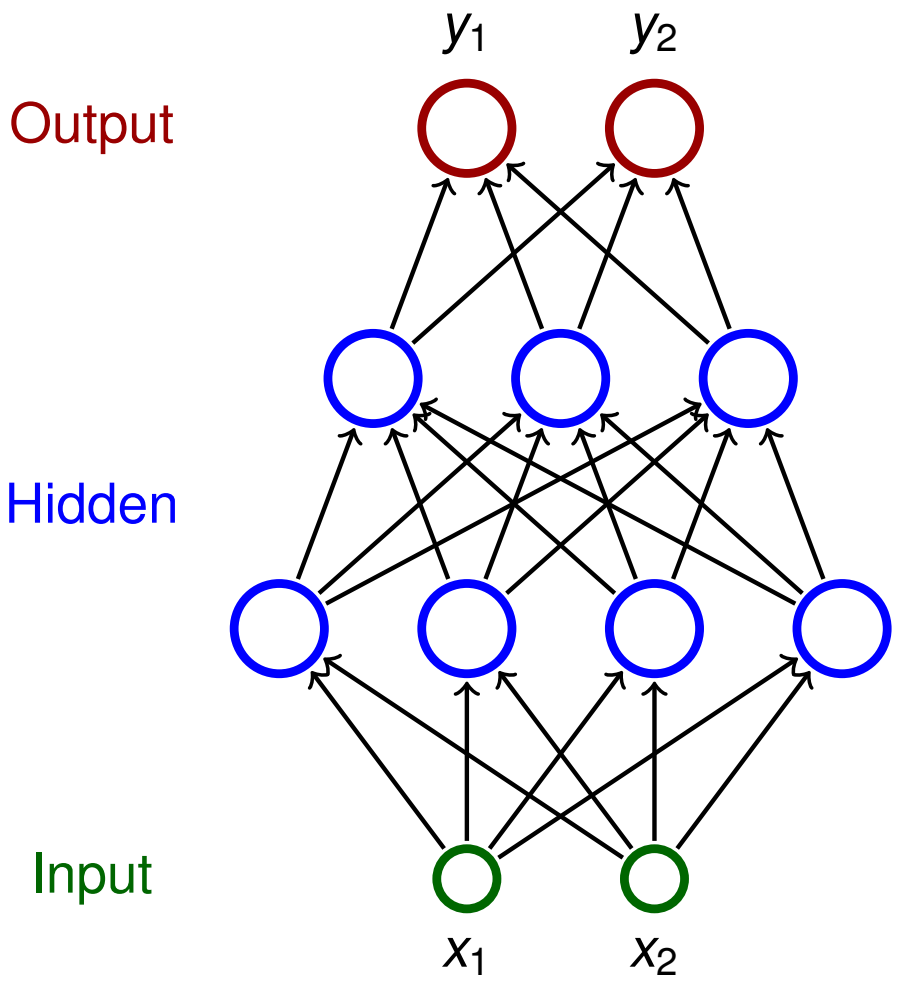
Architecture – Multilayer Perceptron (MLP)



- Neurons partitioned into **layers**; one input layer, one output layer, possibly several hidden layers
- layers numbered from 0; the input layer has number 0
  - E.g. three-layer network has two hidden layers and one output layer
- Neurons in the i-th layer are connected with all neurons in the $i + 1$-st layer
- Architecture of a MLP is typically described by numbers of neurons in individual layers (e.g. 2-4-3-2)

MLP – architecture

Notation:

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)

MLP – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k \right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function:**

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj} \right)^2$$

MLP – batch learning

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

Here

$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \nabla E(\vec{w}^{(t)}) = -\varepsilon(t) \cdot \sum_{k=1}^{p} \nabla E_k(\vec{w}^{(t)})$$

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E(\vec{w}^{(t)})$ is the gradient of the error function
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error function *for the training example k*

MLP – MINIbatch learning – SGD

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

where

$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

Here T is a *minibatch* (of a fixed size),

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially. **Epoch** consists of one round through all data.

MLP – error functions

- **square error:**

$$E(\vec{w}) = \sum_{k=1}^{p} E_k(\vec{w})$$

where $E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj} \right)^2$

- **mean square error (mse):**

$$E(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} E_k(\vec{w})$$

I will use mse throughout the rest of this lecture.

MLP – mse gradient

For every w_{ji} we have

$$\frac{\partial E}{\partial w_{ji}} = \frac{1}{p} \sum_{k=1}^{p} \frac{\partial E_k}{\partial w_{ji}}$$

where for every $k = 1, \ldots, p$ holds

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

and for every $j \in Z \setminus X$ we get

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

(Here all y_j are in fact $y_j(\vec{w}, \vec{x}_k)$).

Practical issues of gradient descent

- Training efficiency:
  - What size of a minibatch?
  - How to choose the learning rate $\varepsilon(t)$ and control SGD ?
  - How to pre-process the inputs?
  - How to initialize weights?
  - How to choose desired output values of the network?
- Quality of the resulting model:
  - When to stop training?
  - Regularization techniques.
  - How large network?

For simplicity, I will illustrate the reasoning on MLP + mse. Later we will see other topologies and error functions with different but always somewhat related issues.

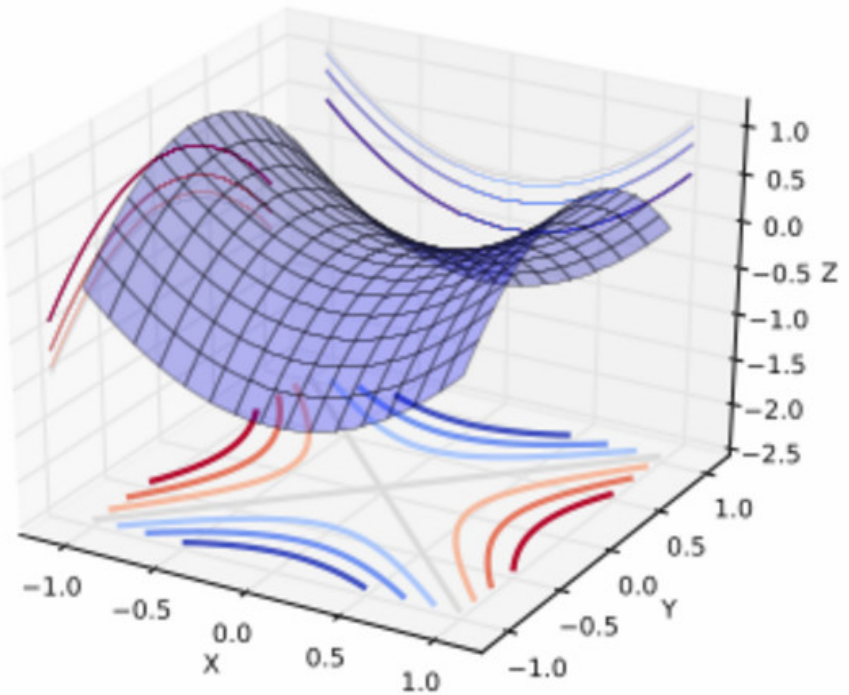
Issues in gradient descent

Lots of local minima where the descent gets stuck:

- The model identifiability problem: Swapping incoming weights of neurons *i* and *j* leaves the same network topology – **weight space symmetry**
- Recent studies show that for sufficiently large networks all local minima have low values of the error function.

Saddle points

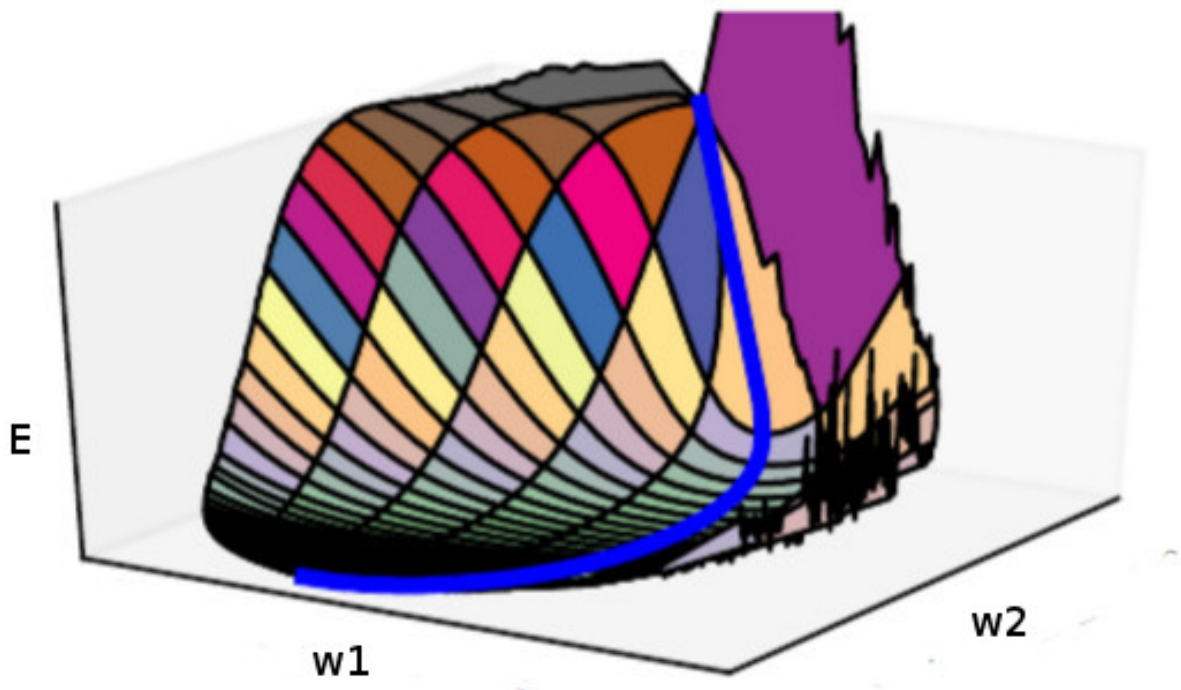
One can show (by a combinatorial argument) that larger networks have exponentially more saddle points than local minima.



Issues in gradient descent – too slow descent

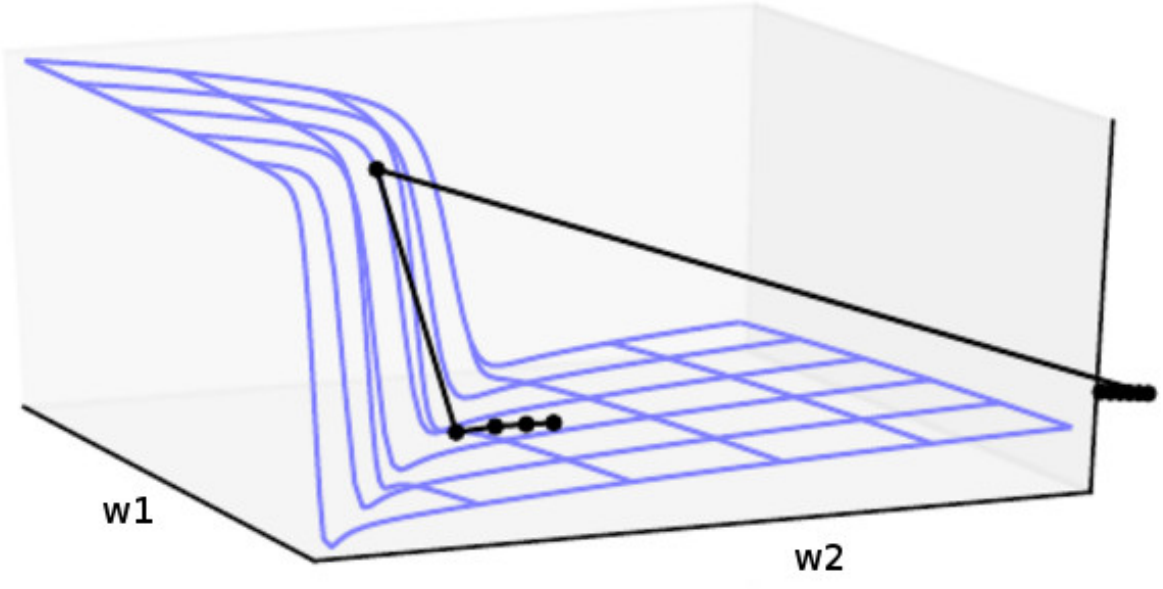
- flat regions

  E.g. if the inner potentials are too large (in abs. value), then their derivative is extremely small.

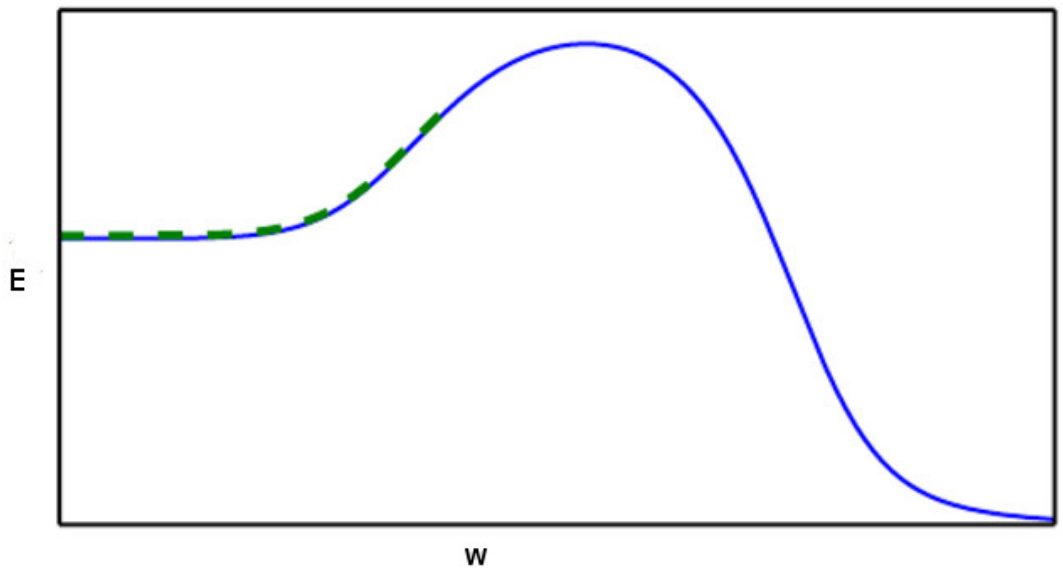


Issues in gradient descent – too fast descent

- steep cliffs: the gradient is extremely large, descent skips important weight vectors



Issues in gradient descent – local vs global structure



What if we initialize on the left?

Issues in computing the gradient

- vanishing and exploding gradients

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj} \qquad \text{for } j \in Y$$

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj} \qquad \text{for } j \in Z \setminus (Y \cup X)$$

- inexact gradient computation:
  - Minibatch gradient is only an estimate of the true gradient.
  - Note that the variance of the estimate is (roughly) $\sigma / \sqrt{m}$ where m is the size of the minibatch and σ is the variance of the gradient estimate for a single training example.
    (E.g. minibatch size 10 000 means 100 times more computation than the size 100 but gives only 10 times less variance.)

