More neural networks and support vector machines (SVMs)

Announcements

- A3 released (due on December 2nd), fill out the teams form by the end of the day (everyone should fill out the form)
- Five classes left! And some work:
  - Assignment 4
  - Final exam

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

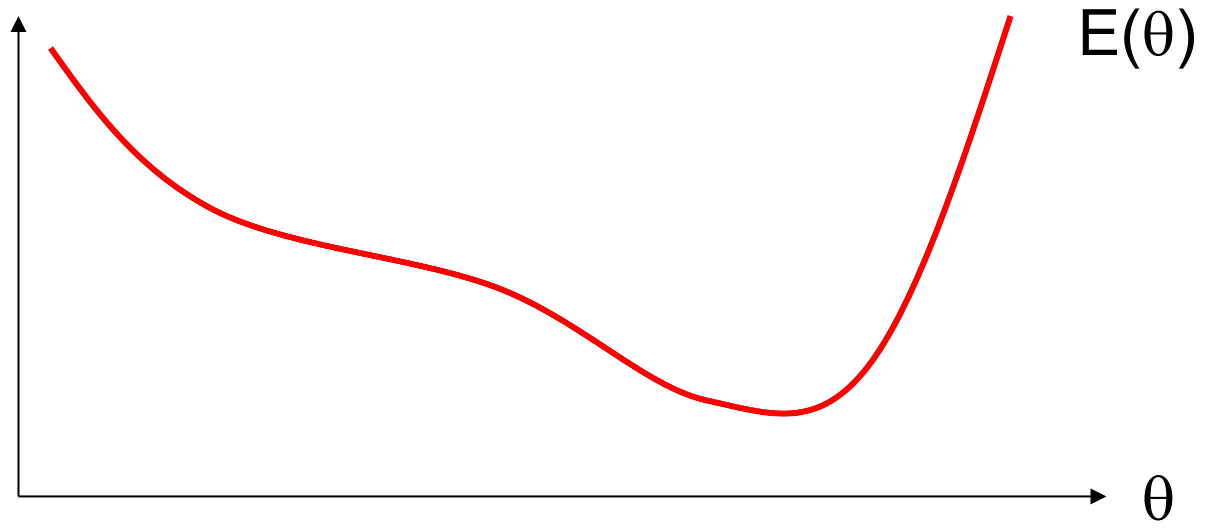
```
function BACK-PROP-LEARNING($examples$, $network$) returns a neural network
  inputs: $examples$, a set of examples, each with input vector $\mathbf{x}$ and output vector $\mathbf{y}$
          $network$, a multilayer network with $L$ layers, weights $w_{i,j}$, activation function $g$
