

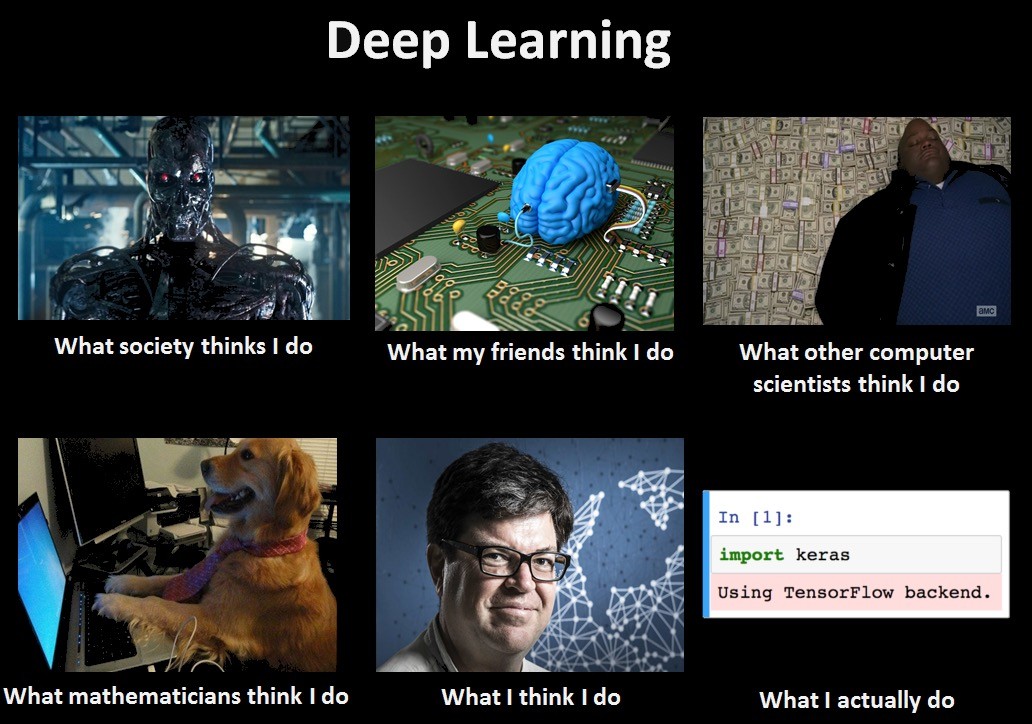
CAP5768/IDC4140:

Introduction to Data Science

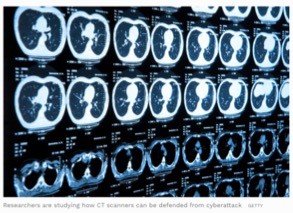
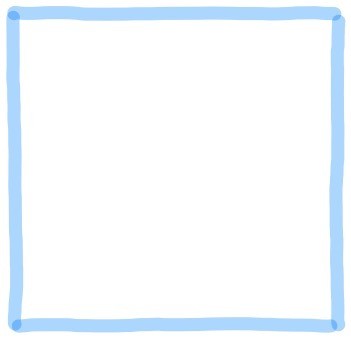
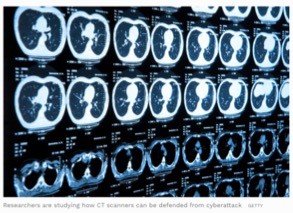
Guang Wang Department of Computer Science

Florida State University

Neural Network

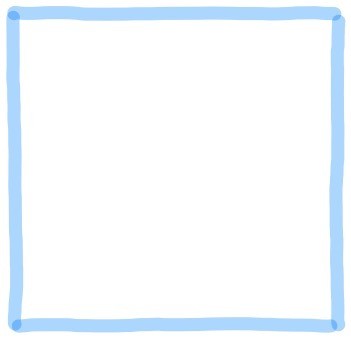


Stopping Cyberattacks



Detecting tampering with the \*5678tgrepcheck1%%&$ diagnostic images, or quietly upped the radiation levels.

Skin Conditions



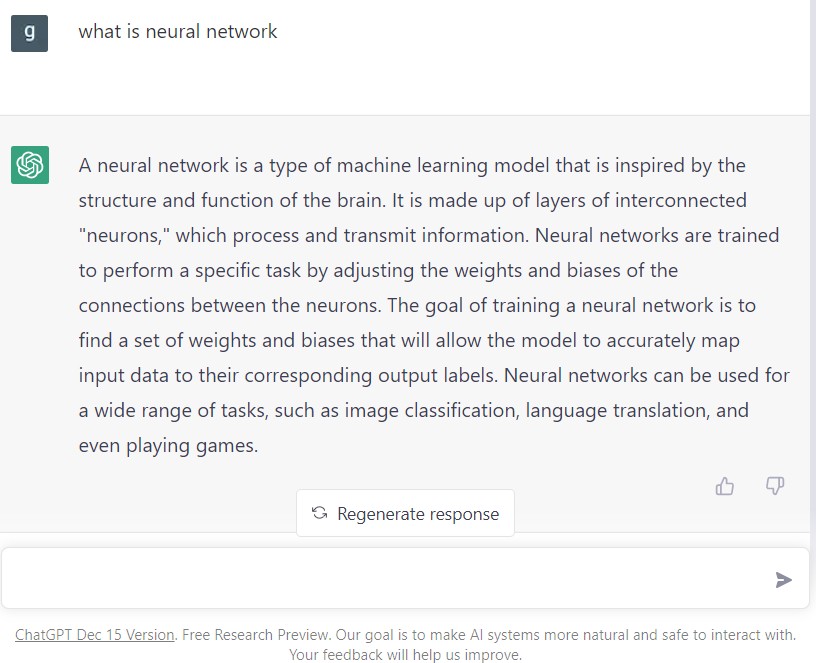
Using Deep Learning in diagnosing skin conditions

Image generation

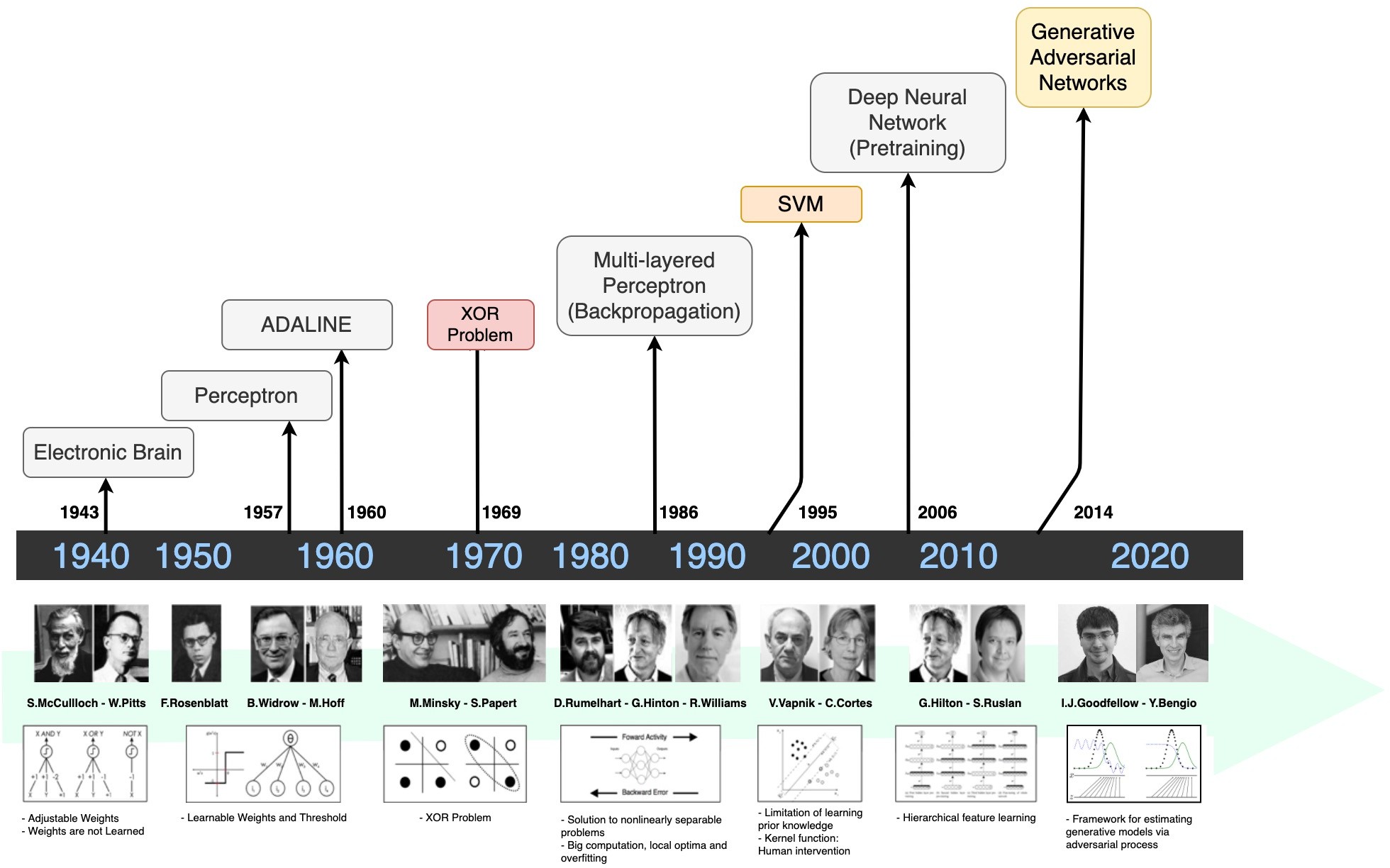


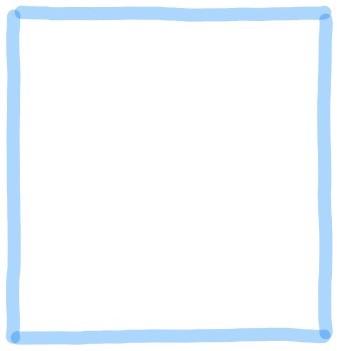
Katie Bouman’s CHIRP produces the first-ever image of a black hole.

ChatBot



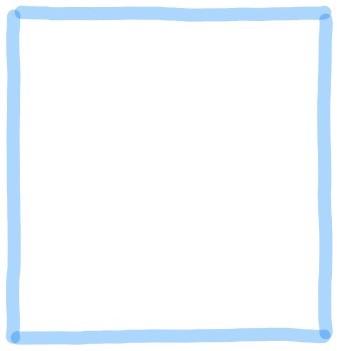
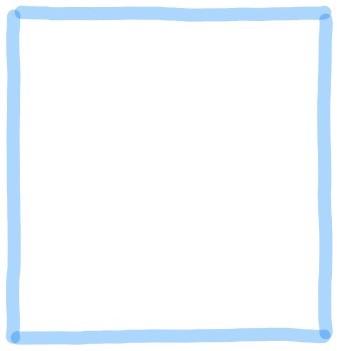
ChatGPT





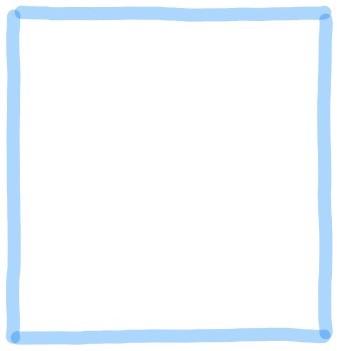
Disease prediction Game strategy

Natural Language Processing

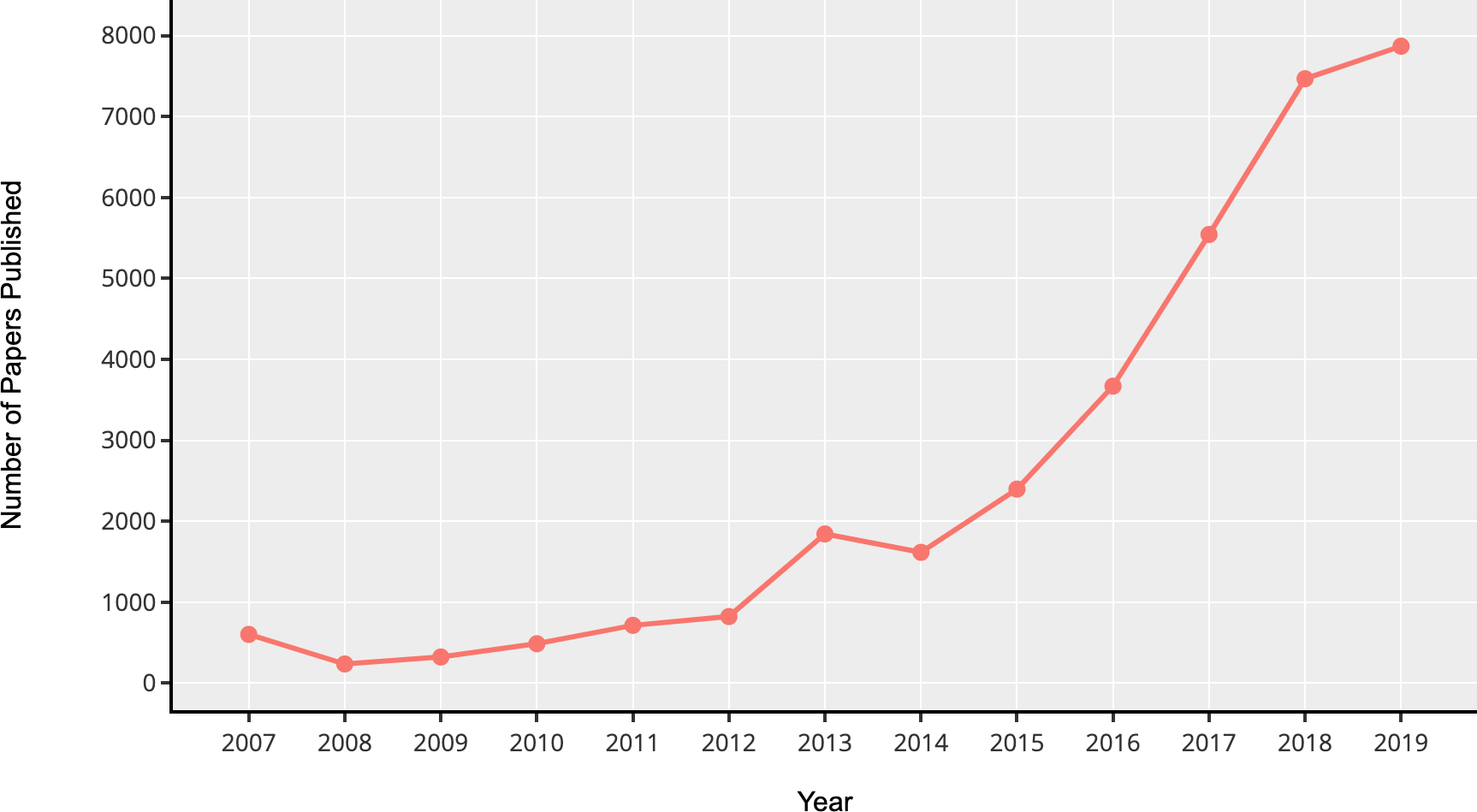


**2016**

7



**2018**

ArXiv papers on Machine Learning and Artificial Intelligence: 2007-2019

Neural Networks

* Origins: Algorithms that try to mimic the brain.
* Very widely used in 80s and early 90s; popularity diminished in late \*5678TGREPCHECK1%%&$ 90s.
* Recent resurgence: State--of--the--art technique for many applications
* Artificial neural networks are not nearly as complex or intricate as the actual brain structure

A simple decision

Say you want to decide whether you are going to attend a cheese festival this upcoming weekend. There are three variables that go into your decision:

1. Is the weather good?
2. Does your friend want to go with you?
3. Is it near public transportation?

We’ll assume that answers to these questions are the only factors that go into your decision.

A simple decision, cont.

I will write the answers to these question as binary variables *xi*, with zero being the answer ‘no’ and one being the answer ‘yes’:

1. Is the weather good? *x*1
2. Does your friend want to go with you? *x*2
3. Is it near public transportation? *x*3

Now, what is an easy way to describe the decision statement resulting from these inputs.

A simple decision, cont.

We could determine weights *wi* indicating how important each feature is to whether you would like to attend. We can then see if:

*x*1 *w·*

1 + *x*2 *w·* 2 + *x*3 *·w*3 *≥* threshold

For some pre-determined threshold. If this statement is true, we would attend the festival, and otherwise we would not.

### A simple decision, cont.

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

**A simple decision, cont.**

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

With a threshold of 5, this causes us to go if and only if the weather is good.

**A simple decision, cont.**

For example, if we really hated bad weather but care less about going with our friend and public transit, we could pick the weights 6, 2 and 2.

With a threshold of 5, this causes us to go if and only if the weather is good.

What happens if the threshold is decreased to 3? What about if it is decreased to 1?

