

# Neural Networks

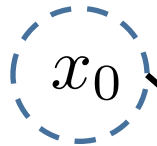
---

Natasha Jaques

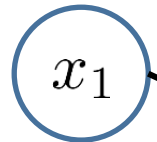


# Single neuron/node = linear regression

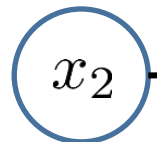
“bias unit”



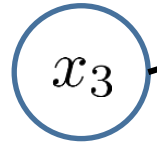
$$x_0 = 1$$



$$\theta_1$$



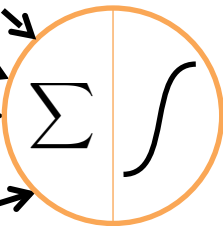
$$\theta_2$$



$$\theta_3$$

$$\theta \equiv w$$

# non-linearity



$$h_{\theta}(\mathbf{x})$$

$$= \sigma(\theta^{\top} \mathbf{x})$$

$$= \frac{1}{1 + e^{-\theta^{\top} \mathbf{x}}}$$

Binary  
Logistic  
Regression

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

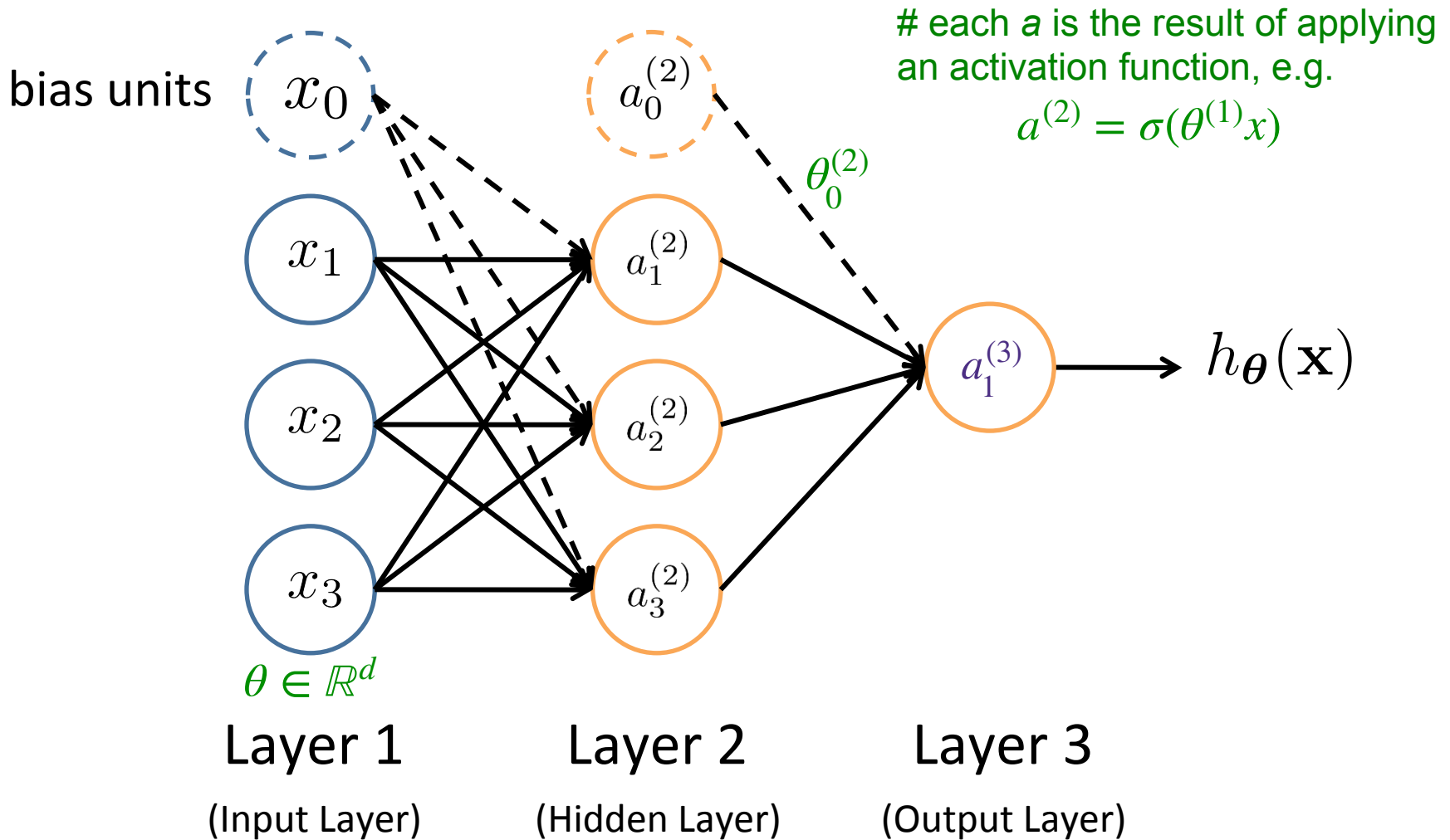
$d=3$

$$x \in \mathbb{R}^d$$

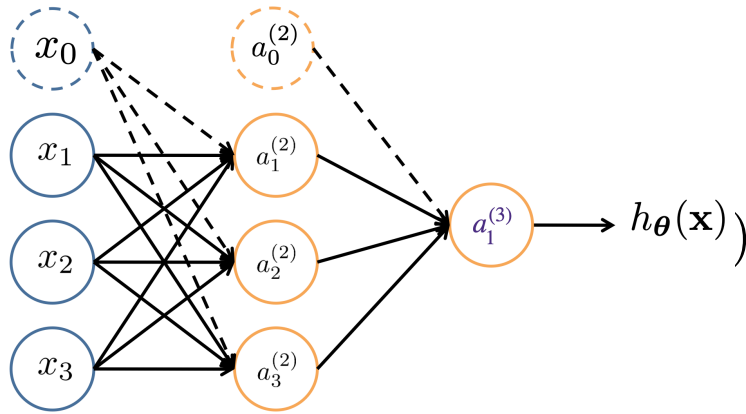
Sigmoid (logistic) activation function:

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

# Neural Network



# 4 inputs, 3 outputs



$a_i^{(j)}$  = “activation” of unit  $i$  in layer  $j$   
 $\Theta^{(j)}$  = weight matrix stores parameters from layer  $j$  to layer  $j + 1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

$$h_{\Theta}(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

If network has  $s_j$  units in layer  $j$  and  $s_{j+1}$  units in layer  $j+1$ , then  $\Theta^{(j)}$  has dimension  $s_{j+1} \times (s_j+1)$

$$\Theta^{(1)} \in \mathbb{R}^{3 \times 4} \quad \Theta^{(2)} \in \mathbb{R}^{1 \times 4}$$

