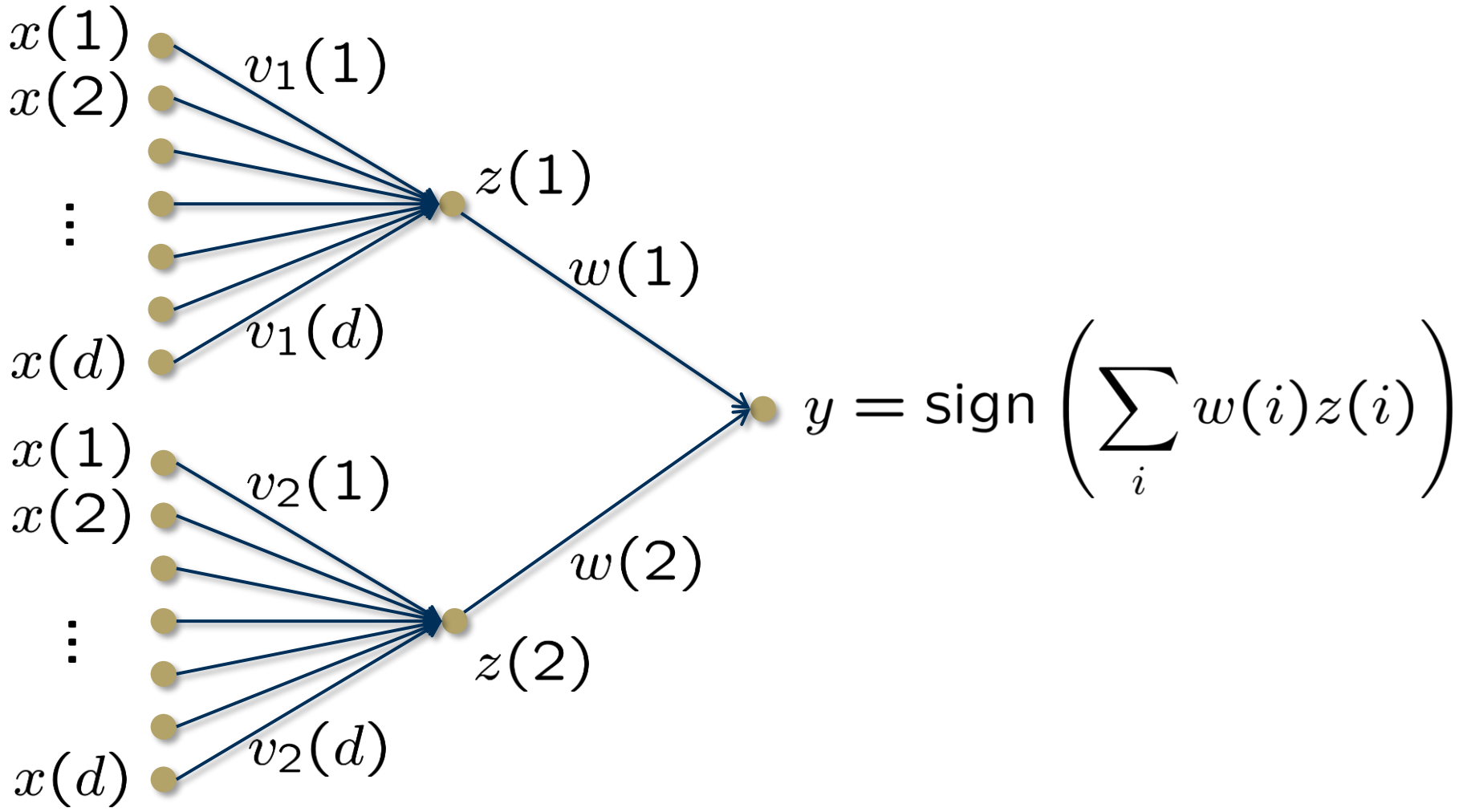
Ensemble methods in machine learning

- Bootstrap aggregating (bagging)
  - train an ensemble of models based on randomly resampled versions of the training set, then take a majority vote
- Boosting
  - iteratively build an ensemble by training each new model to emphasize the parts of the training set that the previous model struggled with
- Stacking
  - train a learning algorithm to combine the predictions of other learning algorithms

