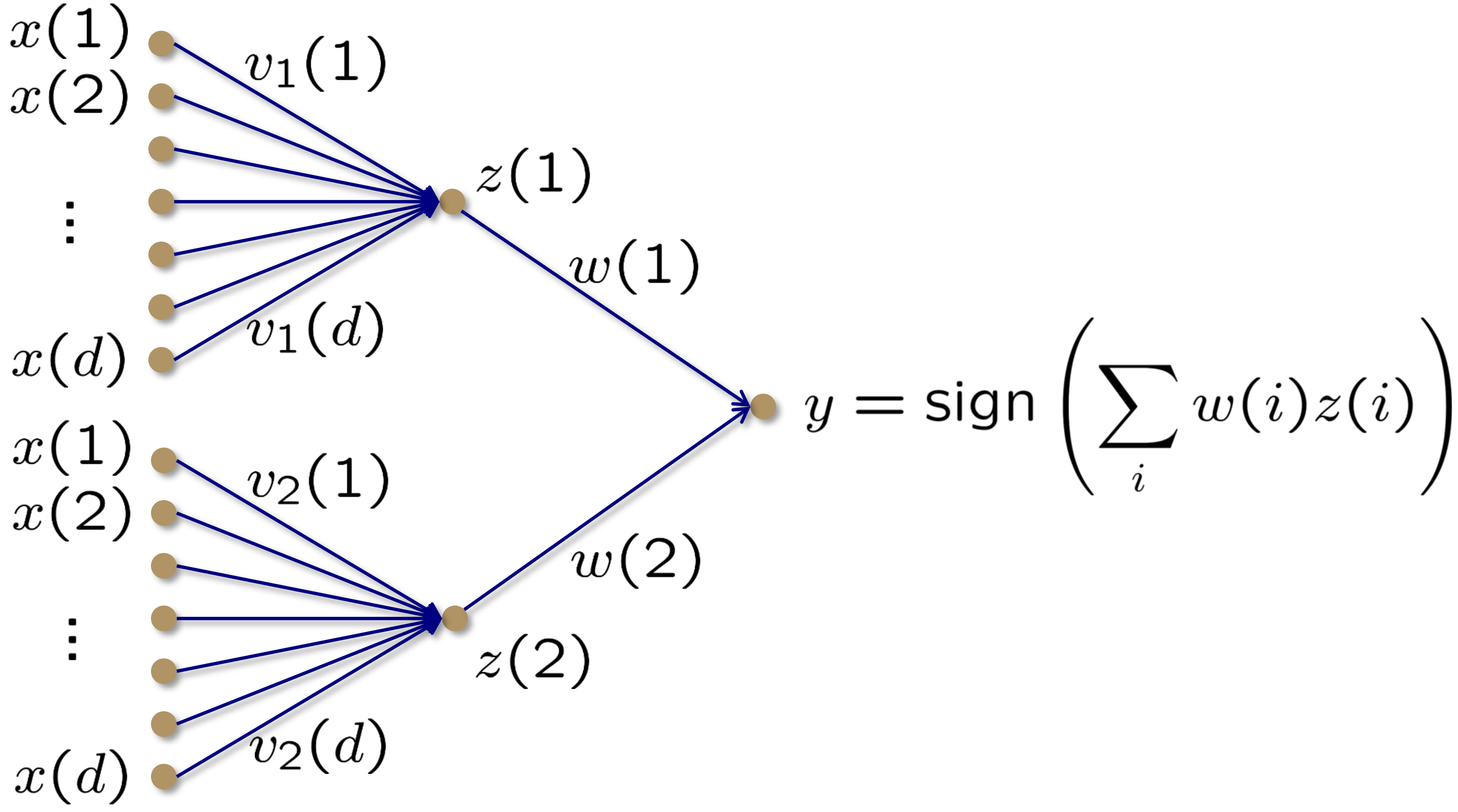
Ensemble methods in machine learning

- 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

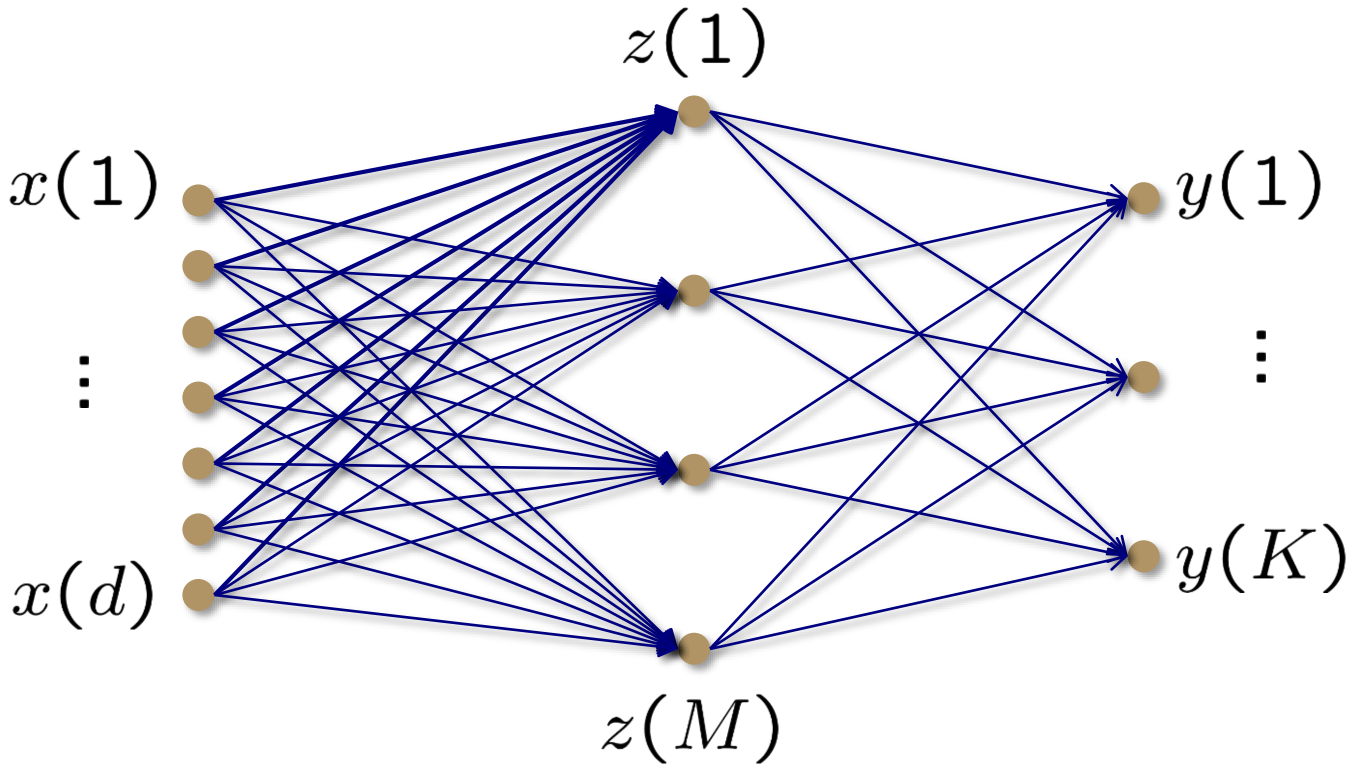
Example

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 ***neural network***



input layer

hidden layer

output layer

Neural networks

Formally, a neural network is expressed mathematically via

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

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

