Deep Learning: Lecture 3

Alexander Schönhuth

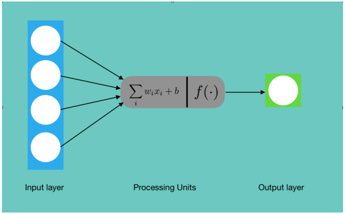
UU

October 10, 2019

Neural Networks

Neurons

Linear + Activation Function



$$\text{output} = a(w^T \cdot x + b)$$

Note: replace f in Figure by a!

Neuron: linear function followed by activation function

Examples

- Linear regression:

$$a = \mathrm{Id}$$

  a is identity function

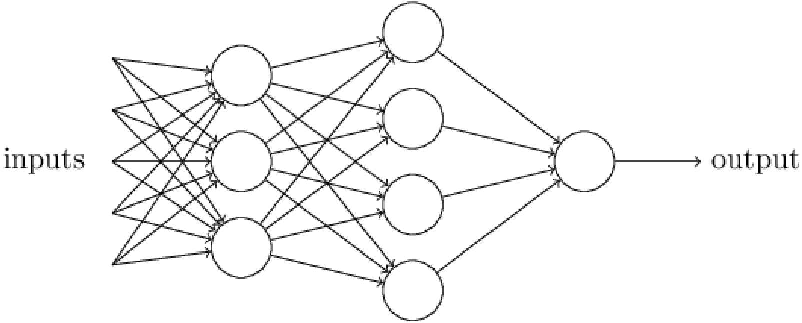
- Perceptron:

$$a(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

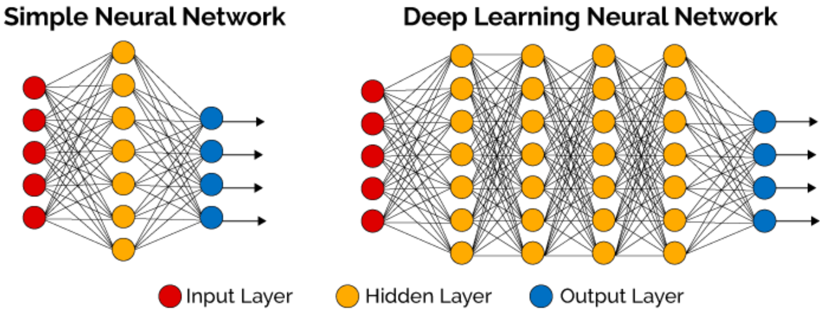
  a is step function

Neural Networks

Concatenating Neurons



Deep Neural Networks



Width = Number of nodes in a hidden layer
Depth = Number of hidden layers
Deep = depth $\geq$ 8 (for historical reasons)

Neural Networks

Formal Definition

- Let $\mathbf{x}^l \in \mathbb{R}^{d(l)}$ be all outputs from neurons in layer l, where $d(l)$ is the *width* of layer l.
- Let $y \in V$ be the output.
- Let $\mathbf{x} =: \mathbf{x}^0$ be the input.
- Then
$$\mathbf{x}^l = \mathbf{a}^l(\mathbf{W}^{(l)}\mathbf{x}^{l-1} + \mathbf{b}^l)$$
where $\mathbf{a}^l(.) = (a_1^l(.), ..., a_{d(l)}^l(.))$, $\mathbf{W}^{(l)} \in \mathbb{R}^{d(l)\times d(l-1)}$, $\mathbf{b}^l \in \mathbb{R}^{d(l)}$
- The function f representing a neural network with L layers (with depth L) can be written
$$y = f(\mathbf{x}^0) = f^{(L)}(f^{(L-1)}(...(f^{(1)}(\mathbf{x}^{(0)}))...))$$
where $\mathbf{x}^l = f^{(l)}(\mathbf{x}^{l-1}) = \mathbf{a}^\mathbf{l}(\mathbf{W}^{(\mathbf{l})}\mathbf{x}^{l-1} + \mathbf{b}^\mathbf{l})$

Why Neural Networks?

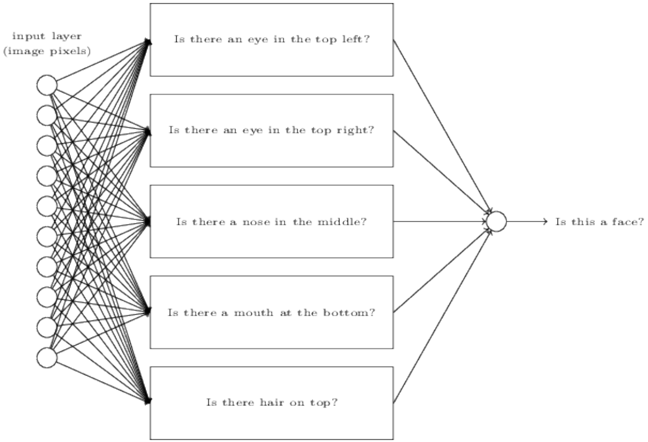
Why Neural Networks?

