ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

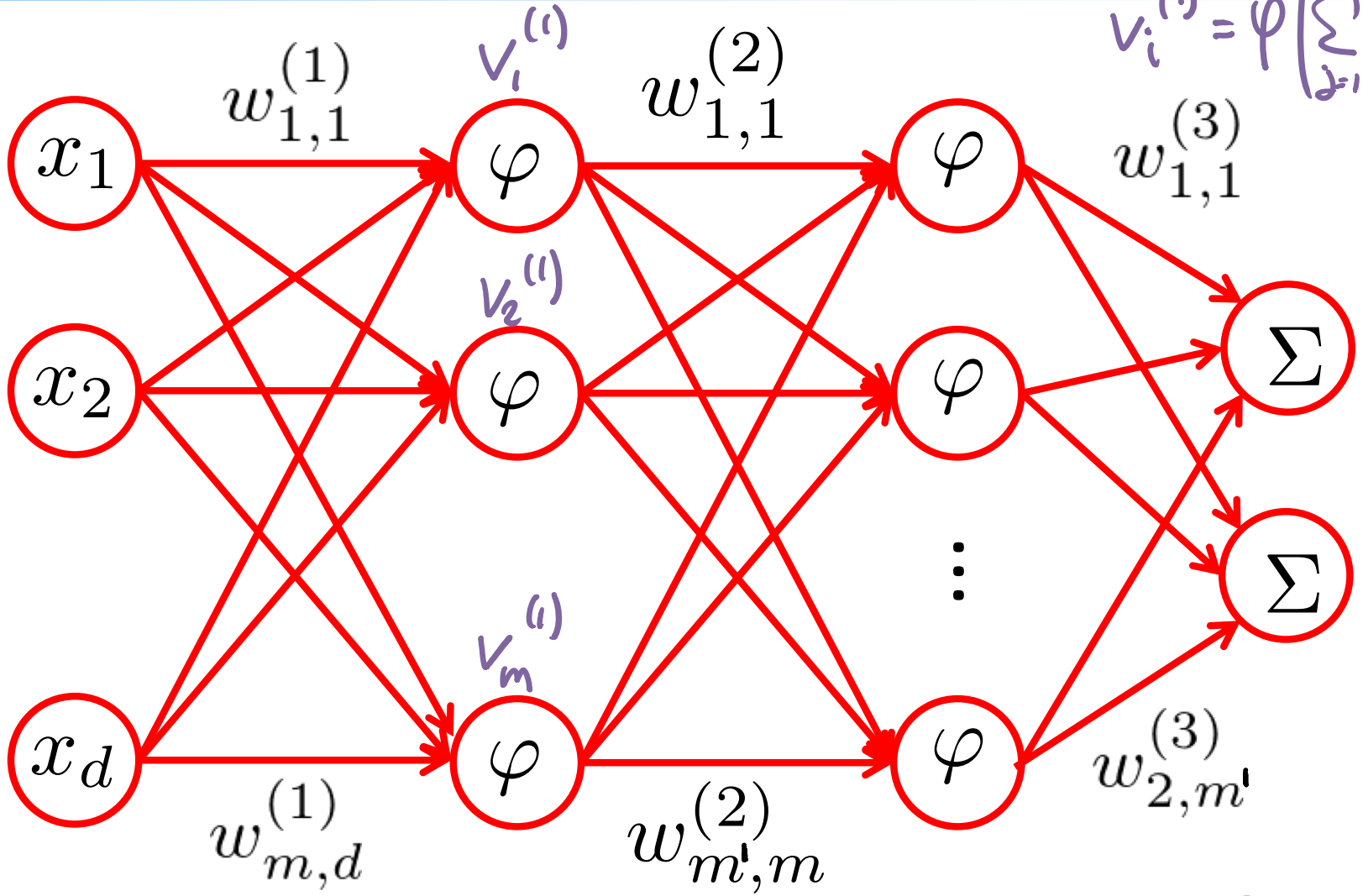
inf | Informatik
Computer Science

# Introduction to Machine Learning

## Neural networks (continued)

Prof. Andreas Krause
Learning and Adaptive Systems (las.ethz.ch)

# Computational graph



# How can we train the weights?

- Given data set $D = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$

  want to optimize weights $\mathbf{W} = (\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)})$
- How do we measure and optimize goodness of fit?

➔ Apply loss function (e.g., Perceptron loss, multi-class hinge loss, square loss, etc.) to output

$$\ell(\mathbf{W}; \mathbf{y}, \mathbf{x}) = \ell\big(\mathbf{y} - f(\mathbf{x}, \mathbf{W})\big)$$

➔ Then optimize the weights to minimize loss over D

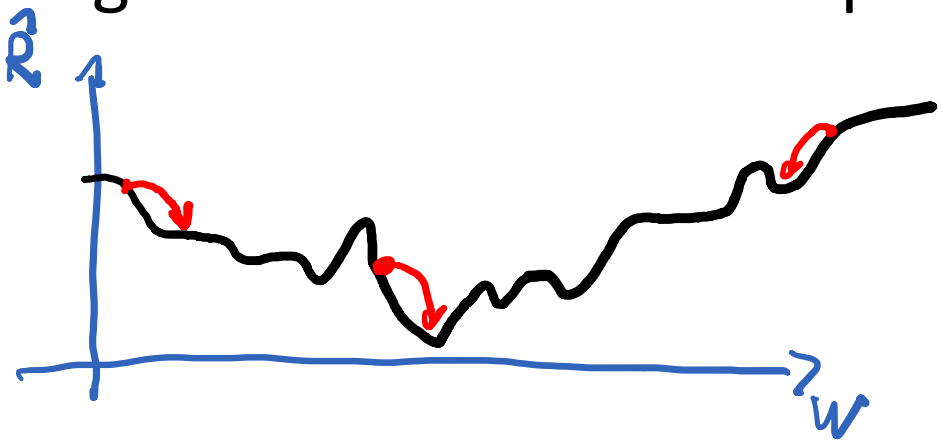
$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{x}_i, y_i)$$

