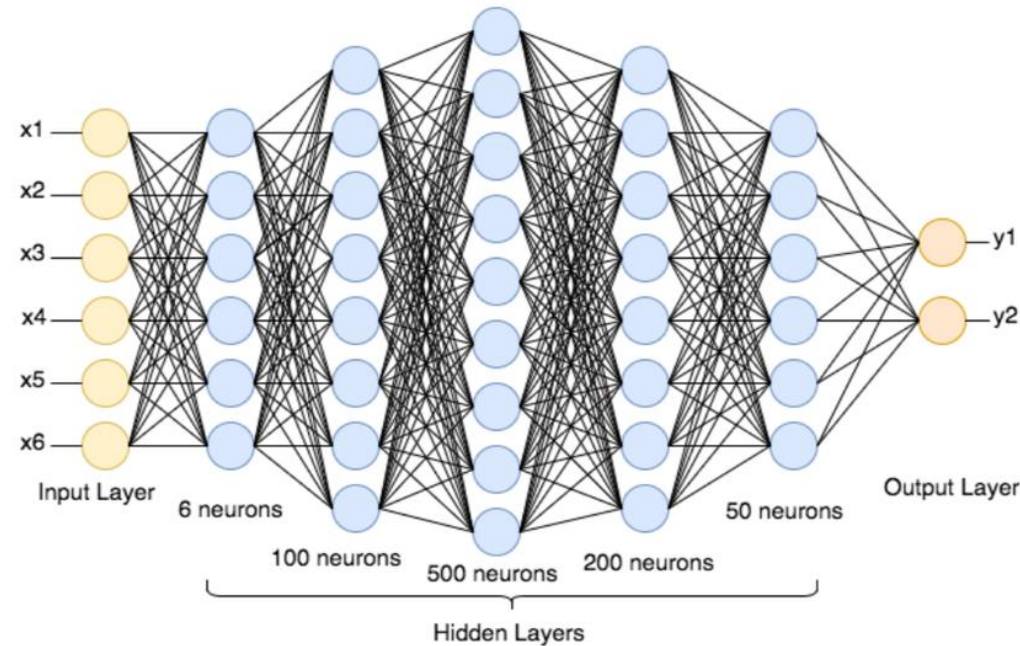


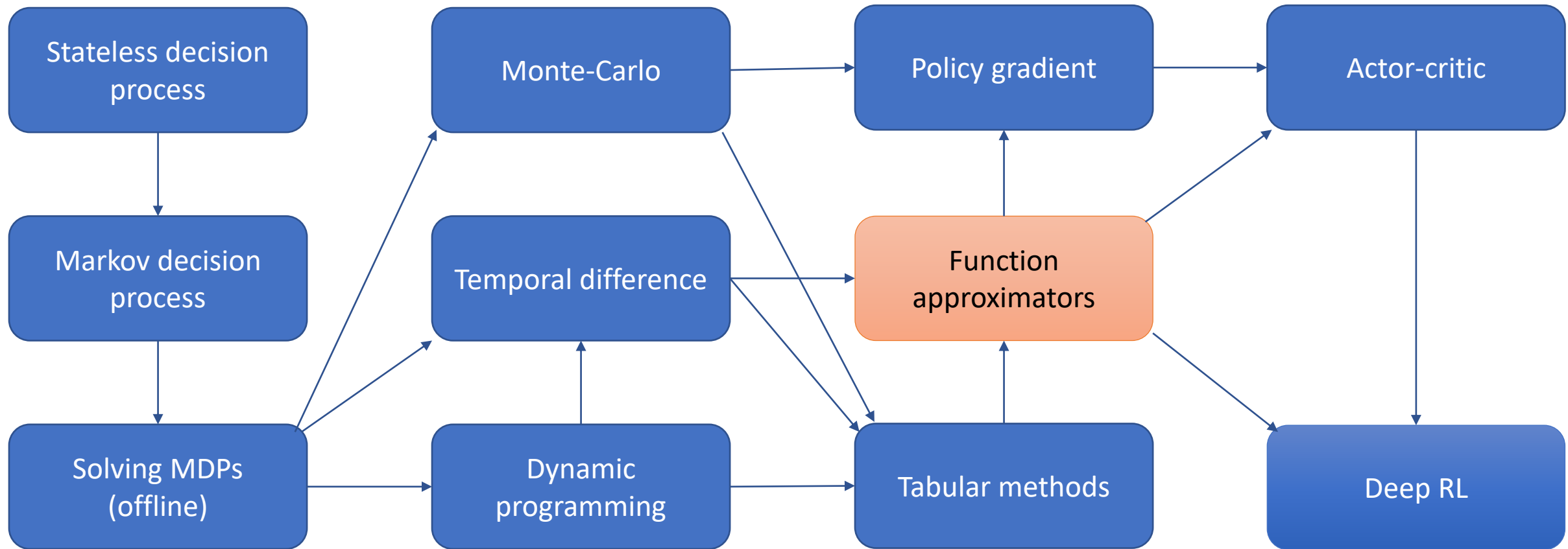
# CSCE-642 Reinforcement Learning

## Deep Neural Nets as Function Approximators



Instructor: Guni Sharon

# CSCE-689, Reinforcement Learning

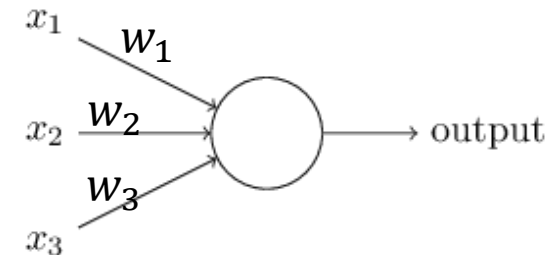


# Function approximator

- Approximate a function  $f: x \rightarrow y$  where  $x$  is a state (possibly + action) and  $y$  is a state value (or action value)
- $y$  can also be an action if  $f$  is a policy (we will see this later)
- We will consider 4 approximation tasks
  1. State value approximation  $\hat{V}: S \rightarrow \mathbb{R}$
  2. Action value approximation  $\hat{Q}: S \times \mathcal{A} \rightarrow \mathbb{R}$
  3. Policy approximation  $\hat{\pi}: S \rightarrow a \in \mathcal{A}$
  4. Model approximation  $\hat{M}: S \times A (\times \text{noise}) \rightarrow R, S'$
- A neural network is a function  $\hat{f}$  with many parameters  $\theta$  that can be tuned to approximate any well-behaved function  $f$  with arbitrary accuracy
  - Universal Function Approximation Theorem

# Perceptrons

- A single artificial neuron (a linear approximator)
- Tunable parameters are the weights,  $\theta = \{w_1, w_2, \dots, w_n\}$
- Output is:  $f = \sum_{i=1}^n x_i w_i$ , i.e., the dot product of vectors  $x$  and  $w$
- **Goal:** set  $w$  such that  $\hat{f}(x; w) \cong f(x)$
- **Approach:** minimize the difference (loss) between  $\hat{f}(x; w)$  and  $f(x)$
- $\min_w \sum_i |f(x_i) - \hat{f}(x_i; w)|$ 
  - Not differentiable
- $\min_w \sum_i \left( f(x_i) - \hat{f}(x_i; w) \right)^2$

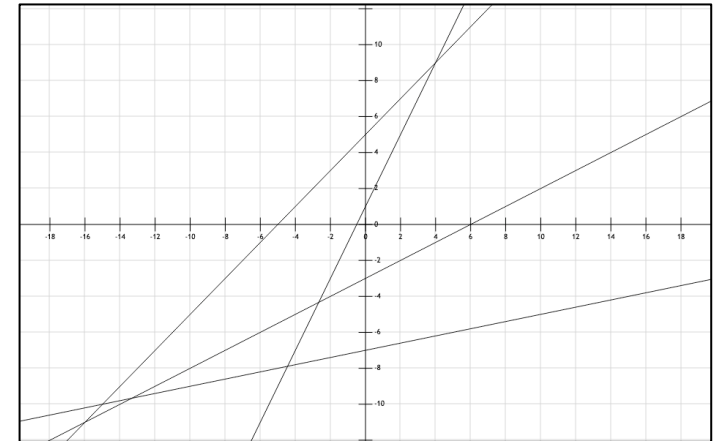
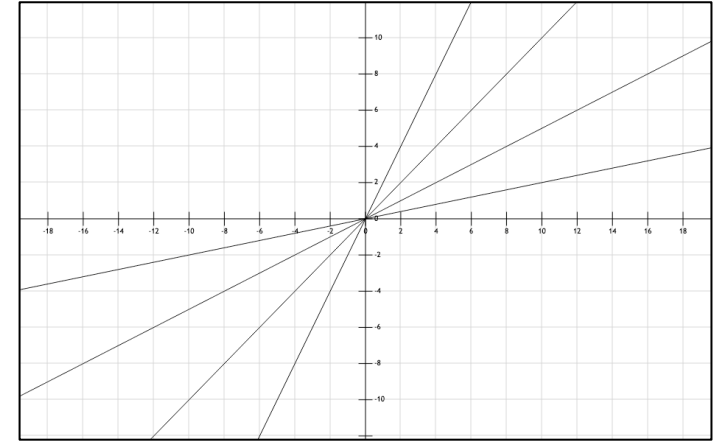


# Perceptrons

- $\min_w 0.5 \left( f(x) - \hat{f}(x; w) \right)^2 :=$  Squared loss function
- GD:  $w = w - \alpha \nabla_w \text{loss} = w + \alpha [f(x) - \hat{f}(x; w)] \nabla_w \hat{f}(x; w)$
- In TD learning,  $f(x) - \hat{f}(x; w)$  is the TD error ( $\hat{f}$  is  $\hat{V}$  or  $\hat{Q}$ )
- With a linear approximator (single perceptron)  $\nabla_w \hat{f}(x; w) = x$ 
  - Enables closed-form optimization (least-squares method)
- What are the limitations of perceptrons as function approximators?
  - Only able to capture linear relations between state features and output
  - The linear function  $f = x \cdot w$  must return 0 for  $x = [0, 0, 0 \dots]$

# Add bias

- Simple perceptron ( $f = wx$ ) capture linear functions that pass through the origin
- **Solution:** add a bias term:
  - $f = w \cdot x + b$
- How do we train the bias term?
  - Gradient decent as before:
  - $b = b - \alpha \nabla loss = b + [f(x) - \hat{f}(x; w, b)] \nabla_b \hat{f}(x; w, b)$
  - With a linear approximator (perceptron)  $\nabla_b \hat{f}(x; w, b) = 1$



# Beyond linearity: connecting perceptrons

- What if we connect several perceptrons to combine a net?
- $x$  is the input,  $z^i$  is the output (activation) vector for layer  $i$ ,  $w^i$  is a weight matrix for layers  $(i - 1, i)$  with dimensions  $(z^{i-1}, z^i)$

- What is the activation for layer 1?

- $z_1^1 = x_1 w_1^1, z_2^1 = x_2 w_2^1 \dots$
- $z^1 = x w^1$

- What is the activation of  $z^3$ ?

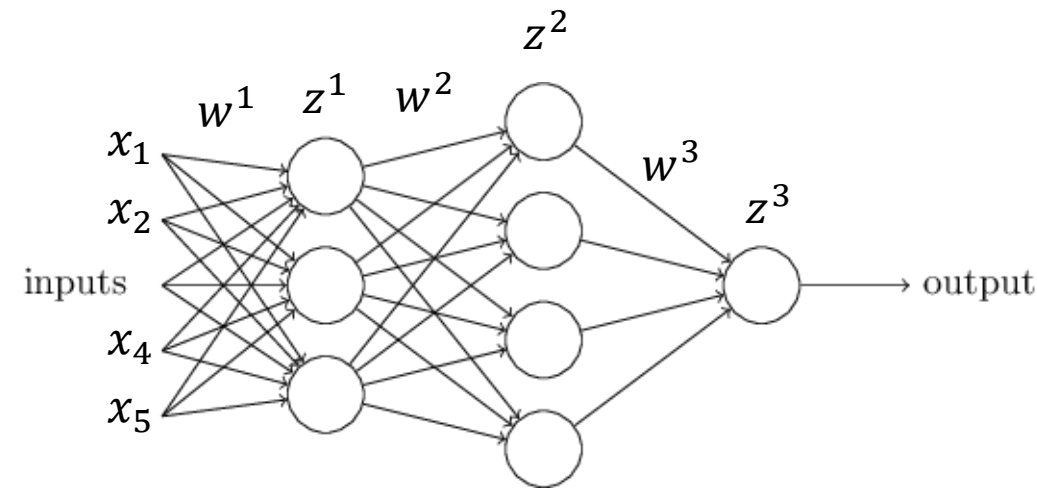
- $z^3 = ((x w^1) w^2) w^3 = x \cdot w'$

