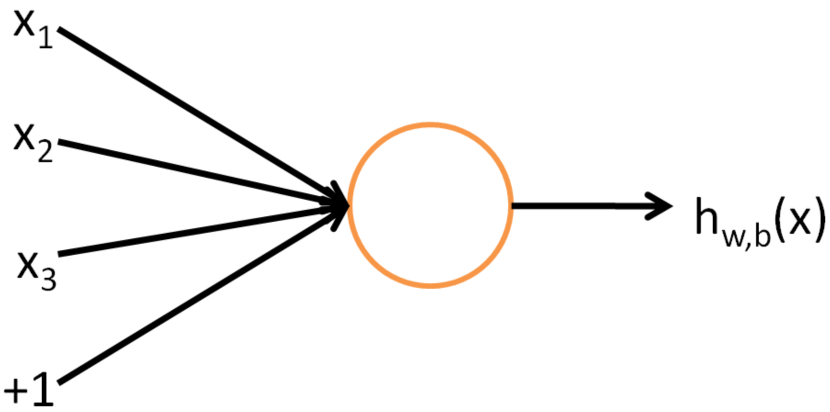
# Neural Networks

University of Santiago de Compostela

Manuel Mucientes

# Introduction

- Basic idea: linear combinations + nonlinear function
- An example of a single unit (neuron):



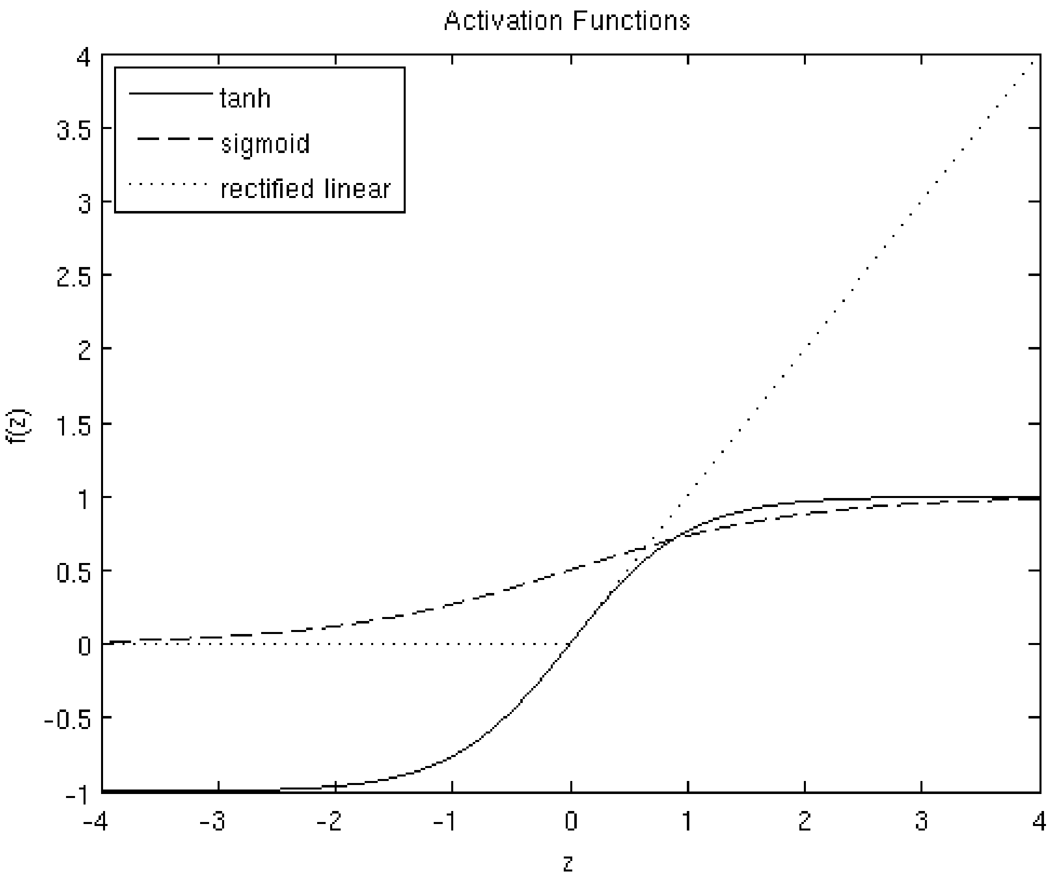
- $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$

# Activation functions

- Sigmoid (also known as standard logistic function): [0, 1]

$$f(z) = \frac{1}{1 + \exp(-z)}$$

- Derivative:
  - $f'(z)=f(z)(1-f(z))$

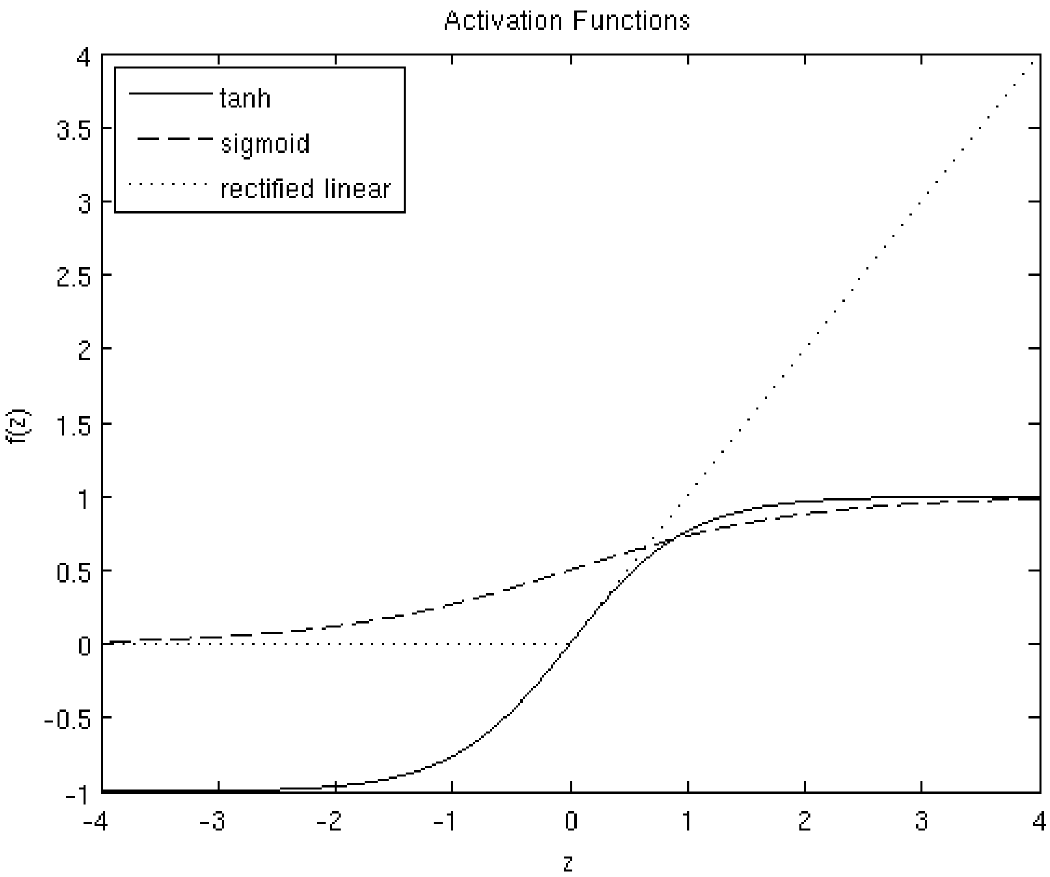


# Activation functions

- Hyperbolic tangent: [-1, 1]

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

  - Derivative:
    - $f'(z)=1-(f(z))^2$



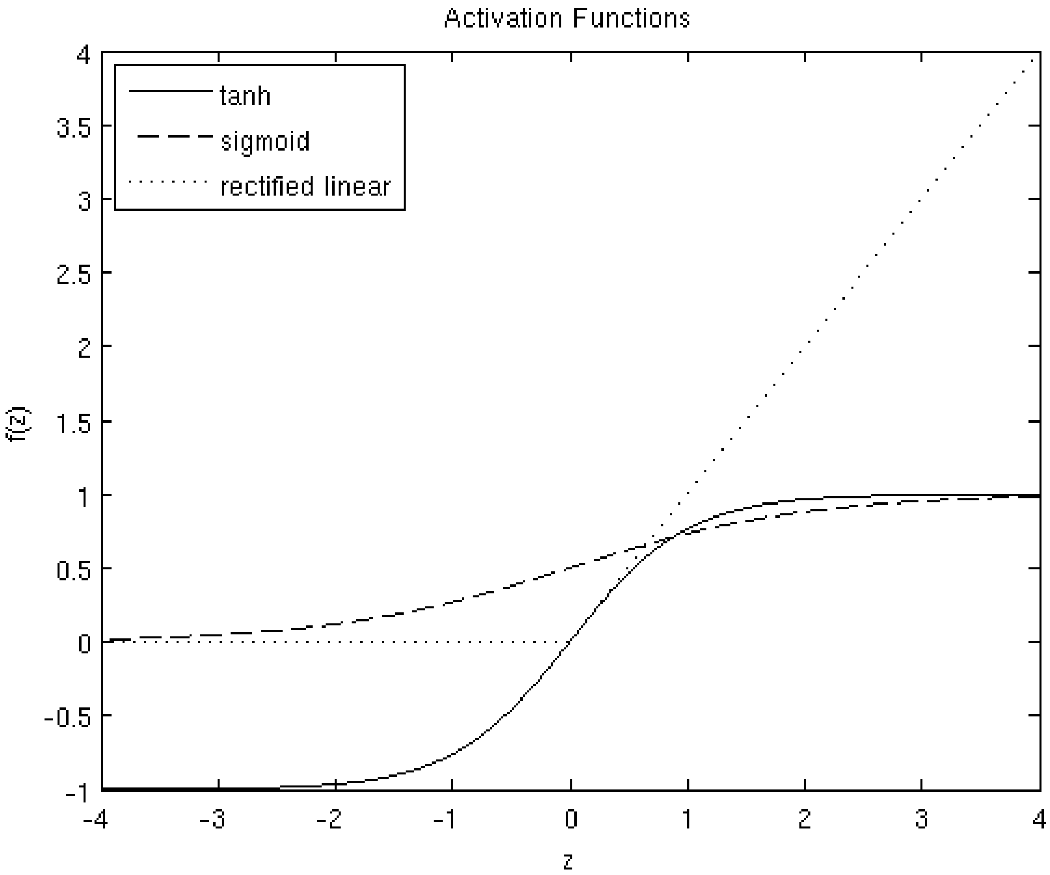
# Activation functions

- Rectified linear unit (ReLU):

  $f(z) = \max(0, z)$

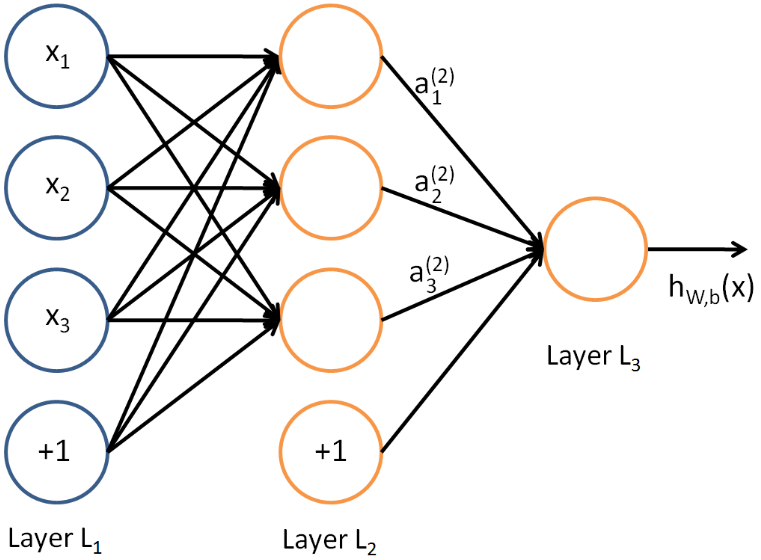
  - Derivative:
    - 0 if z<0, 1 otherwise
    - Undefined at z=0
      - Average the gradient over many training examples during optimization