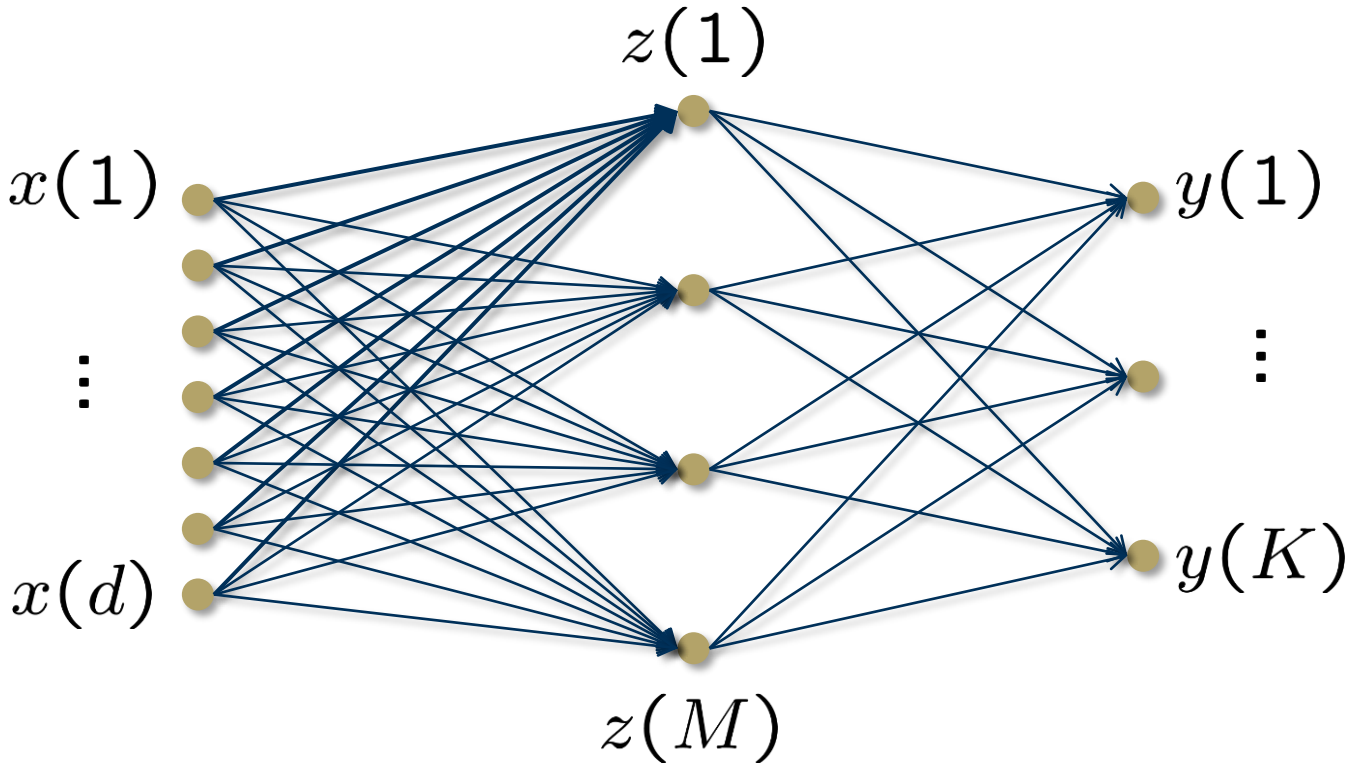
Example

What if you used the output of two linear classifiers as the input to another linear classifier?