If we deﬁne \*5678tgrepcheck1%%&$ a new binary variable *y* that represents whether we go to the festival, we can write this variable as:

*y* = { 0*, x*1 *w·*

1 + *x*2 *w·* 2 + *x*3 *·w*3 *<* threshold

1*, x*1 *w·*

Is this starting to look familiar yet?

1 + *x*2 *·w*2 + *x*3 *w·* 3 *≥* threshold

Now, if I rewrite this in terms of a dot product between the vector of of all binary inputs (*x*), a vector of weights (*w*), and change the threshold to the negative bias (*b*), we have:

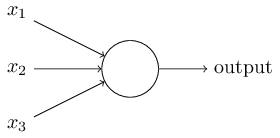
*y* = { 0*, x ·w* + *b <* 0

1*, x ·w* + *b ≥* 0

So we are really just ﬁnding separating hyperplanes again, much as we did with logistic regression and support vector machines!

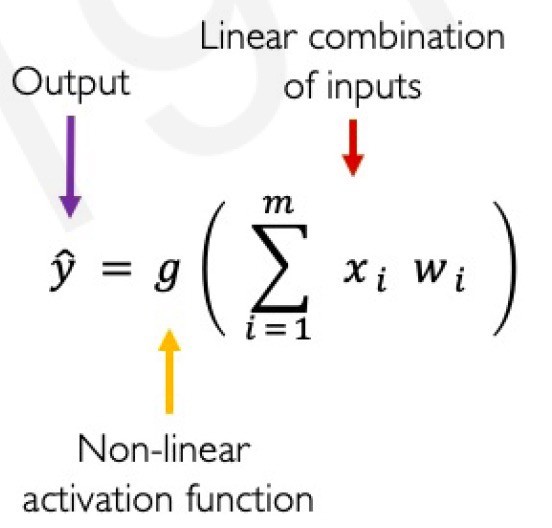
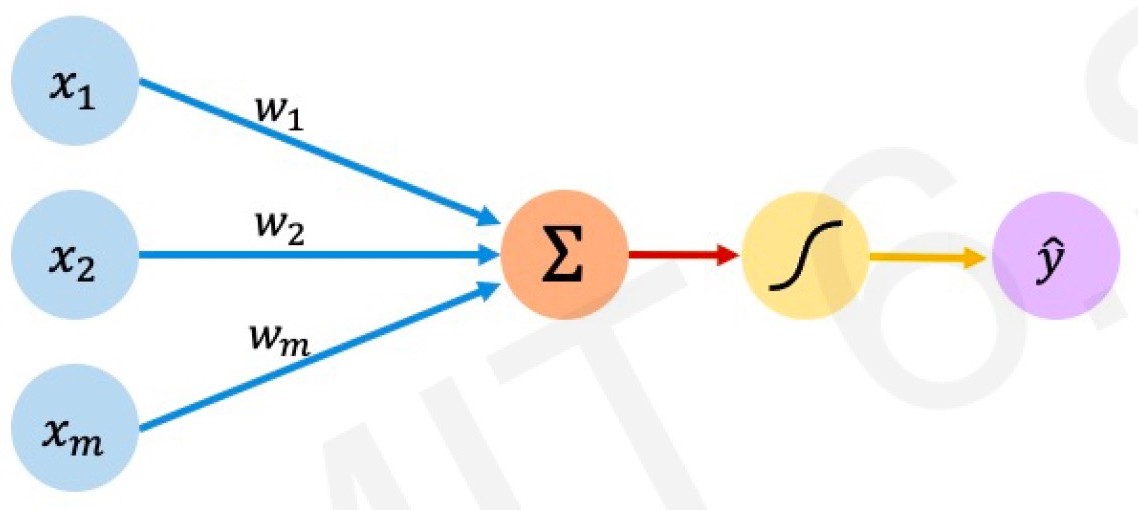
**A perceptron**

We can graphically represent this decision algorithm as an object that takes 3 binary inputs and produces a single binary output:



This object is called a perceptron when using the type of weighting scheme we just developed.

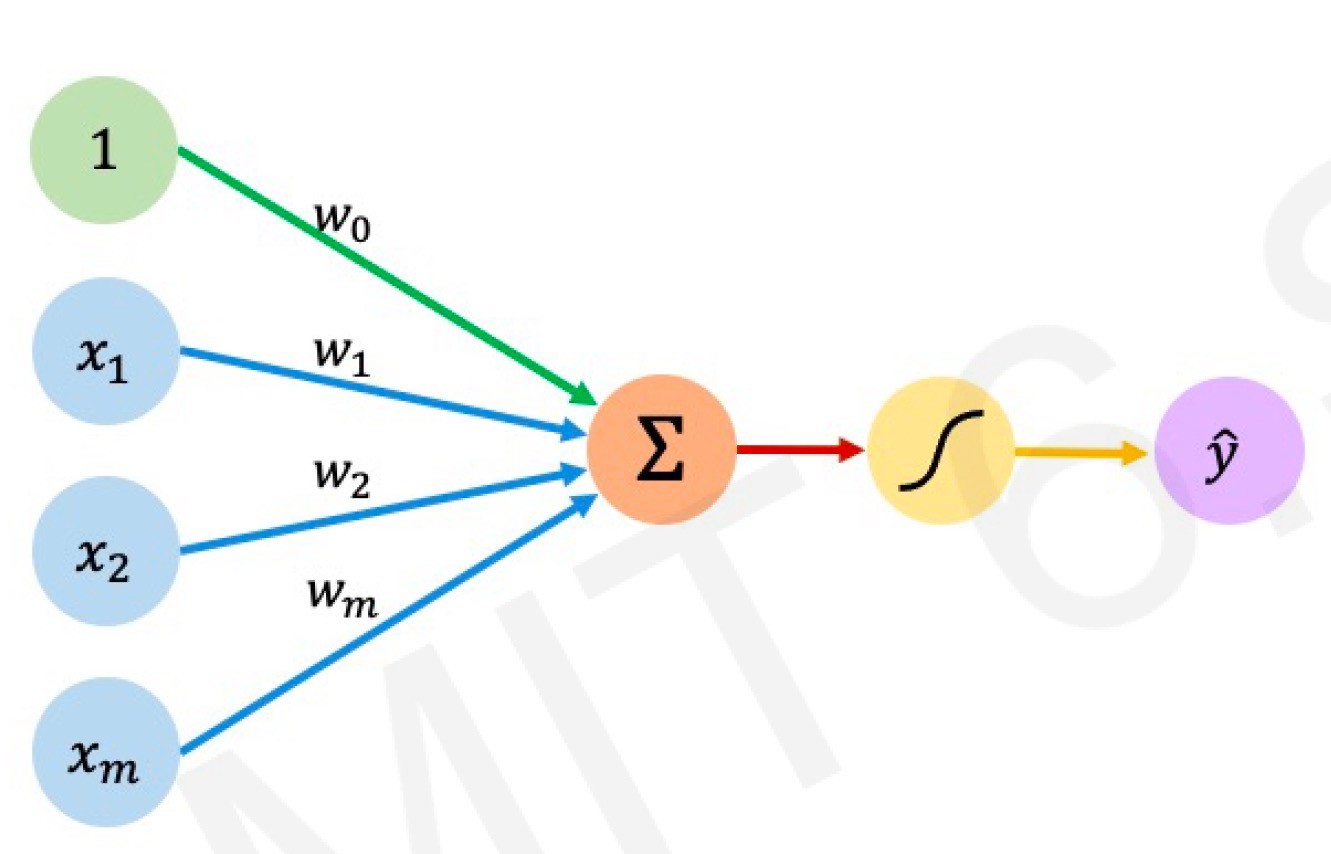
**A network of perceptrons**



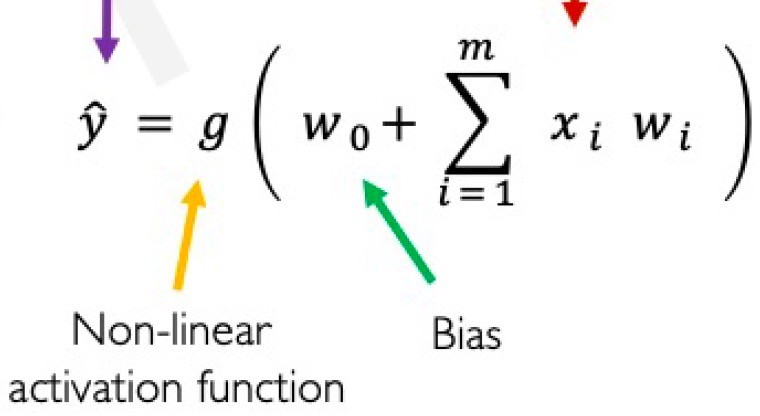
Linear combination of inputs

Input weight sum Non-linearity output

**A network of perceptrons**



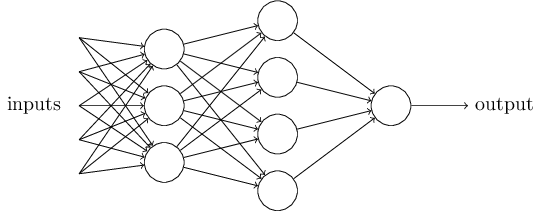
output



Linear combination of inputs

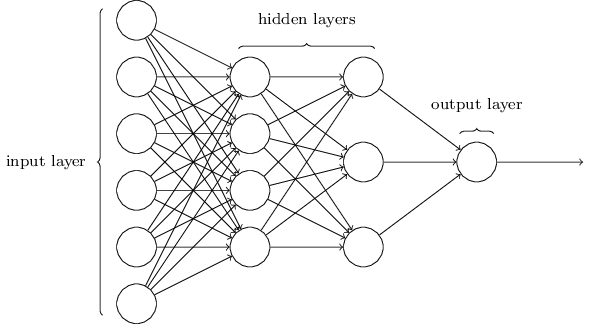
Input weight sum Non-linearity output

A perceptron takes a number of binary inputs and emits a binary output. Therefore it is easy to build a network of such perceptrons, where the output from some perceptrons are used in the inputs of other perceptrons:



Notice that some perceptrons seem to have multiple output arrows, even though we have deﬁned them as having only one output. This is only meant to indicate that a single output is being sent to multiple new perceptrons.

The input and outputs are typically represented as their own neurons, with the other neurons named hidden layers



**A network of perceptrons, cont.**

The biological interpretation of a perceptron is this: when it emits a 1 this is equivalent to ‘ﬁring’ an electrical pulse, and when it is 0 this is when it is not ﬁring. The bias indicates how difﬁcult it is for this particular node to send out a signal.

Neural Networks

Output units Hidden units

Input units Layered feed-forward network

* Neural networks are made up of **nodes** or **units**, connected by **links**
* Each link has an associated **weight** and **activation level**
* Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**

# Neuron Model: Logistic Unit

“bias unit”

x0

**x** = x1 x2



x0

x0 = 1

✓0

✓1

✓2

✓3

2

64

x3

✓0

**✓** = ✓1

6

4

2

7

7

✓2

✓3

h**✓**(**x**) = g (**✓**|**x**)

1

=

1 + e—**✓**Tx

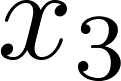
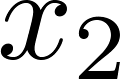
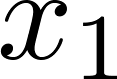
Sigmoid (logistic) activation function: g(z) =

1

1 + e—*z*

Neural Network

bias units



x0

a(2)

0

h**✓**(**x**)

Layer 1

(Input Layer)

Layer 2

(Hidden Layer)

Layer 3

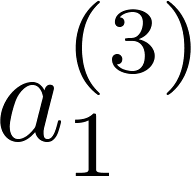
(Output Layer)

## Feed-Forward Process

* + Input layer units are set by some exterior function (think of these as **sensors**), which causes their outp links to be **activated** at the specified level
  + Working forward through the network, the **input function** of each unit is applied to compute the inpu value
    - Usually this is just the weighted sum of the activation on the links feeding into this node
  + The **activation function** transforms this input function into a final value
    - Typically this is a **nonlinear** function, often a **sigmoid**

function corresponding to the “threshold” of that node

Neural Network



⇥(1)

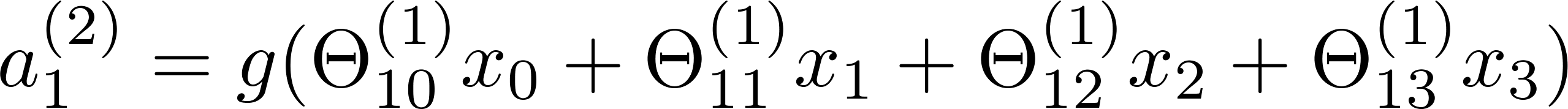
⇥(2)

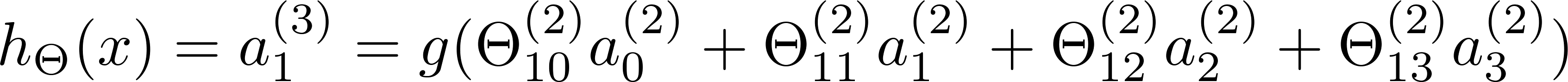
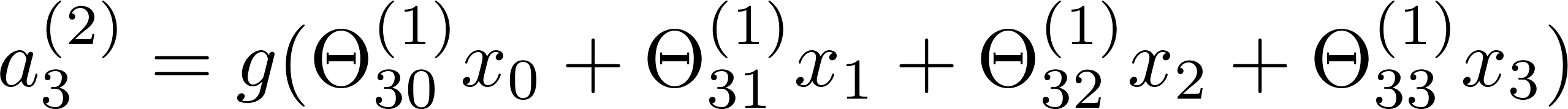
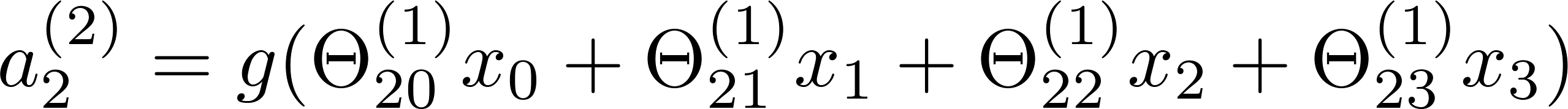
*a* (*j*) = “activation” of unit *i* in layer *j*

*i*

h**✓**(**x**)

Θ(*j*) = weight matrix controlling function mapping from layer *j* to layer *j* + 1





If network has *sj* units in layer *j and sj+1* units in layer *j*+1, then Θ(*j*) has dimension *sj+1 ×* (*sj*+1) .