- Gaussian radial basis functions
  - RBF networks
  - $f(z) = \exp(-(1/2)(z-c)^T \Sigma^{-1}(z-c))$



# Neural Network Model

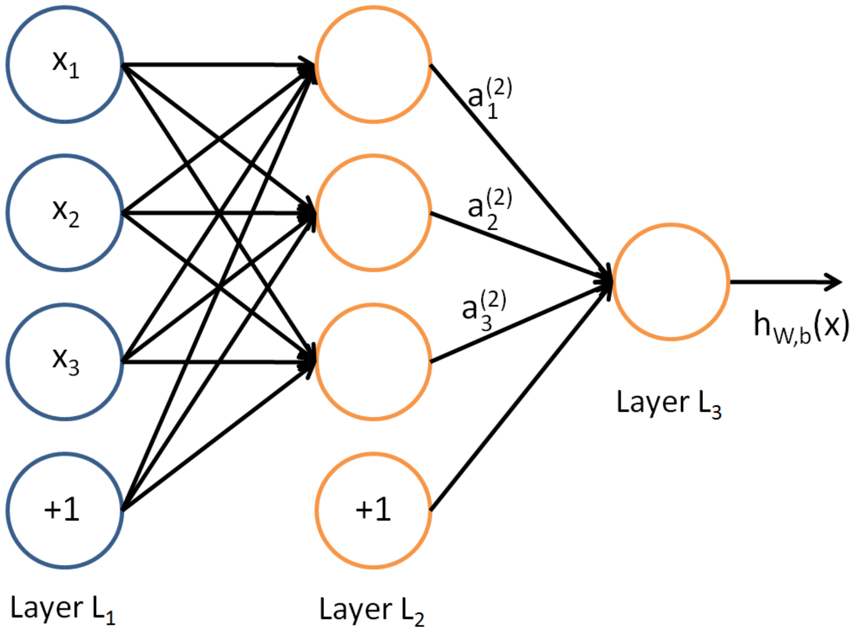
- A single hidden layer feed-forward neural network:



- Input layer, hidden layer, output layer
- Parameters: $(W,b)=(W^{(1)},b^{(1)},W^{(2)},b^{(2)})$
  - $W_{ij}^{(l)}$: weight associated with the connection between unit $j$ in layer $l$ and unit $i$ in layer $l$+1
    - $W^{(1)}\in R^{3\times3}$, and $W^{(2)}\in R^{1\times3}$
  - $b_i^{(l)}$ is the bias associated with unit $i$ in layer $l$+1
    - $b^{(1)}\in R^{3\times1}$, and $b^{(2)}\in R^{1\times1}$

# Neural Network Model

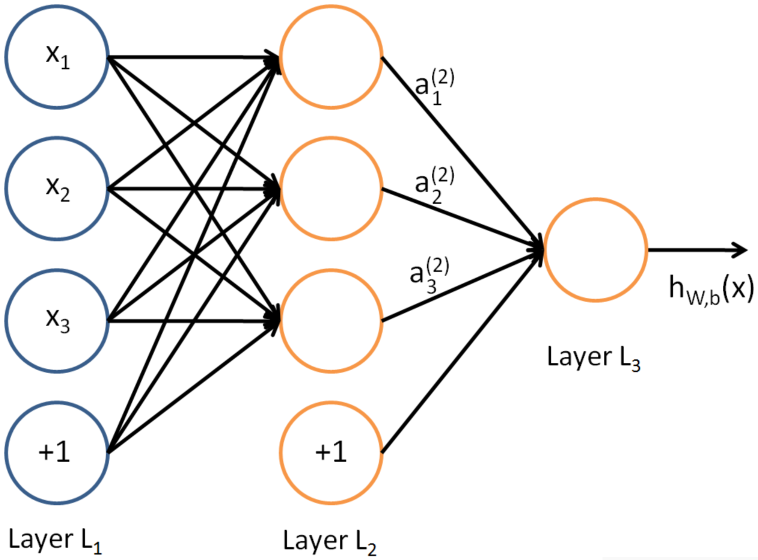
- $a_i^{(l)}$: activation (output value)
  - $a_i^{(1)}=x_i$
- $s_l$: number of nodes in layer $l$ (not counting the bias unit)
- $$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$$



- $a_i^{(l)}=f(z_i^{(l)})$
- Computation:

$$\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) = a_1^{(3)} &= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}$$

# Neural Network Model



- Computation: forward propagation

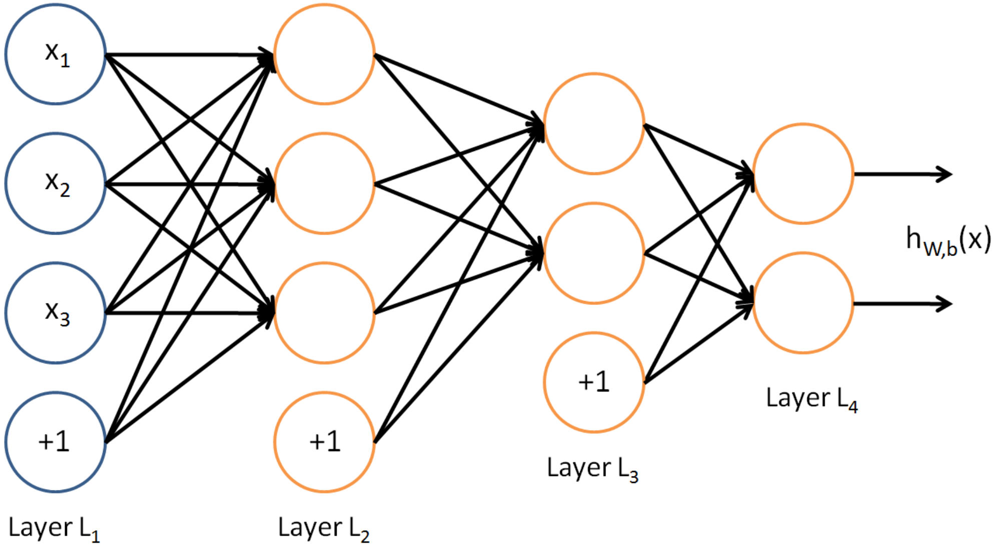
$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

- In general (matrix-vector operations):

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
a^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

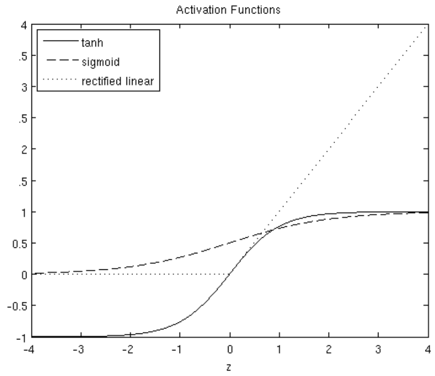
# Architectures

- Patterns of connectivity between neurons
  - Multiple hidden layers
  - Fully (densely) vs. locally connected
  - Weight sharing (locally connected)
  - feed-forward (no loops)
- Most common choice: multilayer feed-forward neural network (multilayer perceptron network)
  - Forward propagation step to calculate outputs of each layer



# Architectures

- Regression:
  - One output per output variable
  - The output function is typically the identity function
  - Data standardization: scale the outputs (and inputs), as range depends on the activation function
- Classification:
  - For K-class classification, K units in the output layer
    - One-hot encoding: E.g. $\begin{array}{cccc} \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, & \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, & \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, & \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \\ \text{pedestrian} & \text{car} & \text{motorcycle} & \text{truck} \end{array}$
    - Output function: *softmax* $$f(z_i^{(l)}) = \frac{e^{z_i^{(l)}}}{\sum_{j=1}^{K} e^{z_j^{(l)}}}$$
  - For binary classification: typically, one unit
    - Output function: sigmoid [0, 1]



# Architectures

- An example of the output of *softmax*
  - Three categories (K=3): bike, car, truck

$$f(z_i^{(l)}) = \frac{e^{z_i^{(l)}}}{\sum_{j=1}^{K} e^{z_j^{(l)}}}$$

| | $z_i$ | $\exp(z_i)$ | $f(z_i)$ | Correct probs. |
|---|---|---|---|---|
| Bike | -0.2 | 0.8 | 0.02 (2%) | 0.00 |
| Car | **3.6** | **36.6** | **0.75 (75%)** | **1.00** |
| Truck | 2.4 | 11.0 | 0.23 (23%) | 0.00 |

© Manuel Mucientes

# Error functions

- Training examples: $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- Maximize likelihood on training data:
  - Each data point drawn independently from same distribution

$$\mathcal{P} = \prod_{i=1}^{m} p\left(x_i, y_i\right) = \prod_{i=1}^{m} p\left(y_i | x_i\right) p\left(x_i\right)$$

- Minimize the negative log. of likelihood: error function

$$J = -\log \mathcal{P} = -\sum_{i=1}^{m} \log p\left(y_i | x_i\right) - \log \sum_{i=1}^{m} p\left(x_i\right)$$

$$J = -\sum_{i=1}^{m} \log p\left(y_i | x_i\right)$$

# Error functions

- Sum of squares error: regression problems
  - Assumption: Gaussian distributed target data

$$p\left(y|x\right)=\prod_{i=1}^{m}\frac{1}{\left(2\pi\sigma^{2}\right)^{(1/2)}}\exp-\frac{\left(h\left(x_{i}\right)-y_{i}\right)^{2}}{2\sigma^{2}}$$

  - Error: -log and eliminate terms independent of weights

$$J=\frac{1}{2}\sum_{i=1}^{m}\left(h\left(x_{i}\right)-y_{i}\right)^{2}$$

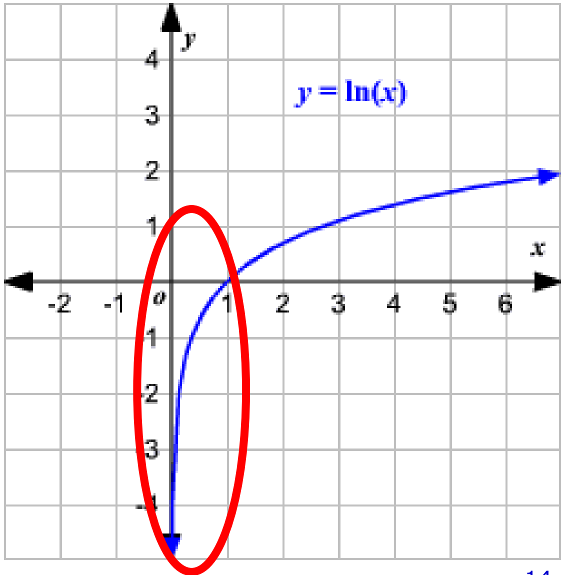
# Error functions

- Binary cross-entropy: binary classification problems
  - Single output unit: $p\left(\mathcal{C}_1|x_i\right) = h\left(x_i\right) \quad p\left(\mathcal{C}_2|x_i\right) = 1 - h\left(x_i\right)$
  - Sigmoid output: output interpreted as probability
  - $p\left(y_i|x_i\right) = \left(h\left(x_i\right)\right)^{y_i}\left(1 - h\left(x_i\right)\right)^{1-y_i}$

$$J = -\sum_{i=1}^{m}\left\{y_i \log h\left(x_i\right) + \left(1 - y_i\right)\log\left(1 - h\left(x_i\right)\right)\right\}$$