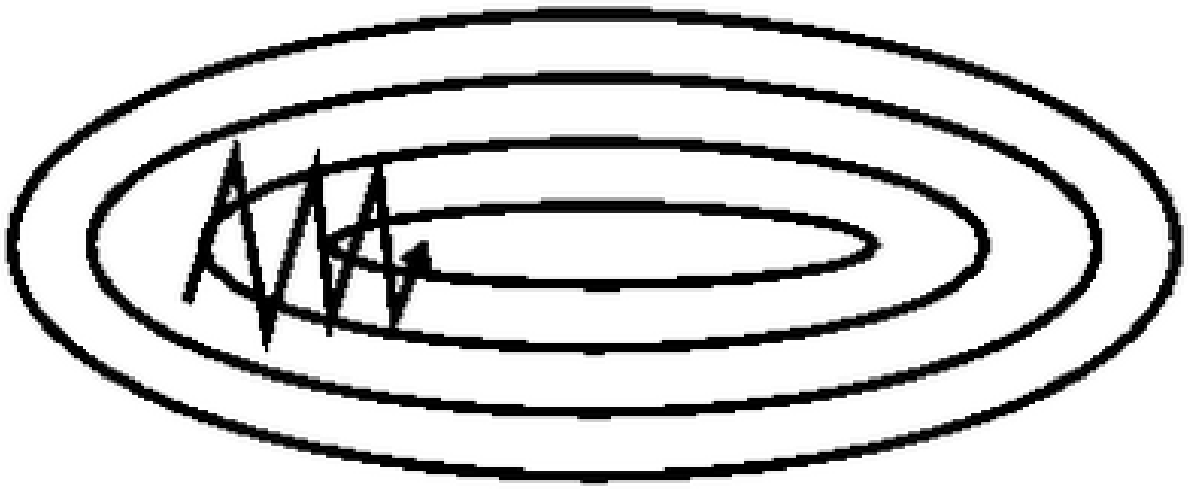
Minibatch size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches.
- If all examples in the batch are to be processed in parallel (as is the typical case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process.

Moment

Issue in the gradient descent:

- $\nabla E(\vec{w}^{(t)})$ constantly changes direction (but the error steadily decreases).

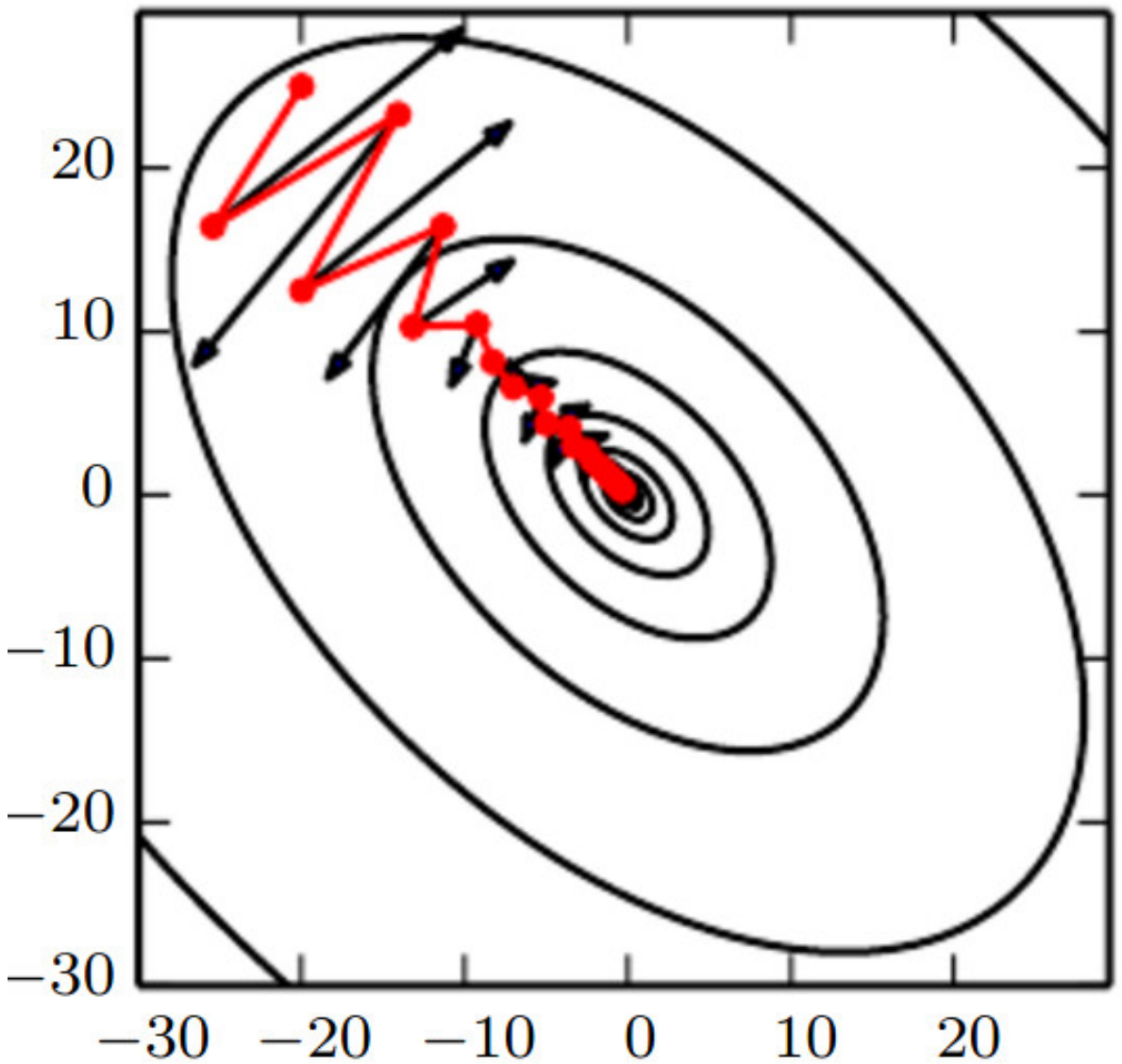


Solution: In every step add the change made in the previous step (weighted by a factor α):

$$\Delta \vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)}) + \alpha \cdot \Delta w_{ji}^{(t-1)}$$

where $0 < \alpha < 1$.

Momentum – illustration



SGD with momentum

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

where

$$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \sum_{k \in T} \nabla E_k(\vec{w}^{(t)}) + \alpha \Delta\vec{w}^{(t-1)}$$

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $0 < \alpha < 1$ measures the "influence" of the moment
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially.

Learning rate

Generic rules for adaptation of $\varepsilon(t)$

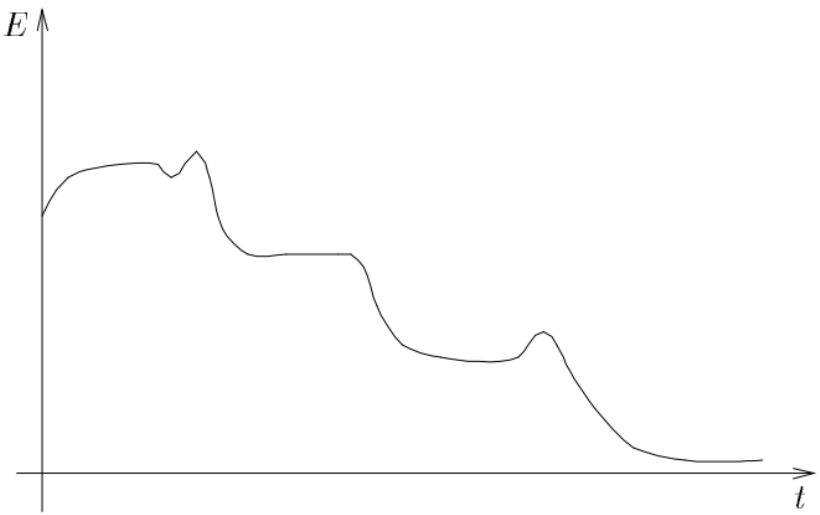
- Start with a larger learning rate (e.g. $\varepsilon = 0.1$).

  Later decrease as the descent is supposed to settle in a minimum of E.

  Some tools allow to set a list of learning rates, each rate for one epoch of the descent.

In case you may observe the error evolving:

- If the error decreases, increase slightly the rate.
- If the error increases, decrease the rate.
- Note that the error may increase for the short period without any harm to convergence of the learning process.



AdaGrad

So far we have considered a uniform learning rate.

It is better to have

- larger rates for weights with smaller updates,
- smaller rates for weights with larger updates.

AdaGrad uses individually adapting learning rate for each weight.

SGD with AdaGrad

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), compute $\vec{w}^{(t+1)}$:
  - Choose (randomly) a minibatch $T \subseteq \{1, \ldots, p\}$
  - Compute

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\frac{\eta}{\sqrt{r_j^{(t)} + \delta}} \cdot \sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)})$$

and

$$r_j^{(t)} = r_j^{(t-1)} + \left(\sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)}) \right)^2$$

- η is a constant expressing the influence of the learning rate, typically 0.01.
- $\delta > 0$ is a smoothing term (typically 1e-8) avoiding division by 0.

RMSProp

The main disadvantage of AdaGrad is the accumulation of the gradient throughout the whole learning process.

In case the learning needs to get over several "hills" before settling in a deep "valley", the weight updates get far too small before getting to it.

RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

SGD with RMSProp

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2 \ldots$), compute $\vec{w}^{(t+1)}$:
  - Choose (randomly) a minibatch $T \subseteq \{1, \ldots, p\}$
  - Compute

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$$

where

$$\Delta w_{ji}^{(t)} = -\frac{\eta}{\sqrt{r_j^{(t)} + \delta}} \cdot \sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)})$$

and

$$r_j^{(t)} = \rho r_j^{(t-1)} + (1 - \rho) \left(\sum_{k \in T} \frac{\partial E_k}{\partial w_{ji}}(\vec{w}^{(t)}) \right)^2$$

- η is a constant expressing the influence of the learning rate (Hinton suggests $\rho = 0.9$ and $\eta = 0.001$).
- $\delta > 0$ is a smoothing term (typically 1e-8) avoiding division by 0.

Other optimization methods

There are more methods such as AdaDelta, Adam (roughly RMSProp combined with momentum), etc.

A natural question: Which algorithm should one choose?

Unfortunately, there is currently no consensus on this point.

According to a recent study, the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta) performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam.

The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm.

Choice of (hidden) activations

Generic requirements imposed on activation functions:

1. differentiability

   (to do gradient descent)

2. non-linearity

   (linear multi-layer networks are equivalent to single-layer)

3. monotonicity

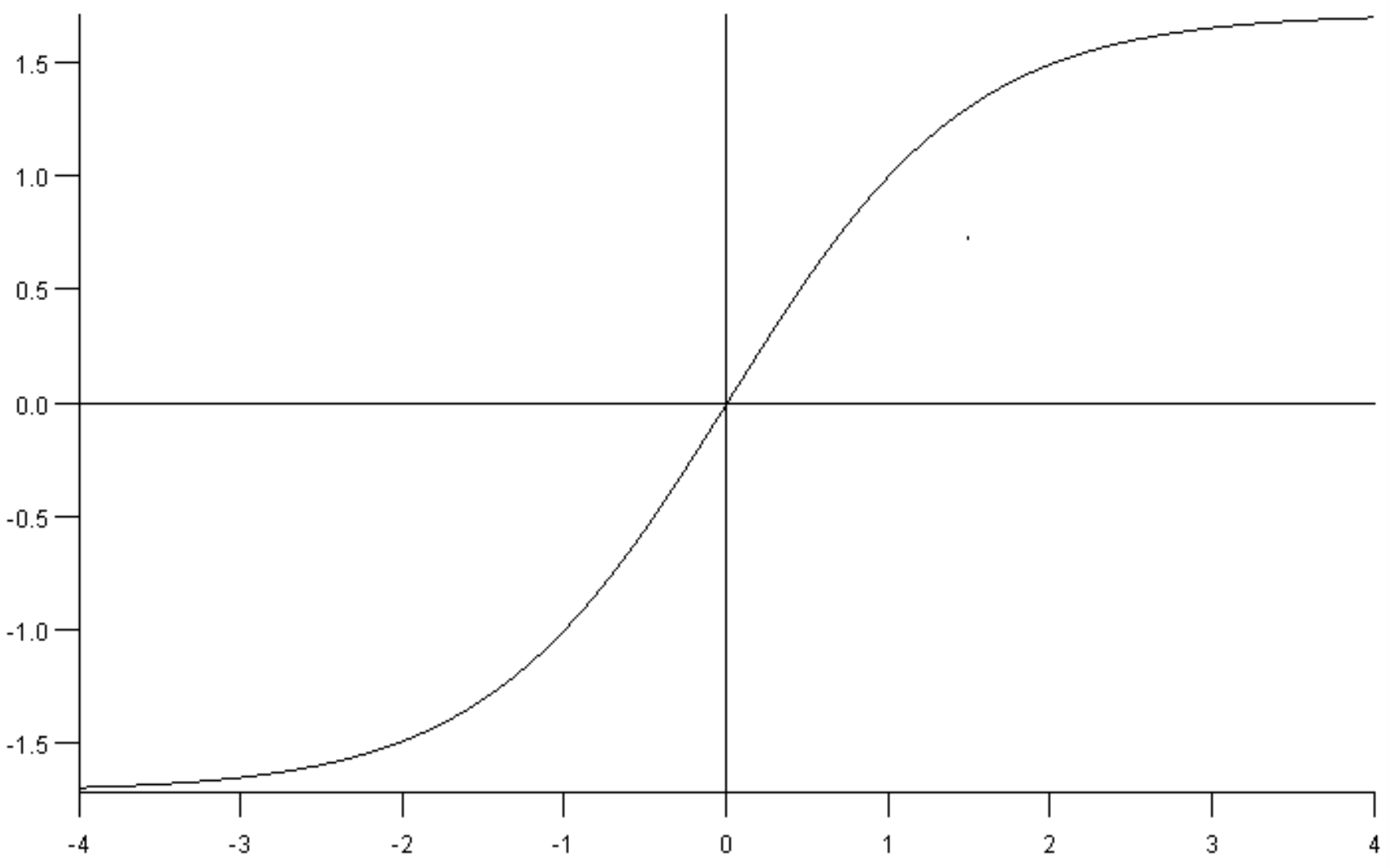
   (local extrema of activation functions induce local extrema of the error function)

4. "linearity"

   (i.e. preserve as much linearity as possible; linear models are easiest to fit; find the "minimum" non-linearity needed to solve a given task)

