Machine Learning for Astrophysics

Deep Learning

Course overview

You are here

Outline

Introduction to ML & Supervised ML

- Introduction
- Regression, Regularization
- Classification, Logistic Regression
- Bias/Variance trade-off

Supervised ML strikes back

- Support Vector Machines
- Gaussian Processes
- Nearest Neighbors
- Ensemble Methods: random forests
- Gradient Boosting

Unsupervised ML

- Clustering: KMeans, DBScan, GMM, Agglomerative Clustering
- Anomaly Detection
- Dimensionality Reduction:
  - linear: PCA, NMF, ICA
  - manifold learning: LLE, IsoMap, t-SNE
- Self-Organizing Maps

Deep Learning

- Basics of NN: computation graphs
- Training a NN: *forth*- and *back*-propagation
- Optimization Algorithms
- Transfer Learning
- ResNets
- Bayesian NN, Probabilistic BNN
- Autoencoders and VAE

Deep Learning, The Revenge

- Reinforcement Learning
- Convolutional Neural Networks
- Inception Module and MobileNet
- Generative Adversarial Networks
- (*hints on*) Recurrent Neural Networks
- Transformers

Supervised Learning

Data: (x, y)

x is the data, with associated labels y

Goal:

learn a function that maps

$x \longrightarrow y$

"This thing is a dog"

Unsupervised Learning

Data: x

there are no labels, only data x

Goal:

learn underlying structure of x

"These two things look alike"

Reinforcement Learning

Data: *state-action* pairs

Goal:

learn a policy π

maximizing future rewards

"Cuddling this thing will make you happy"

Gradient descent is the process of moving towards the minimum of a cost function *J*, as fast as possible, in order to find the best-fit parameters *w* and *b* of your model.

The outline is quite simple:

1. start with some (random or not) initial values for *w* and *b*
2. keep changing *w* and *b* in order to reduce *J(w,b)*
3. until you settle at or near the global minimum (pay attention to local minima)

The weights are updated through an iterative process in this way:

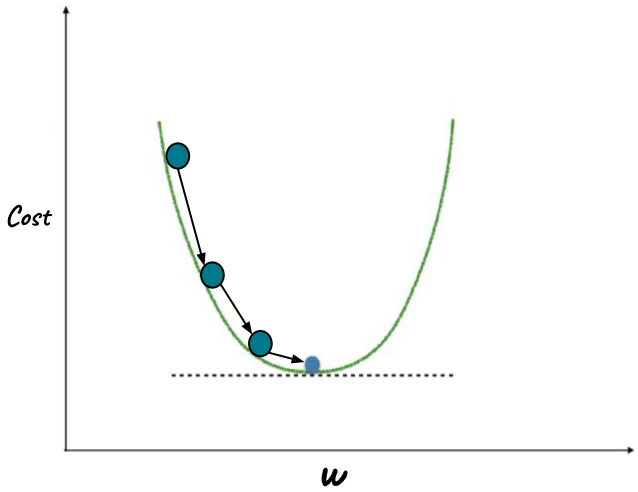
$$w := w - \alpha \, \mathrm{d}J(w)/\mathrm{d}w$$

(the same for b)

with α being the **learning rate**, a fundamental hyperparameter especially in Deep Learning.
Choosing the best learning rate is fundamental for efficiency and reaching the minimum

if α is too small → too slow, lots of steps to arrive to minimum
if α is too large → risk of missing the minimum and jumping over (overshoot)

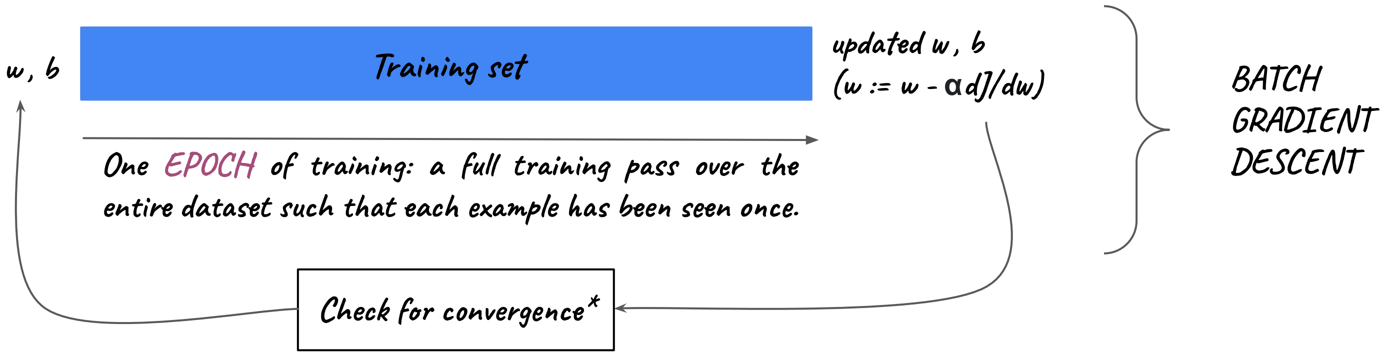
Typical values for α range between 0.01 and 1.

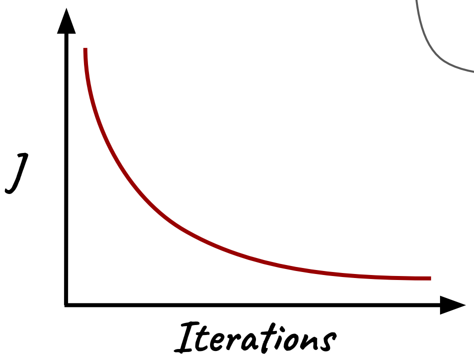


Gradient descent is the process of moving towards the minimum of a cost function *J*, as fast as possible, in order to find the best-fit parameters *w* and *b* of your model.

The outline is quite simple:

1. start with some (random or not) initial values for *w* and *b*
2. keep changing *w* and *b* in order to reduce *J(w,b)*
3. until you settle at or near the global minimum (pay attention to local minima)





Pro: accurate measure of cost J

Cons: extremely slow for large datasets

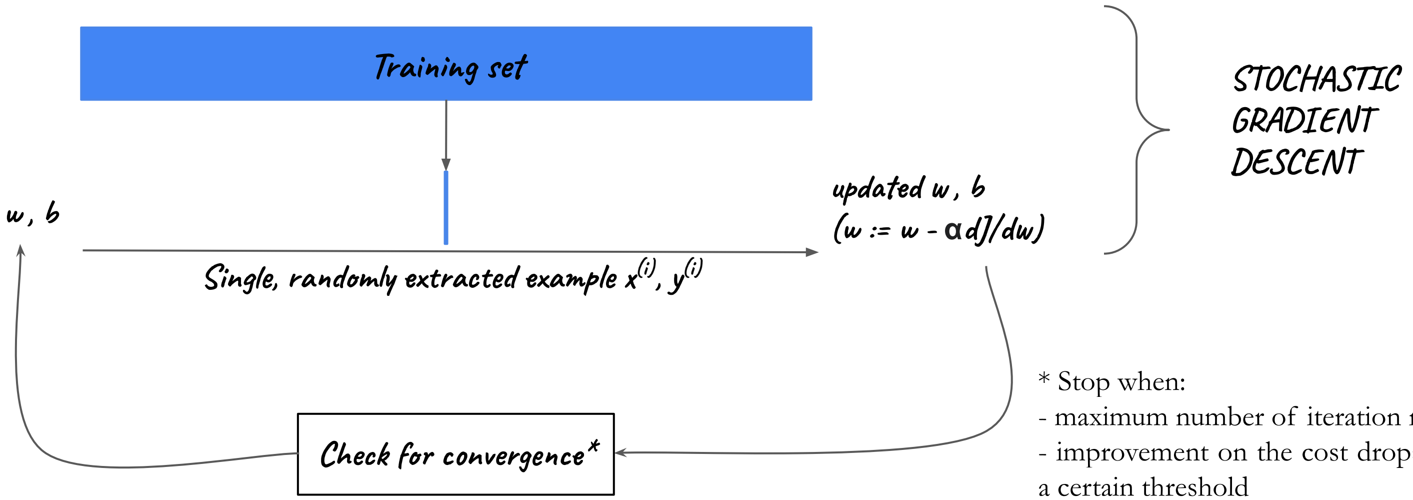
* Stop when:
- maximum number of iteration reached
- improvement on the cost drops below a certain threshold
- cost gets close enough to a certain value where you expect the cost minimum (rare)

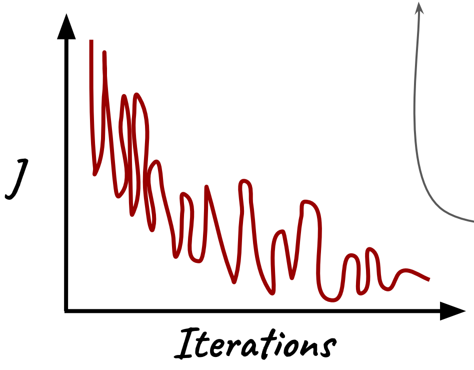
Recap: stochastic gradient descent

Gradient descent is the process of moving towards the minimum of a cost function J, as fast as possible, in order to find the best-fit parameters w and b of your model.

The outline is quite simple:

1. start with some (random or not) initial values for w and b
2. keep changing w and b in order to reduce $J(w,b)$
3. until you settle at or near the global minimum (pay attention to local minima)





Pro: extremely fast
Cons: extremely noisy

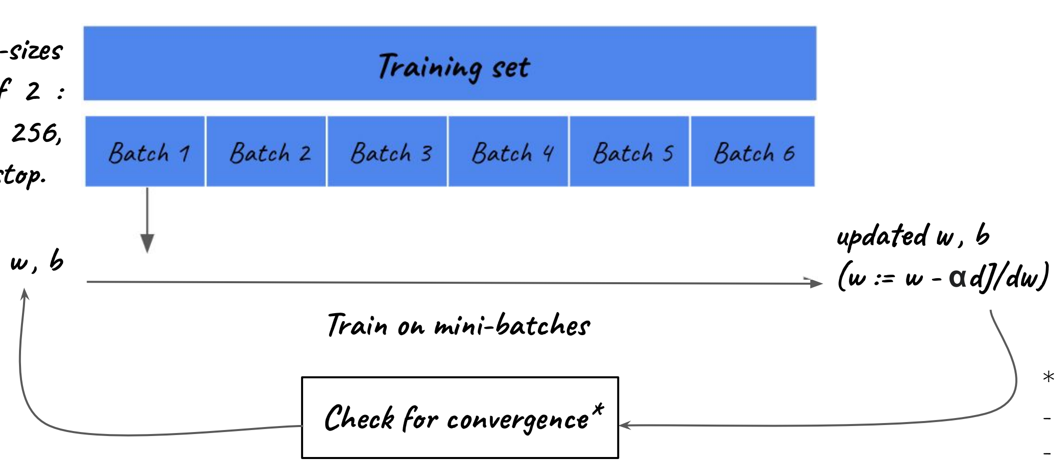
* Stop when:
- maximum number of iteration reached
- improvement on the cost drops below a certain threshold
- cost gets close enough to a certain value where you expect the cost minimum (rare)

Gradient descent is the process of moving towards the minimum of a cost function *J*, as fast as possible, in order to find the best-fit parameters *w* and *b* of your model.