⇥(1) R3⇥4 ⇥(2) R1⇥4

a(2)

1

Vectorization

= g ⇣⇥(1)x0 + ⇥(1)x1 + ⇥(1)x2 + ⇥(1)x3⌘

10

11

12

13

= g ⇣z(2)⌘

a(2)

1

2

= g ⇣⇥(1)x0 + ⇥(1)x1 + ⇥(1)x2 + ⇥(1)x3⌘

= g ⇣z(2)⌘

a(2)

20

21

22

23

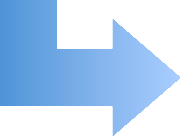
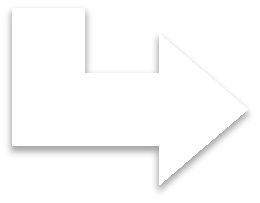
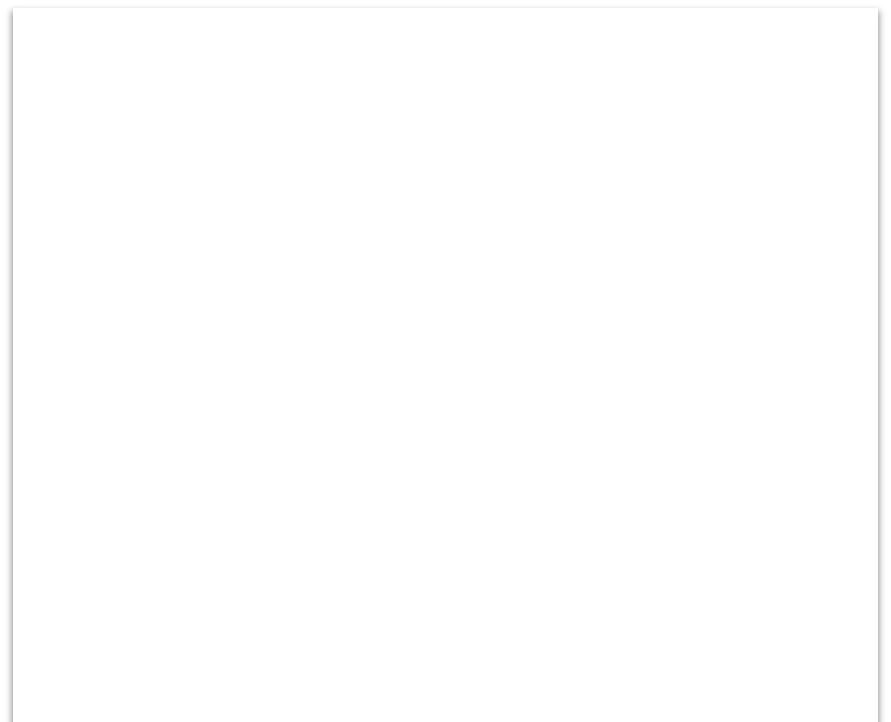
2

3

= g ⇣⇥(1)x0 + ⇥(1)x1 + ⇥(1)x2 + ⇥(1)x3⌘

= g ⇣z(2)⌘

h⇥(**x**) = g ⇣⇥(2)a(2)



30

31

32

33

10

0

11

1

12

2

13

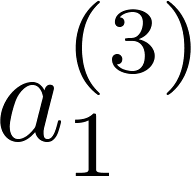
3

1

+ ⇥(2)a(2) + ⇥(2)a(2)

+ ⇥(2)a(2)⌘ = g ⇣z(3)

h**✓**(**x**)



Feed@Forward Steps: **z**(2) = ⇥(1)**x a**(2) = g(**z**(2))

Add a(2) = 1

0

3

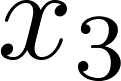
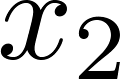
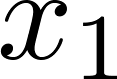
⇥(1) ⇥(2)

**z**(3) = ⇥(2)**a**(2)

h⇥(**x**) = **a**(3) = g(**z**(3))

###### Other Network Architectures

1]



**s** = [3, 3, 2,

h**✓**(**x**)

Layer 1 Layer 2

Layer 3

Layer 4

*L* denotes the number of layers

**s** N+*L*

contains the numbers of nodes at each layer

* Not counting bias units
* Typically, *s0 = d* (# input features) and *sL-1=K* (# classes)

Multiple Output Units: One@vs@Rest



Pedestrian Car Motorcycle Truck

h⇥(**x**) 2 R*K*

We want:

2

2 0 2 0 2 0

h⇥(**x**) ⇡ 46 h⇥(**x**) ⇡ 64 h⇥(**x**) ⇡ 64 h⇥(**x**) ⇡ 64

0 7

1 7

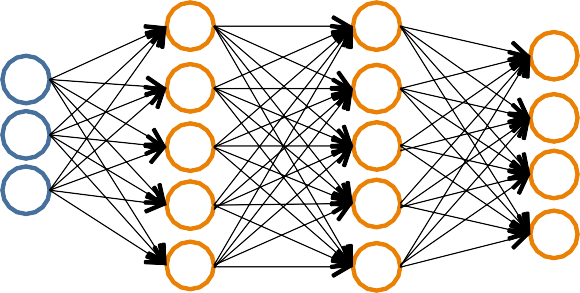
0 7

0 7

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  | | |
|  |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

when pedestrian when car when motorcycle when truck

Multiple Output Units: One@vs@Rest

We want:

2

h⇥(**x**) 2 R*K*

h⇥(**x**) 2 0

⇡ 7

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
|  |  |  |  |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

64

0

h⇥(**x**) ⇡ 6

4

1 7

0

h⇥(**x**) ⇡ 6

4

2

0 7

0

h⇥(**x**) ⇡ 6

4

2

0 7

when pedestrian when car when motorcycle when truck



* Given {(**x**1,*y*1), (**x**2,*y*2), ..., (**x***n*,*yn*)}
* Must convert labels to 1@of@*K* representation

– e.g.,

d on slide by Andrew Ng

**y***i* = 264

0

1. when motorcycle,

1

7

0

**y***i* = 2

64

0

1. when car, etc.

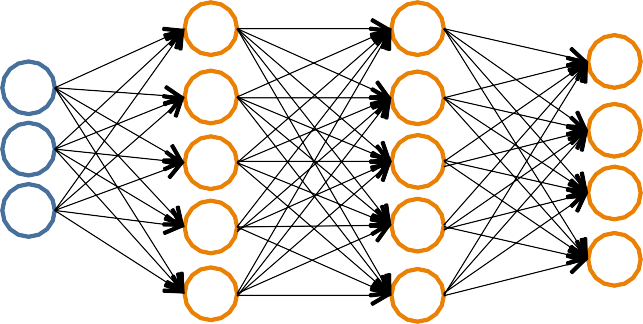
7

0

0

1

Neural Network Classification

**Given:**

{(**x**1,*y*1), (**x**2,*y*2), ..., (**x***n*,*yn*)}

**s** N+*L* contains # nodes at each layer

– *s0 = d* (# features)



Binary classification

Multi@class classification (*K* classes)

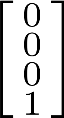
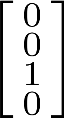
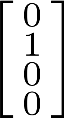
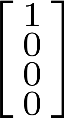
*y* = 0 or 1

1 output unit (*s*

*=* 1)

**y** 2 R*K*

e.g. , , ,

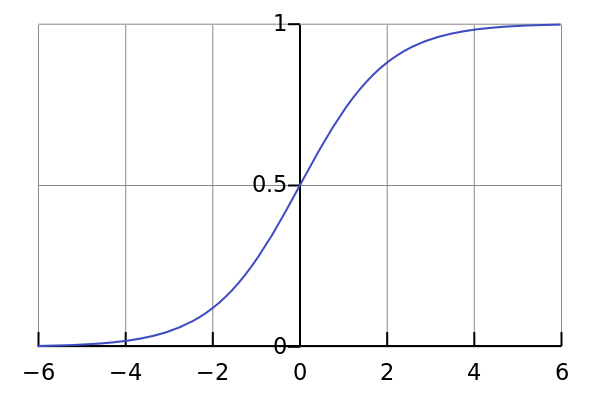
pedestrian car motorcycle truck

*L-1*

*K* output units (*sL-1= K*)

Understanding Representations

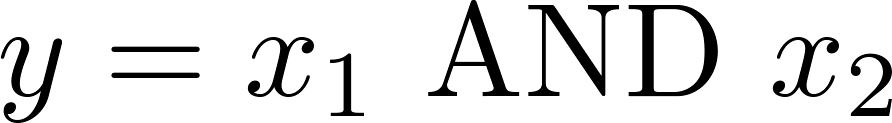
#### Representing Boolean Functions



Logistic / Sigmoid Function

g(z)

**Simple example: AND**



h**✓**(**x**)



@30

+20

+20

h (**x**) = *g*(-30 + 20*x* + 20*x* )

*x*1

*x*2

hΘ(**x**)

|  |  |  |
| --- | --- | --- |
| *x*1 | *x*2 | hΘ(**x**) |
| 0 | 0 | *g*(@30) ≈ 0 |
| 0 | 1 | *g*(@10) ≈ 0 |
| 1 | 0 | *g*(@10) ≈ 0 |
| 1 | 1 | *g*(10) ≈ 1 |

® 1 2

0

0

0

1

1

0

1

1

Representing Boolean Functions

h**✓**(**x**) h**✓**(**x**)



@10

+20

+20

**OR**



@30

+20

+20

**AND**

**NOT**

**(NOT *x*1) AND (NOT *x*2)**



+10

@20



+10

@20 @20

h**✓**(**x**)

h**✓**(**x**)

Combining Representations to Create Non@Linear Functions

**(NOT *x*1) AND (NOT *x*2)**



@10

+20

+20

**OR**



@30 **AND**

+20

+20



+10

@20 @20

h**✓**(**x**)

h**✓**(**x**)

h**✓**(**x**)



**XOR**



@10

@30

+20

+20

+10

in I

+20

@20

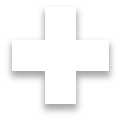
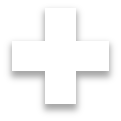
@20

in III

+20

I or III

h**✓**(**x**)



II

I

III

IV

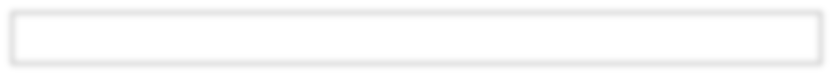
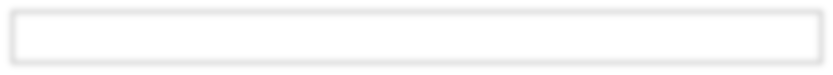
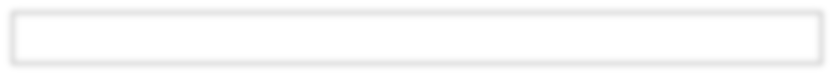
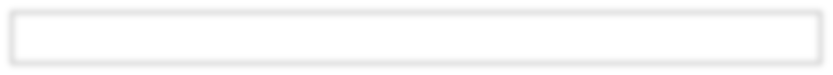


20 × 20 pixel images

*d* = 400 10 classes

*x*1 *... x*20

*x*21 *... x*40



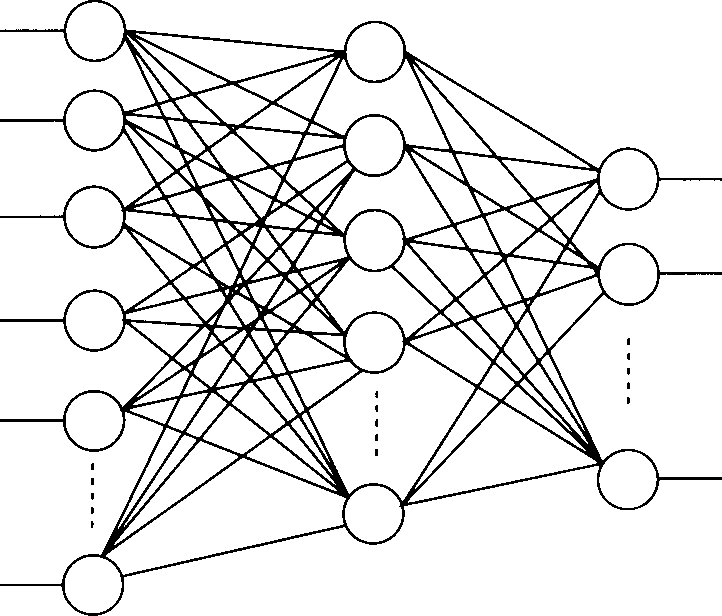
*x*41 *... x*60

...

*x*381 *... x*400

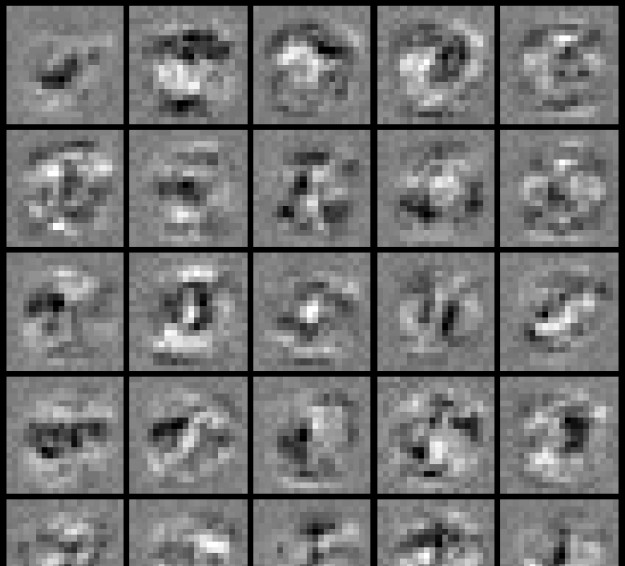
Each image is “unrolled” into a vector **x** of pixel intensities

*x*1 *x*2 *x*3



Hidden Layer

Output Layer

*x*4 *x*5

*x*d

Input Layer

“0”

“1”

“9”

Visualization of Hidden Layer

# Neural Network Learning

#### Perceptron Learning Rule

**✓** ← **✓** + ↵(y — h(**x**))**x**

Equivalent to the intuitive rules:

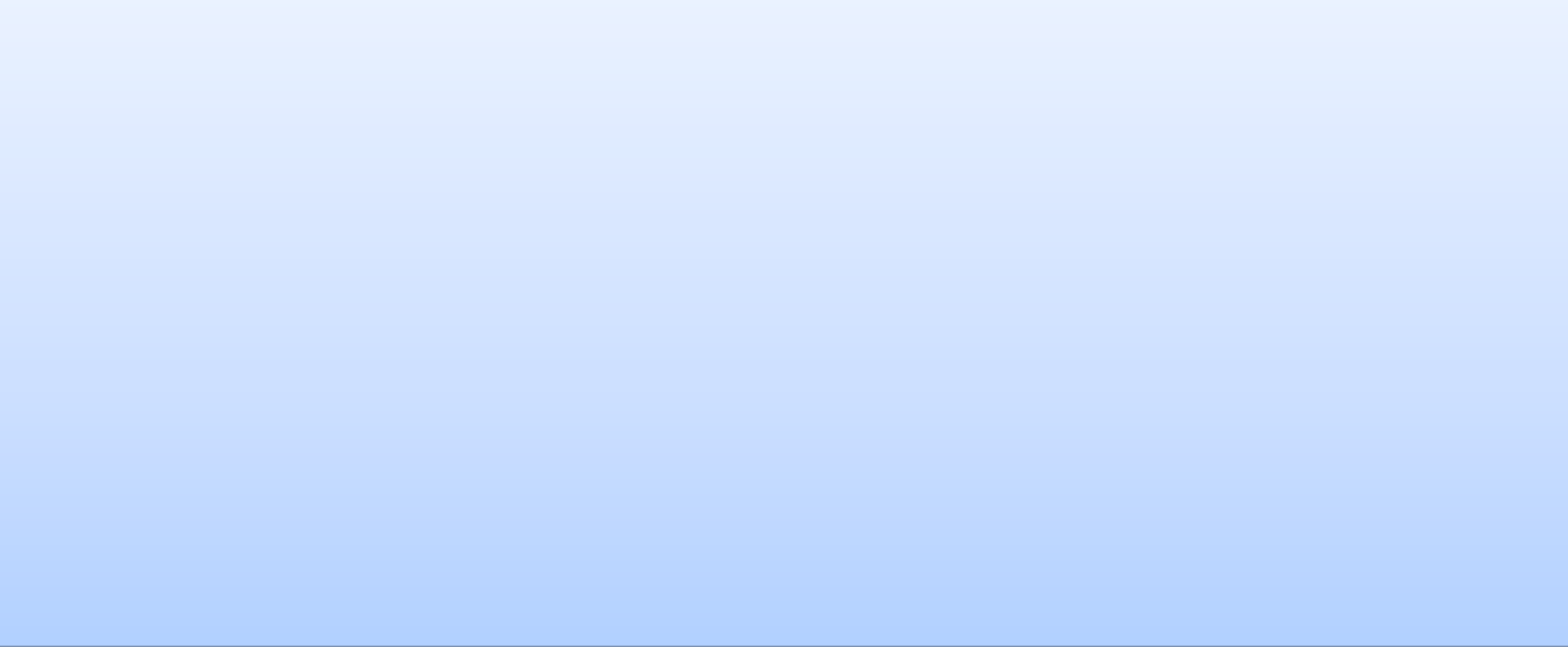
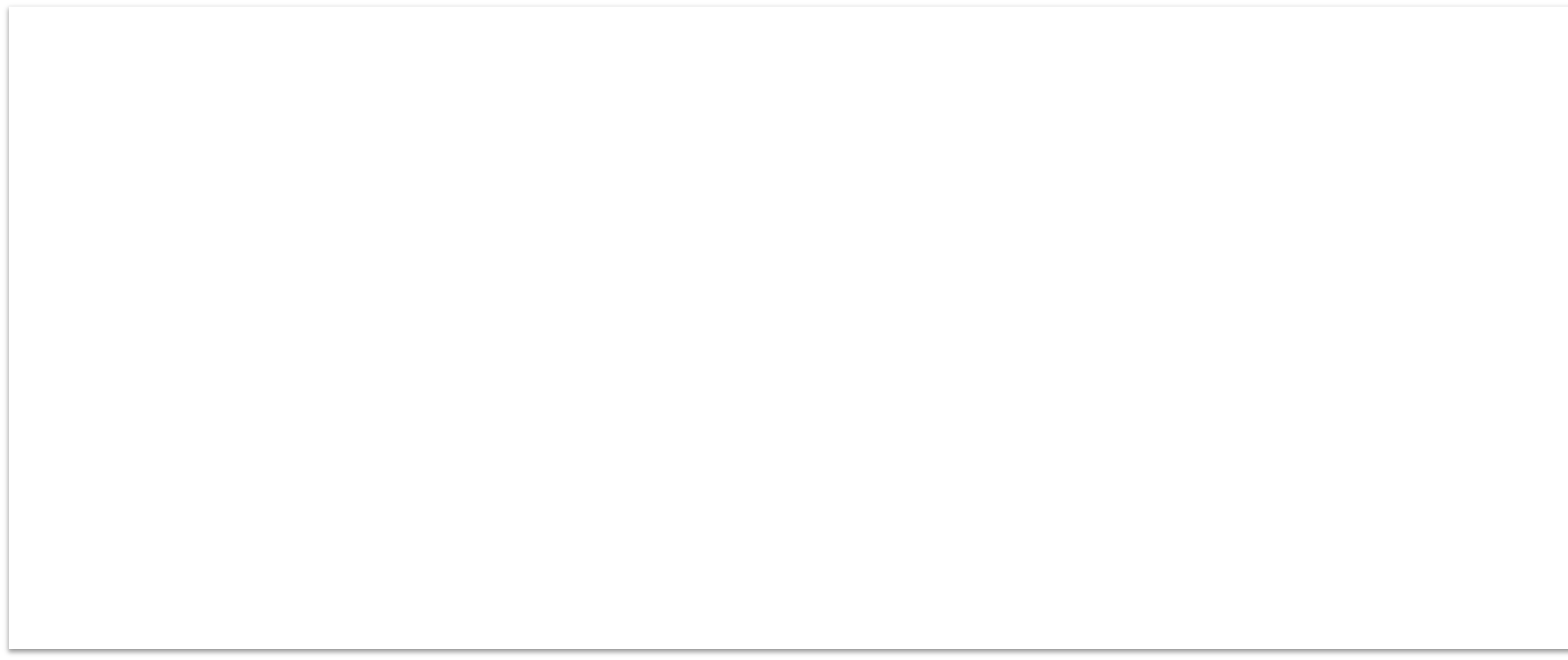
* If output is correct, don’t change the weights
* If output is low (*h*(**x**) = 0, *y* = 1), increment weights for all the inputs which are 1
* If output is high (*h*(**x**) = 1, *y* = 0), decrement weights for all inputs which are 1

