

Artificial Neural Networks: Lecture 4

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Regularization and Tricks of the Trade in deep networks

Objectives for today:

- Bagging
- Dropout
- What are good units for hidden layers?
- Rectified linear unit (RELU)
- Shifted exponential linear (ELU and SELU)
- BackProp: Initialization
- Linearity problem, vanishing gradient problem, bias problem
- Batch normalization

Reading for this lecture:

Goodfellow et al., 2016 *Deep Learning*

- Ch 7.4, 7.8, 7.11 and 7.12,
- **Ch. 8.4**

Further Reading for this Lecture:

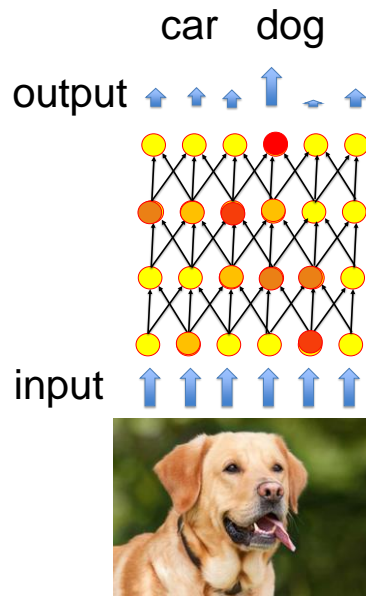
Paper: Klaumbauer, ..., Hochreiter (2017)

Self-normalizing neural networks

<https://arxiv.org/pdf/1706.02515.pdf>

review: Artificial Neural Networks for classification

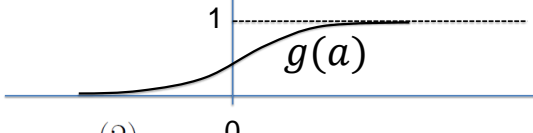
Aim of learning:
Adjust connections such
that output is correct
(for each input image,
even new ones)



Previous slide.

We use an artificial neural network, with multiple layers. This week we will address three important questions.

Review: Multilayer Perceptron



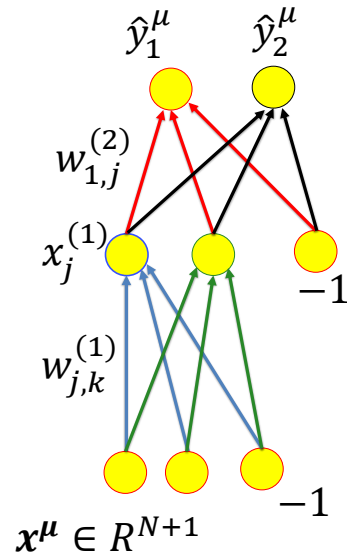
$$\hat{y}_i^\mu = x_i^{(2)} \quad (1)$$

$$= g^{(2)}[a_i^{(2)}] \quad (2)$$

$$= g^{(2)}\left[\sum_j w_{ij}^{(2)} x_j^{(1)}\right] \quad (3)$$

$$= g^{(2)}\left[\sum_j w_{ij}^{(2)} g^{(1)}(a_j^{(1)})\right] \quad (4)$$

$$= g^{(2)}\left[\sum_j w_{ij}^{(2)} g^{(1)}\left(\sum_k w_{jk}^{(1)} x_k^\mu\right)\right] \quad (5)$$



Previous slide.

In each layer, neurons perform a nonlinear transform $g(a)$.

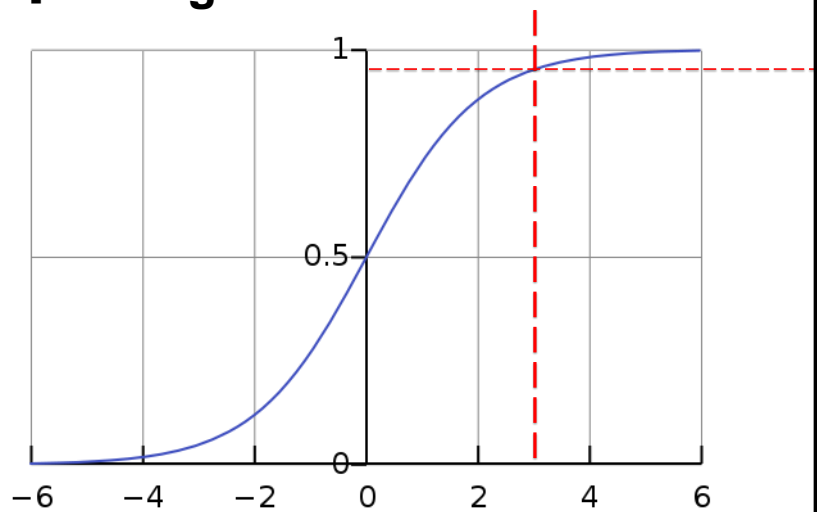
Review. sigmoidal output = logistic function

$$g(a) = \frac{1}{1 + e^{-a}}$$

Rule of thumb:

for $a = 3$: $g(3) = 0.95$

for $a = -3$: $g(-3) = 0.05$



https://en.wikipedia.org/wiki/Logistic_function

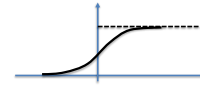
Previous slide.

Based on systematic probabilistic arguments, we have concluded that in the output layer, a good choice is the sigmoidal (for single outputs or multiple attributes) or the softmax function (for exclusive multi-class output).

Review: Modern Neural Networks

output layer

use sigmoidal unit (single-class)
or softmax (exclusive multi-class)



hidden layer

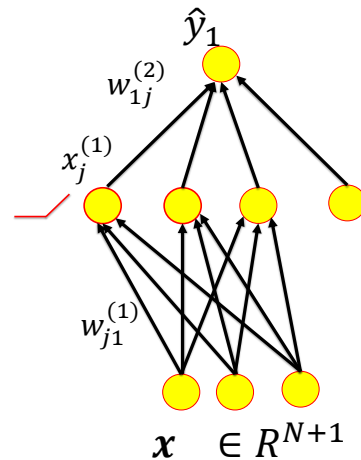
use rectified linear unit in $N+1$ dim.

Why?

$$f(x) = x \text{ for } x > 0$$

$$f(x) = 0 \text{ for } x < 0 \text{ or } x = 0$$

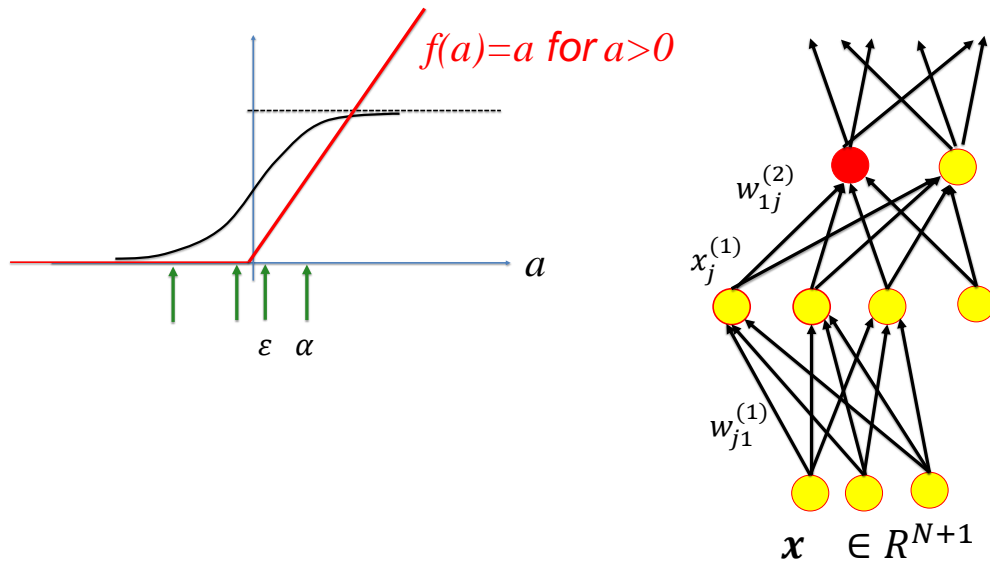
Better choices?



Previous slide.

Why we should use in the hidden layer a rectified linear function is less obvious.

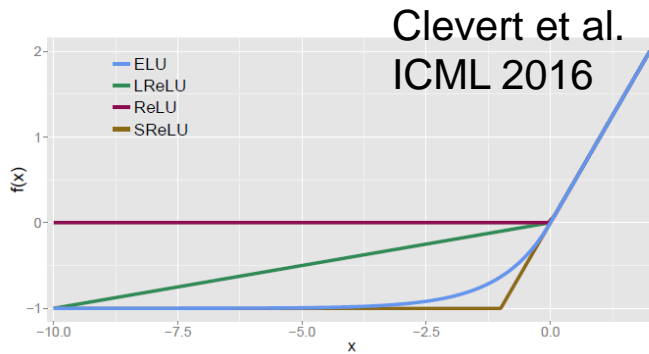
Rectified Linear (RELU) vs. Sigmoidal



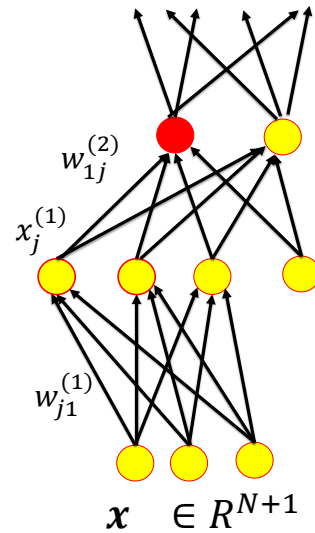
Previous slide.

Indeed, there are other choices. We could also use a sigmoidal unit in the hidden layer.

Exponential Linear (ELU) vs. Sigmoidal



Shifted ReLU (SReLU)
Leaky ReLU (LReLU)



Previous slide.

To complete the picture, we can also consider a Shifted Rectified Linear Unit (SReLU)
Or the
piecewise linear with positive slope for $x < 0$, the Leaky Rectified Linear Unit (LReLU).

Question 1 for this week:

What are good models for hidden neurons?

... and why?

Previous slide.

The question will be addressed in part 4, today, starting with slides 91.

To answer this question, we will look at the BackProp algorithm and focus on values

$$x = \pm \varepsilon$$

where epsilon is a small number. But also at values

$$x = \pm \alpha$$

with alpha of order one.

0. Initialization of weights	BackProp
1. Choose pattern x^μ input $x_k^{(0)} = x_k^\mu$	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="margin-bottom: 10px;">output activity</div> <div style="margin-bottom: 10px;">↑</div> <div style="margin-top: 10px;">input pattern</div> </div>
2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$ $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \quad (1)$ output $\hat{y}_i^\mu = x_i^{(n_{\max})}$	
3. Computation of errors in output $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu] \quad (2)$	
4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$ $\delta_j^{(n-1)} = g'^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)} \quad (3)$	
5. Update weights (for each (i, j) and all layers (n)) $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)} \quad (4)$	
6. Return to step 1.	

Previous slide.

In week 2, we have studied the BackProp algorithm with forward

<p>0. Initialization of weights</p> <p>1. Choose pattern x^μ</p> <p style="padding-left: 40px;">input $x_k^{(0)} = x_k^\mu$</p> <p>2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$</p> $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \quad (1)$ <p style="padding-left: 40px;">output $\hat{y}_i^\mu = x_i^{(n_{\max})}$</p> <p>3. Computation of errors in output</p> $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu] \quad (2)$ <div style="border: 2px solid red; padding: 5px;"> <p>4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$</p> $\delta_j^{(n-1)} = g'^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)} \quad (3)$ </div> <p>5. Update weights (for each (i, j) and all layers (n))</p> $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)} \quad (4)$ <p>6. Return to step 1.</p>	<h2 style="color: red;">BackProp</h2> <h3 style="color: red;">Calculate output error</h3>
---	---

Previous slide.

