Introduction to machine learning

Maksim Kretov

Lecture 5: Basic concepts for neural networks and BP algorithm

5vision, 2017

Course information

Course

10 lectures + 2 seminars; February-May 2017.

Schedule and up-to-date syllabus

https://goo.gl/xExEuL

Contact information and discussion

Maksim Kretov (kretovmk@gmail.com)

Slack group: https://miptmlcourse.slack.com

to get an invite, send e-mail to kretovmk@gmail.com.

Plan of the course

Math and basics of ML	(1-2)		*Theoretical tasks*
Some of ML methods	(3)		
Seminar on ML basics	(4)		
<u>Basics of neural networks</u>	(5)	← **Today**	*+Practical tasks*
Deep learning overview	(6)		
Training deep networks	(7)		
DL for Computer Vision	(8-9)	**Solving more complex ML tasks using NNs**	
DL for time series prediction	(10-11)		
Concluding seminar	(12)		

Plan for the lecture

A. Previous lectures

1. ML tasks
2. ERM framework

B. Basic definitions for neural networks (NNs)

1. Perceptron, MLP
2. Universality of NNs
3. NN structure for MNIST
4. Loss function

C. Training neural networks

1. Overview and SGD
2. Backpropagation (BP) algorithm
3. Other training methods

D. Practical assignment

A.1 Previous lectures: ML tasks

Supervised learning:

Training set: $\boldsymbol{D} = \{(\mathbf{x}_{\boldsymbol{n}}, y_n), n = 1, .. N\}$ *(inputs and labels!)*

Y are class ids or numbers => classification or regression

Task to solve:

Predict y^* for new input $\mathbf{x}^*$ => **Focus on accurate prediction**

Unsupervised learning:

Training set: $D = \{\mathbf{x}_{\boldsymbol{n}}, n = 1, .. N\}$ *(just inputs!)*

Task to solve:

Finding compact description of data

A.2 Previous lectures: ERM framework

Empirical risk minimization approach (ERM)

Formula for fitting the model within ERM framework:

$$\theta^{opt} = \mathbf{argmin}_{\theta} \frac{1}{N} \sum_{n=1}^{N} L(y_n, \hat{y}_n) + \lambda \Omega(\theta)$$

$L\big(y_n, f(\mathbf{x}_{\boldsymbol{n}}, \theta)\big)$ is loss function; $\hat{y}_n = f(\mathbf{x}_{\boldsymbol{n}}, \theta)$ is prediction

$\Omega(\theta)$ is a regularizer (penalize certain values of θ, for example: L1, L2)

=> learning converted into optimization task!

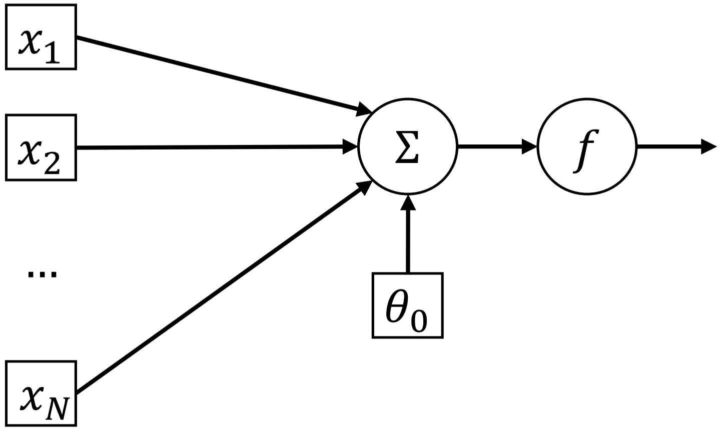
Classification of new input:

