathrm{x})) = \sum_n |y(\mathrm{x}_n) - t_n|$$

- Cross-entropy loss — Logistic regression

$$L(t, y(\mathrm{x})) = -\sum_n \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)$$

- Hinge loss — SVM classification

$$L(t, y(\mathrm{x})) = -\sum_n [1 - t_n y(\mathrm{x}_n)]_+$$

- Softmax loss — Multi-class probabilistic classification

$$L(t, y(\mathrm{x})) = -\sum_n \sum_k \{\mathbb{I}(t_n = k) \ln \frac{\exp(y_k(\mathrm{x}))}{\sum_j \exp(y_j(\mathrm{x}))}\}$$

Regularization

In addition, we can apply regularisers

- E.g., an L2 regularizer

$$E(\mathrm{w}) = \sum_n L(t_n, y(\mathrm{x}_n; \mathrm{w})) + \lambda ||\mathrm{w}||^2$$

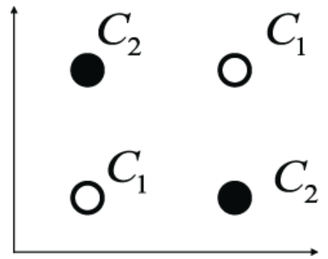
- This is known as weight decay in Neural Networks
- We can also apply other regularisers, e.g. L1 => sparsity
- Since Neural Networks often have many parameters, regularisation becomes very important in practice

Limitations of Perceptrons

What makes the task difficult

- Perceptrons with fixed, hand-coded input features can model any separable function perfectly …
- …given the right input features.
- For some tasks this requires an exponential number of input features.
  - E.g., by enumerating all possible binary input vectors as separate feature units (similar to a look-up table).
  - But this approach won't generalise to unseen test cases!

  => It is the feature design that solves the task!
- Once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn
  - Classical example: XOR function



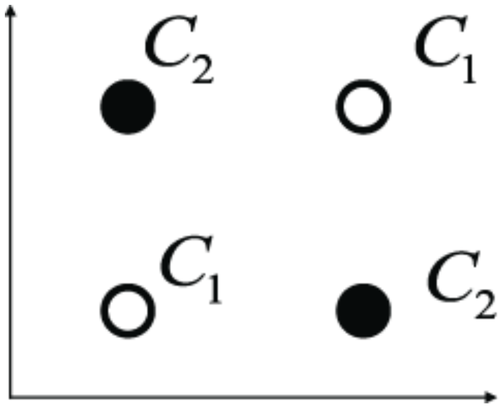
Wait…

Didn’t we just say that …

- Perceptrons corresponds to generalised linear discriminants
- And Perceptrons are very limited…
- Doesn’t this mean that what we have been doing so far in this lecture has the same problems???

Yes, this is the case.

- A linear classifier cannot solve certain problems (e.g. XOR)
- However, with a non-linear classifier based on the right kind of features, the problem becomes solvable



=> So far, we have solved such problems by hand designing good features and kernels

=> *Can we also learn such feature representations?*

Part 2, Video NeuralNetworkI_p2

- Multi-Layer Perceptrons

Today's topics

A Brief History of Neural Networks

Perceptron

- Definition
- Loss functions
- Regularization
- Limits

Multi-Layer Perceptrons

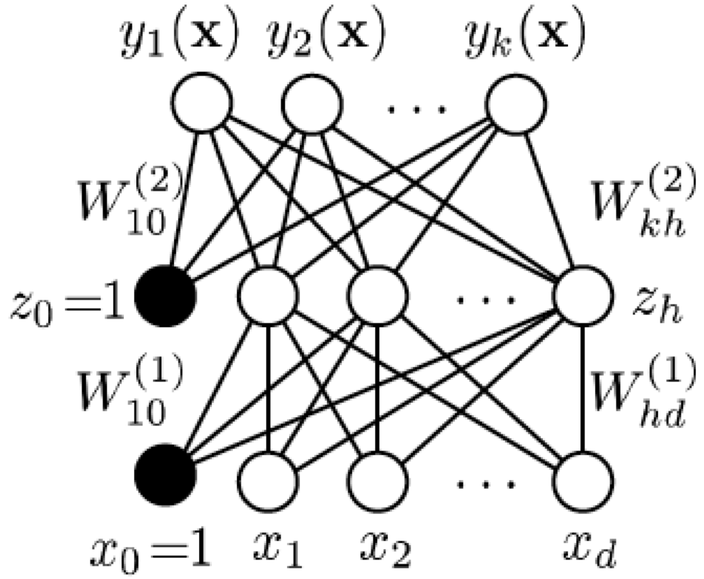
- Definition
- Learning with hidden units