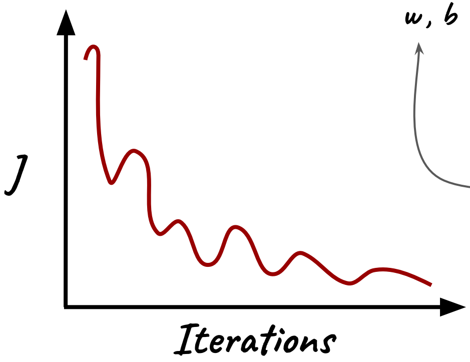
The outline is quite simple:

1. start with some (random or not) initial values for *w* and *b*
2. keep changing *w* and *b* in order to reduce *J(w,b)*
3. until you settle at or near the global minimum (pay attention to local minima)

Popular batch-sizes are all power of 2 : 32, 64, 128, 256, 512, 1024, and stop.



MINI-BATCH STOCHASTIC GRADIENT DESCENT

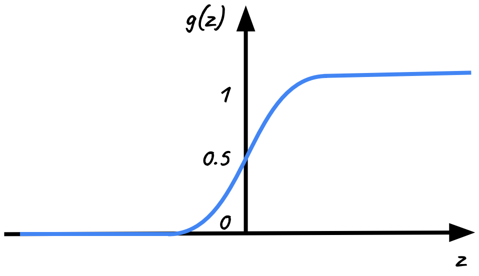


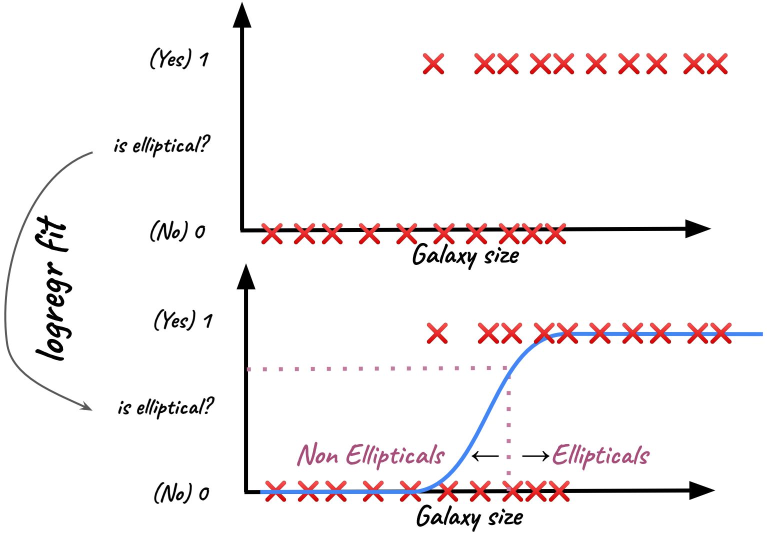
Pro: generally fast
Cons: a bit noisy, but you can live with it

* Stop when:
- maximum number of iteration reached
- improvement on the cost drops below a certain threshold
- cost gets close enough to a certain value where you expect the cost minimum (rare)

Logistic regression is the centerpiece of classification problems. It originates from the sigmoid (or logistic) function:

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





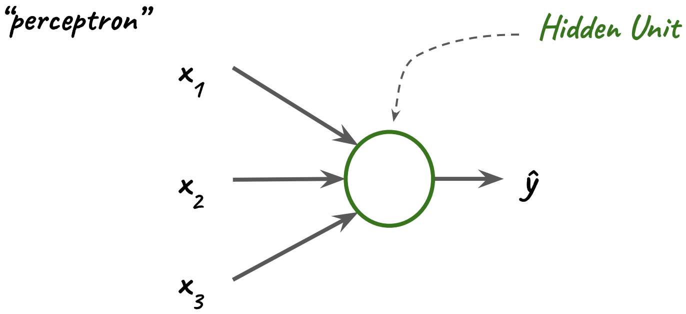
And with this function, define the logistic regression model:

$$f_{\vec{w},b} = g(\vec{w} \cdot \vec{x} + b) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

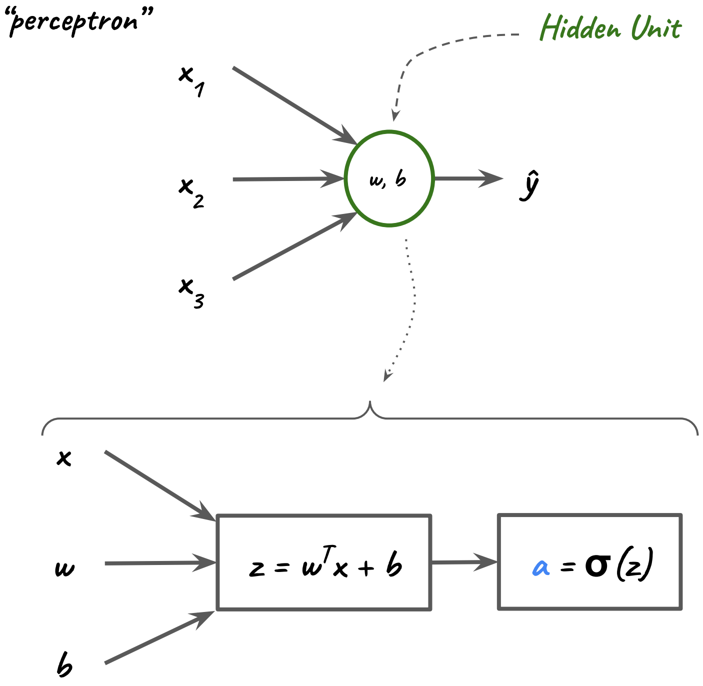
where $f_{w,b}$ gives the probability that x belongs to a certain class (e.g. if the galaxy is elliptical once given a galaxy size in kpc).
With logistic regression you train on the training set, finding the optimal values for the weights w, b.
The logistic loss function (*logloss*) evaluated on a single training example is:

$$\mathcal{L}(f_{\vec{w},b}(x^{(i)}, y^{(i)})) = -y^{(i)} \log(f_{\vec{w},b}(x^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\vec{w},b}(x^{(i)}))$$

The core element of a neural network is the **hidden unit** (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.



The core element of a neural network is the *hidden unit* (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.



The meaning of that circle symbol is:

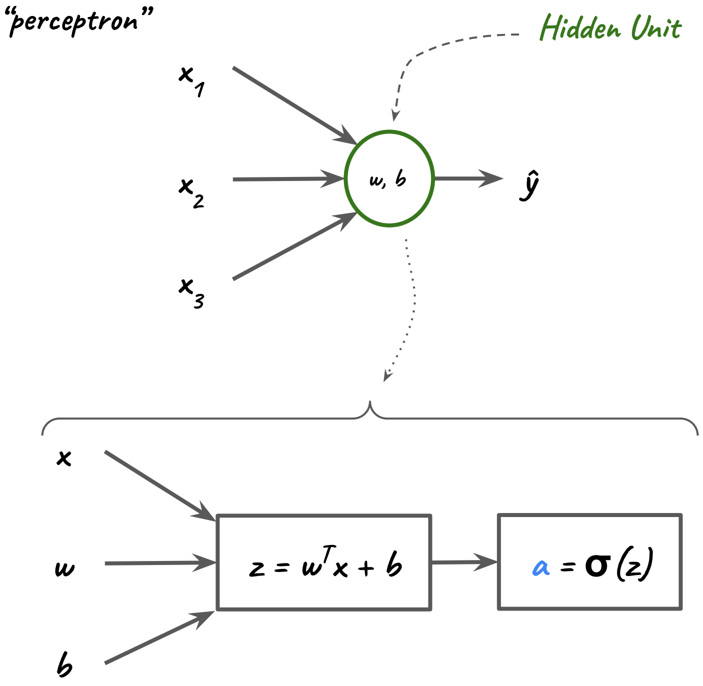
1) take in input the features x
2) combine the features with the hidden unit weights and bias

$$z = w^T x + b$$

3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone *hidden unit*, in this case a is equal to the output value $\hat{y}$.

The core element of a neural network is the *hidden unit* (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.



The meaning of that circle symbol is:

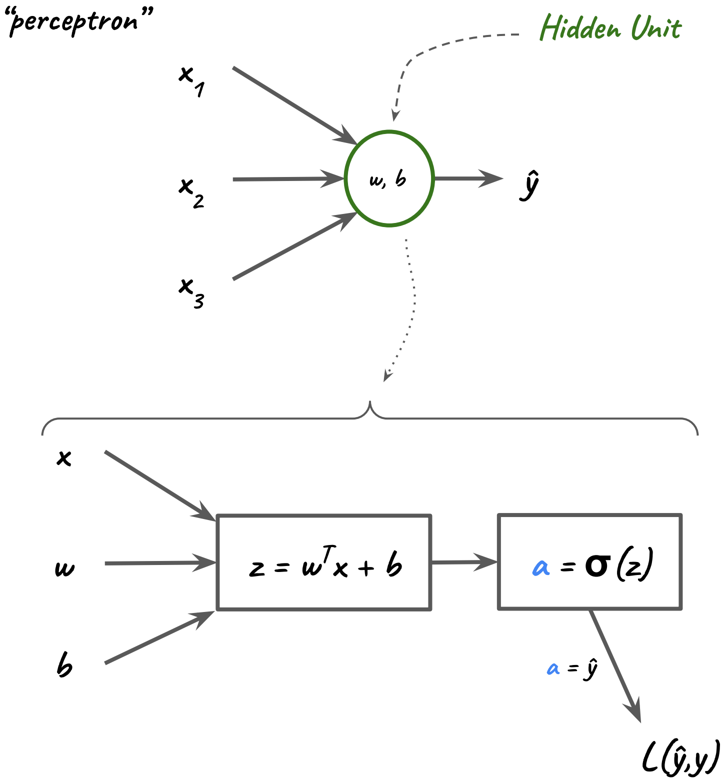
1) take in input the features x
2) combine the features with the hidden unit weights and bias
$$z = w^T x + b$$
3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone *hidden unit*, in this case a is equal to the output value $\hat{y}$.

Let's say that the function that we apply is, e.g., the *sigmoid* function that we saw in the first lesson, and that I reported in the previous slides.

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

The core element of a neural network is the *hidden unit* (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.



The meaning of that circle symbol is:

1) take in input the features x
2) combine the features with the hidden unit weights and bias
$$z = w^T x + b$$
3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone *hidden unit*, in this case a is equal to the output value $\hat{y}$.