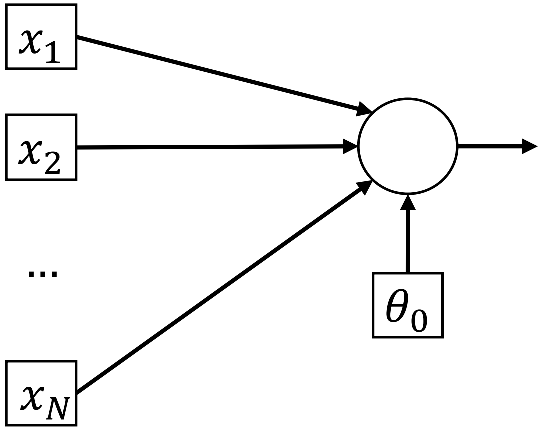
$$\hat{y} = f(\mathbf{x}^{*}, \theta^{opt})$$

B.1 Basic definitions: Perceptron, MLP

<u>Perceptron</u> (McCulloch-Pitts neuron)

Perceptron has very limited learning capacity. For example, cannot learn XOR.





$$\hat{y} = f(\theta^T \mathbf{x})$$

$$f(x) = \begin{cases} 1 \; if \; x > 0 \\ 0 \; if \; x < 0 \end{cases}$$ (Heaviside function)

<u>Problem of linearly separable two-class classification task</u>

$\theta^T \mathbf{x} > 0$, if $\mathbf{x}$ is of class 0
$\theta^T \mathbf{x} < 0$, if $\mathbf{x}$ is of class 1

<u>Training procedure (perceptron rule)</u>

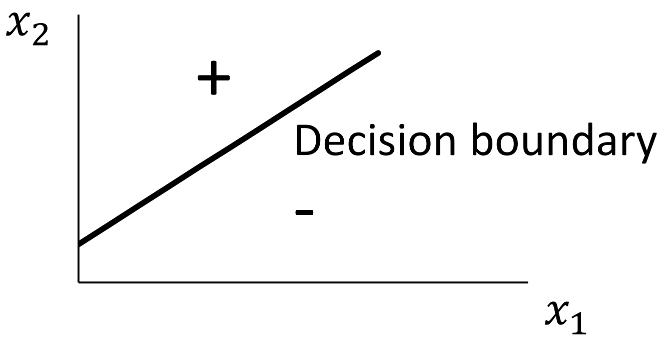
$$\theta^{(i)} = \theta^{(i-1)} + \alpha_i \sum_{\mathbf{x}_n \in M} y_n \mathbf{x}_n$$

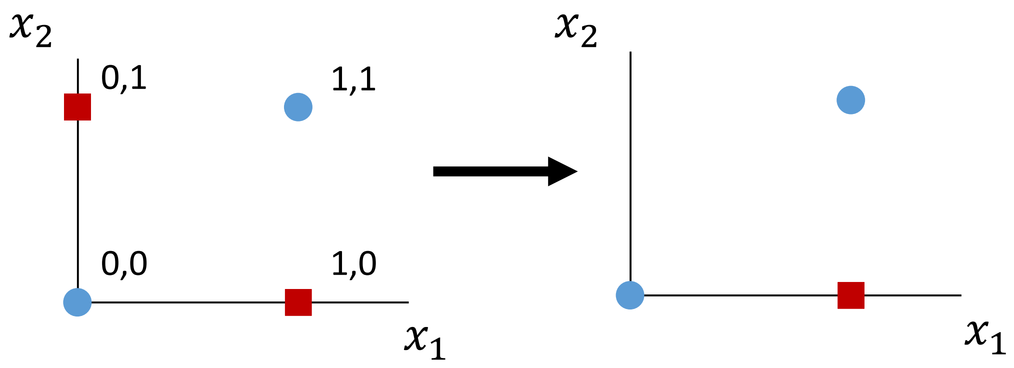
M – misclassified examples

B.1 Basic definitions: Perceptron, MLP

Perceptron (McCulloch-Pitts neuron)

$\hat{y} = f(\theta^T \mathbf{x})$
$\dim(\mathbf{x}) = 2$





Applying least squares to classification task => $\theta = (0,0,1/2)$, i.e. classifier puts 0.5 everywhere.

