PHY654

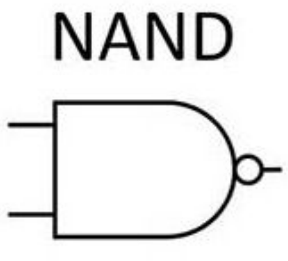
Machine learning (ML) in particle physics

Swagata Mukherjee • IIT Kanpur

17th August 2024

Example code for binary classification

NAND Gate using logistic regression
https://github.com/swagata87/IITKanpurPhy654/blob/main/NAND_gate_Logistic_regression.ipynb



Two features | y

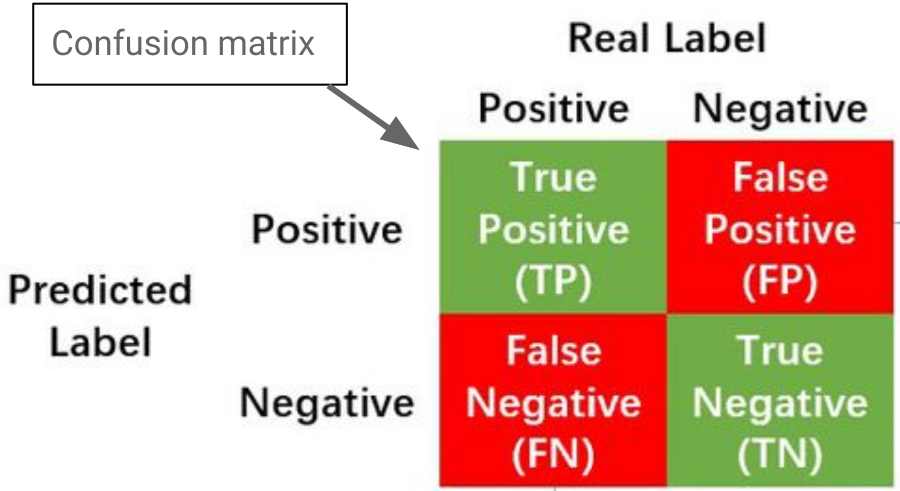
A	B	Output
0	0	1
1	0	1
0	1	1
1	1	0

Try to rewrite the code without using sklearn library

Four training examples
m=4

Useful metrics for evaluating ML models

Confusion matrix



		Real Label	
		Positive	Negative
Predicted Label	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

$$\text{Precision} = \frac{\sum \text{TP}}{\sum \text{TP} + \text{FP}}$$

Generally used when it is very important for positive predictions to be accurate.

$$\text{F1} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

$$\text{Recall} = \frac{\sum \text{TP}}{\sum \text{TP} + \text{FN}}$$

$$\text{Accuracy} = \frac{\sum \text{TP} + \text{TN}}{\sum \text{TP} + \text{FP} + \text{FN} + \text{TN}}$$

Not a good measure if the dataset is not balanced

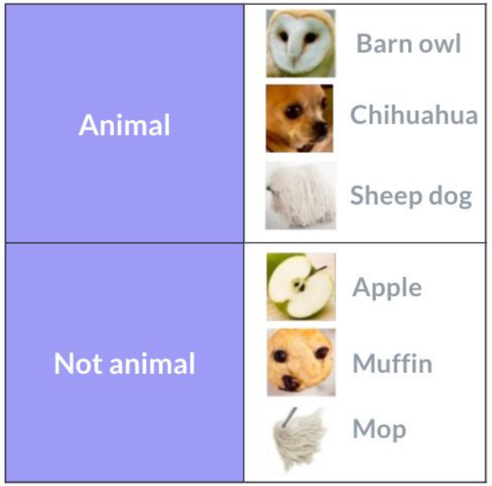
An example

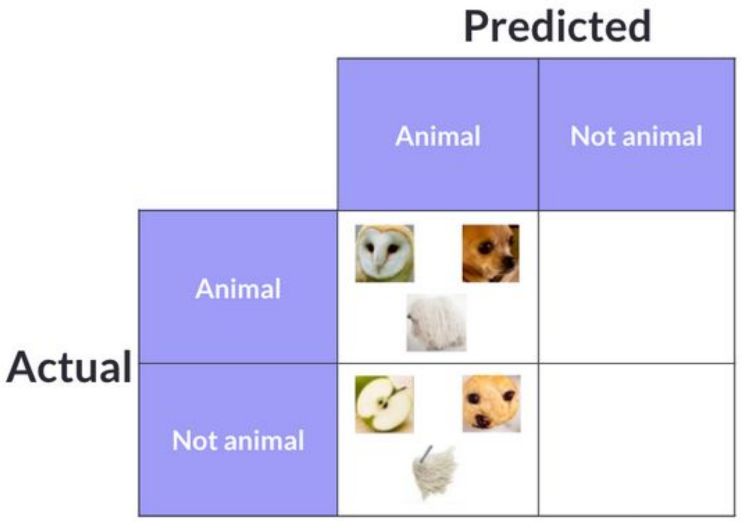
Animal		Barn owl
		Chihuahua
		Sheep dog
Not animal		Apple
		Muffin
		Mop

		Predicted	
		Animal	Not animal
Actual	Animal	True Positives	False Negatives
	Not animal	False Positives	True Negatives

Confusion matrix

Look at all these metrics


Animal
Barn owl
Chihuahua
Sheep dog
Not animal
Apple
Muffin
Mop

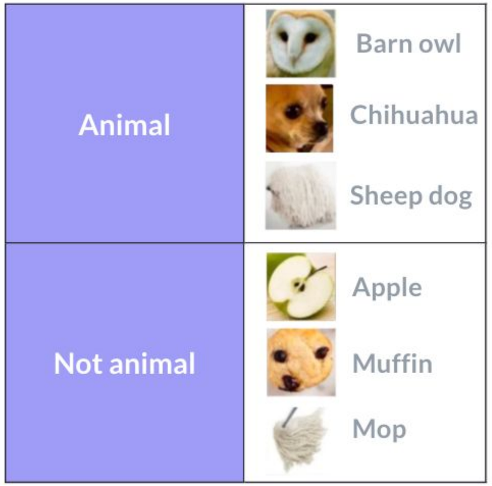


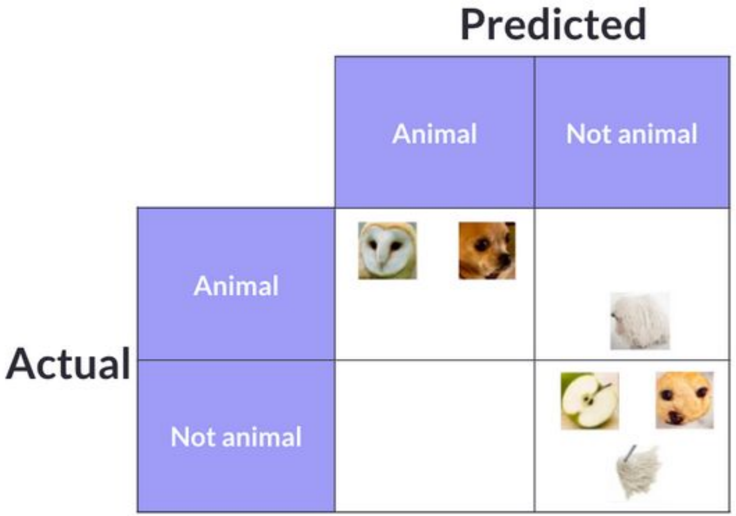
Model 1

True Positives	3
True Negatives	0
False Positives	3
False Negatives	0

Accuracy	50%	$\frac{3+0}{3+0+0+3}$
Precision	50%	$\frac{3}{3+3}$
Recall	100%	$\frac{3}{3+0}$
F1 score	67%	$2 \cdot \frac{0.5 \cdot 1}{0.5+1}$

Look at all these metrics





Model 3

True Positives	2
True Negatives	3
False Positives	0
False Negatives	1

Accuracy	83%	$\frac{2+3}{2+3+1+0}$
Precision	100%	$\frac{2}{2+0}$
Recall	67%	$\frac{2}{2+1}$
F1 score	80%	$2 \cdot \frac{1 \cdot 0.67}{1+0.67}$

These metrics are calculated at a single classification threshold value (predicted output > 0.5 or not).
But if you want to evaluate a model's quality across all possible thresholds, you need different tools.

ROC

Receiver-operating characteristic curve (ROC)

ROC curve is drawn by computing true positive rate (TPR) and false positive rate (FPR) at every possible threshold (in selected intervals), then graphing TPR vs FPR.

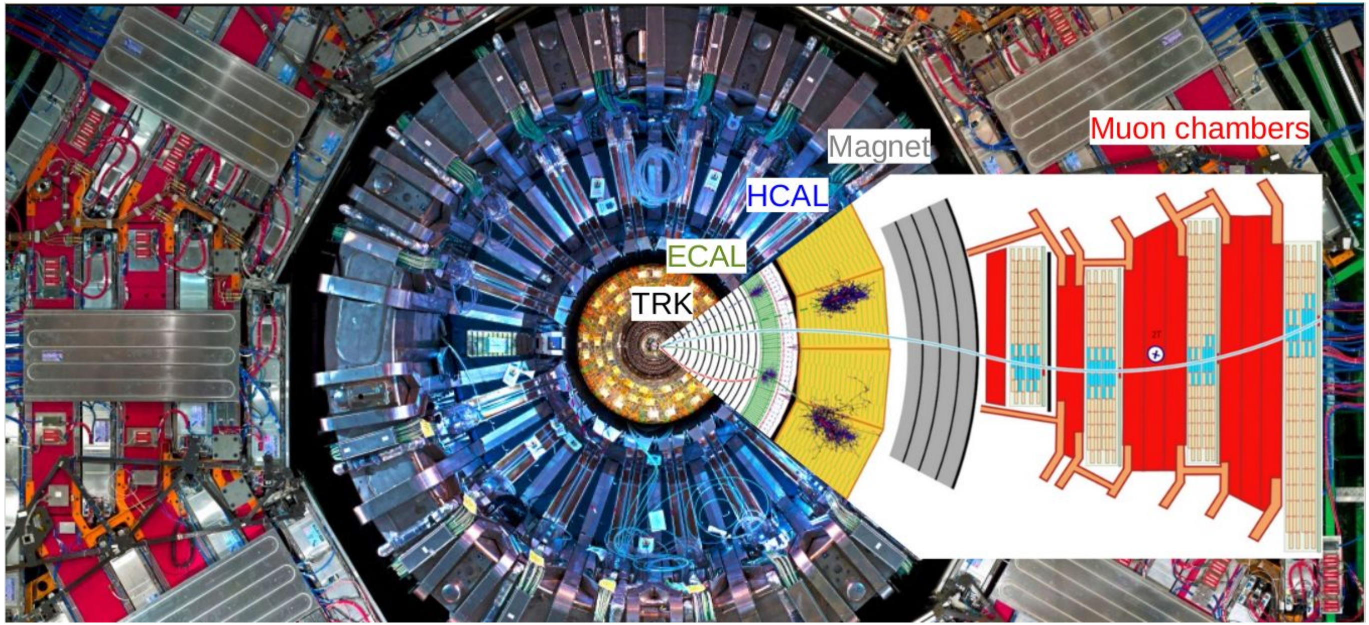
False Positive Rate = False Positives / (False Positives + True Negatives)
True Positive Rate = True Positives / (True Positives + False Negatives)

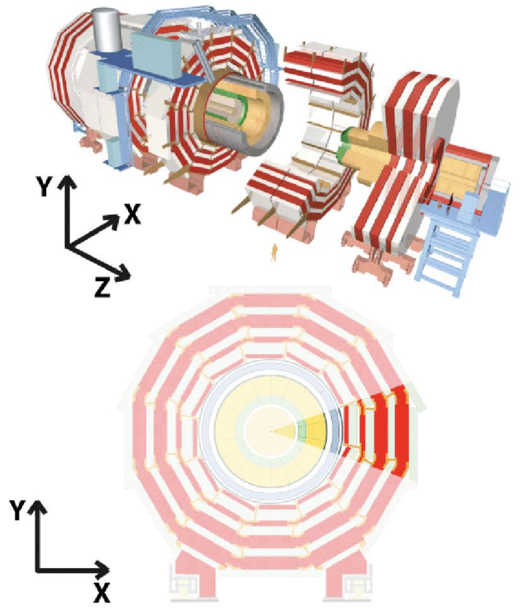
In particle physics, we generally plot **signal efficiency vs background efficiency;** or signal efficiency vs background rejection.

ROC is not specific to ML. It can be plotted for cut-based physics analysis as well. It tells us the usefulness of a cut applied on a variable.

The area under the curve (AUC) can be used as a single-valued metric.