Neural networks

This is a particular example of a ***multi-layer neural network***



input layer

hidden layer

output layer

Neural networks

Formally, the output of this network can be expressed via

$$z(m) = g(\mathbf{v}_m^T \mathbf{x} + b_m) \quad m = 1, \ldots, M$$

$$y(k) = h(\mathbf{w}_k^T \mathbf{z} + c_k) \quad k = 1, \ldots, K$$

where g, h are fixed ***activation functions*** and $\mathbf{v}_m, b_m, \mathbf{w}_k, c_k$ are parameters to be learned from the data

Example

The previous example fits this model with $K = 1$ and

$$h(t) = g(t) = \text{sign}(t)$$

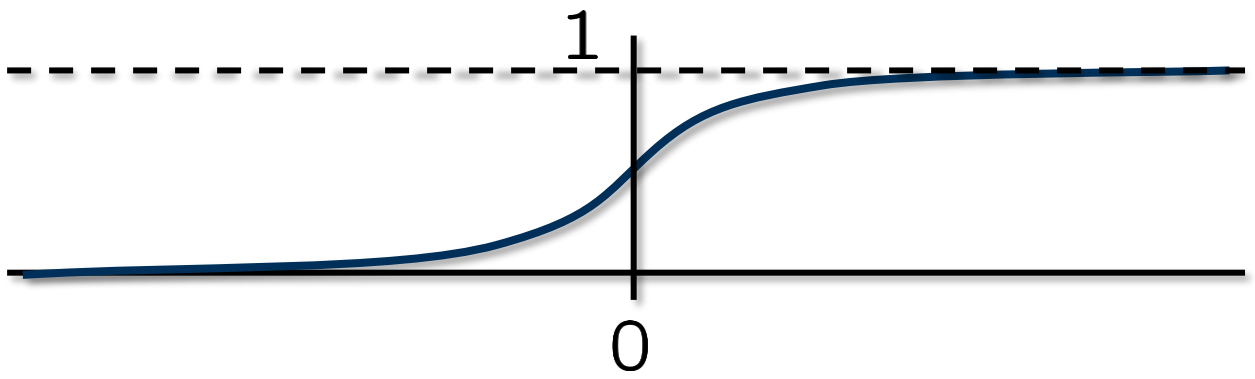
Typical neural networks

In general, learning the parameters for a neural network can be quite difficult

To make life easier, it is nice to choose g, h to be differentiable

A historically common choice for g is the logistic/sigmoid function

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



The choice of h depends somewhat on the application

Typical neural networks

The choice of h depends somewhat on the application

Regression ($y \in \mathbb{R}$)

$$h(t) = t$$

Binary classification ($K = 1, y \in \{-1, +1\}$)

$$h(t) = \frac{1 - e^{-t}}{1 + e^{-t}}$$

Multiclass classification ($\mathbf{y} = [0 \cdots 010 \cdots 0]^T$)

$$h(t_k) = \frac{e^{t_k}}{\sum_{j=1}^{K} e^{t_j}} \qquad t_k = \mathbf{w}_k^T \mathbf{z} + c_k$$

