CS 383 – Machine Learning

Artificial Neural Networks

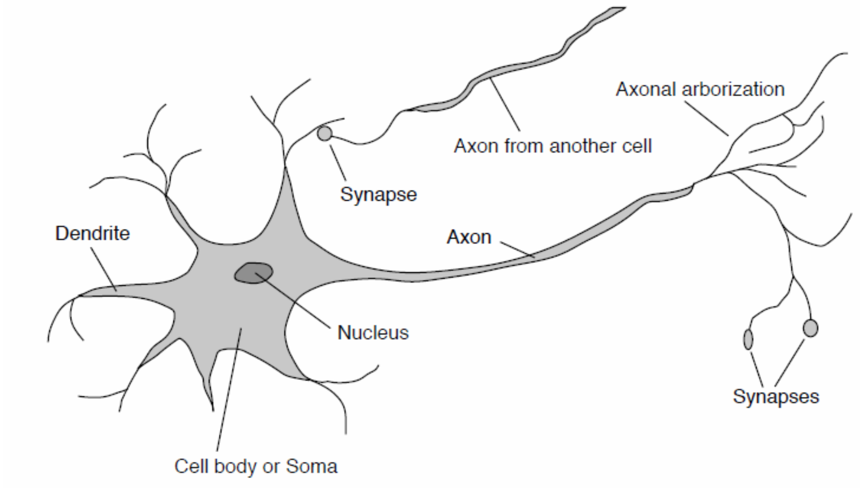
Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

Objectives

- Perceptrons
- Artificial Neural Networks

Artificial Neural Networks

- Based on the belief that it was necessary to model underlying brain architecture for AI/ML
  - "Learning is altering strength of synaptic connections"
- Attempt to build a computation system based on the parallel architecture of brains
- Characteristics
  - Many simple processing elements
  - Many connections
  - Simple messages
  - Adaptive interaction

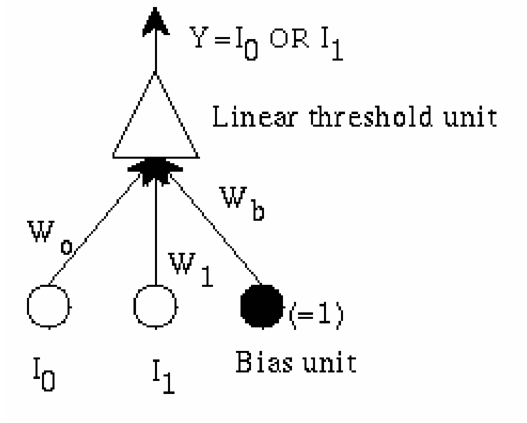


Benefits of ANNs

- Can use for categorical or continuous outputs
- Generalizable/multi-purpose
- Non-linear
- Noise tolerant

Perceptron

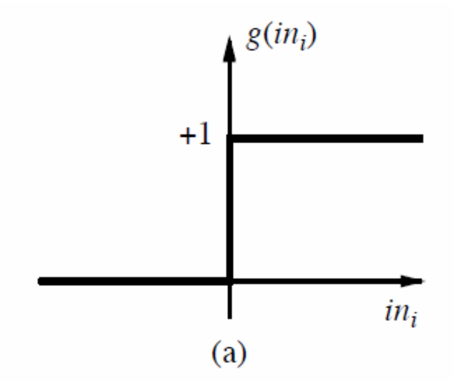
- The foundation of artificial neural networks is the *perceptron.*
- A perceptron takes in several values, weights them, and provides an output
  - Very much like linear or logistic regression
- Typically a function, $g(x)$, is applied to this weighted input, $x\theta$.
  - Or in the figure below, Iw
- One potential function is the step function:
  - If $x\theta > 0$ then output 1
  - Otherwise, output 0

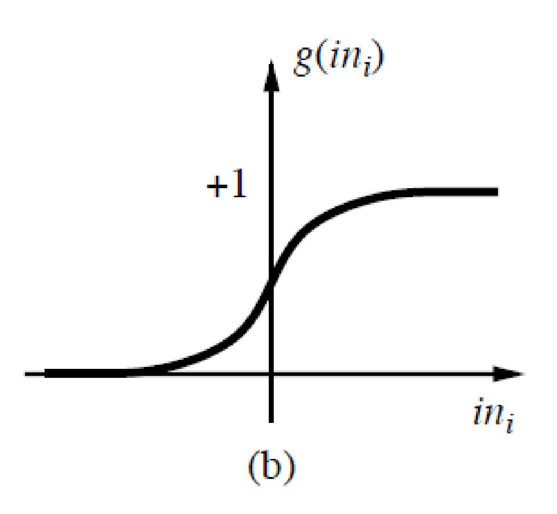


Activation Functions

- This function, $g(x)$, is called the *activation function*
- However, the step function is not differentiable.
- Therefore a more common activation function is our good friend the sigmoid/logistic function:

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





Perceptron Learning Rule

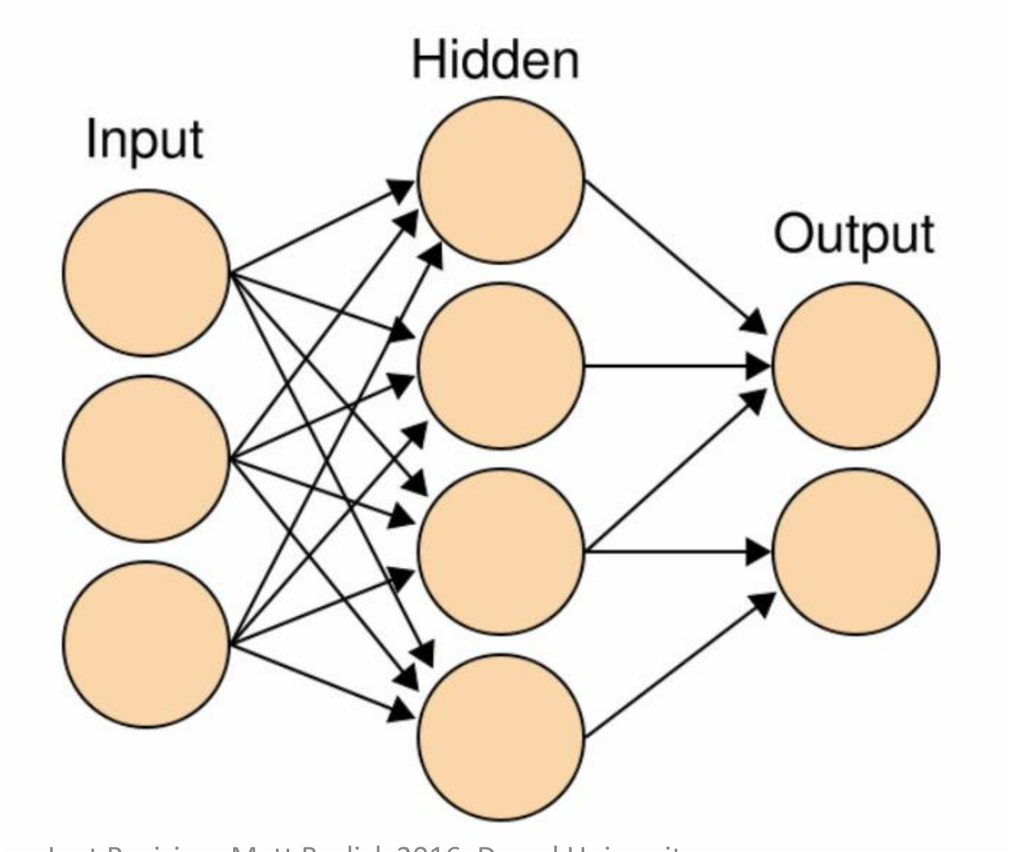
- Ok so we need to learn the perceptron weights
- From logistic regression we know that one way to do this is by gradient descent/ascent
  - Change the weight by an amount proportional to the difference between the desired output and the actual output
  - As an equation:

$$\theta_j = \theta_j + \eta\big(y - g_\theta(x)\big)x_j$$

  - Stops when converges

Multilayer Perceptrons (MLP)