# This is row-vector-of-weights convention, where each row vector is one neuron.  
 Could also do column-vector-of-weights  
 $\Theta^{(1)} \in \mathbb{R}^{4 \times 3} \quad a^{(j+1)} = g(\Theta^{(j)T} x)$

# Multi-layer Neural Network - Binary Classification

# Internal part of NN learns its own features, enables non-linear boundaries

$$a^{(1)} = x$$

$$a^{(2)} = g(\Theta^{(1)} a^{(1)})$$

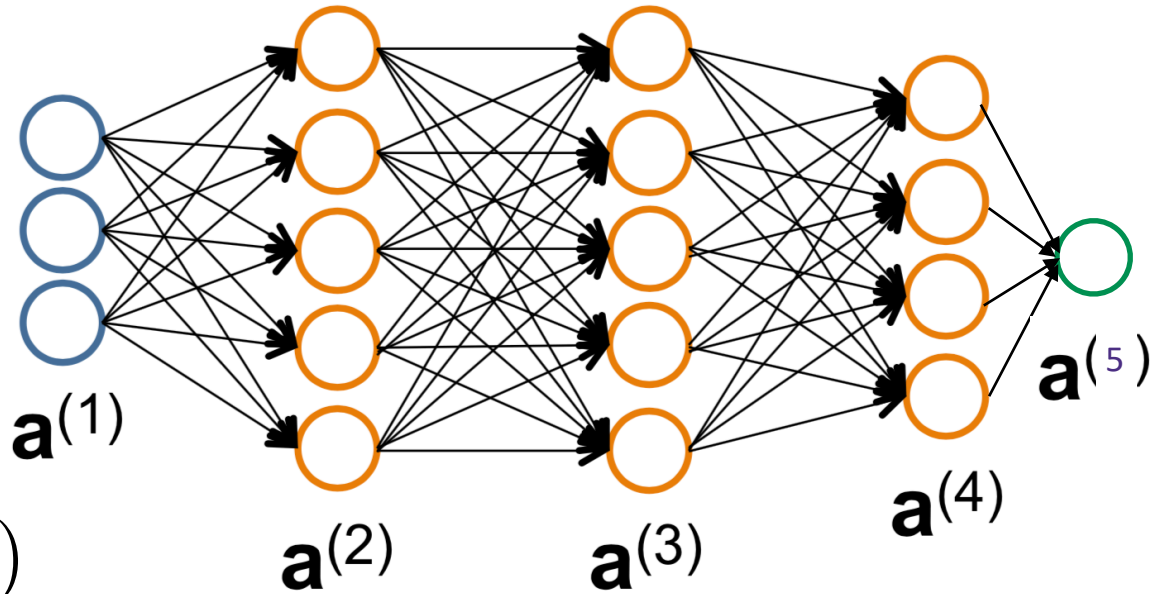
#  $g$  = activation function

⋮

$$a^{(l+1)} = g(\Theta^{(l)} a^{(l)})$$

⋮

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

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

Binary  
Logistic  
Regression

# Multi-layer Neural Network - Activation functions

Activation functions for nodes at intermediate layers include sigmoid, tanh, softplus, **ReLU**...

$$a^{(1)} = x$$

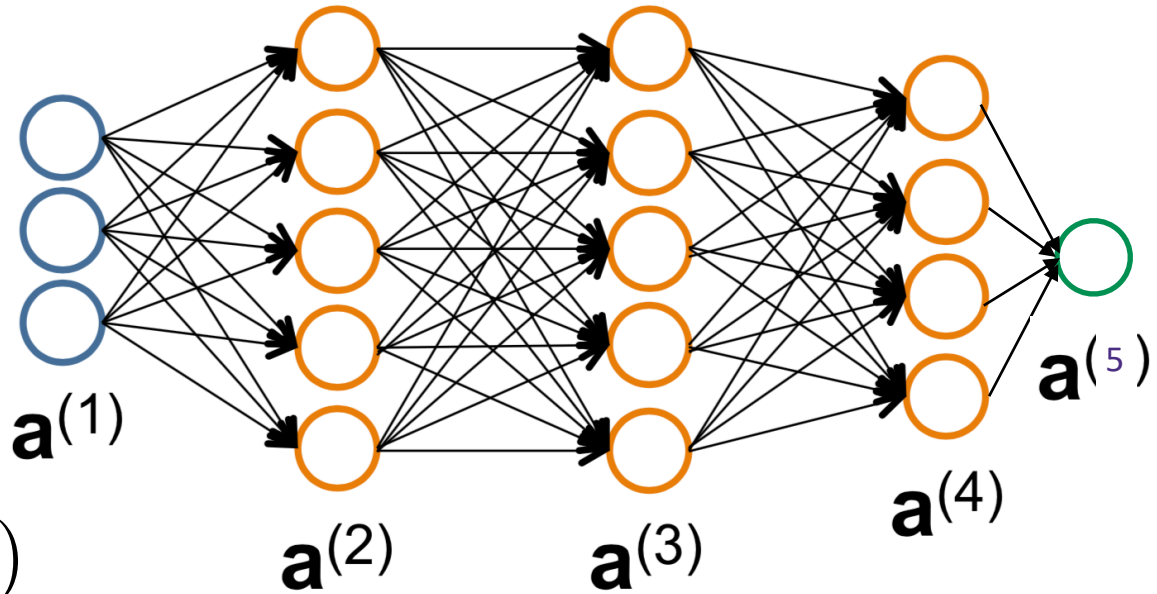
$$a^{(2)} = g(\Theta^{(1)} a^{(1)})$$

⋮

$$a^{(l+1)} = g(\Theta^{(l)} a^{(l)})$$

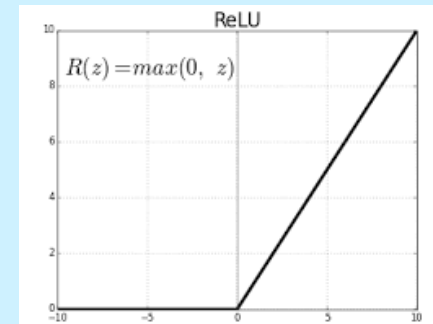
⋮

$$\hat{y} = \sigma(\Theta^{(L)} a^{(L)})$$



ReLU (Rectified Linear Unit) is most common:

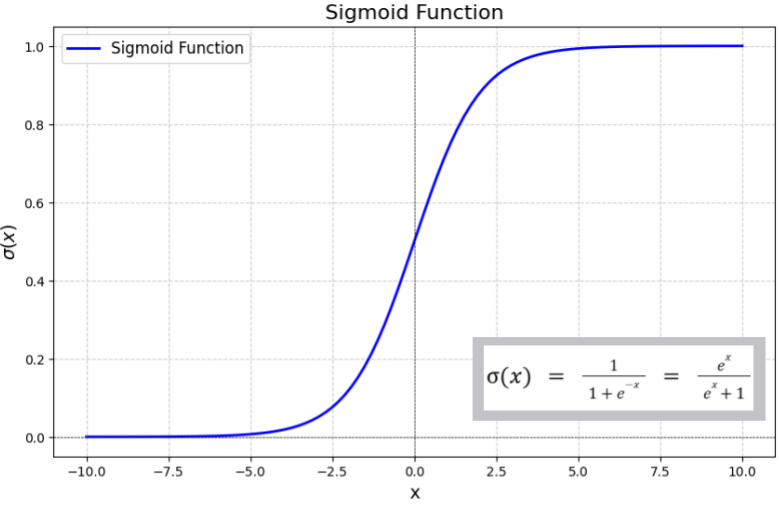
$$g(z) = \max\{0, z\}$$



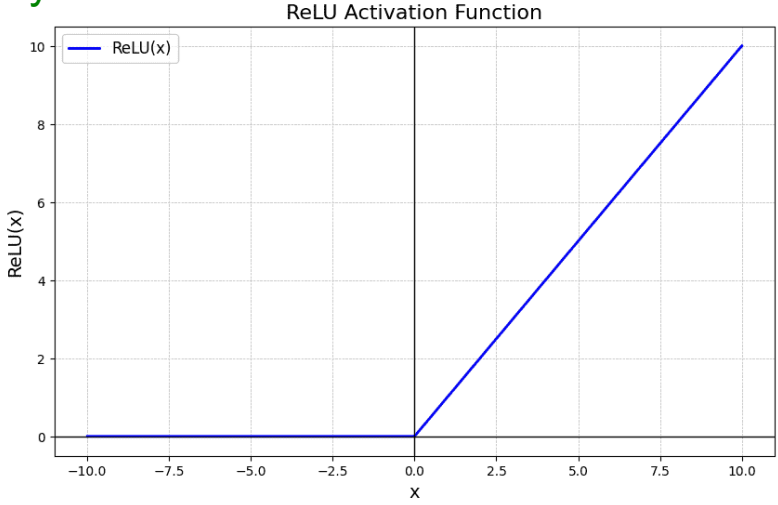
# A history of optimizing NNs has learned ReLU > sigmoid as an activation function... but why?

# gradient saturates easily

# good gradients forever



Sigmoid



ReLU

## GRADIENTS



# Multiple Output Units: One-vs-Rest



Pedestrian



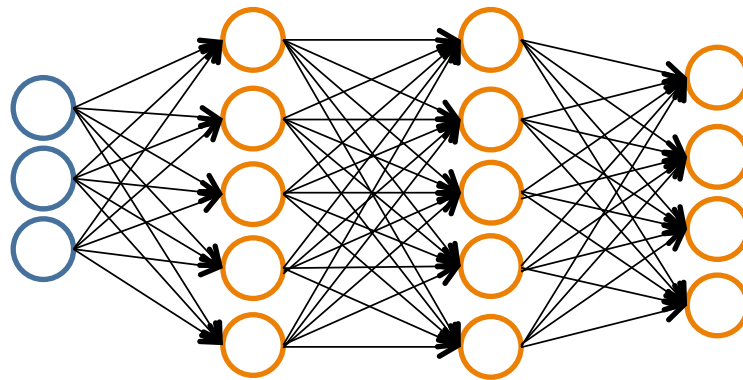
Car



Motorcycle



Truck



$$h_{\Theta}(\mathbf{x}) \in \mathbb{R}^K$$

# LLMs!

Multi-class  
Logistic  
Regression

We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

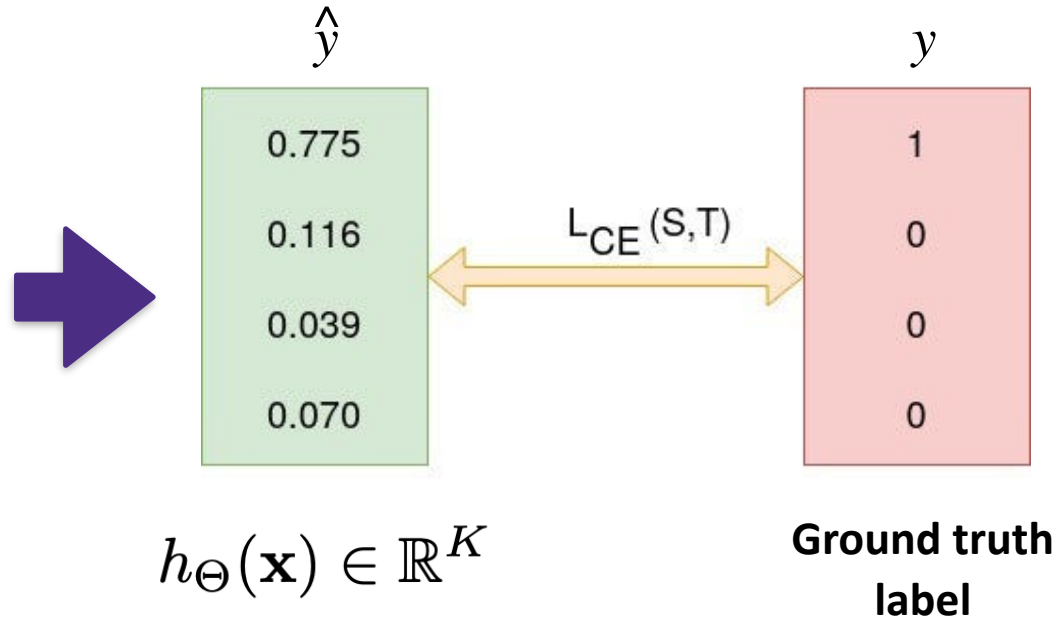
# Multi-class classification: Softmax + Binary Cross-Entropy

Recall: softmax function

$$P(y = c_j | x) = \frac{e^{w_j^\top x}}{\sum_{j'} e^{w_{j'}^\top x}}$$

# sum to 1,  $0 \leq y \leq 1$

Network's final output layer is a **softmax layer**, applies softmax activation function