  local variables: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

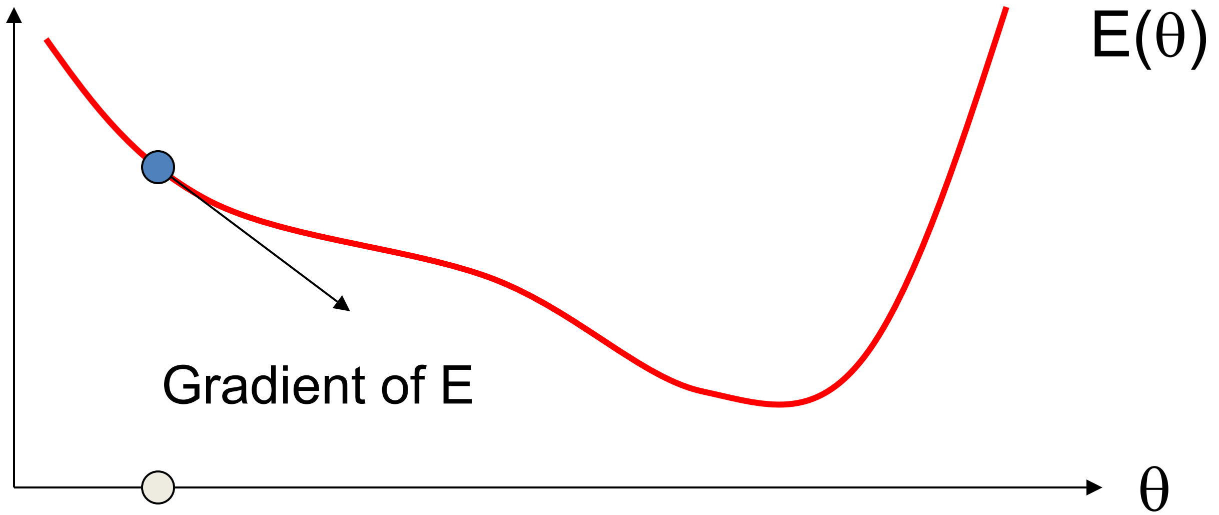
Understanding Backpropagation

- Minimize $E(\theta)$
- Gradient Descent…



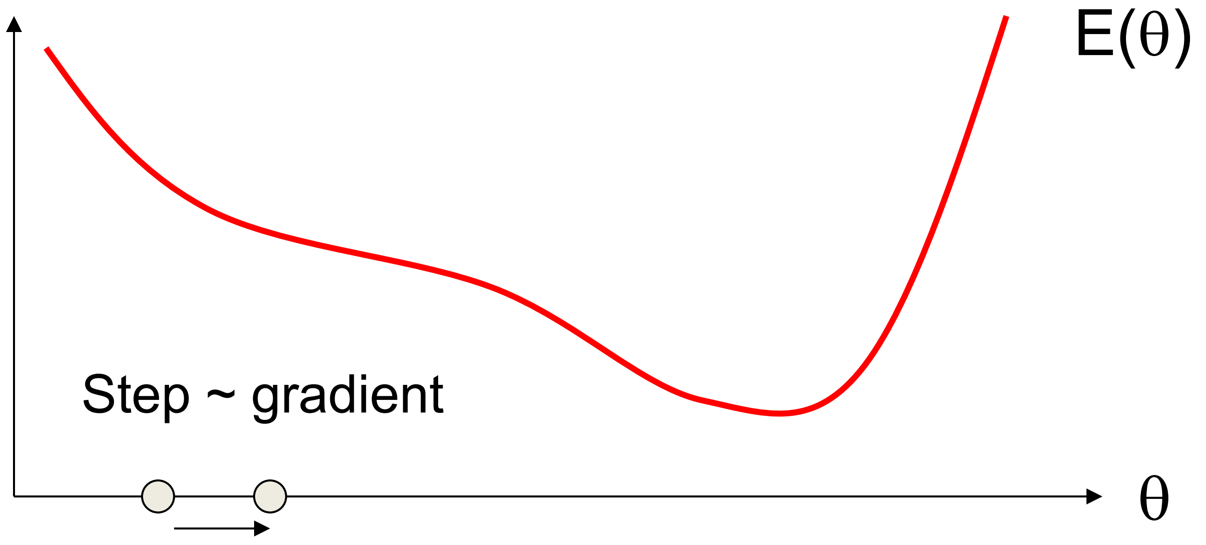
Understanding Backpropagation

- Minimize E(θ)
- Gradient Descent…



Understanding Backpropagation

- Minimize E(θ)
- Gradient Descent…



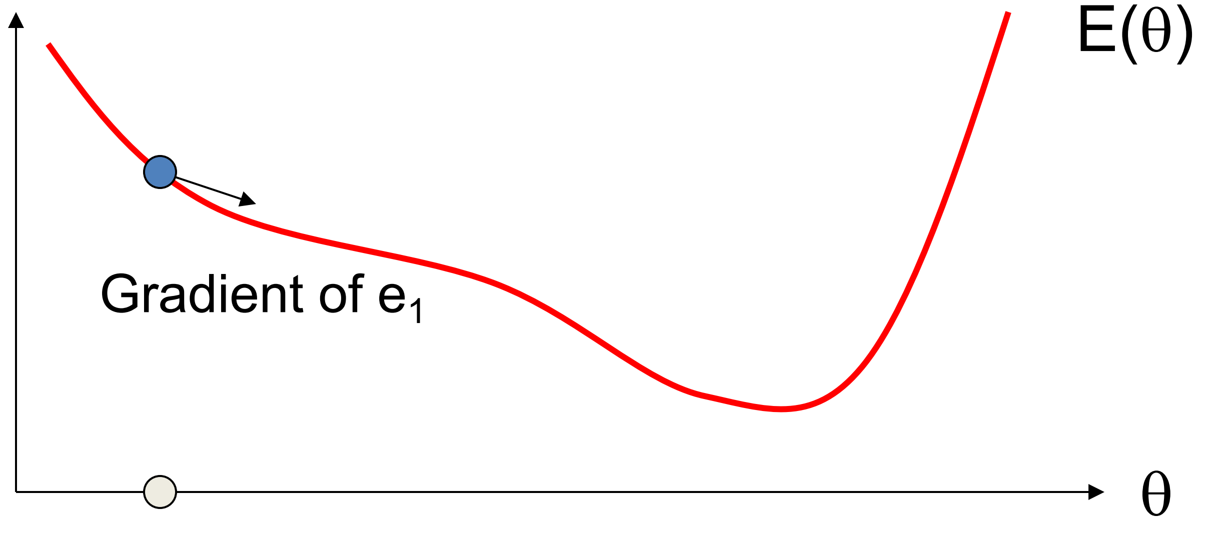
Stochastic Gradient Descent

- 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

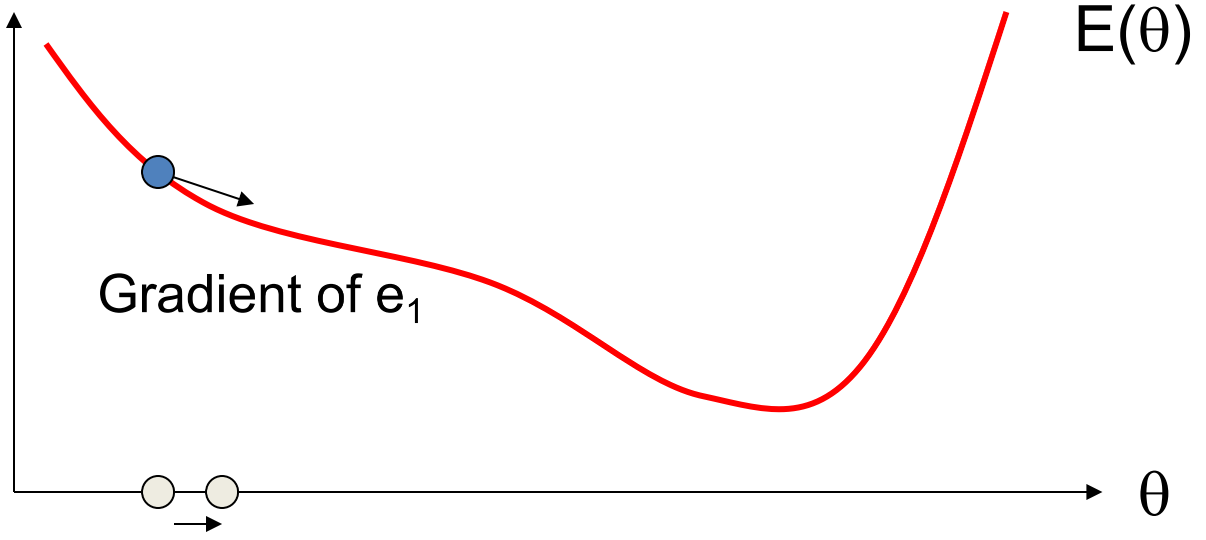
Understanding Backpropagation

- 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



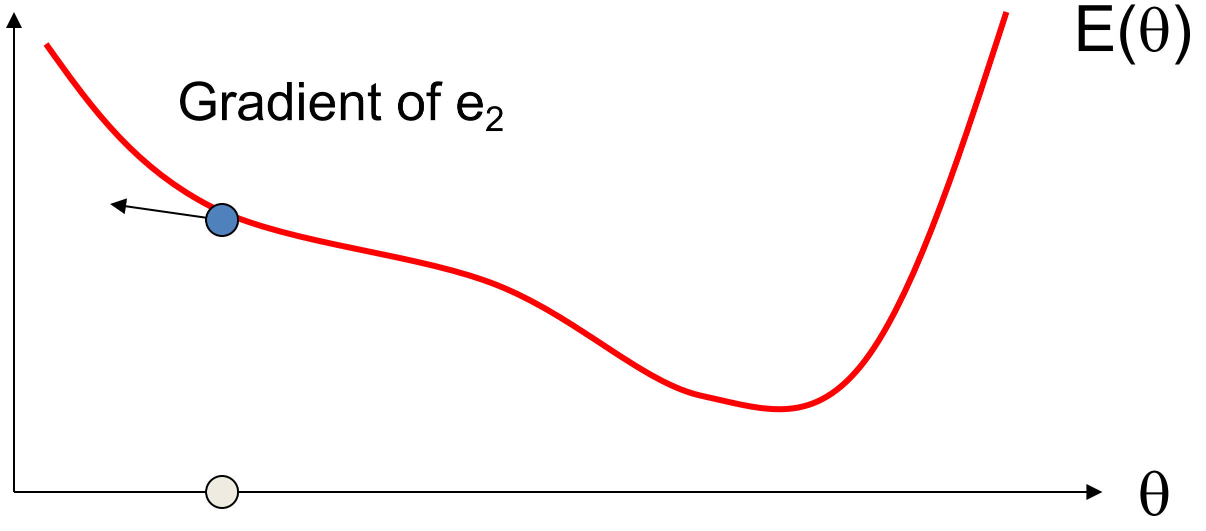