- Minimum of the error: $h\left(x_i\right) = y_i$

$$\frac{\partial J}{\partial h\left(x_i\right)} = \frac{h\left(x_i\right) - y_i}{h\left(x_i\right)\left(1 - h\left(x_i\right)\right)}$$



# Error functions

- Cross-entropy: K-class classification with mutually exclusive classes
  - $p\left(\mathcal{C}_k|x_i\right) = h^k\left(x_i\right)$

    $$p\left(y_i|x_i\right) = \prod_{k=1}^{K}\left(h^k\left(x_i\right)\right)^{y_i^k}$$

  - Error function:

    $$J = -\sum_{i=1}^{m}\sum_{k=1}^{K} y_i^k \log h^k\left(x_i\right)$$

    - Minimum of the error: $h^k\left(x_i\right) = y_i^k$
  - Output values interpreted as probabilities: *softmax* output

# Error functions

- Example:

$$J = -\sum_{i=1}^{m}\sum_{k=1}^{K} y_i^k \log h^k\left(x_i\right)$$

softmax(h$^k$)

| | |
|---|---|
| Bike | 0.02 |
| Car | **0.75** |
| Truck | 0.23 |

- k=1 (bike): 0*log(0.02) = 0.0
- k=2 (car): 1*log(0.75) = -0.29
- k=3 (truck): 0*log(0.23) = 0.0
- J = - [0.0 – 0.29 – 0.0] = 0.29

# Error functions

- Example: $$J = -\sum_{i=1}^{m}\sum_{k=1}^{K} y_i^k \log h^k\left(x_i\right)$$

| | softmax(h) | J | |
|---|---|---|---|
| | {0.02, **0.75**, 0.23} | 0.29 | Good classification |
| | {0.00, **1.00**, 0.00} | 0.00 | Perfect classification |
| | {0.00, **0.75**, **0.25**} | 1,39 | Wrong classification |
| | {0.00, **1.00**, **0.00**} | large number | Totally wrong classif. |

# Error functions

- Cross-entropy: K-class classification with **not** mutually exclusive classes
  - Assumption: independent categories

$$p\left(y_i|x_i\right) = \prod_{k=1}^{K} \left(h^k\left(x_i\right)\right)^{y_i^k} \left(1 - h^k\left(x_i\right)\right)^{1-y_i^k}$$

  - Error function:

$$J = -\sum_{i=1}^{m} \sum_{k=1}^{K} y_i^k \log h^k\left(x_i\right) + \left(1 - y_i^k\right) \log\left(1 - h^k\left(x_i\right)\right)$$

    - Minimum of the error: $h^k\left(x_i\right) = y_i^k$
  - Sigmoid output: output interpreted as probability

# Fitting Neural Networks

- Overall cost function:

$$J\left(W,b\right)=\frac{1}{m}\sum_{e=1}^{m}J\left(W,b;x^{(e)},y^{(e)}\right)+\lambda R\left(W\right)$$

  - Regularization term (weight decay) not applied to the bias terms
  - When R is L2:

$$J\left(W,b\right)=\frac{1}{m}\sum_{e=1}^{m}J\left(W,b;x^{(e)},y^{(e)}\right)+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

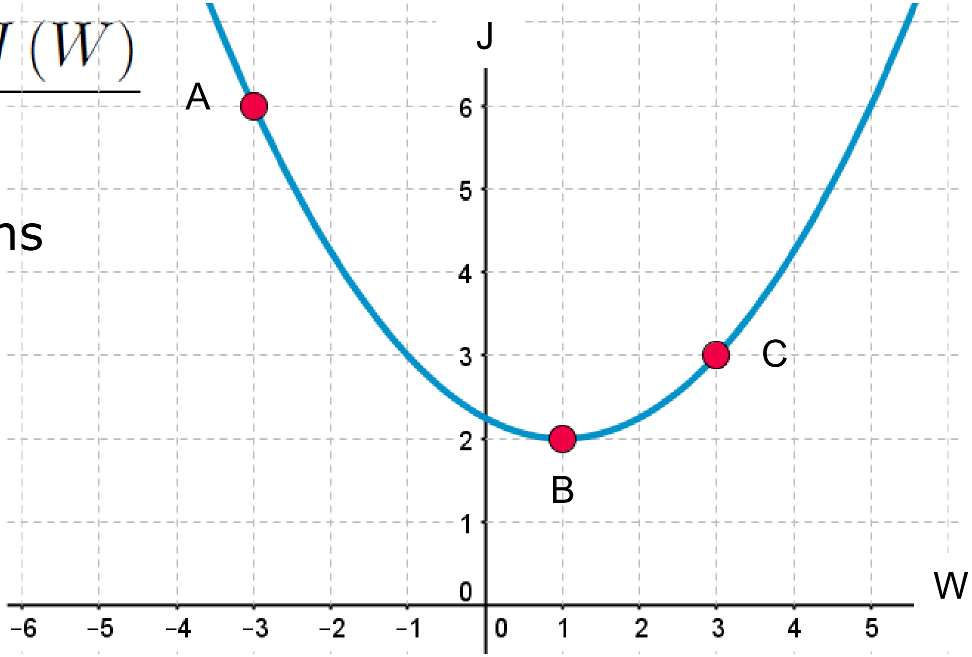
- Minimize *J* through backpropagation
  - *J* is non-convex: local minima
  - Gradient descent: usually works fairly well

# Fitting Neural Networks

- Find W that minimizes the loss function (J)
  - 1-dimensional example
- Radom search:
  - Huge search space
  - W: high dimensionality in deep learning
- Numerical gradient:

$$\frac{dJ(W)}{dW} = \lim_{\epsilon \to 0} \frac{J(W+\epsilon) - J(W)}{\epsilon}$$

  - Slow: loop over all dimensions
  - Approximate



© Manuel Mucientes

# Fitting Neural Networks

no entra

- Multidimensional W: gradient in each dimension
- Example:
  - W = (0.23, -0.74, 0.13), loss = 1.78961
    - First dimension: W + epsilon = = (0.23+0.0001, -0.74, 0.13), loss = 1.78950
      - Gradient = (1.78950 - 1.78961)/0.0001 = -1.1
    - Second dimension: W + epsilon = = (0.23, -0.74+0.0001, 0.13), loss = 1.78969
      - Gradient = (1.78969 - 1.78961)/0.0001 = 0.8
    - Third dimension: W + epsilon = = (0.23, -0.74, 0.13+0.0001), loss = 1.78937
      - Gradient = (1.78937 - 1.78961)/0.0001 = -2.4
  - Gradient = (-1.1, 0.8, -2.4)
    - Increase $W_1$ and $W_3$, decrease $W_2$

# Fitting Neural Networks

- Analytic gradient: exact and fast

$$J\left(W,b\right)=\frac{1}{m}\sum_{e=1}^{m}J\left(W,b;x^{(e)},y^{(e)}\right)+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

$$\frac{\partial J\left(W,b\right)}{\partial W_{ij}^{(l)}}=\left[\frac{1}{m}\sum_{e=1}^{m}\frac{\partial J\left(W,b;x^{(e)},y^{(e)}\right)}{\partial W_{ij}^{(l)}}\right]+\lambda W_{ij}^{(l)}$$

$$\frac{\partial J\left(W,b\right)}{\partial b_i^{(l)}}=\frac{1}{m}\sum_{e=1}^{m}\frac{\partial J\left(W,b;x^{(e)},y^{(e)}\right)}{\partial b_i^{(l)}}$$

- Compute the partial derivatives: backpropagation algorithm

# Fitting Neural Networks

- One iteration of gradient descent: Widrow-Hoff rule

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W,b)$$

  - $\alpha$: learning rate

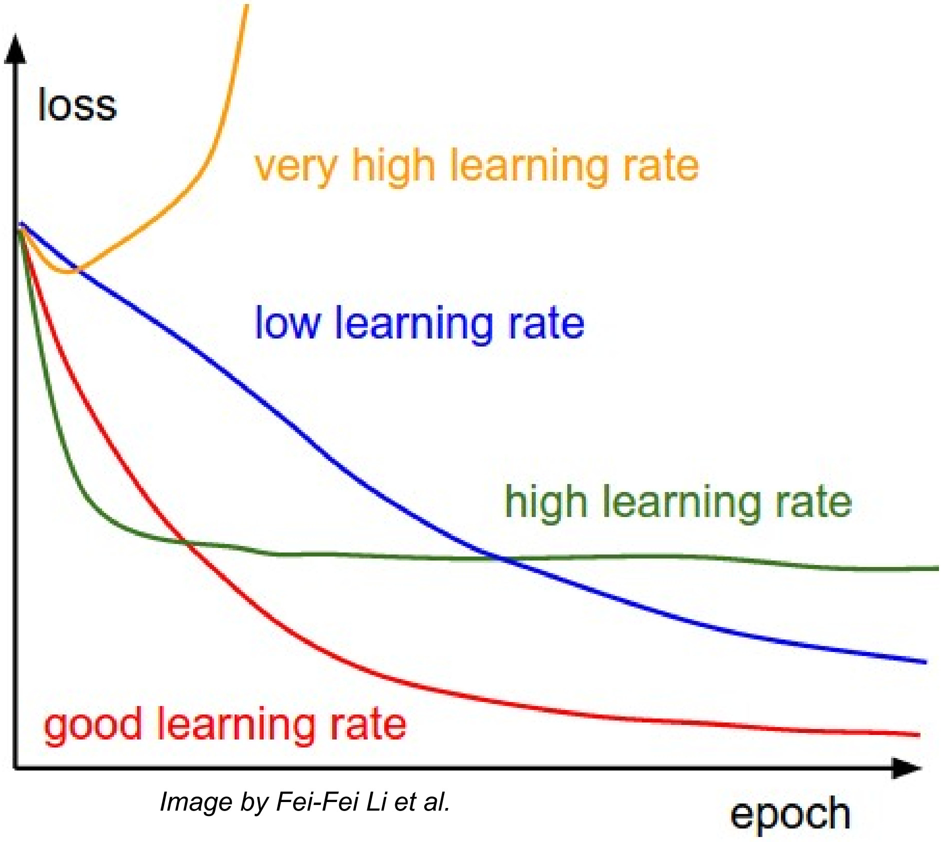
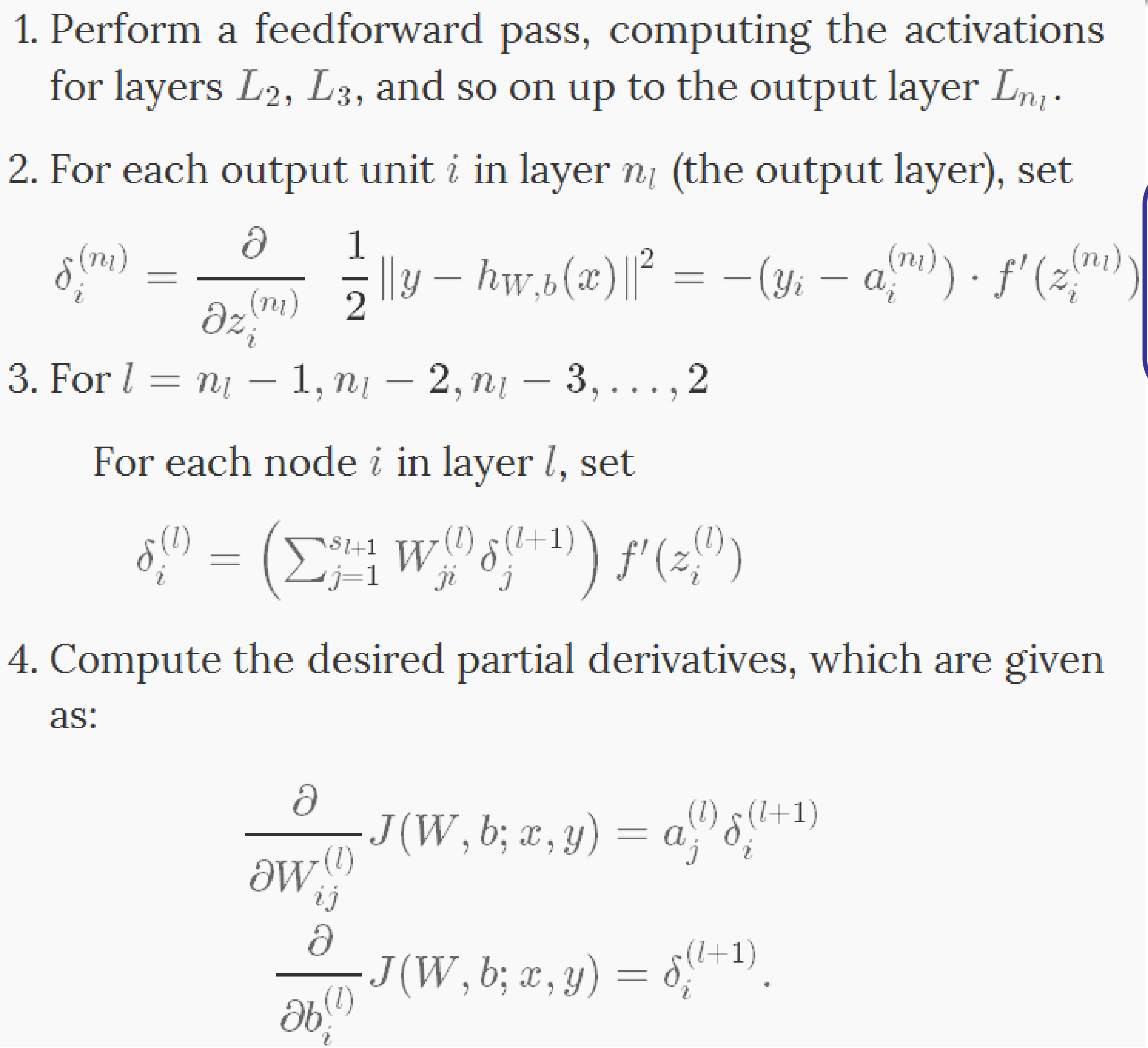