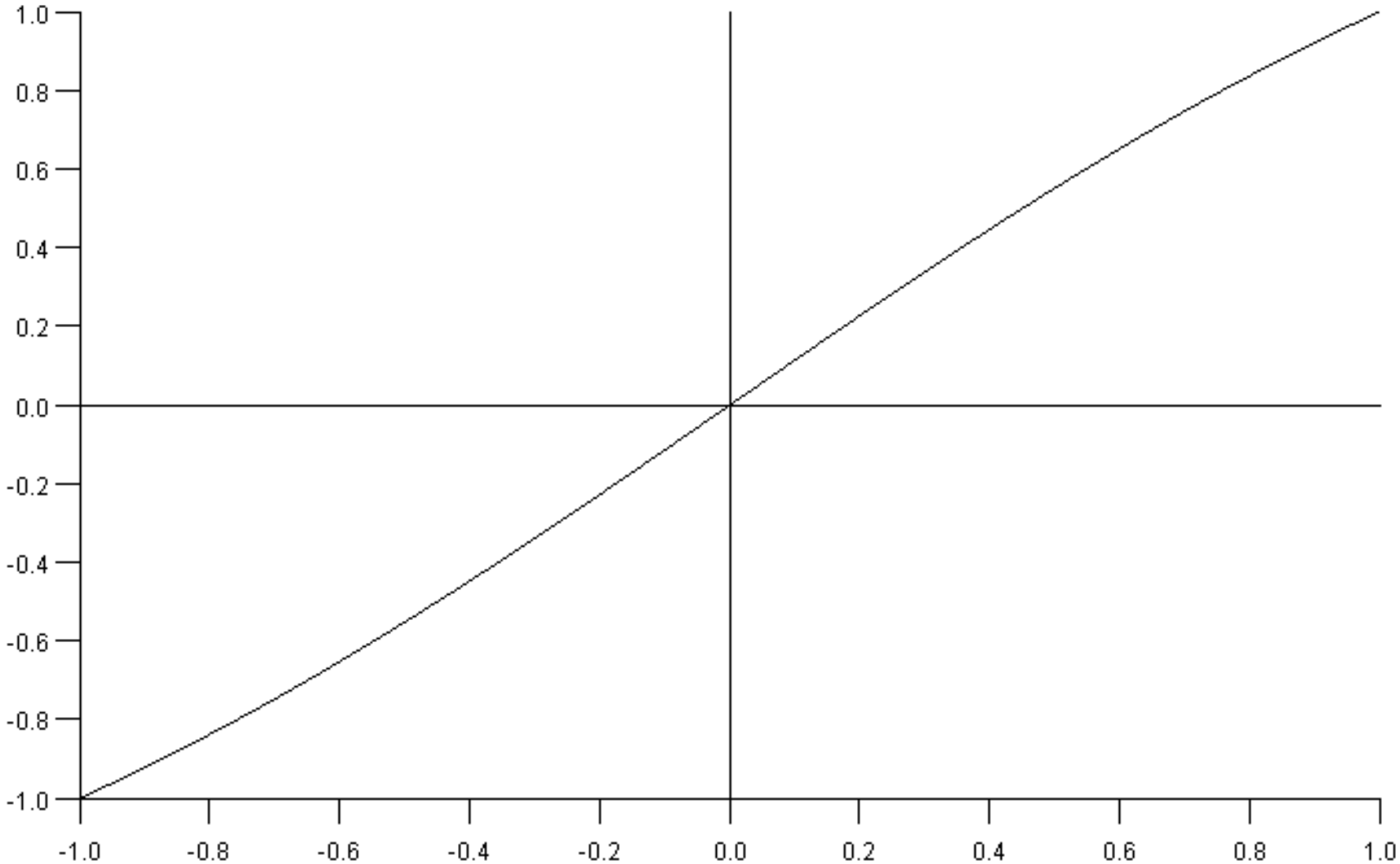
The choice of activation functions is closely related to input preprocessing and the initial choice of weights. I will illustrate the reasoning on sigmoidal functions; say few words about other activation functions later.

Activation functions – tanh



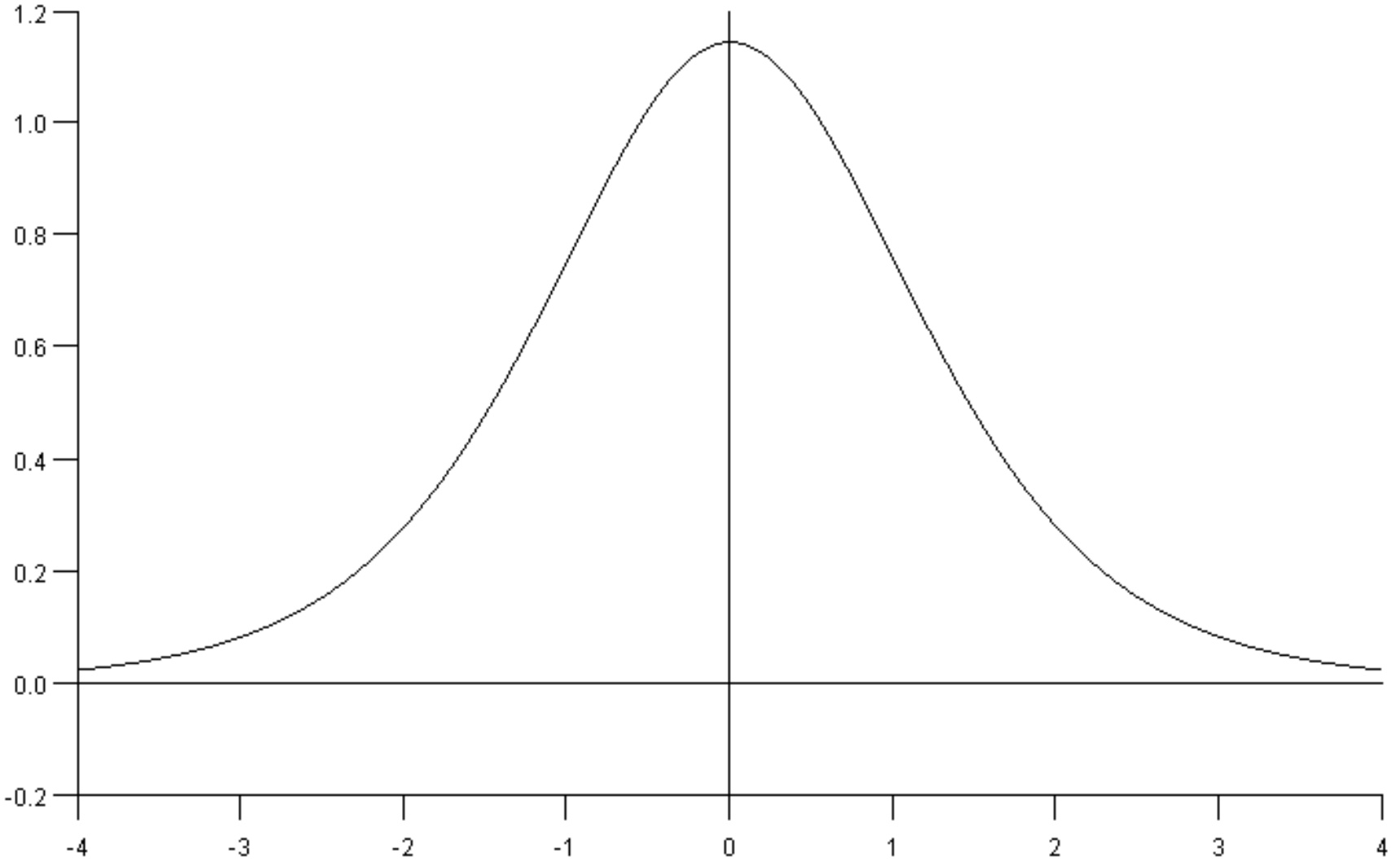
$\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$, we have $\lim_{\xi \to \infty} \sigma(\xi) = 1.7159$ and $\lim_{\xi \to -\infty} \sigma(\xi) = -1.7159$

Activation functions – tanh



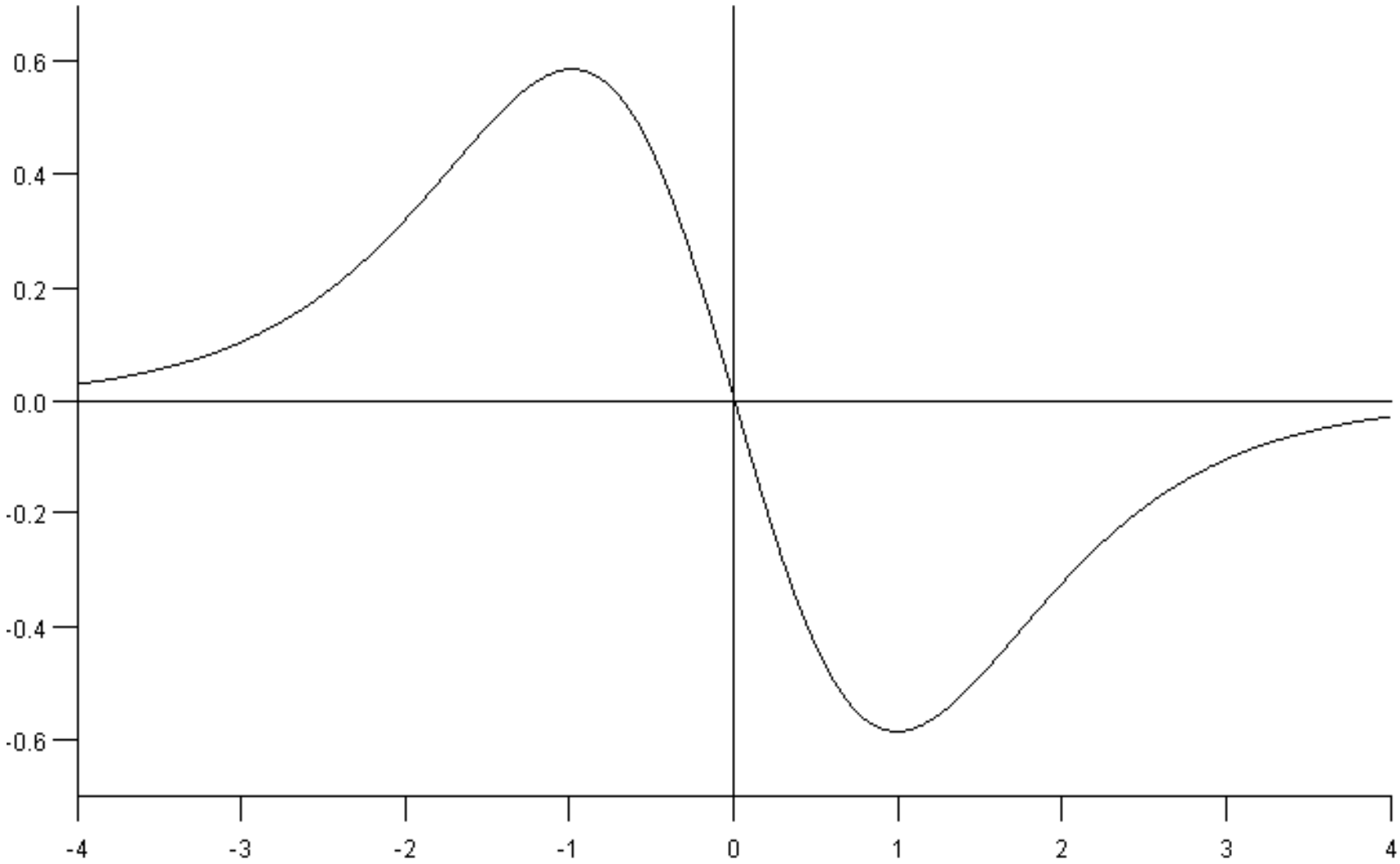
$\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$ is almost linear on $[-1, 1]$

Activation functions – tanh



first derivative: $\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$

Activation functions – tanh



second derivative: $\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$

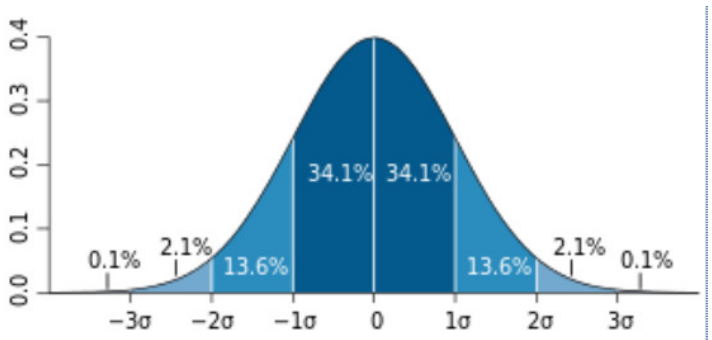
Input preprocessing

- Some inputs may be much larger than others.

  E.g..: Height vs weight of a person, maximum speed of a car (in km/h) vs its price (in CZK), etc.

- Large inputs have greater influence on the training than the small ones. In addition, too large inputs may slow down learning (saturation of activation functions).
- Typical standardization:
  - average $= 0$ (subtract the mean)
  - variance $= 1$ (divide by the standard deviation)

  Here the mean and standard deviation may be estimated from data (the training set).

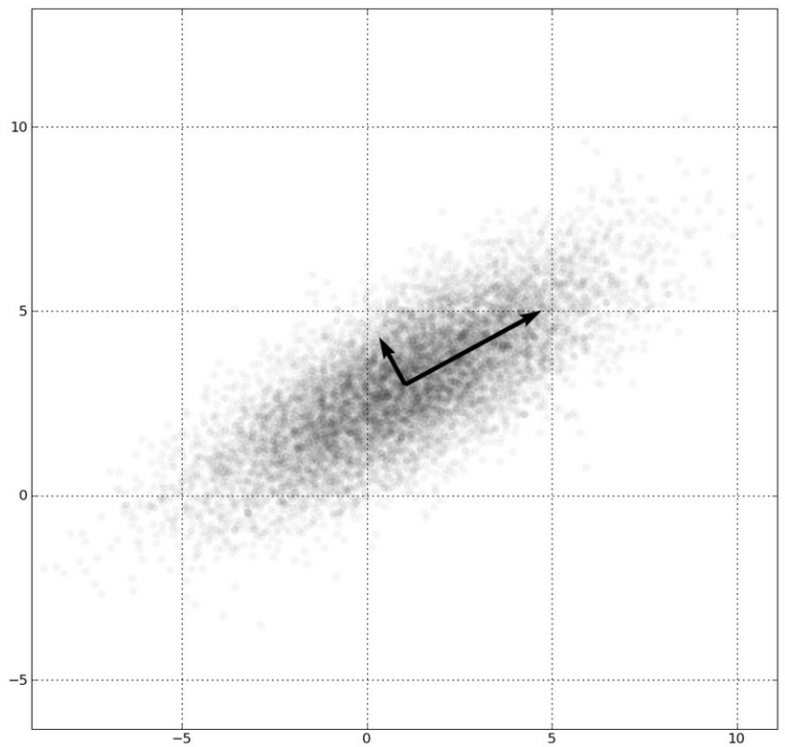


(illustration of standard deviation)

Input preprocessing

- Individual inputs should not be correlated.
- Correlated inputs can be removed as a part of *dimensionality reduction*.