Understanding Backpropagation

- 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



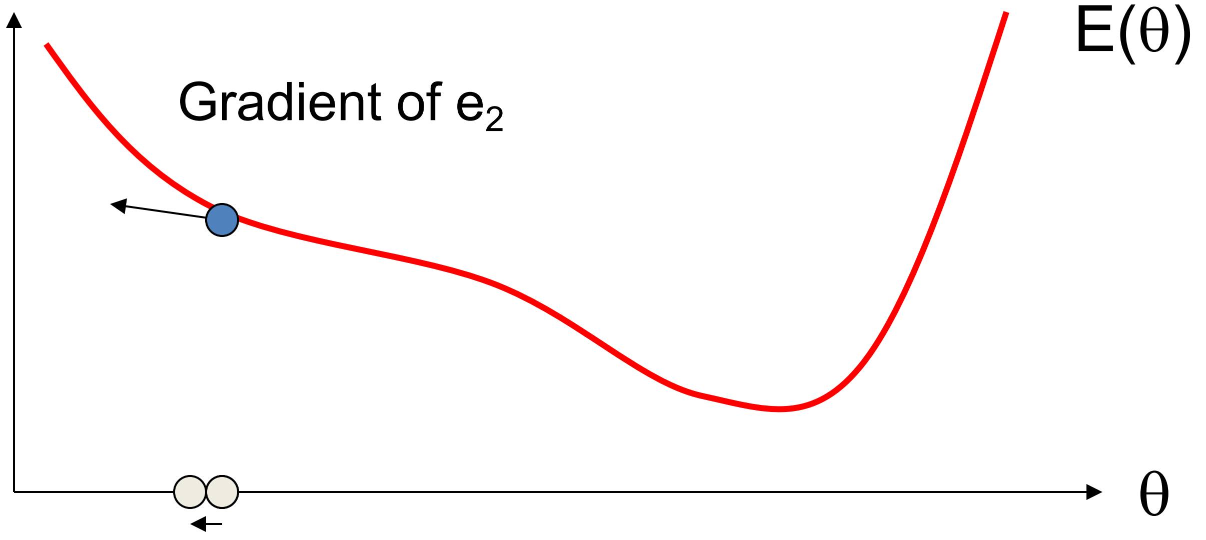
Understanding Backpropagation

- 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



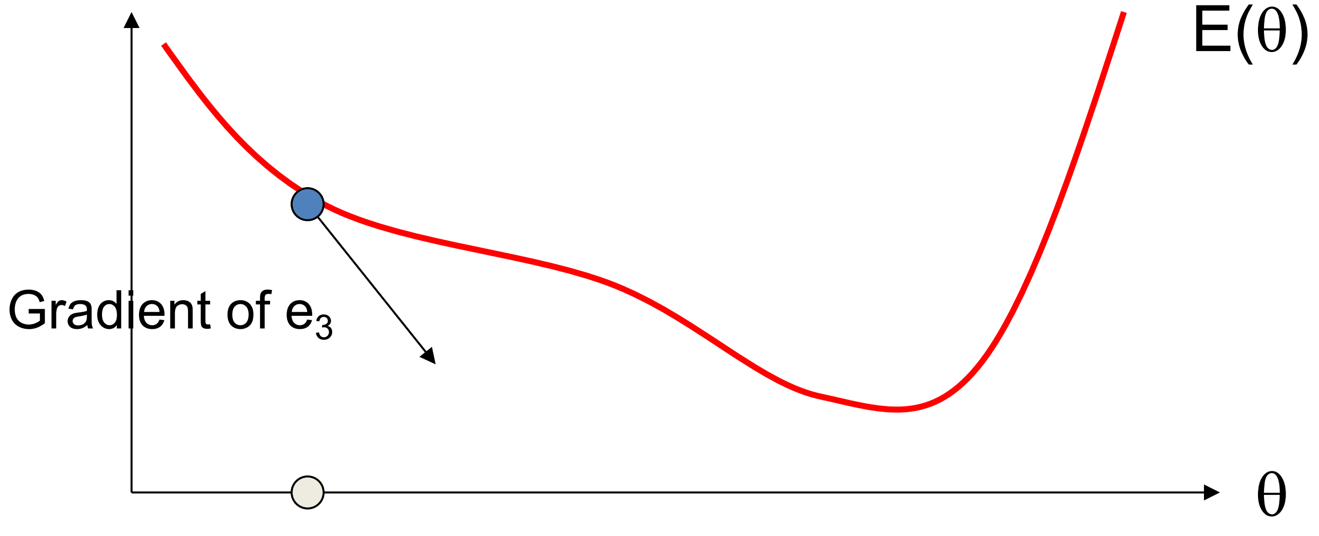
Understanding Backpropagation

- 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



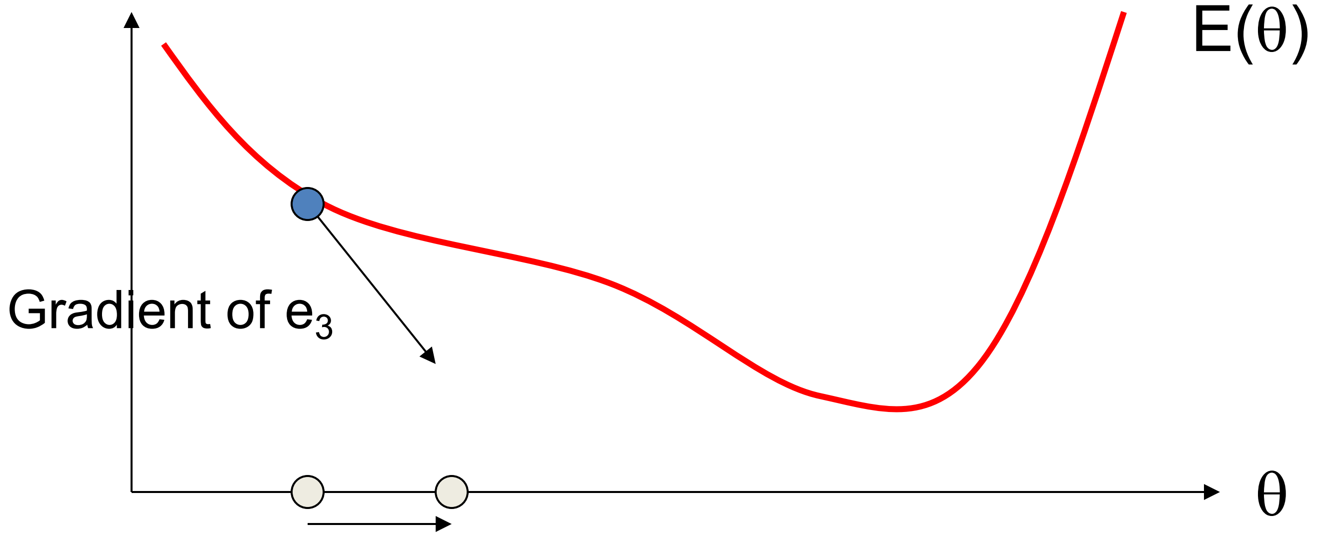
Understanding Backpropagation

- 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



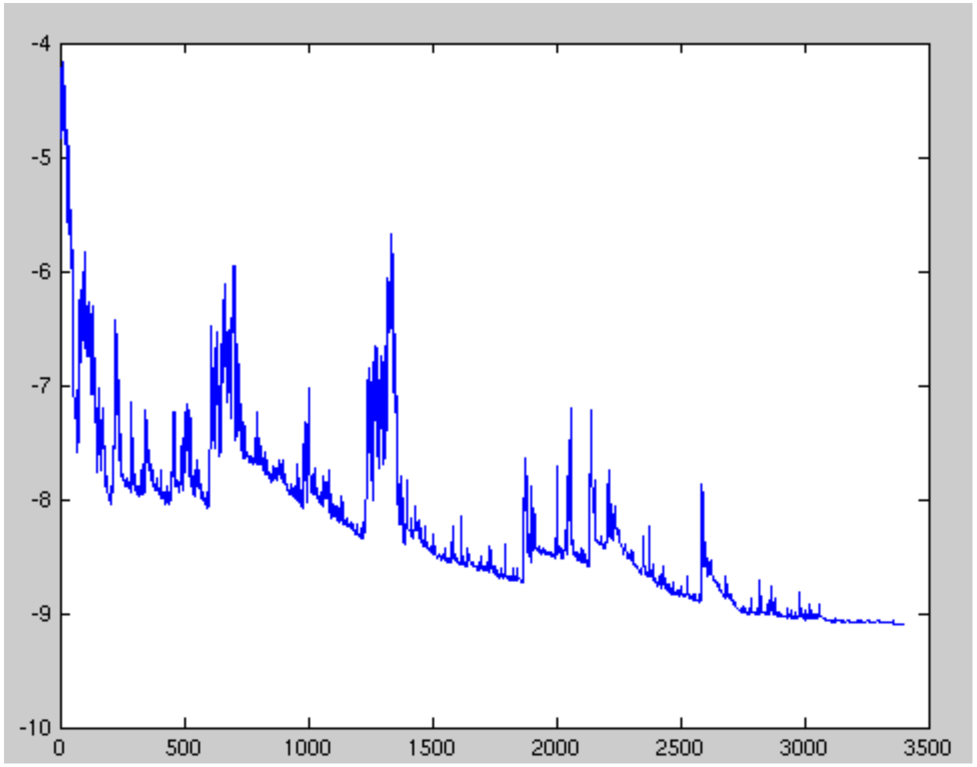
Understanding Backpropagation

- 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



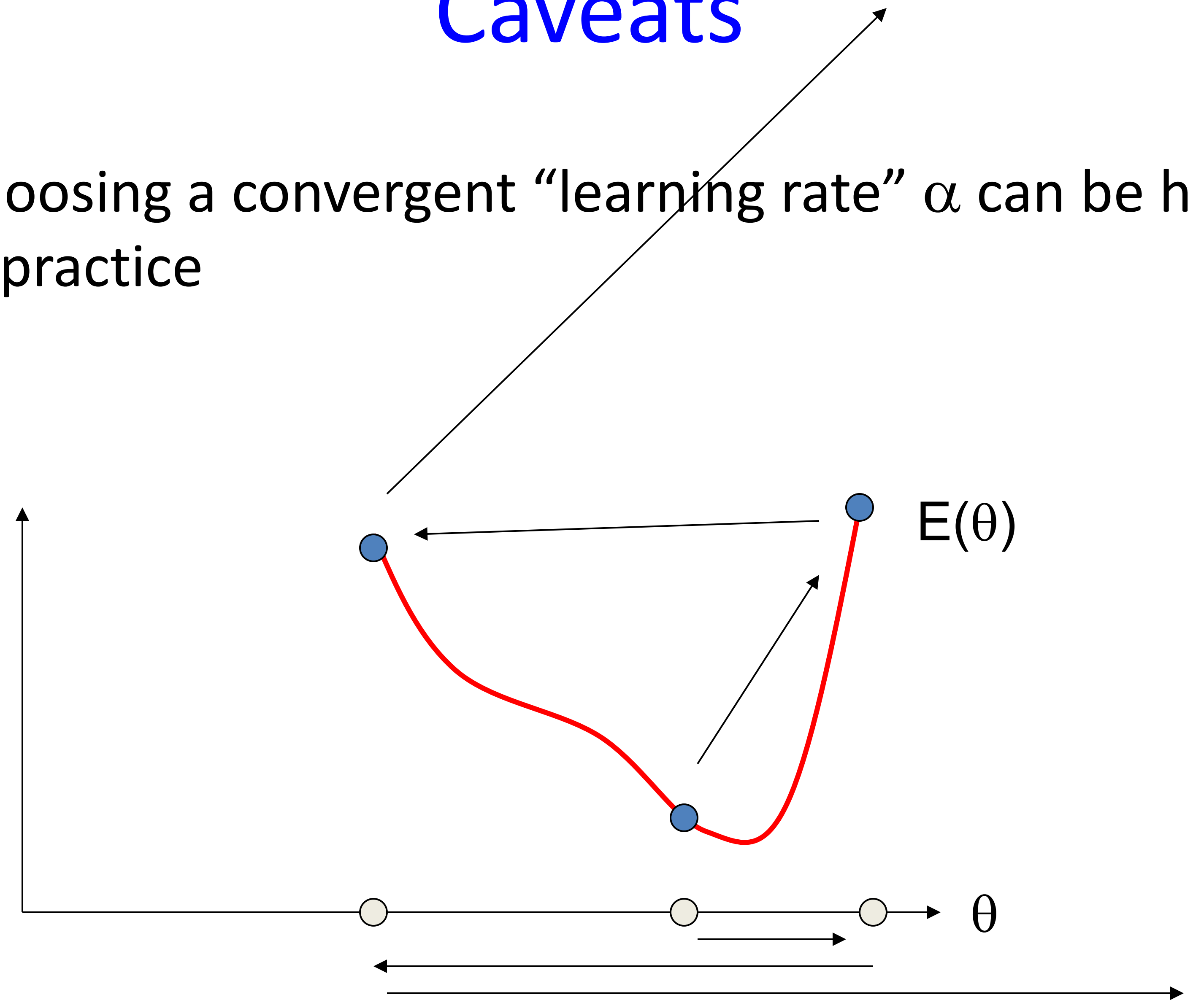
Stochastic Gradient Descent