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

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

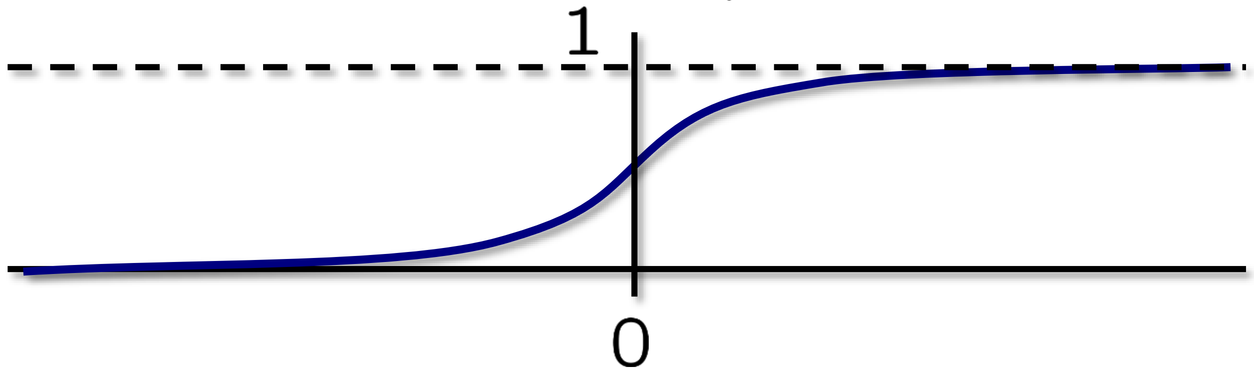
Typical neural networks

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

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

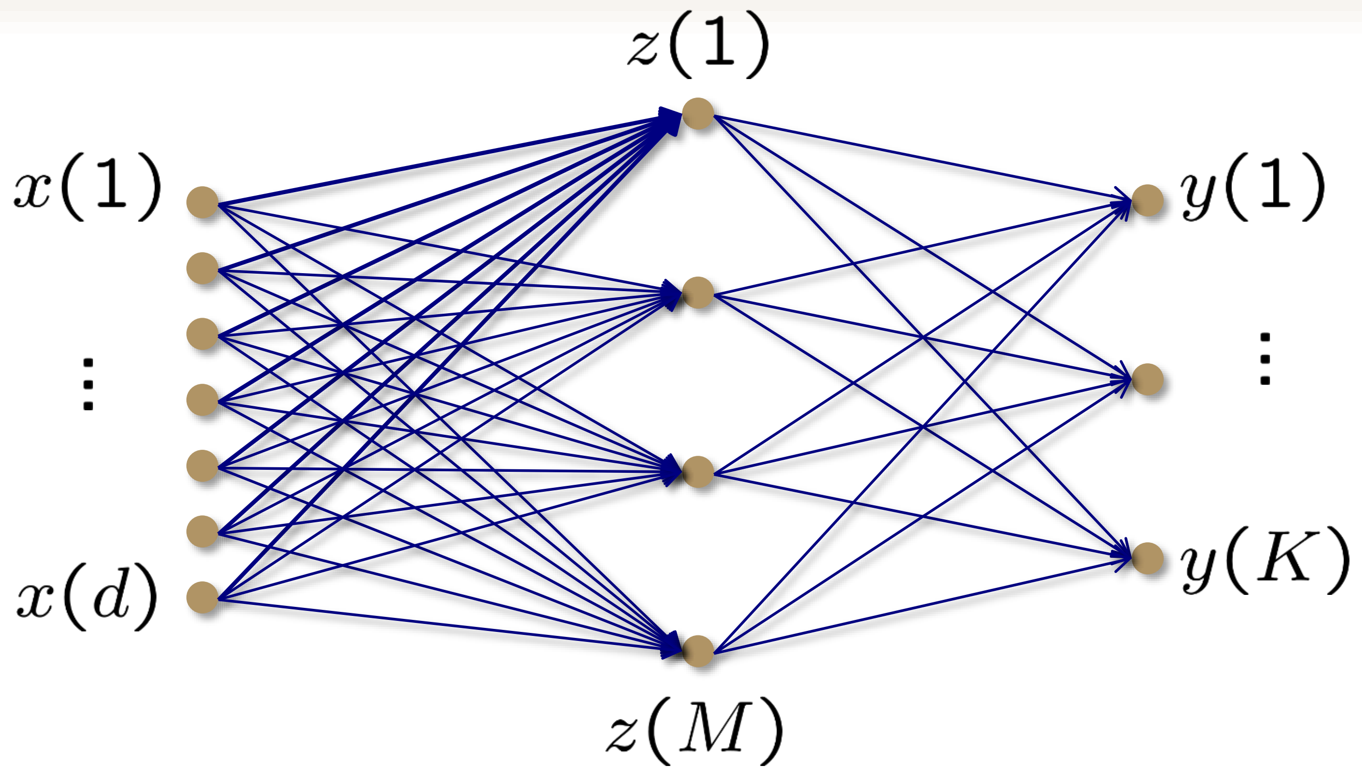
Remarks

- Like SVMs, neural networks fit a linear model in a nonlinear feature space
- 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
