SoSe 2018: Deep Neural Networks

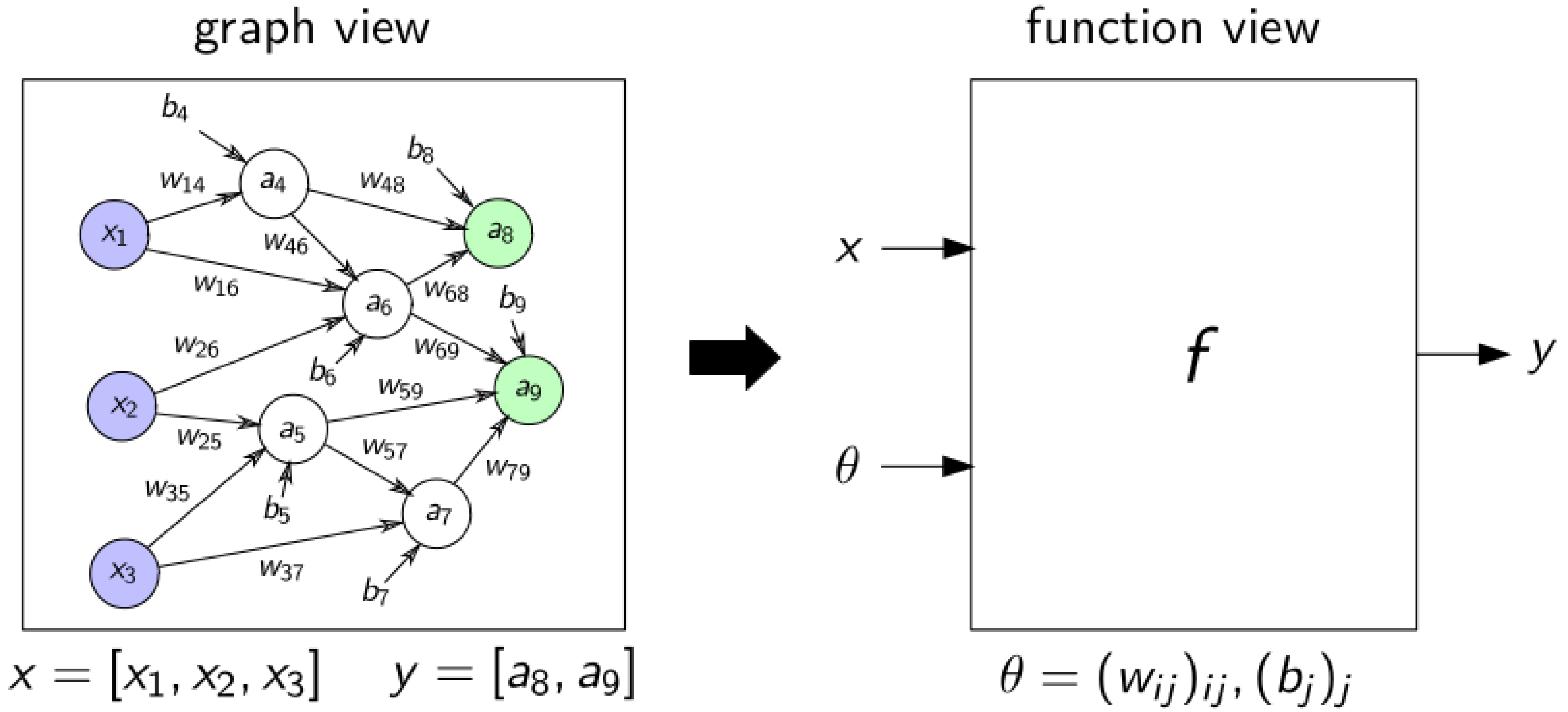
Lecture 3: Optimization

Machine Learning Group

Technische Universität Berlin

Recap: How to Learn in a Neural Network

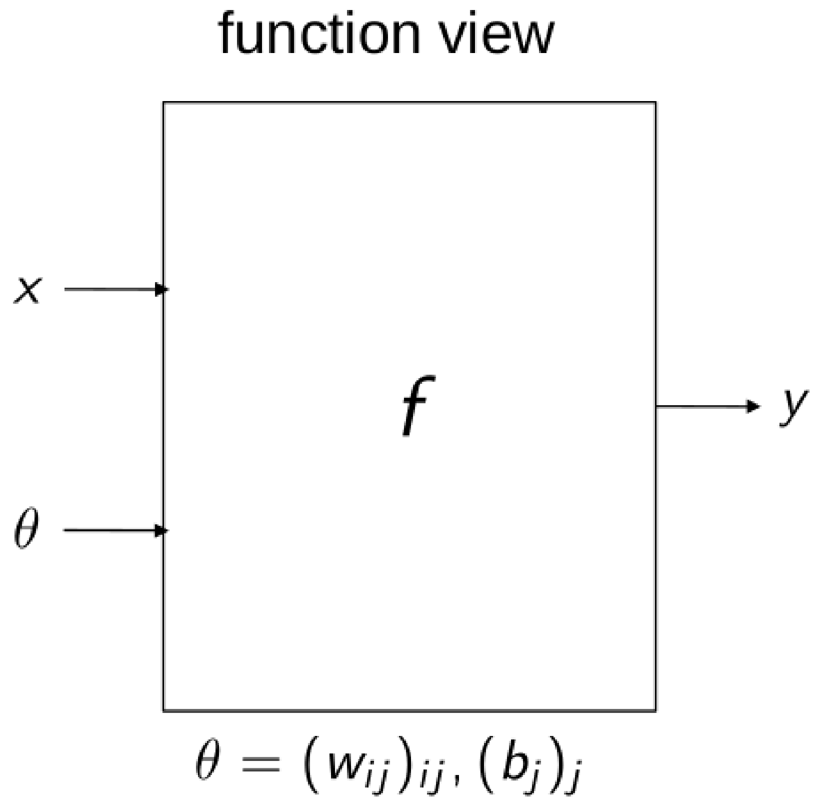
Observation: A neural network is a function of both its inputs and parameters.



$x = [x_1, x_2, x_3] \quad y = [a_8, a_9]$

$\theta = (w_{ij})_{ij}, (b_j)_j$

Recap: How to Learn in a Neural Network

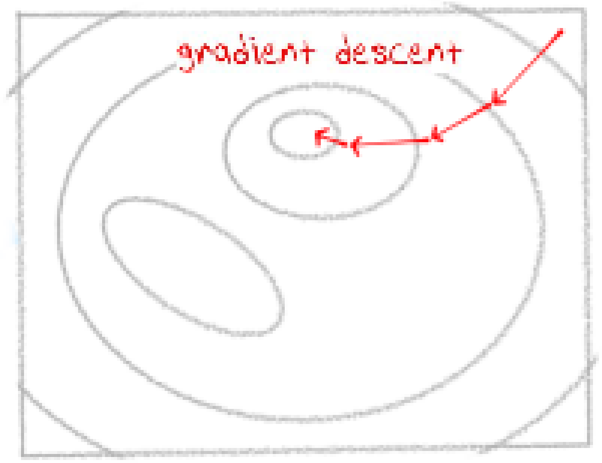


Define an error function

$$E(\theta) = \textstyle\sum_n (f(\boldsymbol{x}_n;\theta) - t_n)^2$$

and minimize by gradient descent

$$\theta \leftarrow \theta - \gamma \cdot \nabla_\theta E(\theta)$$



gradient is computed with error backpropagation

Outline

Characterizing the error function

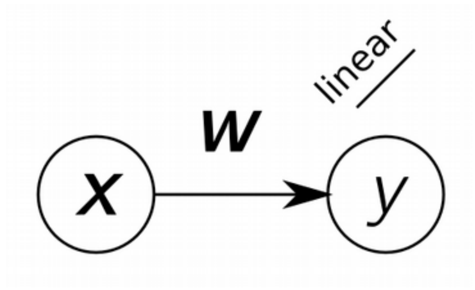
- Local minima and plateaus
- Local curvature and condition number

Improving optimization

- Initialization
- Choice of nonlinearities
- Momentum

Fast implementations

Characterizing the Error Function: 1 Layer



$$y = \boldsymbol{x}^\top \boldsymbol{w}$$

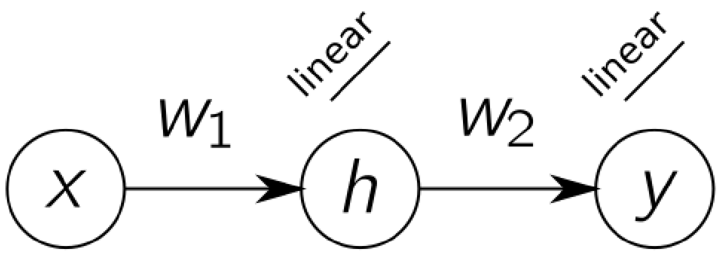
$$E = \tfrac{1}{N} \textstyle\sum_n \|y_n - t_n\|^2 + \lambda \|\boldsymbol{w}\|^2$$

$$= \tfrac{1}{N} \textstyle\sum_n (\boldsymbol{x}_n^\top \boldsymbol{w} - t_n)^\top (\boldsymbol{x}_n^\top \boldsymbol{w} - t_n) + \lambda \|\boldsymbol{w}\|^2$$

$$= \boldsymbol{w}^\top \underbrace{[\tfrac{1}{N} \textstyle\sum_n \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda I]}_{\text{positive semi-definite}} \boldsymbol{w} + \text{linear} + \text{constant}$$

$\Rightarrow$ **convex**

Characterizing the Error Function: 2 Layers

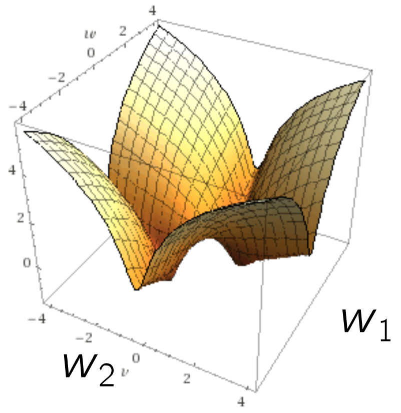


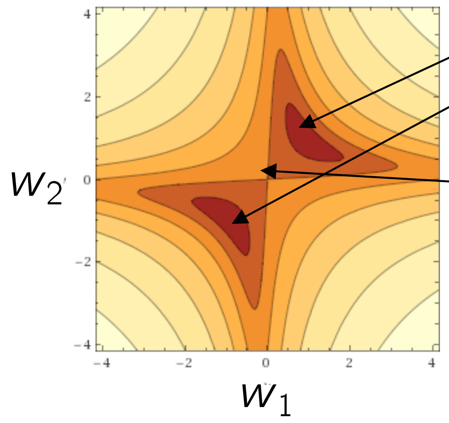
$$\begin{tikzcd} x \arrow[r, "w_1"] & h \arrow[r, "w_2"] & y \end{tikzcd}$$

$$y = w_2 \cdot (w_1 \cdot x)$$

$$E = \frac{1}{N} \sum_n \| w_2 \cdot w_1 \cdot x_n - t_n \|^2 + \lambda(\|w_1\|^2 + \|w_2\|^2)$$