  (Dimensionality reduction and decorrelation can be implemented using neural networks. There are also standard methods such as PCA.)



Initial weights (for tanh)

- Typically, the weights are chosen randomly from an interval $[-w, w]$ where w depends on the number of inputs of a given neuron.
- Consider the activation function $\sigma(\xi) = 1.7159 \cdot \tanh(\frac{2}{3} \cdot \xi)$ for all neurons.
  - σ is almost linear on $[-1, 1]$,
  - extreme values of σ'' are close to -1 and 1,
  - σ saturates out of the interval $[-4, 4]$ (i.e. it is close to its limit values and its derivative is close to 0.

  Thus
  - for too small w we may get (almost) linear model.
  - for too large w (i.e. much larger than 1) the activations may get saturated and the learning will be very slow.

  Hence, we want to choose w so that the inner potentials of neurons will be roughly in the interval $[-1, 1]$.

Initial weights (for tanh)

- Standardization gives mean = 0 and variance = 1 of the input data. Assume that individual inputs are (almost) uncorrelated.
- Consider a neuron j from the first layer with d inputs. Assume that its weights are chosen uniformly from $[-w, w]$.
- **The rule:** choose w so that the *standard deviation* of ξ_j (denote by o_j) is close to the border of the interval on which σ_j is linear.

  In our case: $o_j \approx 1$.
- Our assumptions imply: $o_j = \sqrt{\frac{d}{3}} \cdot w$.

  Thus we put $w = \frac{\sqrt{3}}{\sqrt{d}}$.
- The same works for higher layers, d corresponds to the number of neurons in the layer one level lower.

Glorot & Bengio initialization

The previous heuristics for weight initialization ignores variance of the gradient (i.e. it is concerned only with the "size" of activations in the forward pass.

Glorot & Bengio (2010) presented a **normalized initialization** by choosing w uniformly from the interval:

$$\left(-\sqrt{\frac{6}{m+n'}}, \sqrt{\frac{6}{m+n}}\right)$$

Here m is the number of inputs to the neuron, m is the number of outputs of the neuron.

This is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance.

The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no non-linearities. Real neural networks obviously violate this assumption, but many strategies designed for the linear model perform reasonably well on its non-linear counterparts.

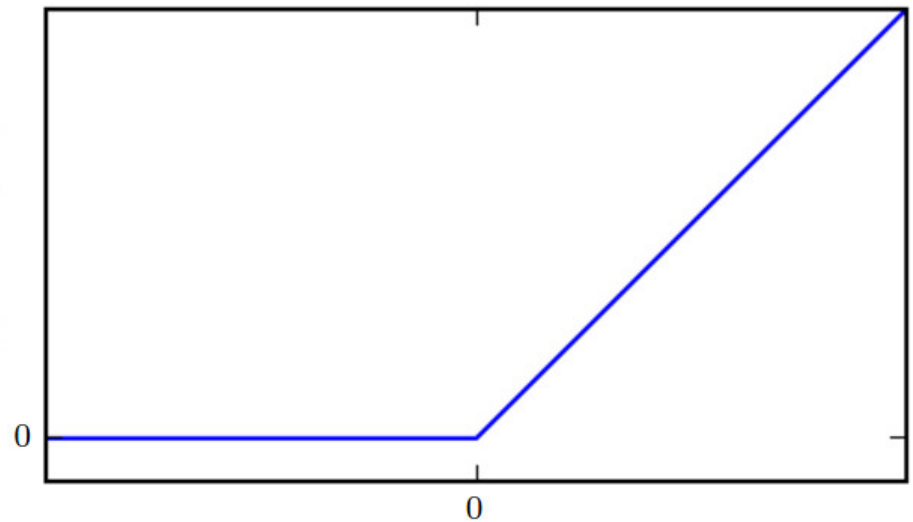
Target values (tanh)

- Target values d_{kj} should be chosen in the range of the output activation functions, in our case $[-1.716, 1.716]$.
- Target values too close to extrema of the output activations, in our case ± 1.716, may cause that the weights will grow indefinitely (slows down learning).
- Thus it is good to choose target values from the interval $[-1.716 + \delta, 1.716 - \delta]$.

  As before, ideally $[-1.716 + \delta, 1.716 - \delta]$ should span the interval on which the activation function is linear, i.e. d_{kj} should be taken from $[-1, 1]$.

Modern activation functions

For hidden neurons sigmoidal functions are often substituted with piece-wise linear activations functions. Most prominent is ReLU:



$$\sigma(\xi) = \max\{0, \xi\}$$

- THE default activation function recommended for use with most feedforward neural networks.
- As close to linear function as possible; very simple; does not saturate for large potentials.

Output neurons

The choice of activation functions for output units depends on the concrete applications.

For regression (function approximation) the output is typically linear (or sigmoidal).

For classification, the current activation functions of choice are

- logistic sigmoid or tanh – binary classification
- softmax:

$$\sigma_j(\xi_j) = \frac{e^{\xi_j}}{\sum_{i \in Y} e^{\xi_i}}$$

  for multi-class classification.

For some reasons the error function used with softmax (assuming that the target values d_{kj} are from $\{0, 1\}$) is typically **cross-entropy**:

$$-\frac{1}{p} \sum_{k=1}^{p} \sum_{j \in Y} \left[d_{kj} \ln(y_j) + (1 - d_{kj}) \ln(1 - y_j) \right]$$

... which somewhat corresponds to the maximum likelihood principle.

Sigmoidal outputs with cross-entropy – in detail

Consider

- Binary classification, two classes $\{0, 1\}$
- One output neuron j, its activation logistic sigmoid

$$\sigma_j(\xi_j) = \frac{1}{1 + e^{-\xi_j}}$$

The output of the network is $y = \sigma_j(\xi_j)$.

- For a training set

$$\mathcal{T} = \left\{ \left(\vec{x}_k, d_k \right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

(here $\vec{x}_k \in \mathbb{R}^{|X|}$ and $d_k \in \mathbb{R}$), the cross-entropy looks like this:

$$E^{cross} = -\frac{1}{p} \sum_{k=1}^{p} [d_k \ln(y_k) + (1 - d_k) \ln(1 - y_k)]$$

where y_k is the output of the network for the k-th training input $\vec{x}_k$, and d_k is the k-th desired output.

Generalization

Intuition: Generalization = ability to cope with new unseen instances.

Data are mostly noisy, so it is not good idea to fit exactly.

In case of function approximation, the network should not return exact results as in the training set.

More formally: It is typically assumed that the training set has been generated as follows:

$$d_{kj} = g_j(\vec{x}_k) + \Theta_{kj}$$

where g_j is the "underlying" function corresponding to the output neuron $j \in Y$ and Θ_{kj} is random noise.

The network should fit g_j not the noise.

Methods improving generalization are called **regularization methods**.

Regularization

Regularization is a big issue in neural networks, as they typically use a huge amount of parameters and thus are very susceptible to overfitting.

von Neumann: **"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."**

... and I ask you prof. Neumann:

What can you fit with 40GB of parameters??

Early stopping

Early stopping means that we stop learning before it reaches a minimum of the error E.

When to stop?

In many applications the error function is not the main thing we want to optimize.

E.g. in the case of a trading system, we typically want to maximize our profit not to minimize (strange) error functions designed to be easily differentiable.

Also, as noted before, minimizing E completely is not good for generalization.

For start: We may employ standard approach of training on one set and stopping on another one.

Early stopping

Divide your dataset into several subsets:

- **training set** (e.g. 60%) – train the network here
- **validation set** (e.g. 20%) – use to stop the training
- (possibly) **test set** (e.g. 20%) – use to compare trained models

What to use as a stopping rule?

You may observe E (or any other function of interest) on the validation set, if it does not improve for last k steps, stop.

Alternatively, you may observe the gradient, if it is small for some time, stop.

(recent studies shown that this traditional rule is not too good: it may happen that the gradient is larger close to minimum values; on the other hand, E does not have to be evaluated which saves time.

To compare models you may use ML techniques such as cross-validation etc.

Size of the network

Similar problem as in the case of the training duration:

- Too small network is not able to capture intrinsic properties of the training set.
- Large networks overfit faster – bad generalization.

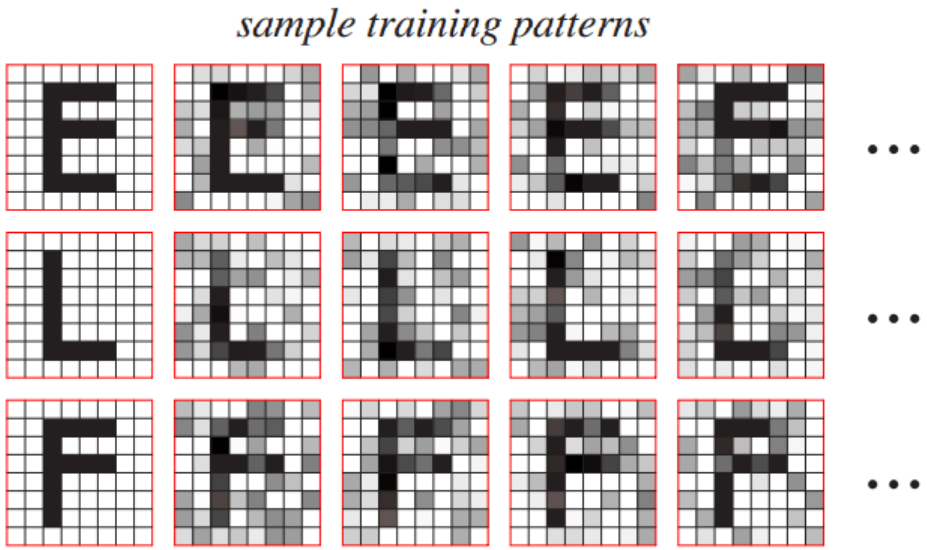
Solution: Optimal number of neurons :-)

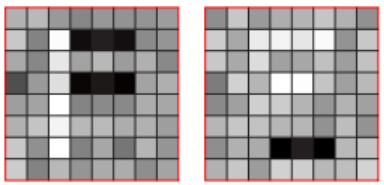
- there are some (useless) theoretical bounds
- there are algorithms dynamically adding/removing neurons (not much use nowadays)
- In practice:
  - start using a rule of thumb: the number of neurons $\approx$ ten times less than the number of training instances.
  - experiment, experiment, experiment.

Feature extraction

Consider a two layer network. Hidden neurons are supposed to represent "patterns" in the inputs.

Example: Network 64-2-3 for letter classification:





Ensemble methods

Techniques for reducing generalization error by combining several models.

The reason that ensemble methods work is that different models will usually not make all the same errors on the test set.

Idea: Train several different models separately, then have all of the models vote on the output for test examples.

Bagging:

- Generate k training sets $T_1, ..., T_k$ of the same size by *sampling from* $\mathcal{T}$ *uniformly with replacement*.

  If $|T_i| = |\mathcal{T}|$, then on average $|T_i| = (1 - 1/e)|\mathcal{T}|$.
- For each i, train a model M_i on T_i.
- Combine outputs of the models: for regression by averaging, for classification by (majority) voting.

Dropout

The algorithm: In every step of the gradient descent

- choose randomly a set N of neurons, each neuron is included in N independently with probability $1/2$,
  (in practice, different probabilities are used as well).
- update weights of neurons in N (in a standard way), leave weights of the other neurons unchanged.

Dropout resembles bagging: Large ensemble of neural networks is trained "at once" on parts of the data.

Dropout is not exactly the same as bagging: The models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

In the case of bagging, each model is trained to convergence on its respective training set. This would be infeasible for large networks/training sets.

Weight decay

Generalization can be improved by removing "unimportant" weights.

Penalising large weights gives stronger indication about their importance.

In every step we decrease weights (multiplicatively) as follows:

$$w_{ji}^{(t+1)} = (1 - \zeta)(w_{ji}^{(t)} + \Delta w_{ji}^{(t)})$$

Intuition: Unimportant weights will be pushed to 0, important weights will survive the decay.

Weight decay is equivalent to the gradient descent with a constant learning rate ε and the following error function:

$$E'(\vec{w}) = E(\vec{w}) + \frac{2\zeta}{\varepsilon}(\vec{w} \cdot \vec{w})$$

Here $\frac{2\zeta}{\varepsilon}(\vec{w} \cdot \vec{w})$ penalizes large weights.

More optimization, regularization ...

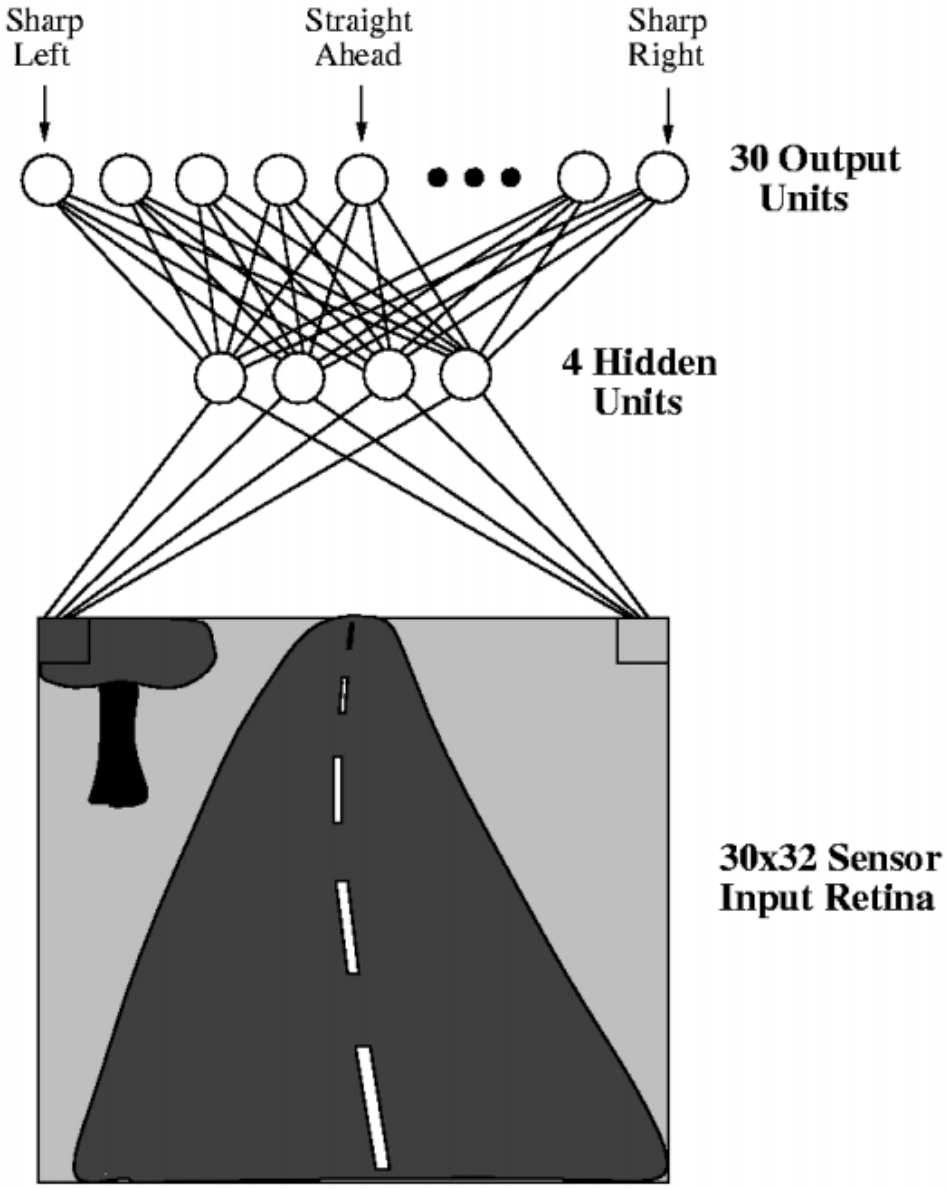
There are many more practical tips, optimization methods, regularization methods, etc.