  - note that our choices for g, h are not exactly step functions, but smooth approximations – this also means our output is continuous (must be quantized in the case of classification)

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

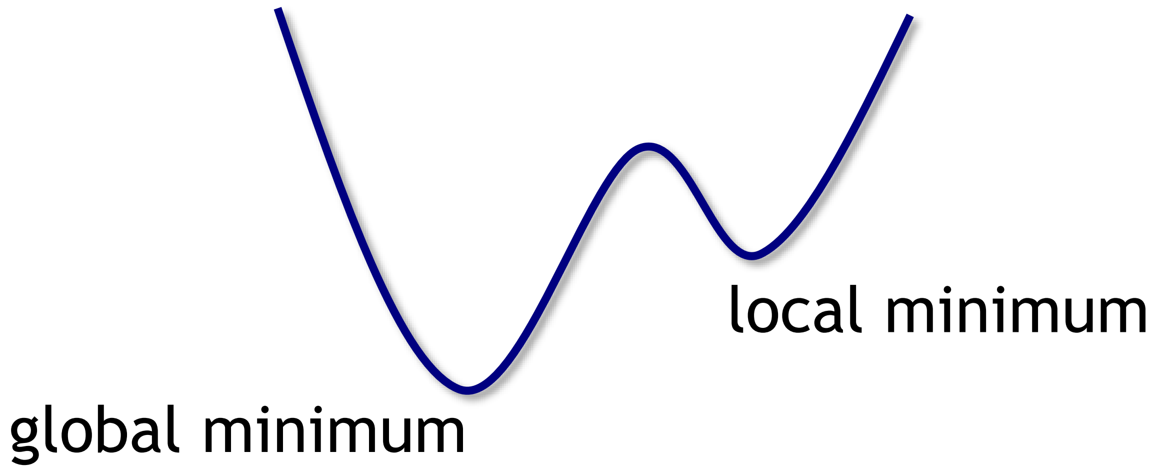
$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \mathbf{y}_i^T \log f(\mathbf{x}_i; \boldsymbol{\theta})$$

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 $\boldsymbol{\theta}$, these objective functions do not lead to convex optimization problems



