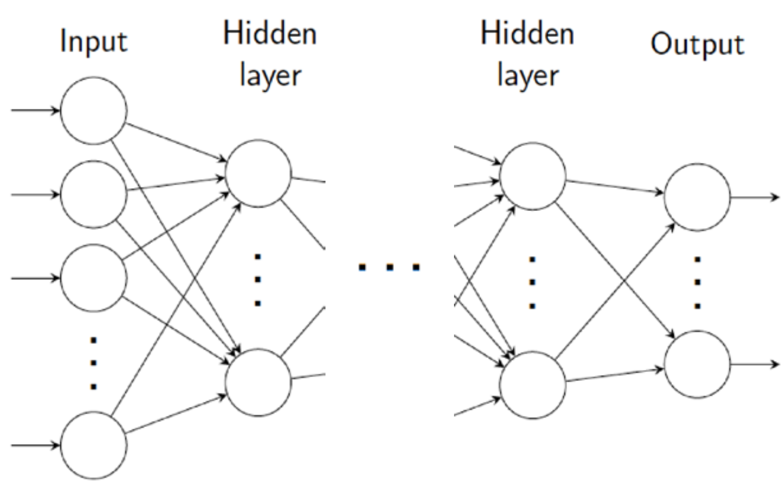
Training Neural Networks

6.036 Introduction to Machine Learning

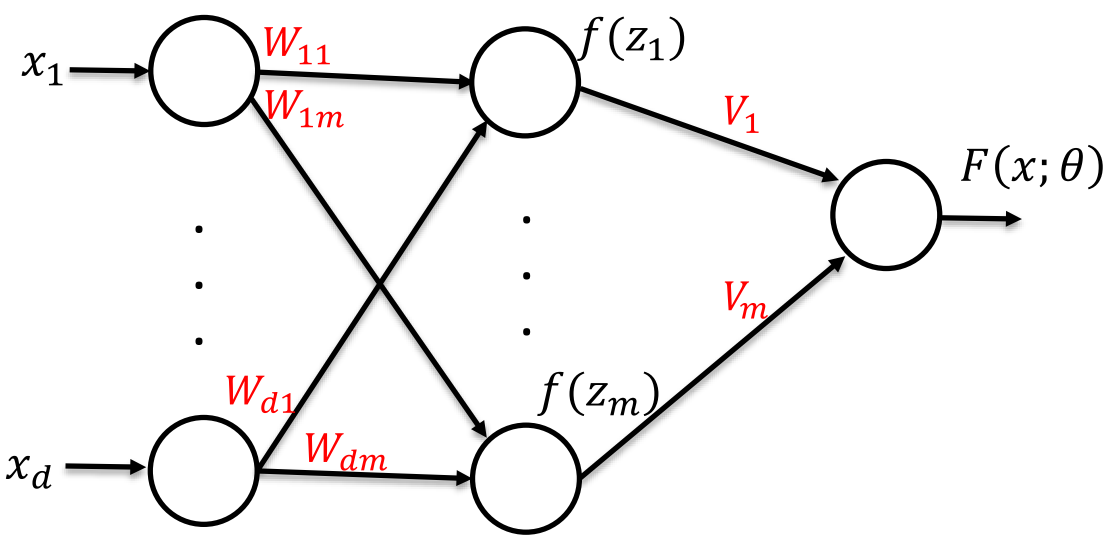
Feedforward Neural Networks

- Representation
  - Input, hidden layers, output
  - Parameters/weights
  - Activation functions
- Each layer computes some function of the previous layer
- Inputs mapped in a feed-forward fashion to output



Training Neural Networks

- Given a training dataset $S_n = \{(x^{(i)}, y^{(i)}), i = 1 \dots n\}$, estimate weights $\boldsymbol{\theta}$ to minimize the average loss over the training examples
- $\theta = \{W_{ij}, W_{0j}, V_j, V_0\}$



Training using SGD

initialize network parameters $\boldsymbol{\theta}$

repeat (until some stopping criteria are met)

pick a random example $\boldsymbol{t}$ from the training set

compute prediction: $F\left(x^{(t)};\theta\right)$

compute error/loss: $\mathbf{Loss}\ \left(y^{(t)}F\left(x^{(t)};\theta\right)\right)$

compute gradient of the error: $\nabla_{\theta}\mathrm{Loss}\ \left(y^{(t)}F\left(x^{(t)};\theta\right)\right)$

update weights: $\theta \leftarrow \theta - \eta\nabla_{\theta}\mathrm{Loss}\ \left(y^{(t)}F\left(x^{(t)};\theta\right)\right)$

Gradients for Single-Layer Networks

- Computing gradient analytically

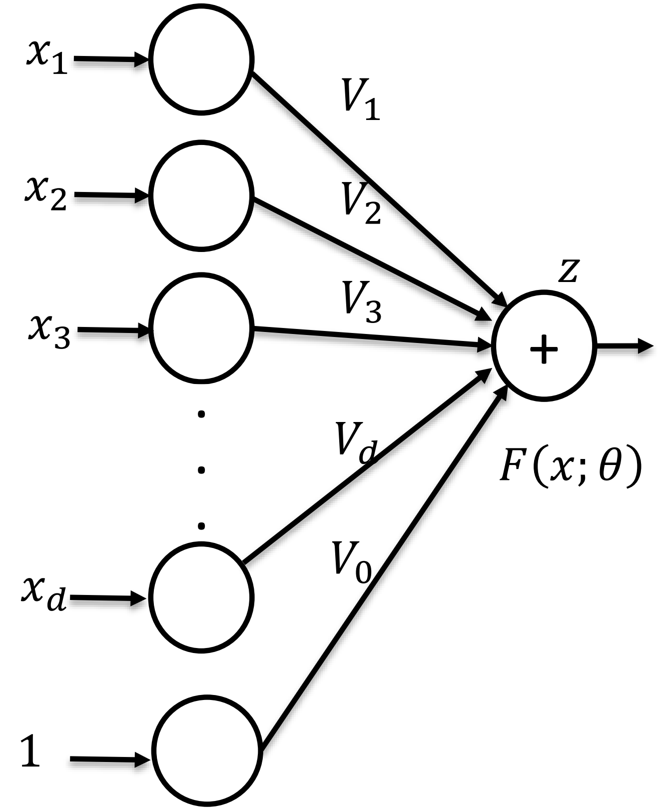
$$\frac{\partial}{\partial V_i}\text{Loss}\left(y^{(t)}\, F\left(x^{(t)};\theta\right)\right) =$$

$$= \frac{\partial}{\partial V_i}\text{Loss}\left(y^{(t)}\, z^{(t)}\right)$$

$$= \left[\frac{\partial}{\partial z^{(t)}}\text{Loss}\left(y^{(t)}\, z^{(t)}\right)\right]\left[\frac{\partial z^{(t)}}{\partial V_i}\right] =$$

$$= \left[\frac{\partial}{\partial z^{(t)}}\text{Loss}\left(y^{(t)}\, z^{(t)}\right)\right]\left[\frac{\partial\left(\sum_{j=1}^{d} x_j^{(t)} V_j + V_0\right)}{\partial V_i}\right] =$$

$$= \begin{bmatrix} -y^{(t)} \text{ if } \text{Loss}\left(y^{(t)} z^{(t)}\right) > 0 \\ 0 \; otherwise \end{bmatrix}\left[x_i^{(t)}\right]$$



$$z^{(t)} = \sum_{j=1}^{d} x_j^{(t)} V_j + V_0$$

$$F\left(x^{(t)};\theta\right) = f\left(z^{(t)}\right) = z^{(t)}$$

Chain Rule: $\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$

Training using SGD

initialize network parameters $\theta = \{V_0, \dots, V_d\}$

repeat (until some stopping criteria are met)

pick a random example t from the training set

compute prediction: $F\left(x^{(t)}; \theta\right)$

compute error/loss: $\mathbf{Loss}\ \left(y^{(t)} F\left(x^{(t)}; \theta\right)\right)$

compute gradient of the error: $\nabla_\theta \mathbf{Loss}\ \left(y^{(t)} F\left(x^{(t)}; \theta\right)\right)$

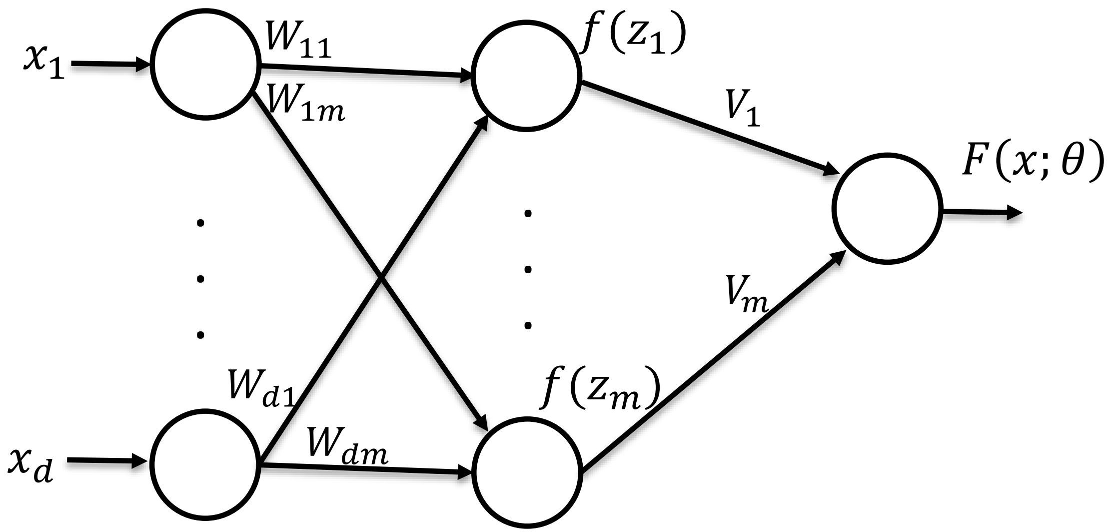
update weights: $\theta \ \leftarrow \theta \ - \ \eta \nabla_\theta \mathbf{Loss}\ \left(y^{(t)} F\left(x^{(t)}; \theta\right)\right)$