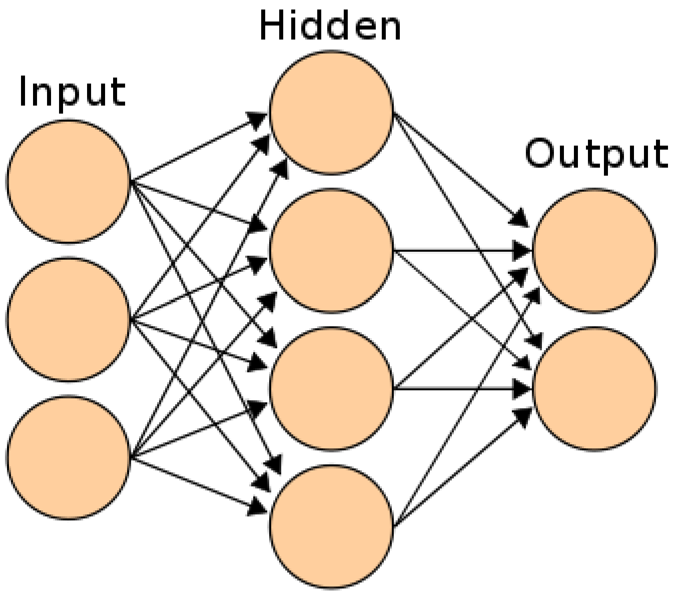
- Artificial Neural networks are also called *multilayer perceptrons*

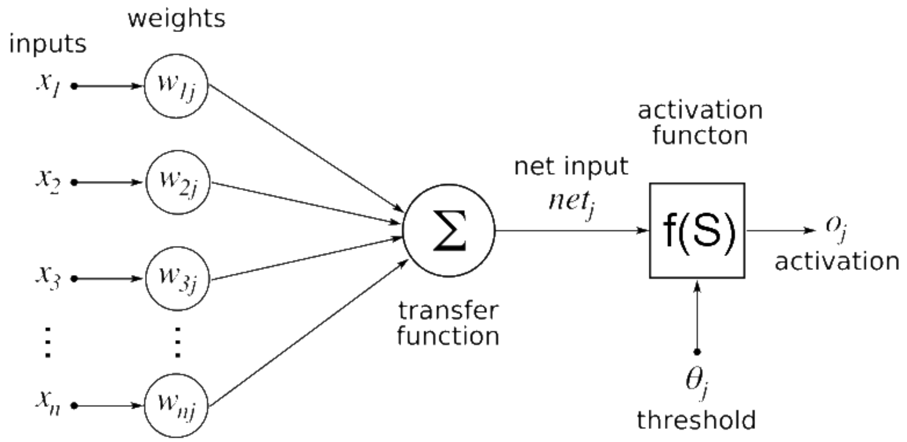


Artificial Neural Network (ANN)

Three layers:

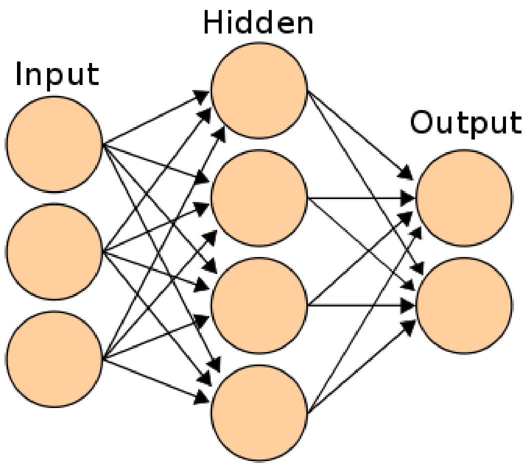
1. Input layer (features)
2. Hidden layer
3. Output layer (predicted values)





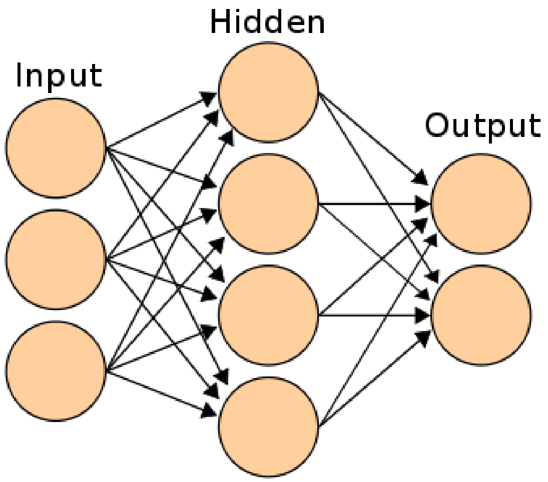
ANN

- For each internal/hidden node
  - Sum of weighted inputs gets processed by an activation function
  - The output is in the range [0,1]
- For each output node
  - Output value is sum of weighted inputs from the last hidden layer is passed through activation function



Training an ANN

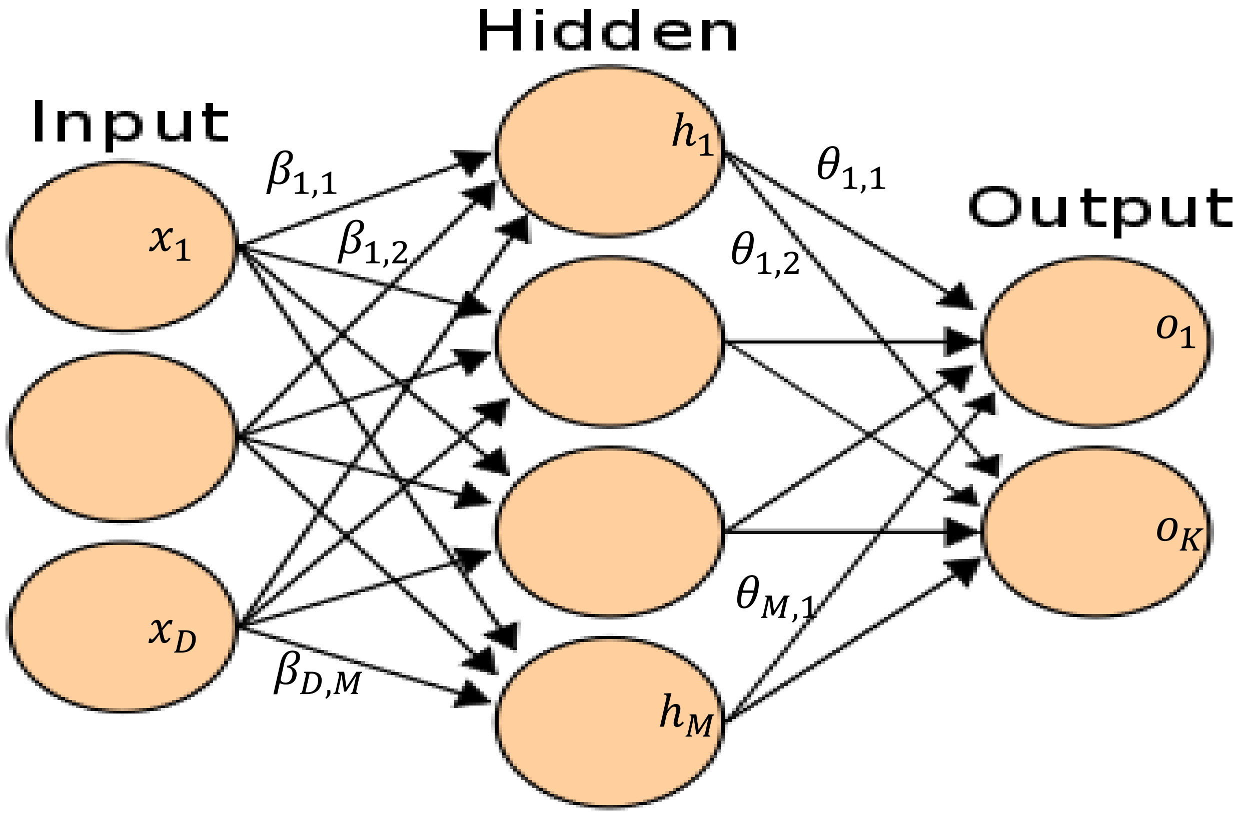
- Now we have lots of weights to learn
  - Each hidden layer node weights the output of the input layer.
  - Each output layer node weights the output of the hidden layer node.
- Our goal is to learn all of these weights in order to minimize the training error
- To do this, we use what's called *forward-backward propagation.*