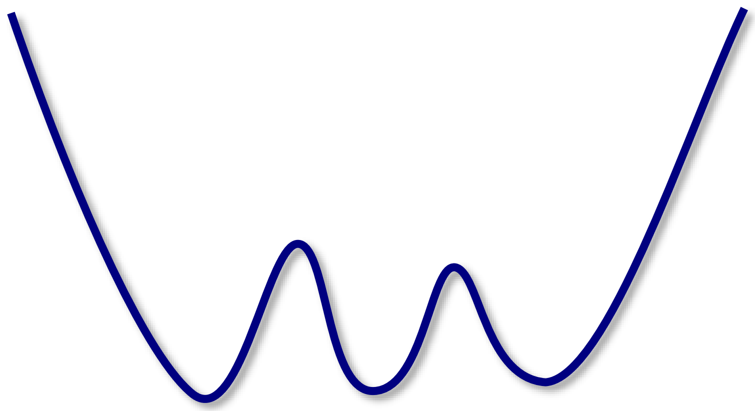
A ***local minimum*** is not necessarily a ***global minimum***

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

A simple way to try to find the minimum of our objective function is to iteratively *"roll downhill"*

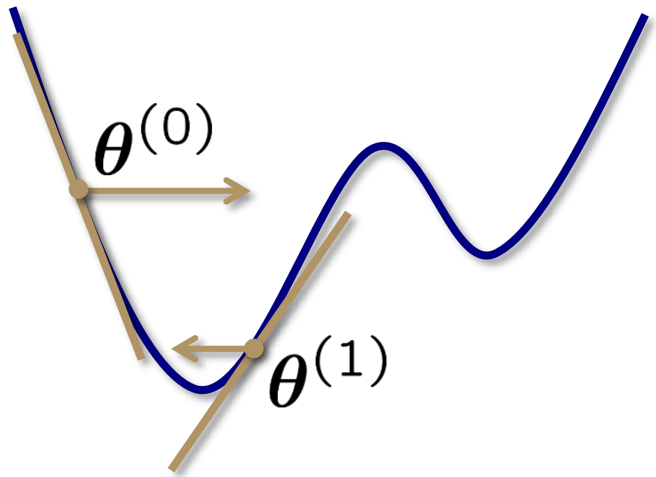
From $\boldsymbol{\theta}^{(0)}$, step in the direction of the negative gradient

$$\boldsymbol{\theta}^{(1)} = \boldsymbol{\theta}^{(0)} - \alpha_0 \nabla L(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(0)}}$$

α_0 : "step size"

$$\boldsymbol{\theta}^{(2)} = \boldsymbol{\theta}^{(1)} - \alpha_1 \nabla L(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(1)}}$$

$$\vdots$$



Squared error loss

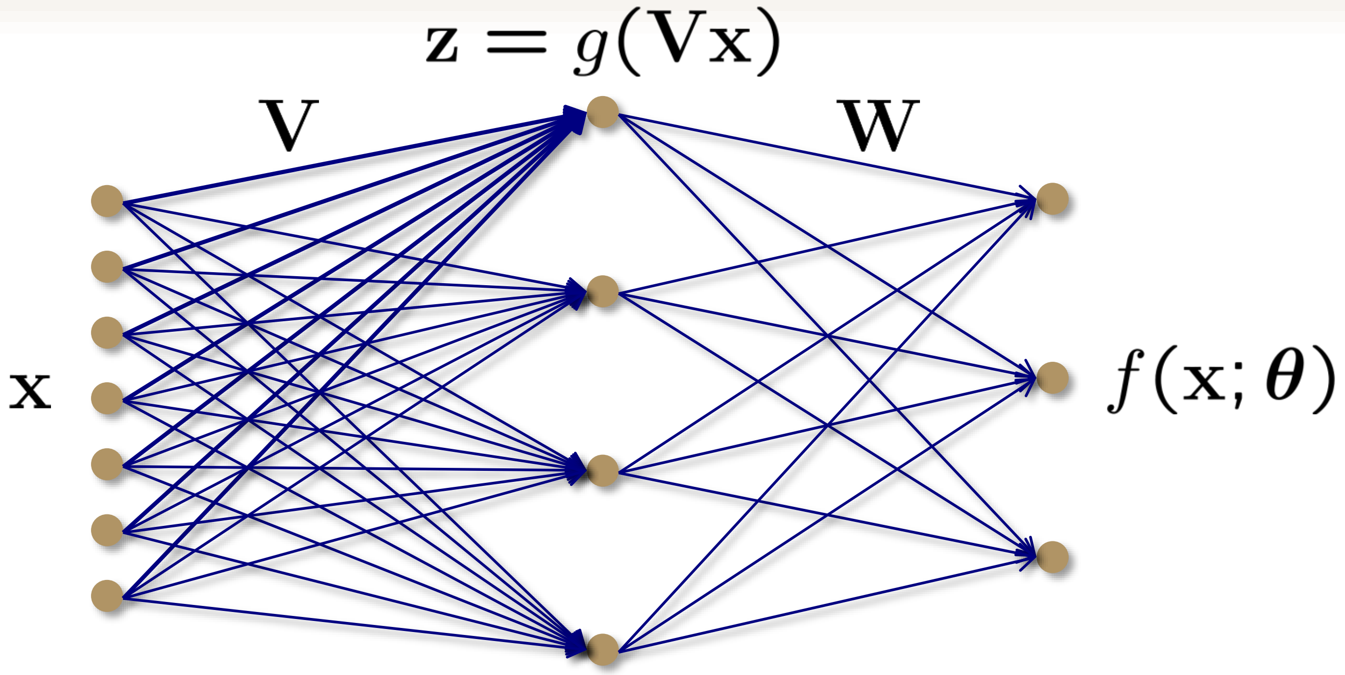
Today, we will focus on how to train to minimize the squared error loss