Where  $w' = w^1 w^2 w^3$

Matrix product is  
associative i.e.,  
 $(AB)C = A(BC)$

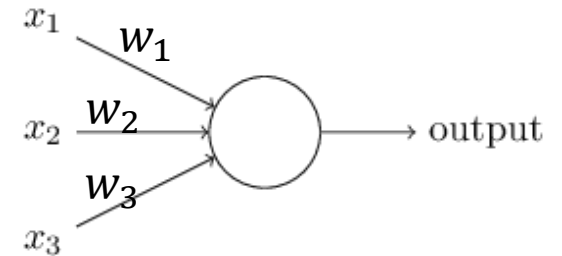
$z^1$   
 $z^2$   
 $z^3$

Not more powerful than a single perceptron



# Introducing non-linearity

- In order to represent general functions, the output of a neuron needs to be nonlinear w.r.t. the activation ( $z$ )
- We need an **Activation Function**  $g(z)$
- $output = g(wx + b)$
- Activation function Job description:
  - Well defined gradient (differentiable)
  - Informative gradient values (how to improve the objective)
  - Monotonic (since local optimums lead to suboptimal solutions with GD)
  - Fast to compute (intense working environment)
  - Covers the required output range
- Let's look at some candidates





# Common Activation Functions

Defined derivatives

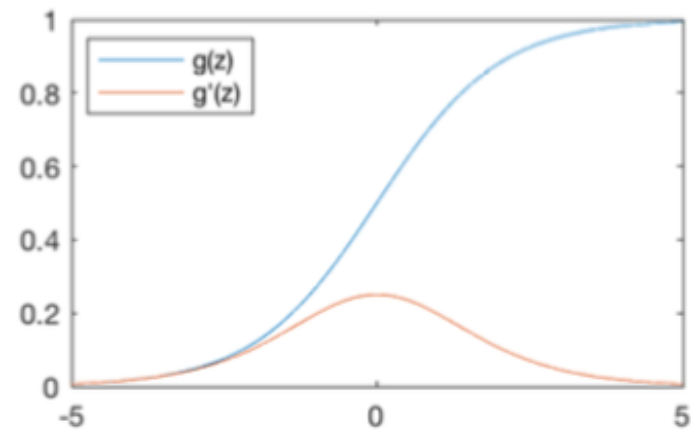
Informative derivative values

Monotonic

Fast to compute

Covers the required output range

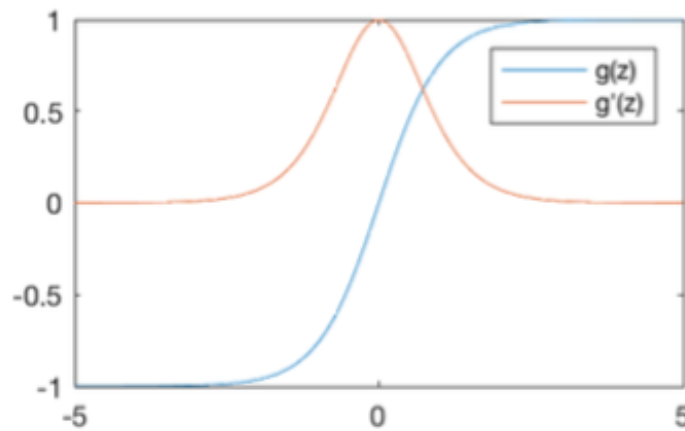
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

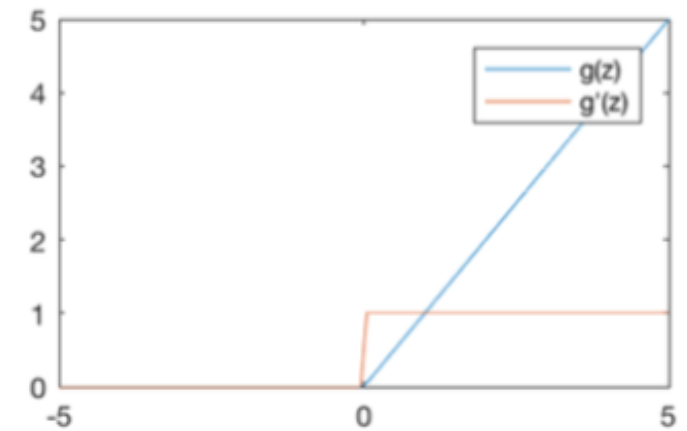
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

# Common Activation Functions

Defined derivatives

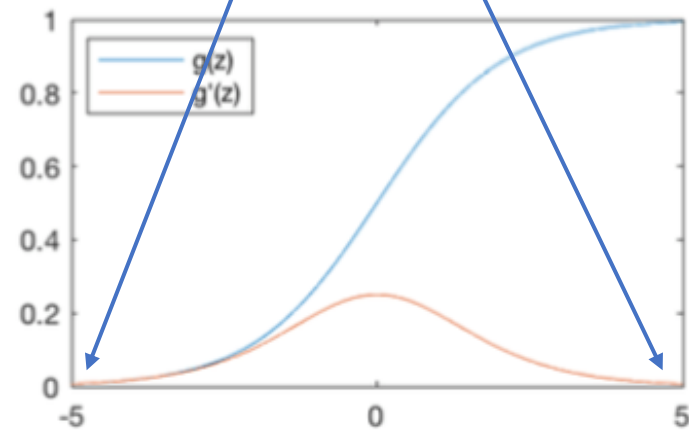
Informative derivative values

Monotonic

Fast to compute

Covers the required output range

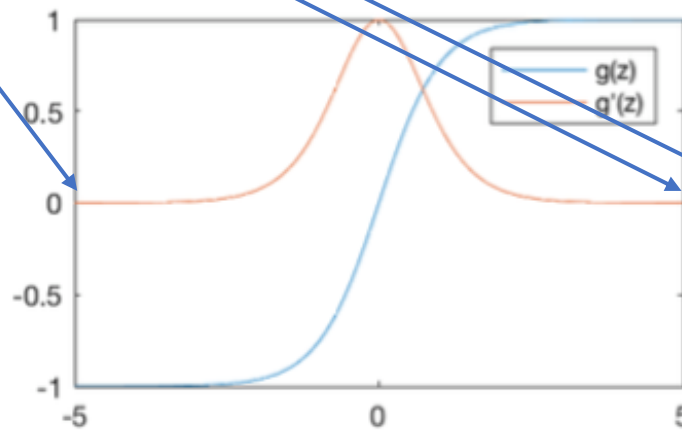
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

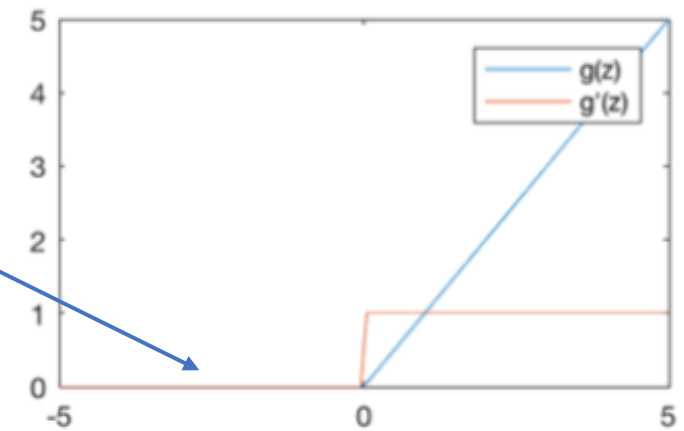
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

# Common Activation Functions

Defined derivatives

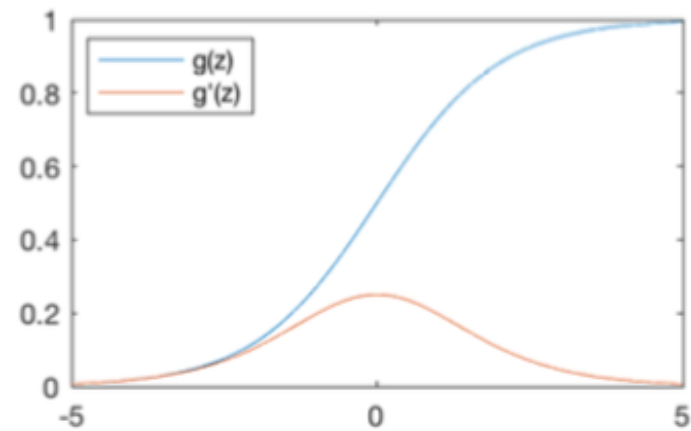
Informative derivative values

Monotonic

Fast to compute

Covers the required output range

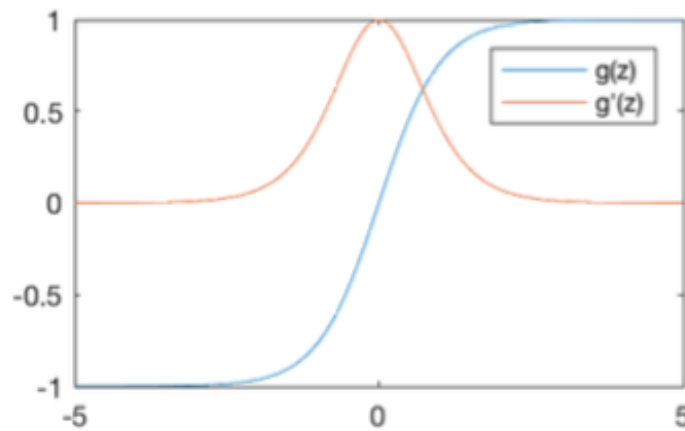
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

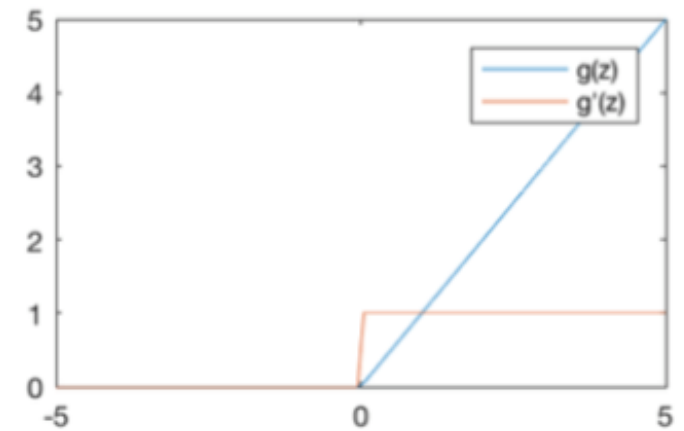
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

# Common Activation Functions

Defined derivatives

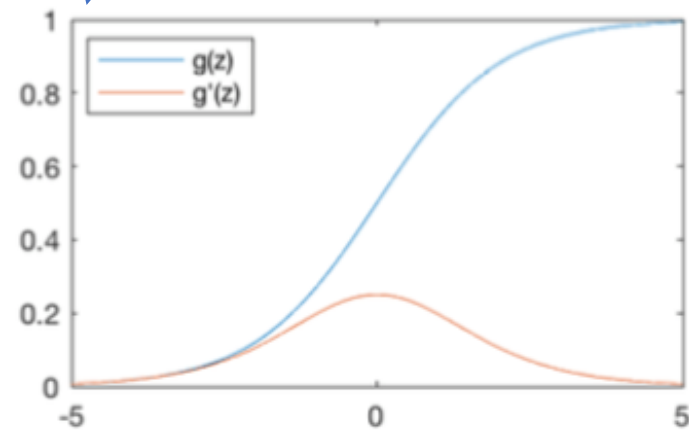
Informative derivative values

Monotonic

Fast to compute

Covers the required output range

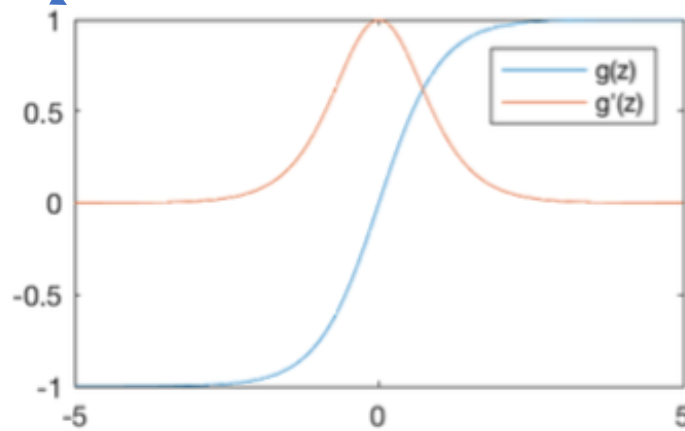
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

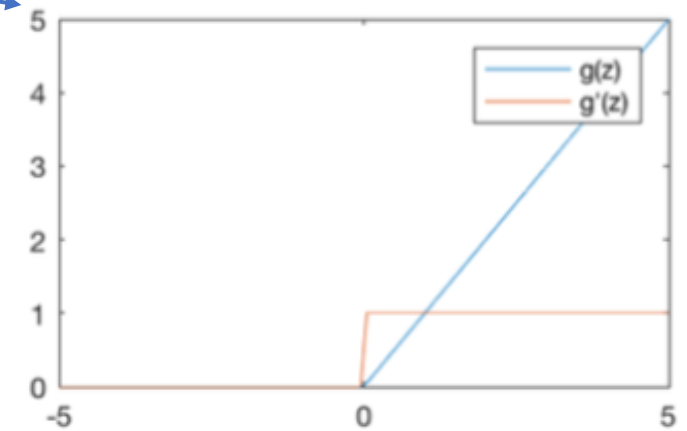
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

# Example: Sigmoid neuron

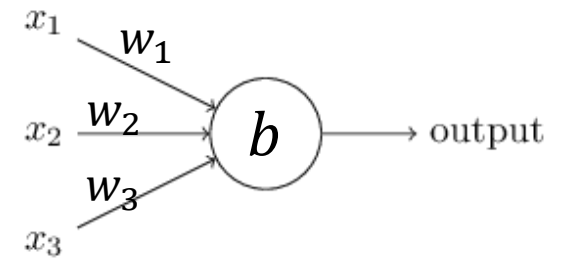
- Assume a single neuron with a Sigmoid activation function

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

- Where  $z = wx + b$

- How can we compute  $\frac{\partial l}{\partial w}$  or  $\frac{\partial l}{\partial b}$  ?