Example:
N=1, x_1=1, t_1=1, λ=0.1





two local minima

saddle point

→ **non-convex**

Characterizing the Error Function: 2 Layers

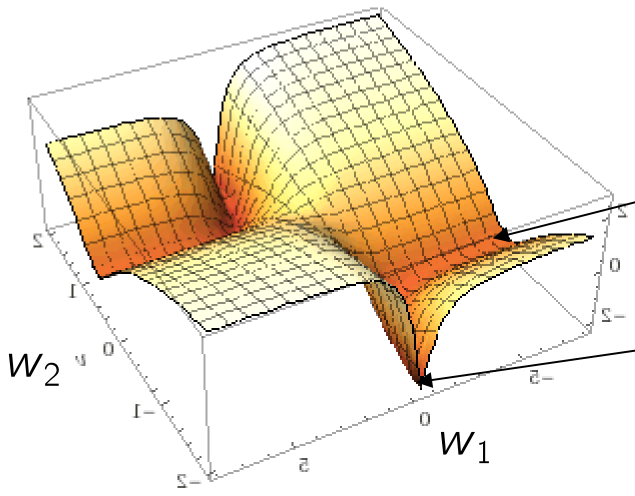
nonlinear

$$\begin{tikzcd} x \arrow[r, "w_1"] & h \arrow[r, "w_2"] & y \end{tikzcd}$$

$$y = w_2 \cdot \tanh(w_1 \cdot x)$$

$$E = \frac{1}{N} \sum_n \| w_2 \cdot \tanh(w_1 \cdot x_n) - t_n \|^2 + \lambda(\|w_1\|^2 + \|w_2\|^2)$$

Example:
N=2, x_1=0.5, t_1=0.5, x_2=1, t_2=1, λ=0.0



Initializing the Neural Network

Even for the simplest two-layer neural network, the error function is already non-convex. Therefore, initialization of the neural network is important.

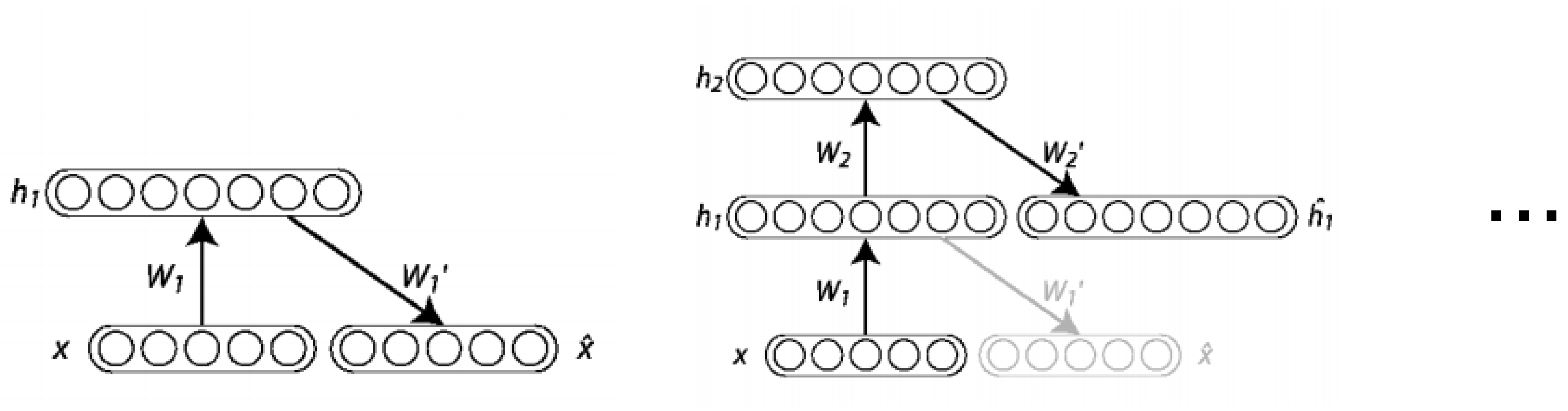
Recommendation for neural networks with *tanh* nonlinearities: Scale parameters such that neuron outputs have variance ≈1 initially (LeCun'98/12 "Efficient Backprop")

$$w \sim \mathcal{N}(0, \sigma^2) \qquad\qquad \sigma^2 = \frac{1}{\#\ \text{input neurons}}$$

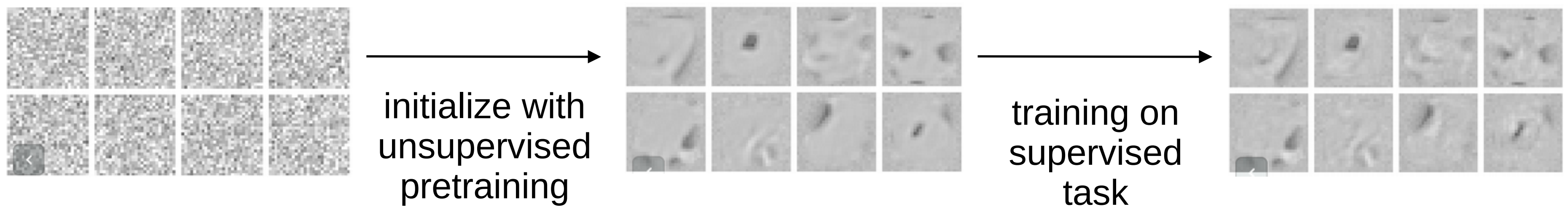
This initialization avoids saddle nodes and plateaus and also works well for ReLU nonlinearities.

Initializing the Neural Network

Technique to reach good optima of the error function: layer-wise unsupervised pre-training (Hinton'06, Bengio'06, Vincent'08).



Example: Learning first-layer parameters on MNIST handwritten digits:



initialize with unsupervised pretraining
training on supervised task

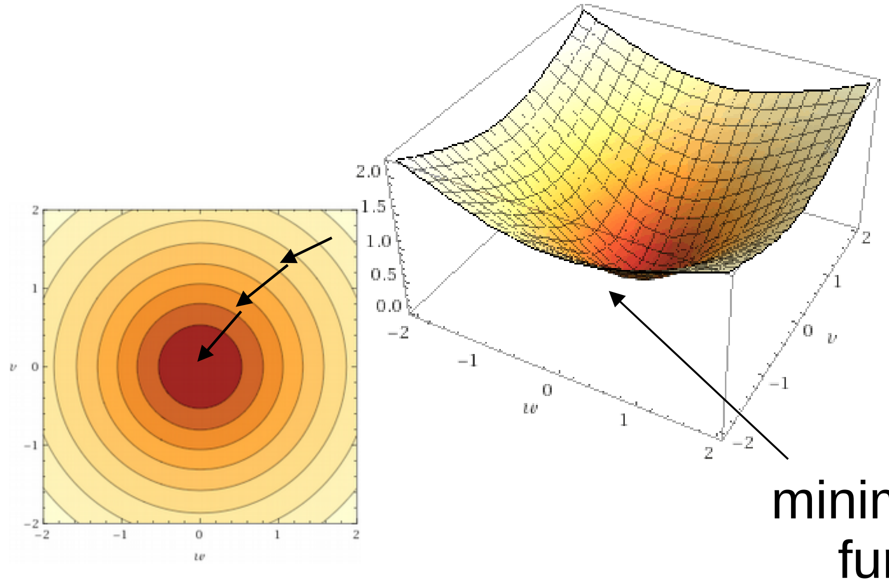
Is Convexity Sufficient?

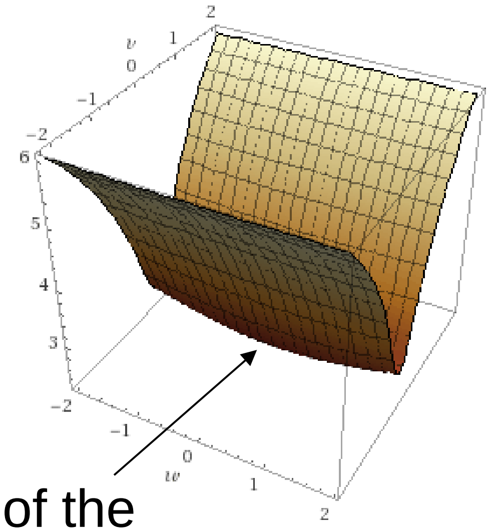
Answer: No. We must also verify that the function is well-conditioned.

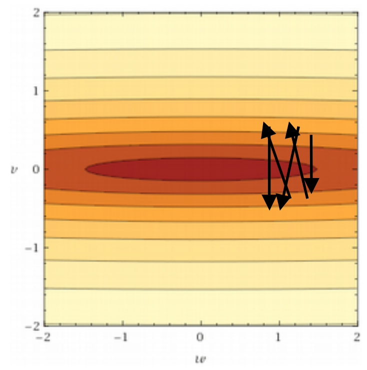
Examples:

well-conditioned error function

poorly conditioned error function







Well-conditioned functions are easier to optimize.

Quantifying “well-conditioned”

Error function of a neural network can be approximated locally by a quadratic function.

$$E(\theta) = E(\theta_0) + \underbrace{\frac{\partial E}{\partial \theta}\Big|_{\theta_0}}_{g} \cdot (\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \underbrace{\frac{\partial^2 E}{\partial \theta \partial \theta^\top}\Big|_{\theta_0}}_{H} \cdot (\theta - \theta_0)$$

g **Gradient** (contains slope information)

H **Hessian** (contains curvature information)

Idea: Look at the disbalance of curvature between different directions in the input space, by computing a ratio of eigenvalues.