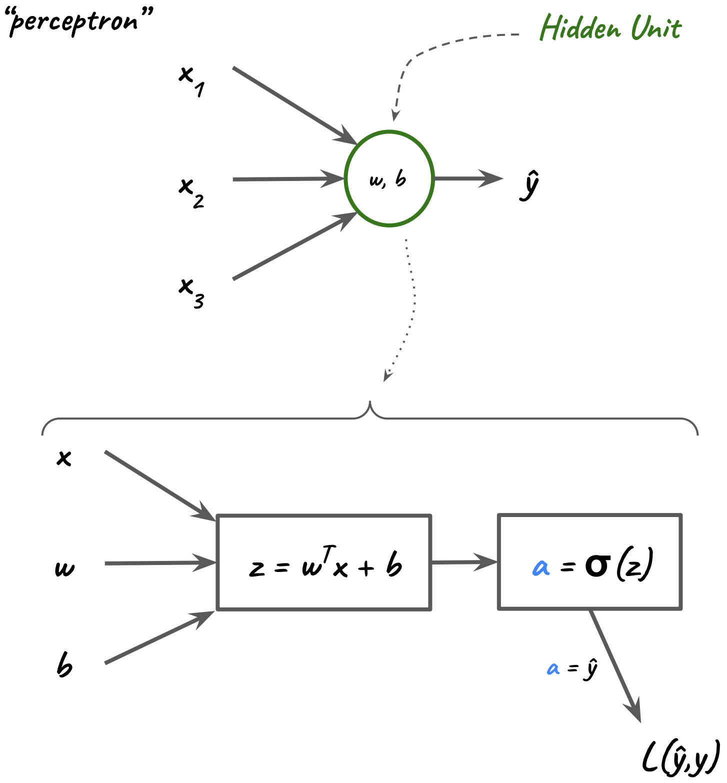
Let's say that the function that we apply is, e.g., the *sigmoid* function that we saw in the first lesson, and that I reported in the previous slides.

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

So, I take the input features, I compute $a = \sigma(w^T x + b)$, that activation value is actually the output $\hat{y}$, a predicted label, that I can compare with the real value associated to that set of features x, measuring a loss $L(\hat{y}, y)$.

A computation graph which is actually logistic regression

The core element of a neural network is the ***hidden unit*** (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.



The meaning of that circle symbol is:

1) take in input the features x
2) combine the features with the hidden unit weights and bias
$$z = w^T x + b$$
3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone ***hidden unit***, in this case a is equal to the output value $\hat{y}$.

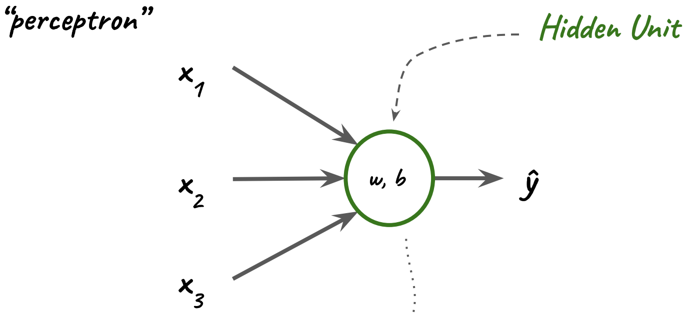
Let's say that the function that we apply is, e.g., the *sigmoid* function that we saw in the first lesson, and that I reported in the previous slides.

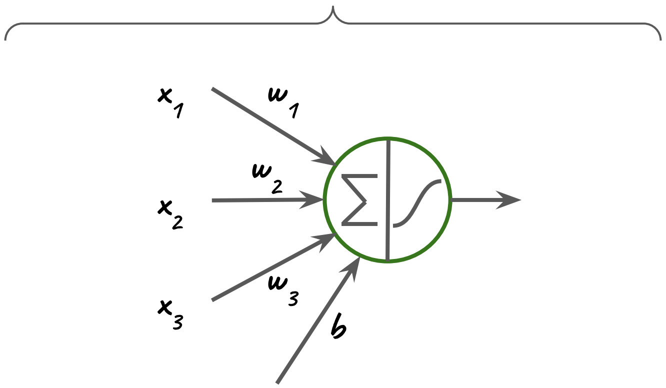
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

So, I take the input features, I compute $a = \sigma(w^T x + b)$, that activation value is actually the output $\hat{y}$, a predicted label, that I can compare with the real value associated to that set of features x, measuring a loss $\mathcal{L}(\hat{y}, y)$.

That is logistic regression.

The core element of a neural network is the ***hidden unit*** (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.





The meaning of that circle symbol is:

1) take in input the features x
2) combine the features with the hidden unit weights and bias
$$z = w^T x + b$$
3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone ***hidden unit***, in this case a is equal to the output value $\hat{y}$.

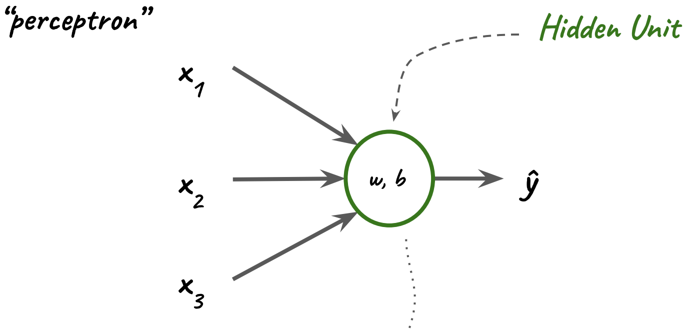
Let's say that the function that we apply is, e.g., the *sigmoid* function that we saw in the first lesson, and that I reported in the previous slides.

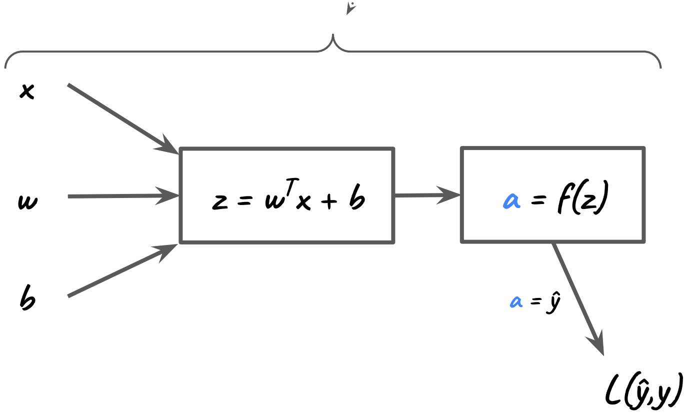
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

So, I take the input features, I compute $a = \sigma(w^T x+b)$, that activation value is actually the output $\hat{y}$, a predicted label, that I can compare with the real value associated to that set of features x, measuring a loss $L(\hat{y},y)$.

That is logistic regression.

The core element of a neural network is the ***hidden unit*** (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.





The meaning of that circle symbol is:

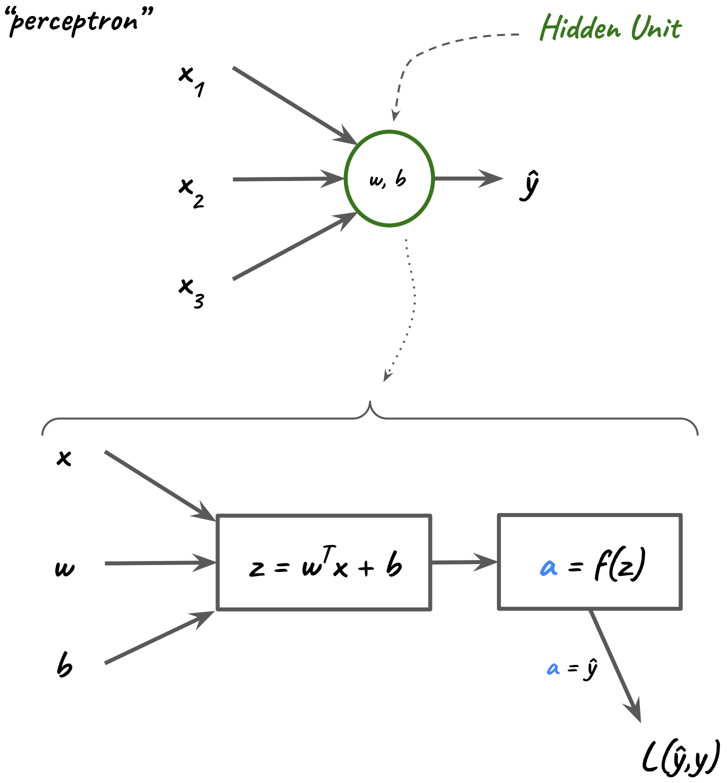
1) take in input the features $\mathbf{x}$
2) combine the features with the hidden unit weights and bias
$$z = w^T x + b$$
3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone ***hidden unit***, in this case a is equal to the output value $\hat{y}$.

Let's say that the function that we apply is instead a linear function
$$y = f(z) = z$$
So, I take the input features, I compute $\hat{y} = a = w^T x + b$, and measure a loss between the predicted and real label: $\mathcal{L}(\hat{y}, y)$.

A computation graph which is actually linear regression

The core element of a neural network is the *hidden unit* (previously known as *neuron*).
To understand what it actually does, let's look at a computation graph, with a simple three features input example.



The meaning of that circle symbol is:

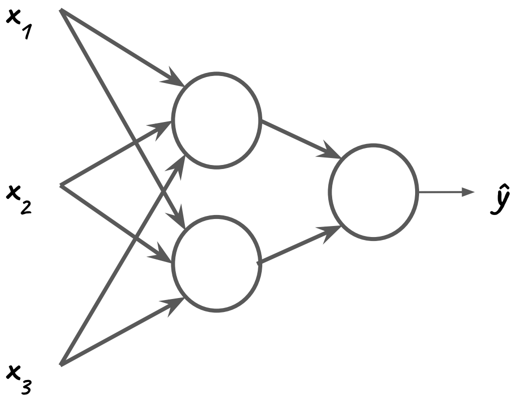
1) take in input the features $\boldsymbol{x}$
2) combine the features with the hidden unit weights and bias
$$z = w^T x + b$$
3) compute the a value (*activation*) applying a function (here $\sigma(z)$)

This is a lone *hidden unit*, in this case a is equal to the output value $\hat{y}$.

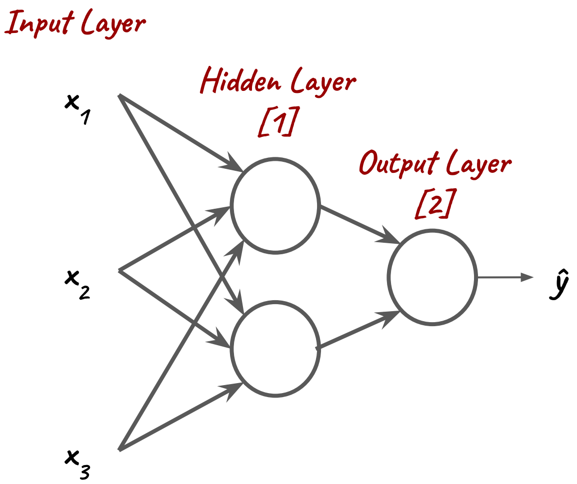
Let's say that the function that we apply is instead a linear function
$$y = f(z) = z$$
So, I take the input features, I compute $\hat{y} = a = w^T x + b$, and measure a loss between the predicted and real label: $L(\hat{y}, y)$.

That is linear regression.

What happens if I have more *hidden units*? Then you have a *neural network*.