... and backward pass.

<p>0. Initialization of weights</p> <p>1. Choose pattern x^μ input $x_k^{(0)} = x_k^\mu$</p> <p>2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$ $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)})$ (1) output $\hat{y}_i^\mu = x_i^{(n_{\max})}$</p> <p>3. Computation of errors in output $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu]$ (2)</p> <p>4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$ $\delta_j^{(n-1)} = g'^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)}$ (3)</p> <p>5. Update weights (for each (i, j) and all layers (n)) $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)}$ (4)</p> <p>6. Return to step 1.</p>	<h2>BackProp</h2> <p>update all weights</p> $\Delta w_{i,j}^{(n)} = \delta_i^{(n)} x_j^{(n-1)}$
--	---

Previous slide.

We emphasized the update of the weights. But so far we did not yet discuss how the weights are initialized. Why does initialization (or normalization) matter in Backprop?

Question 2 for this week:

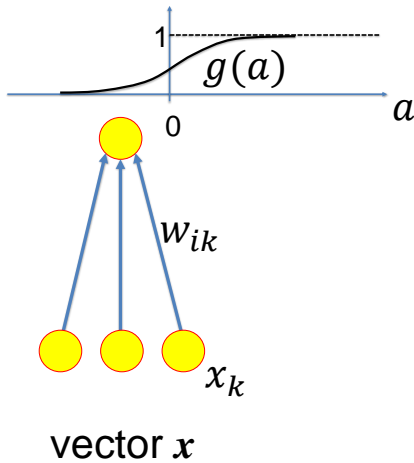
Why does the initialization or normalization matter in backprop?

Previous slide.

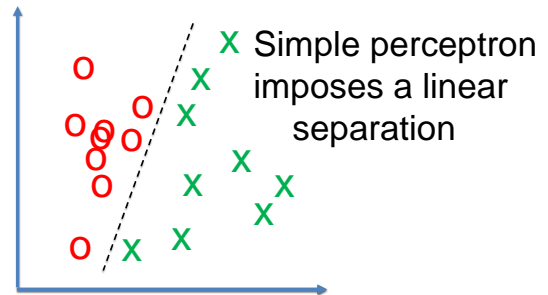
This question will also be addressed in part 4, starting with slide 91.

Review: Single-Layer networks/simple perceptron

$$\hat{y} = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$



$$d(\mathbf{x}) = \sum_k w_k x_k - \vartheta = 0$$



Previous slide.

In the context of generalization, we have seen that a simple perceptron can only solve linearly separable problems

Review: Classification as a geometric problem



Previous slide.

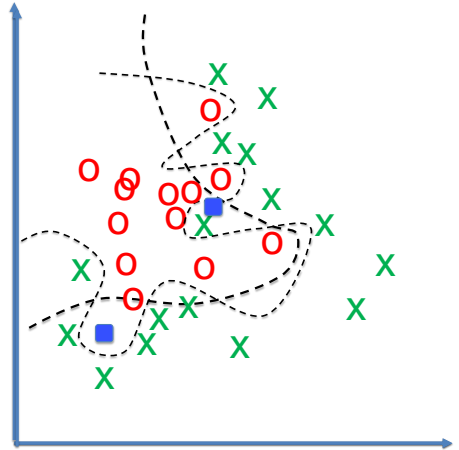
Whereas a multilayer perceptron is flexible enough to solve complex classification problems

Review: The problem of overfitting

Big Multilayer perceptrons are flexible and can be trained by BackProp to minimize classification error

... but is flexibility always good?

Network has to work on future data:
test data base



Previous slide.

But flexibility can lead to overfitting, unless we use a proper regularization method.

Question 3 for this week:

What are good models for regularization?

... and why?

We start with this question!

Previous slide.

We have already seen two powerful regularization methods, early stopping and L2 (or L2) norm penalty on the weights, but there are other regularization methods that are widely used in applications of neural networks.

The question of additional regularization method will be addressed in part 1 today, starting now

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner
EPFL, Lausanne, Switzerland

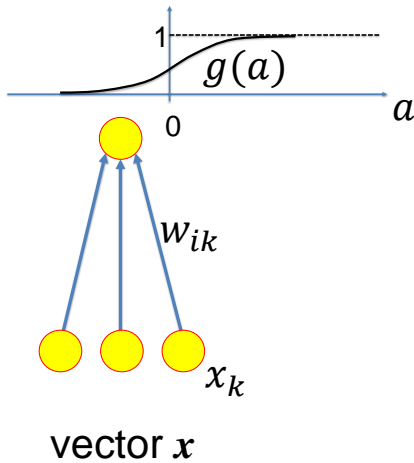
1. Bagging

Previous slide.

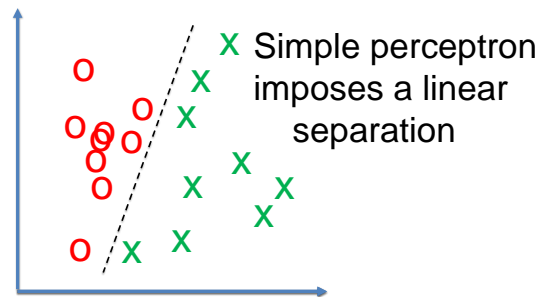
Bagging is a regularization method, that we will discuss now.

1. Bagging Example: simple perceptron

$$\hat{y} = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$



$$d(x) = \sum_k w_k x_k - \vartheta = 0$$



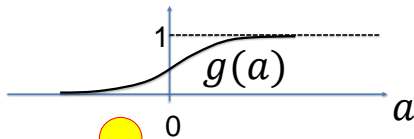
Previous slide.

To introduce bagging, we start with the simple perceptron as an example.

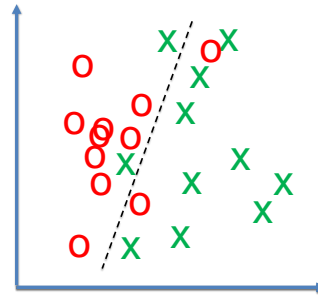
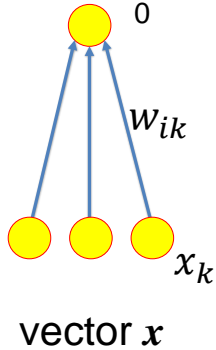
The simple perceptron imposes a linear separation of positive and negative examples.

1. Bagging Example: simple perceptron for noisy data

$$\hat{y} = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$



Find best (approximate) linear separation



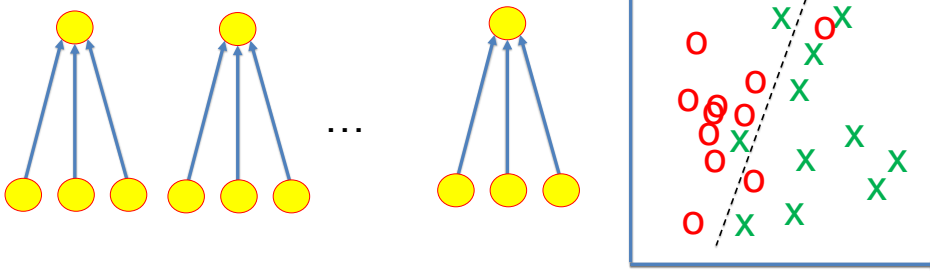
Previous slide.

In the following we work with noisy data and use a sigmoidal in the output.

1. Bagging Idea: (i) Repeat variants of your model K times

$$\hat{y} = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$

Find best (approximate) linear separation



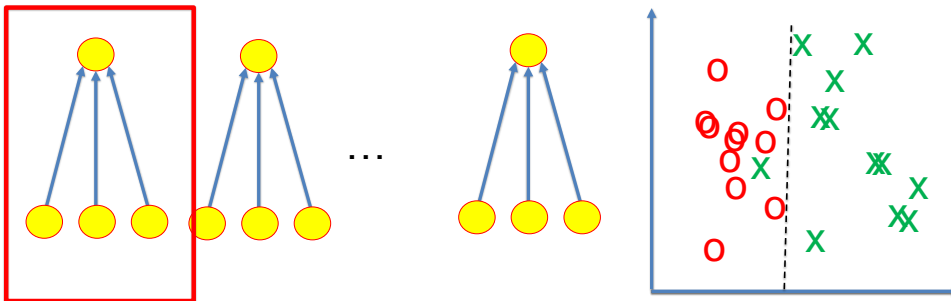
Previous slide.

We work with K repetitions of the simple perceptron

1. Bagging Idea: (ii) Each Variant sees different subsets of data

$$\hat{y}_1 = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$

Find best (approximate) linear separation



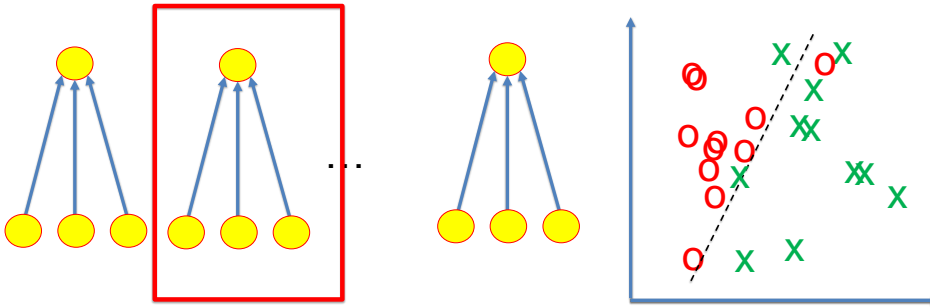
Previous slide.

... where each variant (i.e. each copy of the simple perceptron) is optimized for a different subset of the data; from the first variant

1. Bagging Idea: (ii) Each Variant sees different subsets of data

$$\hat{y}_2 = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$

Find best (approximate) linear separation



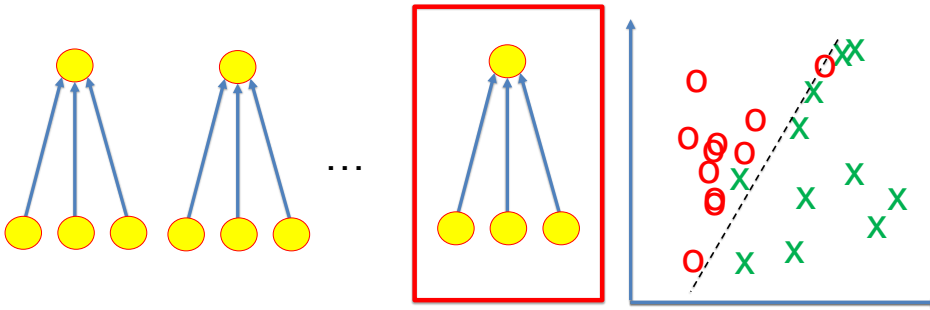
Previous slide.

... or the second one

1. Bagging Idea: (ii) Each Variant sees different subsets of data

$$\hat{y}_K = 0.5[1 + \tanh(\sum_k w_k x_k - \vartheta)]$$

Find best (approximate) linear separation

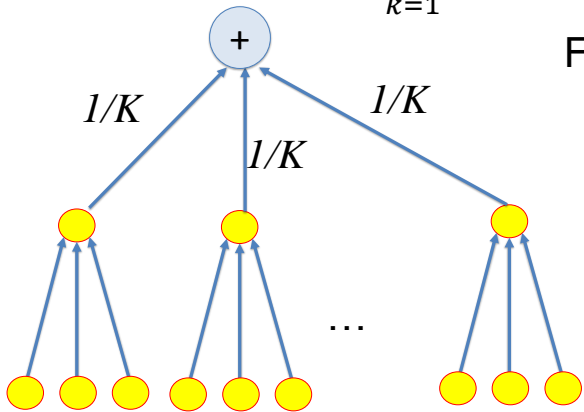


Previous slide.