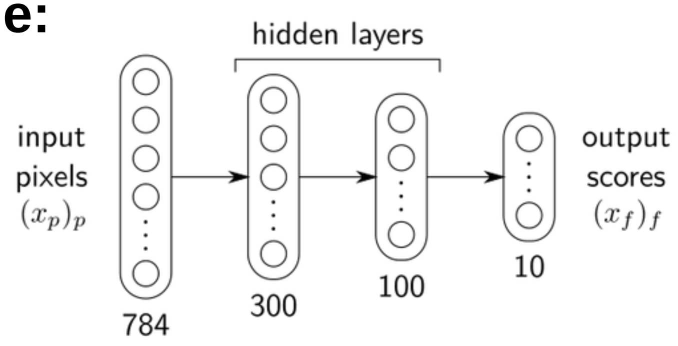
$$\lambda_1, \lambda_2, \ldots, \lambda_d = \mathrm{eigval}(H)$$

$$\boxed{\text{condition number} = \frac{\lambda_1}{\lambda_d}}$$

Computing the Hessian in Practice?

$$E(\theta) = E(\theta_0) + \underbrace{\frac{\partial E}{\partial \theta}\Big|_{\theta_0}}_{\substack{\textbf{gradient} \\ \text{(can be computed} \\ \text{with backprop)}}} \cdot (\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \underbrace{\frac{\partial^2 E}{\partial \theta \partial \theta^\top}\Big|_{\theta_0}}_{\substack{\textbf{Hessian} \\ \text{(hard to compute and very large for fully} \\ \text{connected networks)}}} \cdot (\theta - \theta_0)$$

Example:

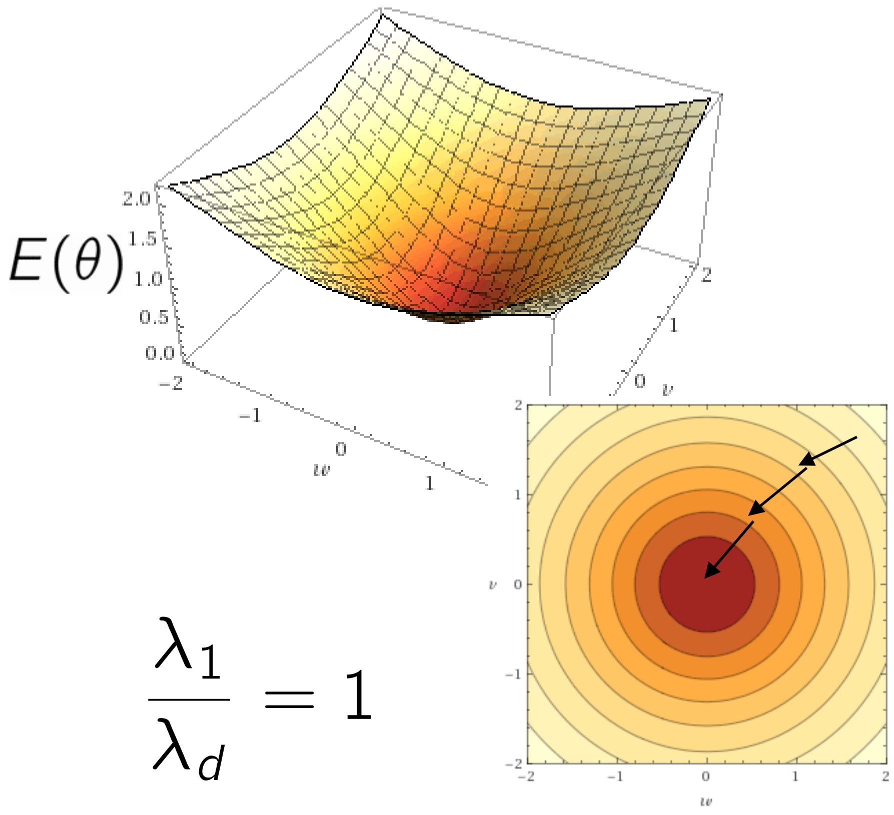


$$\theta = 784 \cdot 300 + 300 \cdot 100 + 100 \cdot 10$$
$$= 266200 \text{ parameters}$$

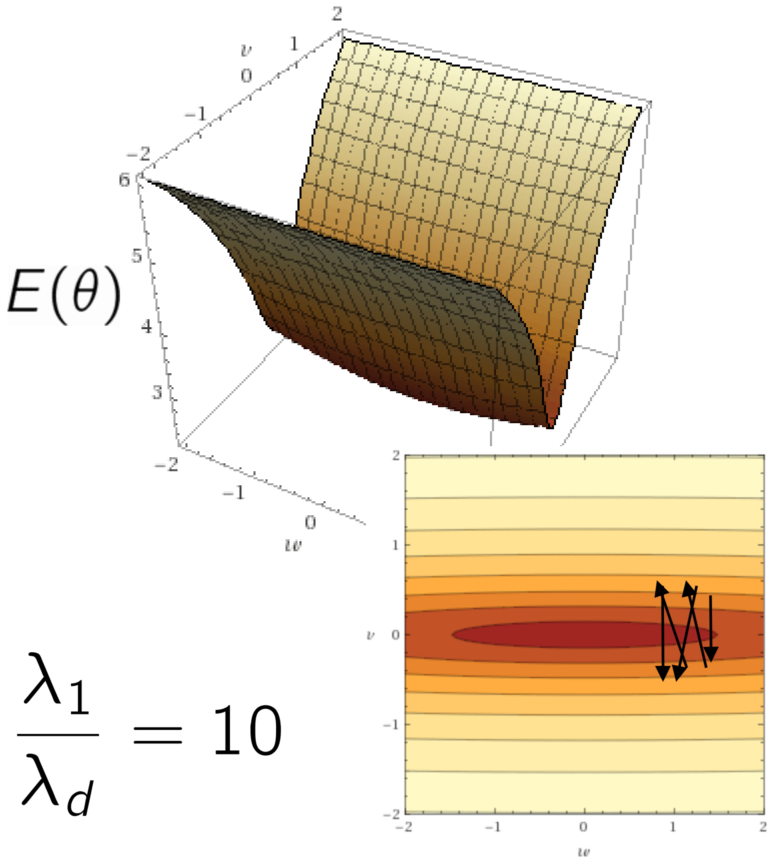
$$H = 266200 \cdot 266200 \text{ entries}$$
$$= 283 \text{ GB}$$

For most practical tasks, we don't need to evaluate the Hessian and the condition number. We only need to apply a set of recommendations and tricks that keep the condition number low.

Quantifying “well-conditioned”



$$\frac{\lambda_1}{\lambda_d} = 1$$

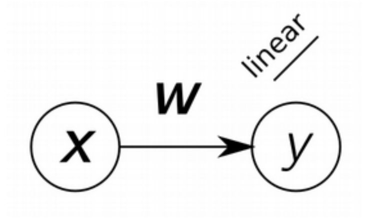


$$\frac{\lambda_1}{\lambda_d} = 10$$

The lower the condition number, the better.

Improving Conditioning of the Error Function

Example: The linear model



$$y = \boldsymbol{x}^\top \boldsymbol{w}$$

$$E = \boldsymbol{w}^\top \underbrace{[\tfrac{1}{N} \textstyle\sum_n \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda I]}_{H} \boldsymbol{w} + \text{linear} + \text{constant}$$

(Hessian of the error function)

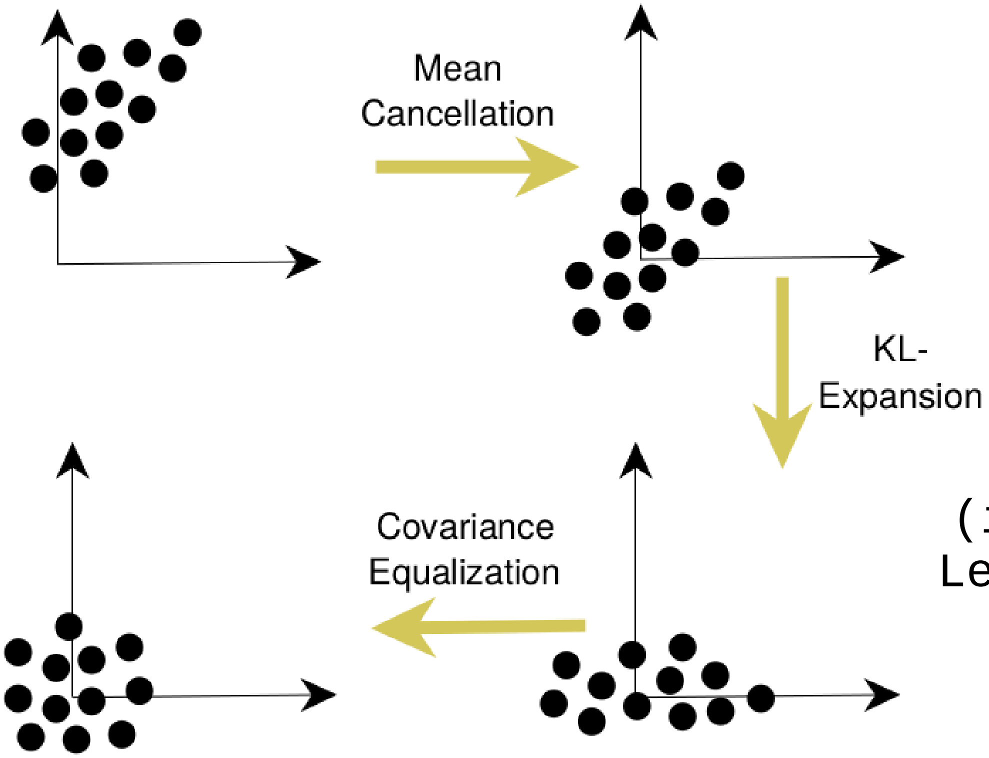
condition number $\dfrac{\lambda_1}{\lambda_d}$ influenced by the *mean* and *covariance* of the input data

$\Rightarrow$ **Trick:** Normalize the data

Data Normalization to Improve Conditioning

Data preprocessing *before* training:

$(x_1, \ldots, x_N)$

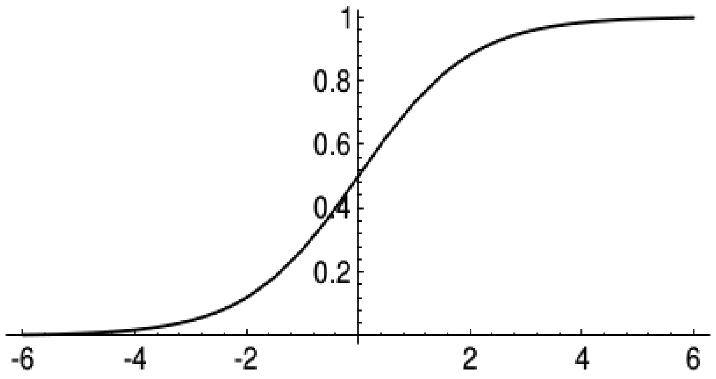


(image from LeCun'98/12)