Binary classification in physics experiments



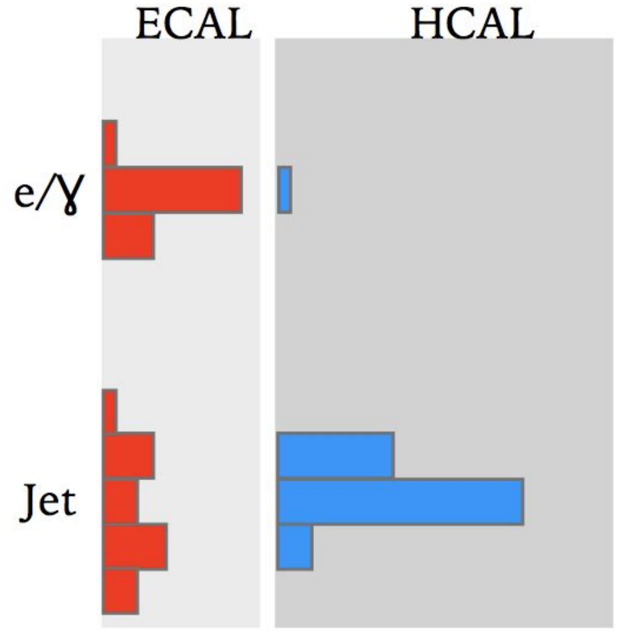


Photon vs hadronic-jet : Binary classification

Photon is signal (y=1) and Hadronic-jet is background (y=0)

Feature 1 : H/E

Photon vs hadronic-jet : Binary classification

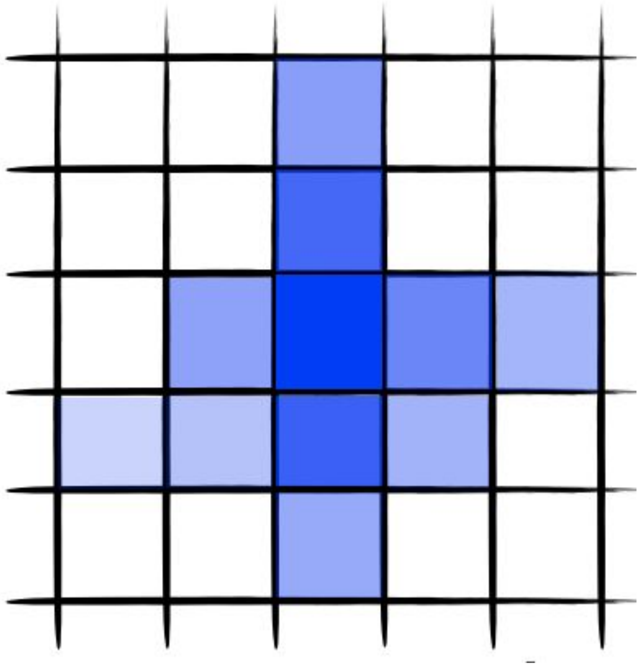


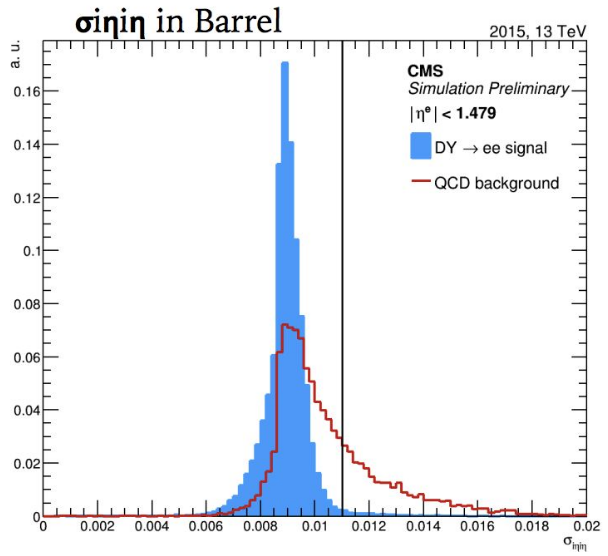
H/E variable is a good discriminator variable.

Feature 2 : Showershape in ECAL

Photon vs hadronic-jet : Binary classification

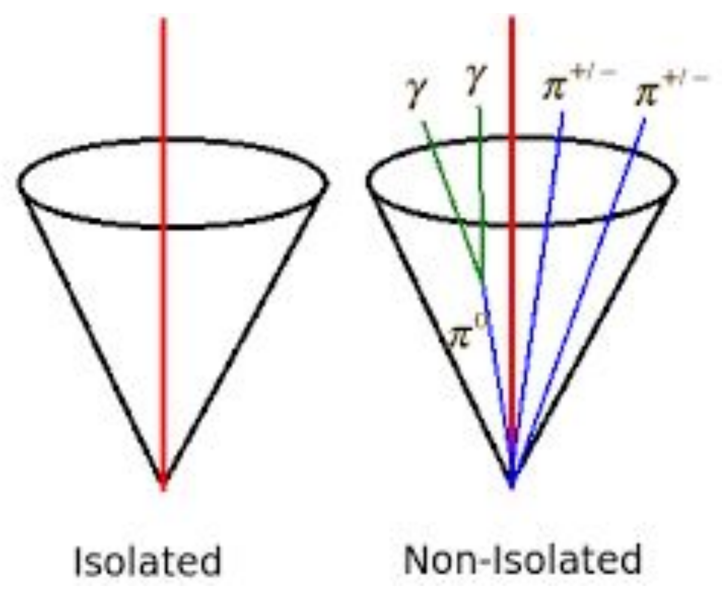
Showershape is also a good discriminator between signal and background





Feature 3 : Isolation sum

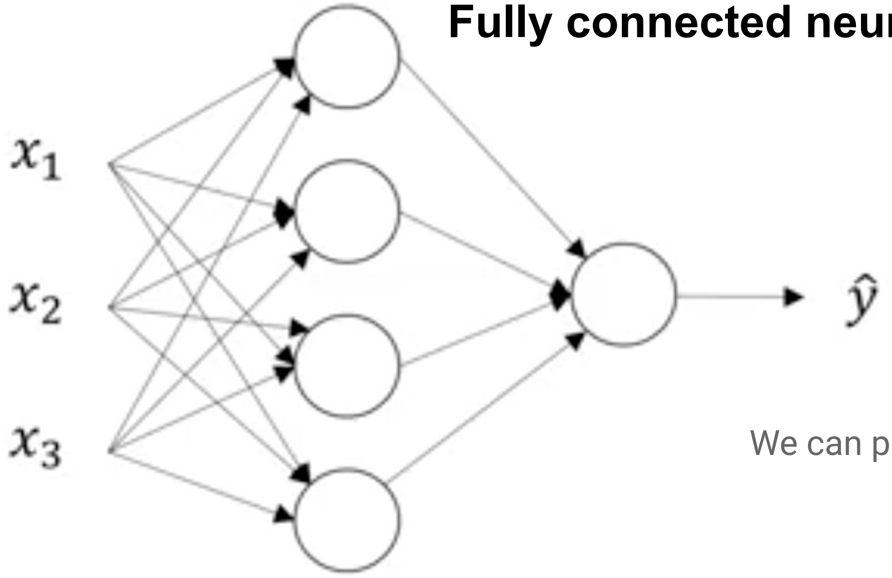
Photon vs hadronic-jet : Binary classification



Isolation sum is also a good discriminator between signal and background

Example use-case in physics experiments

Fully connected neural network



H/E

Showershape in ECAL

Isolation sum

Photon vs hadronic-jet?

We can plot y_predicted for y=0 and y=1.

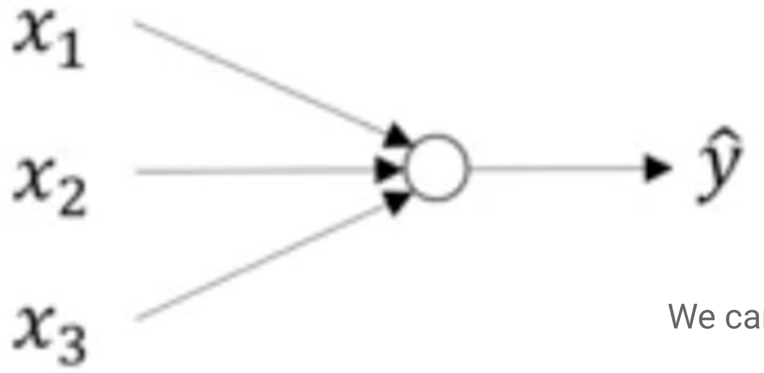
Example use-case in physics experiments

Logistic regression network

H/E

Showershape in ECAL

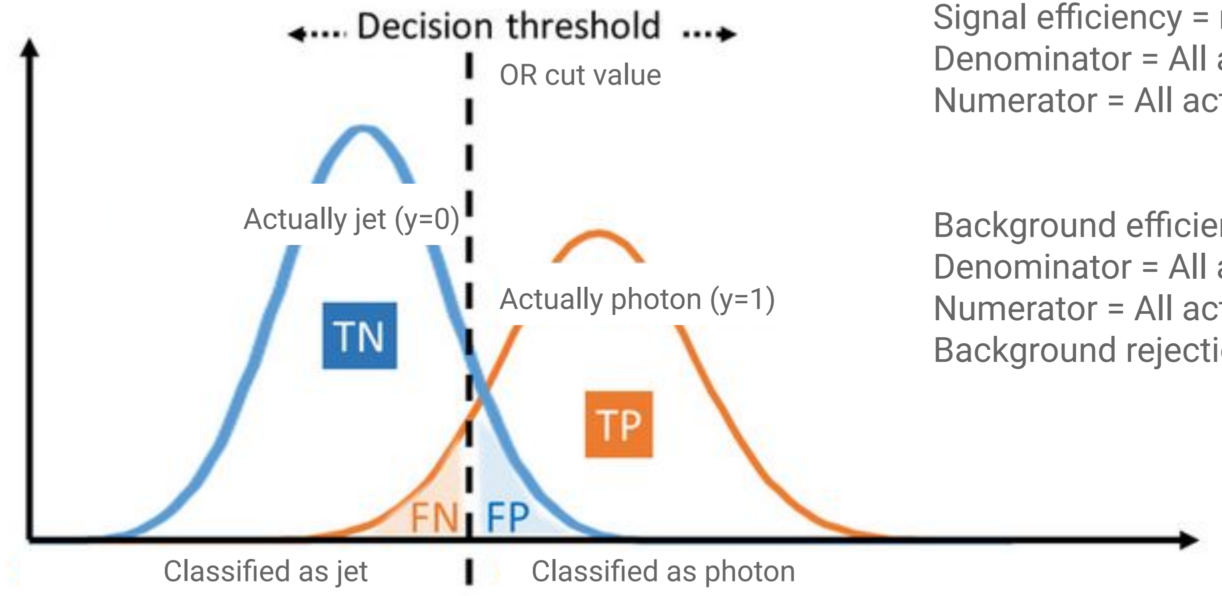
Isolation sum



Photon vs hadronic-jet?

We can plot y_predicted for y=0 and y=1.

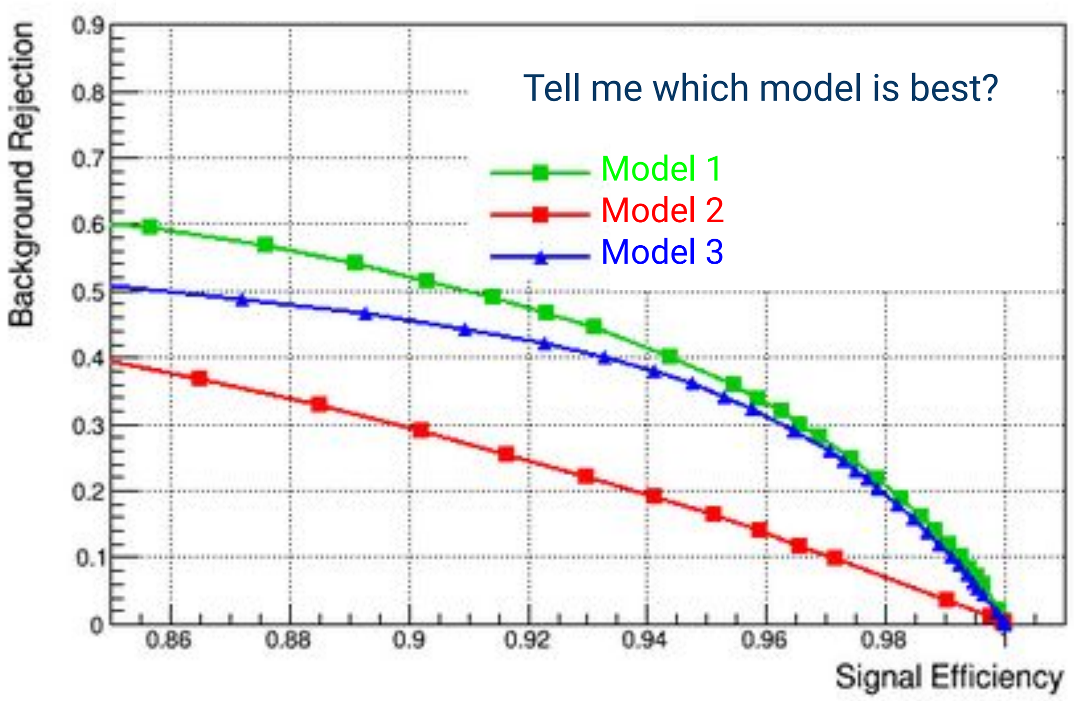
Distribution of predicted output



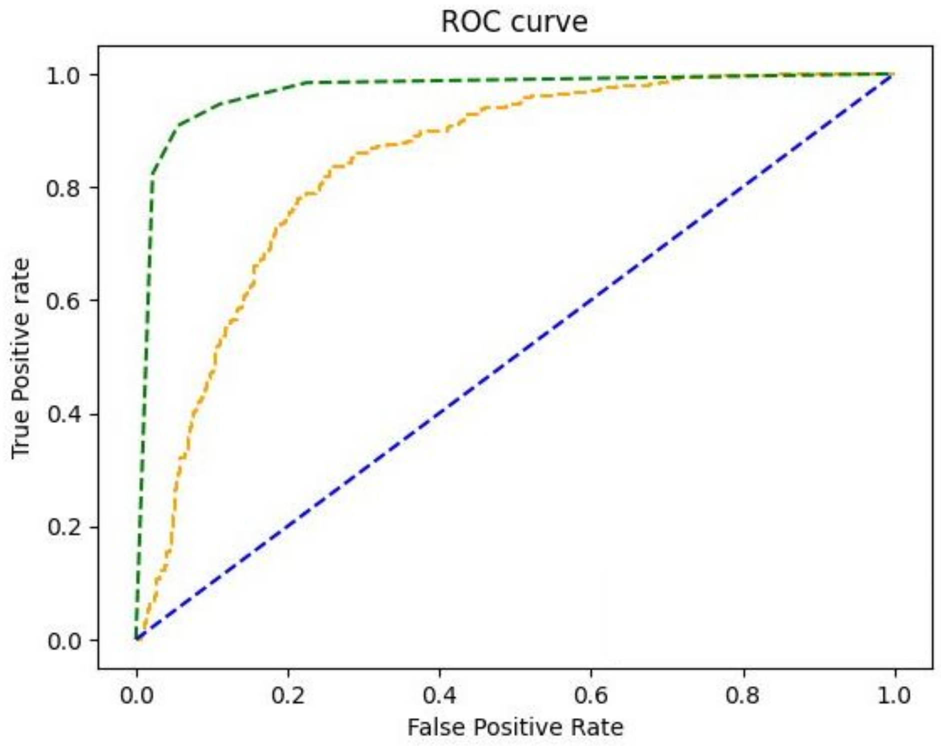
Signal efficiency = numerator / denominator
Denominator = All actual photons
Numerator = All actual photons that pass the cut

Background efficiency = numerator / denominator
Denominator = All actual jets
Numerator = All actual jets that pass the cut
Background rejection = 1 - background efficiency

ROC: an example



ROC: another example



Tell me which model is best?