Given an (unknown) functional relationship $f : \mathbb{R}^d \to V$, why should we learn f by approximating it preferably with a neural network?

Practical, Intuitive Consideration

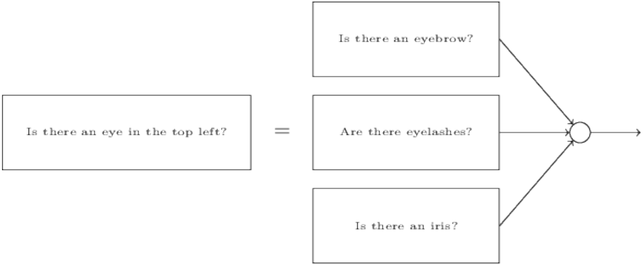
Deep Learning

Intuitive Explanation



- *Face recognition*: decompose classification task into subtasks

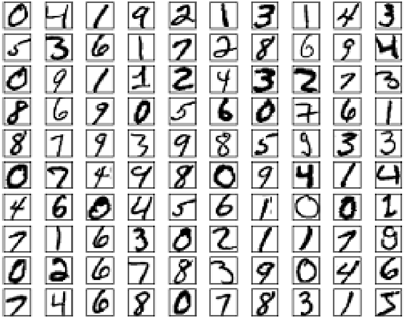
Deep Learning is Intuitive



- *Face recognition*: decompose subtask (eye recognition) into sub-subtasks
- Subtasks are composed into overall task “layer by layer”

Running Example: MNIST Classification

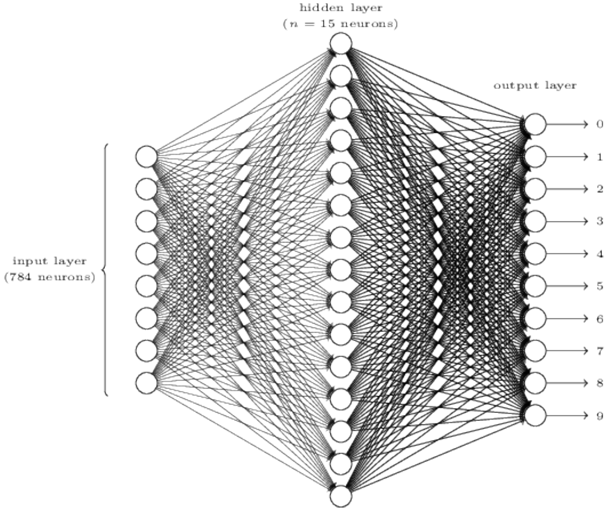
Data, Function



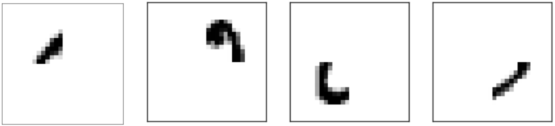
$$f : \mathbb{R}^{28\times 28=784} \longrightarrow \{0, 1, ..., 9\} \quad (1)$$

Running Example

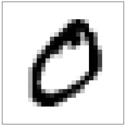
Model Class: NN with 1 hidden layer



Running Example



together makes



Neurons of hidden layer recognize characterizing parts of digit

Theoretical Consideration

The Universal Approximation Theorem

Theorem

A feedforward network with a single hidden layer containing a finite number of neurons can approximate any nonconstant, bounded and continuous function with arbitrary closeness, as long as there are enough hidden nodes.

Black Board Example

Step function with n steps as neural network

- requires n hidden nodes
- hence $O(n)$ training data

Black Board Example

Why Deep Learning?

The Universal Approximation Theorem

Theorem

A feedforward network with a single hidden layer containing a finite number of neurons can approximate any nonconstant, bounded and continuous function with arbitrary closeness, as long as there are enough hidden nodes.