Improving Conditioning of Higher-Layers

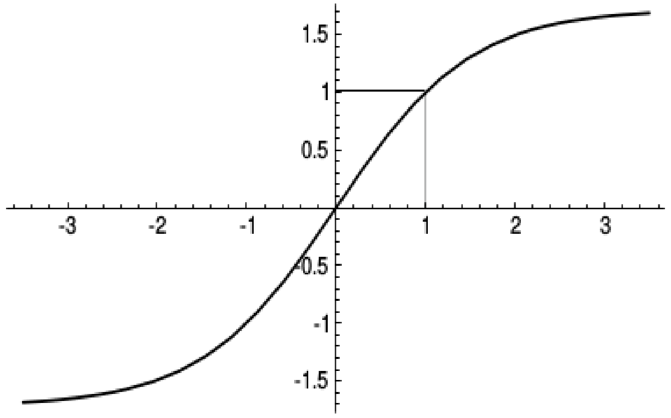
To improve conditioning, not only the input data should be normalized, but also the representations built from this data at each layer. This can be done by carefully choosing the activation function.

logistic sigmoid



(a)

hyperbolic tangent



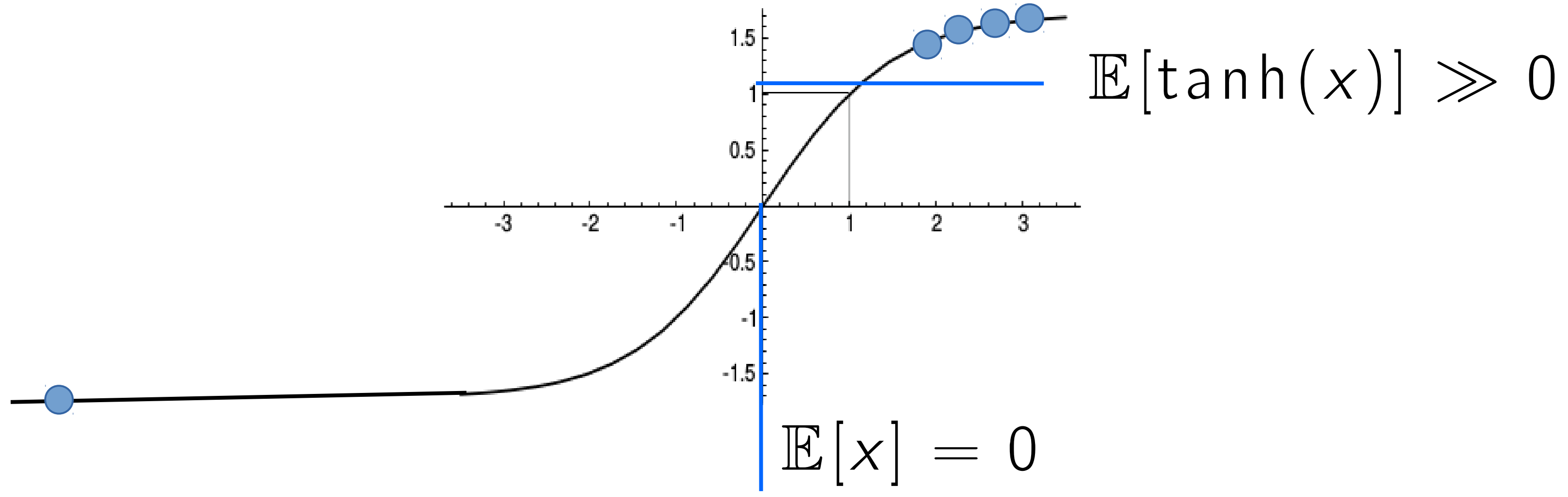
(b)

activations are not centered
→ **high condition number**

activations are approximately centered at 0
→ **low condition number**

Limitation of Tanh

The tanh nonlinearity works well initially, but after some training steps, it might no longer work as expected as the input distribution will drift to negative or positive values.



Remark: If input of tanh is centered but skewed, output of tanh will not be centered. This happens a lot in practice, e.g. when the problem representation needs to be sparse.

Limitation of Tanh

Countermeasures:

- **Use Batch Normalization** [Ioffe'15]
  Create a layer that explicitly centers and rescales the output of hidden units. (Requires batch or minibatch training.)
- Use more rigid nonlinearities such as **rectified linear unit** [Glorot'10] or self-normalizing ones like **SeLU** [Klambauer'17]
- Use **momentum** [Bishop'95]
- Or combination of them

Momentum

Idea: Choose the update direction as a weighted average of previous updates.

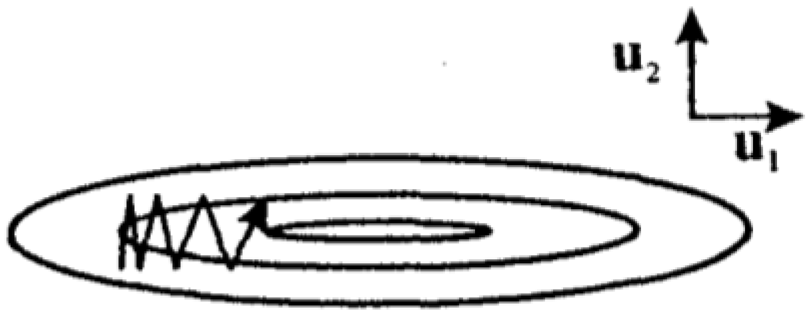


Image from Bishop'95

Accelerates convergence along direction of low curvature.
Momentum can help to overcome a poorly conditioned neural network.

Momentum

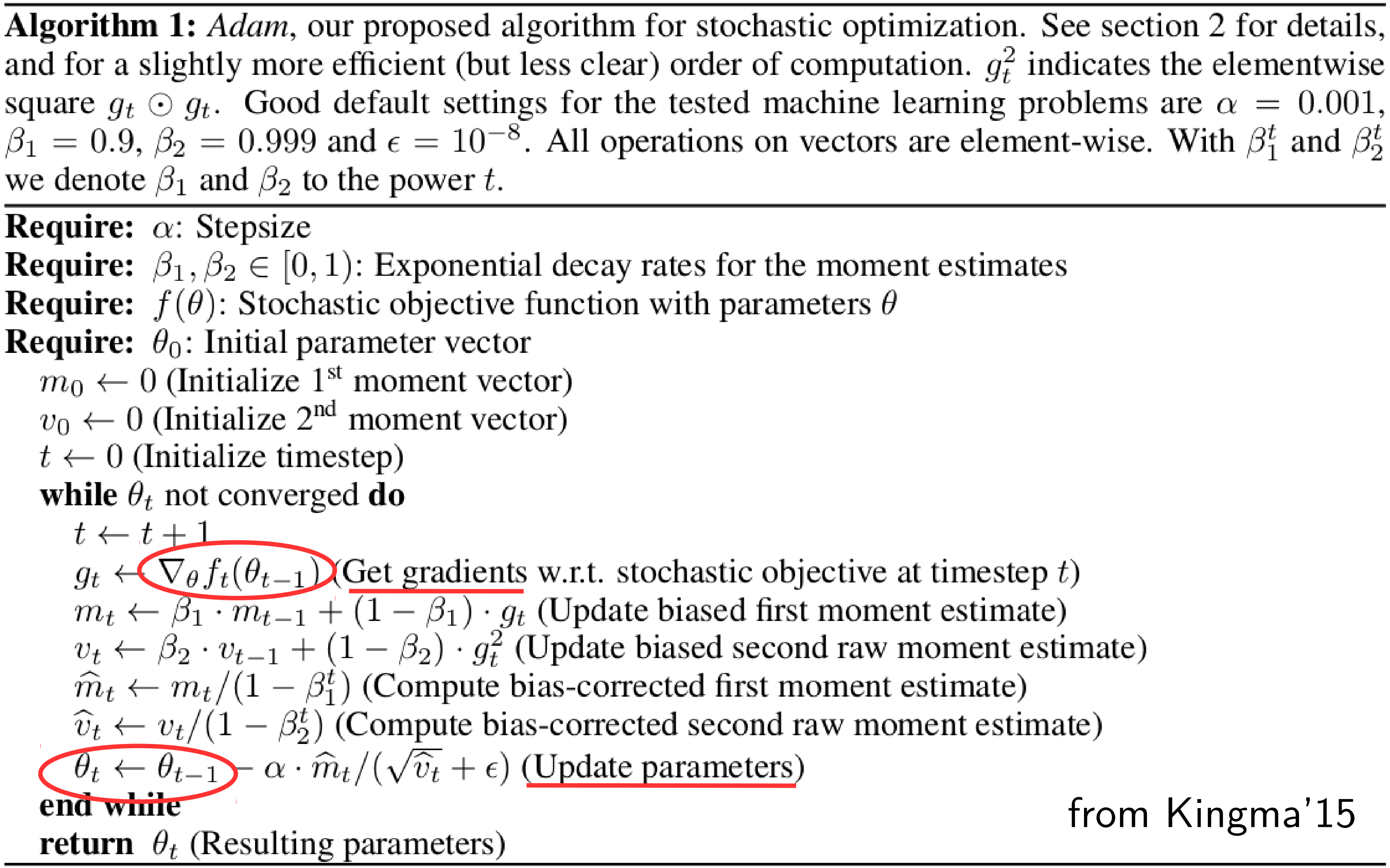
Update the direction of descent as:

$$\underbrace{\Delta^{(t)}}_{\text{new update step}} = \underbrace{\mu}_{\text{momentum}} \cdot \underbrace{\Delta^{(t-1)}}_{\text{previous update step}} + \underbrace{\gamma}_{\text{learning rate}} \cdot \underbrace{\nabla E(\theta^{(t)})}_{\text{error gradient}}$$

and update the neural network parameters following this direction:

$$\theta^{(t)} = \theta^{(t-1)} + \Delta^{(t)}$$

The Adam Algorithm



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

from Kingma'15

Gradient Descent vs. SGD

Objective to Minimize:

$$E(\theta) = \frac{1}{N}\sum_{n=1}^{N} E^{(n)}(\theta)$$

Batch GD:

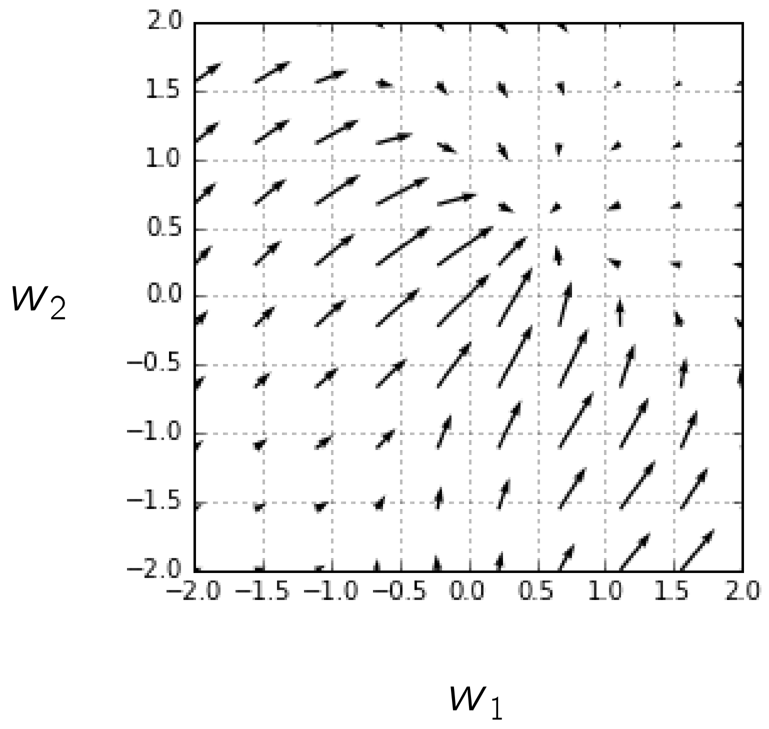
```
while True:
    $\underbrace{\theta \leftarrow \theta - \gamma \frac{\partial}{\partial \theta} \frac{1}{N} \sum_{n=1}^{N} E^{(n)}}_{O(N)}$
```

Stochastic GD (SGD):

```
while True:
    $n \leftarrow \mathrm{random}(1, N)$
    $\underbrace{\theta \leftarrow \theta - \gamma \frac{\partial E^{(n)}}{\partial \theta}}_{O(1)}$
```

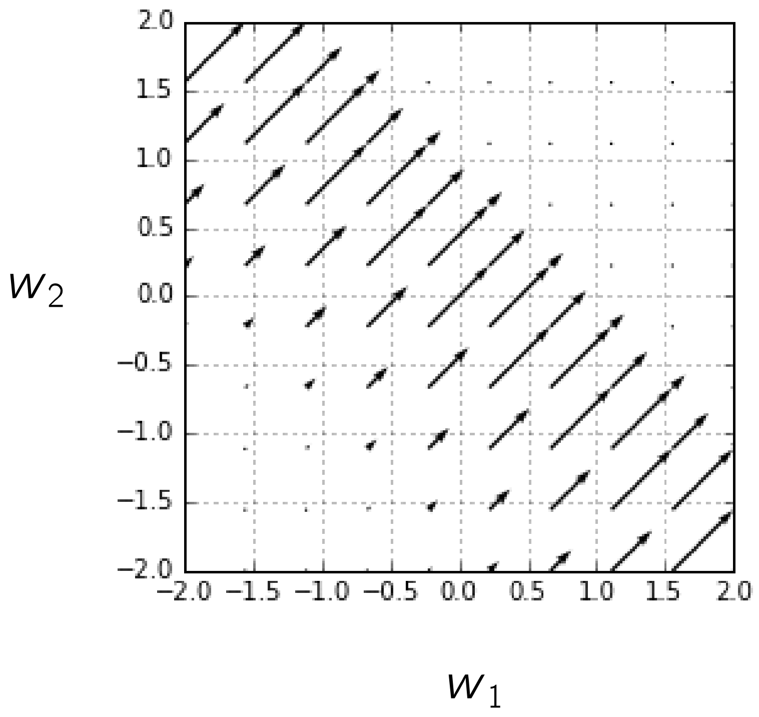
GD vs. SGD

Gradient Descent



Gradient in parameter space is more precise, but more costly to evaluate.

Stochastic Gradient Descent



Gradient in parameter space has some randomness (never converges).

GD vs. SGD

<u>Batch GD:</u>

```
while True:
    $\theta \leftarrow \theta - \gamma \frac{\partial}{\partial \theta} \frac{1}{N} \sum_{n=1}^{N} E^{(n)}$
```

<u>Stochastic GD (SGD):</u>

```
while True:
    $n \leftarrow \text{random}(1, N)$
    $\theta \leftarrow \theta - \gamma \frac{\partial E^{(n)}}{\partial \theta}$
```

use a decreasing schedule

$$\gamma^{(1)}, \gamma^{(2)}, \dots, \gamma^{(T)}$$

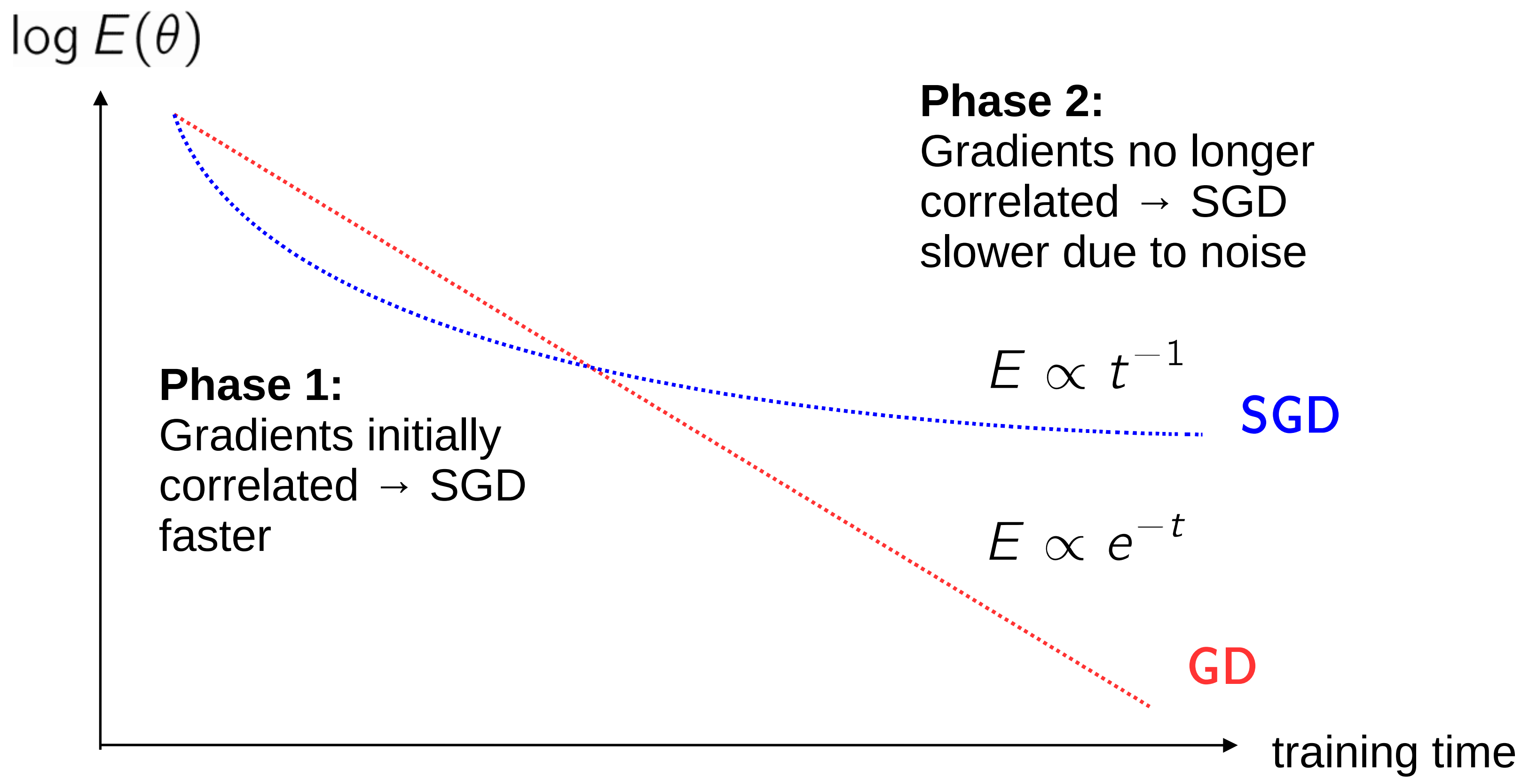
Conditions for SGD convergence:

$$\sum_{t=1}^{\infty} \gamma^{(t)} = \infty \qquad \lim_{t \to \infty} \gamma^{(t)} = 0$$

Which SGD learning schedule to choose?

	$\gamma^{(t)} = 1$	$\gamma^{(t)} = t^{-1}$	$\gamma^{(t)} = e^{-t}$
$\sum_{t=1}^{\infty} \gamma^{(t)} = \infty$	✓	✓	✗
$\lim_{t \to \infty} \gamma^{(t)} = 0$	✗	✓	✓

GD vs. SGD Convergence

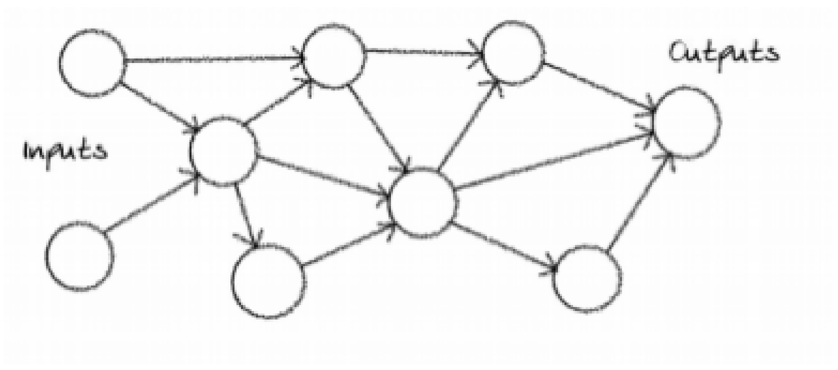


Insight: phase 2 is *not* relevant, because the model already starts overfitting before reaching it. → SGD is the method of choice for most practical purposes.