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

$$L_i = \|\mathbf{y}_i - f(\mathbf{x}_i; \boldsymbol{\theta})\|_2^2$$

Our task is to compute $\nabla L(\boldsymbol{\theta})$ for some given $\boldsymbol{\theta}$

Back to the picture


$\mathbf{z} = g(\mathbf{V}\mathbf{x})$
$\mathbf{V}$
$\mathbf{W}$
$\mathbf{x}$
$f(\mathbf{x};\boldsymbol{\theta})$

$$\begin{aligned} f(\mathbf{x};\boldsymbol{\theta}) &= h(\mathbf{W}\mathbf{z}) \\ &= h(\mathbf{W}g(\mathbf{V}\mathbf{x})) \end{aligned}$$

Deriving the gradient: Second layer

Set $\mathbf{z}_i = g(\mathbf{V}\mathbf{x}_i)$

$$\frac{\partial L_i}{\partial W(k,m)} = \frac{\partial}{\partial W(k,m)} \|\mathbf{y}_i - h(\mathbf{W}\mathbf{z}_i)\|_2^2$$

$$= \frac{\partial}{\partial W(k,m)} (y_i(k) - h(\mathbf{w}_k^T \mathbf{z}_i))^2$$

$$= \underbrace{-2\underbrace{(y_i(k) - h(\mathbf{w}_k^T \mathbf{z}_i))}_{\text{prediction error}} h'(\mathbf{w}_k^T \mathbf{z}_i)}_{\text{weighted prediction error} := \delta_i(k)} z_i(m)$$

$$= \delta_i(k) z_i(m)$$

Deriving the gradient: Second layer

We can more compactly write

$$\frac{\partial L_i}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial L_i}{\partial W(1,1)} & \cdots & \frac{\partial L_i}{\partial W(1,M)} \\ \vdots & \ddots & \vdots \\ \frac{\partial L_i}{\partial W(K,1)} & \cdots & \frac{\partial L_i}{\partial W(K,M)} \end{bmatrix} = \boldsymbol{\delta}_i \mathbf{z}_i^T$$

where

$$\boldsymbol{\delta}_i = \begin{bmatrix} \delta_i(1) \\ \vdots \\ \delta_i(K) \end{bmatrix} = \begin{bmatrix} -2(y_i(1) - h(\mathbf{w}_1^T \mathbf{z}_i))h'(\mathbf{w}_1^T \mathbf{z}_i) \\ \vdots \\ -2(y_i(K) - h(\mathbf{w}_K^T \mathbf{z}_i))h'(\mathbf{w}_K^T \mathbf{z}_i) \end{bmatrix}$$

$$= -2\left(\mathbf{y}_i - h(\mathbf{W}\mathbf{z}_i)\right) \underset{\uparrow}{\odot} h'(\mathbf{W}\mathbf{z}_i)$$

Hadamard product
(element-wise multiplication)

Deriving the gradient: First layer

$$
\begin{aligned}
\frac{\partial L_i}{\partial V(m,\ell)} &= \frac{\partial}{\partial V(m,\ell)} \|\mathbf{y}_i - h(\mathbf{W} g(\mathbf{V}\mathbf{x}_i))\|_2^2 \\
&= \frac{\partial}{\partial V(m,\ell)} \sum_{k=1}^{K} \left(y_i(k) - h\left(\mathbf{w}_k^T g(\mathbf{V}\mathbf{x}_i)\right)\right)^2 \\
&= \sum_{k=1}^{K} \Big(-2\left(y_i(k) - h(\mathbf{w}_k^T \mathbf{z}_i)\right) h'(\mathbf{w}_k^T \mathbf{z}_i) \\
&\qquad \times w_k(m) g'(\mathbf{v}_m^T \mathbf{x}_i) x_i(\ell)\Big) \\
&= \sum_{k=1}^{K} \delta_i(k) w_k(m) g'(\mathbf{v}_m^T \mathbf{x}_i) x_i(\ell)
\end{aligned}
$$

Deriving the gradient: First layer

$$\frac{\partial L_i}{\partial V(m,\ell)} = \sum_{k=1}^{K} \delta_i(k) w_k(m) g'(\mathbf{v}_m^T \mathbf{x}_i) x_i(\ell)$$

