Deep learning summer camp
2 Optimising feed-forward neural networks

Ole Winther

Dept for Applied Mathematics and Computer Science
Technical University of Denmark (DTU)

July 3, 2016

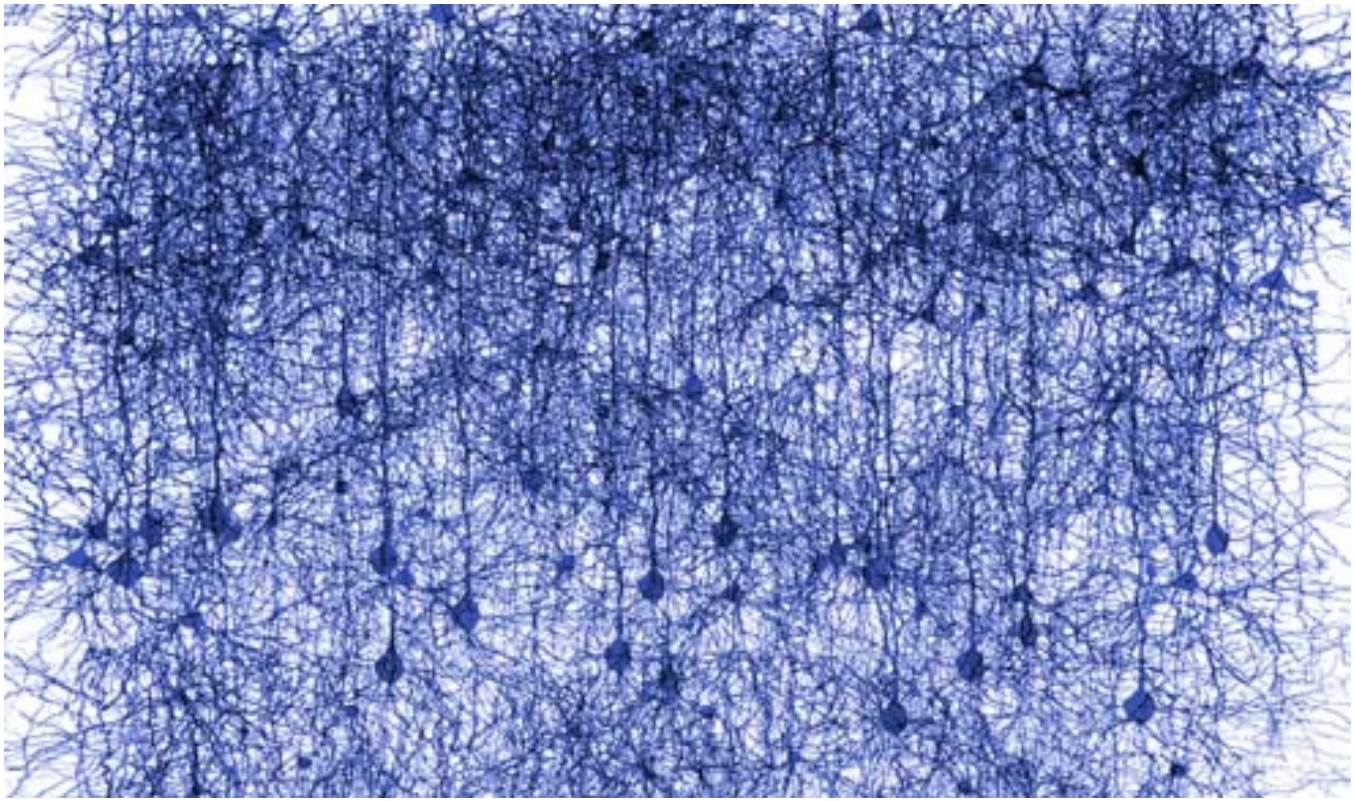
Objectives of talk

- Recap on the feed-forward neural network (FFNN)
- NN training
- Tricks of the trade!

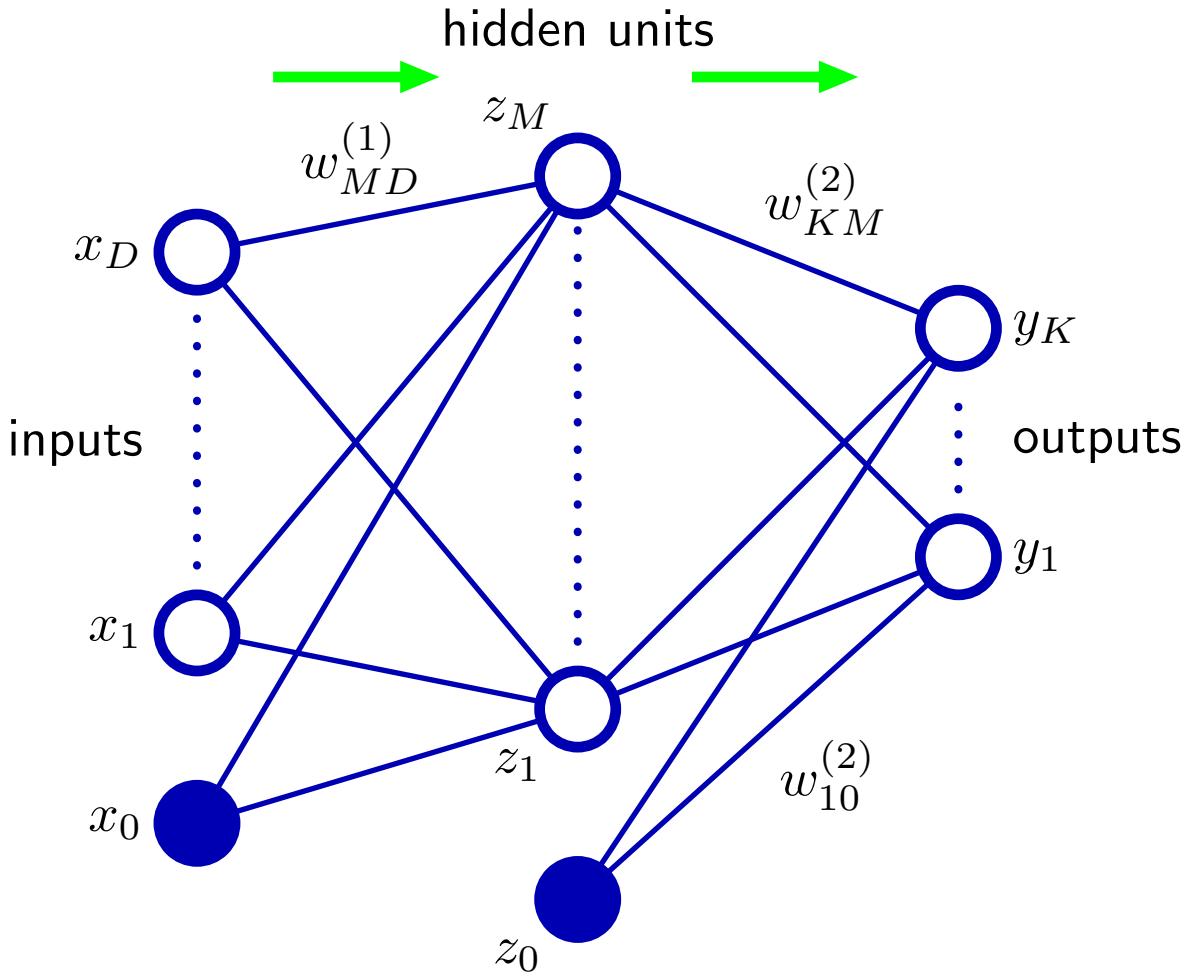
Part 1:
Neural network recap

Neural networks (NNs)

- Feedforward neural networks (FFNNs)
- Convolutional neural networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Auto-encoders (AE)



Feed forward neural networks



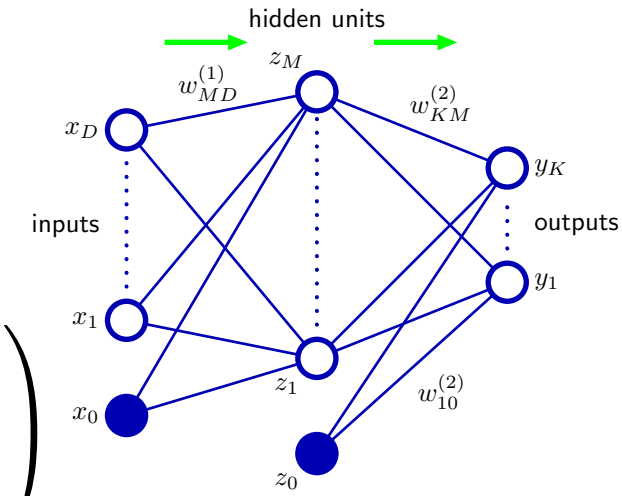
Neural network mapping

- Compute weighted sum of inputs:

$$\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$

- Output k two-layer network:

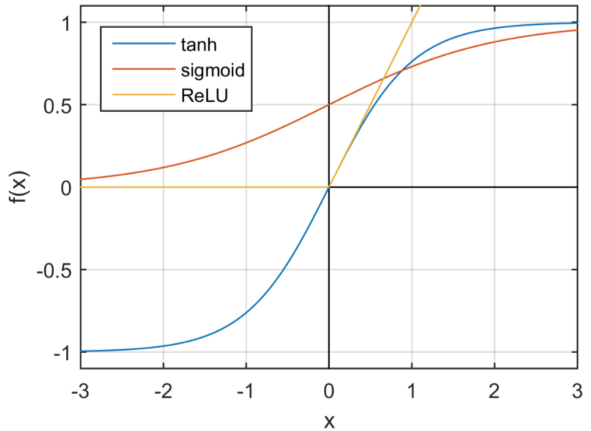
$$h_k^{(2)}(\mathbf{x}, \mathbf{w}) = f_2 \left(\sum_{j=0}^{M} w_{kj}^{(2)} f_1 \left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right)$$



- f_1 and f_2 hidden unit activation functions

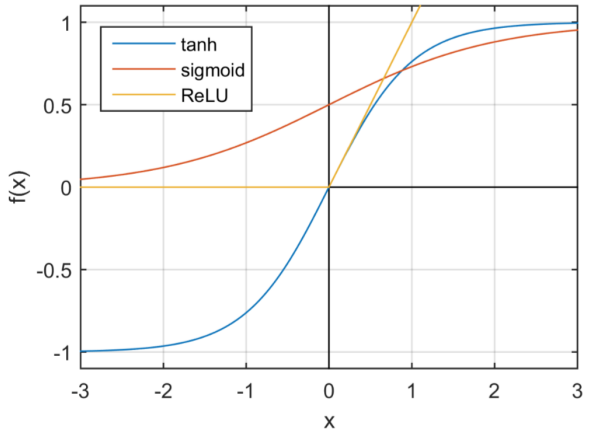
Non-linearity and training

- Linear activation functions will give a linear network.
- Logistic function $\sigma(a) = \frac{1}{1+e^{-a}}$
- Hyperbolic tangent $\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- Rectified linear $\mathrm{relu}(a) = \max(0, a)$



Non-linearity and training

- Linear activation functions will give a linear network.
- Logistic function $\sigma(a) = \frac{1}{1+e^{-a}}$
- Hyperbolic tangent $\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- Rectified linear $\mathrm{relu}(a) = \max(0, a)$



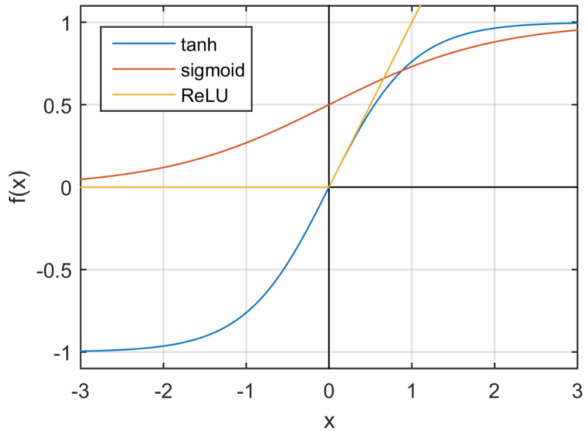
- Supervised learning
- Labeled training set

$$\mathcal{D} = \{(x_i, y_i) | i = 1, \ldots, n\} \ .$$

- Input x_i and output y_i.

Non-linearity and training

- Linear activation functions will give a linear network.
- Logistic function $\sigma(a) = \frac{1}{1+e^{-a}}$
- Hyperbolic tangent $\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- Rectified linear $\mathrm{relu}(a) = \max(0, a)$



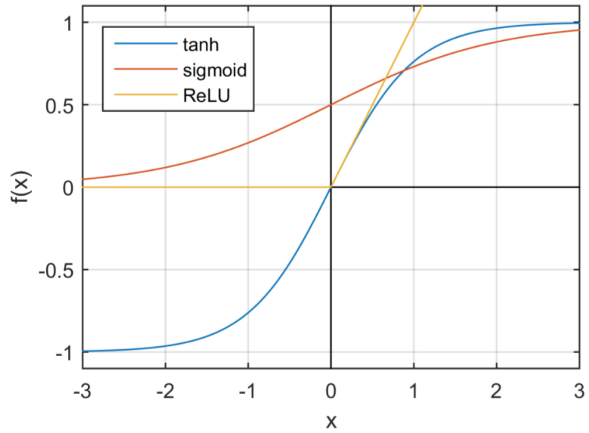
- Supervised learning
- Labeled training set

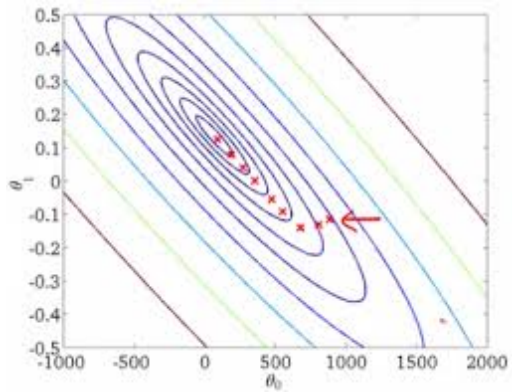
$$\mathcal{D} = \{(x_i, y_i) | i = 1, \ldots, n\} .$$

- Input x_i and output y_i.
- Minimize training error by (stochastic) gradient descent

Non-linearity and training

- Linear activation functions will give a linear network.
- Logistic function $\sigma(a) = \frac{1}{1+e^{-a}}$
- Hyperbolic tangent $\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- Rectified linear $\mathrm{relu}(a) = \max(0, a)$





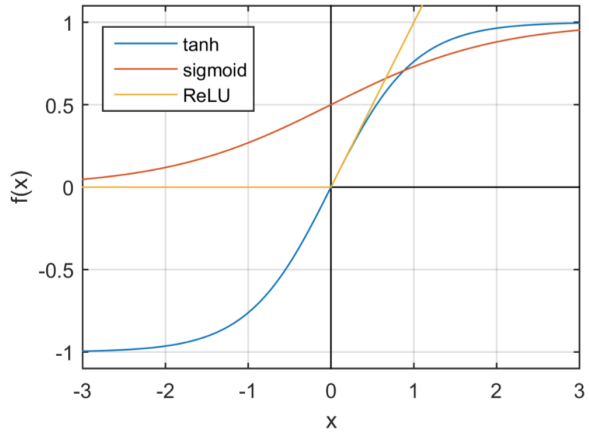
- Supervised learning
- Labeled training set

$$\mathcal{D} = \{(x_i, y_i) | i = 1, \ldots, n\} \ .$$

- Input x_i and output y_i.
- Minimize training error by (stochastic) gradient descent

