Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets2: Tricks of the Trade in Deep Learning

Part 1: Questions and Aims of this Lecture

Objectives for today:

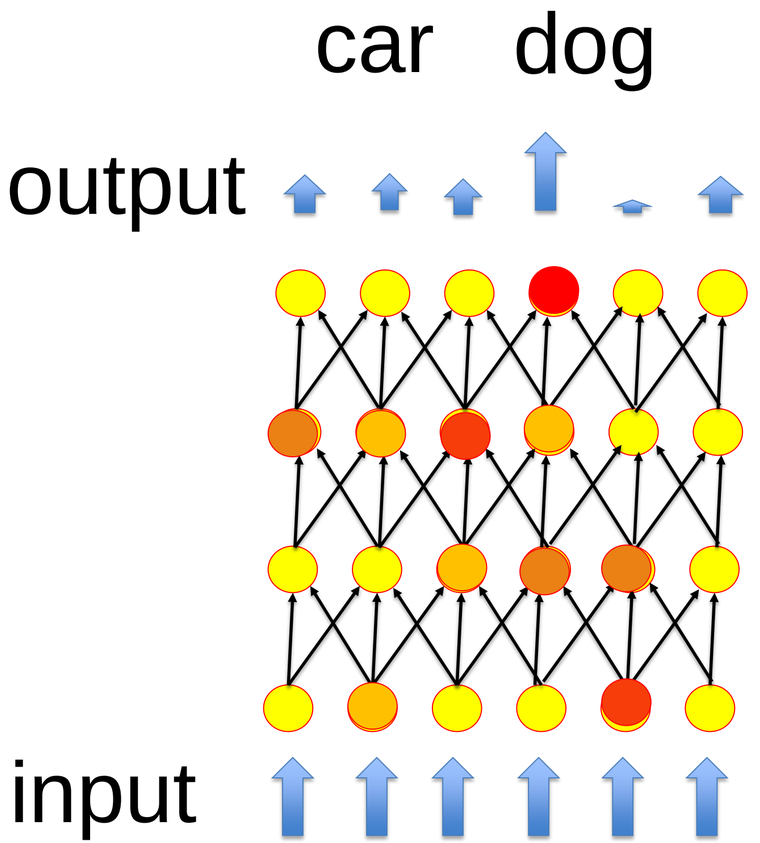
- Bagging
- Dropout
- What are good units for hidden layers?
- Rectified linear unit (RELU)
- Shifted exponential linear (ELU and SELU)
- BackProp: Initialization
- Linearity problem, vanishing gradient problem, bias problem
- Batch normalization

Previous slide.

This first part formulates the aims and big questions for this week.

Review: Artificial Neural Networks for classification

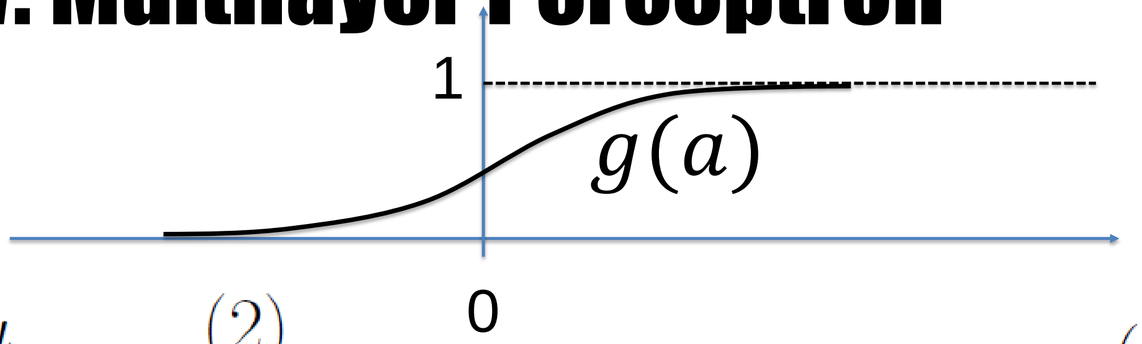
Aim of learning:
Adjust connections such that output is correct (for each input image, **even new ones**)



Previous slide.

We use an artificial neural network, with multiple layers.
We adjust the weights of the network so that it works well for new data.
This week we will address three important questions.

Review: Multilayer Perceptron



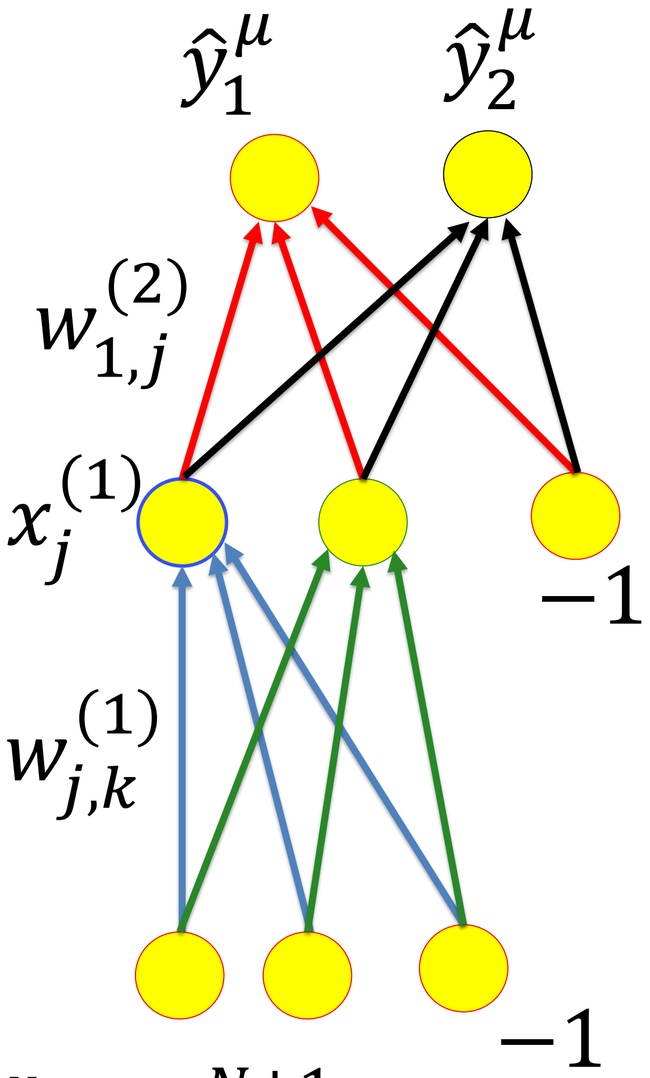
$$\hat{y}_i^{\mu} = x_i^{(2)} \quad (1)$$

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

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

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

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



$\boldsymbol{x^{\mu}} \in R^{N+1}$

Previous slide.

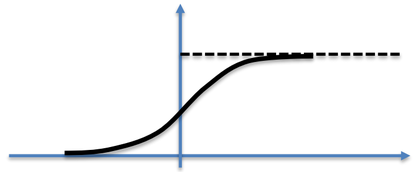
In each layer, neurons perform a nonlinear transform g(a).

Deep Neural Networks: choice of neuron model

output layer

use sigmoidal unit (for single-class)

$$\hat{y}_1 = g(a) = \frac{1}{1+e^{-a}}$$



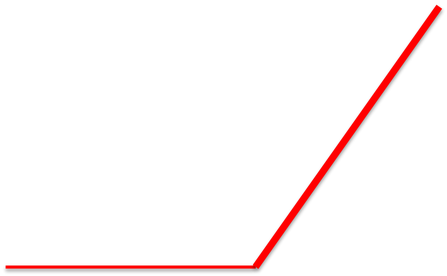
or softmax (for exclusive multi-class)

$$\hat{y}_k = \frac{\boldsymbol{exp}(a_k)}{\sum_j \boldsymbol{exp}(a_j)}$$

hidden layer

Why?

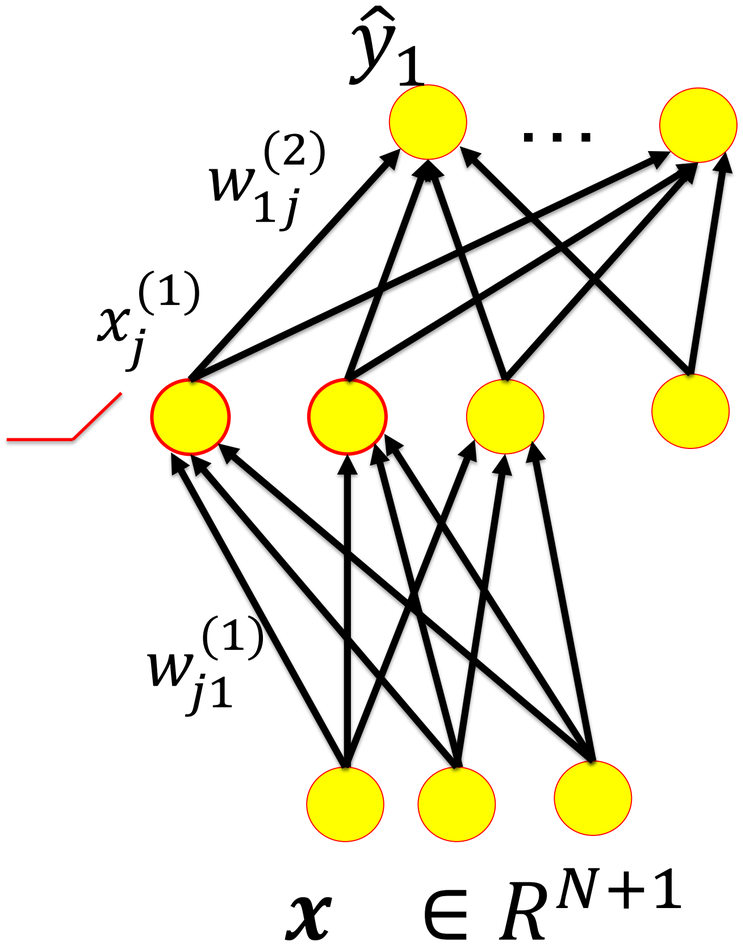
use rectified linear unit in $N+1$ dim.



f(x)=x for x>0

f(x)=0 for x<0 or x=0

Better choices?



Previous slide.

In the output layer of a neural network trained on a classification task, we should always use a sigmoidal unit (for yes-no single-class tasks) or the softmax function for classification into multiple classes.

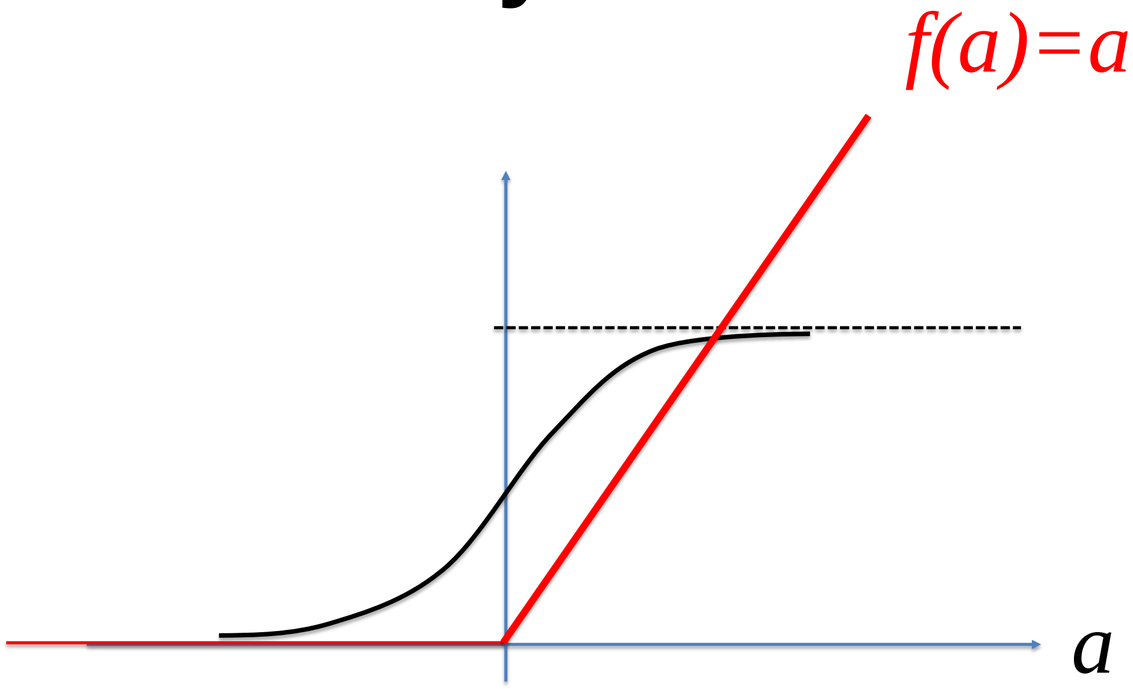
The softmax function will be discussed in the context of statistical classification in the next lecture. In case you want to preview it. Output k is given by

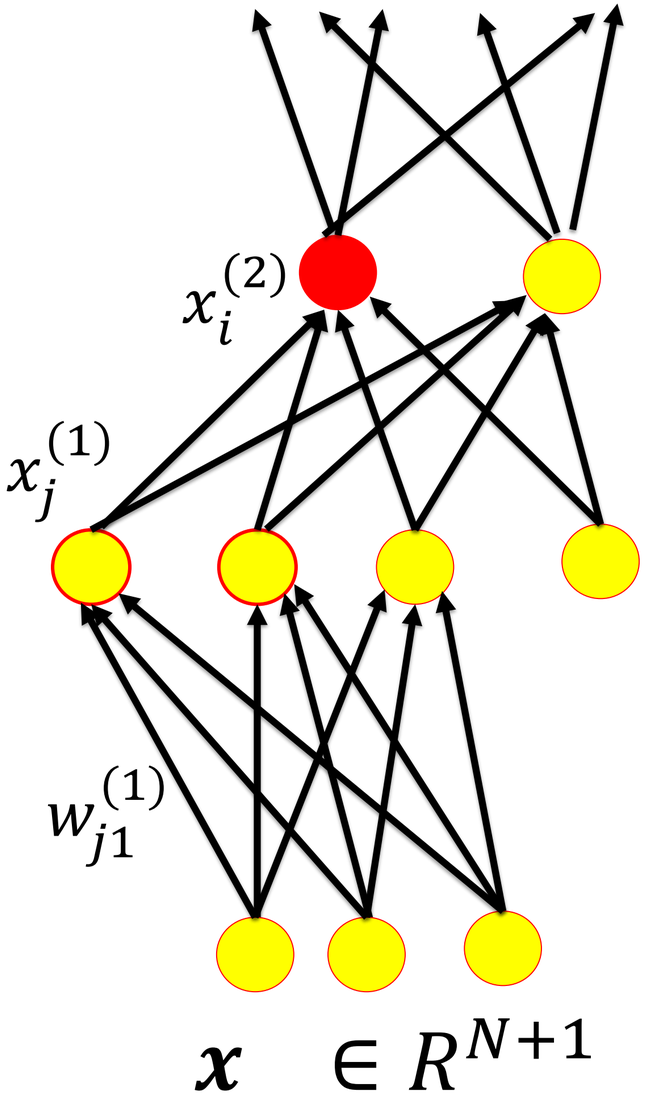
$$\hat{y}_k = \frac{\boldsymbol{exp}(a_k)}{\sum_{\boldsymbol{j}} \boldsymbol{exp}(a_j)}$$

where the sum in the denominator runs over all output units (all classes)

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

Rectified Linear (RELU) vs. Sigmoidal in hidden layer



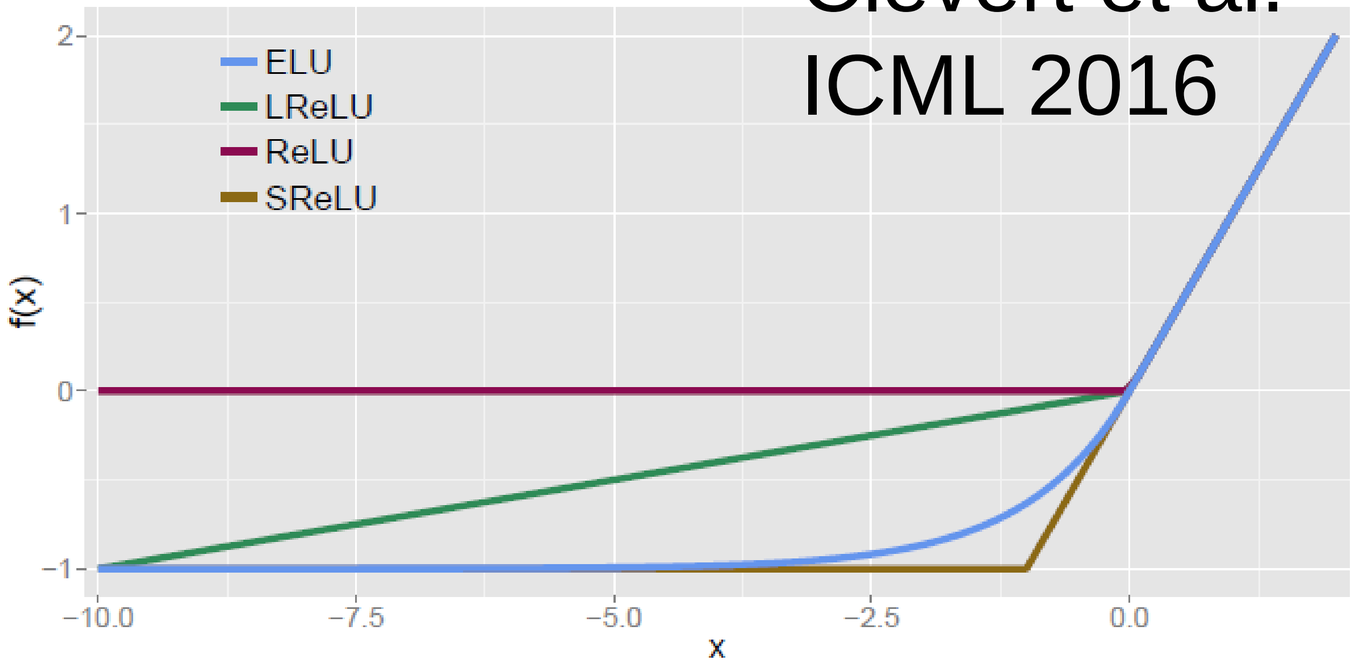


Previous slide.

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

Exponential Linear (ELU) vs RELU

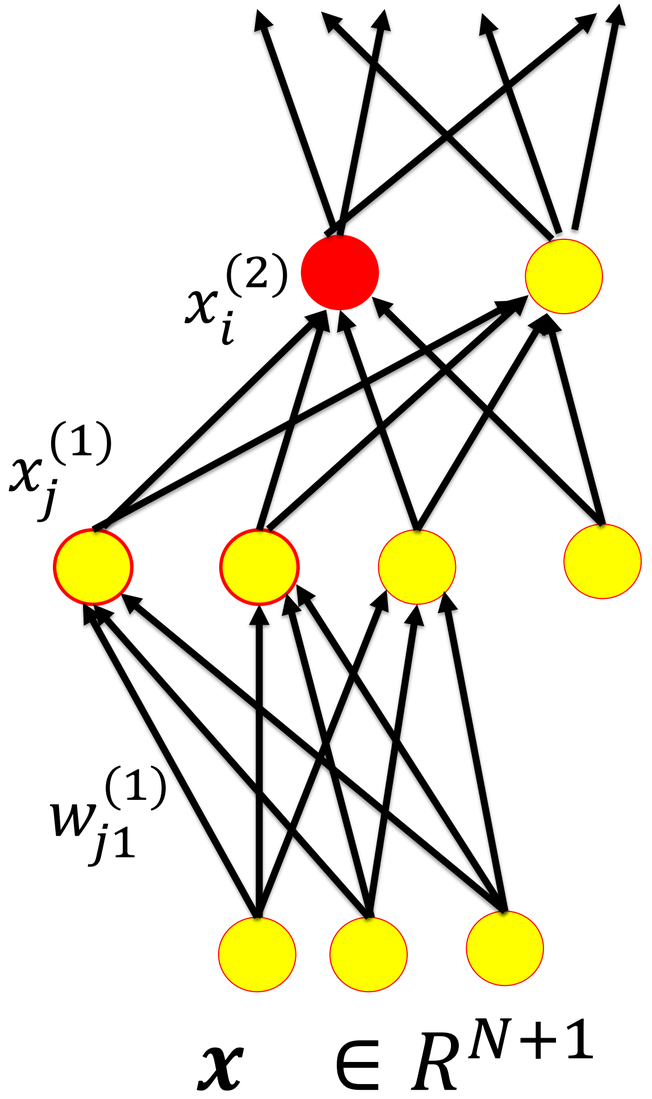
Clevert et al.
ICML 2016



Exponential Linear (ELU)

Shifted ReLU (SReLU)

Leaky ReLu (LReLU)



Previous slide.

… or a shifted exponential-linear function which is bounded from below by -1 and continues linearly for x>0: Exponential Linear Unit (ELU).

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

$$f(a)=exp(a)-1 \text{ for } a<0$$

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 5, today, starting around slides 109.

To answer this question, we have to study all three phases of the BackProp algorithm.

BackProp

0. **Initialization of weights**

1. **Choose pattern** $\mathbf{x}^\mu$

   input $x_k^{(0)} = x_k^\mu$

2. **Forward propagation of signals** $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

   output $\hat{y}_i^\mu = x_i^{(n_{\max})}$

3. **Computation of errors in output**

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})})\ [t_i^\mu - \hat{y}^\mu] \qquad (2)$$

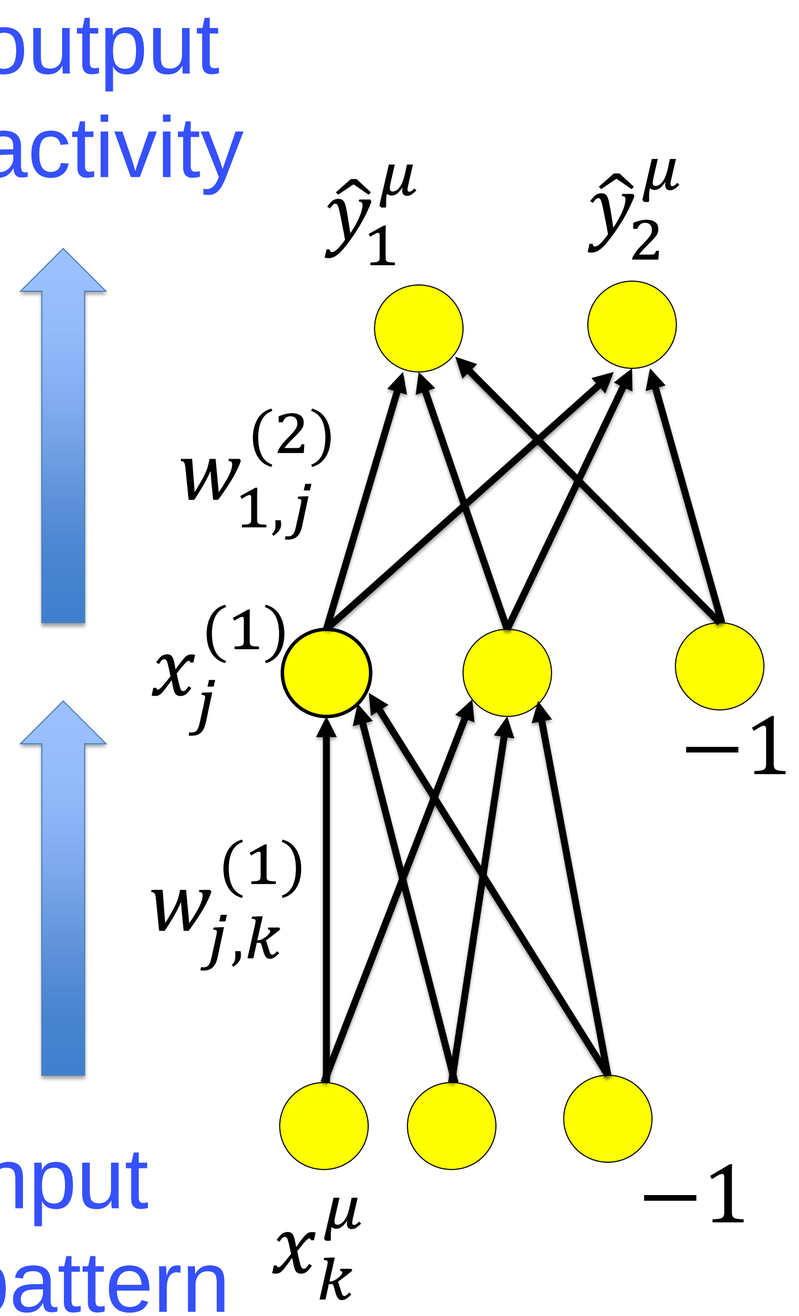
4. **Backward propagation of errors** $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \qquad (3)$$

5. **Update weights (for each** (i,j) **and all layers** (n)**)**

$$\Delta w_{ij}^{(n)} = -\gamma\ \ \delta_i^{(n)}\ x_j^{(n-1)} \qquad (4)$$

6. **Return to step 1.**



Previous slide.

In the previous lecture, we have studied the BackProp algorithm with forward

BackProp

0. **Initialization of weights**

1. **Choose pattern** $\mathbf{x}^{\mu}$

   input $x_k^{(0)} = x_k^{\mu}$

2. **Forward propagation of signals** $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

   output $\hat{y}_i^{\mu} = x_i^{(n_{\max})}$

3. **Computation of errors in output**

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})})\ [t_i^{\mu} - \hat{y}^{\mu}] \qquad (2)$$

4. **Backward propagation of errors** $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

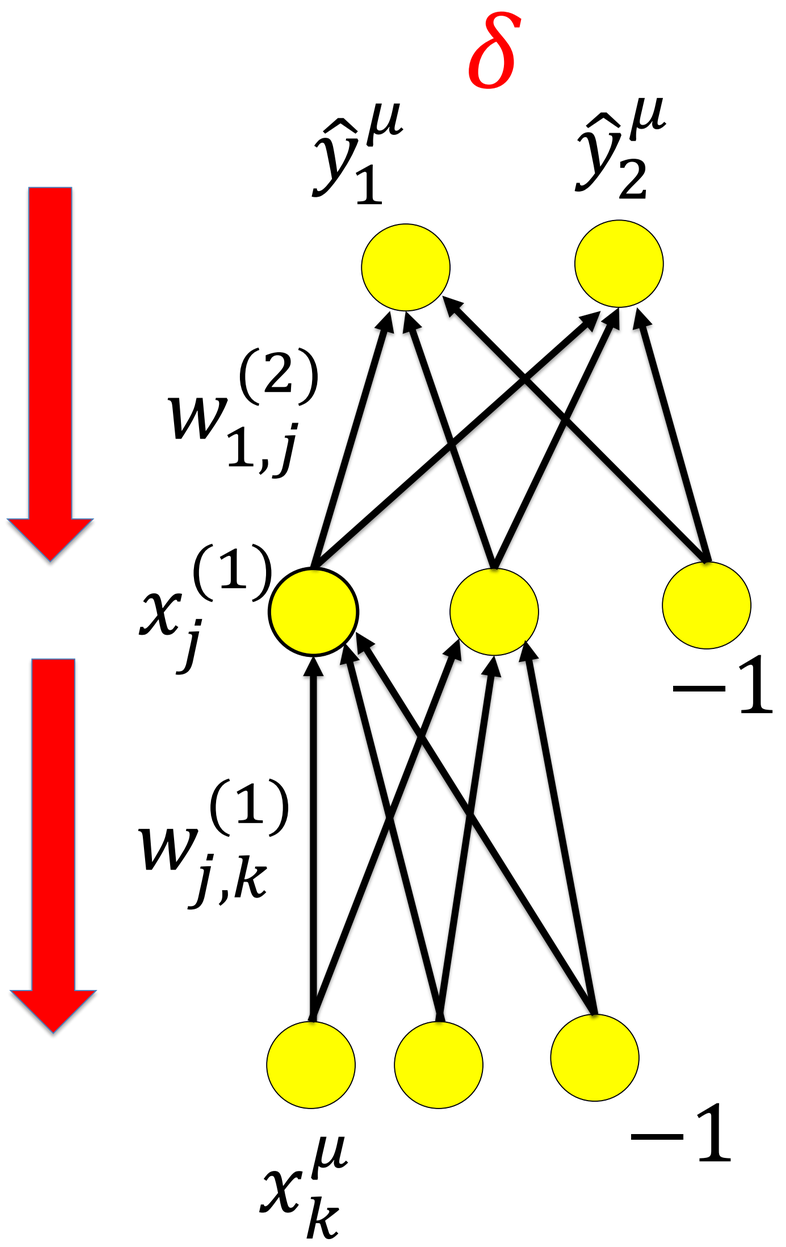
$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \qquad (3)$$

5. **Update weights (for each** (i,j) **and all layers** (n)**)**

$$\Delta w_{ij}^{(n)} = -\gamma\ \ \delta_i^{(n)}\ x_j^{(n-1)} \qquad (4)$$

6. **Return to step 1.**

Calculate output error



Previous slide.