=> Problem: How to deal with non linearly separable problems?

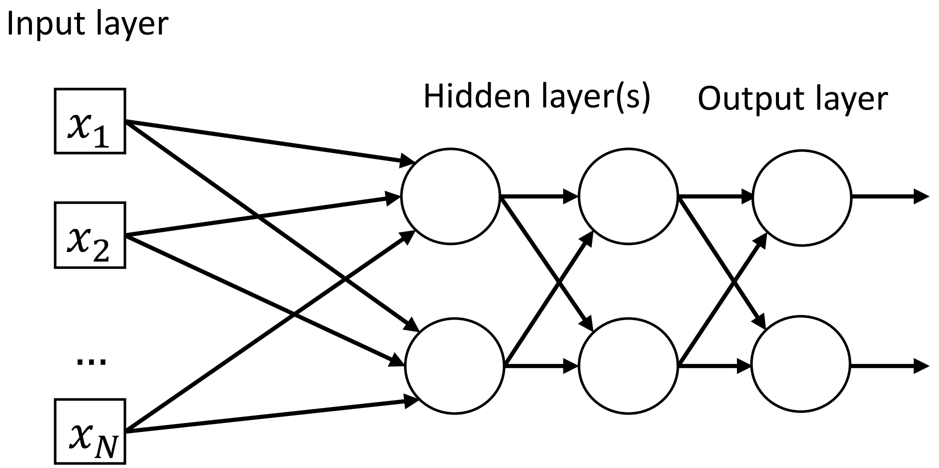
Let's make better representation of input data:

1) Convert to another feature space (**non-linear** transformation): $\max\left[\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}1 & 1\\1 & 1\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix} + \begin{pmatrix}0\\-1\end{pmatrix}\right]$

2) Perform linear classification, because task is now linearly separable.

B.1 Basic definitions: Perceptron, MLP

Feed-forward multilayer neural network (MLP)



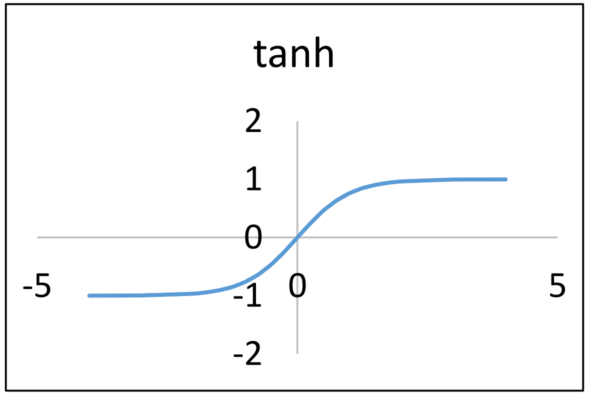
Let's write these transformations using matrix notations:

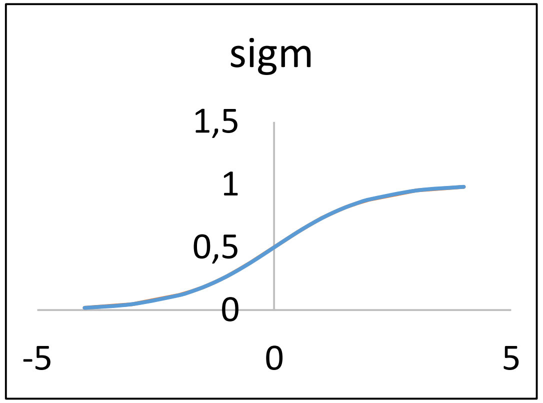
$$a^1 = f^1(W^1 x + b^1)$$

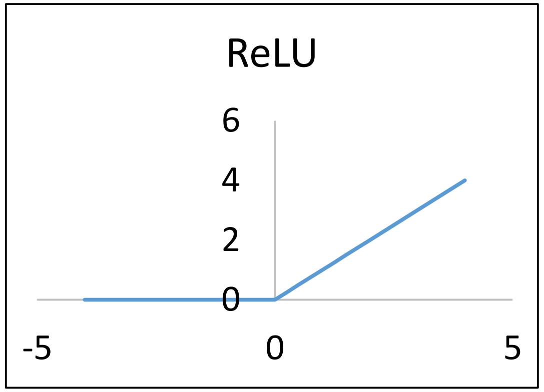
…

$$a^L = f^L(W^L a^{L-1} + b^L)$$

Logistic regression models stacked on top of each other with the final layer being regression / classification model.







$$\mathrm{ReLU}(x) = \max(0, x)$$

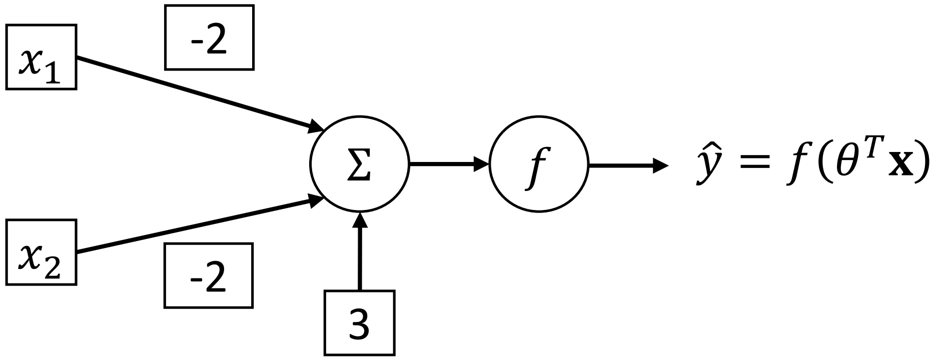
faster than exp!

$$\mathrm{sigm}(x) = \frac{1}{1 + \exp(-x)}$$

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

B.2 Basic definitions: Universality of NNs

Perceptron making NAND gate



X	Y	X|Y
0	0	1
0	1	1
1	0	1
1	1	0

Provides basis for the rest of boolean functions of two variables (NAND gate is universal).

=> We can use combination of perceptrons to calculate any boolean function.

B.2 Basic definitions: Universality of NNs

MLP can approximate any function

Consider NN with one hidden layer (K neurons), single output neuron and activation function f:

$\mathrm{OUT}(\mathbf{x}) = \sum_{k=1}^{K} c_k f(\theta^T \mathbf{x}) + c_0$

$f(z)$ is non-constant, bounded and monotonically-increasing function.