Non-linearity and training

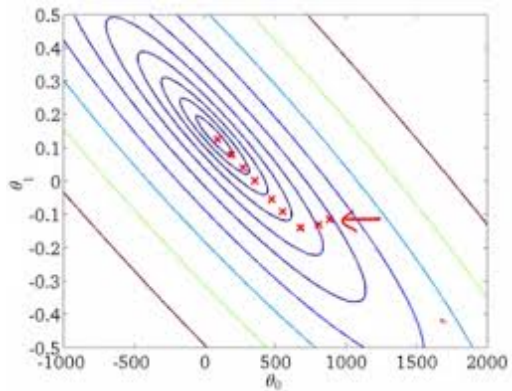
- Linear activation functions will give a linear network.
- Logistic function $\sigma(a) = \frac{1}{1+e^{-a}}$
- Hyperbolic tangent $\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$
- Rectified linear $\mathrm{relu}(a) = \max(0, a)$

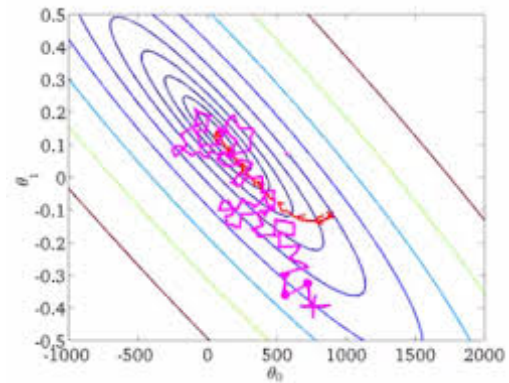


- Supervised learning
- Labeled training set

$$\mathcal{D} = \{(x_i, y_i) | i = 1, \ldots, n\} .$$

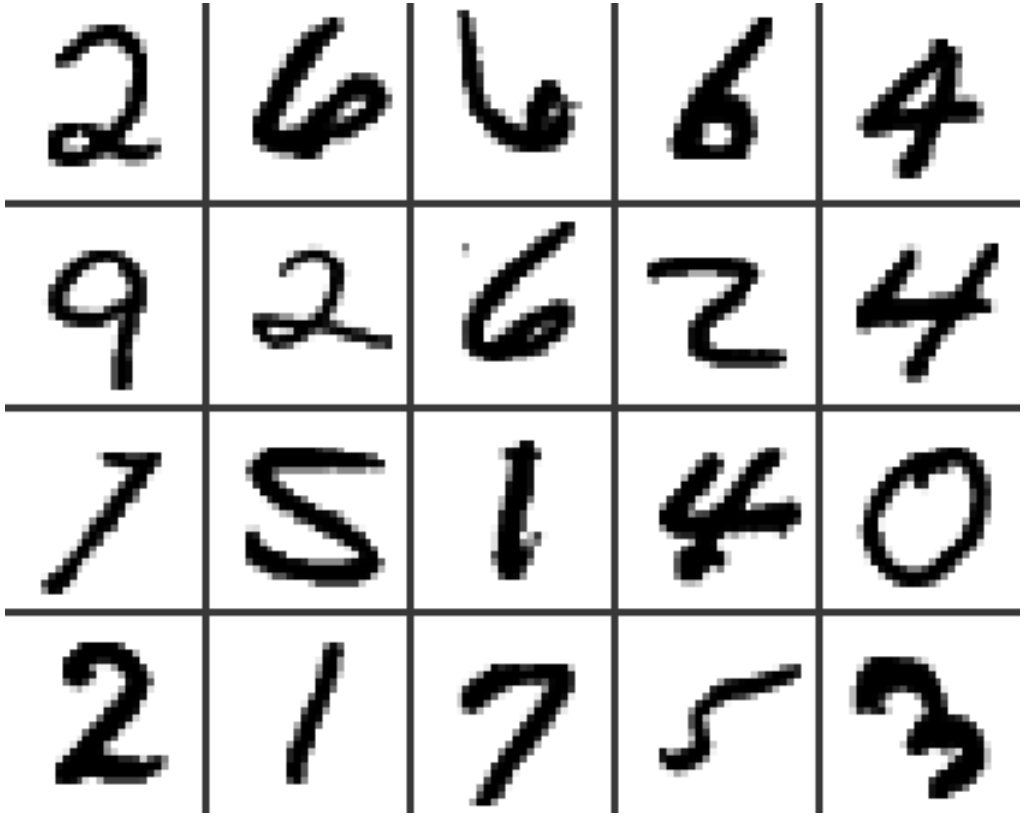
- Input x_i and output y_i.
- Minimize training error by (stochastic) gradient descent





Overfitting!

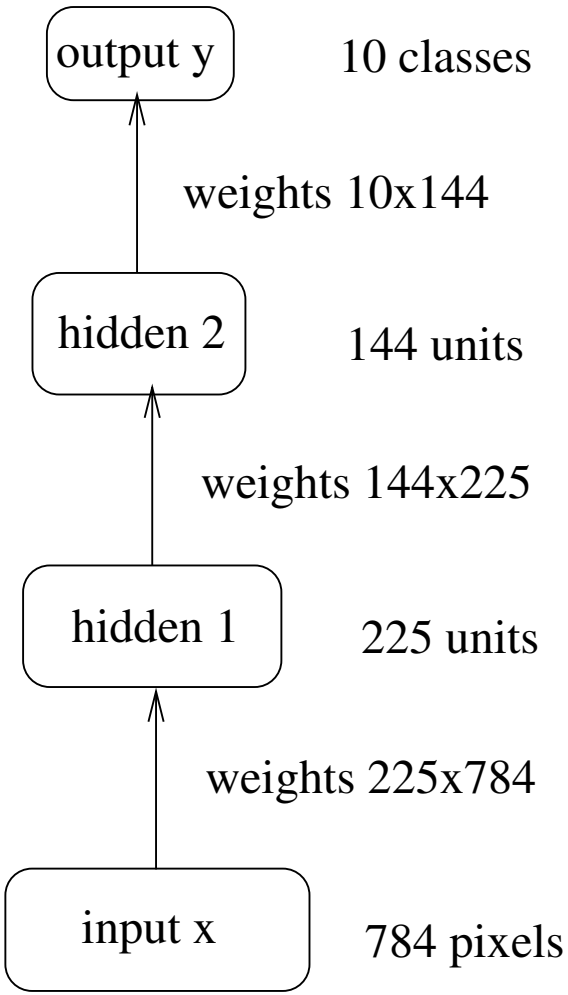
Example: MNIST handwritten digits



Train a network to classify 28×28 images.
Data: 60000 input images $\mathbf{x}_n$ and labels y_n, $n = 1, \ldots, 60k$.
Example model gives around 1.2% test error.

Many thanks to Tapani Raiko for sharing slides!

Example Network



$$\mathbf{h}^{(3)} = \text{softmax}(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)})$$

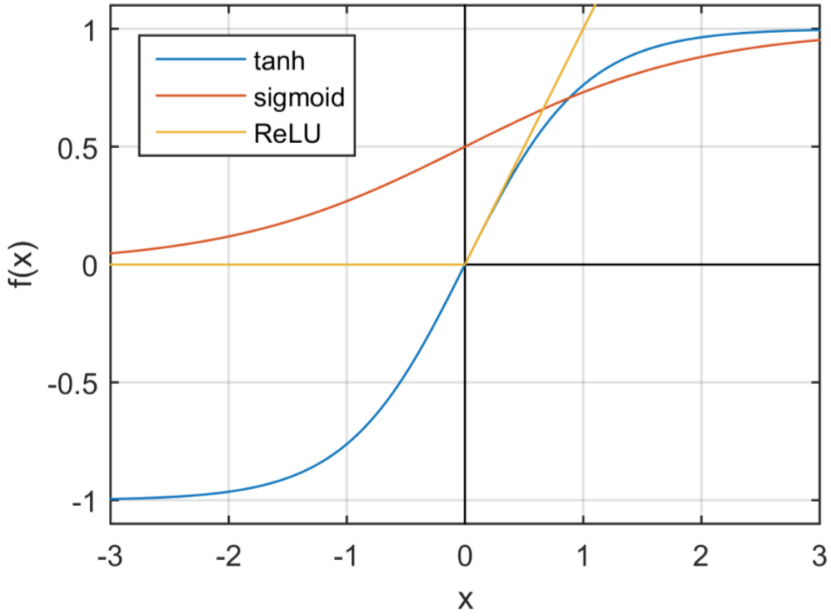
$$\mathbf{h}^{(2)} = \text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

$$\mathbf{h}^{(1)} = \text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\text{relu}(z) = max(0, z)$$

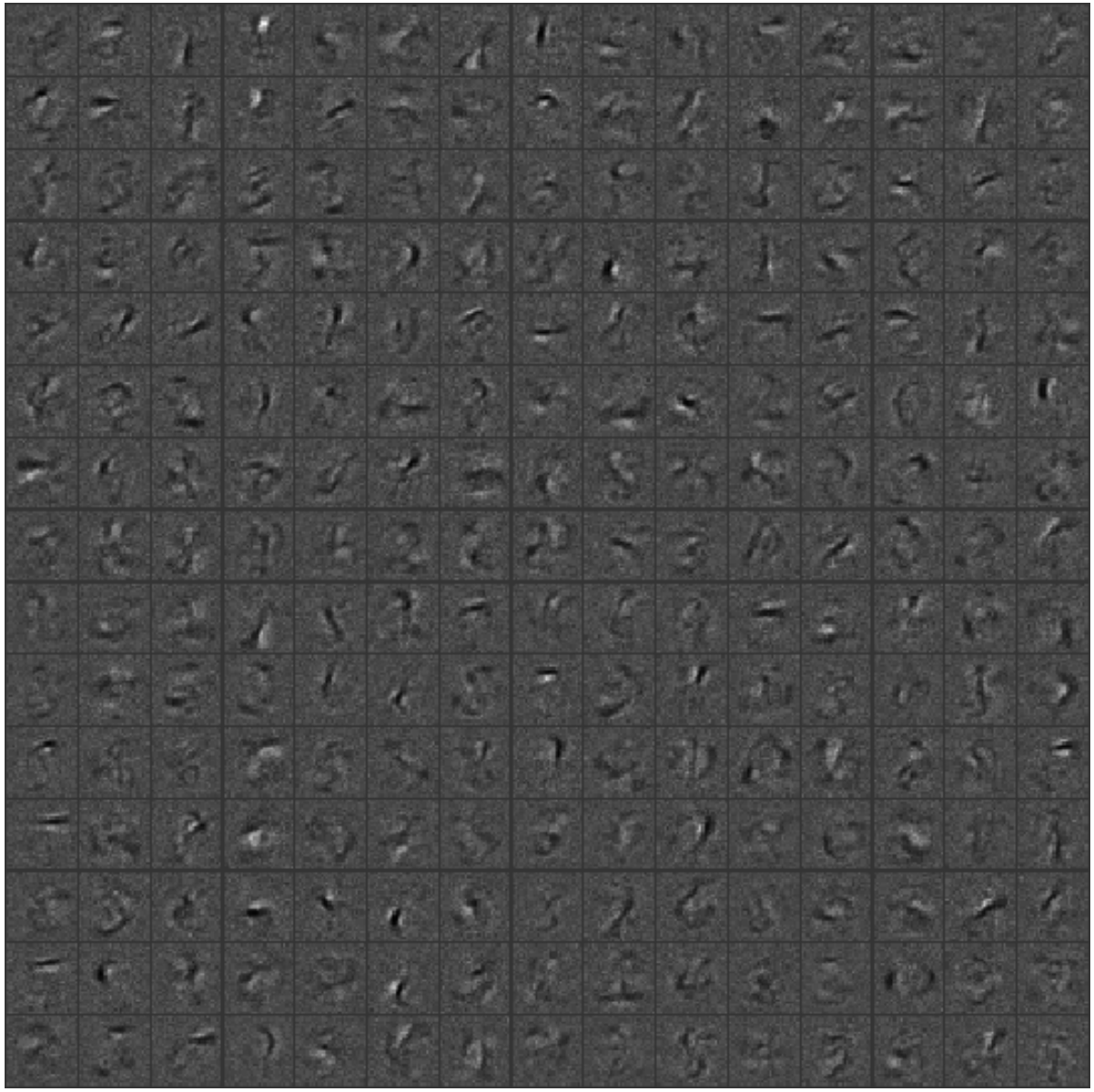
On activation functions



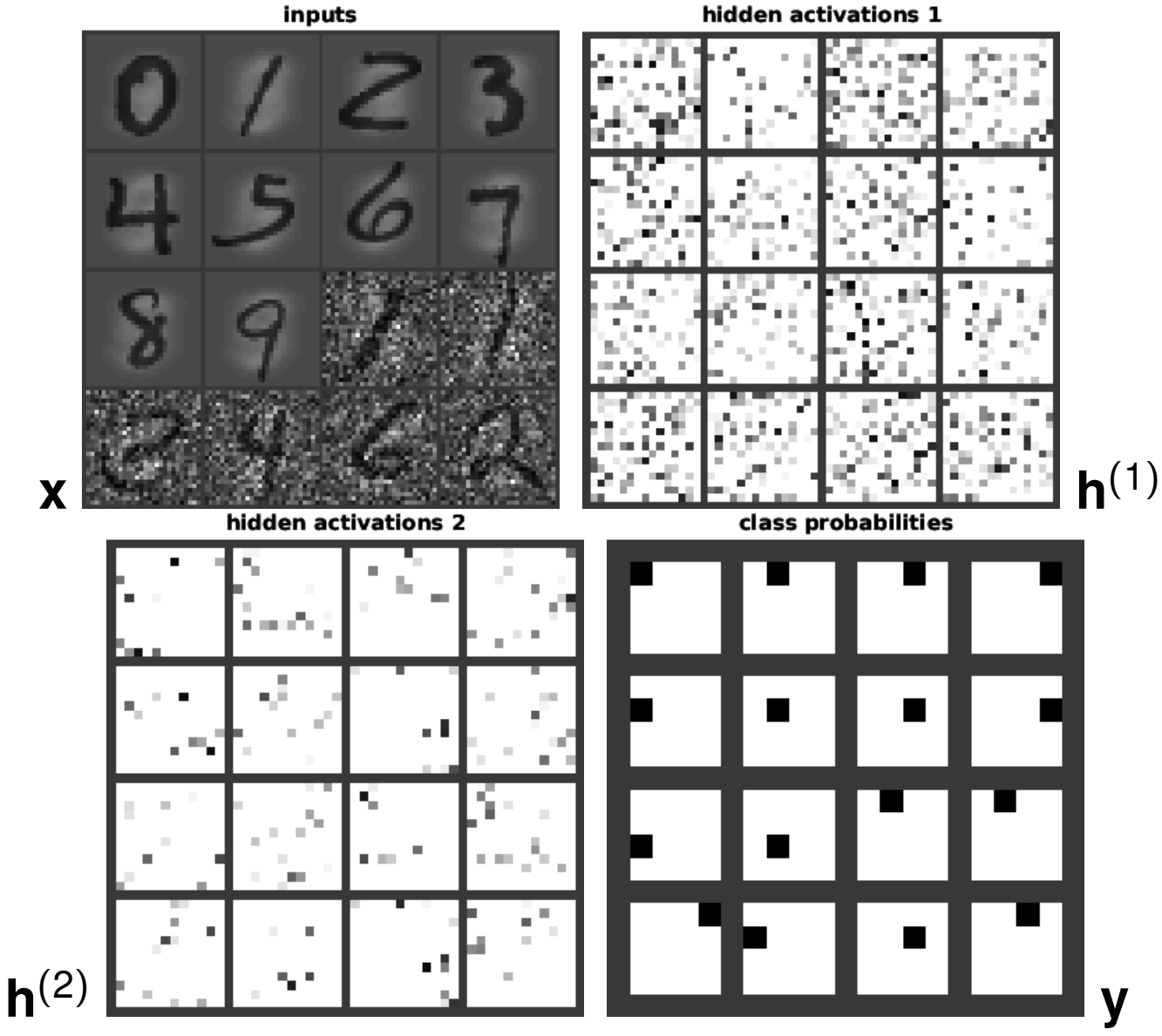
- $\text{relu}(z) = \max(0, z)$ is replacing old sigmoid and tanh.
- Note that identity function would lead into:

$$\begin{aligned}
\mathbf{h}^{(2)} &= \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)} \\
&= \mathbf{W}^{(2)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)} \\
&= (\mathbf{W}^{(2)}\mathbf{W}^{(1)})\mathbf{x} + (\mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}) \\
&= \mathbf{W}'\mathbf{x} + \mathbf{b}'
\end{aligned}$$