Compare predicted values to ground truth using binary **cross-entropy**

$$H(y, \hat{y}) = - \sum_{j=1}^K y_j \log(\hat{y}_j)$$

# Multi-layer Neural Network - Regression

$$a^{(1)} = x$$

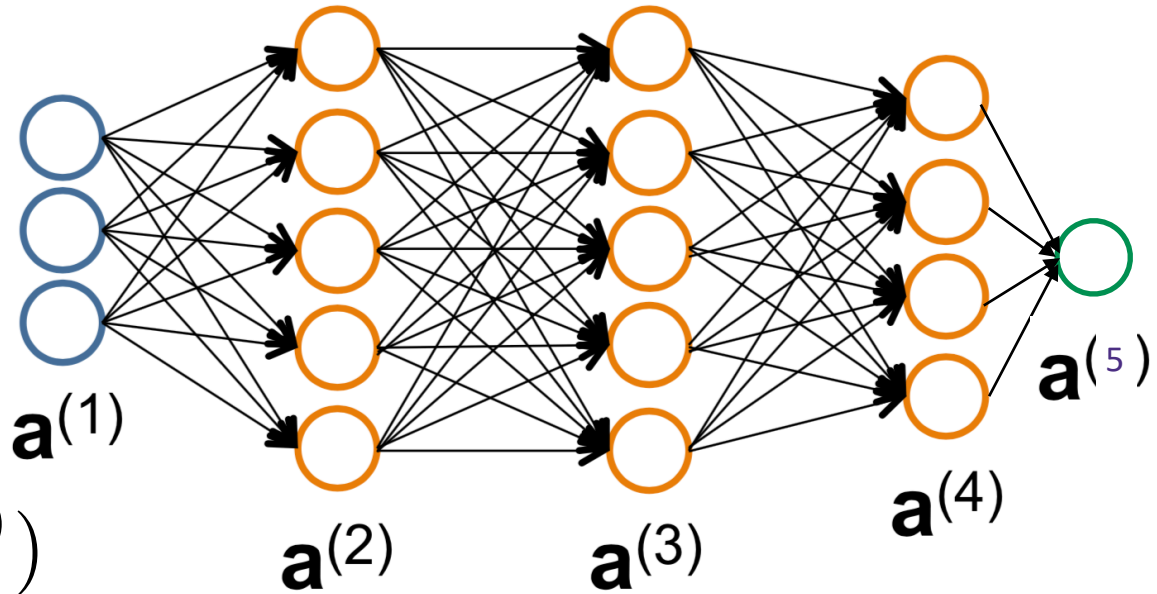
$$a^{(2)} = g(\Theta^{(1)} a^{(1)})$$

⋮

$$a^{(l+1)} = g(\Theta^{(l)} a^{(l)})$$

⋮

$$\hat{y} = \Theta^{(L)} a^{(L)}$$



Regression: apply squared error loss to output

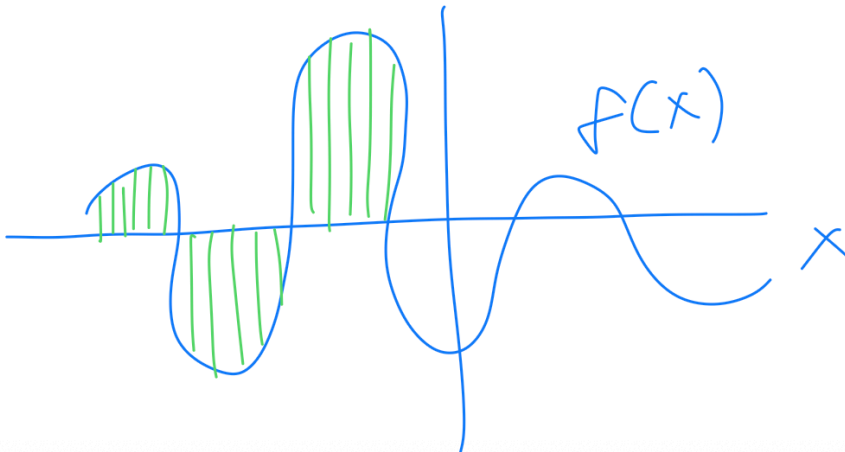
$$L(y, \hat{y}) = (y - \hat{y})^2$$

# MSE

# Neural Networks are arbitrary function approximators

**Theorem 10** (Two-Layer Networks are Universal Function Approximators). *Let  $F$  be a continuous function on a bounded subset of  $D$ -dimensional space. Then there exists a two-layer neural network  $\hat{F}$  with a finite number of hidden units that approximate  $F$  arbitrarily well. Namely, for all  $\mathbf{x}$  in the domain of  $F$ ,  $|F(\mathbf{x}) - \hat{F}(\mathbf{x})| < \epsilon$ . # 1989*

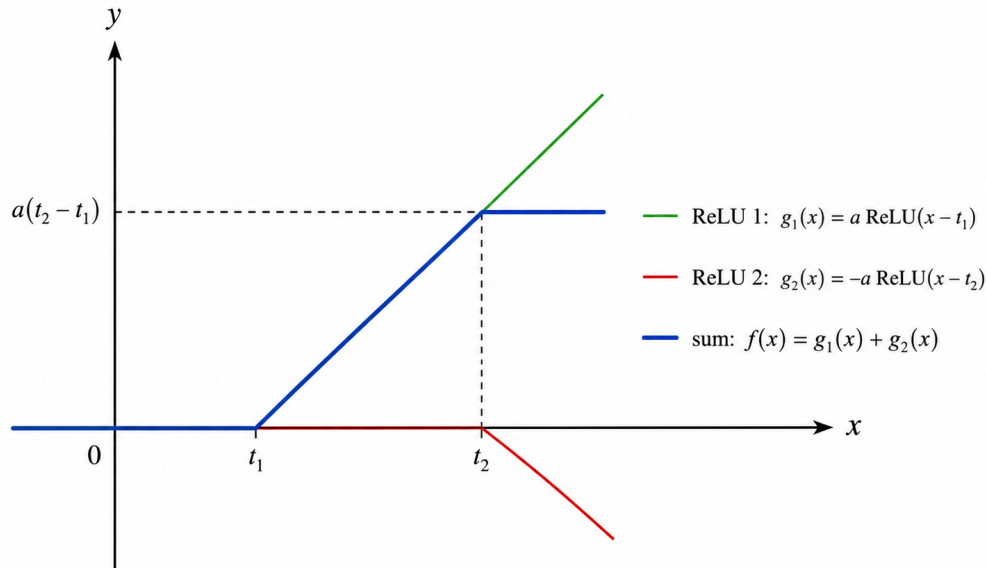
Cybenko, Hornik (theorem reproduced from CIML, Ch. 10)



## Proof sketch:

- Show how you can use 4 neurons to construct a pulse function
- Put together many many of these to approximate any function