- The chain rule!  $\frac{\partial l}{\partial w} = \frac{\partial l}{\partial g(z)} \frac{\partial g(z)}{\partial z} \cdot \frac{\partial z}{\partial w}$



- Given a set of labeled observations  $x, y$  perform a GD step

- $w = w - \alpha \nabla loss = w - [\hat{f}(x; w, b) - y] \nabla_w \hat{f}(x; w, b)$

- $= w - \left[ \frac{1}{1+e^{-z}} - y \right] \frac{1}{1+e^{-z}} \left( 1 - \frac{1}{1+e^{-z}} \right) x$

Derivative of squared error loss  $\frac{\partial l}{\partial g(z)}$

# Example: Sigmoid neuron

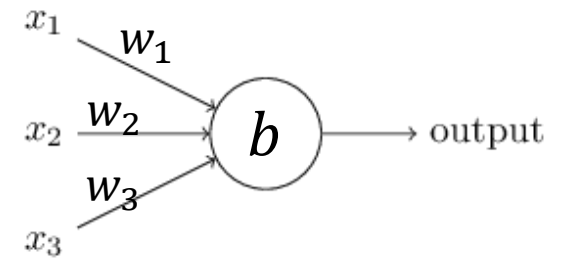
- Assume a single neuron with a Sigmoid activation function

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

- Where  $z = wx + b$

- How can we compute  $\frac{\partial l}{\partial w}$  or  $\frac{\partial l}{\partial b}$  ?

- The chain rule!  $\frac{\partial l}{\partial w} = \frac{\partial l}{\partial g(z)} \frac{\partial g(z)}{\partial z} \cdot \frac{\partial z}{\partial w}$



- Given a set of labeled observations  $x, y$  perform a GD step

- $w = w - \alpha \nabla loss = w - [\hat{f}(x; w, b) - y] \nabla_w \hat{f}(x; w, b)$

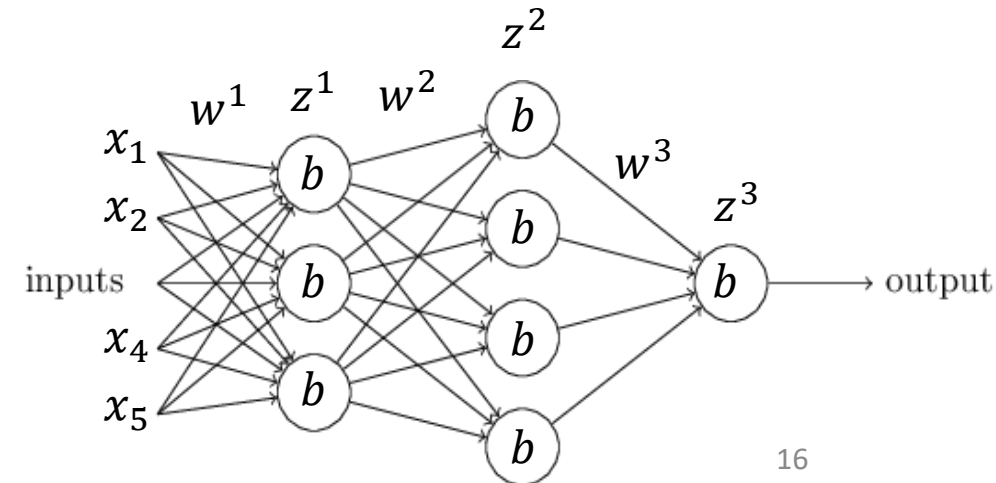
- $= w - \left[ \frac{1}{1+e^{-z}} - y \right] \frac{1}{1+e^{-z}} \left( 1 - \frac{1}{1+e^{-z}} \right) x$

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

$$\frac{\partial z}{\partial w} = x$$
$$\frac{\partial z}{\partial b} = 1$$

# Connecting neurons

- What if we connect several neurons to combine a net?
- $x$  is the input,  $z^i$  is the activation vector for layer  $i$ ,  $w^i$  is a weight matrix for layers  $(i - 1, i)$  with dimensions  $(z^{i-1}, z^i)$
- What is the output of layer 1?
  - $g(z_1^1) = g(b + xw_1^1)$ ,  $g(z_2^1) = g(b + xw_2^1) \dots$
  - $g(z^1) = g(xw^1 + b)$
- What is the output of the network?
  - $g(z^3) = g(g(g(xw^1 + b)w^2 + b)w^3 + b)$
- Can represent any continuous function!



# Universal Function Approximation Theorem

**Hornik theorem 1:** Whenever the activation function is *bounded and nonconstant*, then, for any finite measure  $\mu$ , standard multilayer feedforward networks can approximate any function in  $L^p(\mu)$  (the space of all functions on  $R^k$  such that  $\int_{R^k} |f(x)|^p d\mu(x) < \infty$ ) arbitrarily well, provided that sufficiently many hidden units are available.

**Hornik theorem 2:** Whenever the activation function is *continuous, bounded and non-constant*, then, for arbitrary compact subsets  $X \subseteq R^k$ , standard multilayer feedforward networks can approximate any continuous function on  $X$  arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- In words: Given any continuous function  $f(x)$ , if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate  $f(x)$



## Defining a neural net with Numpy

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                          for x, y in zip(sizes[:-1], sizes[1:])]

    def sigmoid(z):
        return 1.0/(1.0+np.exp(-z))

    def feedforward(self, a):
        """Return the output of the network if "a" is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

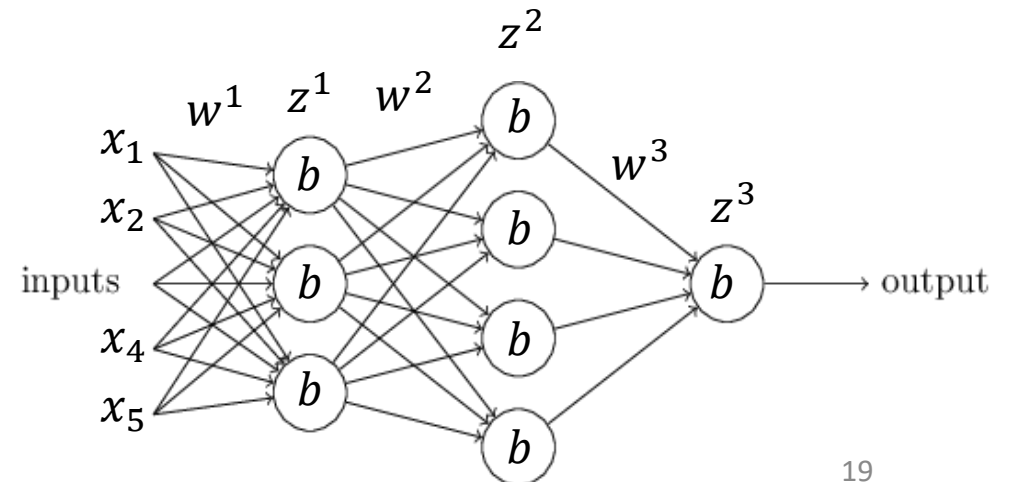
# Gradient decent for a full network

- Step1: compute the loss gradient with respect to  $z^3$

- $\frac{\partial loss}{\partial z^3} = \frac{\partial loss}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3}$

- For  $g=\text{sigmoid}$  and  $\text{loss} = \text{squared}$ :

- $\frac{\partial loss}{\partial z^3} = \left[ \frac{1}{1+e^{-z^3}} - y \right] \frac{1}{1+e^{-z^3}} \left( 1 - \frac{1}{1+e^{-z^3}} \right)$



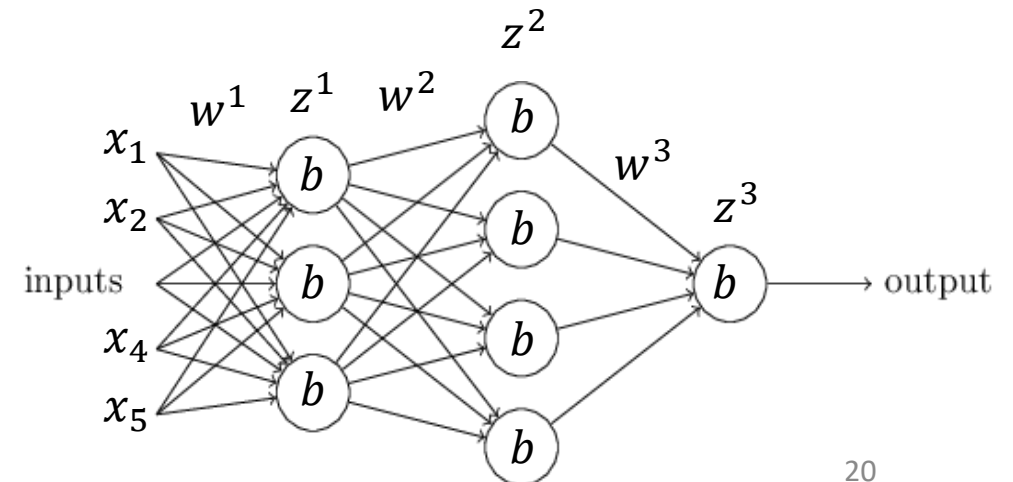
# Gradient decent for a full network

- Step1: compute the loss gradient with respect to  $z^3$

- $\frac{\partial loss}{\partial z^3} = \frac{\partial loss}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3}$

- For  $g$ =sigmoid and loss = squared:

- $\frac{\partial loss}{\partial z^3} = \left[ \frac{1}{1+e^{-z^3}} - y \right] \frac{1}{1+e^{-z^3}} \left( 1 - \frac{1}{1+e^{-z^3}} \right)$



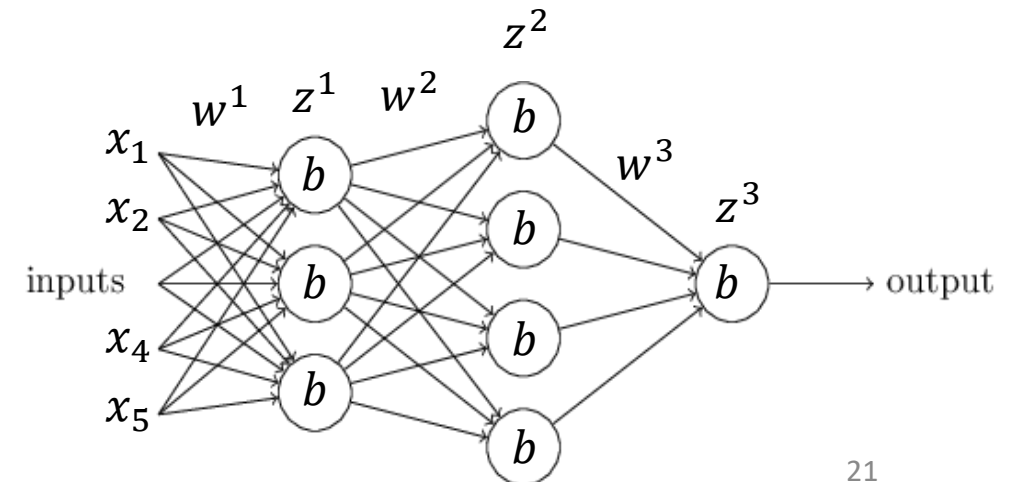
# Gradient decent for a full network

- Step1: compute the loss gradient with respect to  $z^3$

- $\frac{\partial loss}{\partial z^3} = \frac{\partial loss}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3}$

