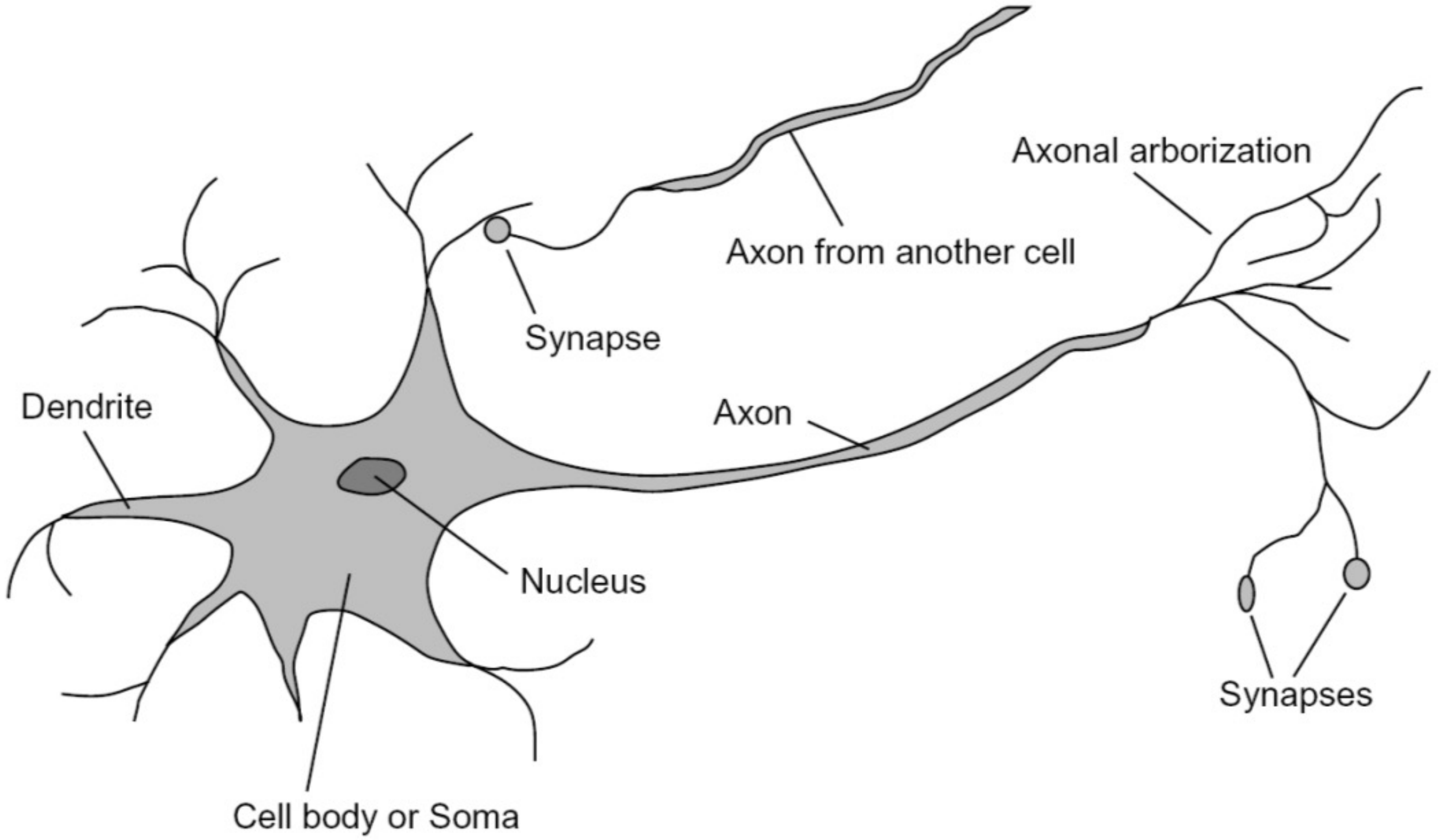
Training neural networks

Announcements

- A3 posted, sign up and create your teams

Inspiration: Neuron cells



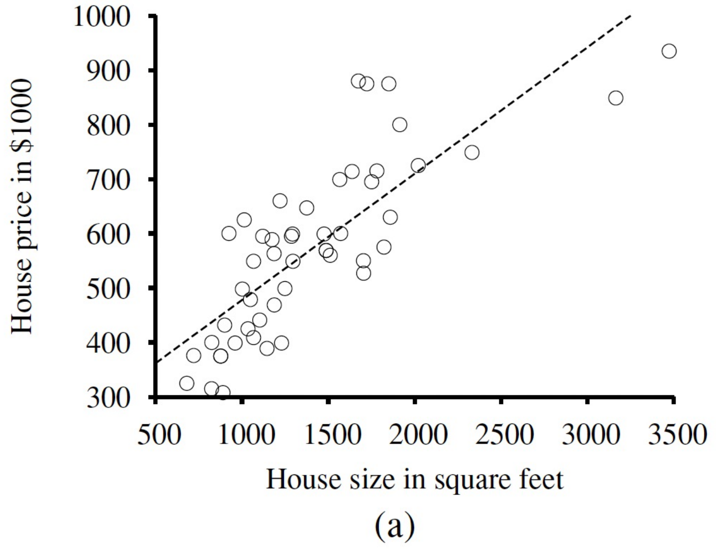
Slide credit: Rob Fergus

Networks with hidden layers

- Can represent XORs, other nonlinear functions
- Many, many variants:
  - Different network structures
  - Different activation functions
  - Etc…
- As the number of hidden units increases, the network’s capacity to learn more complicated functions also increases
- *How to train hidden layers?*

Adapted from K. Hauser’s slide

Sometimes we can estimate parameters analytically: linear regression



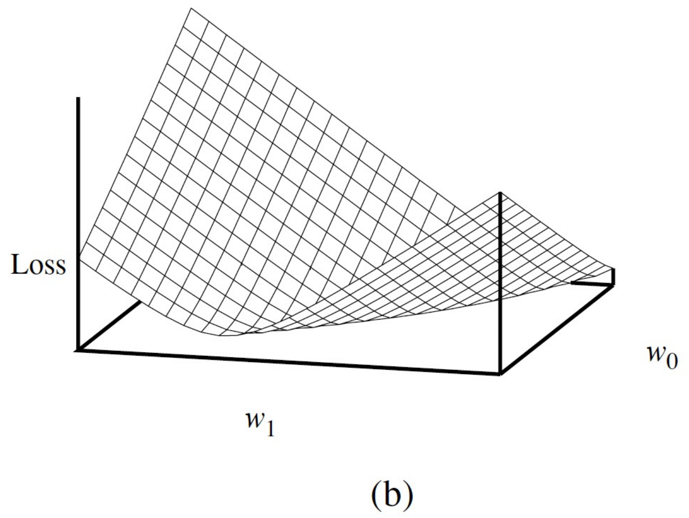


Figure 18.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

$$h_{\mathbf{w}}(x) = w_1 x + w_0$$

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2(y_j, h_{\mathbf{w}}(x_j)) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$$

Sometimes we can estimate parameters analytically: linear regression

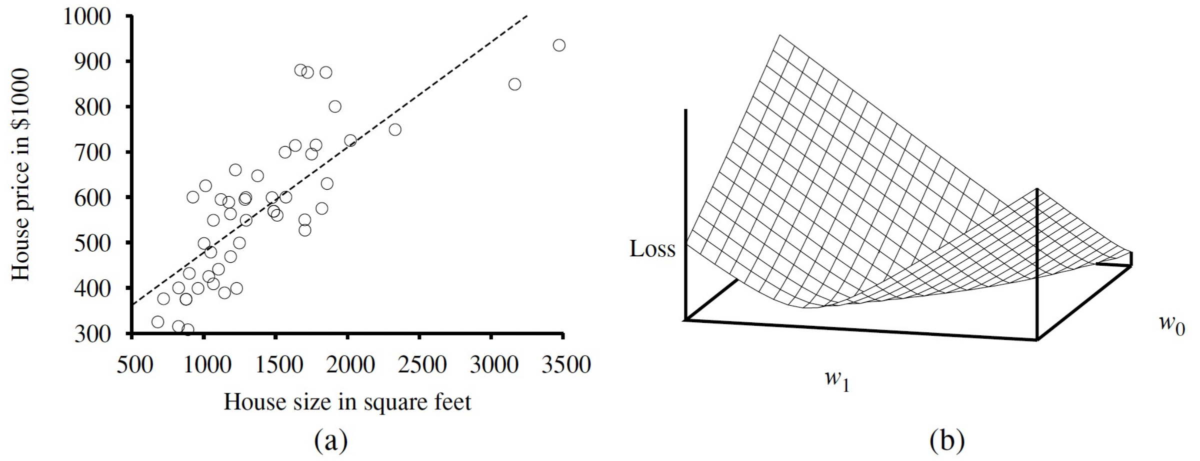


Figure 18.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

We would like to find $\mathbf{w}^* = \text{argmin}_{\mathbf{w}}\, Loss(h_{\mathbf{w}})$. The sum $\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0\,. \qquad (18.2)$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N\,. \qquad (18.3)$$

- For models more complicated than linear regression, typically there is not closed—form solution (we can not estimate parameters analytically).
- Such problems can be addressed by a hill-climbing algorithm that follows the gradient of the function to be optimized.
- To minimize the loss, we will use gradient descent:

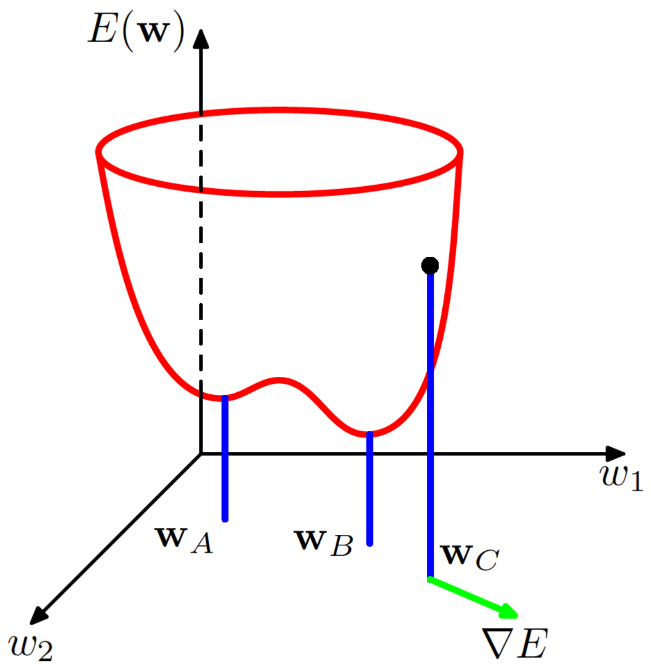
```
$\mathbf{w} \leftarrow$ any point in the parameter space
**loop** until convergence **do**
    **for each** $w_i$ **in w do**
        $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$
```

α is the learning rate

Network training

If we make a small step in weight space from **w** to **w**+δ**w** then the change in the error function (Loss) is $\delta E = \delta \mathbf{w}^T \nabla E(\mathbf{w})$.