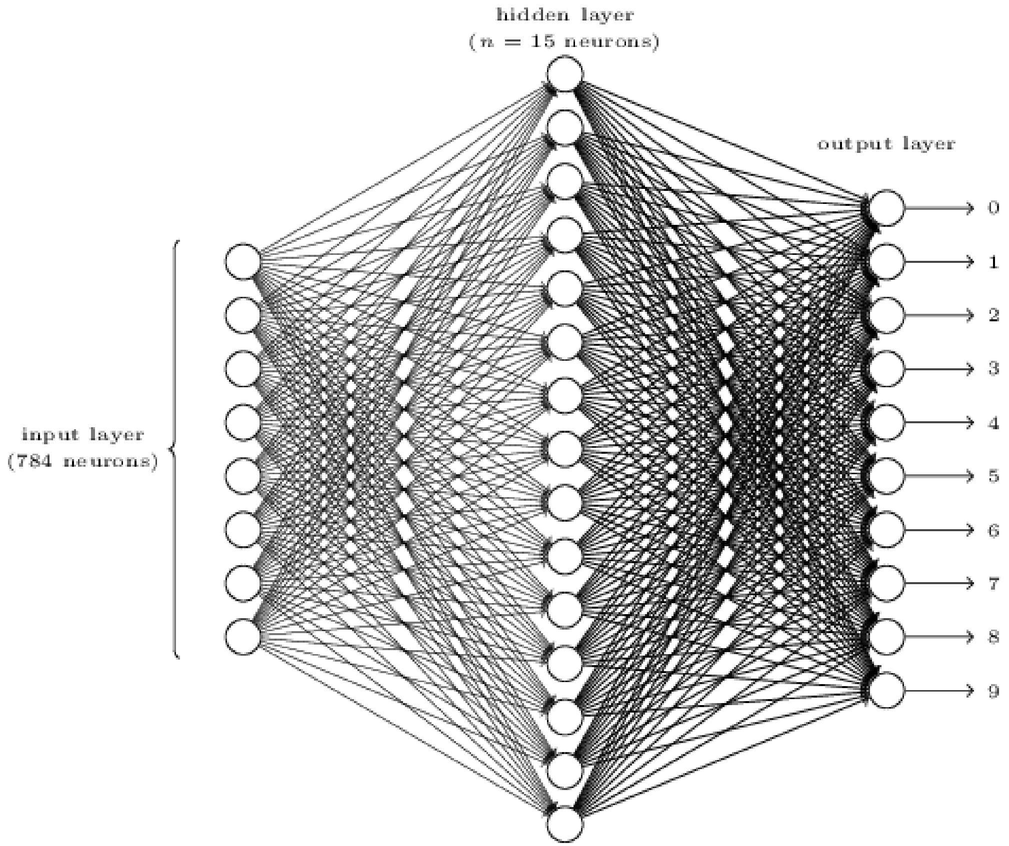
Theorem:

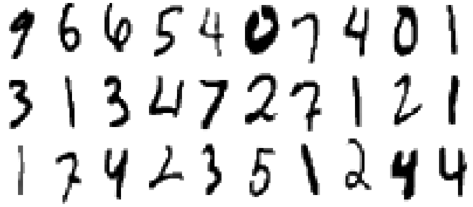
Let $g(\mathbf{x})$ be a continuous function defined in a compact subset $\mathbf{S} \subset \mathbf{R}^n$ and any $\varepsilon > 0$. Then there is a two layer network with $\mathrm{K}(\varepsilon)$ hidden nodes of the form (1), so that:

$|g(\mathbf{x}) - \mathrm{OUT}(\mathbf{x})| < \varepsilon \; \forall \, \mathbf{x} \in \mathbf{S}$

B.3 Basic definitions: NN structure for MNIST

Using MLP for solving MNIST task





70k images of handwritten digits (28x28=784)

Why 10 neurons in out layer? Viable options:

a) 1 neuron (regression)

b) 4 neurons (binary code)

c) 10 neurons (for each numeral)

Softmax activation function for output layer:

$$\hat{y}_{nk} = \frac{\exp(z_{nk})}{\sum_{k=1}^{d} \exp(z_{nk})}$$ normalized probabilities

* Picture from http://neuralnetworksanddeeplearning.com/chap1.html

B.4 Basic definitions: Loss function

Loss functions for NNs:

What properties should loss function have?

- be smooth and easy to differentiate (accuracy is not smooth!)
- be somehow connected to a clear proxy such as accuracy
- ideally have a probabilistic interpretation

B.4 Basic definitions: Loss function

Possible loss functions for NNs:

1. **Quadratic cost** (both classification and regression)

$$L\big(Y, f(\mathbf{X}, \theta)\big) = \frac{1}{N}\sum_{n=1}^{N} L(y_n, \hat{y}_n) = \frac{1}{2N}\sum_{n=1}^{N} \|\hat{y}_n - y_n\|^2$$

2. **Cross-entropy cost** (usual choice for classification)

$$H(p,q) = -\sum_k p_k \ln q_k \qquad \Rightarrow \qquad L\big(Y, f(\mathbf{X}, \theta)\big) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{d} y_{nk} \ln \hat{y}_{nk}$$

From probabilistic point of view, we parameterize $p(y|x, \theta)$ =>

- Maximizing likelihood of correct class in predicted distribution
- KL divergence between true and predicted distributions

Exercise: *Show that minimizing cross-entropy equivalent to minimizing KL div.*

2 minute break..