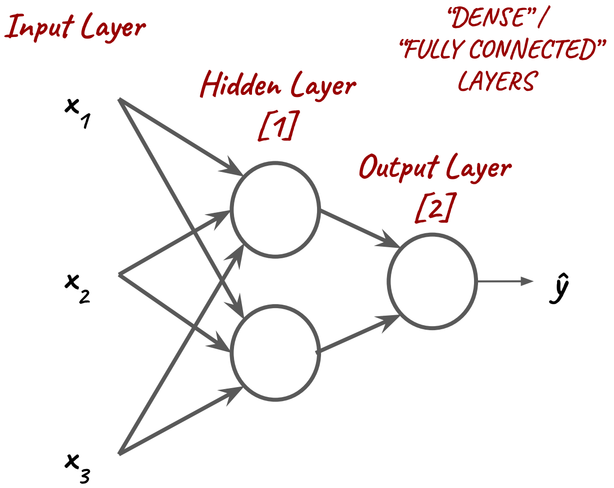


What happens if I have more *hidden units*? Then you have a *neural network*.



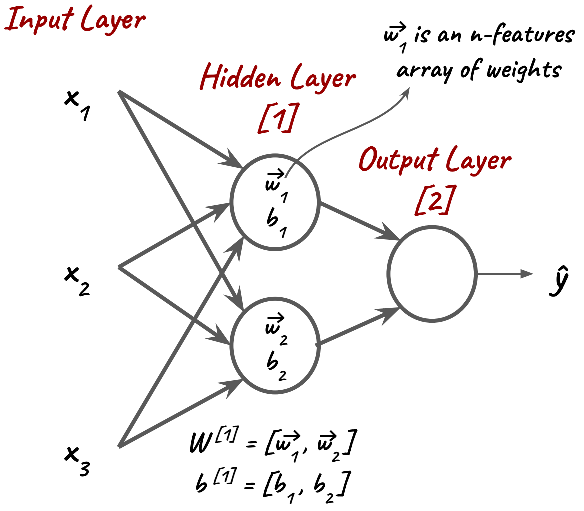
Here we have a two-layers neural network.
The layer with the input features is called the ***input layer***.
The final layer, the one that outputs the prediction $\hat{y}$, is called the ***output layer***.
The layers in between are called the ***hidden layers***. In this case there is only one ***hidden layer***, with two ***hidden units***. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the ***input layer*** features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

What happens if I have more *hidden units*? Then you have a *neural network*.



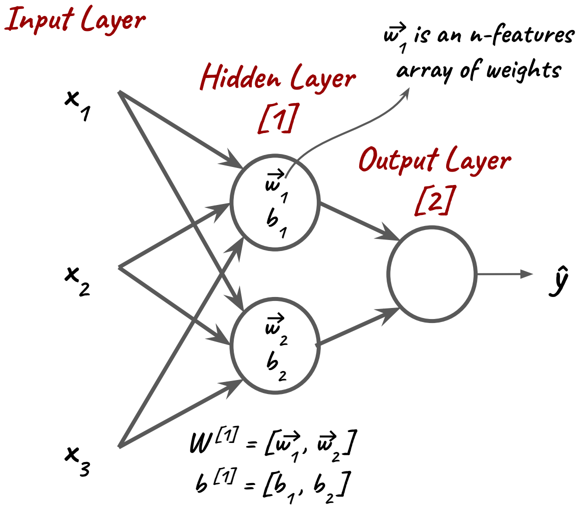
Here we have a two-layers neural network.
The layer with the input features is called the ***input layer***.
The final layer, the one that outputs the prediction $\hat{y}$, is called the ***output layer***.
The layers in between are called the ***hidden layers***. In this case there is only one ***hidden layer***, with two ***hidden units***. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the ***input layer*** features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

What happens if I have more *hidden units*? Then you have a *neural network*.



Here we have a two-layers neural network.
The layer with the input features is called the ***input layer***.
The final layer, the one that outputs the prediction $\hat{y}$, is called the ***output layer***.
The layers in between are called the ***hidden layers***. In this case there is only one ***hidden layer***, with two ***hidden units***. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the ***input layer*** features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

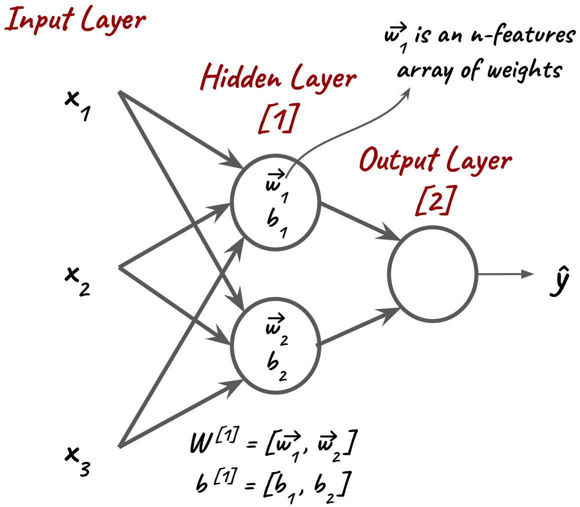
What happens if I have more *hidden units*? Then you have a *neural network*.



Here we have a two-layers neural network.
The layer with the input features is called the *input layer*.
The final layer, the one that outputs the prediction $\hat{y}$, is called the *output layer*.
The layers in between are called the *hidden layers*. In this case there is only one *hidden layer*, with two *hidden units*. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the *input layer* features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

8 parameters + 3 parameters = 11 parameters

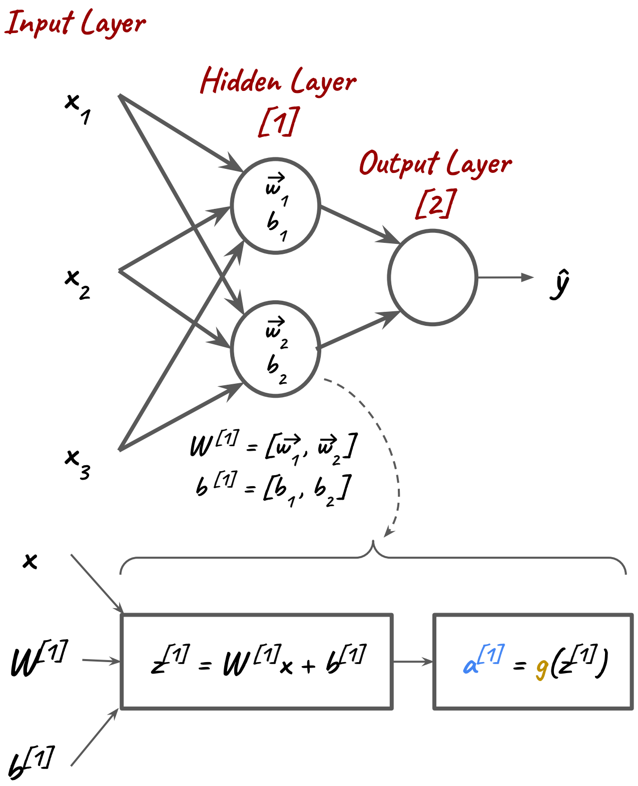
What happens if I have more *hidden units*? Then you have a *neural network*.



Here we have a two-layers neural network.
The layer with the input features is called the *input layer*.
The final layer, the one that outputs the prediction $\hat{y}$, is called the *output layer*.
The layers in between are called the *hidden layers*. In this case there is only one *hidden layer*, with two *hidden units*. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the *input layer* features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

What is this thing actually doing?

What happens if I have more *hidden units*? Then you have a *neural network*.

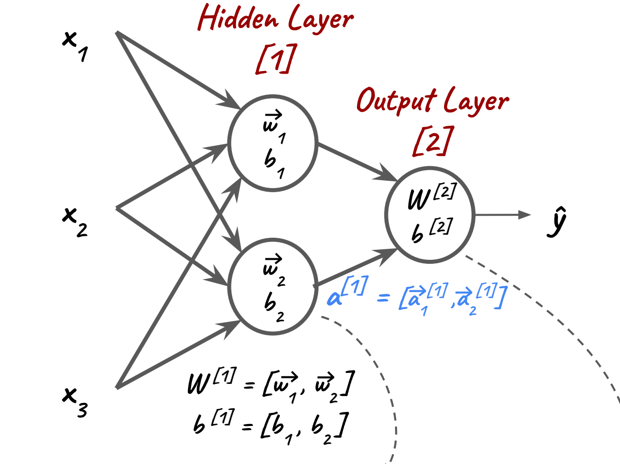


Here we have a two-layers neural network.
The layer with the input features is called the *input layer*.
The final layer, the one that outputs the prediction $\hat{y}$, is called the *output layer*.
The layers in between are called the *hidden layers*. In this case there is only one *hidden layer*, with two *hidden units*. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the *input layer* features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

What is this thing actually doing?
First, it takes in input all the features from the *input layer*, and evaluates the *first layer* activation $a^{[1]}$, with a function $g(z^{[1]})$ called the *activation function* that can be e.g., a logistic, a ReLU, a linear function, a tanh, we'll see in a few minutes the most common ones.

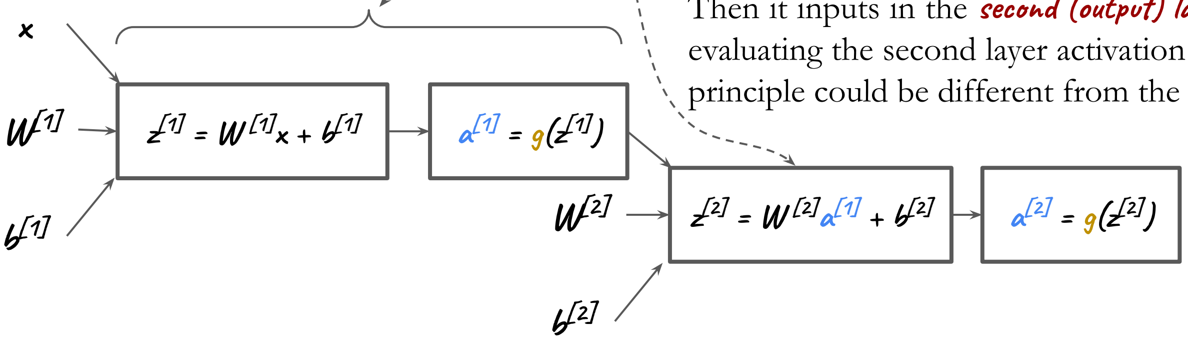
A Neural Network - Mathematics

What happens if I have more *hidden units*? Then you have a *neural network*.