ANN Notation

- Our artificial neural network will have
  - D Input nodes:
  - The output of hidden layer node j is denoted h_j
  - The output of output layer node k is denoted o_k
  - A matrix (or vector of vectors) of weights connecting input nodes to hidden layer nodes, β such that the connection from input layer node i to hidden layer node j is β_{ij}
  - A matrix (or vector of vectors) of weights connecting hidden layer nodes to output layer nodes, θ, such that the connection from hidden layer node j to output layer node k is θ_{jk}

ANN Notation



Forward-Backward Propagation

- Compute the values going into and out of each node given a feature vector is called *forward propagation.*
    - We use this to predict values for new data
- In training our ANN we must also propagate back errors. This is called *backward propagation*
- Let’s first look at forward propagation.

Forward Propagation

- For each hidden layer node, j
  - Compute its output value using activation function g, the input data x and the weights from the input layer to this hidden layer β :

$$h_j = g\left(x\beta_{:j}\right)$$

- Now for each output layer node, k
  - Compute the value being output by it using the output of the hidden layer, h, the node's weight vector, $\theta_{:,k}$, and the activation function, g:

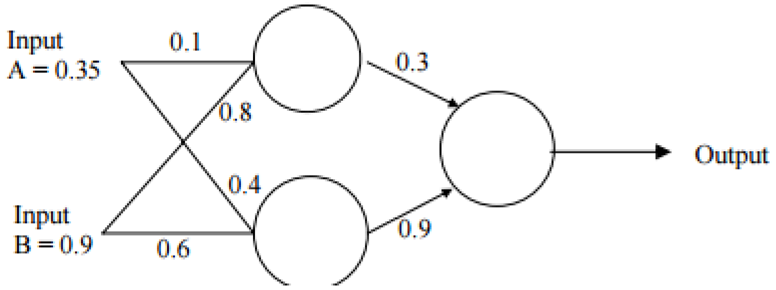
$$o_k = g\left(h\theta_{:,k}\right)$$

- Note we can do this easily with matrix math…

$$h = g(x\beta)$$
$$o = g(h\theta)$$

Forward Propagation Example

- Input to top middle node
  - (0.35*0.1+0.9*0.8)=0.755
- Output of top middle node
  - h_1 =1/(1+exp(-0.755))=0.68
- Input to bottom middle node
  - (0.35*0.4+0.9*0.6)=0.68
- Output of bottom middle node
  - h_2 =0.6637
- Input to output node
  - (0.68*0.3+0.6637*0.9)=0.8013
- Output of output node
  - o_1 =0.6903



Back Propagation

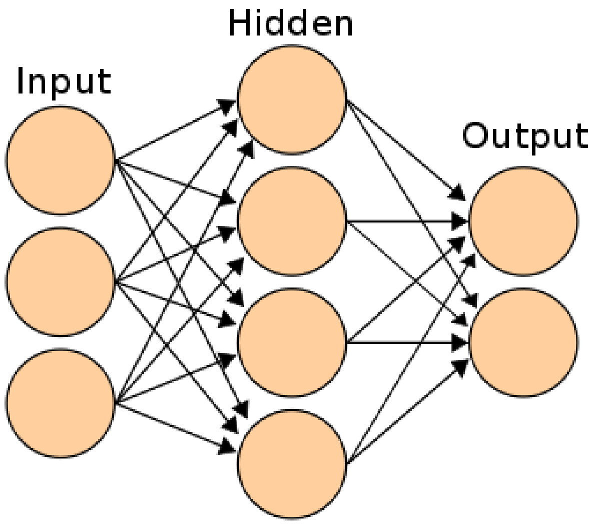
- Now we must *propagate* this error *backwards*
- This process basically distributes the overall error computed at the output nodes to the output weights and the hidden layer weights.
- Before we go into the entire derivation, let's see what happens via some intuition..
- For all of this we'll assume a single observation
  - But we'll show batch mode later.
- Let δ_j be related to the error at node j
- Let y_j be the target output at output layer node j
  - Since ANNs can have multiple output nodes, we can have different target values at each output node.

Back Propagation

- **If node *j* is an output node**, then our signed error is easy:

$$\delta_j = (y_j - o_j)$$

    - We'll see where this formally comes from in a bit…



Back Propagation

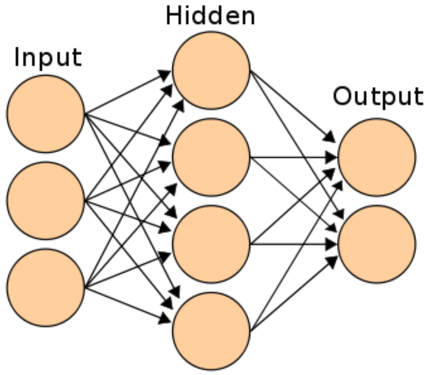
- **If node *j* is a non-output node**, then it's error will be affected by the weighted sum of the errors it goes to, where the weights are weight of the edges going from it to node in the next layer.

$$\sum_{k=1,..,K} \delta_k \theta_{jk}$$

- This is then scaled by $h_j(1 - h_j)$
  - We'll see where this comes from in a bit…
- So the weighted error at the hidden layer will then be:

$$\delta_j = \left(\sum_{k=1,..,K} \delta_k \theta_{jk}\right) h_j(1 - h_j)$$

where K is the number of output nodes.



Back Propagation

- Recall from our other gradient ascent/descent algorithms, the gradient is typically the signed error, weighed by what's coming into it
- Therefore we have:

$$\frac{\partial E}{\partial \theta_{jk}} = \delta_k h_j$$

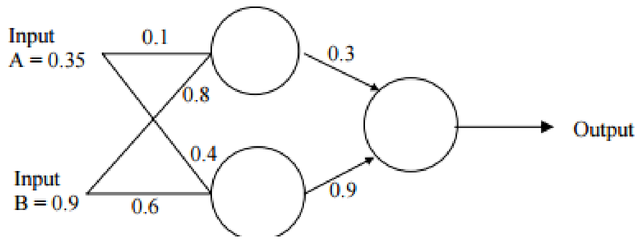
$$\frac{\partial E}{\partial \beta_{ij}} = \delta_j x_i$$

- Resulting in our final update rules:

$$\theta_{jk} += \eta \delta_k h_j$$
$$h_{ij} += \eta \delta_j x_j$$

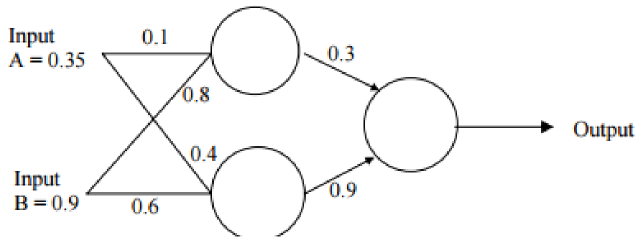
- Returning to our example, let's see this in action!