Questions?

C.1 Training NNs: Overview and SGD

Gradient descent:

$$\theta^{(i)} = \theta^{(i-1)} - \alpha_i \nabla_\theta L\big(Y, f(\mathbf{X}, \theta)\big)$$

Stochastic gradient descent algorithm (SGD):

SGD updates parameters after each example (online learning):

1. Initialize θ
2. For each training example $(\mathrm{y}_n, \mathbf{x}_n)$ do (= one epoch):

$$\theta \leftarrow \theta - \alpha \nabla_\theta L\big(y_n, f(\mathbf{x}_n, \theta)\big)$$

3. Repeat until stop criterion is met

C.2 Training NNs: Backpropagation algorithm

Naïve approach: let's find derivatives numerically!

Much slower.. Full forward pass for each parameter is needed.

$\frac{\partial L}{\partial \omega_i} \approx \frac{L(\omega_i+\varepsilon)-L(\omega_i)}{\varepsilon}$ and same for each parameter (millions of them).

Backpropagation algorithm (BP):

Idea: let's use chain rule in order to calculate derivatives (θ is a vector!):

$$\frac{\partial}{\partial \theta} L\big(y_n, f(\mathbf{x}_n, \theta)\big) = \frac{\partial L(y_n, f)}{\partial f} \frac{\partial f(\mathbf{x}_n, \theta)}{\partial \theta}$$

EASY! HARD..

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – notations:

ω_{jk}^{l} - weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.

b_j^l - bias for j^{th} neuron in the l^{th} layer.

a_j^l - activation of the j^{th} neuron in the l^{th} layer.

$$a_j^l = \sigma\left(\sum_k \omega_{jk}^l a_k^{l-1} + b_j^l\right)$$

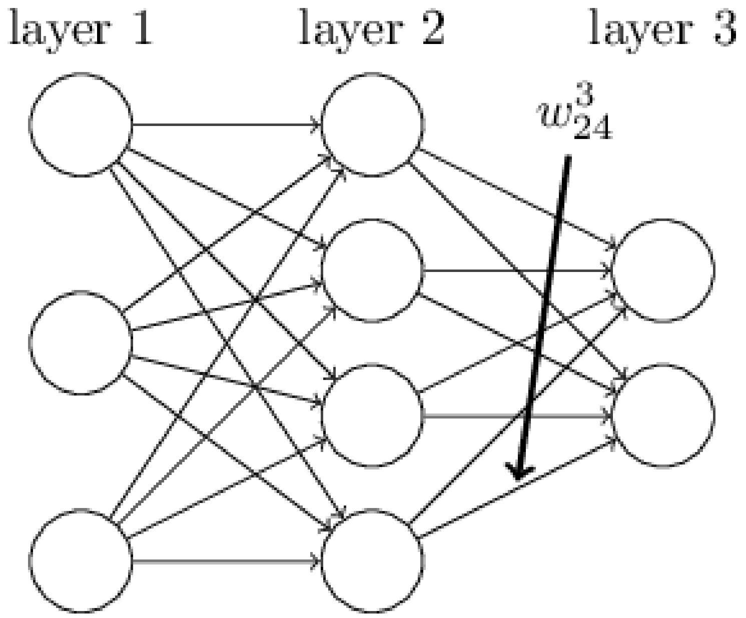


*image from neuralnetworksanddeeplearning.com/chap2.html

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – matrix notations:

Same in matrix form:

$\omega_{jk}^{l} \rightarrow \omega^{l}$ - weight matrix for the l^{th} layer

$b_{j}^{l} \rightarrow b^{l}$ - bias vector for the l^{th} layer.