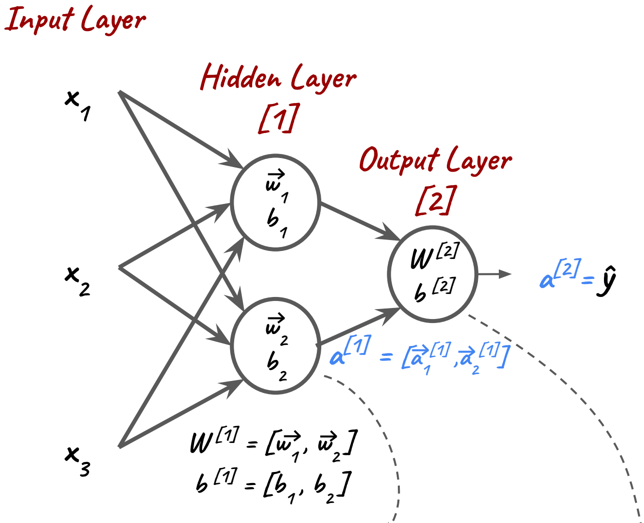
Here we have a two-layers neural network.
The layer with the input features is called the *input layer*.
The final layer, the one that outputs the prediction $\hat{y}$, is called the *output layer*.
The layers in between are called the *hidden layers*. In this case there is only one *hidden layer*, with two *hidden units*. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the *input layer* features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

What is this thing actually doing?
First, it takes in input all the features from the *input layer*, and evaluates the *first layer* activation $a^{[1]}$, with a function $g(z^{[1]})$ called the *activation function* that can be e.g., a logistic, a ReLU, a linear function, a tanh, we'll see in a few minutes the most common ones.
Then it inputs in the *second (output) layer* the *activation* $a^{[1]}$ calculated in the previous layer, evaluating the second layer activation $a^{[2]}$ with another activation function $g(z^{[2]})$, which in principle could be different from the one used in the first layer.



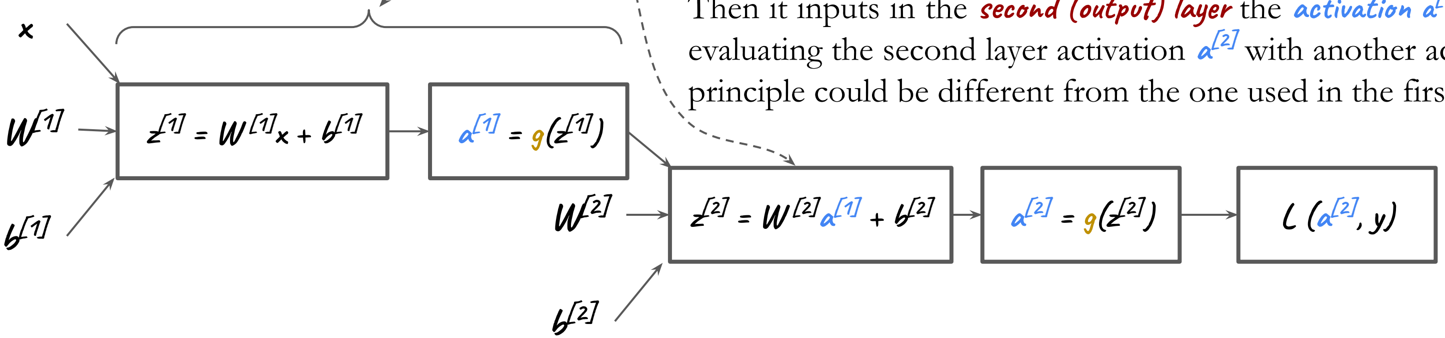
A Neural Network - Mathematics

What happens if I have more *hidden units*? Then you have a *neural network*.



Here we have a two-layers neural network.
The layer with the input features is called the *input layer*.
The final layer, the one that outputs the prediction $\hat{y}$, is called the *output layer*.
The layers in between are called the *hidden layers*. In this case there is only one *hidden layer*, with two *hidden units*. Sometimes the *hidden* in *layer* is dropped for simplicity.
Notice how the *input layer* features enters **ALL** the hidden units of the first layer, and how **ALL** the outputs of the first layer enters the output layer.
Everything is "*fully-connected*" in this Neural Network.

What is this thing actually doing?
First, it takes in input all the features from the *input layer*, and evaluates the *first layer* activation $a^{[1]}$, with a function $g(z^{[1]})$ called the *activation function* that can be e.g., a logistic, a ReLU, a linear function, a tanh, we'll see in a few minutes the most common ones.
Then it inputs in the *second (output) layer* the *activation* $a^{[1]}$ calculated in the previous layer, evaluating the second layer activation $a^{[2]}$ with another activation function $g(z^{[2]})$, which in principle could be different from the one used in the first layer.

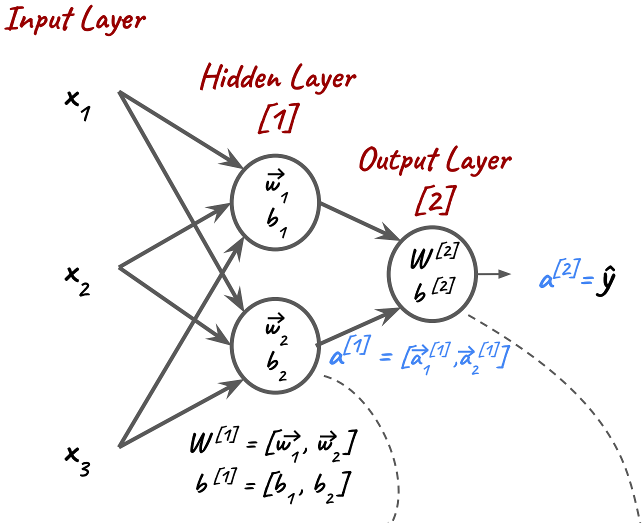


This is the output layer, so:

$$a^{[2]} = \hat{y}$$

and we can compute a loss function between the predicted and observed labels.

What happens if I have more *hidden units*? Then you have a *neural network*.



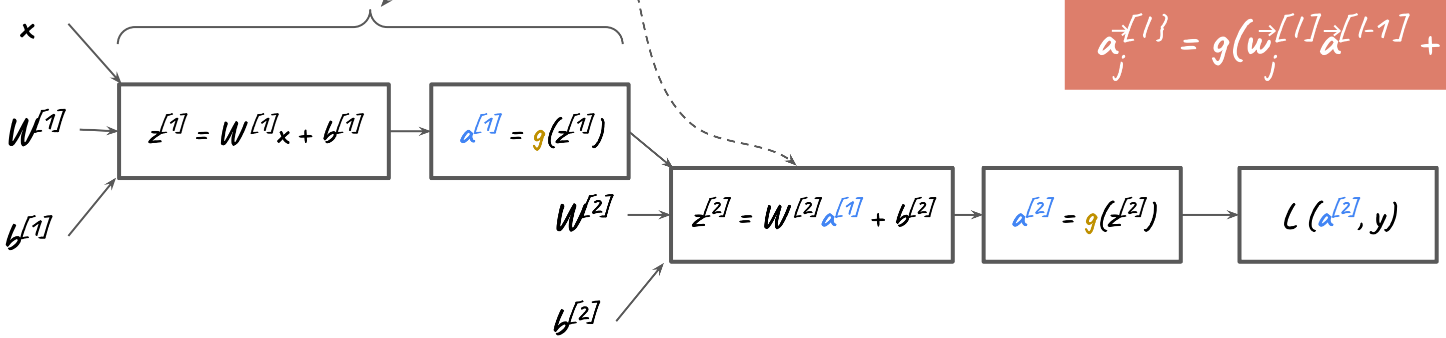
For multiple examples $x^{(i)}$ in the input layer this becomes simply:

$$a^{[1](i)} = g(W^{[1]}x^{(i)} + b^{[1]})$$
$$a^{[2](i)} = g(W^{[1]}a^{[1](i)} + b^{[1]})$$

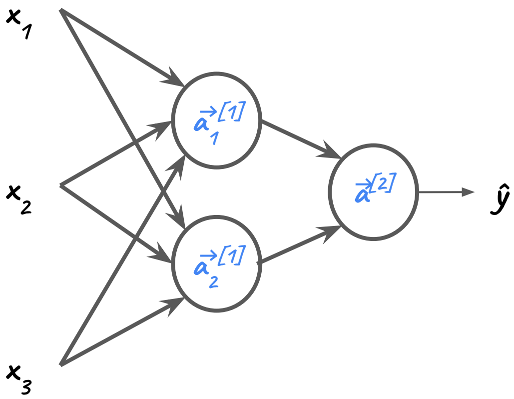
which is easily generalizable to whatever number of hidden layer with whatever number of hidden units you might desire for your application.

This is something you could easily code in NumPy:

$$a_j^{[l]} = g(\vec{w}_j^{[l]}\vec{a}^{[l-1]} + b_j^{[l]})$$

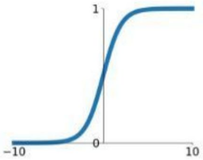


Activation functions are a key element of Neural Networks.
There are plenty to choose, but usually everybody uses the same three or four.

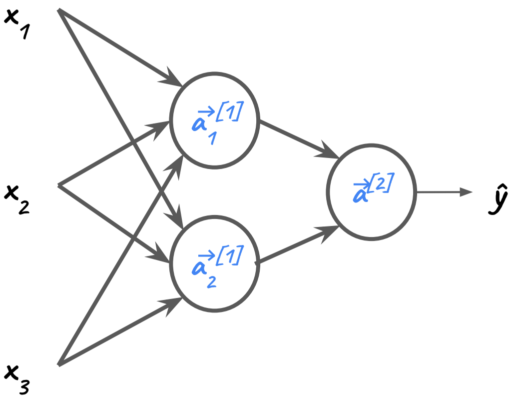


Sigmoid

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

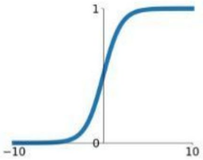


Activation functions are a key element of Neural Networks.
There are plenty to choose, but usually everybody uses the same three or four.



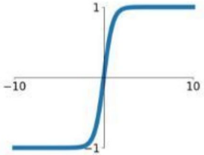
Sigmoid

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



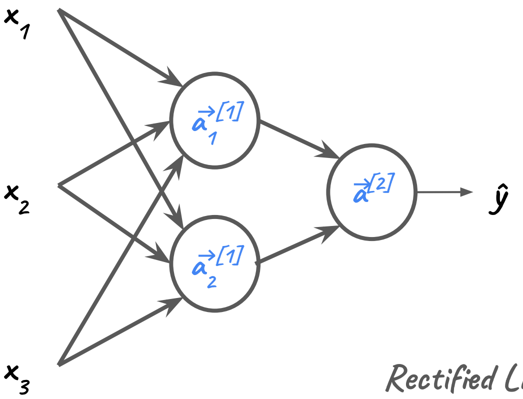
tanh

$\tanh(x)$



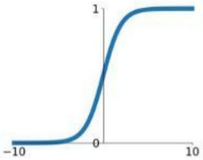
Activation functions

Activation functions are a key element of Neural Networks.
There are plenty to choose, but usually everybody uses the same three or four.



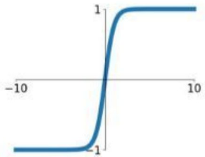
Sigmoid

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



tanh

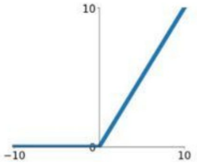
$\tanh(x)$



ReLU

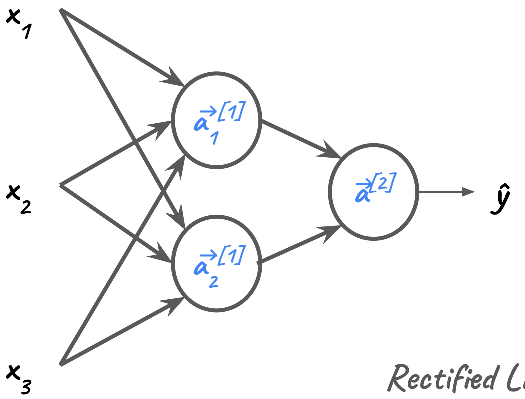
$\max(0, x)$

Rectified Linear Unit



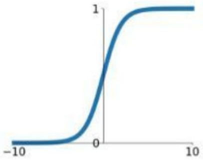
Activation functions

Activation functions are a key element of Neural Networks.
There are plenty to choose, but usually everybody uses the same three or four.