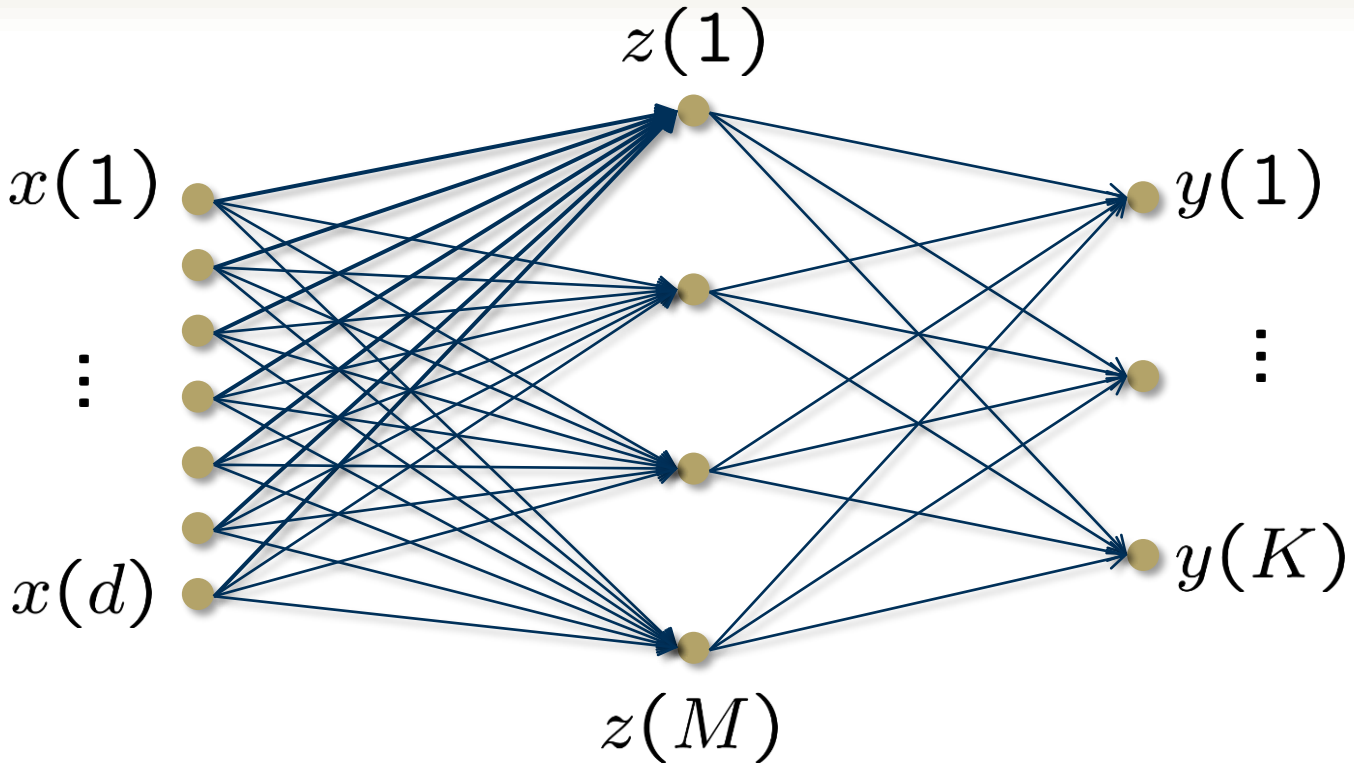
Remarks

- Like SVMs and logistic regression, neural networks ultimately apply a linear classifier to a set of features
- Like SVMs, those features are ***nonlinear***
- Unlike SVMs, those nonlinear features are ***learned***
- Unfortunately, training involves ***nonconvex*** optimization
- Originally conceived as models for the brain
  - nodes are neurons
  - edges are synapses
  - if g is a step function, this represents a neuron "firing" when the total incoming signal exceeds a certain threshold
- Potentially lots of parameters
  - possible red flag for generalization

Training neural networks

Given training data $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $\mathbf{y}_i \in \mathbb{R}^K$, we would like to estimate the parameters $\mathbf{v}_m, b_m, \mathbf{w}_k, c_k$

Training neural networks



$$z(m) = g(\mathbf{v}_m^T \mathbf{x} + b_m) \quad m = 1, \ldots, M$$

$$y(k) = h(\mathbf{w}_k^T \mathbf{z} + c_k) \quad k = 1, \ldots, K$$

Simplifying the notation

Note that we can always augment a “1” to our input data (increasing the dimension to $d+1$) and constrain the $\mathbf{v}_m$ to also ensure that $z(1) = 1$