$$V_i \leftarrow V_i + \ \eta \ x_i^{(t)} y^{(t)}, i = 1 \dots d$$

$$V_0 \leftarrow V_0 + \ \eta \ y^{(t)}$$

SGD for 2-Layer Neural Network

Parameters $\theta = \{W_{ij}, W_{0j}\} \,\&\, \{V_j, V_0\}$



$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j}$$

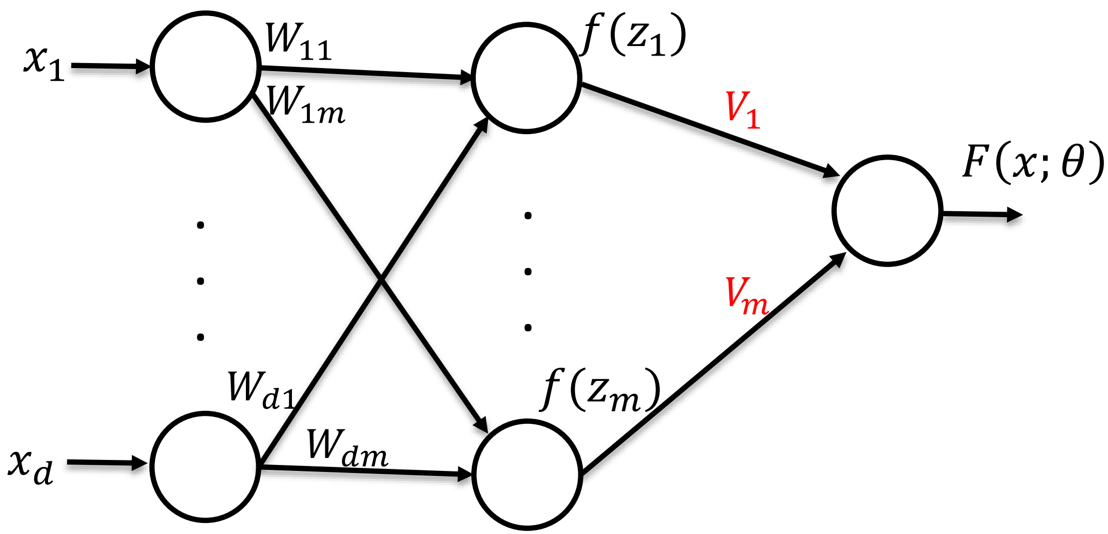
$$f(z_j) = \max\{0, z_j\}$$

$$z = \sum_{j=1}^{m} f(z_j) V_j + V_0$$

$$F(x;\theta) = z$$

Updates for V_j

Parameters $\theta = \{W_{ij}, W_{0j}\}$ & $\{V_j, V_0\}$



$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j}$$

$$f(z_j) = \max\{0, z_j\}$$

$$z = \sum_{j=1}^{m} f(z_j) V_j + V_0$$

$$F(x;\theta) = z$$

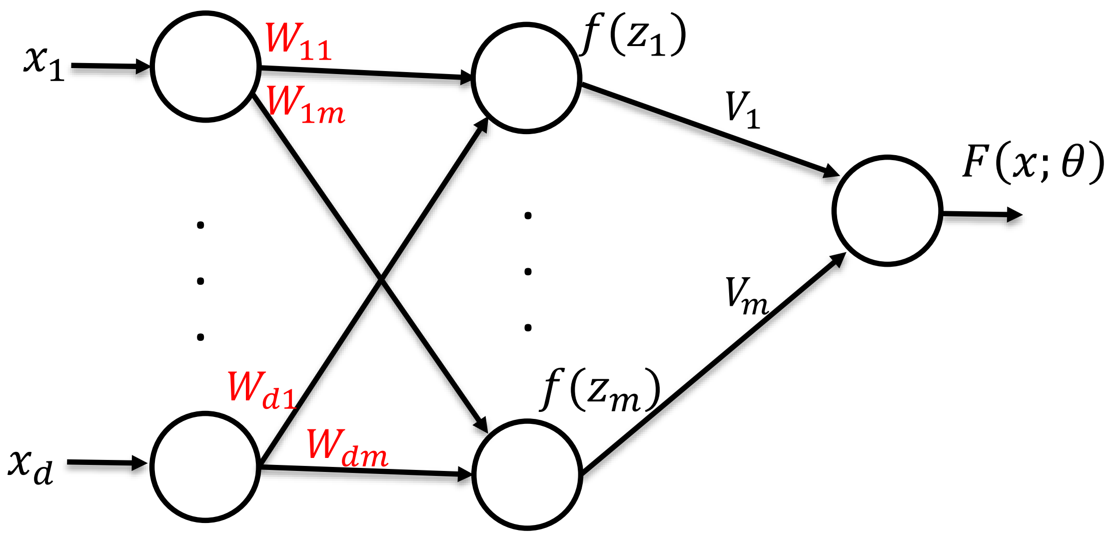
Updates for V_j are the same as for the single-layer network

Replace $x_i^{(t)}$ with $f(z_j^{(t)})$

$$V_j \leftarrow V_j + \eta_k y^{(t)} f(z_j^{(t)}), \quad j = 1, \ldots, m$$

Updates for W_{ij}

Parameters $\theta = \{W_{ij}, W_{0j}\}$ & $\{V_j, V_0\}$



$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j}$$

$$f(z_j) = \max\{0, z_j\}$$

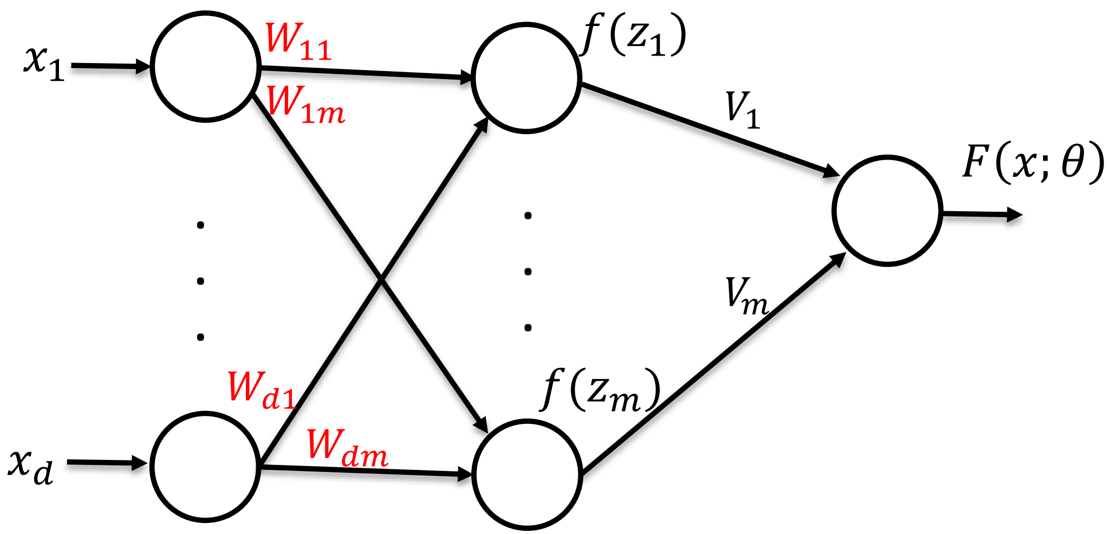
$$z = \sum_{j=1}^{m} f(z_j) V_j + V_0$$

$$F(x;\theta) = z$$

To compute $\frac{\partial}{\partial V_i} \mathrm{\textbf{Loss}}\left(y^{(t)} F\left(x^{(t)};\theta\right)\right)$, we could start by writing $\mathrm{\textbf{Loss}}\left(y^{(t)} F\left(x^{(t)};\theta\right)\right)$ as a function of the network parameters $\boldsymbol{\theta}$. And then compute the partial derivatives … Instead, we can use the chain rule to derive a compact algorithm: back-propagation

Updates for W_{ij}

Parameters $\theta = \{W_{ij}, W_{0j}\}$ & $\{V_j, V_0\}$



$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j}$$

$$f(z_j) = \max\{0, z_j\}$$

$$z = \sum_{j=1}^{m} f(z_j) V_j + V_0$$

$$F(x;\theta) = z$$

$$\frac{\partial}{\partial W_{ij}} \mathrm{Loss}\left(y^{(t)} z^{(t)}\right) = \left[\frac{\partial}{\partial z^{(t)}} \mathrm{Loss}\left(y^{(t)} z^{(t)}\right)\right] \left[\frac{\partial z^{(t)}}{\partial f(z_j^{(t)})}\right] \left[\frac{\partial f(z_j^{(t)})}{\partial z_j^{(t)}}\right] \left[\frac{\partial z_j^{(t)}}{\partial W_{ij}}\right]$$

$$= \begin{bmatrix} -y^{(t)} \text{ if } \mathrm{Loss}\left(y^{(t)} z^{(t)}\right) > 0 \\ 0 \; otherwise \end{bmatrix} [V_j] \left[\!\left[z_j^{(t)} > 0 \right]\!\right] [x_i]$$

Back-propagation

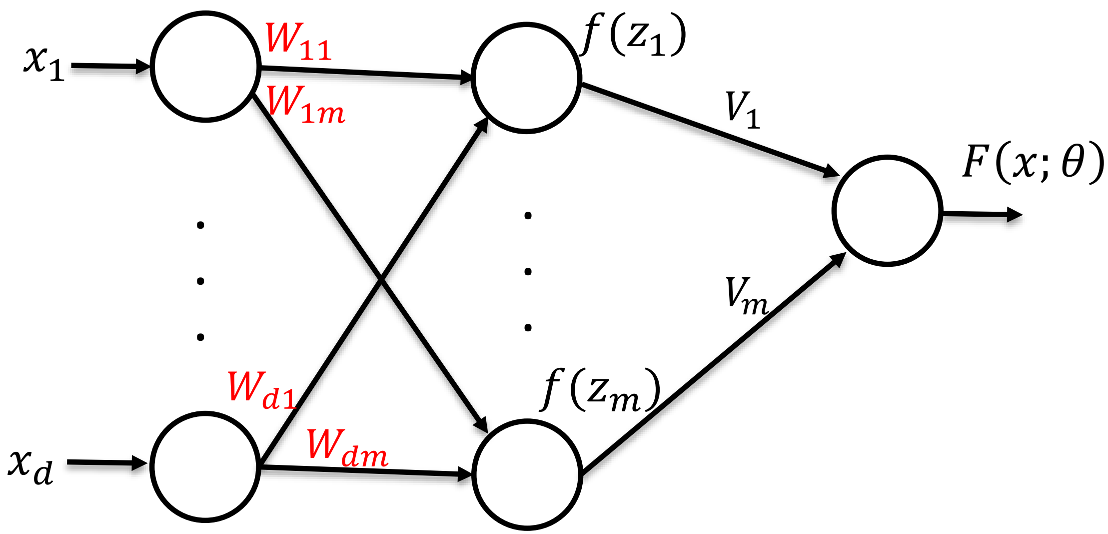
- The process of propagating the gradients backwards towards the input layer is called back-propagation
- Back-propagation is based on applying the chain rule of derivatives back through the model
- Back-propagation can be applied the same way to compute gradients in networks with multiple hidden layers
- Computing derivatives can be reused between layers

$$\frac{\partial}{\partial W_{ij}} \text{Loss}\big(y^{(t)}\, z^{(t)}\big) = \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)}\, z^{(t)}\big)\right] \left[\frac{\partial z^{(t)}}{\partial f(z_j^{(t)})}\right] \left[\frac{\partial f(z_j^{(t)})}{\partial z_j^{(t)}}\right] \left[\frac{\partial z_j^{(t)}}{\partial W_{ij}}\right]$$

$$= \begin{bmatrix} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)} z^{(t)}\big) > 0 \\ 0\ otherwise \end{bmatrix} [V_j]\, [\![z_j^{(t)} > 0]\!]\, [x_i]$$

Updates for W_{ij}

Parameters $\theta = \{W_{ij}, W_{0j}\}$ & $\{V_j, V_0\}$



$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j}$$

$$f(z_j) = \max\{0, z_j\}$$

$$z = \sum_{j=1}^{m} f(z_j) V_j + V_0$$

$$F(x;\theta) = z$$

$$W_{ij} \leftarrow W_{ij} + \eta_k\, x_i^{(t)} [\![z_j^{(t)} > 0]\!] V_j y^{(t)}, \quad i = 1, \ldots, d, \; j = 1, \ldots, m$$

Initialization

- What happens when all weights are initialized to 0?

$$V_j \leftarrow V_j + \eta_k y^{(t)} f(z_j^{(t)}), \quad j = 1, \ldots, m$$

$$W_{ij} \leftarrow W_{ij} + \eta_k \, x_i^{(t)} [\![\, z_j^{(t)} > 0 \,]\!] V_j y^{(t)}, \quad i = 1, \ldots, d, \; j = 1, \ldots, m$$