… and backward pass.

BackProp

0. **Initialization of weights**

1. **Choose pattern** $\mathbf{x}^\mu$

   input $x_k^{(0)} = x_k^\mu$

2. **Forward propagation of signals** $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

   output $\hat{y}_i^\mu = x_i^{(n_{\max})}$

3. **Computation of errors in output**

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})})\ [t_i^\mu - \hat{y}^\mu] \qquad (2)$$

4. **Backward propagation of errors** $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \qquad (3)$$

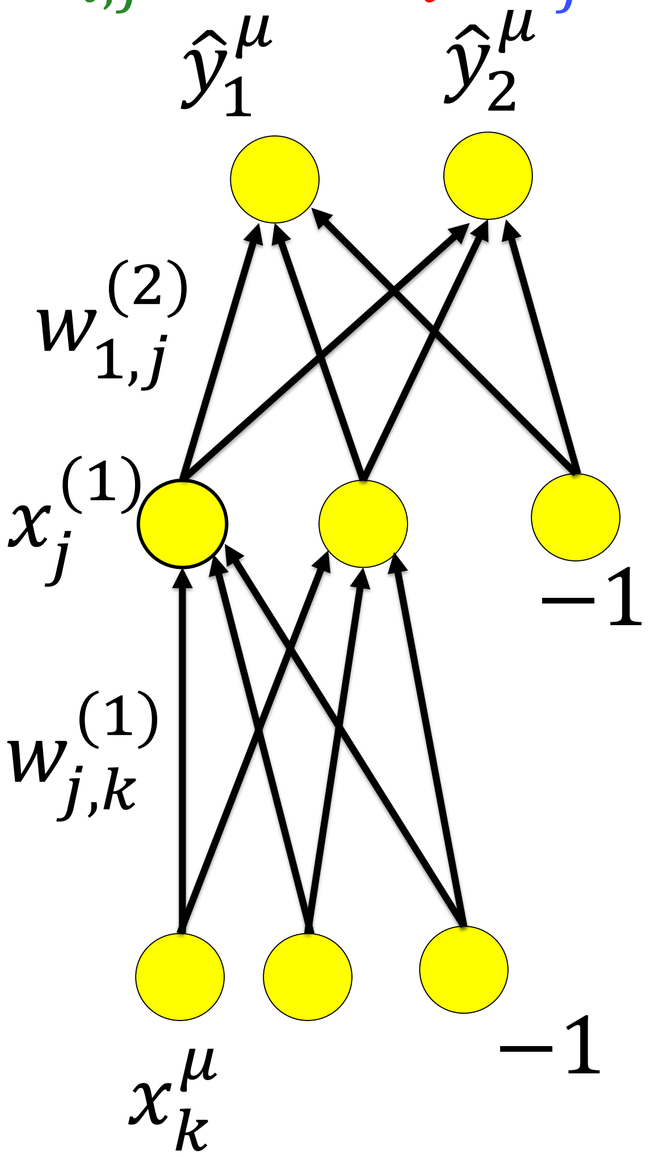
5. **Update weights (for each** (i,j) **and all layers** (n)**)**

$$\Delta w_{ij}^{(n)} = -\gamma\ \ \delta_i^{(n)}\ x_j^{(n-1)} \qquad (4)$$

6. **Return to step 1.**

update **all** weights

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



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:

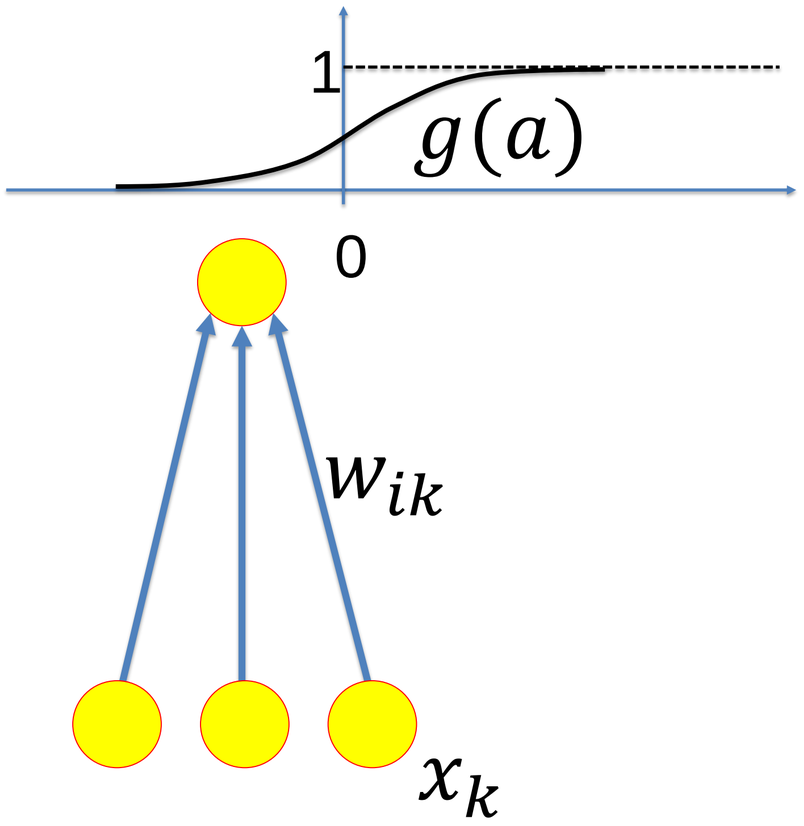
Why does the initatialization or normalization matter in backprop?

Previous slide.

This question will also be addressed in part 4, starting with slide 109.

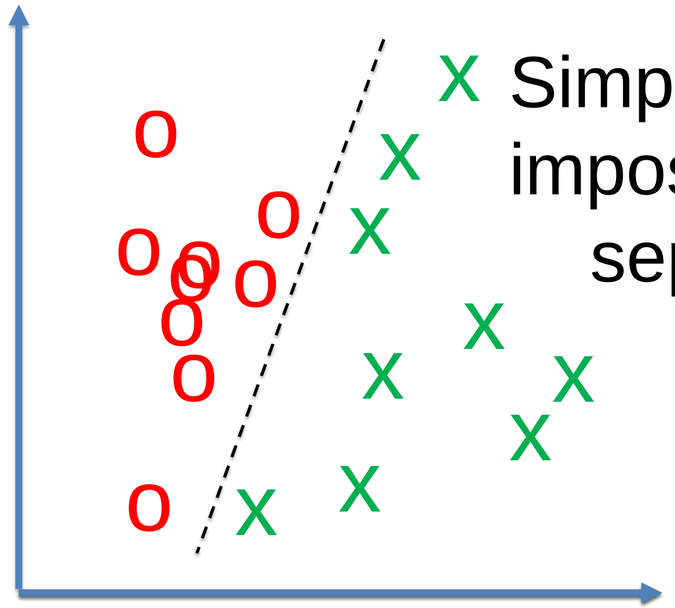
Review: Single-Layer networks/simple perceptron

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



vector $\boldsymbol{x}$

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

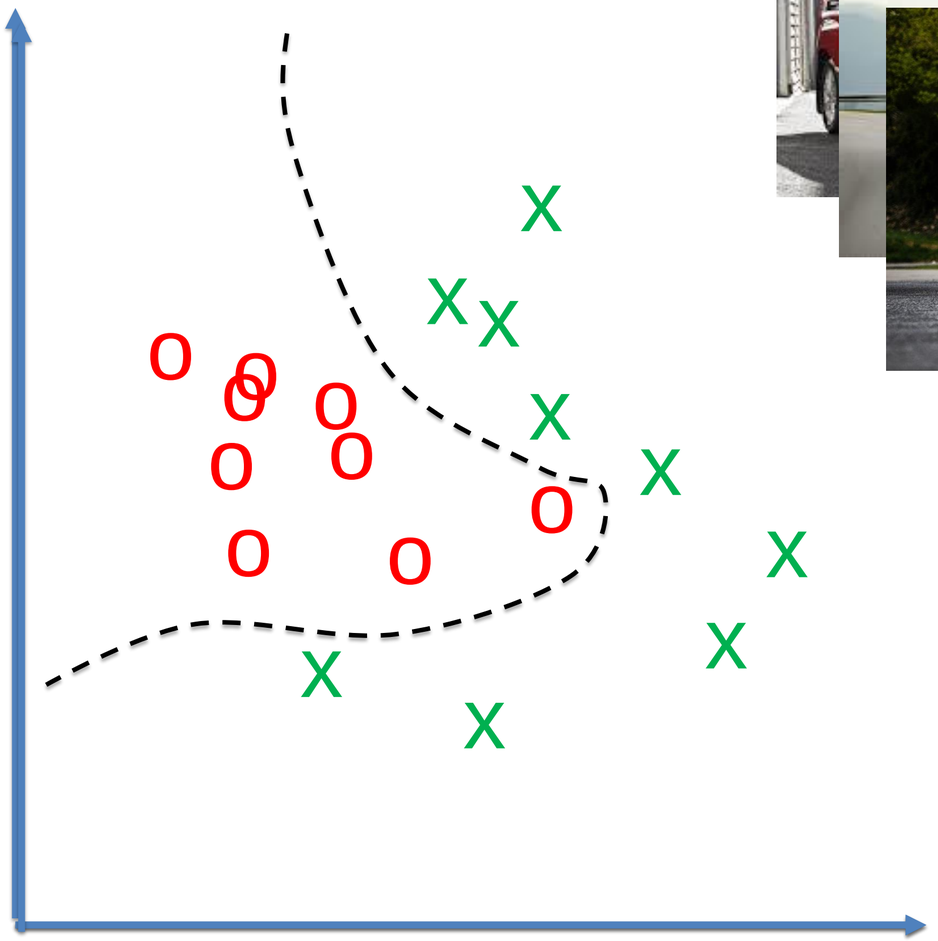


Simple perceptron imposes a linear separation

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.

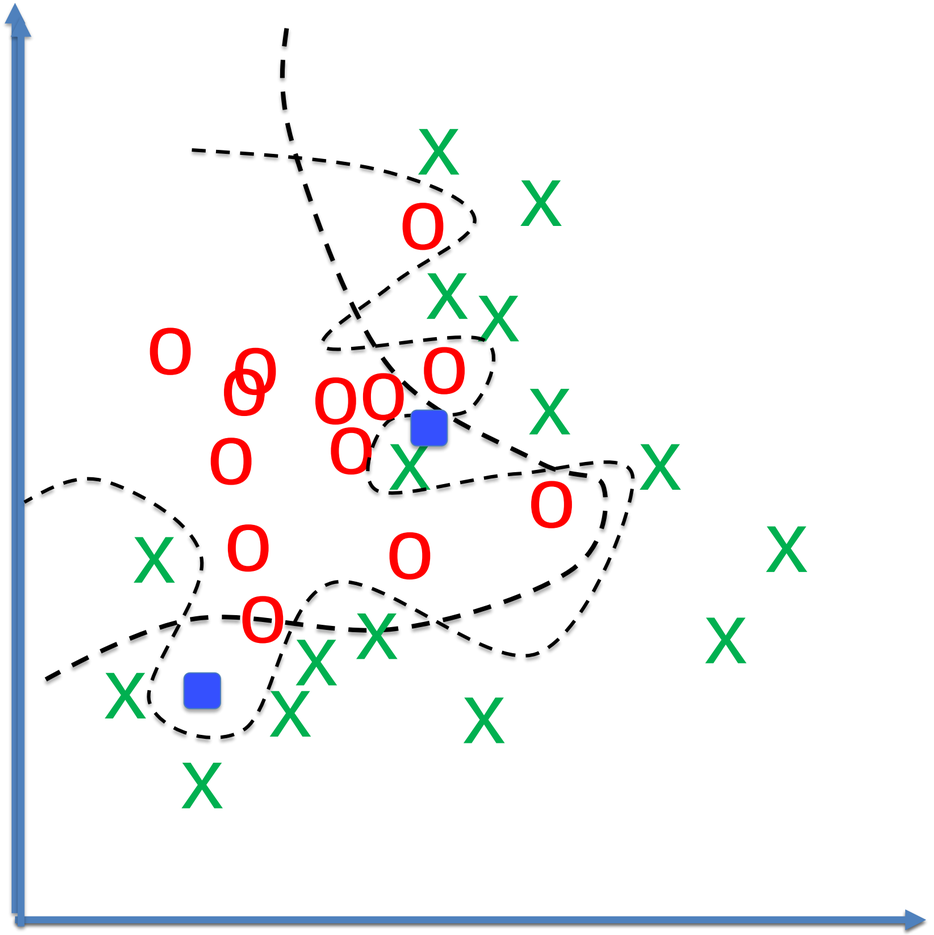
Whereas a multilayer perceptron is flexible enough to solve complex classification problems

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 of deep networks?

… and why?

We start with this question!

Previous slide.

We have already seen two powerful regularization methods, early stopping and L2 (or L1) 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

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Part 1: Questions and Aims of this Lecture

Objectives for this lecture:

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

Previous slide.

We start with bagging which is an traditional and generic methods of regularization in machine learning.

We use this as a preparation for Dropout which is a method that is widely used in neural networks and a rather specific trick for deep networks.

We will then turn to questions of why some neuron models might be preferable compared to others.

And why you need batch normalization when you work with piecewise linear units.

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Part 2: Bagging

Wulfram Gerstner

EPFL, Lausanne, Switzerland

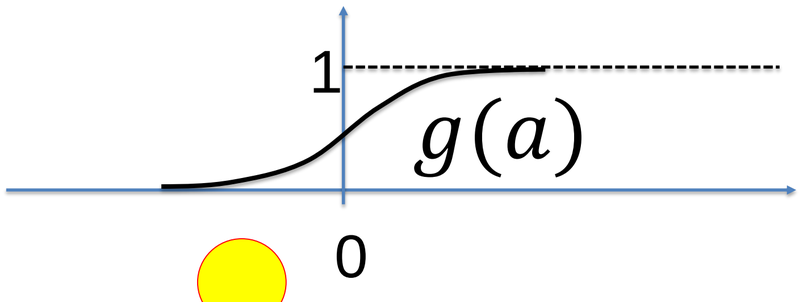
1. Questions and Aims of this Lecture
2. Bagging

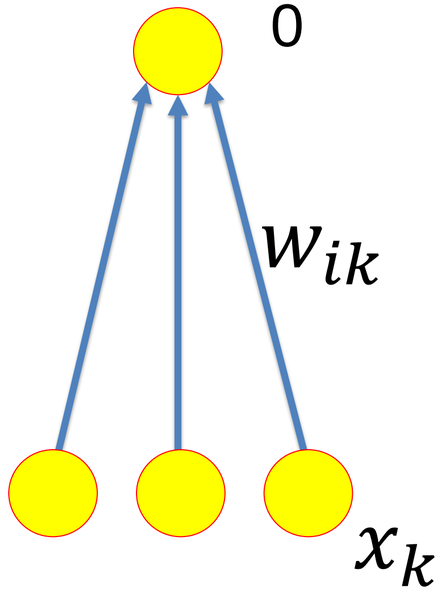
Previous slide.

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

Bagging Example: simple perceptron

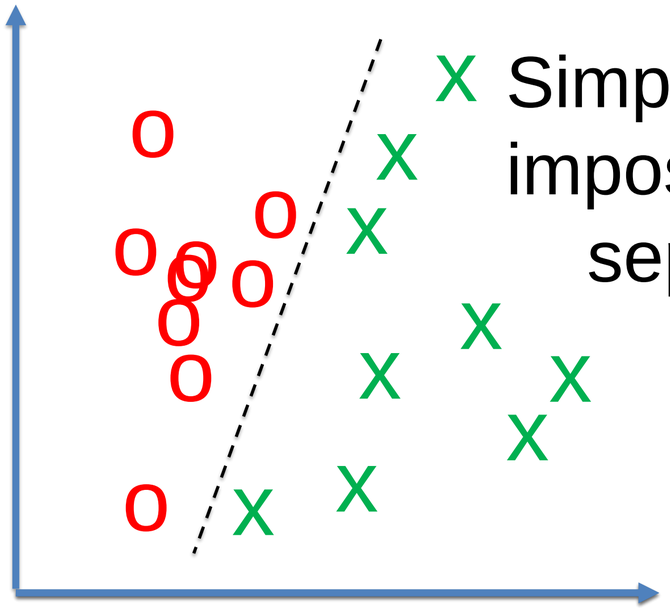
$$\hat{y} \quad = 0.5[1 + tanh(\sum_k w_k\, x_k - \vartheta)]$$





vector $\boldsymbol{x}$

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

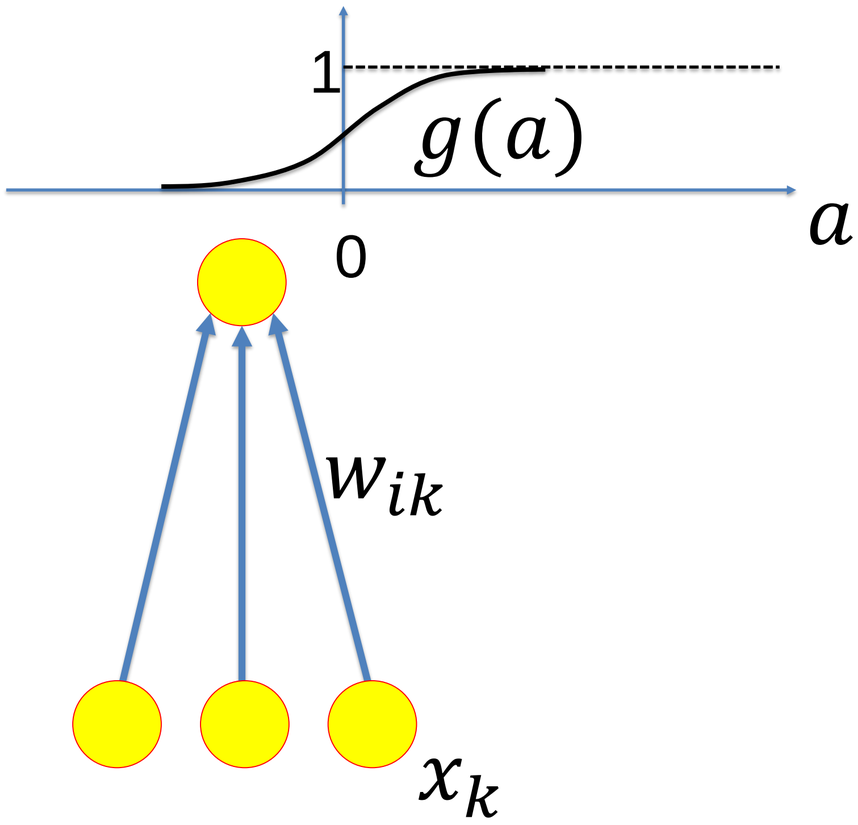


Simple perceptron imposes a linear separation

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.

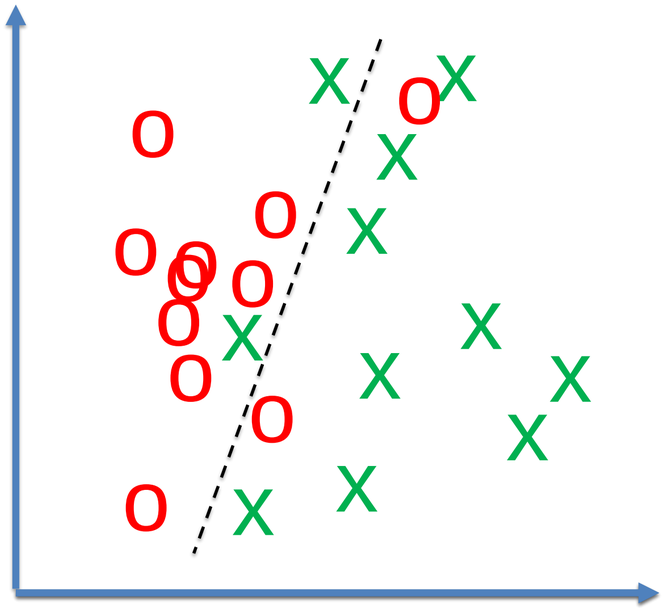
Bagging Example: simple perceptron for noisy data

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



vector $\boldsymbol{x}$

Find best (approximate) linear separation



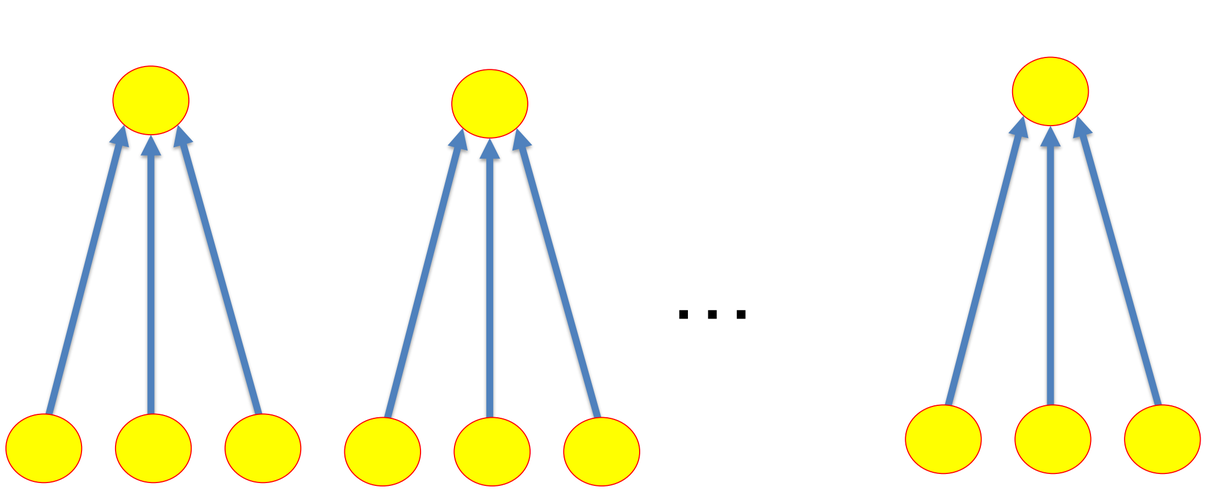
Previous slide.

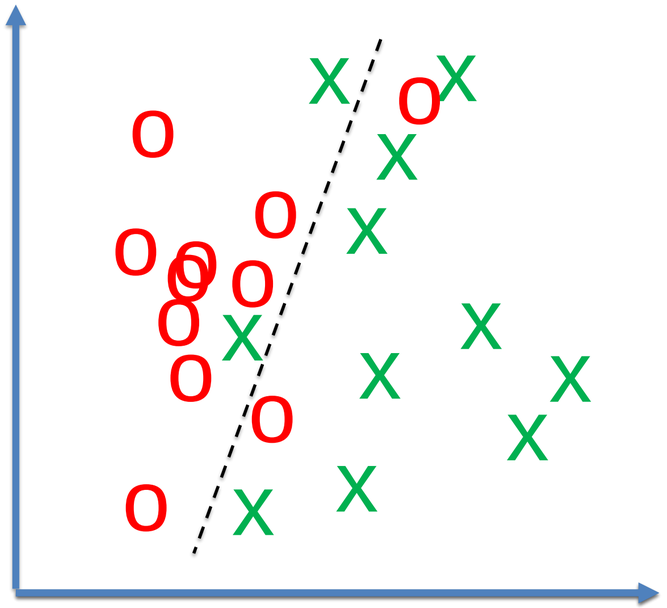
In the following we work with noisy data and use a sigmoidal in the output.

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

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

Each variant tries to find best (approximate) linear separation





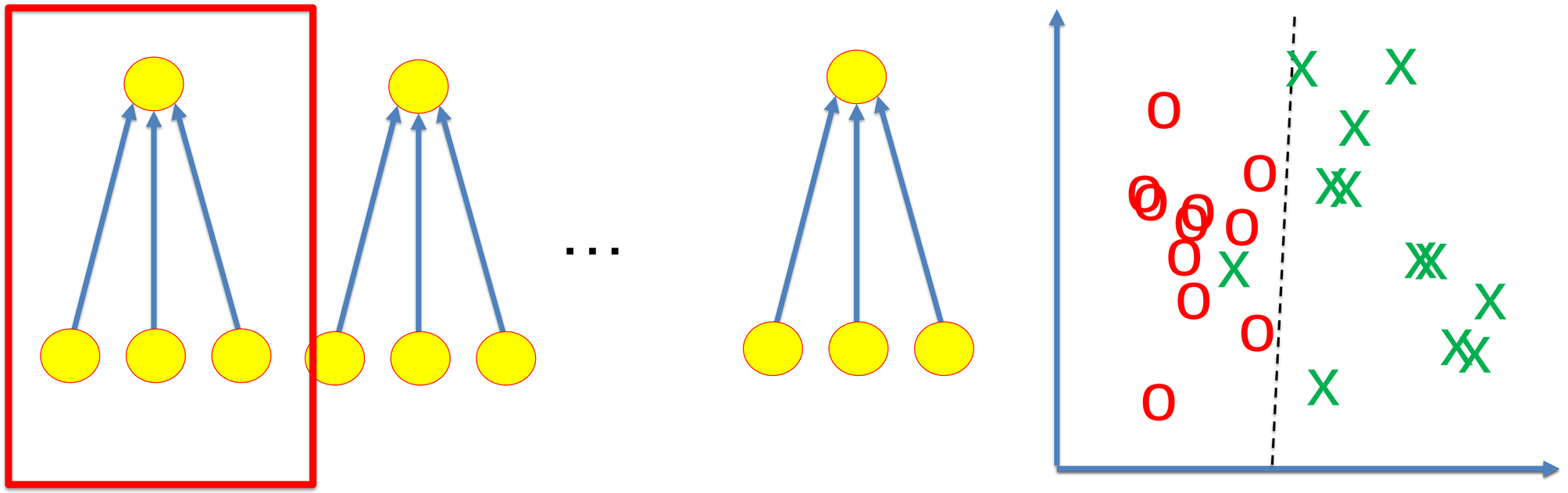
Previous slide.
We work with K repetitions of the simple perceptron.