Obtaining the Gradients

- Naive analytical differentiation
- Numerical differentiation
- Backpropagation

Multi-Layer Perceptron

Adding more layers



- output layer
- hidden layer
- mapping (learned)
- inputs layer

Output

$$y_k(\mathrm{x}) = g^{(2)}(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}(\sum_{i=0}^{d} W_{ij}^{(1)} x_j))$$

Multi-Layer Perceptron

$$y_k(\mathrm{x}) = g^{(2)}(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}(\sum_{i=0}^{d} W_{ij}^{(1)} x_j))$$

Activation functions $g^{(k)}$

- For example: $g^{(2)}(a) = \sigma(a),\ g^{(1)}(a) = a\ \ g(a) = tanh(a)\ \ g(a) = exp(-z^2/2)$

The hidden layer can have an arbitrary number of nodes

- There can also be multiple hidden layers.

Universal approximators

- A 2-layer network (1 hidden layer) can approximate any continuous function of a compact domain arbitrarily well! (assuming sufficient hidden nodes)

Learning with Hidden Units

Networks without hidden units are very limited in what they can learn

- More layers of linear units do not help => still linear
- Fixed output non-linearities are not enough.

We need multiple layers of adaptive non-linear hidden units. But how can we train such nets?

- Need an efficient way of adapting all weights, not just the last layer
- Learning the weights to the hidden units = learning features
- This is difficult, because nobody tells us what the hidden units should do.

=> Main challenge in deep learning.

Learning with Hidden Units

How can we train multi-layer networks efficiently?

- Need an efficient way of adapting all weights, not just the last layer.

Idea: Gradient Descent

- Set up an error function $E(\mathrm{W}) = \sum_n L(t_n, y(\mathrm{x}_n; \mathrm{W})) + \lambda\Omega(\mathrm{W})$
- with a loss L(.) and a regularise $\Omega(.)$
- E.g., $L(t, y(\mathrm{x}; \mathrm{W})) = \sum_n (y(\mathrm{x}_n; \mathrm{W}) - t_n)^2$ L2-loss

  $\Omega(\mathrm{W}) = ||\mathrm{W}||_F^2$ L2-regulariser

=> Update each weight $W_{ij}^{(k)}$ in the direction of the gradient $\frac{\partial E(\mathrm{W})}{\partial W_{ij}^{(k)}}$

Gradient Descent

Two main steps

1. Computing the gradient for each weight
2. Adjusting the weights in the direction of the gradient

Part 3, Video NeuralNetworkI_p3

- Obtaining the Gradients

Today's topics

A Brief History of Neural Networks

Perceptron

- Definition
- Loss functions
- Regularization
- Limits

Multi-Layer Perceptrons

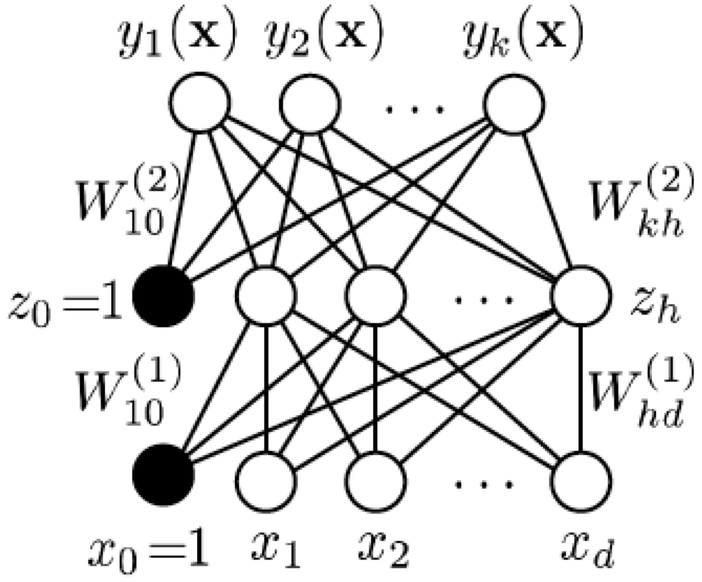
- Definition
- Learning with hidden units