*Image by Fei-Fei Li et al.*

# Backpropagation algorithm



1. Perform a feedforward pass, computing the activations for layers $L_2$, $L_3$, and so on up to the output layer $L_{n_l}$.

2. For each output unit $i$ in layer $n_l$ (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \; \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node $i$ in layer $l$, set

$$\delta_i^{(l)} = \left( \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}.$$

When error function is squared error. Replace with adequate error func.

# Backpropagation algorithm

- $\delta_i^{(l)} \equiv \dfrac{\partial J\left(W,b,x,y\right)}{\partial z_i^{(l)}}$
- For each output unit:
  - For the squared error function:

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}\left\|y - h_{W,b}(x)\right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

- For each node in layer *l*:

$$\delta_i^{(l)} \equiv \frac{\partial J\left(W,b,x,y\right)}{\partial z_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \frac{\partial J\left(W,b,x,y\right)}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \delta_j^{(l+1)} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}}$$

# Backpropagation algorithm

- For each node in layer *l*:
  - Given: $a_i^{(l)} = f\left(z_i^{(l)}\right)$

$$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i(l) = \sum_{j=1}^{s_l} W_{ij}^{(l)} f\left(z_j^{(l)}\right) + b_i(l)$$

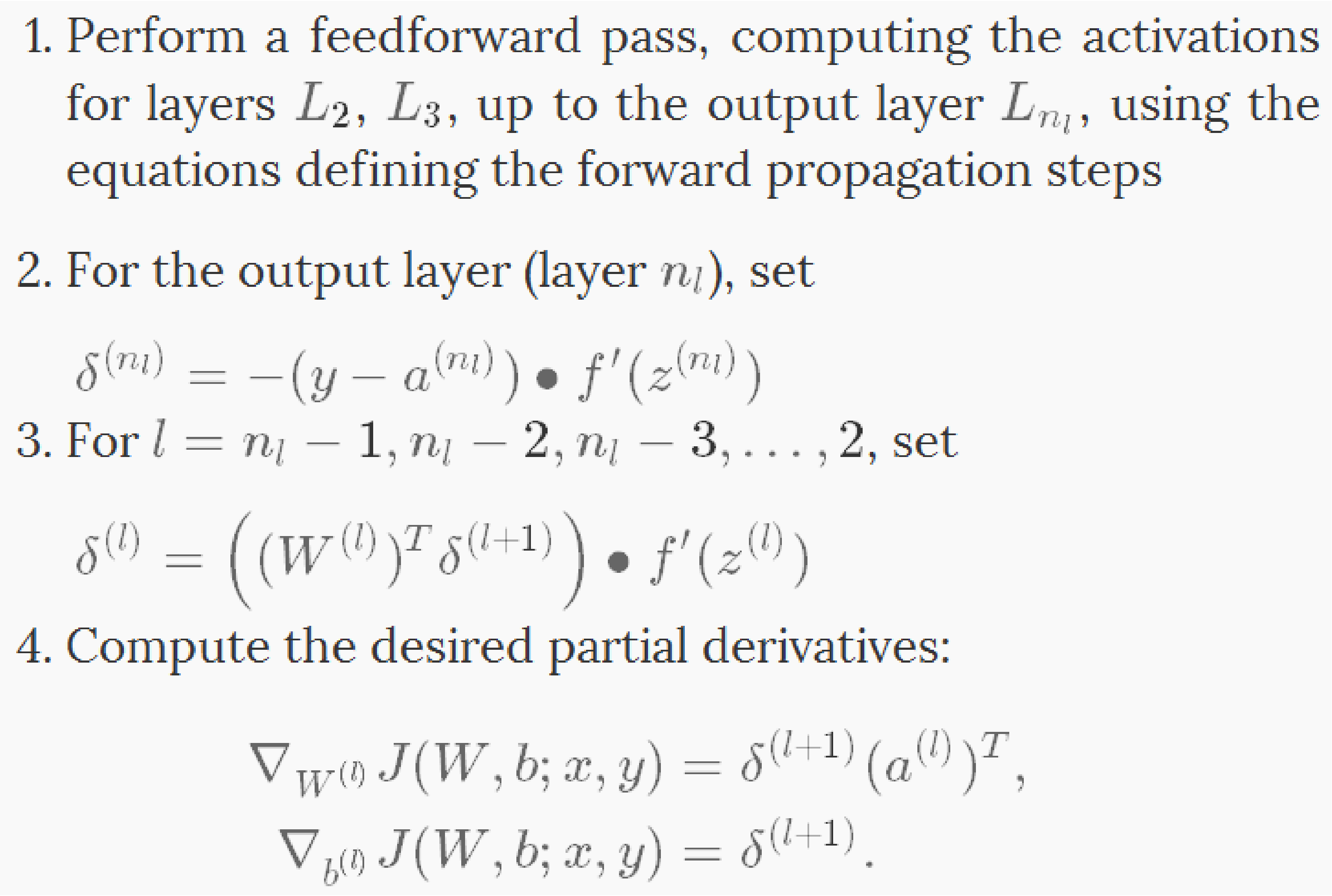
  - Then: $\dfrac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = W_{ji}^{(l)} f'\left(z_i^{(l)}\right)$

$$\delta_i^{(l)} = \sum_{j=1}^{s_{l+1}} \delta_j^{(l+1)} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = f'\left(z_i^{(l)}\right) \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}$$

- Finally: $\dfrac{\partial J(W,b,x,y)}{\partial W_{ij}^{(l)}} = \dfrac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}} \dfrac{\partial J(W,b,x,y)}{\partial z_i^{(l+1)}} = a_j^{(l)} \delta_i^{(l+1)}$

# Backpropagation algorithm

- Matrix-vectorial notation



1. Perform a feedforward pass, computing the activations for layers $L_2$, $L_3$, up to the output layer $L_{n_l}$, using the equations defining the forward propagation steps

2. For the output layer (layer $n_l$), set

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

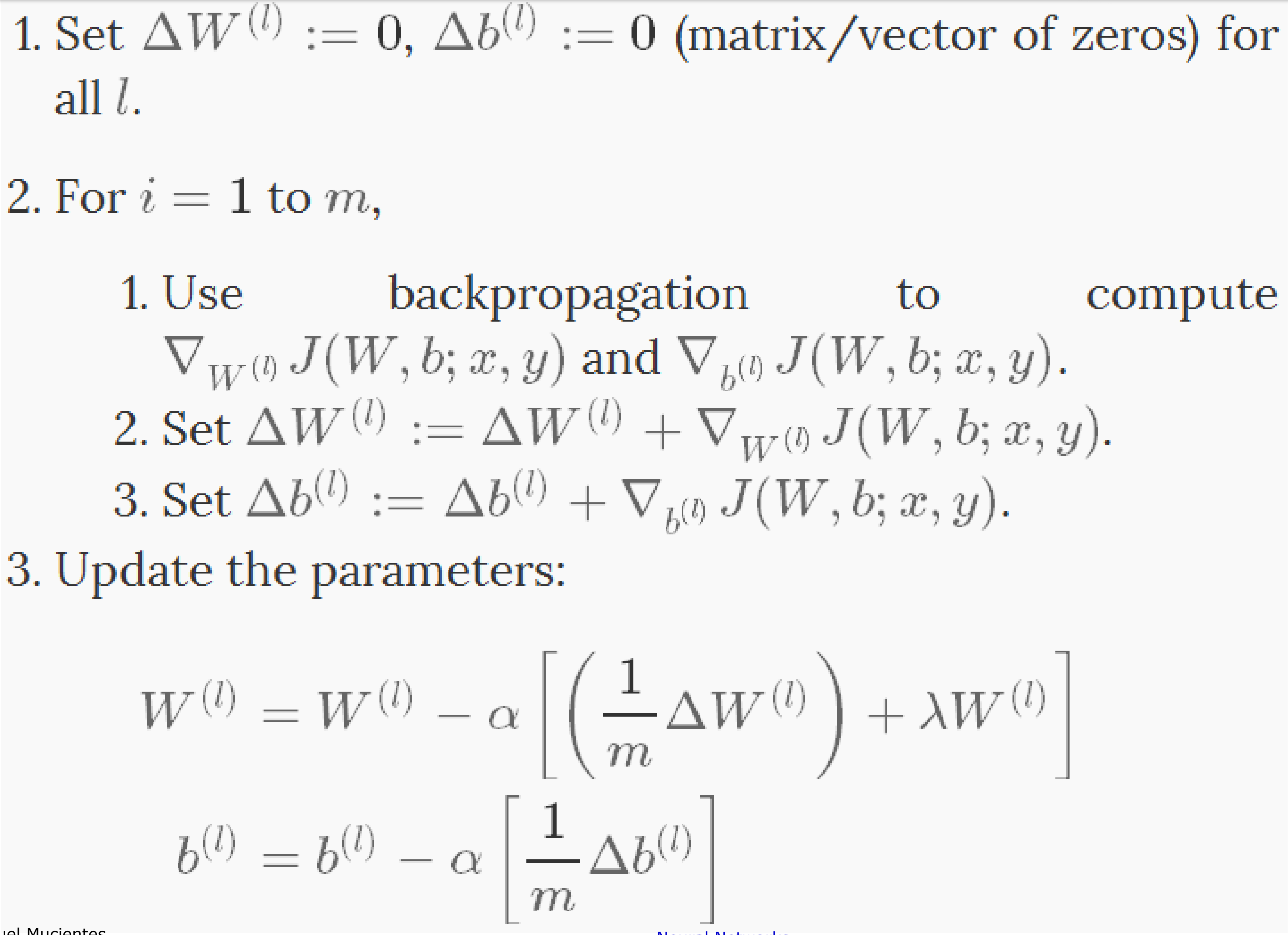
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives:

$$\begin{aligned}\nabla_{W^{(l)}} J(W,b;x,y) &= \delta^{(l+1)} (a^{(l)})^T, \\ \nabla_{b^{(l)}} J(W,b;x,y) &= \delta^{(l+1)}.\end{aligned}$$

# Full NN learning algorithm



1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all $l$.