$a_{j}^{l} \rightarrow a^{l}$ - activation vector for the l^{th} layer.

$z^{l} = \omega^{l} a^{l-1} + b^{l}$ (weighted input)

$a^{l} = \sigma(z^{l})$ (element-wise)

=> Use ext. libraries for fast matrix calculation.

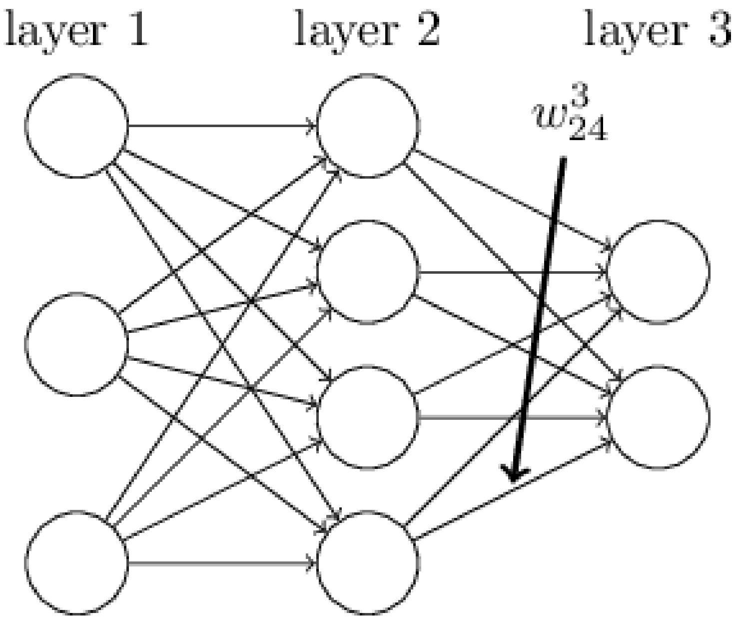


*image from neuralnetworksanddeeplearning.com/chap2.html

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – easy part:

$a^L\,(\mathbf{x}_{\boldsymbol{n}}) = f(\mathbf{x}_{\boldsymbol{n}}, \theta)$ – activation of neurons of L^{th} layer.

And let's find derivative of loss function first:

$$\frac{\partial L(y_n, f)}{\partial f} = \frac{1}{2N}\frac{\partial}{\partial f}\sum_{n=1}^{N}(f(\mathbf{x}_{\boldsymbol{n}}, \theta) - y_n)^2 = \frac{1}{2N}\frac{\partial}{\partial a^L}\sum_{n=1}^{N}\left(a^L\,(\mathbf{x}_{\boldsymbol{n}}) - y_n\right)^2 =$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left(a^L\,(\mathbf{x}_{\boldsymbol{n}}) - y_n\right)$$

Definition

Hadamar product of matrices:

$(A \circ B)_{ij} = A_{ij}B_{ij}$

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – formulas:

More complex task: Calculating $\frac{\partial a^L}{\partial \theta}$, where θ are weights ω_{jk}^l and biases b_j^l.

Let's introduce auxiliary quantities:

$\delta_j^l \equiv \frac{\partial L}{\partial z_j^l}$ (error on l^{th} layer)

$\sigma(x) = \frac{1}{1+\exp(-x)}$

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Equation #1:

$\delta_j^L = \frac{\partial L}{\partial a_j^L} \sigma'(z_j^L)$ or in matrix notations: $\delta^L = \frac{\partial L}{\partial a^L} \circ \sigma'(z^L)$

May cause slow learning if neuron is "saturated" (activation is close to 0 or 1)

Proof: just apply chain rule.