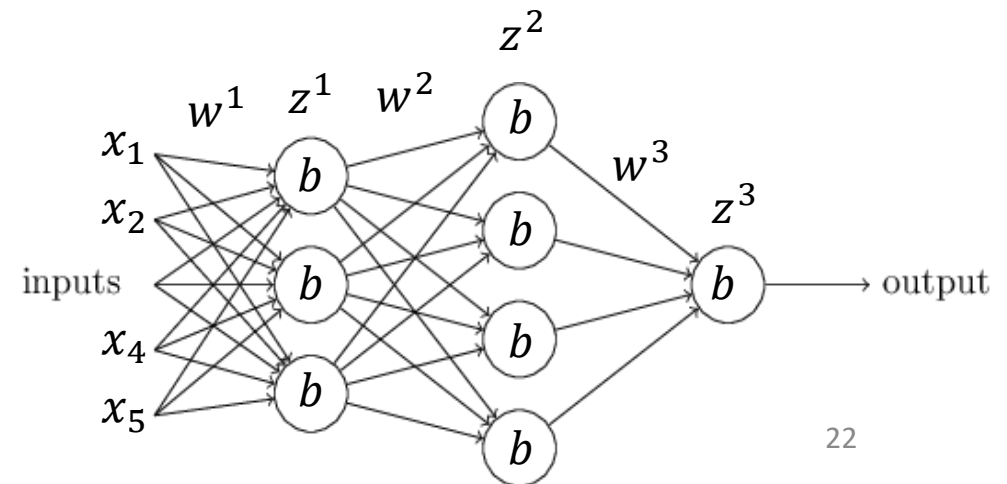
- For  $g$ =sigmoid and loss = squared:

- $\frac{\partial loss}{\partial z^3} = \left[ \frac{1}{1+e^{-z^3}} - y \right] \frac{1}{1+e^{-z^3}} \left( 1 - \frac{1}{1+e^{-z^3}} \right)$



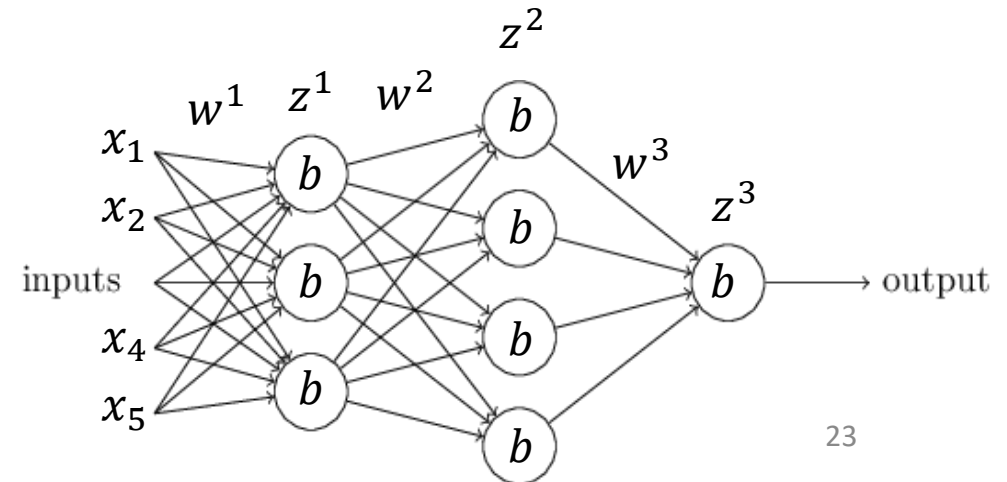
# Gradient decent for a full network

- Step1: compute the loss gradient with respect to  $z^3$ 
  - $\frac{\partial loss}{\partial z^3} = \frac{\partial loss}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3}$
  - For  $g$ =sigmoid and loss = squared:
    - $\frac{\partial loss}{\partial z^3} = \left[ \frac{1}{1+e^{-z^3}} - y \right] \frac{1}{1+e^{-z^3}} \left( 1 - \frac{1}{1+e^{-z^3}} \right)$
- Step2: Compute the loss gradient with respect to  $w^3$  and  $b^3$ 
  - $\frac{\partial loss}{\partial w^3} = \frac{\partial loss}{\partial z^3} \cdot \frac{\partial z^3}{\partial w^3} = \frac{\partial loss}{\partial z^3} \cdot [g(z^2)]$
  - $\frac{\partial loss}{\partial b^3} = \frac{\partial loss}{\partial z^3} \cdot \frac{\partial z^3}{\partial b^3} = \frac{\partial loss}{\partial z^3} \cdot 1$



# Gradient decent for a full network

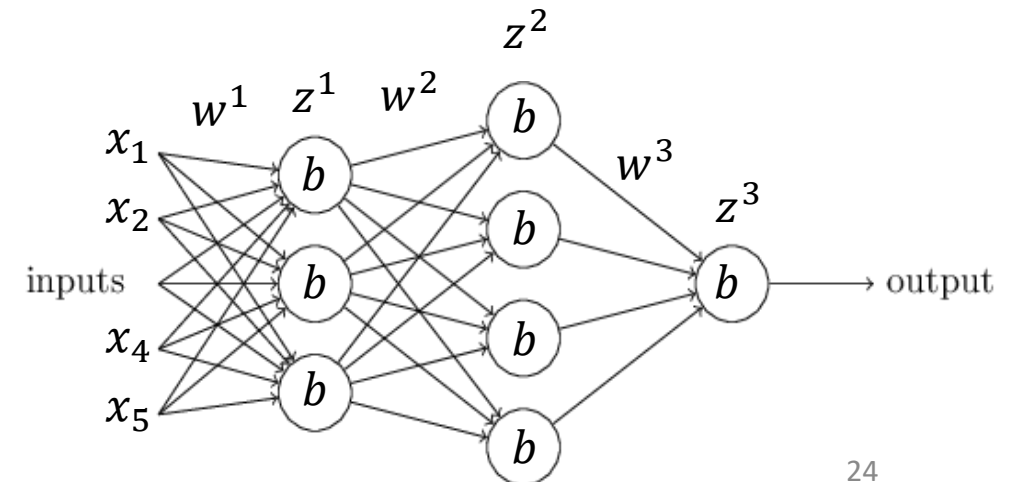
- Step1: compute the loss gradient with respect to  $z^3$ 
  - $\frac{\partial loss}{\partial z^3} = \frac{\partial loss}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3}$
  - For  $g$ =sigmoid and loss = squared:
  - $\frac{\partial loss}{\partial z^3} = \left[ \frac{1}{1+e^{-z^3}} - y \right] \frac{1}{1+e^{-z^3}} \left( 1 - \frac{1}{1+e^{-z^3}} \right)$
- Step2: Compute the loss gradient with respect to  $w^3$  and  $b^3$ 
  - $\frac{\partial loss}{\partial w^3} = \frac{\partial loss}{\partial z^3} \cdot \frac{\partial z^3}{\partial w^3} = \frac{\partial loss}{\partial z^3} \cdot [g(z^2)]$
  - $\frac{\partial loss}{\partial b^3} = \frac{\partial loss}{\partial z^3} \cdot \frac{\partial z^3}{\partial b^3} = \frac{\partial loss}{\partial z^3} \cdot 1$
- Update  $w^3, b^3$  such that loss is reduced
  - $w^3 = w^3 - \alpha \nabla_{w^3} loss$



# Gradient decent for a full network

- Next: compute the loss gradient with respect to  $z^2$

$$\bullet \frac{\partial \text{loss}}{\partial z^2} = \frac{\partial \text{loss}}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3} \cdot \frac{\partial z^3}{\partial g(z^2)} \cdot \frac{\partial g(z^2)}{\partial z^2}$$



# Gradient decent for a full network

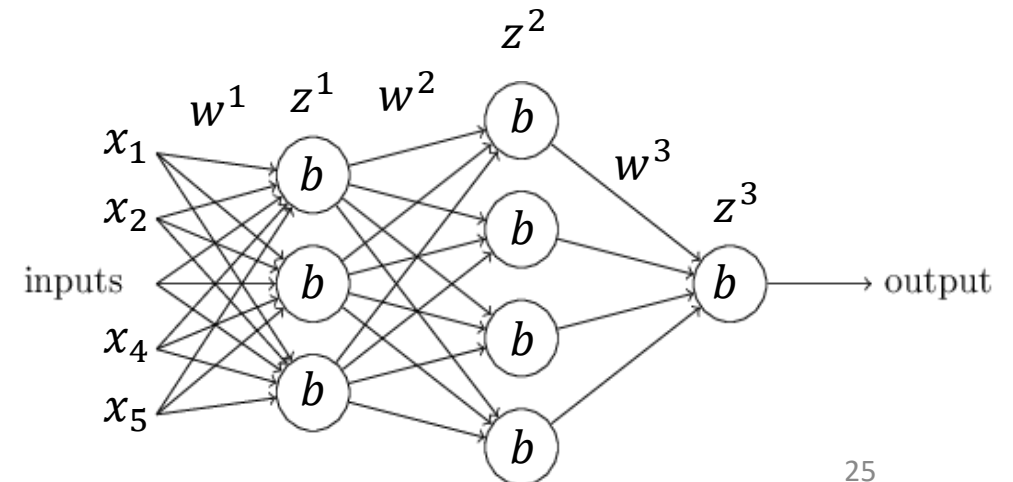
- Next: compute the loss gradient with respect to  $z^2$

$$\bullet \frac{\partial loss}{\partial z^2} = \frac{\partial loss}{\partial g(z^3)} \cdot \frac{\partial g(z^3)}{\partial z^3} \cdot \frac{\partial z^3}{\partial g(z^2)} \cdot \frac{\partial g(z^2)}{\partial z^2}$$

We know this!

We computed it in the previous step

$$= \frac{\partial loss}{\partial z^3}$$





# Gradient decent for a full network

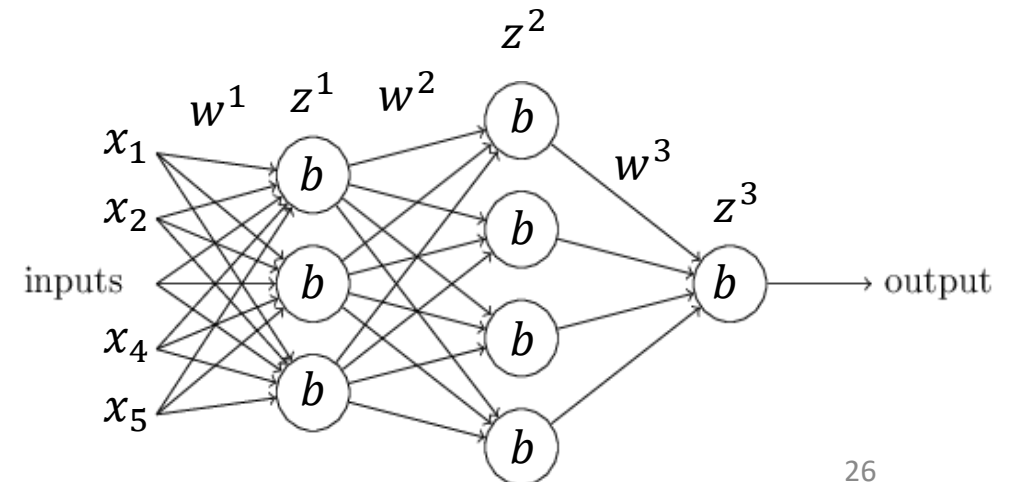
- Next: compute the loss gradient with respect to  $z^2$

$$\bullet \frac{\partial loss}{\partial z^2} = \frac{\partial loss}{\partial z^3} \cdot \frac{\partial z^3}{\partial g(z^2)} \cdot \frac{\partial g(z^2)}{\partial z^2}$$

We know this!

We computed it in the previous step

$$= \frac{\partial loss}{\partial z^3}$$



# Gradient decent for a full network

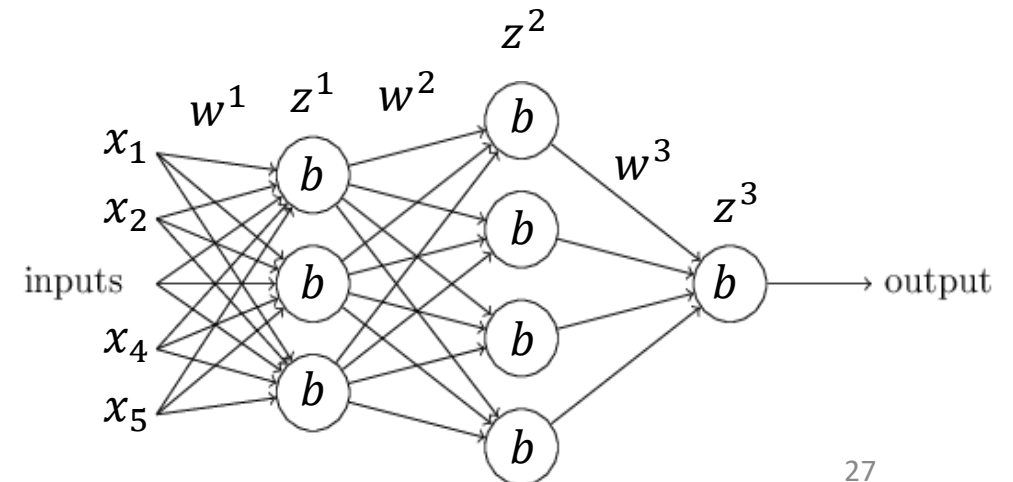
- In general: compute the loss gradient with respect to  $z^i$

$$\bullet \frac{\partial loss}{\partial z^i} = \frac{\partial loss}{\partial z^{i+1}} \cdot \frac{\partial z^{i+1}}{\partial g(z^i)} \cdot \frac{\partial g(z^i)}{\partial z^i}$$

We know this!