If we do this, we may omit b_m and c_k to arrive at the simpler formulation

$$z(m) = g(\mathbf{v}_m^T \mathbf{x})$$
$$y(k) = h(\mathbf{w}_k^T \mathbf{z})$$

Letting $\mathbf{V}$ denote the matrix having rows $\mathbf{v}_m^T$ and $\mathbf{W}$ the matrix having rows $\mathbf{w}_k^T$, we can also write this as

$$\mathbf{z} = g(\mathbf{V}\mathbf{x}) \qquad \mathbf{y} = h(\mathbf{W}\mathbf{z})$$
$$\mathbf{y} = f(\mathbf{x}) = h(\mathbf{W}g(\mathbf{V}\mathbf{x}))$$

Training neural networks

Given training data $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $\mathbf{y}_i \in \mathbb{R}^K$, we would like to estimate the parameters $\mathbf{V}, \mathbf{W}$

For simplicity, let $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$

To emphasize the dependence of our network on the parameters $\boldsymbol{\theta}$, we will write the output of the network when given input $\mathbf{x}$ as $f(\mathbf{x}; \boldsymbol{\theta})$

We would like to choose $\boldsymbol{\theta}$ to ensure that $f(\mathbf{x}; \boldsymbol{\theta}) \approx \mathbf{y}$

We can quantify this by choosing a ***loss function*** which we will seek to minimize by picking $\boldsymbol{\theta}$ appropriately

Loss functions

Regression or ***binary classification***

For $K = 1$

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i; \boldsymbol{\theta}))^2$$

or for vector-valued regression problems

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \|\mathbf{y}_i - f(\mathbf{x}_i; \boldsymbol{\theta})\|_2^2$$

Loss functions

General classification

In the case of classification where $K > 1$, we can define the $\mathbf{y}_i$ as indicator vectors in $\mathbb{R}^K$ (e.g., $\mathbf{y}_i = [0 \cdots 010 \cdots 0]^T$)

In this case a natural loss function is

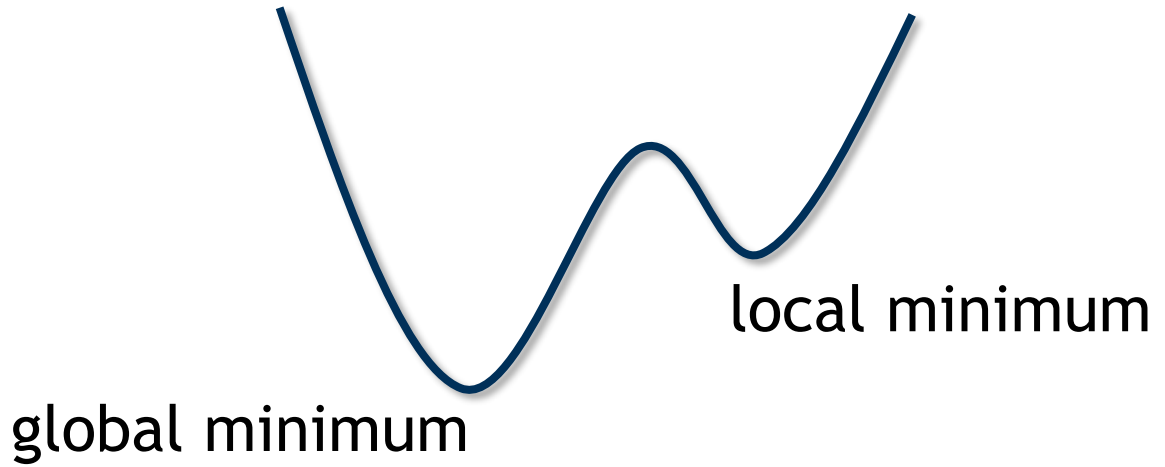
$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \mathbf{y}_i^T \log f(\mathbf{x}_i; \boldsymbol{\theta})$$

This is called the ***cross entropy***

If we interpret the outputs of our neural network $f(\mathbf{x}_i; \boldsymbol{\theta})$ as class conditional probabilities, this is computing the negative log-likelihood of $\boldsymbol{\theta}$ given the training data, and is hence a natural quantity to minimize