- 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)



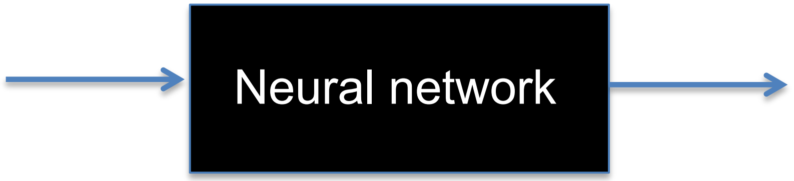
Caveats

- 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

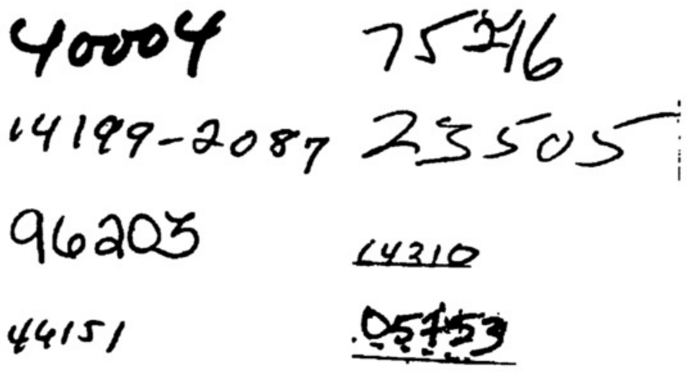
Pros and cons of different classifiers

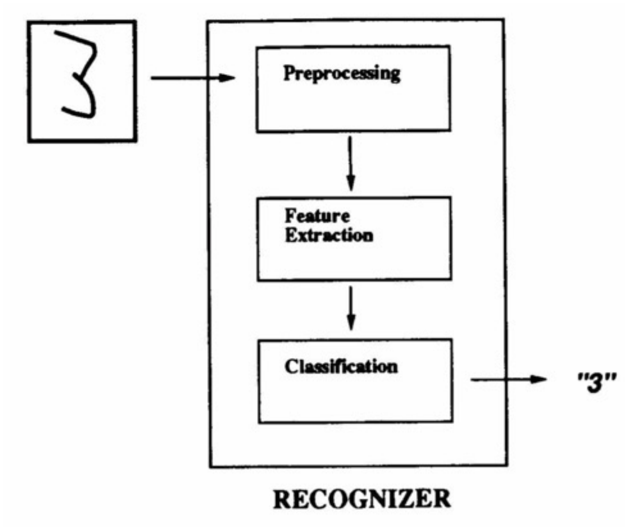
- Nearest neighbors
  - Can model any data, very prone to overfitting, requires distance function, fast learning, slow classification
- Neural networks
  - Models any function, requires structure, can suffer from local minima, slow learning, fast classification, difficult to interpret.
- Bayes nets
  - 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.