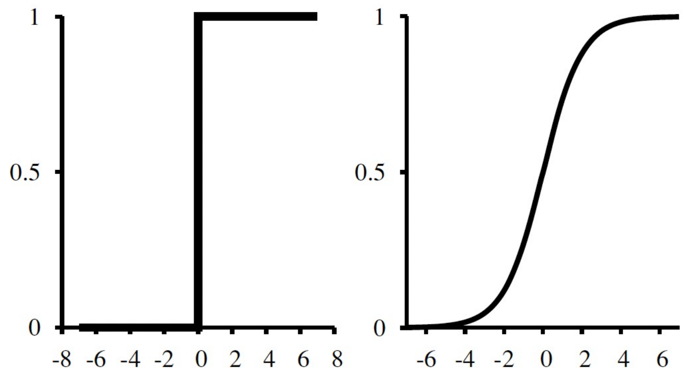
- where the vector $\nabla E(\mathbf{w})$ points in the direction of greatest rate of increase of the error function.
- Our goal is to find a vector **w** such that E(**w**) takes its smallest value.
- However, the error function typically has a highly nonlinear dependence on the weights and bias parameters, and so there will be many points in weight space at which the gradient vanishes (or is numerically very small).
- A minimum that corresponds to the smallest value of the error function for any weight vector is said to be a **global minimum** ($\mathbf{w}_B$ in the figure).
- Any other minima corresponding to higher values of the error function are said to be **local minima** ($\mathbf{w}_A$ in the figure).



Linear classification with logistic regression

Unlike step function, logistic function (also known as sigmoid) is differentiable.

$$g(z) = \frac{1}{1+e^{-z}}$$



Derivative of the logistic function:

$$g'(z) = g(z)(1 - g(z))$$

Linear classification with logistic regression

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

Remember chain rule: $\partial g(f(x))/\partial x = g'(f(x))\,\partial f(x)/\partial x$

Linear classification with logistic regression

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

Remember chain rule: $\partial g(f(x))/\partial x = g'(f(x))\,\partial f(x)/\partial x$

Let's derive weight update for minimizing the loss in logistic regression

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w} \cdot \mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i \, .
\end{aligned}$$

Linear classification with logistic regression

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

Remember chain rule: $\partial g(f(x))/\partial x = g'(f(x))\,\partial f(x)/\partial x$

Let's derive weight update for minimizing the loss in logistic regression

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i} \mathbf{w} \cdot \mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i \;.
\end{aligned}$$

$$g'(z) = g(z)(1 - g(z))$$

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

Linear classification with logistic regression

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w}\cdot\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}$$

Remember chain rule: $\partial g(f(x))/\partial x = g'(f(x))\,\partial f(x)/\partial x$

Let's derive weight update for minimizing the loss in logistic regression

$$\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w}\cdot\mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w}\cdot\mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w}\cdot\mathbf{x}) \times x_i \ .
\end{aligned}$$

$$g'(z) = g(z)(1-g(z))$$

$$g'(\mathbf{w}\cdot\mathbf{x}) = g(\mathbf{w}\cdot\mathbf{x})(1-g(\mathbf{w}\cdot\mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1-h_{\mathbf{w}}(\mathbf{x}))$$

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1-h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

Learning in multilayer networks: error backpropagation

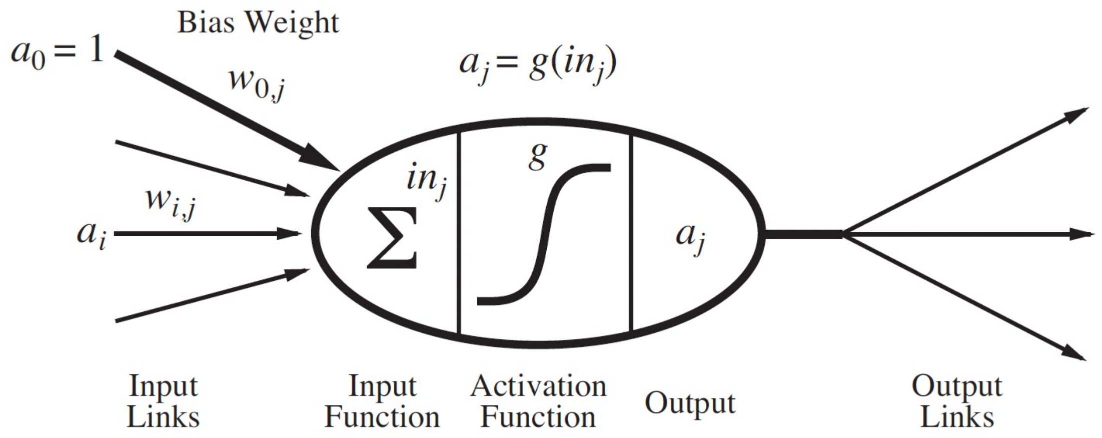


Figure 18.19 A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

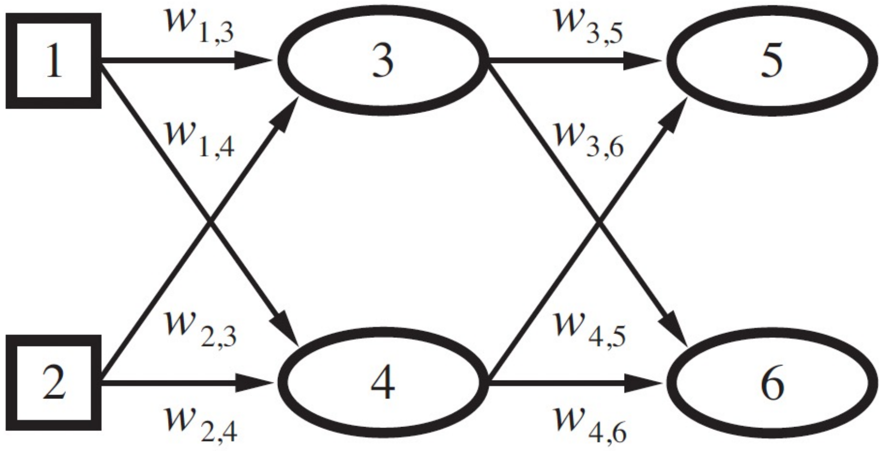
$$in_j = \sum_{i=0}^{n} w_{i,j} a_i$$

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

Learning in multilayer networks: error backpropagation

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \frac{\partial}{\partial w}|\mathbf{y} - \mathbf{h_w}(\mathbf{x})|^2 = \frac{\partial}{\partial w}\sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w}(y_k - a_k)^2$$

where the index k ranges over nodes in the output layer.



Learning in multilayer networks: error backpropagation

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \frac{\partial}{\partial w}|\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})|^2 = \frac{\partial}{\partial w}\sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w}(y_k - a_k)^2$$

First, let's compute the gradient for loss at k-th output:

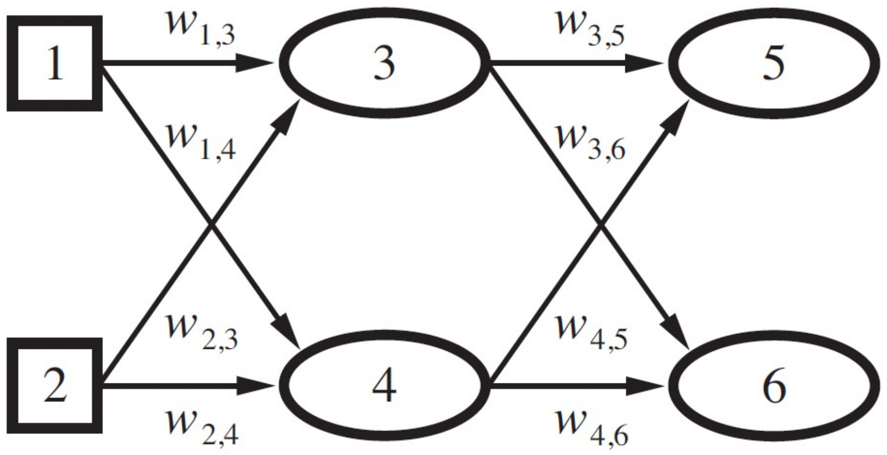
Remember chain rule: $\partial g(f(x))/\partial x = g'(f(x))\,\partial f(x)/\partial x$

$$\begin{aligned}
\frac{\partial Loss_k}{\partial w_{j,k}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{j,k}} \\
&= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k)g'(in_k)\frac{\partial}{\partial w_{j,k}}\left(\sum_j w_{j,k}a_j\right) \\
&= -2(y_k - a_k)g'(in_k)a_j = -a_j\Delta_k \ ,
\end{aligned}$$

$$\Delta_k = Err_k \times g'(in_k)$$

Learning in multilayer networks: error backpropagation

Next we compute gradient with respect to the $w_{i,j}$ weights connecting the input layer to the hidden layer