We computed it in the previous step

$$= \frac{\partial loss}{\partial z^{i+1}}$$



# Back propagation

- In general: compute the loss gradient with respect to  $z^i$

- $\frac{\partial loss}{\partial z^i} = \frac{\partial loss}{\partial z^{i+1}} \cdot \frac{\partial z^{i+1}}{\partial g(z^i)} \cdot \frac{\partial g(z^i)}{\partial z^i}$

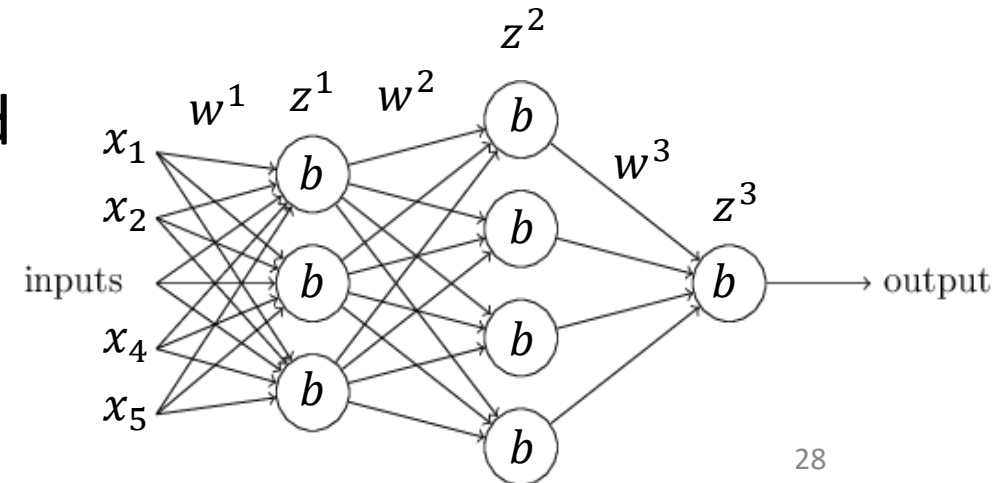
- Compute the loss gradient with respect to  $w^i$  and  $b^i$

- $\frac{\partial loss}{\partial w^i} = \frac{\partial loss}{\partial z^i} \cdot \frac{\partial z^i}{\partial w^i} = \frac{\partial loss}{\partial z^i} \cdot g(z^{i-1})$

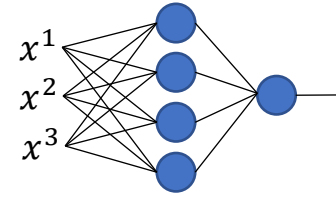
- $\frac{\partial loss}{\partial b^i} = \frac{\partial loss}{\partial z^i} \cdot \frac{\partial z^i}{\partial b^i} = \frac{\partial loss}{\partial z^i} \cdot 1$

- Later: update  $w, b$  such that loss is reduced

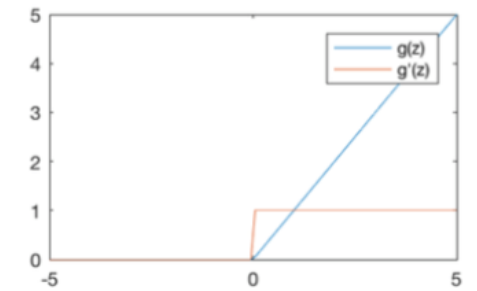
- $w^i = w^i - \alpha \nabla_{w^i} loss$



# Numerical example



Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

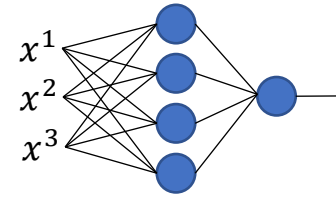
- One training sample:  $x = [1, 2, 3]^T$   $y = 20$
- Network =  $[3, 4, 1]$  // [input, hidden layer, output]
- Hidden layer uses a ReLU activation function

$$\text{Init: } w^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad b^1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad w^2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

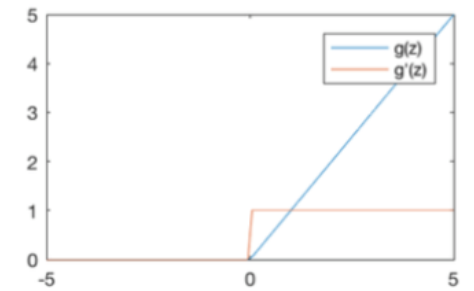
Uniform values are a bad idea in general

- $\hat{y} = z^2 = g(w^1 x + b^1)^T w^2 = g([6 \ 6 \ 6 \ 6]) \cdot w^2 = 24$
- Squared loss =  $0.5(y - \hat{y})^2 = 8$

# Numerical example



Rectified Linear Unit (ReLU)



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

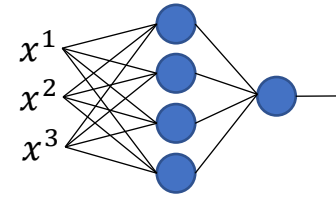
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Init:  $w^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$   $b^1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$   $w^2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$
- $\hat{y} = g(w^1 x + b^1)^\top w^2 = g([6 \ 6 \ 6 \ 6]) w^2 = 24$
- Squared loss =  $0.5(y - z^2)^2 = 8$ ,  $\frac{\partial \text{loss}}{\partial z^2} = -1 \cdot (20 - 24) = 4$
- $\frac{\partial \text{loss}}{\partial w^2} = \frac{\partial \text{loss}}{\partial z^2} \cdot \frac{\partial z^2}{\partial w^2} = 4 \cdot g(z^1) = [24 \ 24 \ 24 \ 24]$
- $\frac{\partial \text{loss}}{\partial z^1} = \frac{\partial \text{loss}}{\partial z^2} \cdot \frac{\partial z^2}{\partial g(z^1)} \cdot \frac{\partial g(z^1)}{\partial z^1} = 4 \cdot w^2 \cdot [1 \ 1 \ 1 \ 1] = [4 \ 4 \ 4 \ 4]$
- $\frac{\partial \text{loss}}{\partial w^1} = \text{For each row } i \left[ \frac{\partial \text{loss}}{\partial z_i^1} \cdot \frac{\partial z_i^1}{\partial w_i^1} \right] = [4x \ 4x \ 4x \ 4x]$

For all  $i$ ,  $z_i^1 > 0$

3 by 4 matrix

# Numerical example



- One training sample:  $x = [1, 2, 3]^T$   $y = 20$
- In order to reduce the loss  $0.5(y - \hat{y})^2 = 0.5(20 - 24)^2 = 8$
- Update:  $w = w - \alpha \nabla_w \text{loss}$
- $w^2 = [1 \ 1 \ 1 \ 1] - 0.01[24 \ 24 \ 24 \ 24] = [0.76 \ 0.76 \ 0.76 \ 0.76]$
- $w^1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} - 0.01 \begin{bmatrix} 4 & 8 & 12 \\ 4 & 8 & 12 \\ 4 & 8 & 12 \\ 4 & 8 & 12 \end{bmatrix} = \begin{bmatrix} 0.96 & 0.92 & 0.88 \\ 0.96 & 0.92 & 0.88 \\ 0.96 & 0.92 & 0.88 \\ 0.96 & 0.92 & 0.88 \end{bmatrix}$
- After update:
  - $\hat{y} = g(xw^1 + b^1)w^2 = g([5.44 \ 5.44 \ 5.44 \ 5.44]^T)w^2 = 16.54$
  - New loss  $0.5(y - \hat{y})^2 = 0.5(20 - 16.54)^2 = 6$

```
def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x. ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
```

```
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))
```

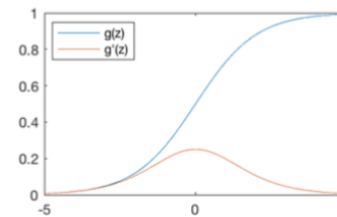
```
def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