We can express this in matrix form as

$$\frac{\partial L_i}{\partial \mathbf{V}} = \begin{bmatrix} \frac{\partial L_i}{\partial V(1,1)} & \cdots & \frac{\partial L_i}{\partial V(1,d)} \\ \vdots & \ddots & \vdots \\ \frac{\partial L_i}{\partial V(M,1)} & \cdots & \frac{\partial L_i}{\partial V(M,d)} \end{bmatrix}$$

$$= \left(\mathbf{W}^T \boldsymbol{\delta}_i \odot g'(\mathbf{V}\mathbf{x}_i)\right) \mathbf{x}_i^T$$

Gradient descent update

The update at iteration $r+1$ is

$$\mathbf{W}^{(r+1)} = \mathbf{W}^{(r)} - \alpha_r \sum_{i=1}^{n} \frac{\partial L_i}{\partial \mathbf{W}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(r)}}$$

$$\mathbf{V}^{(r+1)} = \mathbf{V}^{(r)} - \alpha_r \sum_{i=1}^{n} \frac{\partial L_i}{\partial \mathbf{V}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(r)}}$$

The weights in our formula for the partial derivatives depend on the previous parameter estimate $\boldsymbol{\theta}^{(r)}$

We can calculate this gradient very efficiently because of the special structure imposed by the network

Computing the gradient

Notice that given $\mathbf{z}_i = g(\mathbf{V}\mathbf{x}_i)$, we can compute

$$\boldsymbol{\delta}_i = -2\left(\mathbf{y}_i - h(\mathbf{W}\mathbf{z}_i)\right) \odot h'(\mathbf{W}\mathbf{z}_i)$$

Given $\boldsymbol{\delta}_i$, we can then compute

$$\frac{\partial L_i}{\partial \mathbf{W}} = \boldsymbol{\delta}_i \mathbf{z}_i^T$$

$$\frac{\partial L_i}{\partial \mathbf{V}} = \left(\mathbf{W}^T \boldsymbol{\delta}_i \odot g'(\mathbf{V}\mathbf{x}_i)\right) \mathbf{x}_i^T$$

This suggests an efficient two-pass algorithm for computing the gradient at each iteration

Backpropagation

Forward pass

Using current weights $\boldsymbol{\theta}^{(r)}$, feed each $\mathbf{x}_i$ into the network and compute

- $\mathbf{V}^{(r)}\mathbf{x}_i$
- $\mathbf{z}_i = g(\mathbf{V}^{(r)}\mathbf{x}_i)$
- $\mathbf{W}^{(r)}\mathbf{z}_i$
- $f(\mathbf{x}_i; \boldsymbol{\theta}^{(r)}) = h(\mathbf{W}^{(r)}\mathbf{z}_i)$