Learning in multilayer networks: error backpropagation

Next, we compute gradient with respect to the $w_{i,j}$ weights connecting the input layer to the hidden layer

$$
\begin{aligned}
\frac{\partial Loss_k}{\partial w_{i,j}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}} \\
&= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k \frac{\partial}{\partial w_{i,j}}\left(\sum_j w_{j,k}a_j\right) \\
&= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k} g'(in_j)\frac{\partial in_j}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k} g'(in_j)\frac{\partial}{\partial w_{i,j}}\left(\sum_i w_{i,j}a_i\right) \\
&= -2\Delta_k w_{j,k} g'(in_j)a_i = -a_i\Delta_j \ ,
\end{aligned}
$$

$$\Delta_j = g'(in_j)\sum_k w_{j,k}\Delta_k$$

Backpropagation Algorithm

- Werbos, Rumelhart, Hinton, Williams (1974)
- Until convergence:
  - Present a training pattern to network
  - Calculate the error of the output nodes
  - Calculate the error of the hidden nodes, based on the output node error which is propagated back
  - Continue back-propagating error until the input layer
  - Update all weights in the network

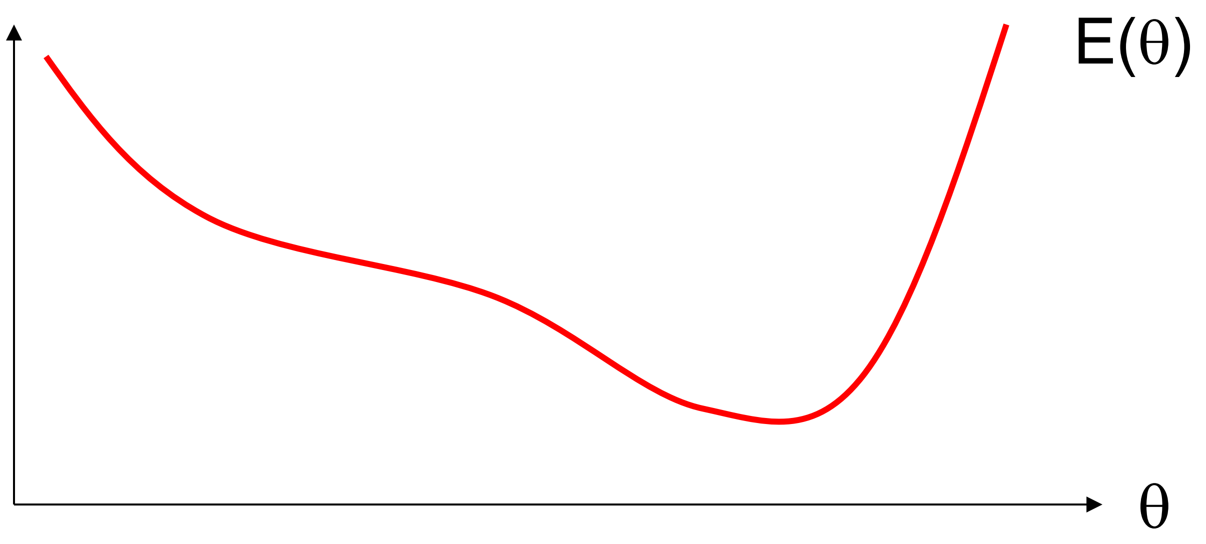
```
function BACK-PROP-LEARNING($examples$, $network$) returns a neural network
  inputs: $examples$, a set of examples, each with input vector x and output vector y
          $network$, a multilayer network with $L$ layers, weights $w_{i,j}$, activation function $g$
  local variables: $\Delta$, a vector of errors, indexed by network node

  repeat
      for each weight $w_{i,j}$ in $network$ do
          $w_{i,j} \leftarrow$ a small random number
      for each example ($\mathbf{x}, \mathbf{y}$) in $examples$ do
          /* Propagate the inputs forward to compute the outputs */
          for each node $i$ in the input layer do
              $a_i \leftarrow x_i$
          for $\ell = 2$ to $L$ do
              for each node $j$ in layer $\ell$ do
                  $in_j \leftarrow \sum_i w_{i,j} \, a_i$
                  $a_j \leftarrow g(in_j)$
          /* Propagate deltas backward from output layer to input layer */
          for each node $j$ in the output layer do
              $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
          for $\ell = L - 1$ to $1$ do
              for each node $i$ in layer $\ell$ do
                  $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \, \Delta[j]$
          /* Update every weight in network using deltas */
          for each weight $w_{i,j}$ in $network$ do
              $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
  until some stopping criterion is satisfied
  return $network$
```

Figure 18.24 The back-propagation algorithm for learning in multilayer networks.