**Perceptron Convergence Theorem**:

* If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minicksy & Papert, 1969]

Batch Perceptron

Given training data (**x**(*i*), y(*i*))}*n*



*i*=1

←

Let **✓** [0, 0,. .., 0]

Repeat:

Let **A** [0, 0,. .., 0]

←

for i = 1 .. . n, do

if y(*i*)**x**(*i*)**✓** 0 // prediction for i*th* instance is incorrect

**A** ← **A** + y(*i*)**x**(*i*)

**A** ← **A**/n // compute average update

**✓** ← **✓** + ↵**A** Until k**A**k2 < ✏

* Simplest case: α = 1 and don’t normalize, yields the fixed increment perceptron
* Each increment of outer loop is called an **epoch**

##### Learning in NN: Backpropagation

* + Similar to the perceptron learning algorithm, we cycle through our examples
    - If the output of the network is correct, no changes are made
    - If there is an error, \*5678TGREPCHECK1%%&$ weights are adjusted to reduce the error
  + The trick is to assess the blame for the error and divide it among the contributing weights

##### Cost Function

Logistic Regression:

*n d*

X

X ✓

1

J(✓) = — n

[y*i*

*i*=1

log h**✓**

(**x***i*

) + (1 — y*i*

) log (1 — h**✓**

(**x***i*

λ

))] +

2n

2

*j*

*j*=1

Neural Network:

h⇥ 2 R*K* (h⇥(**x**))*i* = i*th*output

*K*

J(⇥) = —

1 "X*n* X

n

y*ik* log (h⇥(**x***i*))*k* + (1 — y*ik*) log

⇣1 — (h⇥(**x***i*))*k*⌘

*i*=1 *k*=1

λ *L*—1 *sl—*1 *sl*

X X X

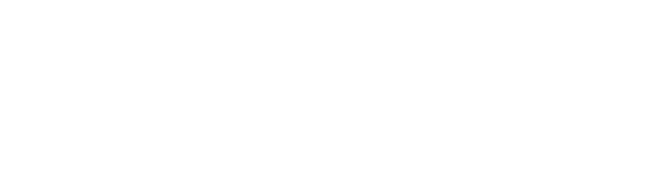
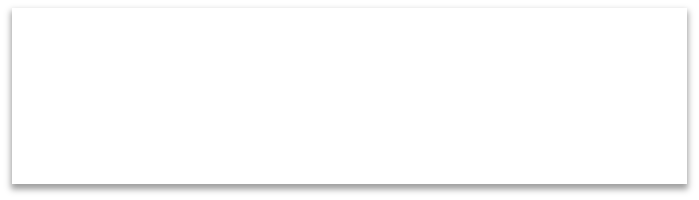
⇣

+

2n

(*l*) 2

*ji*



*k*th class:

true, predicted

not *k*th class: true, predicted

⇥

⌘

*l*=1 *i*=1 *j*=1

##### Optimizing the Neural Network

J(⇥) = —

1 "X*n* X

n

y*ik* log(h⇥(**x***i*))*k* + (1 — y*ik*) log

⇣1 — (h⇥(**x***i*))*k*⌘

*i*=1 *k*=1

*K*

λ *L*—1 *sl—*1 *sl*

X X X

⇣

+

2n

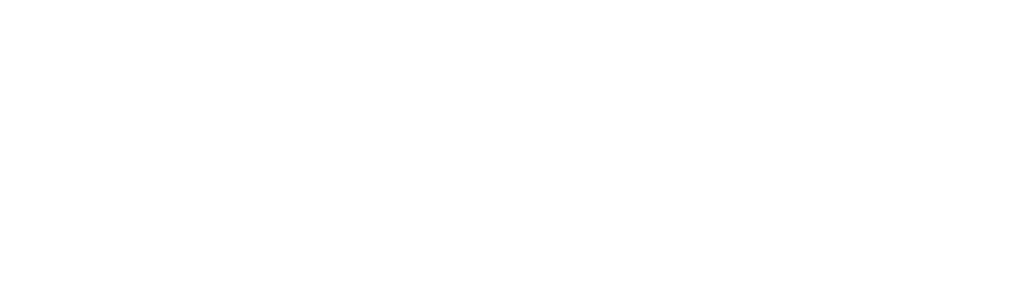
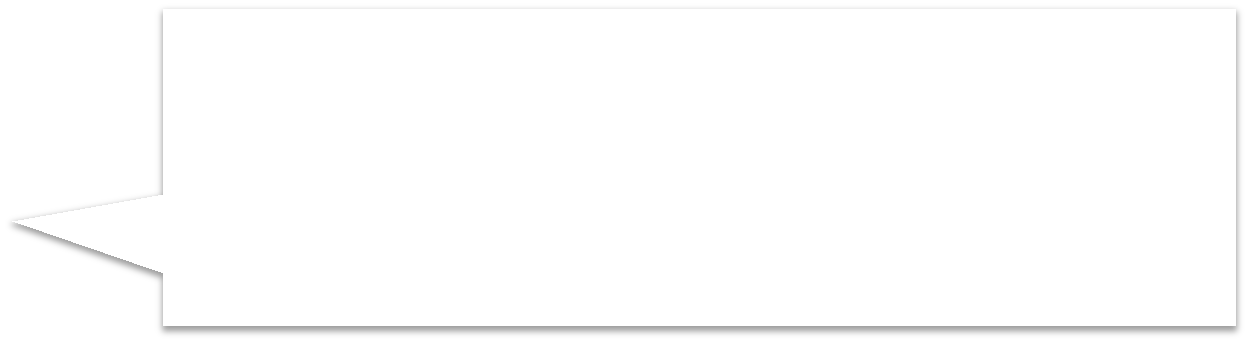
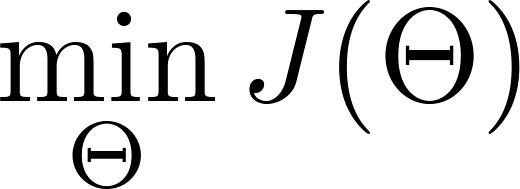
1. 2

*ji*

⇥

⌘

*l*=1 *i*=1 *j*=1

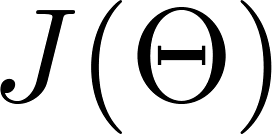


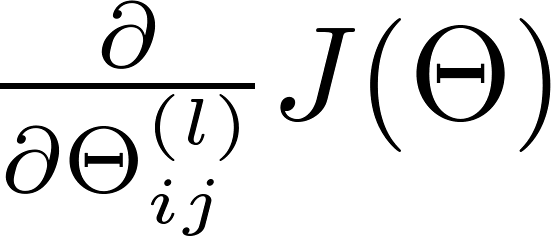
Solve via:

*J*(Θ) is not convex, so GD on a neural net yields a local optimum

* + But, tends to work well in practice

Need code to compute:

• 

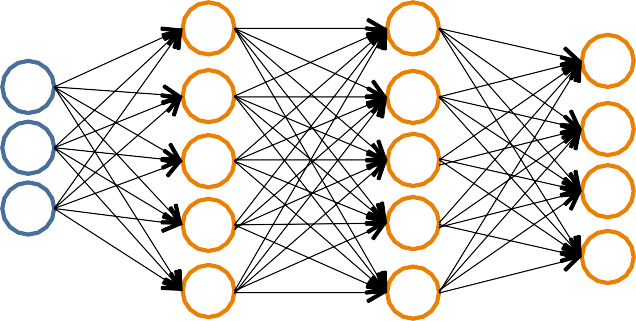
•

Forward Propagation

* + - Given one labeled training instance (**x**, *y*):

Forward Propagation

* + - **a**(1) = **x**
    - **z**(2) = Θ(1)**a**(1)
    - **a**(2) = *g*(**z**(2)) [add a0

(2)]

**a**(1)

**a**(2) **a**(3) **a**(4)

* + - **z**(3) = Θ(2)**a**(2)
    - **a**(3) = *g*(**z**(3)) [add a0(3)]
    - **z**(4) = Θ(3)**a**(3)
    - **a**(4) = hΘ(**x**) = *g*(**z**(4))
      * Each hidden node *j* is “responsible” for some

fraction of the error *δ* (*l*) in each of the output nodes to which it connects

*j*

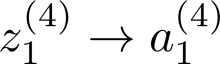
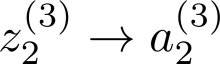
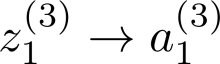
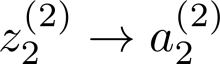
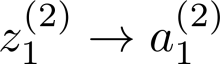
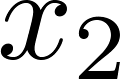
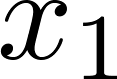
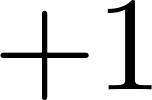
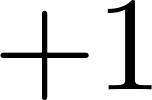
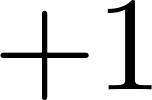
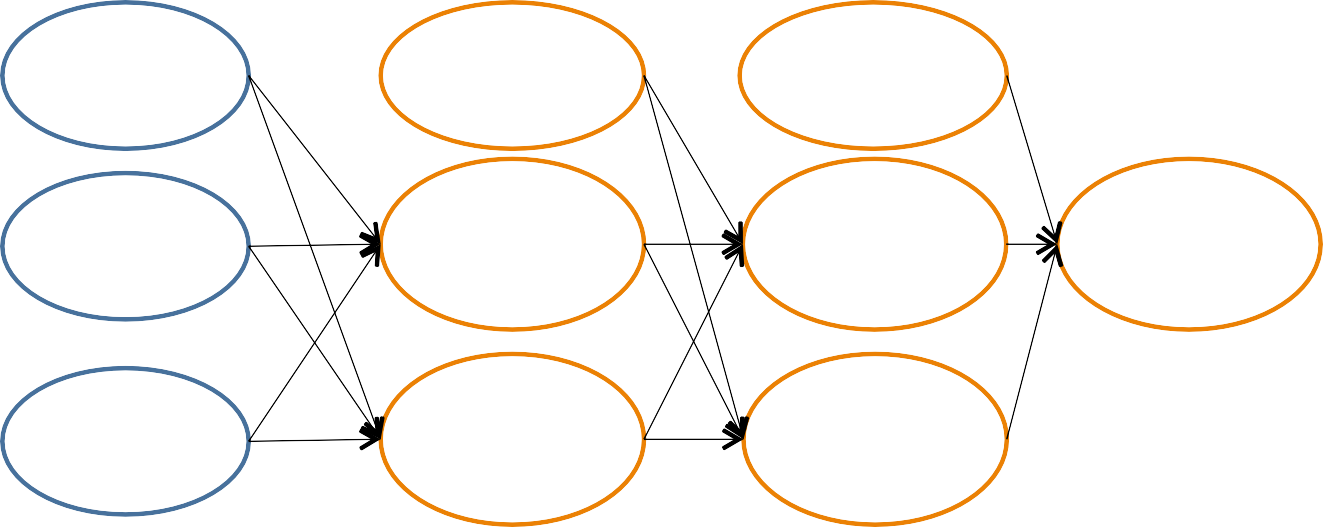
* + - * *δ* (*l*) is divided according to the strength of the

*j*

connection between \*5678TGREPCHECK1%%&$ hidden node and the output

node

* + - * Then, the “blame” is propagated back to provide the error values for the hidden layer



6(2)

1

6(3)

1

6(4)

1

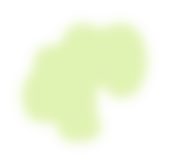
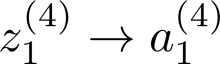
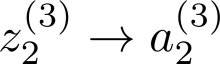
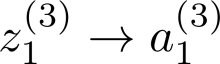
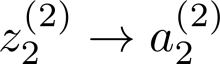
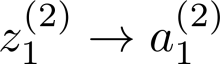
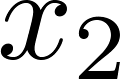
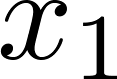
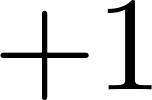
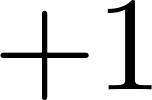
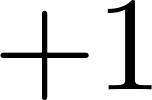
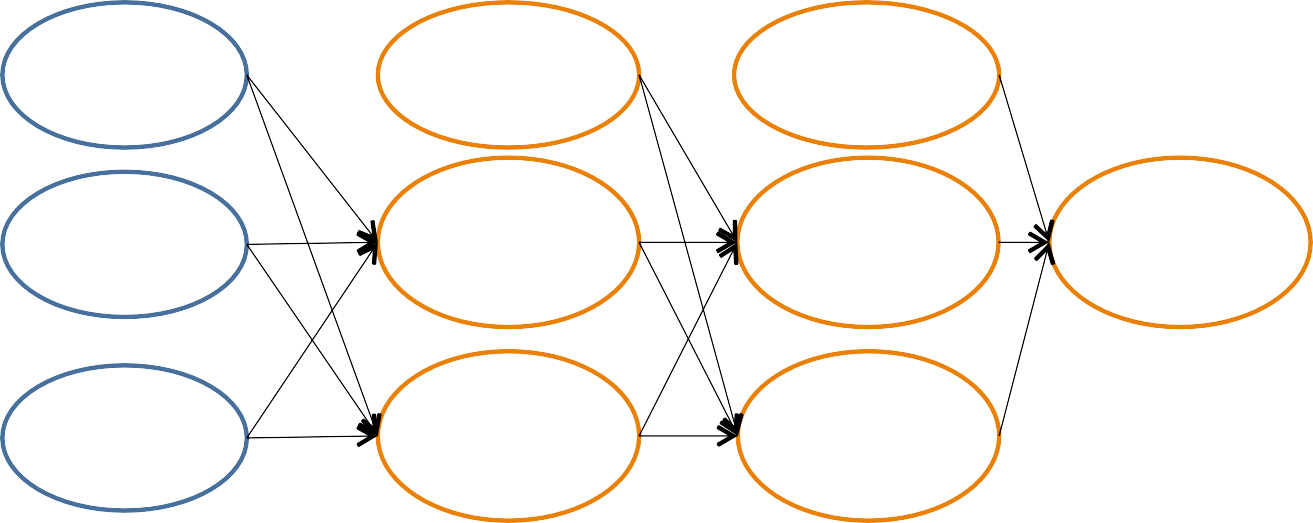
6(2)

2

6(3)

2

(2)



6

1

(2)

(3)

1

6

(3)

(4)