Black Board Example

Step function with n steps as neural network

- requires n hidden nodes
- hence $O(n)$ training data

Black Board Example

Why Deep Learning

- *Great*: as long as there are on the order of m training data, we can learn any step function with m steps using an NN with one hidden layer
- *However*: Both SVM's and Nearest Neighbor can do this, too.
  - Obvious for Nearest Neighbor
  - For SVM's use *Gaussian kernel* or *radial basis function (RBF)* kernel

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}) \tag{2}$$

  - ☞ RBF kernel measures closeness, hence is similar to Nearest Neighbor
- *Moreover*: In particular RBF kernel SVM's enjoy rapid, closed form optimization and fast prediction
- *While*: neural networks do not

Why Deep Learning

- *Great*: as long as there are on the order of m training data, we can learn any step function with m steps using an NN with one hidden layer
- *However*: Both SVM's and Nearest Neighbor can do this, too.
  - Obvious for Nearest Neighbor
  - For SVM's use *Gaussian kernel* or *radial basis function (RBF)* kernel

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}) \tag{2}$$

  - ☞ RBF kernel measures closeness, hence is similar to Nearest Neighbor
- *Moreover*: In particular RBF kernel SVM's enjoy rapid, closed form optimization and fast prediction
- *While*: neural networks do not

WHY DEEP LEARNING

- *Great*: as long as there are on the order of m training data, we can learn any step function with m steps using an NN with one hidden layer
- *However*: Both SVM's and Nearest Neighbor can do this, too.
  - Obvious for Nearest Neighbor
  - For SVM's use *Gaussian kernel* or *radial basis function (RBF)* kernel

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}) \tag{2}$$

  - ☞ RBF kernel measures closeness, hence is similar to Nearest Neighbor
- *Moreover*: In particular RBF kernel SVM's enjoy rapid, closed form optimization and fast prediction
- *While*: neural networks do not

Why Deep Learning

- *Great*: as long as there are on the order of m training data, we can learn any step function with m steps using an NN with one hidden layer
- *However*: Both SVM's and Nearest Neighbor can do this, too.
  - Obvious for Nearest Neighbor
  - For SVM's use *Gaussian kernel* or *radial basis function (RBF)* kernel

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}) \tag{2}$$

  - ☞ RBF kernel measures closeness, hence is similar to Nearest Neighbor
- *Moreover*: In particular RBF kernel SVM's enjoy rapid, closed form optimization and fast prediction
- *While*: neural networks do not

Why Deep Learning

- *Great*: as long as there are on the order of m training data, we can learn any step function with m steps using an NN with one hidden layer
- *However*: Both SVM's and Nearest Neighbor can do this, too.
  - Obvious for Nearest Neighbor
  - For SVM's use *Gaussian kernel* or *radial basis function (RBF)* kernel
$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}) \tag{2}$$
  - ☞ RBF kernel measures closeness, hence is similar to Nearest Neighbor
- *Moreover*: In particular RBF kernel SVM's enjoy rapid, closed form optimization and fast prediction
- *While*: neural networks do not

Why Deep Learning

- *Great*: as long as there are on the order of m training data, we can learn any step function with m steps using an NN with one hidden layer
- *However*: Both SVM's and Nearest Neighbor can do this, too.
  - Obvious for Nearest Neighbor
  - For SVM's use *Gaussian kernel* or *radial basis function (RBF)* kernel
$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}) \tag{2}$$
  - ☞ RBF kernel measures closeness, hence is similar to Nearest Neighbor
- *Moreover*: In particular RBF kernel SVM's enjoy rapid, closed form optimization and fast prediction
- *While*: neural networks do not

Rule of Thumb

One needs approximately

as many training data
as there are parameters

in the class of models

More Layers

Motivation

- We have 2 parameters per hidden neuron, amounting to requiring approximately $2n$ data points
- Can we save on neurons/parameters, while increasing number of steps, by increasing depth?

Black Board Example

Symmetric step function with $2n$ steps
modeled by NN with 2 hidden layers

Black Board Example

Why Deep Learning

- We need only $O(n+1)$ (and not $O(2n)$) many parameters to model a constellation with $2n$ steps and one symmetry axis
- Hence, we only need $O(n+1)$ many training data, and not $O(2n)$ (like SVM's or Nearest Neighbor)
- In general $O(n^l)$ (symmetric) steps need only $O(nl)$ training data
- This illustrates why deeper NN's can deal with symmetry invariance in images

Why Deep Learning

- We need only $O(n+1)$ (and not $O(2n)$) many parameters to model a constellation with $2n$ steps and one symmetry axis
- Hence, we only need $O(n+1)$ many training data, and not $O(2n)$ (like SVM's or Nearest Neighbor)
- In general $O(n^l)$ (symmetric) steps need only $O(nl)$ training data
- This illustrates why deeper NN's can deal with symmetry invariance in images