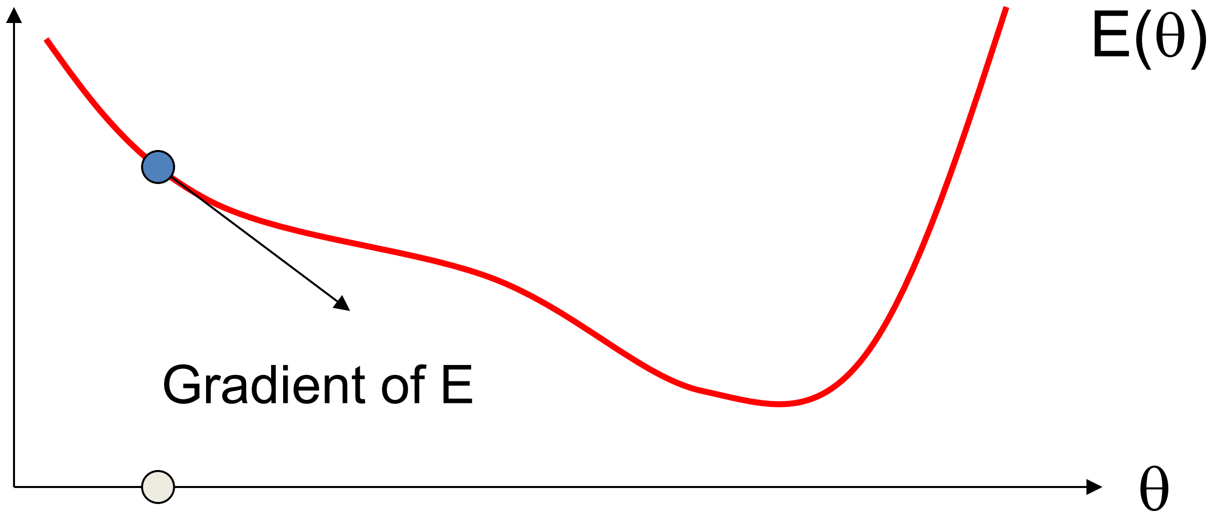
Understanding Backpropagation

- Minimize E(θ)
- Gradient Descent…



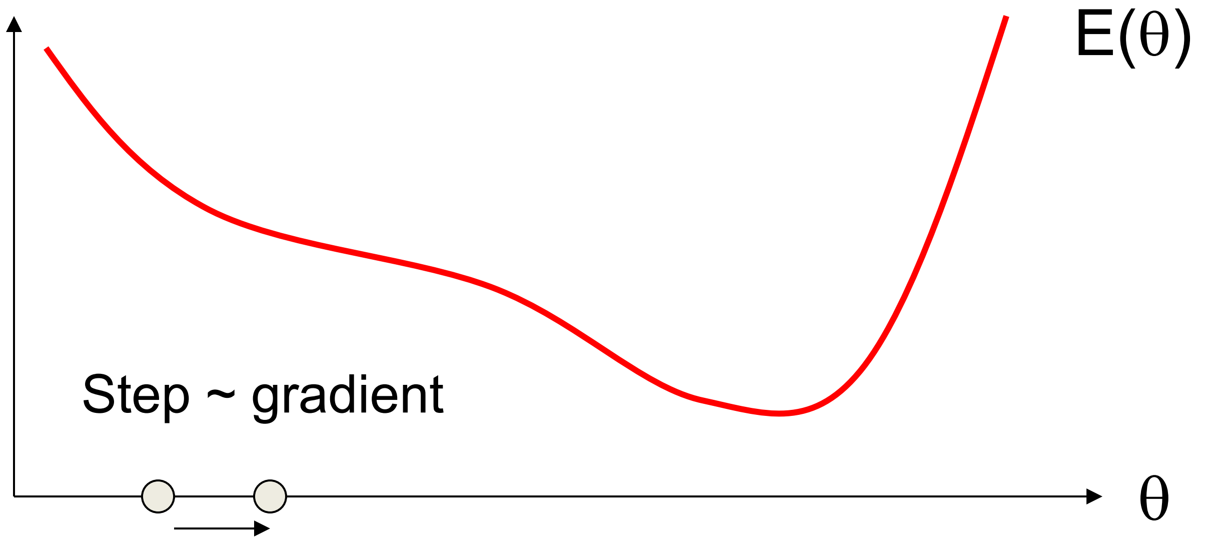
Understanding Backpropagation

- Minimize E(θ)
- Gradient Descent…



Understanding Backpropagation

- Minimize E(θ)
- Gradient Descent…



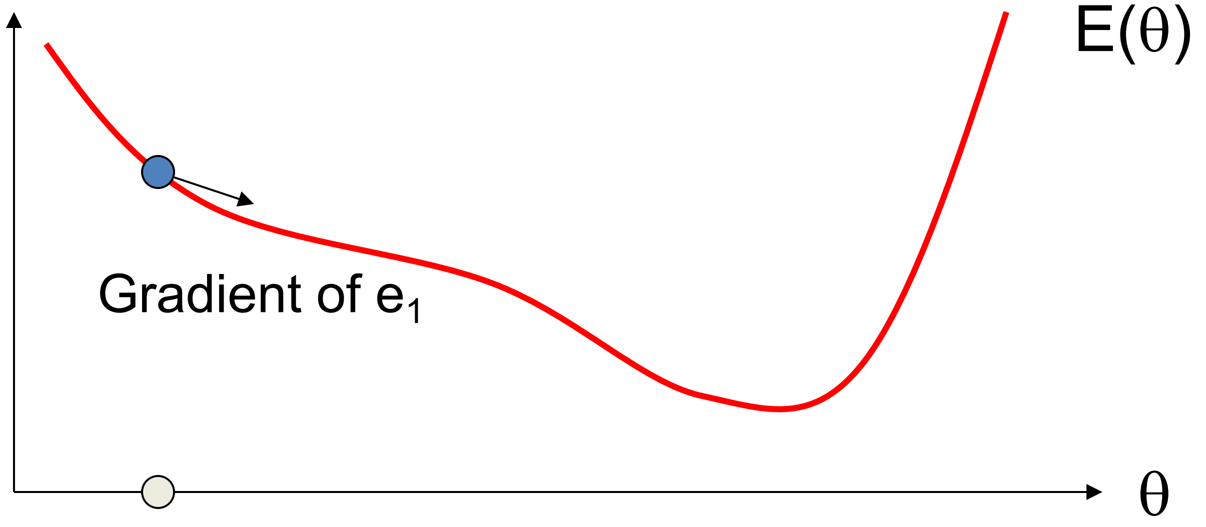
Stochastic Gradient Descent

- Classic backprop computes weight changes after scanning through the entire training set
  - Theoretically justified
  - But this is very slow

- Stochastic gradient descent randomizes the input data, then takes a step after each training exemplar

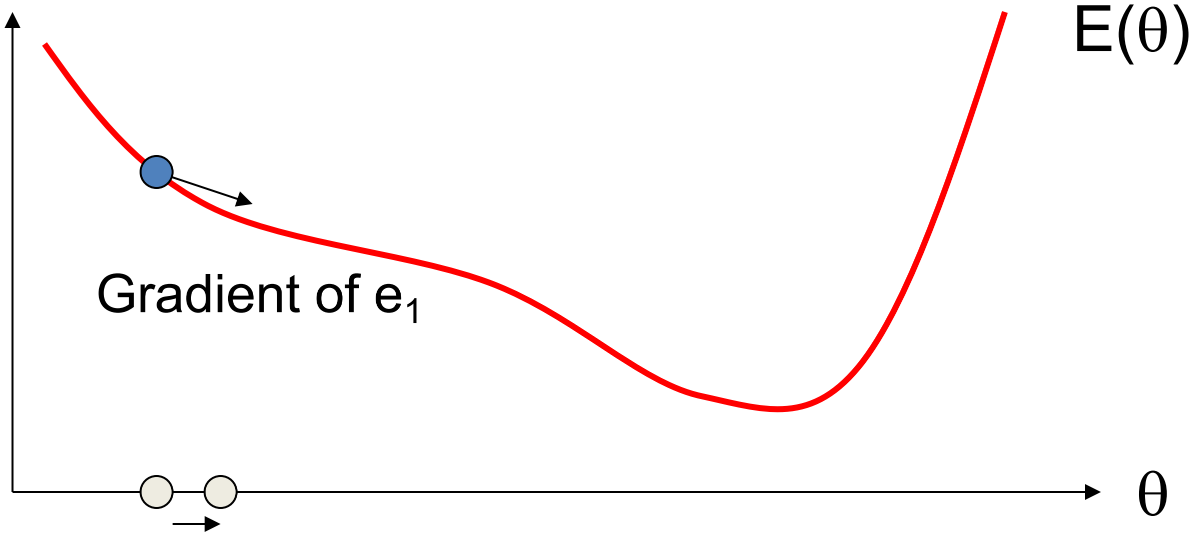
Understanding Backpropagation

- Example of Stochastic Gradient Descent
- Decompose $E(\theta) = e_1(q)+e_2(q)+\ldots+e_N(q)$
  - Here $e_k = (g(\mathbf{x}^{(k)},\theta)-y^{(k)})^2$
- On each iteration take a step to reduce e_k



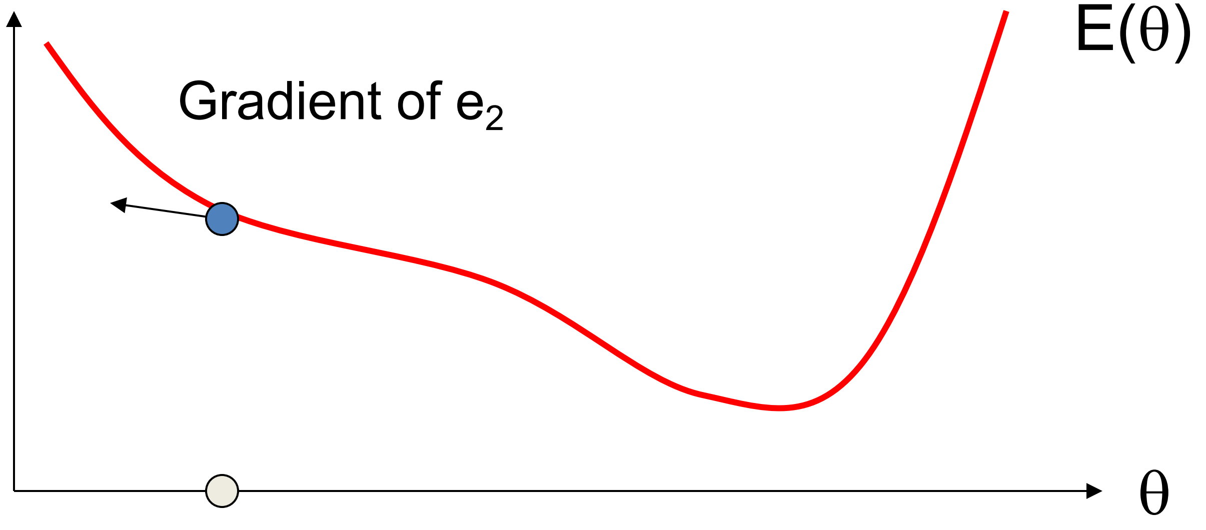
Understanding Backpropagation

- Example of Stochastic Gradient Descent
- Decompose $E(\theta) = e_1(q)+e_2(q)+\ldots+e_N(q)$
  - Here $e_k = (g(\mathbf{x}^{(k)},\theta)-y^{(k)})^2$
- On each iteration take a step to reduce e_k



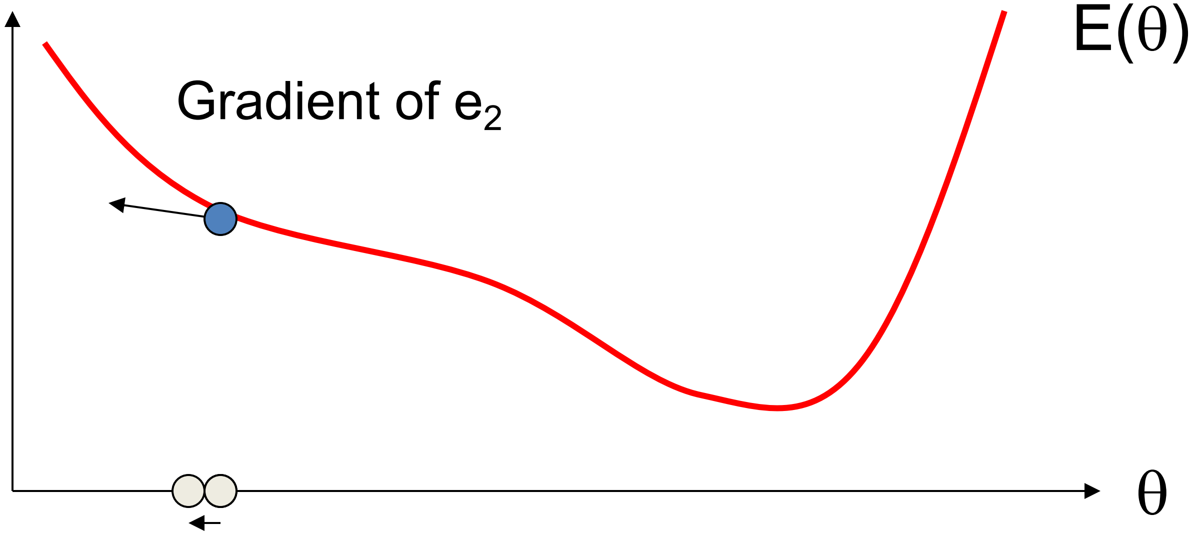
Understanding Backpropagation

- Example of Stochastic Gradient Descent
- Decompose $E(\theta) = e_1(q)+e_2(q)+\ldots+e_N(q)$
  - Here $e_k = (g(\mathbf{x}^{(k)},\theta)-y^{(k)})^2$
- On each iteration take a step to reduce e_k