Initialization

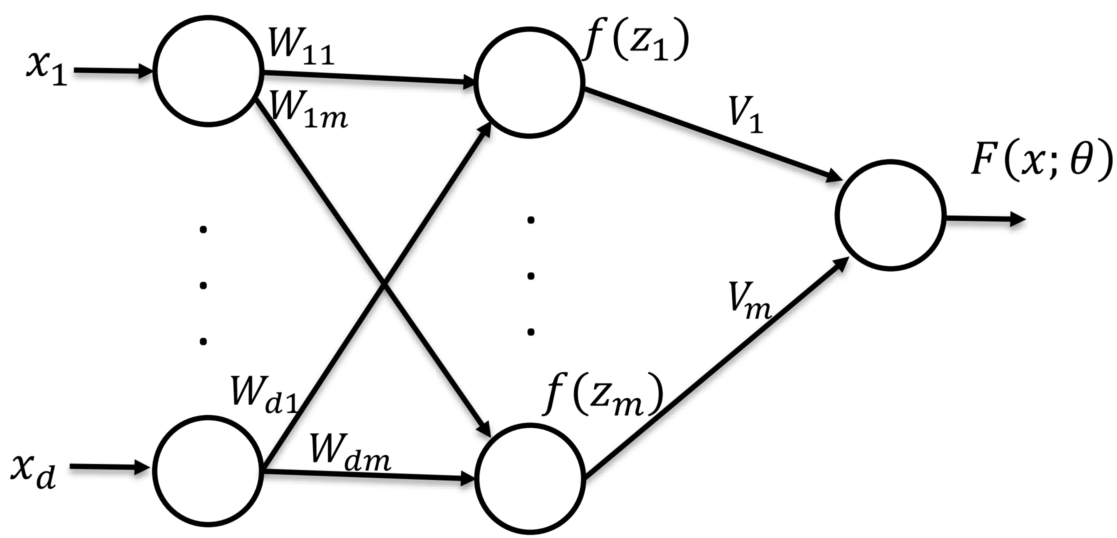
- What happens when all weights are initialized to 0?

$$V_j \leftarrow V_j$$

$$W_{ij} \leftarrow W_{ij}$$

Initialization

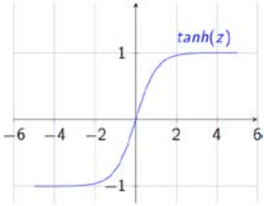
- Typically random
  - Zero mean and variance $1/d^2$

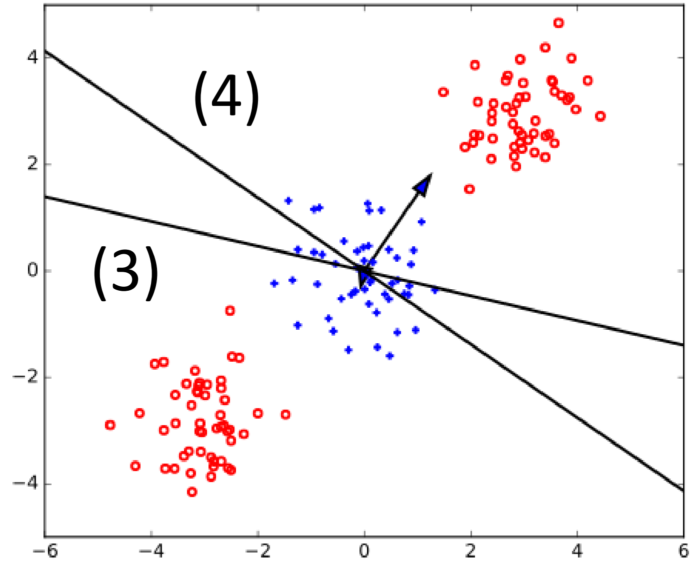


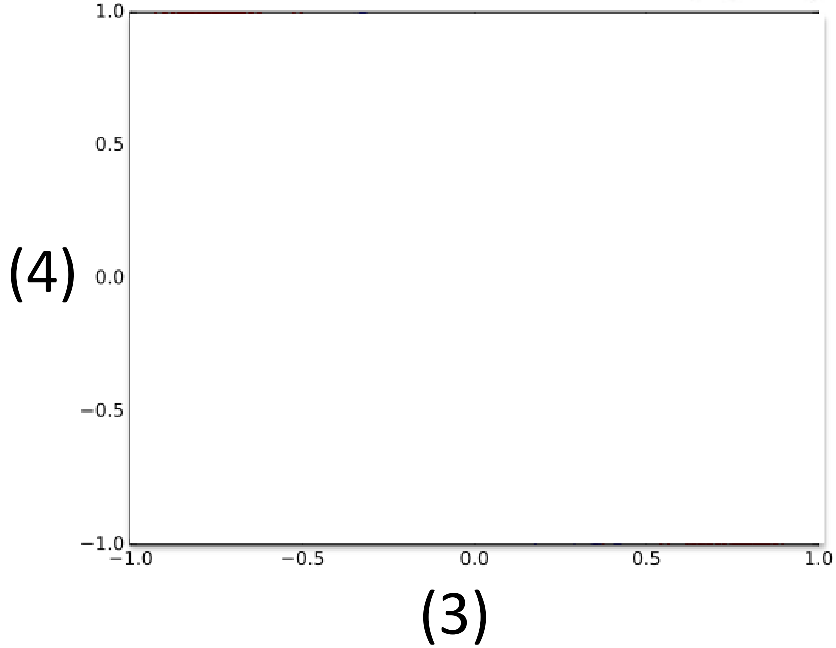
2 Hidden Units

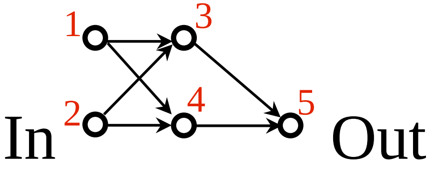
- Randomly initialized weights (zero offset)

tanh activation





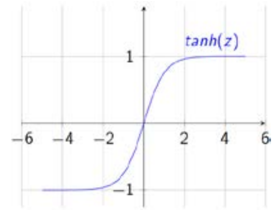


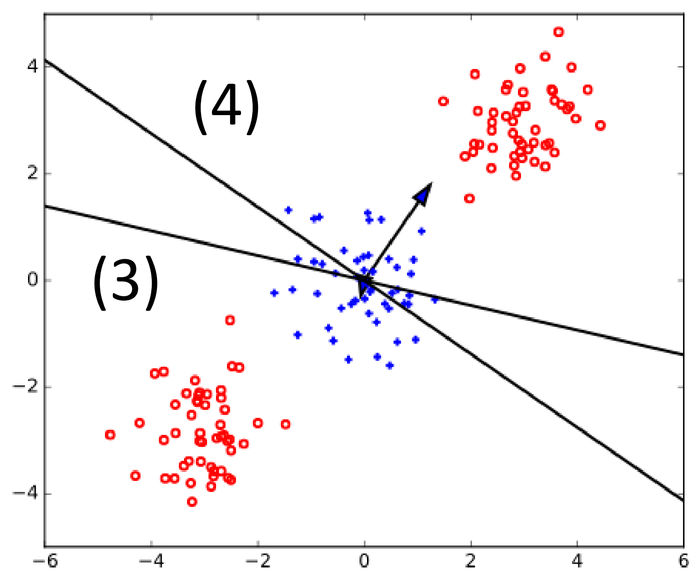


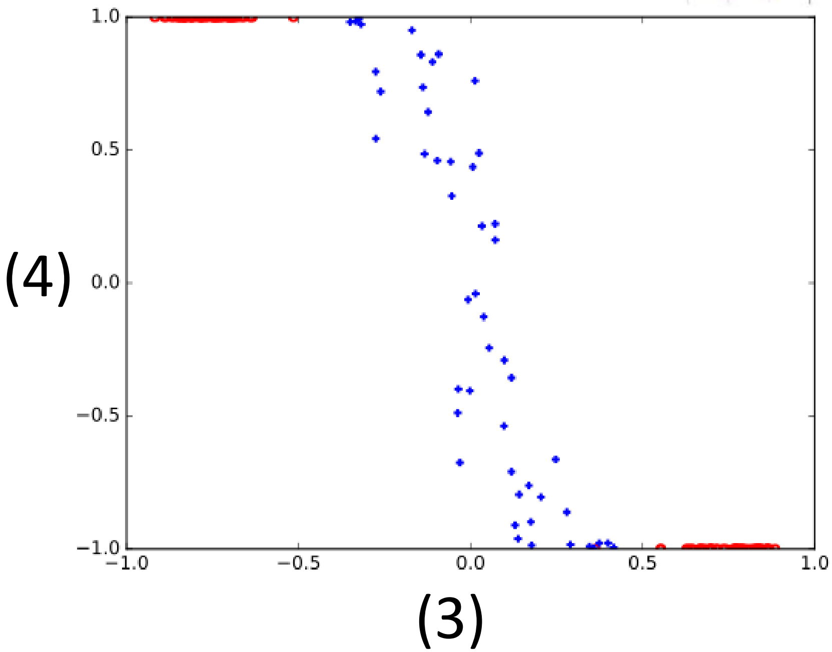
2 Hidden Units

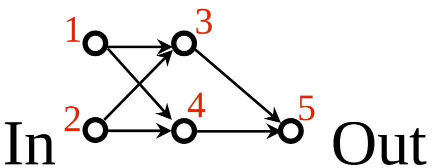
- Randomly initialized weights (zero offset)