Why Deep Learning

Curse of Dimensionality

The increase in areas is exponential in the dimensions

- On increasing dimensions, one needs exponentially increasing training data
- Deep NN's, beyond symmetry in one dimension, can deal with invariances in terms of exchanging features (dimensions)
- This explains why they can detect cats in the lower-right corner although training data only showed cats in the upper-left corner

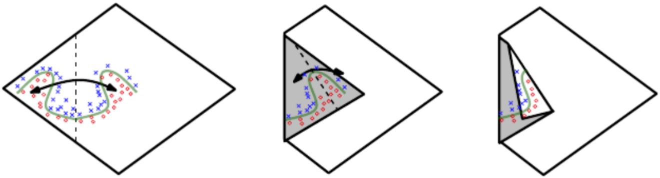
Why Deep Learning

Theorem (Universal Approximation; Montufar (2014))

Let f be an NN with d inputs, l hidden layers (depth l) of width l each. Then the number of differently labeled regions is

$$O(\binom{n}{d}^{d(l-1)} n^d) \tag{3}$$

That is, the number of regions that can receive different labels is exponential in the depth (the number of hidden layers) l.



[Montufar 2014]: Every neuron can fold space along an axis

- Model classes make certain assumptions about properties of the functions they aim to approximate
- Many model classes (such as Nearest Neighbors and SVM's) require *local consistency* and *smoothness*: nearby points are likely to receive the same label
- Deep neural networks make further assumptions such as invariance to shifts, rotations and mirroring

Important Example: XOR Function

$$\begin{aligned} XOR: \quad \{0,1\}^2 \quad &\longrightarrow \quad \{0,1\} \\ (0,0) \quad &\mapsto \quad 0 \\ (0,1) \quad &\mapsto \quad 1 \\ (1,0) \quad &\mapsto \quad 1 \\ (1,1) \quad &\mapsto \quad 0 \end{aligned}$$

Black Board Example

NN with one hidden layer implementing XOR

Black Board Example

Challenges: Optimization

Deep Learning: Challenges

- So, as we have seen, given that we can make some reasonable assumptions about the functions to be learnt, deep learning is just awesome, both
  - powerful and
  - intuitive

Deep Learning: Challenges

- So, as we have seen, given that we can make some reasonable assumptions about the functions to be learnt, deep learning is just awesome, both
  - powerful and
  - intuitive

Where is the trouble?

DEEP LEARNING: CHALLENGES

- The functions $f_{\mathbf{w}}$ representing NN's cannot be described in closed form
- Hence the loss $C(\mathbf{w}) := C(f_{\mathbf{w}}) := C(f_{\mathbf{w}}, f^*)$ cannot be described in closed form either
- However, we need to both
  - evaluate $f_{\mathbf{w}}$ when predicting
  - optimize with respect to a loss function $C(f_{\mathbf{w}})$ ☞ we require to get control of the gradient $\nabla_{\mathbf{w}} C(f_{\mathbf{w}})$
- both difficult when not in possession of closed form description

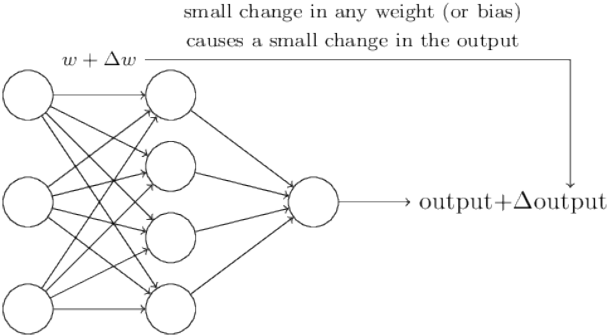
Deep Learning: Challenges

- The functions $f_{\mathbf{w}}$ representing NN's cannot be described in closed form
- Hence the loss $C(\mathbf{w}) := C(f_{\mathbf{w}}) := C(f_{\mathbf{w}}, f^*)$ cannot be described in closed form either
- However, we need to both
  - evaluate $f_{\mathbf{w}}$ when predicting
  - optimize with respect to a loss function $C(f_{\mathbf{w}})$ ☞ we require to get control of the gradient $\nabla_{\mathbf{w}} C(f_{\mathbf{w}})$
- both difficult when not in possession of closed form description

How to overcome the issue?