Obtaining the Gradients

- Naive analytical differentiation
- Numerical differentiation
- Backpropagation

Obtaining the Gradients

Approach 1: Naive Analytical Differentiation



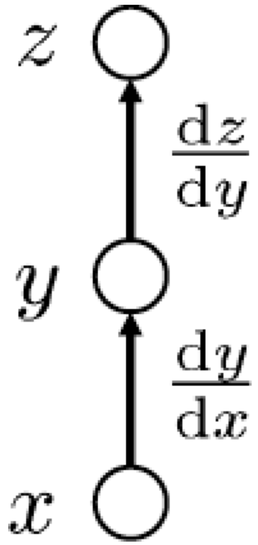
$$\frac{\partial E(\mathrm{W})}{\partial W_{10}^{(2)}} \cdots \frac{\partial E(\mathrm{W})}{\partial W_{kh}^{(2)}}$$

$$\frac{\partial E(\mathrm{W})}{\partial W_{10}^{(1)}} \cdots \frac{\partial E(\mathrm{W})}{\partial W_{hd}^{(1)}}$$

- Compute gradient for each variable analytically.
- What is the problem when doing this

Excursion: Chain Rule of Differentiation

One-dimensional case: Scalar function



$$\Delta z = \frac{dz}{dy}\Delta y$$

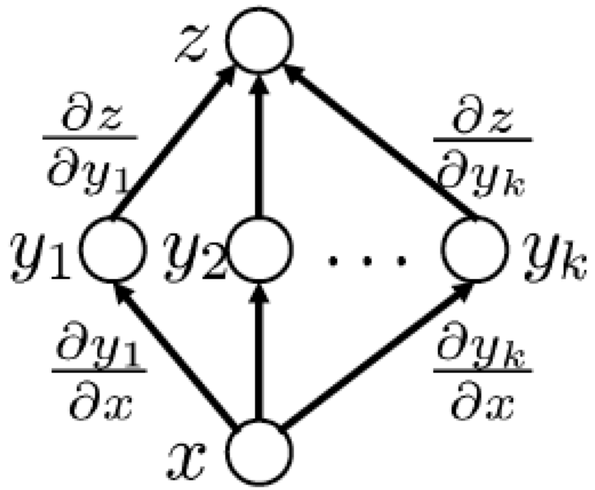
$$\Delta y = \frac{dy}{dx}\Delta x$$

$$\Delta z = \frac{dz}{dy}\frac{dy}{dx}\Delta x$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Excursion: Chain Rule of Differentiation

Multi-dimensional case: Total derivative



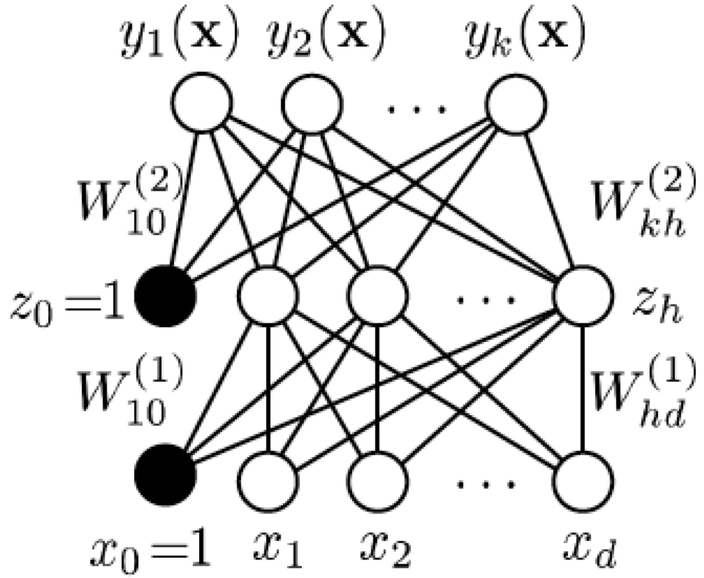
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \ldots$$

$$= \sum_{i=1}^{k} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

=> Need to sum over all paths that lead to the target variable x

Obtaining the Gradients

Approach 1: Naive Analytical Differentiation



$$\frac{\partial E(\mathrm{W})}{\partial W_{10}^{(2)}} \cdots \frac{\partial E(\mathrm{W})}{\partial W_{kh}^{(2)}}$$

$$\frac{\partial E(\mathrm{W})}{\partial W_{10}^{(1)}} \cdots \frac{\partial E(\mathrm{W})}{\partial W_{hd}^{(1)}}$$

- Compute gradient for each variable analytically.
- What is the problem when doing this

=> With increasing depth, there will be exponentially many paths!