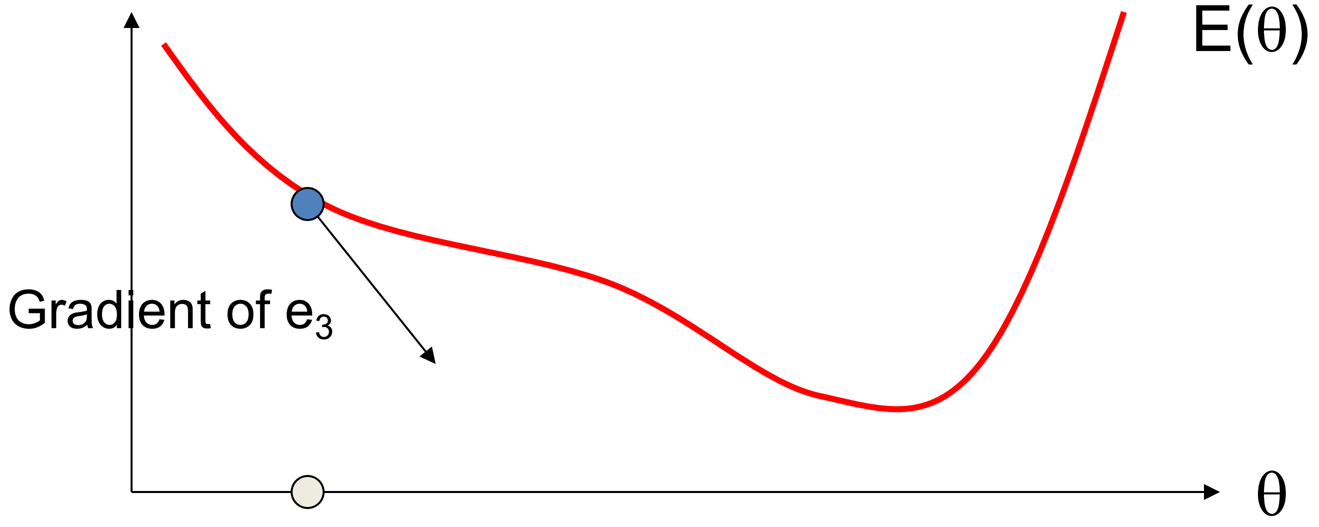
Understanding Backpropagation

- Example of Stochastic Gradient Descent
- Decompose $E(\theta) = e_1(q)+e_2(q)+\ldots+e_N(q)$
  - Here $e_k = (g(\mathbf{x}^{(k)},\theta)-y^{(k)})^2$
- On each iteration take a step to reduce e_k



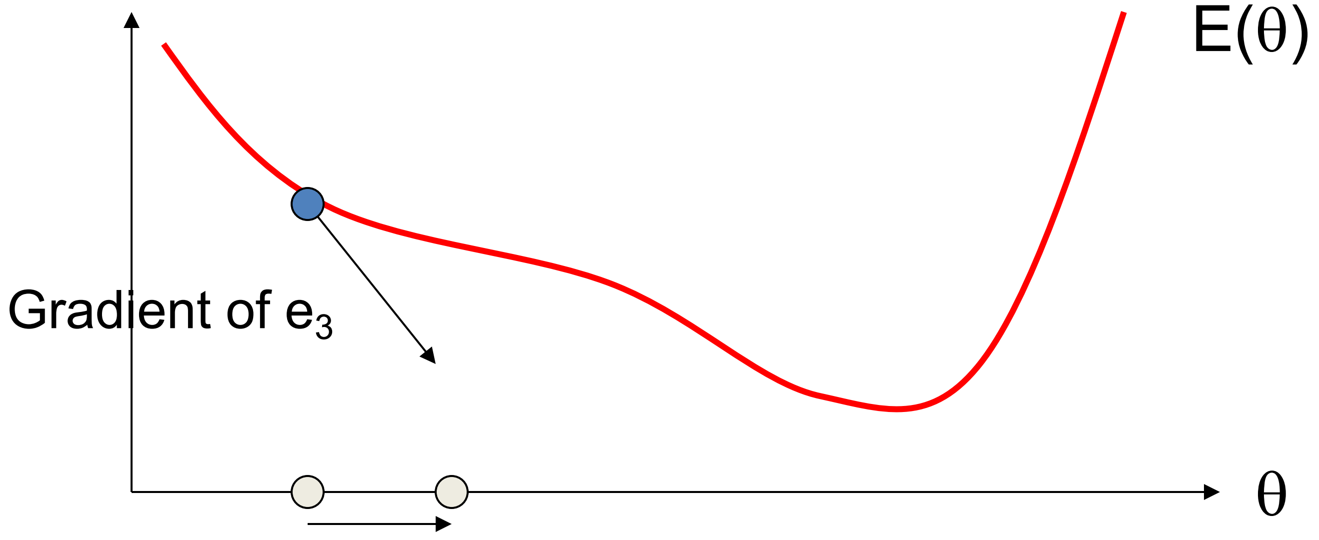
Understanding Backpropagation

- Example of Stochastic Gradient Descent
- Decompose $E(\theta) = e_1(q)+e_2(q)+\ldots+e_N(q)$
  - Here $e_k = (g(\mathbf{x}^{(k)},\theta)-y^{(k)})^2$
- On each iteration take a step to reduce e_k



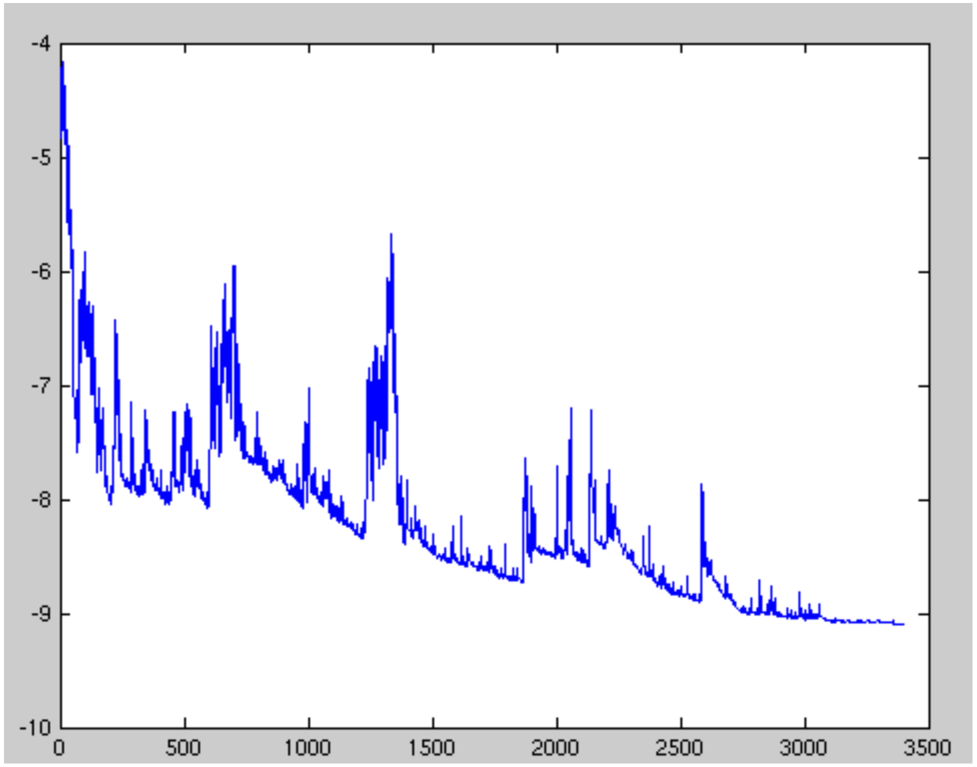
Understanding Backpropagation

- Example of Stochastic Gradient Descent
- Decompose $E(\theta) = e_1(q)+e_2(q)+\ldots+e_N(q)$
  - Here $e_k = (g(\mathbf{x}^{(k)},\theta)-y^{(k)})^2$
- On each iteration take a step to reduce e_k



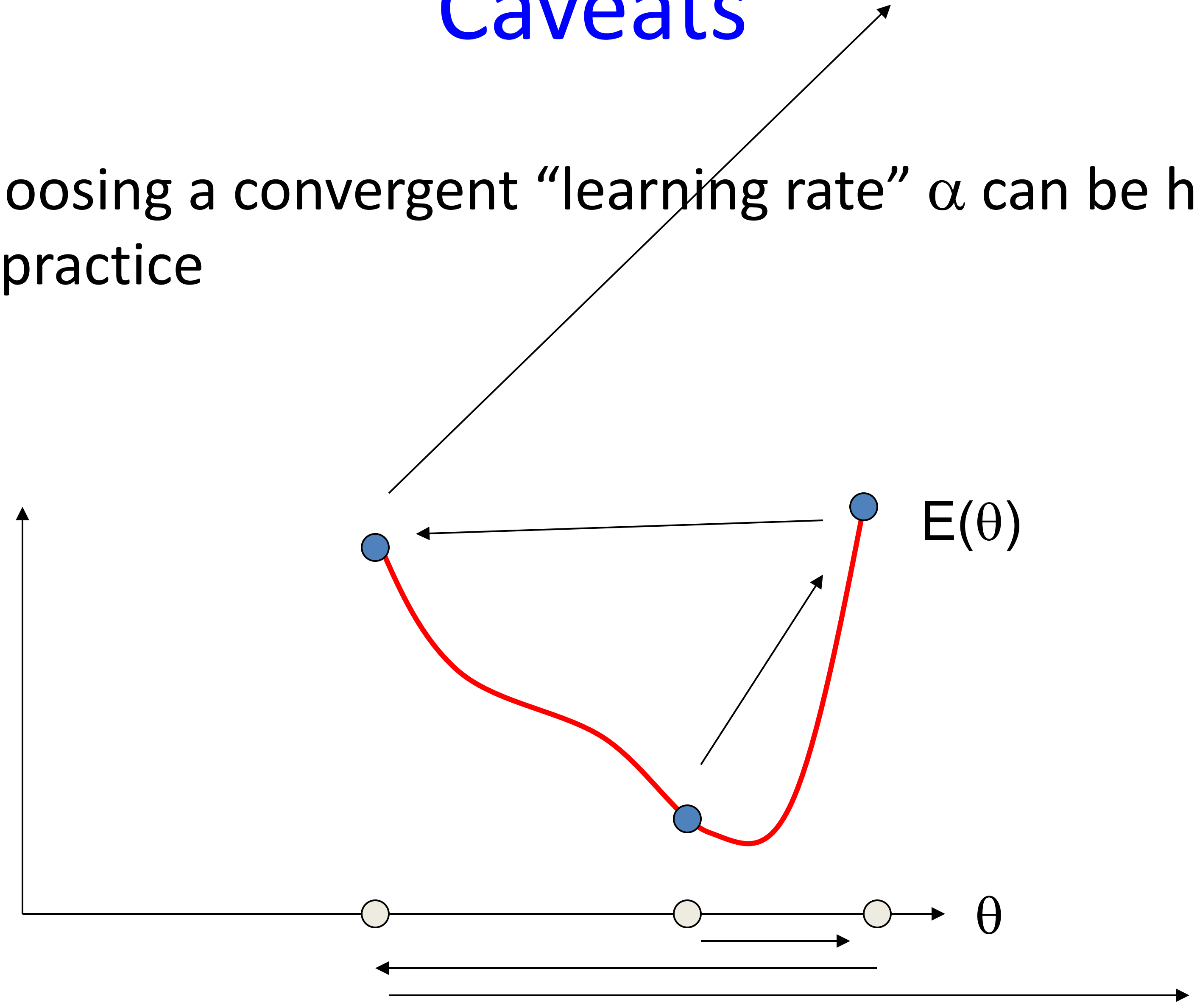
Stochastic Gradient Descent

- Objective function values (measured over all examples) over time settle into local minimum
- Step size must be reduced over time, e.g., O(1/t)