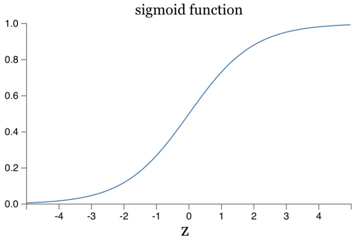
Activation Function

Motivation



- Output needs to be differentiable in the weights
- *Recall*: We would like to compute gradients

Sigmoid Neurons



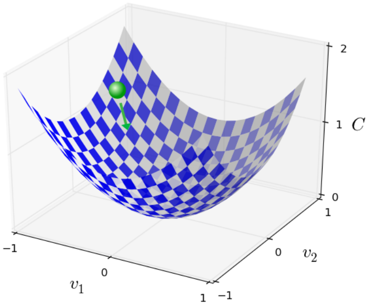
- Perceptrons, where activation functions are step functions do not work as neuron model, because they are not differentiable
- *Idea*: Use *sigmoid functions* (i.e. ”smoothed step functions”)

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad \text{where} \quad z = \sum_j w_j x_j + b \tag{4}$$

as activation function ☞ *sigmoid neurons*

Gradient Descent

Gradient Descent



- Let $C(v_1, \ldots, v_n)$ be a differentiable function in n variables, here $n = 2$. We look for the minimum of C.
- *Idea*: At point v_1, v_2 (green ball), move into direction of steepest decline (green arrow).

Gradient Descent

Algorithm

- When at $\mathbf{v} = (v_1, v_2)$, compute gradient

$$\nabla_{\mathbf{v}} C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{5}$$

- We know that

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 = \nabla_{\mathbf{v}} C^T \cdot \Delta v \tag{6}$$

- Choosing $\Delta v = \eta \nabla_{\mathbf{v}} C$ yields [note: η is another hyperparameter!]

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2 \leq 0 \tag{7}$$

- So, updating

$$v \longrightarrow v' = v - \eta \nabla C \tag{8}$$

guarantees to decrease C.

- Repeat until done (for example in case of convergence)

Gradient Descent

Algorithm

- When at $\mathbf{v} = (v_1, v_2)$, compute gradient

$$\nabla_{\mathbf{v}} C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{5}$$

- We know that

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 = \nabla_{\mathbf{v}} C^T \cdot \Delta v \tag{6}$$

- Choosing $\Delta v = \eta \nabla_{\mathbf{v}} C$ yields [note: η is another hyperparameter!]

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2 \leq 0 \tag{7}$$

- So, updating

$$v \longrightarrow v' = v - \eta \nabla C \tag{8}$$

  guarantees to decrease C.
- Repeat until done (for example in case of convergence)

Gradient Descent

Algorithm

- When at $\mathbf{v} = (v_1, v_2)$, compute gradient

$$\nabla_{\mathbf{v}} C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \tag{5}$$

- We know that

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 = \nabla_{\mathbf{v}} C^T \cdot \Delta v \tag{6}$$

- Choosing $\Delta v = \eta \nabla_{\mathbf{v}} C$ yields [note: η is another hyperparameter!]

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2 \leq 0 \tag{7}$$

- So, updating

$$v \longrightarrow v' = v - \eta \nabla C \tag{8}$$

guarantees to decrease C.

- Repeat until done (for example in case of convergence)

Gradient Descent

Algorithm

- When at $\mathbf{v} = (v_1, v_2)$, compute gradient

$$\nabla_{\mathbf{v}} C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \tag{5}$$

- We know that

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 = \nabla_{\mathbf{v}} C^T \cdot \Delta v \tag{6}$$

- Choosing $\Delta v = \eta \nabla_{\mathbf{v}} C$ yields [note: η is another hyperparameter!]

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2 \leq 0 \tag{7}$$

- So, updating

$$v \longrightarrow v' = v - \eta \nabla C \tag{8}$$

  guarantees to decrease C.
- Repeat until done (for example in case of convergence)

Gradient Descent for Neural Networks

Practical Scheme

Input

- A NN with appropriately chosen initial parameters $\mathbf{w_0}$
- Training data $\mathbf{X}^{(\text{train})} \in \mathbb{R}^{m \times n}, \mathbf{y}^{(\text{train})} \in \mathbb{R}^m$
  ☞ m training data points $x \in \mathbb{R}^n$
- Cost function

$$C = \frac{1}{m} \sum_x C_x = \frac{1}{m} \sum_x C(f(x), y(x))$$

Gradient Descent for Neural Networks

Practical Scheme

Iteration i

1. Compute $\nabla_{\mathbf{w}} C(\mathbf{w}_{i-1})$
   - Need training data to update C, based on having updated $\mathbf{w}$
2. Update: $\mathbf{w}^{(\mathbf{i})} \leftarrow \mathbf{w}^{(\mathbf{i-1})} + \eta \nabla_{\mathbf{w}} C$
   - $w_k^{(i)} \leftarrow w_k^{(i-1)} - \eta \frac{\partial C}{\partial w_k}$
   - $b_l^{(i)} \leftarrow b_l^{(i-1)} - \eta \frac{\partial C}{\partial b_l}$
3. Stop, if appropriate