Some useful links

- Python Tutorial (https://docs.python.org/3.7/tutorial/index.html): an introduction to the Python programming language
- Google Colab (https://colab.research.google.com/): for Python development in your web-browser
- numpy (https://numpy.org/doc/stable/user/quickstart.html): a widely used library for mathematical operations in Python
- Keras (https://keras.io/): a beginner-friendly deep learning library
- Tensor Flow (https://www.tensorflow.org/): a useful backend for deep learning development
- SciKit Learn (https://scikit-learn.org/stable/): helpful machine learning library
- Seaborn (https://seaborn.pydata.org/): a library for creating graphs and figures

A non-physics example of classification

→ Input image →

Binary classifier
1 (cat) vs 0 (non cat)

What kind of data we will be dealing with?

Structured data

H/E	isolation	shower-shape	Matched tracks?	Photon or jet
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-

Unstructured data

Image

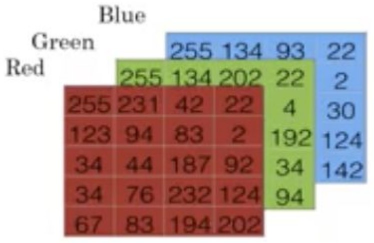
Audio

Images as input to a Neural Network?

→ Input image →

Binary classifier
1 (cat) vs 0 (non cat)

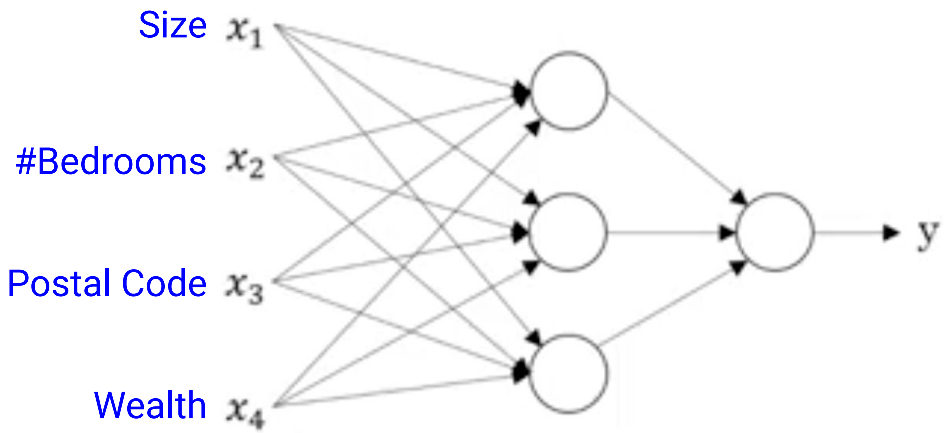
How does a computer "**see**" an image?
→ It sees 3 matrices. These are pixel-intensity values.



→ Each element is a feature. We need to unroll to a column vector.

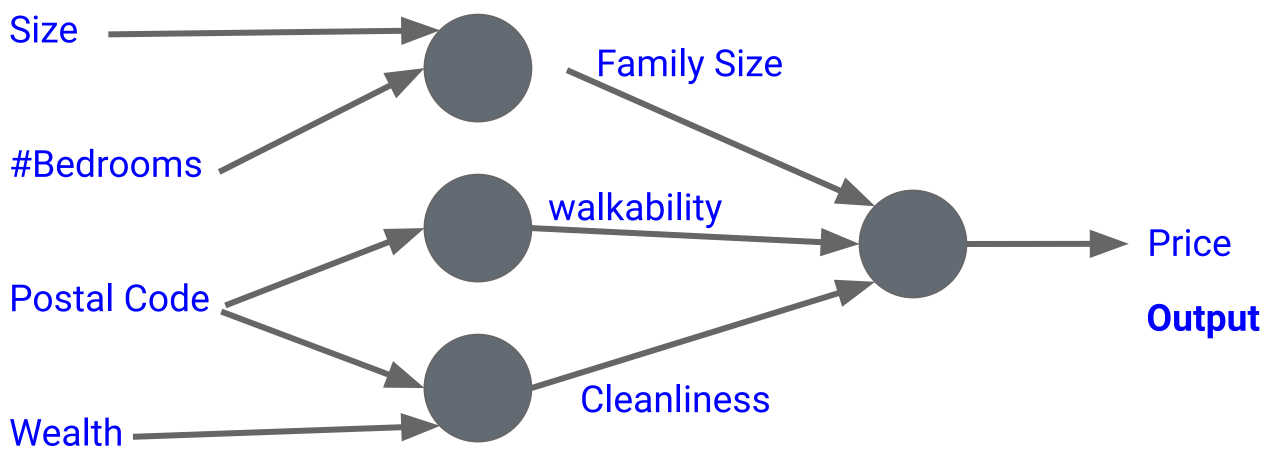
If input image is 64x64 then number of features=64x64x3 = 12288.
So, n=12288.

Hidden layer: what does it do?



Example (regression):
Housing price prediction, based on 4 input variables.

Hidden layer: what does it do?



Example (regression):
Housing price prediction, based on 4 input variables.

Input features (X)

You will provide X and Y for some training examples, and the hidden layer(s) will figure out all the other things by itself.

Fully connected or densely connected

This is what we will actually implement

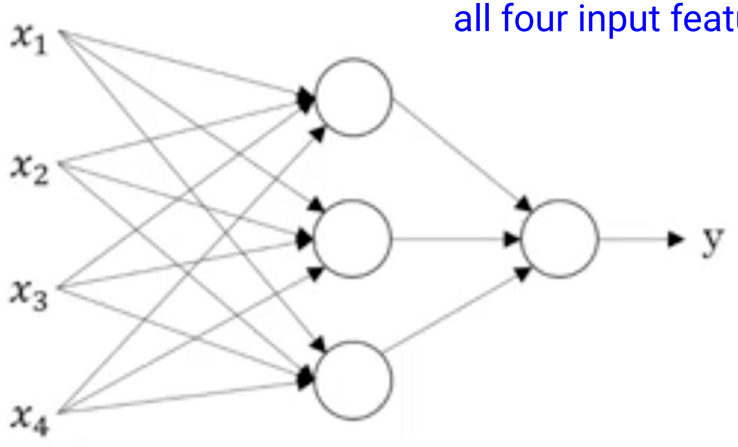
Each **hidden unit** takes all four input features

Size

#Bedrooms

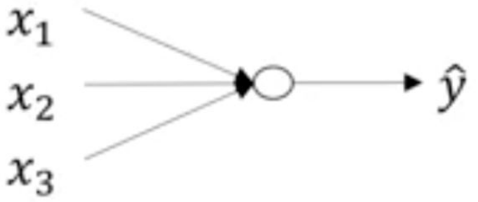
Postal Code

Wealth

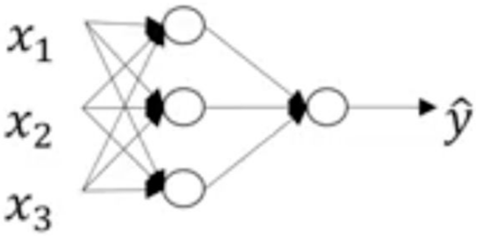


Instead of we deciding that the last hidden unit is “cleanliness”, which depends only on postal code and wealth, we leave it on the neural network to decide what the last hidden unit would be. We just provide it with all input features.

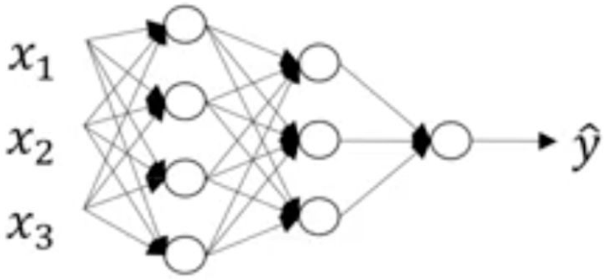
neural network: shallow vs deep



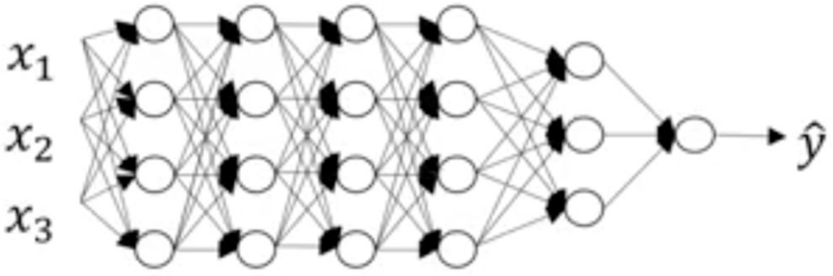
logistic regression



1 hidden layer



2 hidden layers

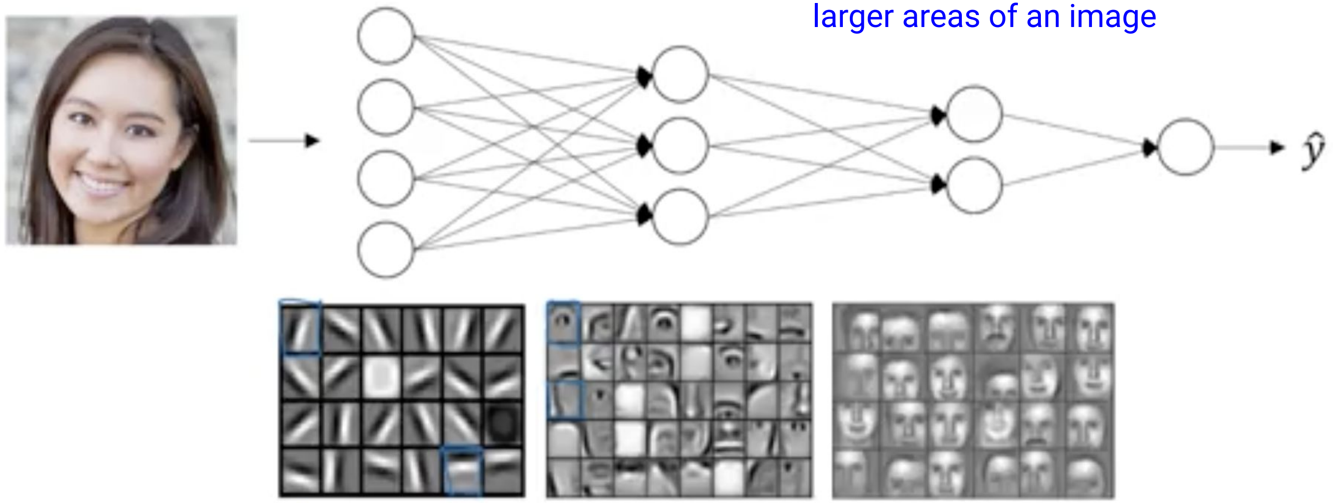


5 hidden layers

Why deep?

Earlier layers learning simple features, later layers learn more complex features.

Earlier layers focus on small areas of an image. Later layers focus on larger areas of an image



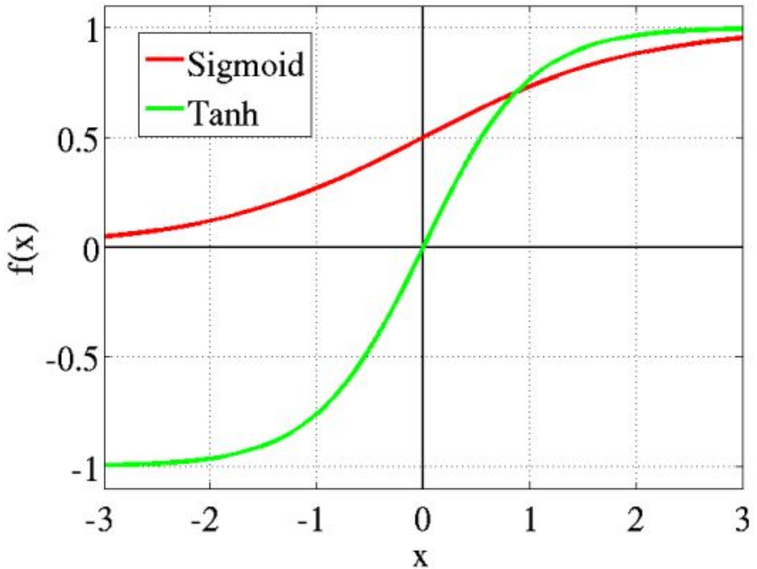
1st layer may be detecting edges

2nd layer may be detecting parts of faces (eye, nose etc)

3rd layer may be detecting different types of faces

Activation function

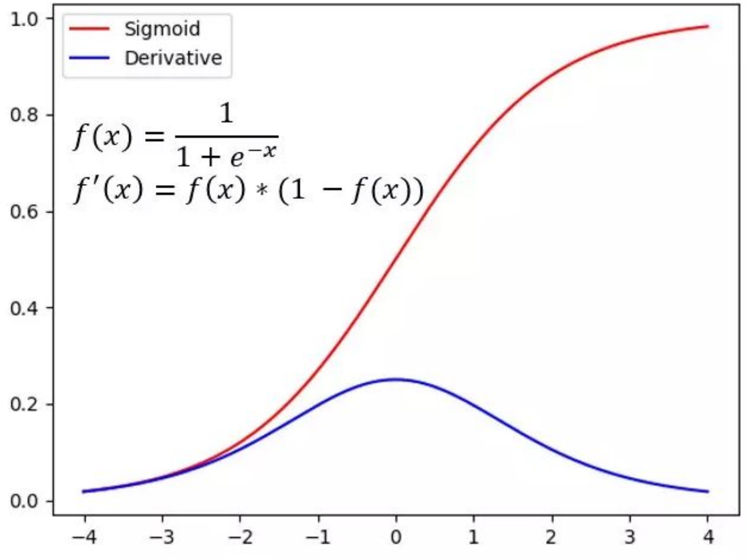
We have been using **sigmoid** so far. But there are other options too. Example **tanh** (sometimes works better than sigmoid in hidden layers).