# How do we optimize over weights?

- Want to do Empirical Risk Minimization

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{x}_i, y_i)$$

- I.e., jointly optimize over all weights for all layers to minimize loss over the training data
- This is in general a non-convex optimization problem




- Nevertheless, can try to find a local optimum

# Stochastic gradient descent for ANNs

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{x}_i, y_i)$$

- Initialize weights $\mathbf{W}$
- For t = 1,2,...
  - Pick data point $(\mathbf{x}, \mathbf{y}) \in D$ uniformly at random
  - Take step in negative gradient direction

$$\mathbf{W} \leftarrow \mathbf{W} - \eta_t \nabla_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$$

# Backpropagation: Matrix form

- For the output layer
  - Compute „error“ $\delta^{(L)} = \mathbf{l}'(\mathbf{f}) = [l'(f_1), \ldots, l'(f_p)]$
  - Gradient: $\nabla_{\mathbf{W}^{(L)}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x}) = \delta^{(L)} \mathbf{v}^{(L-1)T}$
- For each hidden layer $\ell = L-1 : -1 : 1$
  - Compute „error“ $\delta^{(\ell)} = \varphi'(\mathbf{z}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)T} \delta^{(\ell+1)})$

    pointwise multiplication
  - Gradient $\nabla_{\mathbf{W}^{(\ell)}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x}) = \delta^{(\ell)} \mathbf{v}^{(\ell-1)T}$

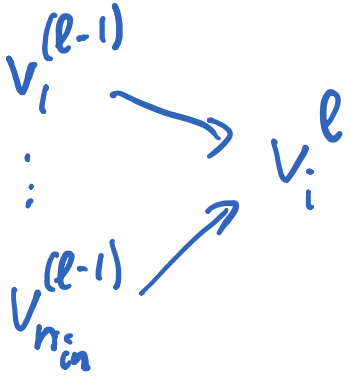
# Some questions

- How can we compute the gradient?
- How should we initialize the weights?
- When should we terminate?
- How do we choose parameters (number of units/layers/activation functions/learning rate/ ...)?
- What about overfitting?

# Initializing weights

- Since optimization problem is non-convex, initialization matters!
- What happens if we pick inappropriate weights?

# Propagation of variance (ReLU)




$$z_i^{(l)} = \sum_{j=1}^{n_{in}} w_{ij}\, v_j^{(l-1)} \quad \Rightarrow \quad v_i^{(l)} = \varphi\left(z_i^{(l)}\right)$$

$\varphi(z) := \max(0, z)$

Assume: $\mathbb{E}[x_i] = \mathbb{E}[v_i^{(0)}] = 0$, $\mathrm{Var}[x_i] = \mathrm{Var}[v_i^{(0)}]$

Further: $x_1 \dots x_d$ independ.

Also: $w_{ij} \sim N(0, \sigma^2)$ independ

Sps $X, Y$ indep., $Y$ mean 0

(a) if $Y$ mean 0: $\mathrm{Var}[XY] = \mathrm{Var}[X]\,\mathrm{Var}[Y]$

(b) if $Y$ mean $\neq 0$: $\mathrm{Var}[XY] = \mathrm{Var}[X]\,\mathbb{E}[Y^2]$

For ReLU and symmetric $Y$:

$\mathbb{E}[\varphi(Y)^2] = \frac{1}{2}\mathrm{Var}(Y)$

$$\mathbb{E}[z_i^{(l)}] = \mathbb{E}\left[\sum_{j=1}^{n_{in}} w_{ij}\, x_j\right] = \sum_j \mathbb{E}[w_{ij}\, x_j] = \sum_j \mathbb{E}[w_{ij}]\,\mathbb{E}[x_j] = 0$$

$$\mathrm{Var}[z_i^{(l)}] = \sum_j \mathrm{Var}[w_{ij}\, x_j] \overset{(a)}{=} \sum_j \mathrm{Var}(w_{ij})\,\mathrm{Var}(x_j) = n \cdot \sigma^2$$

First layer

$$\mathrm{Var}[z_i^{(l)}] = \sum_j \mathrm{Var}[w_{ij}\, v_j^{(l-1)}] = \sum_j \underbrace{\mathrm{Var}(w_{ij})}_{\sigma^2} \underbrace{\mathbb{E}[v_j^{(l-1)2}]}_{\frac{1}{2}\mathrm{Var}(z_j^{(l-1)})} = \boxed{\frac{1}{2} n_{in}\, \sigma^2 := 1} \Rightarrow \sigma^2 = \frac{2}{n_{in}}$$

# Initializing weights

- Goal: Keep variance of weights approximately constant across layers to avoid vanishing and exploding gradients