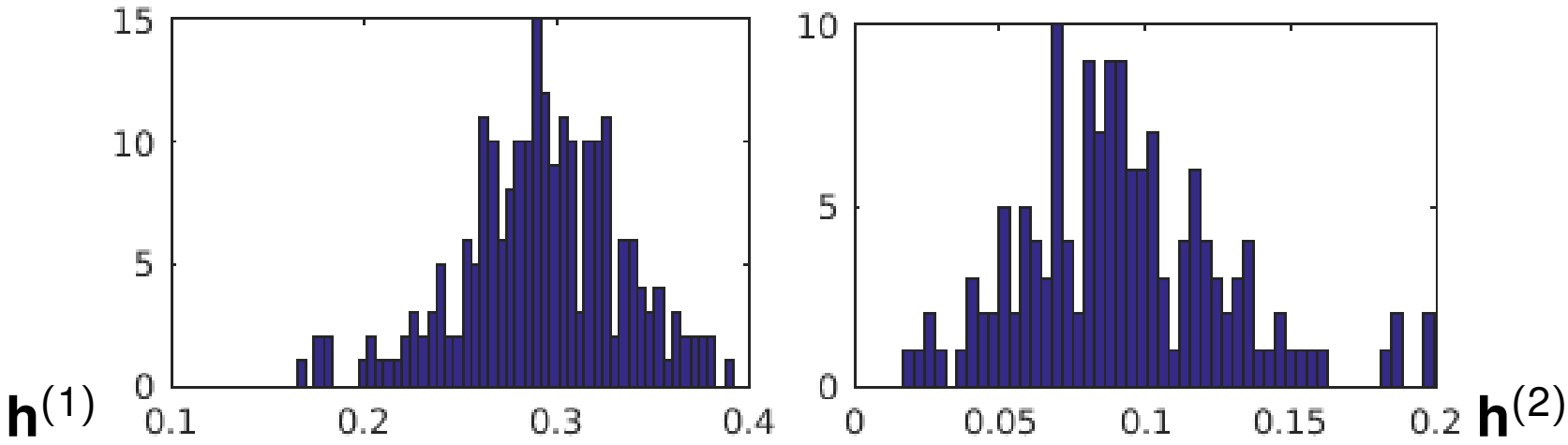
Weight matrix $\mathbf{W}^{(1)}$ size 225×784



Signals $\mathbf{x} \rightarrow \mathbf{h}^{(1)} \rightarrow \mathbf{h}^{(2)} \rightarrow \mathbf{h}^{(3)}$



On sparsity



How often $h_i > 0$? Histogram over units i.
(Sometimes units become completely dead.)

Part 2:
Neural network training

Training criterion

Find parameters

$$\theta = \{\mathbf{W}^{(L)}, \mathbf{b}^{(L)}\}$$

that minimize expected negative log-likelihood:

$$C = \mathbb{E}_{\text{data}} \left[-\log P(\mathbf{y}|\mathbf{x}, \theta) \right].$$

Learning becomes optimization.

Say we have a true distribution $P(\mathbf{y} \mid \mathbf{x})$ and we would like to find a model $Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta})$ that matches P. Let us study how maximizing expected negative log-likelihood $C = \mathbb{E}_P [-\log Q]$ works as a learning criterion.

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_\theta C(\theta) = \operatorname*{argmin}_\theta \mathbb{E}_{P(\mathbf{y}|\mathbf{x})} \left[-\log Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) \right].$$

Let us assume that there is a $\boldsymbol{\theta}^*$ for which $Q(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}^*) = P(\mathbf{y}|\mathbf{x})$. We can note that the gradient at $\boldsymbol{\theta}^*$

$$\begin{aligned}
&\frac{\partial}{\partial \boldsymbol{\theta}} \mathbb{E}_{P(\mathbf{y}|\mathbf{x})} \left[\log Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}^*) \right] \\
&= \mathbb{E}_{P(\mathbf{y}|\mathbf{x})} \left[\frac{\partial}{\partial \boldsymbol{\theta}} \log Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}^*) \right] \\
&= \int P(\mathbf{y} \mid \mathbf{x}) \frac{\frac{\partial}{\partial \boldsymbol{\theta}} Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}^*)}{Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}^*)} \mathrm{d}\mathbf{y} \\
&= \int \frac{\partial}{\partial \boldsymbol{\theta}} Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}^*) \mathrm{d}\mathbf{y} \\
&= \frac{\partial}{\partial \boldsymbol{\theta}} \int Q(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}^*) \mathrm{d}\mathbf{y} = \frac{\partial}{\partial \theta} 1 = 0
\end{aligned}$$

becomes zero, that is, the learning converges when $Q = P$. Therefore the expected log-likelihood is a reasonable training criterion.

Classification - one hot encoding and cross-entropy

- MNIST, output labels: $0, 1, \ldots, 9$.
- Convenient to use a sparse one hot encoding:

$$0 \to \mathbf{y} = (1,0,0,0,0,0,0,0,0,0)^T$$
$$1 \to \mathbf{y} = (0,1,0,0,0,0,0,0,0,0)^T$$
$$2 \to \mathbf{y} = (0,0,1,0,0,0,0,0,0,0)^T$$
$$\ldots.$$
$$9 \to \mathbf{y} = (0,0,0,0,0,0,0,0,0,1)^T$$

- Output

$$\mathbf{h}^{(3)} = \text{softmax}(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)})$$

  interpreted as class(-conditional) probability.
- Cross-entropy cost - sum over data and label

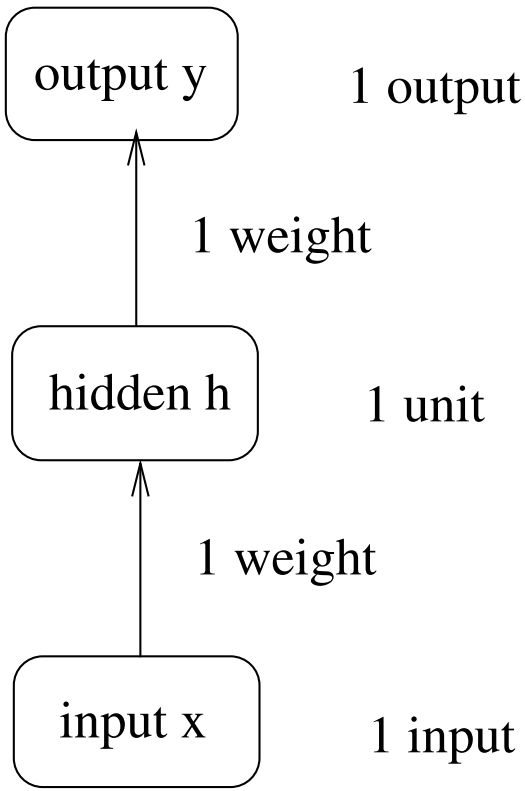
$$C = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk} \log h_{nk}^{(3)}$$

Gradient descent

- Simple algorithm for minimizing the training criterion C.
- Gradient $\mathbf{g} = \nabla_{\boldsymbol{\theta}} C(\theta) = \begin{pmatrix} \frac{\partial C}{\partial \theta_1} \\ \vdots \\ \frac{\partial C}{\partial \theta_n} \end{pmatrix}$
- Iterate $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{g}_k$
- Notation: iteration k, stepsize (or learning rate) η_k

Backpropagation (Linnainmaa, 1970)

Computing gradients in a network.



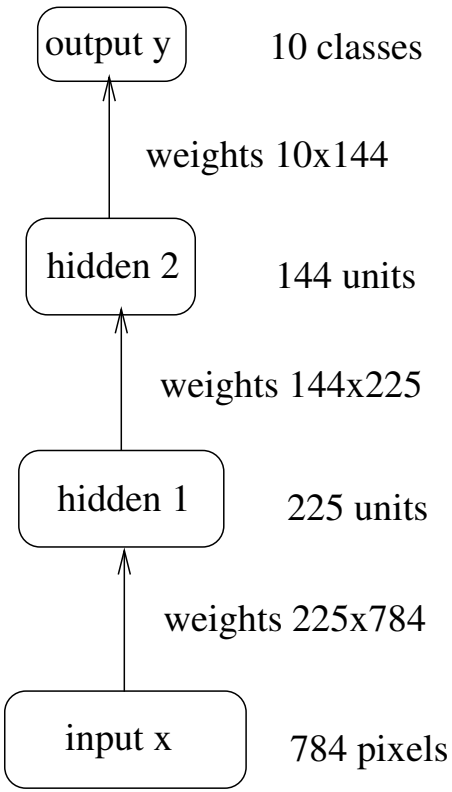
- First with scalars. Use chain rule:

$$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial w_2}$$

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial w_1}$$

- Chain rule: $\frac{\partial h^{(2)}}{\partial x} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial x}$

Backpropagation



- Multi-dimensional:

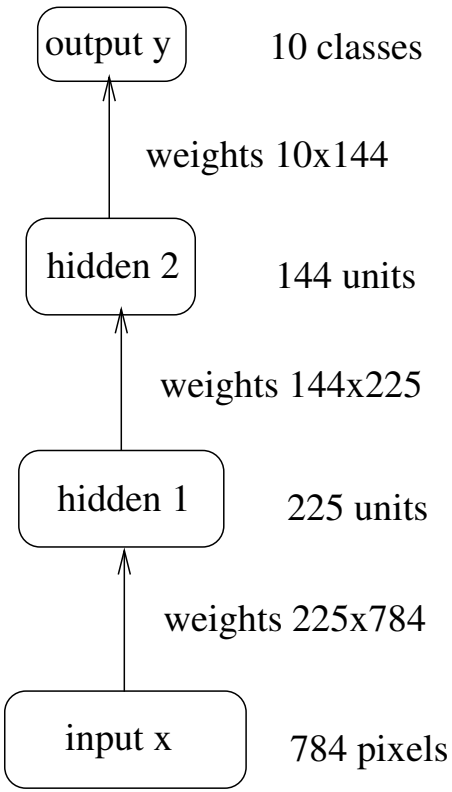
$$\frac{\partial C}{\partial W_{ij}^{(3)}} = \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial W_{ij}^{(3)}}$$

$$\frac{\partial C}{\partial W_{jk}^{(2)}} = \sum_i \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial W_{jk}^{(2)}}$$

$$\frac{\partial C}{\partial W_{kl}^{(1)}} = \sum_j \sum_i \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial h_k^{(1)}} \frac{\partial h_k^{(1)}}{\partial W_{kl}^{(1)}}$$

- *How many paths - for two hidden layers*
- *as a function of depth?*

Backpropagation



- Multi-dimensional:

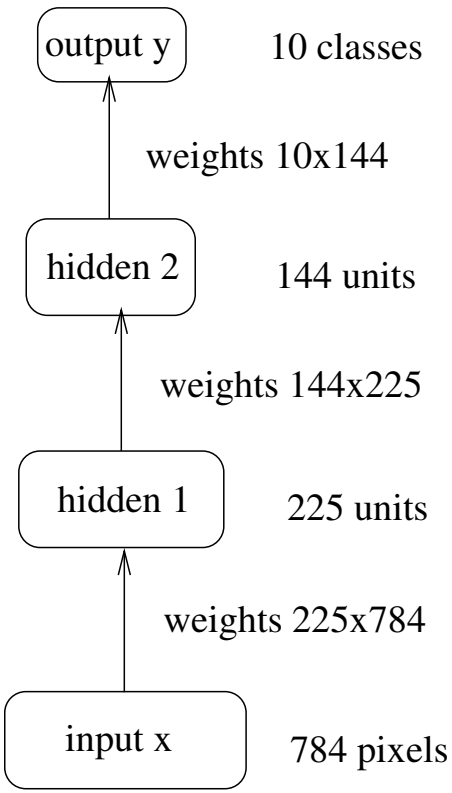
$$\frac{\partial C}{\partial W_{ij}^{(3)}} = \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial W_{ij}^{(3)}}$$

$$\frac{\partial C}{\partial W_{jk}^{(2)}} = \sum_i \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial W_{jk}^{(2)}}$$

$$\frac{\partial C}{\partial W_{kl}^{(1)}} = \sum_j \sum_i \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial h_k^{(1)}} \frac{\partial h_k^{(1)}}{\partial W_{kl}^{(1)}}$$

- *How many paths - for two hidden layers*
- *as a function of depth?*

Backpropagation - dynamic programming



- Store intermediate results

$$\frac{\partial C}{\partial h_j^{(2)}} = \sum_i \frac{\partial C}{\partial h_i^{(3)}} \frac{\partial h_i^{(3)}}{\partial h_j^{(2)}}$$

$$\frac{\partial C}{\partial h_k^{(1)}} = \sum_j \frac{\partial C}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial h_k^{(1)}}$$

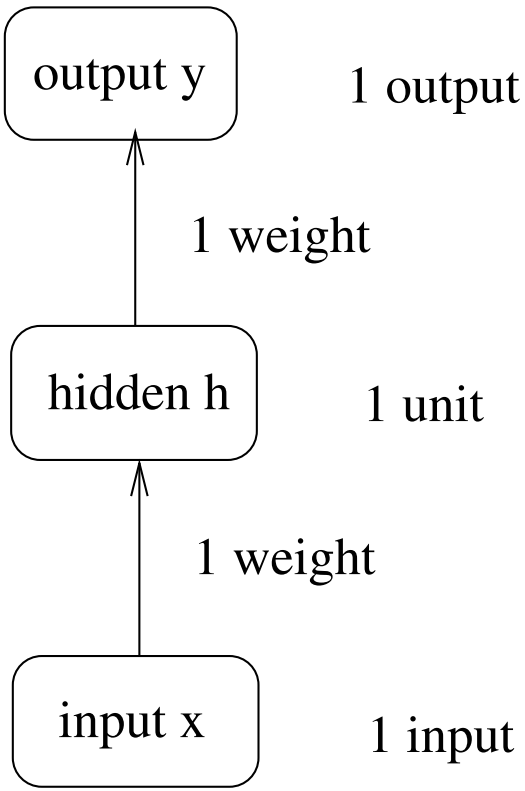
- In general

$$\frac{\partial C}{\partial h_j^{(l)}} = \sum_i \frac{\partial C}{\partial h_i^{(l+1)}} \frac{\partial h_i^{(l+1)}}{\partial h_j^{(l)}}$$