Nonconvex optimization

Because of the complex interactions between the parameters in θ, these objective functions do not lead to convex optimization problems



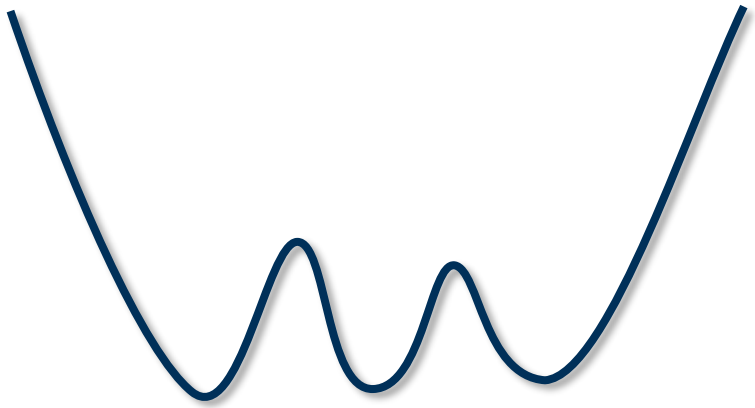
A ***local minimum*** is not necessarily a ***global minimum***

The best we can hope for is to try to find a local minimum, and hope that this gives us good performance in practice

Aside…

This is currently a very active area of research, but there is some preliminary evidence that the picture I just showed you is not really an accurate depiction of the nonconvexity that typically arises in practice

It may be the case that the picture really looks more like



Perhaps most/all local minima are equally good!
(highly speculative…)

Gradient descent

Recall that in gradient descent we simply iteratively *"roll downhill"*

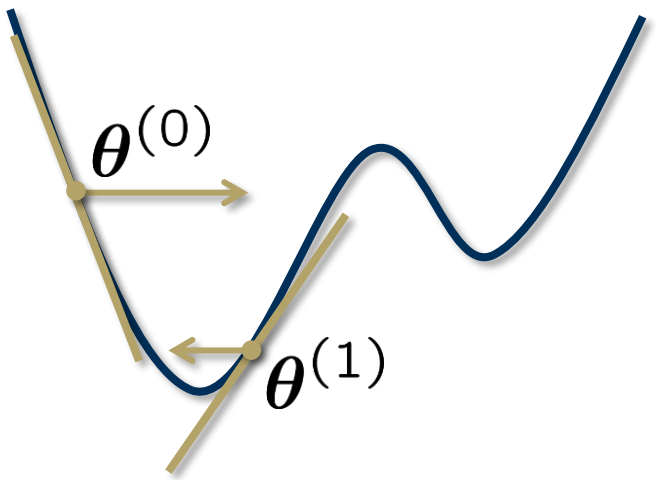
From $\boldsymbol{\theta}^{(0)}$, step in the direction of the negative gradient

$$\boldsymbol{\theta}^{(1)} = \boldsymbol{\theta}^{(0)} - \alpha \nabla L(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(0)}}$$

$$\boldsymbol{\theta}^{(2)} = \boldsymbol{\theta}^{(1)} - \alpha \nabla L(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(1)}}$$