=> Infeasible to compute this way.

Today's topics

A Brief History of Neural Networks

Perceptron

- Definition
- Loss functions
- Regularization
- Limits

Multi-Layer Perceptrons

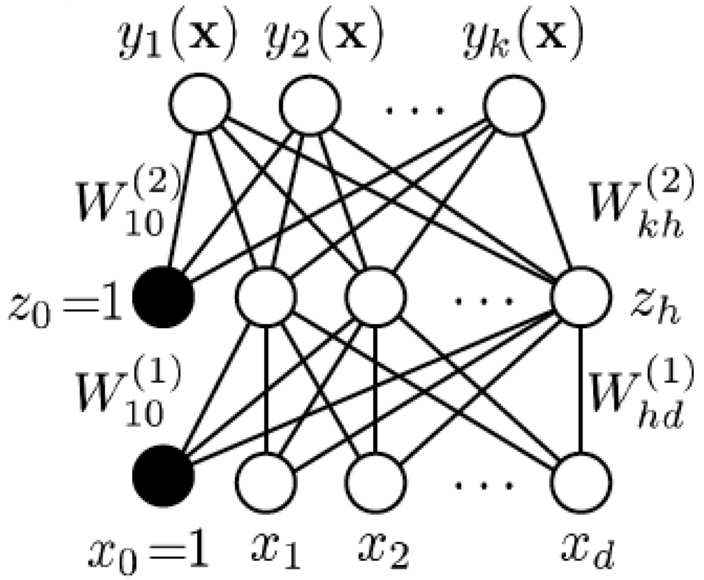
- Definition
- Learning with hidden units

Obtaining the Gradients

- Naive analytical differentiation
- Numerical differentiation
- Backpropagation

Obtaining the Gradients

Approach 2: Numerical Differentiation



- Given the current state $W^{(\tau)}$, can we evaluate $E(W^{(\tau)})$
- Idea: Make small changes to $W^{(\tau)}$ and accept those that improve $E(W^{(\tau)})$

=> Horribly inefficient! Need several forward passes for each weight.
Each forward pass is one run over the entire dataset!

Today's topics

A Brief History of Neural Networks

Perceptron

- Definition
- Loss functions
- Regularization
- Limits

Multi-Layer Perceptrons

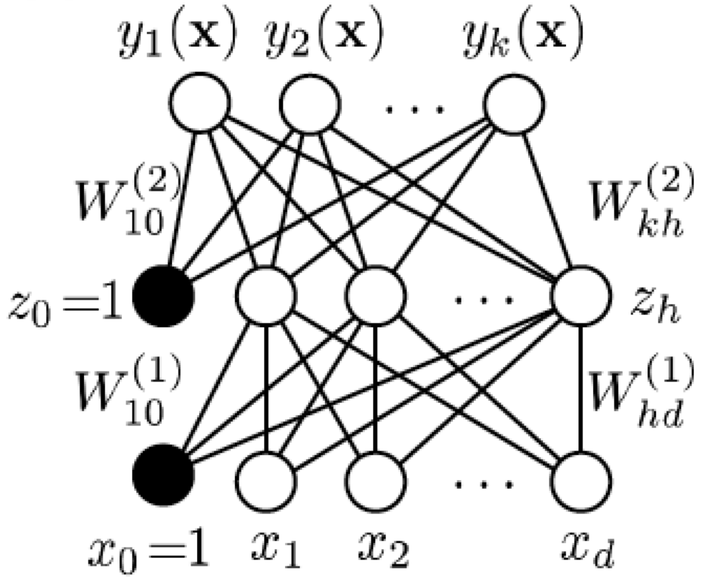
- Definition
- Learning with hidden units

Obtaining the Gradients

- Naive analytical differentiation
- Numerical differentiation
- Backpropagation

Obtaining the Gradients

Approach 3: Incremental Analytical Differentiation



$$\frac{\partial E(W)}{\partial y_j} \rightarrow \frac{\partial E(W)}{\partial W_{ij}^{(2)}}$$

$$\frac{\partial E(W)}{\partial z_j} \rightarrow \frac{\partial E(W)}{\partial W_{ij}^{(1)}}$$

$$\frac{\partial E(W)}{\partial x_i}$$

- Idea: Compute the gradients layer by layer.
- Each layer below builds upon the results of the layer above.

=> The gradient is propagated backwards through the layers.