Backpropagation



- Recall from forward propagation:
$$h_1 = 0.68, h_2 = 0.6637, o_1 = 0.6903$$
- For backwards propagation training we'll let the target be $y_1 = 0.5$ and rate $\eta = 1.0$
- Let's first compute the error at the output node:
- Error in output node $\delta_k = (y_k - o_k)$
  - δ_1 =(0.5-0.69) = -0.19
- Update weights from hidden to output $\theta_{jk} += \alpha\delta_k h_j$
  - $\theta_{1,1}$=0.3+1.0(-0.19*0.68)=0.1708
  - $\theta_{2,1}$=0.9+1.0(-0.19*0.6637)=0.7739

Backpropagation



- Recall from forward propagation:

$$h_1 = 0.68, h_2 = 0.6637, o_1 = 0.6903$$

- And from back-propagating from the output layer…

$$\theta_{1,1} = 0.1708, \theta_{2,1} = 0.7739, \delta_1 = -0.19$$

- Error in hidden layer $\delta_j = \left(\sum_{k \in outputs} (\theta_{jk}\delta_k)\right)\ h_j(1 - h_j)$
  - δ_1 = (0.1708*-0.19)*0.68*(1-0.68) = -0.0071
  - δ_2 = (0.7739*-0.19)*0.6637*(1-0.6637) = -0.0328
- Update weights from input layer to hidden layer

$$\beta_{ij} += \alpha\delta_j x_i$$

  - $\beta_{1,1}$=0.1+1.0(-0.0071*0.35)=0.992
  - Etc..

Back Propagation Derivation

- Ok so now that we have some “intuition”, lets see where these rules come from.
- First lets define net_j to be what goes into a node j
  - If j is an output node this is:

$$net_j = h\theta_{:,j}$$
$$o_j = g(net_j)$$

  - Otherwise (we’ll assume only a single hidden layer) it is:

$$net_j = x\beta_{:,j}$$
$$h_j = g(net_j)$$

- Ultimately we want to find the gradients of the error with respect to the parameters:

$$\frac{\partial E}{\partial \theta_{jk}} \text{ and } \frac{\partial E}{\partial \beta_{ij}}$$

Back Propagation Derivation

- The key to finding these is via the *chain rule*

$$\frac{\partial E}{\partial \theta_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial \theta_{jk}}$$

$$\frac{\partial E}{\partial \beta_{ij}} = \frac{\partial E}{\partial h_j} \cdot \frac{\partial h_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial \beta_{ij}}$$

- Let's solve the partials one at a time (and first for θ then for β)!

Back Propagation Derivation

$$\frac{\partial E}{\partial \theta_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k} \cdot \frac{\boldsymbol{\partial net_k}}{\boldsymbol{\partial \theta_{jk}}}$$

$$\frac{\partial net_k}{\partial \theta_{jk}} = \frac{\partial}{\partial \theta_{jk}}\left(h\theta_{:,k}\right) = \frac{\partial}{\partial \theta_{jk}}\left(\sum_{\{i \in hiddenlayer\}} h_i \theta_{ik}\right)$$

- This will only exist for $i = j$ and therefore:

$$\frac{\partial net_k}{\partial \theta_{jk}} = h_j$$

Back Propagation Derivation

$$\frac{\partial E}{\partial \theta_{jk}} = \frac{\partial E}{\partial o_k} \cdot \boldsymbol{\frac{\partial o_k}{\partial net_k}} \cdot h_j$$

- Since net_k is the value coming *into* node k and o_k is the value coming *out* of it, we need the activation function!

$$\frac{\partial o_k}{\partial net_k} = \frac{\partial}{\partial net_k}\big(g(net_k)\big)$$

- From logistic regression recall that if $g(z) = \frac{1}{1+e^{-z}}$ *then*

$$\frac{\partial}{\partial z} g(z) = g(z)\big(1 - g(z)\big)$$

- Therefore $\frac{\partial o_k}{\partial net_k} = g(net_k)\big(1 - g(net_k)\big)$
- For the output layer $g(net_k) = o_k$, so we have:

$$\frac{\partial o_k}{\partial net_k} = o_k(1 - o_k)$$

Back Propagation Derivation

$$\frac{\partial E}{\partial \theta_{jk}} = \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial o_k}} \cdot o_k(1 - o_k) \cdot h_j$$

- Since we're doing classification, for our error function let's use that MLE equation from logistic regression!
- For a single sample this would be:

$$E = yln(o) + (1 - y)\ln(1 - o)$$

- The partial of this will be (work through it!):

$$\frac{\partial E}{\partial o_k} = \frac{(y - o_k)}{o_k(1 - o_k)}$$

Back Propagation Derivation

- Finally we have a rule for the gradient with respect to the weights going from the training layer to the output layer:
$$\frac{\partial E}{\partial \theta_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial \theta_{jk}} = \frac{(y - o_k)}{o_k(1 - o_k)} \cdot o_k(1 - o_k) \cdot h_j$$
- If we let $\delta_k = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k} = \frac{\partial E}{\partial net_k}$ then

$$\delta_k = \frac{\partial E}{\partial net_k} = \frac{(y - o_k)}{o_k(1 - o_k)} \cdot o_k(1 - o_k) = (y - o_k)$$

- And
$$\frac{\partial E}{\partial \theta_{jk}} = \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial \theta_k} = \delta_k \cdot h_j$$

Back Propagation Derivation

- Let's do this whole thing again for the weights going from the input layer to the hidden layer…

$$\frac{\partial E}{\partial \beta_{ij}} = \frac{\partial E}{\partial h_j} \cdot \frac{\partial h_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial \beta_{ij}}$$

- $\frac{\partial net_j}{\partial \beta_{ij}} = \frac{\partial}{\partial \beta_{ij}}\left(x\beta_{:,j}\right) = x_i$
- $\frac{\partial h_j}{\partial net_j} = \frac{\partial}{\partial net_j}\left(g\left(net_j\right)\right) = g\left(net_j\right)\left(1 - g\left(net_j\right)\right) = h_j\left(1 - h_j\right)$

Back Propagation Derivation

$$\frac{\partial E}{\partial \beta_{ij}} = \frac{\partial E}{\partial h_j} \cdot h_j(1-h_j) \cdot x_i$$

- The trickiest part is computing this $\frac{\partial E}{\partial h_j}$ since E comes all the way from the output stage!
  - This is where the real back-propagation comes from.
- For simplicity we'll just show this for an ANN with a single hidden layer, although you should be able to generalize to propagate further back to earlier layers.

Back Propagation Derivation

$$\frac{\partial E}{\partial \beta_{ij}} = \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial h_j}} \cdot h_j\left(1 - h_j\right) \cdot x_i$$

- Intuitively, the error at hidden layer node j gets distributed (via it's weights) to the output nodes.
- Let's use the chain rule to re-write this partial in terms of the outputs o_k

$$\frac{\partial E}{\partial h_j} = \sum_k \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k} \frac{\partial net_k}{\partial h_j}$$

- Recall that from the output stage $\delta_k = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial net_k}$ so we can re-write this as:

$$\frac{\partial E}{\partial h_j} = \sum_k \delta_k \frac{\partial net_k}{\partial h_j}$$

Back Propagation Derivation

$$\frac{\partial E}{\partial \beta_{ij}} = \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial h_j}} \cdot h_j(1-h_j) \cdot x_i$$

- $\frac{\partial E}{\partial h_j} = \sum_k \delta_k \frac{\partial net_k}{\partial h_j}$
- So all we need is $\frac{\partial net_k}{\partial h_j}$
- $\frac{\partial net_k}{\partial h_j} = \frac{\partial}{\partial h_j} h\theta_{:,k} = \theta_{jk}$
- Therefore $\frac{\partial E}{\partial h_j} = \left(\sum_k \delta_k \theta_{jk}\right)$ and finally…

$$\frac{\partial E}{\partial \beta_{ij}} = \left(\sum_k \delta_k \theta_{jk}\right) \cdot h_j(1-h_j) \cdot x_i$$

Back Propagation Derivation

$$\frac{\partial E}{\partial \beta_{ij}} = \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial h_j}} \cdot h_j(1-h_j) \cdot x_i$$

- If we again let $\delta_j = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial net_j} = \left(\sum \delta_k \theta_{jk}\right) \cdot h_j(1-h_j)$ then we can easily write the update rule as:

$$\beta_{ij} \mathrel{+}= \eta \delta_j x_i$$

- And if you wanted a network with multiple hidden layers (a deep network) then you could propagate these δ_j backwards.