Backpropagation

Backward pass

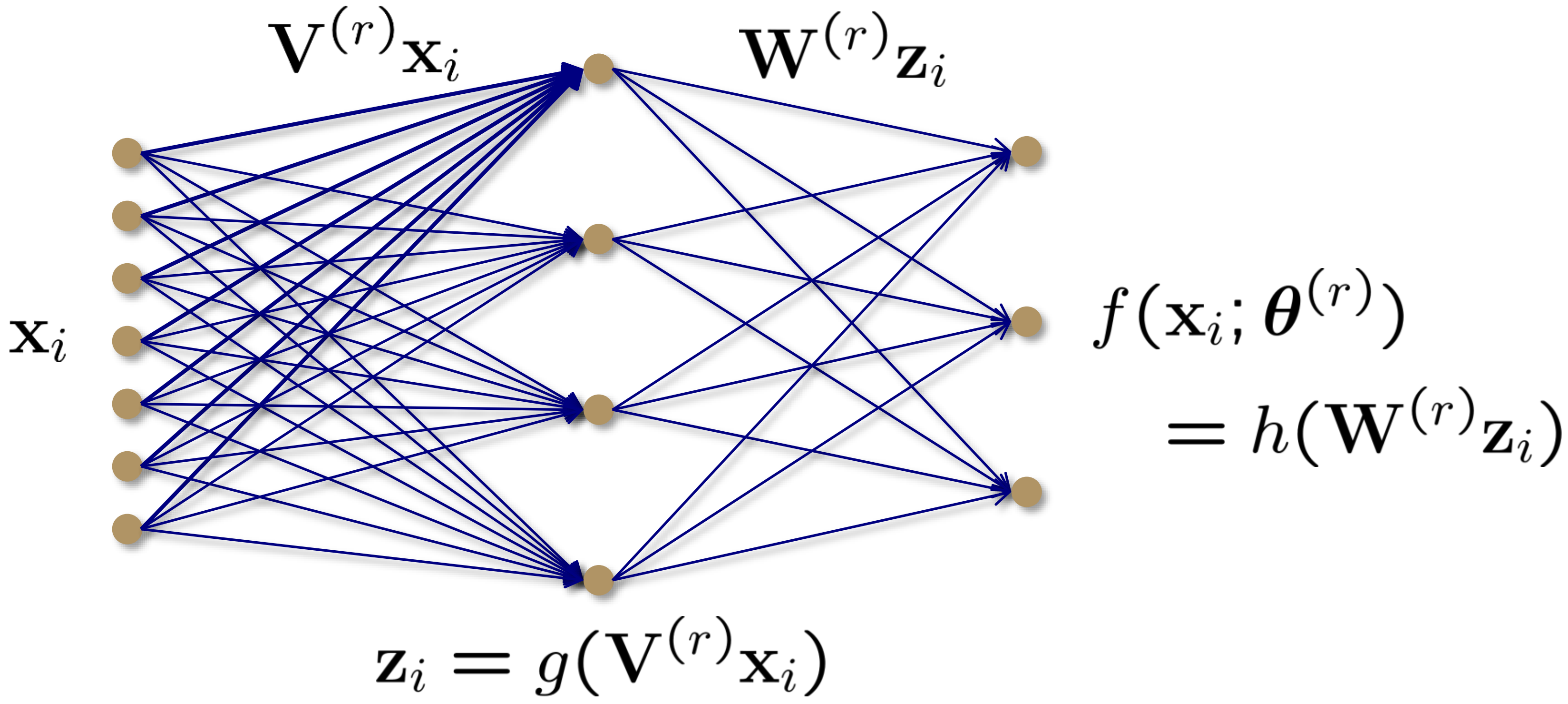
Now compute

- $\boldsymbol{\delta}_i^{(r)} = -2\left(\mathbf{y}_i - f(\mathbf{x}_i; \boldsymbol{\theta}^{(r)})\right) \odot h'(\mathbf{W}^{(r)}\mathbf{z}_i)$
- $\frac{\partial L_i}{\partial \mathbf{W}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(r)}} = \boldsymbol{\delta}_i^{(r)}\mathbf{z}_i^T$

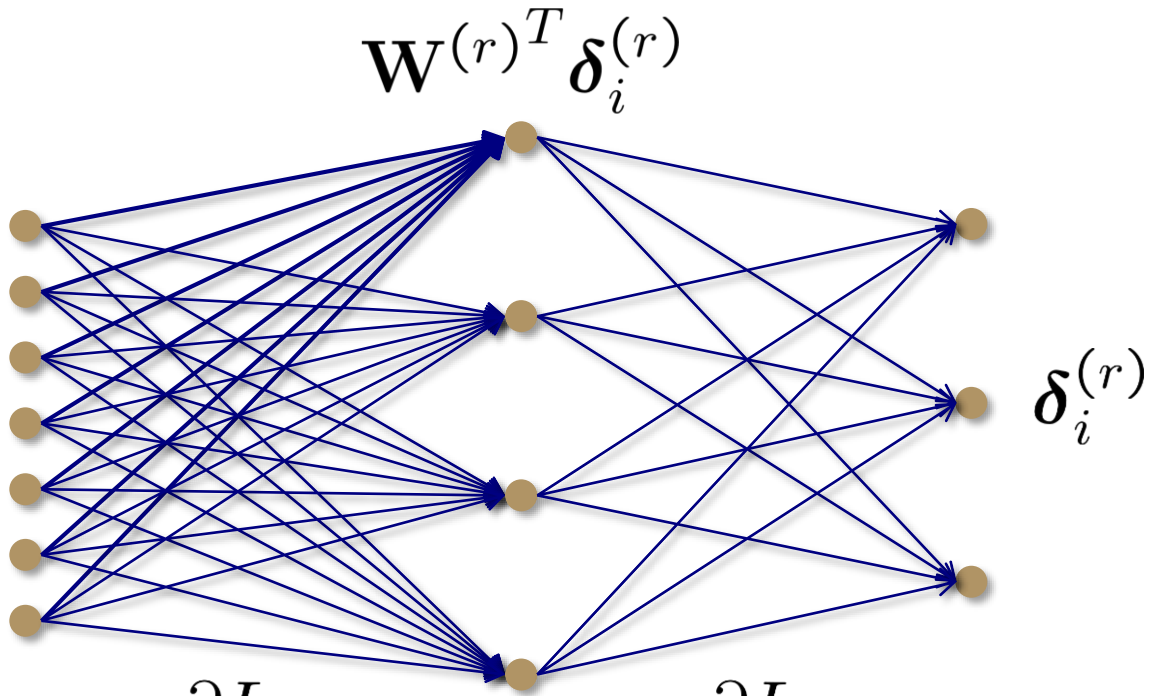
Next, “back-propagate” these values to the previous layer

- ${\mathbf{W}^{(r)}}^T\boldsymbol{\delta}_i^{(r)}$
- $\frac{\partial L_i}{\partial \mathbf{V}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(r)}} = \left({\mathbf{W}^{(r)}}^T\boldsymbol{\delta}_i^{(r)} \odot g'(\mathbf{V}^{(r)}\mathbf{x}_i)\right)\mathbf{x}_i^T$