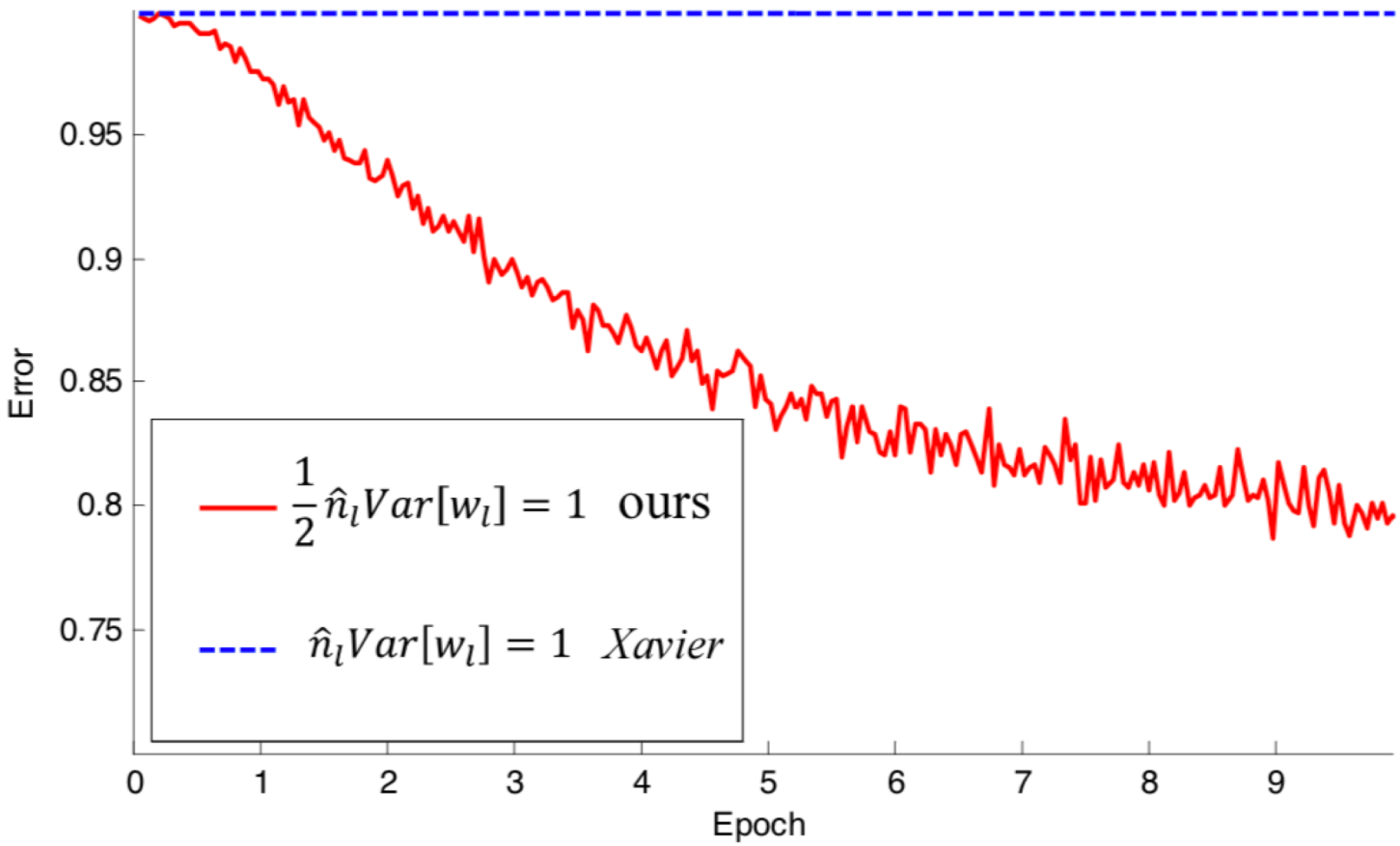
- Usually random initialization works well, e.g.,

Glorot (tanh): $w_{i,j} \sim \mathcal{N}(0, 1/(n_{in}))$

$w_{i,j} \sim \mathcal{N}(0, 2/(n_{in} + n_{out}))$

He (ReLU): $w_{i,j} \sim \mathcal{N}(0, 2/n_{in})$

# Effects of (im)proper initialization




[He, Zhang, Ren, Sun 2015]

# Learning rate

- To implement SGD rule

$$\mathbf{W} \leftarrow \mathbf{W} - \eta_t \nabla_{\mathbf{W}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$$

  need to choose learning rate $\eta_t$

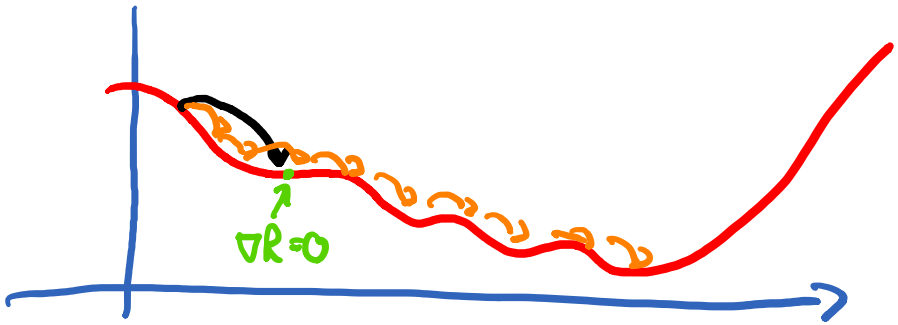
- Usually good idea to start with fixed (small) learning rate, and decrease slowly after some iterations, e.g.,

$$\eta_t = \min(0.1, 100/t)$$

- Often use piecewise constant learning rate „schedules“ (i.e., drop after some # of epochs)

# Learning with momentum

- Common extension to training with SGD
- Can help to escape local minima
- *Idea*: Move not only into direction of gradient, but also in direction of last weight update




- Updates:

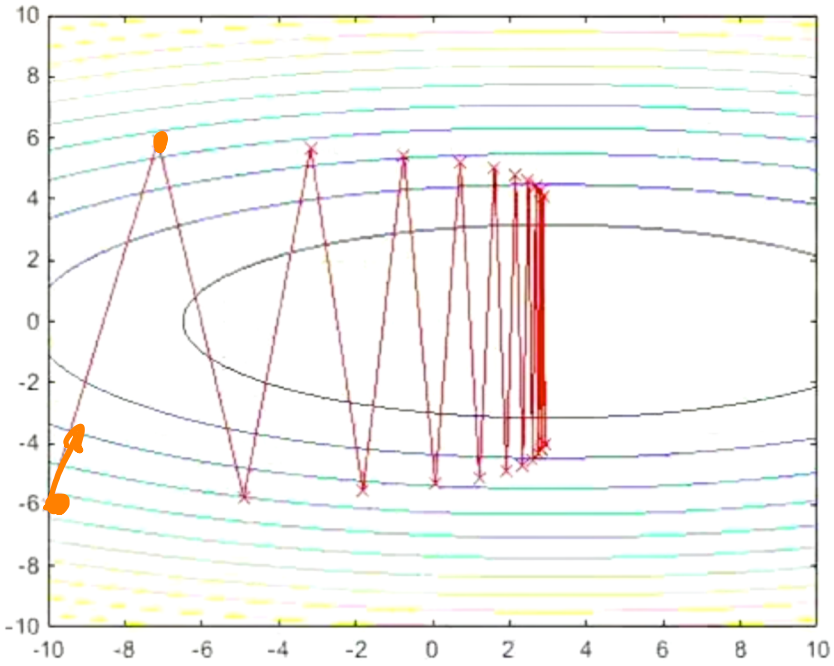
Momentum < 1

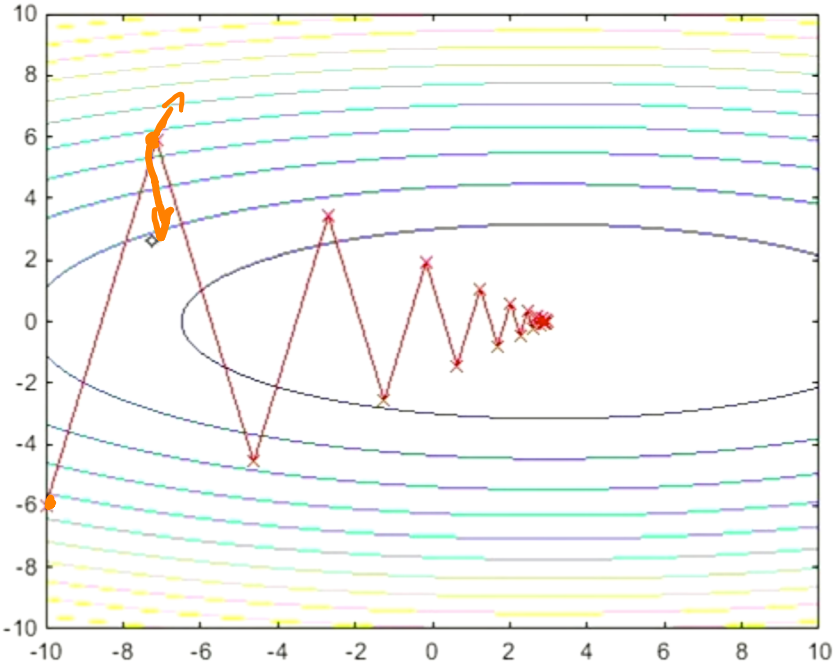
$$a \leftarrow m \cdot a + \eta_t \nabla_{\mathbf{w}} \ell(\mathbf{W}; \mathbf{y}, \mathbf{x})$$

$$\mathbf{W} \leftarrow \mathbf{W} - a$$

# Learning with momentum

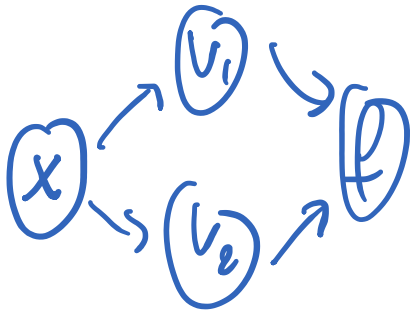
- Preventing oscillation



# Weight-space symmetries

- Multiple distinct weights compute the same predictions




$$v_i = \varphi(w_i x)$$

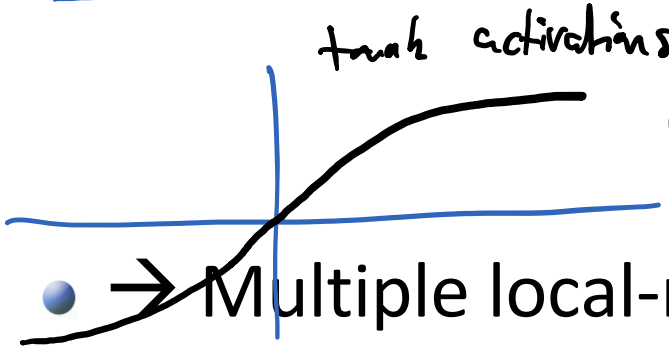
$$f = w_1' v_1 + w_2' v_2$$

$w = [w_i, w_i']_i$ , $\tilde{w} = [\tilde{w}_i, \tilde{w}_i']_i$

Sps. $\tilde{w}_2 = w_1$ , $\tilde{w}_1 = w_2$ , $\tilde{w}_1' = w_2'$ , $\tilde{w}_2' = w_1'$

Then : $f(x; \tilde{w}) = f(x; w)$

---




$$\varphi(z) = -\varphi(-z)$$

$\tilde{w}_1 = -w_1$ , $\tilde{w}_2 = -w_2$ , $\tilde{w}_1' = -w_1'$ , $\tilde{w}_2' = -w_2'$

$\Rightarrow f(x; \tilde{w}) = f(x; w)$

- → Multiple local-minima can be equivalent in terms of input-output mapping

# Avoiding overfitting

- Neural networks have many parameters
  ➔ Potential danger of overfitting
- Countermeasures
  - Early stopping: Don‘t run SGD until convergence
  - Regularization: Add penalty term to keep weights small