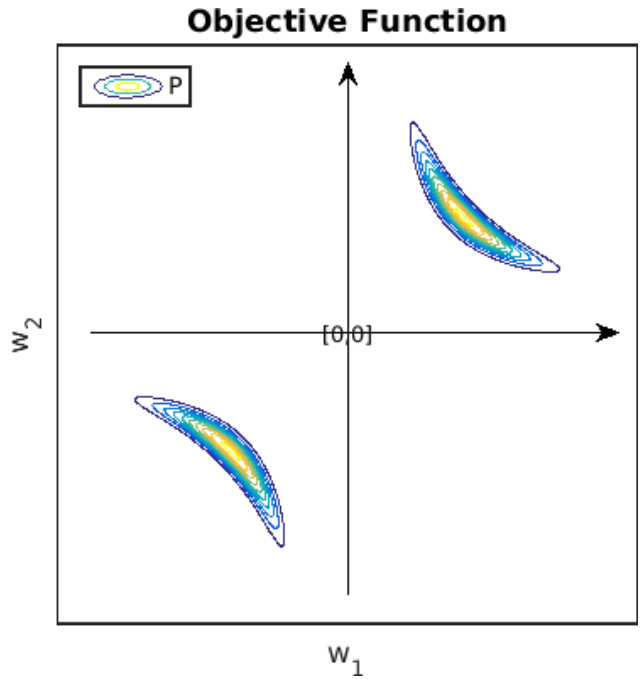
- and gradient:

$$\frac{\partial C}{\partial W_{ij}^{(l)}} = \frac{\partial C}{\partial h_i^{(l)}} \frac{\partial h_i^{(l)}}{\partial W_{ij}^{(l)}}$$

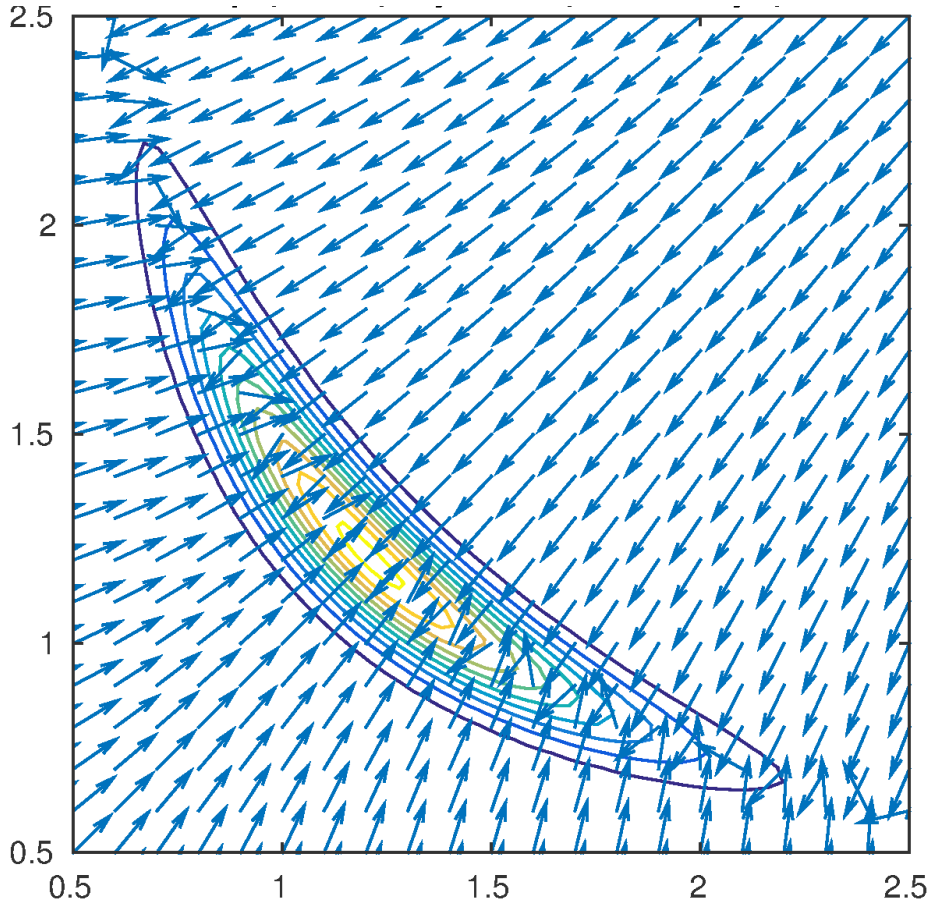
Tiny Example



- $y \sim \mathcal{N}(w_2 h, 1)$
- $h = w_1 x$
- "Data set": $\{x = 1, y = 1.5\}$
- Some weight decay.
- $C = (w_1 w_2 - 1.5)^2 + 0.04(w_1^2 + w_2^2)$

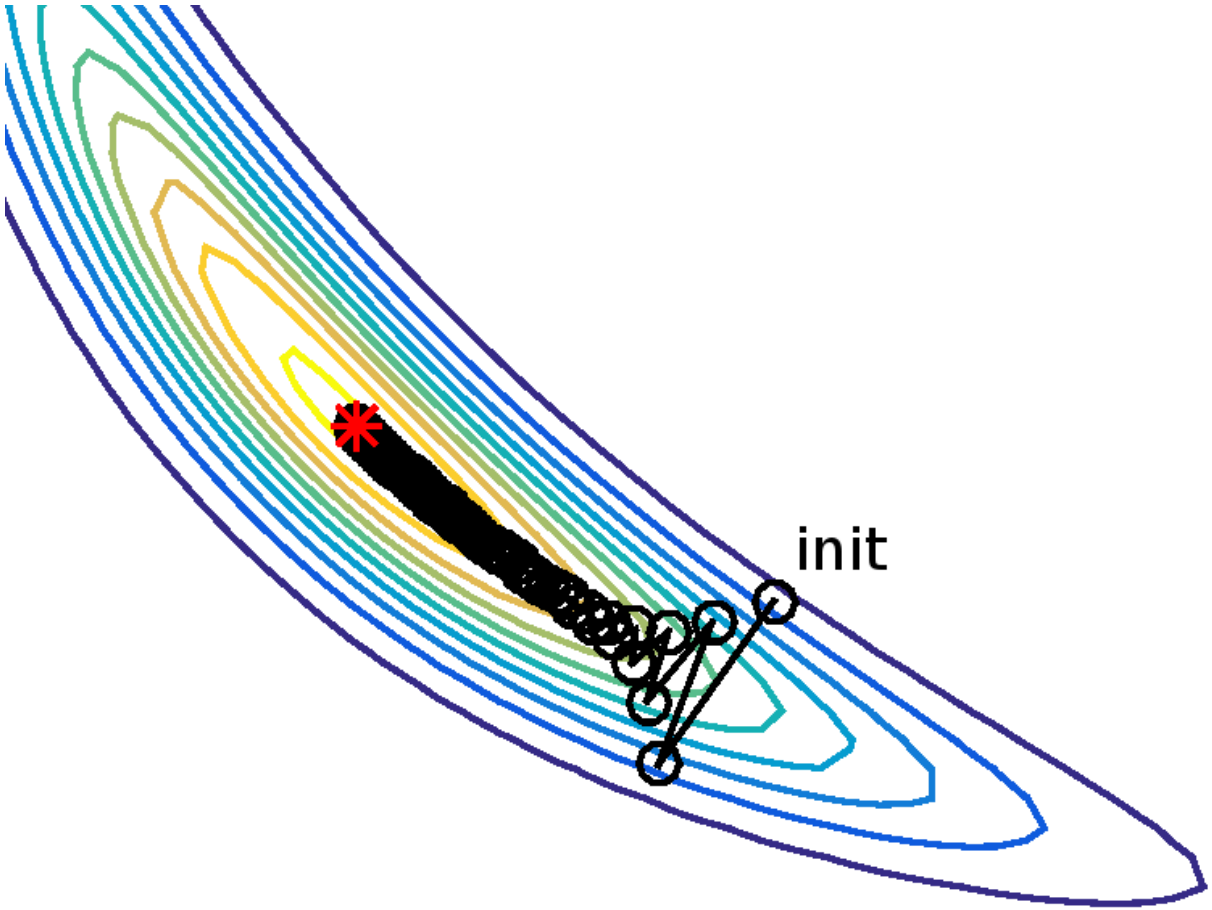


Gradient $\mathbf{g} = \nabla_\theta C(\theta) = \begin{pmatrix} \frac{\partial C}{\partial \theta_1} \\ \vdots \\ \frac{\partial C}{\partial \theta_n} \end{pmatrix}$



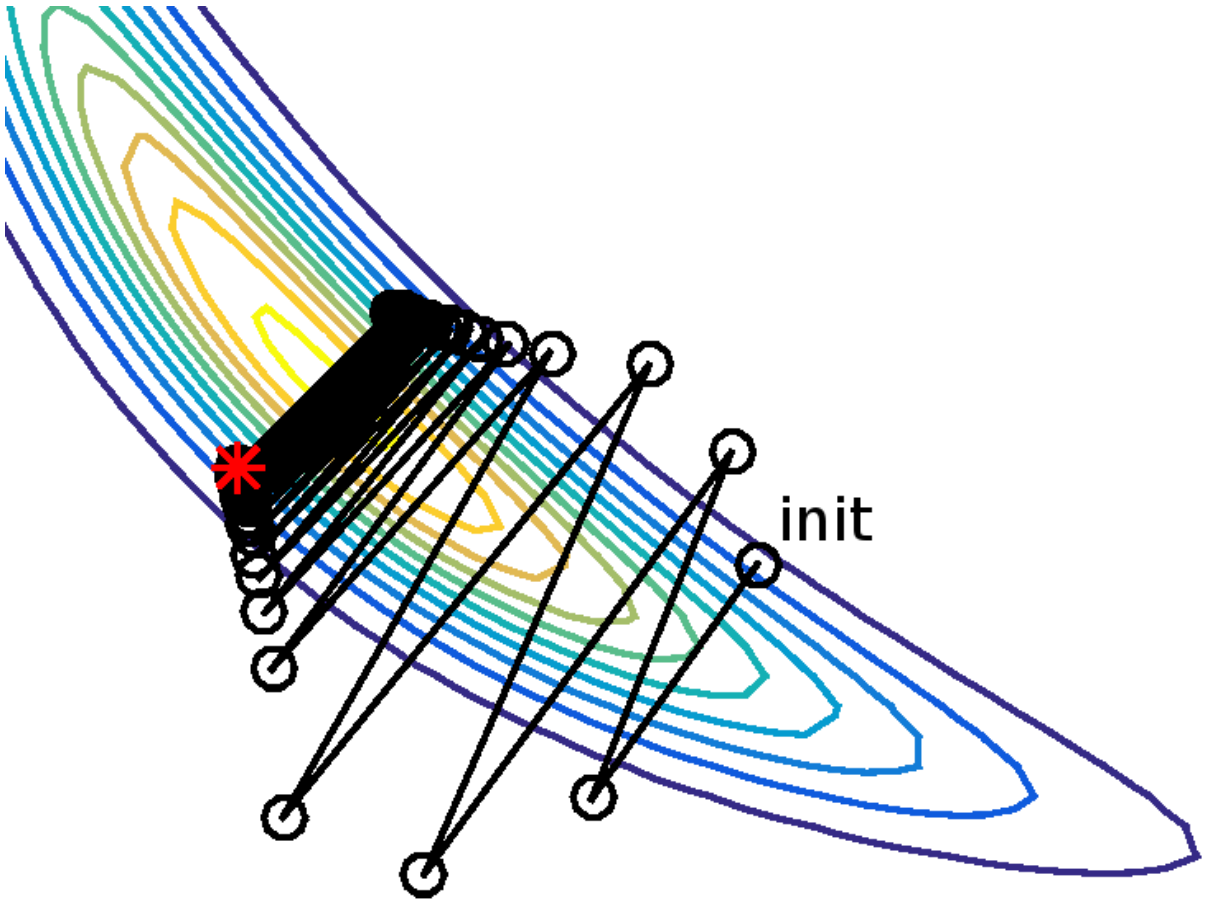
Gradient descent, $\eta_k = 0.25$ ($\rightarrow$ too slow)

$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$, iteration k, stepsize (or learning rate) η_k



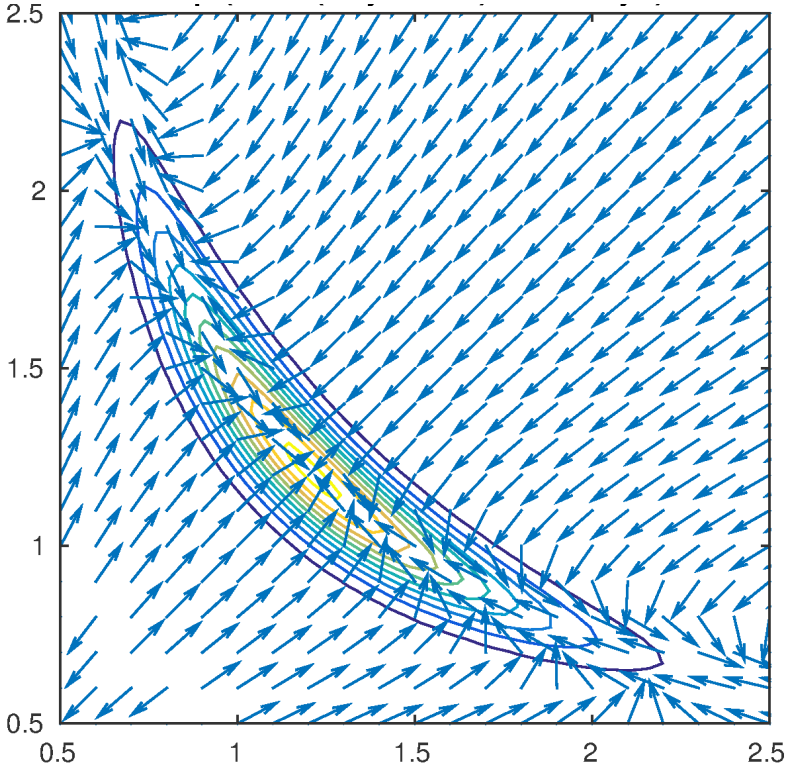
Gradient descent, $\eta_k = 0.35$ ($\rightarrow$ oscillates)