2. For $i = 1$ to $m$,
    1. Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.
    2. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.
    3. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.

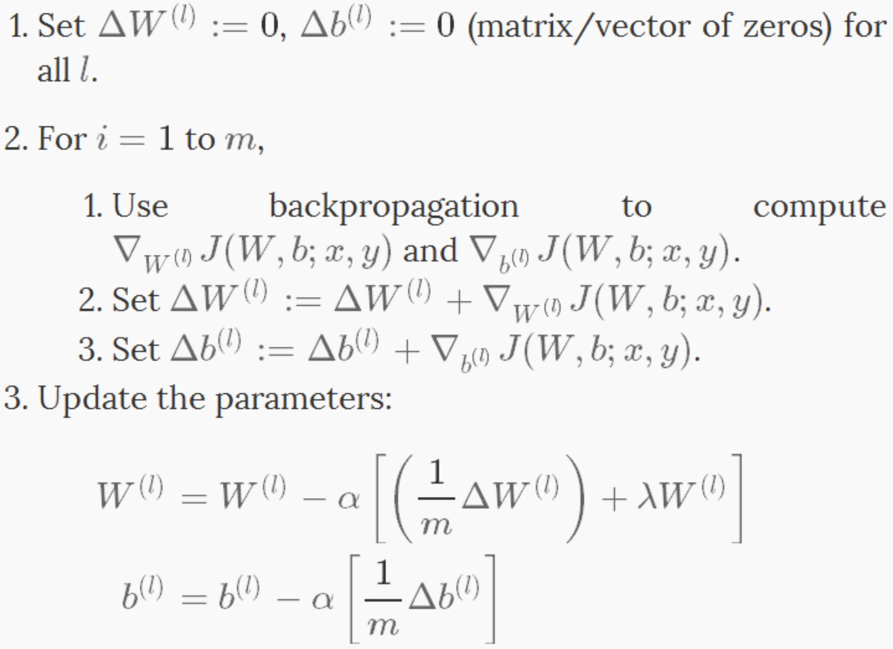
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[ \left( \frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[ \frac{1}{m} \Delta b^{(l)} \right]$$

# Full NN learning algorithm

- Batch learning: 1 epoch = 1 iteration
- Minibatch learning
  - Approximate the sum with a minibatch of examples
  - Extreme case: batch size =1; 1 epoch = m iterations
- Learning rate should be modified according to batch size
  - Change in W equivalent for 1 epoch



```
1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all $l$.
2. For $i = 1$ to $m$,
    1. Use backpropagation to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ and $\nabla_{b^{(l)}} J(W,b;x,y)$.
    2. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y)$.
    3. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y)$.
3. Update the parameters:
    $W^{(l)} = W^{(l)} - \alpha \left[ \left( \frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$
    $b^{(l)} = b^{(l)} - \alpha \left[ \frac{1}{m} \Delta b^{(l)} \right]$
```

# Training Neural Networks

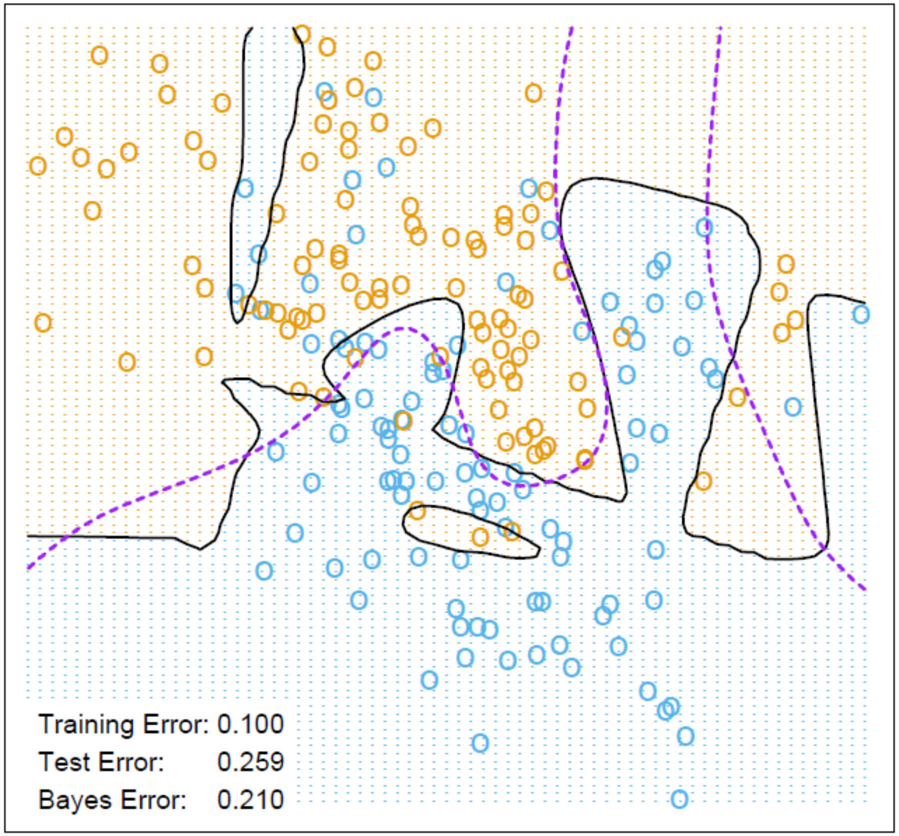
- Over-parametrized model, non-convex and unstable optimization problem

- Starting values:
  - Random values near 0 for W, biases to 0
    - With standardized inputs, random uniform weights in [-0.7, 0.7]
      - Ok for small networks
    - Xavier initialization: random() * sqrt(1/n)
      - random(): mean 0, variance 1
      - n: number of inputs
  - Model starts out nearly linear, and becomes nonlinear
  - 0 weights give perfect symmetry, large weights lead poor solutions

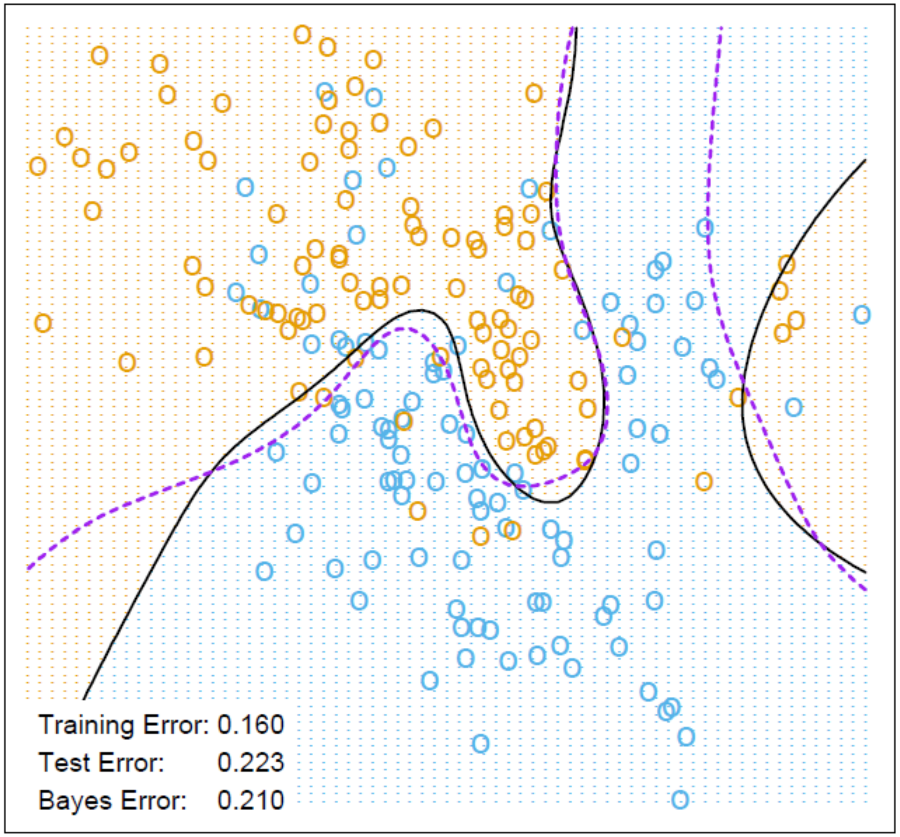
# Training Neural Networks

- Overfitting: regularization term (weight decay)
  - Cross-validation to estimate the regularization parameter

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



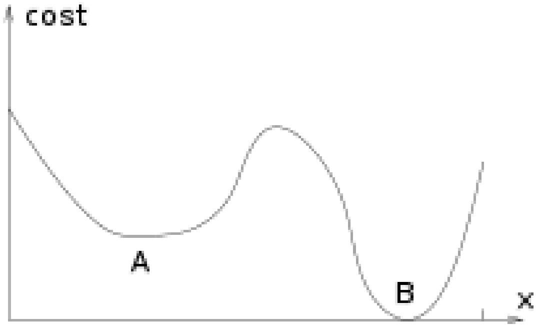
# Training Neural Networks

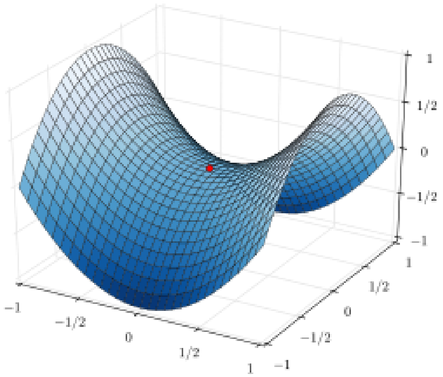
- Scaling of the inputs:
  - Can have a large effect in the quality of the final solution: scaling of the weights
  - Standardize inputs (and outputs):
    - Treat all inputs equally: regularization
    - Choose a meaningful range for the starting weights

- Number of hidden units:
  - Better to have too many hidden units than too few
  - Typical number of hidden units: 5 to 100 (per layer)
    - Number increasing with number of inputs and number of training cases
    - Same number in every layer (reasonable default)

# Training Neural Networks

- Number of hidden layers: construction of hierarchical features
  - background knowledge and experimentation
- Multiple minima: non-convex error function
  - Try a number of random starting configurations: choose the best solution
  - Better approach: average the predictions over a collection of networks
  - Another approach: bagging
- Problems with gradient descent: advanced optimization alg.
  - Loss changes quickly in one direction and slowly in another
  - Local minima/saddle points: common in high dimensions
  - Noisy gradients: come from minibatches





# Training Neural Networks

- Learning rate is a fundamental hyperparameter
- Learning rate decay:
  - Step decay:
    - E.g., decay by 10 every few epochs
  - Exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