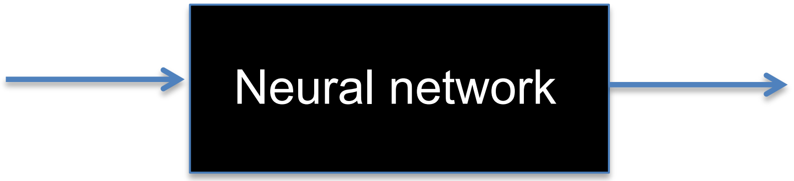
Caveats

- Choosing a convergent "learning rate" α can be hard in practice



Neural networks

- Neural networks are *universal function approximators*
  - Given any function, and a complicated enough network, they can accurately model that function



- How to choose the size and structure of networks?
  - If network is too large, risk of over-fitting (data caching)
  - If network is too small, representation may not be rich enough

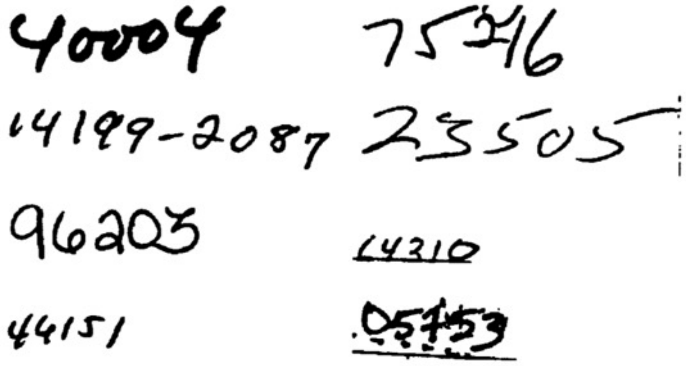
Pros and cons of different classifiers

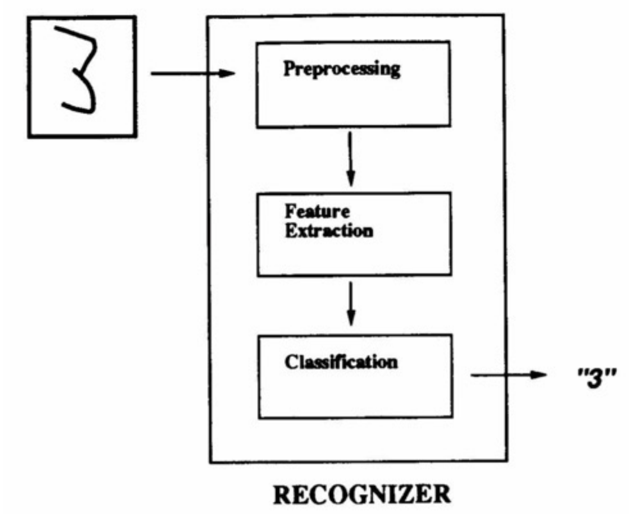
- Nearest neighbors
  - Can model any data, very prone to overfitting, requires distance function, fast learning, slow classification
- Neural networks
  - Models any function, requires structure, can suffer from local minima, slow learning, fast classification, difficult to interpret.
- Bayes nets
  - Requires setting network structure, fast learning, fast classification, intuitive interpretation of parameters.
- Decision trees
  - Limited modeling power, mostly automatic, moderate learning speed, fast classification, intuitive interpretation of parameters.
- Perceptrons
  - Very limited modeling power, fast training, fast classification, intuitive interpretation of parameters.

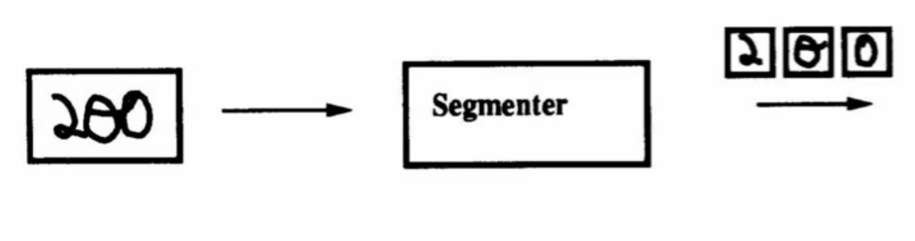
Neural Nets: 1960s-1990s

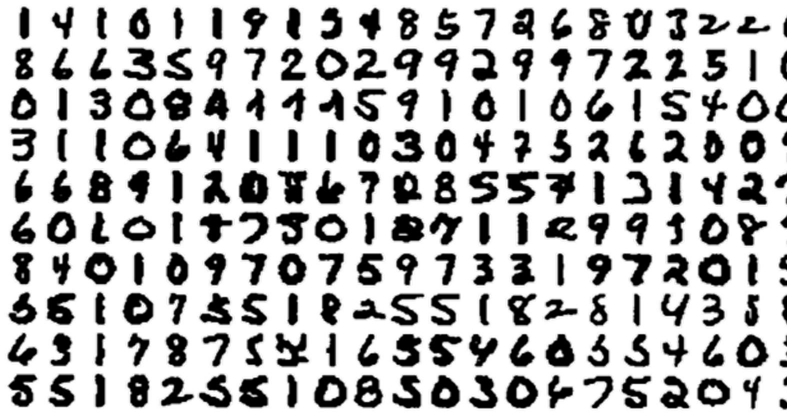
- Failure to deliver perceptron promises during during 1960s-1970s led to “AI winter”
- In 1980s, multi-layer networks and the backpropagation algorithm led to new excitement, new era of neural network research

Success story: handwritten digit recognition (LeCun, 1989)