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – formulas:

Equation #2:

$\delta_j^l = \sum_k \omega_{kj}^{l+1} \delta_k^{l+1} \, \sigma'(z_j^l)$ or in matrix notations: $\delta^l = \left(\left(\omega^{l+1} \right)^T \delta^{l+1} \right) \circ \sigma'(z^l)$

Proof:

$$\delta_j^l \equiv \frac{\partial L}{\partial z_j^l} = \sum_k \frac{\partial L}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} = \sum_k \omega_{kj}^{l+1} \sigma'(z_j^l) \delta_k^{l+1}$$

Use definition of z_j^l to make the last transition:

$z^l = \omega^l a^{l-1} + b^l$

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – formulas:

Equation #3:

$\frac{\partial L}{\partial b_j^l} = \delta_j^l$ or in matrix notations: $\frac{\partial L}{\partial b^l} = \delta^l$

Proof: $\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l * 1 = \delta_j^l$

Equation #4:

$\frac{\partial L}{\partial \omega_{jk}^l} = a_k^{l-1}\delta_j^l$ or in matrix notations: $\frac{\partial L}{\partial \omega^l} = a^{l-1}\delta^l$

Proof: $\frac{\partial L}{\partial \omega_{jk}^l} = \frac{\partial L}{\partial z_j^l}\frac{\partial z_j^l}{\partial \omega_{jk}^l} = a_k^{l-1}\delta_j^l$

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – pseudocode 1:

Given: input $\mathbf{x}$

Forward pass:

$z^1 = \omega^1 \mathbf{x} + b^1$ for $l = 1$

$a^1 = \sigma(z^1)$ for $l = 1$

$z^l = \omega^l a^{l-1} + b^l$ for $l > 1$

$a^l = \sigma(z^l)$ for $l > 1$ till $l = L$

C.2 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – pseudocode 2:

Given: input $\mathbf{x}$

Backward pass:

$\delta^L = \frac{\partial L}{\partial a^L} \circ \sigma'(z^L)$ for $l = L$

$\delta^l = \left(\left(\omega^{l+1}\right)^T \delta^{l+1}\right) \circ \sigma'(z^l)$ for $l < L$

Using derivatives calculated above:

$\frac{\partial L}{\partial b^l} = \delta^l$ biases

$\frac{\partial L}{\partial \omega^l} = a^{l-1}\delta^l$ weights

C.3 Training NNs: Backpropagation algorithm

Backpropagation algorithm (BP) – pseudocode 2:

Given: input $\mathbf{x}$

Backward pass:

$$\delta^L = \frac{\partial L}{\partial a^L} \circ \sigma'(z^L) \qquad \text{for } l = L$$

$$\delta^l = \left(\left(\omega^{l+1}\right)^T \delta^{l+1} \right) \circ \sigma'(z^l) \qquad \text{for } l < L$$

Using derivatives calculated above:

$$\frac{\partial L}{\partial b^l} = \delta^l \qquad \text{biases}$$

$$\frac{\partial L}{\partial \omega^l} = a^{l-1} \delta^l \qquad \text{weights}$$

Next week

Training Neural Networks: better and faster

Improving convergence of training process

- Weights initialization
- Loss function
- Regularization
- Advanced GD

D.1 Homework

1. Exercises from presentation.
2. Rewrite equations in “batch” form.
3. Practical assignment, see jupyter notebook.

Refs

1. Thorough review of relevant math topics:

http://info.usherbrooke.ca/hlarochelle/ift725/review.pdf

2*. Ian Goodfellow, Yoshua Bengio and Aaron Courville, Deep Learning.

3. Kevin P. Murphy, Machine Learning: A probabilistic perspective.

4. David Barber, Bayesian Reasoning and Machine Learning.

5. Sergios Theodoridis, Machine Learning: A Bayesian and optimization perspective.

6*. See also refs in practical assignment and online courses, especially one from Hugo Larochelle (presentation from lecture 1).