tanh activation

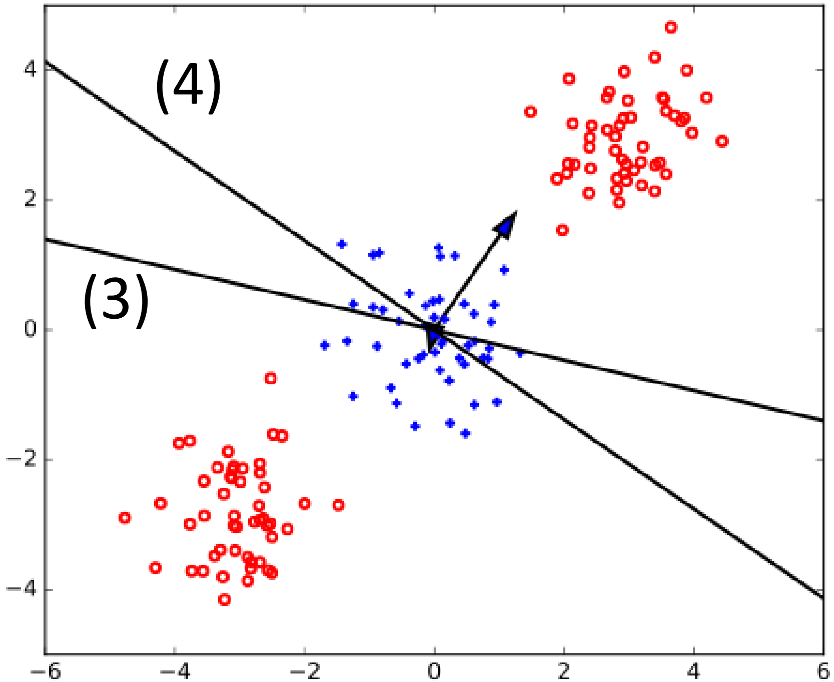




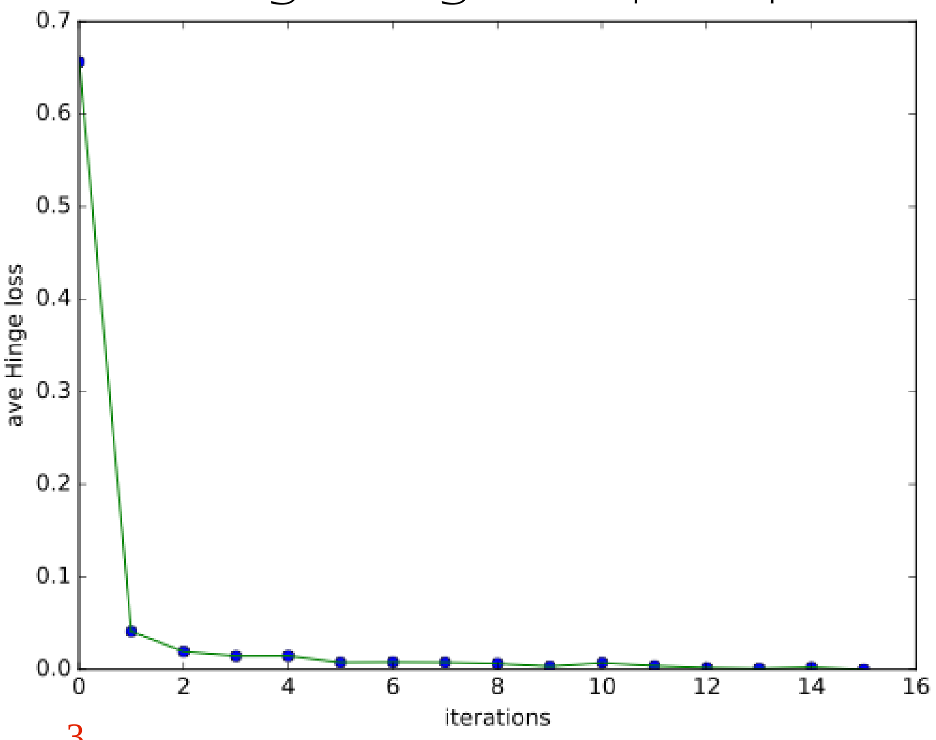


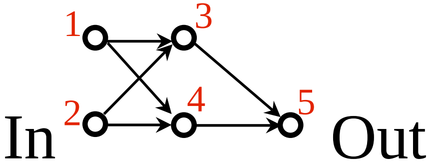


2 Hidden Units: Training



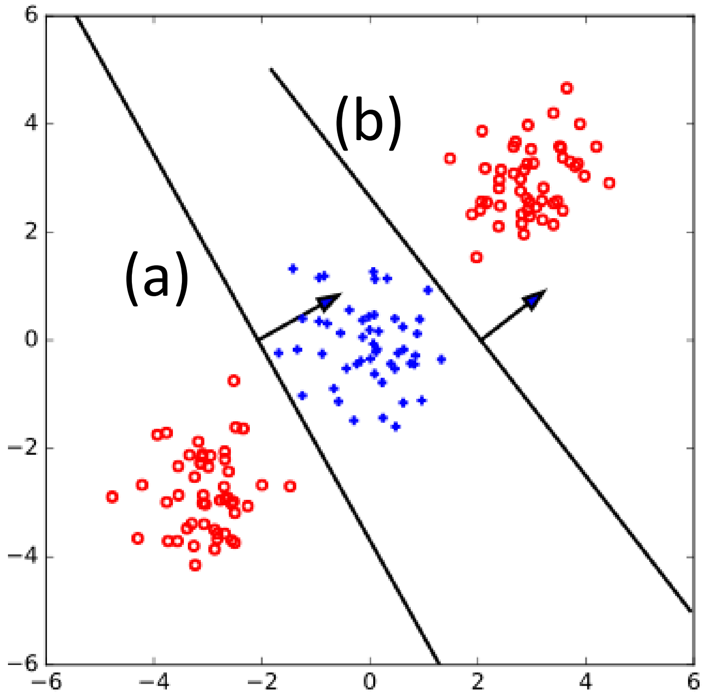
Average hinge loss per epoch

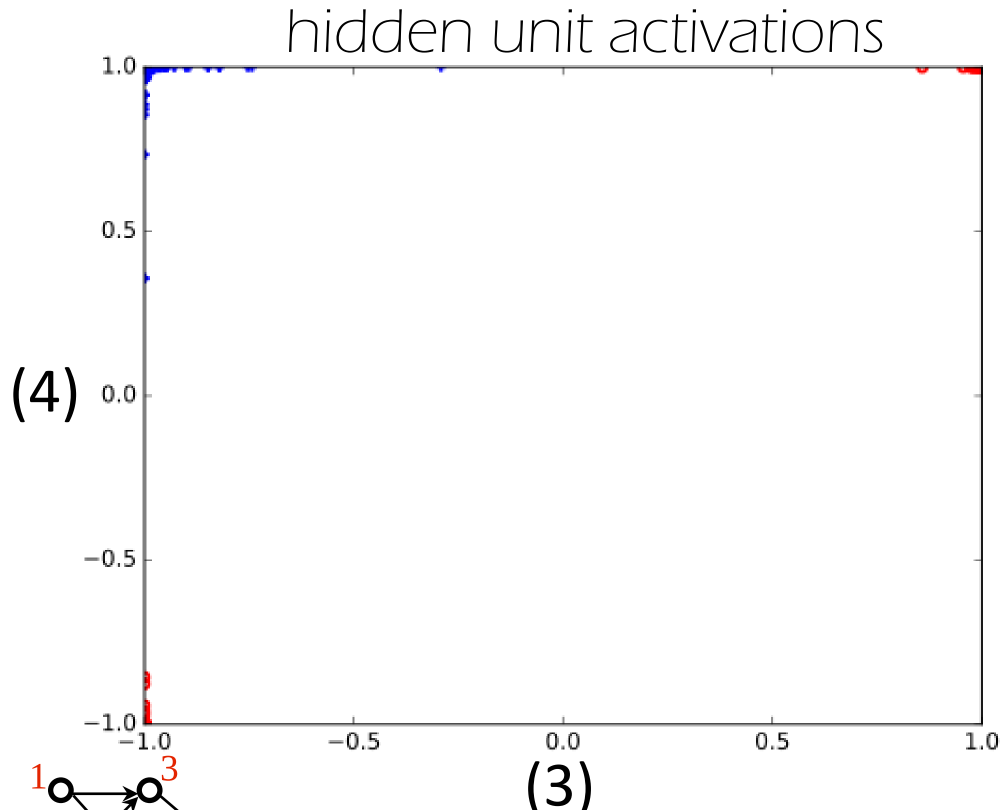


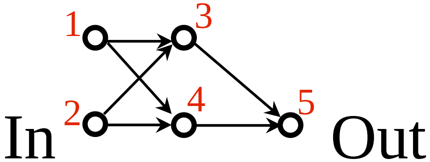


2 Hidden Units: Training

- *After ~10 passes through the data*

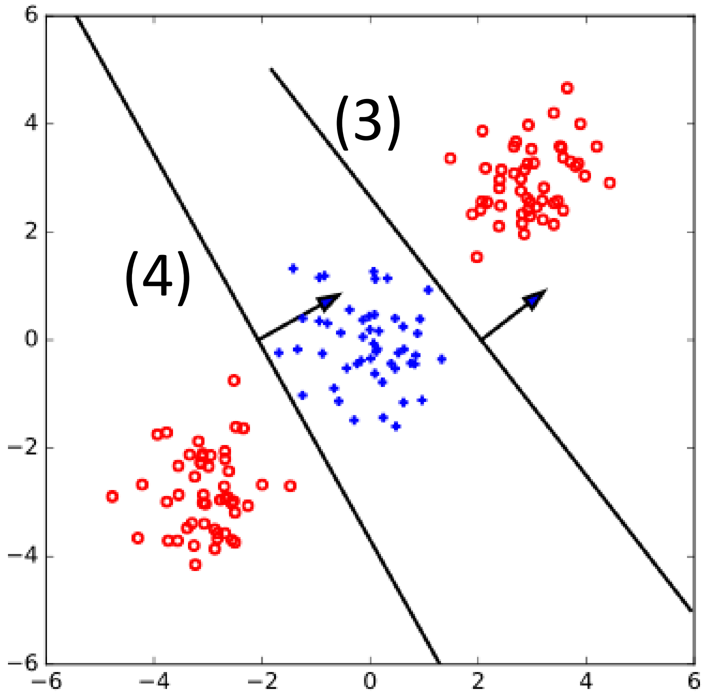




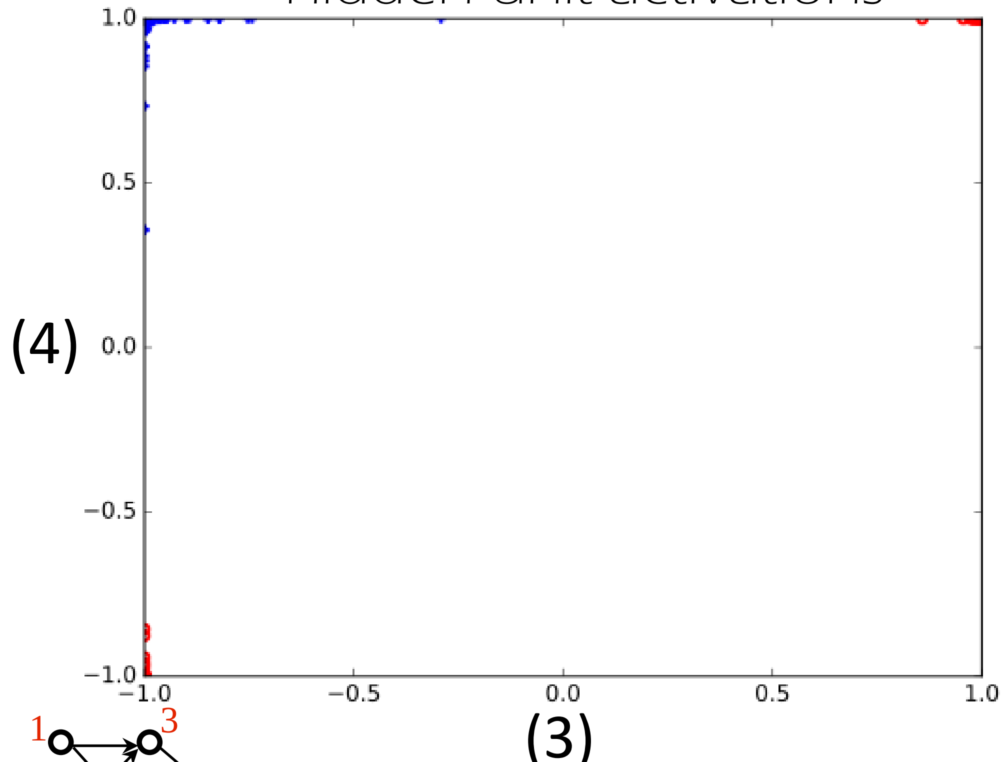


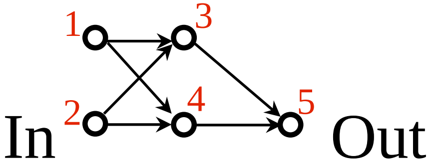
2 Hidden Units: Training

- *After ~10 passes through the data*



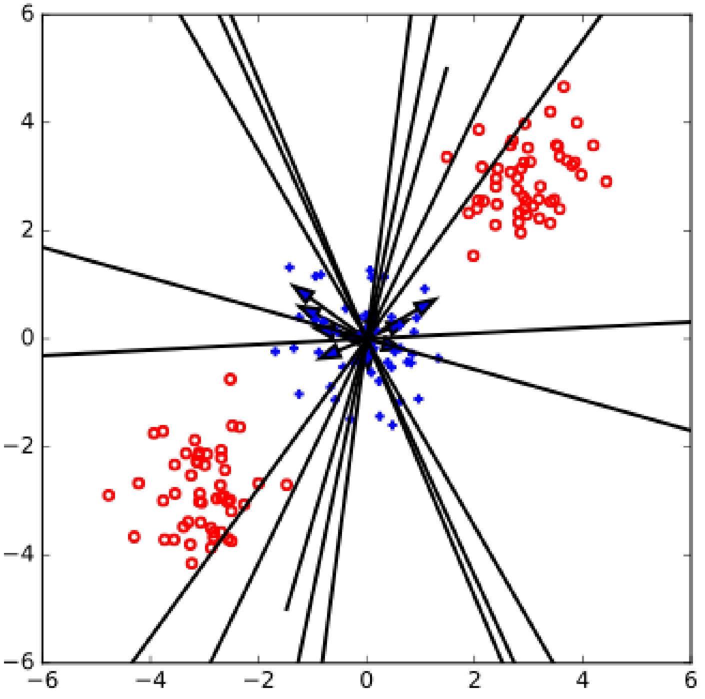
hidden unit activations