Forward pass


$\mathbf{V}^{(r)}\mathbf{x}_i$
$\mathbf{W}^{(r)}\mathbf{z}_i$
$\mathbf{x}_i$
$f(\mathbf{x}_i; \boldsymbol{\theta}^{(r)})$
$= h(\mathbf{W}^{(r)}\mathbf{z}_i)$
$\mathbf{z}_i = g(\mathbf{V}^{(r)}\mathbf{x}_i)$

Backward pass


$\mathbf{W}^{(r)^T}\boldsymbol{\delta}_i^{(r)}$
$\boldsymbol{\delta}_i^{(r)}$

$$\frac{\partial L_i}{\partial \mathbf{V}} = \left(\mathbf{W}^{(r)^T}\boldsymbol{\delta}_i^{(r)} \odot g'(\mathbf{V}^{(r)}\mathbf{x}_i)\right)\mathbf{x}_i^T$$

$$\frac{\partial L_i}{\partial \mathbf{W}} = \boldsymbol{\delta}_i^{(r)}\mathbf{z}_i^T$$

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

Initial conditions make a big difference

To avoid "saturation", it is often a good idea to pre-process the data to have zero mean and unit variance or to be scaled to lie in $[0, 1]$

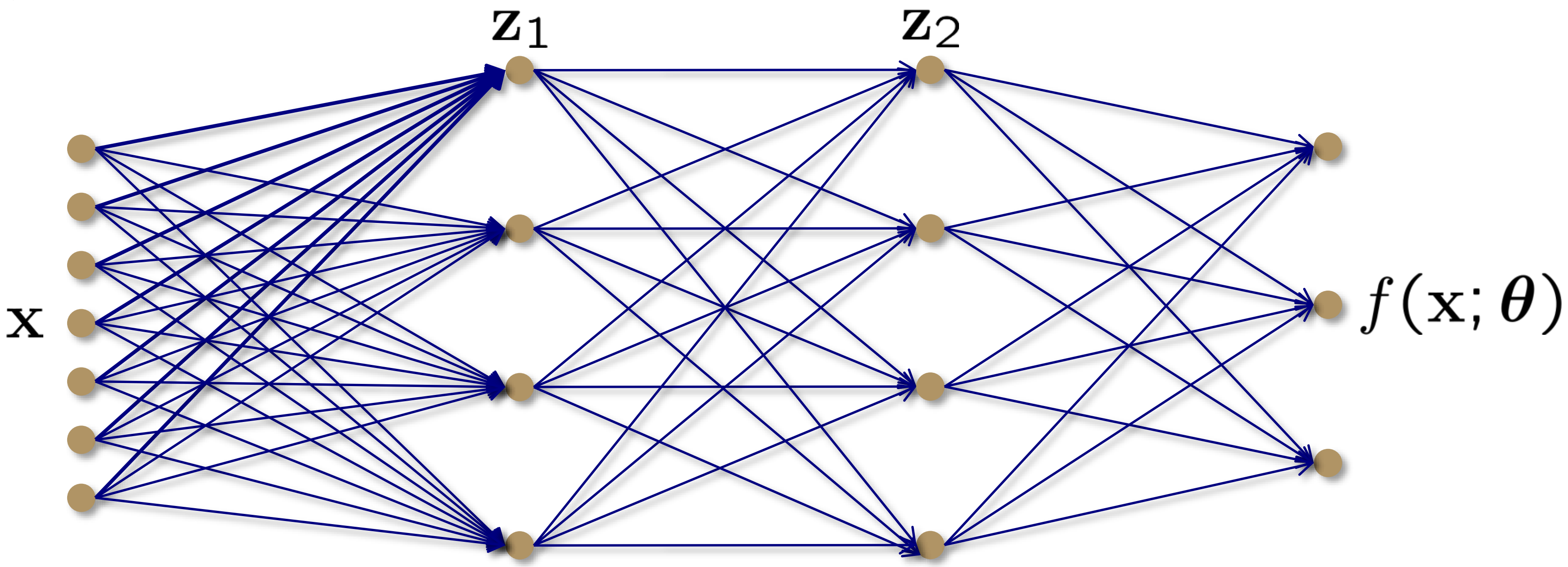
Since our objective function will have many local minima, it is generally a good idea to try several random starting values

Common heuristics for initializing the weights in a fully connected layer mapping m_1 inputs to m_2 outputs is to sample each weight from

- $U\left(-\frac{1}{\sqrt{m_1}}, \frac{1}{\sqrt{m_1}}\right)$
- $U\left(-\sqrt{\frac{6}{m_1+m_2}}, \sqrt{\frac{6}{m_1+m_2}}\right)$

Multi-layer neural networks

This framework, along with the backpropagation algorithm, can easily be extended to networks with multiple hidden layers



Has had a resurgence in recent years under the name ***"deep learning"***