# How to construct a pulse with 4 ReLUs



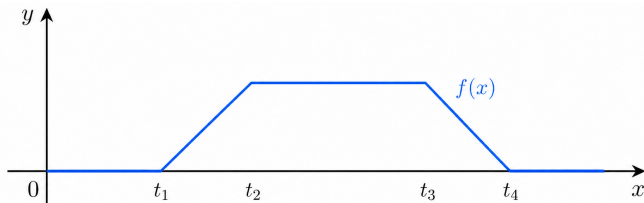
Recall:

$$g(x) = \max(0, x) \quad \# \text{ ReLU}$$

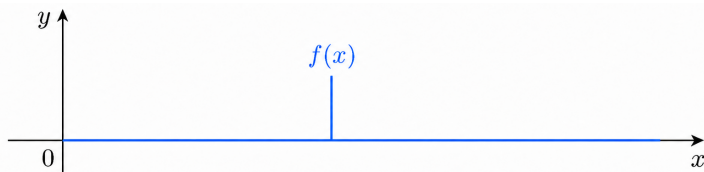
$$a(x) = g(mx + b) \quad \# \theta = \langle m, b \rangle$$

#  $\theta$  enables inverting and flipping the ReLU function, and arbitrarily scaling height and slope

# Combine two ReLUs to get a function stepping up



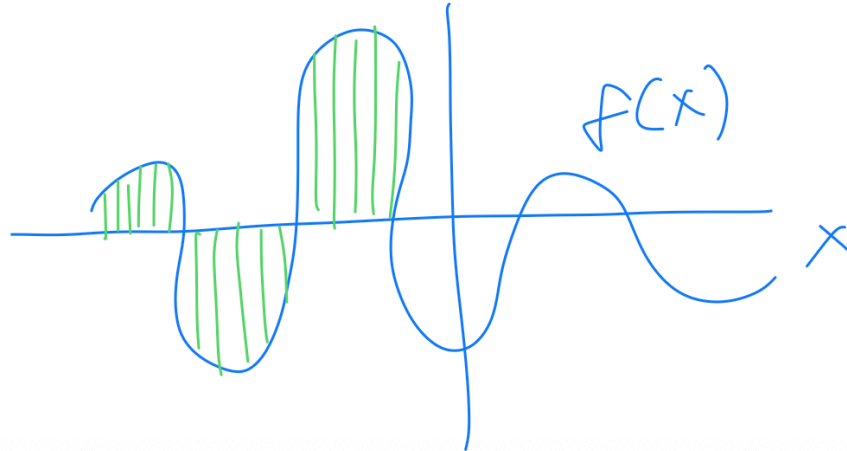
# Getting it to step back down takes another two



# Now scale it so they're really squished together, and you get a beautiful pulse

# Universal Approximation Theorem

---

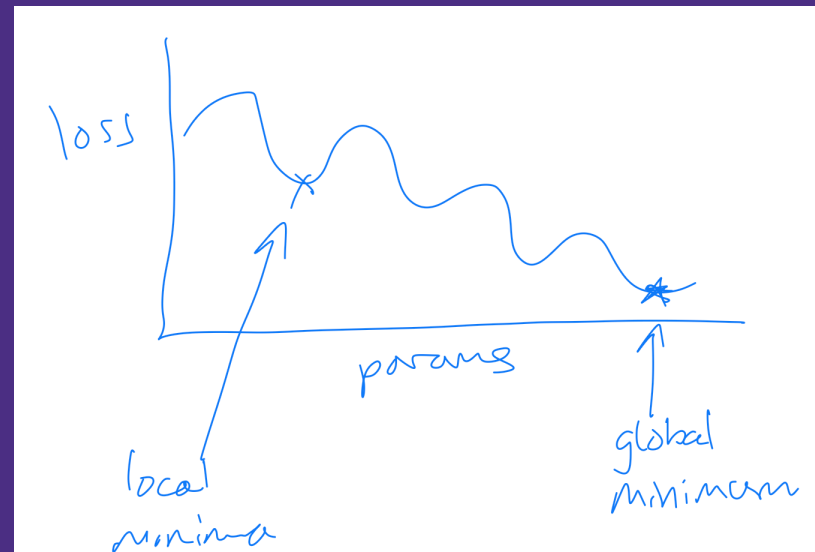


## Caveats:

- How many neurons would it take to approximate some function?
  - We don't know
- Pulses might not be what the network learns in practice
- This can work in theory... pending how well we can optimize it

# Training Neural Networks

---



$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

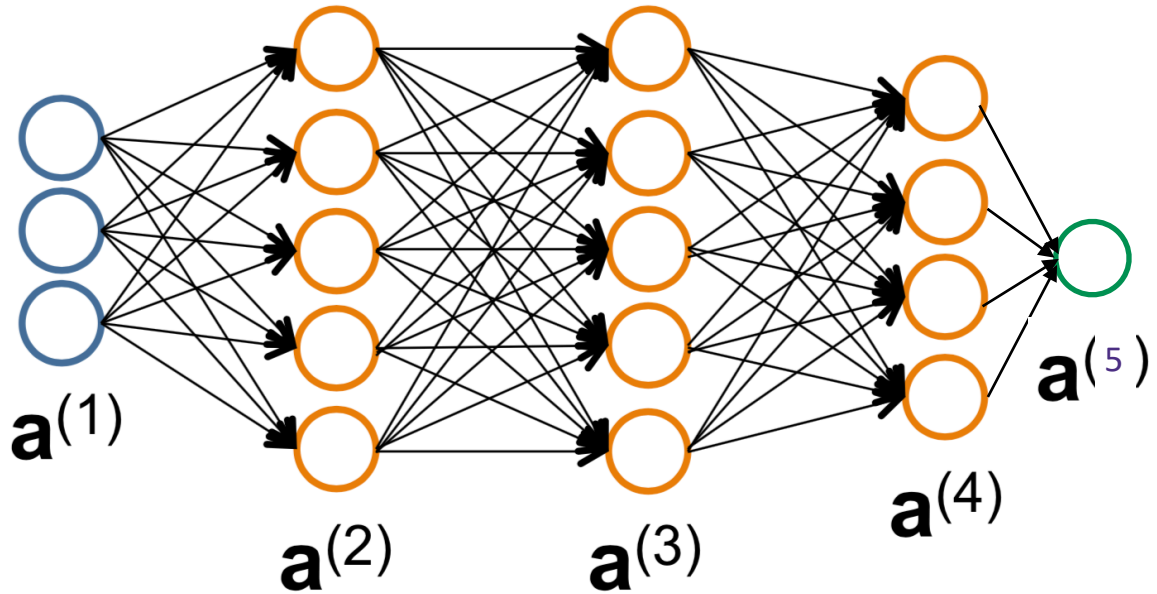
$$\vdots$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

$$\vdots$$

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

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

$$\text{Gradient Descent: } \Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y})$$