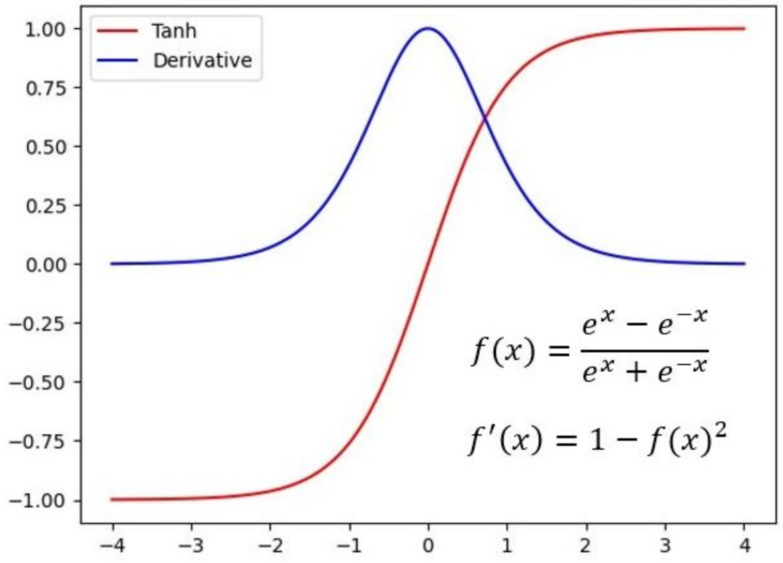
Sigmoid and tanh have issues for values too large or too small, because slope becomes close to zero and gradient descent is very slow.

Vanishing gradient problem.

Activation function



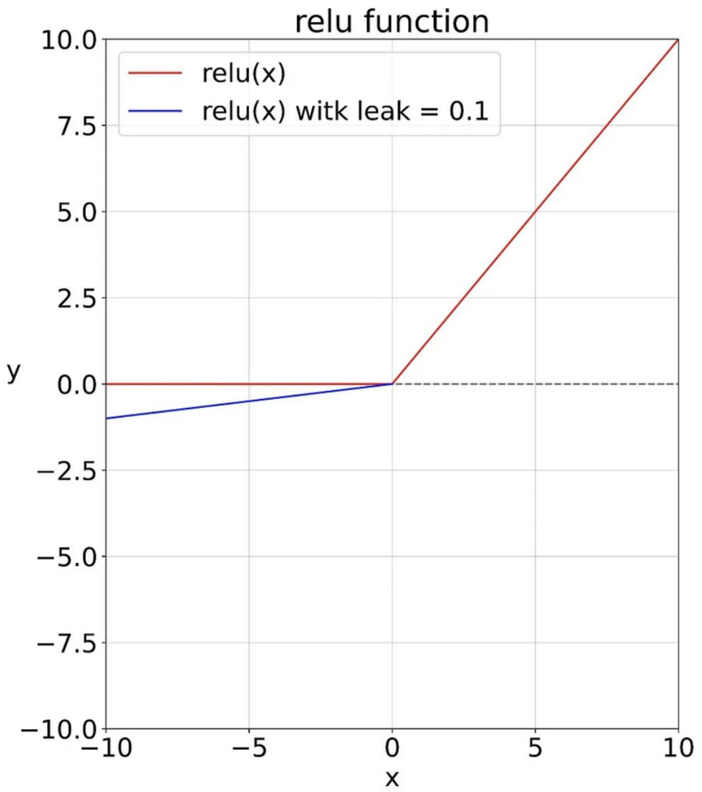
Sigmoid function (red) and first derivative(blue)



Tangent hyperbolic function(red) and first derivative (blue)

Activation function

ReLU or **Leaky ReLU** can be used to avoid vanishing gradient issue



Extra slides

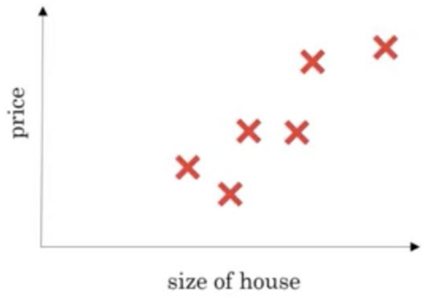
https://developers.google.com/machine-learning/crash-course/classification/accuracy

https://www.geeksforgeeks.org/confusion-matrix-machine-learning/

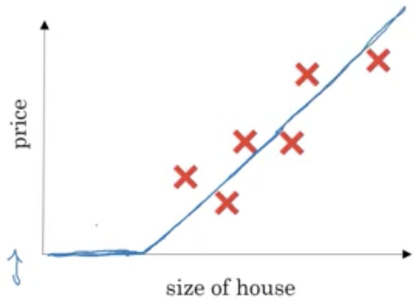
https://www.statology.org/sklearn-classification-report/

The term deep learning refers to training neural networks.
What exactly is a neural network?

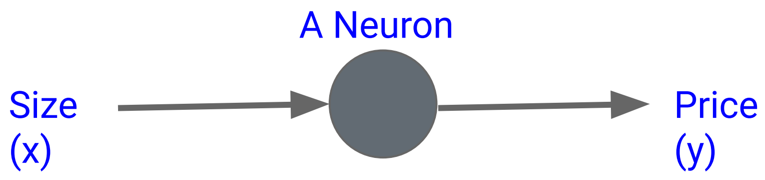
Let's take an example of Housing price prediction.
Say we have a dataset with 6 houses, we know the size of the houses and their price.
And we want to **fit a function** to predict price of a house as a function of its size.

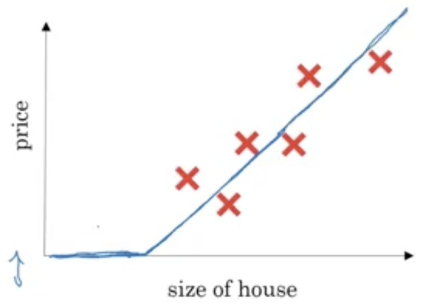


You can fit a straight line. But since price can't be negative we actually draw the following



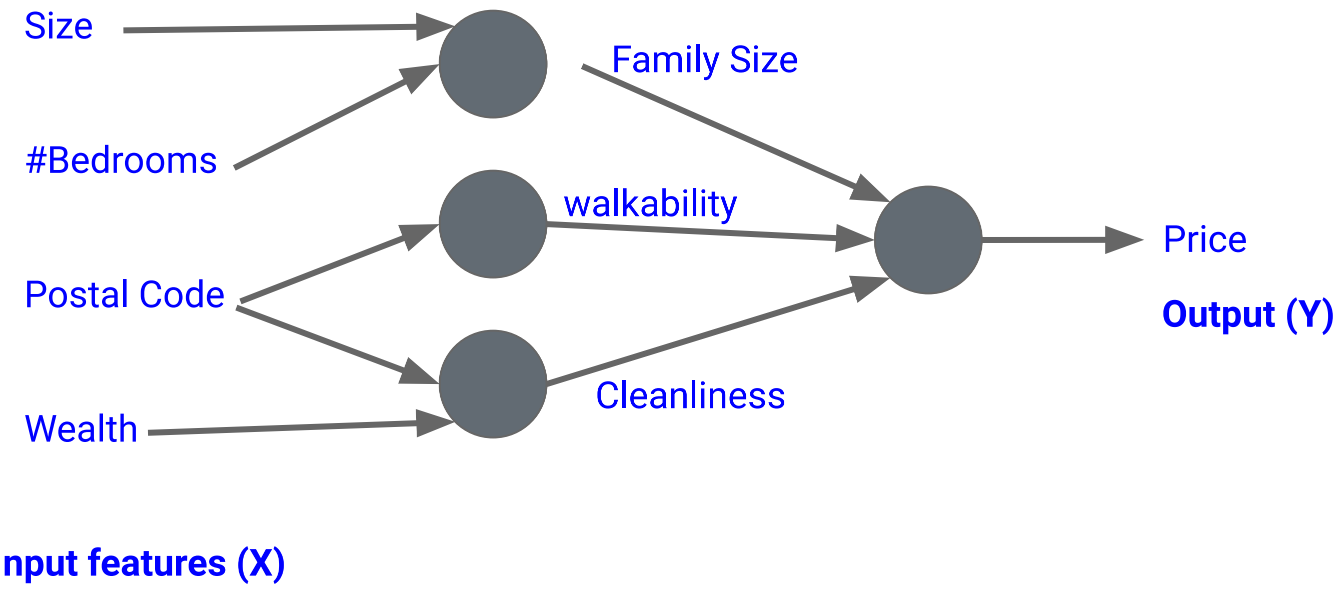
You can think of this function as a neural network (the simplest possible neural network)



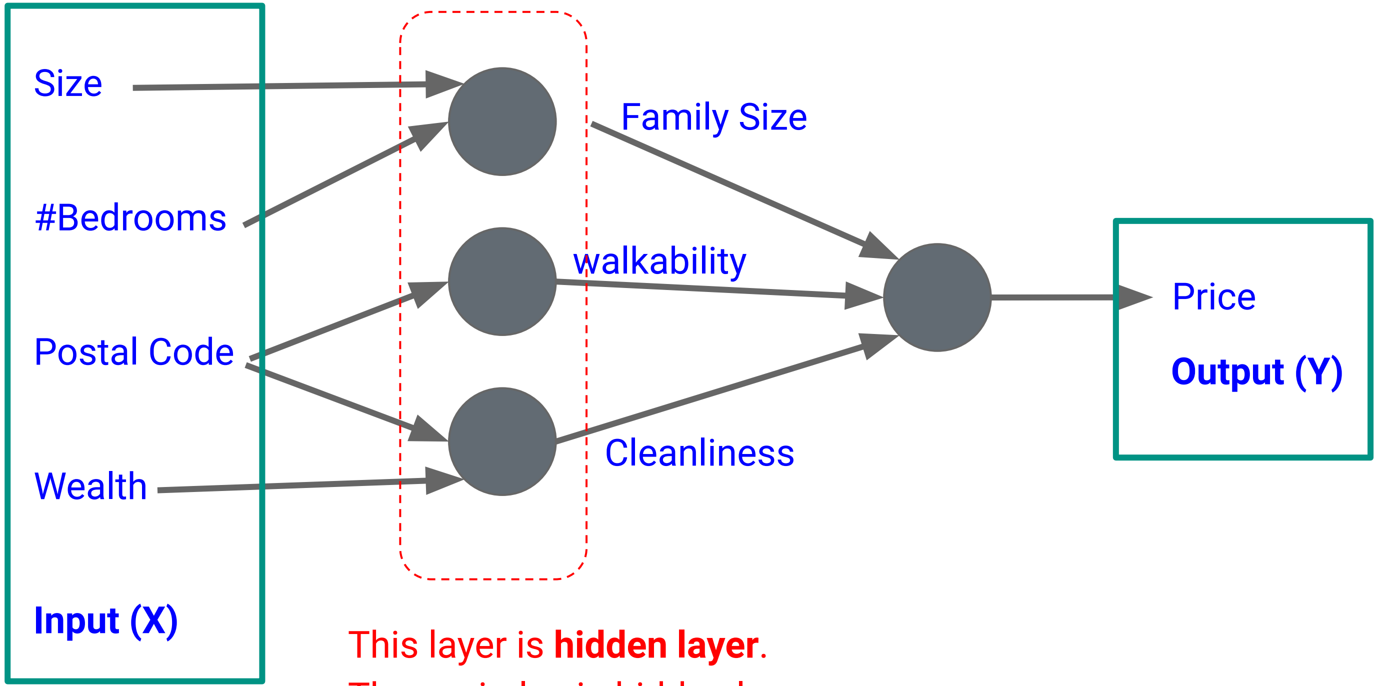


This function is very common in deep learning, and it is called ReLU.
ReLU = Rectified Linear Unit.
Rectified just means taking a max with 0.

How about a slightly bigger neural network?
Eg: Housing price prediction, based on >1 input variables.



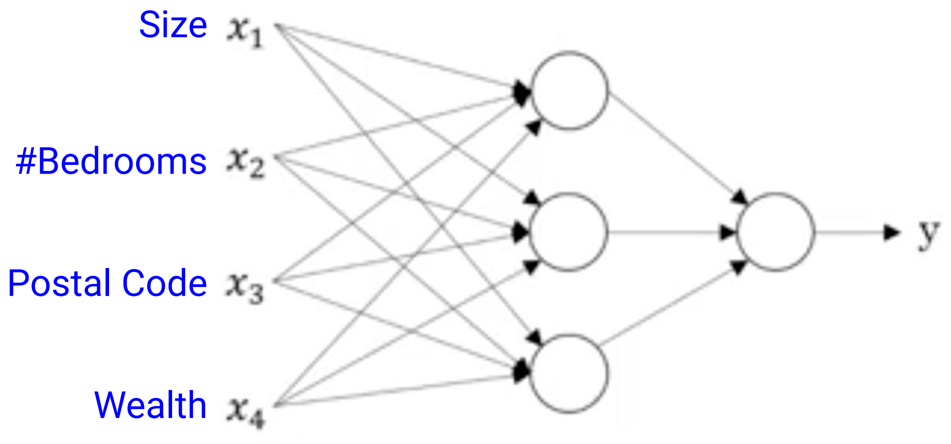
The magic of neural network is that you will provide X and Y for some examples in training set, and it will figure out all the other things by itself.