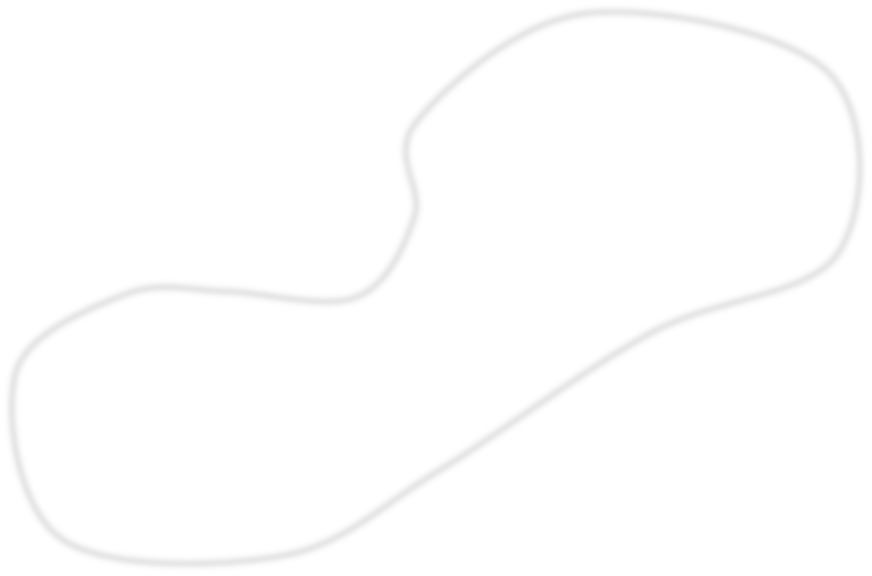
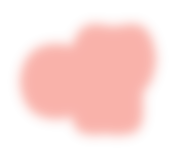
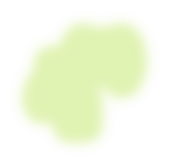
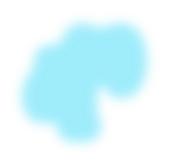
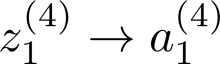
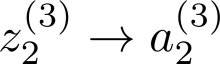
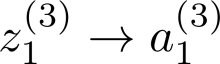
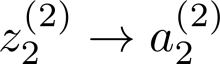
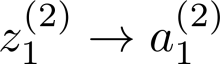
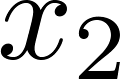
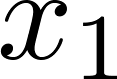
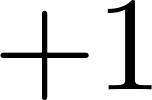
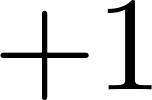
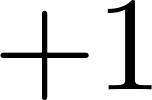
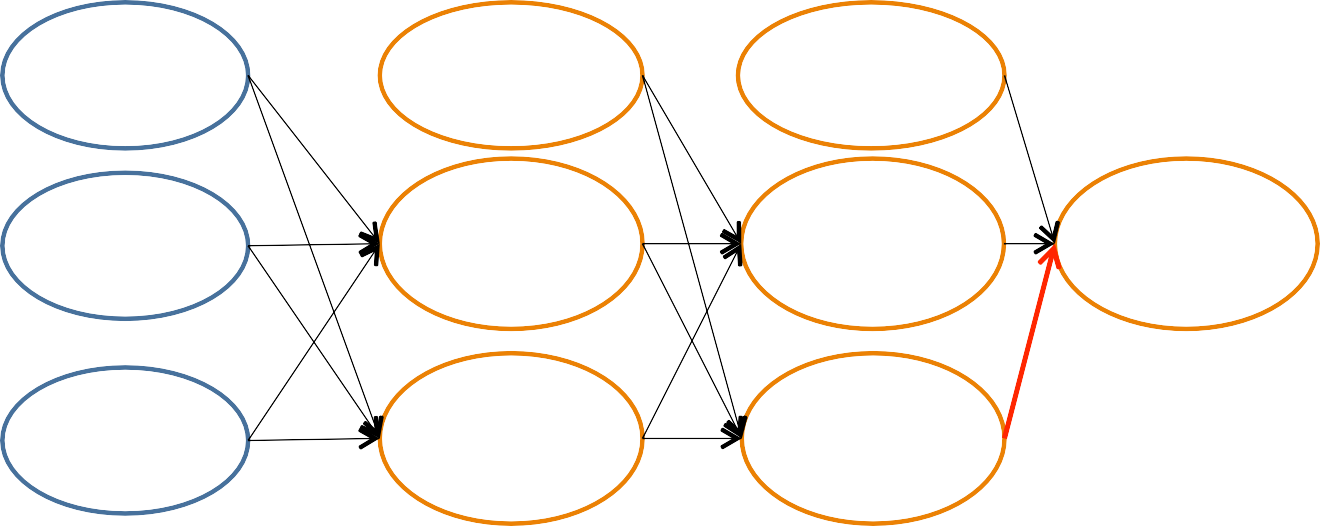
1

6

***δ***(4) = ***a***(4) – **y**

62 62

×*δ* (4)



6(2)

1

6(3)

1

6(4)

1

⇥(3)

12

6(2)

2

6(3)

2

*δ* (3) = Θ

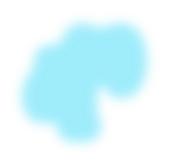
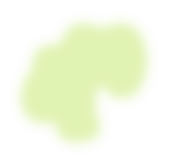
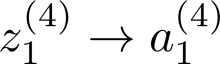
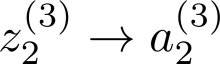
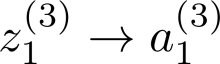
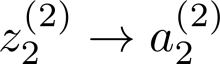
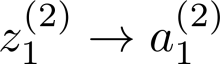
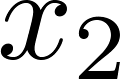
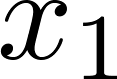
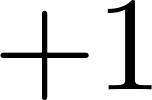
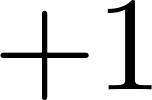
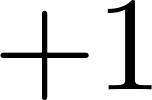
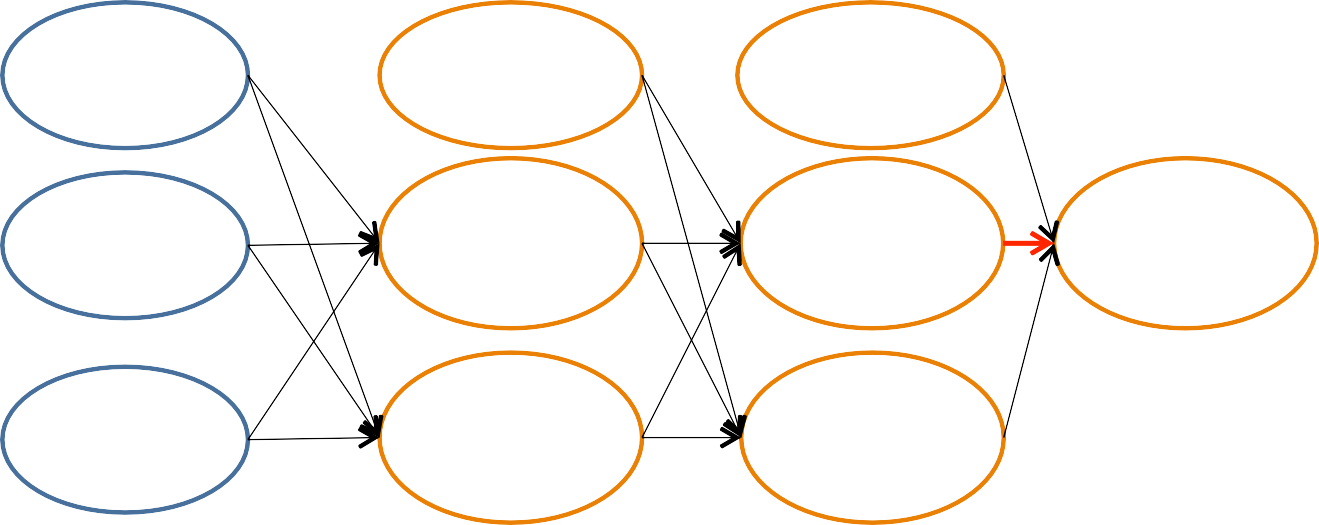
2

(3)

12

1

(2)



6

1

(3)

1

6

(4)

1

6

(2)

6

2

(3)

2

6

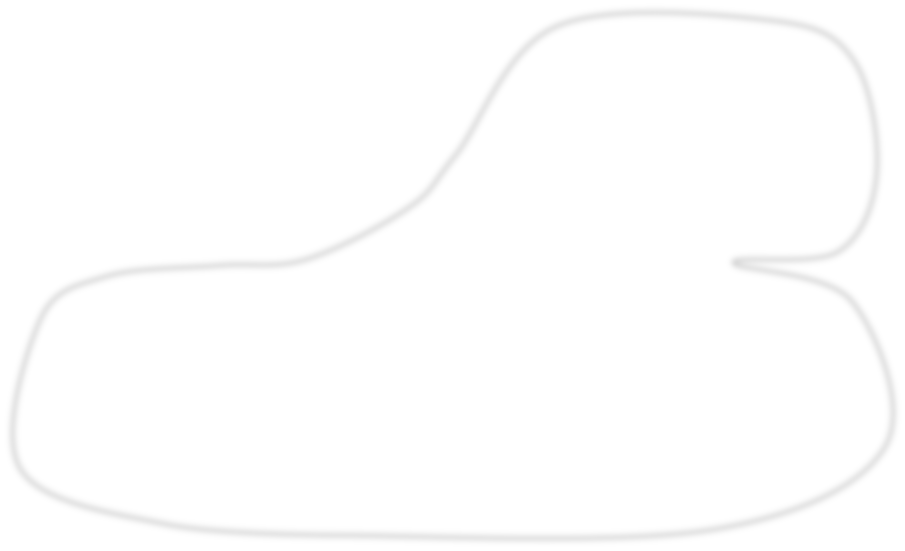
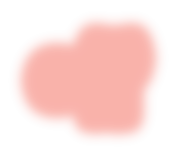
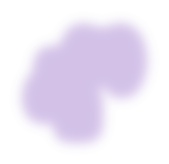
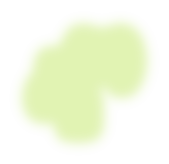
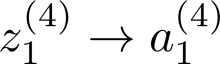
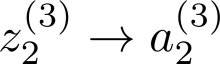
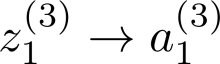
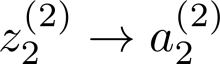
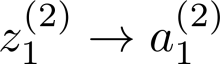
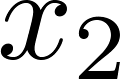
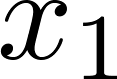
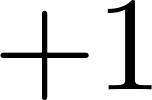
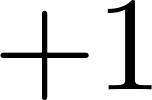
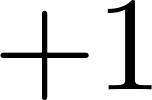
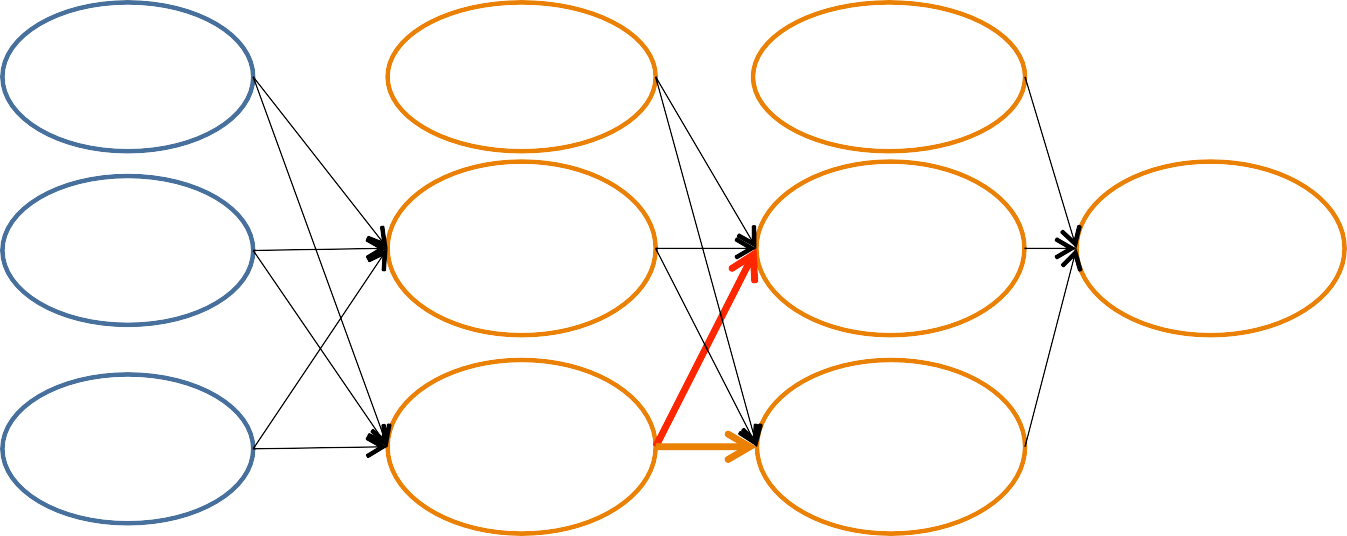
*δ* (3) = Θ (3) ×*δ*

(4)

2 12 1

*δ*1(3) = Θ11(3) ×*δ*1(4)

*δ* (2) = Θ (2) ×*δ* (3) + Θ (2) ×*δ* (3)



6(2)

1

6(3)

6(4)

⇥(2)

1

1

12

6(2)

2

⇥(2)

22

6(3)

2

2 12 1 22 2

*δj*(*l*) = “error” of node *j* in layer *l*

Formally,

(*l*)

*j*

6

@

= (*l*)

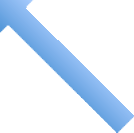
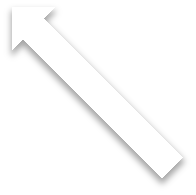
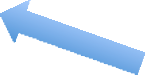
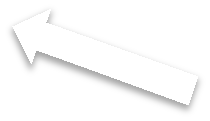
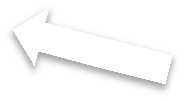
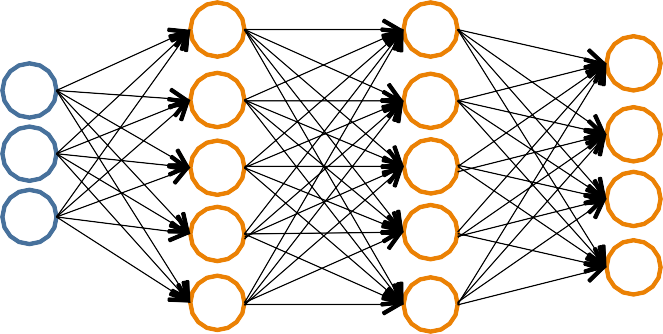
@z*j*

cost(**x***i*)

where cost(**x***i*) = y*i* log h⇥(**x***i*) + (1 — y*i*) log(1 — h⇥(**x***i*))

Backpropagation: Gradient Computation

Let *δ* (*l*) = “error” of node *j* in layer *l*



*j*

(#layers *L* = 4)

Backpropagation

Element@wise product .\*

***δ***(2)

***δ***(3)

***δ***(4)

* ***δ***(4) = ***a***(4) – **y**
* ***δ***(3) = (Θ(3))T***δ***(4) .\*

*g’*(**z**(3))

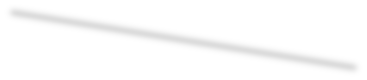
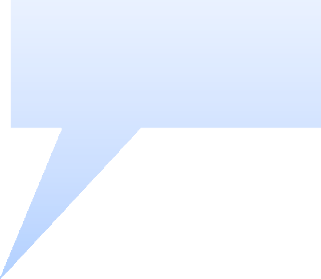
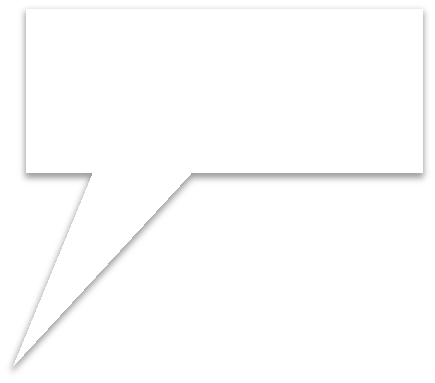
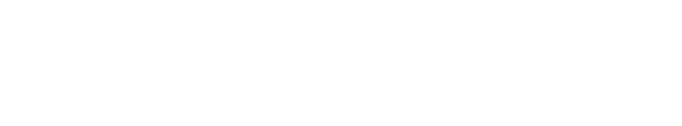
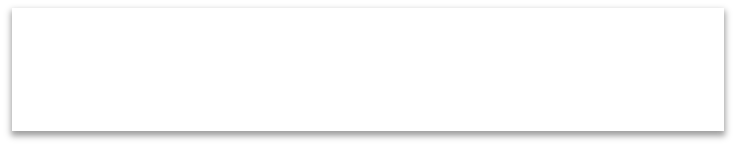
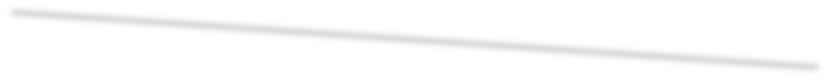
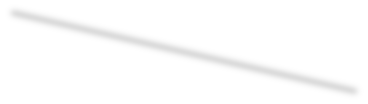
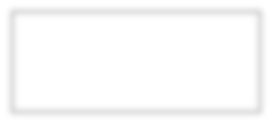
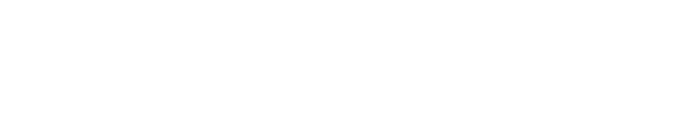
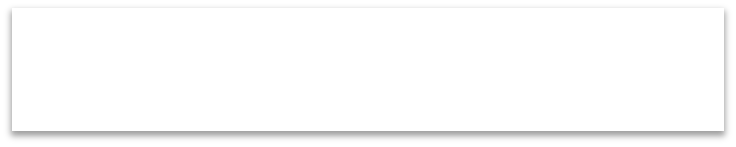
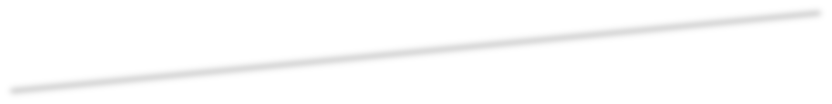
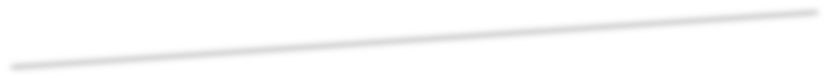
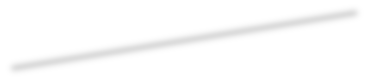
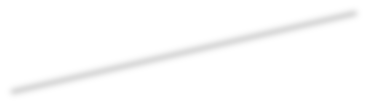
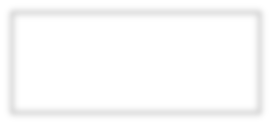
* ***δ***(2) = (Θ(2))T***δ***(3) .\*
* (No ***δ***(1))