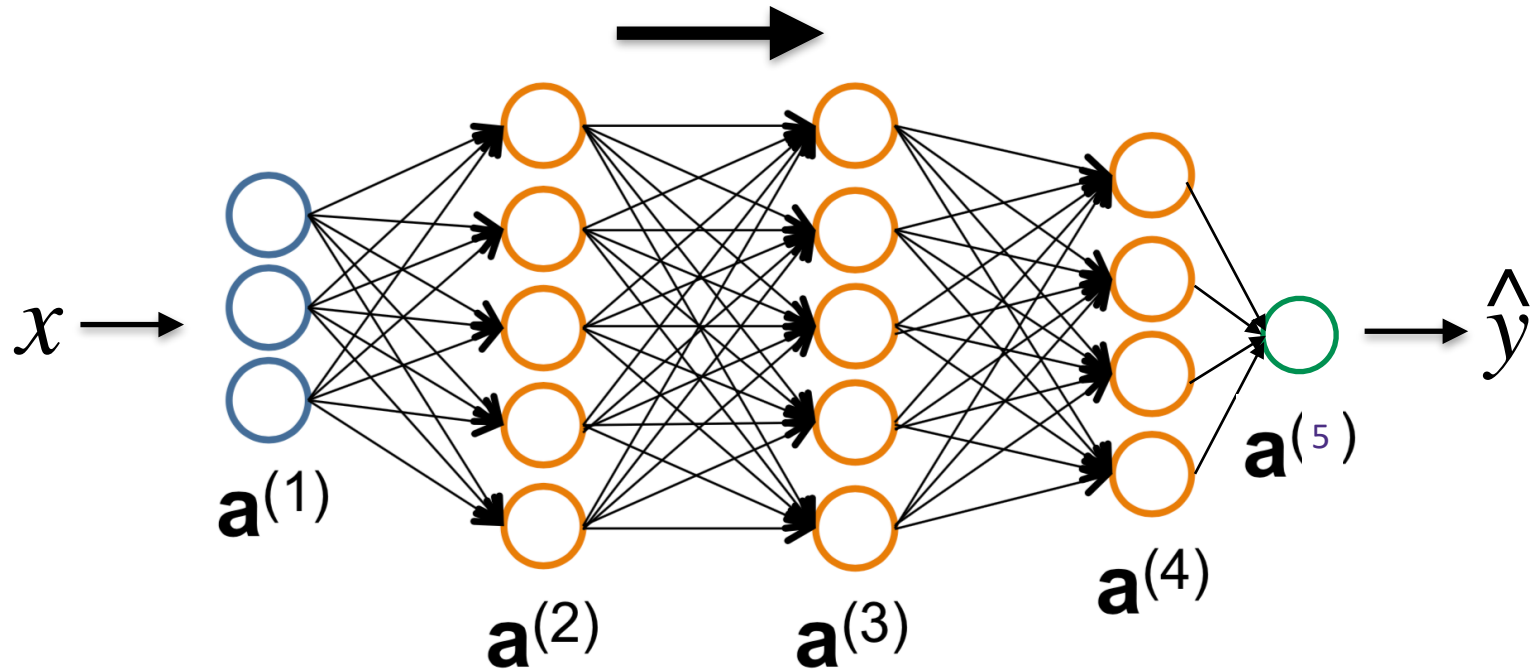
$$\forall l$$

# learning rate

# vector of partial derivatives  
for all network parameters

# Forward and backward pass

**Forward pass:** input  $\rightarrow$  output. compute all activation functions



$$\Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \leftarrow \quad L(y, \hat{y})$$

**Backward pass:** gradients  $\leftarrow$  loss. Backpropagate gradients from loss to update parameters

$$\text{Gradient Descent: } \Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$$

Seems simple enough, why are packages like PyTorch, Jax, Tensorflow, Theano, Cafe, MxNet synonymous with deep learning?

## 1. Automatic differentiation

# Applies back propagation via the chain rule

- For every function you use in the language (e.g. `torch.relu`), knows how to take the derivative and how to compose via the chain rule
- Does bookkeeping to save redundant calculations for efficiency

Gradient Descent:  $\Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$

Seems simple enough, why are packages like PyTorch, Tensorflow, Theano, Cafe, MxNet synonymous with deep learning?

1. Automatic differentiation

2. Convenient libraries

## Gradient Descent:

Seems simple enough,  
Theano, Cafe, MxNet s

1. Automatic differ

2. Convenient libra

```
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120) # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad() # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step() # Does the update
```

Gradient Descent:  $\Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} L(y, \hat{y}) \quad \forall l$

Seems simple enough, why are packages like PyTorch, Tensorflow, Theano, Cafe, MxNet synonymous with deep learning?