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$



Newton's method, too complex

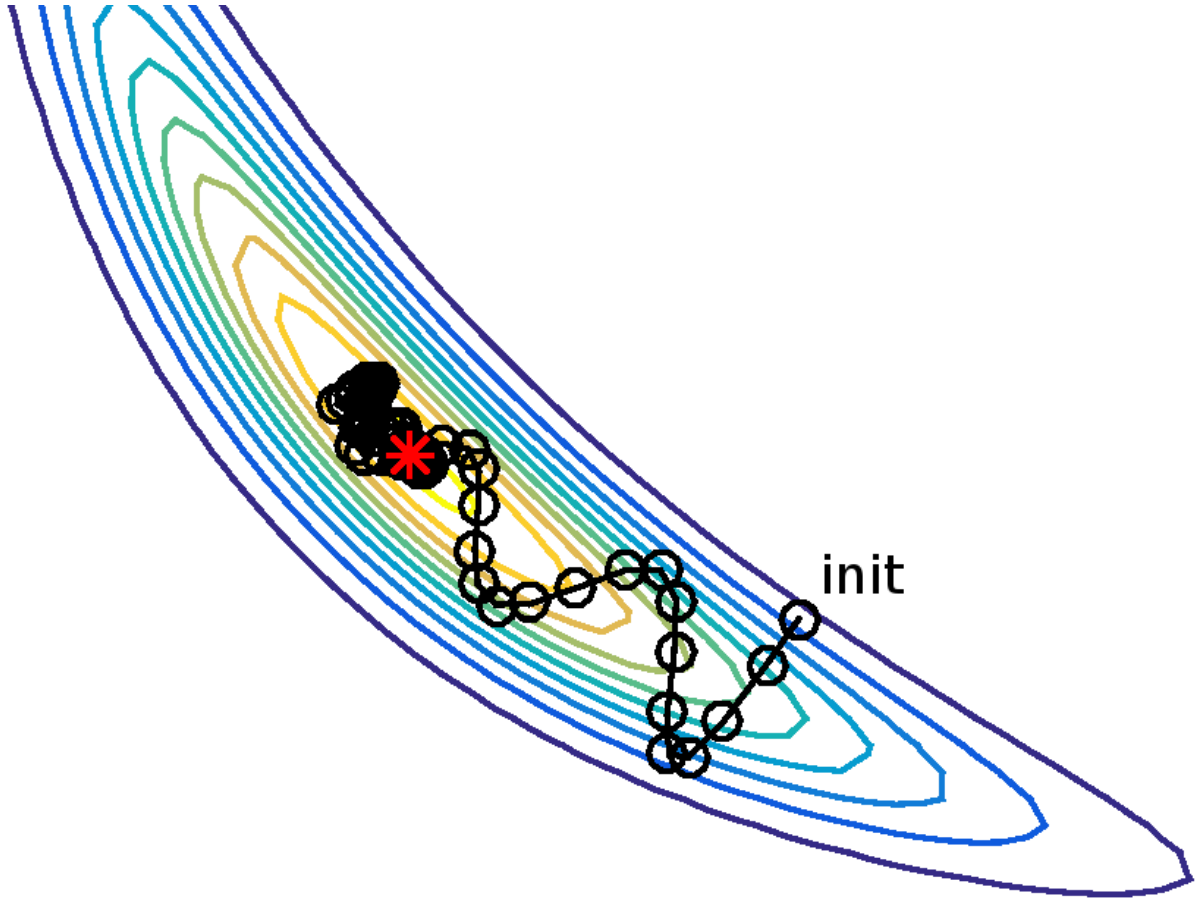
$$\theta_{k+1} = \theta_k - \mathbf{H}_k^{-1}\mathbf{g}_k, \quad \mathbf{H} = \begin{pmatrix} \frac{\partial^2 C}{\partial\theta_1\partial\theta_1} & \cdots & \frac{\partial^2 C}{\partial\theta_1\partial\theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 C}{\partial\theta_n\partial\theta_1} & \cdots & \frac{\partial^2 C}{\partial\theta_n\partial\theta_n} \end{pmatrix}$$



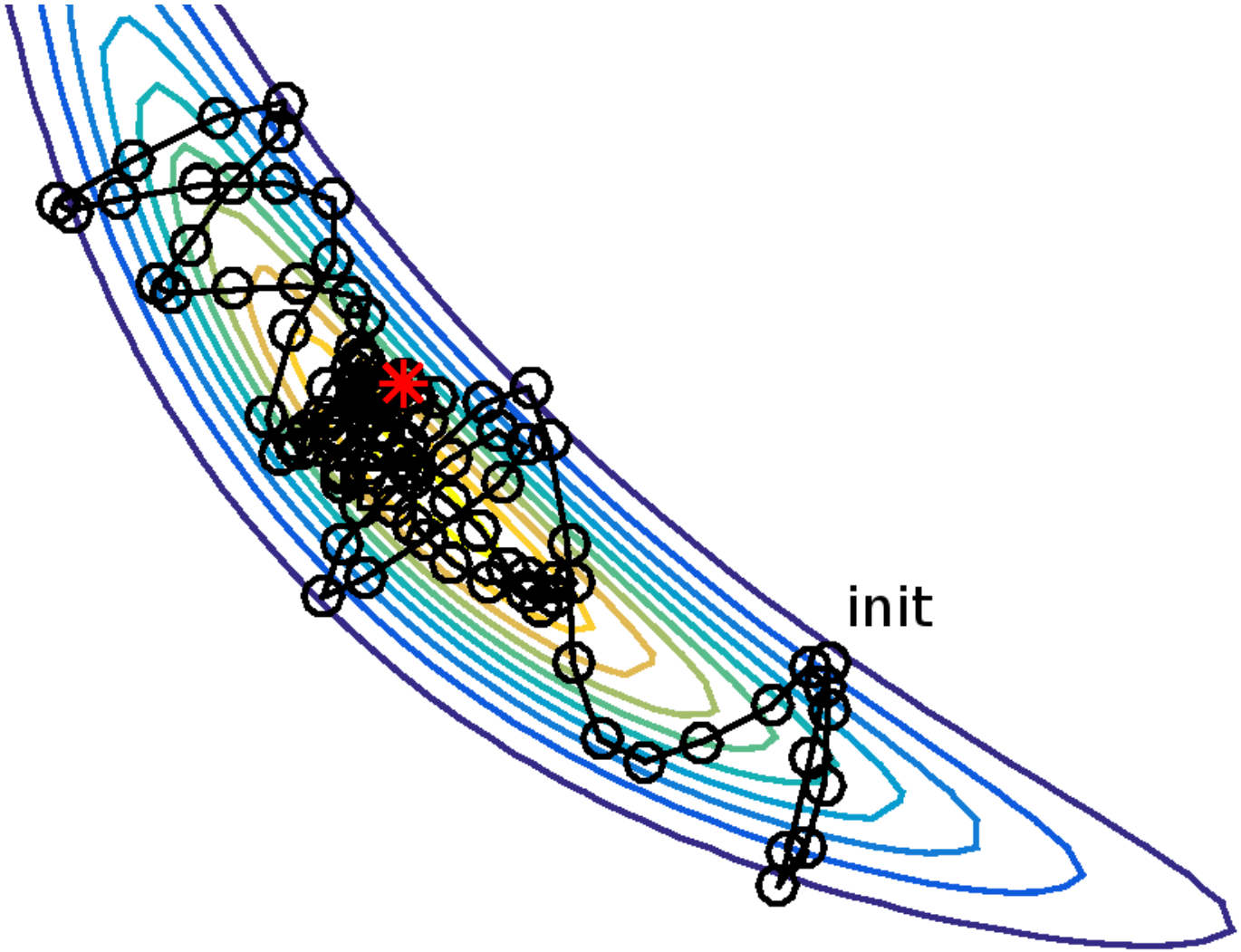
- Less oscillations.
- Points to the wrong direction in places (solvable).
- Computational complexity: #params^3 (prohibitive).
- There are approximations, but not very popular.

Momentum method (Polyak, 1964)

$$\mathbf{m}_{k+1} = \alpha \mathbf{m}_k - \eta_k \mathbf{g}_k$$
$$\theta_{k+1} = \theta_k + \mathbf{m}_{k+1}$$



Momentum method with noisy gradient

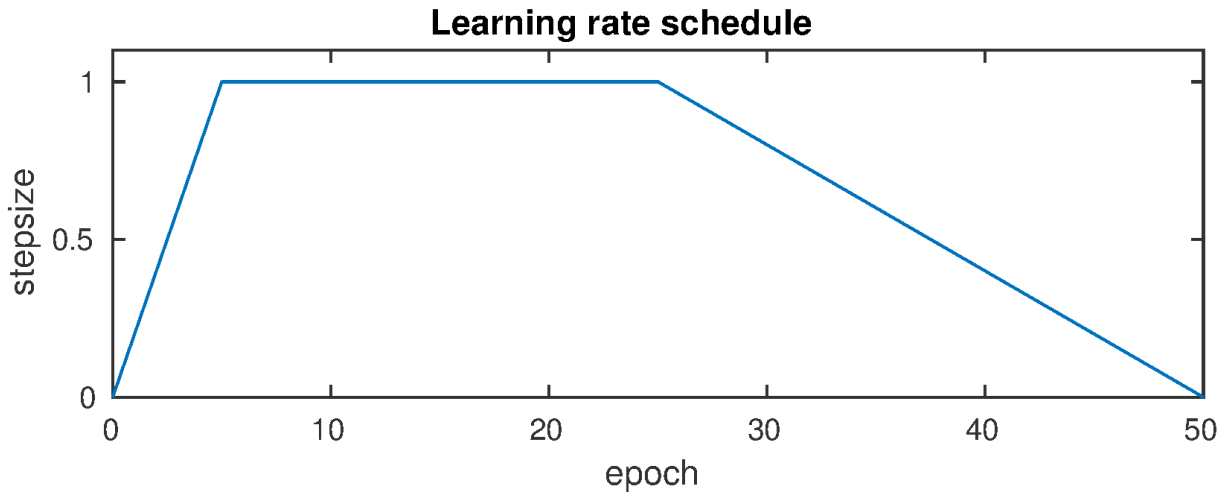


Mini-batch training

- No need to have an accurate estimate of $\mathbf{g}$.
- Use only a small batch of training data at once.
- Leads into many updates per epoch (=seeing data once).
- E.g. 600 updates with 100 samples per epoch in MNIST.

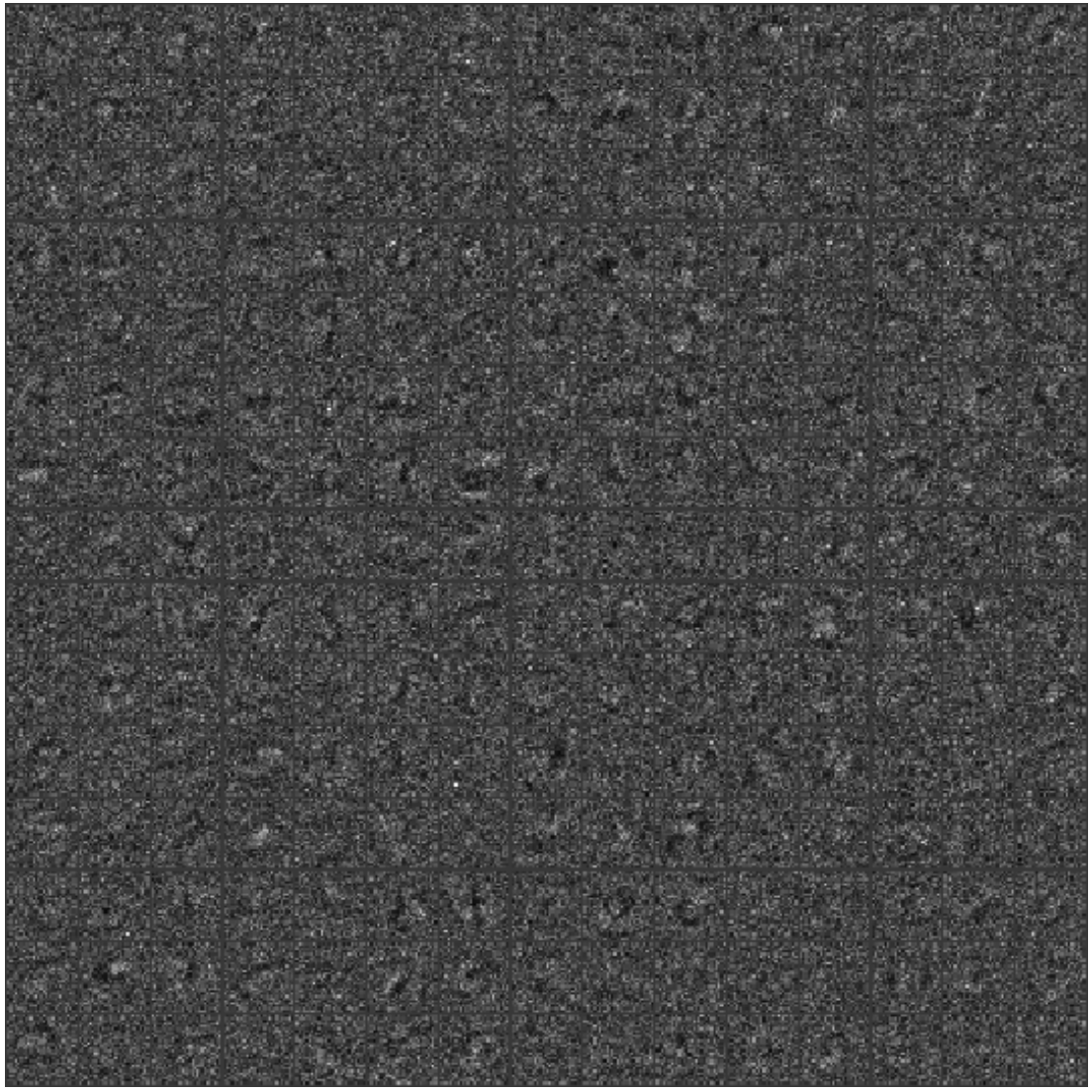
Mini-batch training

- No need to have an accurate estimate of **g**.
- Use only a small batch of training data at once.
- Leads into many updates per epoch (=seeing data once).
- E.g. 600 updates with 100 samples per epoch in MNIST.
- Important to anneal stepsize η_k towards the end, e.g.

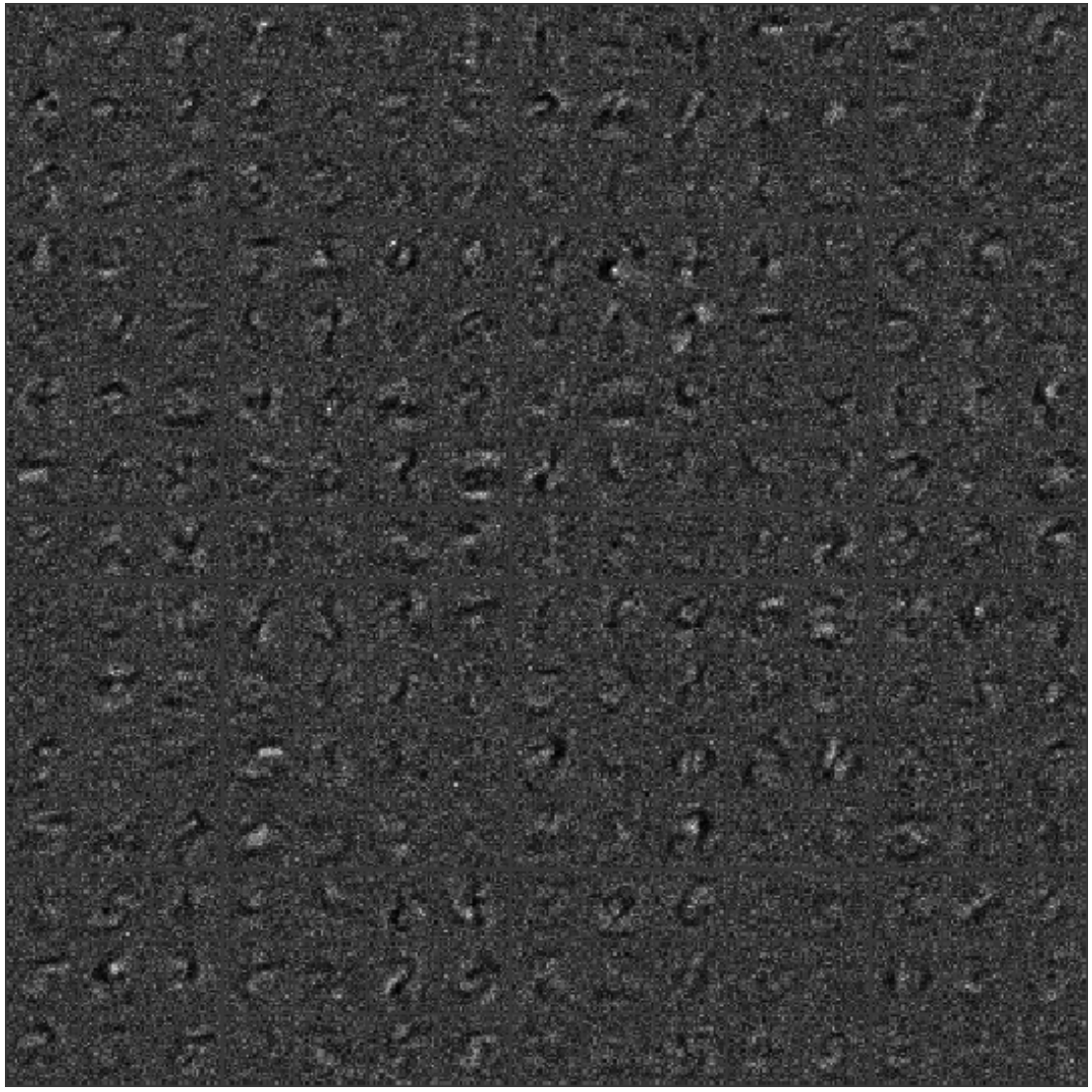


- Adaptation of η_k possible (Adam, Adagrad, Adadelta).