For a very nice survey see

`http://www.deeplearningbook.org/`

... and also all other infinitely many urls concerned with deep learning.

ALVINN (history)



ALVINN

Architecture:

- MLP, 960 – 4 – 30 (also 960 – 5 – 30)
- inputs correspond to pixels

Activity:

- activation functions: logistic sigmoid
- Steering wheel position determined by "center of mass" of neuron values.

ALVINN

Learning: Trained during (live) drive.

- Front window view captured by a camera, 25 images per second.
- Training samples of the form $(\vec{x}_k, \vec{d}_k)$ where
  - $\vec{x}_k$ = image of the road
  - $\vec{d}_k$ = corresponding position of the steering wheel
- position of the steering wheel "blurred" by Gaussian distribution:

$$d_{ki} = e^{-D_i^2/10}$$

where D_i is the distance of the i-th output from the one which corresponds to the correct position of the wheel.

(The authors claim that this was better than the binary output.)

ALVINN – Selection of training samples

Naive approach: take images directly from the camera and adapt accordingly.

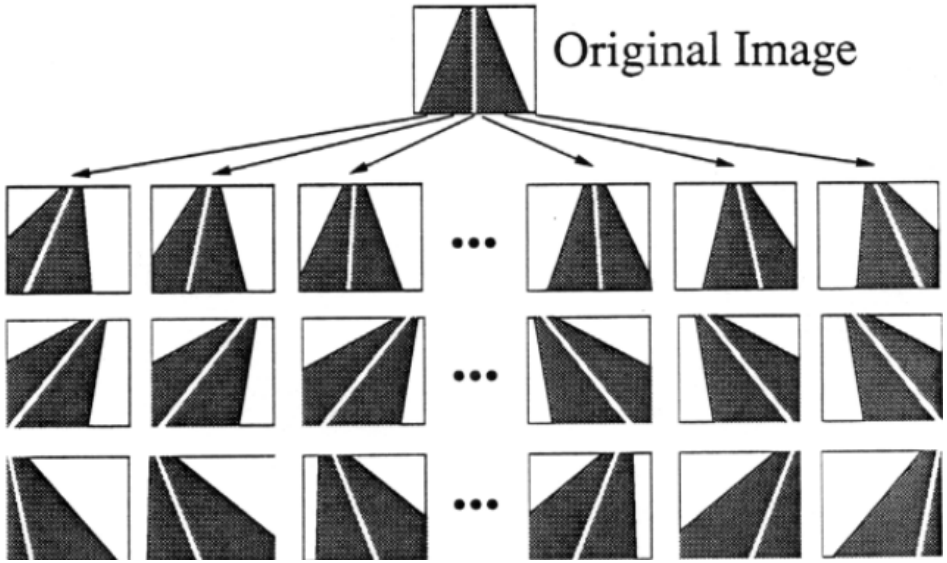
Problems:

- If the driver is gentle enough, the car never learns how to get out of dangerous situations. A solution may be
  - turn off learning for a moment, then suddenly switch on, and let the net catch on,
  - let the driver drive as if being insane (dangerous, possibly expensive).
- The real view out of the front window is repetitive and boring, the net would overfit on few examples.

ALVINN – Selection of training examples

Problem with a "good" driver is solved as follows:

- 15 distorted copies of each image:



- desired output generated for each copy

"Boring" images solved as follows:

- a buffer of 200 images (including 15 copies of the original), in every step the system trains on the buffer
- after several updates a new image is captured, 15 copies are made and they will substitute 15 images in the buffer (5 chosen randomly, 10 with the **smallest** error).

ALVINN - learning

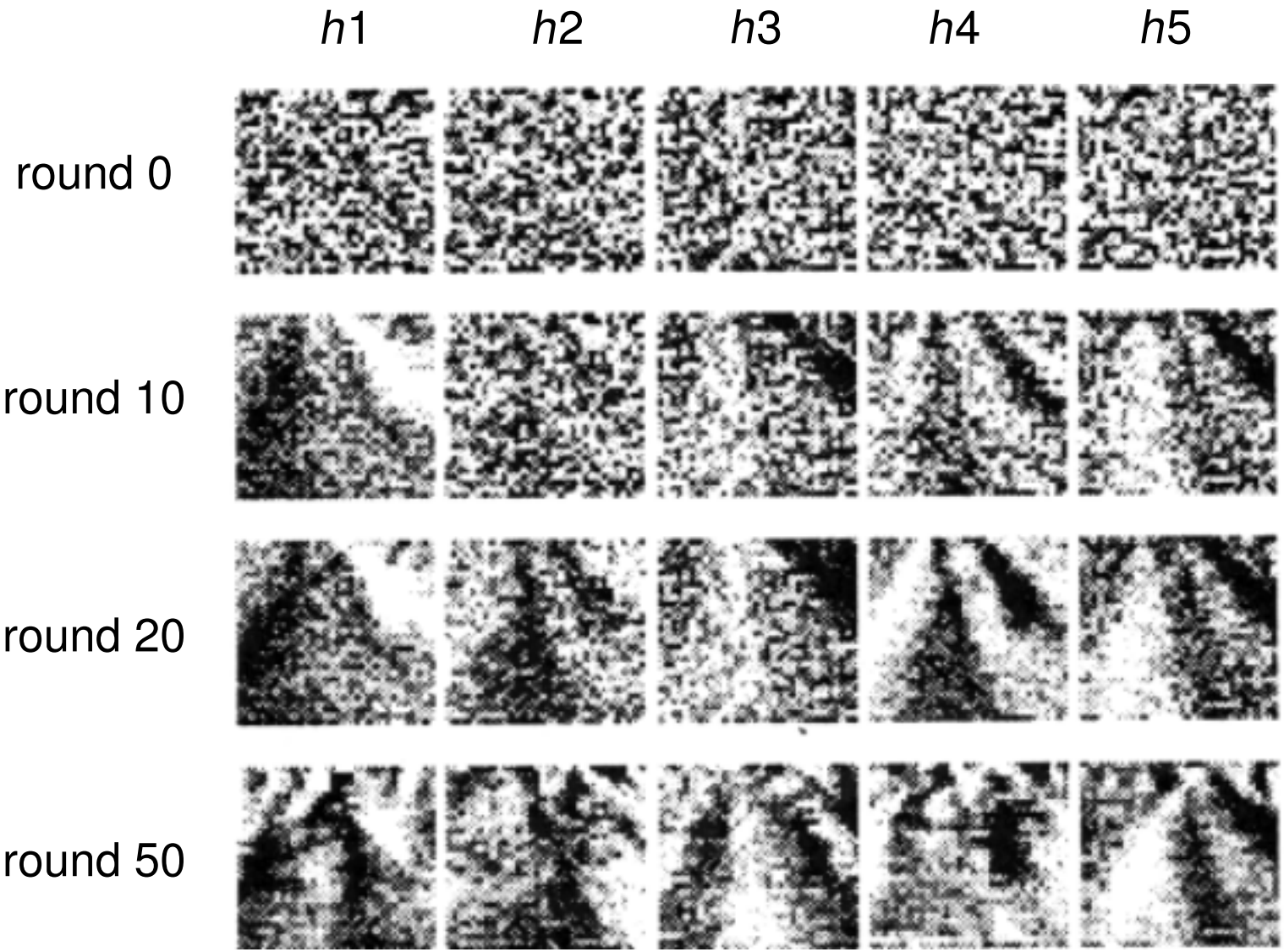
- pure backpropagation
- constant learning speed
- momentum, slowly increasing.

We used a learning rate of 0.015, a momentum term of 0.9, and we ramped up the learning rate and momentum using a rate term of 0.05. This means that the learning rate and momentum increase linearly over 20 epochs until they reach their maximum value (0.015 and 0.9, respectively). We also used a weight decay term of 0.0001.

Results:

- Trained for 5 minutes, speed 4 miles per hour.
- ALVINN was able to drive well on a new road it has never seen (in different weather conditions).
- The maximum speed was limited by the hydraulic controller of the steering wheel, not the learning algorithm.

ALVINN - weight development



Here $h1, \ldots, h5$ are hidden neurons.

ALVINN - comments

Compare ALVINN with explicit system development:
For driving you need to

- find key features for driving
  (ALVINN finds automatically)
- detect the features
  (ALVINN creates its own detectors)
- implement driving algorithm
  (ALVINN learns form the driver)

ALVINN was rather limited (but keep in mind that the net is **small**):

- just one type of road, no obstacles
- no higher level control

MNIST – handwritten digits recognition

- Database of labelled images of handwritten digits: 60 000 training examples, 10 000 testing.
- Dimensions: 28 x 28, digits are centered to the "center of gravity" of pixel values and normalized to fixed size.
- More at `http://yann.lecun.com/exdb/mnist/`

Fig. 4. Size-normalized examples from the MNIST database.

The database is used as a standard benchmark in lots of publications.

Allows comparison of various methods.

MNIST

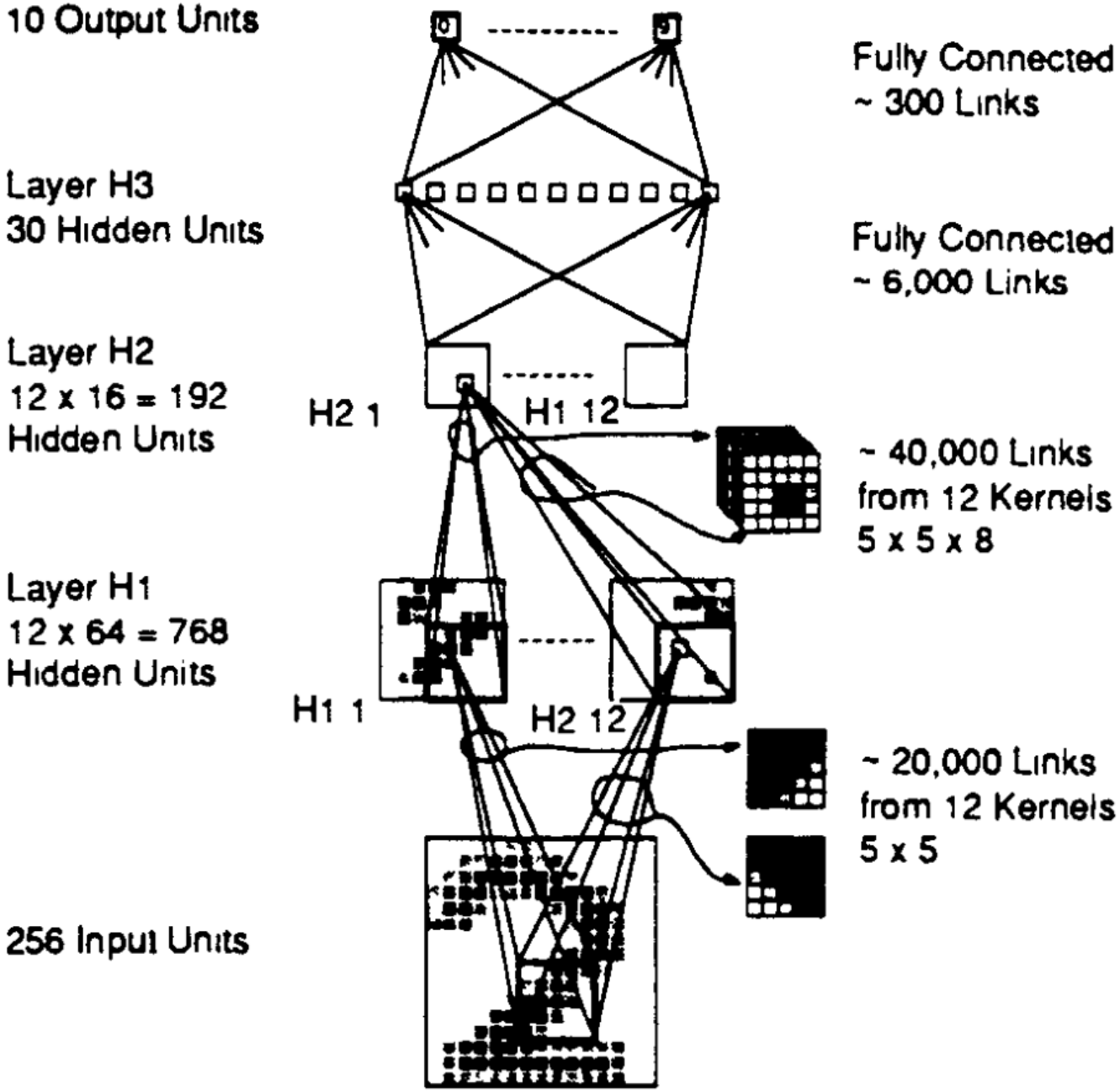
One of the best "old" results is the following:

6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU)
(Ciresan et al. 2010)

Abstrakt: Good old on-line back-propagation for plain multi-layer perceptrons yields a very low 0.35 error rate on the famous MNIST handwritten digits benchmark. All we need to achieve this best result so far are many hidden layers, many neurons per layer, numerous deformed training images, and graphics cards to greatly speed up learning.

A famous application of the first convolutional network LeNet-1 in 1998.

MNIST – LeNet1



MNIST – LeNet1

Activity: activation function: hyperbolic tangents

Interpretation of output:

- the output neuron with the highest value identifies the digit.
- the same, but if the two largest neuron values are too close together, the input is rejected (i.e. no answer).