*g’*(**z**(3)) = **a**(3) .\* (1–**a**(3))

*g’*(**z**(2))

*g’*(**z**(2)) = **a**(2) .\* (1–**a**(2))

@ J(⇥) = a(*l*)6(*l*+1)



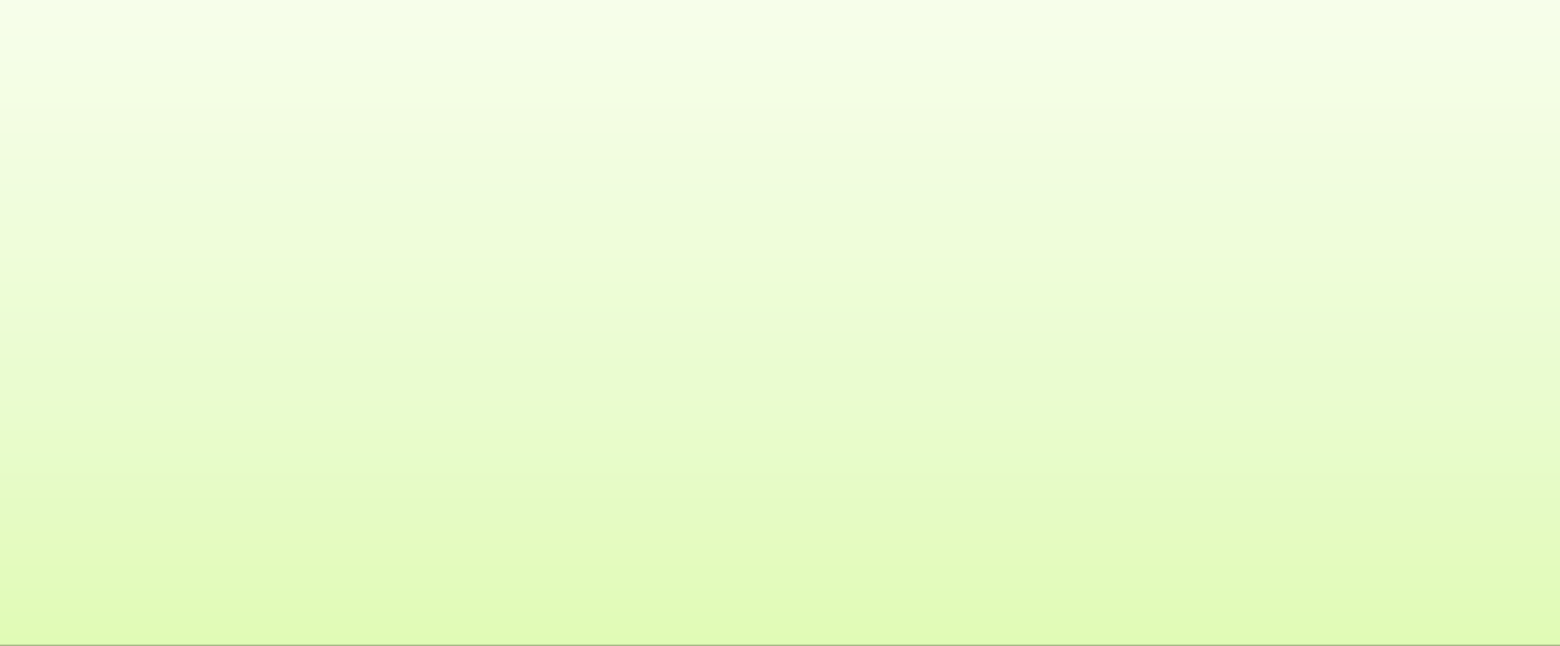
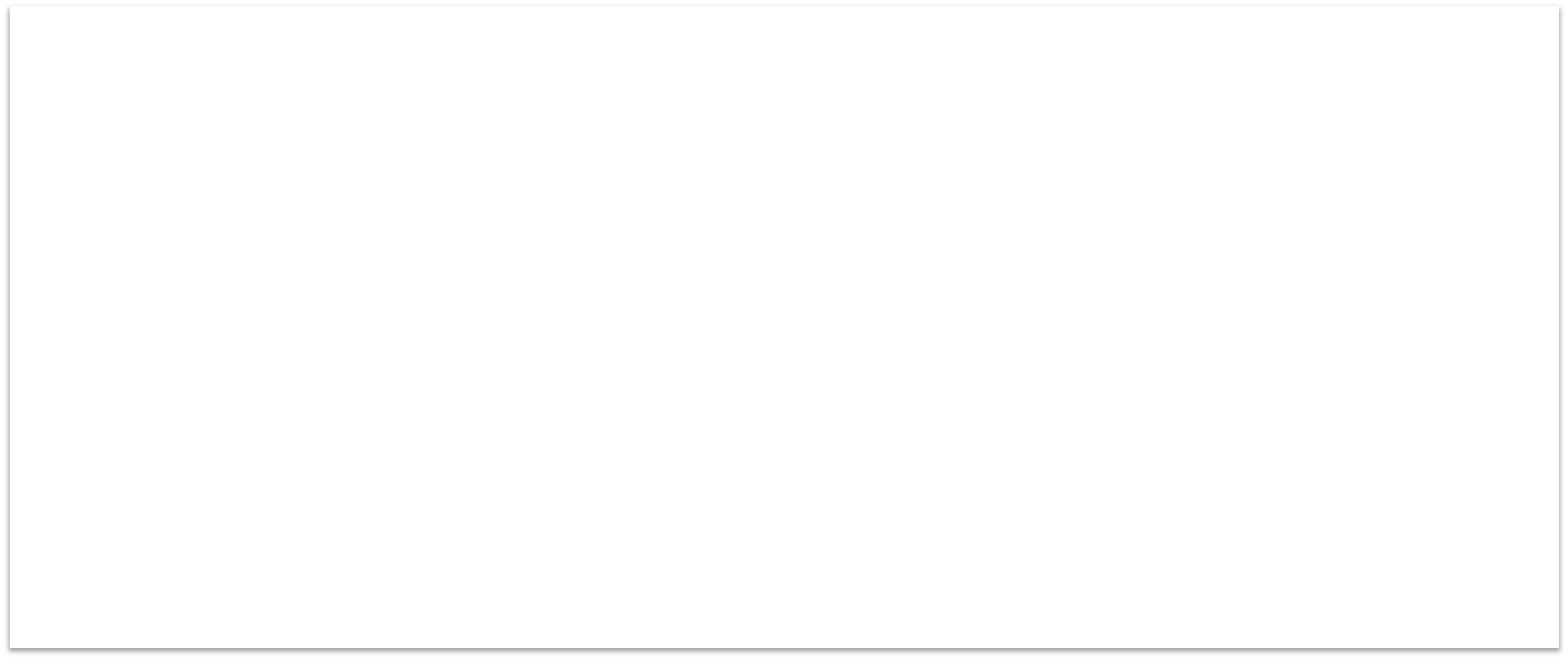
(ignoring λ; if λ = 0)

(*l*) *j i*

@⇥

*ij*

##### Backpropagation



Set ∆(*l*) = 0

*ij*

For each training instance (**x***i*, y*i*):

Set **a**(1) = **x***i*

8l, i, j

(Used to accumulate gradient)

Compute {**a**(2),. .., **a**(*L*)} via forward propagation

Compute **6**(*L*) = **a**(*L*) — y*i*

Compute errors {**6**(*L*—1),. .., **6**(2)}

Compute gradients ∆(*l*) = ∆(*l*) + a(*l*)6(*l*+1)

*ij*

*ij*

*j i*

Compute avg regularized gradient D*ij* =

(*l*)

(

*n*

1 ∆(*l*) + λ⇥(*l*)

*ij*

*ij*

*n*

1 ∆(*l*)

*ij*

if j =6 0

otherwise

***D***(*l*) is the matrix of partial derivatives of *J*(Θ)

Note: Can vectorize

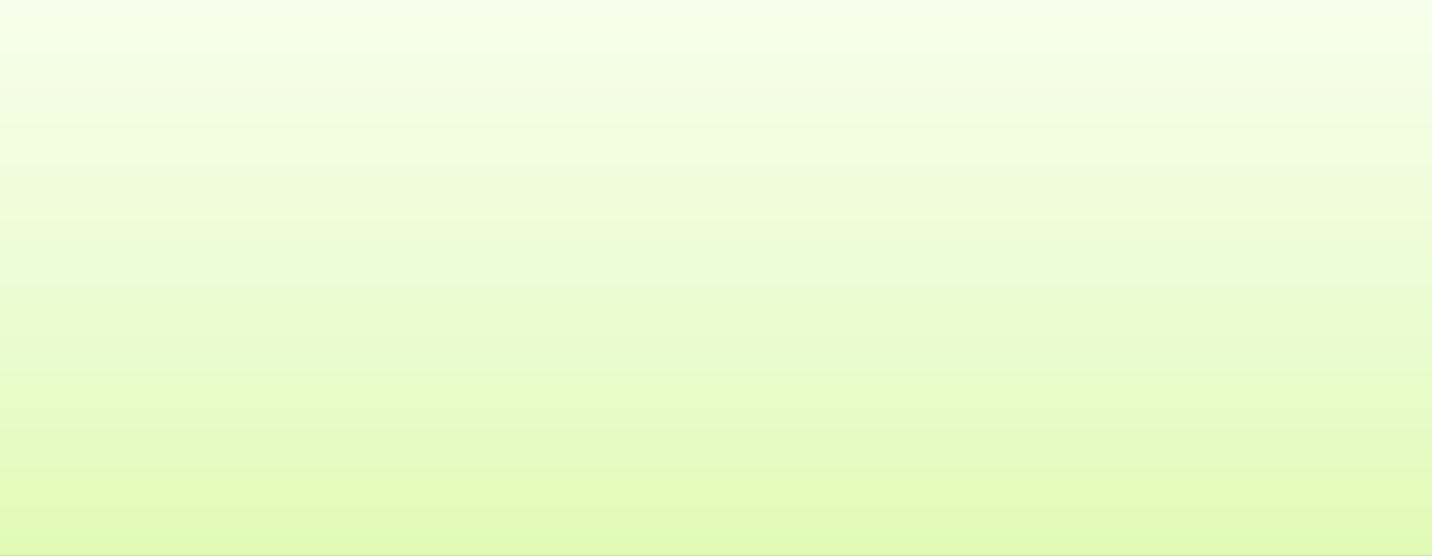
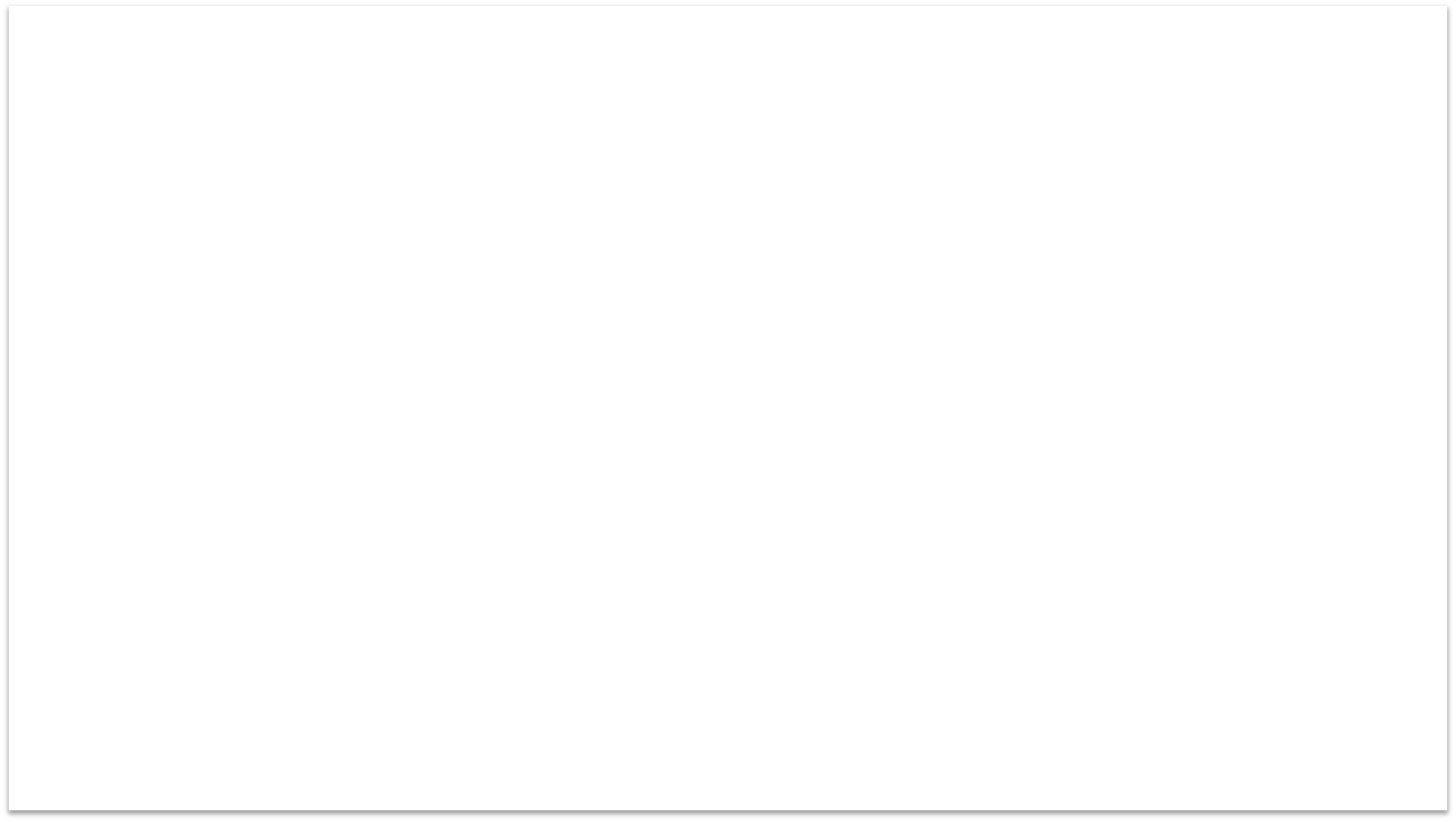
∆(*l*) = ∆(*l*) + a(*l*)6(*l*+1)

as **A**(*l*) = **A**(*l*) + **6**(*l*+1)**a**(*l*)|

*ij ij j i*

Training a Neural Network via Gradient Descent with Backprop

Given: training set (**x**1, y1),. .., (**x***n*, y*n*) Initialize all ⇥(*l*) randomly (NOT to 0!) Loop // each iteration is called an epoch



{ }

Set ∆(*l*)

*ij*

= 0 8l, i, j

(Used to accumulate gradient)

For each training instance (**x***i*, y*i*):

Backpropagation

Set **a**(1) = **x***i*

Compute {**a**(2),. .., **a**(*L*)} via forward propagation Compute **6**(*L*) = **a**(*L*) — y*i*

Compute errors {**6**(*L*—1),. .., **6**(2)}

Compute gradients ∆(*l*)

= ∆(*l*) + a(*l*)6(*l*+1)

*ij ij*

*i*

*j*

(*l*)

Compute avg regularized gradient D

*ij*

( 1 ∆(*l*) + λ⇥(*l*)

if j 6= 0

otherwise

*n ij*

=

*n*

*ij*

*ij*

1 ∆(*l*)

Update weights via gradient step ⇥(*l*)

*ij*

= ⇥(*l*) — ↵D(*l*)

Until weights converge or max #epochs is reached

*ij*

*ij*

Backprop Issues

“Backprop is the cockroach of machine learning. It’s ugly, and annoying, but you just can’t get rid of it.”