Neural Network Training Time

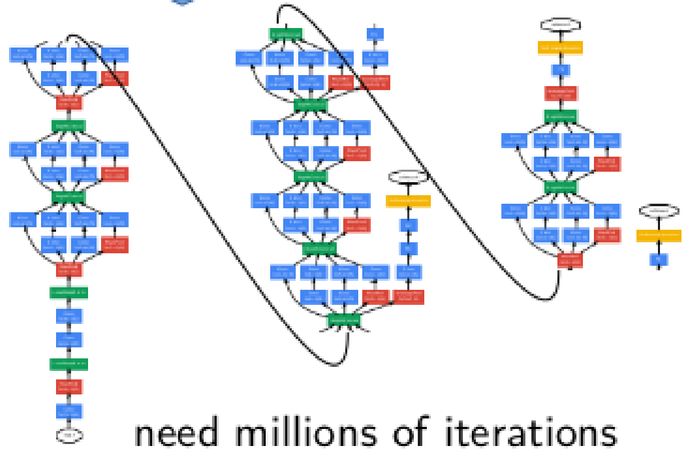
SGD $\Rightarrow$ $\boxed{O(\#\text{iterations} \times \#\text{edges})}$ $\Leftarrow$ **Backprop**

This network: 13 connections



can probably be trained with a few hundreds iterations.

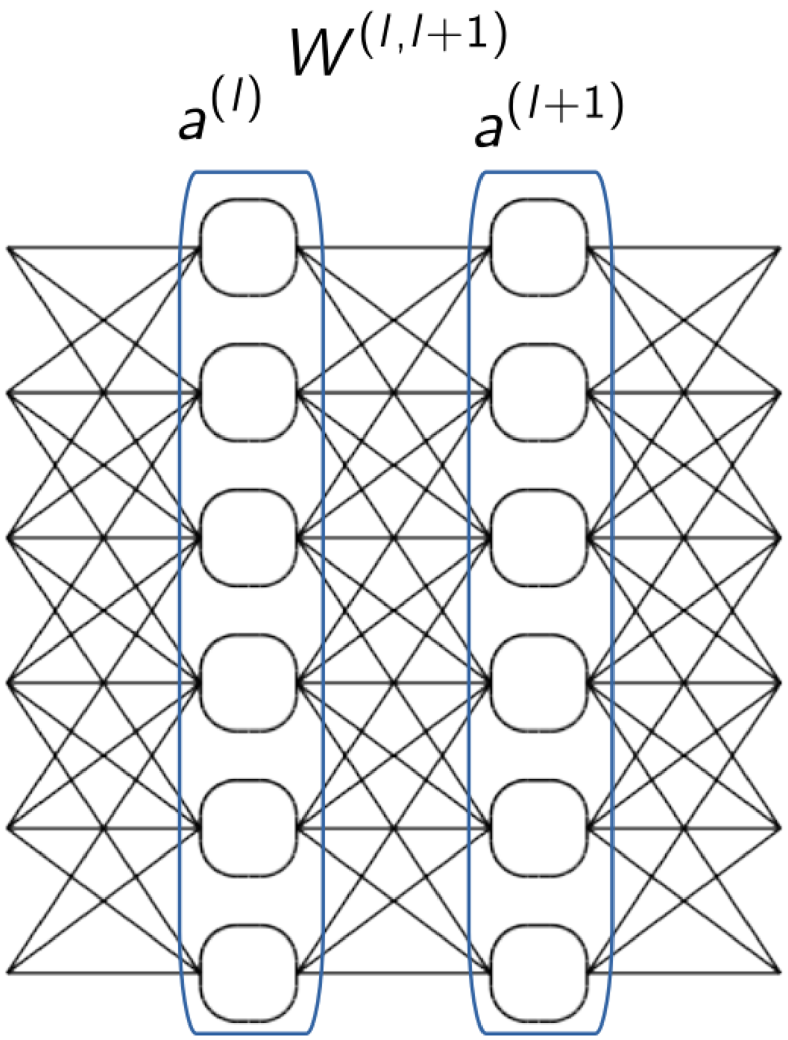
Googlenet: $> 10^9$ connections



need millions of iterations to be trained.

Part 2: Scaling Deep Nets to Big Data

Step 1: Systematize Computations



Per-neuron forward computations

$$\forall_j : a_j = g\left(\textstyle\sum_i a_i w_{ij} + b_j\right)$$

Whole-layer computation

$$a^{(l+1)} = g\big(\underbrace{W^{(l,l+1)} \cdot a^{(l)} + b^{(l+1)}}_{\text{matrix-vector products (e.g. numpy.dot)}}\big)$$

$\downarrow$

element-wise application of nonlinearity

Step 2: Mini-Batches

Idea: Take advantage of fast matrix-matrix multiplications by feeding several examples at a time to the neural network.

Example for layer with weight matrix: $W^{(l,l+1)} \in \mathbb{R}^{h \times h}$

Pure SGD (propagate 1 data point)

$$a^{(l)}, a^{(l+1)} \in \mathbb{R}^{h} \qquad a^{(l+1)} = g\left(W^{(l,l+1)} \cdot a^{(l)} + b^{(l+1)}\right)$$

$\boxed{O(h^2)}$ computations

Minibatch SGD (propagate M data points)

$$A^{(l)}, A^{(l+1)} \in \mathbb{R}^{h \times M} \qquad A^{(l+1)} = g(W^{(l,l+1)} \cdot A^{(l)} + b^{l+1})$$

$M \approx h \quad \Rightarrow \boxed{O(h^{2.4})} < O(Mh^2)$ computations

Step 2: How to Choose Mini-Batch Size?

$$\boxed{M \approx h}$$

Advantages:
(1) Largest speed up in terms of matrix multiplications.

Disadvantages:
(1) Multiplication might not fit in memory
(2) If gradients are correlated, same direction as pure SGD.
(3) Layers have different size

In practice, it is better to use a constant minibatch size, for example, $M = 25$.

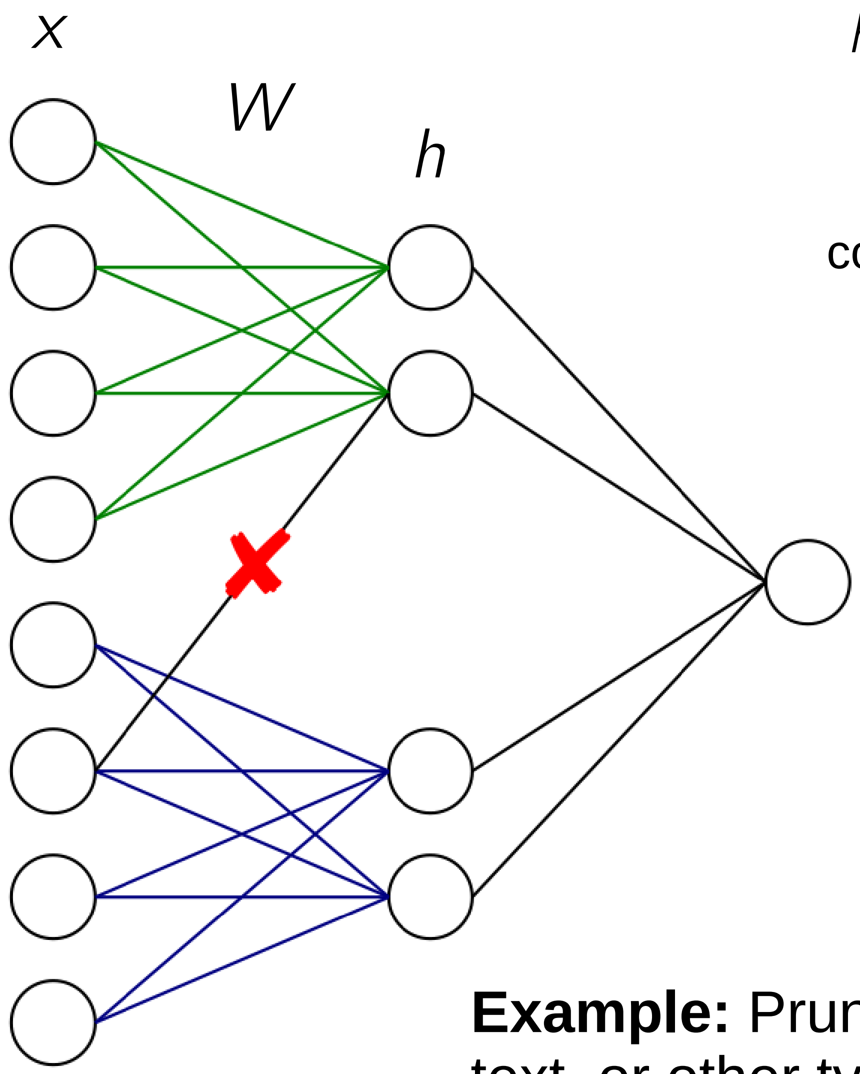
If overfitting doesn't occur (e.g. because the model is smaller than the data). We can switch to a larger minibatch size (e.g. M=100 or M=1000) in the late stage of optimization to speed up convergence. [e.g. Salakhutdinov'09, Kindermans'18], or increase momentum.

Step 2: How to Choose Mini-Batch Size?