Back Propagation

- Putting it all together…
- For a standard ANN with a single hidden layer, our update rules are then:
  - Output layer
    - $\delta_k = (y_k - o_k)$
    - $\theta_{jk} \mathrel{+}= \eta \delta_k h_j$
  - Inner layer
    - $\delta_j = \left(\sum_k \delta_k \theta_{jk}\right) h_j \left(1 - h_j\right)$
    - $\beta_{ij} \mathrel{+}= \eta \delta_j x_i$

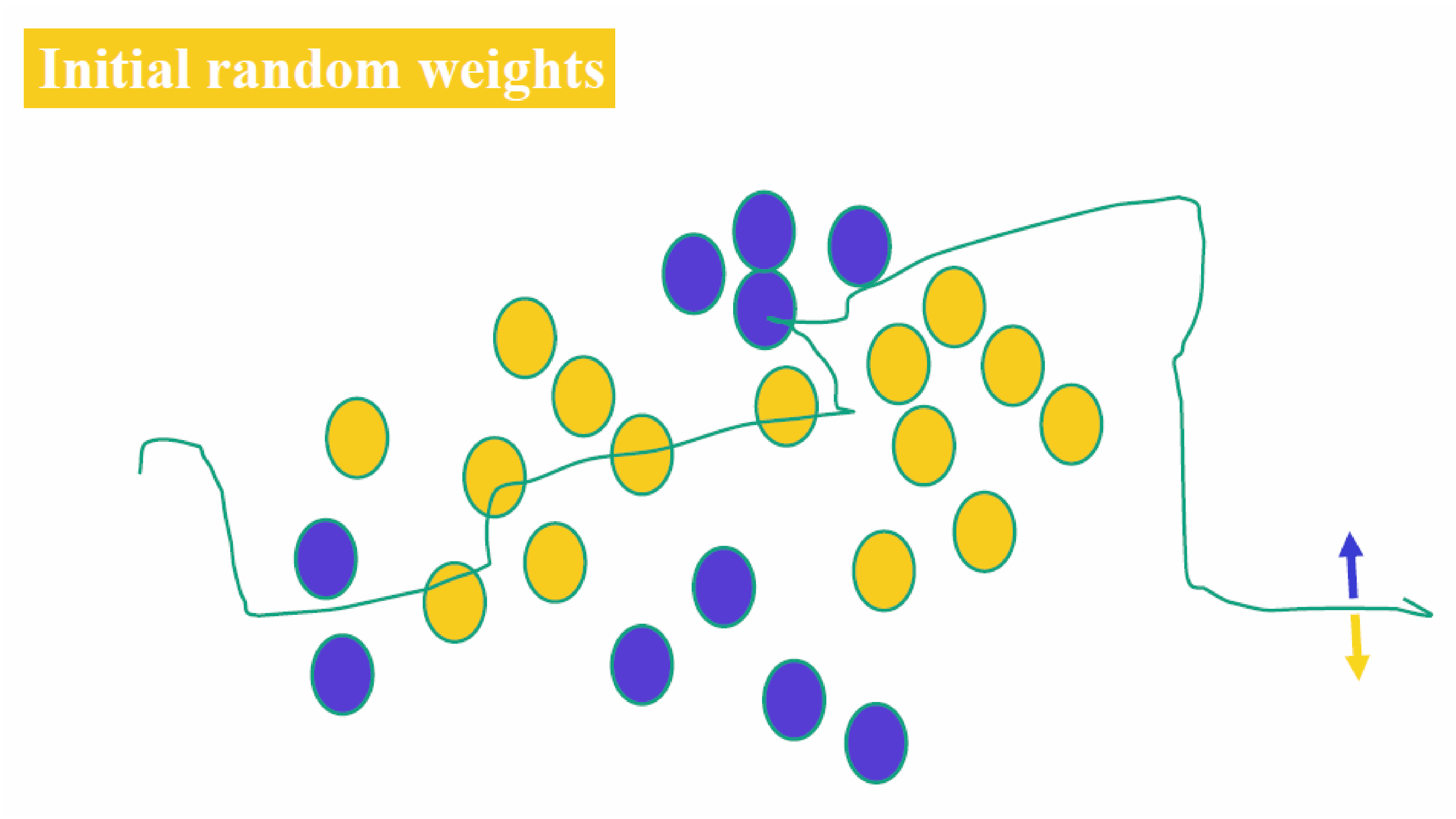
Batch Processing

- Instead of doing one observation at a time, we might want to do batch (or mini-batch) gradient descent.
- The rules are pretty much the same but now o, h are matrices
  - Output layer
    - Compute the loss at output node k due to each training sample s:
$$\delta_{s,k} = \left(Y_{s,k} - o_{s,k}\right)$$
    - Update the parameter by an average of the weighted losses:
$$\theta_{jk} = \theta_{ji} + \frac{\eta}{N}\delta_{:,k}^{T} h_{:,j}$$
  - Inner layer
    - Again compute the loss at hidden layer node j due to each training sample s:
$$\delta_{s,j} = \left(\sum_{k \in outputs} \left(\theta_{jk}\delta_{s,k}\right)\right) h_{s,j}\left(1 - h_{s,j}\right) = \left(\theta_{j,:}\delta\right)^{T} h_{s,j}\left(1 - h_{s,j}\right)$$
    - And update the parameter by an average of the weighted losses:
$$\beta_{ij} = \beta_{ij} + \frac{\eta}{N}\delta_{:,j}^{T} X_{:,i}$$