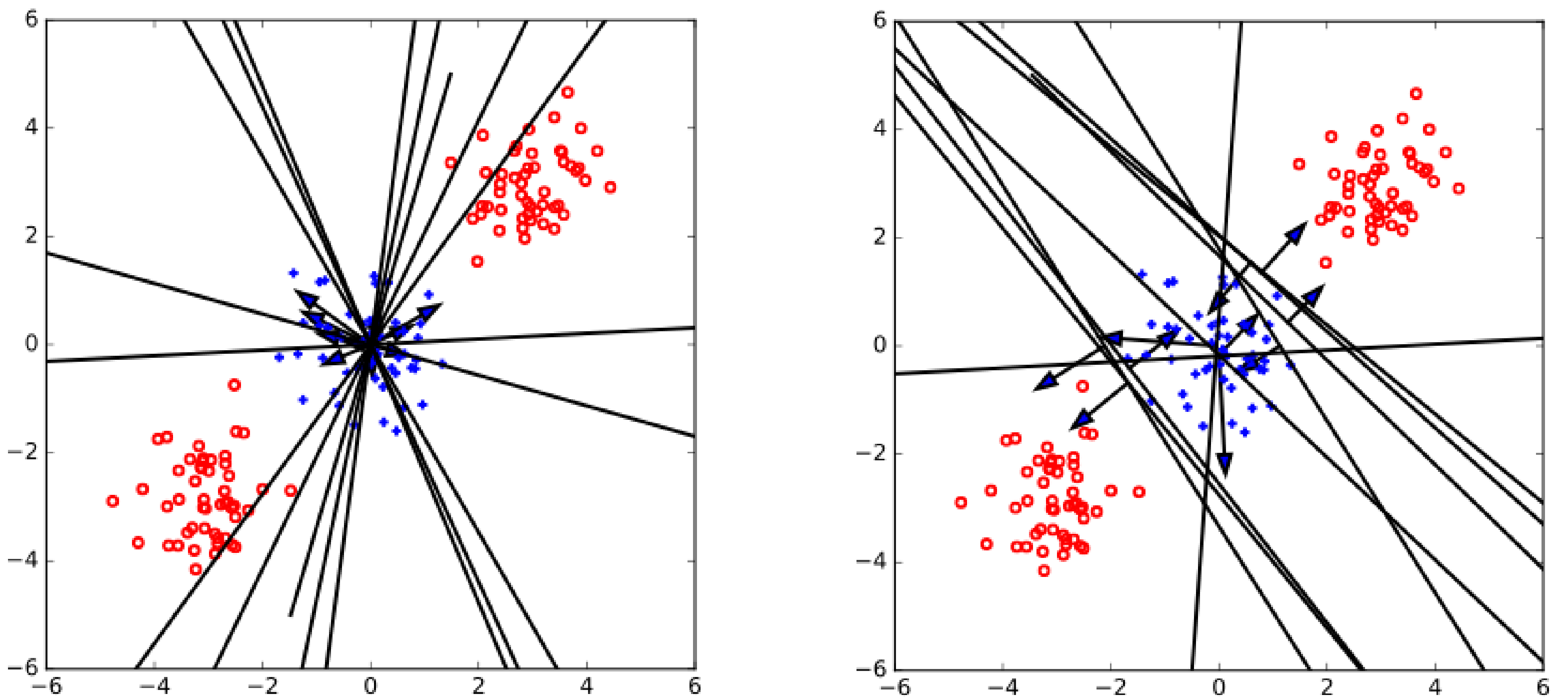
10 Hidden Units

- Randomly initialized weights (zero offset) for the hidden units



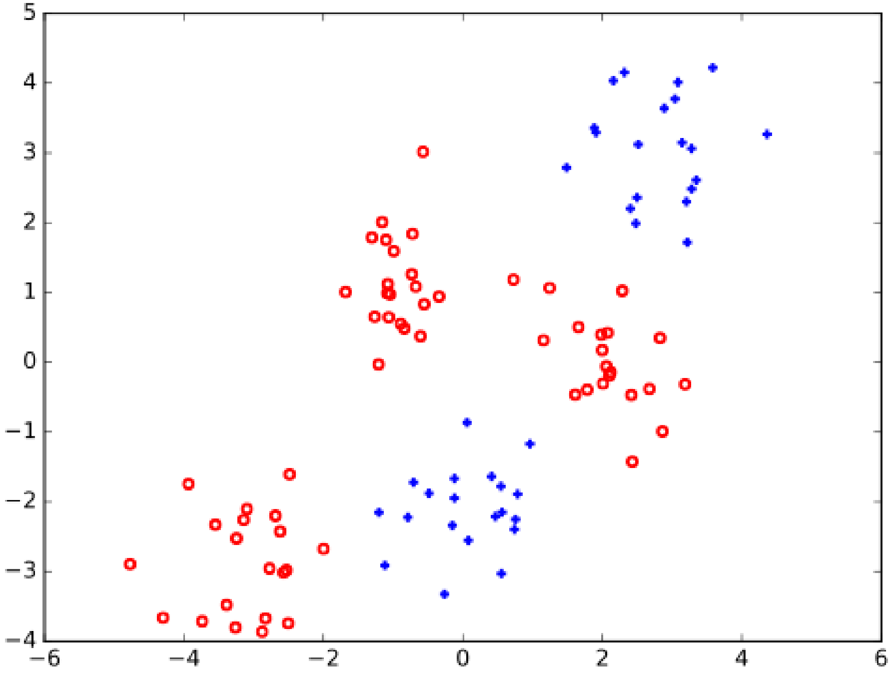
10 Hidden Units

- *After ~ 10 epochs the hidden units are arranged in a manner sufficient for the task (but not otherwise perfect)*



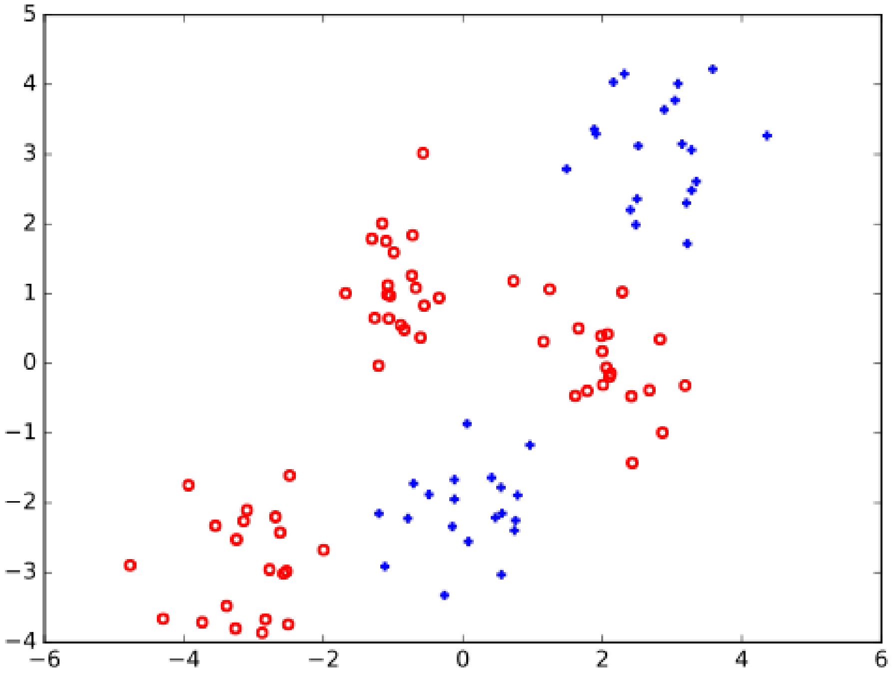
Decisions (and a harder task)

- 2 hidden units can no longer solve this task

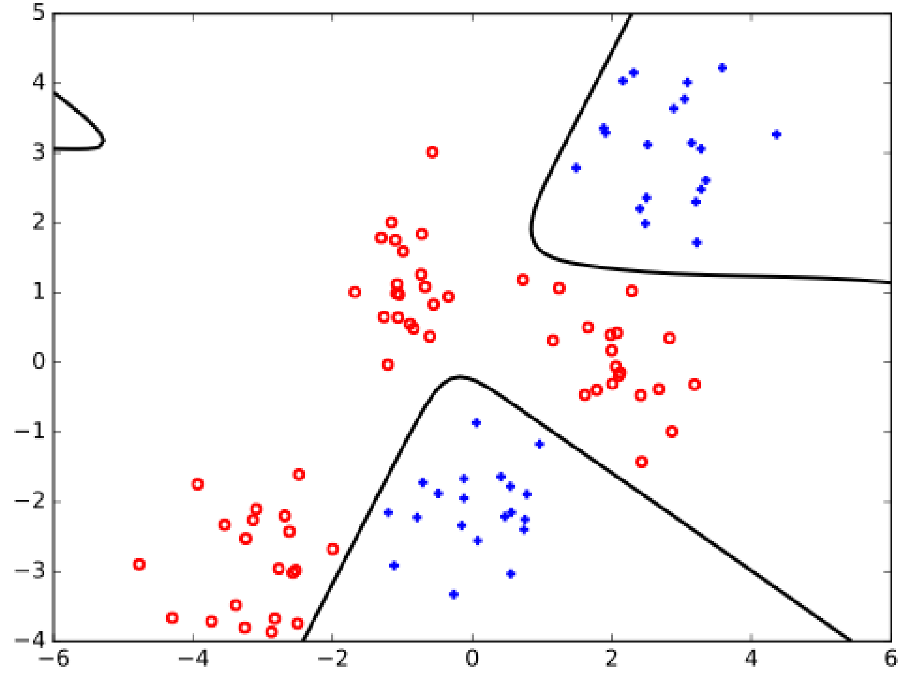


Decisions (and a harder task)

- 2 hidden units can no longer solve this task

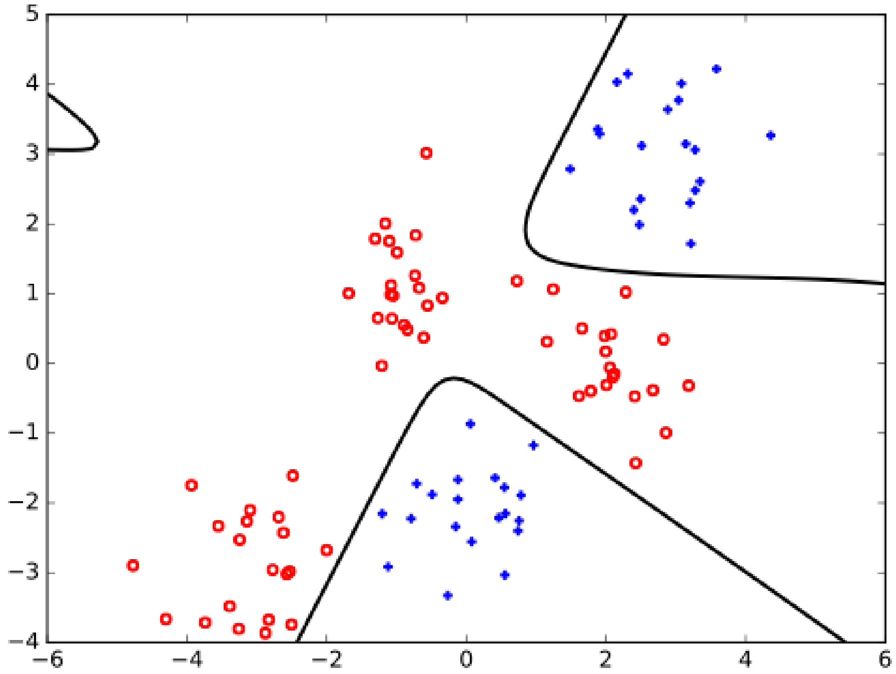


10 hidden units

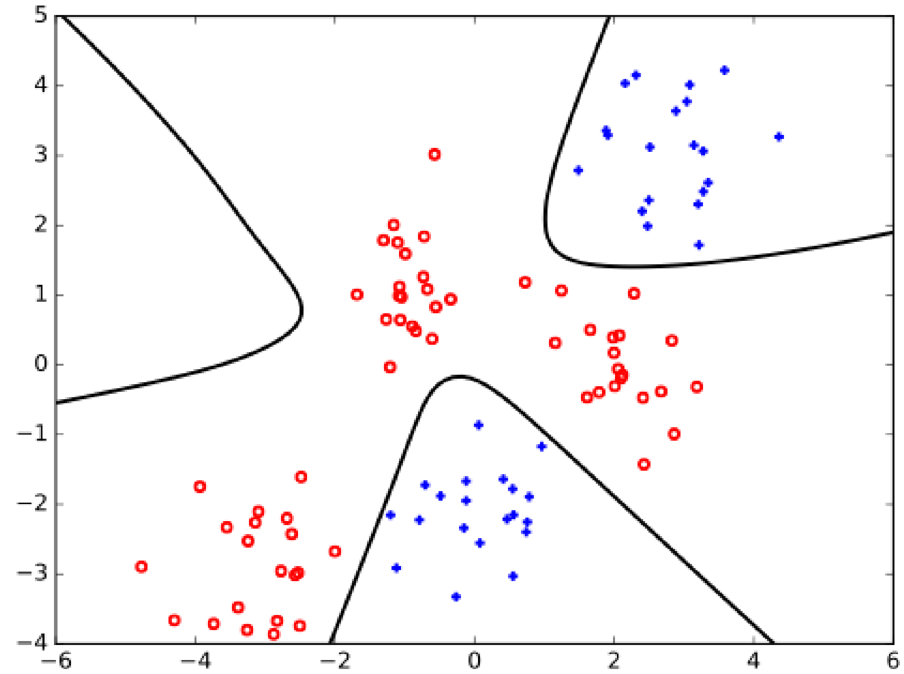


Decisions (and a harder task)