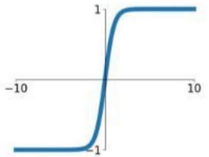
Sigmoid

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



tanh

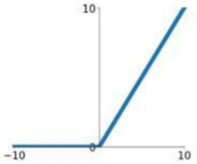
$\tanh(x)$



ReLU

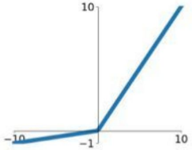
$\max(0, x)$

Rectified Linear Unit



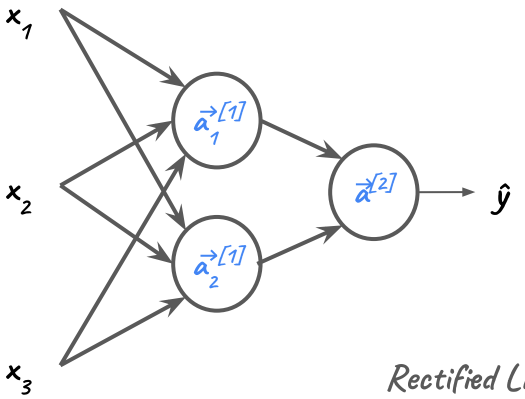
Leaky ReLU

$\max(0.1x, x)$



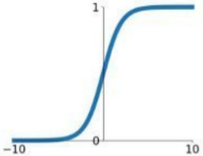
Activation functions

Activation functions are a key element of Neural Networks.
There are plenty to choose, but usually everybody uses the same three or four.



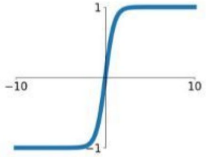
Sigmoid

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



tanh

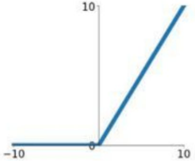
$\tanh(x)$



ReLU

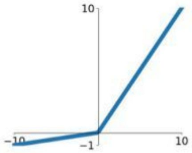
$\max(0, x)$

Rectified Linear Unit



Leaky ReLU

$\max(0.1x, x)$

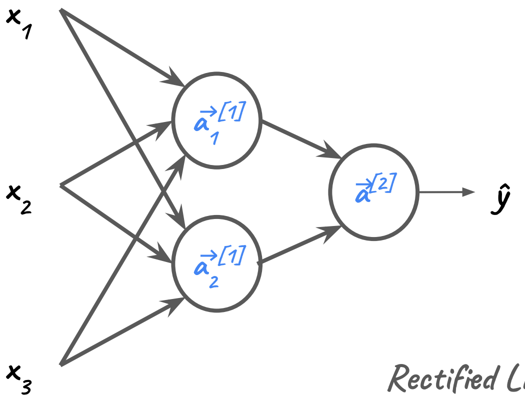


Fast to compute, but sometimes problems could arise in gradient descent

(but in majority of cases they work fine, and there other ways to solve those "vanishing" gradient issues)

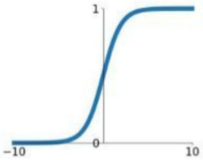
Activation functions

Activation functions are a key element of Neural Networks.
There are plenty to choose, but usually everybody uses the same three or four.



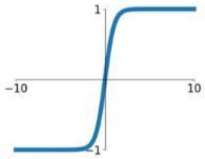
Sigmoid

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



tanh

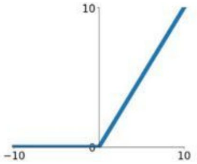
$\tanh(x)$



ReLU

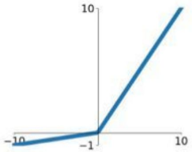
$\max(0, x)$

Rectified Linear Unit



Leaky ReLU

$\max(0.1x, x)$



They insert non-linearities into the whole process. Well, except for the linear activation function, which still is fundamental, e.g., for regression task as the activation function of the final layer.

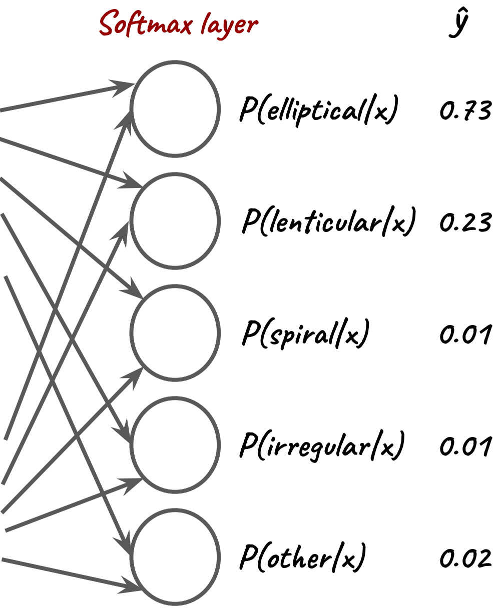
Output layer activation function

- Classification
  - Binary → *Sigmoid Activation*
  - Multiclass → *Softmax Activation*
  - Multilabel → *Sigmoid Activation*
- Regression → *Linear Activation*

We saw in the first lecture that a possible solution to problems with multiple classes is to use a **one-vs-all** (a.k.a. **one-vs-rest**) approach, or a **one-vs-one** algorithm, where each class probability is evaluated against all the others combined or taken individually.
Neural Networks offers a way out of training *#Classes(#Classes-1)/2* models, similar to what **OvA** does, but all hardcoded within the network as the ***softmax* output layer.**

We saw in the first lecture that a possible solution to problems with multiple classes is to use a **one-vs-all** (a.k.a. **one-vs-rest**) approach, or a **one-vs-one** algorithm, where each class probability is evaluated against all the others combined or taken individually.
Neural Networks offers a way out of training *#Classes(#Classes-1)/2* models, similar to what **OvA** does, but all hardcoded within the network as the *softmax output layer.*

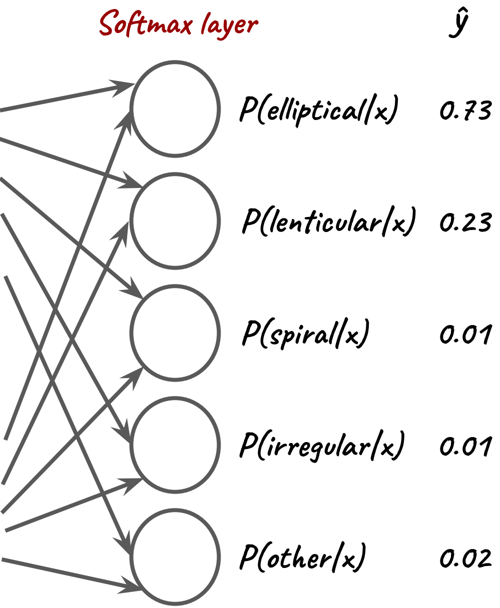
The *softmax layer* is a layer where the number of hidden units is equal to the number of *C classes*; the hidden units in the *softmax* layer will give back the probability that an example belongs to that particular class, so in the De Vaucoulers galaxy classification example:



The softmax layer

We saw in the first lecture that a possible solution to problems with multiple classes is to use a **one-vs-all** (a.k.a. **one-vs-rest**) approach, or a **one-vs-one** algorithm, where each class probability is evaluated against all the others combined or taken individually.
Neural Networks offers a way out of training *#Classes(#Classes-1)/2* models, similar to what **OvA** does, but all hardcoded within the network as the *softmax output layer.*

The *softmax layer* is a layer where the number of hidden units is equal to the number of *C classes*; the hidden units in the *softmax* layer will give back the probability that an example belongs to that particular class, so in the De Vaucoulers galaxy classification example:



How does this actually work? The input of the final layer is:

$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

and for each class *C*, the output of the softmax layer will be:

$$a_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{j=1}^{C} e^{Z_j^{[L]}}}$$

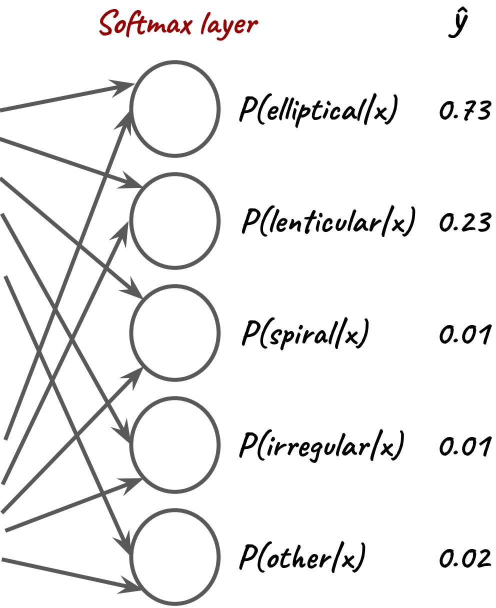
normalized to 1

which naturally outputs values between 0 and 1 → **probabilities**

The softmax layer

We saw in the first lecture that a possible solution to problems with multiple classes is to use a **one-vs-all** (a.k.a. **one-vs-rest**) approach, or a **one-vs-one** algorithm, where each class probability is evaluated against all the others combined or taken individually.
Neural Networks offers a way out of training *#Classes(#Classes-1)/2* models, similar to what **OvA** does, but all hardcoded within the network as the *softmax output layer.*

The *softmax layer* is a layer where the number of hidden units is equal to the number of *C classes*; the hidden units in the *softmax* layer will give back the probability that an example belongs to that particular class, so in the De Vaucoulers galaxy classification example:



How does this actually work? The input of the final layer is:

$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

and for each class *C*, the output of the softmax layer will be:

$$a_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{j=1}^{C} e^{Z_j^{[L]}}}$$

normalized to 1

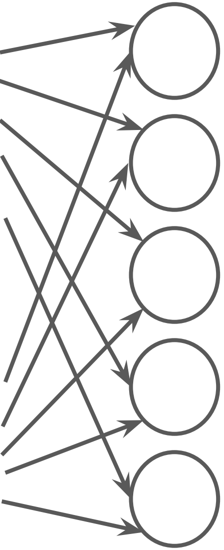
which naturally outputs values between 0 and 1 → **probabilities**
How does *softmax* training work?