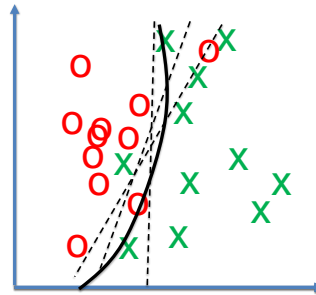
... or the last one.

1. Bagging Idea: (iii) Average over all K variants

$$\hat{y}_{bag} = \frac{1}{K} \sum_{k=1}^K \hat{y}_k$$



Find average (nonlinear) separation



Previous slide.

Rather than looking for a single copy of the simple perceptron that would be the 'best' in some sense, we take all K copies in parallel and average their outputs.

1. Bagging : Algorithm

Given: Training data set $\{ (x^\mu, t^\mu) , \quad 1 \leq \mu \leq P1 \}$;

1 Generate K different training sets

for $k=1, \dots, K$

pick $P1$ times into your data set with replacement

(you can pick the same data point several times)

2 Initialize K different variants of your model

3 Train model k on data set k up to criterion

4 For a future data point (test set)

for $k=1, \dots, K$

put input x into model k , read out \hat{y}_k

5 Report average $\hat{y}_{bag} = \frac{1}{K} \sum_{k=1}^K \hat{y}_k$

Previous slide.

Pseudoalgorithm for bagging.

Steps 1-3 describe training.

Steps 4-5 describe testing (or final application).

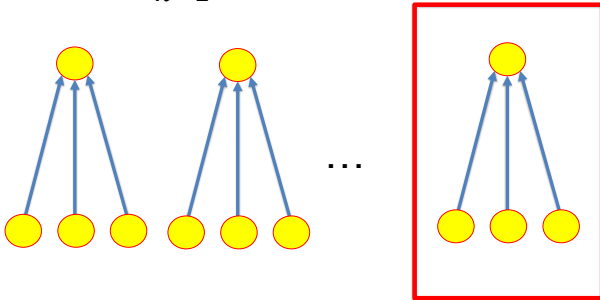
1. Bagging: Theory

Model k

$$\hat{y}_k = 0.5[1 + \tanh(\sum_j w_j x_j - \vartheta)]$$

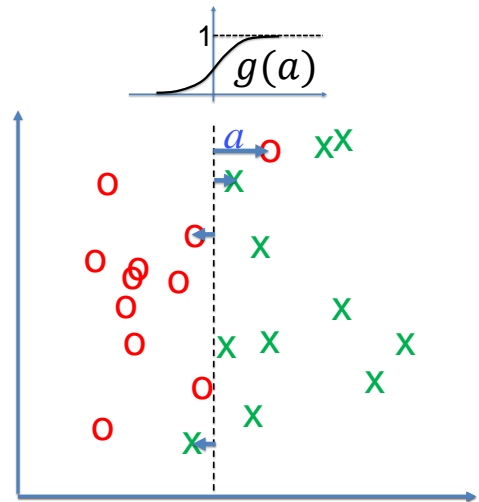
Bagged output

$$\hat{y}_{\text{bag}} = \frac{1}{K} \sum_{k=1}^K \hat{y}_k$$



Blackboard: Bagging

$$\delta_k^\mu = t_k^\mu - \hat{y}_k^\mu = \sigma(a)$$



Previous slide.

Bagging is supported by a theoretical analysis.

Suppose the actual output of copy k of the model is \hat{y}_k^μ while the target output is t_k^μ (either zero or one)

We introduce the signed difference $\delta_k^\mu = t_k^\mu - \hat{y}_k^\mu = \sigma(a)$

which is some function of the distance a of the data point from the separating hyperplane. Toward the end of learning δ_k^μ will be small, but can be positive or negative.

We are interested in the quadratic error in the output of copy k : $E_k = \frac{1}{P} \sum_{\mu=1}^P [\delta_k^\mu]^2$

We compare this error with the quadratic error E_{bag} of the total 'bagged' output

$$\hat{y}^\mu = \frac{1}{K} \sum_{k=1}^K \hat{y}_k^\mu$$

1. Bagging : Theory

Blackboard: Bagging

Claim: bagged output has smaller quadratic error than a typical individual model

$$\text{bagged output } \hat{y}_{\text{bag}} = \frac{1}{K} \sum_{k=1}^K \hat{y}_k$$

Previous slide.

1. Bagging : Result

assumption: the average delta-difference, defined as

$$\frac{1}{P} \sum_{\mu=1}^P [\delta_k^{\mu}] = d$$

is the same for all K copies of the model.

THEN

- bagged output has smaller quadratic error than a typical individual model
- if all K individual models are uncorrelated, the gain in performance scales as 1/K

Previous slide.

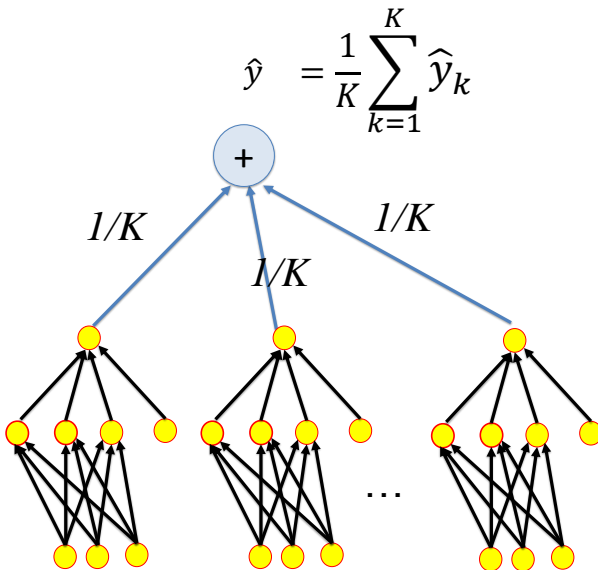
Thus, using a bagged output is always better than using the output of a single model.

NOTE: the assumption is rather natural. If all K models are trained with the same learning algorithm, same error function, and same regularization, there is no reason that the average delta-difference would be bigger for one model than the other, if the average is over many data points (apart from statistical fluctuations).

NOTE: with a suitable error function, the average delta-difference might even be zero.

NOTE: the assumption is nevertheless a bit special because we say that the **average** delta-distance should be identical for all copies of the model --- as opposed to the **average squared-delta distance**.

1. Bagging: each of the models can be a deep network

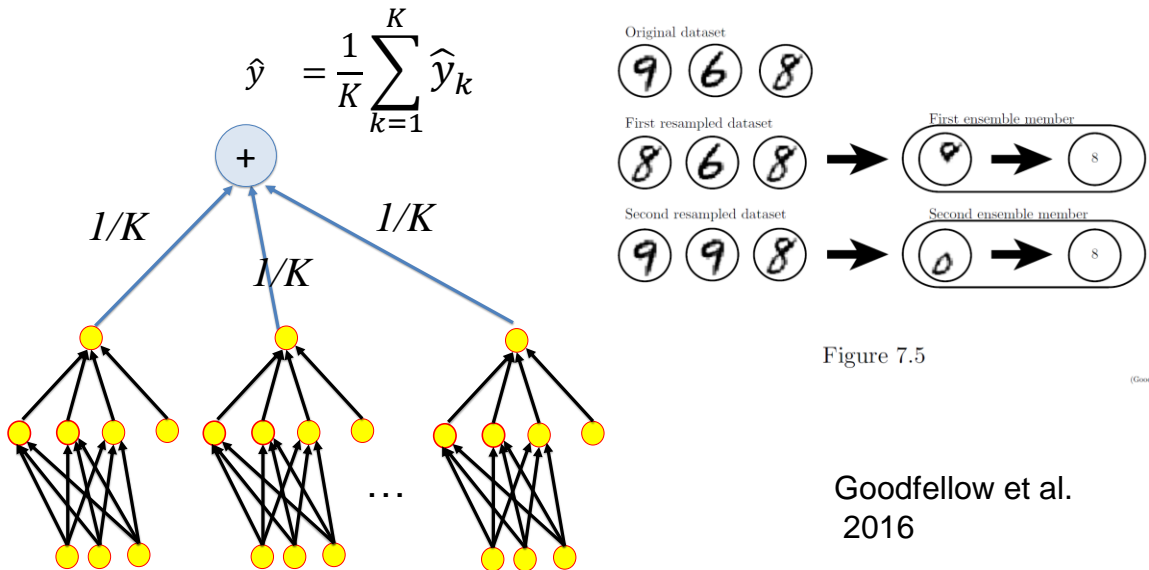


Previous slide.

Bagging does not only work for simple perceptrons, but also for multi-layer neural networks. You simply need to train the networks separately and then average their outputs.

Note that averaging over the output is identical to adding an additional linear output neuron on top of the existing networks, so that instead of K copies of a smaller network we can also view it as a single larger network.

1. Bagging: each of the models sees a different data set



Previous slide.

As an illustration of bagging, Goodfellow et al. give the following example.

The task is to build a detector for eights, '8'.

One member of the ensemble (i.e., one copy of the network) is exposed of a data set which contains many sixes and eights (plus possibly a few nines). It therefore learns to build a detector that mainly focuses on the upper half of the input images.

Another copy of the network is exposed to a data based which contains many many nines as well as a eights (and also possibly a few sixes). It therefore learns to build a detector that mainly focuses on the lower half of the input images.

Once you average of the results of different copies of the network, you get a better detector of eights, than any single network alone.

Quiz:

- If you want to win a machine learning competition, it is better to average the prediction on new data over ten different models, rather than just using the model that is best on your validation data.
- If you want to win a machine learning competition, it is better to hand in 10 contributions (using different author names) rather than a single contribution

Your notes.

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner

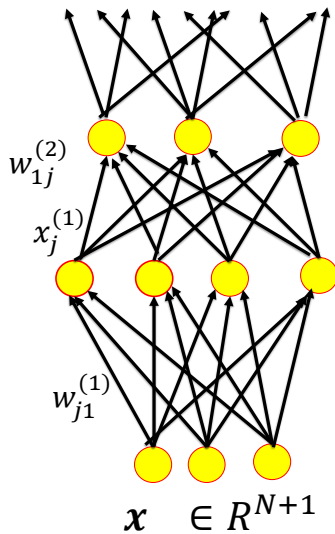
EPFL, Lausanne, Switzerland

1. Bagging
2. Dropout

Previous slide.

Dropout is a regularization method that has been specifically developed for neural networks. It is very loosely related to bagging.