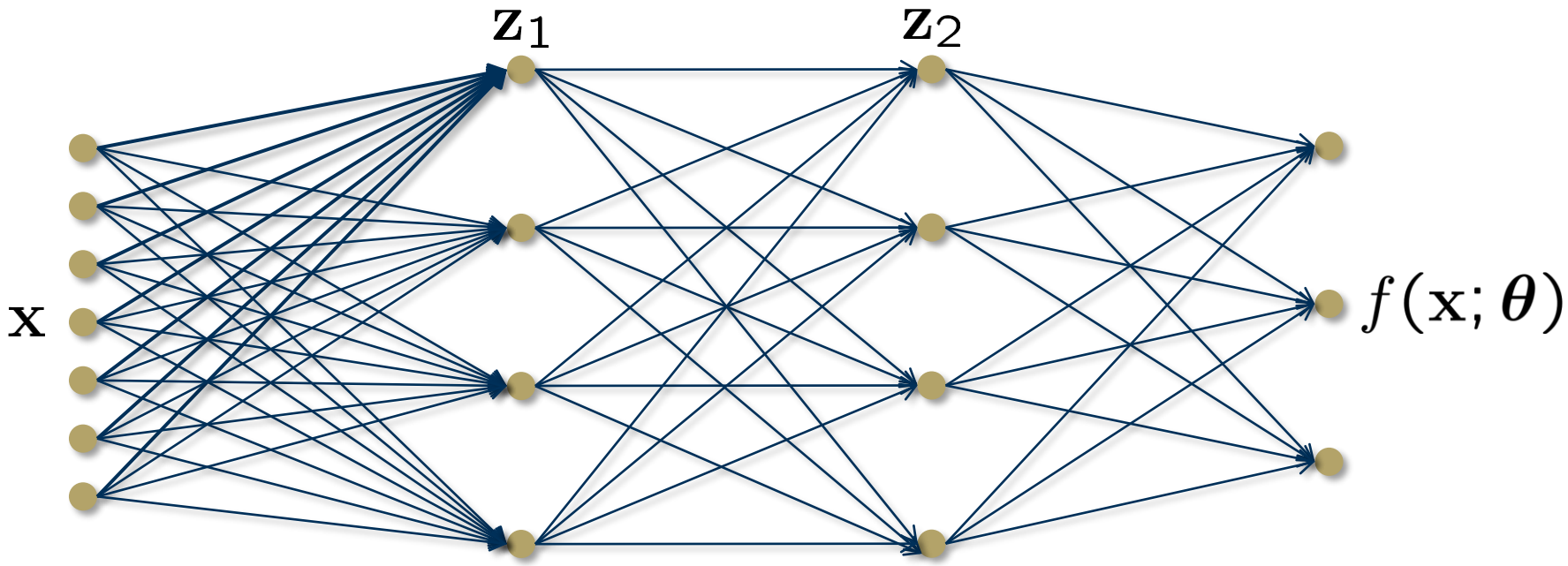
$$\vdots$$

α : "step size"



Multi-layer neural networks

This framework can easily be extended to networks with multiple hidden layers




WE JUST MIGHT NEED
TO GO DEEPER
memegenerator.net

Deep neural networks

To simplify things, we will use a slightly altered notation

$$\mathbf{x}^{(0)} \xrightarrow{\mathbf{W}^{(1)}} \mathbf{x}^{(1)} \xrightarrow{\mathbf{W}^{(2)}} \cdots \xrightarrow{\mathbf{W}^{(L)}} \mathbf{x}^{(L)} = f(\mathbf{x})$$

Here $\mathbf{W}^{(\ell)}$ is a $d^{(\ell-1)} \times d^{(\ell)}$ matrix and

$$\mathbf{x}^{(\ell)} = g^{(\ell)}\left((\mathbf{W}^{(\ell)})^T \mathbf{x}^{(\ell-1)}\right)$$

$$x_j^{(\ell)} = g^{(\ell)}\left(\underbrace{\sum_{i=1}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)}}_{s_j^{(\ell)}}\right)$$

Computing the gradient

Given a loss function L, to implement gradient descent we need to compute ∇L

i.e., we need to compute $\frac{\partial L}{\partial w_{ij}^{(\ell)}}$ for every single (i, j, ℓ)

We could do this independently for each (i, j, ℓ), but this gets very slow as the network gets large

Backpropagation is a much more efficient strategy for computing the gradient

The chain rule

Recall that $s_j^{(\ell)} = \sum_{i=1}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)}$

Note that $\frac{\partial L}{\partial w_{ij}^{(\ell)}} = \frac{\partial L}{\partial s_j^{(\ell)}} \times \frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}}$

$$\frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} = x_i^{(\ell-1)}$$

All we need is $\frac{\partial L}{\partial s_j^{(\ell)}}$, which we will denote by $\delta_j^{(\ell)}$