More on Backpropagation

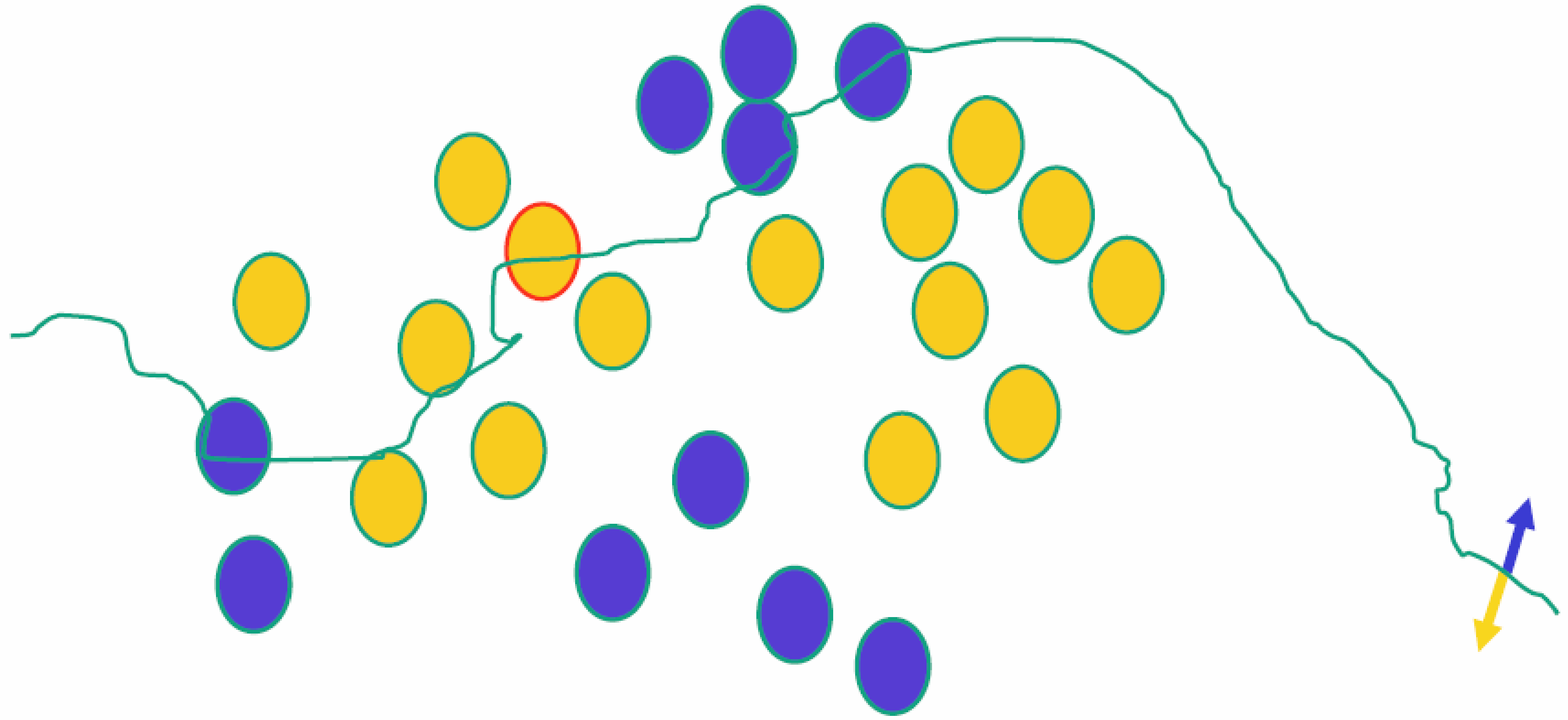
- Training can take thousands of iterations
  - Slow! ☹
- But using network after training is very fast! ☺