More generally, repeating a simple model is the first idea of bagging

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



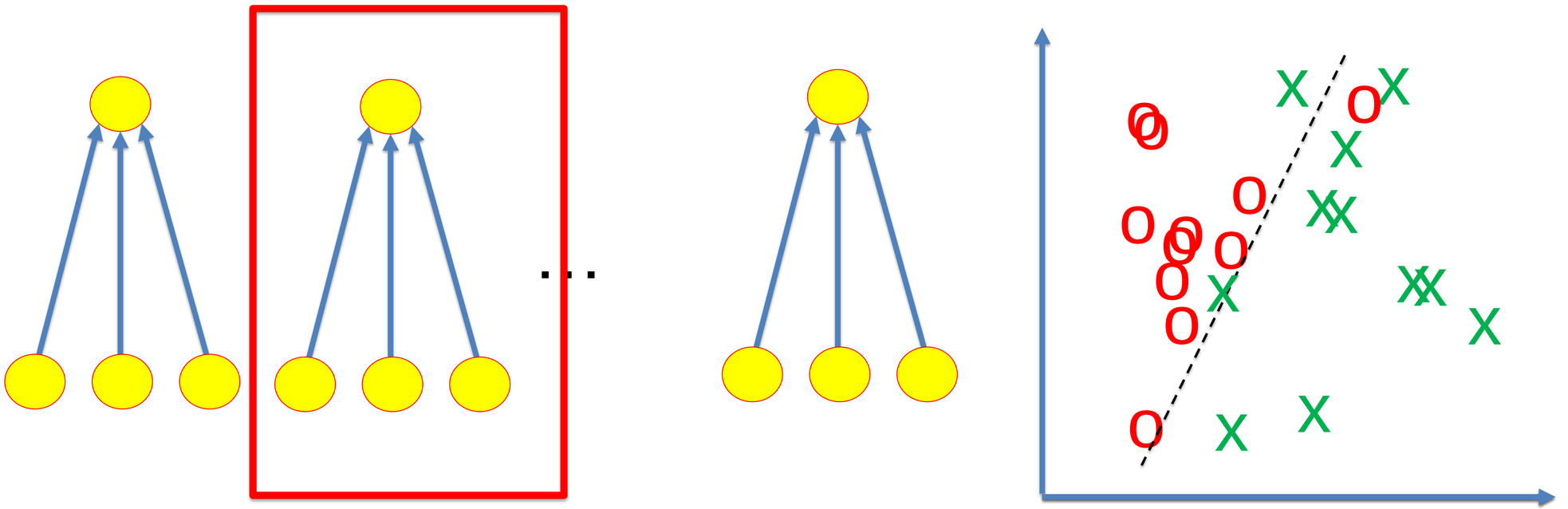
Previous slide.

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

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



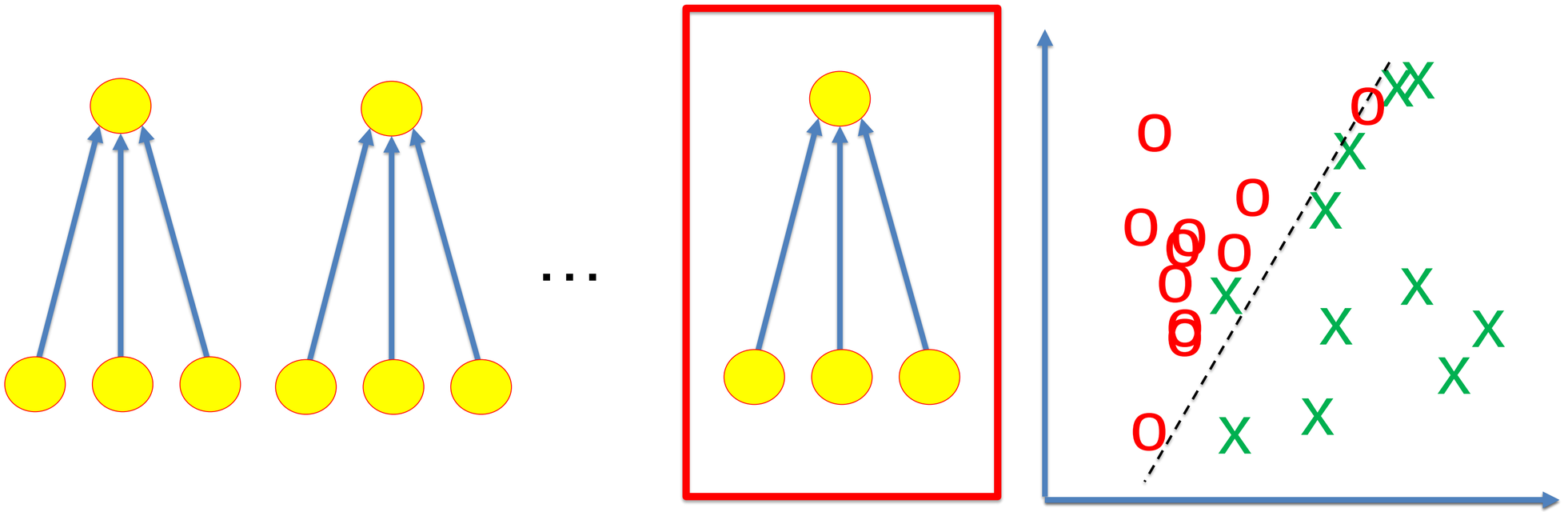
Previous slide.

… or the second one

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

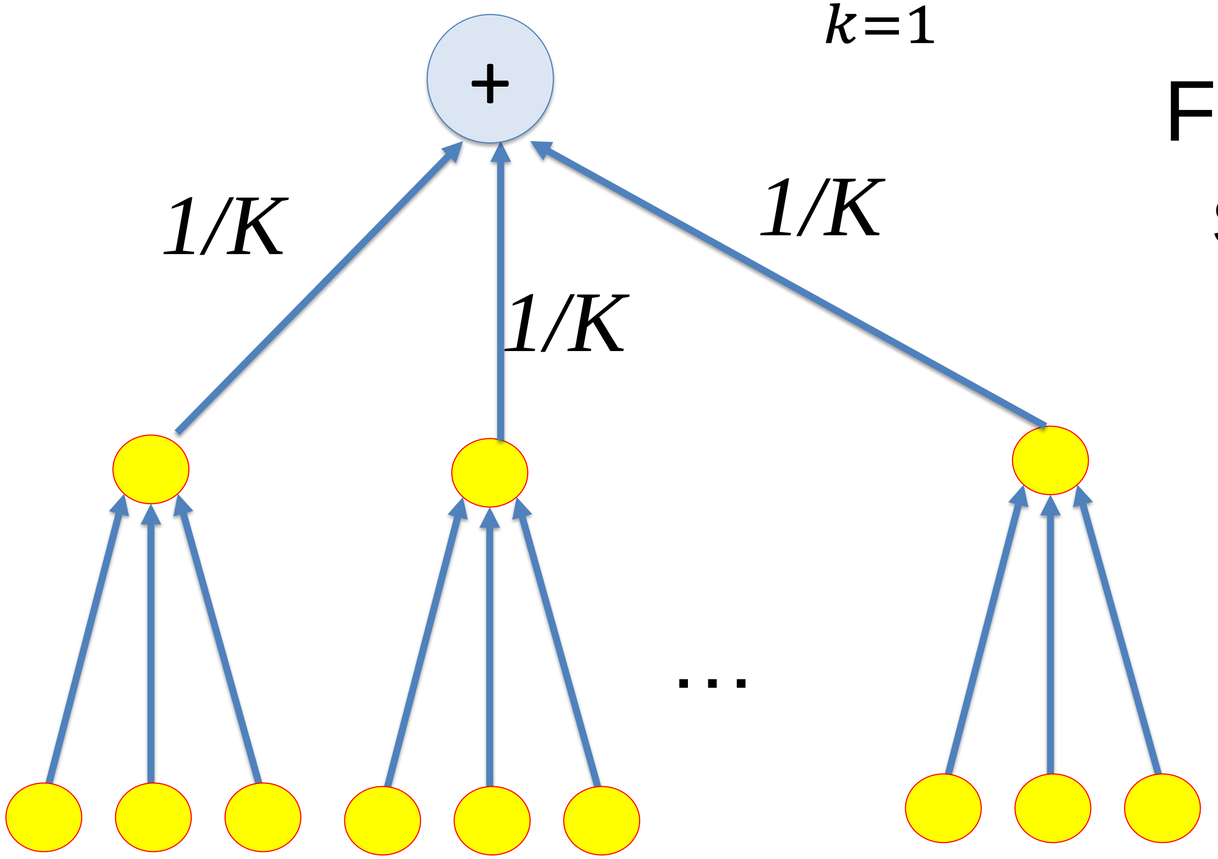


Previous slide.

… or the last one.

Bagging Idea: (iii) Average over all K variants

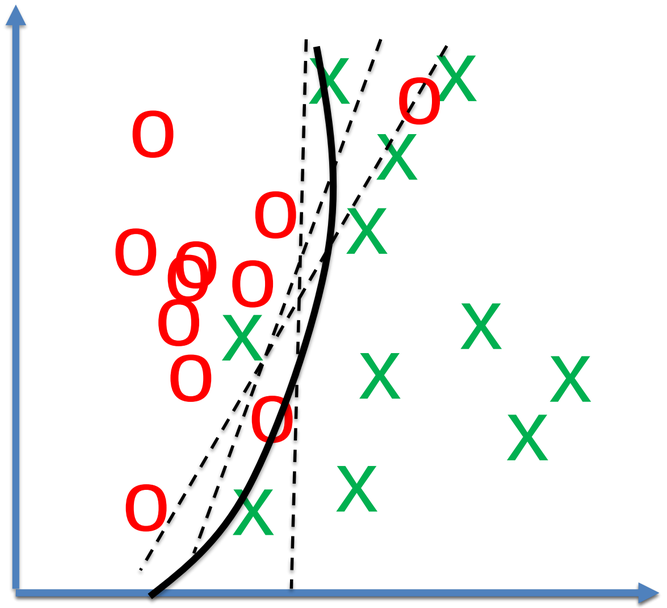
$$\hat{y}_{bag} = \frac{1}{K}\sum_{k=1}^{K}\hat{y}_k$$



For classification tasks:

$\hat{y}_{bag}$>0.5 = majority vote step

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.

For classification task, the value $\hat{y}_{bag}$ must be compared with a threshold to assign the class. If we use a threshold of 0.5, then we can think of bagging as ‘majority’ vote across the ensemble outputs.

Note that $\hat{y}_{bag}$ can implement more than just a separating hyperplane! This is not surprising since bagging corresponds to adding an extra layer to the original smaller network.

Bagging : Algorithm

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

1 Generate K different training sets

for $k=1,\ldots,K$

pick $\boldsymbol{P1}$ times into your data set with replacement

(your 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 $k=1,\ldots,K$

put input $\boldsymbol{x}$ 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).

Bagging : Theory

Claim: the error of a bagged output is always smaller than that of a typical individual model

→ Part 2*: Bagging - Theory

Previous slide.

Bagging is supported by theoretical analysis. The short summary is that

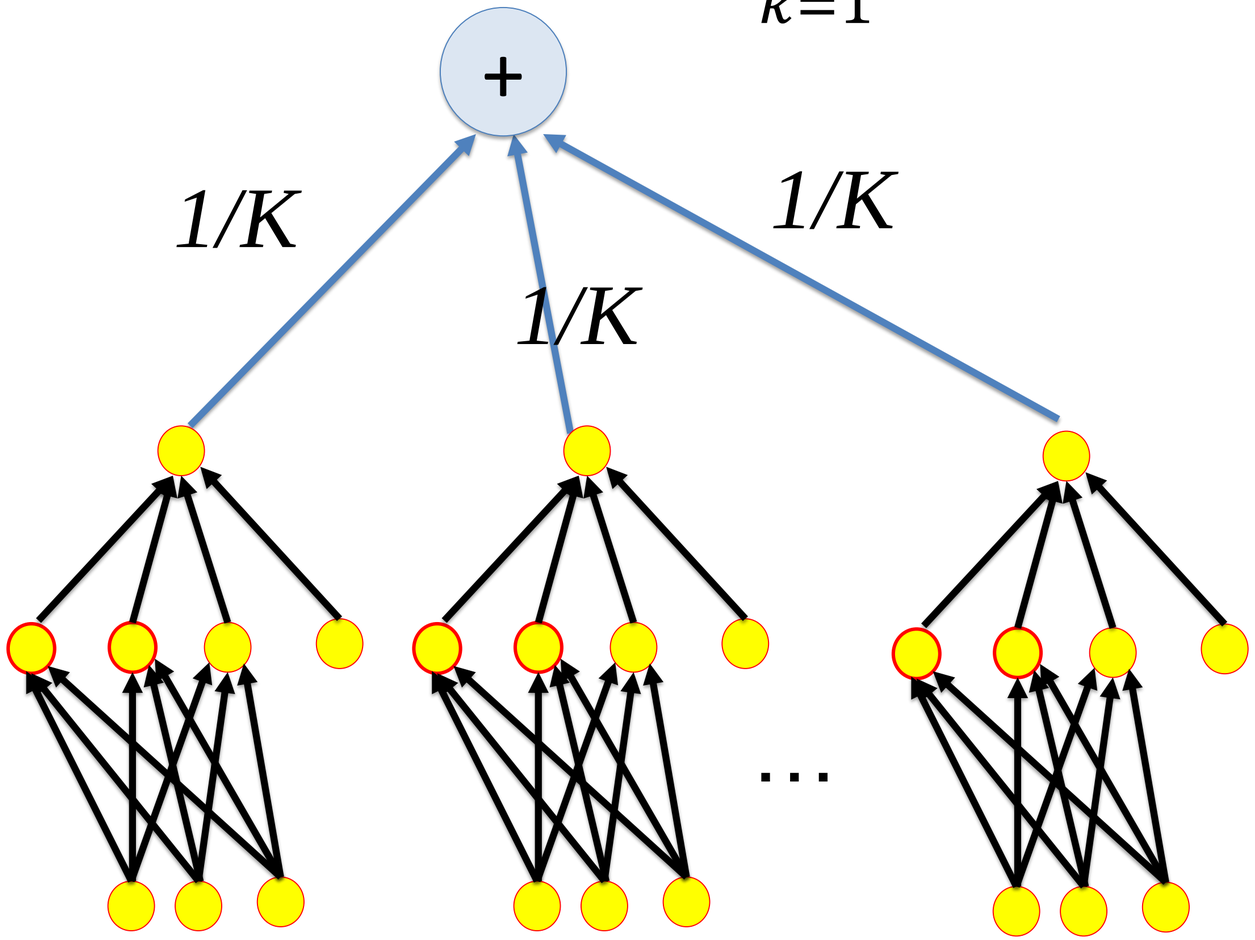
using a bagged output is always better than using the output of a single model.

There are different variants of Bagging Theorems. We will discuss a specific one later in the Theory Part (Blackboard Part).

The proof technique is closely related to the theorems of ‘bias-variance-tradeoff’.

Bagging: each of the models can be a deep network

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



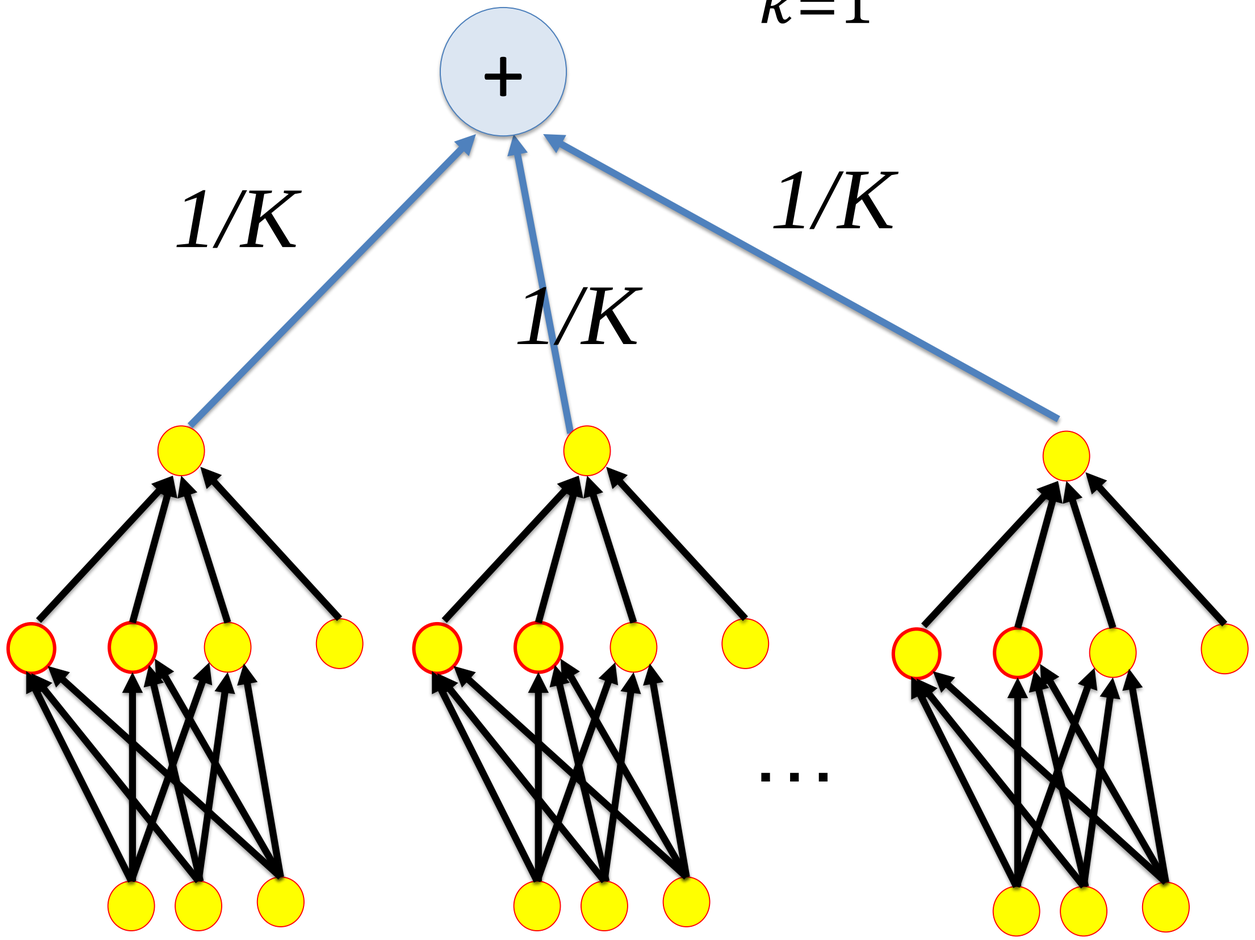
Previous slide.

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.

Bagging: each of the models sees a different data set

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



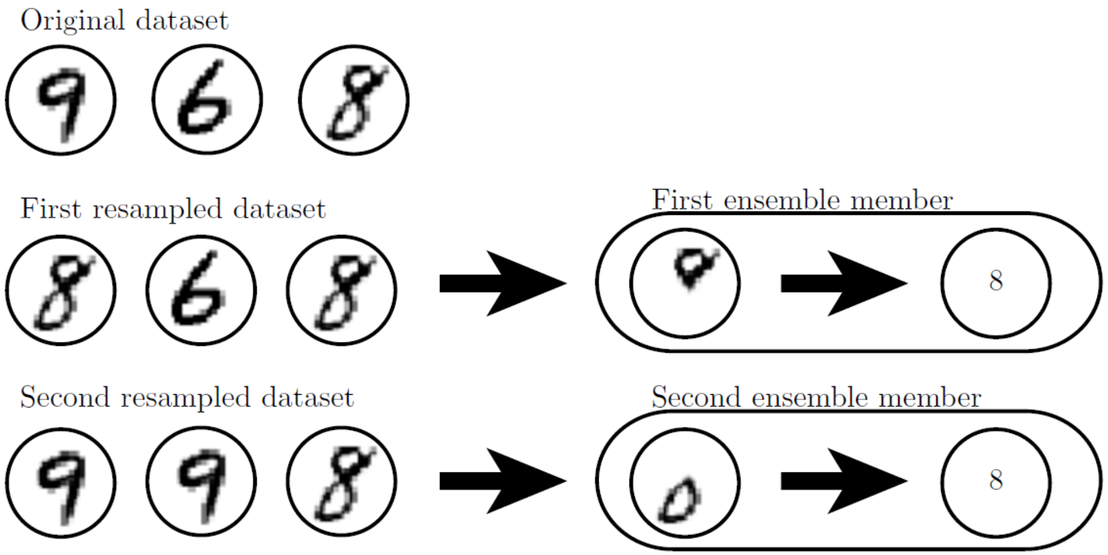


Figure 7.5

(Goodfellow 2016)

Goodfellow et al.
Deep Learning, 2016

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 (assuming that you hesitate between
these ten because they look roughly equally good),
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

Tricks of the Trade in Deep Learning

Part 2*: Bagging - Theory

Claim: the error of a bagged output is always smaller than that of a typical individual model

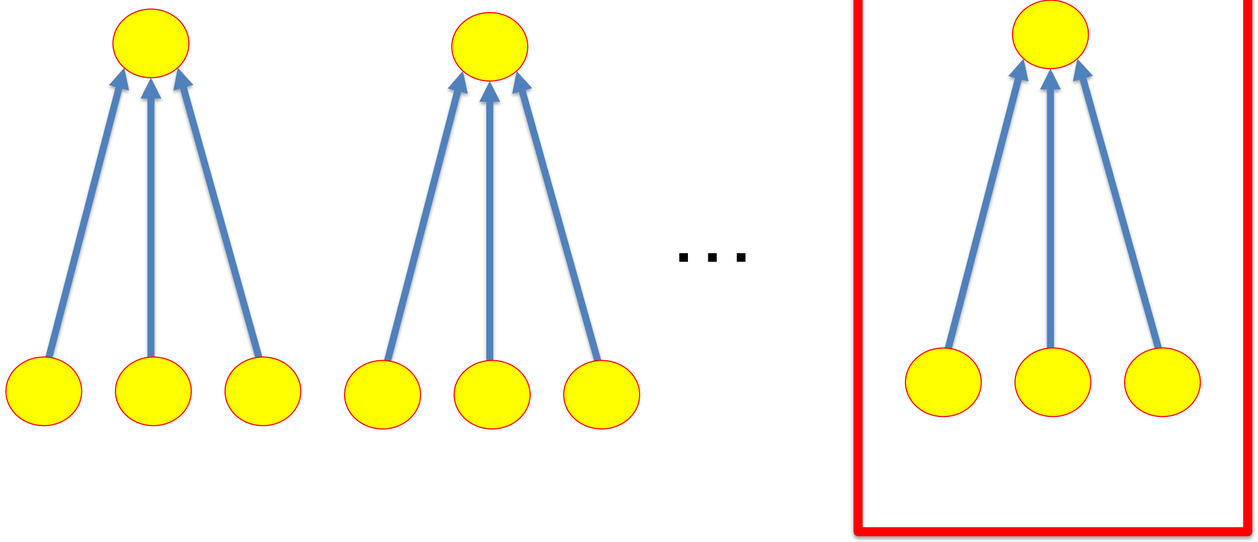
Bagging: Preparation for Theory

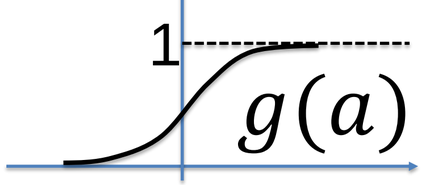
Model k

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

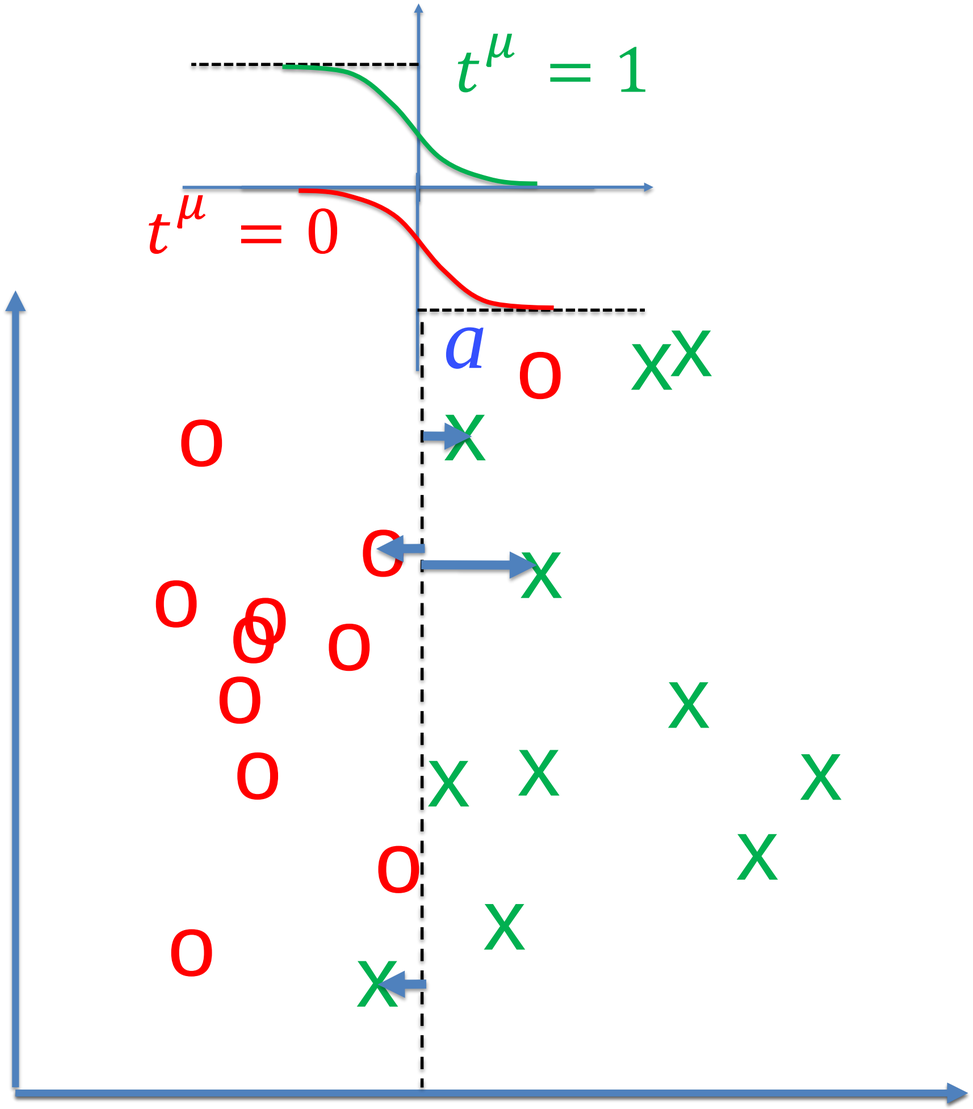
Bagged output

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





$$\delta_k^\mu = t^\mu - \hat{y}_k^\mu = t^\mu - g(a_k^\mu)$$



Bagging: Theory

Blackboard: Bagging 1A (preparation)

Model k

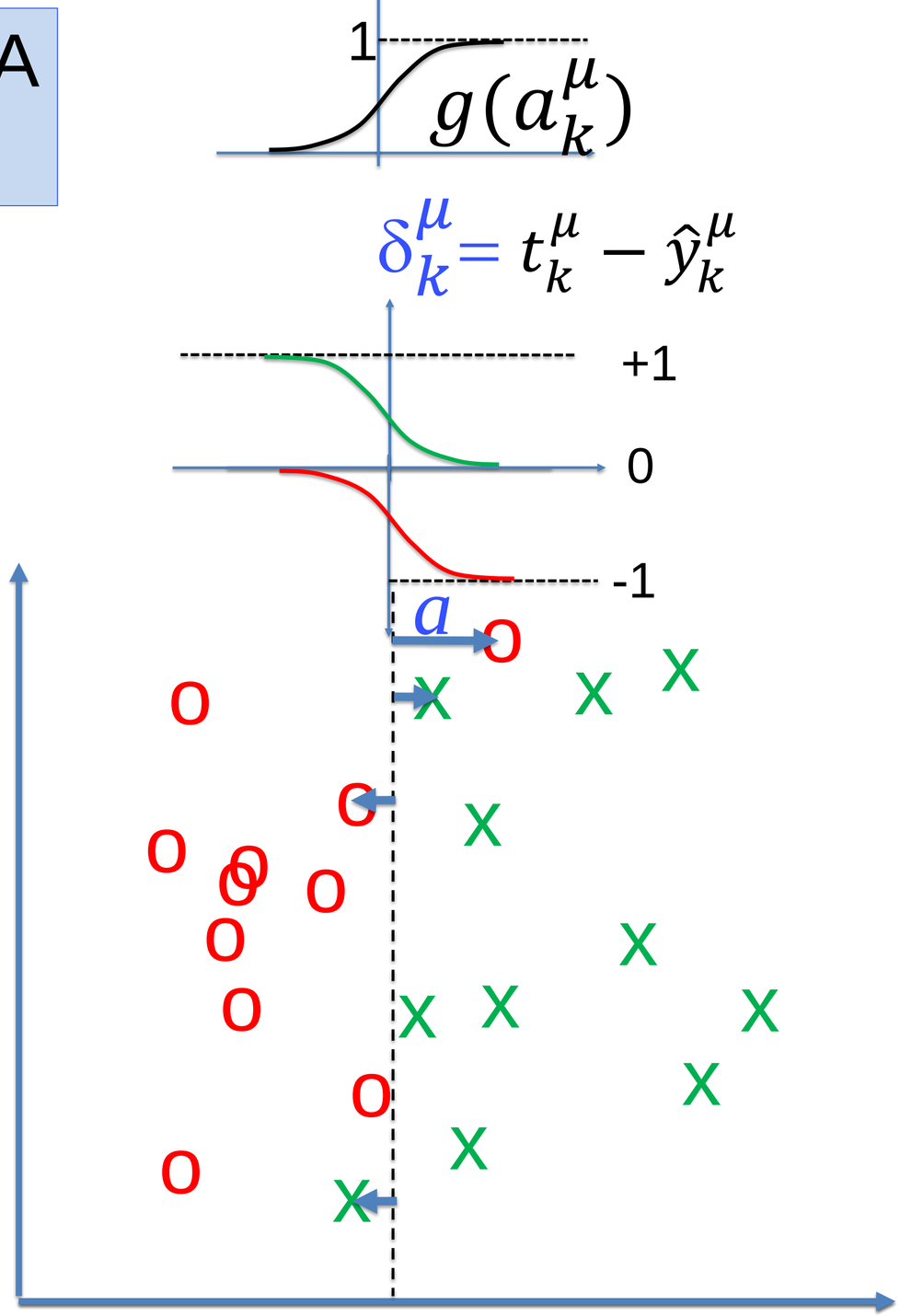
$$\hat{y}_k = 0.5[1 + tanh(\sum_j w_j x_j - \vartheta)]$$

Signed mismatch in output (copy k)

$$\delta_k^\mu = t^\mu - \hat{y}_k^\mu \qquad (1)$$

Total error in output (copy k)

$$E_k = \frac{1}{P}\sum_{\mu=1}^{P} [\delta_k^\mu]^2$$



Previous slide and next slide.
Bagging is supported by a theoretical analysis.