=> Backpropagation algorithm

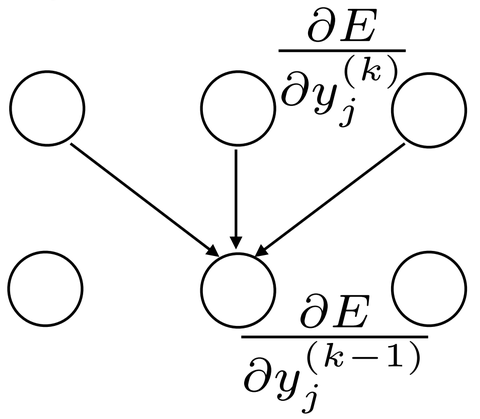
Backpropagation Algorithm

Core steps

- Convert the discrepancy between each output and its target value into an error derivate
- Compute error derivatives in each hidden layer from error derivatives in the layer above
- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights

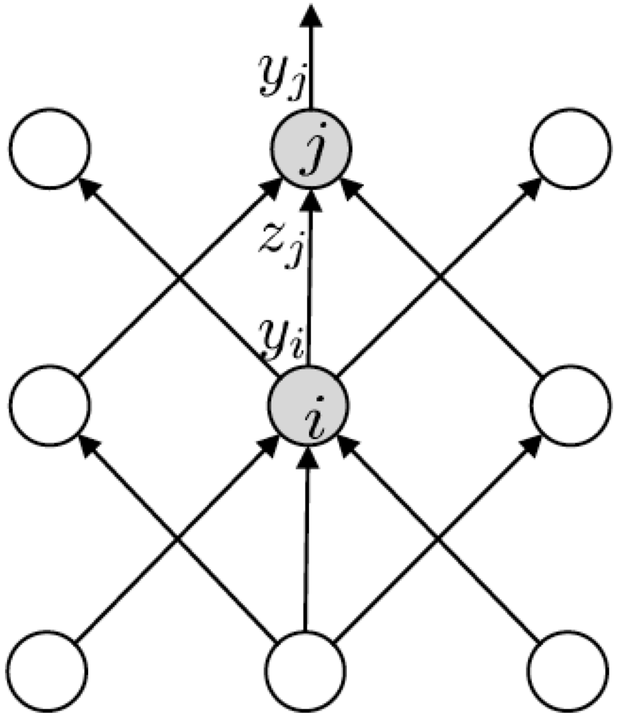
$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



$$\frac{\partial E}{\partial y_j^{(k)}} \longrightarrow \frac{\partial E}{\partial y_j^{(k-1)}}$$