2. Dropout: suppress 50 percent of hidden units during training

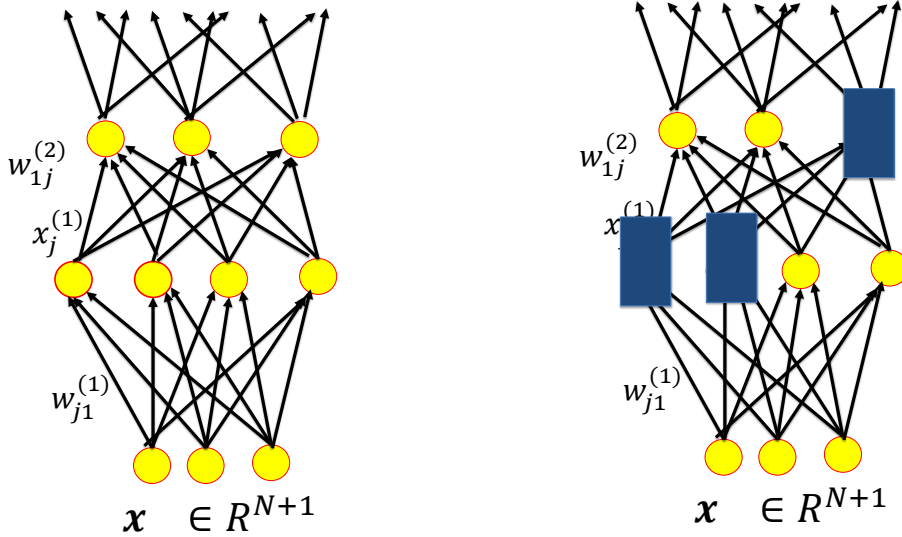


Previous slide.

Remember that in all cases where we want to use regularization, we start with a network that is too flexible (too many neurons and layers) so that we would see overfitting without regularization.

We therefore start with a big and flexible network. During training, you randomly suppress, for each input pattern, 50 percent of the hidden units.

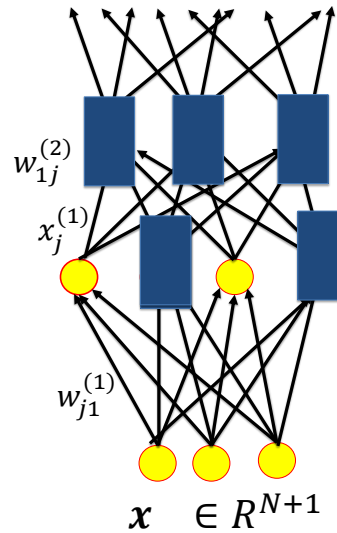
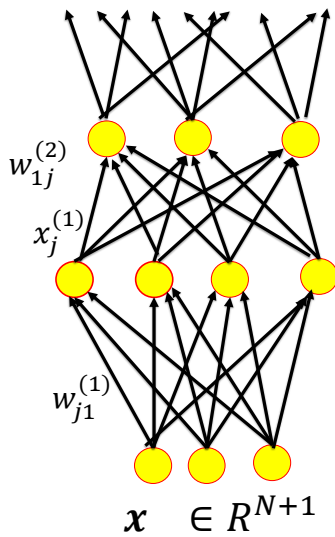
2. Dropout: suppress 50 percent of hidden units during training



Previous slide.

Thus for pattern number μ you randomly pick a subset of hidden units which you remove (their outputs are set to zero).

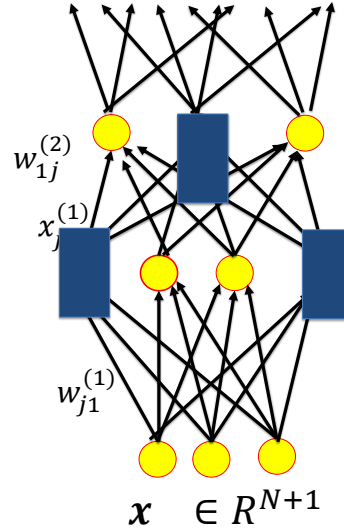
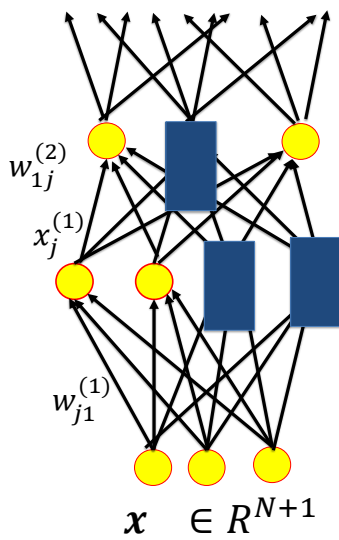
2. Dropout: suppress 50 percent of hidden units during training



Previous slide.

And for pattern number $\mu+1, \mu+1, \square$ you randomly pick each time a different subset of hidden units which you remove (their outputs are set to zero).

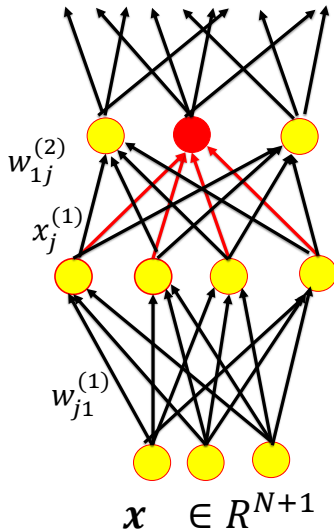
2. Dropout: suppress 50 percent of hidden units during training



Previous slide.

You train over many epochs.

2. Dropout: use full network for validation and test



For test:

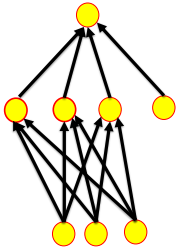
- full network
 - but multiply output weights from hidden units by $1/2$
- Total input to each unit is roughly same as during training

Previous slide.

For testing you use the full network with all hidden units. However, since there are now twice as many hidden units as during training, you need to multiply the output weights by factor $1/2$, so that the typical input to a unit in the next layers is roughly the same as during training.

2. Dropout: two different interpretations

1. An approximate, but practical implementation of bagging
2. A tool to enforce representation sharing in the hidden neurons



Previous slide.

Dropout is an effective regularization method. There are two different interpretations of why dropout works.

2. Dropout as approximate bagging

Dropout can be seen as a practical application of the ideas of bagging to deep networks

Differences to standard bagging:

- not a fixed data base for each 'dropout' configuration
- models not independent: share weights
- output not a 'sum over model outputs'

Previous slide.

The first interpretation sees dropout as a practical implementation of the ideas of bagging to deep networks.

Note that dropout implements ideas of bagging not just for the output layer, but also for neurons in the hidden layer.

The main differences to standard bagging are:

1. **not a fixed data base for each 'dropout' configuration.**

In a network with N_h hidden neurons, there are $\binom{N_h}{N_h/2}$ different dropout configurations.

If the same configuration reappears, it will be trained with a different input pattern.

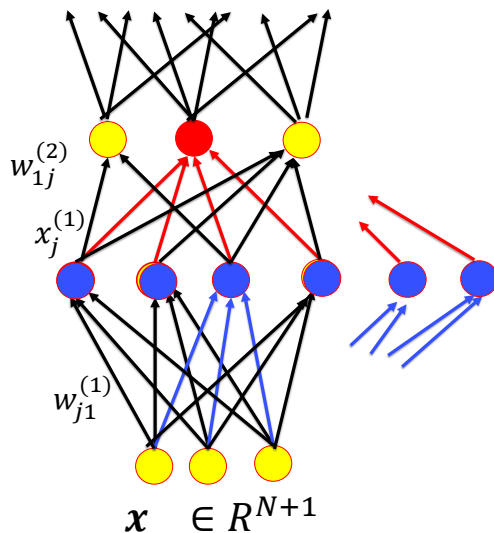
2. **models not independent, because they share weights.**

In bagging, models are first trained independently and only combined at the end. Here, each pair of configurations shares half the neurons.

3. **output not a 'sum over model outputs'**

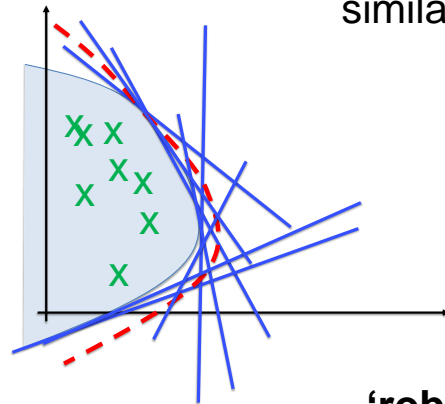
In dropout, the output can be a sigmoidal unit.

2. Dropout as forced feature sharing



Feature sharing:

Take 2 times as many neurons,
But make sure they all solve
similar tasks



'robust'

Previous slide.

The second interpretation is: Dropout is a tool to enforce representation sharing in the hidden neurons.

To understand this statement, let us focus on the red neuron somewhere inside the network. It receives inputs from the four blue hidden neurons one layer below. Each of the blue neurons represents a hyperplane in input space (or more generally: in the space of the previous hidden layer).

The red neuron takes a weighted average over the output of the blue neurons which corresponds to a non-linear separation in the input space as indicated by the dashed red line.

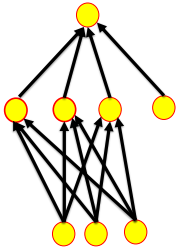
Suppose now that we add another four blue neurons in the first hidden layer.

Dropout forces them to learn very similar separating hyperplanes: for example we add two neurons, but remove at the same time two of the old ones. The two new ones will take over the role of those that they have to replace, but they might implement slightly different hyperplanes. Hyperplanes can be interpreted as features.

In the end, the set of eight neurons will share features, by implementing similar hyperplanes.

2. Dropout: two different interpretations

1. An approximate, but practical implementation of bagging
2. A tool to enforce representation sharing in the hidden neurons



→ useful regularization method,
→ simple to implement

Previous slide.

In practice, dropout is a useful regularization method because it is simple to implement.

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

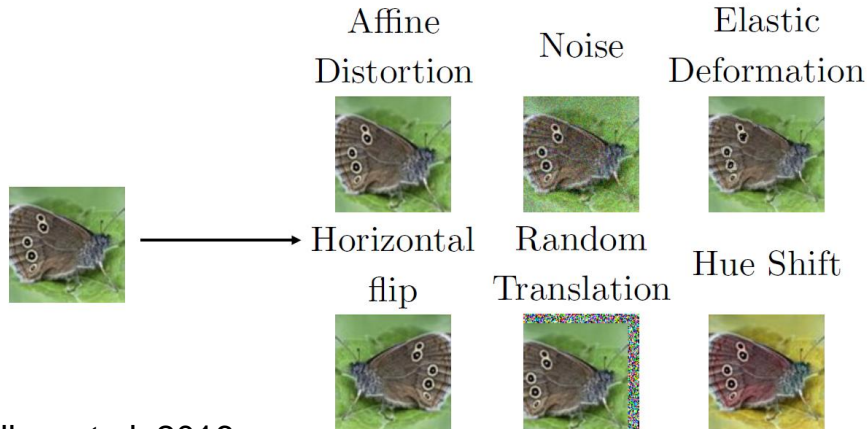
1. Bagging
2. Dropout
3. Other simple regularization methods

Previous slide.

Just to complete the picture, we need to discuss a few other simple regularization methods.

3. Other easy regularization methods: dataset augmentation

Dataset Augmentation



Goodfellow et al. 2016

Previous slide.

Dataset augmentation is a simple regularization method. You start with a dataset of P data points.