This layer is **hidden layer**.
These circles in hidden layer are called hidden units.

This is what we will actually implement

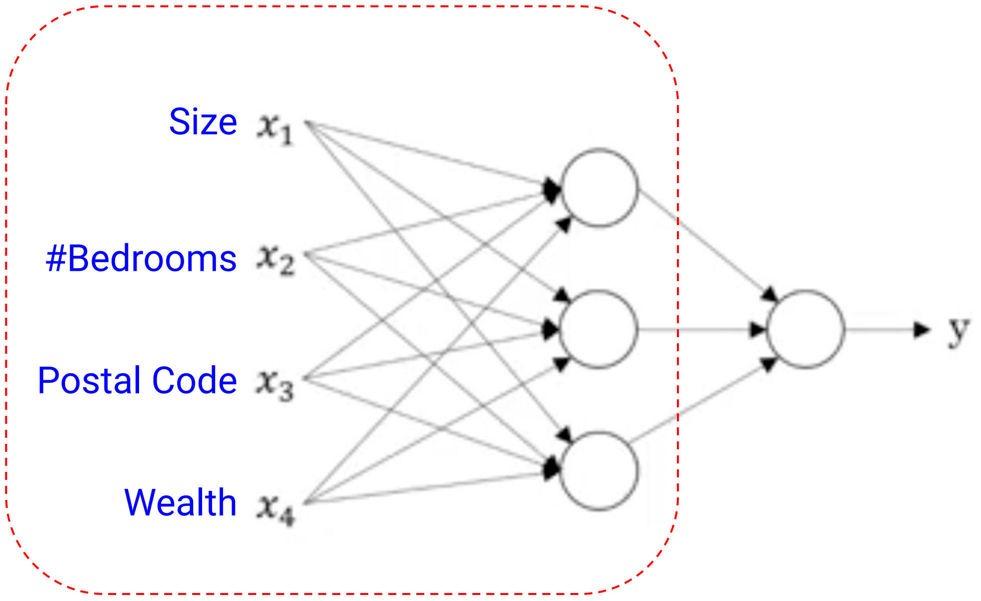
Each hidden unit takes
all four input features



Instead of we deciding that the last hidden unit is “cleanliness”, which depends only on postal code and wealth, we leave it on the neural network to decide what the last hidden unit would be. We just provide it with all input features.

This is what we will actually implement

Densely connected or Fully connected



This type of problems are called **supervised learning**. $\longrightarrow$ Labelled data available for training the model

Structured data

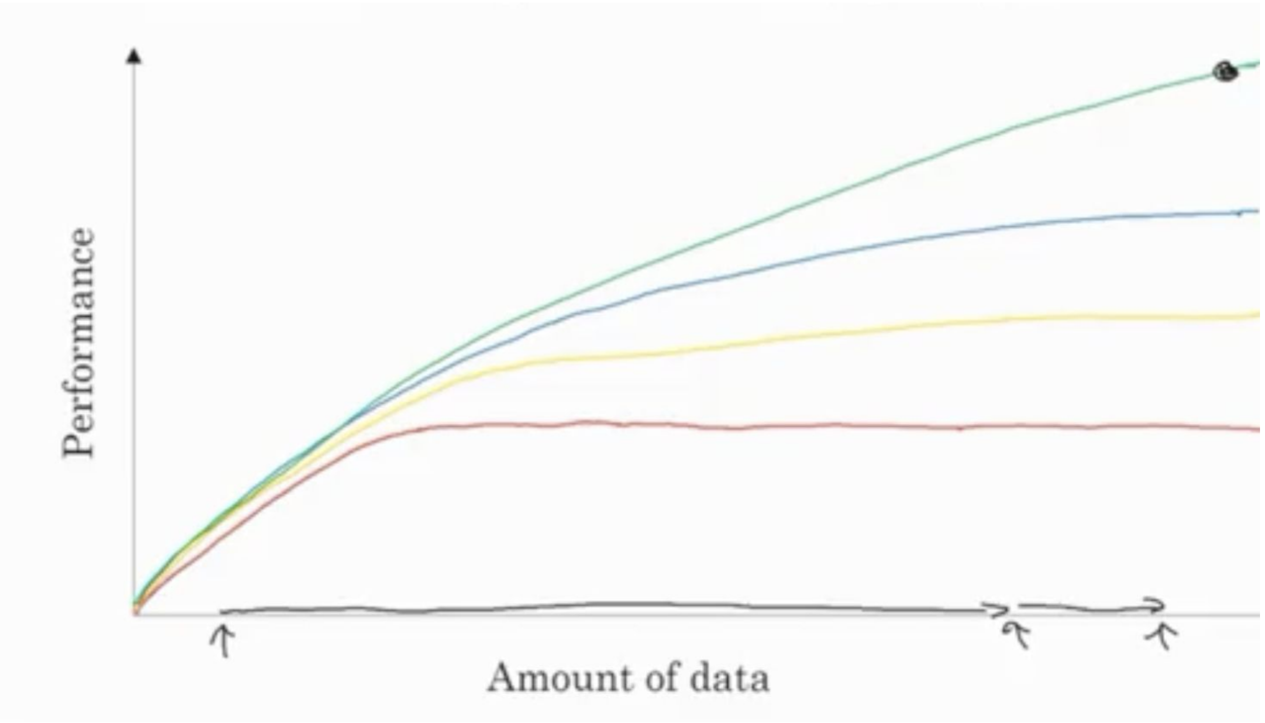
size	#room	pin	wealth	price
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-

Unstructured data

Image

Audio

Scale drives deep-learning progress



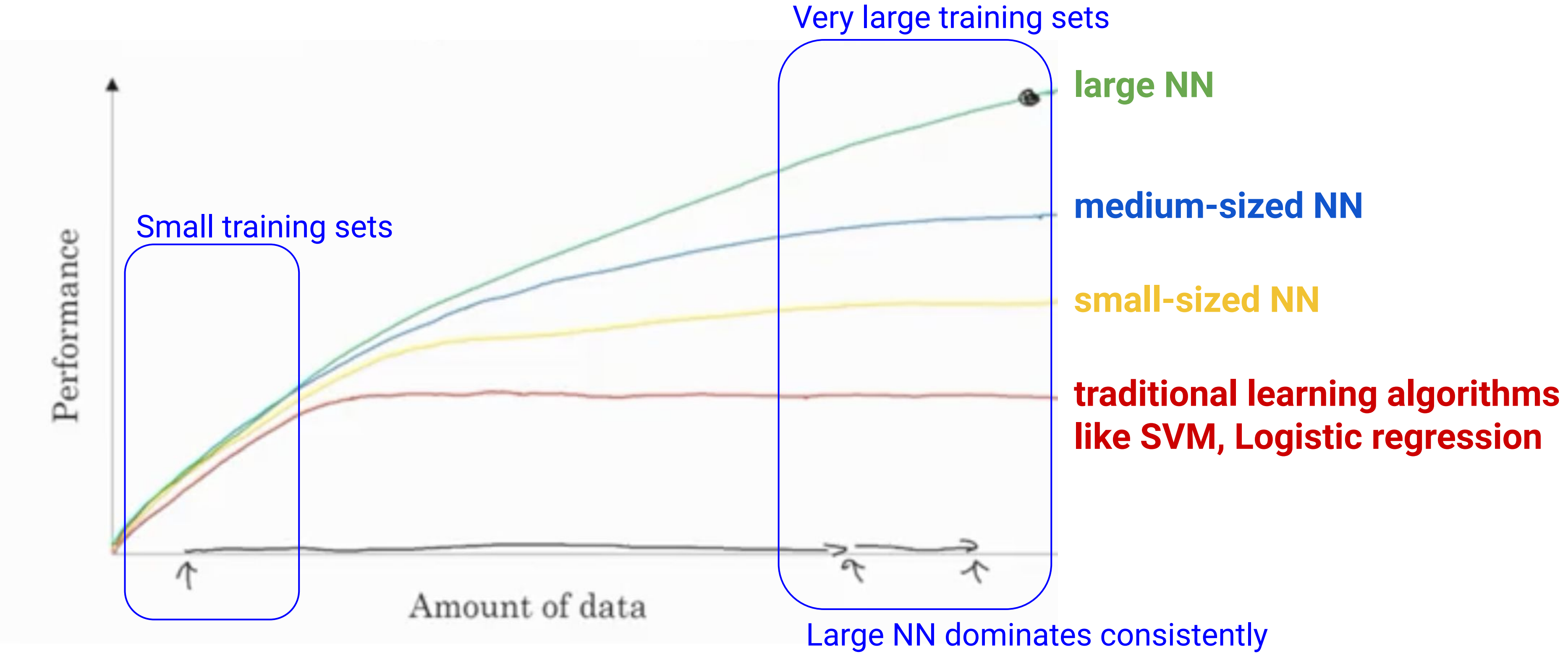
large NN

medium-sized NN

small-sized NN

traditional learning algorithms like SVM, Logistic regression

Scale drives deep-learning progress



Relative ordering of algorithms not well-defined.
Example: SVM could do slightly better than large NN.

More Data

More Computation power

Innovation of Algorithm

Process of training a NN is often **iterative** and **empirical** process.
Idea → Implement → Run and check performance → Repeat

Let’s start with Logistic regression which is the simplest possible NN.

Let’s start with Logistic Regression

Notations

For binary classification problem:

(x,y) is a single training example.

x is n-dimensional feature vector.

y is either 0 or 1.

Number of entries in training dataset = m.

We can write $x^{(1)}$ as a column vector with n entries.

Define X = $[x^{(1)}, x^{(2)}...., x^{(m)}]$

X is a (n x m) matrix,

n=number of rows, m=number of column.

In python, check X.shape. It should return (n, m)

Define Y = $[y^{(1)}, y^{(2)}...., y^{(m)}]$

Y is a (1 x m) dimensional matrix.

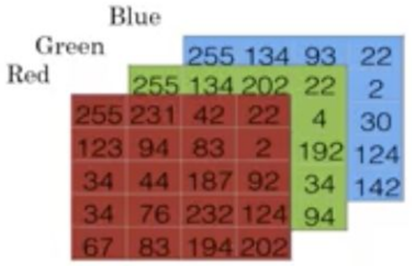
While implementing an algorithm, avoid using explicit for loop.

Implementing algorithms by hand is useful for your understanding, but in real research work it is not needed as we have libraries for almost everything.

→ Input image →

Binary classifier
1 (cat) vs 0 (non cat)

How does a computer "see" an image?
→ It sees 3 matrices. These are pixel-intensity values.



→ Each element is a feature. We need to unroll to a column vector.

If input image is 64x64 then number of features=64x64x3 = 12288.
So, n=12288.

Given x, want a prediction ŷ → estimate of y. We want predictions to be correct (i.e we want ŷ ≈ y).

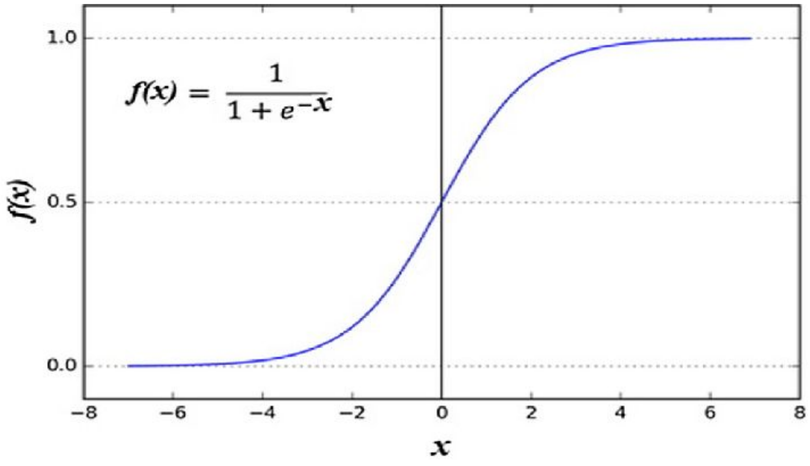
Formally, ŷ = P(y=1 | x)

Parameters: w and b; where w is n-dimensional vector and b is a real number.

In logistic regression, **$\hat{y} = \sigma(w^T x+b)$**

$\downarrow$

Sigmoid



We need a Cost Function so that the model parameters (w, b) can be learned.