10 hidden units

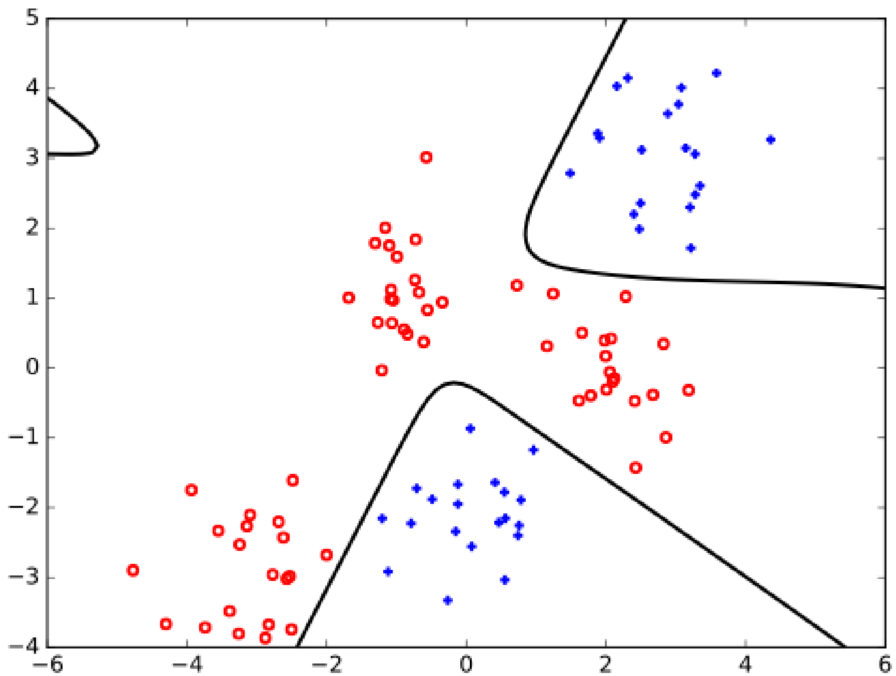


100 hidden units



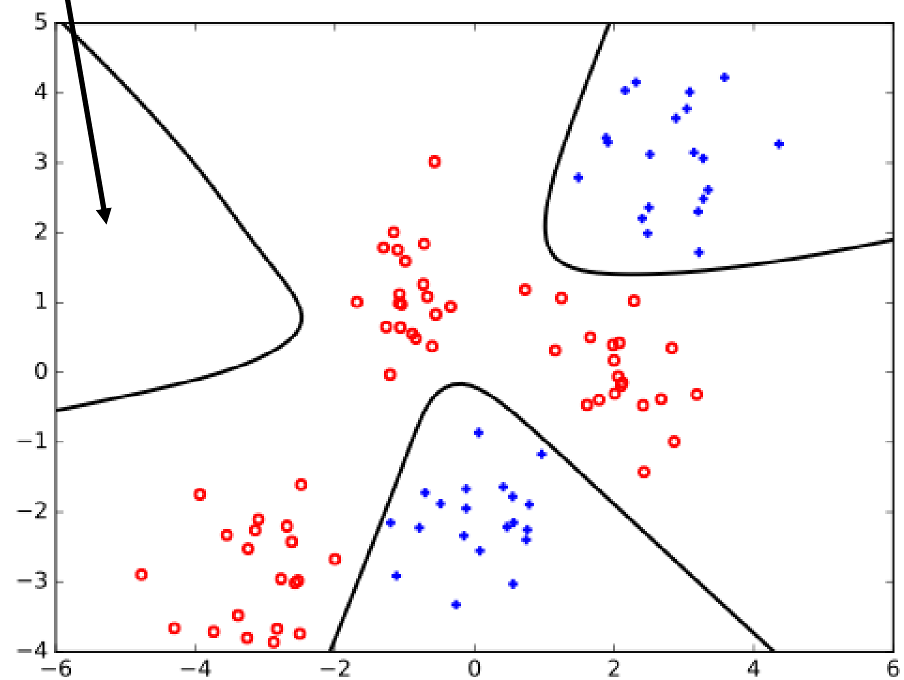
Decisions (and a harder task)

10 hidden units



???

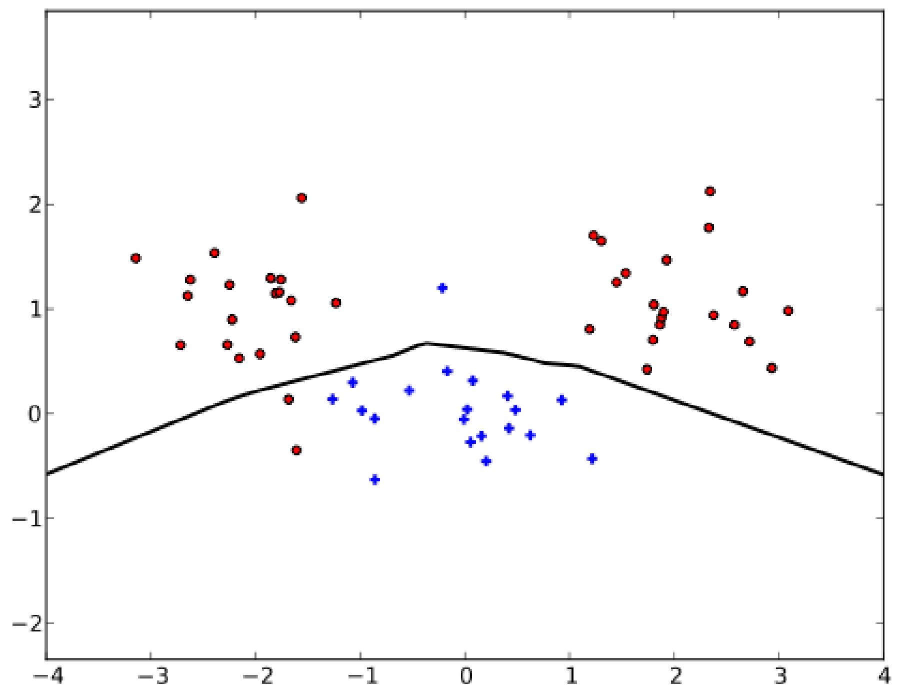
100 hidden units



Architectural Variations...

- Increasing the number of hidden units
  - More powerful decision boundary
  - Easier to fit the data

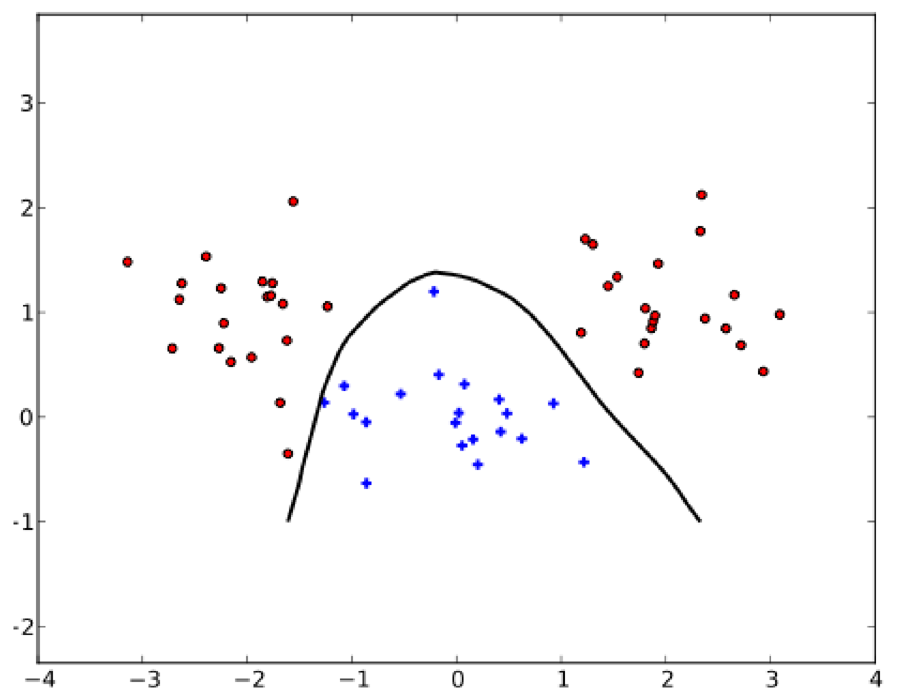
10 hidden units



Architectural Variations...

- Increasing the number of hidden units
  - More powerful decision boundary
  - Easier to fit the data

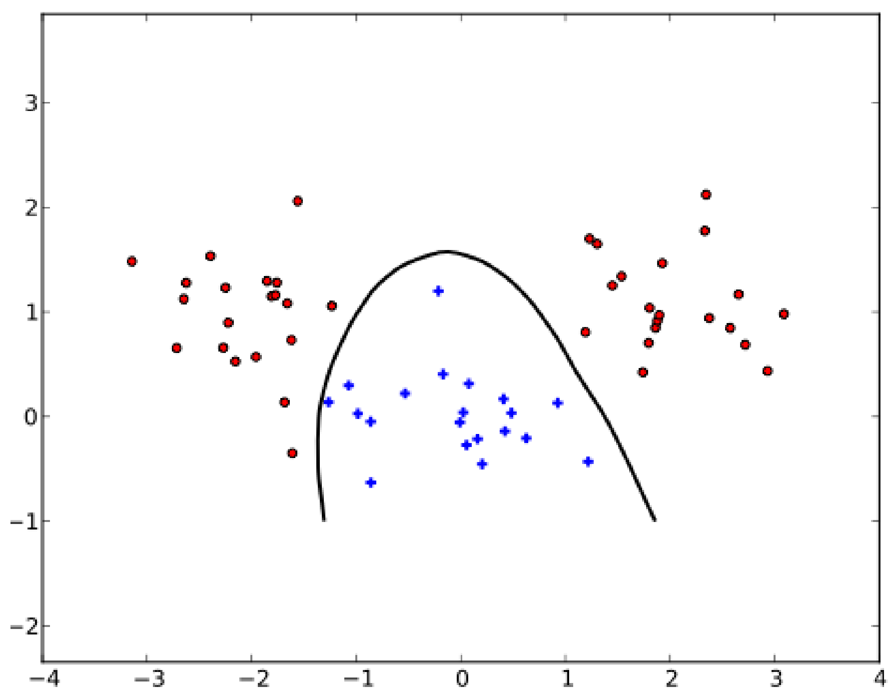
100 hidden units



Architectural Variations...