Learning:

Inputs:

- training on 7291 samples, tested on 2007 samples

Training:

- modified backpropagation (conjugate gradients), online
- weights initialized uniformly from $[-2.4, 2.4]$, divided by the number of inputs to a given neuron

MNIST – results

- error on test set without rejection: 5%
- error on test set with rejection: 1% (12% rejected)

- compare with dense MLP with 40 hidden neurons: error 1% (19.4% rejected)

Modern convolutional networks

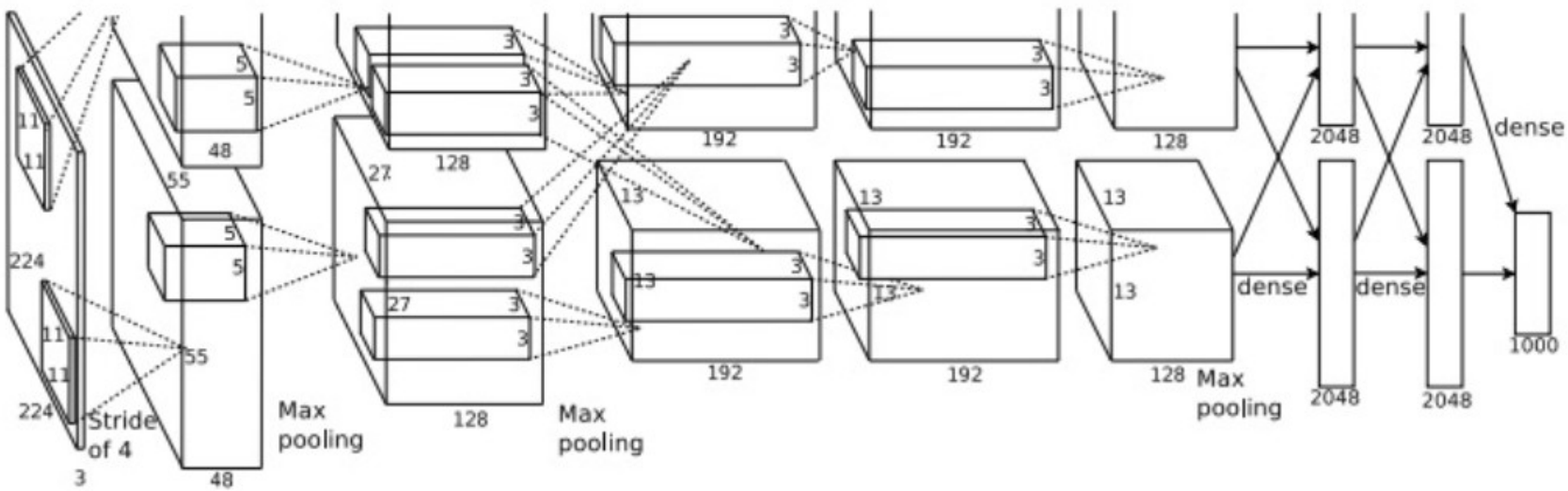
The rest of the lecture is based on the online book Neural Networks and Deep Learning by Michael Nielsen.
`http://neuralnetworksanddeeplearning.com/index.html`

- Convolutional networks are currently the best networks for image classification.
- Their common ancestor is LeNet-5 (and other LeNets) from nineties.

  Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998

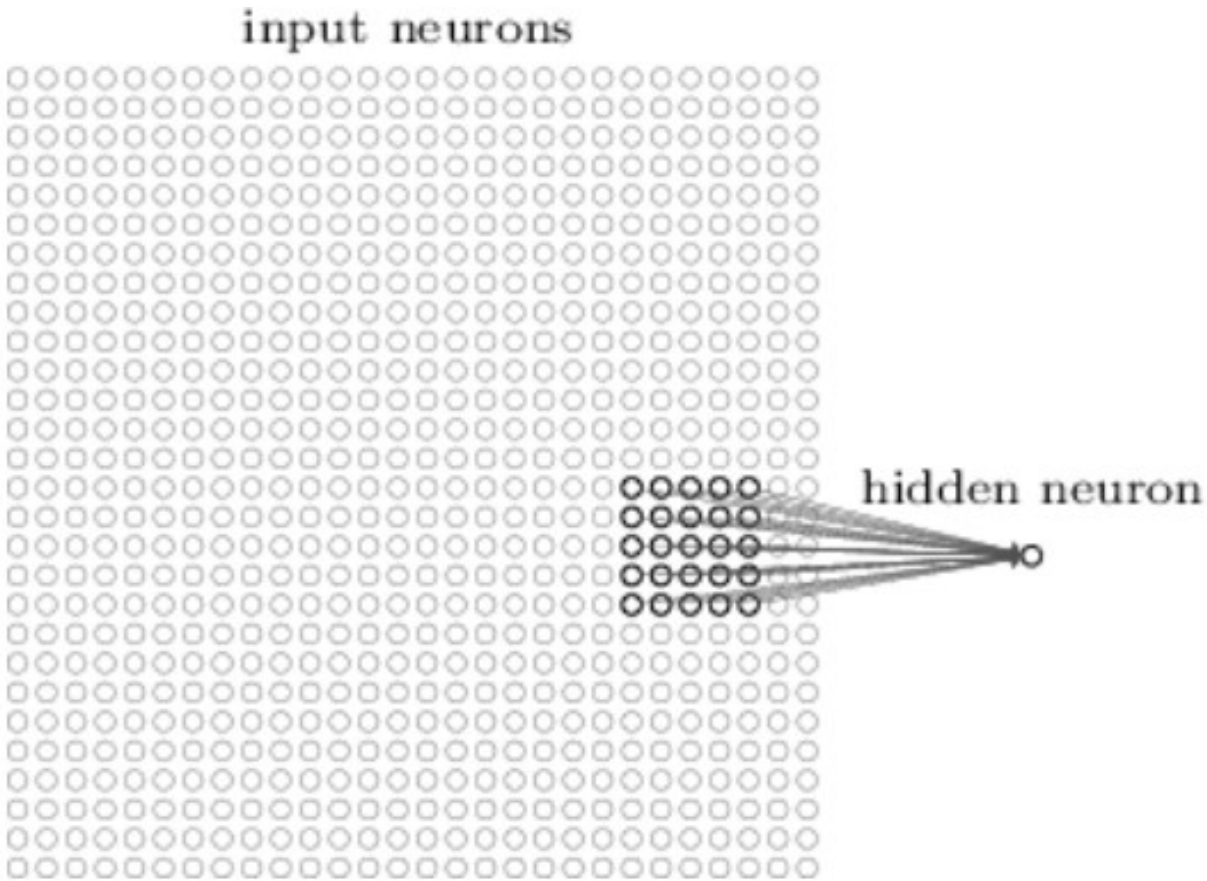
AlexNet

In 2012 this network made a breakthrough in ILVSCR competition, taking the classification error from around 28% to 16%:



A convolutional network, trained on two GPUs.

Convolutional networks - local receptive fields

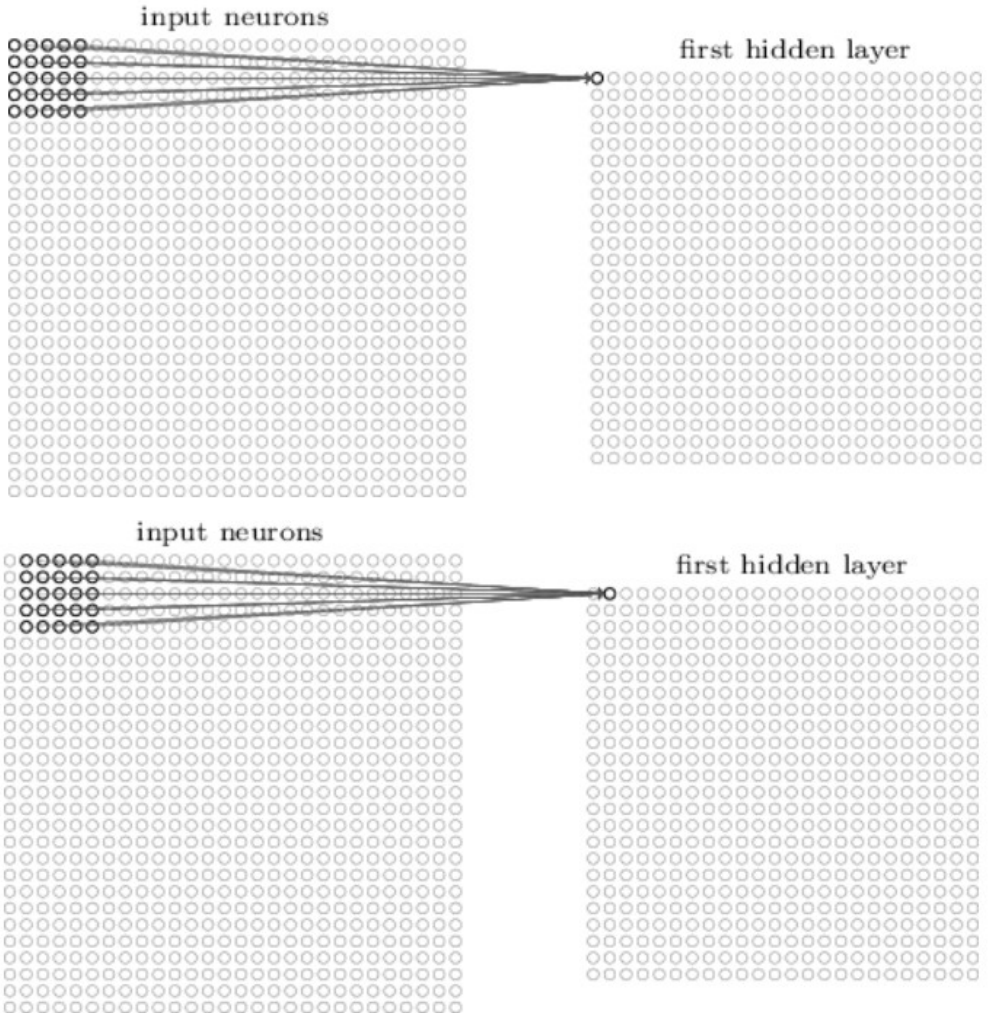


Every neuron is connected with a field of $k \times k$ (in this case 5×5) neurons in the lower layer (this filed is *receptive field*).

Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

Convolutional networks - stride length

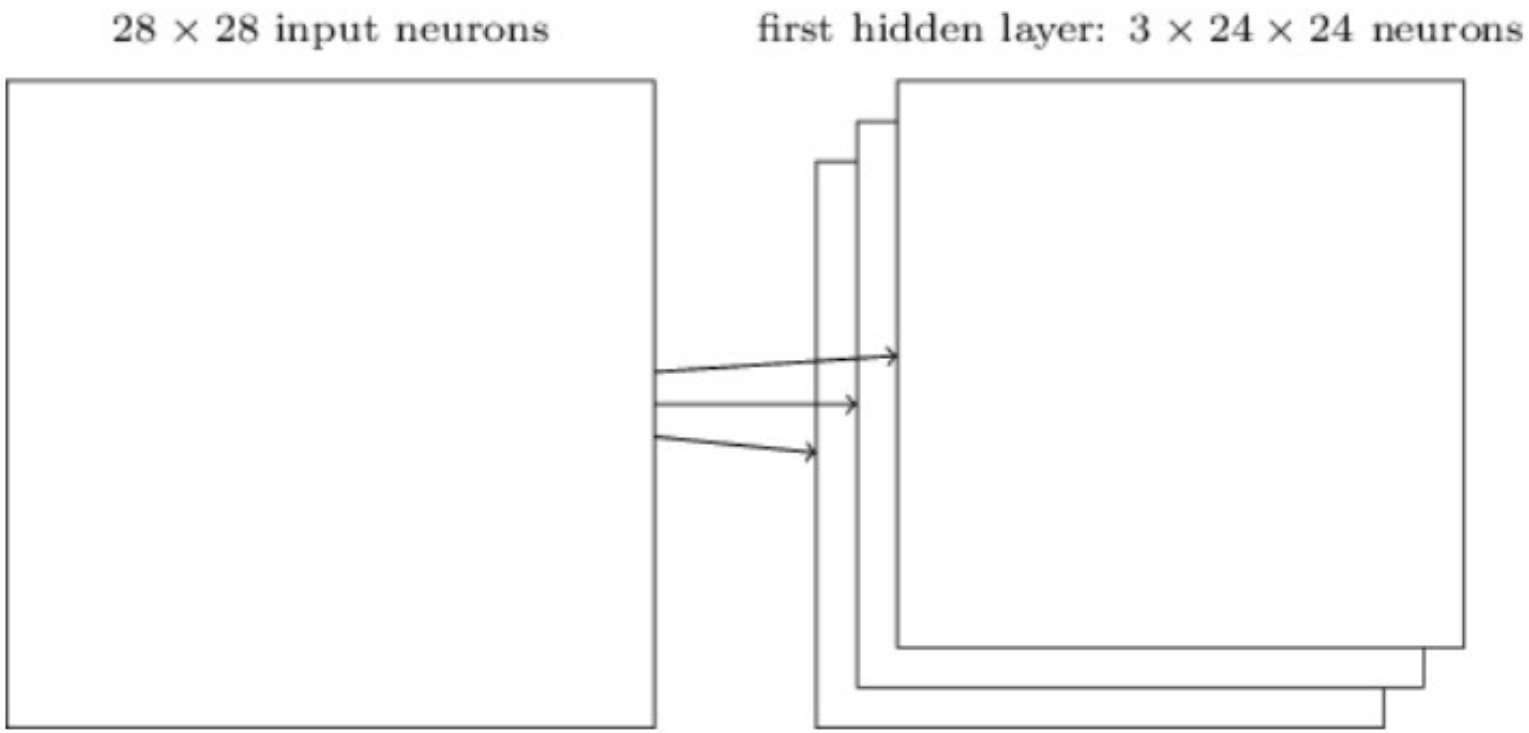
Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:



The "size" of the slide is called *stride length*.

The group of all such neurons is *feature map*. all these neurons *share weights and biases*!