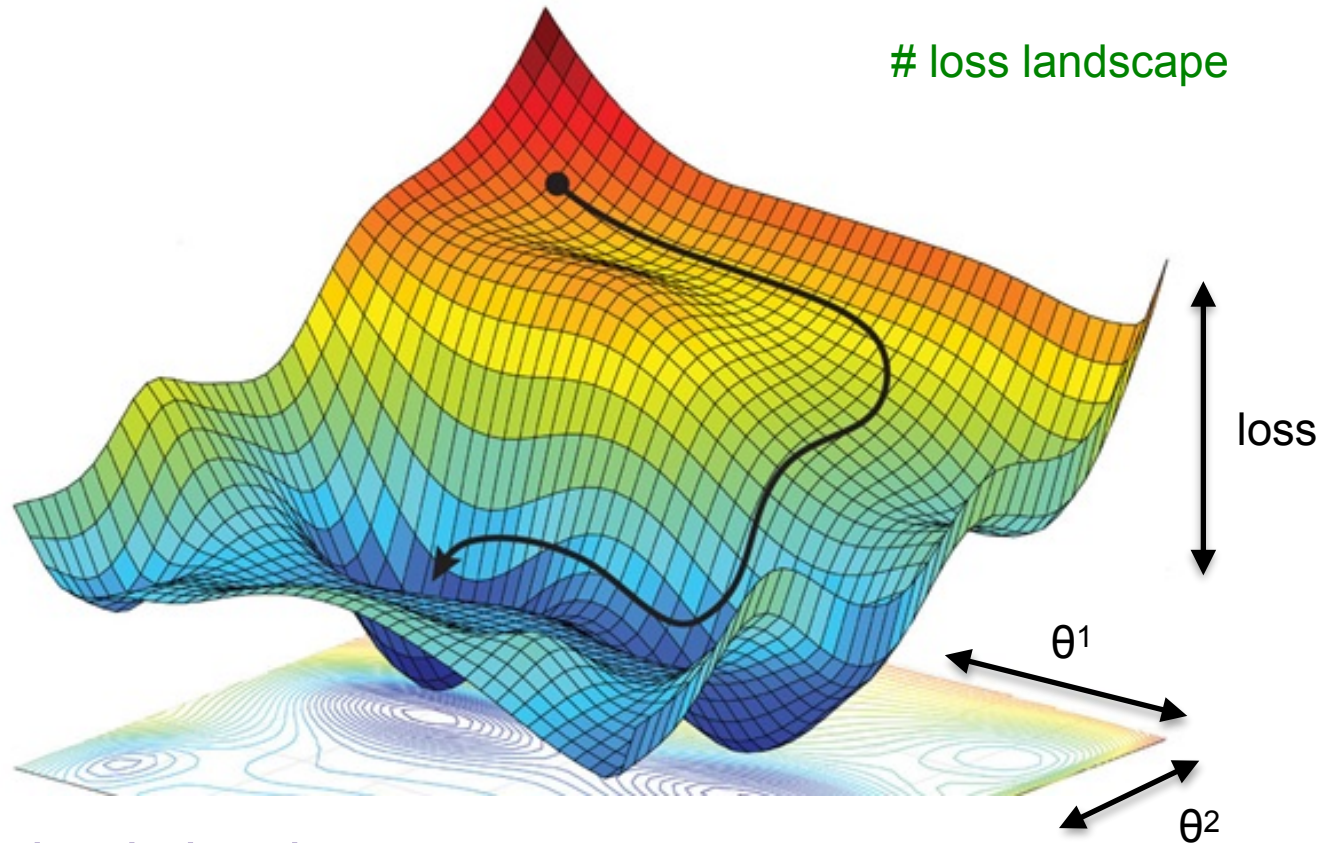
1. Automatic differentiation

2. Convenient libraries

3. GPU support

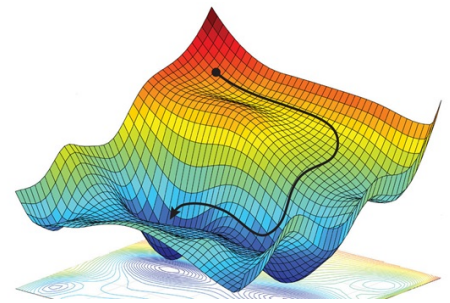
# Common training issues

## Neural networks are **non-convex**



- Optimization is hard
- Local minima can be a problem
- Rich history of work on getting these things to train effectively

# Common training issues



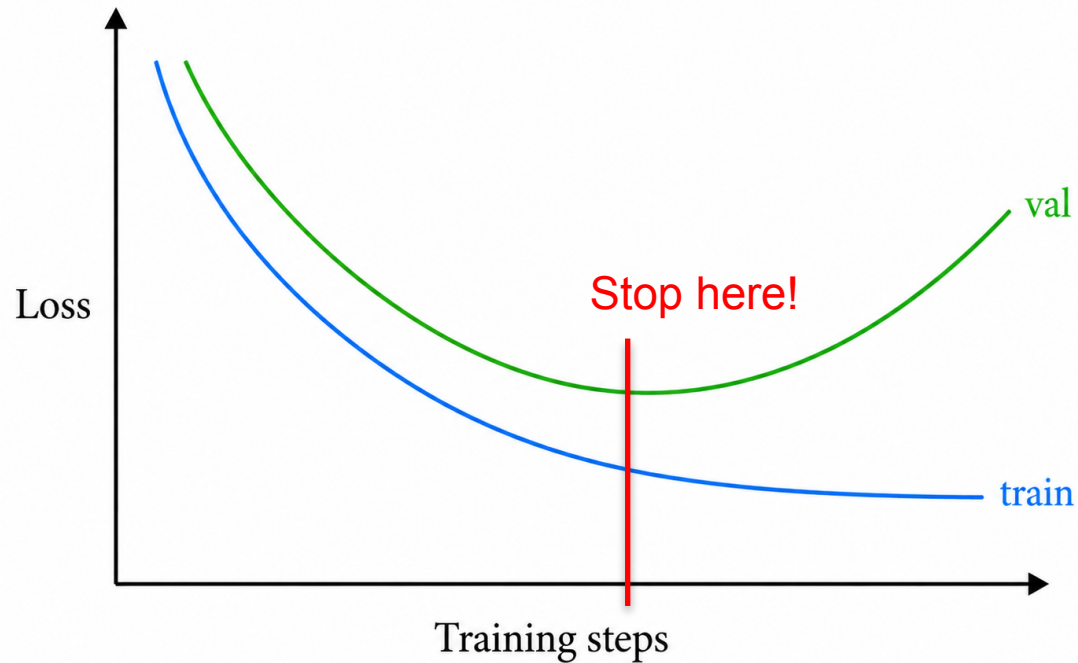
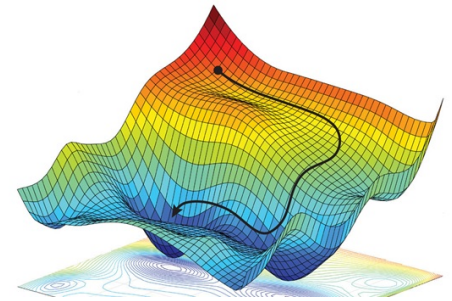
## Neural networks are **non-convex**

# explode

# vanish

- For large networks, **gradients** can **blow up** or **go to zero**.  
This can be helped by gradient clipping, **batchnorm**, ResNet architecture... many tricks
- **Hyperparameters** like **Stepsize**, **batchsize**, **momentum** all have large impact on optimizing the training error *and* generalization
- Fancier alternatives to SGD (Adagrad, Adam, LAMB, etc.) can significantly improve training
- Overfitting is common and not undesirable: typical to achieve 100% training accuracy even if test accuracy is just 80%
- **Overparameterization**: Making the network *bigger* may make training *faster* and it might generalize *better*!

# So how will I know if I succeed?



Each training step is a forward & backward pass of gradient descent

# Common training issues

---

## Training is too slow:

- Use larger learning rate, develop step size reduction schedule
- Use GPU resources
- Change batch size
- Use momentum and more exotic optimizers (e.g., Adam)
- Apply batch normalization
- Make network larger or smaller (# layers, # filters per layer, etc.)

# May also need to do performance optimization. Did you use a for loop instead of a matrix multiply??

## Test accuracy is low

- Try modifying all of the above, plus changing other hyperparameters
- Consider data augmentation strategies, dropout