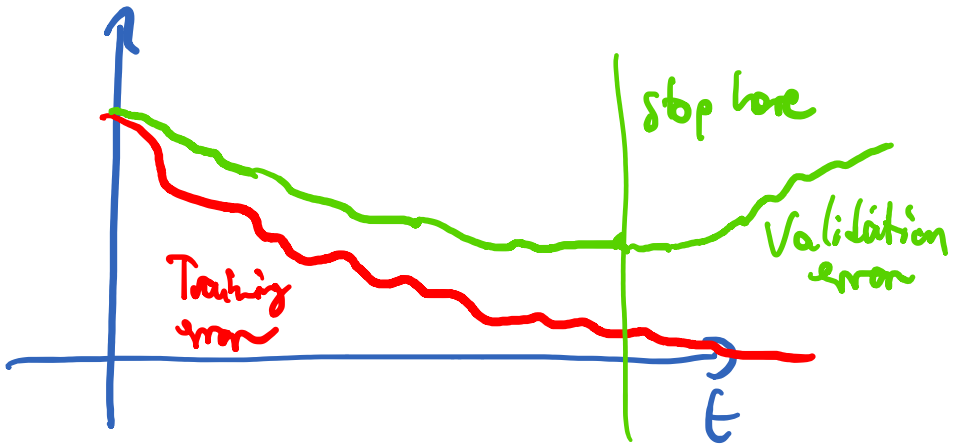
$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \sum_{i=1}^{n} \ell(\mathbf{W}; \mathbf{x}_i, y_i) + \lambda ||W||_F^2$$

  - „Dropout“

# Early stopping

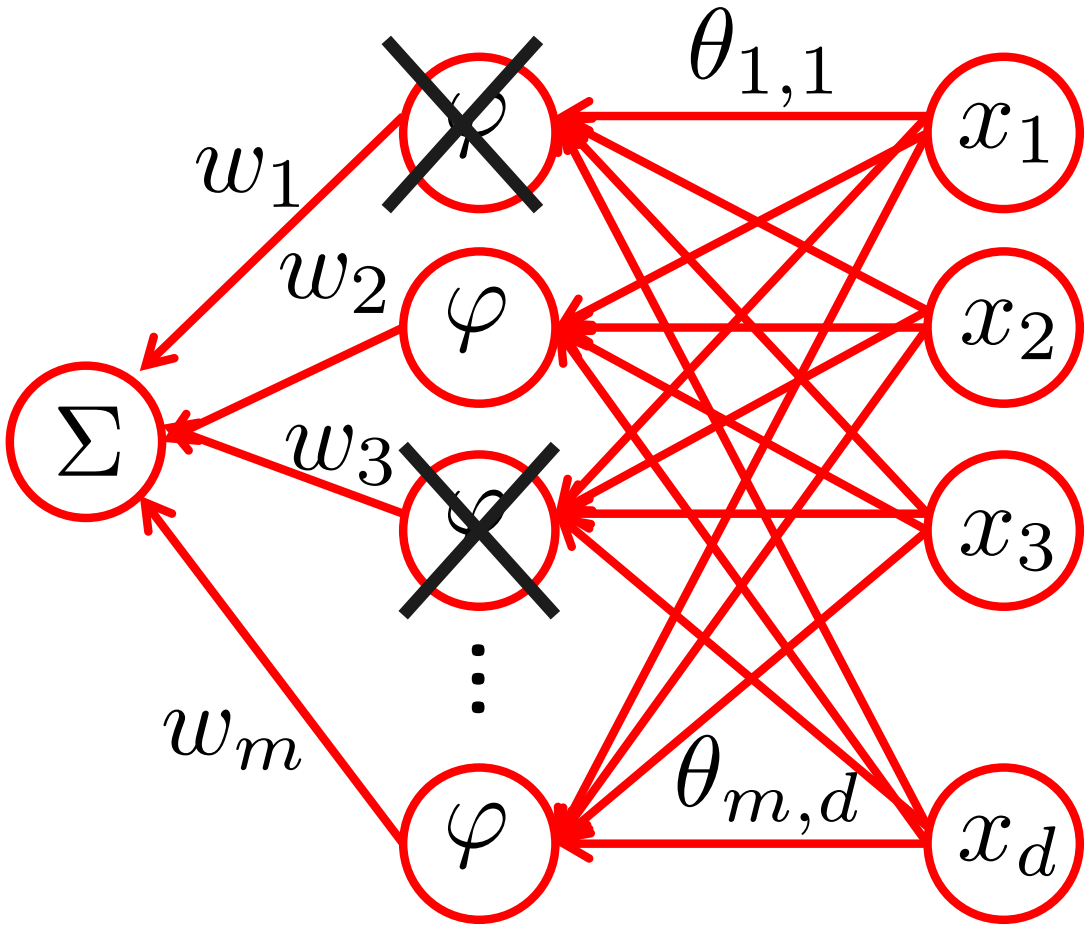
- In general, might not want to run training until weights converge ➔ Overfitting!