Loss function

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

We want this loss to be as small as possible so that ŷ is close to y

Cost function

$J(w,b) = (1/m)\sum L(\hat{y}^{(i)},y^{(i)})$

We want to find parameters w and b that minimize this Cost function

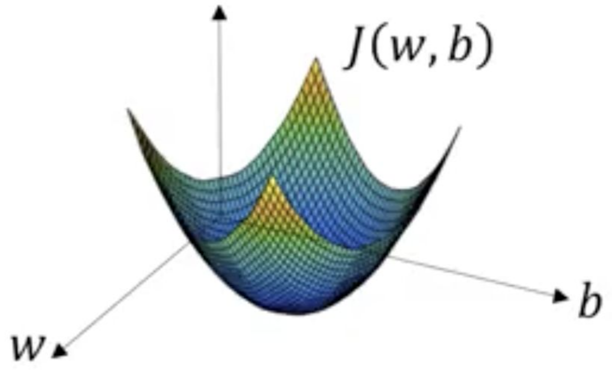
Gradient Descent

Epoch: One pass through entire training set

$$\hat{y} = \sigma(w^T x + b), \ \sigma(z) = \frac{1}{1+e^{-z}}$$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})$$

We want to find w and b that minimize J(w,b)



First, initialize w, b, then

$$\begin{aligned}
&\text{Repeat } \{ \\
&w := w - \alpha \frac{\partial J(w,b)}{\partial w} \\
&b := b - \alpha \frac{\partial J(w,b)}{\partial b} \\
&\}
\end{aligned}$$

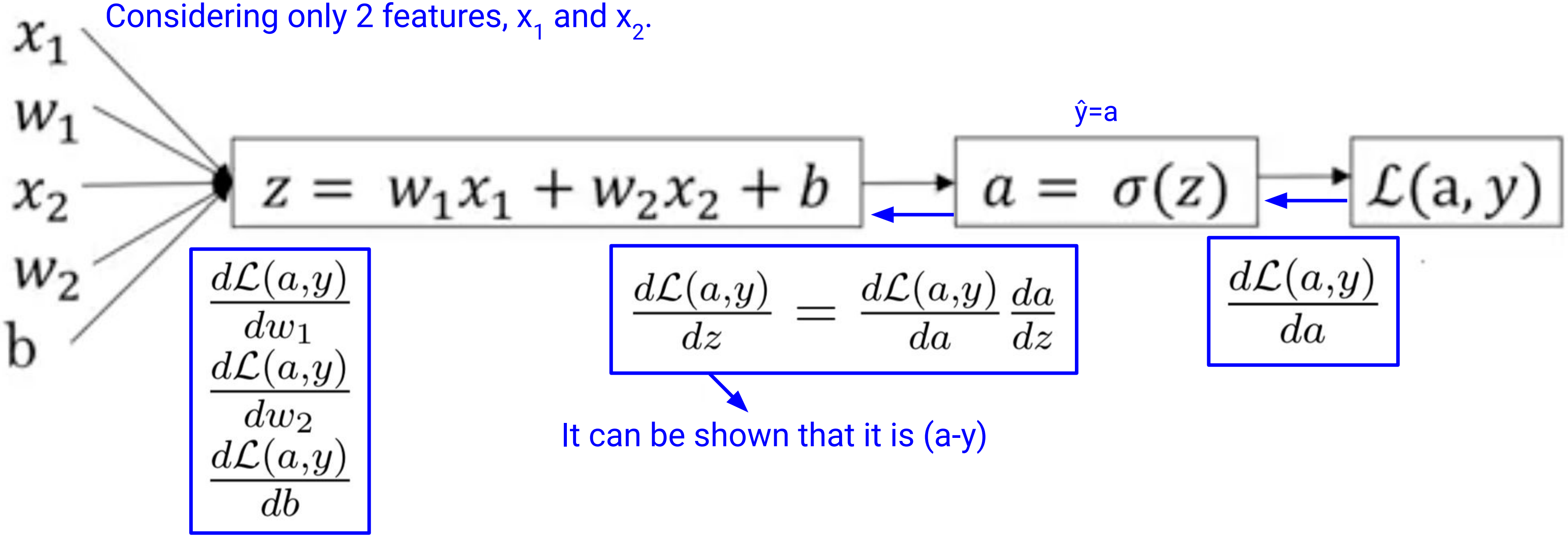
Forward propagation: compute the loss

Backward propagation: compute derivatives

Computation graph can be used for this

For one training example

Considering only 2 features, x_1 and x_2.



x_1
w_1
x_2
w_2
b
$\hat{y}=a$
$z = w_1x_1 + w_2x_2 + b$
$a = \sigma(z)$
$\mathcal{L}(a, y)$
$\frac{d\mathcal{L}(a,y)}{dw_1}$ $\frac{d\mathcal{L}(a,y)}{dw_2}$ $\frac{d\mathcal{L}(a,y)}{db}$
$\frac{d\mathcal{L}(a,y)}{dz} = \frac{d\mathcal{L}(a,y)}{da}\frac{da}{dz}$
$\frac{d\mathcal{L}(a,y)}{da}$

It can be shown that it is (a-y)

We have to do it for m training examples, and then sum up and divide by m

How can you write a code to compute J and its derivatives w.r.t w and b?

```
J = 0, dw_1 = 0, dw_2 = 0, db = 0     Initialize variables
for i = 1 to m:     for loop over training examples (m)
    $z^{(i)} = w^T x^{(i)} + b$
    $a^{(i)} = \sigma(z^{(i)})$
    $J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$
    $dz^{(i)} = a^{(i)} - y^{(i)}$
    $dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$     Possible for loop over number of features (n), when n is large
    $dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$
    $db \mathrel{+}= dz^{(i)}$
J = J/m, dw_1 = dw_1/m, dw_2 = dw_2/m
db = db/m
```

Drawback of the above approach: Explicit for loops.
In deep-learning era, m and n will be very large. So for loop is inefficient and **very slow**.
Use vectorization technique to get rid of the for loops. It will speed up the code.

Vectorization: one example

How to vectorize computation of z for one training example?
Where **$z = w^Tx+b$,**
w and x are column matrices with n entries where n is the number of features..

Instead of a for loop over n, we can vectorize in the following way using numpy
z = np.dot(w.T,x) + b

Note that it might not be possible to get rid of all for loops in all situations.
For example, a for loop for **number of iterations** will still be needed.

Vectorizing forward propagation of logistic regression

$z^{(1)} = w^T x^{(1)} + b$

$a^{(1)} = \sigma(z^{(1)})$

1st training example

$z^{(2)} = w^T x^{(2)} + b$

$a^{(2)} = \sigma(z^{(2)})$

2nd training example

$z^{(3)} = w^T x^{(3)} + b$

$a^{(3)} = \sigma(z^{(3)})$

3rd training example

…. m times

Recall we defined $X = [x^{(1)}, x^{(2)}...., x^{(m)}]$, which is (n x m) matrix.

$Z = [z^{(1)}, z^{(2)}...., z^{(m)}] = w^T X + [b, b, b ....] = [w^T x^{(1)}, w^T x^{(2)}, w^T x^{(3)}....] + [b, b, b...]$

In numpy

Z = np.dot(w.T, X) + b

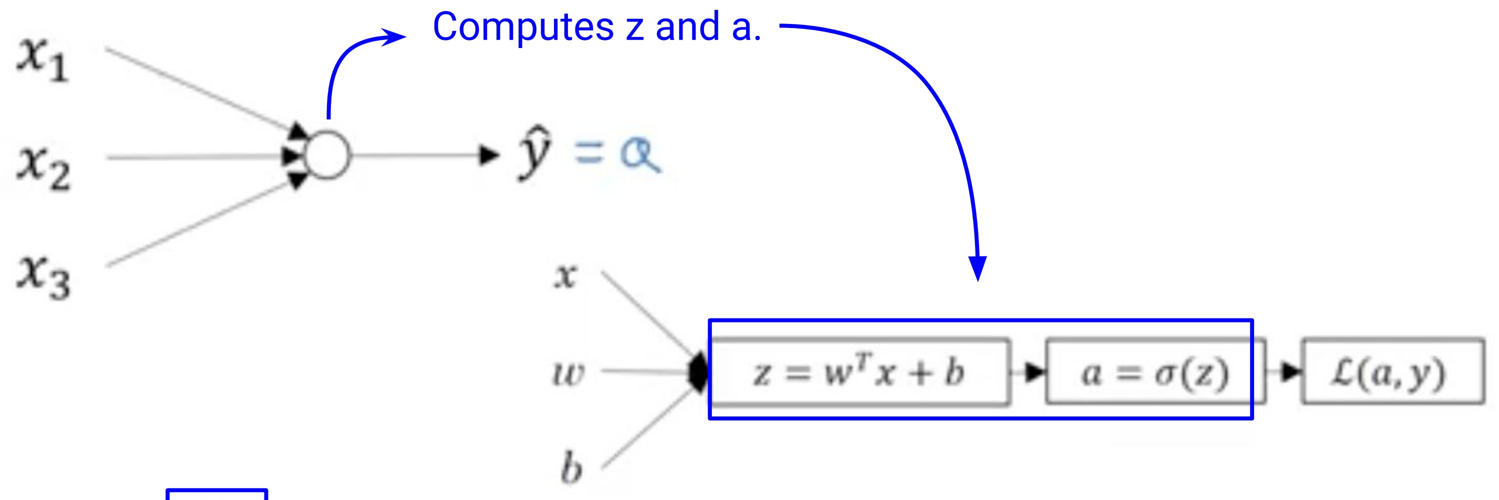
Broadcasting

$A = [a^{(1)}, a^{(2)}...., a^{(m)}] = \sigma(Z)$

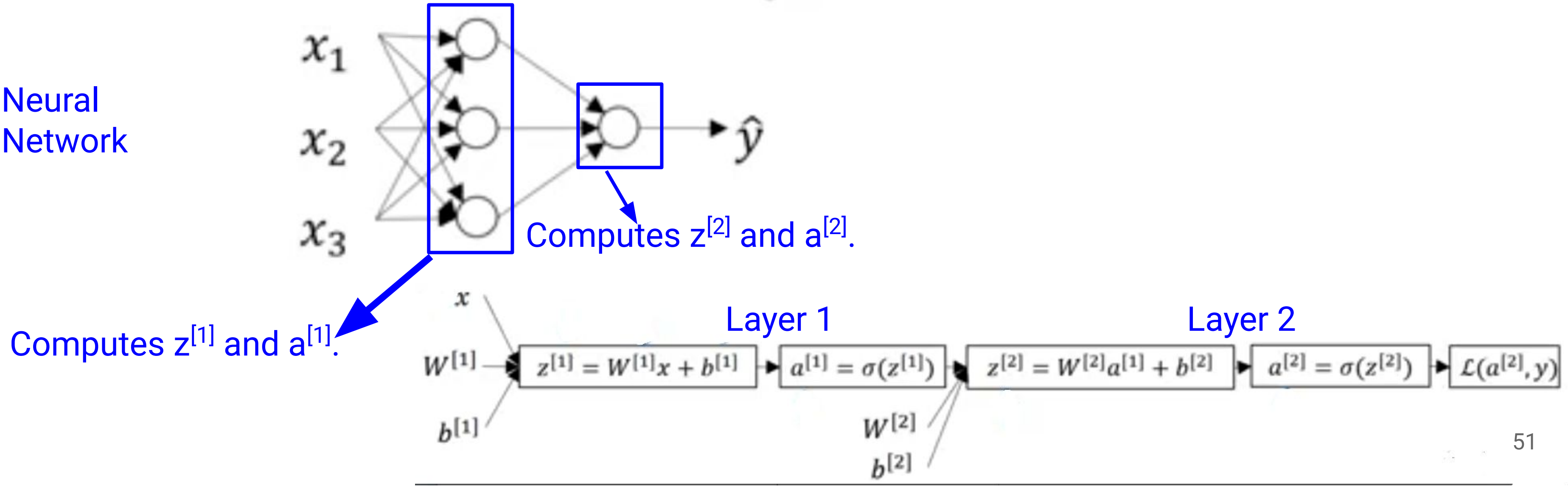
Vector valued sigmoid function

Back propagation can also be vectorised in a similar way. Make use of in-built functions like np.sum(..)

Logistic Regression



Neural Network



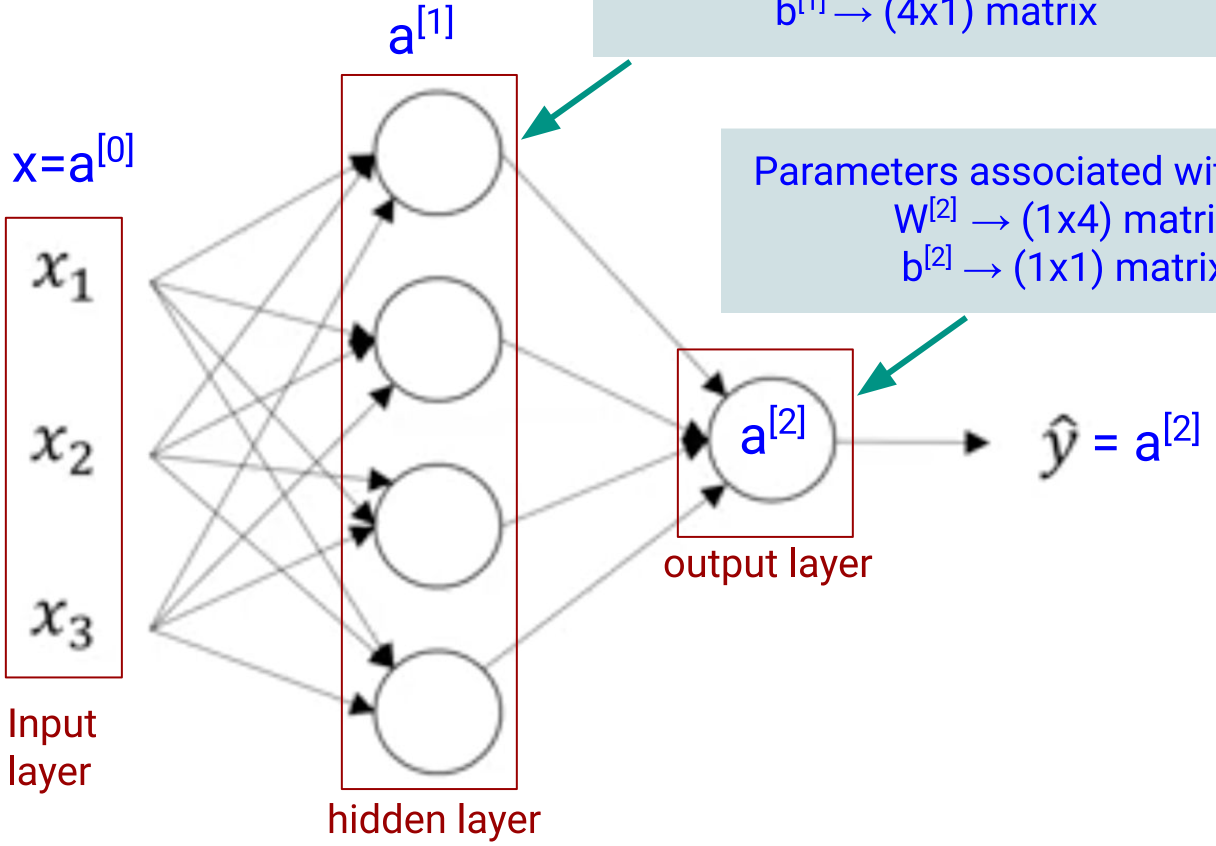
x_1
x_2
x_3
$\hat{y}$
Computes $z^{[2]}$ and $a^{[2]}$.
Computes $z^{[1]}$ and $a^{[1]}$.
Layer 1
Layer 2
x
$W^{[1]}$
$b^{[1]}$
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$W^{[2]}$
$b^{[2]}$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$\mathcal{L}(a^{[2]}, y)$