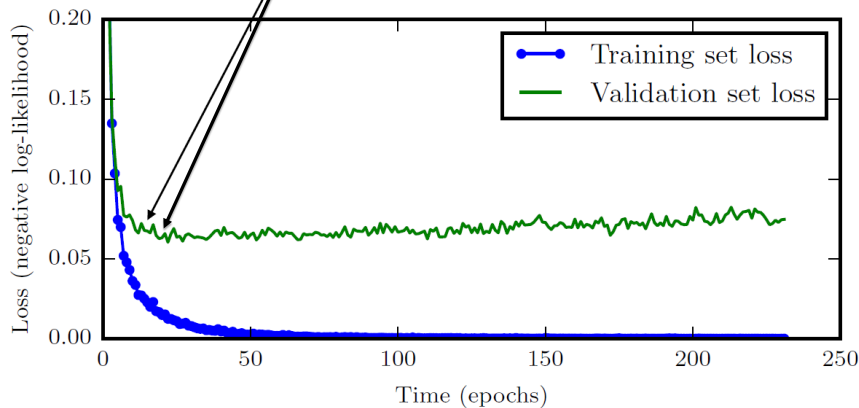
For each data point you apply a few transformations. For the case of images, these are:

1. An image is laterally, vertically, or diagonally shifted (you need to fill in the background to do so). The new images are added to the data base (with the same label)
2. An image is flipped. The new image is added to the data base (the the same label).
3. You add pixel noise (white or locally correlated). The new images are added to the data base (with the same label).
4. You apply one or several elastic deformations. The new images are added to the data base (with the same label).
5. You slightly shift the color scheme. The new images are added to the data base (with the same label).

Thus, a single image gives rise to twenty or more images. The transformations must correspond to the known invariances: a butterfly remains a butterfly if it is shifted, if the background illumination changes, if its shape changes slightly, etc.

3. Other easy regularization methods: early stopping

Go back to **weights** where validation error was minimal



Example: MNIST data base, see Goodfellow et al. 2016

Previous slide.

Early stopping is also a regularization method that is easy to implement. Note that you have to continue for a LONG time, before you can go back to the best weights. This is necessary because the validation loss could make (together with the training loss) a big jump downward a long time after having passed through a first minimum.

This is not the case for the MNIST data base shown here because the training loss is already practically zero.

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner

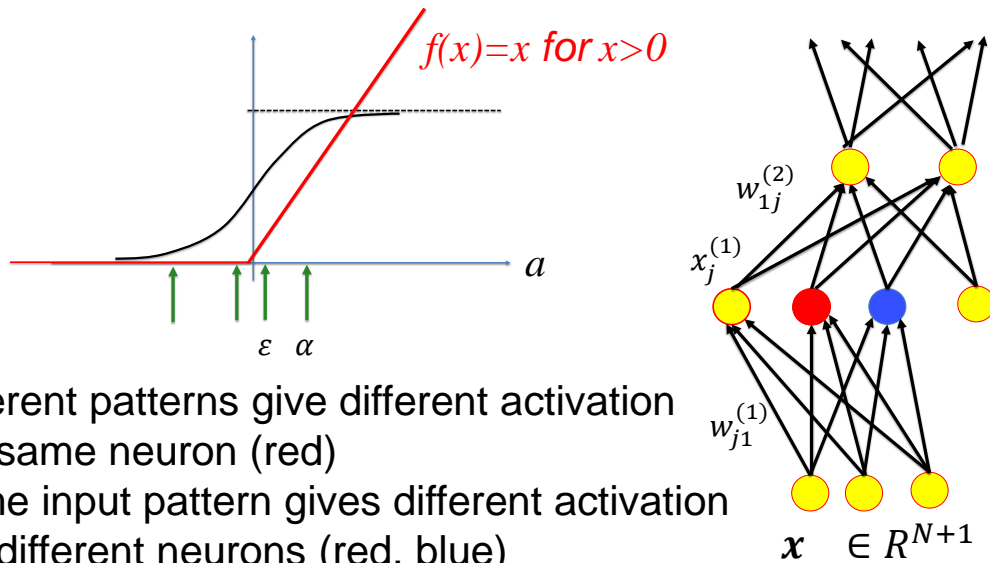
EPFL, Lausanne, Switzerland

1. Bagging
2. Dropout
3. Other simple regularization methods
4. Weight initialization and choice of hidden units

Previous slide.

We now focus on the hidden neurons.

4. Choice of units



- different patterns give different activation of same neuron (red)
- same input pattern gives different activation of different neurons (red, blue)

Previous slide.

Let us focus on the red neuron in one of the hidden layers.

If I apply pattern μ , the total activation a of the red neuron might be α .

If I apply pattern $\mu+1$, the total activation a of the red neuron might be $-\epsilon$.

If I apply pattern $\mu+2$, the total activation a of the red neuron might be $+\epsilon$.

Etc.

Thus different patterns cause different activation values of same neuron (red)

On the other hand,

If I apply pattern μ , the total activation a of the red neuron might be α ,

and the total activation a of the blue neuron might be -2α .

Etc.

Thus the same patterns causes different activation values for different neuron.

Let us keep this in mind for the following discussions.

4. Initialization (input layer) Blackboard

Normalization of data base:

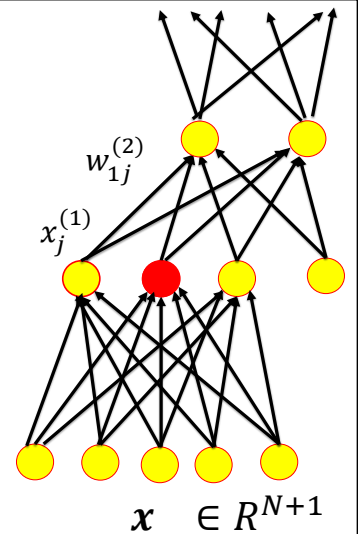
$$(1) \langle x_j \rangle = \frac{1}{P} \sum_{\mu=1}^P x_j^{\mu} = 0$$

Random initialization of weights:

$$(2) \langle w_{ij}^{(n)} \rangle = 0$$

How should you choose the variance?

Claim: square root of N is important



Previous slide.

Let us now focus on a single neuron (red), and look at different input patterns.

We suppose that patterns in the data base have been pre-treated in a normalization step so as to ensure that for each component (e.g. each pixel) the empirical mean across all patterns is zero.

$$\langle x_j \rangle = \frac{1}{P} \sum_{\mu=1}^P x_j^{\mu} = 0$$

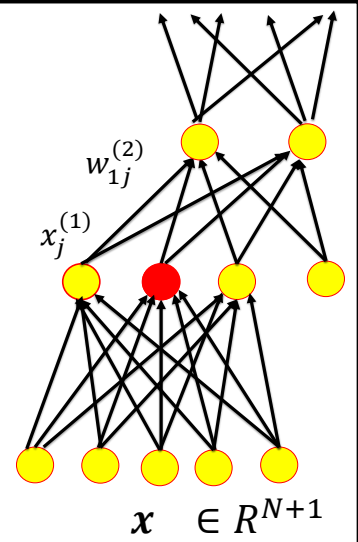
We will initialize the weights by drawing weight values randomly and independently from a Gaussian distribution with mean zero, so that the expectation value is:

$$\langle w_{ij}^{(n)} \rangle = 0$$

We ask the question: how should we choose the variance of the initial weight distribution?

Blackboard: Initialization

Claim: square root of N is important



Your notes.

4. Initialization (input layer)

Normalization of data base:

$$(1) \langle x_j \rangle = \frac{1}{P} \sum_{\mu=1}^P x_j^{\mu} = 0$$

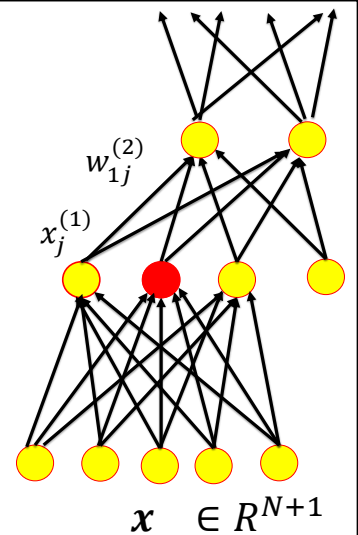
Random initialization of weights:

$$(2) \langle w_{ij}^{(1)} \rangle = 0$$

And standard deviation propto $1/\sqrt{N}$

→ Distribution of $x_j^{(1)}$ in layer 1

→ Distribution of $x_j^{(k)}$ in layer k



Previous slide.

Appropriate random initialization of the input weights (layer 1), gives an expected activation

$$\langle a_i^{(1)} \rangle = \frac{1}{P} \sum_{\mu=1}^P w_{ij}^{(1)} x_j^{\mu} = 0$$

and a standard deviation

$$\sqrt{\langle [a_i^{(1)}]^2 \rangle} = 2$$

As a result we will have a suitable distribution of values $x_j^{(1)}$ in layer 1.

Random initialization of weights in layer 2, gives a distribution of activation $a_j^{(2)}$ in layer 2, which in turn are transformed into a distribution of values $x_j^{(2)}$ in layer 2; and this process continues (see Exercises this week).

0. Initialization of weights	BackProp
1. Choose pattern x^μ input $x_k^{(0)} = x_k^\mu$	
2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$ $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \quad (1)$ output $\hat{y}_i^\mu = x_i^{(n_{\max})}$	
3. Computation of errors in output $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu] \quad (2)$	
4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$ $\delta_j^{(n-1)} = g^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)} \quad (3)$	
5. Update weights (for each (i, j) and all layers (n)) $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)} \quad (4)$	
6. Return to step 1.	

Previous slide.

In the forward pass, we need to evaluate

$$x_j^{(n)} = g\left[\sum_k w_{j,k}^{(n)} x_k^{(n-1)}\right]$$

Now we can use the same argument as previously used for the input layer. For neuron j in layer n , the value $x_j^{(n)}$ will depend on the pattern so that we have a distribution of values across different patterns.

<p>0. Initialization of weights</p> <p>1. Choose pattern x^μ</p> <p style="padding-left: 40px;">input $x_k^{(0)} = x_k^\mu$</p> <p>2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$</p> $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \quad (1)$ <p style="padding-left: 40px;">output $\hat{y}_i^\mu = x_i^{(n_{\max})}$</p> <p>3. Computation of errors in output</p> $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu] \quad (2)$ <div style="border: 2px solid red; padding: 5px;"> <p>4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$</p> $\delta_j^{(n-1)} = g^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)} \quad (3)$ </div> <p>5. Update weights (for each (i, j) and all layers (n))</p> $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)} \quad (4)$ <p>6. Return to step 1.</p>	<h2 style="color: red;">BackProp</h2> <h3 style="color: red;">Calculate output error</h3>
--	---

Previous slide.

In the backward pass, we need to evaluate

$$g' \left[a_j^{(n)} \right] = g' \left[\sum_k w_{j,k}^{(n)} x_k^{(n-1)} \right]$$

<p>0. Initialization of weights</p> <p>1. Choose pattern x^μ input $x_k^{(0)} = x_k^\mu$</p> <p>2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$ $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)})$ (1) output $\hat{y}_i^\mu = x_i^{(n_{\max})}$</p> <p>3. Computation of errors in output $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu]$ (2)</p> <p>4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$ $\delta_j^{(n-1)} = g'^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)}$ (3)</p> <p>5. Update weights (for each (i, j) and all layers (n)) $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)}$ (4)</p> <p>6. Return to step 1.</p>	<h2>BackProp</h2>	<p>update all weights</p> $\Delta w_{i,j}^{(n)} = \delta_i^{(n)} x_j^{(n-1)}$
--	-------------------	---

Previous slide.

Before we finally update the weights.