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

We introduce the signed difference $\delta_k^\mu = t^\mu - \hat{y}_k^\mu = t^\mu - g(a_k^\mu)$

which is some function of the distance a of this data point from the separating hyperplane. Positive samples have $0 \leq \delta_k^\mu \leq 1$ and negative samples $-1 \leq \delta_k^\mu \leq 0$.
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: $\mathrm{E}_k = \frac{1}{P}\sum_{\mu=1}^{P}[\delta_k^\mu]^2$

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

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

Bagging : Theory

Compare total quadratic error in a typical copy k with total quadratic error of bagged output

Output of copy k

$\hat{y}_k^{\mu}$

Bagged output

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

Signed mismatch

$$\delta_k^{\mu} = t^{\mu} - \hat{y}_k^{\mu} \qquad (1)$$

Signed mismatch (bagged)

$$\delta_{bag}^{\mu} = t^{\mu} - \hat{y}_{bag}^{\mu} \qquad (1')$$

Total error in output (copy k)

$$\mathrm{E}_k = \frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_k^{\mu}\right]^2$$

Total error in bagged output

$$\mathrm{E}_{bag} = \frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_{bag}^{\mu}\right]^2$$

Bagging : Theory

Blackboard: Bagging 1B

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

Theorem

Assumption: the average delta-difference (‘bias’), defined as

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

is the same for all K copies of the model. THEN

Typical error in individual $\geq$ **Total error in bagged output**

$$\left\langle \mathrm{E}_k \right\rangle_k = \left\langle \frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_k^{\mu}\right]^2 \right\rangle_k \geq \mathrm{E}_{bag} = \frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_{bag}^{\mu}\right]^2$$

Proof Sketch (1) : typical error in individual copy

Signed mismatch (delta-difference)

$$\delta_k^\mu = t^\mu - \hat{y}_k^\mu \qquad (1)$$

Assumption:

the average delta-difference (bias),

$$\frac{1}{P}\sum_{\mu=1}^{P}[\delta_k^\mu] = \mathrm{d} \qquad (2)$$

is the same for all K copies. Hence

quadratic error (averaged over copies)

$$\langle \mathrm{E}_k \rangle_k = \frac{1}{K}\sum_k \left(\frac{1}{P}\sum_{\mu=1}^{P}[\delta_k^\mu]^2\right)$$

$$= \frac{1}{K}\sum_k \left(\frac{1}{P}\sum_{\mu=1}^{P}[\varepsilon_k^\mu + d]^2\right)$$

$$= \mathrm{V} + \mathrm{d}^2$$

shift to mean zero

$$\varepsilon_k^\mu = \delta_k^\mu - d \qquad (3a)$$

$$\frac{1}{P}\sum_{\mu=1}^{P}[\varepsilon_k^\mu] = 0 \qquad (3b)$$

with variance

$$\frac{1}{K}\sum_{k=1}^{K}\frac{1}{P}\sum_{\mu=1}^{P}[\varepsilon_k^\mu]^2 = \mathrm{V} \qquad (4)$$

(*) “typical error in individual copy"

Next slide and previous slide.
So far we have evaluated the typical error of a single copy.
We will now evaluate the quadratic error of bagged output. The term surrounded by the red box disappears if all the ε are independent.

Proof Sketch (2): After bagging

Quadratic error of bagged output is

$$\mathrm{E}_{bag} = \frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_{bag}^{\mu}\right]^2$$

$$\mathrm{E}_{bag} = \frac{1}{P}\sum_{\mu=1}^{P}\left[\varepsilon_{bag}^{\mu} + d\right]^2$$

$$\mathrm{E}_{bag} = \frac{1}{K}\mathrm{V} + \mathrm{d}^2 + \frac{1}{P}\sum_{\mu=1}^{P}\frac{1}{K}\sum_{k}\varepsilon_{k}^{\mu}\,\frac{1}{K}\sum_{m\neq k}\varepsilon_{m}^{\mu}$$

$$\delta_{bag}^{\mu} = t^{\mu} - \hat{y}_{bag}^{\mu} \qquad (1')$$

$$\varepsilon_{bag}^{\mu} = \delta_{bag}^{\mu} - d \qquad (3'a)$$

$$\varepsilon_{bag}^{\mu} = \frac{1}{K}\sum_{k}\varepsilon_{k}^{\mu} \rightarrow$$

$$\frac{1}{P}\sum_{\mu=1}^{P}\varepsilon_{bag}^{\mu} = 0 \qquad (3'b)$$

Compare with (*) “typical error in single copy”:

$$\left\langle \mathrm{E}_k \right\rangle_k = \frac{1}{K}\sum_{k}\left(\frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_{k}^{\mu}\right]^2\right)$$

$$= \mathrm{V} + \mathrm{d}^2$$

Summary: Bagging Theory

Next Lecture at 12h20

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 (same bias for all).

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

Notes on Theory Result

1. Assumption: the average delta-difference, defined as

$$\frac{1}{P}\sum_{\mu=1}^{P}\left[\delta_{k}^{\mu}\right] = d$$

is the same (the assumption is **not** for the quadratic error)

2. With a good error function d might even vanish.

3. Uncorrelated data is necessary for error reduction.
   Having more independent data is always better,
   but resampling is a good trick to construct
   ‘somewhat independent’ data sets from finite data.

4. Our theorem does not include the majority vote at the end

Previous slide.
Bagging is an example of ‘Ensemble’ methods which is a widely used and rather generic principle. The upshot is that averaging over individual, simple models, always helps if the data used for each individual are independent. Ideally, independence should be strict; but resampling different subsets from a fixed finite data set is often a good approximation to independence.

In the proof we assumed a typical signed distance d after convergence. However, with a good combination of error function and neuronal output function, d might even vanish (e.g., quadratic error combined with linear output). In the context of bias-variance trade-off d would be called the ‘bias’.

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).
BUT: 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.**

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Wulfram Gerstner

EPFL, Lausanne, Switzerland

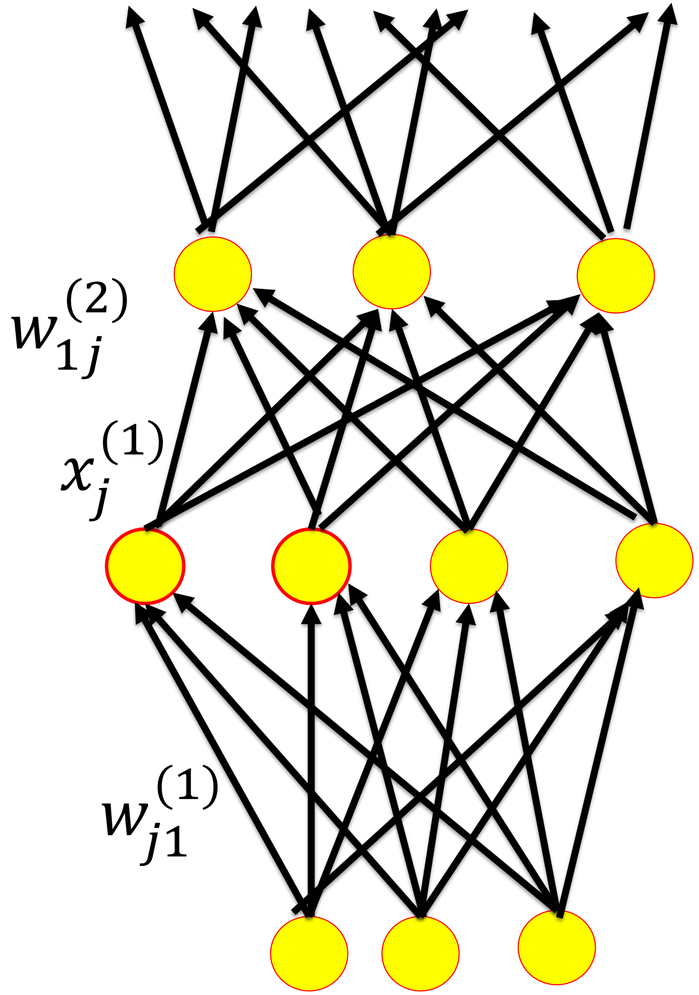
Part 3: Dropout

1. Questions and aims
2. Bagging
3. Dropout

Previous slide.

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

Dropout: suppress 50 percent of hidden units during training

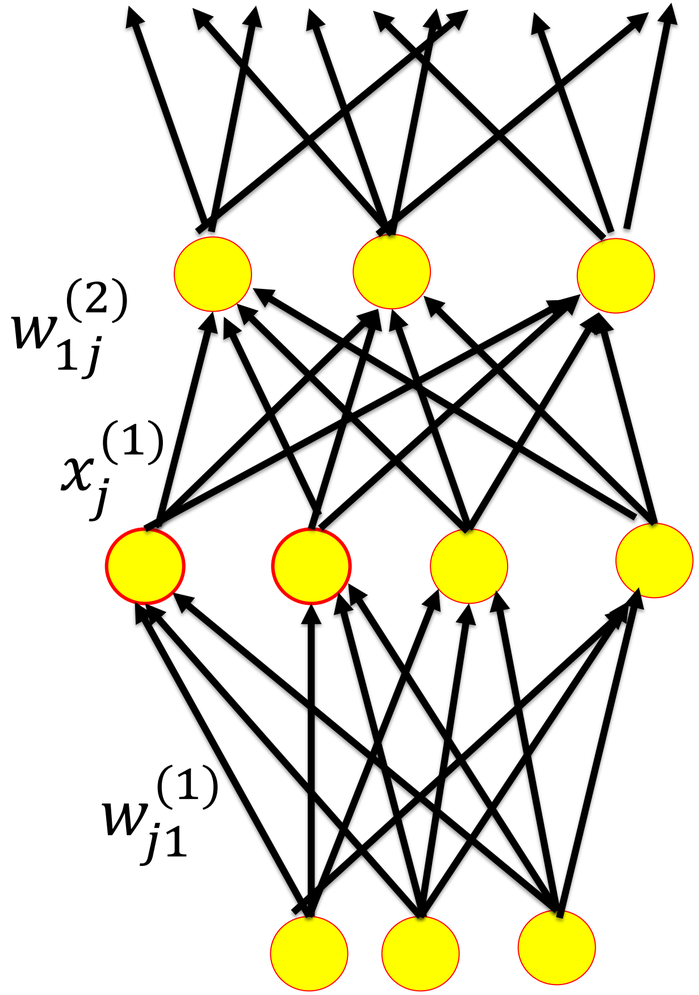


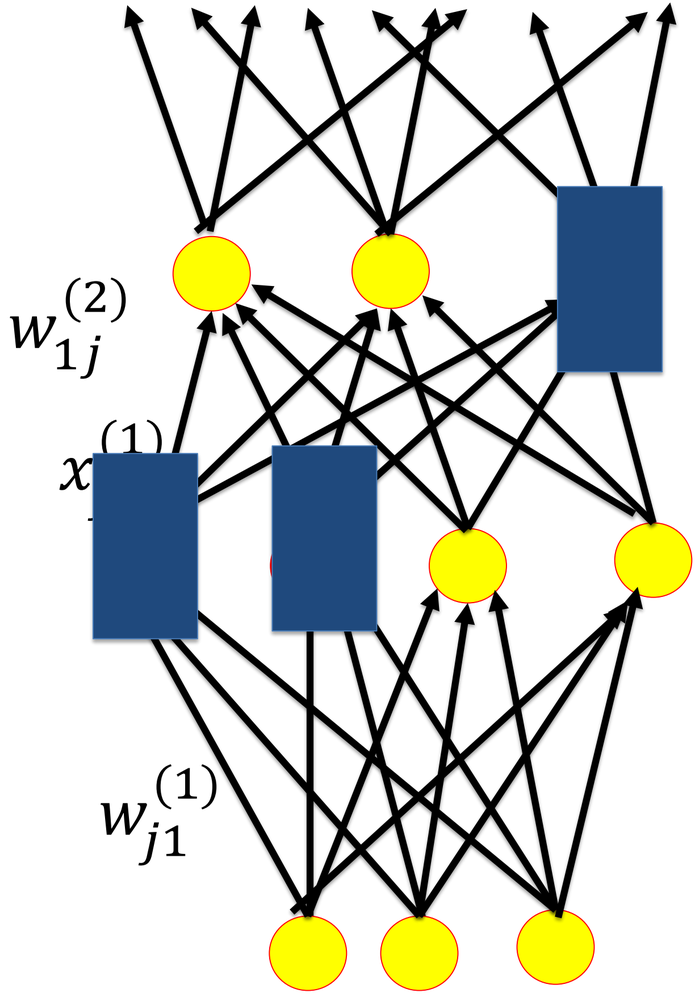
Previous slide.

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.

Dropout: suppress 50 percent of hidden units during training

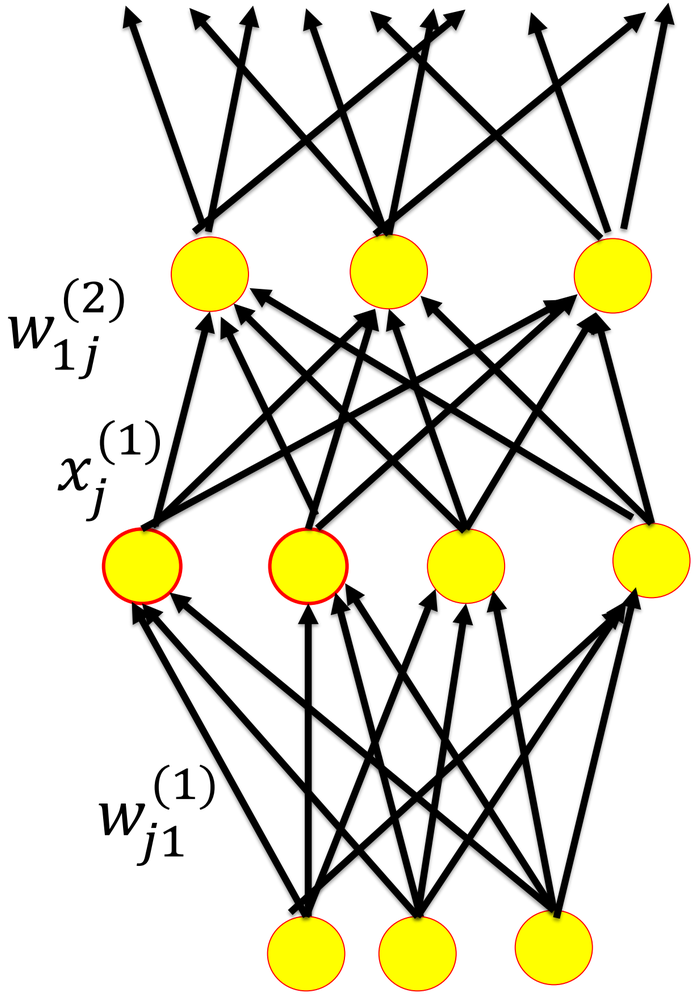


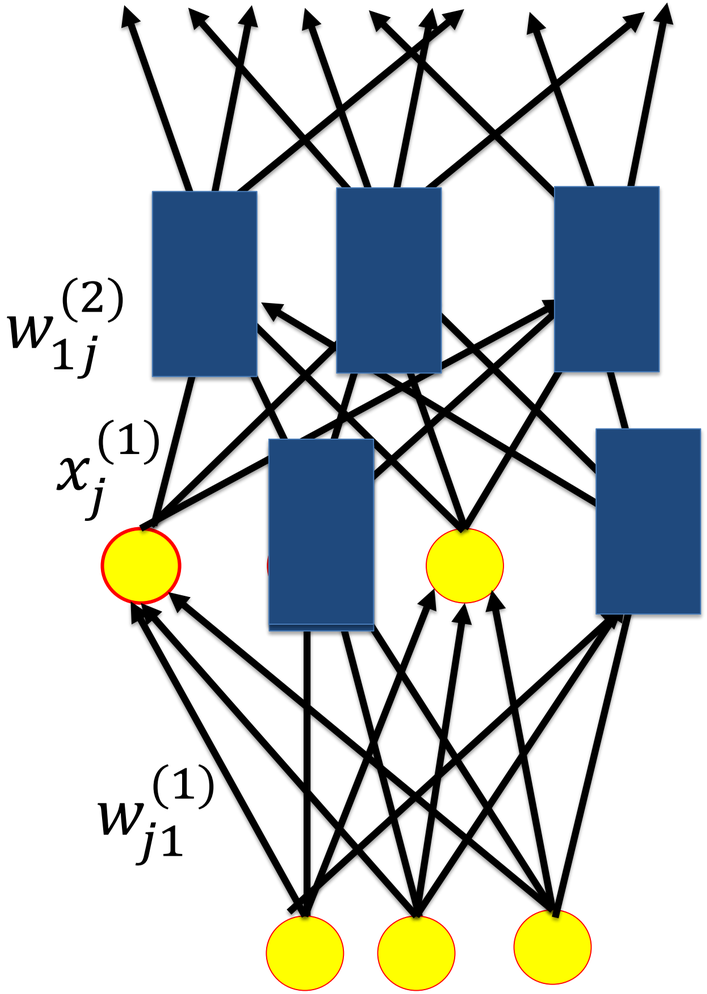


Previous slide.

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

Dropout: suppress 50 percent of hidden units during training

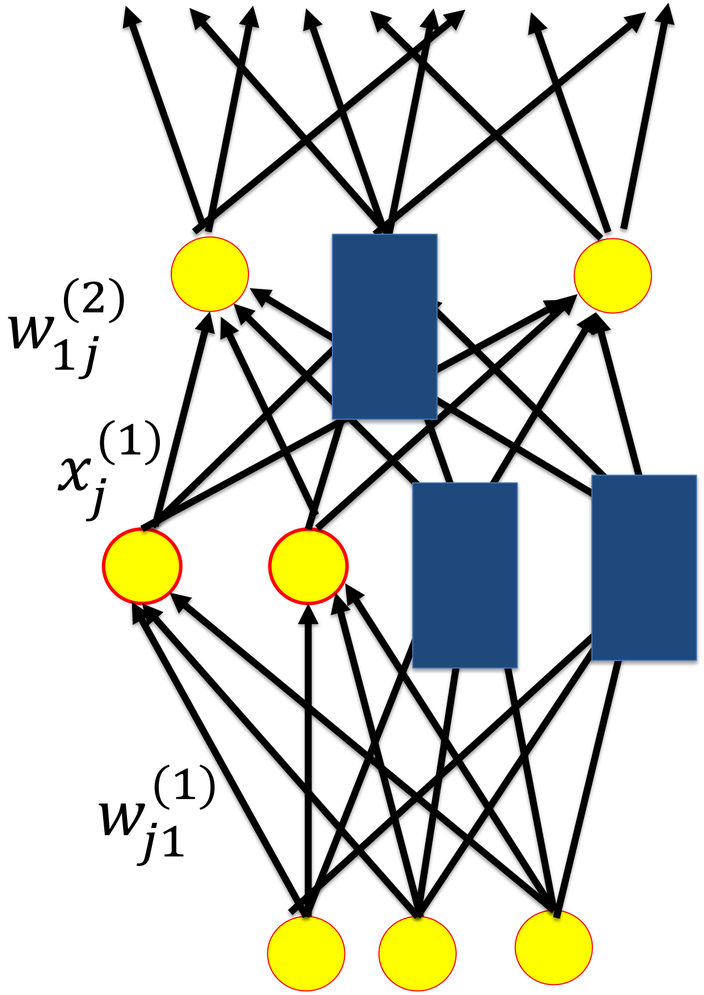




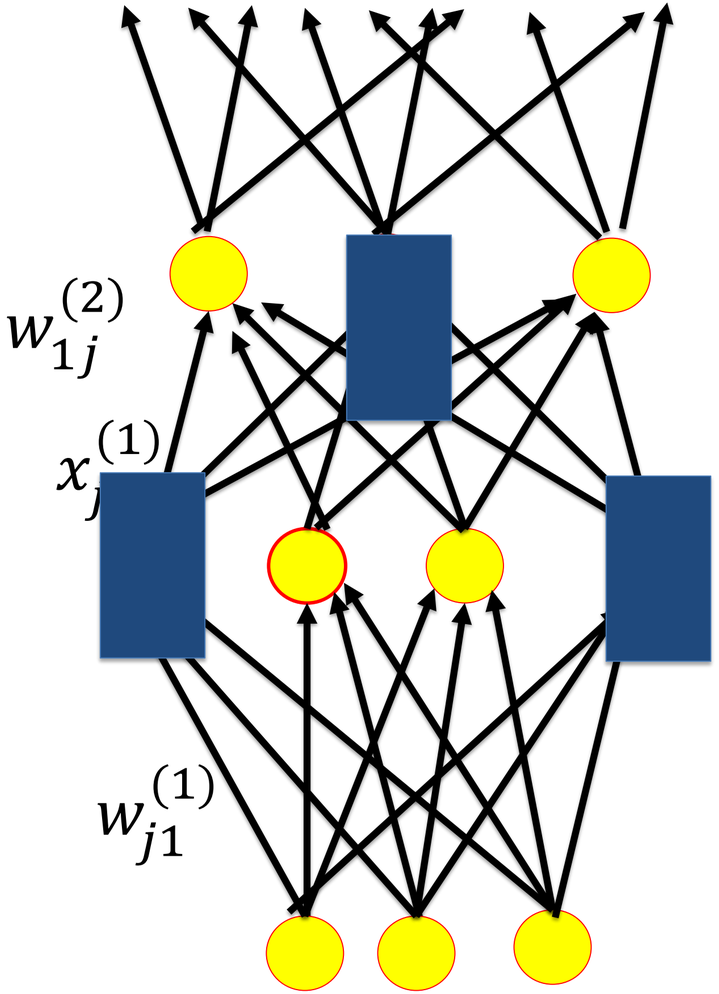
Previous slide.

And for pattern number $\mu+1, \mu+2, \square$ you randomly pick each time a different subset of hidden units which you remove (their outputs are set to zero).

Dropout: suppress 50 percent of hidden units during training



$\boldsymbol{x} \in R^{N+1}$

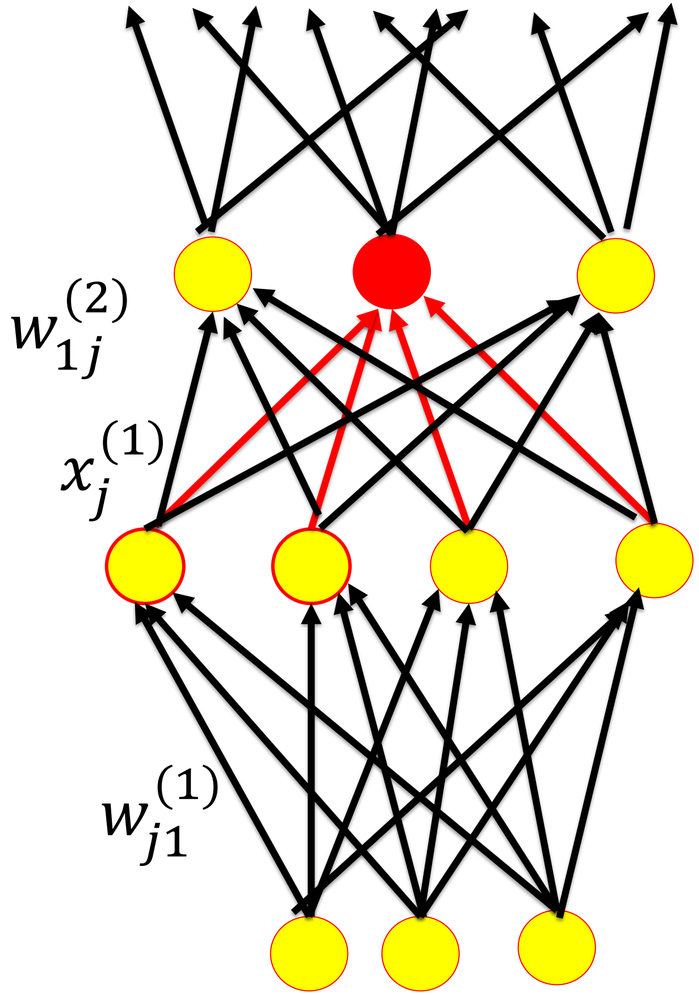


$\boldsymbol{x} \in R^{N+1}$

Previous slide.

You train over many epochs.

Dropout: use full network for validation and test



For test:

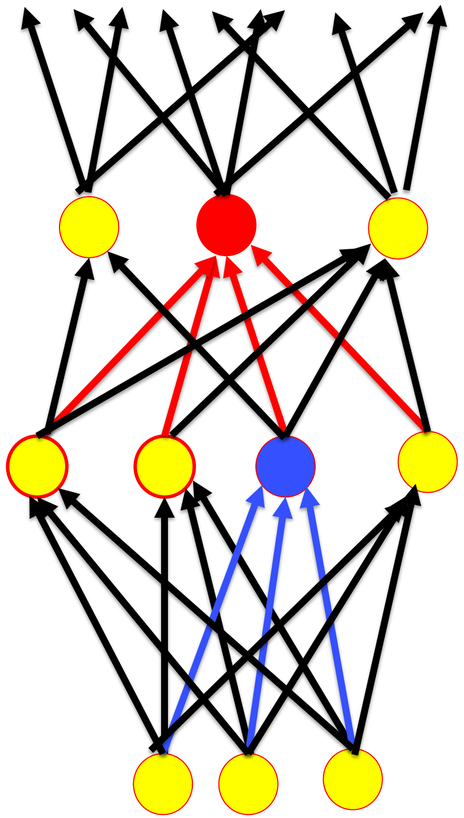
- full network
- but multiply output weights from hidden units by 1/2

→ 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 ½, so that the typical input to a unit in the next layers is roughly the same as during training.

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 widely used in Deep Artificial Neural Networks. There are two different interpretations of why dropout works.