- One possibility:
  - Monitor prediction performance on validation set;
  - stop training once validation error starts to increase



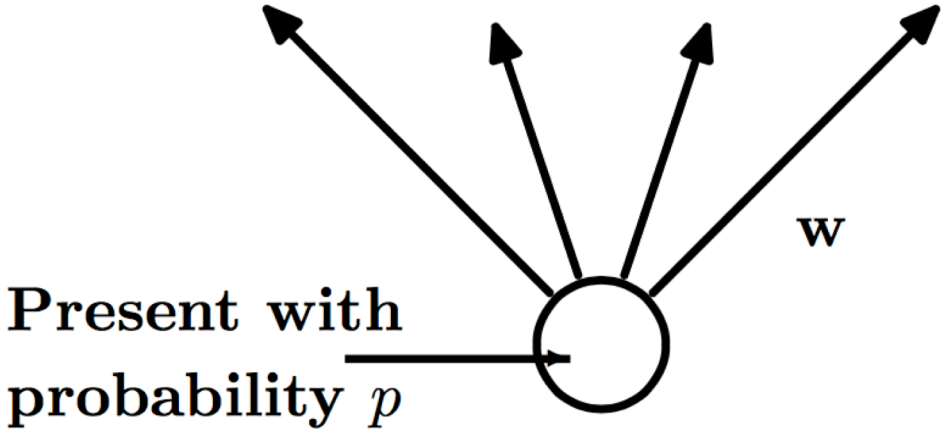
# Dropout regularization

- **Key idea**: randomly ignore („drop out“) hidden units during each iteration of SGD with probability 1/2