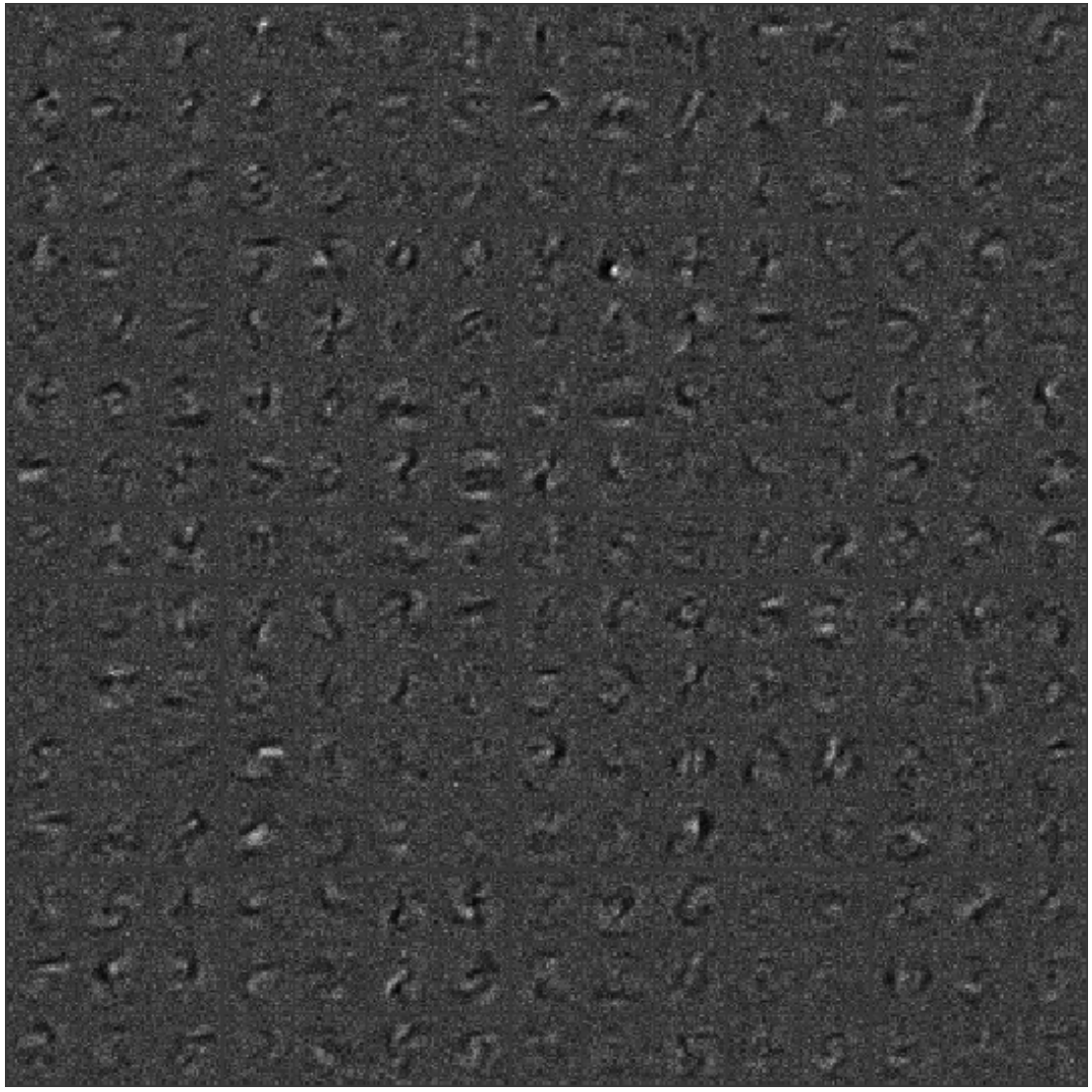
$\mathbf{W}^{(1)}$ after epoch 1



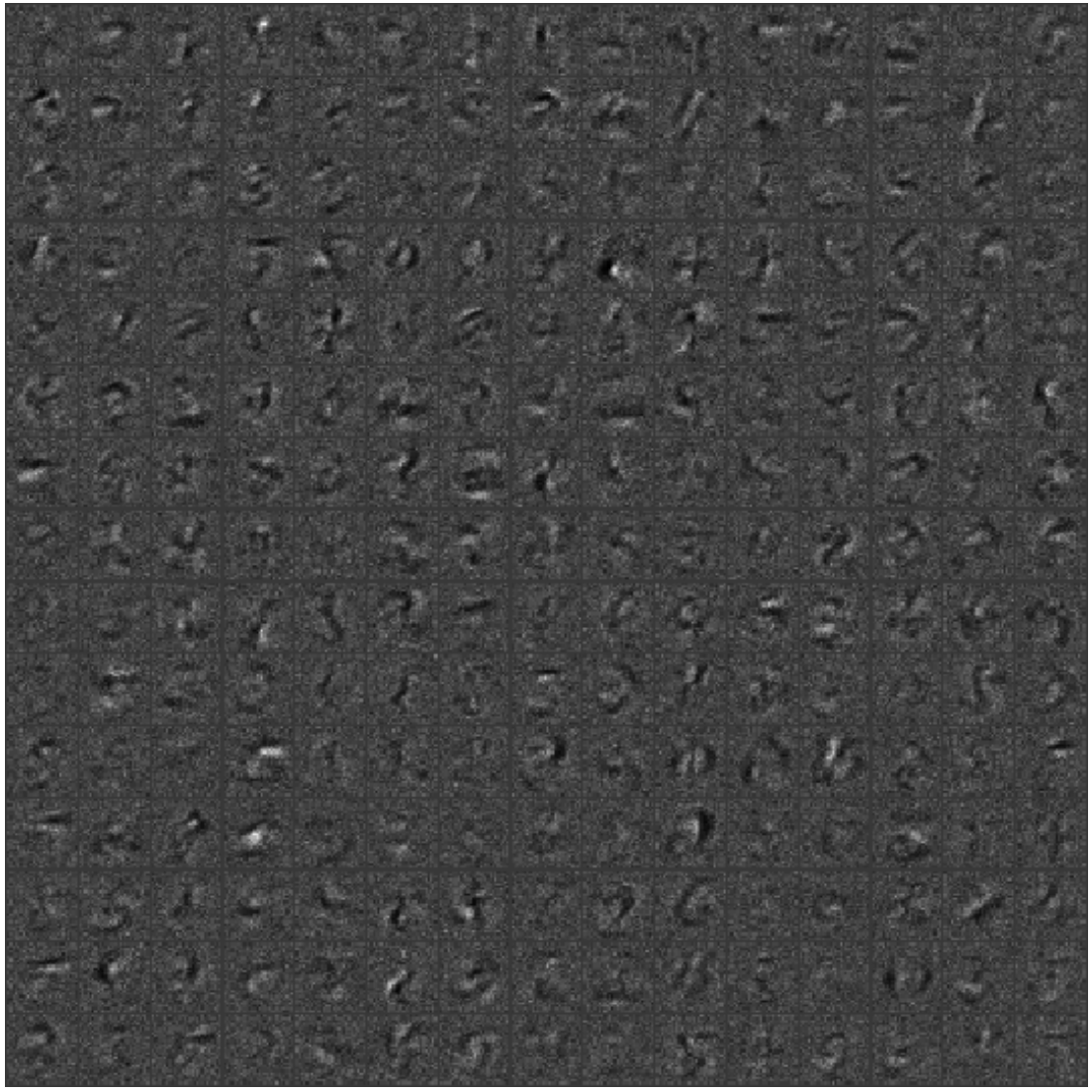
$\mathbf{W}^{(1)}$ after epoch 2



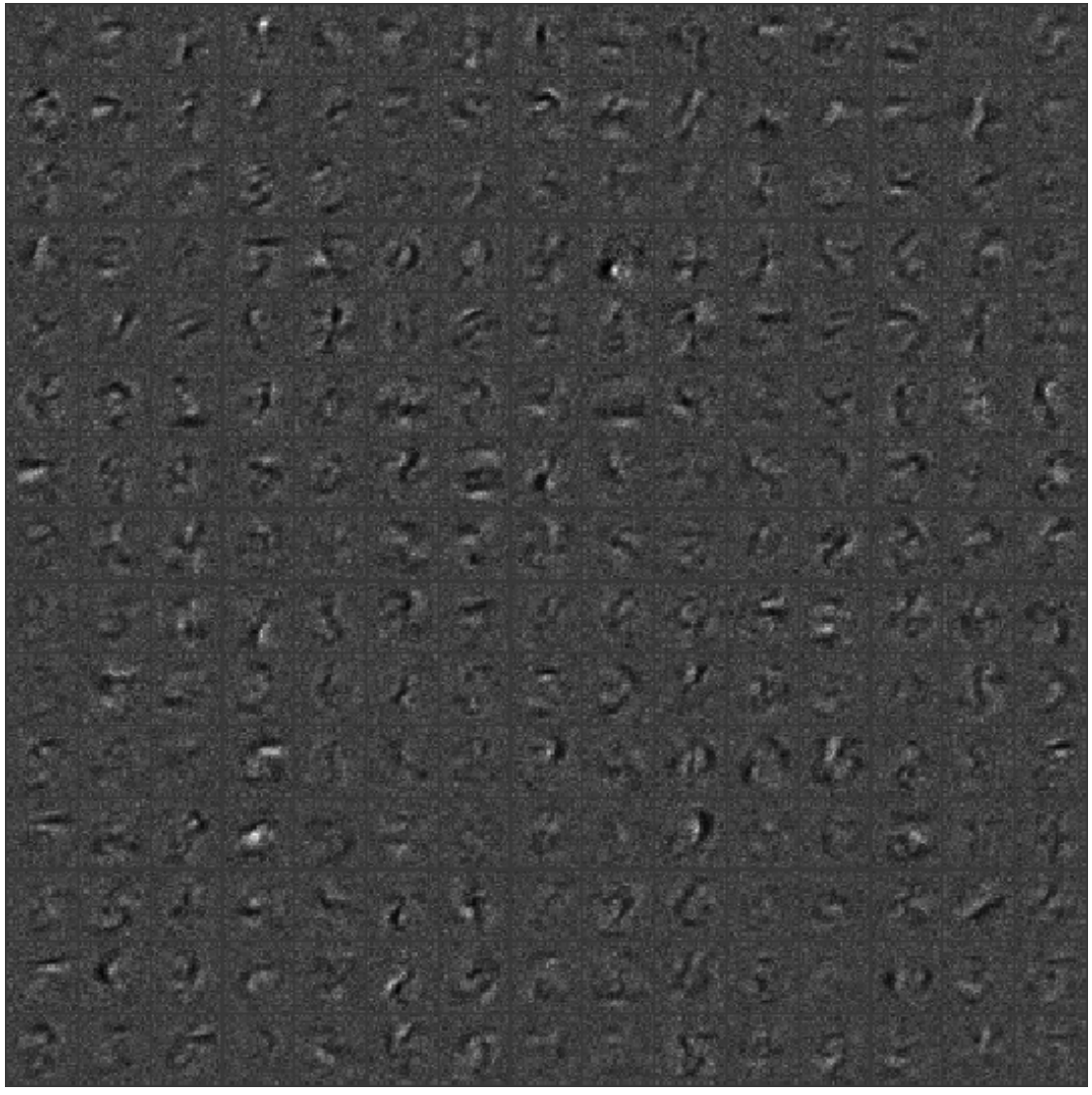
$\mathbf{W}^{(1)}$ after epoch 3



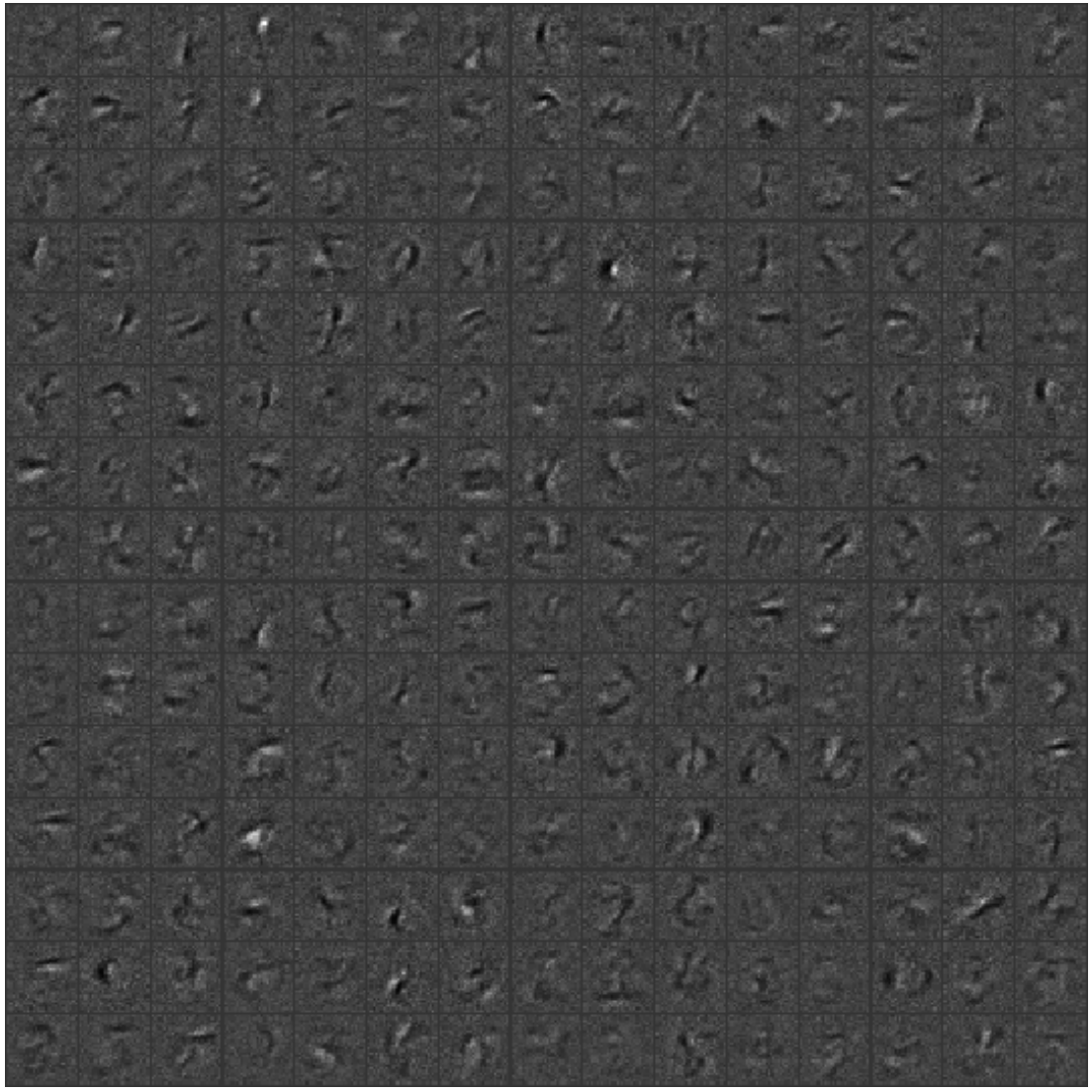
$\mathbf{W}^{(1)}$ after epoch 4



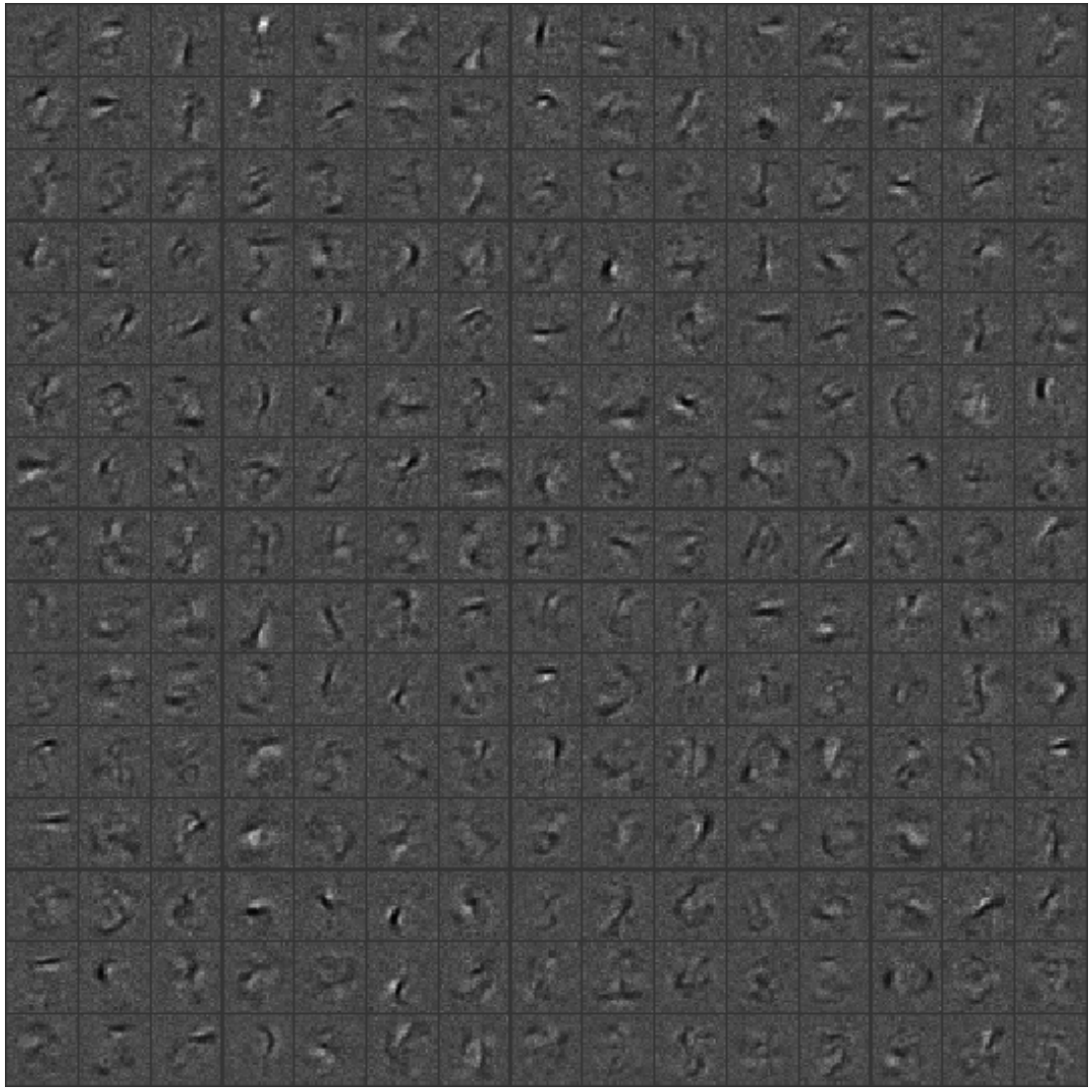
$\mathbf{W}^{(1)}$ after epoch 5



$\mathbf{W}^{(1)}$ after epoch 10

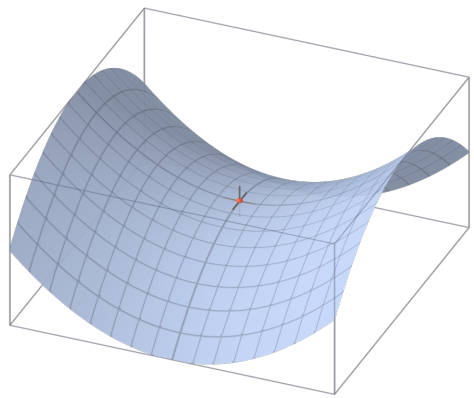


$\mathbf{W}^{(1)}$ after epoch 50 (final)



Definitions

- Learning stops at a critical point (gradient $\mathbf{g} = 0$)
- If all eigenvalues of Hessian **H** are positive it is a local minimum
- If all eigenvalues of Hessian **H** are negative it is a local maximum



- If Hessian has both positive and negative eigenvalues it is a **saddle point**

Theory: Local minima not an issue

(Dauphin et al., 2014, Choromanska et al., 2015)

- Local minima dominate in low-dimensional optimization, but saddle points dominate in high dimensions
- Most local minima are close to the global minimum

- Noisy gradient $\mathbf{g}$ helps in escaping saddle points

Backpropagation, tiny example

$$
\begin{aligned}
y &= w_2 h + \text{noise} \\
h &= w_1 x \\
C &= (y - 1.5)^2 \\
\frac{\partial C}{\partial w_2} &= 2(w_2 w_1 x - 1.5) w_1 x \\
\frac{\partial C}{\partial w_1} &= 2(w_2 w_1 x - 1.5) w_2 x
\end{aligned}
$$

Note a scaling issue: If w_1 is doubled and w_2 is halved,
- output y stays the same.
- system is twice as sensitive to changes in w_2.
- gradient of w_2 is doubled!

Exponential growth/decay forward

- Recall the model

$$\begin{aligned}\mathbf{y} &= \text{softmax}(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}) \\ \mathbf{h}^{(2)} &= \text{relu}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \\ \mathbf{h}^{(1)} &= \text{relu}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})\end{aligned}$$

- Ignoring softmax and biases, we can write

$$y_i = \sum_{j,k,l} \mathbf{1}\left(h_j^{(2)} > 0\right) \mathbf{1}\left(h_k^{(1)} > 0\right) W_{ij}^{(3)} W_{jk}^{(2)} W_{kl}^{(1)} x_l$$

- Exponential growth/decay of forward signals!

Exponential growth/decay backward

- Given indicators $\mathbf{1}(\cdot)$, model is linear

$$y_i = \sum_{j,k,l} \mathbf{1}\left(h_j^{(2)} > 0\right) \mathbf{1}\left(h_k^{(1)} > 0\right) W_{ij}^{(3)} W_{jk}^{(2)} W_{kl}^{(1)} x_l$$