Visualizing the Process…



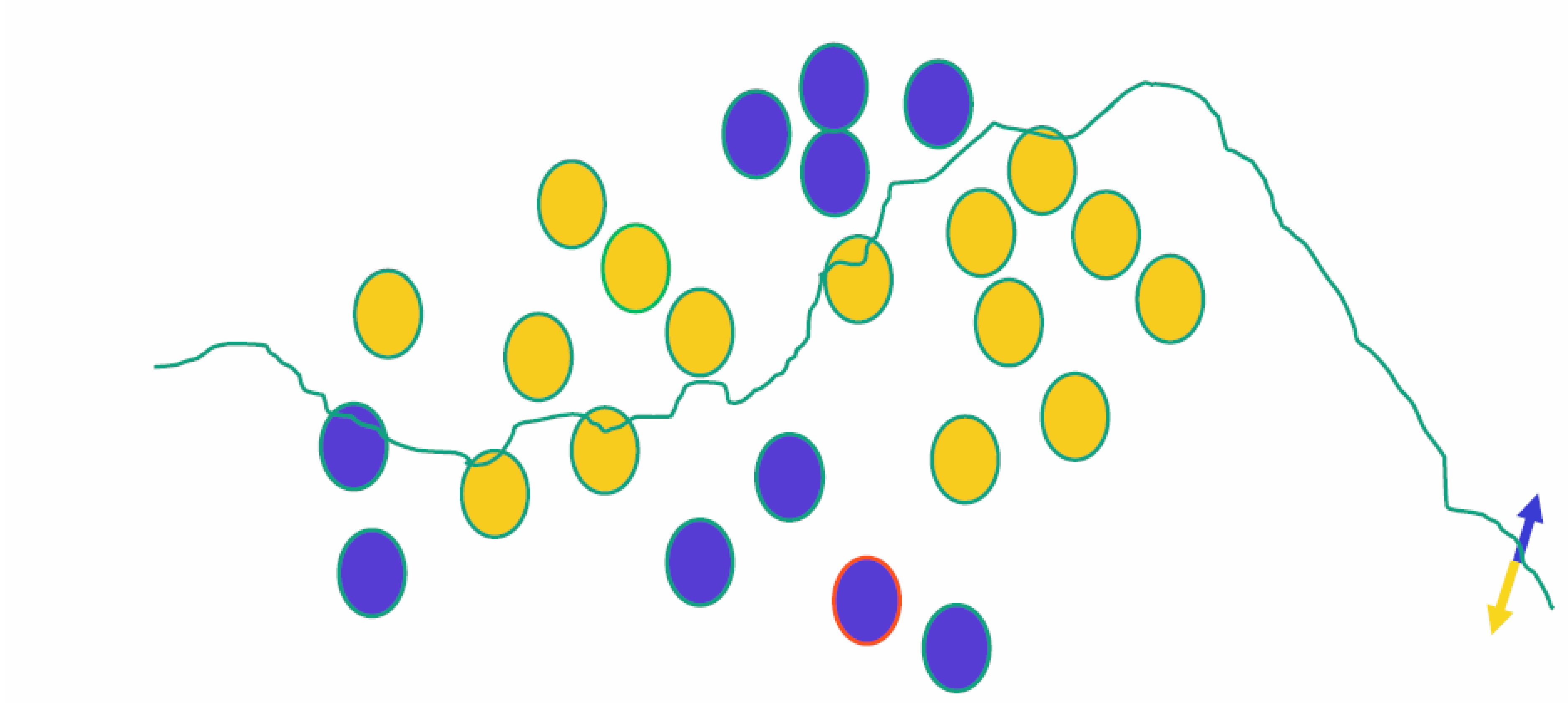
Visualizing the Process…

Present a training instance / adjust the weights



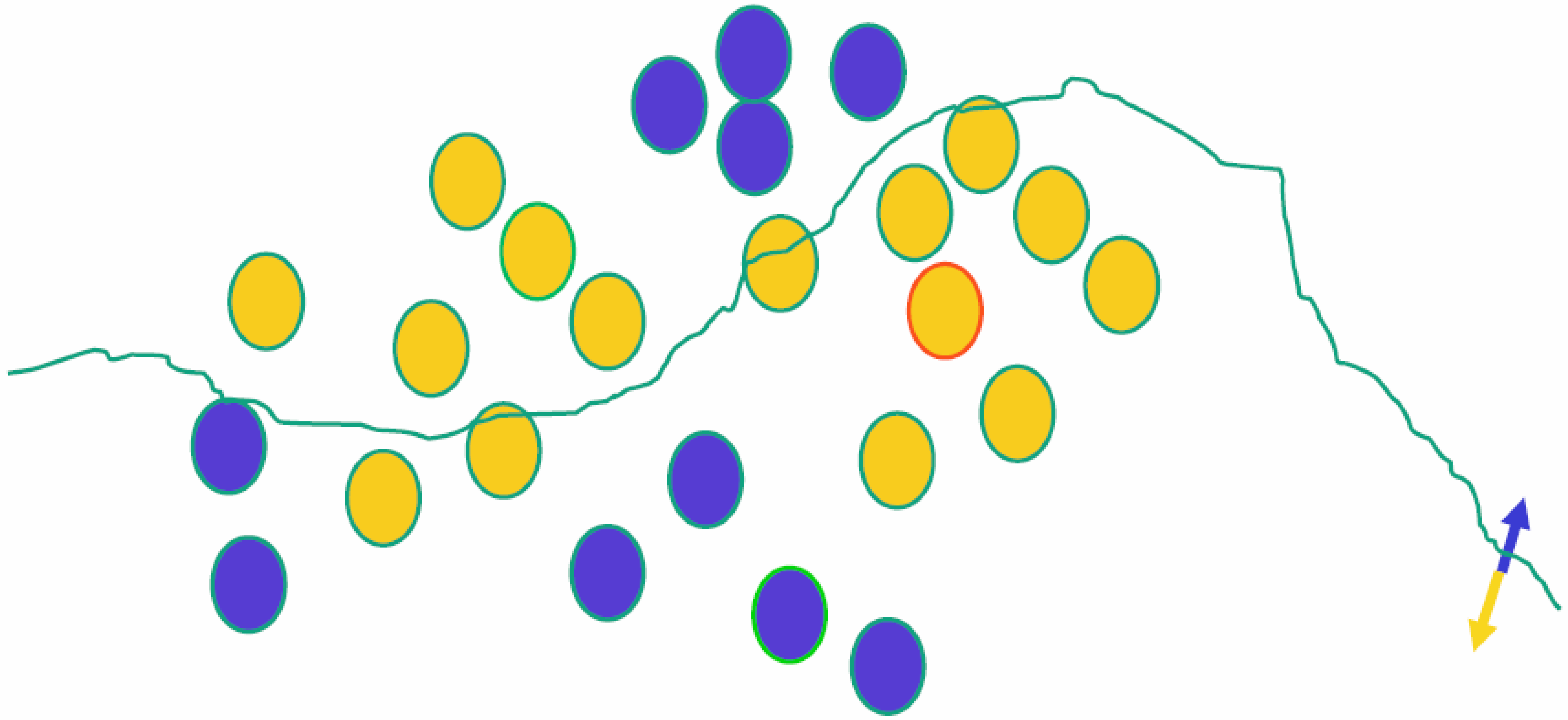
Visualizing the Process…

Present a training instance / adjust the weights



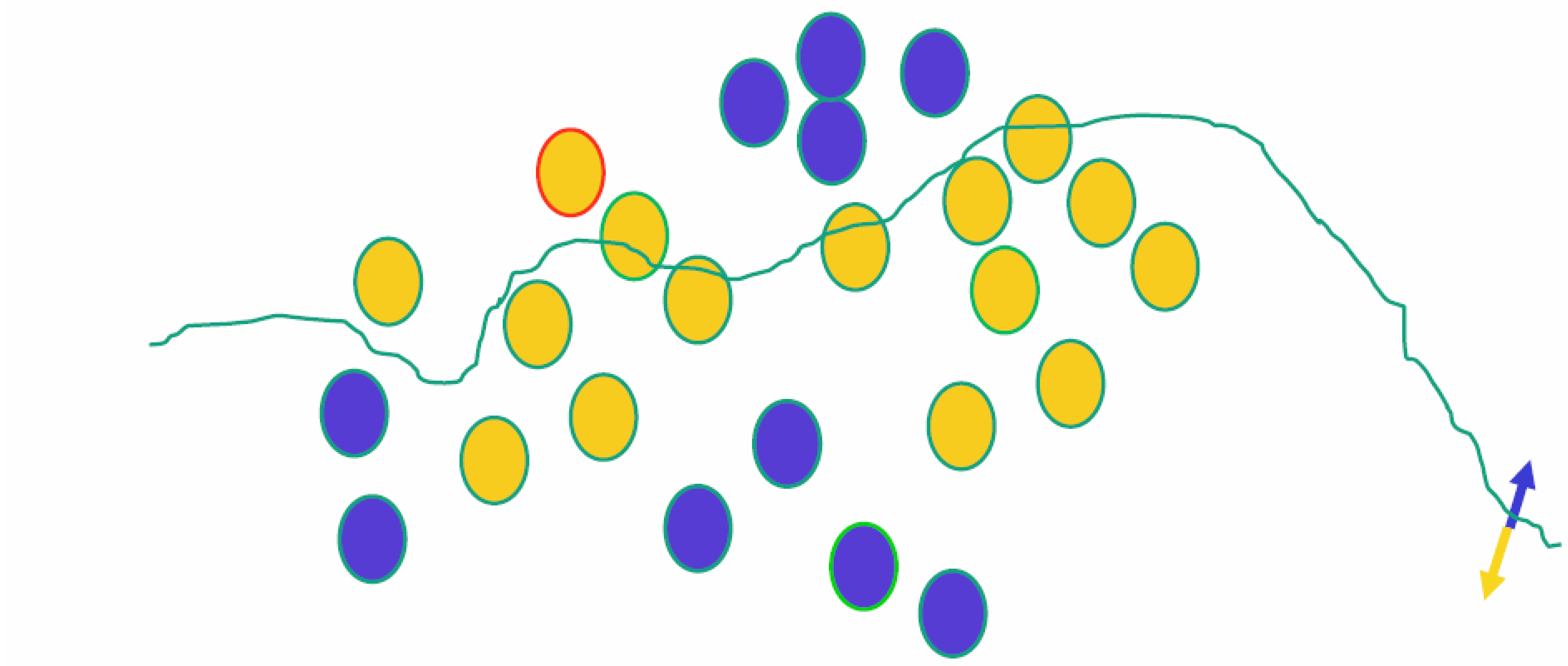
Visualizing the Process…

Present a training instance / adjust the weights

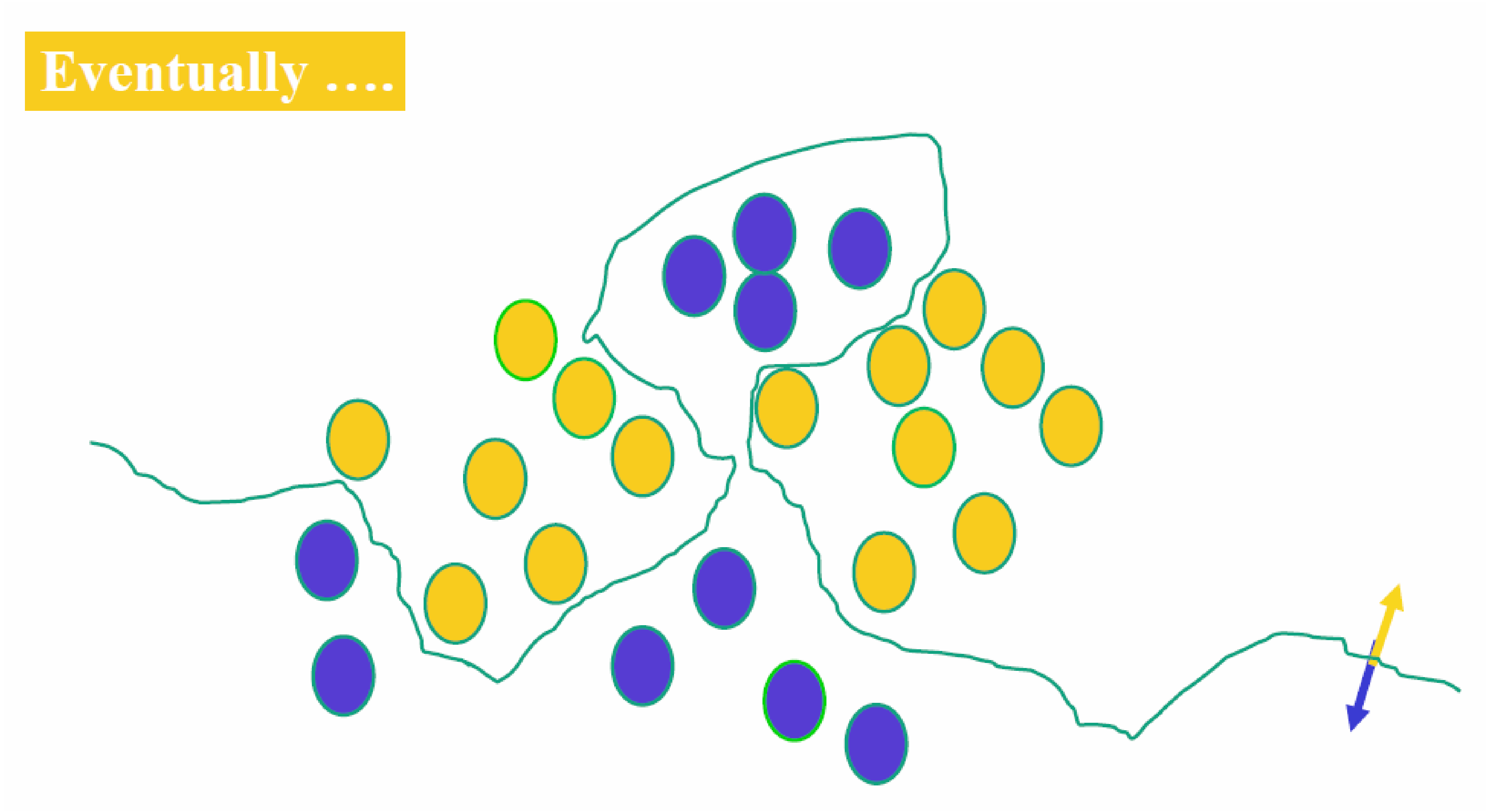


Visualizing the Process…

Present a training instance / adjust the weights

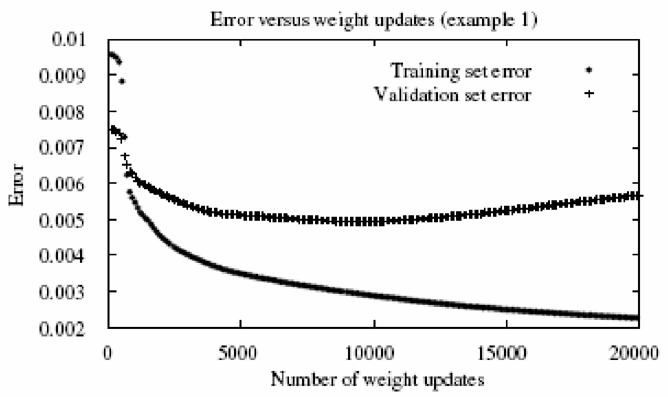


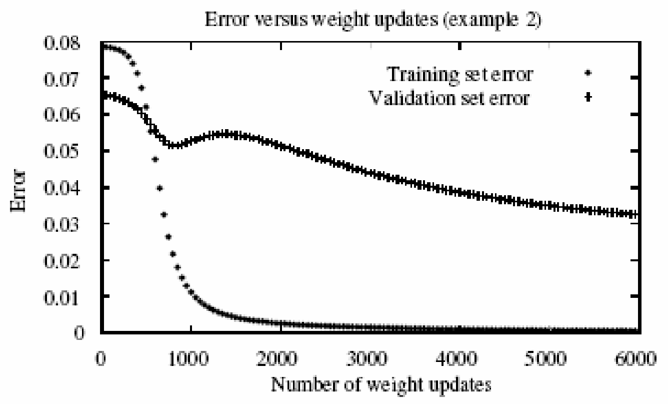
Visualizing the Process…



Overfitting

- Some things that can result in overfitting are:
  - Too many iterations
  - Too many hidden nodes

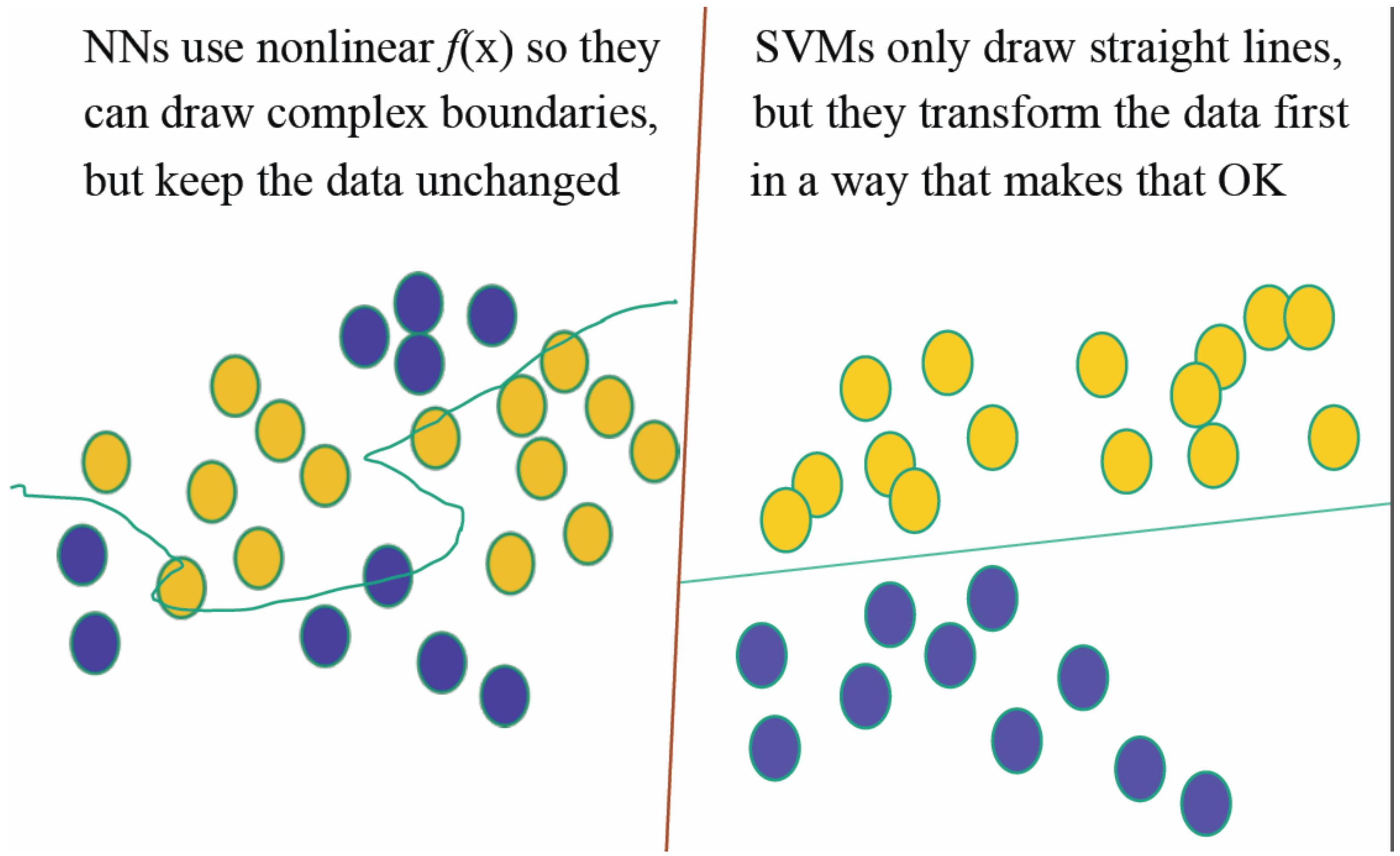




Overfitting

- How can we use a validation set to avoid overfitting algorithms that iteratively update parameters?
  - For determining number of iterations?
  - For determining number of hidden layer nodes?
- What if we don't have that much data?

Additional Remarks



NNs use nonlinear *f*(x) so they can draw complex boundaries, but keep the data unchanged

SVMs only draw straight lines, but they transform the data first in a way that makes that OK

Final Remarks

- So when might it be good to use ANNs
    - Input is high-dimensional
    - Input is continuous (though it can handle discrete too)
    - Output is a vector of values
    - Human readability of result is unimportant

Final Observations

- Let's think about this algorithm
  - Supervised or non?
  - Classification or regression?
  - Model-based or instance-based?
    - When it comes time to test/use, are we using the original data?
  - Linear vs Non-Linear?
  - Can this work on categorical data?
  - Can this work on continuous valued data?
  - Training Complexity?
  - Testing Complexity?
  - How to deal with overfitting?
  - Directly handles multi-class?