2-layer NN

Input layer not counted

Parameters associated with layer 1
$W^{[1]} \rightarrow$ (4x3) matrix
$b^{[1]} \rightarrow$ (4x1) matrix

Parameters associated with layer 2
$W^{[2]} \rightarrow$ (1x4) matrix
$b^{[2]} \rightarrow$ (1x1) matrix



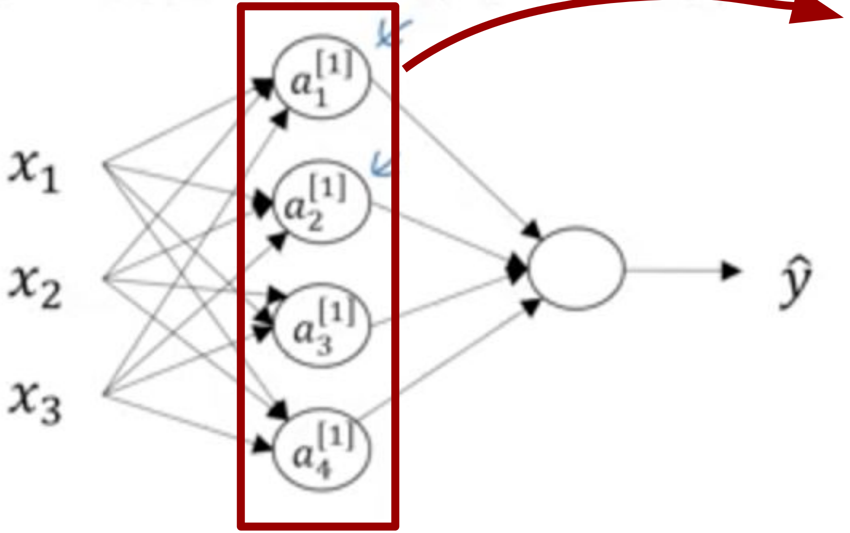
What NN does is similar to logistic regression, but repeated **multiple times**

Notation

$$z_i^{[l]}$$

$\rightarrow$ layer number

$\rightarrow$ node number



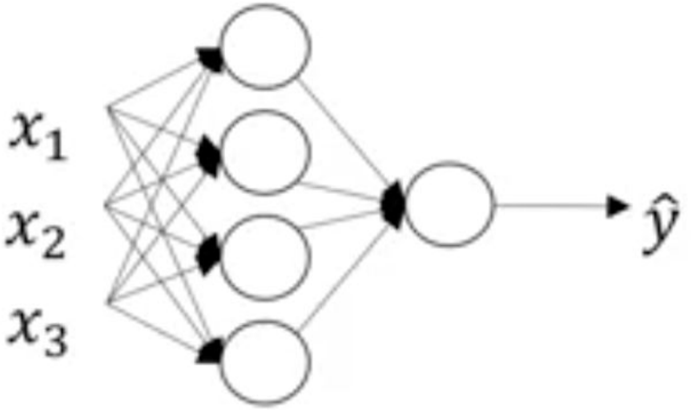
Vectorised implementation of this is possible

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \ a_4^{[1]} = \sigma(z_4^{[1]})$$



Forward propagation for 1 training example

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

For m training examples

```
for i = 1 to m:
    $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$
    $a^{[1](i)} = \sigma(z^{[1](i)})$
    $z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$
    $a^{[2](i)} = \sigma(z^{[2](i)})$
```

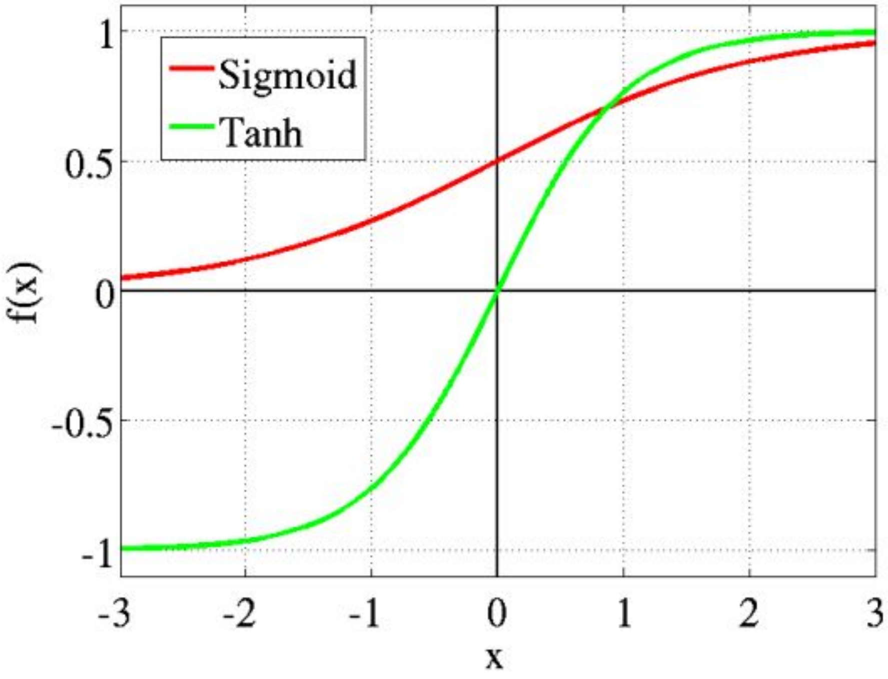
We can also vectorize across m training examples, avoiding for loop, like we did for logistic regression

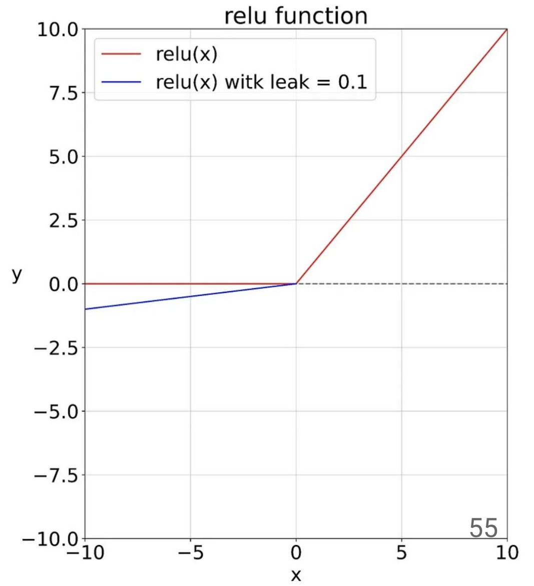
Activation function

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$\boxed{a^{[1]} = \sigma(z^{[1]})}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$\boxed{a^{[2]} = \sigma(z^{[2]})}$$

We have been using **sigmoid** so far. But there are other better options too. Example **tanh** works better than sigmoid in hidden layer.

Sigmoid and tanh have issues when z is large or small, because slope close to zero and gradient descent is very slow. **ReLU** or **Leaky ReLU** can be used.



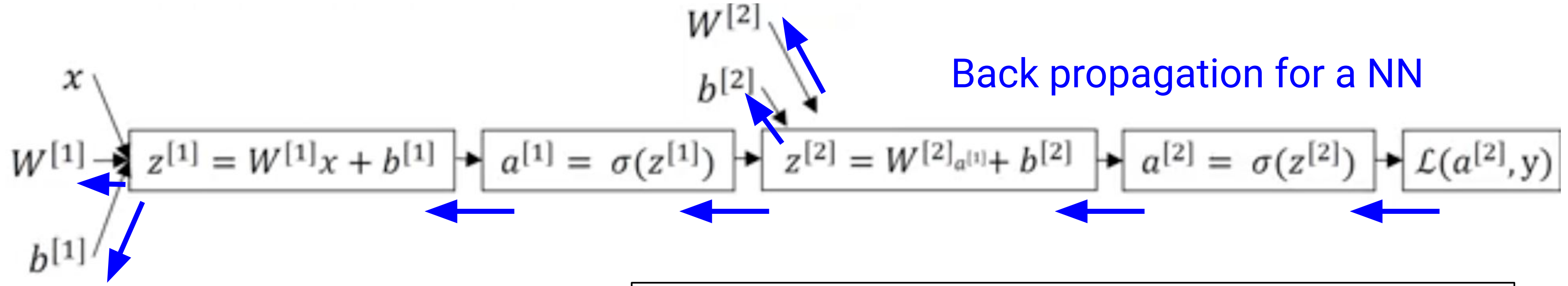


Using non-linear activation function is an essential part for NN.

NN learns interesting features from the given features by using the non-linearity of the activation.

In the absence of non-linear activation function, the power of NN is lost. This can be checked by using identity activation $g(z)=z$.

Back propagation for a NN


x
$W^{[1]}$
$b^{[1]}$
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$W^{[2]}$
$b^{[2]}$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$\mathcal{L}(a^{[2]}, y)$

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]}a^{[1]^T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = dz^{[1]}x^T$$

$$db^{[1]} = dz^{[1]}$$

Random initialisation of weights (w) is necessary for NN.

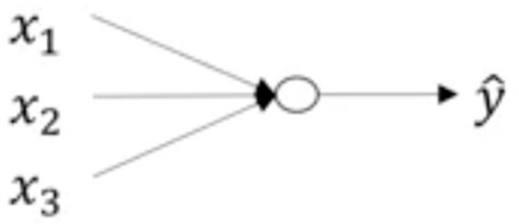
Initializing w parameters to 0 does not work for NN. (all hidden units become same)

Use **small** random values. Large values may lead to vanishing gradient issue.

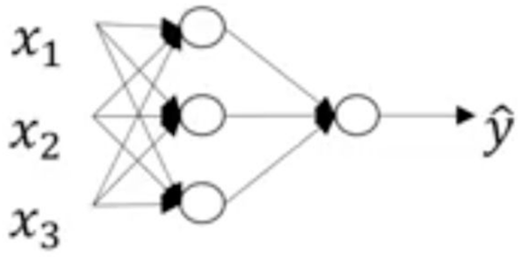
Use `np.random.randn(i,j) * 0.01`

However, b can be initialized to 0.