The softmax layer

We saw in the first lecture that a possible solution to problems with multiple classes is to use a **one-vs-all** (a.k.a. **one-vs-rest**) approach, or a **one-vs-one** algorithm, where each class probability is evaluated against all the others combined or taken individually.
Neural Networks offers a way out of training *#Classes(#Classes-1)/2* models, similar to what **OvA** does, but all hardcoded within the network as the *softmax output layer.*

The *softmax layer* is a layer where the number of hidden units is equal to the number of *C classes*; the hidden units in the *softmax* layer will give back the probability that an example belongs to that particular class, so in the De Vaucoulers galaxy classification example:

Softmax layer



	$\hat{y}$	y
P(elliptical|x)	0.73	1
P(lenticular|x)	0.23	0
P(spiral|x)	0.01	0
P(irregular|x)	0.01	0
P(other|x)	0.02	0

$$\mathcal{L}(y, \hat{y}) = -\sum_{j=1}^{C} y_i \log \hat{y}_j$$

$$J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

How does this actually work? The input of the final layer is:

$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

and for each class C, the output of the softmax layer will be:

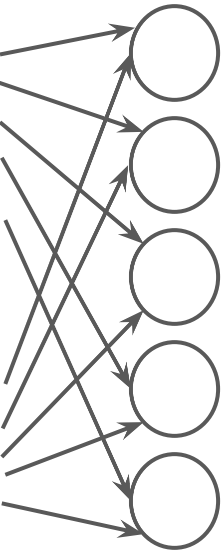
$$a_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{j=1}^{C} e^{Z_j^{[L]}}}$$

normalized to 1

which naturally outputs values between 0 and 1 → **probabilities**
How does *softmax* training work?

We saw in the first lecture that a possible solution to problems with multiple classes is to use a **one-vs-all** (a.k.a. **one-vs-rest**) approach, or a **one-vs-one** algorithm, where each class probability is evaluated against all the others combined or taken individually.
Neural Networks offers a way out of training *#Classes(#Classes-1)/2* models, similar to what **OvA** does, but all hardcoded within the network as the *softmax output layer.*

The *softmax layer* is a layer where the number of hidden units is equal to the number of *C classes*; the hidden units in the *softmax* layer will give back the probability that an example belongs to that particular class, so in the De Vaucoulers galaxy classification example:



Hardmax layer	$\hat{y}$	y
$P(elliptical \mid x)$	1	1
$P(lenticular \mid x)$	0	0
$P(spiral \mid x)$	0	0
$P(irregular \mid x)$	0	0
$P(other \mid x)$	0	0

$$\mathcal{L}(y, \hat{y}) = -\sum_{j=1}^{C} y_i \log \hat{y}_j$$

$$J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

How does this actually work? The input of the final layer is:

$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

and for each class C, the output of the softmax layer will be:

$$a_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{j=1}^{C} e^{Z_j^{[L]}}}$$

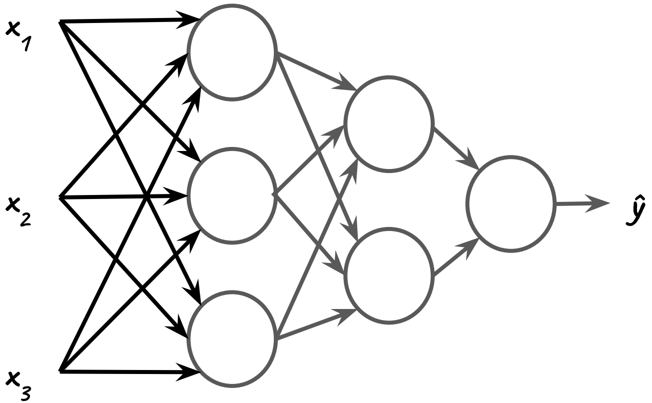
normalized to 1

which naturally outputs values between 0 and 1 → **probabilities**
How does *softmax* training work?

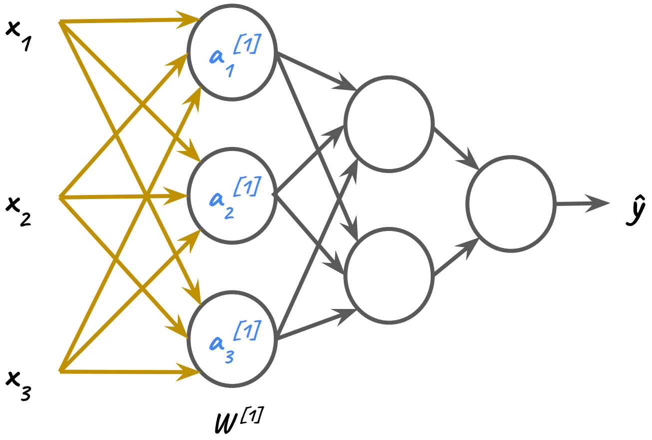
Hardmax = a *softmax* that only returns 0 and 1.

Training a Neural Network

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}$ $l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



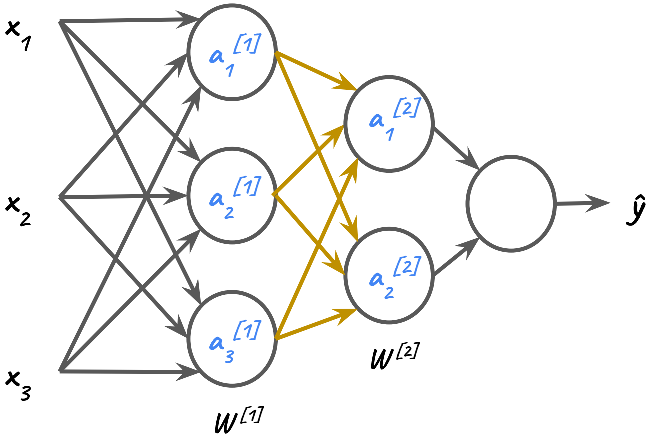
Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.

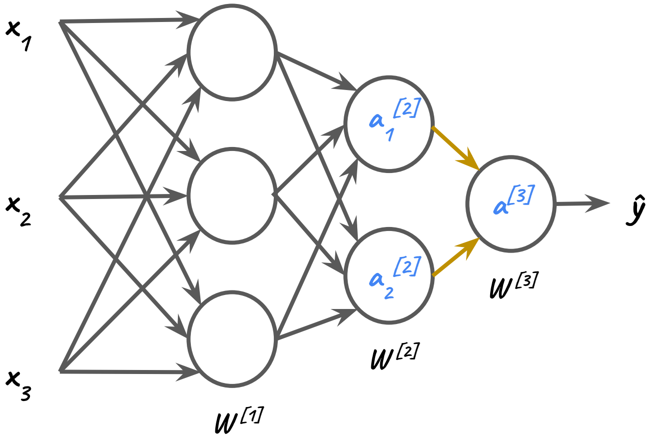
Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.

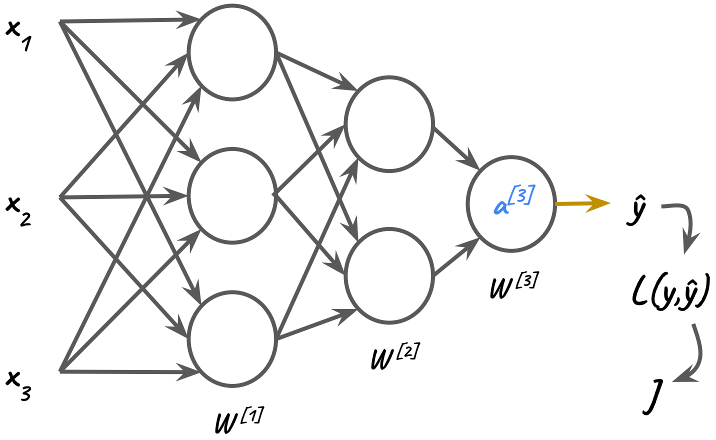
Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}$ $l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the *FORTHPROPAGATION* and the *BACKPROPAGATION*.



FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.

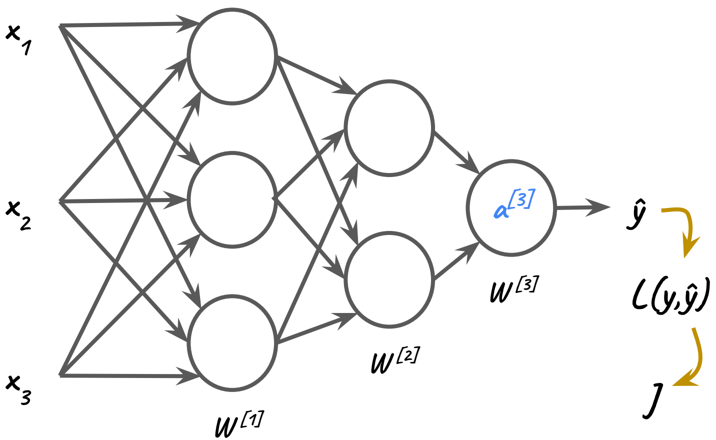
Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.

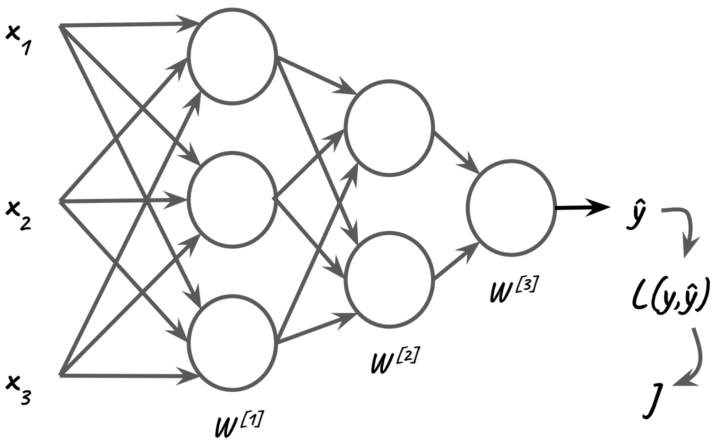
Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses $\mathcal{L}$.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



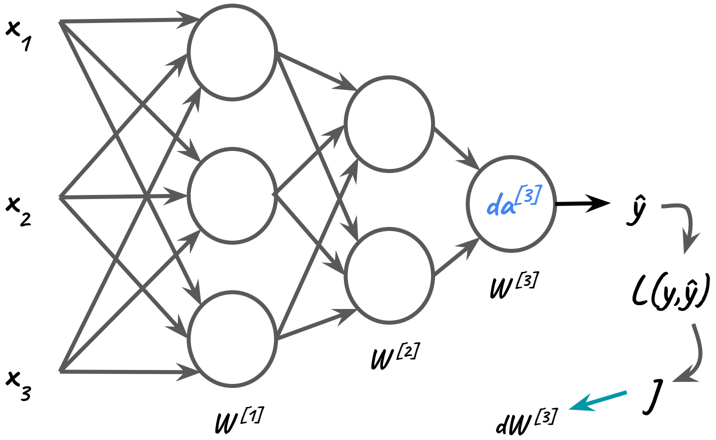
FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

BACKPROPAGATION

Is the process of passing through the *hidden layers* backwards, updating each time the weights $W^{[l]}$ of every *hidden layer* with the new values computed during gradient descent by multiplying the gradient of the cost with the learning rate.