@Geoﬀ Hinton

Problems:

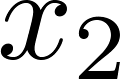
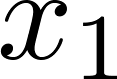
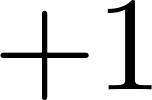
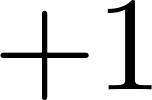
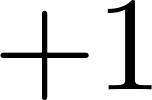
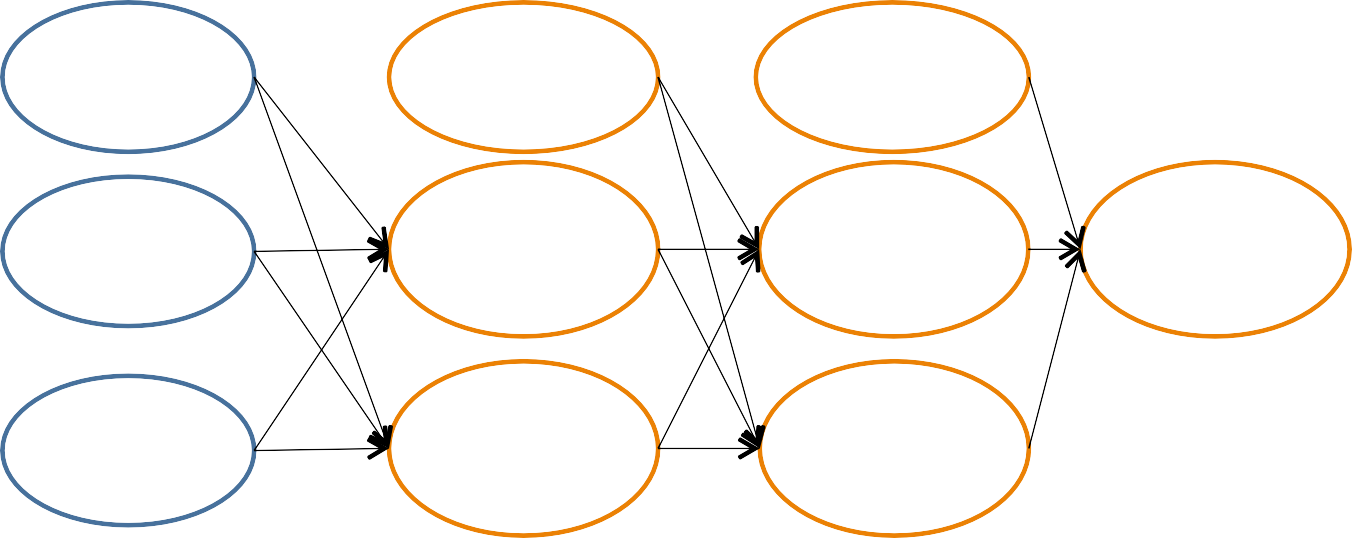
* black box
* local minima

# Implementation Details

### Random Initialization

* Important to randomize initial weight matrices
* Can’t have uniform initial weights, as in logistic regression

– Otherwise, all updates will be identical & the net won’t learn



6(2)

1

6(3)

1

6(4)

1

6(2)

2

6(3)

2

Implementation Details

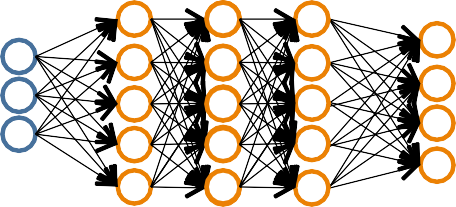
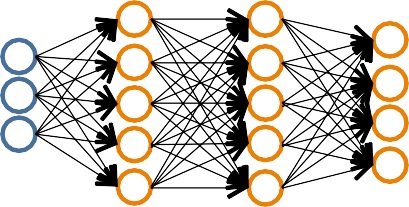
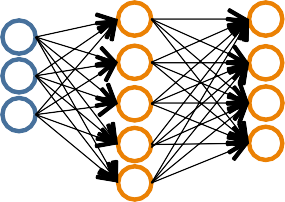
* + For convenience, compress all parameters into **θ**
    - “unroll” Θ(1), Θ(2),... , Θ(L-1) into one long vector **θ**
      * E.g., if Θ(1) is 10 x 10, then the first 100 entries of **θ** contain the value in Θ(1)
    - Use the reshape command to recover the original matrices
      * E.g., if Θ(1) is 10 x 10, then

theta1 = reshape(theta[0:100], (10, 10))

* + Each step, check to make sure that *J*(**θ**) decreases
  + Implement a gradient@checking procedure to ensure that the gradient is correct...

# Putting It All Together

Pick a network architecture (connectivity pattern between nodes)



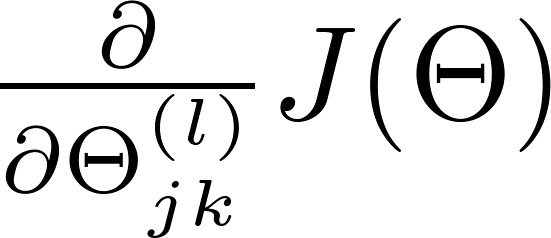
* + # input units = # of features in dataset
  + # output units = # classes

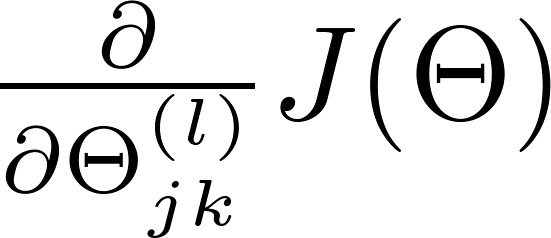
**Reasonable default:** 1 hidden layer

* + or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)

1. Randomly initialize weights
2. Implement forward propagation to get *h*Θ(**x***i*)

for any instance **x***i*

1. Implement code to compute cost function *J*(Θ)
2. Implement backprop to compute partial derivatives
3. Use gradient checking to compare

computed using backpropagation vs. the numerical gradient estimate.

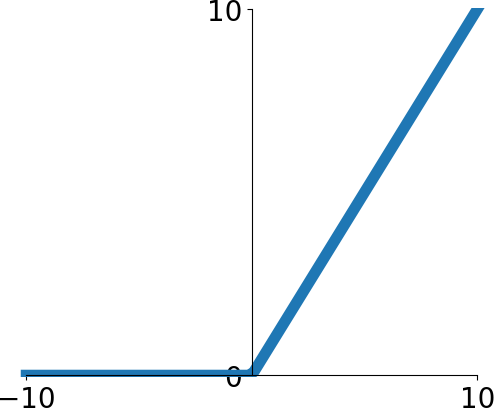
– Then, disable gradient checking code

1. Use gradient descent with backprop to fit the network

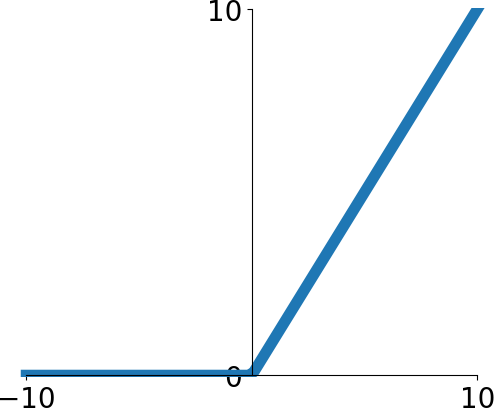
The function

is called “Rectified Linear Unit”

This is called the **activation function** of the neural network



The function

is called “Rectified Linear Unit”

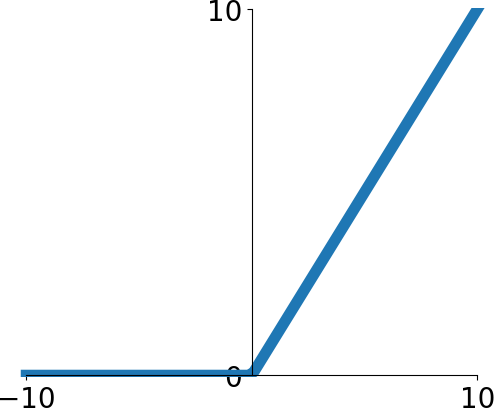
This is called the **activation function** of the neural network

**Q**: What happens if we build a neural network with no activation function?



The function

is called “Rectified Linear Unit”



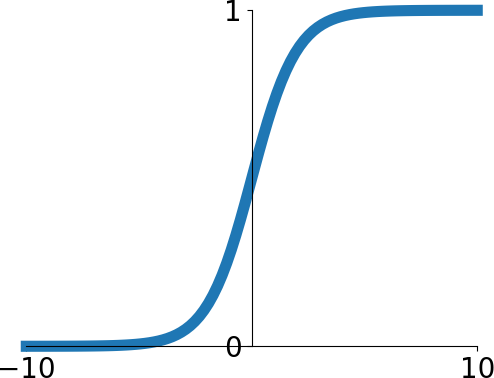
This is called the **activation function** of the neural network

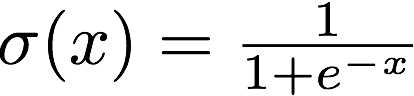
**Q**: What happens if we build a neural network with no activation function?

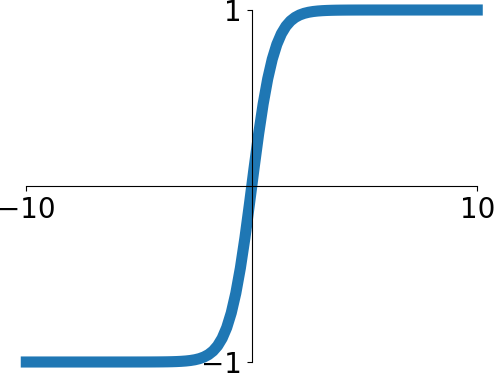


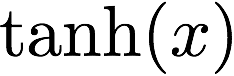
**A**: We end up with a linear classifier!

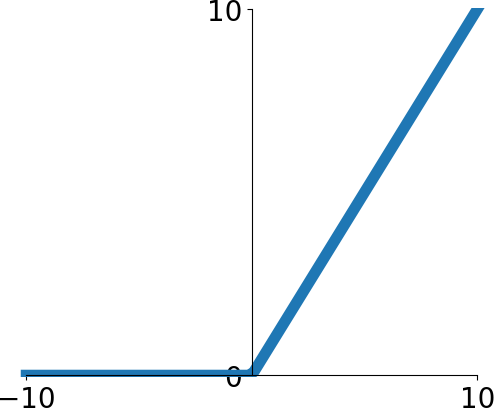
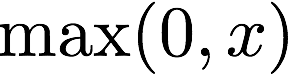
Activation Functions

**Sigmoid**

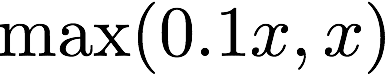


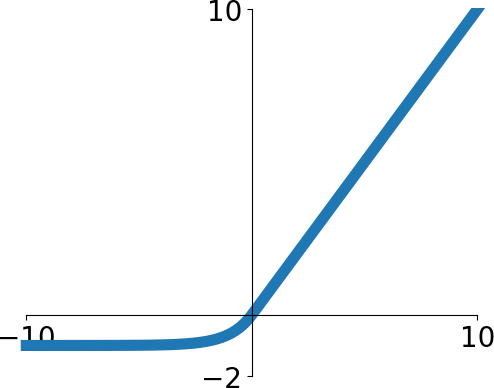
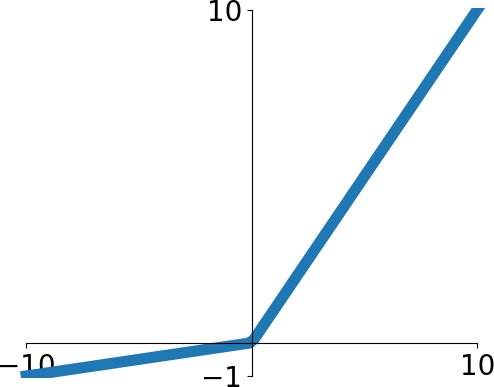
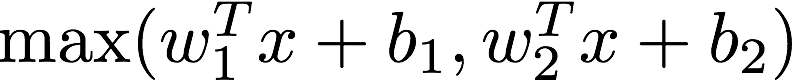
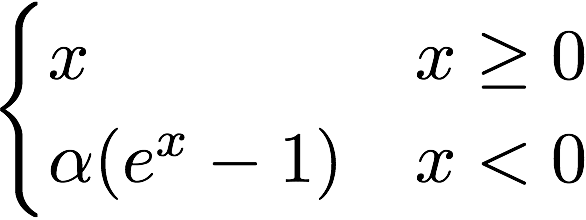
**tanh**



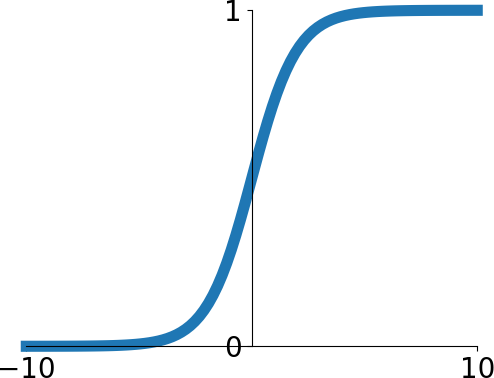
**ReLU**

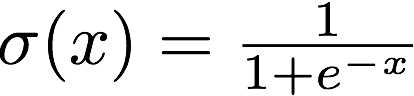
**Leaky ReLU**

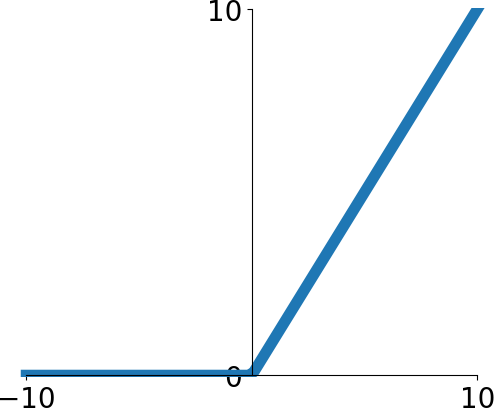
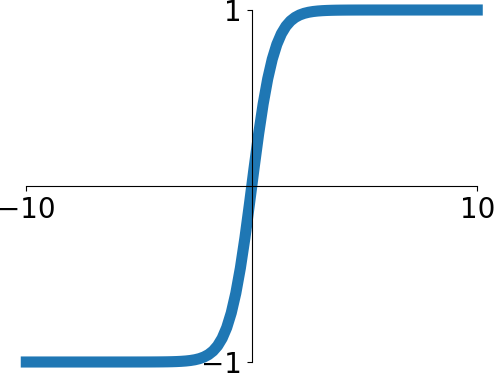


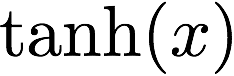
**Maxout ELU**

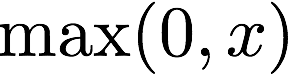
Activation Functions

**Sigmoid**



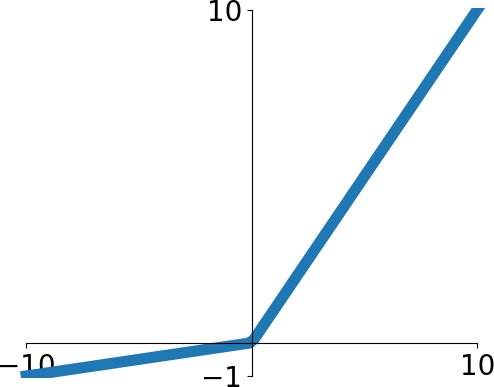
**tanh**

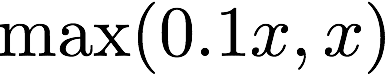


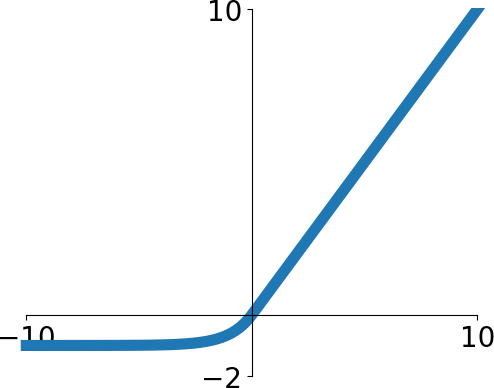
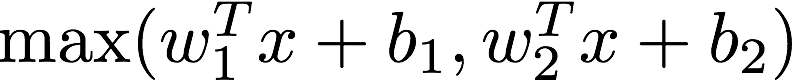
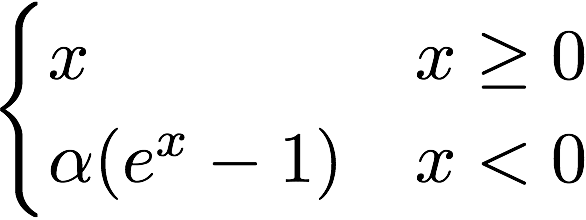