Gradient Descent

Things to consider in practice

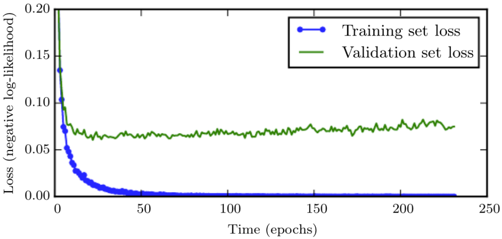
- Choose appropriate η
  - Too small η: too slow convergence
  - Too large η: (6) no longer good approximation
- Direction of gradient minimizes ΔC the most
- *Stochastic Gradient Descent*: Divide m training data points into small batches of sizes $m_1, ..., m_l$ where $m_1 + ... + m_l = m$.
  - Run gradient descent on each batch separately. For each batch $h = 1, ..., H$, update
    - $w_k^{(i)} \leftarrow w_k^{(i-1)} - \frac{\eta}{m_h} \sum_{j=1}^{m_h} \frac{\partial C_{x_j}}{\partial w_k}$
    - $b_l^{(i)} \leftarrow b_l^{(i-1)} - \frac{\eta}{m_h} \sum_{j=1}^{m_h} \frac{\partial C_{x_j}}{\partial b_l}$

    until all batches are processed
  - Variations conceivable!
  - *Epoch*: One round of using *all training data* (that is using all batches)

Early Stopping Revisited

Reminder: Early Stopping

Regularization



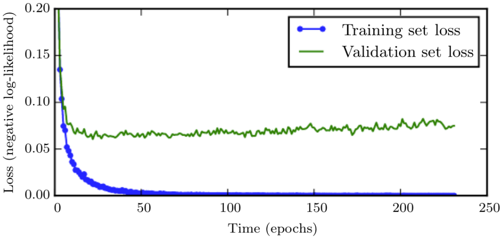
Epoch: One iteration of using *all* training data

How to Stop Early?

- Run gradient descent on training data
- After each iteration (epoch), evaluate C on validation data $\mathbf{X}^{(\mathrm{val})}, \mathbf{y}^{(\mathrm{val})}$
- Stop if no improvements on $\mathbf{X}^{(\mathrm{val})}, \mathbf{y}^{(\mathrm{val})}$ can be seen

Reminder: Early Stopping

Regularization



Epoch: One iteration of using *all* training data

General Wisdom

- Points nearby training optimum generalize better
- *But*: No consistent theory to support this intuition available

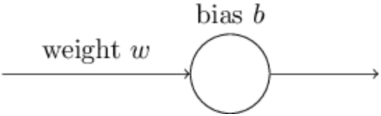
Preventing Slow Learning

Slow Learning

Slow Learning II

Sigmoid Function

Drawback

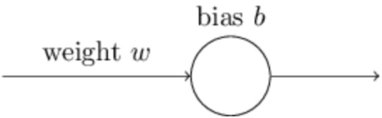


Cost function:

$$C = \frac{(y-a)^2}{2} = \frac{(y-\sigma(z))^2}{2} = \frac{(y-\sigma(wx+b))^2}{2} \tag{9}$$

Sigmoid Function

Drawback



Cost function:

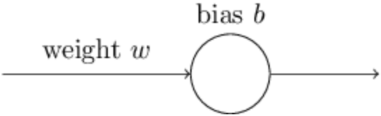
$$C = \frac{(y-a)^2}{2} = \frac{(y-\sigma(z))^2}{2} = \frac{(y-\sigma(wx+b))^2}{2} \tag{9}$$

Training input $x = 1$, desired output $y = 0$:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \quad \text{and} \quad \frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z) \tag{10}$$

Sigmoid Function

Drawback



Cost function:

$$C = \frac{(y-a)^2}{2} = \frac{(y-\sigma(z))^2}{2} = \frac{(y-\sigma(wx+b))^2}{2} \tag{9}$$

Training input $x = 1$, desired output $y = 0$:

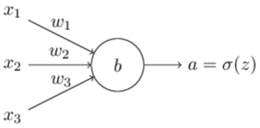
$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \quad \text{and} \quad \frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z) \tag{10}$$

When $\sigma(z) \approx 0$ or $\sigma(z) \approx 1$, we have $\sigma'(z) \approx 0$ ☞ learning slows down

Sigmoid Neurons

Remedy: Alternative cost function

Consider



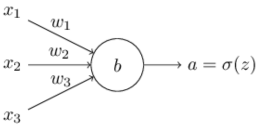
where $z = \sum_j w_j x_j + b$.