```
# backward pass
delta = self.cost_derivative(activations[-1], y) * \
    sigmoid_prime(zs[-1])
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
# Note that the variable l in the loop below is used a little
# differently to the notation in Chapter 2 of the book. Here,
# l = 1 means the last layer of neurons, l = 2 is the
# second-last layer, and so on. It's a renumbering of the
# scheme in the book, used here to take advantage of the fact
# that Python can use negative indices in lists.
for l in xrange(2, self.num_layers):
    z = zs[-l]
    sp = sigmoid_prime(z)
    delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
    nabla_b[-l] = delta
    nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
return (nabla_b, nabla_w)
```

# Activation functions

- Each layer can be assigned a different activation function
- The output layer should be assigned an activation that is appropriate to the desired output range (can be linear)
- What is the appropriate activation:
  - If output represents probability?
  - If output represents steering control from  $[-1,1]$ ?
  - If output represents a Q value?

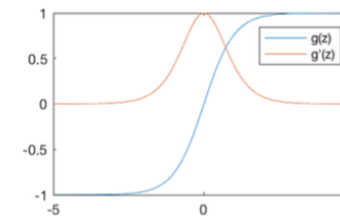
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

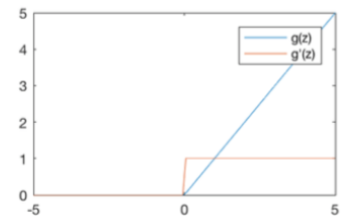
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



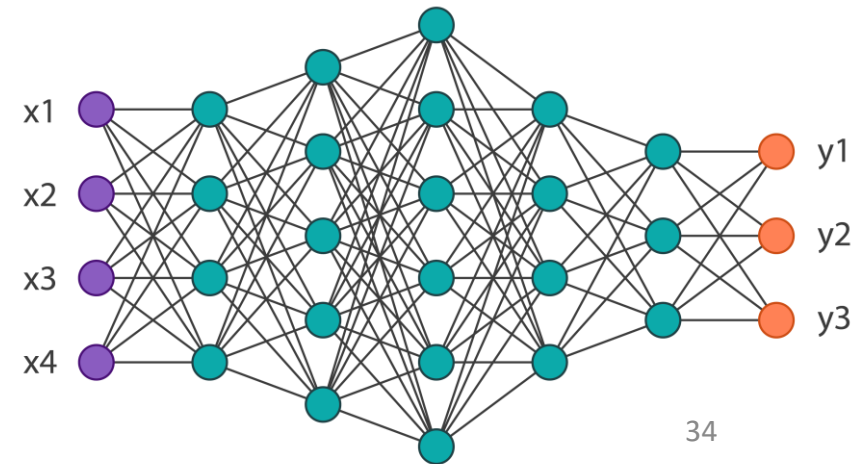
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$



# Output vector

- Neural nets can be generalized to produce an output vector
- For instance, it is generally more efficient (why?) to assign an output value for each action when representing:
  - Q-values:  $Q(s, a) \rightarrow \mathbb{R} \iff Q(s) \rightarrow \mathbb{R}^{\mathcal{A}}$
  - A soft policy (the actor):  $\pi(s, a) \rightarrow [0,1] \iff \pi(s) \rightarrow [0,1]^{\mathcal{A}}$
  - For soft policy, what activation function should we use at the final layer?
    - Sigmoid, TanH, Relu, Softmax
  - What loss function should we use?



# Common loss functions

- Regression Loss Functions
  - When required to fit a continuous function
    - Absolute Error Loss
    - Squared Error Loss
    - Huber Loss
- Multi-class Classification Loss Functions
  - When required to predict one of many classes
    - Cross Entropy Loss (usually over a Softmax activation for the output layer)
- Distribution approximation
  - When required to fit a distribution
    - Kullback-Liebler (KL) Divergence

# Squared Error Loss

$$L = (y - f(x))^2$$

- also known as **L2 Loss**, is the square of the difference between the actual and the predicted values
- The Squared Error loss function penalizes the model for making large errors by squaring them
- Change in loss grows with the prediction error i.e.,  $\frac{dL}{df(x)}$  grows with  $(f(x) - y)$
- This property makes the Squared Error Loss function less robust to outliers. Therefore, ***it should not be used if our data is prone to many outliers***

# Absolute Error Loss

$$L = |y - f(x)|$$

- Also known as the **L1 loss**
- More robust to outliers as compared to Squared Error Loss
- Not differentiable at  $x=0$
- The derivative is not continuous around zero which is problematic for convergence

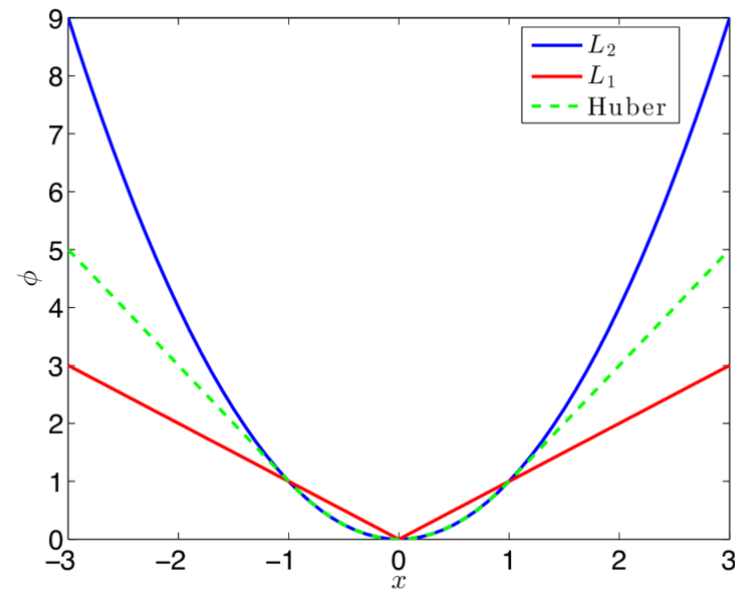
# Huber Loss

$$L_{\delta} = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Quadratic

Linear

- Quadratic for smaller errors and linear otherwise
- Requires tuning of the *delta* hyper-parameter
- More robust to outliers as compared to Squared Error Loss
- Differentiable and continuous



# Cross Entropy Loss

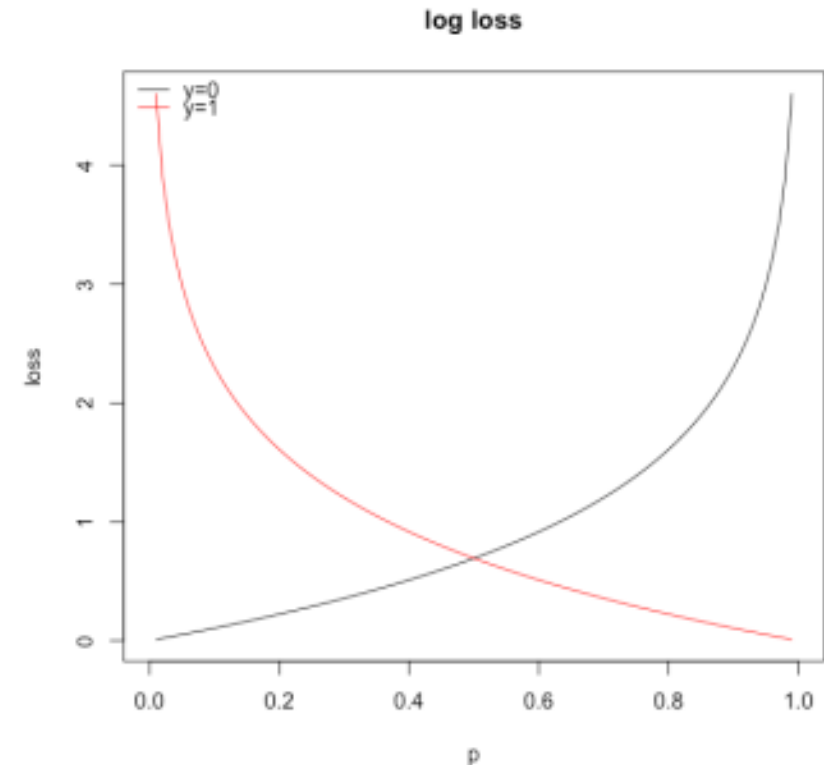
- also called **Log-Loss**
- Penalizes large errors in predicted distribution
- Goes in tandem with the softmax function
  - See Lecture “2Multi\_armed\_bandits.pptx”, Slide 34

$$L(X_i, Y_i) = - \sum_{j=1}^c y_{ij} * \log(p_{ij})$$

where  $Y_i$  is one – hot encoded target vector  $(y_{i1}, y_{i2}, \dots, y_{ic})$ ,

$$y_{ij} = \begin{cases} 1, & \text{if } i_{th} \text{ element is in class } j \\ 0, & \text{otherwise} \end{cases}$$

$p_{ij} = f(X_i)$  = Probability that  $i_{th}$  element is in class  $j$



# Kullback-Liebler (KL) Divergence

- Measures similarity between two distributions
- A KL-divergence of zero indicates that the distributions are identical

*For probability distributions  $P$  and  $Q$ ,  
KL – Divergence of  $P$  from  $Q$  is given by*

$$\begin{aligned} D_{KL}(P||Q) &= \begin{cases} -\sum_x P(x) \cdot \log \frac{Q(x)}{P(x)} = \sum_x P(x) \cdot \log \frac{P(x)}{Q(x)}, & \text{for discrete distributions} \\ -\int P(x) \cdot \log \frac{Q(x)}{P(x)} \cdot dx = \int P(x) \cdot \log \frac{P(x)}{Q(x)} \cdot dx, & \text{for continuous distributions} \end{cases} \\ &= \text{Expectation of logarithmic difference between } P \text{ and } Q \text{ with respect to } P \end{aligned}$$

- Relative entropy of  $P$  with respect to  $Q$ ,  $D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P)$
- Measuring dissimilarity (not distance) between two soft policies

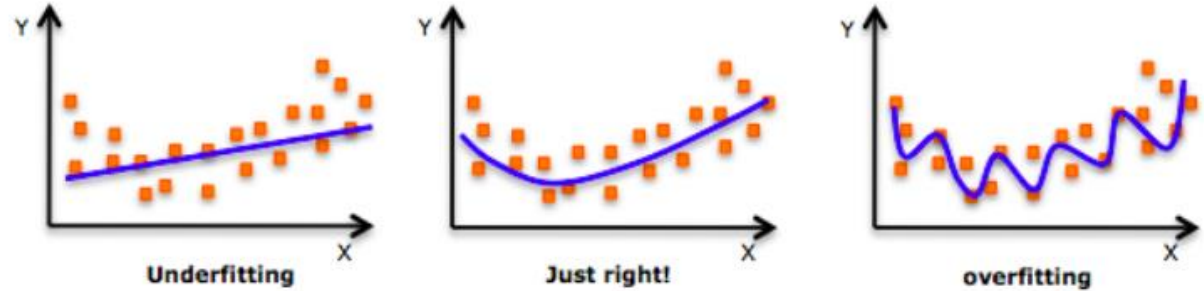
# Overfitting

- We can now define a neural network with non-linear activation and set an appropriate loss function
- We know how to compute the gradient of the loss with respect to the tunable parameters
- Can we perform successive gradient decent steps until  $\text{loss}=0$  ?
  - Universal Function Approximation Theorem tells us that it is possible to reach  $\text{loss}=0$
  - However, parameter assignment space is nonconvex with respect to the loss
    1. We might converge on a local optimum