  - 1/t decay:

$$\alpha = \frac{\alpha_0}{1 + kt}$$

  - Learning rate schedule

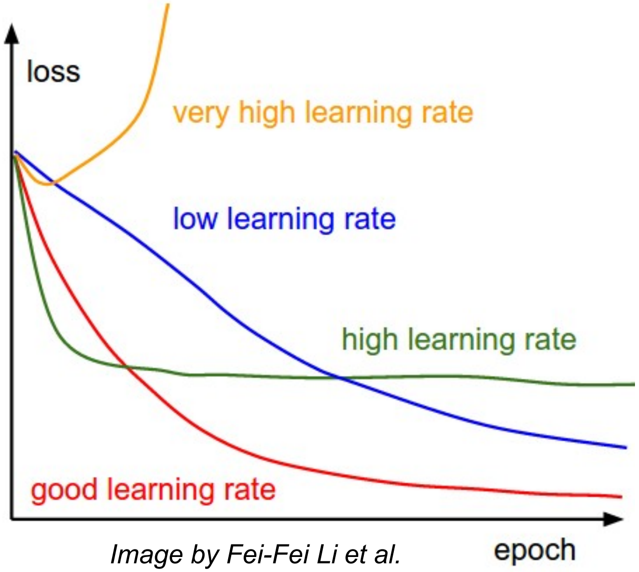


*Image by Fei-Fei Li et al.*

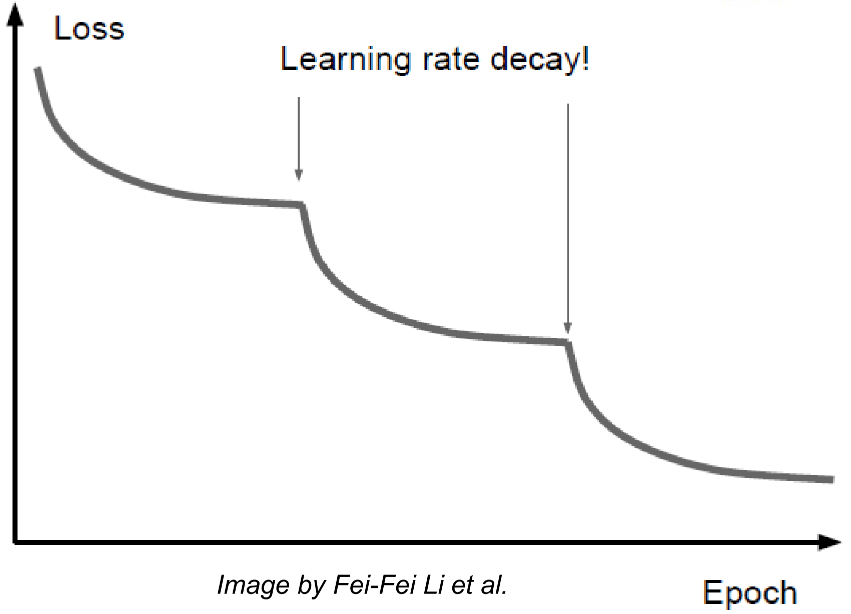


*Image by Fei-Fei Li et al.*

# Hyperparameters setting

- Coarse search: few epochs (log space)
- Finer search: higher number of epochs
- Monitor and visualize training and validation loss

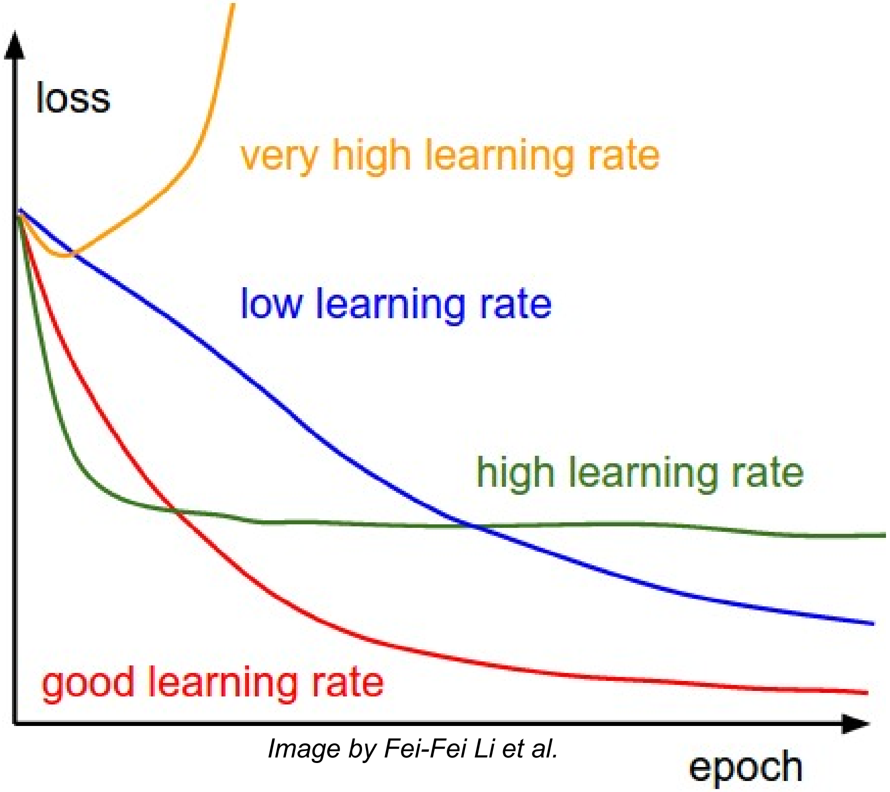
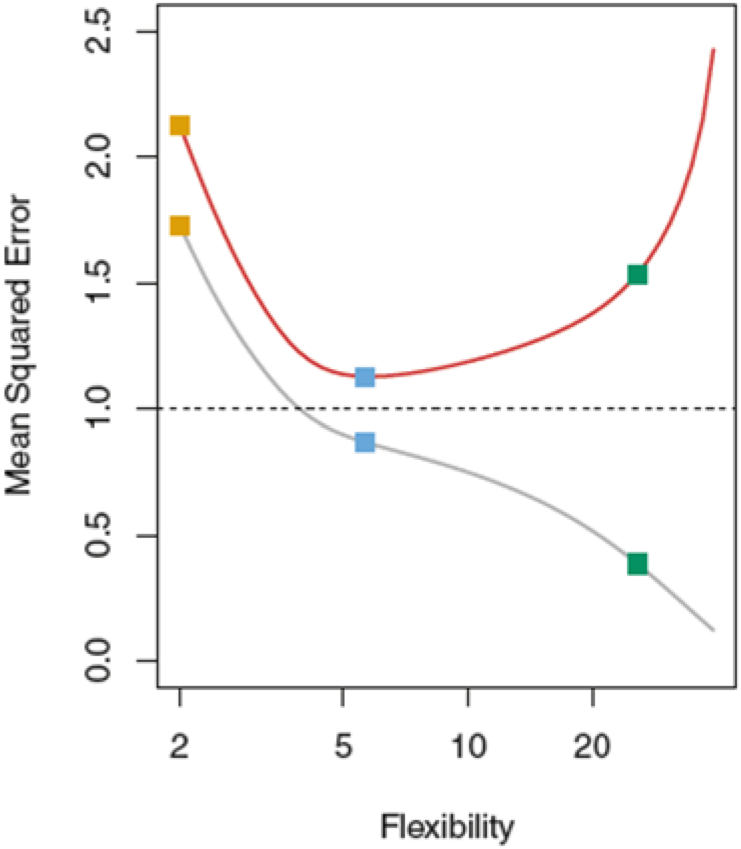


*Image by Fei-Fei Li et al.*



# Optimization

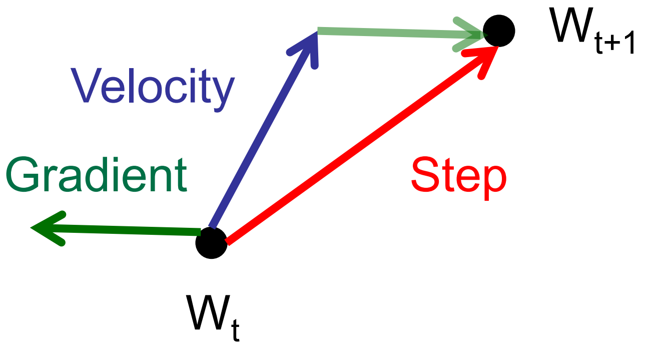
- Vanilla Stochastic Gradient Descent (SGD):

$$W_{t+1} = W_t - \alpha \nabla J\left(W_t\right)$$

- SGD with momentum:

$$v_{t+1} = \rho v_t - \alpha \nabla J\left(W_t\right)$$

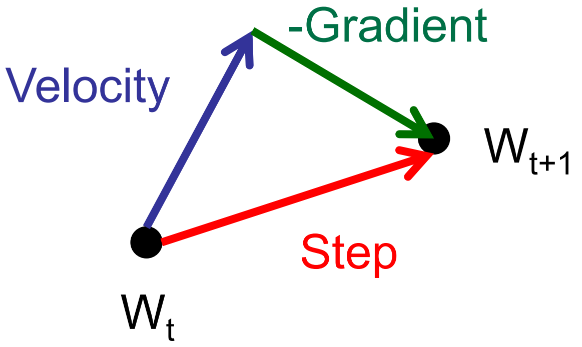
$$W_{t+1} = W_t + v_{t+1}$$



- SGD with Nesterov momentum:

$$v_{t+1} = \rho v_t - \alpha \nabla J\left(W_t + \rho v_t\right)$$

$$W_{t+1} = W_t + v_{t+1}$$



# Optimization

- AdaGrad:

$$\phi = \phi + \left(\nabla J\left(W_t\right)\right)^2$$

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\phi} + \epsilon} \nabla J\left(W_t\right)$$

  - Accelerate progress along flat directions
  - Damp progress along steep directions
  - Step size decays to zero along time
- RMSProp
  - Discard history from extreme past

$$\phi = \mu\phi + (1 - \mu)\left(\nabla J\left(W_t\right)\right)^2$$

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\phi} + \epsilon} \nabla J\left(W_t\right)$$

# Optimization

- Adam:
  - Combination of RMSProp with momentum (with distinctions)
  - Momentum incorporated as an estimate of the first-order moment
  - Bias corrections to the first and second order moments (initial moments are 0)
  - Suggested default values:
    - $\alpha$=0.001
    - $\mu_1$=0.9
    - $\mu_2$=0.999

$$\sigma = \mu_1 \sigma + (1 - \mu_1) \nabla J(W_t)$$

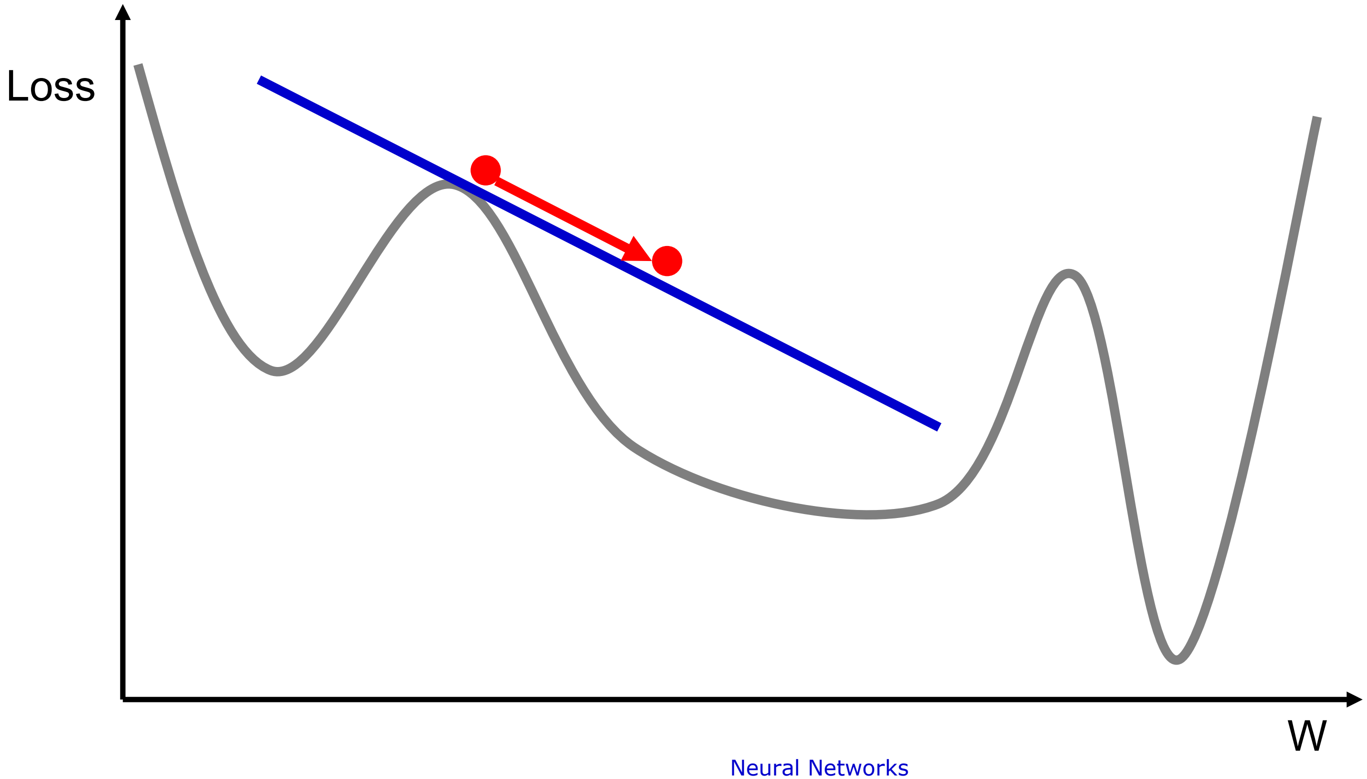
$$\phi = \mu_2 \phi + (1 - \mu_2) (\nabla J(W_t))^2$$

$$\hat{\sigma} = \frac{\sigma}{1 - \mu_1^t} \qquad \hat{\phi} = \frac{\phi}{1 - \mu_2^t}$$