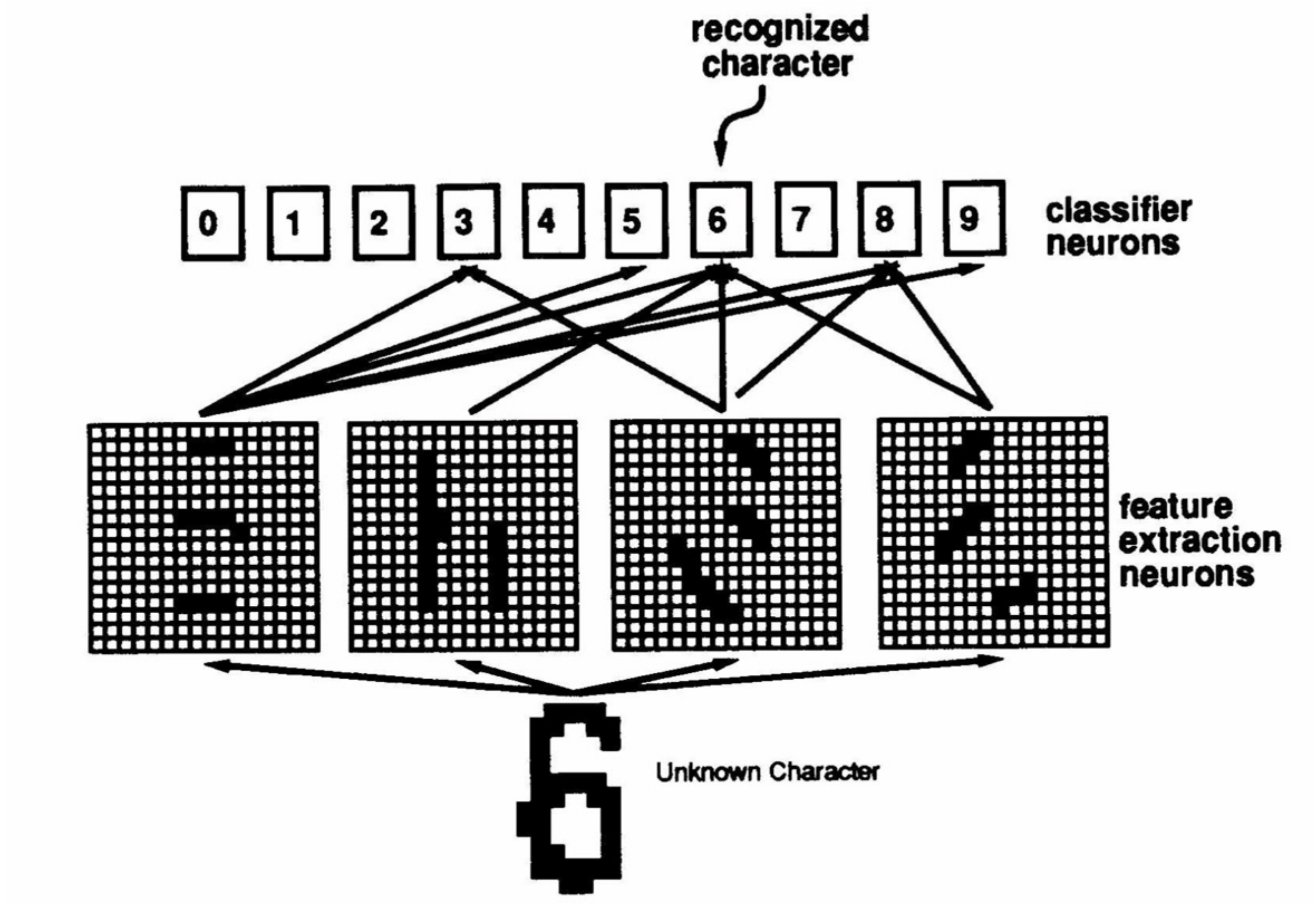




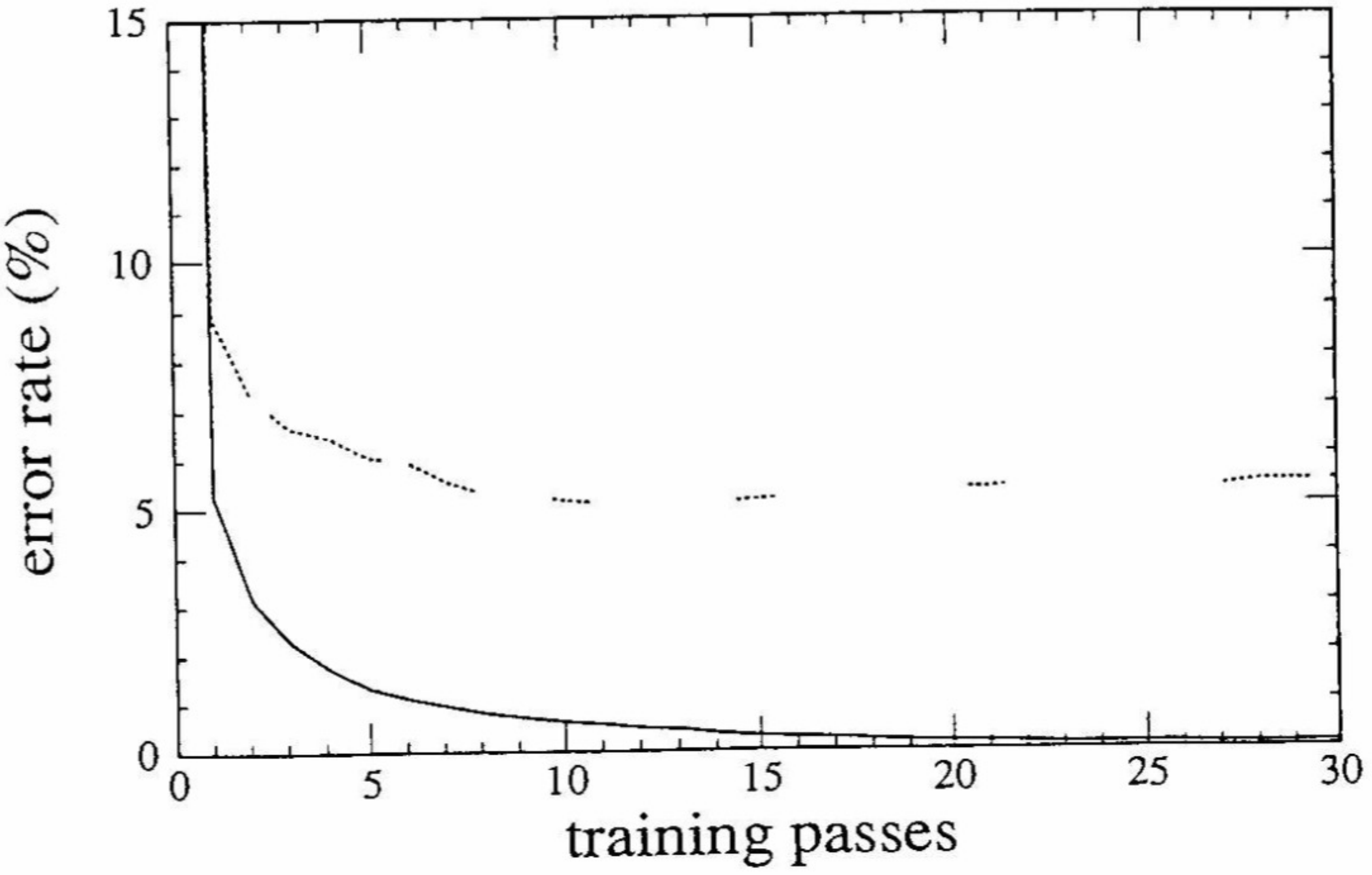




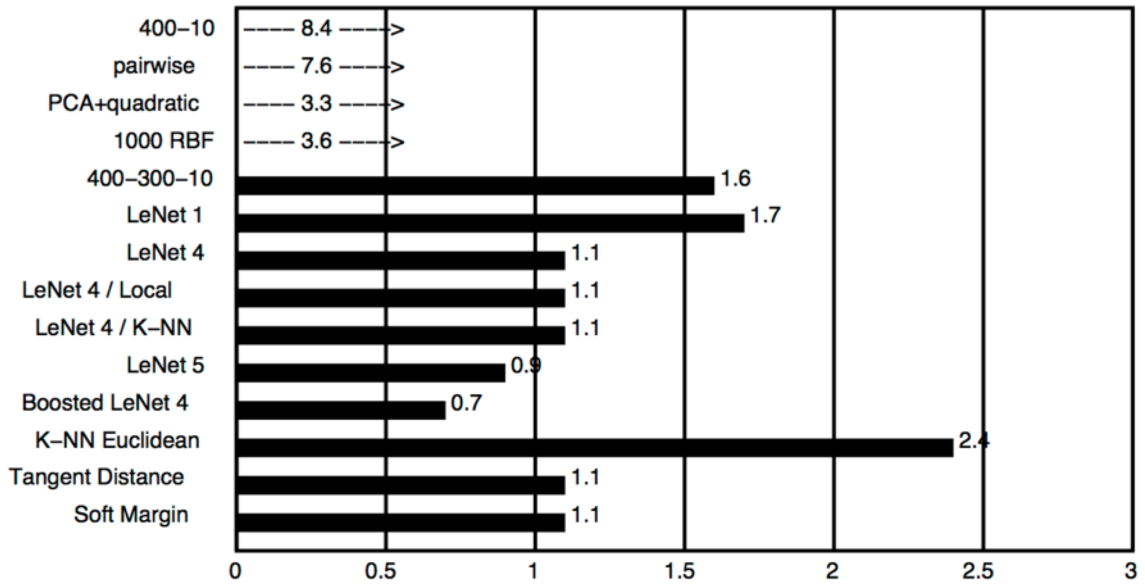
Network structure

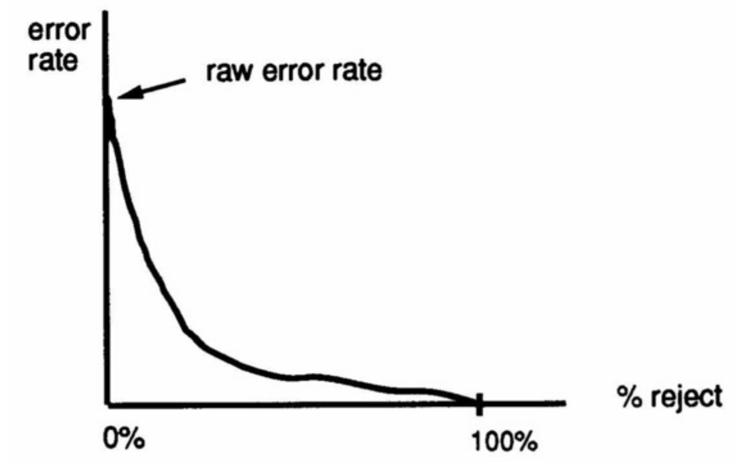


Use backprop to train

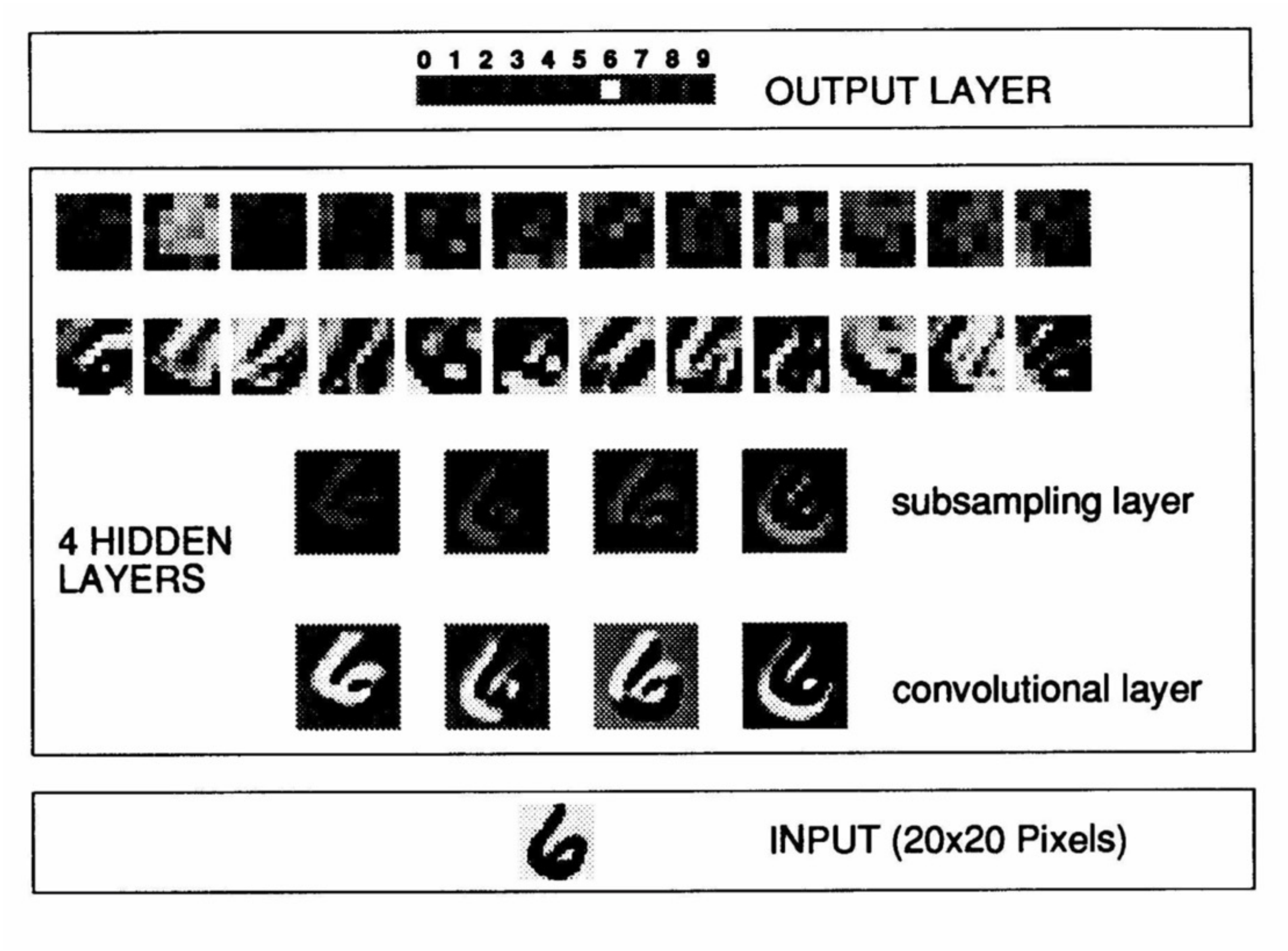


Worked better than other techniques (LeCun 1989)



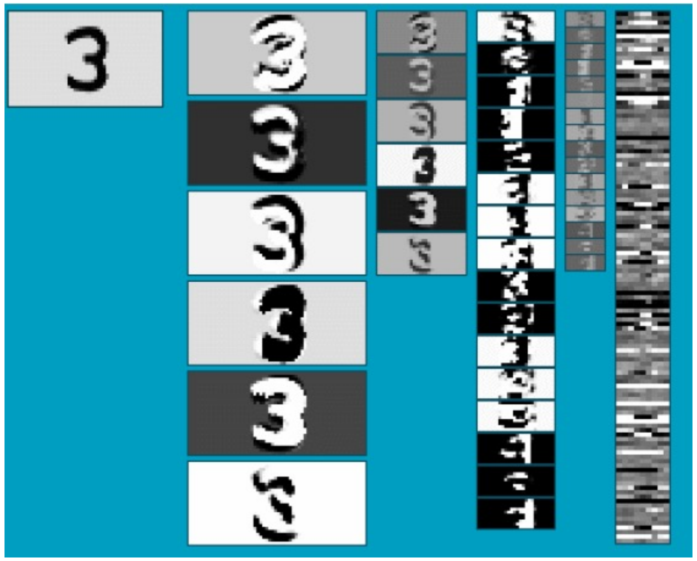


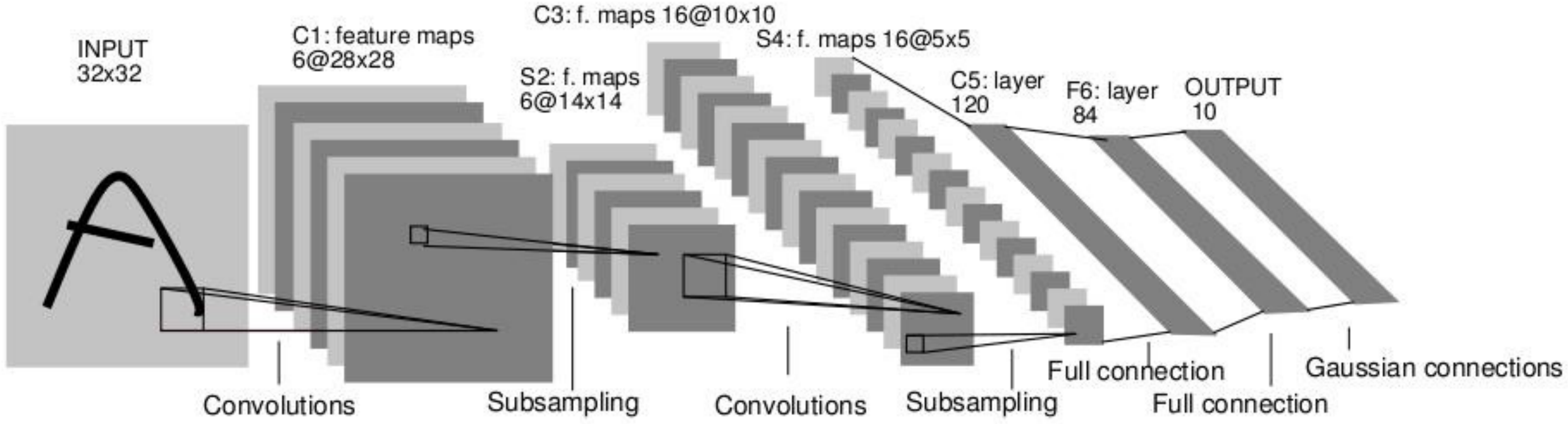
More complex architectures…



Convolutional Neural Networks

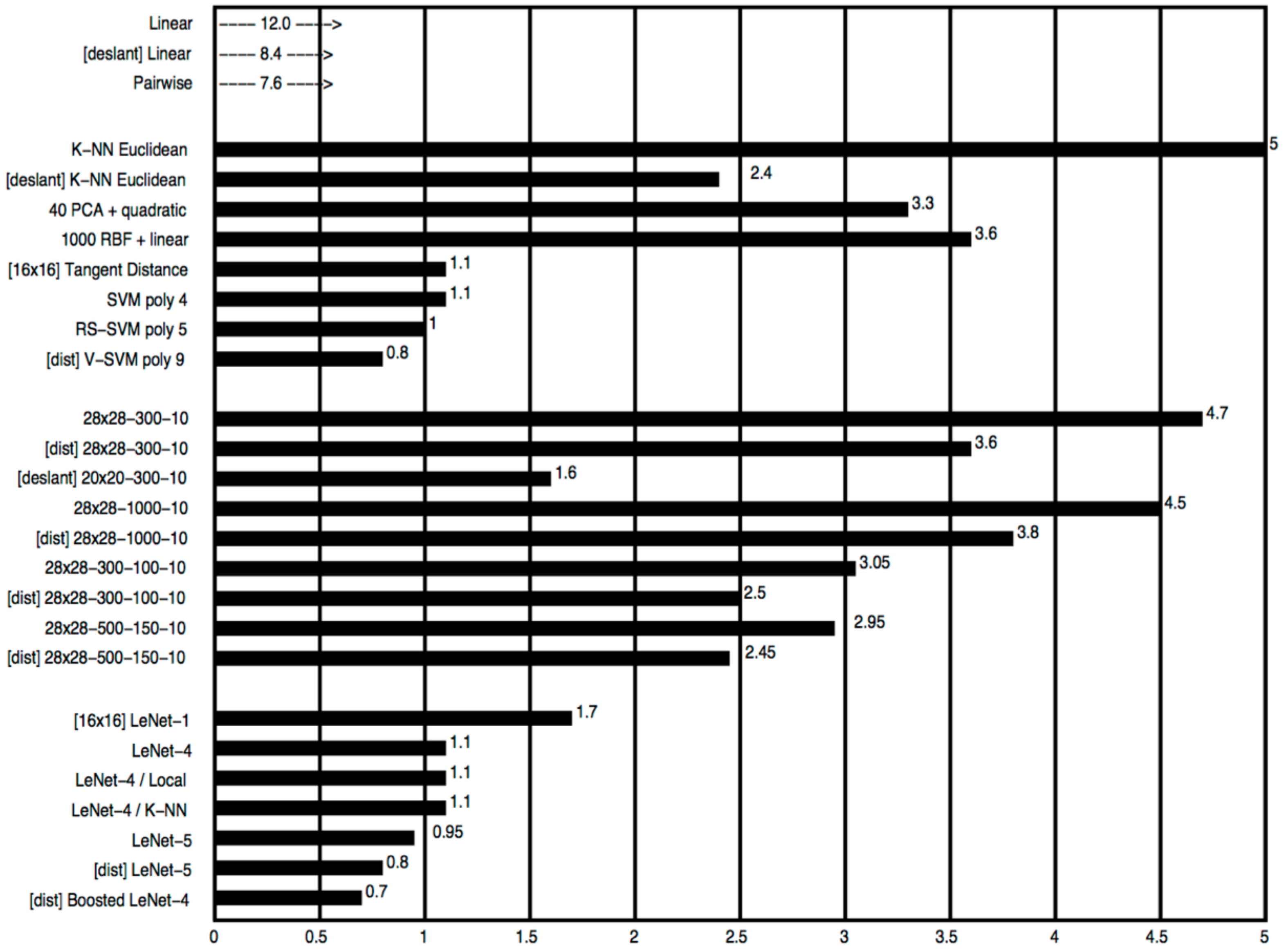
- Neural network with specialized connectivity structure
- Stack multiple stages of feature extractors
- Higher stages compute more global, more invariant features
- Classification layer at the end





Slide credit: Rob Fergus

But other techniques catching up (LeCun 1998)



Late 1990s-2010: Another decline

- Neural networks failed to work equally well on more complicated problems
  - E.g. recognition in real images, real audio streams, etc.

- Mix of practical and theoretical problems
  - How to decide network structure and many learning parameters (e.g. step sizes)?
  - Required too much computation
  - Required too much data
  - Very difficult to “debug” failures

2000's: Return to the simple

- Return to simpler techniques, like linear classifiers
  - But in high dimensions
  - Simpler learning algorithms, easier to justify theoretically
- Learn classifiers on manually-created features
  - E.g. not images themselves, but statistical features like color histograms, edge distributions, etc.

Next class

- Support Vector Machines (SVM)