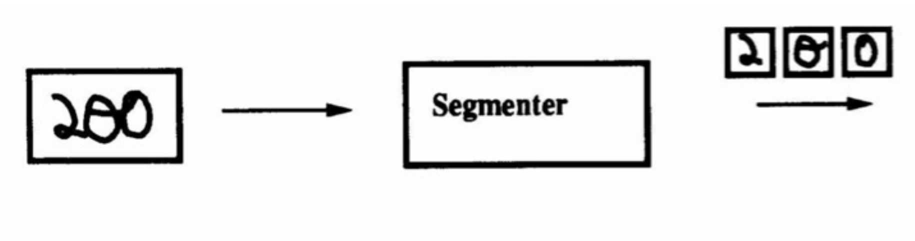
Neural Nets: 1960s-1990s

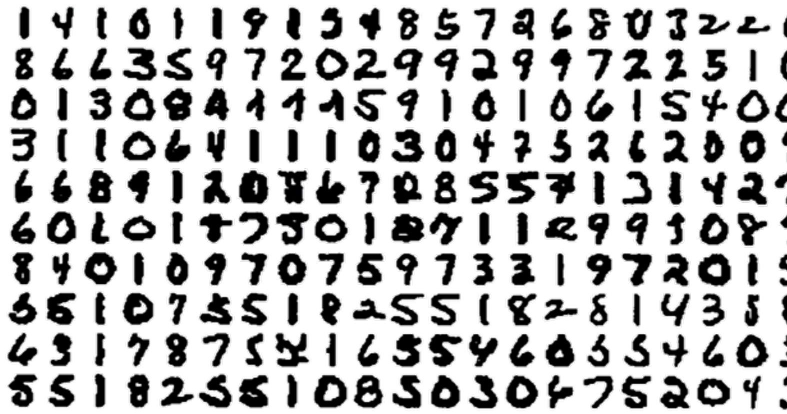
- Failure to deliver perceptron promises during during 1960s-1970s led to “AI winter”
- In 1980s, multi-layer networks and the backpropagation algorithm led to new excitement, new era of neural network research

Success story: handwritten digit recognition (LeCun, 1989)

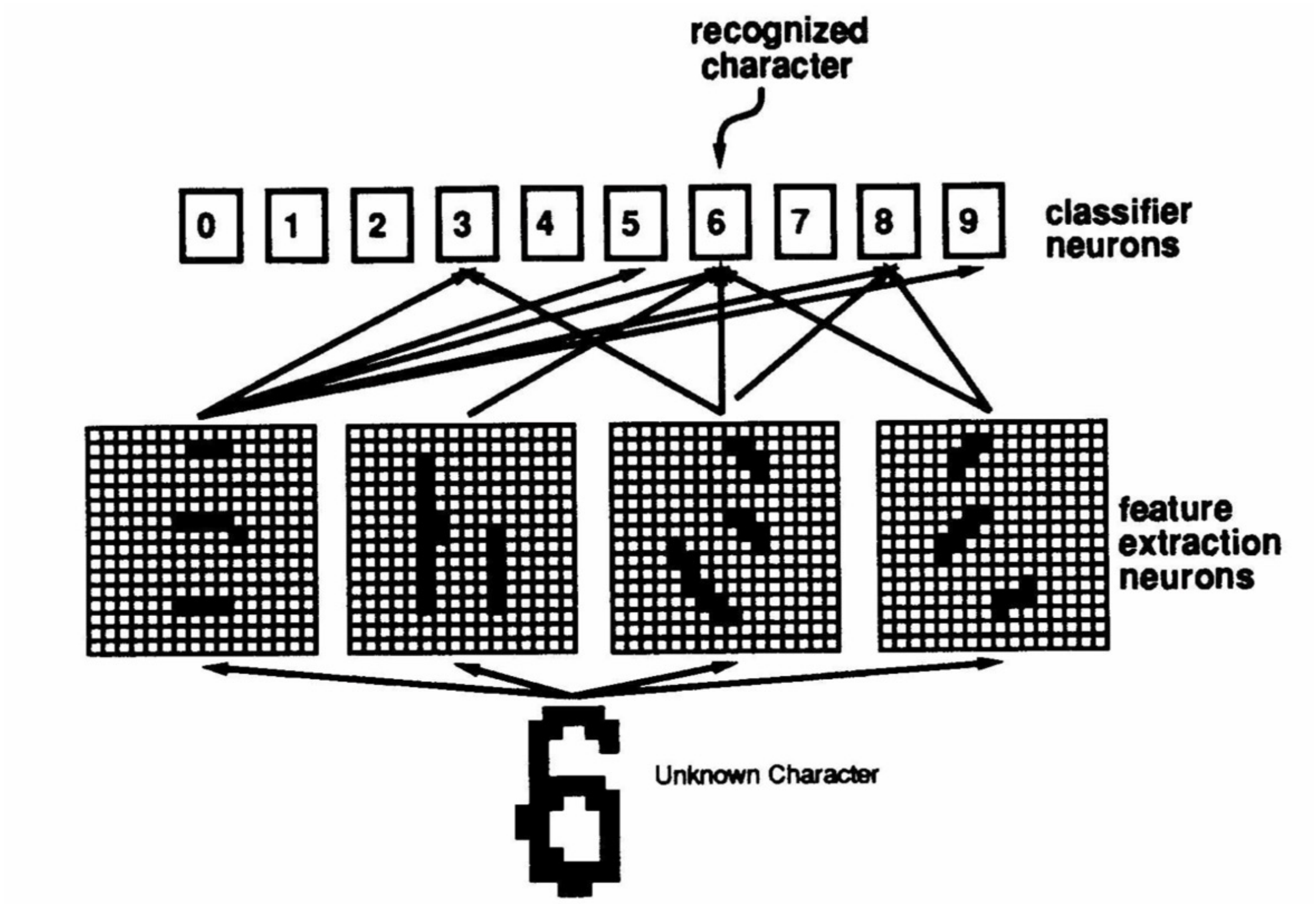




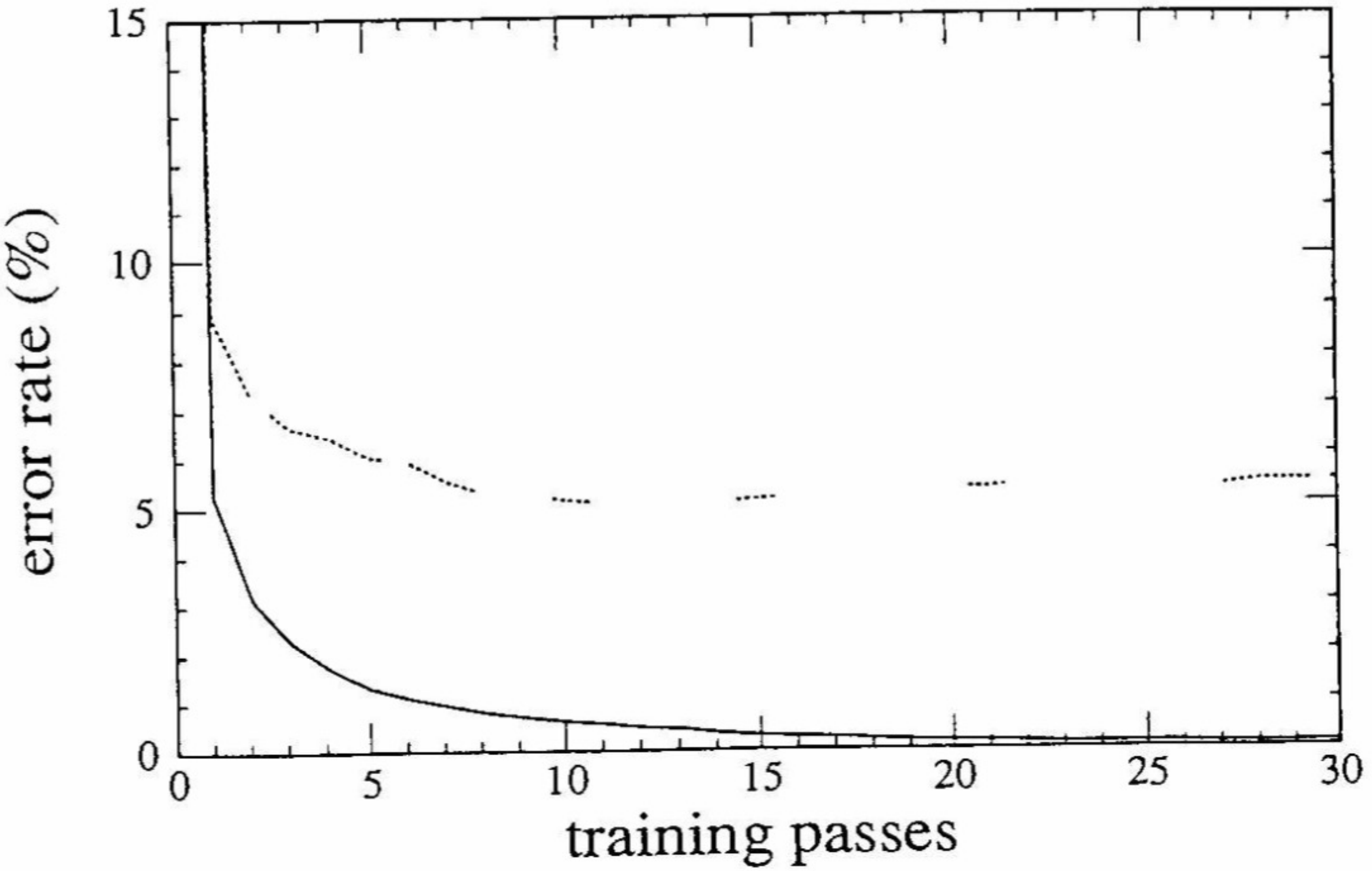




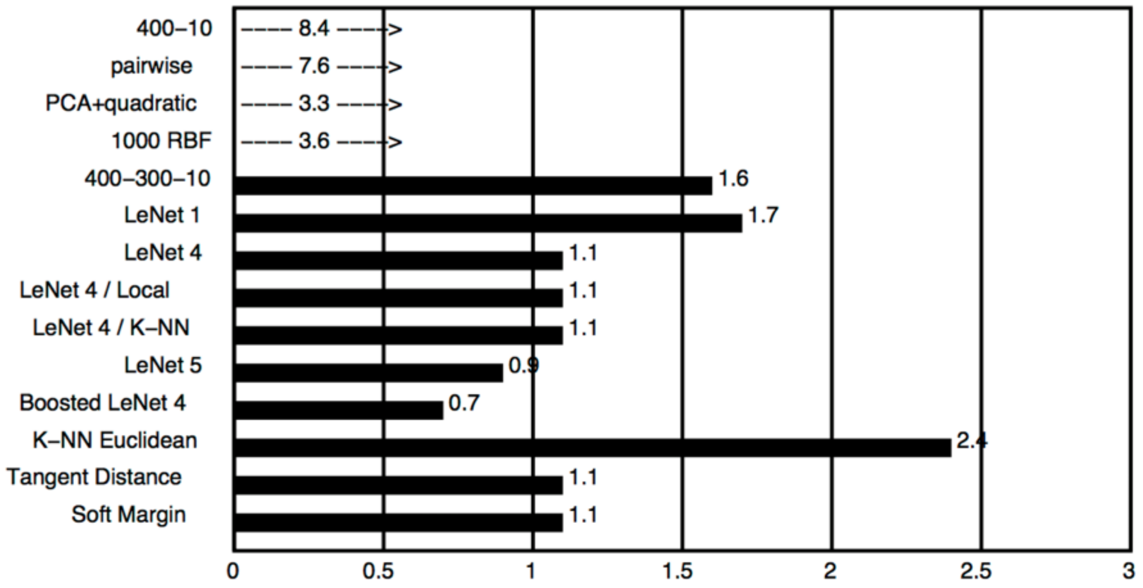