- Increasing the number of hidden units
  - More powerful decision boundary
  - Easier to fit the data

500 hidden units



Representational Power

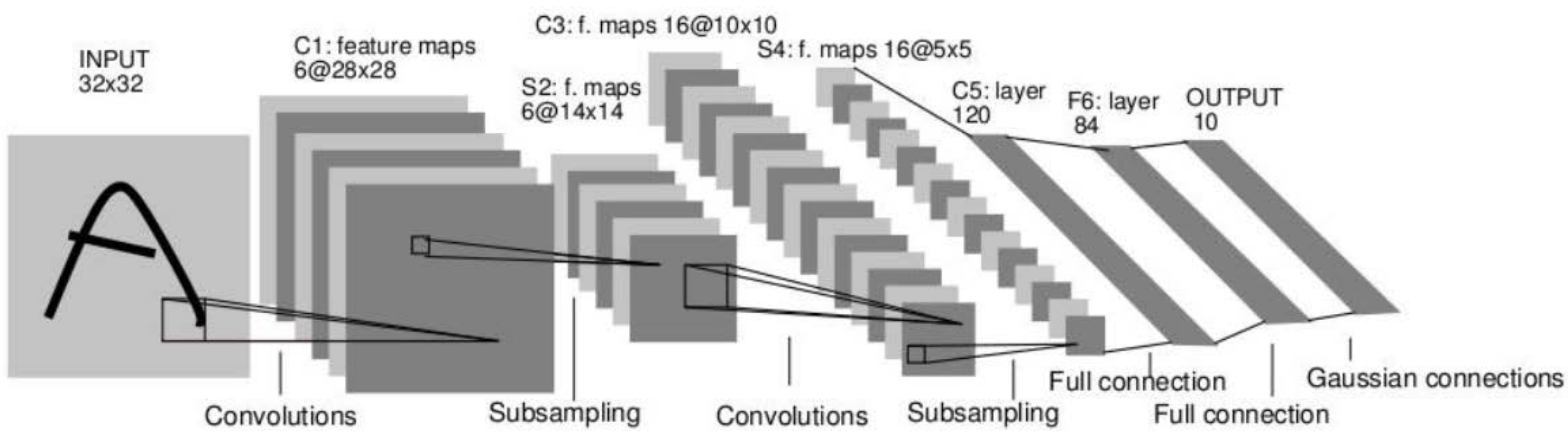
- 1 layer: Linear decision surface
- 2+ layers can represent any function assuming non-trivial non-linearity
- For fully connected models 2 or 3 layers seems the most that can be effectively trained
- Regarding number of units/layer:
  - Parameters grows with $(\text{units/layer})^2$

Architectures

- How to select:
  - Depth
  - Width
  - Parameter count
- Manual selection of features has turned into manual selection of architectures

Convolutional Neural Networks

- LeCun et al. 1989
- Neural network with specialized connectivity structure

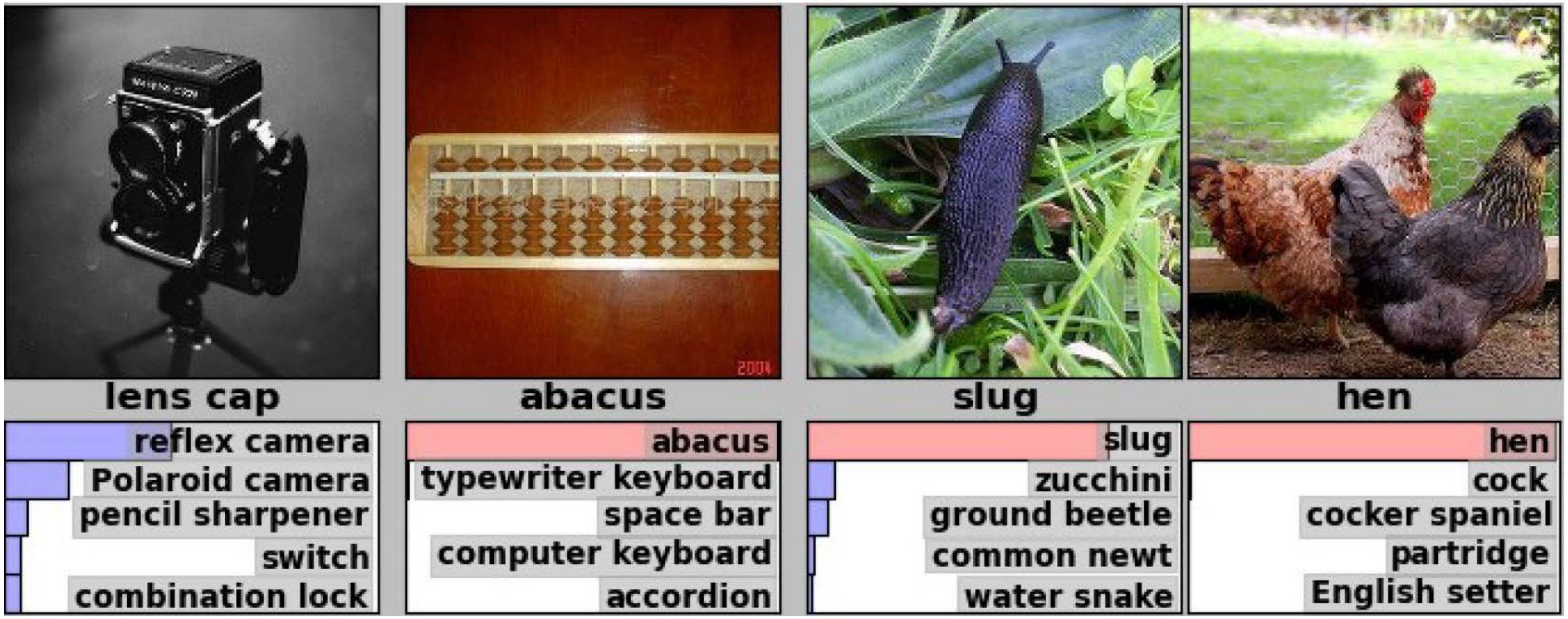


Application to ImageNet

- ~14 million images, 20K classes
- Images gathered from Internet
- Human labeled via Amazon Mechanical Turk

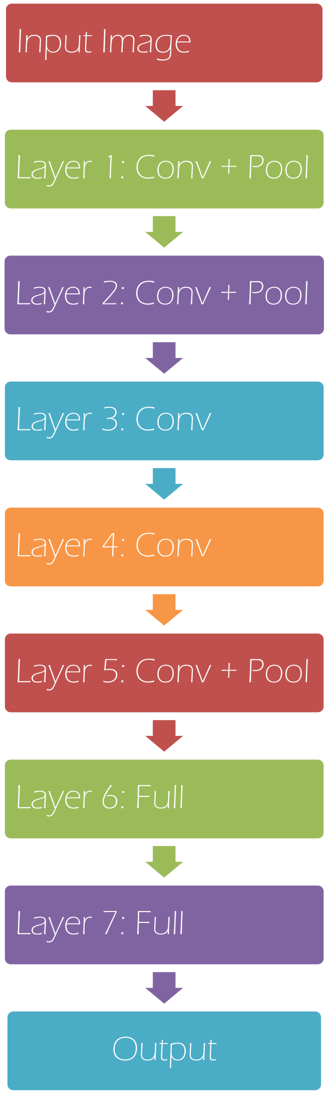
Goal

- *Image recognition*
- *Image → class label*



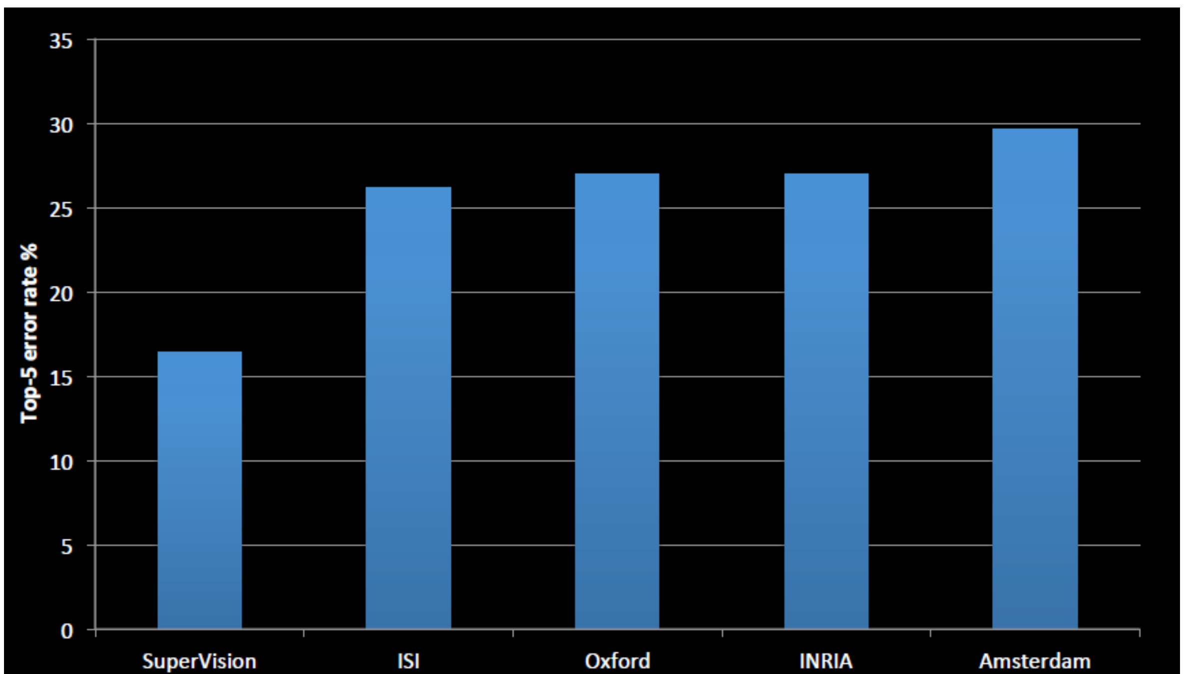
Krizhevsky et al. [NIPS 2012]

- Same model as LeCun'98 but
  - Bigger (8 layers)
    - 7 hidden layers, 650K neurons, 60M parameters
  - More data (10^6 vs 10^3 images)
  - GPU implementation (50x speedup over CPU)
    - Trained on 2 GPUs for a week
  - Better regularization



ImageNet Classification 2012

- Krizhevsky et al. – 16.4% error (top-5)
- Next best (non-convent) – 26.2% error

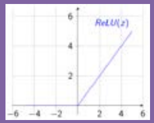


Components of Each Layer

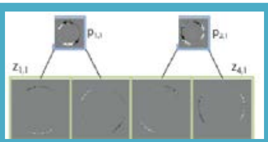
Pixels/Features

Filter with learned dictionary

Non-linearity



Spatial local max pooling



[Optional] Normalization across data/features

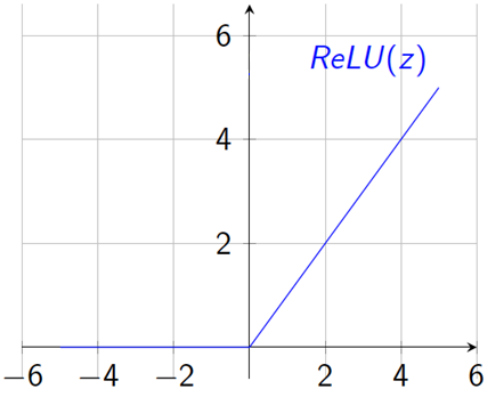
Output Features

Filtering

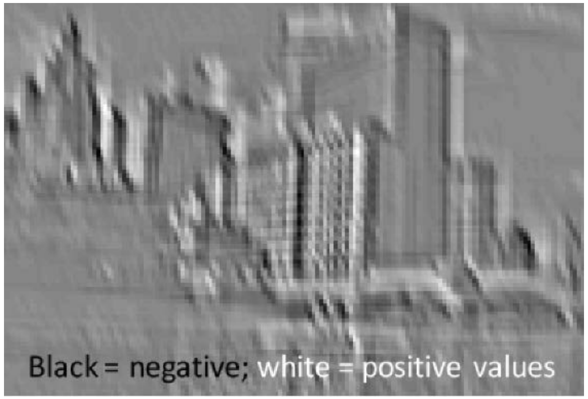
- Convolutional
  - Dependencies are local
  - Translation invariance
  - Tied filter weights (few parameters)