    2. In any case, that is probably not a good idea. Doing so usually implies memorizing the training data (overfitting). In such cases predicting unseen observations is usually inaccurate

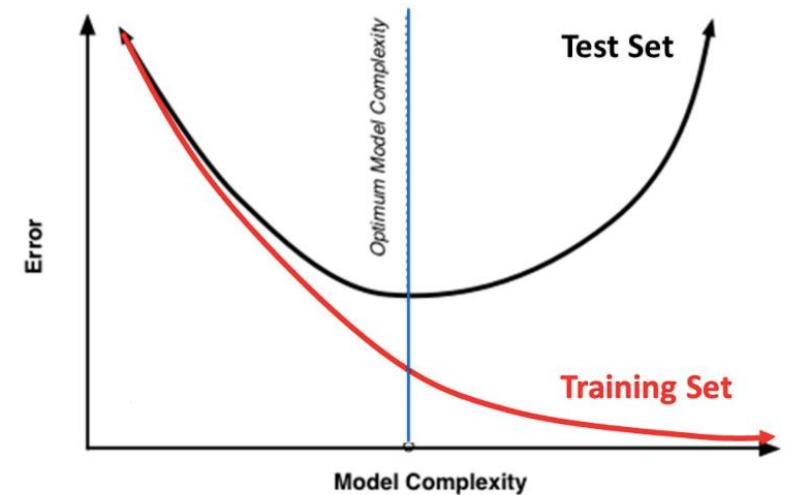


# Regularization

1. Early stopping
2. Penalize complex model
  - Add model complexity penalty to the loss function
    - L1 regularization
    - L2 regularization
3. Enforce model simplicity
  - Dropout

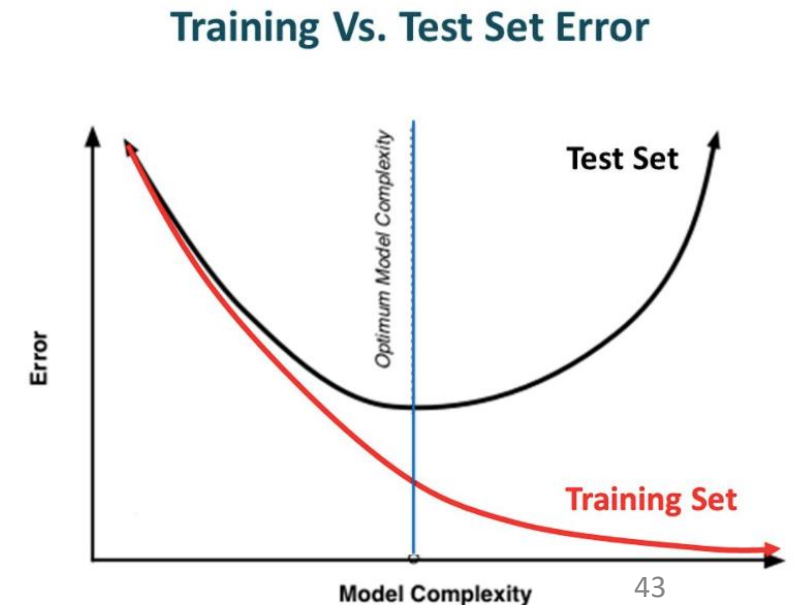


Training Vs. Test Set Error



# Early stopping

- Usually, more training = higher accuracy on training set
- At some point the model starts memorizing the training examples
- In order to avoid such overfitting
  - Keep track of test set accuracy
  - Stop training when it declines



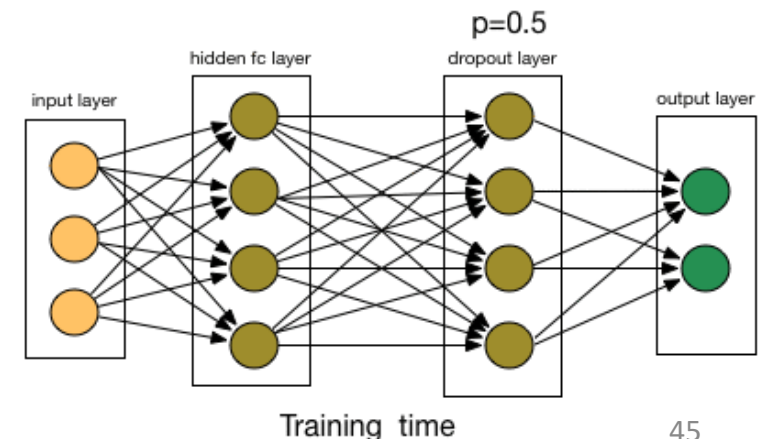
# L1 & L2 regularization

- Assumes that a neural network with a smaller weight norm leads to simpler models
- Therefore, it will also reduce overfitting
- Add model complexity penalty to the loss function
- $penalty_{L1} = \frac{\lambda}{2n} \sum_W |w|$  ,  $penalty_{L2} = \frac{\lambda}{2n} \sum_W w^2$ 
  - $\lambda > 0$  is the *regularization parameter*,  $n$  is the size of the training set,  $W$  is the set of all weights in the network
- How will adding these penalties affect the weight gradient?

\*L2 regularization has a deeper theoretical meaning which we won't cover

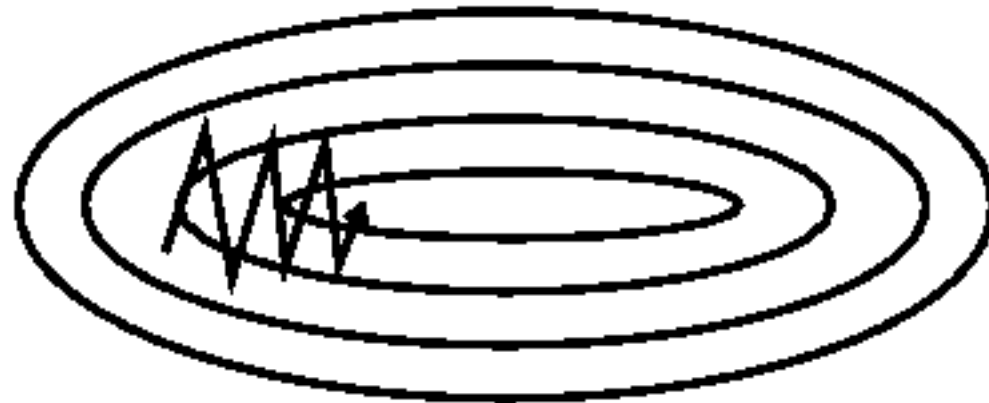
# Dropout

- Common regularization technique (these days)
- At every iteration, randomly (and temporarily) remove some nodes along with their incoming and outgoing connections
  - Train a minibatch of observation on the augmented network
- Each iteration trains a different subset of the network
- Dropout probability is a hyper parameter



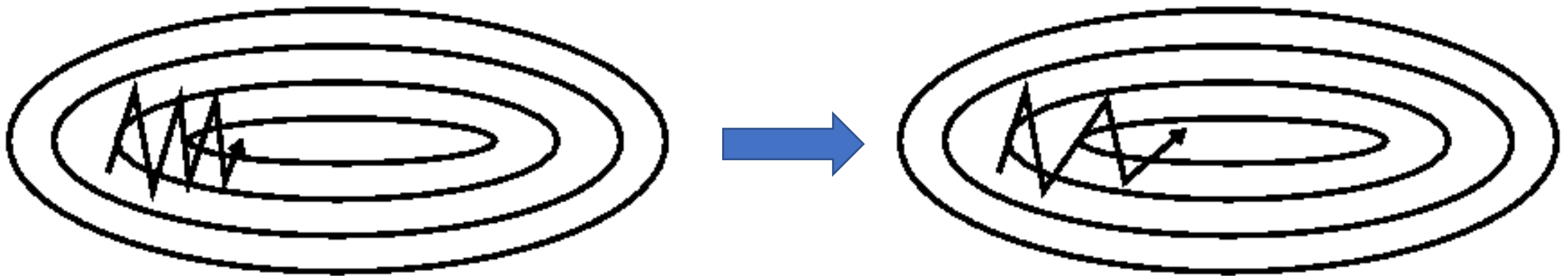
# Adaptive learning rates

- SGD has trouble navigating ravines, i.e., areas where the surface curves much more steeply in one dimension than in another, which are common around local optima
- In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum



# Momentum

- Keep an exponential weighted average of the gradient over successive iterations
  - $v_t = \beta v_{t-1} + (1 - \beta) \nabla_{\theta} \text{loss}$
- Update the model using SGD towards  $-v_t$ 
  - $\theta = \theta - \alpha v_t$

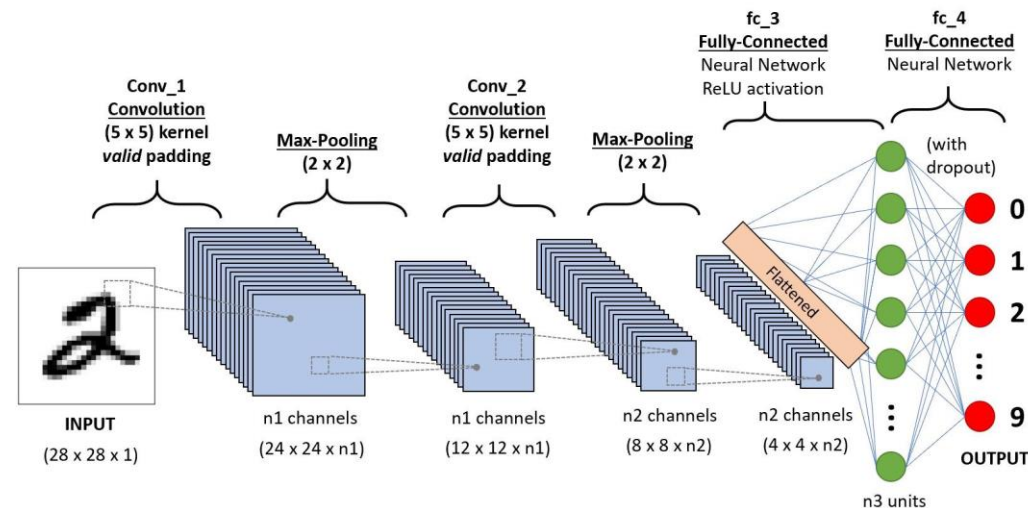


# Adaptive Moment Estimation (ADAM)

- Keeps an exponentially decaying average of past gradients, as in vanilla momentum
- Keeps an exponentially decaying average of past squared gradients
- Computes adaptive learning rates for each parameter
  - Invariant to the magnitude of the gradient, which helps a lot when going through areas with tiny gradients
- Commonly used these days
  - $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )
  - $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)
  - $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)
  - $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)
  - $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)
  - $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

# Conv layers

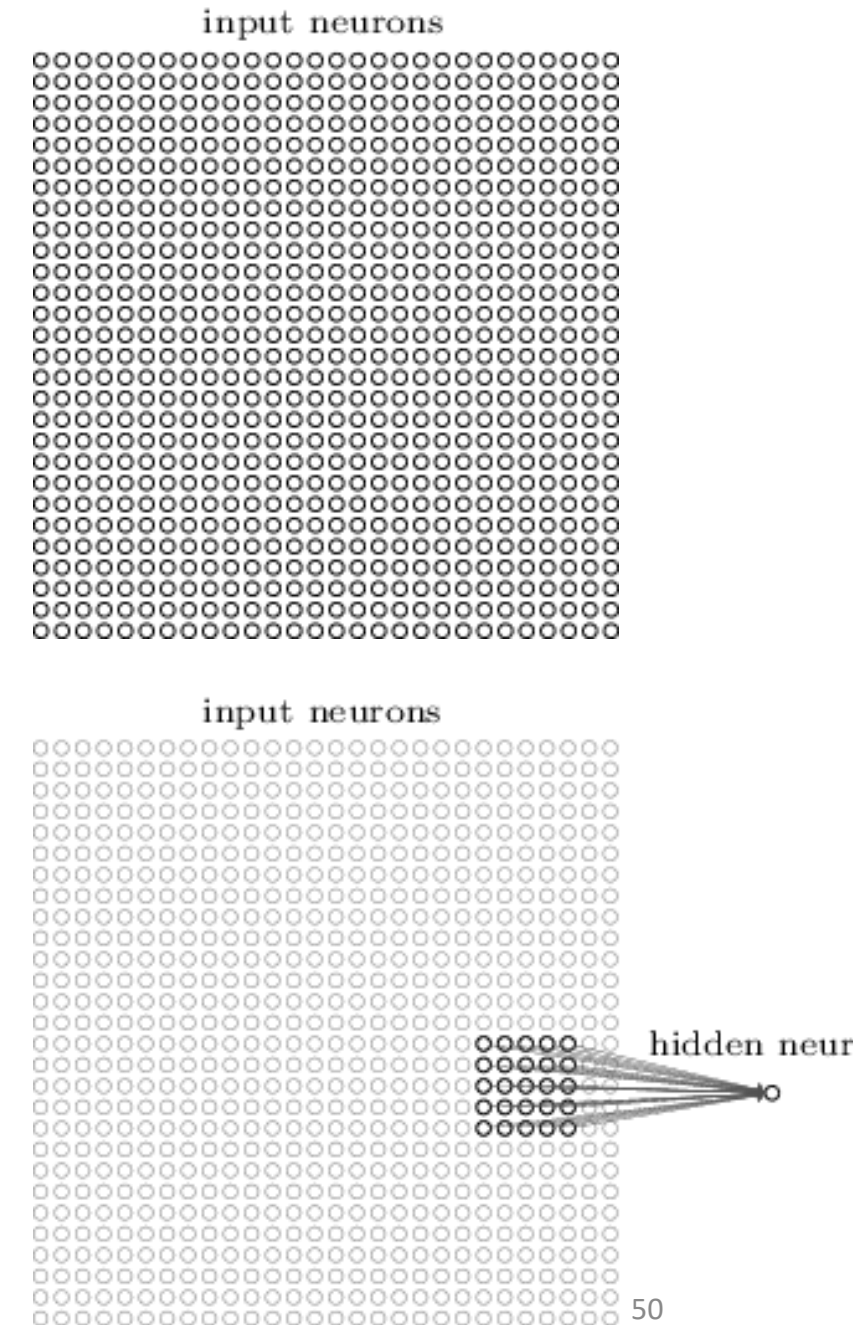
- Spatial structure can be inferred for some inputs (e.g., images)
- We would like to take advantage of the spatial structure in such cases
- CNN use three basic ideas: *local receptive fields*, *shared weights*, and *pooling*





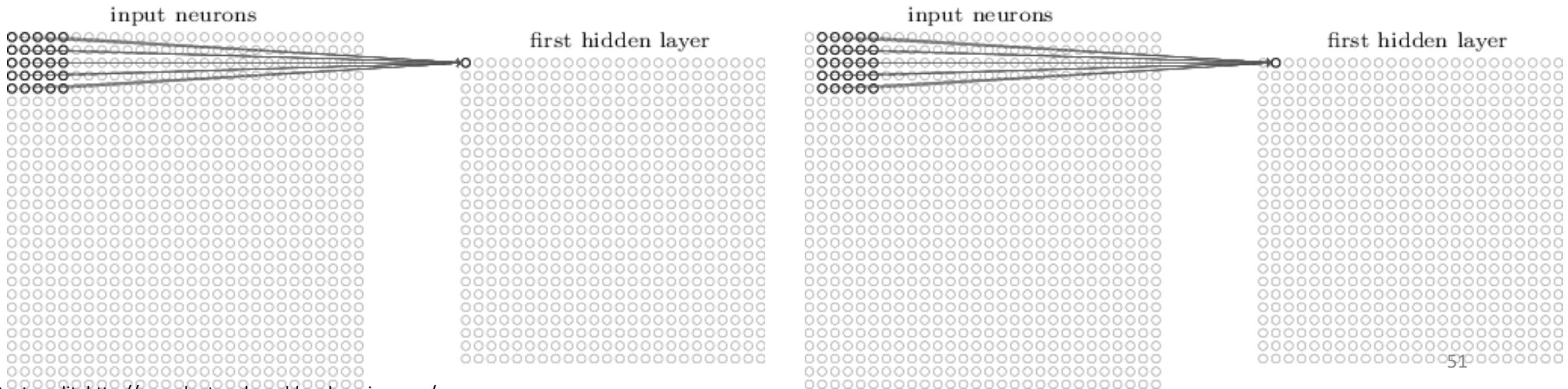
# Local receptive fields

- Instead of connecting all inputs to every hidden neuron at  $l_1$ , connect only a subset of spatially connected input neurons
- That region in the input image is called the *local receptive field* for the hidden neuron
- You can think of that particular hidden neuron as learning to analyze its particular local receptive field



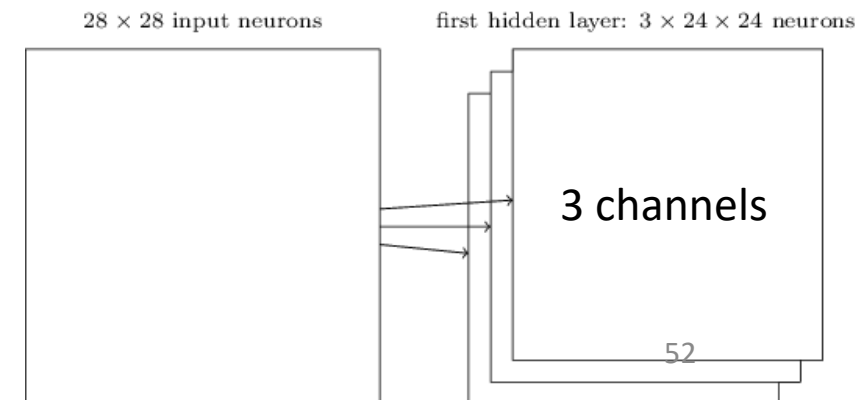
# Local receptive fields

- Start with a local receptive field in the top-left corner
- Then slide the local receptive field over by one pixel
- if we have a  $28 \times 28$  input image, and  $5 \times 5$  local receptive fields, then there will be  $24 \times 24$  neurons in the hidden layer



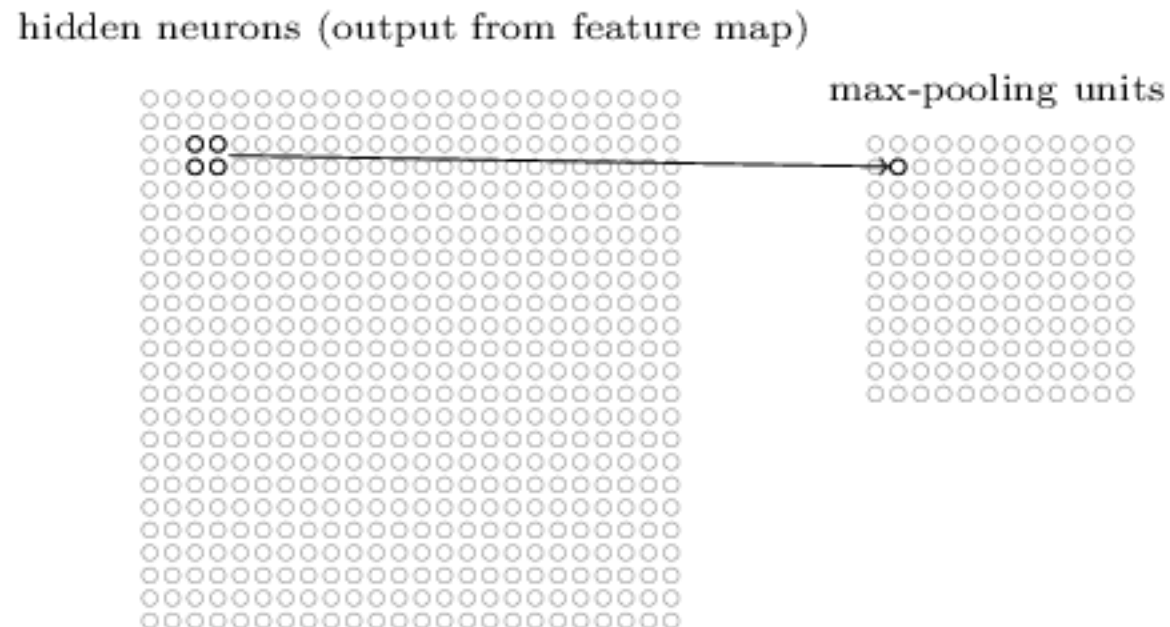
# Shared weights and biases

- Use the *same* weights and bias for each hidden neuron in the conv layer
- This means that all the neurons in the first hidden layer detect the same feature
- What if we want to identify more than one feature?
  - Affiliate each local receptive field with  $n$  hidden neurons (channels)
  - Each feature has its own shared weights and biases across all local fields



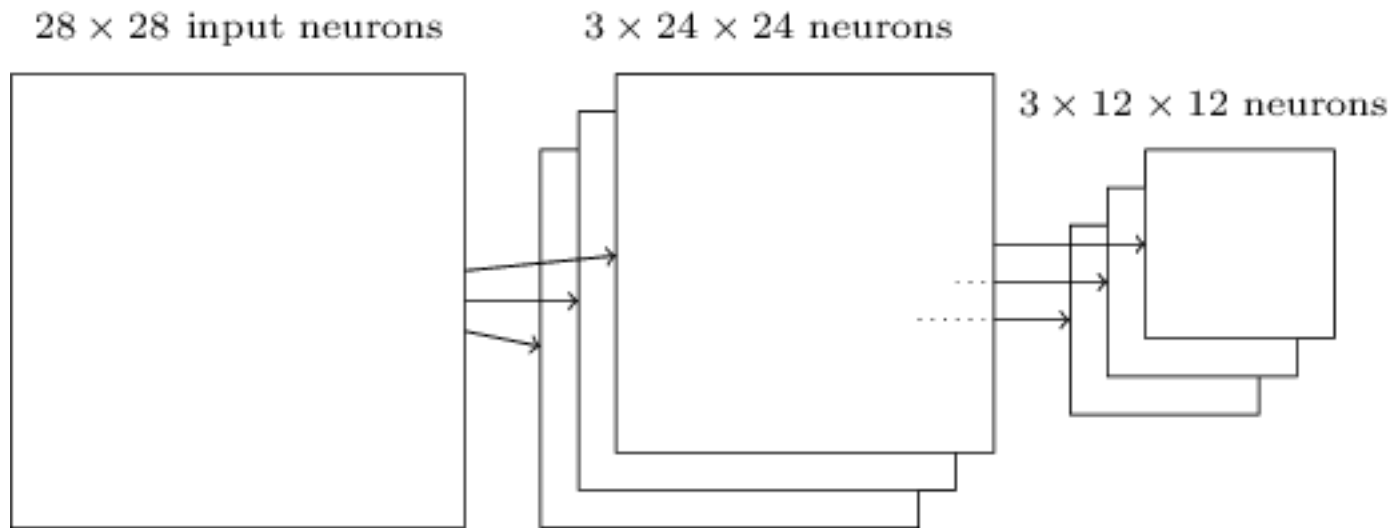
# Pooling layers

- Pooling layers are usually used immediately after convolutional layers
- Creates a condensed feature map
- E.g., max-pooling outputs the maximum activation in an input region



# Pooling layers

- We apply pooling to each feature map separately
- Finally, don't forget to add at least one fully connected layer



# Keras (Python library based on TensorFlow)

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1),
                activation='relu',
                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Create a CNN with 5 hidden layers:

1. 32 channels (hidden features) over 3×3 local receptive fields that slide 1 pixel from each other in each direction.
2. A second conv layer connected to the first one. Further extraction of local features.
3. Pooling layer with dropout regularization
4. A fully connected layer with dropout regulation
5. Finally a softmax layer

The loss function is defined to be the cross-entropy loss. The SGD optimizer is ADAM. The metric for evaluation is total accuracy on the test set.

# Keras: train