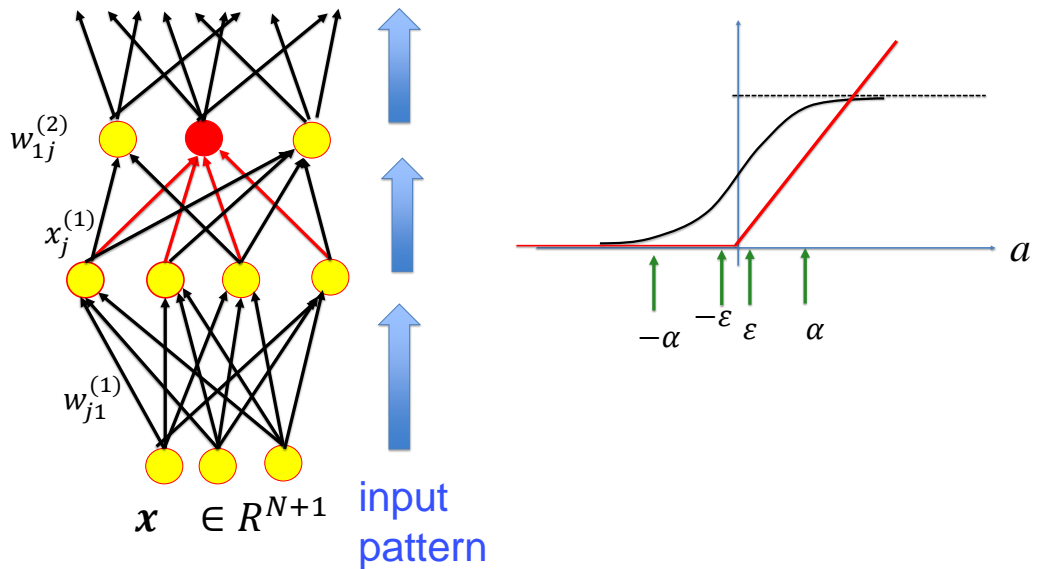
Why does the initialization or normalization matter in backprop?

Previous slide.

So why is the initialization of the weights so important?

Analogously, why is the normalization of the weights so important?

4. Forward pass: Linear and nonlinear processing



Previous slide.

As we have seen,

If I apply pattern μ , the total activation a of the red neuron might be α .

If I apply pattern $\mu+1$, the total activation a of the red neuron might be $-\epsilon$.

If I apply pattern $\mu+2$, the total activation a of the red neuron might be $+\epsilon$.

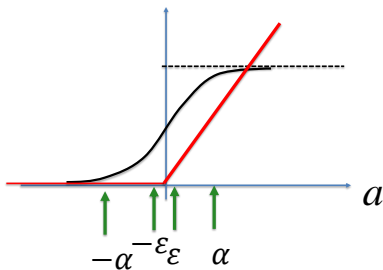
Etc.

Thus different patterns cause different activation values of same neuron (red)

4. Forward pass: Linear and nonlinear processing

Observations:

if all patterns in all layers touch the linear regime of $g(a)$, then the whole network is linear
 → different patterns should touch different regions of $g(a)$.



- this is automatically true for ReLu, if the mean (across patterns) is $a=0$
- this is automatically true for sigmoids, if the variance (across patterns) is > 2

Previous slide.

Suppose that we work with the **sigmoidal unit (black)**

If all the patterns cause activations in the range $[-\epsilon, \epsilon]$, then all the patterns fall in the linear regime of the gain function g .

Suppose that we work with the **ReLU (red)**.

If all the patterns cause activations in the range $[\epsilon, \alpha]$, then all the patterns fall in the linear regime of the gain function g .

In both cases, the result is that this neuron implements a linear transformation (because its nonlinearity is not exploited). However, **a multi-layer network of linear units can be replaced by a single layer of linear units**. Therefore the additional layers are useless.

To exploit the nonlinearities of the neurons, we have to make sure that:

- For sigmoids, some of the input patterns cause activations $|a| > 2$.
- For ReLu's, some of the input patterns cause positive a , others negative a .

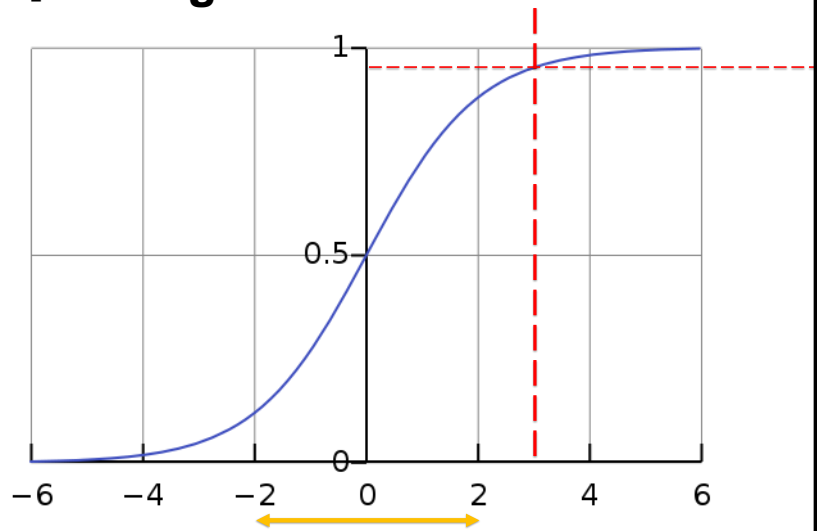
Review. sigmoidal output = logistic function

$$g(a) = \frac{1}{1 + e^{-a}}$$

Rule of thumb:

for $a = 3$: $g(3) = 0.95$

for $a = -3$: $g(-3) = 0.05$



https://en.wikipedia.org/wiki/Logistic_function

Previous slide.

Note that a sigmoidal unit is strongly nonlinear in the regime $|a| = 2$.

4. Forward pass: exploit nonlinearities ('linearity problem')

To **exploit nonlinearities** of all units in the network, we must

1. Make sure that the **initialization** of weights is well chosen
→ expectation (across patterns) of the activation variable

$$0 = \langle a_j^{(n)} \rangle; a_j^{(n)} = \sum_k w_{j,k}^{(n)} x_k^{(n-1)}$$

- standard deviation of the activation variable

$$a_j^{(n)} \text{ of order } 1.$$

2. Make sure that **weight updates** do not shift mean (and standard deviation) of distribution too much

Previous slide.

A multilayer network in the linear regime acts like a linear network ('linearity problem')

To exploit nonlinearities of all neurons in the network, we have to make sure that

- The initial choice of the weights is such that each unit has a range of activation values (across different patterns) that touch the nonlinear regime.

- During training the weights remain in a regime such that each unit has a range of activation values (across different patterns) that touch the nonlinear regime.

Note:

- 1) for **ReLU's** the only nonlinearity is at zero. Thus, if the mean activation (across all patterns) is zero, we can be sure that some patterns cause positive, and others a negative a , and the nonlinearity is exploited.

- 2) For **sigmoidals**, the nonlinearity is around $|a|=2$. Thus, if the mean activation (across all patterns) is zero AND the variance is around 1 or 2, we can be sure that some patterns cause a big positive, and others a big negative a , and the nonlinearity is exploited.

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner
EPFL, Lausanne, Switzerland

1. Bagging
2. Dropout
3. Other simple regularization methods
4. Choice of hidden units and initialization: 'linearity problem'
5. Vanishing gradient problem

Previous slide.

So far, our arguments have been based on the forward pass. As we will see, similar arguments can also be applied by studying the backward pass.

<p>0. Initialization of weights</p> <p>1. Choose pattern \mathbf{x}^μ input $x_k^{(0)} = x_k^\mu$</p> <p>2. Forward propagation of signals $x_k^{(n-1)} \rightarrow x_j^{(n)}$ $x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum_k w_{jk}^{(n)} x_k^{(n-1)})$ (1) output $\hat{y}_i^\mu = x_i^{(n_{\max})}$</p> <p>3. Computation of errors in output $\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) [t_i^\mu - \hat{y}_i^\mu]$ (2)</p> <p>4. Backward propagation of errors $\delta_i^{(n)} \rightarrow \delta_j^{(n-1)}$ $\delta_j^{(n-1)} = g'^{(n-1)}(a_j^{(n-1)}) \sum_i w_{ij} \delta_i^{(n)}$ (3)</p> <p>5. Update weights (for each (i, j) and all layers (n)) $\Delta w_{ij}^{(n)} = \eta \delta_i^{(n)} x_j^{(n-1)}$ (4)</p> <p>6. Return to step 1.</p>	<h2 style="margin: 0;">BackProp</h2>
---	--------------------------------------

Previous slide.

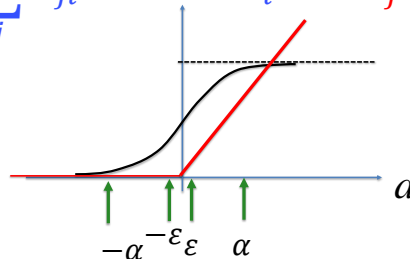
As discussed earlier, at each step of the backward pass, a factor

$$g'_j{}^{(n)} := g' \left[a_j^{(n)} \right] = g' \left[\sum_k w_{j,k}^{(n)} x_k^{(n-1)} \right]$$

appears

5. Backward pass: Vanishing gradient problem

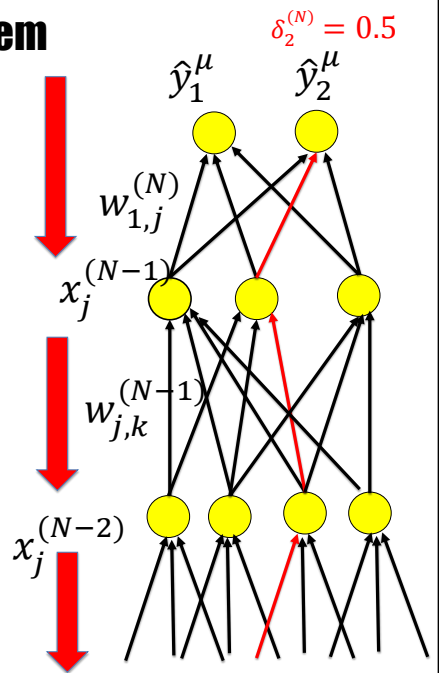
$$\delta_i^{(n-1)} = \sum_j w_{ji}^{(n)} g'^{(n-1)}(a_i^{(n-1)}) \delta_j^{(n)}$$



After N layers: each path contributes

$$\delta_i^{(1)} \sim g'^{(1)} g'^{(2)} \dots g'^{(N-1)} \delta_j^{(N)}$$

Many terms to be summed,
but most terms are tiny if N large



Previous slide.

For calculating the deltas in the first layers, we have to sum over the deltas in the second layer. To find these over those in the third layer etc.

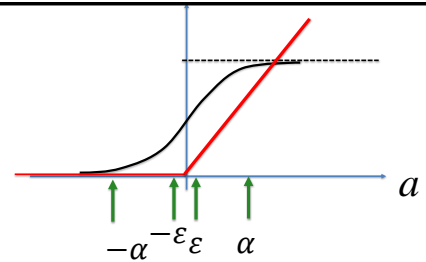
After $N-1$ layers of backpropagation, starting at the output layer N and finishing in the first layer, the deltas will contain terms of the form

$$\delta_i^{(1)} \sim g'^{(1)} g'^{(2)} \dots g'^{(N-1)} \delta_j^{(N)}$$

There are many of these summation paths, but each path contains a multiplication of several g' . If a single g' is zero, or if three g' in a path are very small (say 0.1 each), the contribution of this path to the total is negligible. Thus there is a risk that the calculated $\delta_i^{(1)}$ is very close to zero. This is called the vanishing gradient problem.

The more layers we have in a network, the higher the risk of a vanishing gradient.

5. Vanishing gradient problem



Observations:

- for each single path many terms g'
- g' is small for sigmoidal at $-\alpha$ or $+\alpha$ ($|a|=4$)
- g' vanishes for ReLU if one inactive unit sits in path
- $g'=1$ for all ReLU on 'active paths'
- for ReLU highly active forward paths coincide with good gradient transmission on backward path

Previous slide. To summarize

For **sigmoidal units**, we ideally need for a given pattern μ that for most units

1. for most units on a path $|a| < 3$ so as to make sure that the g' in the backward pass is not too small.
2. for some units on a path $|a| > 2$ so as to make sure that the forward pass exploits nonlinearities.