Network structure

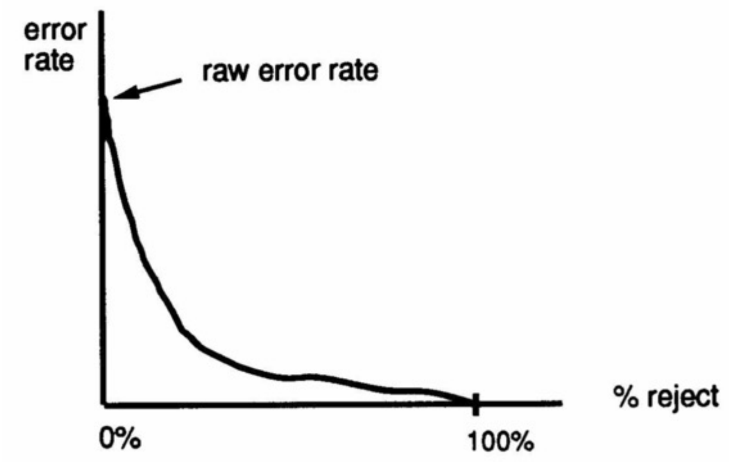


Use backprop to train

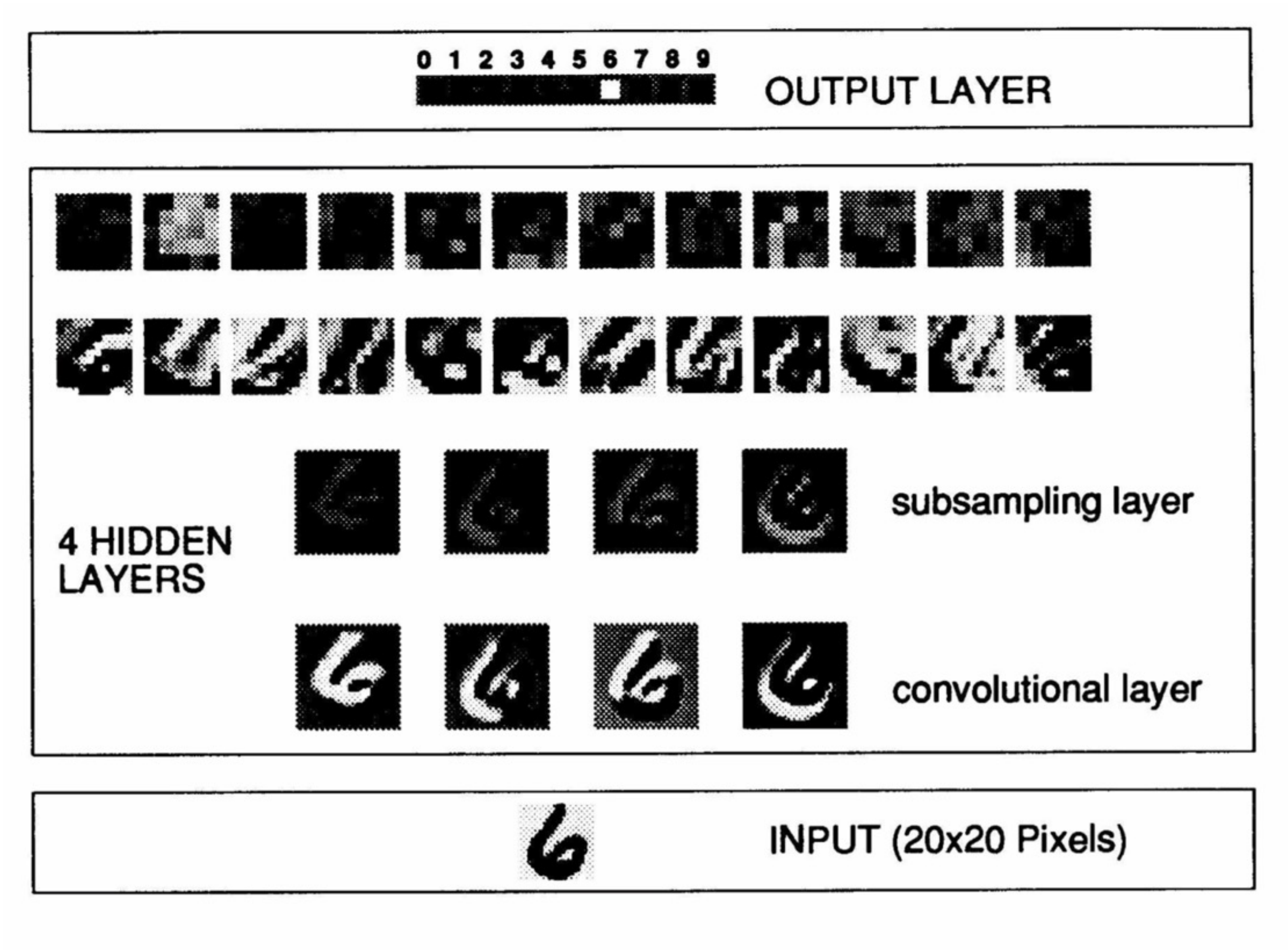


Worked better than other techniques (LeCun 1989)



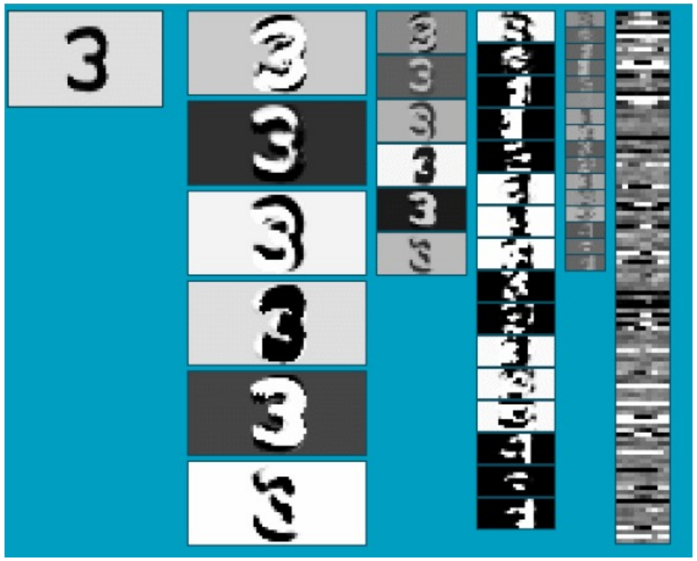


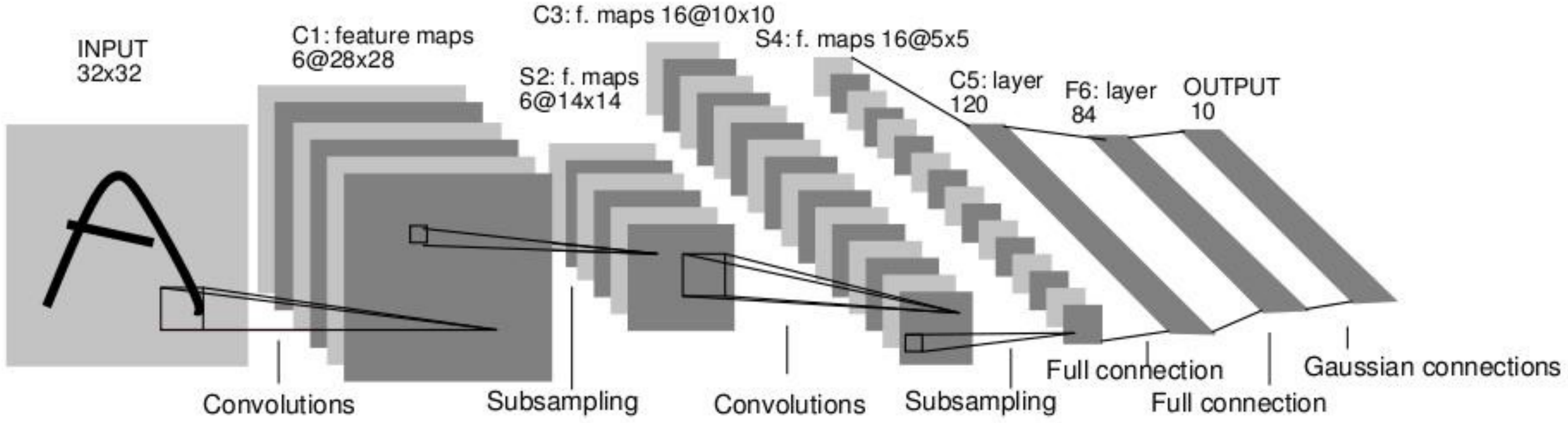
More complex architectures…



Convolutional Neural Networks

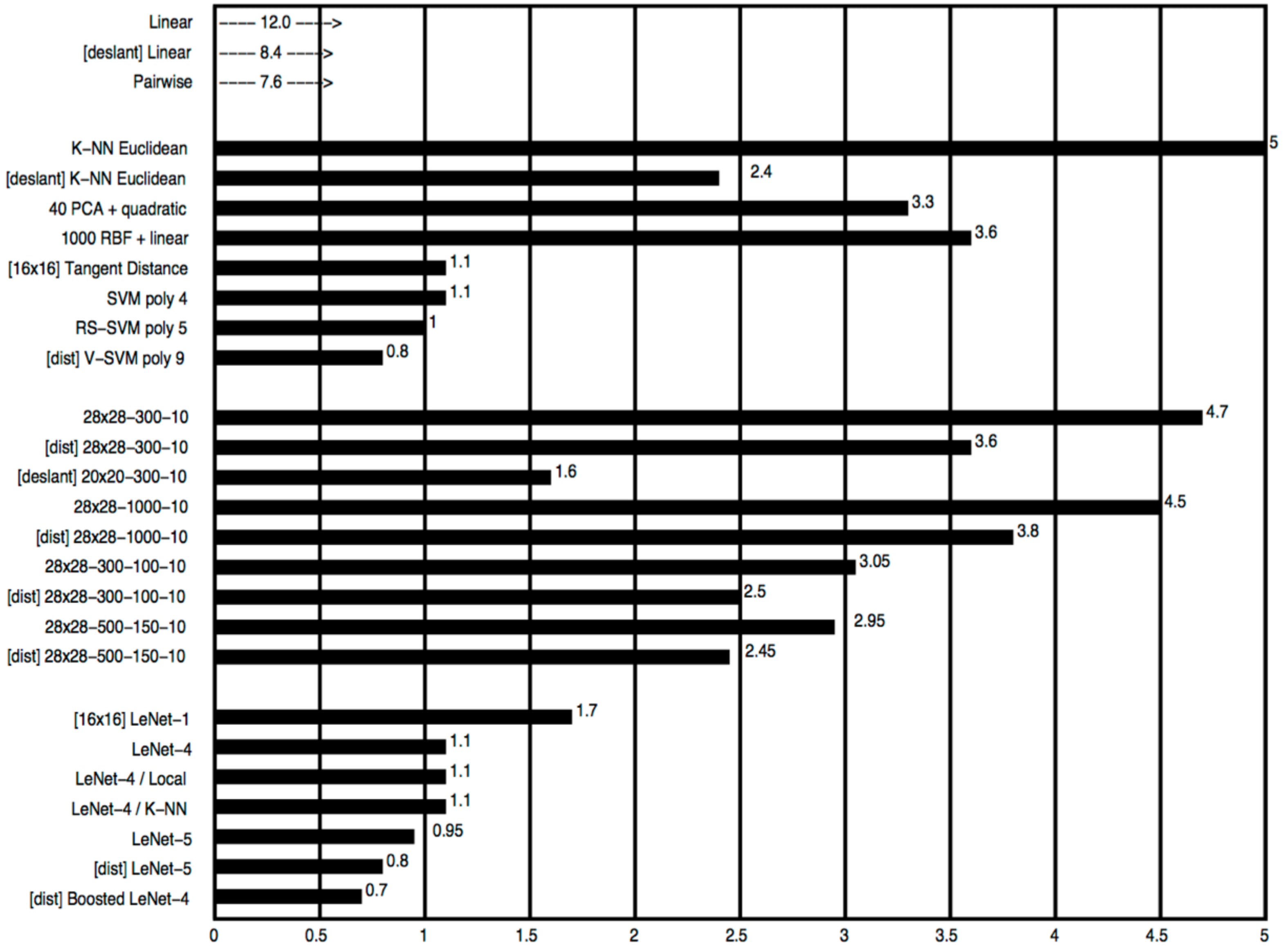
- Neural network with specialized connectivity structure
- Stack multiple stages of feature extractors