$$\frac{\partial y_i}{\partial x_l} = \sum_{j,k} \mathbf{1}\left(h_j^{(2)} > 0\right) \mathbf{1}\left(h_k^{(1)} > 0\right) W_{ij}^{(3)} W_{jk}^{(2)} W_{kl}^{(1)}$$

- Exponential growth/decay of gradient, too!
- $\Rightarrow$ Scale of initialization important.

Part 3:
Tricks of the trade I

Initialization

- Initialize weights $W_{ij} \sim \sqrt{\frac{4}{n_i+n_j}}\mathcal{N}(0,1)$
- where size of $\mathbf{W}$ is $n_i \times n_j$.

Initialization

- Initialize weights $W_{ij} \sim \sqrt{\frac{4}{n_i+n_j}}\mathcal{N}(0,1)$
- where size of $\mathbf{W}$ is $n_i \times n_j$.
- Outline of derivation:
- Assume independent signals.

Initialization

- Initialize weights $W_{ij} \sim \sqrt{\frac{4}{n_i+n_j}}\mathcal{N}(0,1)$
- where size of $\mathbf{W}$ is $n_i \times n_j$.
- Outline of derivation:
- Assume independent signals.
- Retain variance forward: $\sqrt{\frac{2}{n_j}}$
- Retain variance backward: $\sqrt{\frac{2}{n_i}}$
- (2 is from relu indicators being on half the time.)

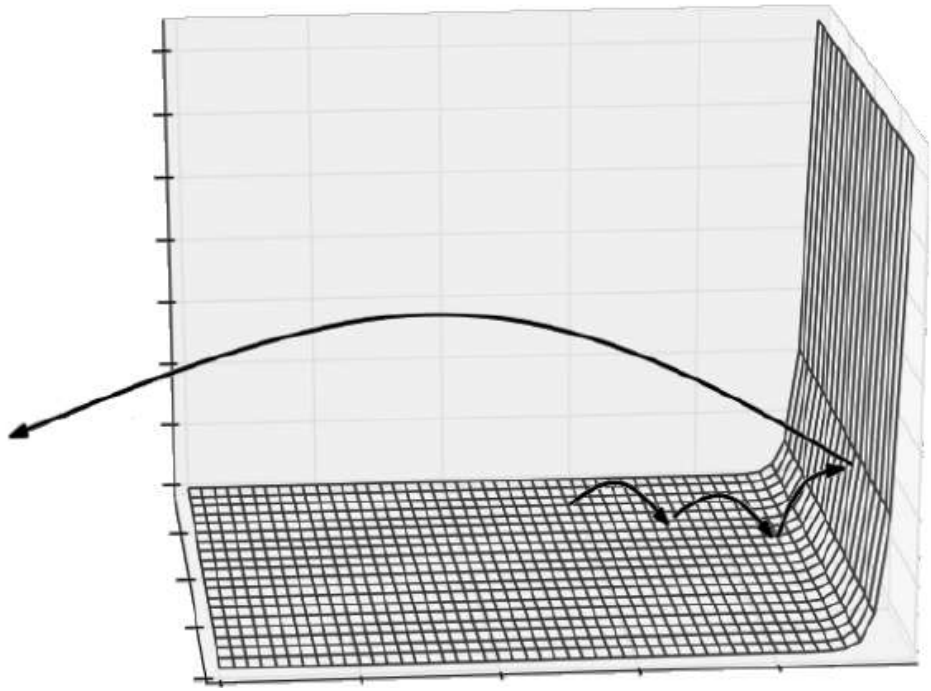
Initialization

- Initialize weights $W_{ij} \sim \sqrt{\frac{4}{n_i+n_j}}\mathcal{N}(0,1)$
- where size of $\mathbf{W}$ is $n_i \times n_j$.
- Outline of derivation:
- Assume independent signals.
- Retain variance forward: $\sqrt{\frac{2}{n_j}}$
- Retain variance backward: $\sqrt{\frac{2}{n_i}}$
- (2 is from relu indicators being on half the time.)
- Strike a balance between the two (Glorot and Bengio, 2010).

Initialization

- Initialize weights $W_{ij} \sim \sqrt{\frac{4}{n_i+n_j}}\mathcal{N}(0,1)$
- where size of $\mathbf{W}$ is $n_i \times n_j$.
- Outline of derivation:
- Assume independent signals.
- Retain variance forward: $\sqrt{\frac{2}{n_j}}$
- Retain variance backward: $\sqrt{\frac{2}{n_i}}$
- (2 is from relu indicators being on half the time.)
- Strike a balance between the two (Glorot and Bengio, 2010).
- (Other ideas, relevant for RNNs: sparse initialization (Martens, 2010), orthogonal initialization (Saxe et al., 2014))

Gradient clipping



- Highly nonlinear model: Gradient update can catapult parameters very far.
- Heuristic: Clip the magnitude of the gradient.
- Figure from (Pascanu, 2014)

Batch normalization (Ioffe and Szegedy, 2015)

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

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

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

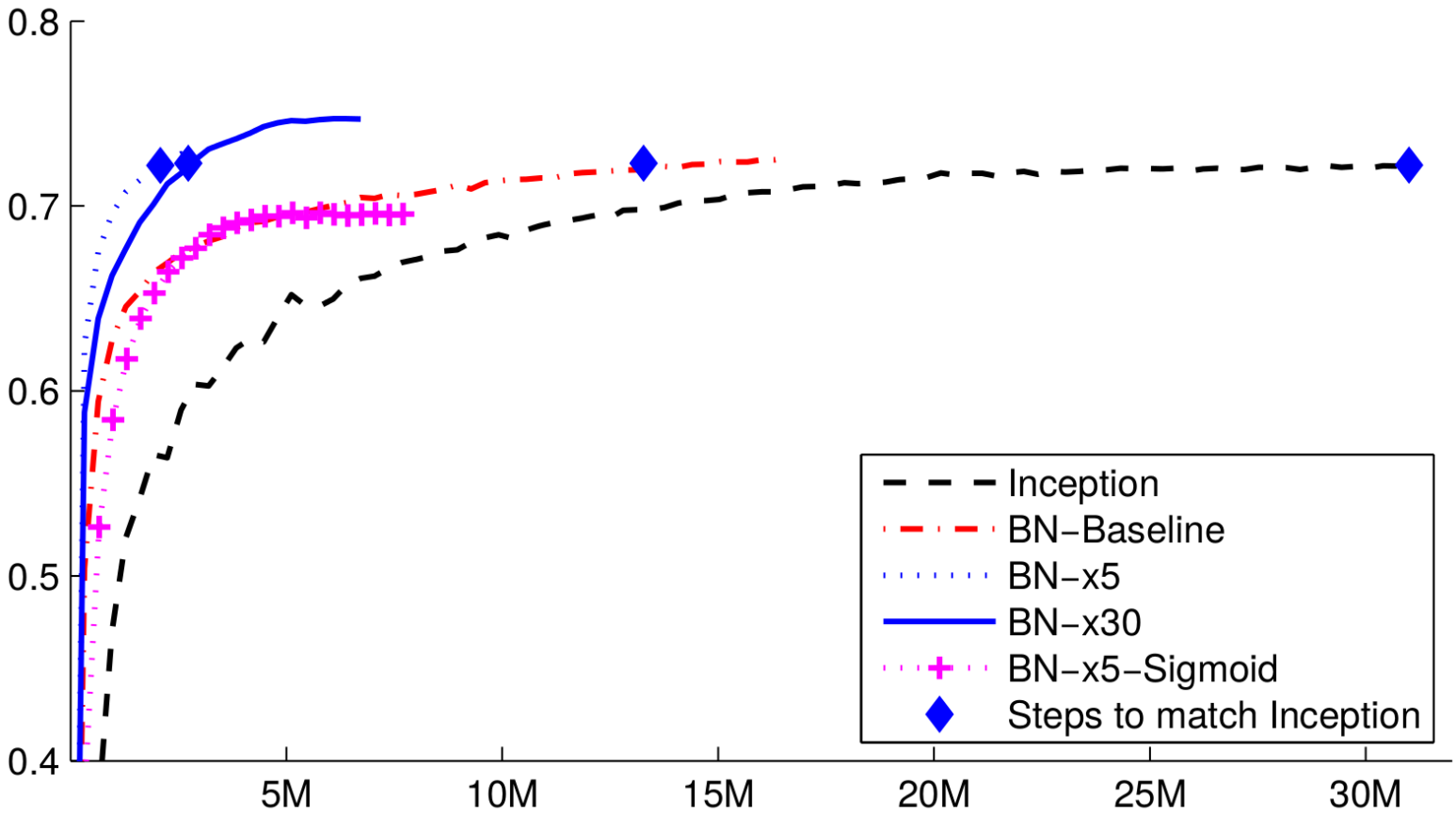
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