For **Rectified Linear units (ReLU)**, we ideally need for a given pattern μ that for some paths all units have:

1. $|a| > 0$ so as to make sure that the g' in the backward pass is not zero.
2. $|a| > 0$ so as to make sure that the forward pass goes through; but the same path should have some units with $|a| < 0$ when a different pattern is applied so as to exploit nonlinearities.

Note that the 'nonlinearity' argument is by looking at the distribution of activations across different patterns.

Conclusion: it is easier to avoid the vanishing gradient problem of BackProp when using ReLU's.

5. Vanishing gradient problem

Conclusion:

Successful forward pass

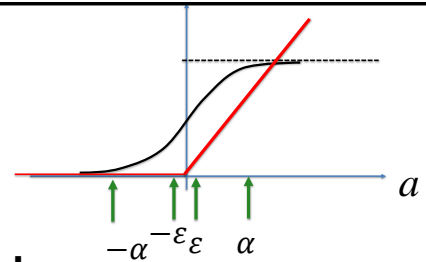
→ needs to avoid the linearity problem.

(*exploit nonlinearities*)

Successful backward pass

→ needs to avoid the vanishing gradient problem.

A good hidden units must be good for forward and backward pass!



Previous slide.

But it is not so easy to have hidden units that are good on the forward pass and the backward pass!

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner
EPFL, Lausanne, Switzerland

1. Bagging
2. Dropout
3. Other simple regularization methods
4. Initialization and choice of hidden units are important.
5. Vanishing gradient problem
6. Weight update: mean input and bias problem

Previous slide.

So far we have focused on forward and backward pass,
but the picture gets even more complicated if we include the weight update step.

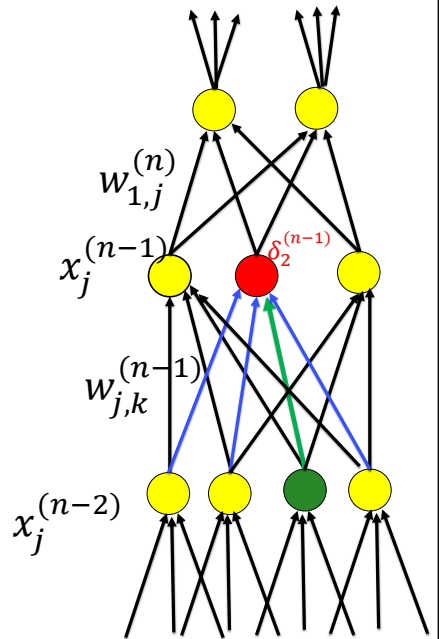
6. Weight update step

update **all** weights

$$\Delta w_{i,j}^{(n-1)} = \delta_i^{(n-1)} x_j^{(n-2)}$$

Weights onto the same neuron (red) are all updated with same delta

→ if $x_j^{(n-2)}$ are all positive,
all the weights onto red neuron increase or decrease together



Previous slide.

The update formula of the BackProp algorithm

$$\Delta w_{i,j}^{(n-1)} = \delta_i^{(n-1)} x_j^{(n-2)}$$

implies that all weights onto the same neuron i (red), share the same $\delta_i^{(n-1)}$.

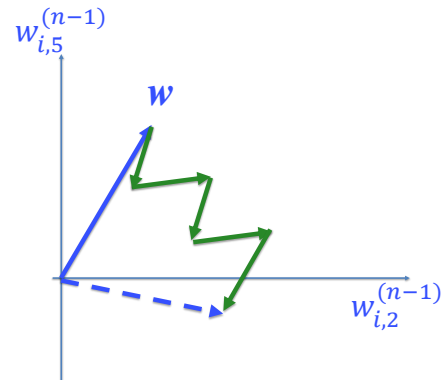
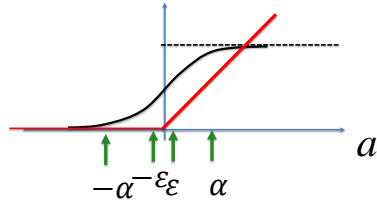
This has **two implications**.

The first one concerns the possible movements of the weight vector, to be discussed now.

6. Weight update step

update **all** weights

$$\Delta w_{i,j}^{(n-1)} = \delta_i^{(n-1)} x_j^{(n-2)}$$



Weights onto the same neuron
are all updated with same delta

- Problem for ReLu and other units with non-negative x
- No problem for tanh
- No problem for shifted exponential linear Selu

Previous slide.

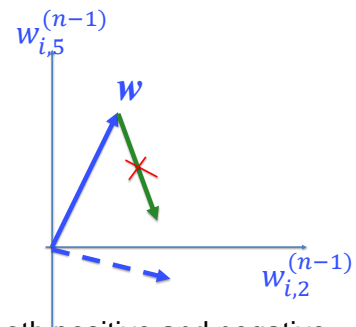
Assume that we work with ReLu's, so that all x are non-negative.

Then during the update step, two weights onto the same neuron either move both up or down together. For example for weights with index $j=2$ and $j=5$

$$\text{If } \Delta w_{i,2}^{(n-1)} = \delta_i^{(n-1)} x_2^{(n-2)} \geq 0 \text{ , then also } \Delta w_{i,5}^{(n-1)} = \delta_i^{(n-1)} x_5^{(n-2)} \geq 0$$

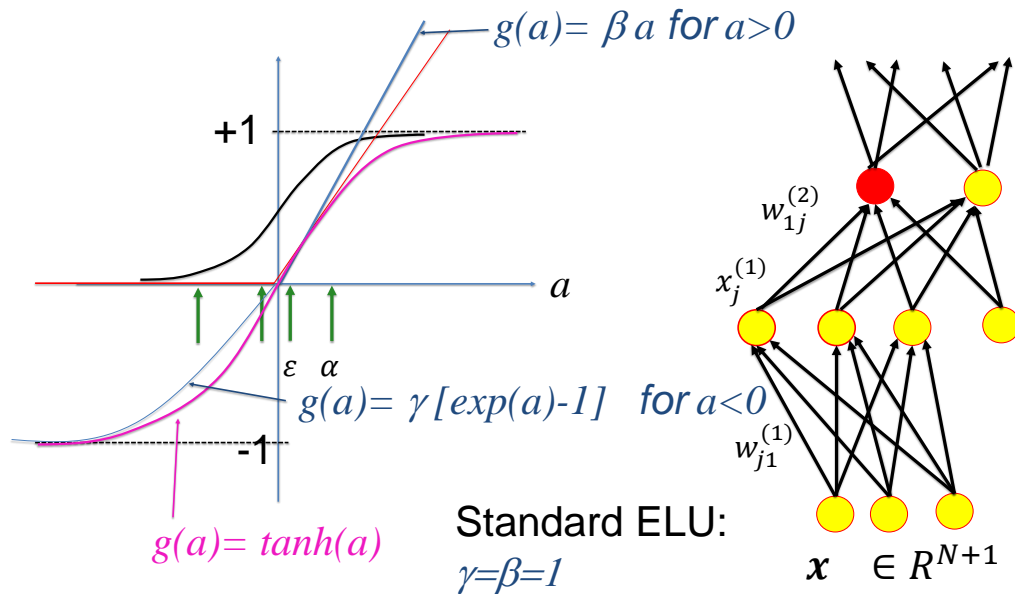
Thus changes in direction downward-right, as on the graph on the right are excluded.

To move downward right, several iterations are necessary, as shown on the previous slide.



This problem is absent for units with a gain function that has both positive and negative values. For example, the problem is absent if we choose for the gain function of hidden units $g(a) = \tanh(a)$

Shifted Exponential Linear (SELU) vs. tanh



Previous slide.

Instead of $\tanh(a)$, we can also work with the shifted exponential linear units (ELU) or a scaled version called SELU. SELU has additional parameters γ, β

Similar to the ReLU, the ELU and SELU are linear for positive activation values a .

Similar to the tanh-unit (and in contrast to the ReLU), the ELU and SELU are smooth and also generates negative outputs.

6. Bias problem

update **all** weights

$$\Delta w_{i,j}^{(n)} = \delta_i^{(n)} x_j^{(n-1)}$$

Before update

$$a_i^{(n)} = \sum_j w_{ij}^{(n)} x_j^{(n-1)} - \vartheta$$

after update

$$a_i^{(n)} = \sum_j [w_{ij}^{(n)} + \Delta w_{i,j}^{(n)}] x_j^{(n-1)} - \vartheta$$

same sign for all j

non-negative
(for ReLu etc)

Weights onto the same neuron
are all updated with same **delta**

- Problem for ReLu and other units with non-negative x
- The mean changes! ('bias problem')
- But controlling the mean was important for correct initialization!
- Return of vanishing gradient and linearity problem!

Previous slide.

As we have seen, the update formula of the BackProp algorithm implies that all weights onto the same neuron i (red), share the same $\delta_i^{(n-1)}$.

This has **two implications**.

The first one concerns the possible movements of the weight vector, discussed above.

The second implication concerns a shift in the mean. If we use a ReLu or a sigmoidal (where all the x -values are non-negative), then the mean activation changes in each update step, even if the threshold θ does not change!

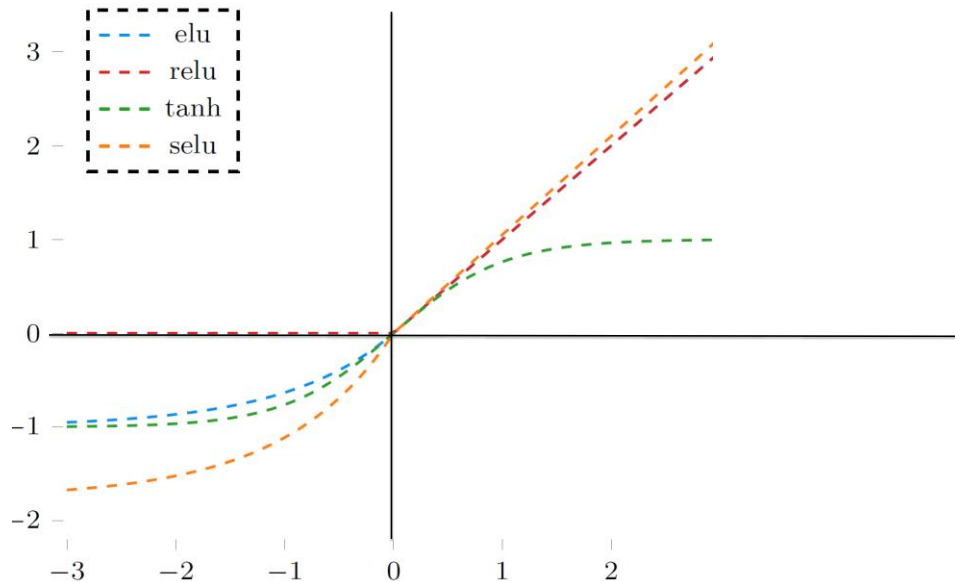
However, we have seen earlier that controlling the mean activity (where the mean is taken over the distribution of patterns) is important to correctly exploit the nonlinearities of a ReLu. In fact the mean should be close to zero, so that some patterns cause an activation, and others not.