- Higher stages compute more global, more invariant features
- Classification layer at the end





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

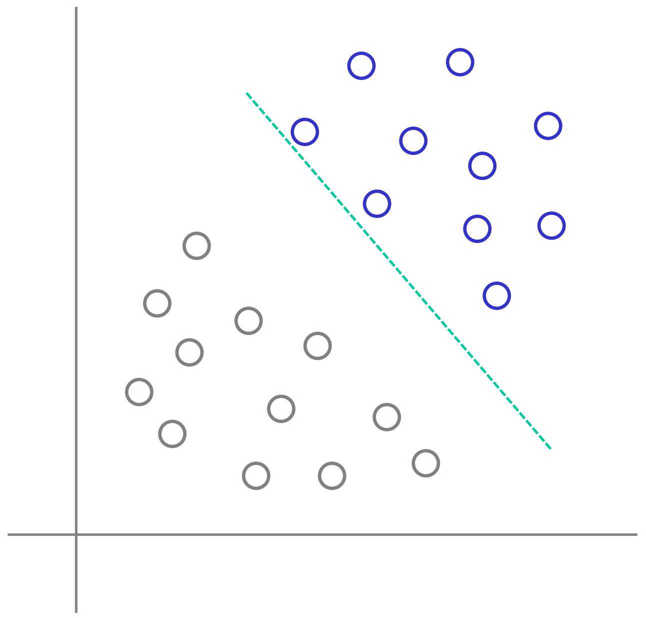
- Return to simpler techniques, like linear classifiers
  - But in high dimensions
  - Simpler learning algorithms, easier to justify theoretically
- Learn classifiers on manually-created features
  - E.g. not images themselves, but statistical features like color histograms, edge distributions, etc.