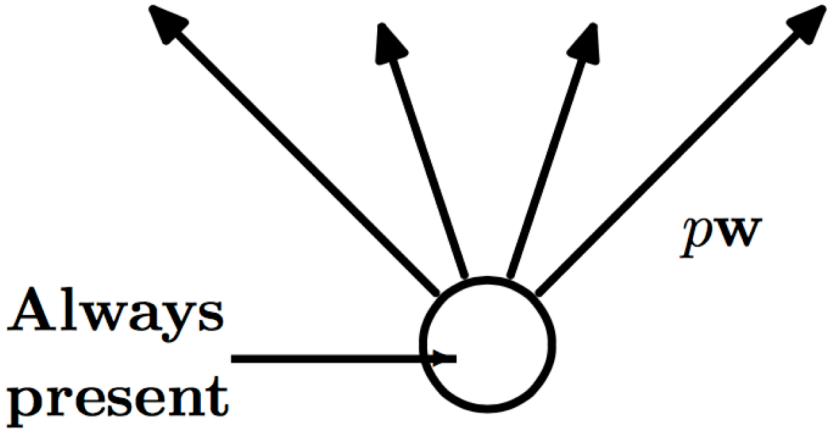



- After training, half the weights to compensate

# Dropout: Train vs Test




(a) At training time

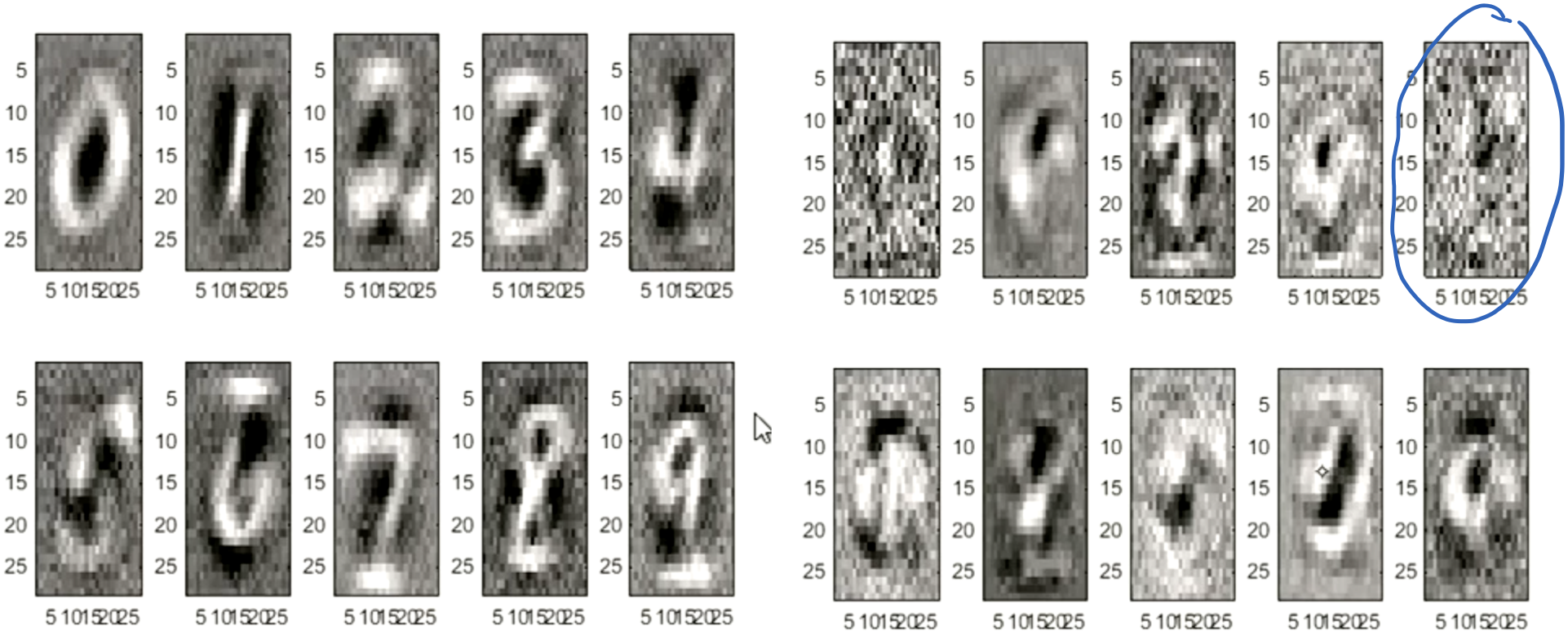



(b) At test time

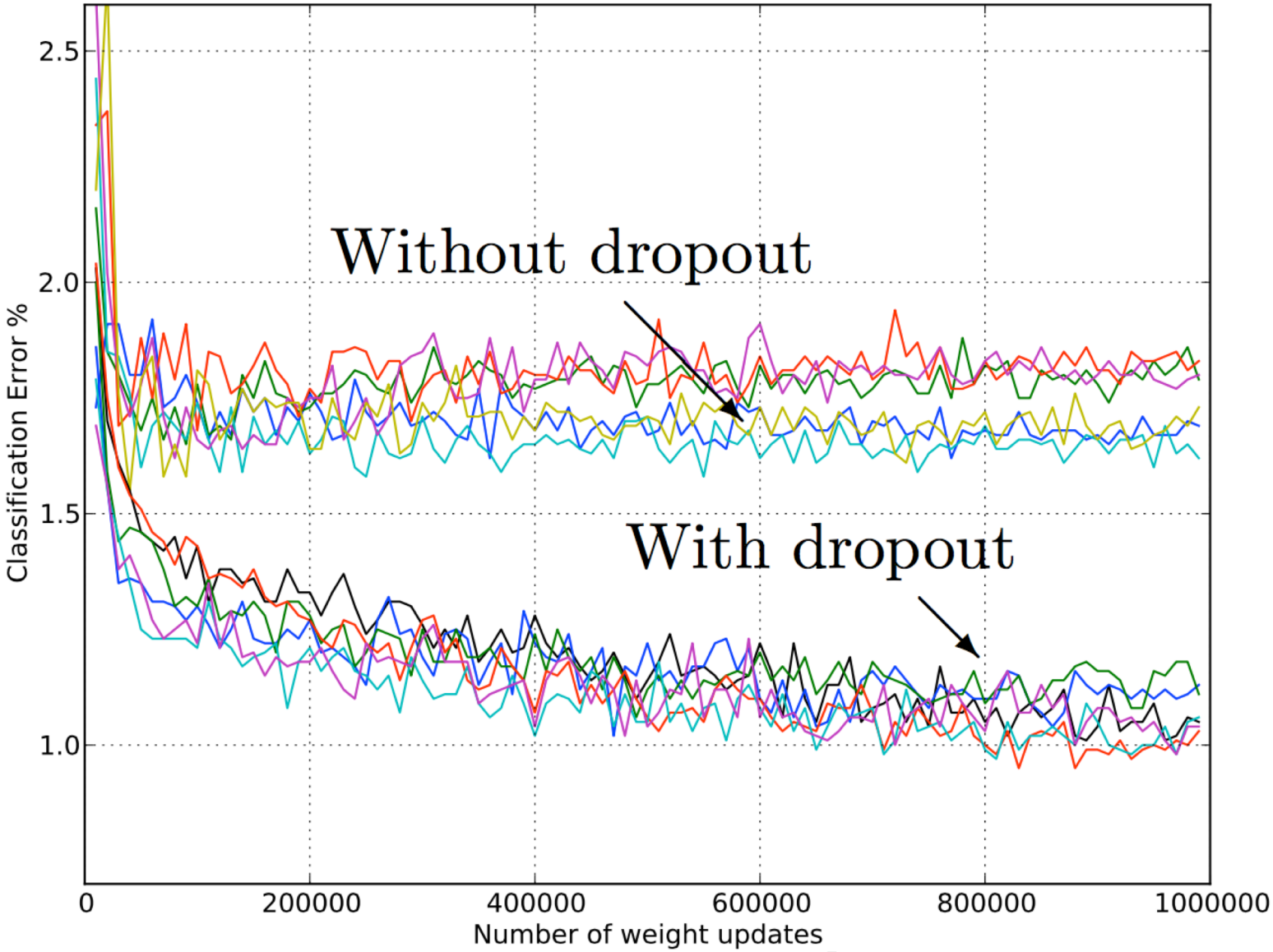
$$\mathbb{E}[z] = \mathbb{E}[w^T x]$$
$$= \mathbb{E}\left[\sum_i w_i x_i [i \text{ is selected}]\right]$$
$$= \sum_i w_i x_i \, p = p \cdot w^T x = (pw)^T x$$

[Srivastava et al JMLR '14]