```
model.fit(x_train, y_train,  
         batch_size=batch_size,  
         epochs=epochs,  
         validation_data=(x_test, y_test))
```

Predication:

```
model.predict(x)  
model.predict_on_batch(x)
```

# Pytorch

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.drop_out = nn.Dropout(0.5)
        self.fc1 = nn.Linear(7 * 7 * 64, 1000)
        self.fc2 = nn.Linear(1000, 10)
```

**Conv1:** 1 input channel (the image), 32 output channels (unique local features), 5x5 conv filters, padding =2 such that the output will be 28x28, ReLU activation, max pool layer over 2x2 kernels.

**Conv2:** similar to conv1 with 32 input channels and 64 output channels each is a 7x7 feature matrix.

**Drop\_out:** dropout regularization. Removes every value in a specific layer with 0.5 probability when applied.

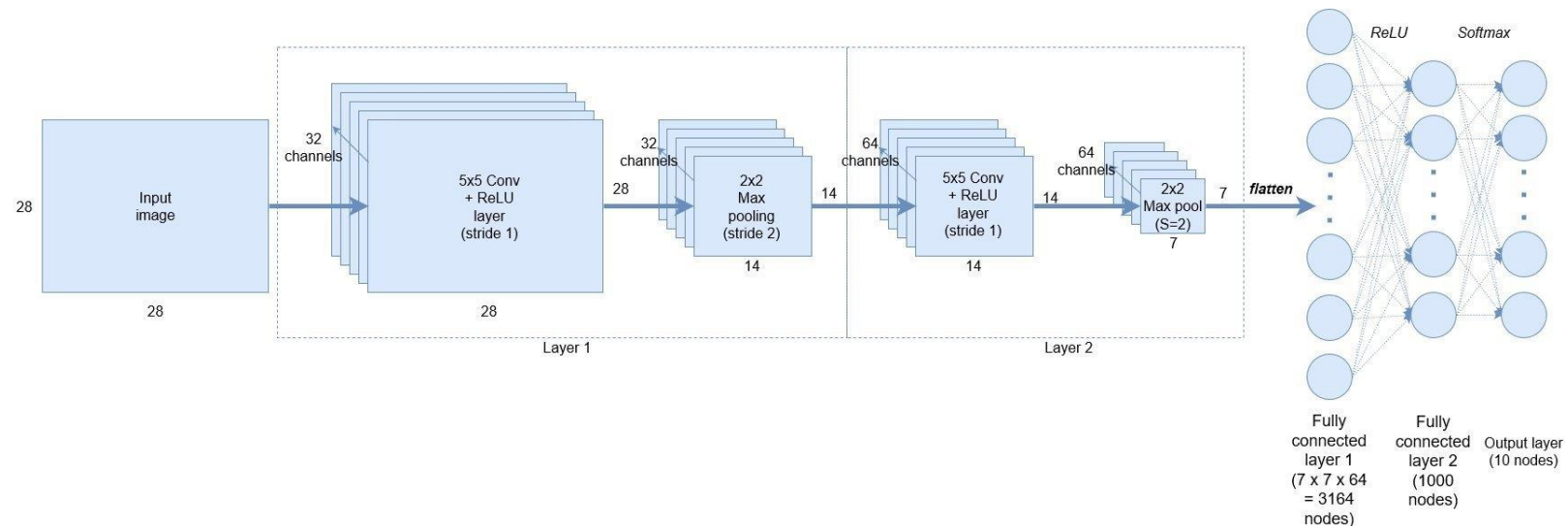
**Fully connected (fc1):** 7\*7\*64 input size and 1000 output size. (fc2 is similar)



# Pytorch: connect the layers (also your prediction function)

```
def forward(self, x):  
    out = self.layer1(x)  
    out = self.layer2(out)  
    out = out.reshape(out.size(0), -1)  
    out = self.drop_out(out)  
    out = self.fc1(out)  
    out = self.fc2(out)  
    return out
```

Define the data flow along the network:  
conv1 -> max pool 1 -> conv2 -> max pool 2 ->  
flatten 64\*7\*7 -> 1 vector -> drop some values  
(to avoid overfitting) -> 2 fully connected  
layers.



# Pytorch: train

```
model = ConvNet()
```

```
# Loss and optimizer
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
# Run the forward pass
```

```
outputs = model(images)
```

```
loss = criterion(outputs, labels)
```

```
# Backprop and perform Adam optimisation
```

```
optimizer.zero_grad()
```

```
loss.backward()
```

```
optimizer.step()
```

the CrossEntropyLoss function combines both a SoftMax activation with the log loss function. Set the Adam optimizer over all tunable parameters.

Get the outputs for a batch of images and compute the loss with respect to the labels. Next, init all gradients to zero, compute new gradients (backpropagate), update parameters based on new gradients.

# Chat-GPT

- “Define a CNN with Pytorch”

Certainly! Below is an example of a simple Convolutional Neural Network (CNN) defined using PyTorch. This CNN consists of three convolutional layers followed by two fully connected layers. It's a basic architecture that you can further customize based on your specific task and dataset.

```
```python
import torch
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        super(SimpleCNN, self).__init__()

        # First Convolutional Layer
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Second Convolutional Layer
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Third Convolutional Layer
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully Connected Layers
        self.fc1 = nn.Linear(64 * 4 * 4, 128) # Adjust input size based on your image dimensions
        self.relu4 = nn.ReLU()
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.pool3(self.relu3(self.conv3(x)))
        x = x.view(x.size(0), -1) # Flatten the output for the fully connected layers
        x = self.relu4(self.fc1(x))
        x = self.fc2(x)
        return x

# Create an instance of the SimpleCNN model
num_classes = 10 # Change this to the number of classes in your classification task
model = SimpleCNN(num_classes)

# Print the model architecture
print(model)
```
```

In this code, we define a `SimpleCNN` class with three convolutional layers, each followed by a ReLU activation and max-pooling operation. Then, there are two fully connected layers for classification. You should adapt the number of channels, kernel sizes, and fully connected layer sizes to match your specific problem. Additionally, adjust the input size of the first fully connected layer to match your image dimensions.

# Chat-GPT

- “Add an ADAM optimizer using the Cross Entropy Loss function. Define the training function.”

To add an optimizer using the Cross Entropy Loss function and define a training function for the CNN, you can follow this example. We'll use the Adam optimizer and train the model on a sample dataset. Make sure you have a dataset and dataloaders for your specific task.

```
```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define the CNN architecture as previously shown
class SimpleCNN(nn.Module):
    # ... (same as before)

# Create an instance of the SimpleCNN model
num_classes = 10 # Change this to the number of classes in your dataset
model = SimpleCNN(num_classes)

# Define the loss function (CrossEntropyLoss) and optimizer (Adam)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001) # You can adjust the learning rate as needed

# Define the training function
def train_model(model, train_loader, num_epochs):
    for epoch in range(num_epochs):
        model.train() # Set the model to training mode

        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad() # Zero the gradients

            # Forward pass
            outputs = model(inputs)

            # Compute the loss
            loss = criterion(outputs, labels)

            # Backpropagation and optimization
            loss.backward()
            optimizer.step()

        running_loss += loss.item()

    # Print the average loss for this epoch
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader)}")

# Example usage:
# Assuming you have a DataLoader named 'train_loader' and 'num_epochs' is the number of training epochs
# train_model(model, train_loader, num_epochs)
```
```

In this code, we added the loss function (`nn.CrossEntropyLoss()`) and defined the Adam optimizer (`optim.Adam()`) with a learning rate of 0.001. The `train\_model` function takes the model, training data loader (`train\_loader`), and the number of training epochs (`num\_epochs`) as input and performs the training loop.

You should replace `train\_loader` with your actual training data loader and specify the number of epochs based on your training requirements. After defining and running this training function, your CNN will be trained using the specified optimizer and loss function.

# What did we learn?

- DNN are nonlinear functions with many tunable parameters (weights and biases)
- Using the backpropagation algorithm, we can easily compute the partial derivative in the approximation error (loss) w.r.t the tunable parameters ( $w, b$ )
- Using (stochastic) gradient decent steps, we optimize our model such that the error is reduced (in expectancy)
- DNN can approximate any continuous function to arbitrary precision
  - Choose an appropriate loss function
  - Avoid overfitting! Regulate
  - Hyper parameter tuning: network's structure (depth, width, convolution layers, pooling layers, fully connected layers), momentum, activation functions, Regularization technique.

# What next?

- **Lecture:** RL Control with approximation
- **Assignments:**
  - Tabular Q-Learning, by Monday Oct. 11, EOD
  - SARSA, by Monday Oct. 11, EOD
  - Q-Learning with Approximation, by Wednesday, Oct. 11, EOD
- **Quiz (on Canvas):**
  - Value approximation, by Tuesday Oct. 3, EOD
  - Deep Neural Nets, by Tuesday Oct. 12, EOD
- **Project:**
  - Literature survey, by Monday, November 6 EOD