# Common training issues

---

<https://playground.tensorflow.org/>

# Backprop

---



# Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

$$\vdots$$

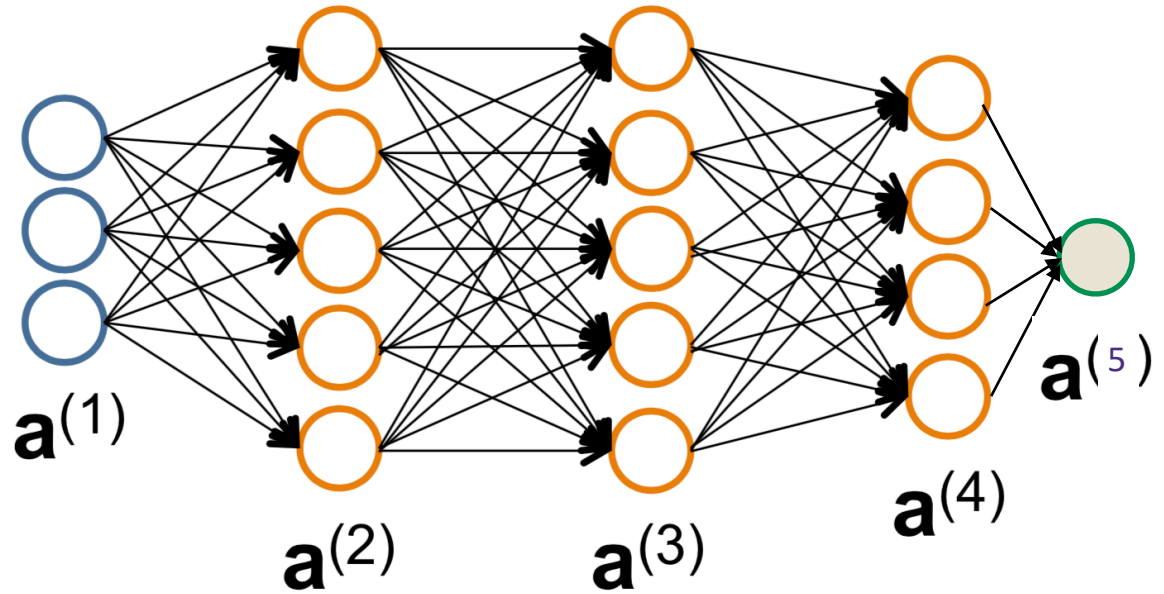
$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$



$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

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

# Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

Train by Stochastic Gradient Descent:

$$\Theta_{i,j}^{(l)} \leftarrow \Theta_{i,j}^{(l)} - \eta \frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

$$\vdots$$
$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

$$\vdots$$
$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

# forward pass: activations before this node

# backward pass: gradients ahead of this node

**Train by Stochastic Gradient Descent:**

$$\Theta_{i,j}^{(l)} \leftarrow \Theta_{i,j}^{(l)} - \eta \frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\delta_i^{(l)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l)}} = \sum_k \frac{\partial L(y, \hat{y})}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}$$

$$z_k^{(l+1)} = \Theta_{ki}^{(l)} a_i^{(l)} = [\Theta_{ki}^{(l)}]^T g(z^{(l)})$$

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = \Theta_{ki}^{(l)} g'(z_i^{(l)})$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\begin{aligned} \delta_i^{(l)} &= \frac{\partial L(y, \hat{y})}{\partial z_i^{(l)}} = \sum_k \frac{\partial L(y, \hat{y})}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} \\ &= \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)} g'(z_i^{(l)}) \\ &= a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)} \end{aligned}$$

# sigmoid derivative

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

$$\vdots$$
$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

$$\vdots$$
$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$a^{(l)} = g(z^{(l)})$$

$$\delta_i^{(L+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(L+1)}} = \frac{\partial}{\partial z_i^{(L+1)}} [y \log(g(z^{(L+1)})) + (1-y) \log(1-g(z^{(L+1)}))]$$

# final layer

$$= \frac{y}{g(z^{(L+1)})} g'(z^{(L+1)}) - \frac{1-y}{1-g(z^{(L+1)})} g'(z^{(L+1)})$$

$$= y - g(z^{(L+1)}) = y - a^{(L+1)}$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$L(y, \hat{y}) = y \log(\hat{y}) + (1-y) \log(1-\hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

⋮

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g(z^{(l+1)})$$

⋮

$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial L(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)}$$

$$\delta^{(L+1)} = y - a^{(L+1)}$$

Recursive Algorithm!

$$L(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}} \quad \delta_i^{(l+1)} = \frac{\partial L(y, \hat{y})}{\partial z_i^{(l+1)}}$$

# Backprop



# Backpropagation

Set  $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$  (Used to accumulate gradient)

For each training instance  $(\mathbf{x}_i, y_i)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  via forward propagation

Compute  $\delta^{(L)} = \mathbf{a}^{(L)} - y_i$  # derivative at last layer using loss

Compute errors  $\{\delta^{(L-1)}, \dots, \delta^{(2)}\}$

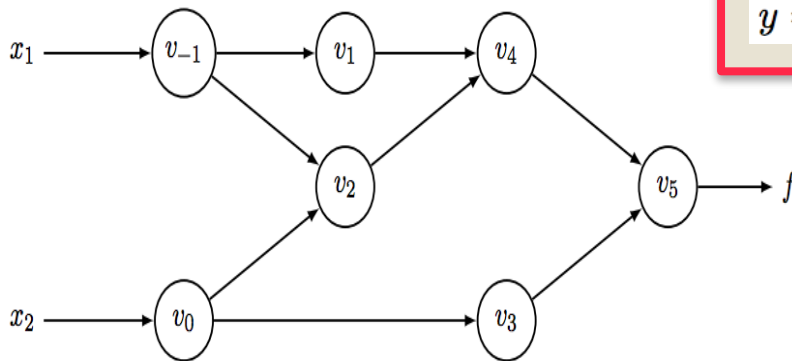
Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

$\mathbf{D}^{(l)}$  is the matrix of partial derivatives of  $J(\Theta)$

# Autodiff

Backprop for this simple network architecture is a special case of *reverse-mode auto-differentiation*:



$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

Forward Primal Trace		
$v_{-1} = x_1$	$=$	2
$v_0 = x_2$	$=$	5
<hr/>		
$v_1 = \ln v_{-1}$	$=$	$\ln 2$
$v_2 = v_{-1} \times v_0$	$=$	$2 \times 5$
<hr/>		
$v_3 = \sin v_0$	$=$	$\sin 5$
$v_4 = v_1 + v_2$	$=$	$0.693 + 10$
<hr/>		
$v_5 = v_4 - v_3$	$=$	$10.693 + 0.959$
<hr/>		
$y = v_5$	$=$	11.652

Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$	$=$	5.5
$\bar{x}_2 = \bar{v}_0$	$=$	1.716
<hr/>		
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$=$	$\bar{v}_{-1} + \bar{v}_1 / v_{-1} = 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$=$	$\bar{v}_0 + \bar{v}_2 \times v_{-1} = 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$=$	$\bar{v}_2 \times v_0 = 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$=$	$\bar{v}_3 \times \cos v_0 = -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$=$	$\bar{v}_4 \times 1 = 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$=$	$\bar{v}_4 \times 1 = 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$=$	$\bar{v}_5 \times (-1) = -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$=$	$\bar{v}_5 \times 1 = 1$
<hr/>		
$\bar{v}_5 = \bar{y}$	$=$	1

This is the special sauce in Tensorflow, PyTorch, Theano, ...