Summary:

	smallest minibatch (M=1) = pure SGD	typical minibatch (M=25)	largest minibatch (M=N) = GD
avoid redundant gradient computations	✓	✓	✗
speedup from matrix-matrix multiplications	✗	✓	✓

Step 3: Prune Irrelevant Computations

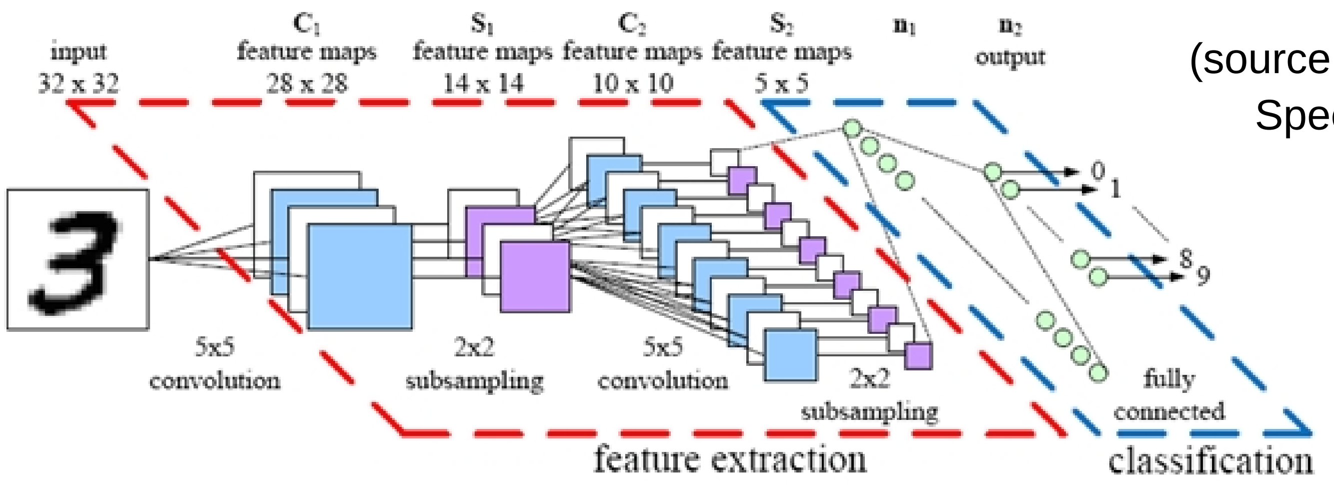


$$h = \underbrace{W^\top x}_{\substack{8 \text{ x } 4 = 32 \\ \text{computations}}} = \underbrace{W_A^\top x_A + W_B^\top x_B}_{\substack{2 \text{ x } (4 \text{ x } 2) = 16 \\ \text{computations}}}$$

$$W = \begin{array}{|c|c|} \hline W_A & 0 \\ \hline 0 & W_B \\ \hline \end{array}$$

Example: Prune long-range interactions in images or text, or other types of sequential data.

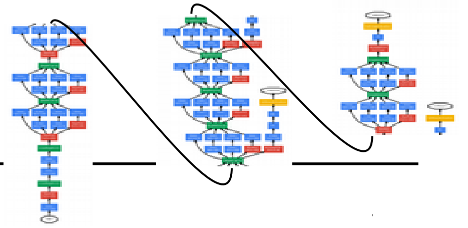
Step 3: Avoid Computational Bottlenecks

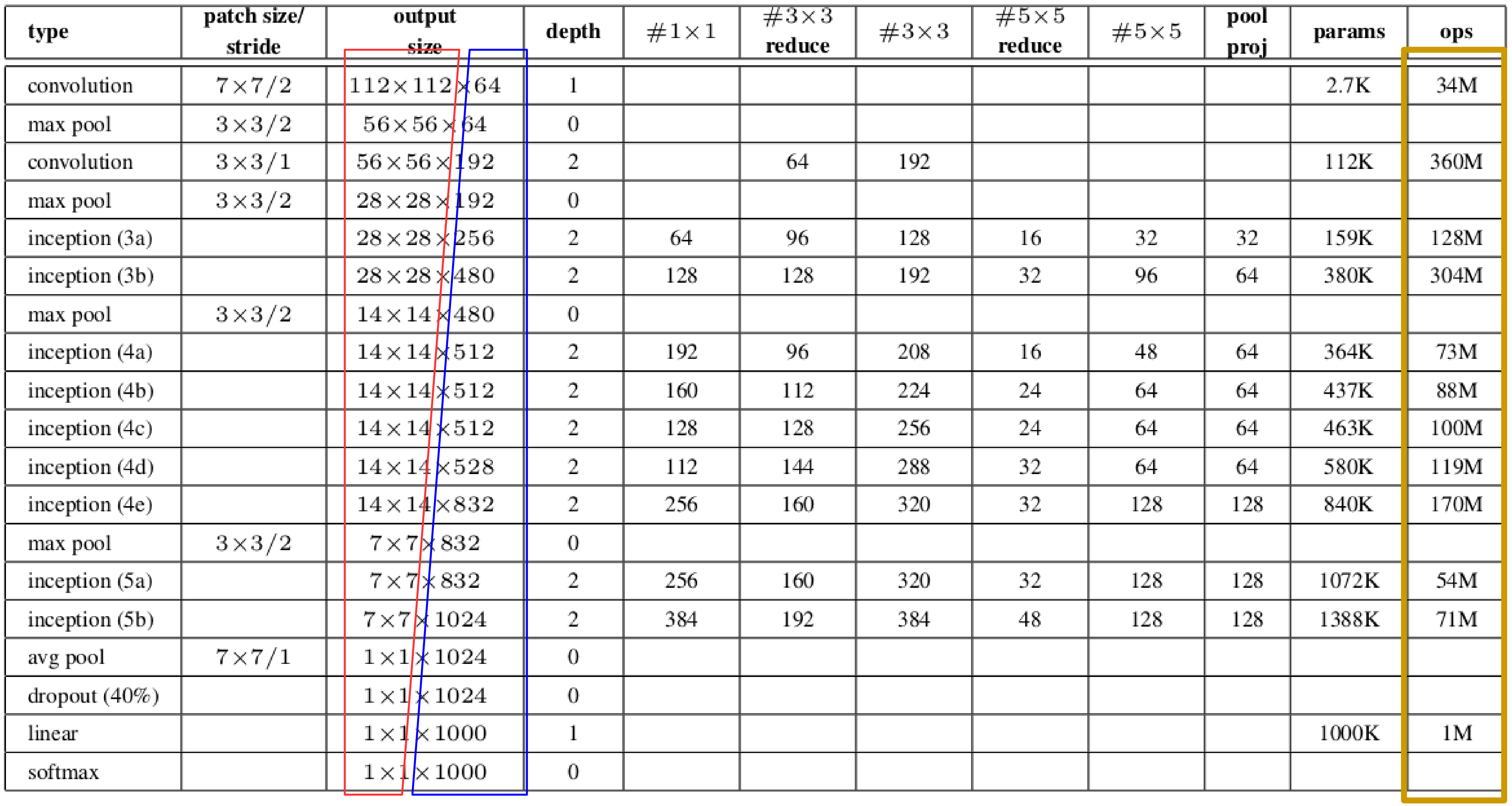


(source: Peemen et al. 2011: Speed sign detection and recognition by convolutional neural networks).

- Lower layers detect simple features at exact locations.
- Higher layers detect complex features at approximate locations.
- Layers progressively replace spatial information with semantic information → <u>keep dimensionality and number of connections low at each layer</u>.

Step 3: GoogleNet Example





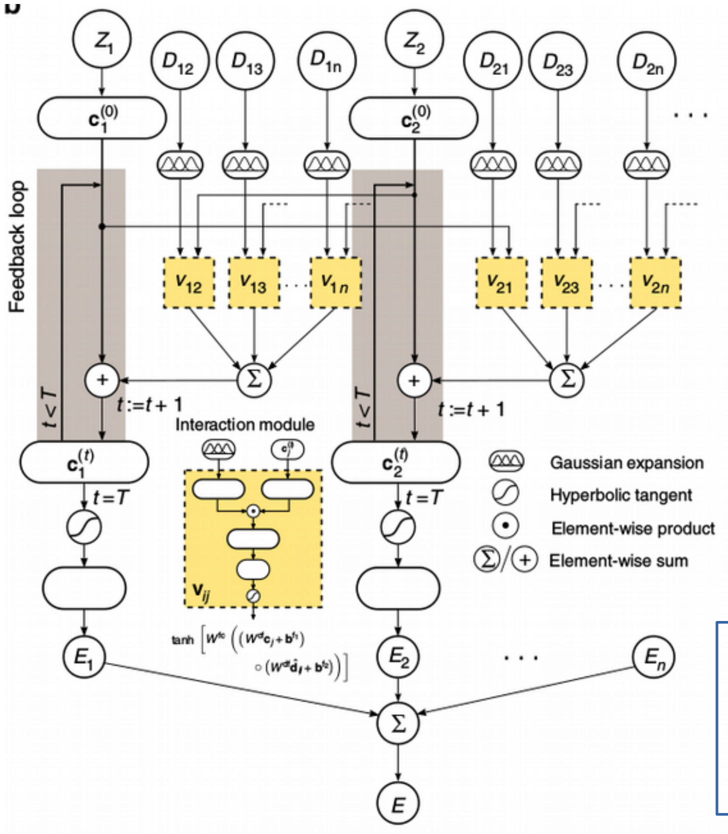
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

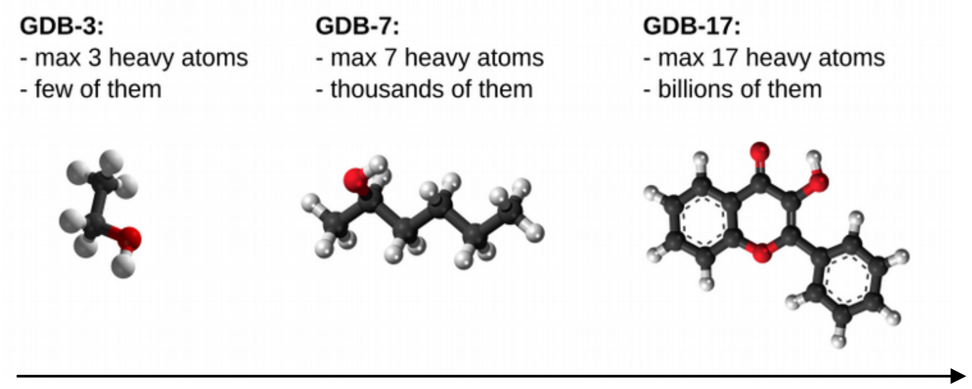
Table 1: GoogLeNet incarnation of the Inception architecture

Szegedy'14

Step 3: SchNet Example

SchNet: neural network that predicts molecular properties, and where each layer models an exchange of local information in the molecular graph [Schütt'17].





SchNet scales to a combinatorial number of molecules, while keeping dimensionality low.