Here is the first one:
Dropout can be seen as approximate bagging

Dropout as approximate bagging

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

Differences to standard bagging:

- bagging not just for the output layer
- not a fixed data base for each ‘dropout’ configuration
- models are not independent: share weights
- output not a ‘average’ or ‘majority vote’ over model outputs

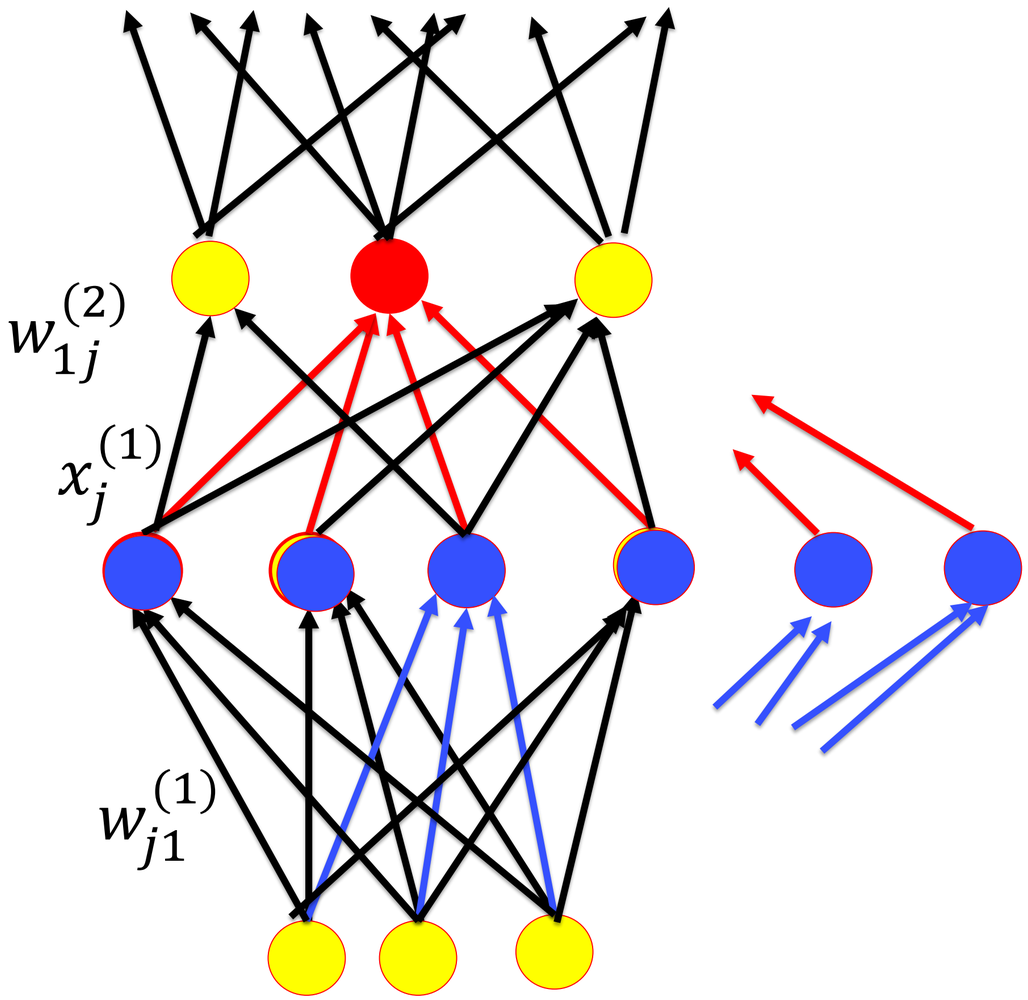
Previous slide.

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

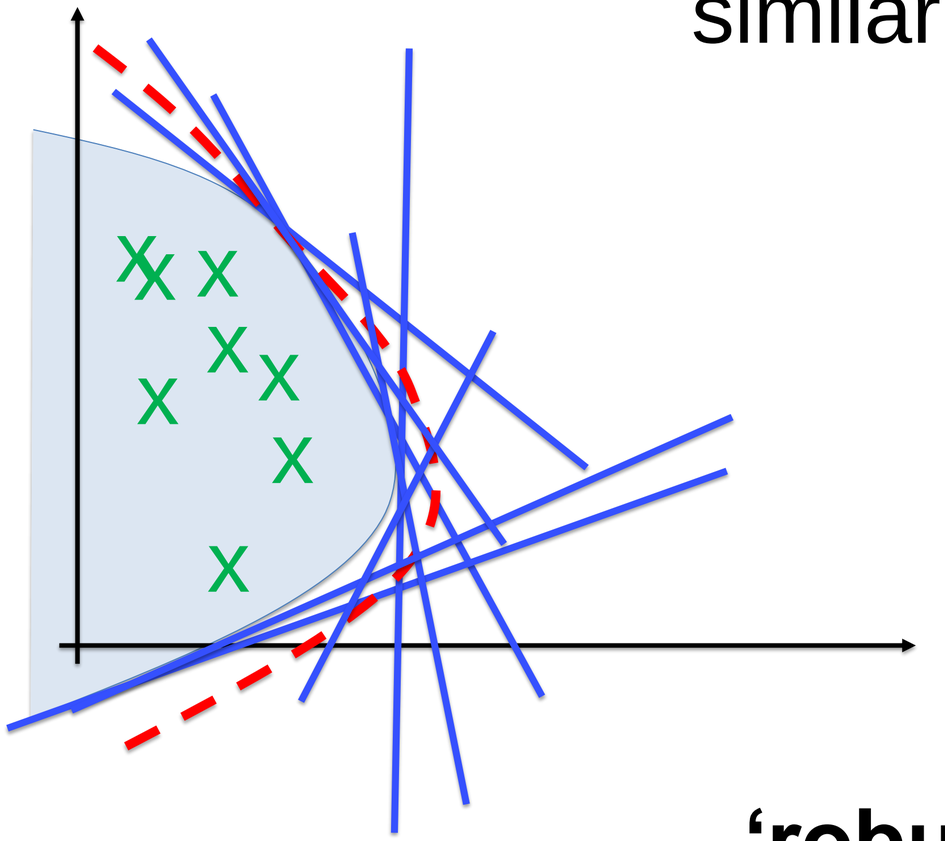
Note that dropout implements ideas of bagging not just for the output layer, but also for neurons in the hidden layer.
Further differences to standard bagging are:

1. **not a fixed data base for each ‘dropout’ configuration.**
   In a network with N_h hidden neurons, there are
   $(N_h\ !\ )/[\ (N_h/2)!\ (N_h/2)!\ ]$ different dropout configuration.
   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.

Dropout as forced feature sharing



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



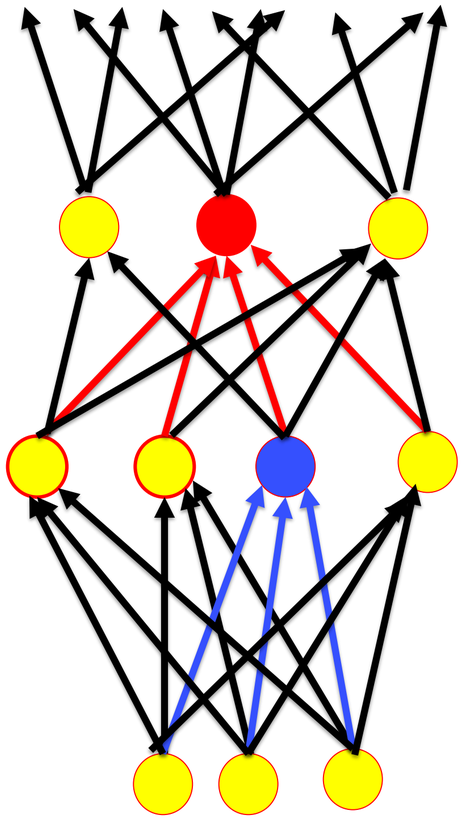
‘robust’

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 input to the red neurons is a weighted average over the output of the blue neurons. Together with the nonlinearity of the red neurons this corresponds to a nonlinear 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.

Summary: two different interpretations of Dropout



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 summary, there are two different interpretations of Dropout.

The first one sees Dropout as an approximate version of Bagging, suitable for deep networks.

The second one highlights that Dropout enforces feature sharing between different hidden neurons in the same layer

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

Teaching monitoring – monitoring of understanding

[ ] today, up to here, at least 60% of material was new to me.

[ ] up to here, I have the feeling that I have been able to follow (at least) 80% of the lecture.

(previous slide)

Exercise: Dropout (example)

Next Lecture at 14h15

We have a deep network of $2n$ hidden layers ($n > 2$) of neurons with sharp threshold functions $g(a) = 1$ for $a > 0$ and zero otherwise. After training with dropout, somewhere in hidden layer n, we have a hidden neuron i which receives input from 4 hidden neurons in layer $n-1$. All weights onto neuron i are equal to one and the threshold of neuron i is 2.7.

Each of the four hidden neurons j in layer $n-1$ receives input from the same 2 neurons in layer $n-2$. The weight vectors and thresholds of the four neurons in layer $n-1$ are:

j=1 (1,0) and threshold 0
j=2 (1,0) and threshold 0.5
j=3 (1,1) and threshold 1
j=4 (1,-1)and threshold 1

a. Qualitatively sketch the two-dimensional space representing the activity of the 2 neurons in layer $n-2$ and indicate the region (by shading it with crosses x x x) in which neuron i responds positively.

b. Dropout: Remove neurons $j = 1$ and $j = 4$ in layer $n-1$, rescale the weights appropriately, and sketch the input space where neuron i responds positively (by shading it with crosses x x x).

c. Dropout: Remove neurons $j = 2$ and $j = 3$ in layer $n-1$, rescale the weights appropriately, and sketch the input space where neuron i responds positively (by shading it with crosses x x x).

d. Your friend Adam claims: 'Dropout might be a useful trick, but nobody understands how it works'. Your friend Berthilde claims 'Dropout is good for generalization and easy to understand'. Comment on your results (think also of the other 4 combinations of dropping out two neurons) and relate your results to the claims of your friends.

notes

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Wulfram Gerstner

EPFL, Lausanne, Switzerland

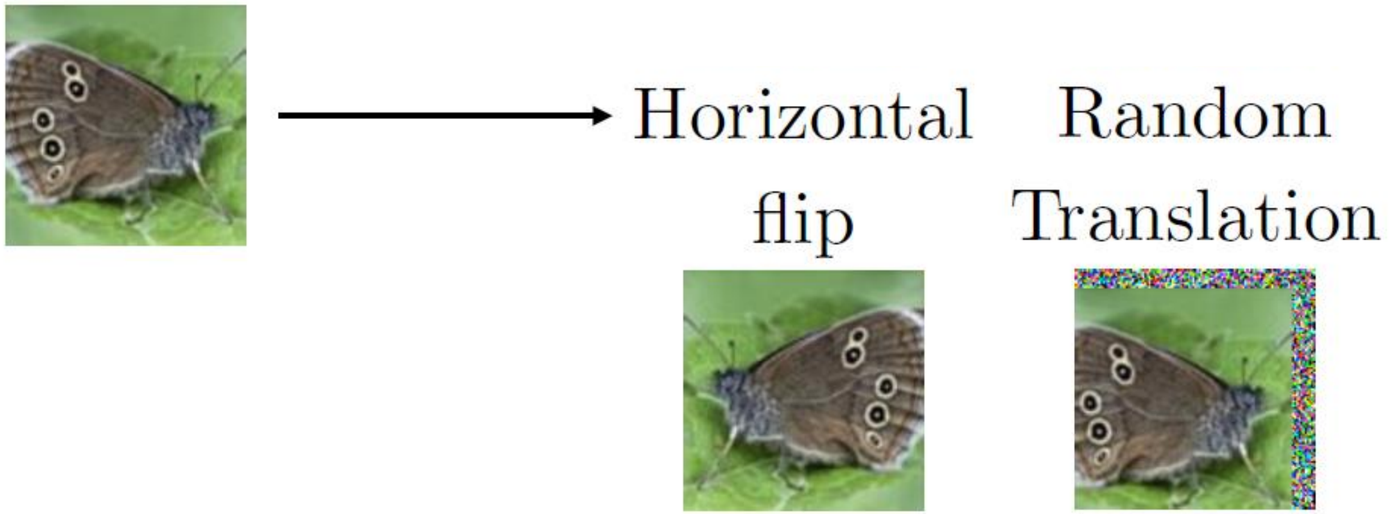
Part 4: Data augmentation

1. Questions and aims
2. Bagging
3. Dropout
4. Data augmentation

Previous slide.

Data augmentation is a very effective regularization method and comes at relatively low cost. **This part has not been shown in class.**

Dataset augmentation: Translations and Rotationas



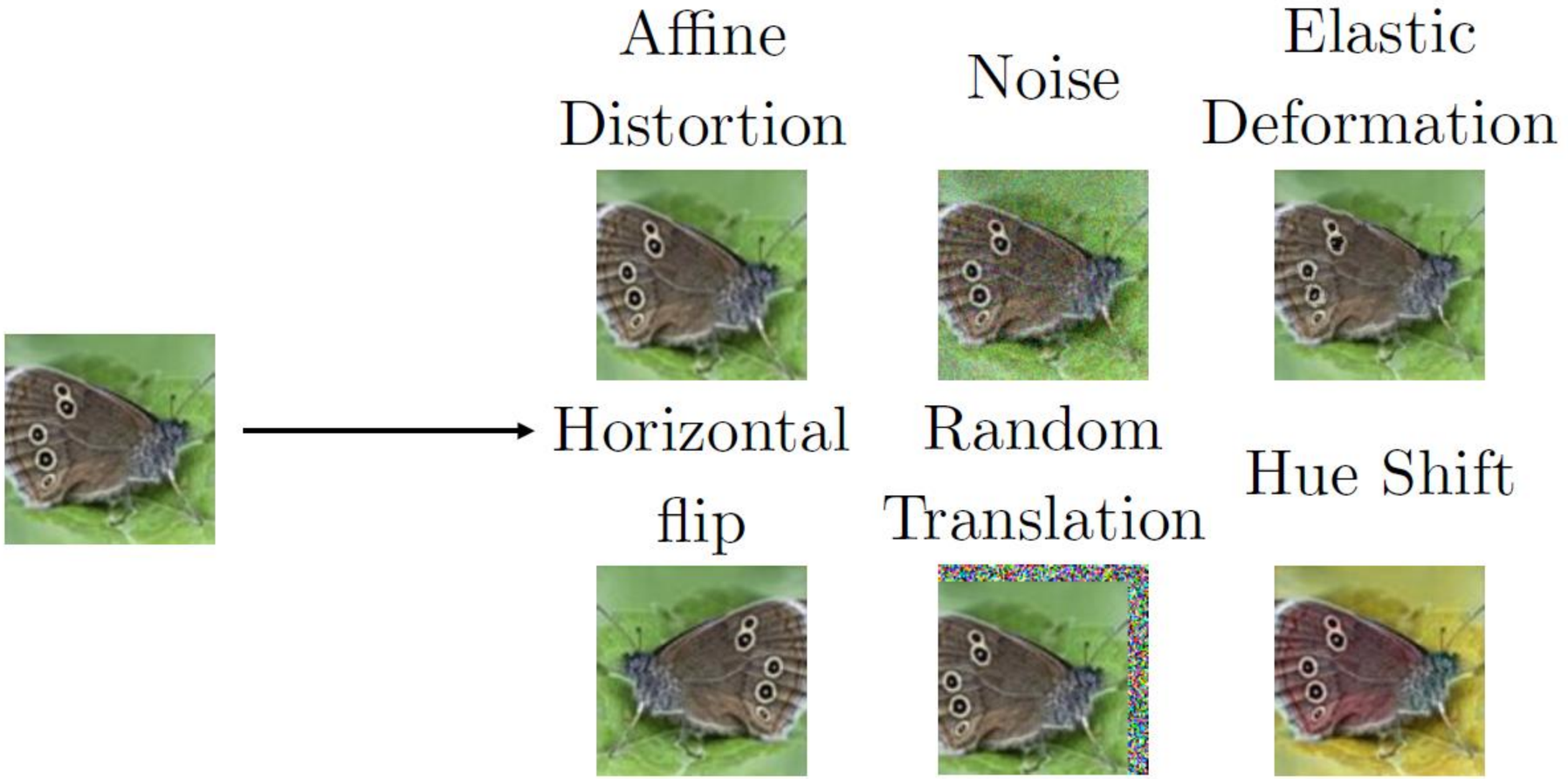
Deep Learning, Goodfellow et al. 2016

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).

Dataset augmentation: Translations and Rotations



Deep Learning, 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.

Dataset augmentation: artistic styles

Style

Previous slide.

But there are other ways to do data augmentation on images. We may exploit that humans recognize not only objects and scenes from photographs, but also from artistic images.

For example, we recognize a boathouse in the country, or a face, or the sky in the night, even if these are stylized paintings.

Dataset augmentation: artistic styles

https://deepart.io/

Gatys, Ecker, Bethge "A Neural Algorithm of Artistic Style" (2015).

Previous slide.

Therefore, we can use as training data for object recognition not just original photos, but also transformed versions of these photos.
With AI methods, stylized versions of images can be automatically generated.

Dataset augmentation

Augment images from ImageNet
with stylized versions, generated automatically

Robert Geirhos et al., ICLR, 2019

thousands of potential styles for each image!

easy original images → difficult stylized images (same label)

Previous slide.

When trained on stylized versions of images (automatically generated from the originals and a random artistic style), the object recognition of deep networks becomes much more stable, and more comparable to humans, compared to the case where only original image in ImageNet were used.

Summary: Data augmentation as regularization

- Data set augmentation is a regularization method
- Data augmentation should reflect all known invariances of the task
- Labels for augmented images are ‘copied’ from originals
- Not just images, also for other input vectors
- Generated automatically by algorithms

Previous slide.

Data set augmentation is a regularization method, because it avoids overfitting

Data set augmentation is effective, ant not costly in terms of human labor: we just use machine time to generate the additional images using some algorithms

And of course, it also is possible outside the domain of images as inputs.

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Wulfram Gerstner

EPFL, Lausanne, Switzerland

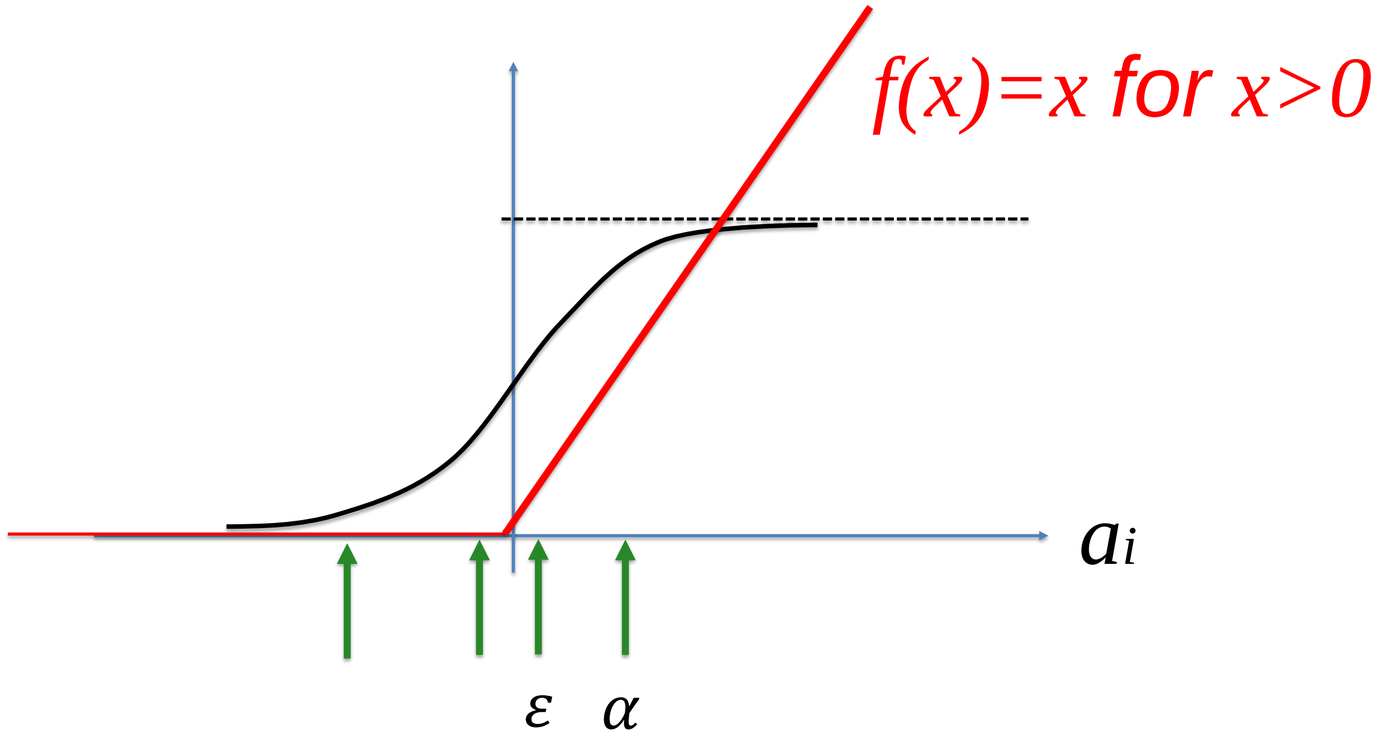
Part 5: Weight initialization and choice of hidden units

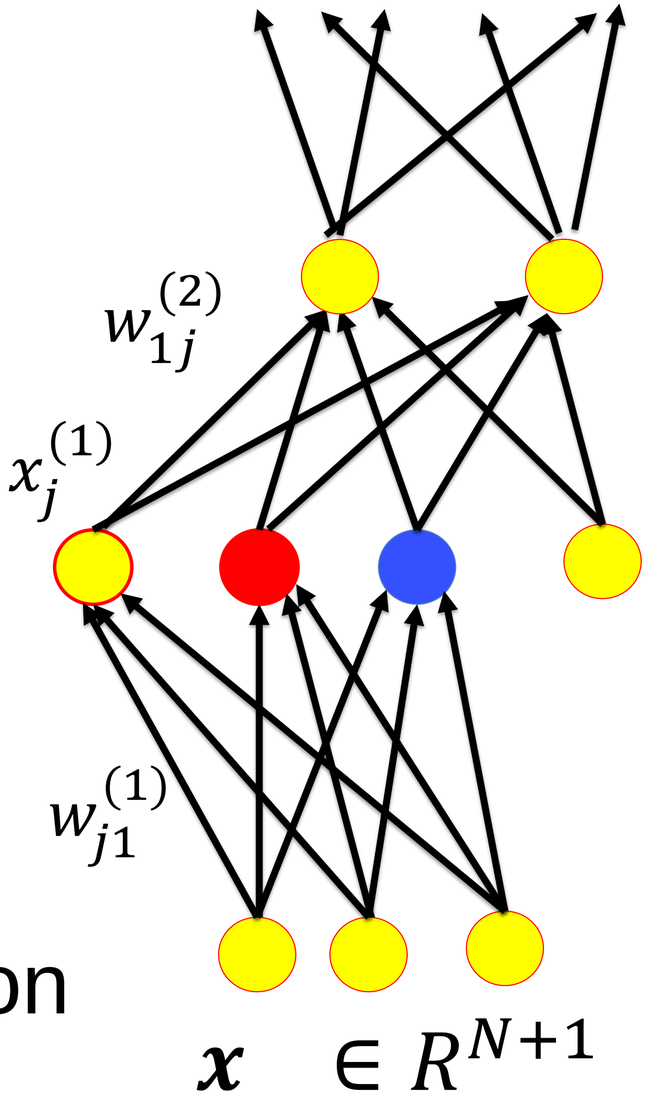
1. Questions and aims
2. Bagging
3. Dropout
4. Data augmentation
5. Weight initialization

Previous slide.

We now focus on the hidden neurons.

Choice of hidden units





-different patterns give different activation of same neuron (red)

$$a_i^{(1)} = \sum_{j=1}^{N} w_{ij}^{(1)} \; x_j^{\mu}$$

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

Initialization (input layer)

Blackboard 2

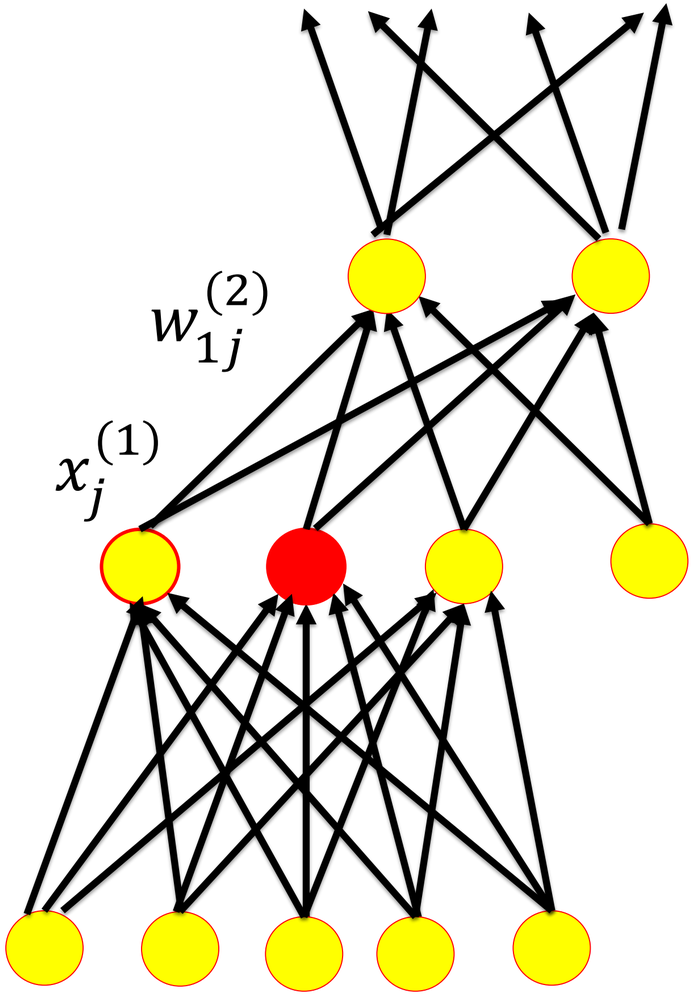
Normalization of data base (preprocessing):

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

(2) $< (x_j)^2 > = \frac{1}{P} \sum_{\mu=1}^{P} (x_j^{\mu})(x_j^{\mu}) = 1$

Random initialization of weights:

(3) $< w_{ij}^{(n)} > = 0$



How should we choose the standard deviation?

$\boldsymbol{x} \in R^{N+1}$

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.

$$< x_j >= \frac{1}{P} \sum_{\mu=1}^{P} x_j^{\mu} = 0$$

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:

$$< w_{ij}^{(n)} > = 0$$

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

Claim: square root of *N* is important

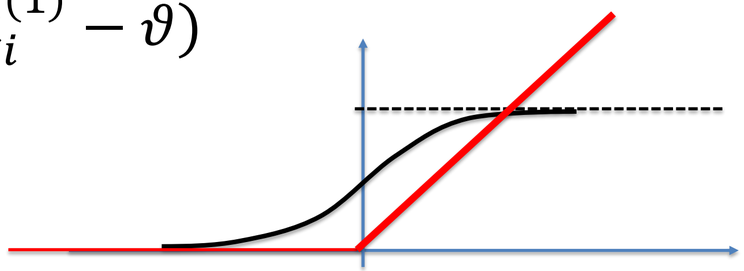
Assumptions:

(1) $\langle x_j \rangle = 0\,; \quad \left\langle \left(x_j^{\mu}\right)^2 \right\rangle = 1$ for all j

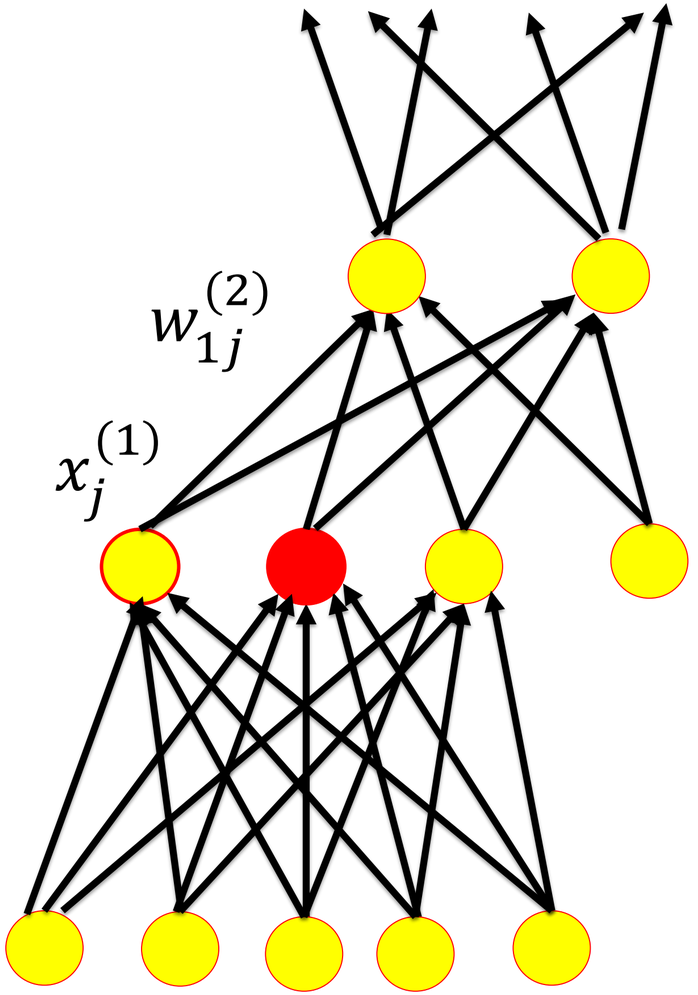
(2) $\langle w_{ij}^{(1)} \rangle = 0$

→ **Distribution of $a_i^{(1)}$ in layer 1?**

$$x_i^{(1)} = g\left(\sum_{k=1}^{N} w_{ij}^{(1)} x_j^{(0)} - \vartheta\right) = g(a_i^{(1)} - \vartheta)$$



$$a_i^{(1)} = \sum_{j=1}^{N} w_{ij}^{(1)} \; x_j^{(0)}$$



Your notes.