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{\hat{\phi} + \epsilon}} \hat{\sigma}$$

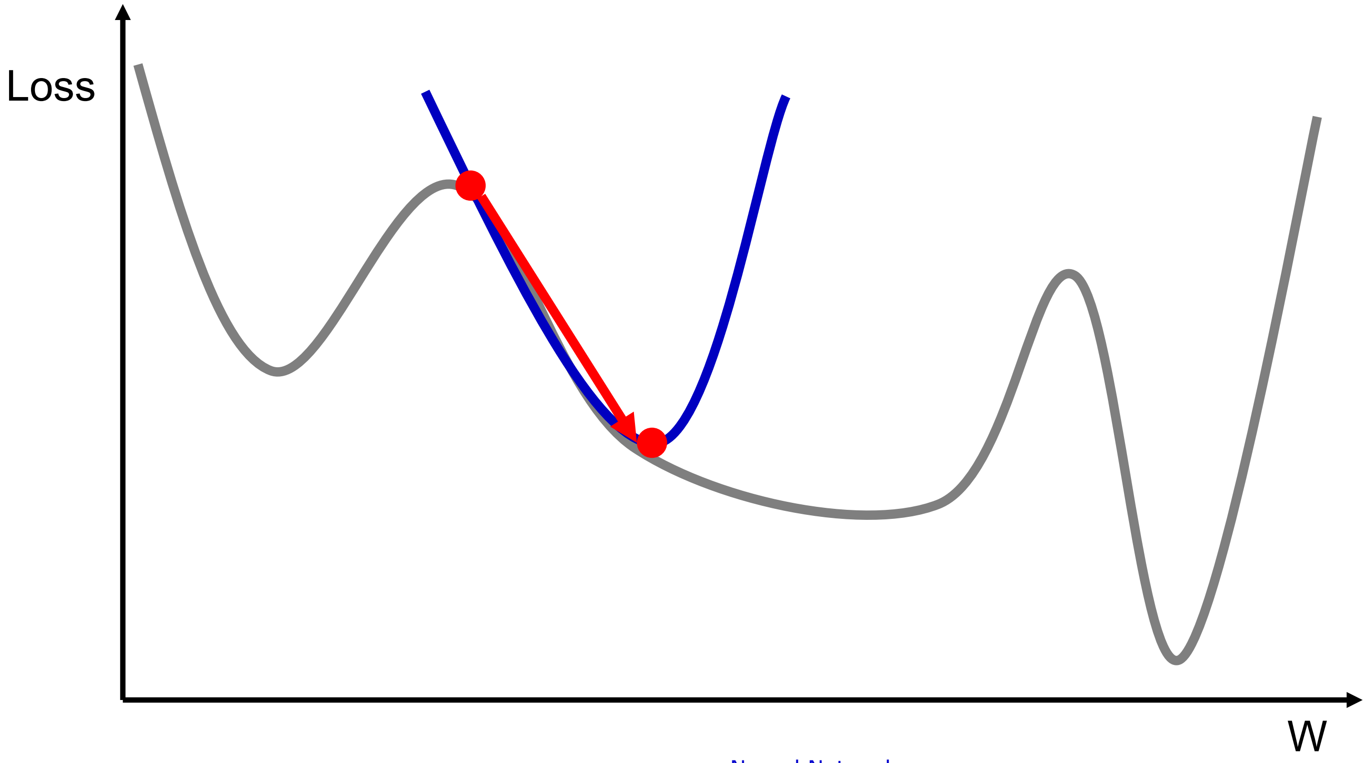
# Optimization

- First-order optimization: $W_{t+1} = W_t - \alpha \nabla J\left(W_t\right)$
  - Gradient as linear approximation
  - Step to minimize the approximation



# Optimization

- Second-order optimization:
  - Gradient and Hessian as quadratic approximation
  - Step to the minima of the approximation



© Manuel Mucientes

# Optimization

- Newton’s method:
  - Second-order Taylor expansion:

$$J\left(W_{t+1}\right) \approx J\left(W_t\right) + \left(W_{t+1} - W_t\right) \nabla J\left(W_t\right) + \frac{\left(W_{t+1} - W_t\right)^2}{2} \nabla^2 J\left(W_t\right)$$

  - Solve for critical point (derivative w.r.t the step): Newton update

$$W_{t+1} = W_t - \left(\nabla^2 J\left(W_t\right)\right)^{-1} \nabla J\left(W_t\right)$$

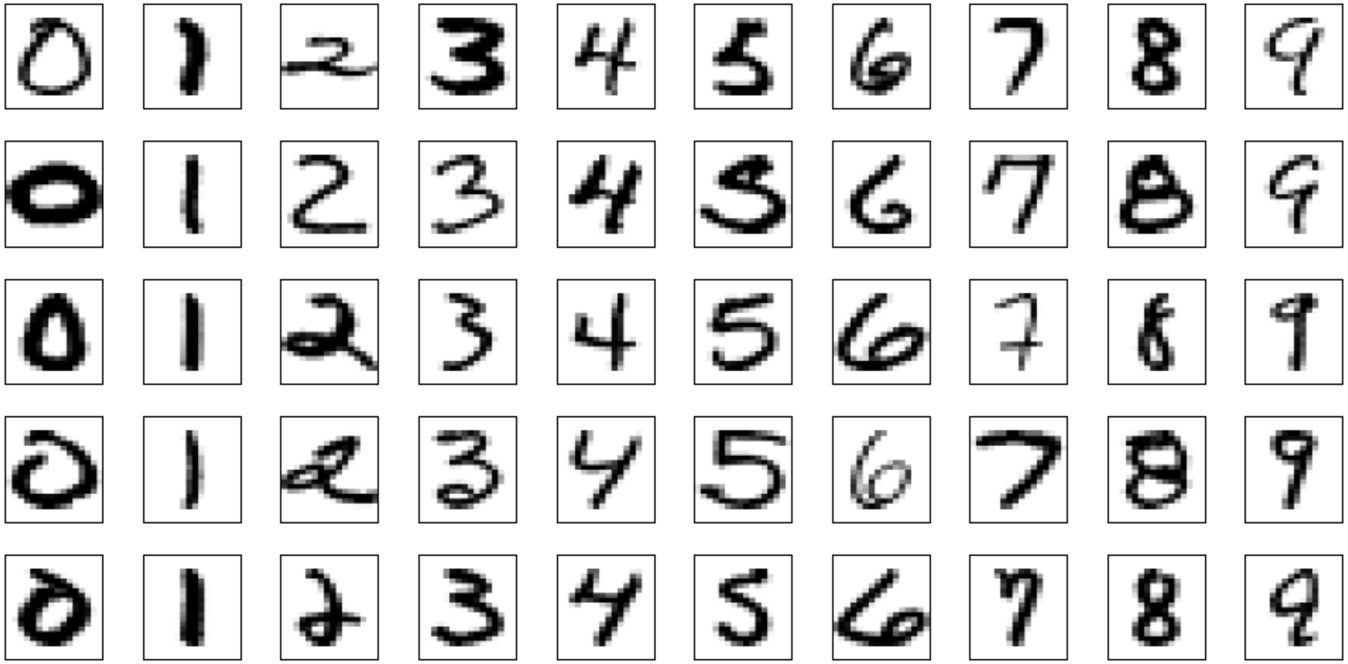
  - No hyperparameters
  - If Hessian is kxk, inverting takes O(k^3)
    - Non-viable for deep NN
- Quasi-Newton methods:
  - Approximate inverse Hessian
  - Limited memory BFGS (L-BFGS) is the most popular

# Optimization

- Practical guidelines:
  - Adam is a good choice
  - SGD with momentum with learning rate decay requires more tuning, but often outperforms Adam
  - L-BFGS does not work well with mini-batches
    - Fine for full batch

# Example: ZIP Code Data

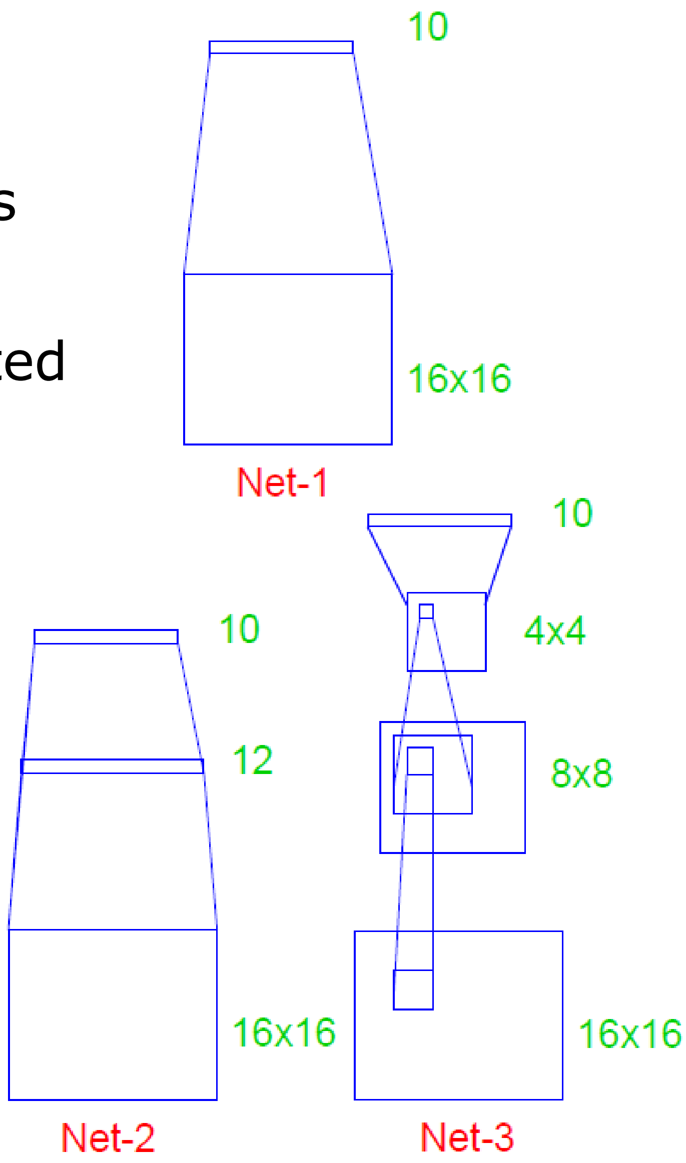
- Le Cun, 1989
- 16x16 grayscale images
- Training with 320 digits, test with 160
- Five different networks: sigmoidal output units, sum-of-squares error function