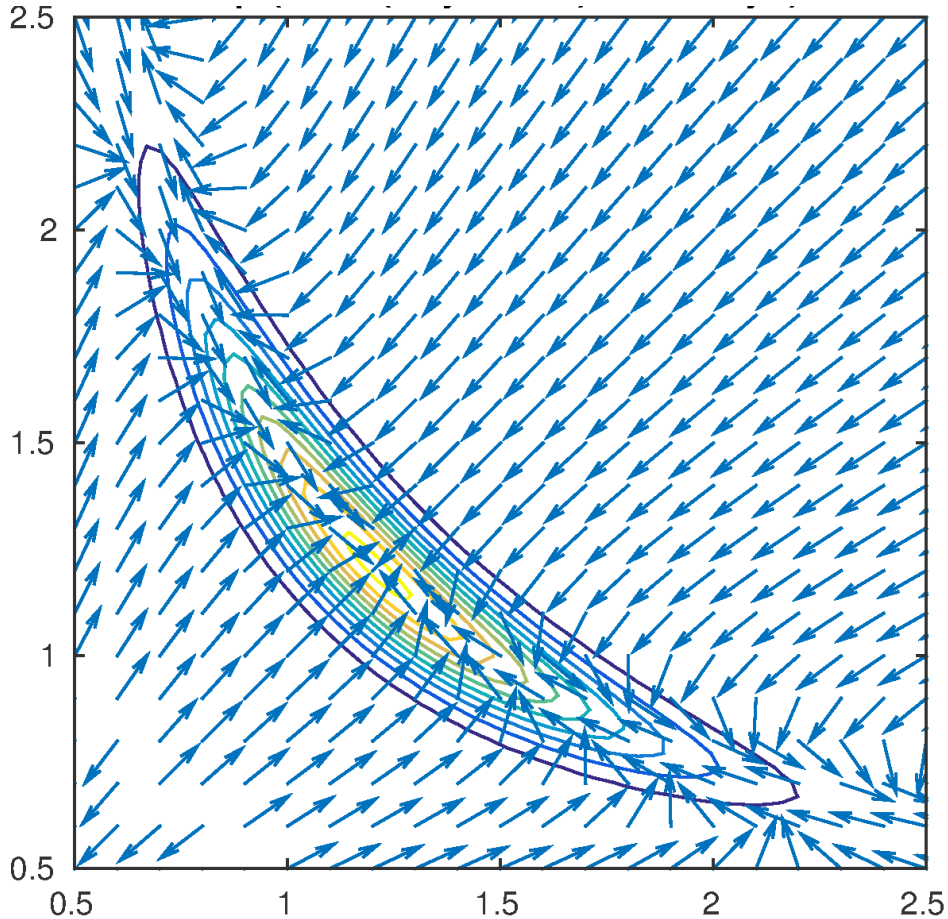
Batch normalization (Ioffe and Szegedy, 2015)

- Improves learning (BN-baseline vs. Inception)
- Allows bigger learning rates (5x and 30x)

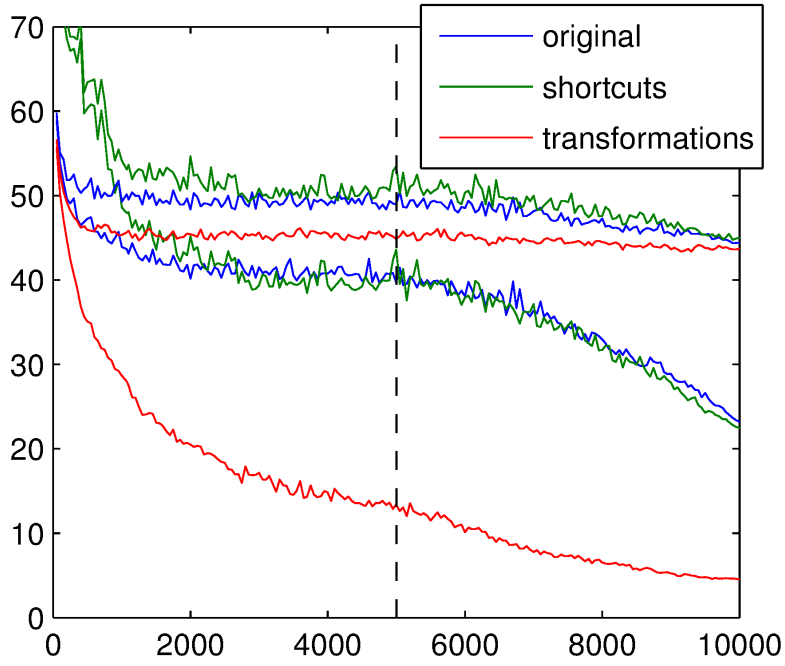


Why does it work? Recall Newton's

$$\theta_{k+1} = \theta_k - \mathbf{H}_k^{-1}\mathbf{g}_k, \quad \mathbf{H} = \begin{pmatrix} \frac{\partial^2 C}{\partial\theta_1\partial\theta_1} & \cdots & \frac{\partial^2 C}{\partial\theta_1\partial\theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 C}{\partial\theta_n\partial\theta_1} & \cdots & \frac{\partial^2 C}{\partial\theta_n\partial\theta_n} \end{pmatrix}$$

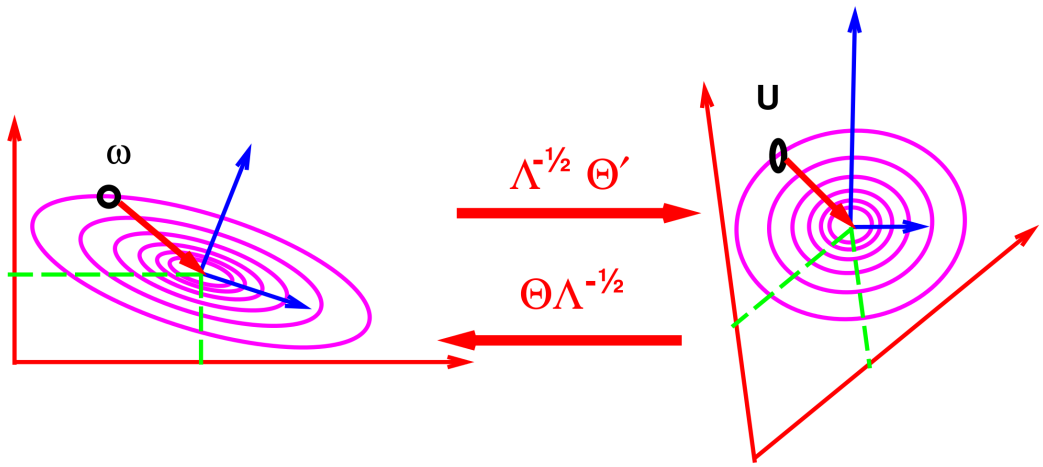


Why does it work? (Raiko et al., 2012)



- Transformations do not change the model, but the optimisation
- Hessian **H** is closer to a diagonal
- Traditional gradient is thus closer to Newton's and parameter updates are more independent

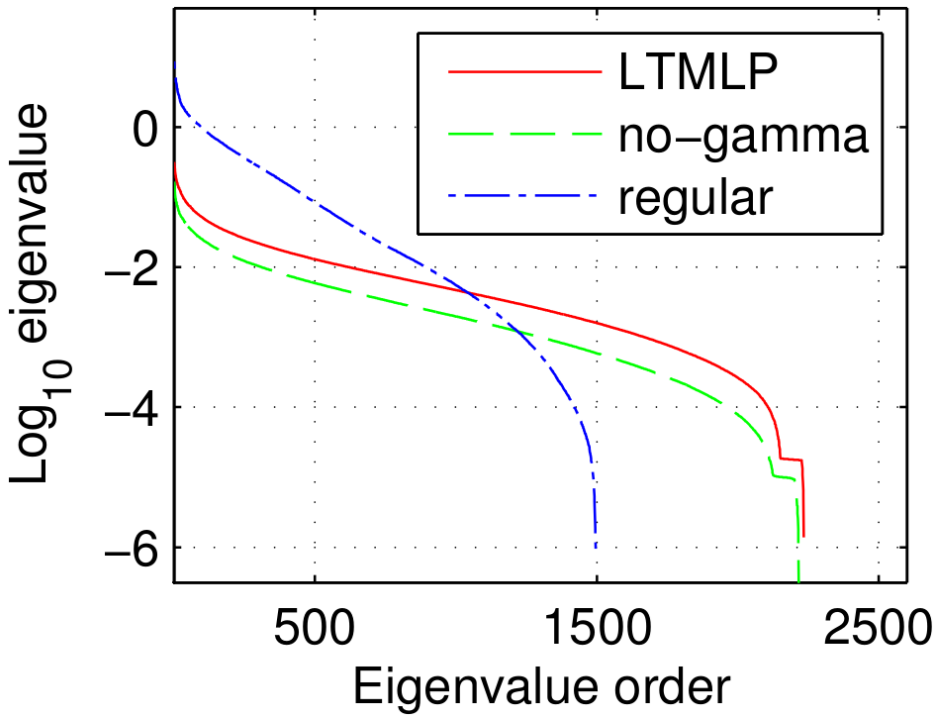
Eigenvalues of the Hessian **H** (LeCun et al., 1998)



- Eigenvectors corresponds to (update) directions
- In Newton's method, each direction has its own learning rate: inverse of the eigenvalue
- Some eigenvalues can be negative: Newton's method points the wrong way

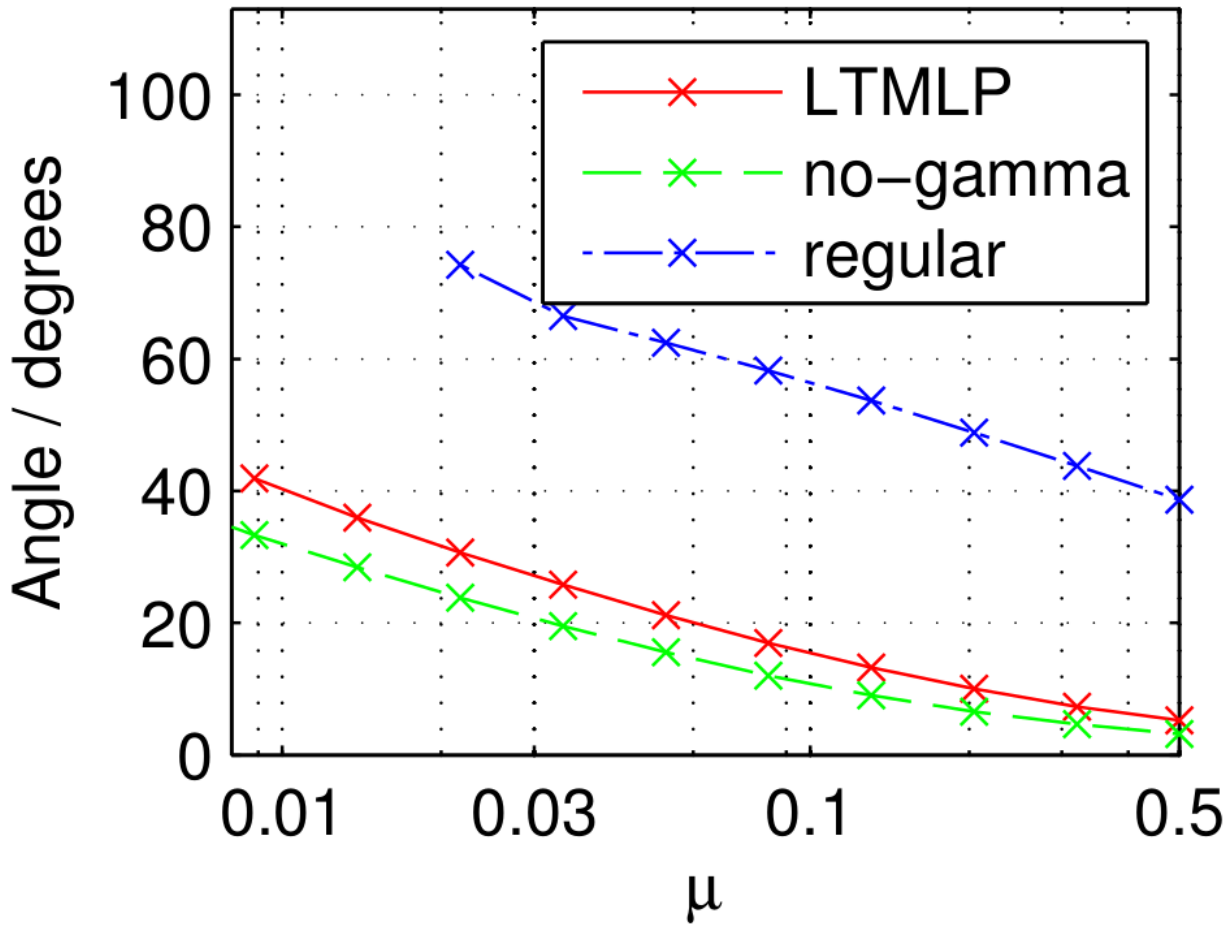
Why does it work? (Vatanen et al., 2013)

- Analysis of eigenvalues of Hessian in 2600-parameter model
- Curvature is much more even with transformations



Why does it work? (Vatanen et al., 2013)

Angle between gradient and second order update:
$\theta_{k+1} = \theta_k - (\mathbf{H}_k + \mu\mathbf{I})^{-1}\mathbf{g}_k$,



References

- S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 1970.
- Polyak, B.T. Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics, 4(5):1–17, 1964.
- Kingma, D. and Ba, J. (2015). ADAM: A method for stochastic optimization. In the International Conference on Learning Representations (ICLR), San Diego. arXiv:1412.6980.
- Dauphin, Pascanu, Gulcehre, Cho, Ganguli, and Bengio. Identifying and attacking the saddle point problem in high dimensional non-convex optimization. In NIPS 2014.
- Choromanska, Henaff, Mathieu, Ben Arous, and LeCun. The Loss Surface of Multilayer Nets. In AISTATS 2015.
- Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International conference on artificial intelligence and statistics. 2010.
- Martens, J. Deep learning via Hessian-free optimization. In ICML, 2010.
- Saxe, Andrew M., James L. McClelland, and Surya Ganguli. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." In ICLR 2014.
- Pascanu, R., "On Recurrent and Deep Neural Networks", PhD thesis, University of Montreal, 2014.
- Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." Proc. of ICML, 2015.
- T. Raiko, H. Valpola, and Y. LeCun. Deep Learning Made Easier by Linear Transformations in Perceptrons. Proc. of AISTATS, 2012.
- Y. LeCun, L. Bottou, G. Orr and K. Muller: Efficient BackProp, in Orr, G. and Muller K. (Eds), Neural Networks: Tricks of the trade, Springer, 1998.
- T. Vatanen, T. Raiko, H. Valpola, and Y. LeCun. Pushing Stochastic Gradient towards Second-Order Methods - Backpropagation Learning with Transformations in Nonlinearities. In Proc. ICONIP, 2013.

Thanks!
Ole Winther