The average drive (activation) of neurons in layer 1 is

$$\left\langle a_i^{(1)} \right\rangle = \sum_{j=1}^{N} < w_{ij}^{(1)} >< x_j^{(0)} > = 0$$

and its variance

$$\left\langle \left(a_i^{(1)}\right)^2 \right\rangle = \sum_{j=1}^{N} < \left(w_{ij}^{(1)}\right)^2 > < \left(x_j^{(0)}\right)^2 > \qquad = N < \left(w_{ij}^{(1)}\right)^2 >$$

We initialize the variance of the weights such that the above equation yields a value of 1 or 2 or 4 (order 1).

Initialization (input layer)

Normalization of data base:

(1) $< x_j >= \frac{1}{P} \sum_{\mu=1}^{P} x_j^{\mu} = 0$; and $\left\langle x_j^2 \right\rangle = 1$

Random initialization of weights:

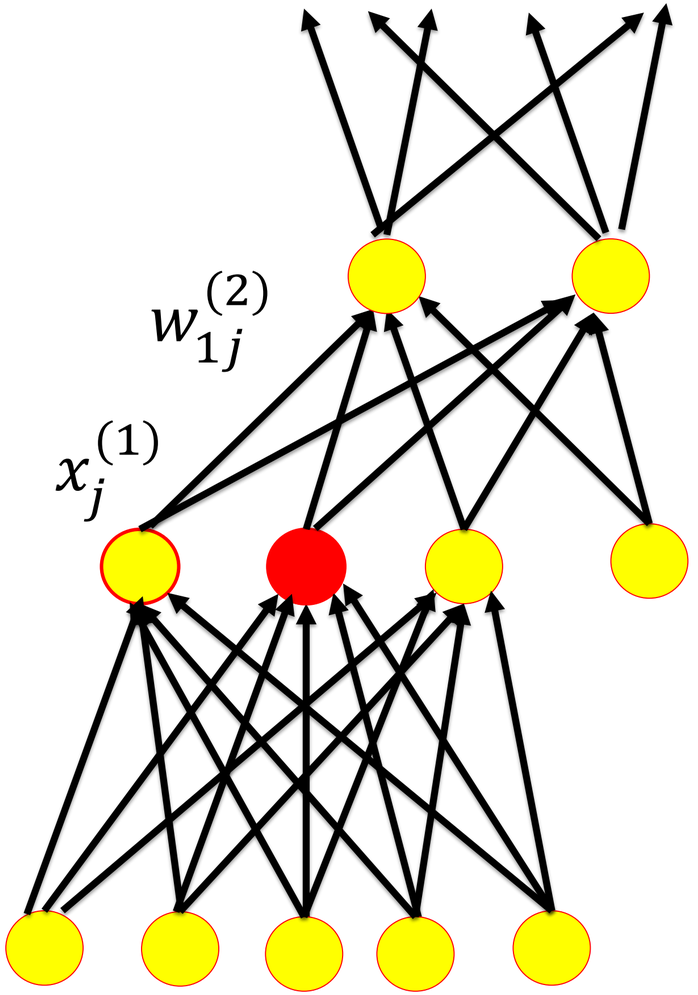
(2) $< w_{ij}^{(1)} > = 0$

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

→ **controlled spread of $a_j^{(1)}$ in layer 1**

→ **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

$$< a_i^{(1)} >= \quad \frac{1}{P} \sum_{\mu=1}^{P} \sum_j \; w_{ij} \, x_j^{\mu} \quad = 0$$

and a standard deviation

$$\sqrt{< [a_i^{(1)}]^2 >} = 2$$

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).

BackProp

0. **Initialization of weights**

1. **Choose pattern** $\mathbf{x}^\mu$

$$\textbf{input } x_k^{(0)} = x_k^\mu$$

2. **Forward propagation of signals** $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

$$\textbf{output } \hat{y}_i^\mu = x_i^{(n_{\max})}$$

3. **Computation of errors in output**

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})})\ [t_i^\mu - \hat{y}^\mu] \qquad (2)$$

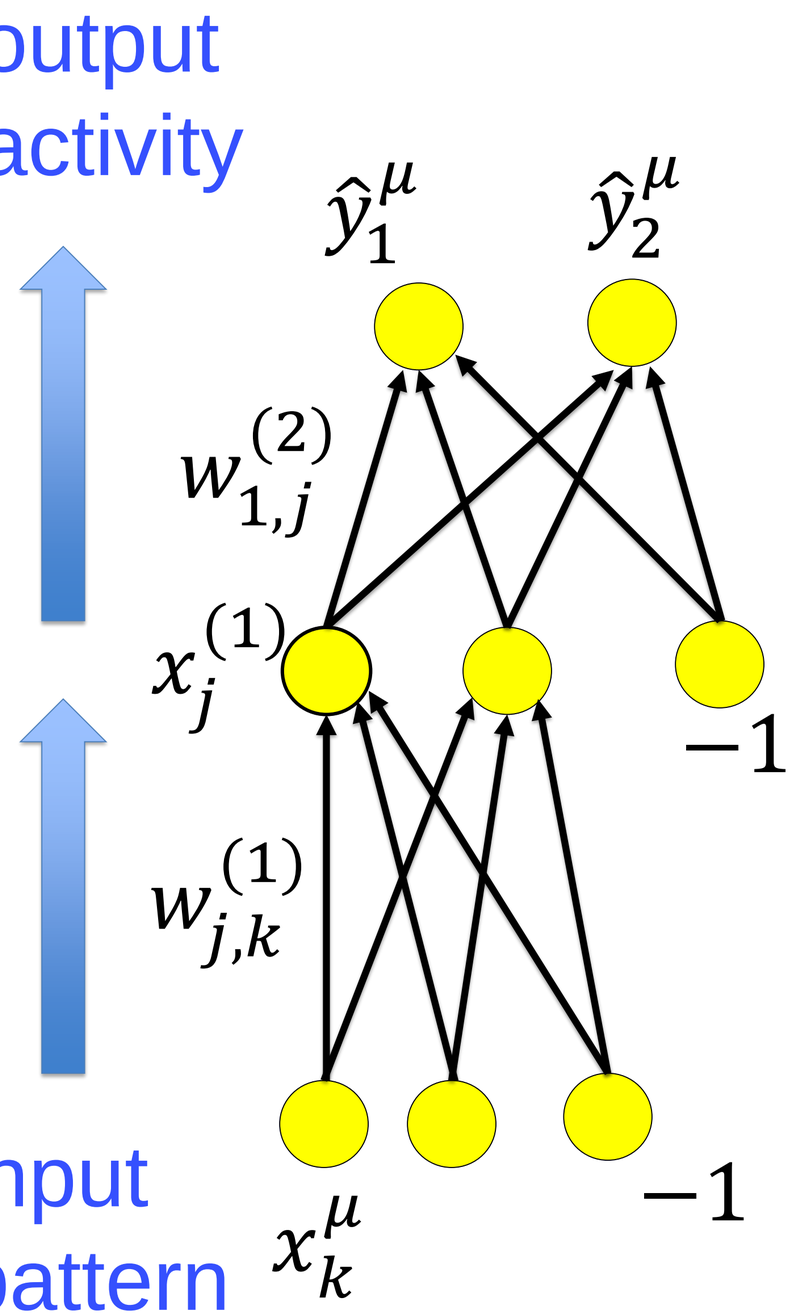
4. **Backward propagation of errors** $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \qquad (3)$$

5. **Update weights (for each** (i,j) **and all layers** (n)**)**

$$\Delta w_{ij}^{(n)} = \eta\, \delta_i^{(n)}\, x_j^{(n-1)} \qquad (4)$$

6. **Return to step 1.**



Previous slide.

In the forward pass, we need to evaluate

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

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.

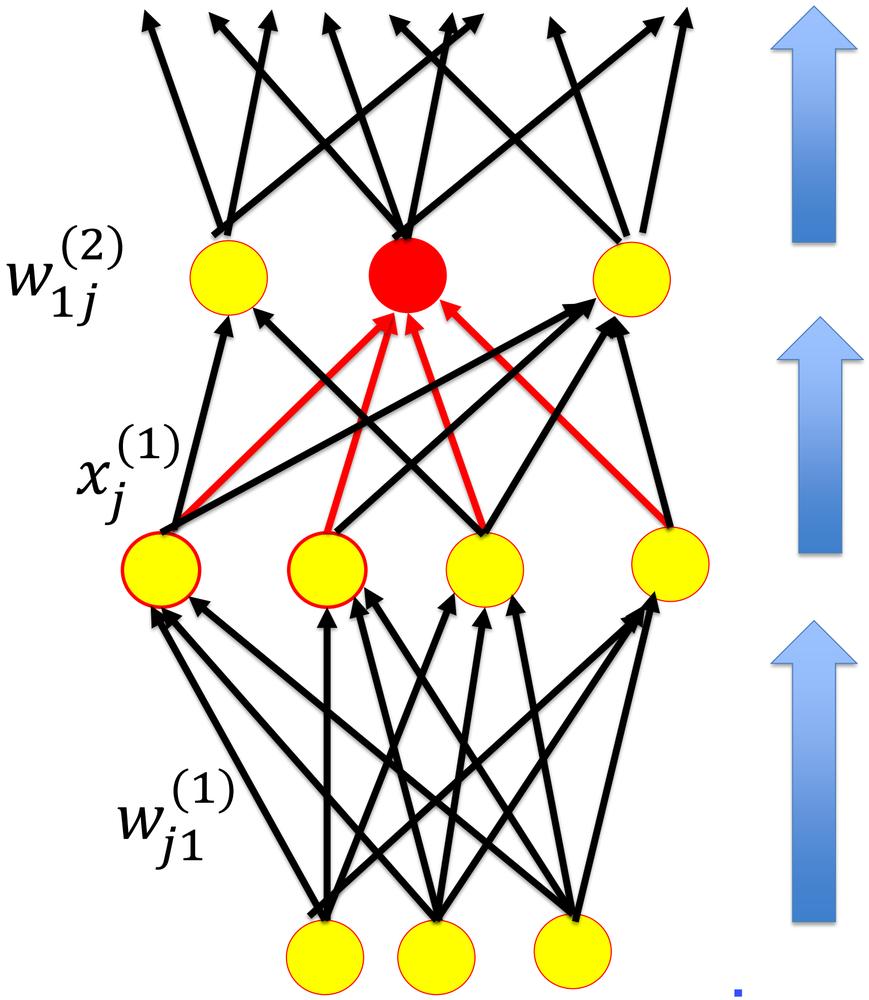
Why does the initialization of weights matter in backprop?

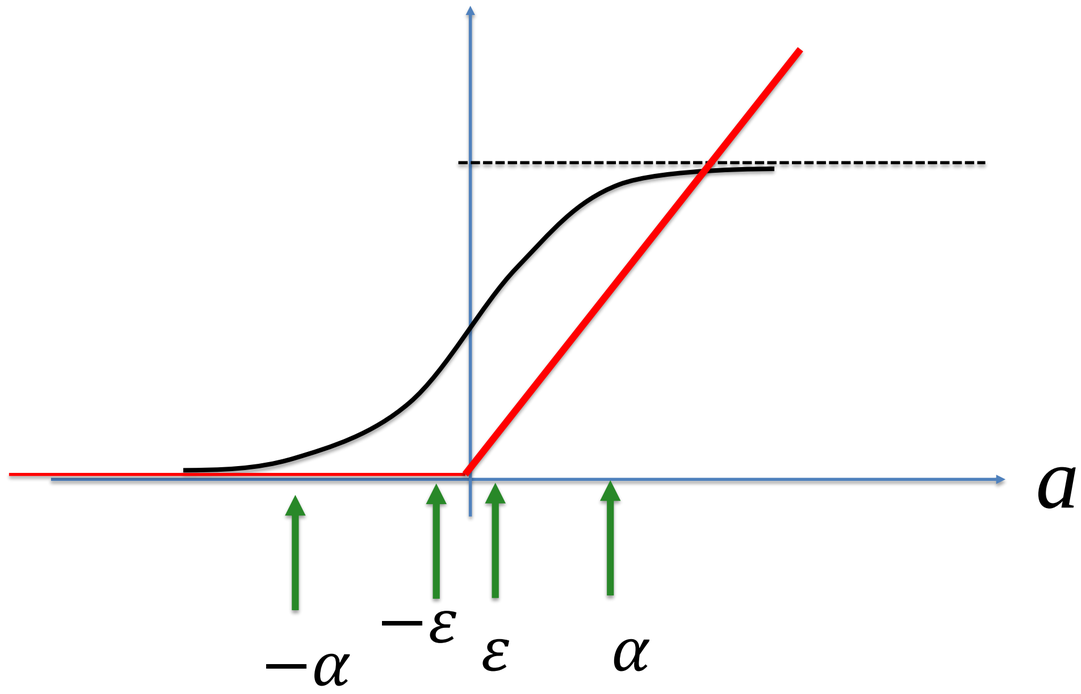
Previous slide.

So why is the initialization of the weights so important?

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

Forward pass: Linear and nonlinear processing





Previous slide.

As we have seen,

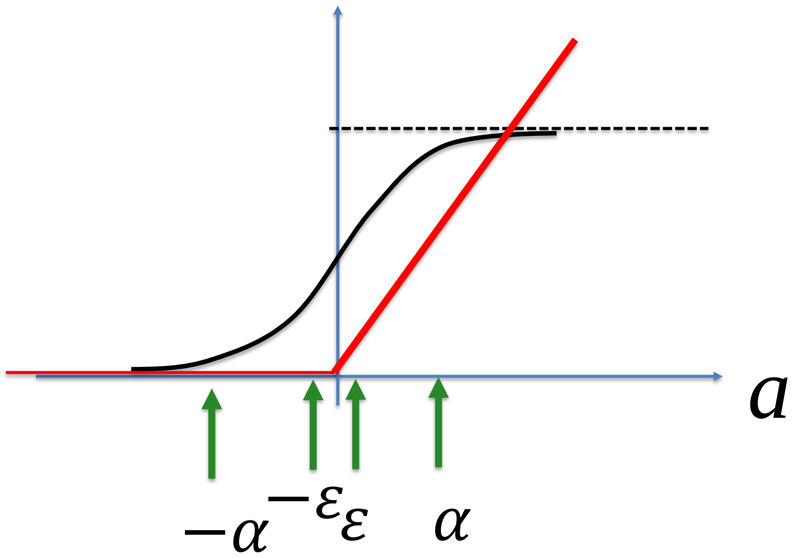
If I apply pattern μ , the total activation a of the red neuron might be α.
If I apply pattern $\mu+1$, the total activation a of the red neuron might be $-\varepsilon$.
If I apply pattern $\mu+2$, the total activation a of the red neuron might be $+\varepsilon$.
Etc.
Thus different patterns cause different activation values of same neuron (red)

Forward pass: Linear and nonlinear processing

Observations:

if all patterns in all layers touch the (same) 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 sigmoidals, 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 [-ε,ε], 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 [ε,α], 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 replace by a single layer of linear units**. Therefore the additional layers are useless.

- For ReLu's, it is good if some of the input patterns case have positive a, others negative a.

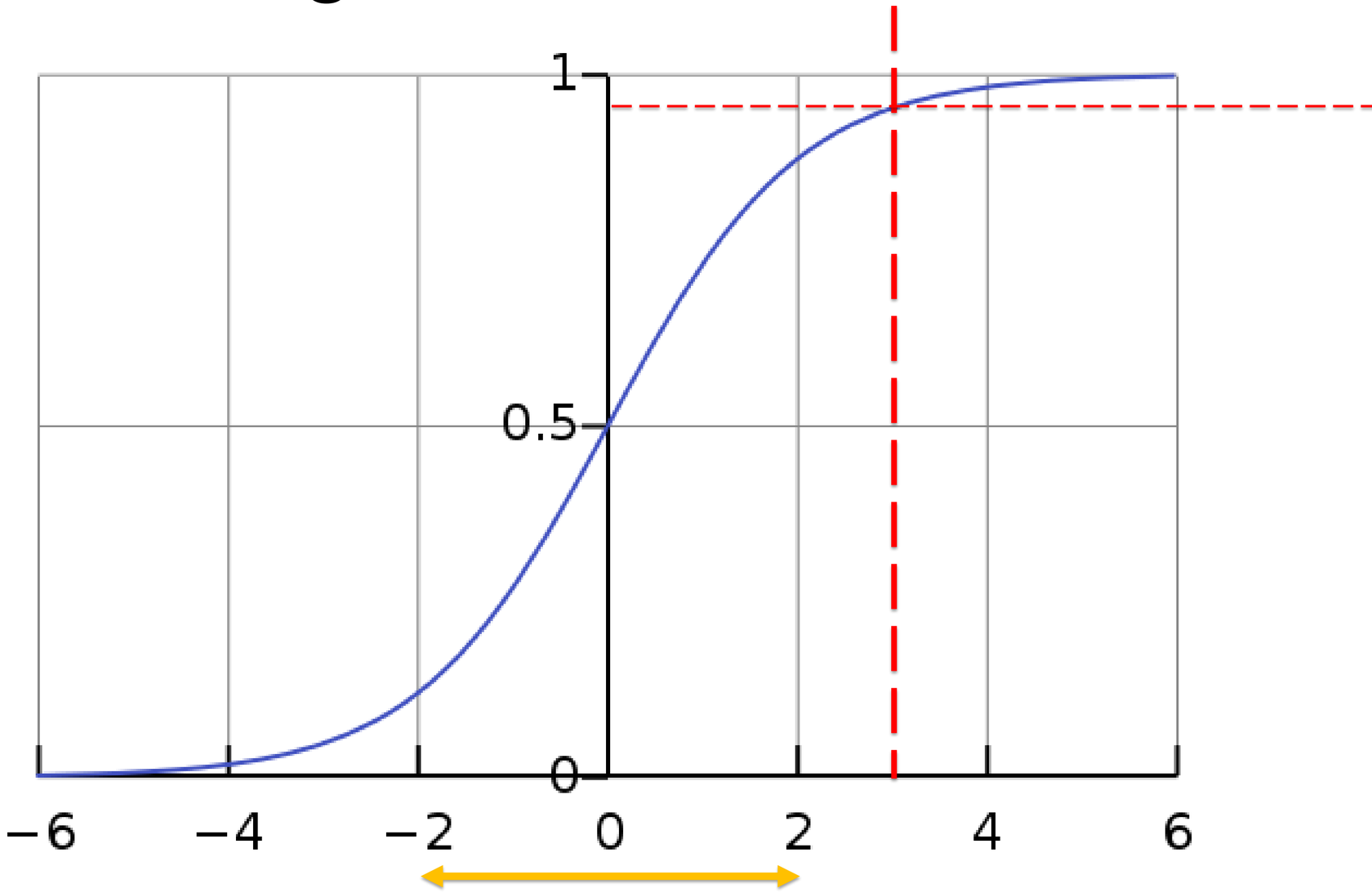
Review. sigmoidal output = logistic function

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

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 $|a| = 2$.

Summary: Exploit nonlinearities in forward pass ('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
   $0 = < a_j^{(n)} >;\ a_j^{(n)} = \sum_k\ w_{jk}^{(n)} x_k^{(n-1)}$
   → standard deviation of the activation variable
   $a_j^{(n)}$ 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.

Quiz:

Let n_i denote the number of input connections onto neuron i; this number will be called **the fan-in of neuron *i*.**

[ ] A good initialization of the weights onto neuron i is a Gaussian random distribution with mean zero and variance $1/n_i$

[ ] A good initialization of the weights onto neuron i is a Gaussian random distribution with mean zero and standard deviation $1/\sqrt{n_i}$

[ ] The fan-in number n_i and the above rule for random initialization includes the threshold variables.

[ ] A good initialization of the threshold variables is zero (or very close to zero)

[ ] The choice of the standard deviation of the random weights during initialization is less critical for the ReLu than for the sigmoidal units.

Previous slide.

Your comments.

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 6: Vanishing gradient problem

1. Questions and aims
2. Bagging
3. Dropout
4. Data augmentation
5. Weight initialization
6. Vanishing gradient problem

Previous slide.

So far, our arguments have been based on the forward pass. Let us now focus on the backward pass. As we will see, similar arguments can also be applied to the backward pass.

BackProp

0. **Initialization of weights**

1. **Choose pattern** $\mathbf{x}^\mu$

   input $x_k^{(0)} = x_k^\mu$

2. **Forward propagation of signals** $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \tag{1}$$

   output $\hat{y}_i^\mu = x_i^{(n_{\max})}$

3. **Computation of errors in output**

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})})\ [t_i^\mu - \hat{y}^\mu] \tag{2}$$

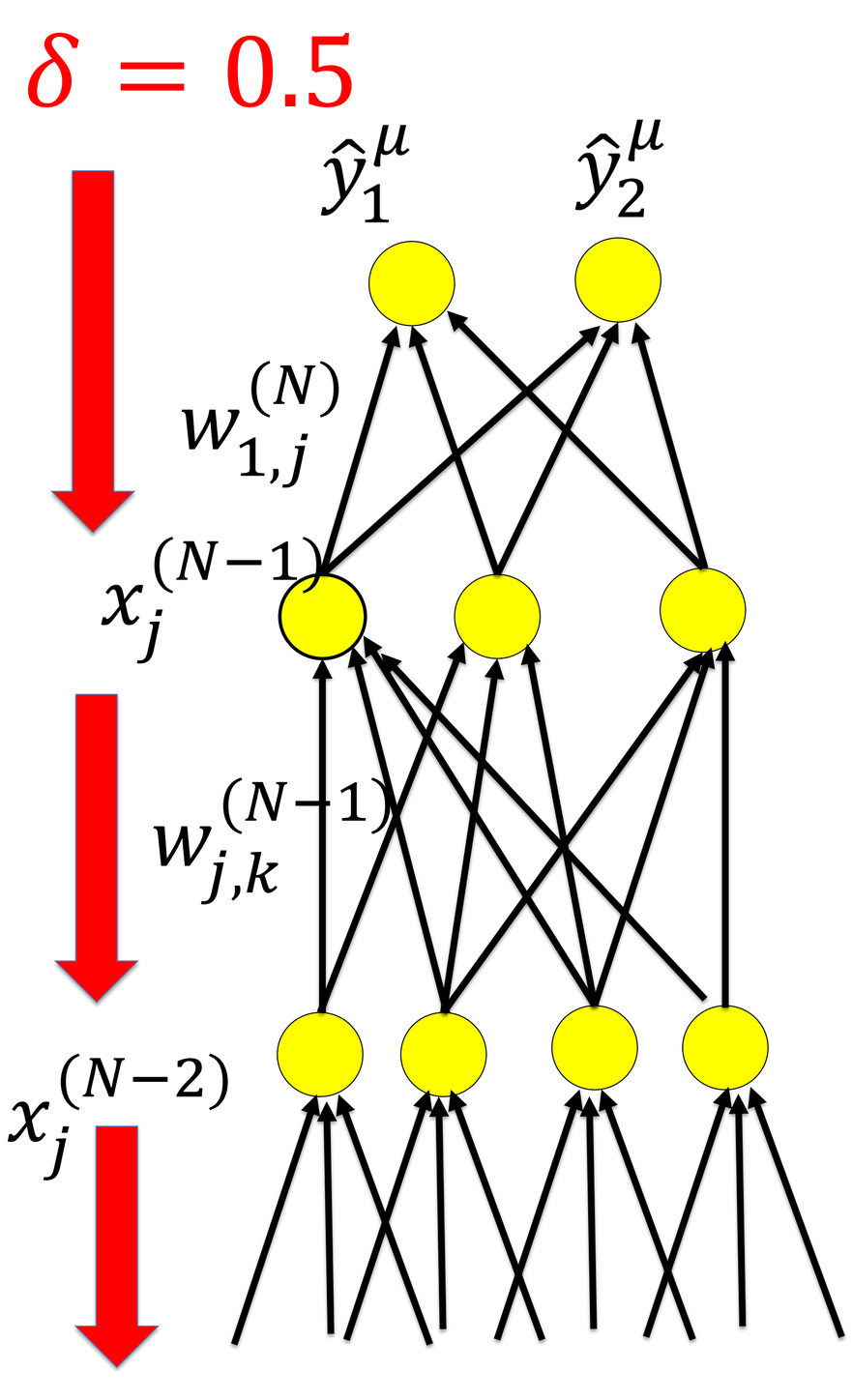
4. **Backward propagation of errors** $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \tag{3}$$

5. **Update weights (for each** (i,j) **and all layers** (n)**)**

$$\Delta w_{ij}^{(n)} = \eta\, \delta_i^{(n)}\, x_j^{(n-1)} \tag{4}$$

6. **Return to step 1.**



Previous slide.