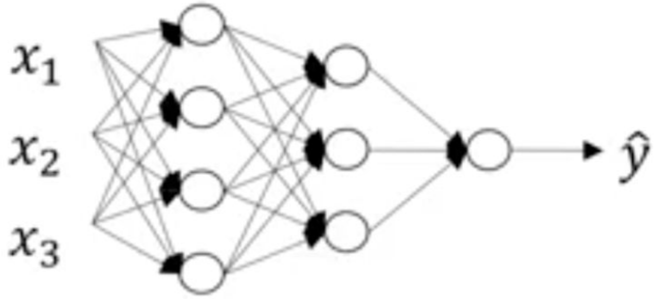
Now we are ready to look into Deep NN



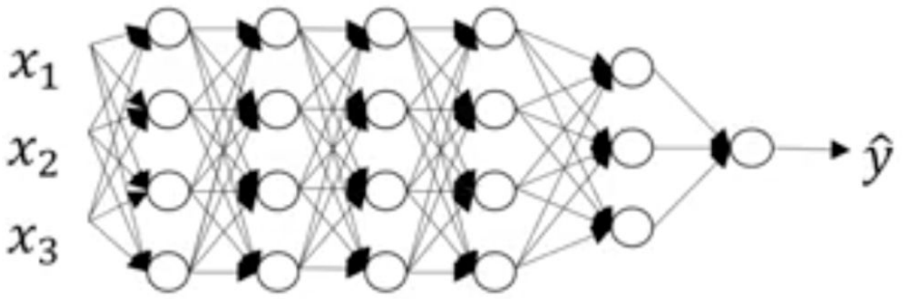
logistic regression



1 hidden layer

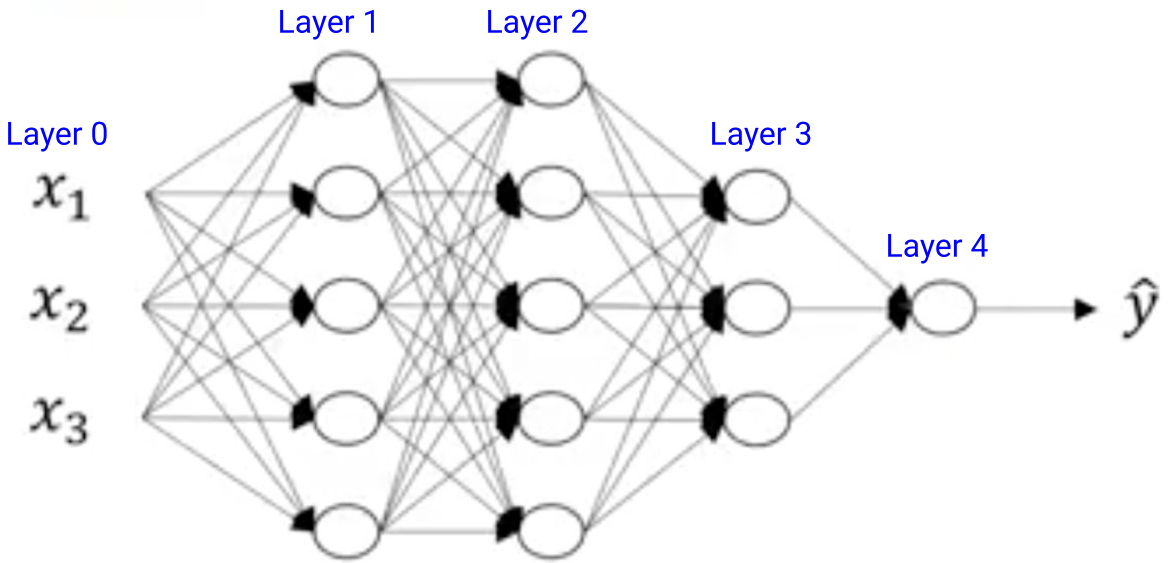


2 hidden layers



5 hidden layers

Notations



Number of layers = l = 4

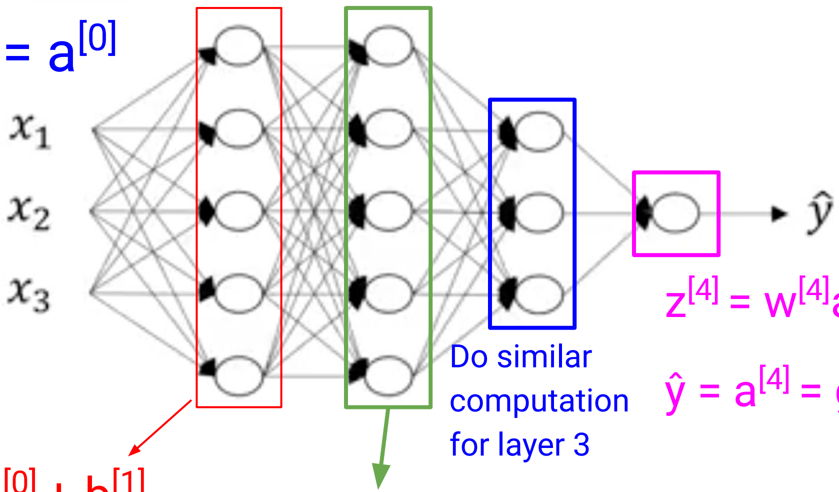
Number of units in a layer = $n^{[l]}$

Activation in layer l = $a^{[l]}$

$n^{[1]} = 5$, $n^{[2]} = 5$, $n^{[3]} = 3$, $n^{[4]} = 1$, also $n^{[0]} = 3$

Forward propagation in deep NN for one training example

$x = a^{[0]}$



Do similar computation for layer 3

$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$

$\hat{y} = a^{[4]} = g^{[4]}(z^{[4]})$

$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$

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

So, the general rule is

$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$

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

Forward propagation in deep NN for multiple training example

For one training example

$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$

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

For multiple training example

$$\boxed{\begin{aligned} Z^{[l]} &= w^{[l]}A^{[l-1]} + b^{[l]} \\ A^{[l]} &= g^{[l]}(Z^{[l]}) \end{aligned}}$$

Recall we defined

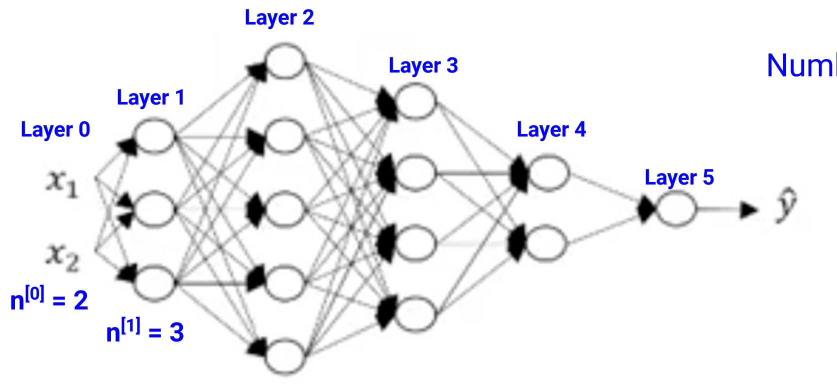
$A^{[0]} = X = [x^{(1)}, x^{(2)}...., x^{(m)}]$, which is (n x m) matrix.

$Z = [z^{(1)}, z^{(2)}...., z^{(m)}]$

$A = [a^{(1)}, a^{(2)}...., a^{(m)}]$

An explicit for loop for number of layers (l) is unavoidable. Otherwise, this is a vectorised implementation.

Keep an eye on matrix dimensions while implementing NN



Number of layers = l = 5

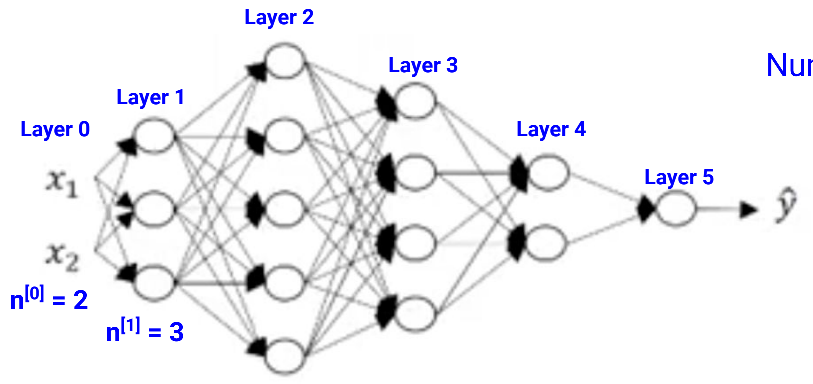
For layer 1
$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$

$z^{[1]}$ is (3, 1) matrix, ie ($n^{[1]}$, 1) matrix.
$a^{[0]}$ is x, it is (2, 1) matrix, ie ($n^{[0]}$, 1) matrix.
So $w^{[1]}$ has to be (3, 2) matrix, ie ($n^{[1]}$, $n^{[0]}$) matrix
More generally $w^{[l]}$ has to be ($n^{[l]}$, $n^{[l-1]}$) matrix
So, in this case, $w^{[2]}$ has to be (5, 3) matrix

And $b^{[1]}$ is ($n^{[l]}$, 1) matrix

For 1 training example

Keep an eye on matrix dimensions while implementing NN



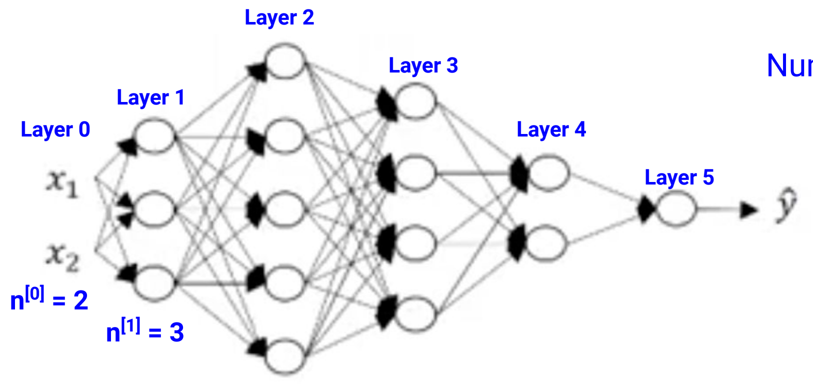
Number of layers = l = 5

For layer 1

$$\underset{(n^{[1]},\,1)}{z^{[1]}} = \underset{(n^{[1]},\,n^{[0]})}{w^{[1]}}\ \underset{(n^{[0]},\,1)}{a^{[0]}} + \underset{(n^{[1]},\,1)}{b^{[1]}}$$

For 1 training example

Keep an eye on matrix dimensions while implementing NN



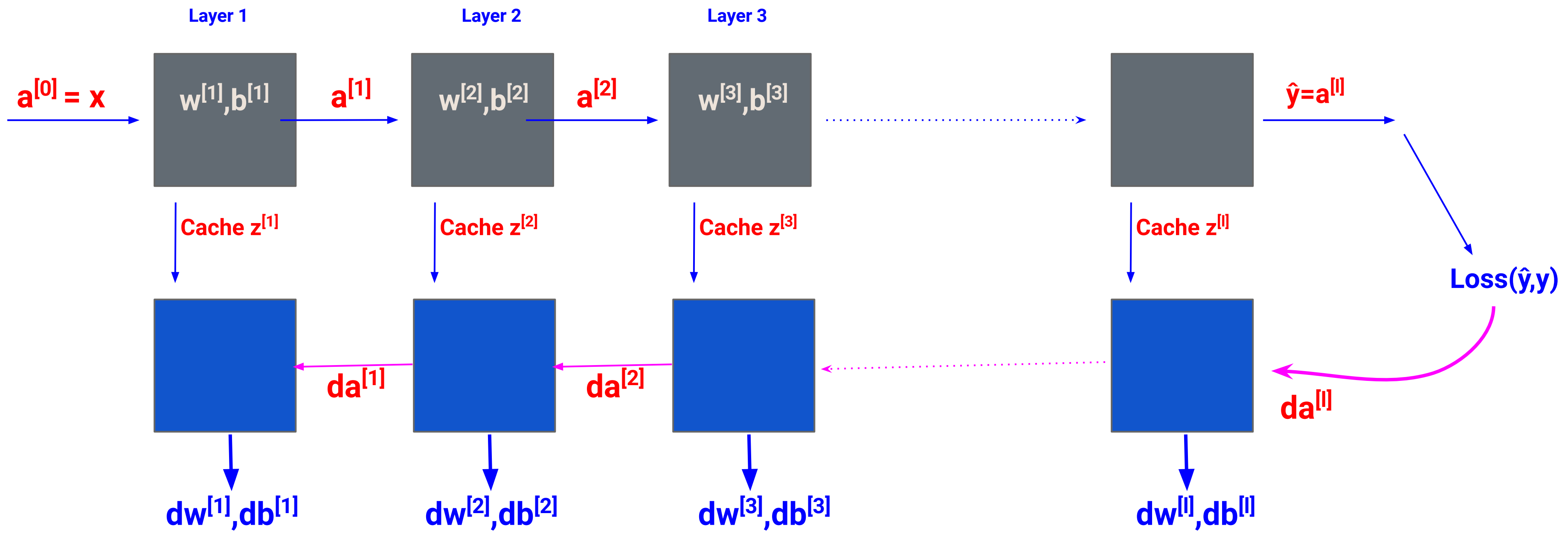
Number of layers = l = 5

For layer 1

$$\underset{(n^{[1]},\, m)}{Z^{[1]}} = \underset{(n^{[1]},\, n^{[0]})}{w^{[1]}}\ \underset{(n^{[0]},\, m)}{A^{[0]}} + \underset{(n^{[1]},\, m)}{b^{[1]}}$$

For m training examples

Summary of forward and backward propagation (one iteration)

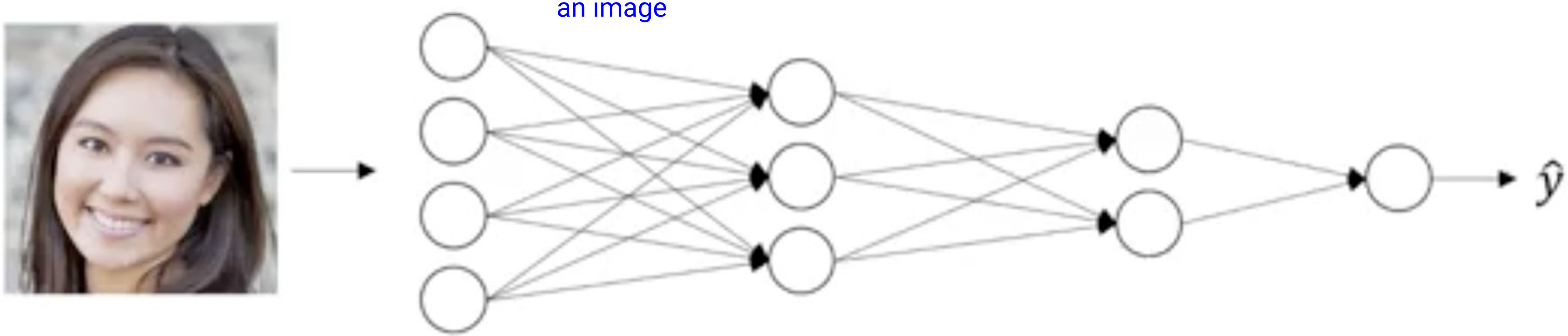


$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

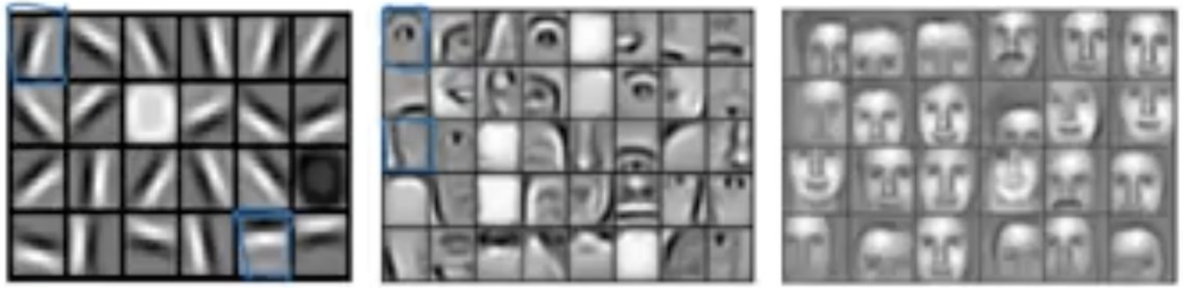
Why deep NN work so well?

Face recognition

Earlier layers learning simple features, later layers learn more complex features. Earlier layers focus on small areas of an image. Later layers focus on larger areas of an image



We will come back to this while discussing CNN



1st layer may be detecting edges

2nd layer may be detecting parts of faces (eye, nose etc)

3rd layer may be detecting different types of faces

Parameters vs Hyperparameters

Parameters: $w^{[l]}$, $b^{[l]}$

Hyperparameters: learning rate, number of iterations, number of hidden layers, number of hidden units in each layer, choice of activation function.

Hyperparameters control / influence parameters.

Best value of hyperparameter not known in advance. You might have to figure it out with some trials.
We can provide some guidance.

Train/Dev/Test sets

Recall that applied ML is an iterative process.
Idea → Code → Check performance

training
dev
test

Use training set to keep on training your NN
Use dev set to check which network is performing best.
Once you are happy with a NN, check performance in test set.

In modern era of big data, where we have **millions** of data,
Train: 98%
Dev: 1%
Test: 1%

If you have small amount of data (few **thousand**), then 60%, 20%, 20% will be more appropriate.
In some cases, people do not have a test set, ie, you can have only train and dev.

back up slides

https://www.oeaw.ac.at/en/hephy/research/machine-learning