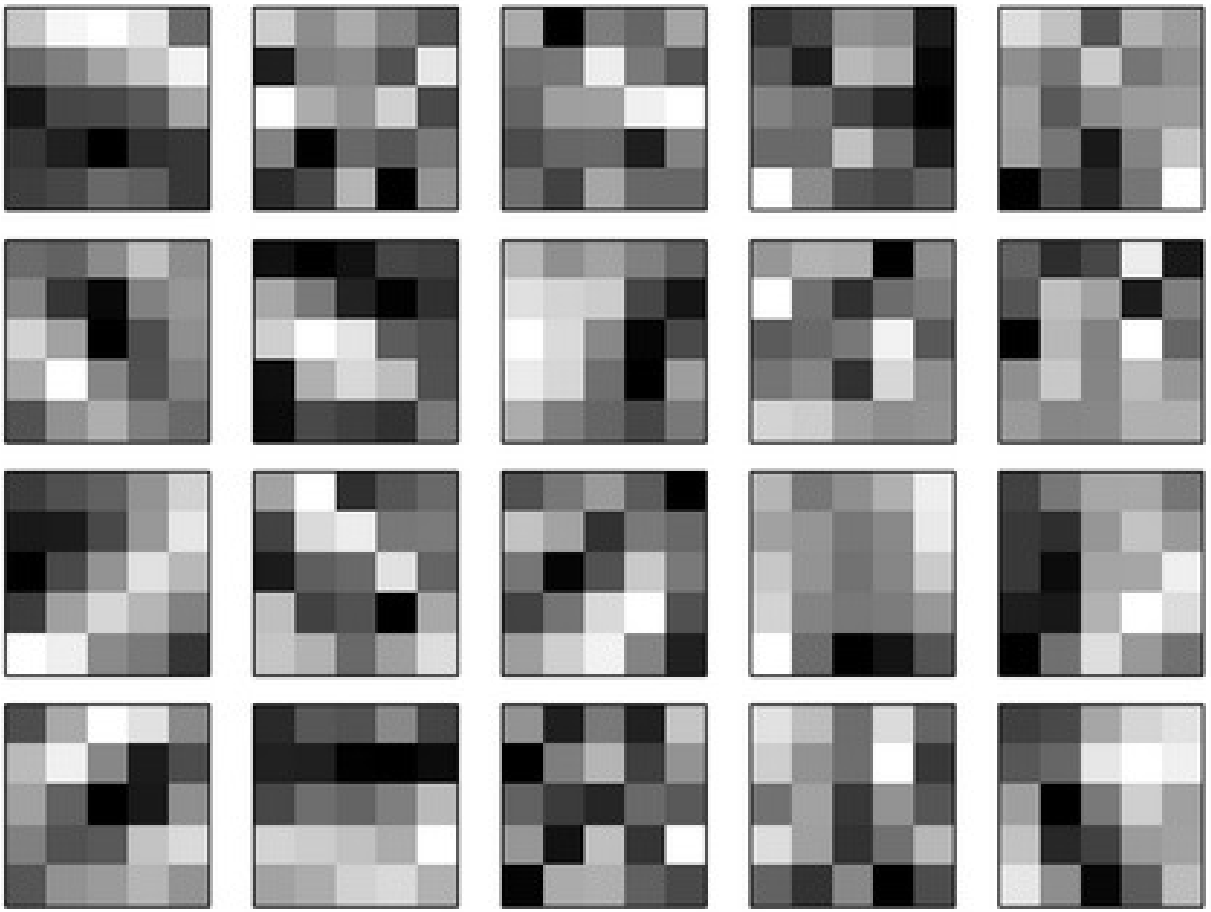
Feature maps



Each feature map represents a property of the input that is supposed to be spatially invariant.

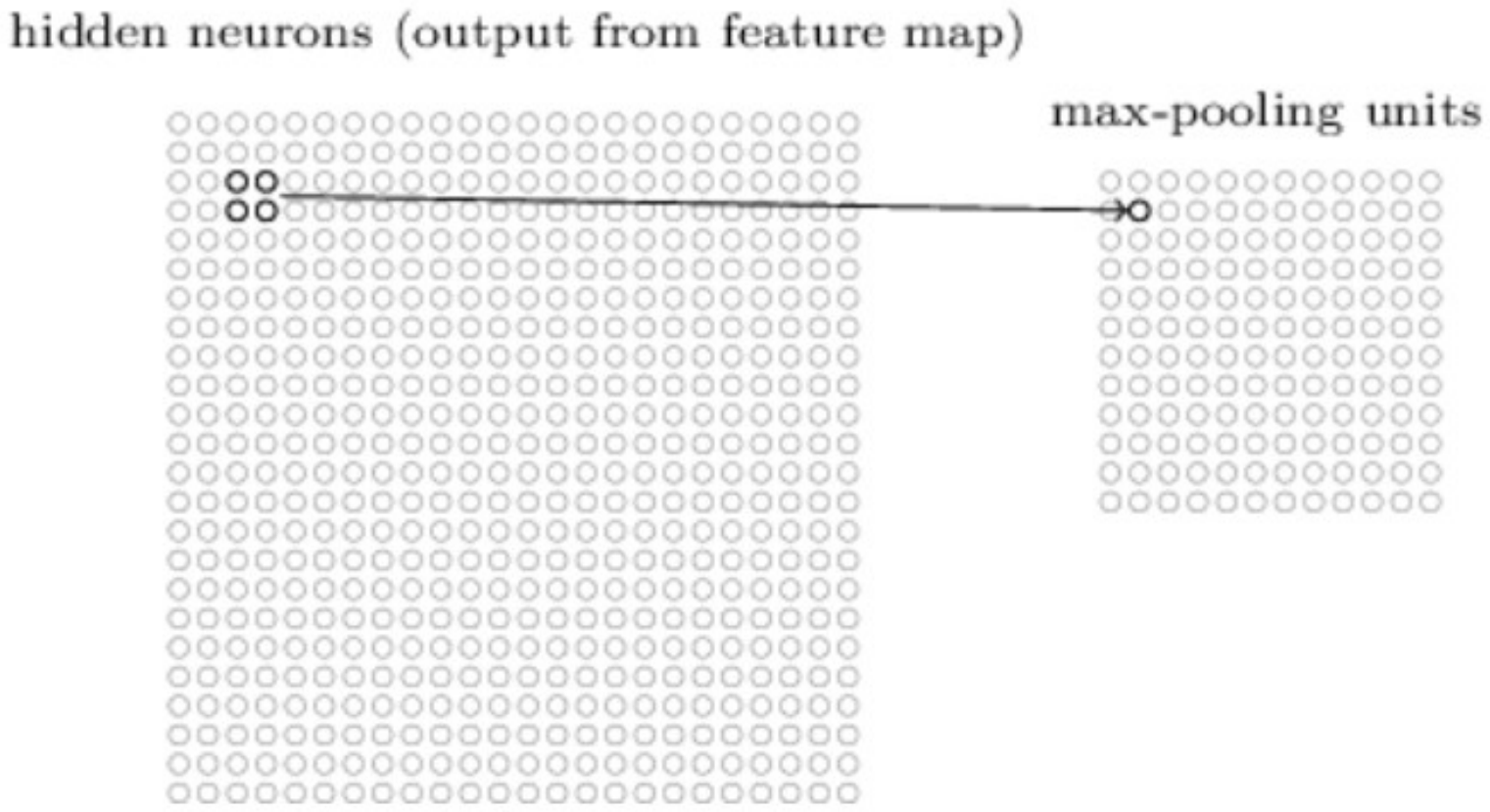
Typically, we consider several feature maps in a single layer.

Trained feature maps



(20 feature maps, receptive fields 5×5)

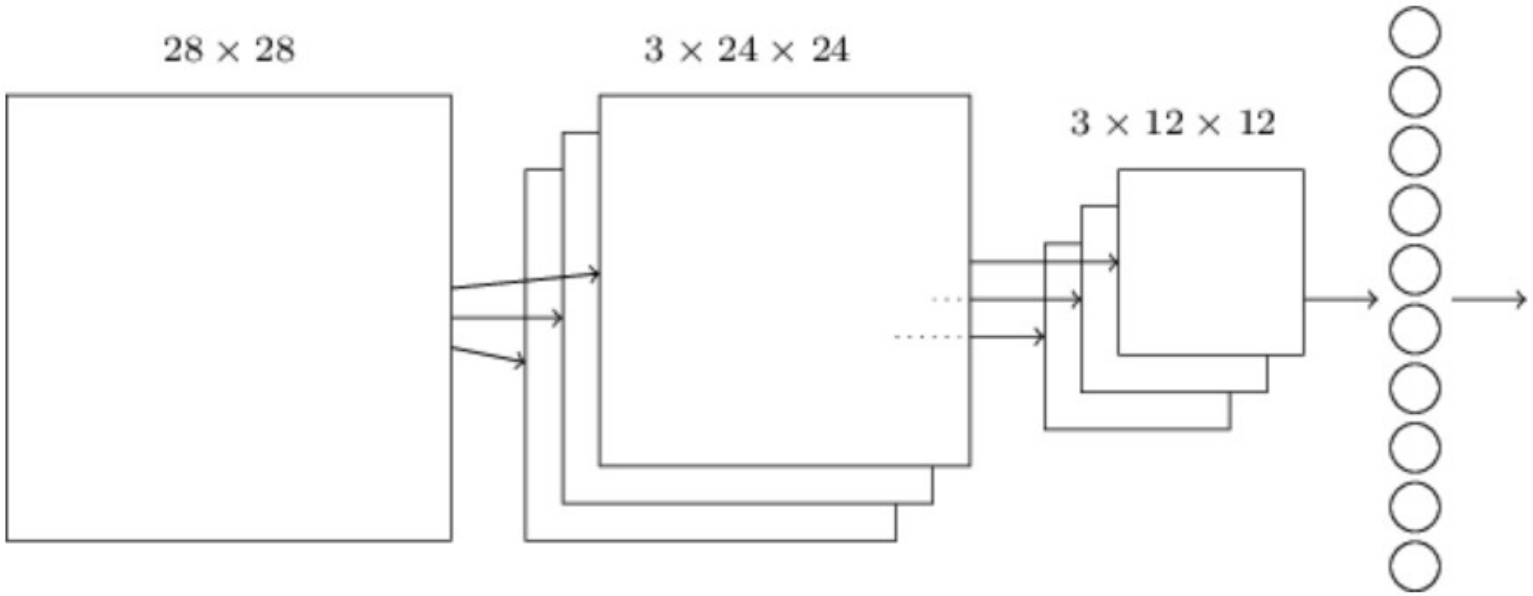
Pooling



Neurons in the pooling layer compute functions of their receptive fields:

- **Max-pooling** : maximum of inputs
- **L2-pooling** : square root of the sum of squres
- **Average-pooling** : mean
- ...

Simple convolutional network

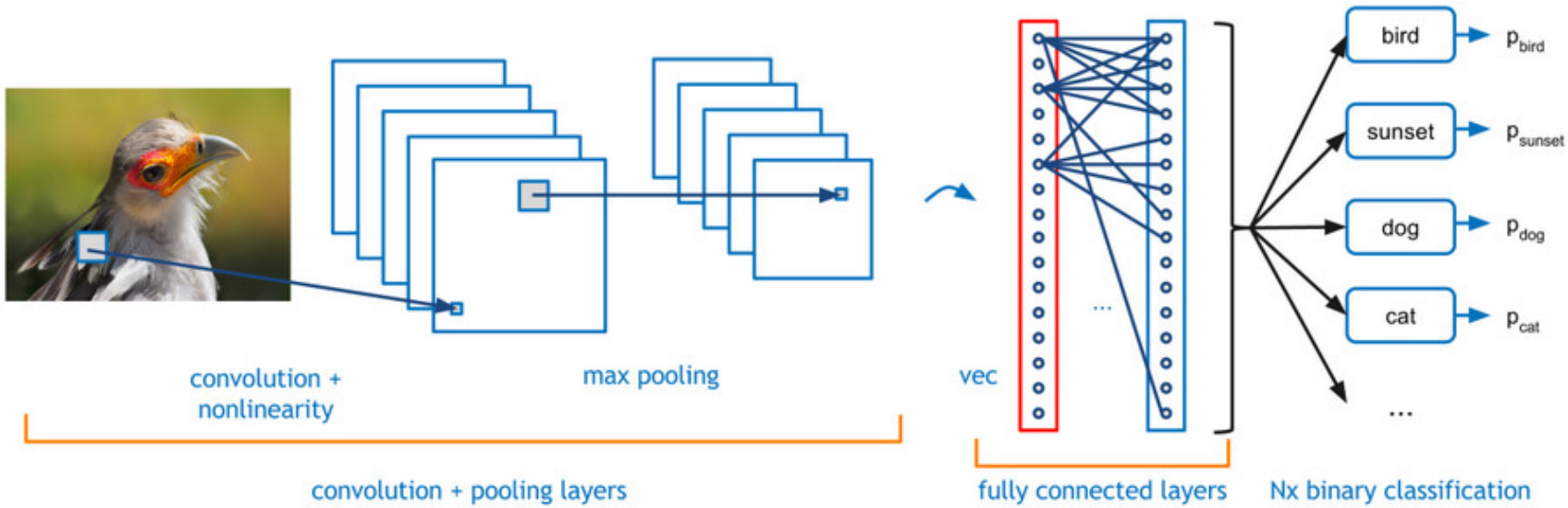


28×28 input image, 3 feature maps, each feature map has its own max-pooling (field 5×5, stride = 1), 10 output neurons.

Each neuron in the output layer gets input from each neuron in the pooling layer.

Trained using backprop, which can be easily adapted to convolutional networks.

Convolutional network



Simple convolutional network vs MNIST

two convolutional-pooling layers, one 20, second 40 feature maps, two dense (MLP) layers (1000-1000), outputs (10)

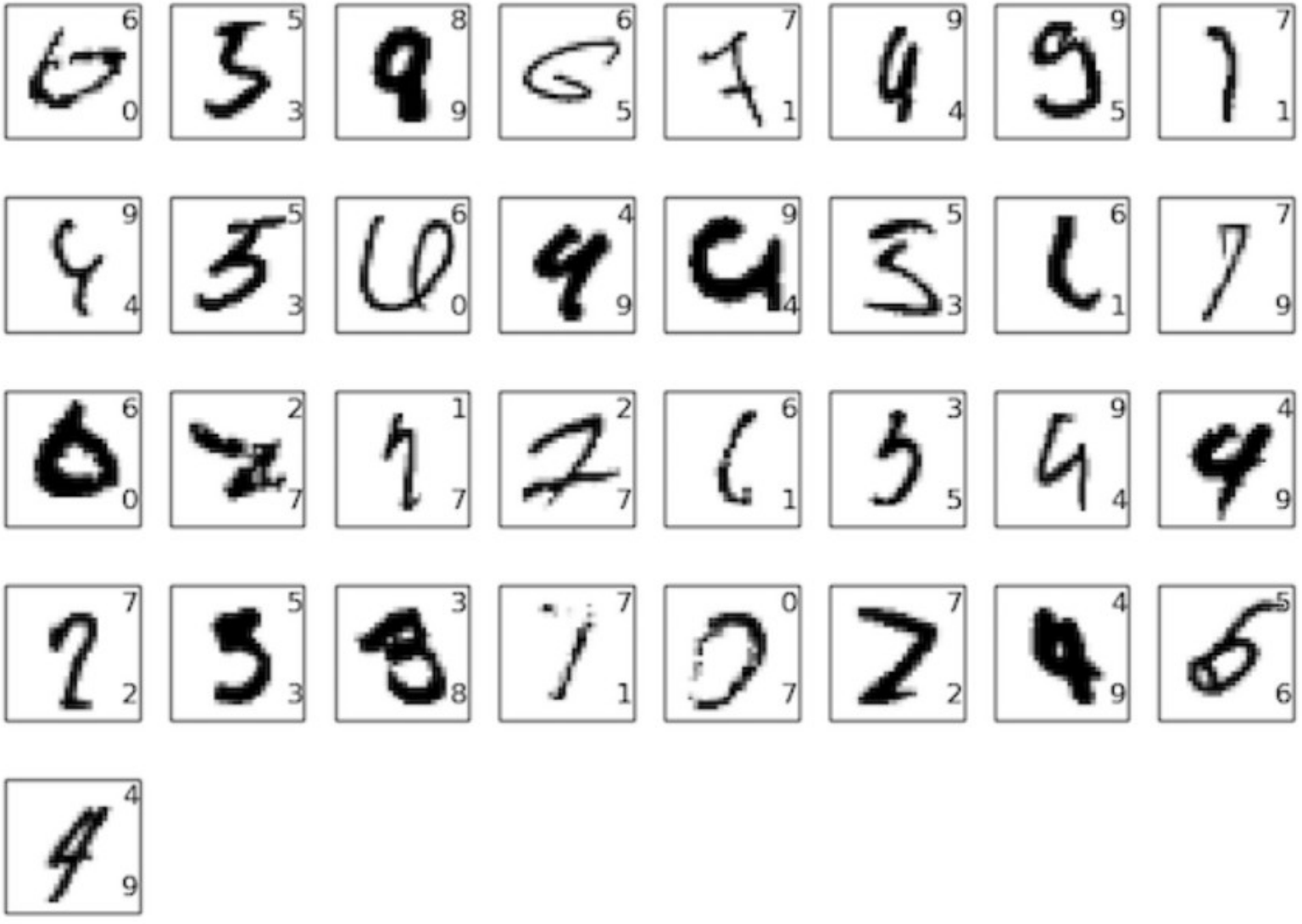
- Activation functions of the feature maps and dense layers: ReLU
- max-pooling
- output layer: soft-max
- Error function: negative log-likelihood (= cross-entropy)
- Training: SGD, mini-batch size 10
- learning rate 0.03
- L2 regularization with "weight" $\lambda = 0.1$ + dropout with prob. $1/2$
- training for 40 epochs (i.e. every training example is considered 40 times)
- Expanded dataset: displacement by one pixel to an arbitrary direction.
- Committee voting of 5 networks.