Non-linearity

- Rectified linear function
  - Applied per-pixel
  - Output = max(0,input)



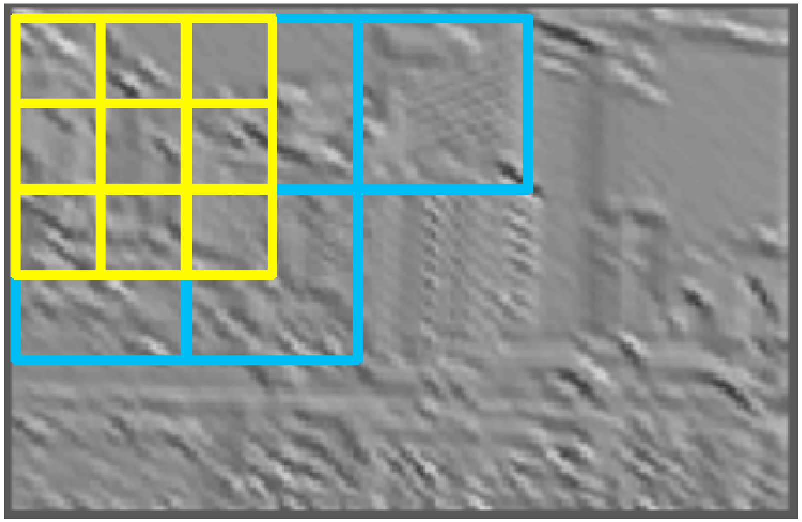
Input feature map



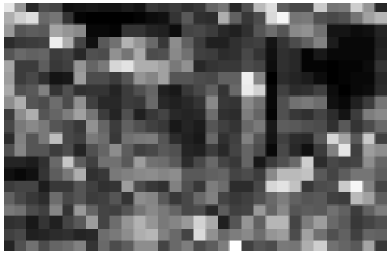
Output feature map

Pooling

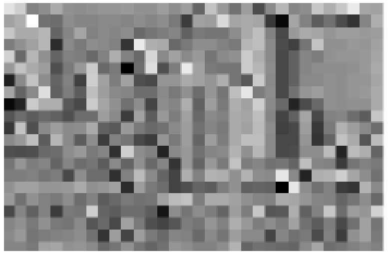
- Spatial Pooling
  - Non-overlapping/overlapping regions
  - Sum or max
  - Invariance to small transformations
  - Larger receptive field



Max

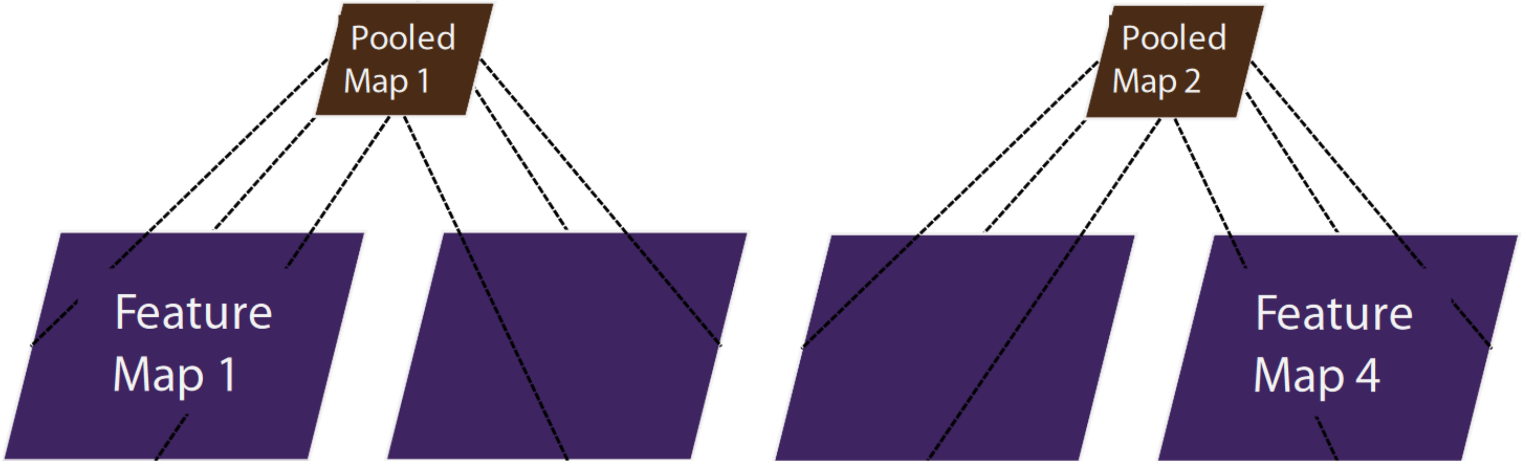


Sum



Pooling

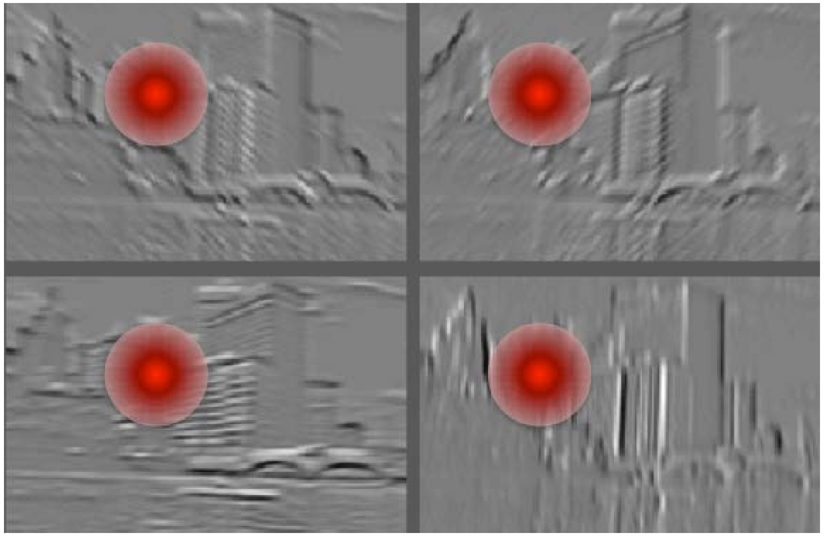
- Pooling across feature groups



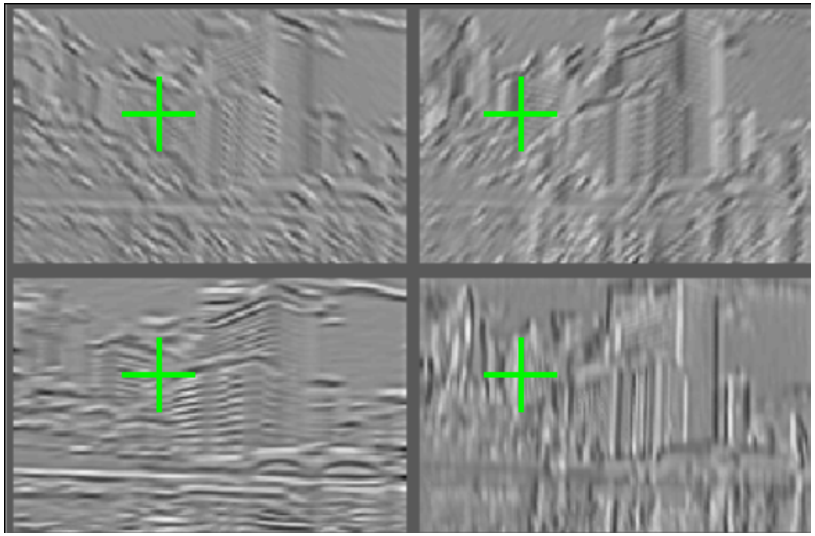
Normalization

- Contrast normalization (across feature maps)

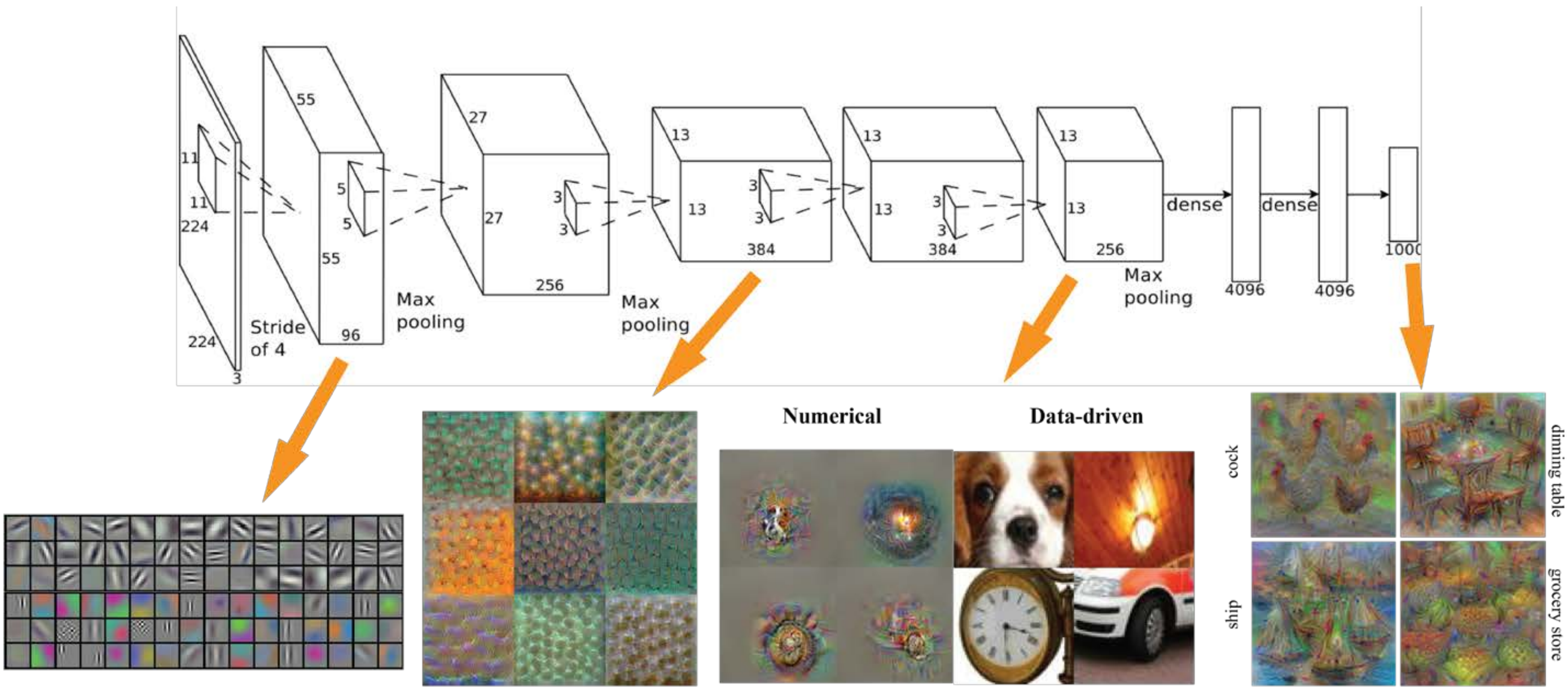
feature map



feature map after normalization



AlexNet Architecture



Conv 1: Edge+Blob **Conv 3: Texture** **Conv 5: Object Parts** **Fc8: Object Classes**

http://vision03.csail.mit.edu/cnn_art/

Summary

- Feedforward neural networks can be trained effectively using SGD
- Back-propagation
  - relies on applying chain rule to compute error gradient with respect to the network parameters
- CNN – neural network with a specialized connectivity structure
  - state-of-the-art for many computer vision problems