Support Vector Machines (SVM)

Reminder about perceptrons

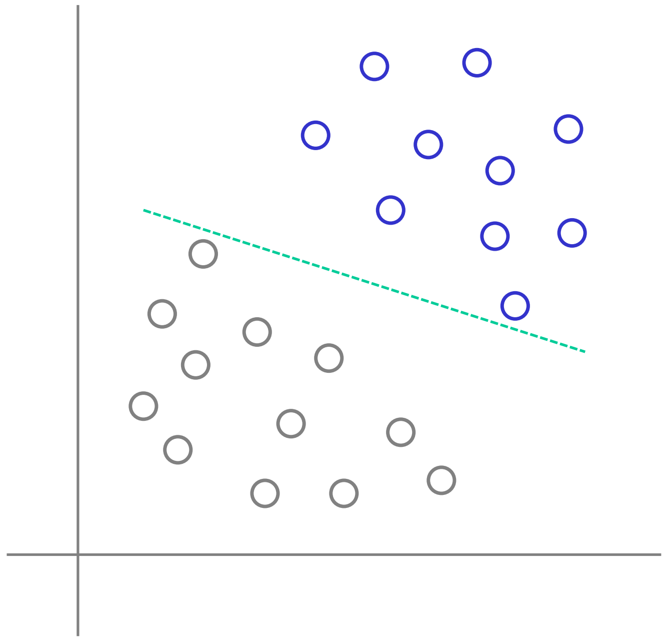
- **Perceptron Convergence Theorem**: If a classification problem is linearly separable, a perceptron will reach a solution in a finite number of iterations

- The solution weight vector is **<u>not</u>** unique. There are infinite possible solutions and decision boundaries.
  - Perceptrons find any separating hyperplane
  - The hyperplane depends on initialization and ordering of training points



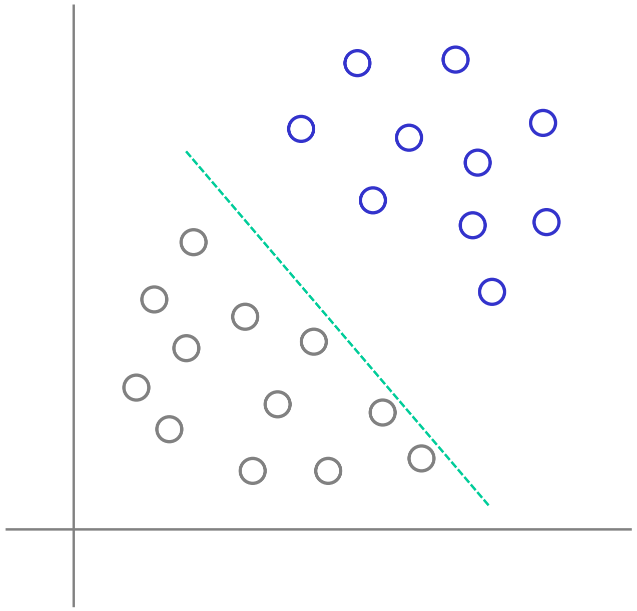
Reminder about perceptrons

- **Perceptron Convergence Theorem**: If a classification problem is linearly separable, a perceptron will reach a solution in a finite number of iterations
- The solution weight vector is **not** unique. There are infinite possible solutions and decision boundaries.
  - Perceptrons find any separating hyperplane
  - The hyperplane depends on initialization and ordering of training points
- ***If done differently***



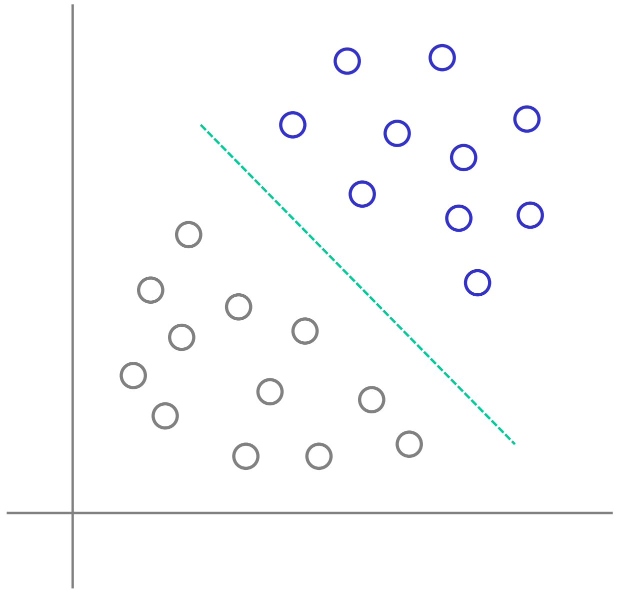
Reminder about perceptrons

- **Perceptron Convergence Theorem**: If a classification problem is linearly separable, a perceptron will reach a solution in a finite number of iterations
- The solution weight vector is **not** unique. There are infinite possible solutions and decision boundaries.
  - Perceptrons find any separating hyperplane
  - The hyperplane depends on initialization and ordering of training points
- ***If done differently….Again***



Reminder about perceptrons

- **Perceptron Convergence Theorem**: If a classification problem is linearly separable, a perceptron will reach a solution in a finite number of iterations
- The solution weight vector is <u>**not**</u> unique. There are infinite possible solutions and decision boundaries.
  - Perceptrons find any separating hyperplane
  - The hyperplane depends on initialization and ordering of training points
- <u>***If done differently….Again...And Again***</u>

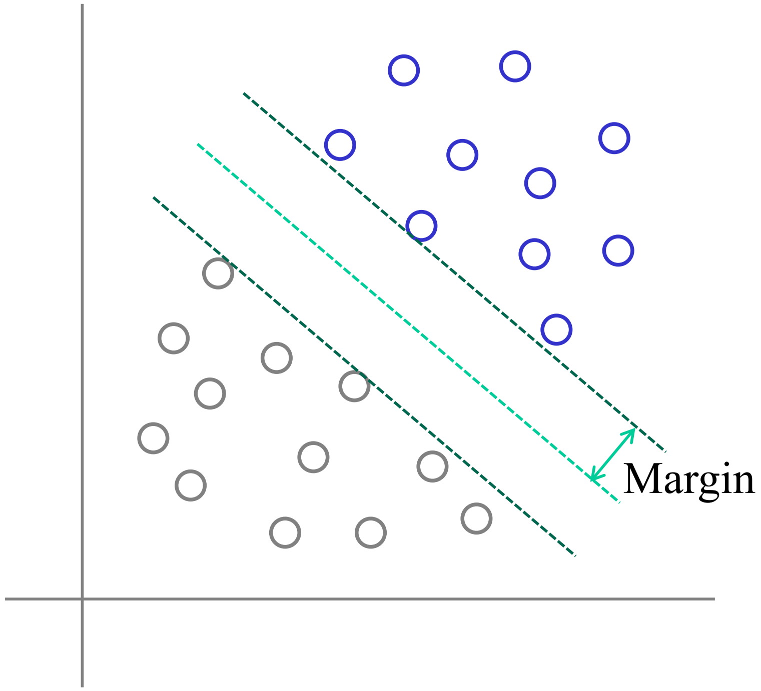


Motivation

- For a linearly separable classification task, there are generally infinitely many separating hyperplanes
  - Perceptron learning, however, stops as soon as one of them is reached
  - Some hyperplanes may be better than others

- To improve generalization, we want to place a decision boundary as far away from training classes as possible.
  - In other words, place the boundary at equal distances from class boundaries

Optimal Hyperplane



- Given a training sample, the support vector machine constructs a hyperplane as the decision surface in such a way that the margin of separation between positive and negative examples is maximized.

Next class

- More about SVM and neural networks