Simple convolutional network in Theano

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(
            n_in=40*4*4, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        FullyConnectedLayer(
            n_in=1000, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        SoftmaxLayer(n_in=1000, n_out=10, p_dropout=0.5)],
        mini_batch_size)
>>> net.SGD(expanded_training_data, 40, mini_batch_size, 0.03,
            validation_data, test_data)
```

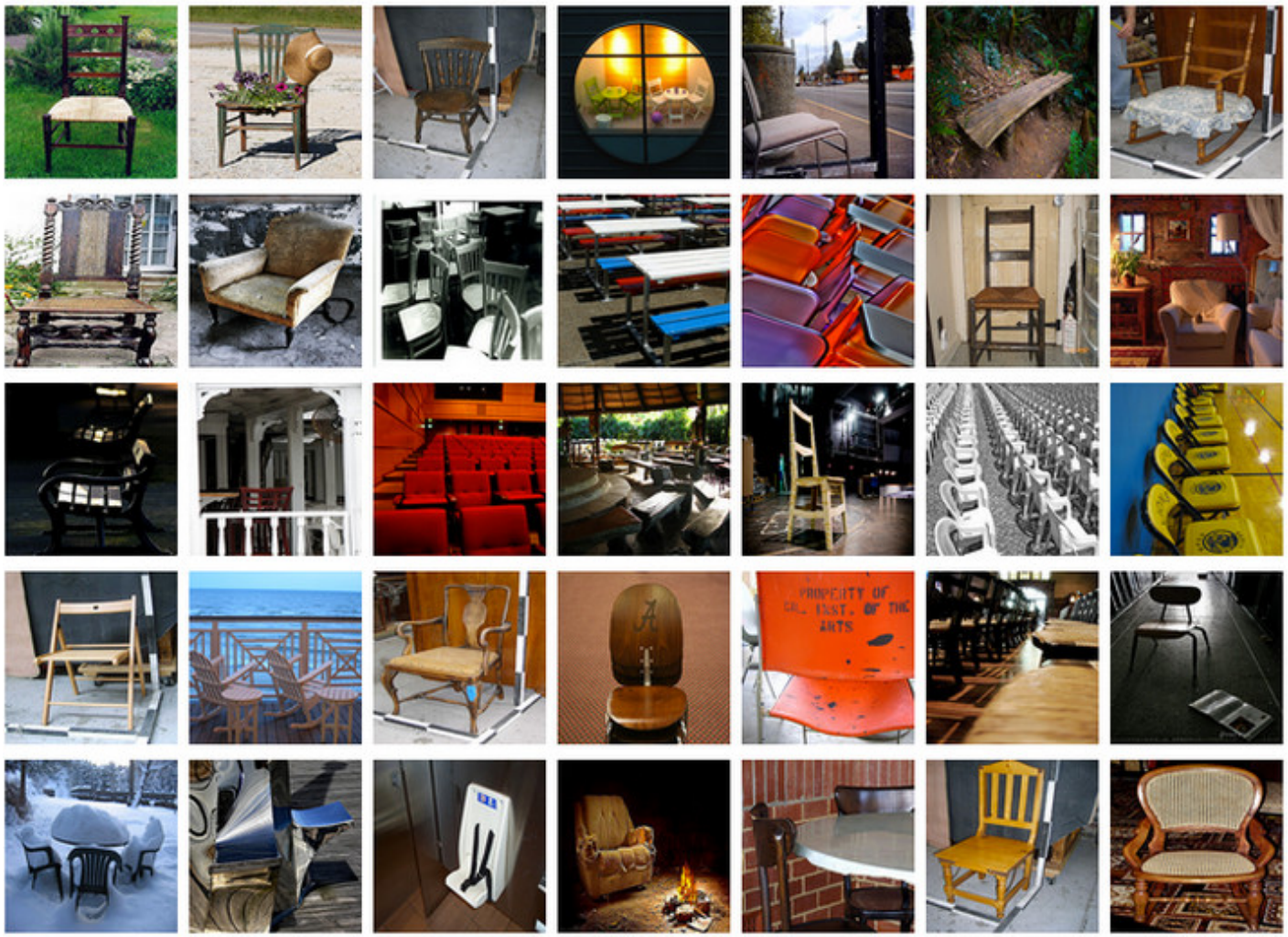
MNIST

Out of 10 000 images in the test set, only these 33 have been incorrectly classified:



More complex convolutional networks

Convolutional networks have been used for classification of images from the ImageNet database (16 million color images, 20 thousand classes)



ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)

Competition in classification over a subset of images from ImageNet.

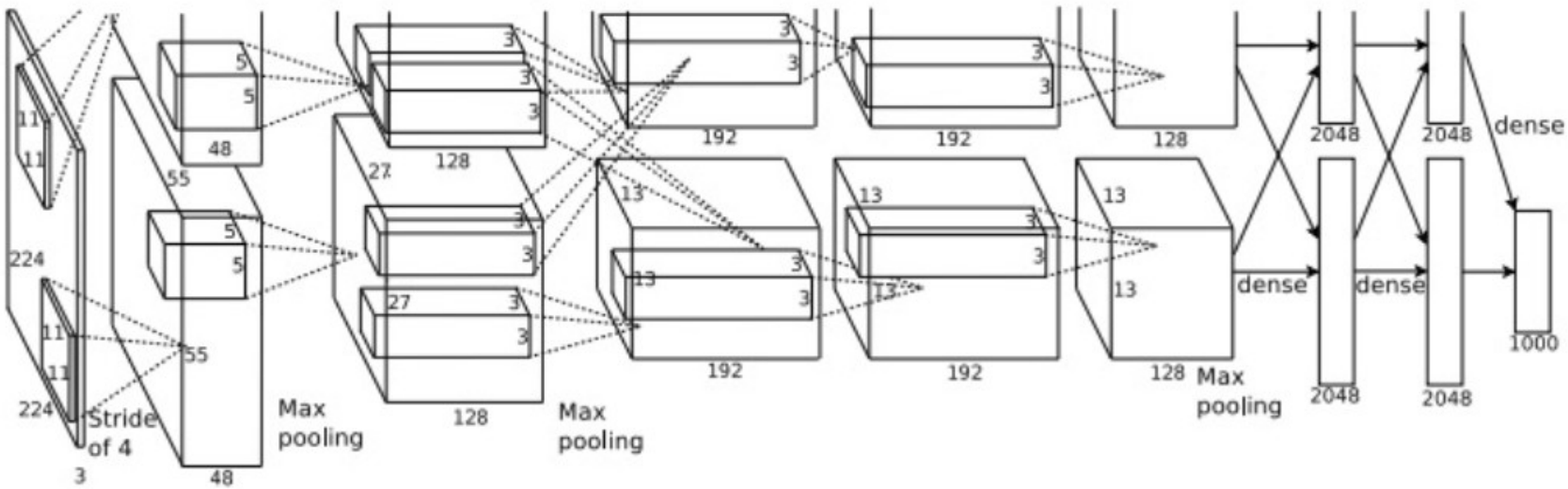
Started in 2010, assisted in breakthrough in image recognition.

Training set 1.2 million images, 1000 classes. Validation set: 50 000, test set: 150 000.

Many images contain more than one object $\Rightarrow$ model is allowed to choose five classes, the correct label must be among the five. (top-5 criterion).

AlexNet

ImageNet classification with deep convolutional neural networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton (2012).



Trained on two GPUs (NVIDIA GeForce GTX 580)

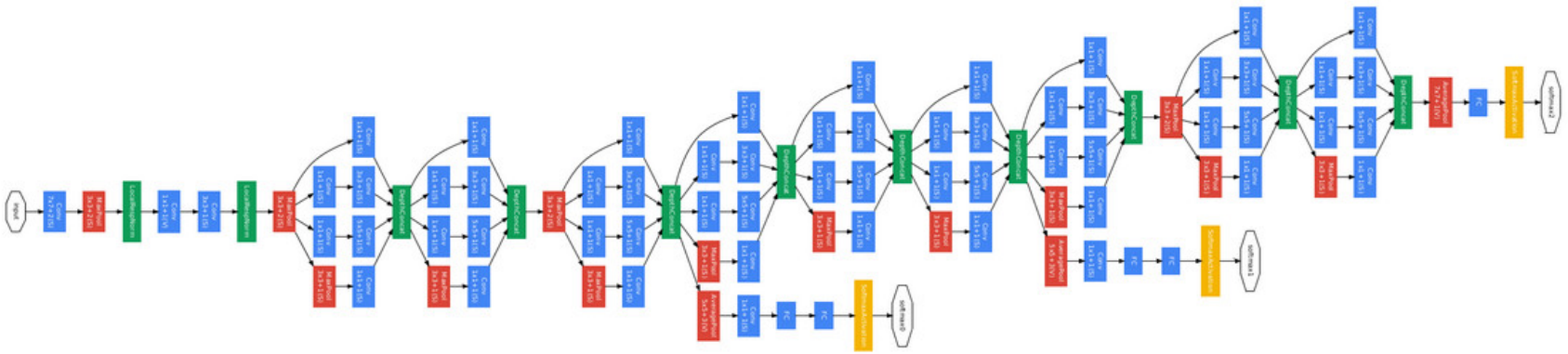
Výsledky:

- accuracy 84.7% in top-5 (second best algorithm at the time 73.8%)
- 63.3% "perfect" (top-1) classification

ILSVRC 2014

The same set as in 2012, top-5 criterion.

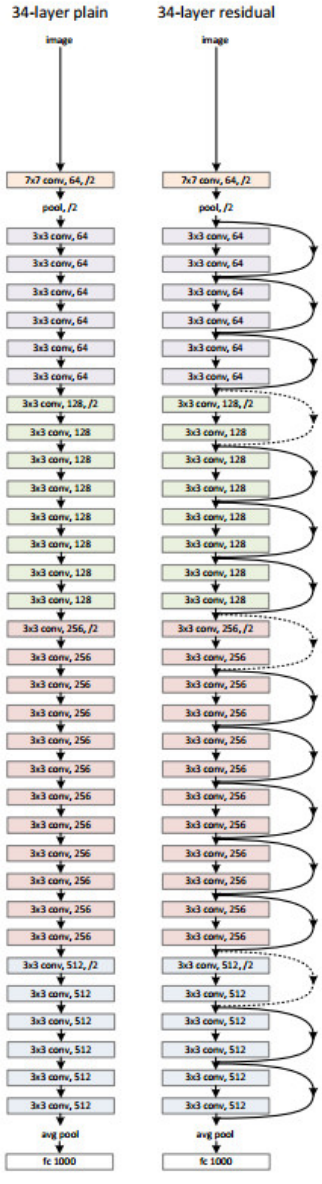
GoogLeNet: deep convolutional network, 22 layers



Results:

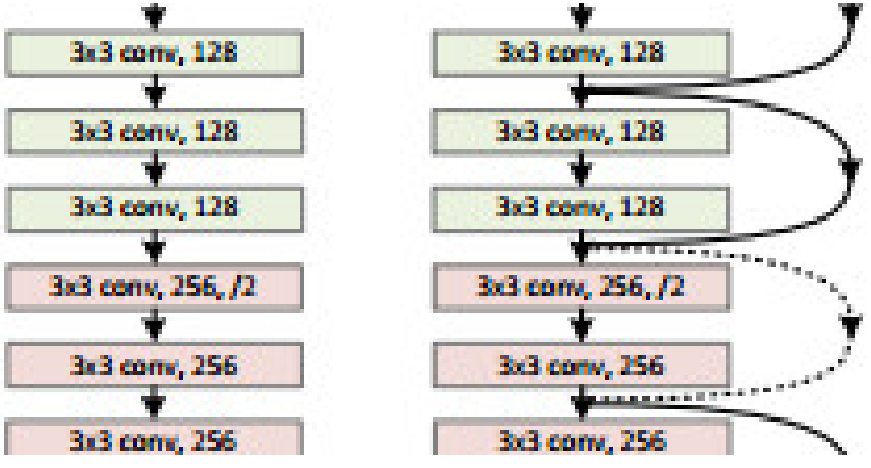
- Accuracy 93.33% top-5

ILSVRC 2015



- Deep convolutional network
- Various numbers of layers, the winner has 152 layers
- Skip connections implementing residual learning

- Error **3.57%** in top-5.



Superhuman convolutional nets?!

Andrej Karpathy: ...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought we could recruit paid undergrads. Then I organized a labeling party of intense labeling effort only among the (expert labelers) in our lab. Then I developed a modified interface that used GoogLeNet predictions to prune the number of categories from 1000 to only about 100. It was still too hard - people kept missing categories and getting up to ranges of 13-15% error rates. In the end I realized that to get anywhere competitively close to GoogLeNet, it was most efficient if I sat down and went through the painfully long training process and the subsequent careful annotation process myself... The labeling happened at a rate of about 1 per minute, but this decreased over time... Some images are easily recognized, while some images (such as those of fine-grained breeds of dogs, birds, or monkeys) can require multiple minutes of concentrated effort. I became very good at identifying breeds of dogs... Based on the sample of images I worked on, the GoogLeNet classification error turned out to be 6.8%... My own error in the end turned out to be 5.1%, approximately 1.7% better.