Starting at the final layer

Let's look at a concrete example

For our loss function, take $L = (f(\mathbf{x}_0) - y_0)^2$ for a particular input $\mathbf{x}_0$ and desired scalar output y_0

Assume for simplicity that $g^{(1)} = \cdots g^{(L)} = g$

$$\delta_1^{(L)} = \frac{\partial L}{\partial s_1^{(L)}}$$

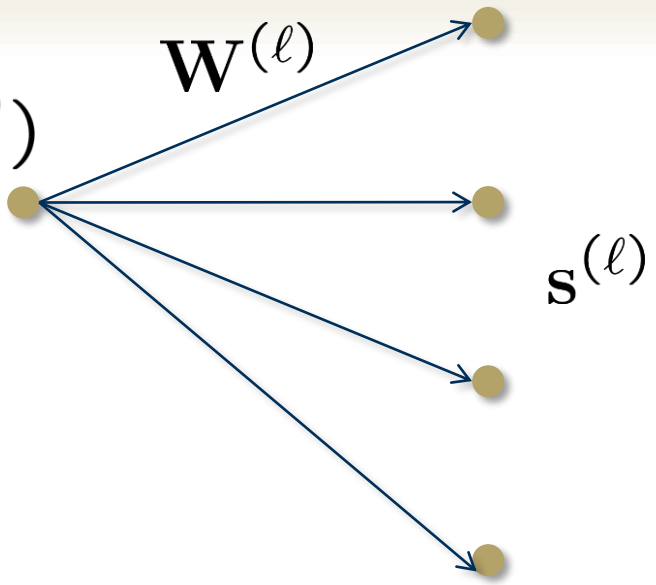
$$= \frac{\partial}{\partial s_1^{(L)}} \left(x_1^{(L)} - y_0 \right)^2 \qquad x_1^{(L)} = g\left(s_1^{(L)} \right)$$

$$= 2g'\left(s_1^{(L)} \right) \left(x_1^{(L)} - y_0 \right)$$

Backpropagation

Now consider

$$x_i^{(\ell-1)} = g(s_i^{(\ell-1)})$$



$$\delta_i^{(\ell-1)} = \frac{\partial L}{\partial s_i^{(\ell-1)}}$$

$$= \sum_{j=1}^{d^{(\ell)}} \frac{\partial L}{\partial s_j^{(\ell)}} \times \frac{\partial s_j^{(\ell)}}{\partial x_i^{(\ell-1)}} \times \frac{\partial x_i^{(\ell-1)}}{\partial s_i^{(\ell-1)}}$$

$$= \sum_{j=1}^{d^{(\ell)}} \delta_j^{(\ell)} \times w_{ij}^{(\ell)} \times g'\left(s_i^{(\ell-1)}\right)$$

$$= g'\left(s_i^{(\ell-1)}\right) \sum_{j=1}^{d^{(\ell)}} w_{ij}^{(\ell)} \delta_j^{(\ell)}$$

Backpropagation algorithm

Initialize all weights $w_{ij}^{(\ell)}$ ***at random***

Until convergence (or deadline):

1. Pick an example input-output $(\mathbf{x}_0, y_0)$
2. Forward: Pass $\mathbf{x}_0$ through the network to compute all $x_j^{(\ell)}$
3. Backward: Compute all $\delta_j^{(\ell)}$
4. Update the weights: $w_{ij}^{(\ell)} \leftarrow w_{ij}^{(\ell-1)} - \alpha x_i^{(\ell-1)} \delta_j^{(\ell)}$
5. Iterate

Return final weights

Convergence speed

In backpropagation, each hidden unit passes and receives information to and from only those units to which it is connected

Much faster way to compute the gradient than a naïve approach

Lends itself naturally to ***parallel*** implementations

Gradient descent can still be pretty slow to converge…

Initialization and various “tricks” are very important!