Quiz:

- forward propagation with ReLu leaves only a few active paths
- back propagation with ReLu leaves only a few active paths
- a non-zero weight update step of ReLu shifts most often the mean
- forward propagation with ReLu is always linear on the active paths
- in a ReLu network all patterns are processed with the same linear filter
- in a sigmoidal network with small weights (and normalized inputs) all patterns are processed with the same linear filter
- in a sigmoidal network with big weights, there are active units in the forward pass that contribute a vanishing gradient in the backward path
- in a network with SELU, there are active units in the forward path which contribute a vanishing gradient in the backward path
- a non-zero the weight update step of SELU shifts the mean

Your notes.

Shifted Exponential Linear vs. tanh



Previous slide.

The generalized 'Shifted exponential linear unit' (SELU) has two parameters, $\beta > 1, \gamma > 1$:

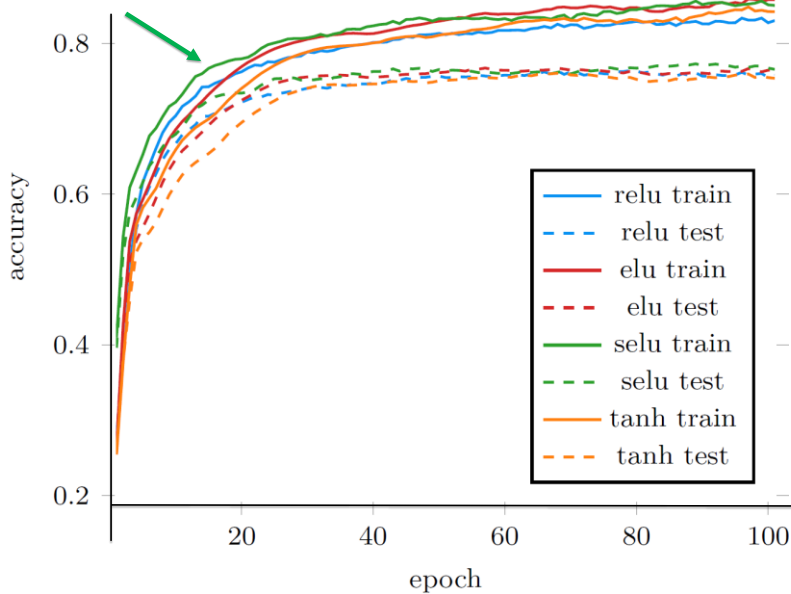
$$g(a) = \beta a \quad \text{for } a > 0$$

$$g(a) = \gamma [\exp(a) - 1] \quad \text{for } a < 0$$

The orange curve shows that the SELU starts at values below (-1) and, for positive a , increases slightly faster than the RELU.

The SELU parameters beta and gamma are chosen such as to minimize the bias problem, as well as the linearity and vanishing gradient problem.

Shifted Exponential Linear (SELU)



Previous slide.

A network learns faster with SELU as hidden units. The test error after convergence is not affected. The training time is shorter because many of the problems such as vanishing gradient, unexploited nonlinearities, or shifting mean that plague learning during the initial epochs are minimized.

6. Conclusion

- initialization is important so as to exploit nonlinearities
- choice of hidden unit is important in **initial phase** of training
- ReLU has disadvantages in keeping the mean
 - batch normalization
- Tanh has problems with vanishing gradient
- Sigmoidal has problems with vanishing gradient **and** mean
- SELU solves all problems and is currently best choice

Paper: Klambauer, ..., Hochreiter (2017)

Self-normalizing neural networks

<https://arxiv.org/pdf/1706.02515.pdf>

Previous slide.

Thus, if you have the choice, take SELU's.

The shifting mean can also be addressed by batch normalization, which is the topic of the next section.

Artificial Neural Networks: Lecture 4

Tricks of the Trade in deep networks

Wulfram Gerstner
EPFL, Lausanne, Switzerland

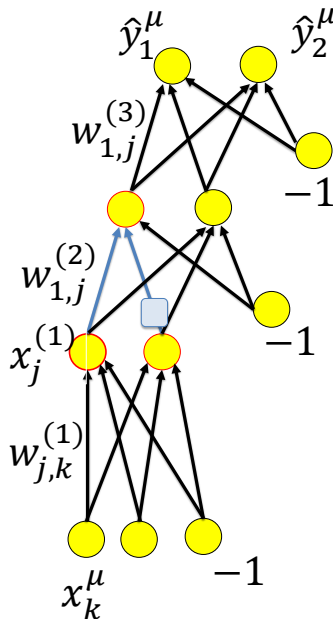
1. Bagging
2. Dropout
3. Other simple regularization methods
4. Hidden units: linearity problem (exploit nonlinearities)
5. Hidden units: Vanishing gradient problem
6. Weight update: bias problem
7. Batch normalization

Previous slide.

For unbalanced hidden units such as ReLu or Sigmoidals with non-negative outputs, the mean will shift during training even if we initialize well.

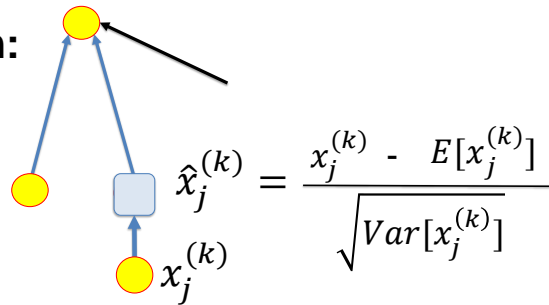
Batch normalization solves this issue.

7. Batch normalization: Idea



Normalize input on each input line

Zoom:



Previous slide.

At the output $x_j^{(k)}$ of each neuron, we add a normalization step:

We calculate the mean and the variance of $x_j^{(k)}$ (taken over a batch or minibatch). Then we renormalize to mean zero and unit variance.

This renormalization step is denoted in the following by a small box in the network graph.

When you do backprop, the blue box has to be taken into account for both forward and backward pass.

7. Batch normalization

Ioffe&Szegedi, 2015

Work with minibatch:
**Normalize per
 minibatch**

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;
 Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

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

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

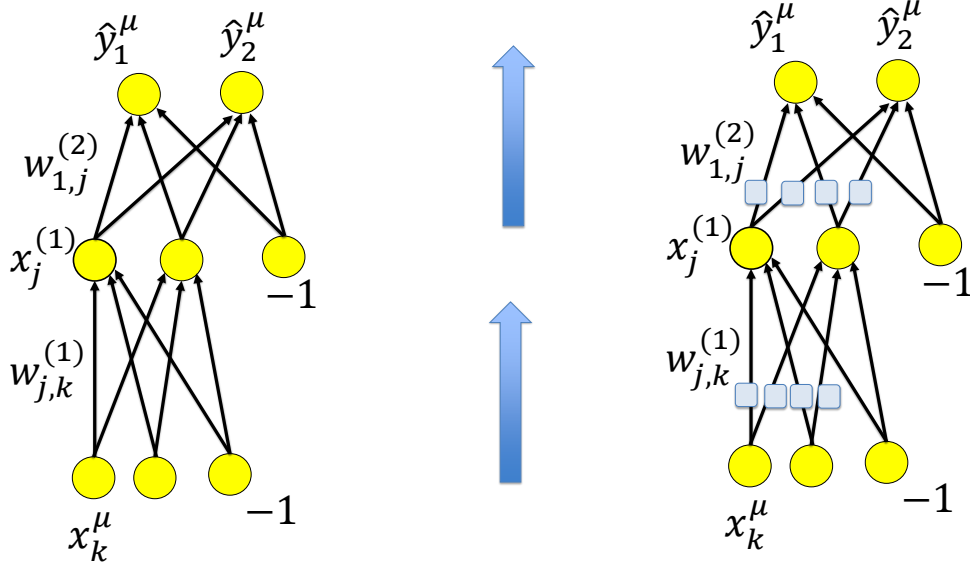
Previous slide.

The blue box corresponds to a mathematical transformation $y = \text{BN}(x)$. BN stands for Batch Normalization.

Since we are not sure that we want to normalize the mean to exactly zero and the variance to exactly one, we allow for additional parameters gamma and beta.

These parameters are learned using backprop.

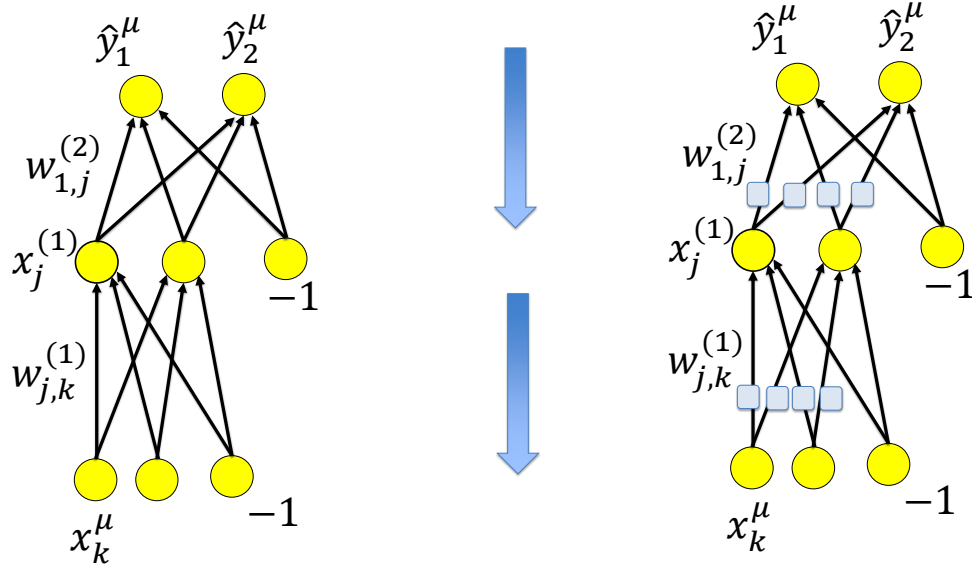
7. Batch normalization Ioffe&Szegedi, 2015



Previous slide.

Note that it does not make sense to add a normalization step for the thresholds (i.e., the inputs fixed at -1 in the graph).

7. Batch normalization Ioffe&Szegedi, 2015



Previous slide.

The normalization steps lead to additional terms in the backprop algorithm which is directly taken care of (again) by an efficient implementation of the chain rule.

7. Batch normalization

Ioffe&Szegedi, 2015

5: **end for**
 6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters Θ
 $\{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^K$
 7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen
 // parameters
 8: **for** $k = 1 \dots K$ **do**
 9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc
 10: Process multiple training mini-batches \mathcal{B} , each
 size m , and average over them:

$$E[x] \leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\text{Var}[x] \leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

Input: Network N with trainable parameters Θ ;
 subset of activations $\{x^{(k)}\}_{k=1}^K$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network

2: **for** $k = 1 \dots K$ **do**

3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)}, \beta^{(k)}}(x^{(k)})$ to
 $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)

4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take
 $y^{(k)}$ instead

5: **end for**

1: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma, \beta}(x)$ with
 $y = \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma E[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right)$

2: **end for**

Algorithm 2: Training a Batch-Normalized Network

Previous slide.

The full algorithm of Batch Normalization.

7. Batch normalization

Ioffe&Szegedi, 2015

Necessary for ReLu and other unbalanced hidden units

Normalization step in forward pass is also taken care of during backward pass

Objectives for today:

- Bagging: **multiple models help always to improve results!**
- Dropout: **two interpretations**
 - (i) **a practical implementation of bagging**
 - (ii) **forced feature sharing**
- BackProp: Initialization, nonlinearity, and symmetry
- What are good units for hidden layers?
 - problems of vanishing gradient and shift of mean**
 - solved by Shifted exponential linear (SELU)**
- Batch normalization **→ necessary for ReLu**

The end