© Manuel Mucientes

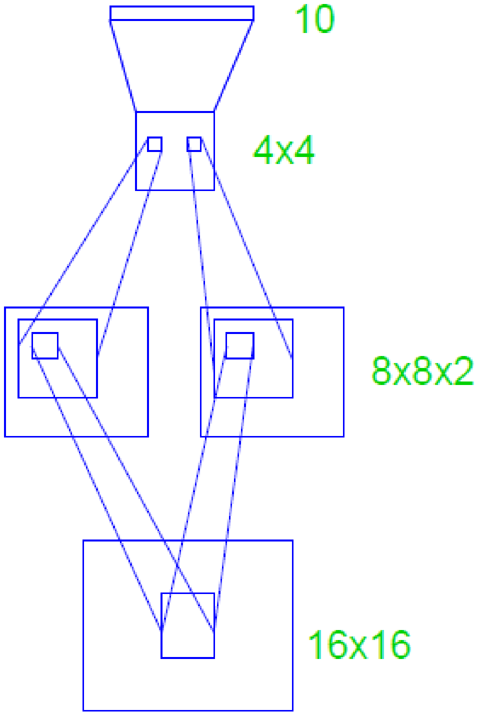
# Example: ZIP Code Data (ii)

- Net-1: no hidden layer, equivalent to multinomial logistic regression
- Net-2: one hidden layer, 12 hidden units fully connected
- Net-3: two hidden layers locally connected
  - First hidden layer:
    - Inputs from a 3x3 patch
    - Units 1 unit apart are 2 pixels apart
  - Second hidden layer:
    - Inputs from a 5x5 patch
    - Units 1 unit apart are 2 pixels apart
  - Local connectivity



# Example: ZIP Code Data (iii)

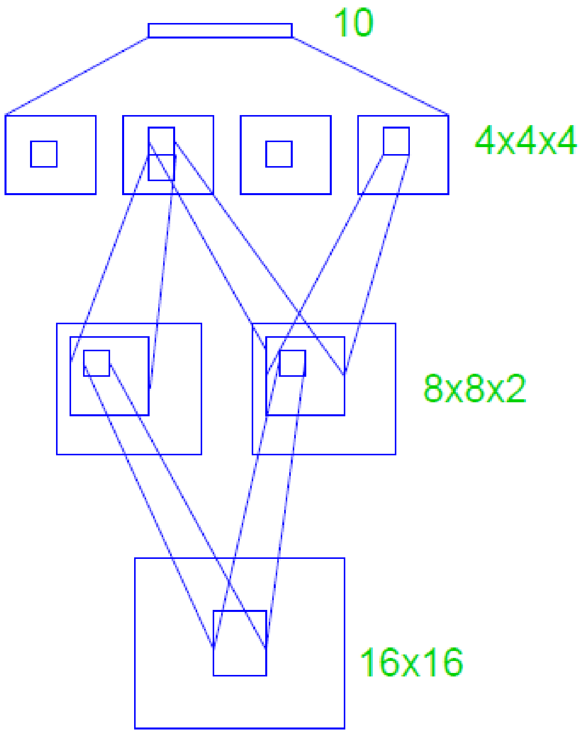
- Net-4: two hidden layers, locally connected, weight sharing
  - First hidden layer: two 8x8 feature maps
    - Input from 3x3 patch
    - Units in the same 8x8 feature map share the same set of 9 weights (bias not shared)
    - **Convolutional networks**
  - Second hidden layer: no weight sharing
- The gradient of the error for a shared weight is the sum of the gradients to each connection controlled by the weight



Net-4

# Example: ZIP Code Data (iv)

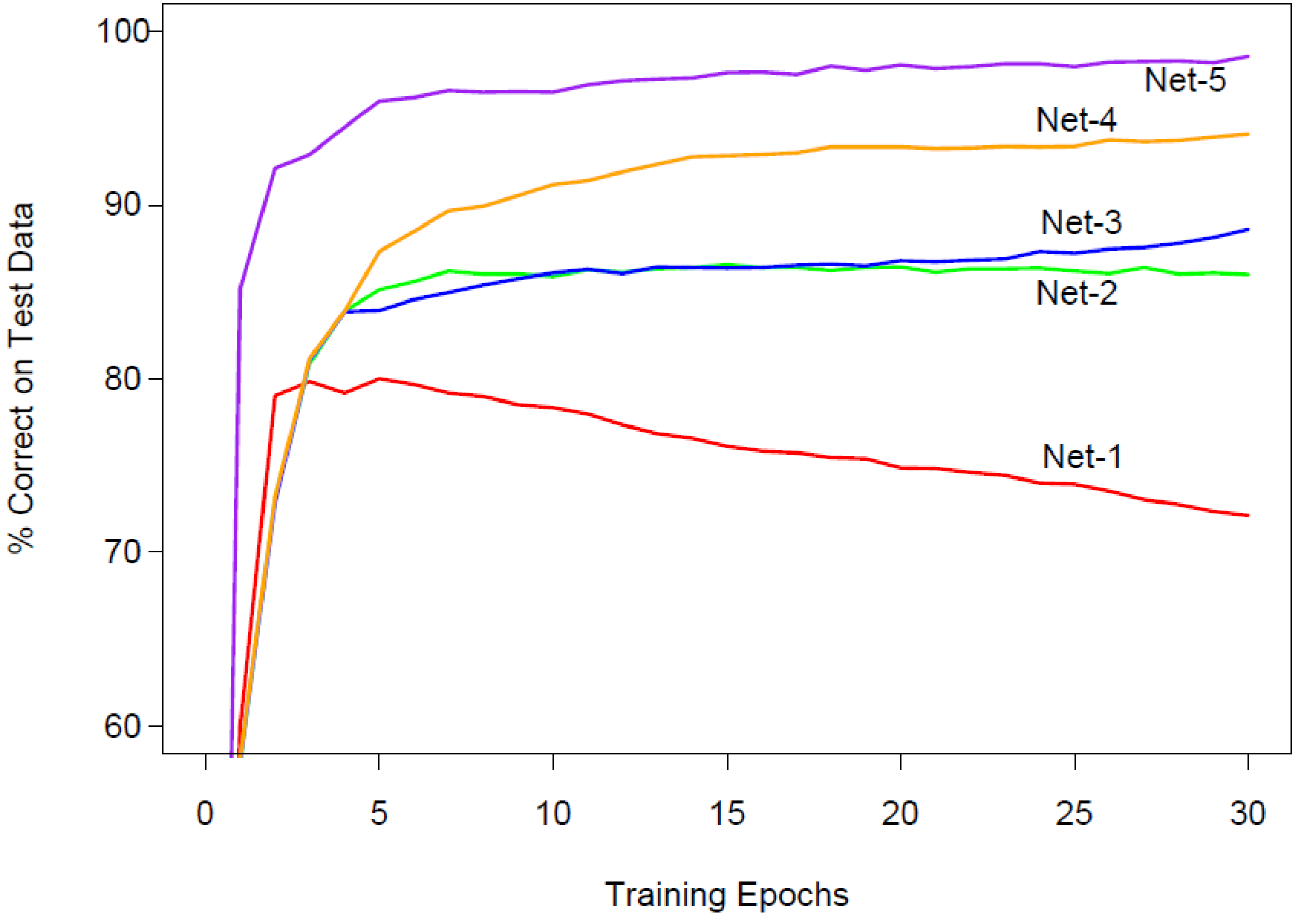
- Net-5: two hidden layers, locally connected, two levels of weight sharing
  - First hidden layer: same as Net-4
  - Second hidden layer: four 4x4 feature maps
    - Input from a 5x5 patch
    - Weights shared in each of the feature maps



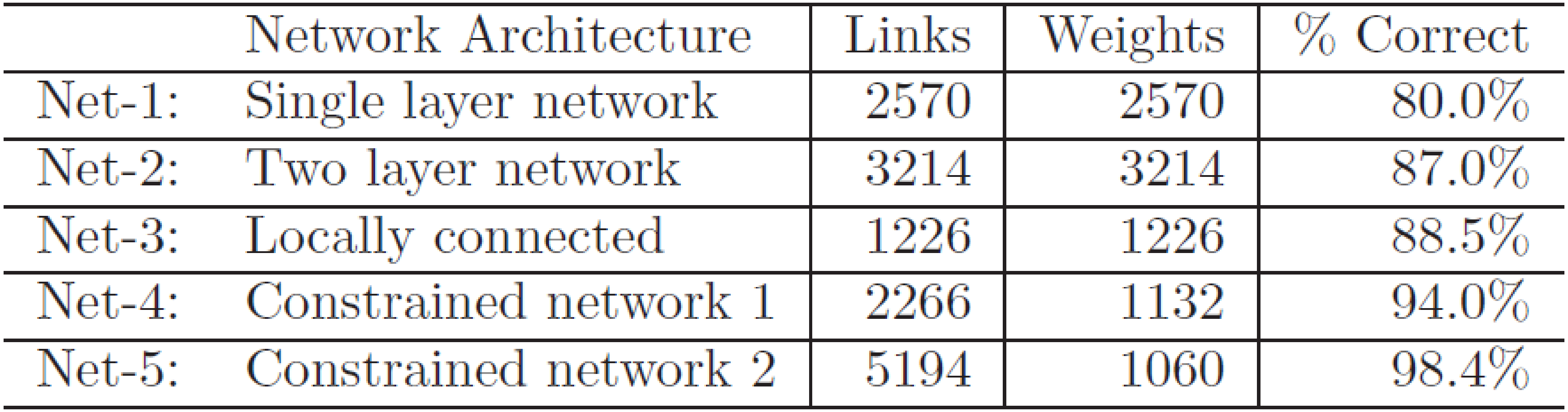
Net-5

# Example: ZIP Code Data (v)

- Training error was 0% in all the cases
  - More parameters than training observations



# Example: ZIP Code Data (vi)



| | Network Architecture | Links | Weights | % Correct |
|---|---|---|---|---|
| Net-1: | Single layer network | 2570 | 2570 | 80.0% |
| Net-2: | Two layer network | 3214 | 3214 | 87.0% |
| Net-3: | Locally connected | 1226 | 1226 | 88.5% |
| Net-4: | Constrained network 1 | 2266 | 1132 | 94.0% |
| Net-5: | Constrained network 2 | 5194 | 1060 | 98.4% |

- Best results on a large database: Le Cun et al., 1998
  - 60,000 training and 10,000 test examples
  - LeNet-5: a more complex convolutional network
    - 99.2% correct
  - Boosted LeNet-4: boosting with a predecesor of LeNet-5
    - 99.3% correct

# Exercise

- We have a neural network with a hidden layer, $s^l=\{2, 3, 1\}$, $W^{(1)}= (-2, 1; 1, -1; 3, -1)$, $W^{(2)}= (2, 3, 1)$, $b^{(1)}= (0; -1; 1)$, and $b^{(2)}= (0)$. The network uses the sigmoid activation function in all neurons except the output neuron, whose activation function is the identity. In addition, the cost function is the quadratic error. Given the example (1, 1, 1):
  - Calculate $z_i^{(l)}$ and $a_i^{(l)}$ for all neurons
  - Using the error back-propagation algorithm, compute $\delta_i^{(l)}$ for all neurons
  - Using the error back-propagation algorithm, determine the final values of each weight ($W_{ij}^{(l)}$) and bias ($b_i^{(l)}$) of the neural network after completion of the first iteration of the algorithm, assuming a value of $\lambda=1$, and a learning rate $\alpha=0.5$

# Bibliography

- C. Bishop, Neural Networks for Pattern Recognition. Oxford University Press, 1995.
  - Chapter 4
- T. Hastie, R. Tibshirani, y J. Friedman, The elements of statistical learning. Springer, 2009.
  - Chapter 11
- Unsupervised Feature Learning and Deep Learning Tutorial
  - http://ufldl.stanford.edu/tutorial/