Issue: Sigmoid activation and quadratic cost make unfortunate combination

Sigmoid Neurons

Remedy: Alternative cost function

Consider



where $z = \sum_j w_j x_j + b$.

Issue: Sigmoid activation and quadratic cost make unfortunate combination

Solution: Use alternative cost function: *cross entropy*:

$$C = -\frac{1}{m} \sum_x [y(x) \log a(x) + (1 - y(x)) \log(1 - a(x)) \tag{11}$$

where x runs over all m training examples.

Lecture 3 ended here, see further below for Summary and Homework

Cross Entropy

Cross entropy:

$$C = -\frac{1}{m}\sum_x [y(x)\log a(x) + (1-y(x))\log(1-a(x))] \tag{12}$$

where x runs over all m training examples.

Remarks

- $C \geq 0$: log's are negative because of $a(x) = \sigma(z) \in [0,1]$, minus sign in front
- C close to zero if $y(x) \approx a(x)$ (considering $y(x) \in \{0,1\}$)
- If $y(x) \in [0,1]$, cross entropy C is minimal iff $a(x) = y(x)$.

Cross Entropy

Substituting $a = \sigma(z)$ into (11), we obtain

$$\begin{aligned}\frac{\partial C}{\partial w_j} &= -\frac{1}{m}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y(x))}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j} \\ &= -\frac{1}{m}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y(x))}{1-\sigma(z)}\right)\sigma'(z)x_j\end{aligned} \tag{13}$$

Further simplifying yields:

$$\frac{\partial C}{\partial w_j} = \frac{1}{m}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z) - y) \tag{14}$$

Cross Entropy

Realizing that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, we finally obtain

$$\frac{\partial C}{\partial w_j} = \frac{1}{m} \sum_x x_j(\sigma(z) - y(x)) \tag{15}$$

Similarly

$$\frac{\partial C}{\partial b} = \frac{1}{m} \sum_x (\sigma(z) - y(x)) \tag{16}$$

Fast Learning

Fast Learning II

Cross Entropy

Multineuron output

Cross entropy also works for more than one output neuron. Let $y(x) = (y_1(x), ..., y_d(x))$ be the true labels, while $a^L(x) = (a_1^L(x), ..., a_d^L(x))$ are the actual output values.

Then multi output neuron cross entropy is defined by

$$C = -\frac{1}{m}\sum_x \sum_j [y_j(x) \log a_j^L(x) + (1 - y_j(x)) \log(1 - a_j^L(x))] \quad (17)$$

where $j = 1, ..., d$.

Softmax

Consider the case of J outputs $a_j^L, j = 1, ..., J$. Let (as usual)

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L \tag{18}$$

be the input to the corresponding J neurons making the output layer.

Then the *softmax activation* is defined by

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{19}$$

SOFTMAX

Note that

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1 \qquad (20)$$

- All outputs are positive
- A softmax layer can be thought of as a probability distribution over the J different possible outputs.
- *Observation*: Softmax output values depend on the inputs to all output neurons, and not only on the particular one that generates the output.

Softmax

Cost Function

Let $(x, y(x))$ be one training example, where $y(x) \in \{1, ..., J\}$. Then the *log-likelihood cost* is defined to be

$$-\log a^L_{y(x)} \tag{21}$$

Let here $y_j = 1$ iff $j = y(x)$ and $y_j = 0$ iff $j \neq y(x)$ (in abuse of earlier notation). Then we obtain

$$\frac{\partial C}{\partial b^L_j} = a^L_j - y_j \tag{22}$$

$$\frac{\partial C}{\partial w^L_{jk}} = a^{L-1}_k (a^L_j - y_j) \tag{23}$$

Note that (22) and (23) are, apart from not summing over many training examples here, identical to (15) and (16).

So, what is better, sigmoid + cross-entropy, or softmax + loglikelihood? It depends, in fact both can lead to good results in many cases.

Alternative Activation Functions

Tangens Hyperbolicus

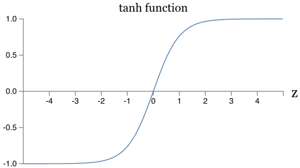
Tangens hyperbolicus is defined by

$$\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{24}$$

It holds that

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2} \tag{25}$$

so tanh turns out to be a scaled version of the sigmoid function σ.