**ReLU**

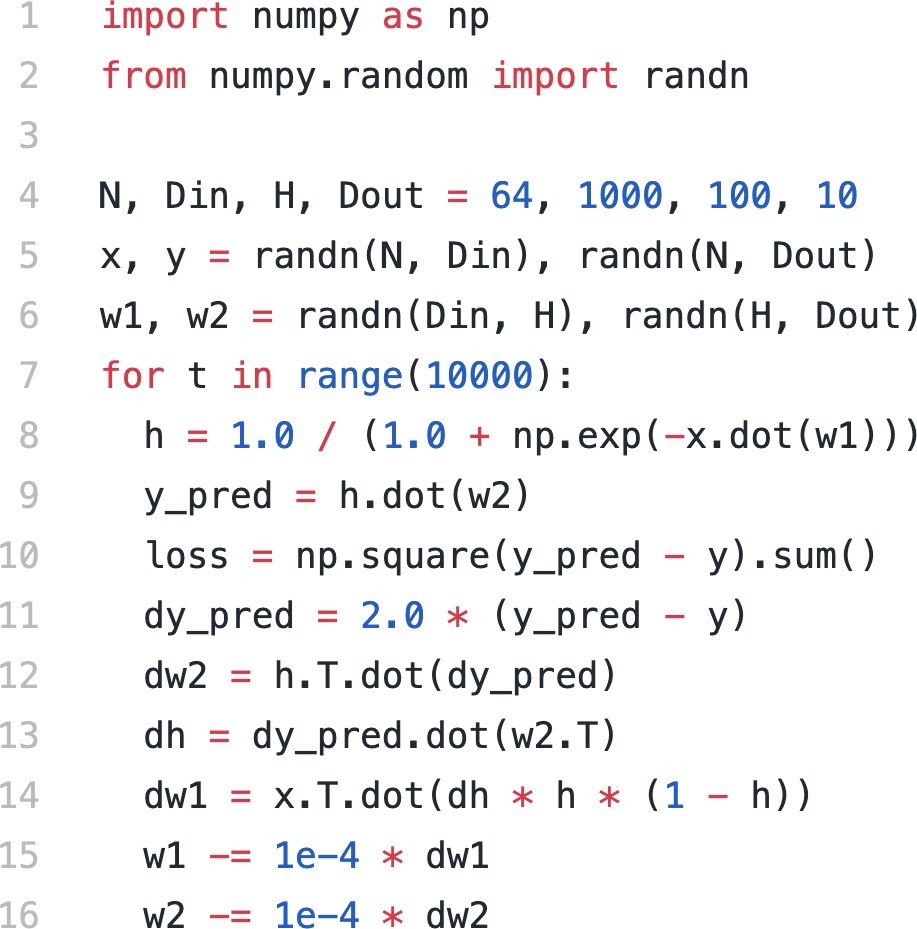
ReLU is a good default choice for most problems

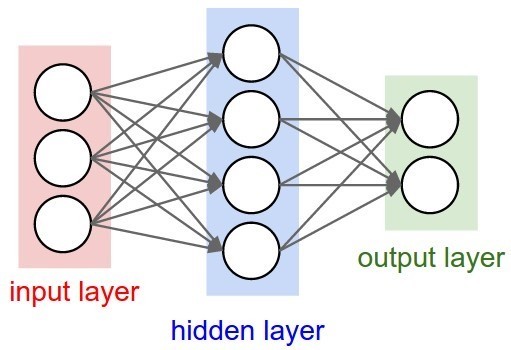
**Leaky ReLU**

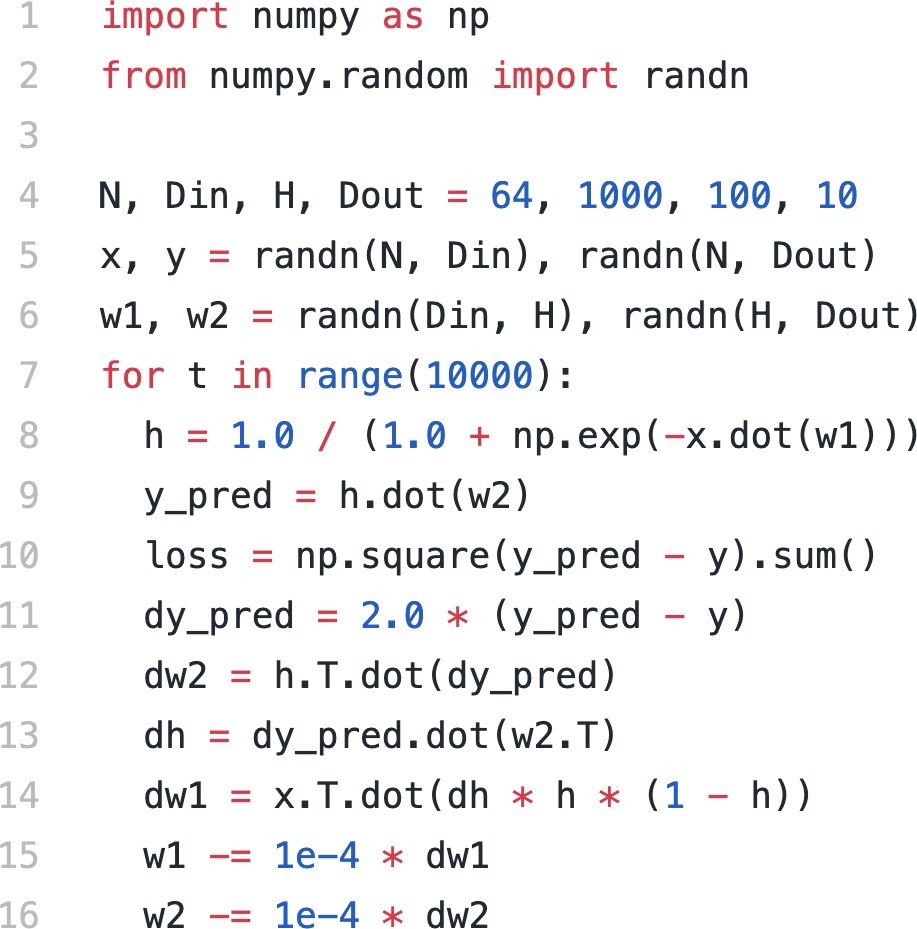


**Maxout ELU**

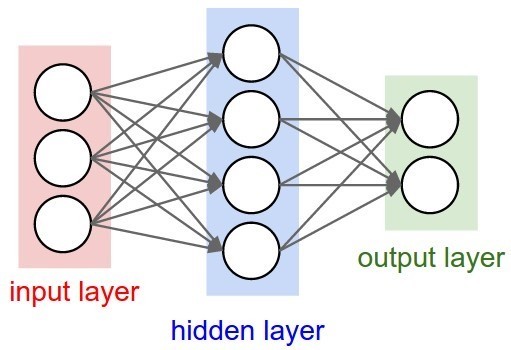
67

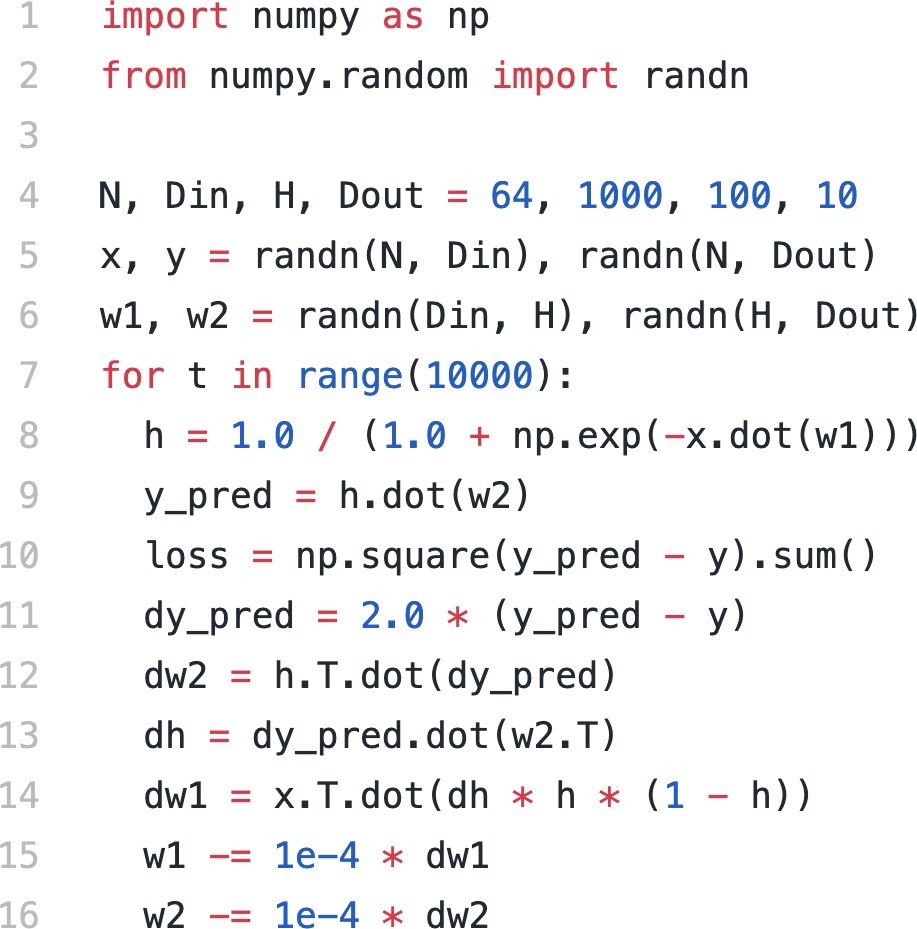


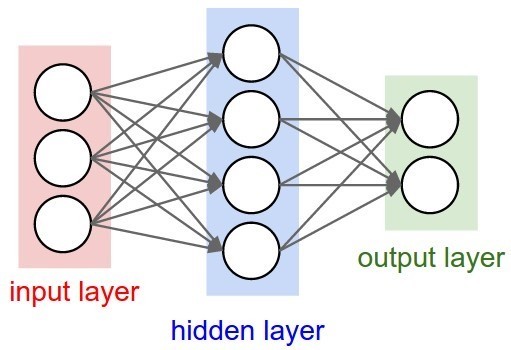




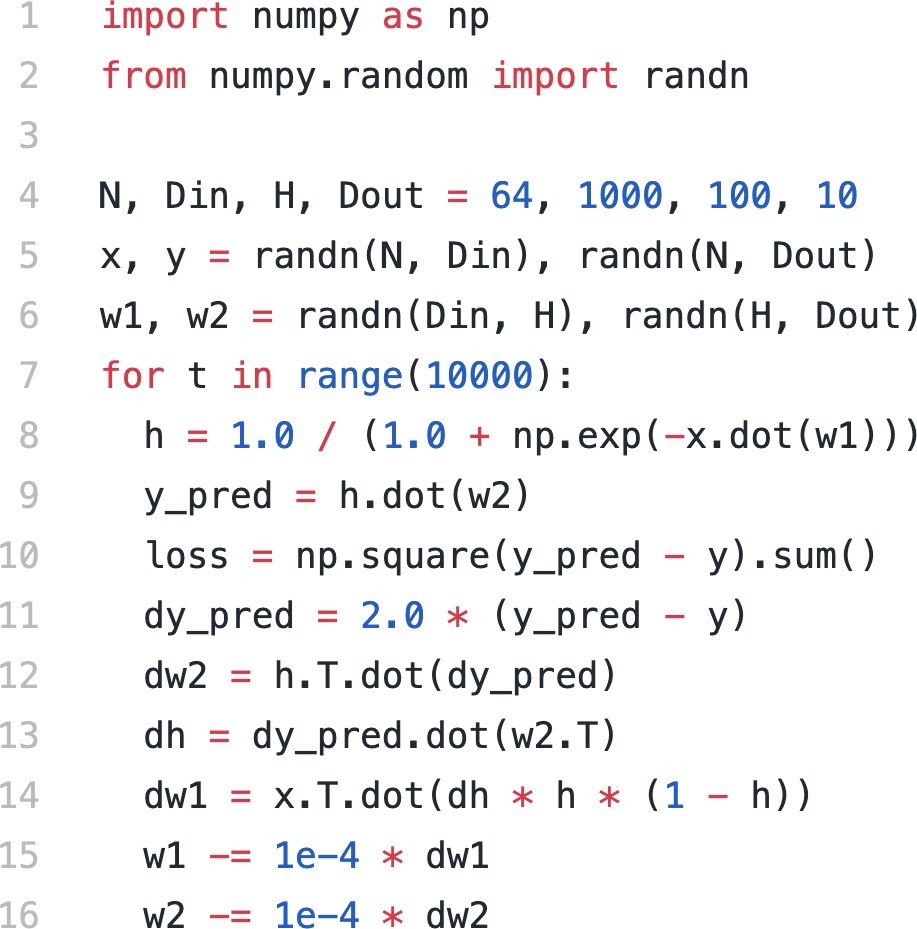
Initialize weights and data

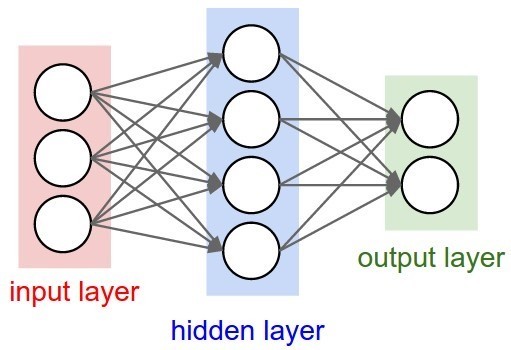




Initialize weights and data

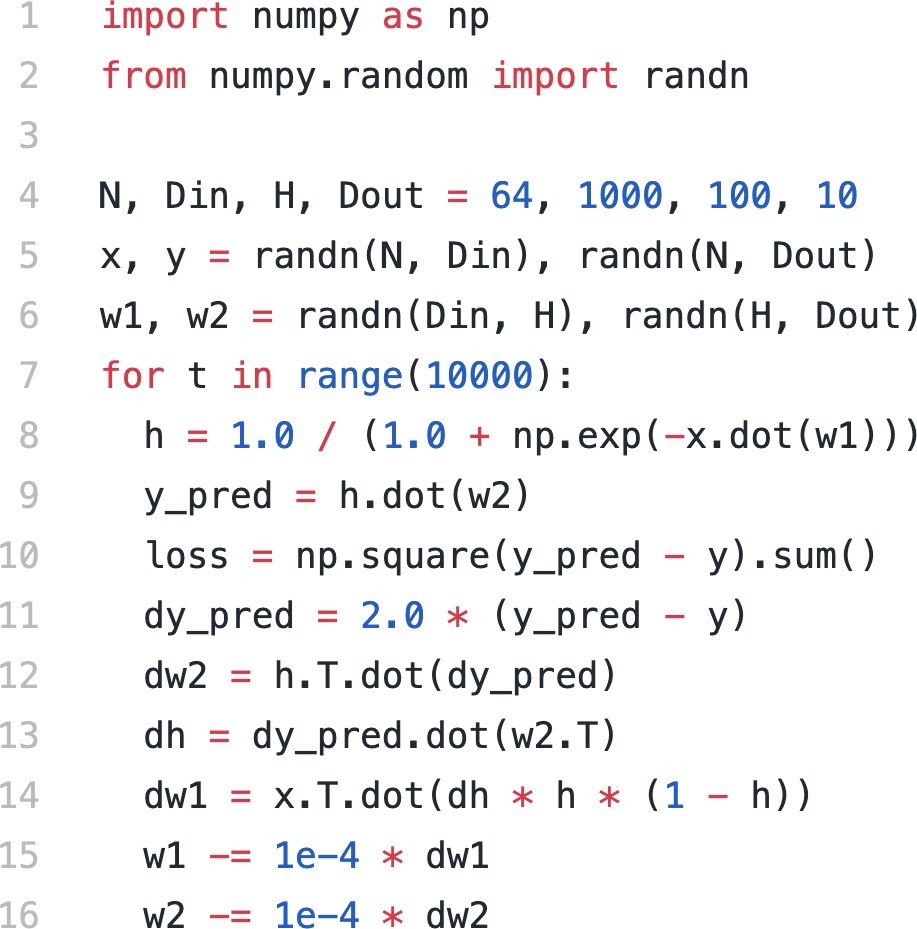
Compute loss (sigmoid activation, L2 loss)

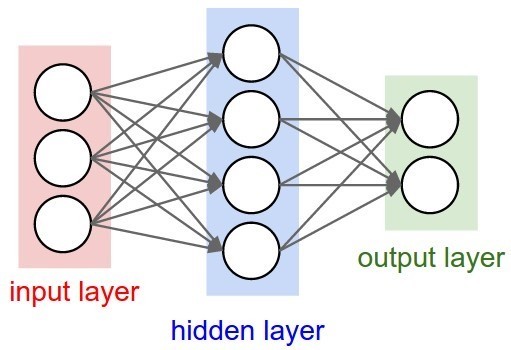


Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

Compute gradients



Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

Compute gradients

SGD

step