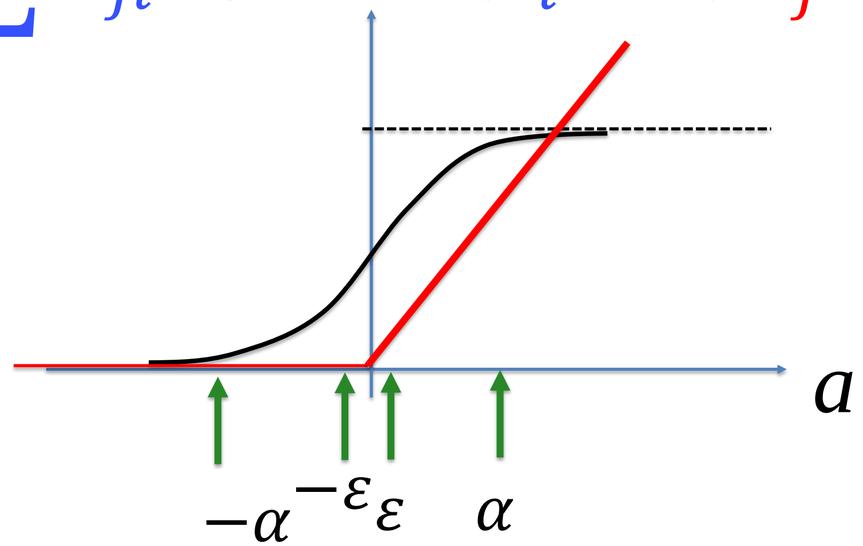
As discussed earlier, at each step of the backward pass, a factor

$$g'^{(n)}_j := g'\left[a^{(n)}_j\right] = \quad g'[\sum_k w^{(n)}_{j,k} x^{(n-1)}_k]$$

appears

Backward pass: Vanishing gradient problem

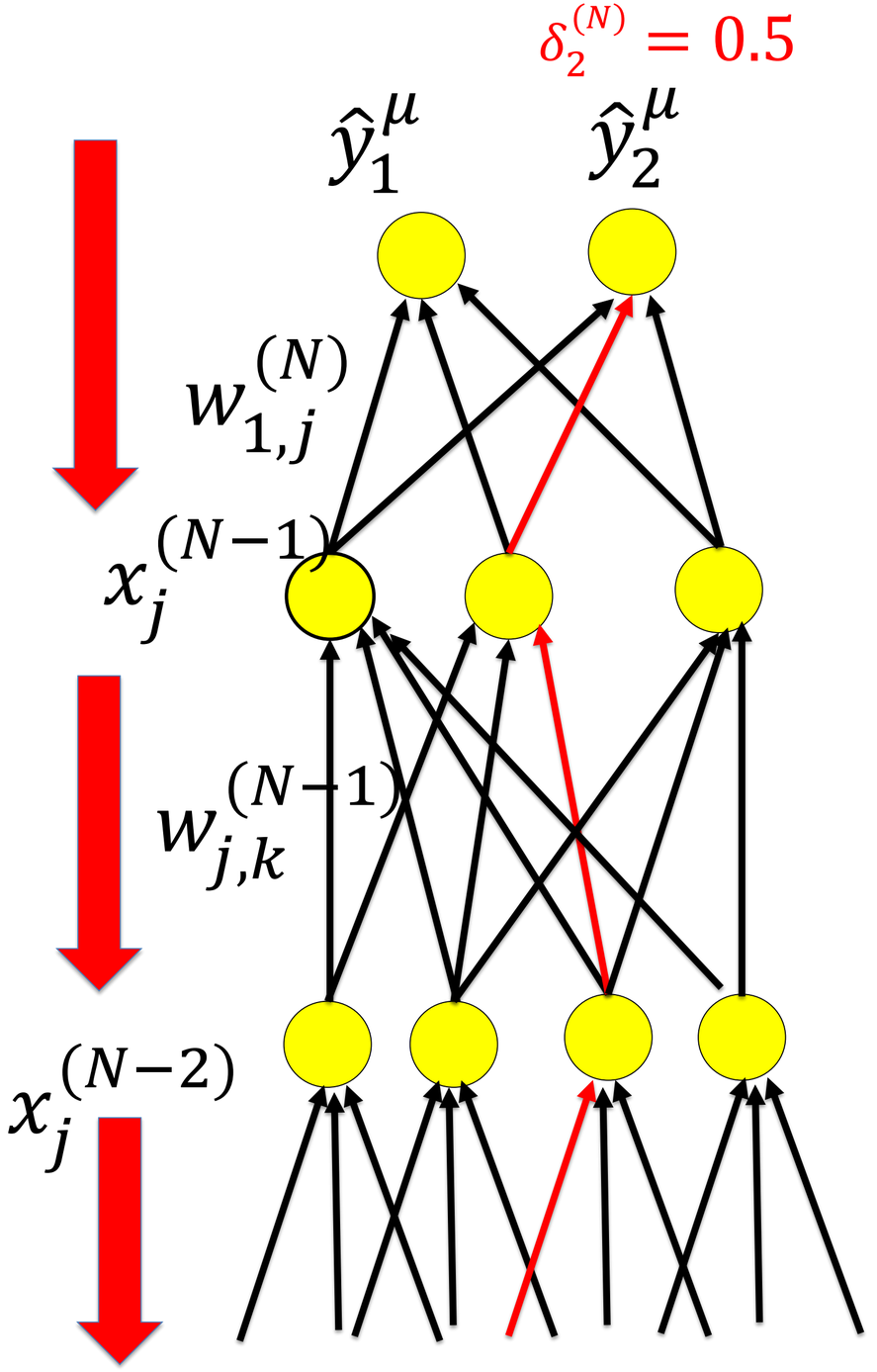
$$\delta_i^{(n-1)} = \sum_j w_{ji}^{(n)} g'^{(n-1)}(a_i^{(n-1)}) \delta_j^{(n)}$$



After N layers: each path contributes

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

Many paths need to be summed,
but most paths give tiny terms, 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

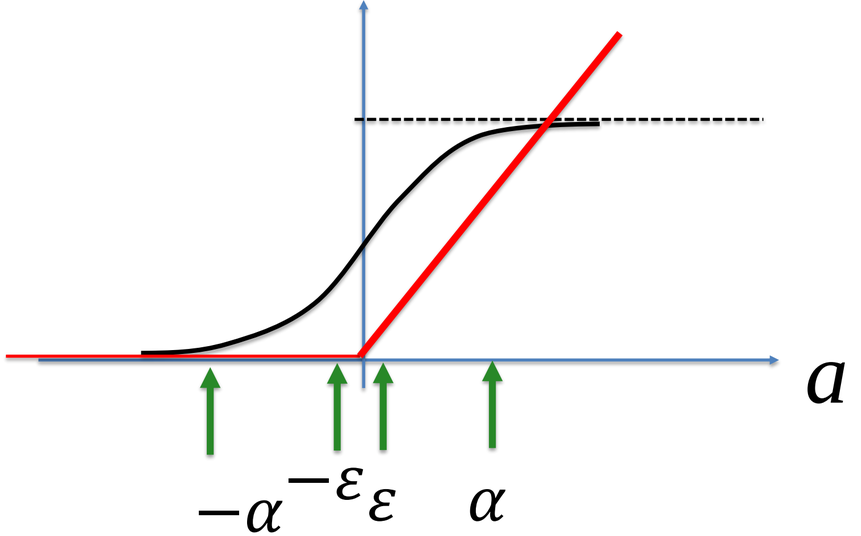
$$\delta_i^{(1)} \sim g'^{(1)} g'^{(2)} \dots g'^{(N-1)} \delta_j^{(N)}$$

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 gradient 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.

Vanishing gradient problem

Backward pass yields for
each single path many factors g’



Observations:

- g’ is small for **sigmoidal** at $-\alpha$ or $+\alpha$ ($|a|$=4 or bigger)
- but nonlinearity in forward path requires $|a|$>2 for some μ

- g’ vanishes for **ReLu** if one inactive unit sits in path
- g’=1 for all ReLu on ‘active paths’ during forward pass

→ 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

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.

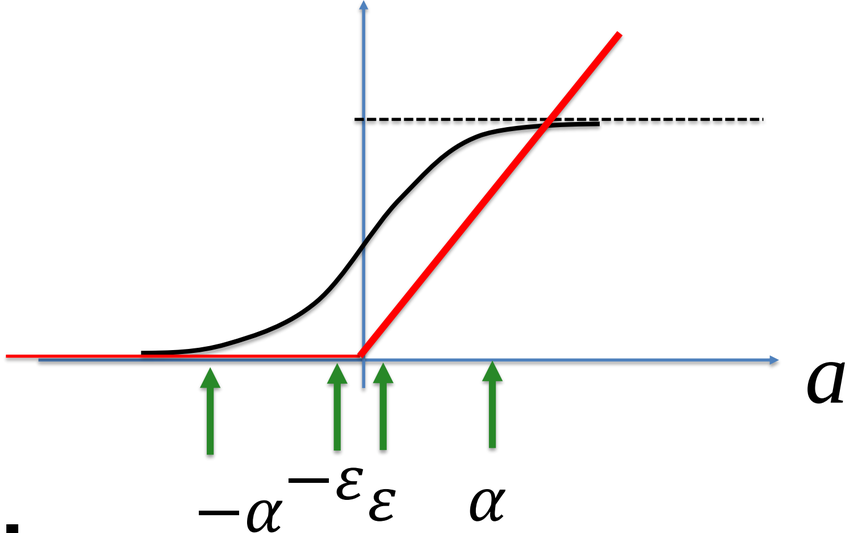
For **Rectified Linear units (ReLu)**, we ideally need for a given pattern μ that for some paths all units have:

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 based on 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.

Vanishing gradient problem



Conclusion:

Sucessful 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!
So far, ReLU looks like an excellent choice!

Quiz:

[ ] forward propagation with ReLu leaves only a few active paths

[ ] back propagation with ReLu leaves only a few active paths

[ ] 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 pass

Previous slide.

Your comments.

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 7: Weight update: mean input and bias problem

1. Questions and aims
2. Bagging
3. Dropout
4. Data augmentation
5. Weight initialization
6. Vanishing gradient problem
7. 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.

Weight update step

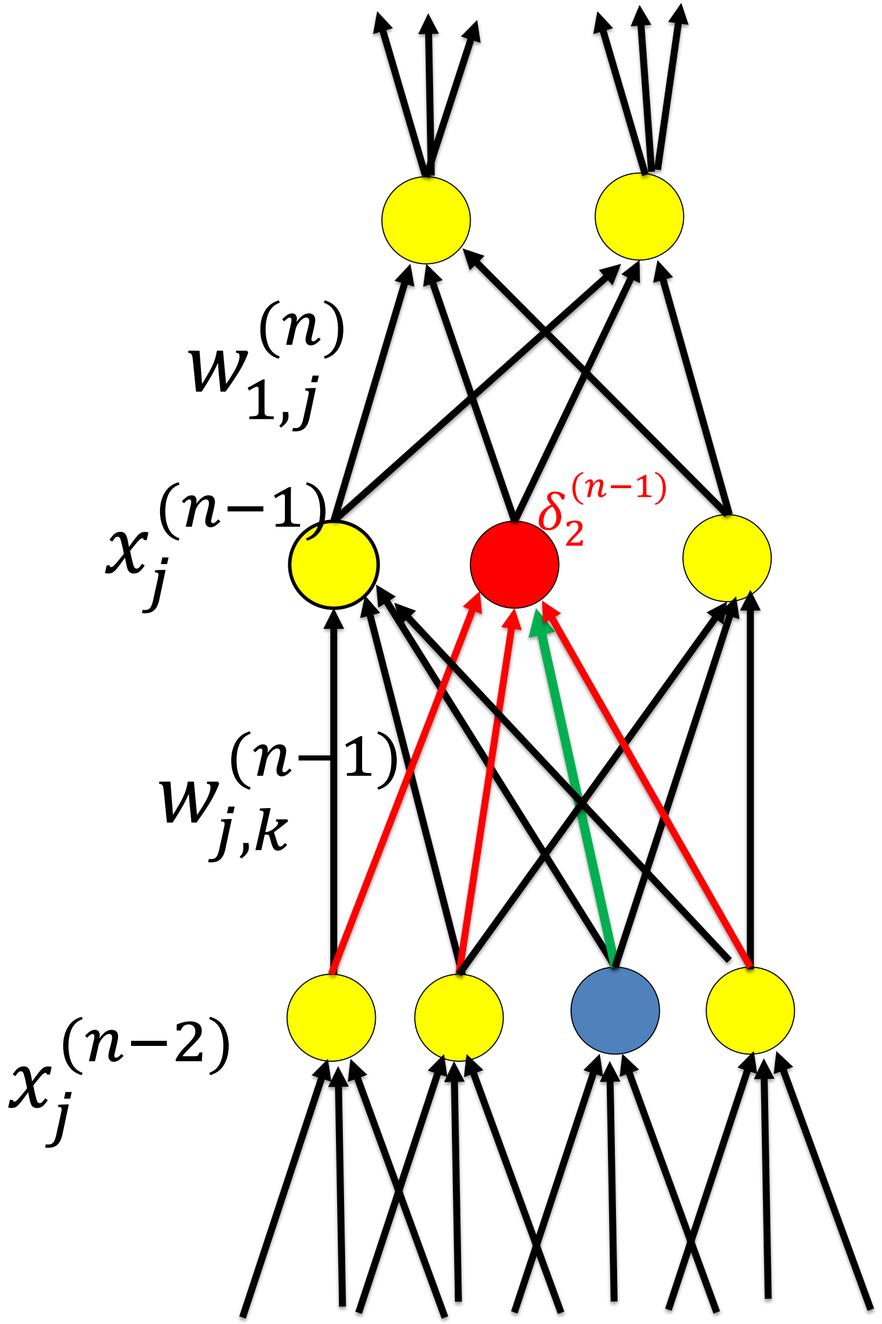
update **all** weights

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

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 $\delta_i^{(n-1)}$.

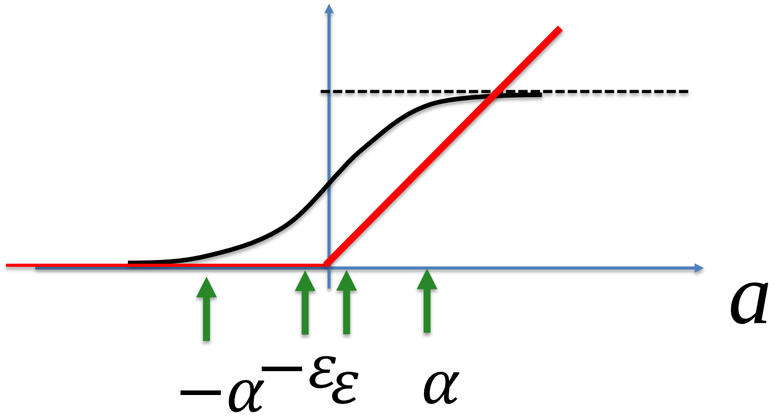
This has **two implications**.

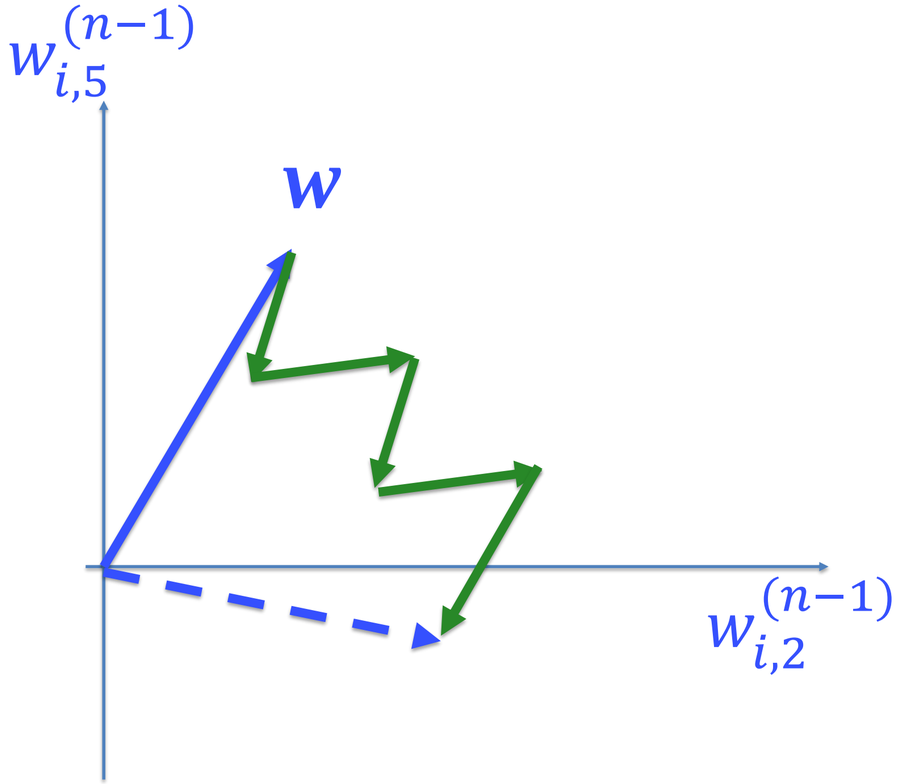
The first one concerns the possible movements of the weight vector, to be discussed now.

Weight update step

update **all** weights

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





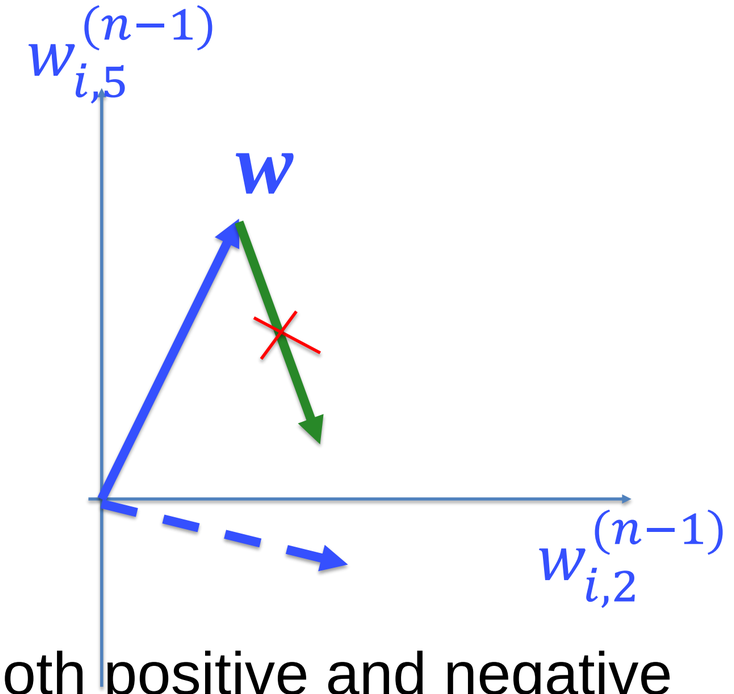
Weights onto the same neuron
are all updated with same delta

→ Problem for ReLu and other units with non-negative x:

'Either all the weights onto neuron i that change
move up together or all move down together'

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

If $\Delta w_{i,2}^{(n-1)} = \delta_i^{(n-1)} x_2^{(n-2)} \geq 0$,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.

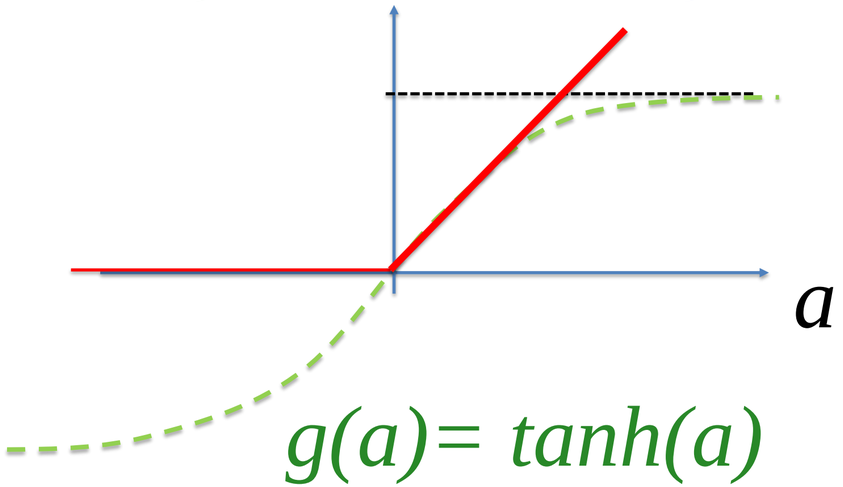
To move downward right, several iterations are necessary, as shown on the previous slide.

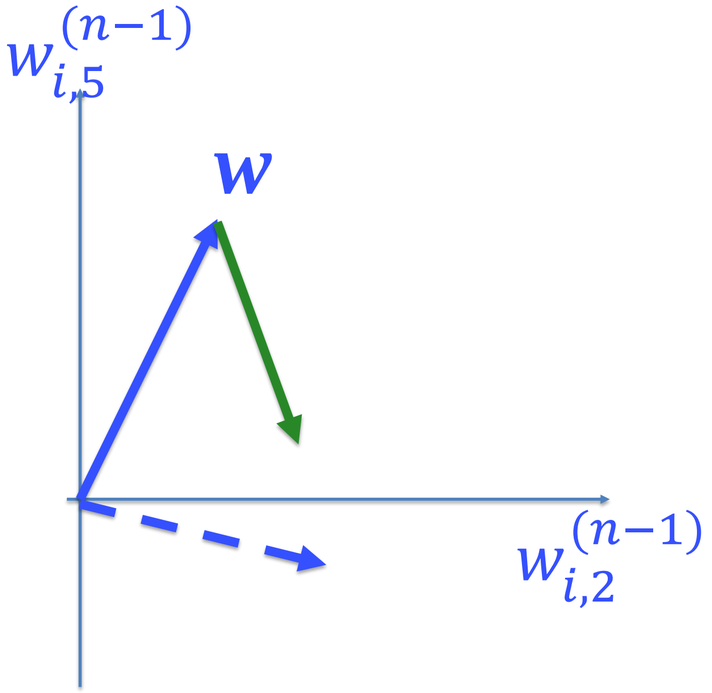
This problem is absent for units with a gain function that has both positive and negative values. For example, the problem is absent if we choose for the gain function of hidden units $g(a) = tanh(a)$

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

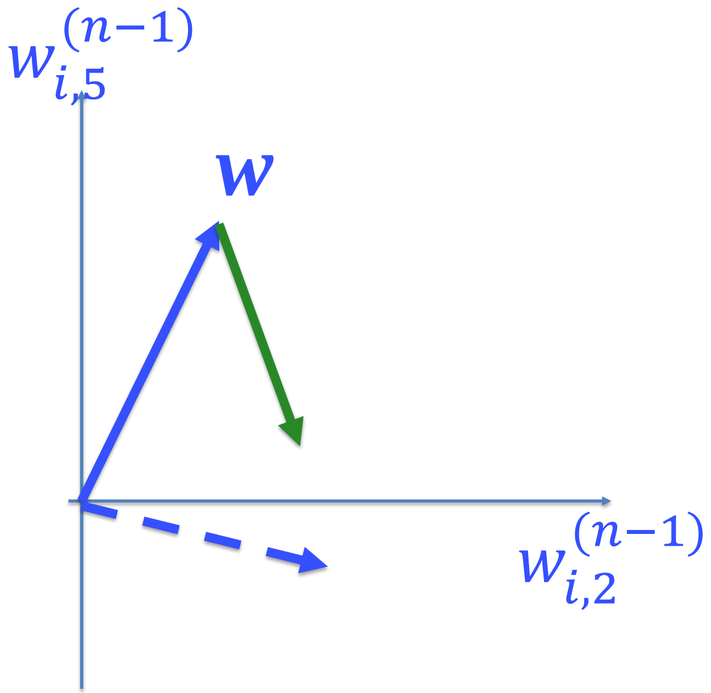
→ No problem for tanh

→ No problem for shifted exponential linear unit (SELU)

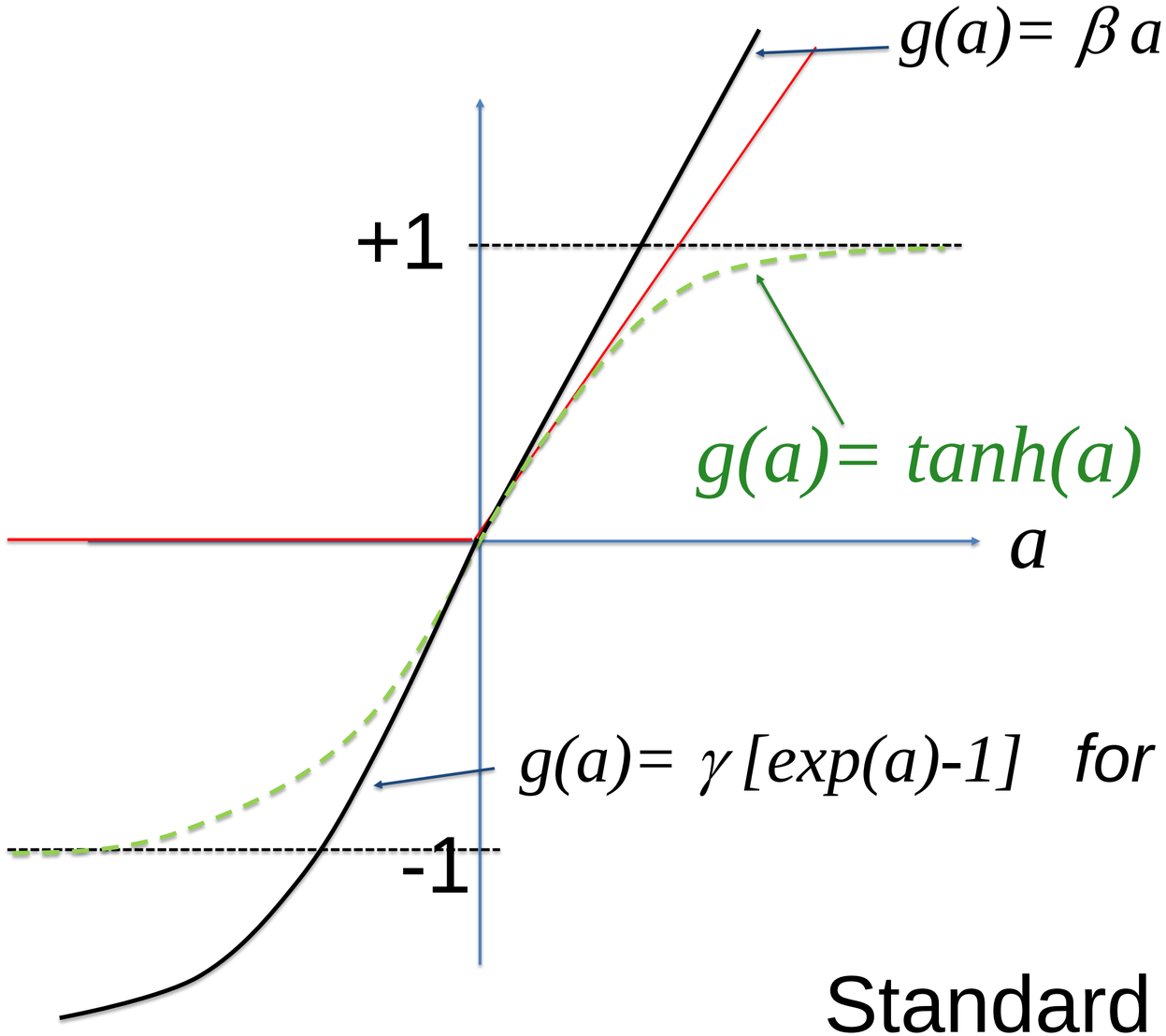
Previous slide.

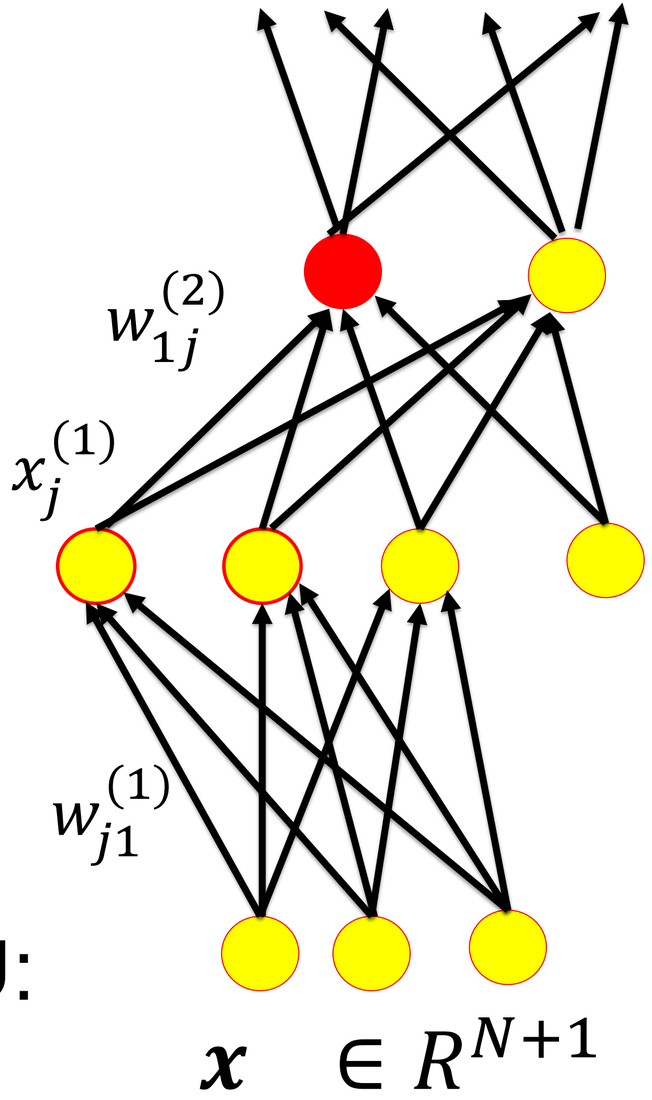
This problem is absent for units with a gain function that has both positive and negative values. For example, the problem is absent if we choose for the gain function of hidden units

$g(a) = tanh(a)$



Shifted Exponential Linear (SELU) vs. tanh





Standard ELU:

$\gamma=\beta=1$

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 generate negative outputs.