Step 4: Map Neural Network to Hardware

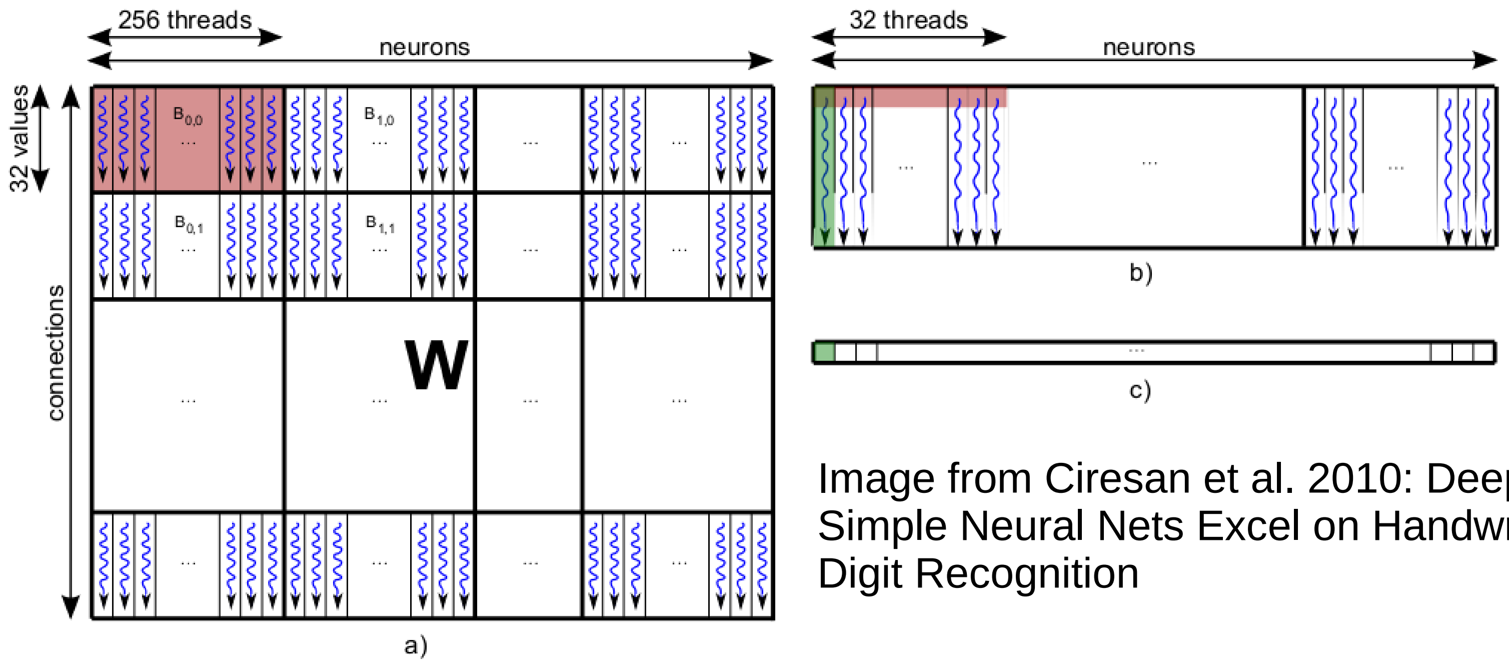


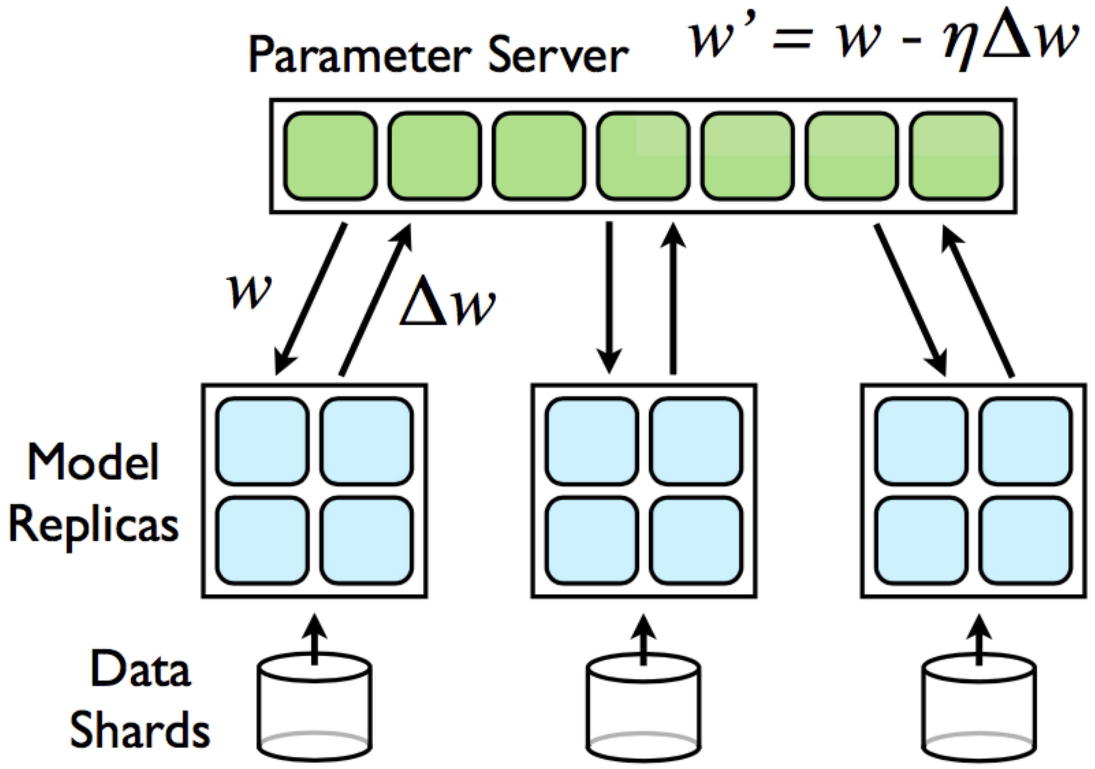
Image from Ciresan et al. 2010: Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition

In order for the training procedure to match the hardware specifications (e.g. CPU cache, GPU block size) optimally, neural network computations (e.g. matrix multiplications) must be decomposed into blocks of appropriate size.

These hardware-specific optimizations are already built in most fast neural network libraries (e.g. CUDNN, Torch, Tensorflow, MxNet, ...).

Step 5: Distributed Training

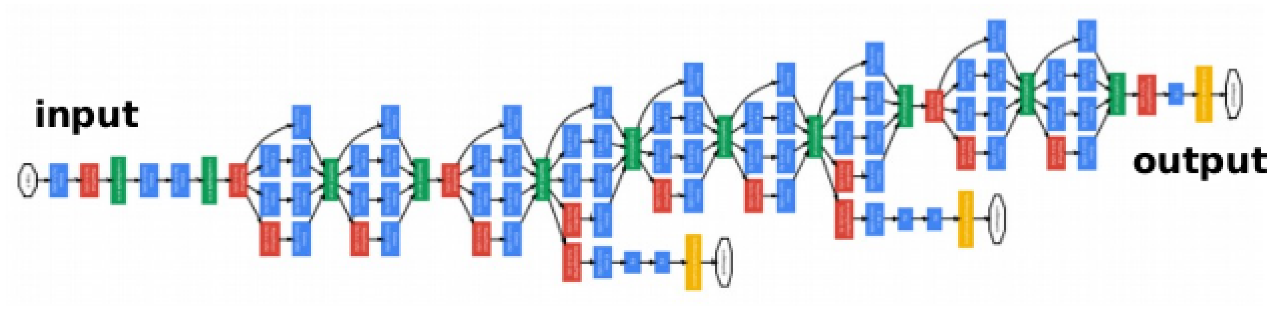
Example: Google's DistBelief Architecture [Dean'12]

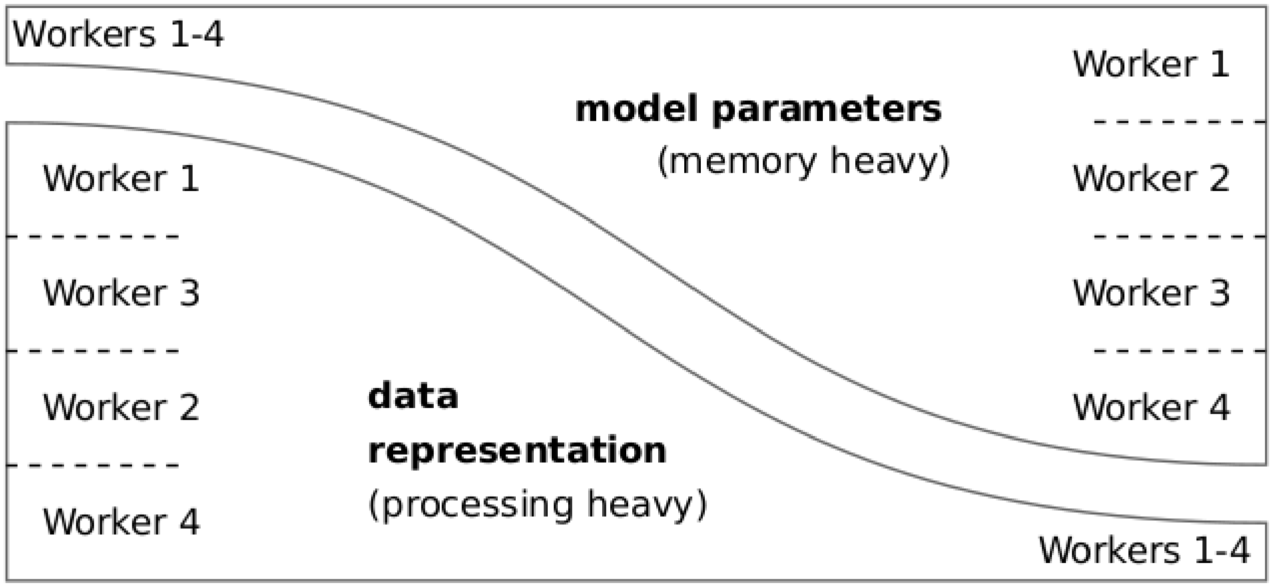


Each model replica trains on its own data, and synchronizes the model parameters it has learned with other replica via a dedicated parameter server.

Step 5: Distributed Training

Combining data-parallelism and model-parallelism





see also Krizhevsky'14: One weird trick for parallelizing convolutional neural networks

Summary

- Optimization of neural networks is harder than linear models, because of (1) nonconvexity and (2) conditioning issues.
- Therefore, we must carefully choose the initialization, the structure of the model (e.g. nonlinearities), and if necessary add momentum to gradient descent.
- Neural networks principal goal is to enable the transformation of large datasets into complex highly predictive models.
- To achieve this, it is important to make sure they can be trained as quickly as possible (e.g., minibatches, layered structure, avoiding bottlenecks, matching the hardware, distributed architectures).