Backpropagation Algorithm



E.g. with sigmoid output nonlinearity

$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

Notation

- y_j Output of layer *j*
- z_j Input of layer *j*

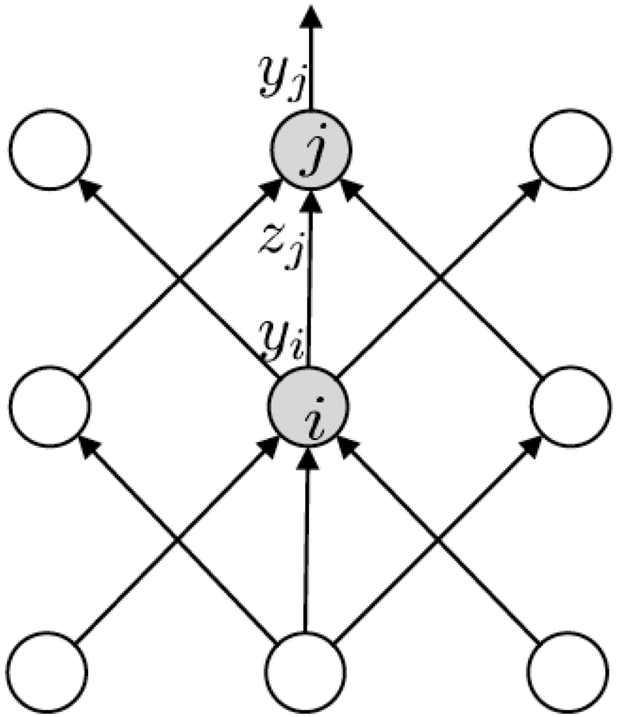
- Connections

$$z_j = \sum_i w_{ij} y_i$$

$$y_j = g(z_j) \qquad y_j = \frac{1}{1+e^{-z}}$$

$$\frac{dy_j}{dz_j} = y_j(1-y_j)$$

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

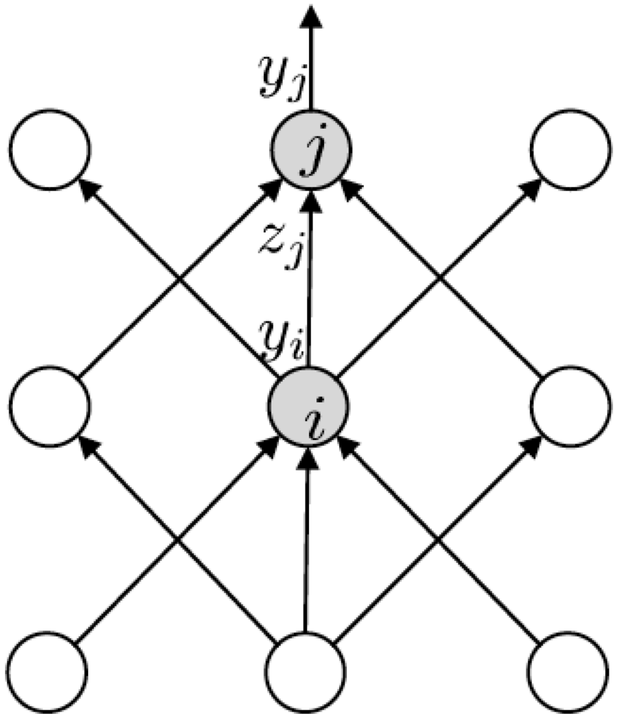
Notation

- y_j Output of layer *j*
- z_j Input of layer *j*

- Connections $z_j = \sum_i w_{ij} y_i$

$$\frac{\partial z_j}{\partial y_i} = w_{ij}$$

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

Notation

- y_j Output of layer *j*
- z_j Input of layer *j*

- Connections $z_j = \sum_i w_{ij} y_i$

$$\frac{\partial z_j}{\partial w_{ij}} = y_i$$

Summary: MLP Backpropagation

Forward Pass

$$\mathbf{y}^{(0)} = \mathbf{x}$$
$$\text{for} \quad k = 1, ..., l \text{ do}$$
$$\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{y}^{(k-1)}$$
$$\mathbf{y}^{(k)} = g_k(\mathbf{z}^{(k)})$$
$$\text{endfor}$$
$$\mathbf{y} = \mathbf{y}^{(l)}$$
$$E = L(\mathbf{t}, \mathbf{y}) + \lambda\Omega(\mathbf{W})$$

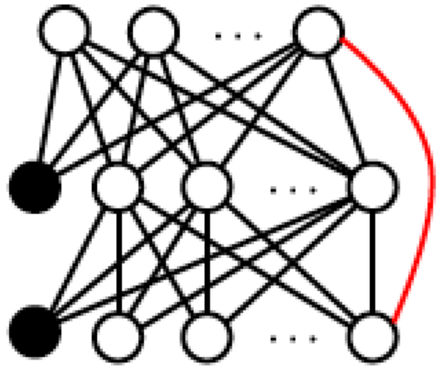
Backward Pass

$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}} L(\mathbf{t}, \mathbf{y}) + \lambda \frac{\partial}{\partial \mathbf{y}} \Omega$$
$$\text{for} \quad k = l, l\text{-}1, ..., 1 \text{ do}$$
$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{z}^{(k)}} = \mathbf{h} \odot g'(\mathbf{y}^{(k)})$$
$$\frac{\partial E}{\partial \mathbf{W}^{(k)}} = \mathbf{h}\mathbf{y}^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial \mathbf{W}^{(k)}}$$
$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}^{(k-1)}} = \mathbf{W}^{(k)\top}\mathbf{h}$$
$$\text{endfor}$$

Notes

- For efficiency, an entire batch of data X is processed at once.
- $\odot$ denotes the element-wise product

Analysis: Backpropagation



Backpropagation is the key to make deep NNs tractable

- However…

The Backprop algorithm given here is specific to MLPs

- It does not work with more complex architectures, e.g. skip connections or recurrent networks!
- Whenever a new connection function induces a different functional form of the chain rule, you have to derive a new Backdrop algorithm for it.

Let's analyse Backprop in more detail

- This will lead us to a more flexible algorithm formulation

References and Future Reading

More information on Neural Networks can be found in Chapters 6 and 7 of the Goodfellow & Bengio book

I. Goodfellow, Y. Bengio, A. Courville
Deep Learning
MIT Press, 2016

http://www.deeplearningbook.org/