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 theta 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:

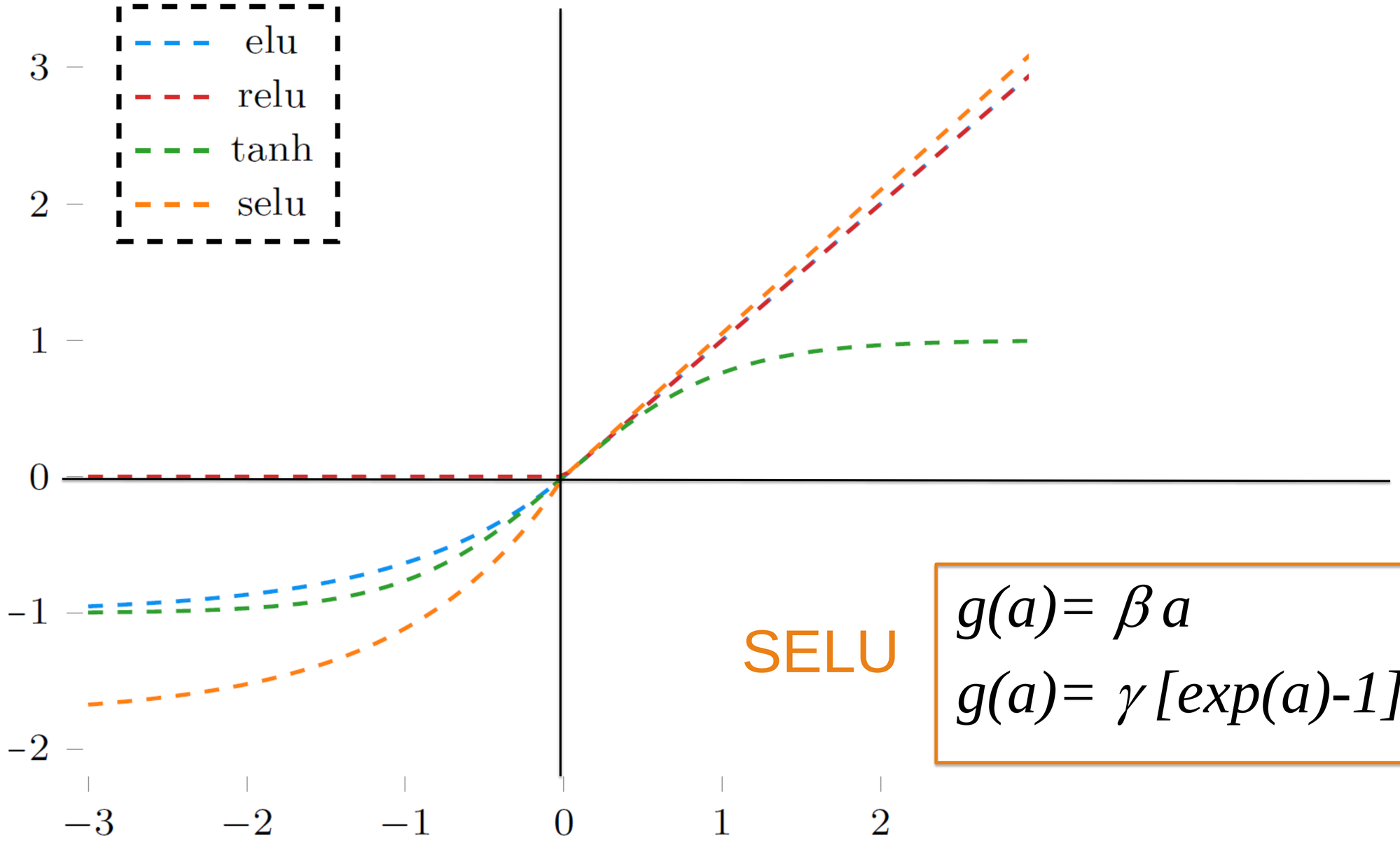
[ ] a non-zero weight update step of ReLu shifts most often the mean

[ ] 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 weight update step of SELU shifts most often the mean

Your notes.

Shifted Exponential Linear vs. tanh



SELU

$g(a) = \beta\, a$ for $a>0$

$g(a) = \gamma\, [exp(a)-1]$ for $a<0$

Previous slide.
The generalized ‘Shifted exponential linear unit’ (SELU) has two parameters, $\beta>1$, $\gamma>1$:

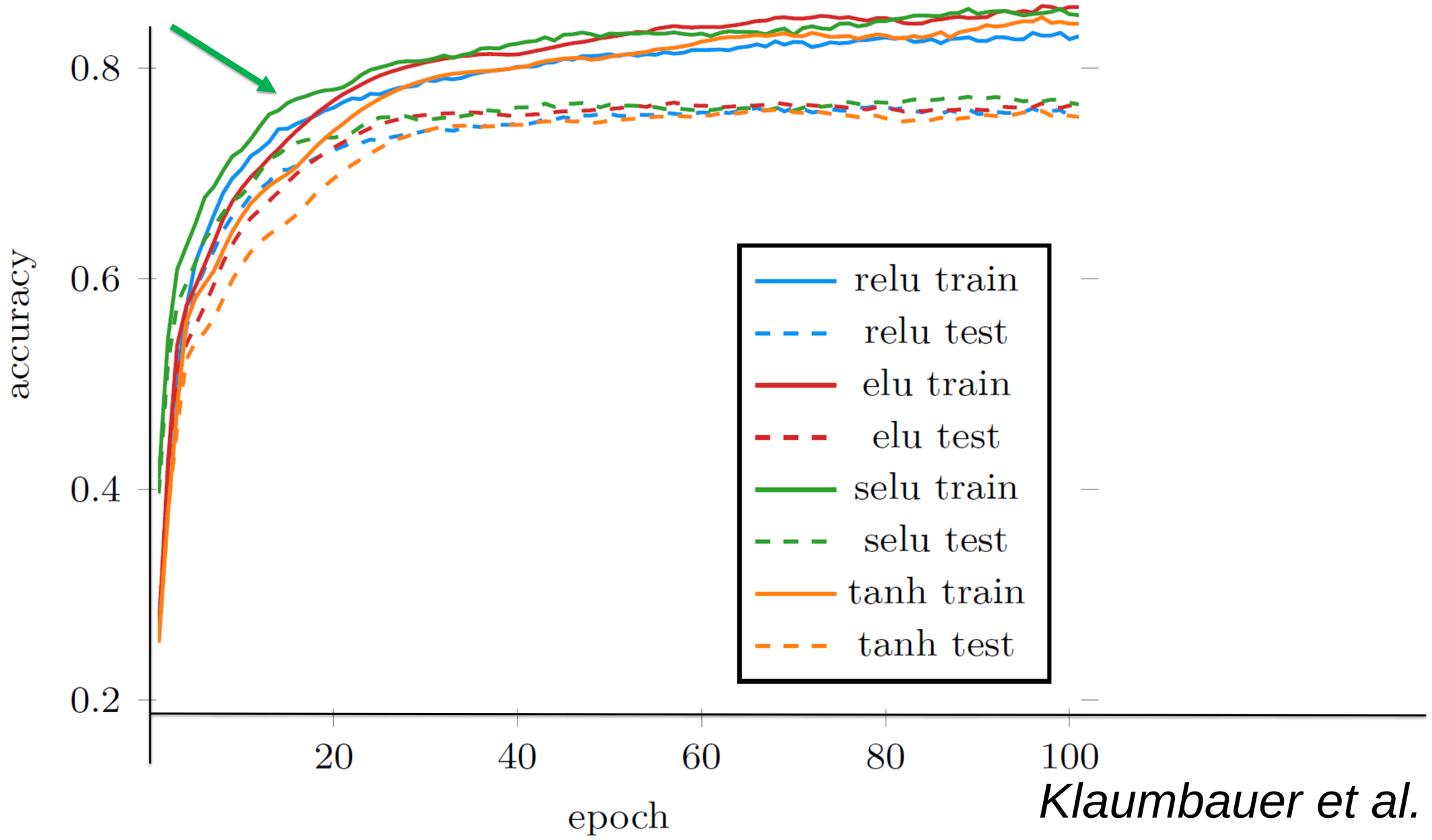
$$g(a) = \beta\, a \qquad \text{for } a>0$$

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

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)



Klaumbauer et al. (2017)

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 (linearity problem), or shifting mean (bias problem) that plague learning during the initial epochs are minimized.

6. Conclusion

- initialization is important in the **initial phase** of training
- choice of hidden unit is important in **initial phase** of training
- ReLu has disadvantages in keeping the mean
  → requires 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: Klaumbauer, …, Hochreiter (2017)
Self-normalizaing neural networks
https://arxiv.org/pdf/1706.02515.pdf

Previous slide.

Thus, if you have the choice, take SELUs.

The shifting mean can also be addressed by batch normalization, which is the topic of the next section. Since batch normalization is part of standard implementation patterns, there is no obligation to switch to SELUs.

Artificial Neural Networks

Tricks of the Trade in Deep Learning

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 8: Batch normalization

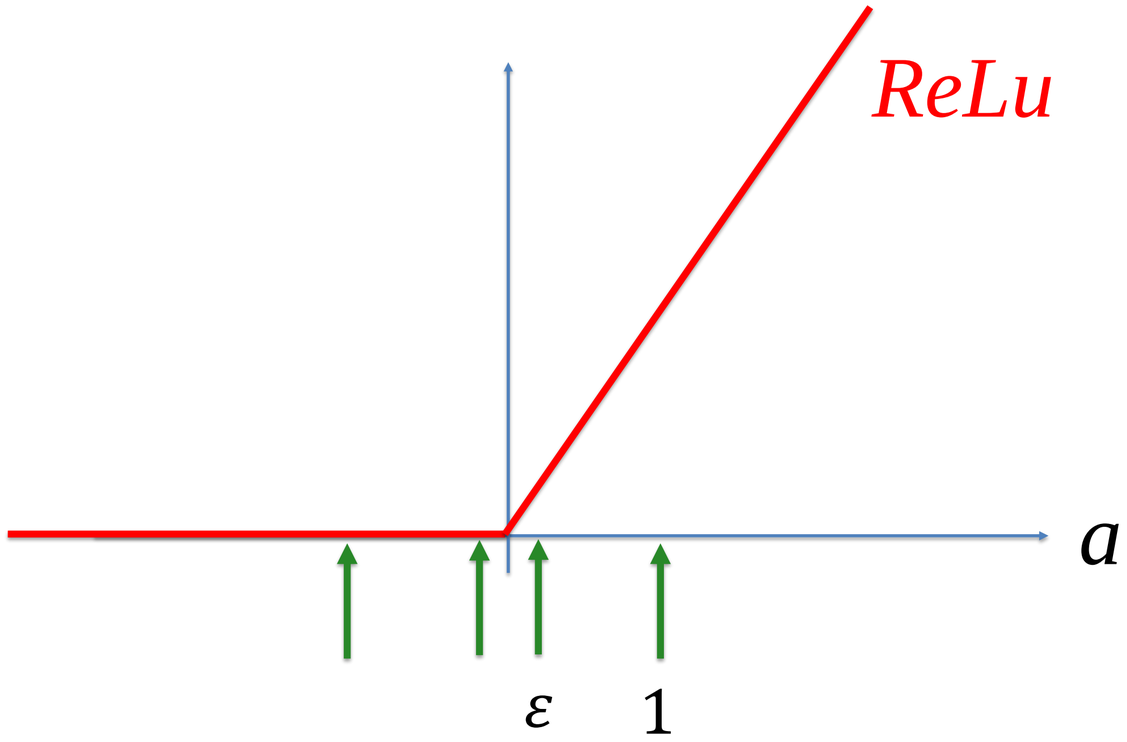
1. Questions and aims
2. Bagging
3. Dropout
4. Data augmentation
5. Weight initialization
6. Vanishing gradient problem
7. Weight update: mean input and bias problem
8. Batch normalization

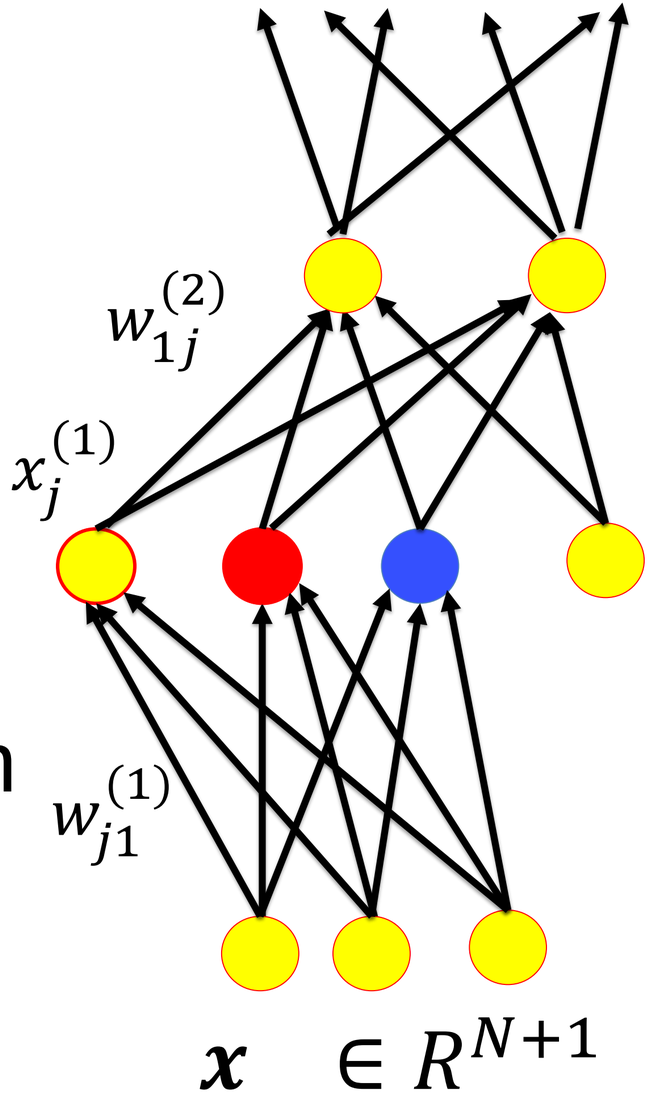
Previous slide.

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.

Aim of Batch normalization: Keep mean input stable





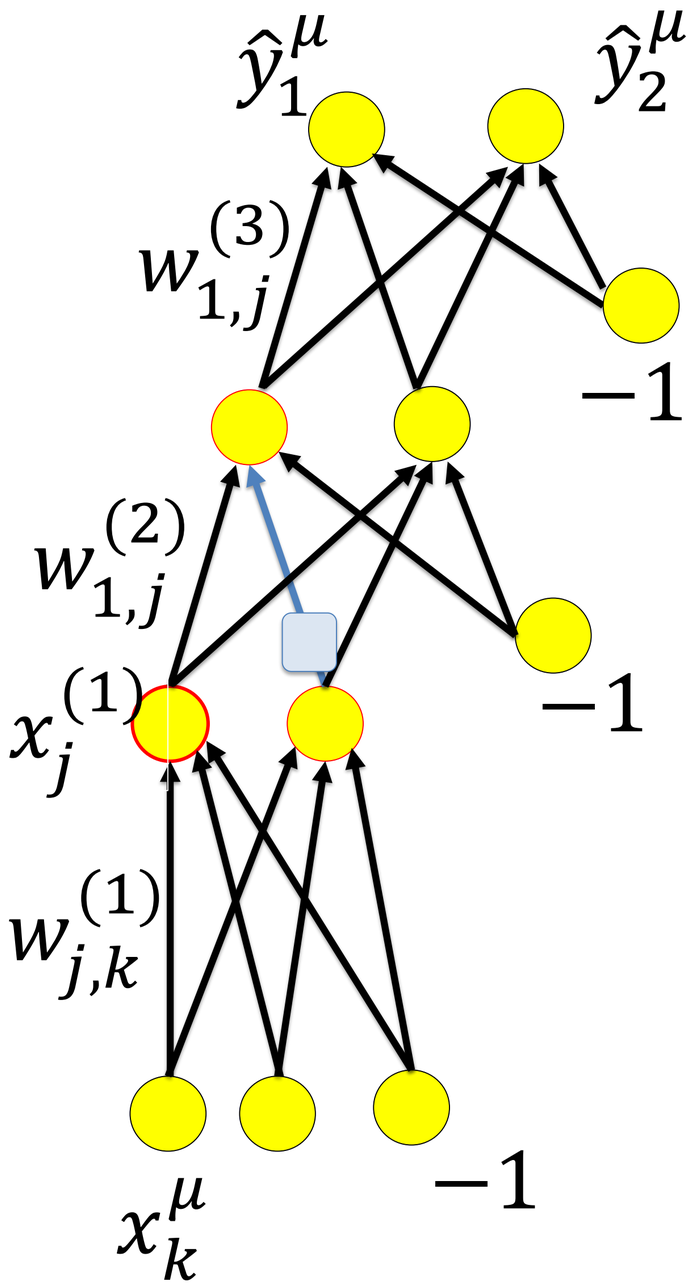
- Different patterns give different activation of same neuron (red)
- Aim: During learning, keep mean at approximately zero

Previous slide.

As a reminder, different patterns will activate the red neuron to different levels, such as +1, epsilon, or -1.

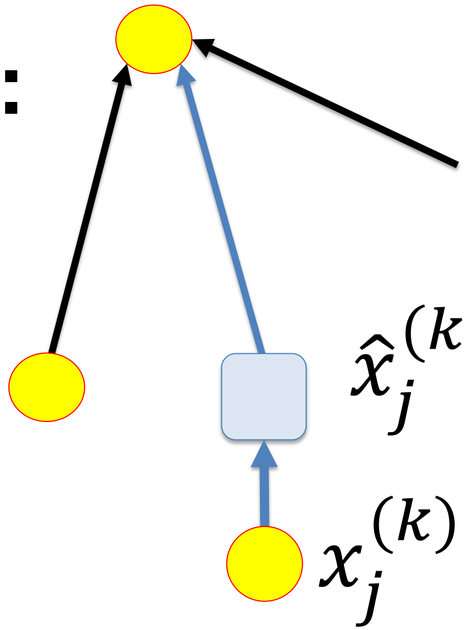
The aim of batch normalization is to keep, DURING LEARNING, the mean of the activation always around zero and a standard deviation always around 1.

7. Batch normalization: Idea



Normalize input on each input line

Zoom:



$$\hat{x}_j^{(k)} = \frac{x_j^{(k)} - E[x_j^{(k)}]}{\sqrt{Var[x_j^{(k)}]}}$$

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.

You also need to keep the normalization for inference on novel data.

7. Batch normalization

Ioffe&Szegedi, 2015

Work with minibatch:
Normalize per minibatch

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

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

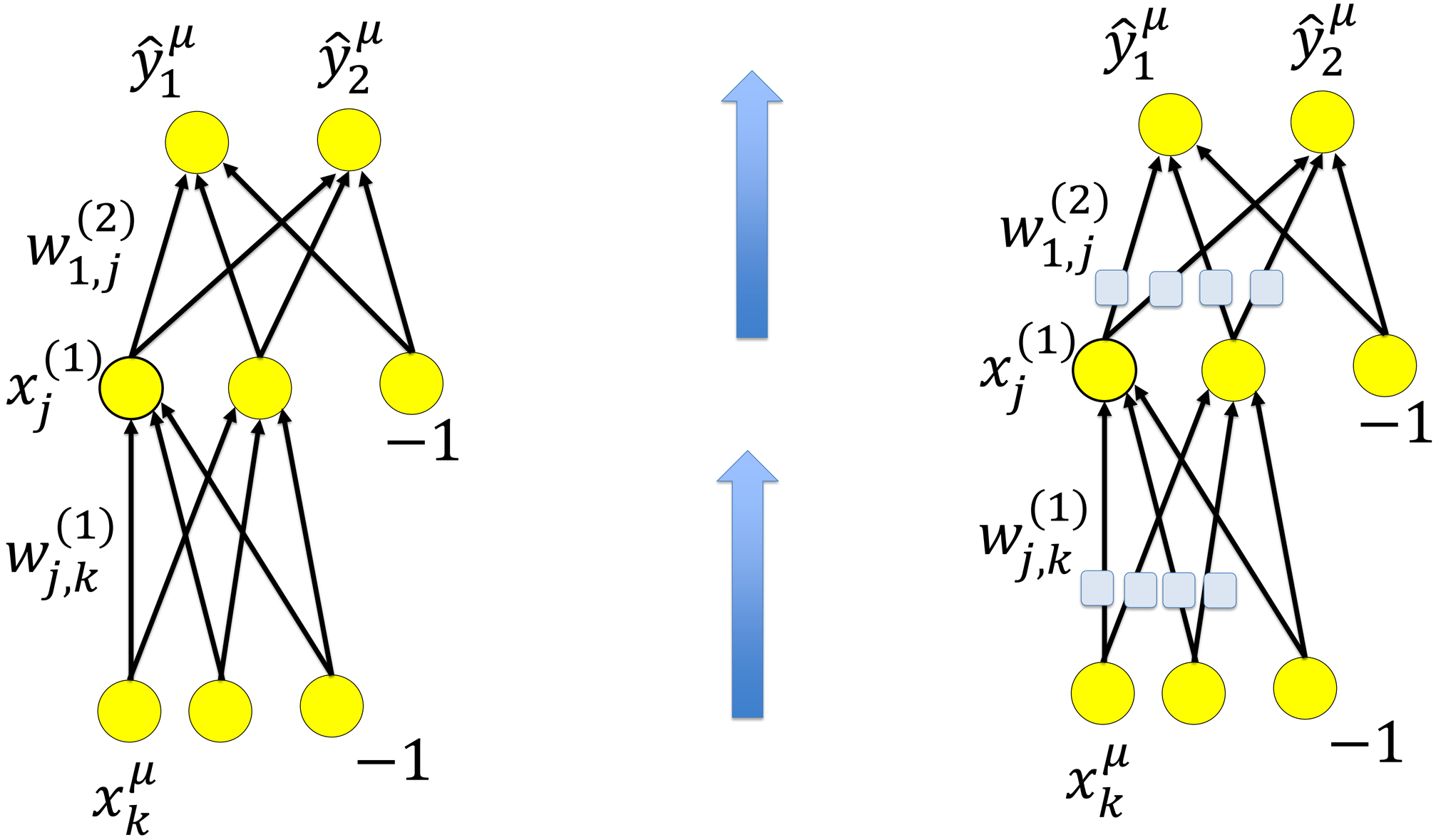
Previous slide.

The blue box corresponds to a mathematical transformation y=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.

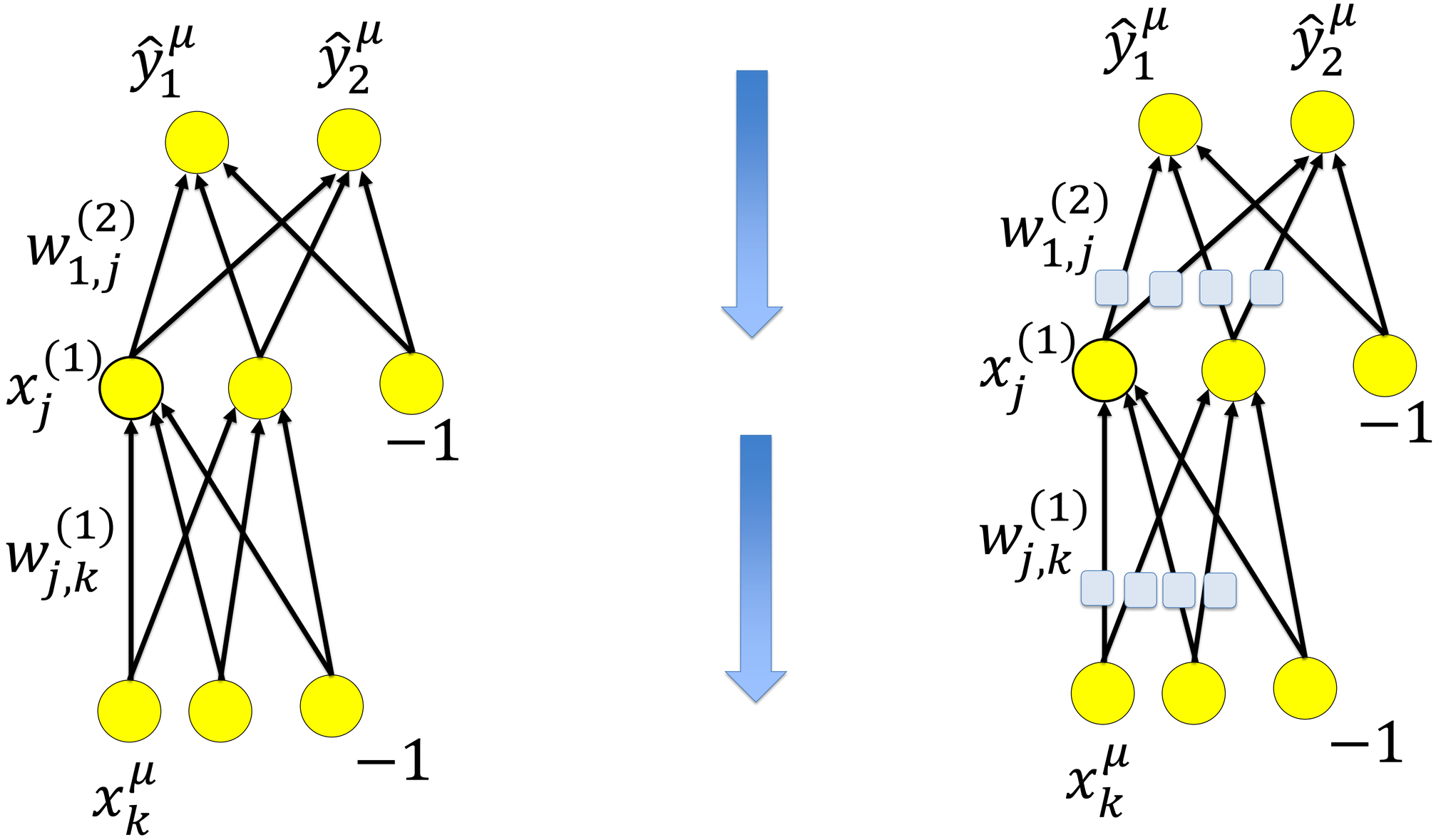
7. Batch normalization Ioffe&Szegedi, 2015



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;
Goodfellow et al, 2016

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** $k = 1 \ldots K$ **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**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **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 of size m, and average over them:

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

In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with

$$y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$$

end for

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
- During inference (normalized) weights are frozen.

Objectives for today:

- Bagging: multiple models help always to improve results!
- Dropout: two interpretations
  - (i) a practical implementation of bagging
  - (ii) forced feature sharing
- Data augmentation: exploit invariances
- 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

Teaching monitoring – monitoring of understanding

[ ] today, up to here, at least 60% of material was new to me.

[ ] up to here, I have the feeling that I have been able to follow (at least) 80% of the lecture.

(previous slide)

Reading for this lecture:

Goodfellow et al.,2016 *Deep Learning, MIT Press*

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

Further Reading for this Lecture:

Papers:

Robert Geirhos et al., ImageNet trained images are biased … ICLR, 2019
https://arxiv.org/pdf/1811.12231.pdf

Klaumbauer, …, Hochreiter (2017), Self-normalizaing neural networks
https://arxiv.org/pdf/1706.02515.pdf

Ioffe&Szegedi, 2015, Batch Normalization: accelerating …
https://arxiv.org/abs/1502.03167

The end

Climate change - Climate strike, course edition 2019

Switzerland in 2017: 46 Mio tons of CO2 (1 ton = 1000kg)

Aim: - CO2 neutrality in 2035 (or 2050)

- pressure on politics: implement Paris climate agreement

CO2 tax can help to readjust behavior

- Tax on carbon,
- redistribute money to citizens
- Start with low taxes: 4 cent per kg
- Increase every year, linearly
- Adjust for borders (duties for imported CO2-heavy goods)

Switzerland is small, but Europe as a whole is big,
other countries will follow.

Arctic sea ice thickness