Tangens hyperbolicus is a scaled version of a sigmoid

Tangens Hyperbolicus

Motivation

Remember that

$$\frac{\partial C}{\partial w_{jk}^{l+1}} = a_k^l \delta_j^{l+1} \tag{26}$$

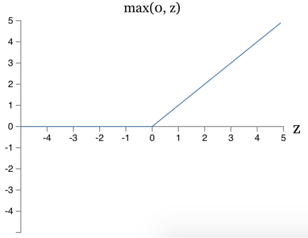
When using sigmoid neurons, $a_k^l \in [0, 1]$, hence non-negative, while for tangens hyperbolicus $a_k^l \in [-1, 1]$, so possibly also negative. Hence, if $\delta_j^{l+1} > 0$ (or $\delta_j^{l+1} < 0$) then

- all weights w_{jk}^{l+1} will decrease (or increase) for sigmoid neurons
- some weights will decrease, and some weights will increase (or vice versa) for tanh neurons

The latter case can be advantageous.

However, empirically, tanh were not found to have decisive advantages over sigmoid neurons.

Alternative Activation Functions
Rectified Linear Units



Rectifying function

The rectified linear function with input z is defined by

$$\max(0, z) \tag{27}$$

so a *rectified linear neuron* with input $\mathbf{x}$, weight vector $\mathbf{w}$ and bias b is defined by

$$\max(0, \mathbf{w}\mathbf{x} + b) \tag{28}$$

Rectified Linear Neurons

Properties

- No theoretical deep understanding available
- Rectified linear neurons do not saturate on positive input
  ☞ no learning slowdown
- When input is negative, rectified linear neurons stop learning entirely!
- Empirically, rectified linear neurons have been proven to be of great use in image recognition

Rectified Linear Neurons

Literature

- "What is the best multi-stage architecture for object recognition?", `http://yann.lecun.com/exdb/publis/pdf/jarrett-iccv-09.pdf`
- "Deep sparse rectifier neural networks", `http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf`
- "ImageNet classification with deep convolutional neural networks", `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional pdf`
- Papers provide interesting details about choice of cost functions, setting up the output layer, and regularization.

HOMEWORK

- Modeling 'XOR': read through chapter 6.1 in Bengio's book, see `http://www.deeplearningbook.org/contents/mlp.html`
- Install and learn Python (see Lecture 1 for links)
- Download *tutorial_0.py* (see 'Tutorial Basic Python Program') from BlackBoard and make sure it runs on your laptop
- Try to understand *tutorial_0.py* if you wish ...
- If not, I'll explain you next time!

HOMEWORK

- Modeling 'XOR': read through chapter 6.1 in Bengio's book, see `http://www.deeplearningbook.org/contents/mlp.html`
- Install and learn Python (see Lecture 1 for links)
- Download *tutorial_0.py* (see 'Tutorial Basic Python Program') from BlackBoard and make sure it runs on your laptop
- Try to understand *tutorial_0.py* if you wish ...
- If not, I'll explain you next time!

HOMEWORK

- Modeling 'XOR': read through chapter 6.1 in Bengio's book, see `http://www.deeplearningbook.org/contents/mlp.html`
- Install and learn Python (see Lecture 1 for links)
- Download *tutorial_0.py* (see 'Tutorial Basic Python Program') from BlackBoard and make sure it runs on your laptop
- Try to understand *tutorial_0.py* if you wish ...
- If not, I'll explain you next time!

Homework

- Modeling 'XOR': read through chapter 6.1 in Bengio's book, see `http://www.deeplearningbook.org/contents/mlp.html`
- Install and learn Python (see Lecture 1 for links)
- Download *tutorial_0.py* (see 'Tutorial Basic Python Program') from BlackBoard and make sure it runs on your laptop
- Try to understand *tutorial_0.py* if you wish ...
- If not, I'll explain you next time!

Lecture 3: Summary

- *Topics*:
  - Deep Learning: Motivation
  - Gradient Descent
  - Addressing Slow Learning
- *Reading*:
  - `http://neuralnetworksanddeeplearning.com`: Chapter 1, Chapter 2 until 'Overfitting and Regularization'
  - `https://www.deeplearningbook.org/`: 6.1, 6.2 (until 6.2.1.1), 6.3 (not treated today, but next time), 6.4, see also 6.6, if interested

Thanks for your attention