# Ovefitting illustration

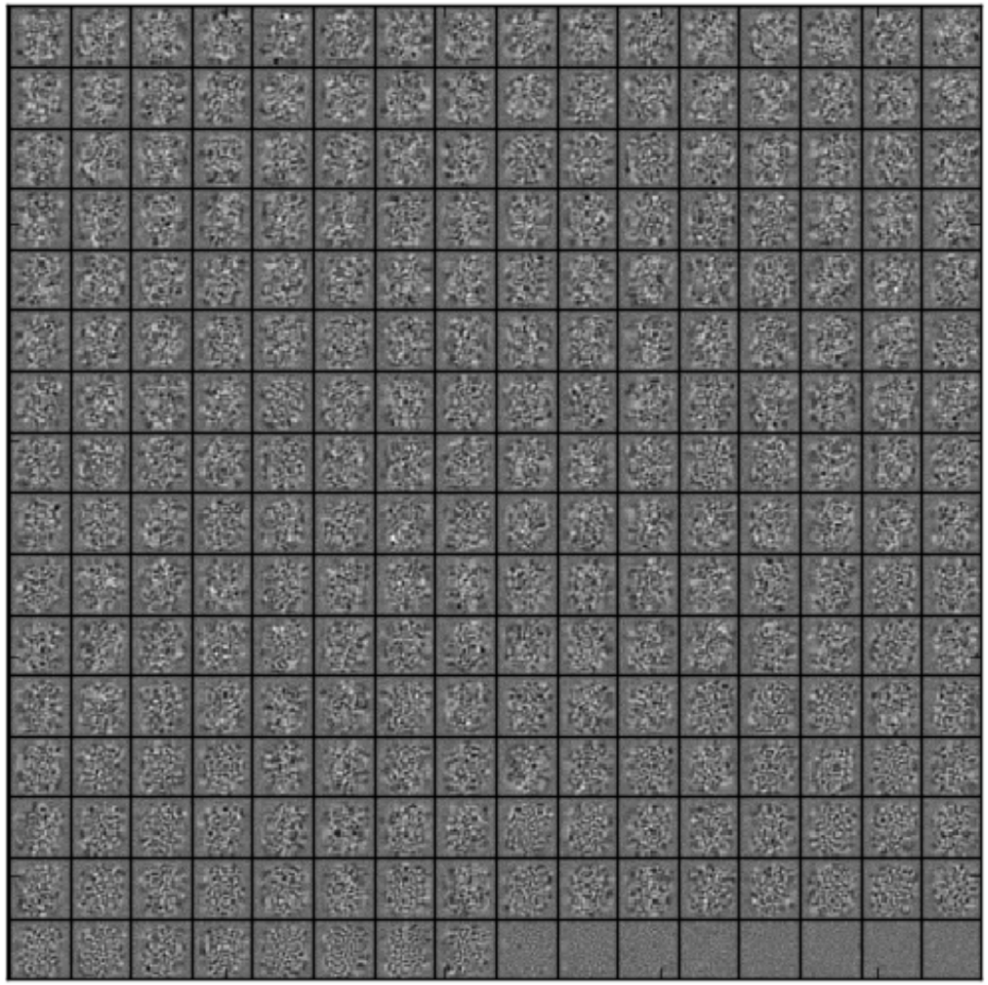


# Dropout Results

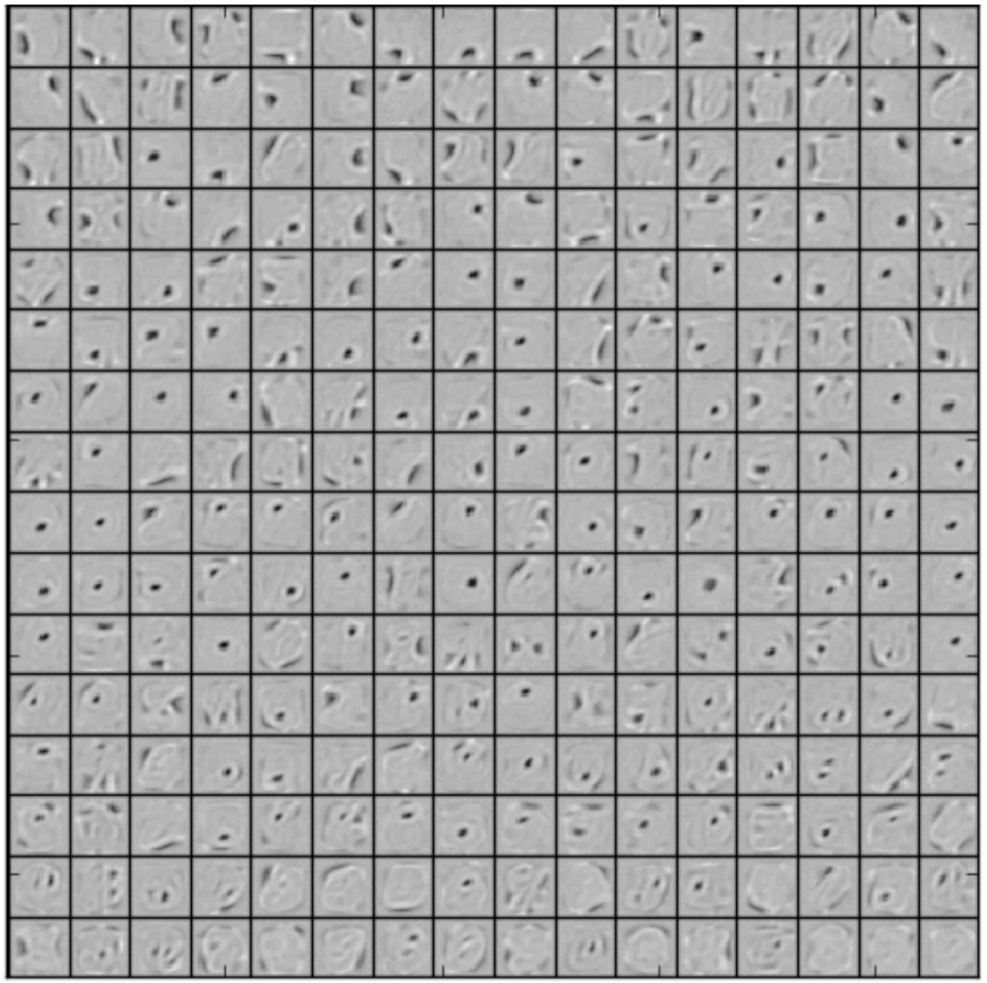



[Srivastava et al JMLR ’14]

# Dropout: Effect on filters



(a) Without dropout



(b) Dropout with $p = 0.5$.

[Srivastava et al JMLR '14]

# Batch normalization

- We have discussed data normalization (mean 0, variance 1) as a common approach for stabilizing training
- In deep learning, inputs are shifted and scaled through each layer
- Batch Normalization is a widely used technique that normalizes inputs to each layer according to mini-batch statistics
  - "Reduces internal covariate shift"
  - Enables larger learning rates
  - Helps with regularization

# Batch normalization [Ioffe & Szegedy 2015]

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

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

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

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

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

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

$\varphi(Wx) \rightsquigarrow \varphi(W B_{\gamma,\beta}(x))$

Params: $W$ ↪ Params $W, \gamma, \beta$

# Batch normalization [Ioffe & Szegedy 2015]