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



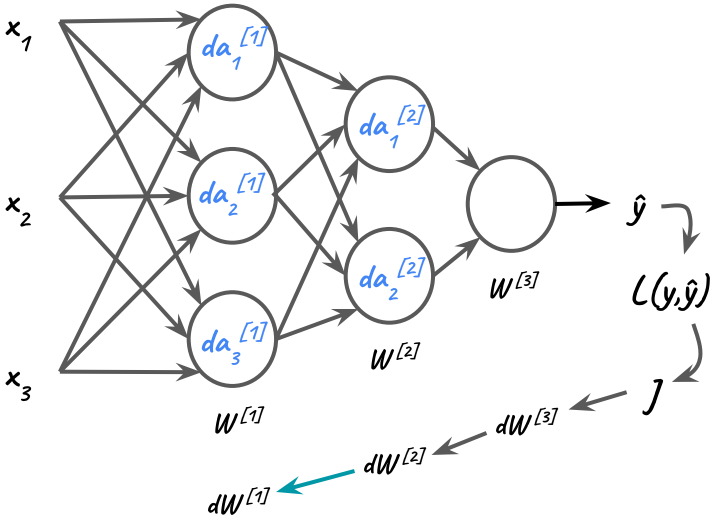
FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

BACKPROPAGATION

Is the process of passing through the *hidden layers* backwards, updating each time the weights $W^{[l]}$ of every *hidden layer* with the new values computed during gradient descent by multiplying the gradient of the cost with the learning rate.

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}$ $l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses $\mathcal{L}$.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



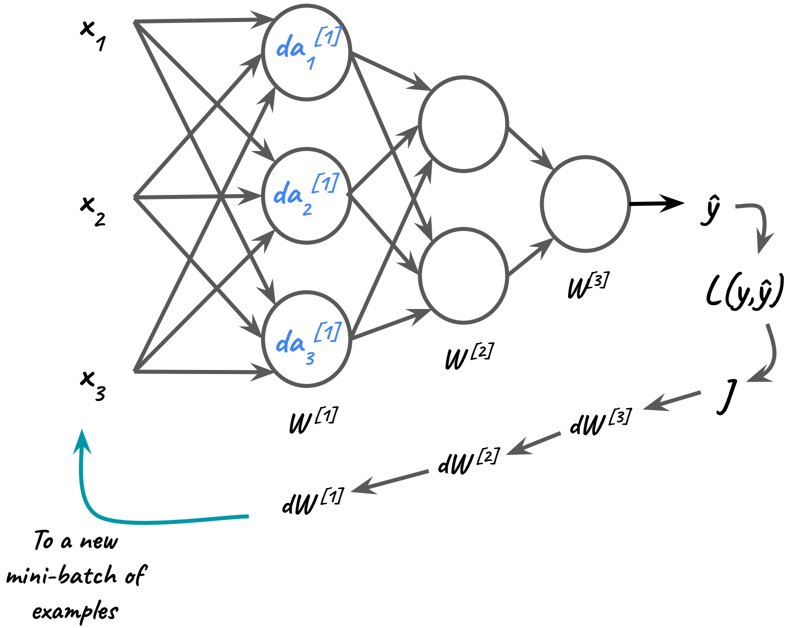
FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

BACKPROPAGATION

Is the process of passing through the *hidden layers* backwards, updating each time the weights $W^{[l]}$ of every *hidden layer* with the new values computed during gradient descent by multiplying the gradient of the cost with the learning rate.

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the *FORTHPROPAGATION* and the *BACKPROPAGATION*.



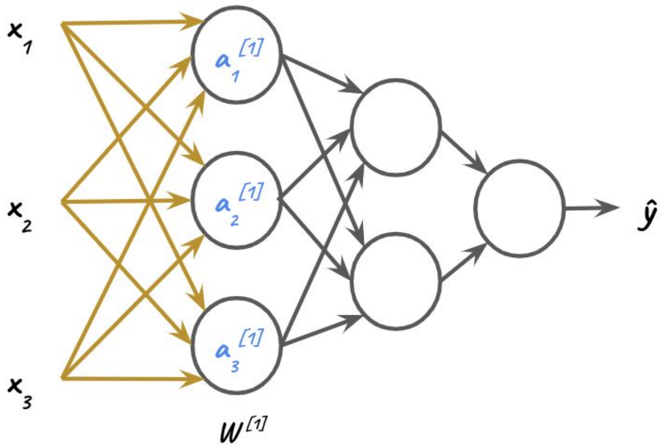
FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

BACKPROPAGATION

Is the process of passing through the *hidden layers* backwards, updating each time the weights $W^{[l]}$ of every *hidden layer* with the new values computed during gradient descent by multiplying the gradient of the cost with the learning rate.

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



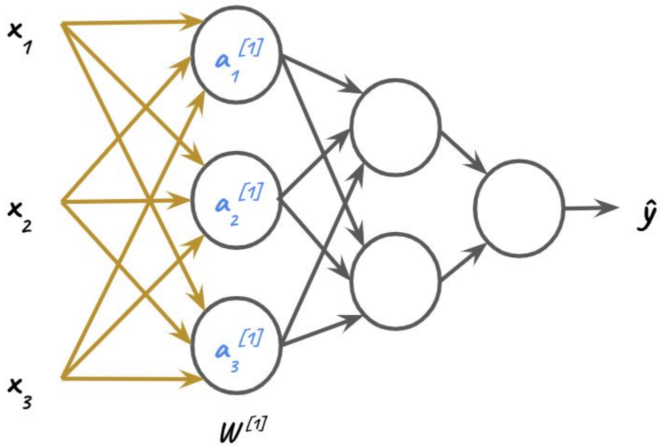
FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

BACKPROPAGATION

Is the process of passing through the *hidden layers* backwards, updating each time the weights $W^{[l]}$ of every *hidden layer* with the new values computed during gradient descent by multiplying the gradient of the cost with the learning rate.

Training a Neural Network means finding the best set of parameters $W^{[l]}, b^{[l]}\ l = 1 \dots L$ for all the hidden layers. The concept is the same we saw in the first lesson: perform gradient descent, minimizing a cost function J, sum of all the single examples losses L.
The two key concepts here are the FORTHPROPAGATION and the BACKPROPAGATION.



FORTHPROPAGATION

Is the propagation of the input data from the *input layer* across all the *hidden layers*, up to the *output layer*, the one we saw previously.
At this point, the predicted label $\hat{y}$ is evaluated, and therefore a measure of the cost function for that set of examples, thus a measurement of the gradient.
And that's where backpropagation starts.

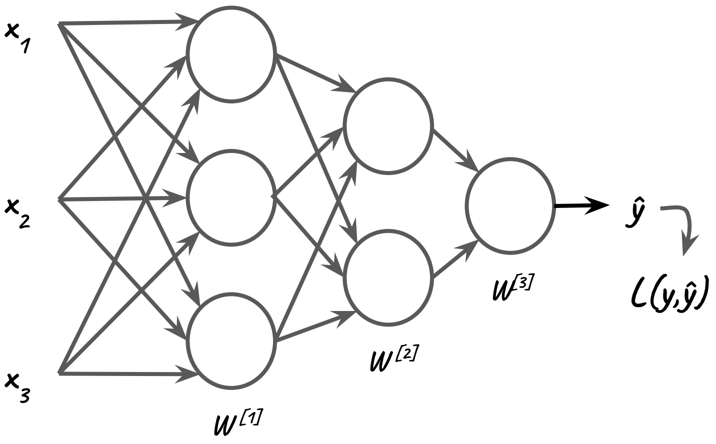
BACKPROPAGATION

Is the process of passing through the *hidden layers* backwards, updating each time the weights $W^{[l]}$ of every *hidden layer* with the new values computed during gradient descent by multiplying the gradient of the cost with the learning rate.

DEEP LEARNING

Perhaps you might wonder why it is called *"deep learning"*.
The reason is that once you start to stack layers on layers, you'll notice how the first layers *learn simple structures* from your data, and the deeper you go with the model, the more *complex things* it is able to learn and reproduce.

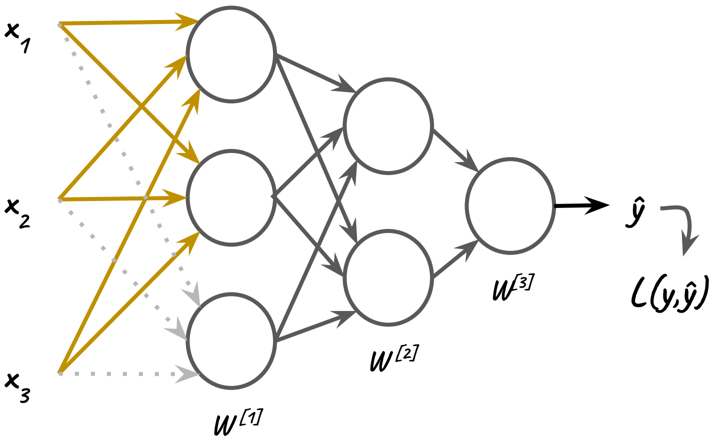
As we saw in the first day, it is important to keep a balance between the need to fit to the training data and the need have a model as generalizable as possible, avoiding overfitting. This is achieved by keeping the model as simple as possible with ***regularization***.
We saw that ***regularization*** forces the model weights to be close to (L_2) or exactly zero (L_1), under the assumption that small values for the model weight → simpler model, and it actually works fine.



L_2 and L_1 also apply for Neural Networks, being additional terms to add to the cost function J.
Early stopping, surprisingly, works also fine for Neural Networks, with the necessary caveat that you must always know what you're doing, so don't try this at home.
Another efficient technique is the ***dropout regularization***.

The concept is: for each training step, do not update *ALL* the weights in the layers, but turn a random fraction (e.g. 20%) off, and update all the others.

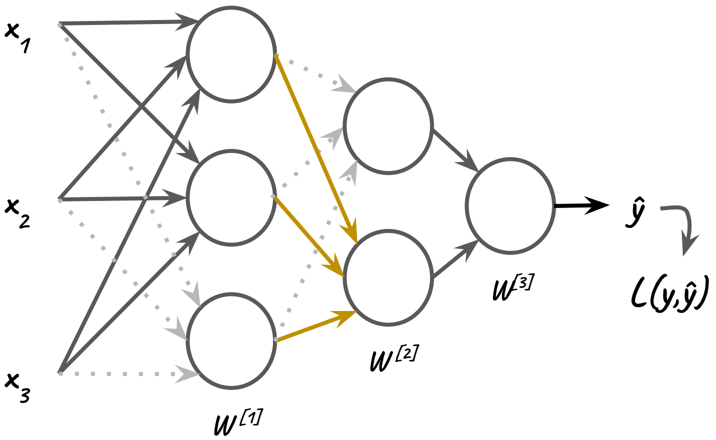
As we saw in the first day, it is important to keep a balance between the need to fit to the training data and the need have a model as generalizable as possible, avoiding overfitting. This is achieved by keeping the model as simple as possible with ***regularization***.
We saw that ***regularization*** forces the model weights to be close to (ℓ_2) or exactly zero (ℓ_1), under the assumption that small values for the model weight → simpler model, and it actually works fine.



ℓ_2 and ℓ_1 also apply for Neural Networks, being additional terms to add to the cost function J.
Early stopping, surprisingly, works also fine for Neural Networks, with the necessary caveat that you must always know what you're doing, so don't try this at home.
Another efficient technique is the *dropout regularization*.

The concept is: for each training step, do not update *ALL* the weights in the layers, but turn a random fraction (e.g. 20%) off, and update all the others.

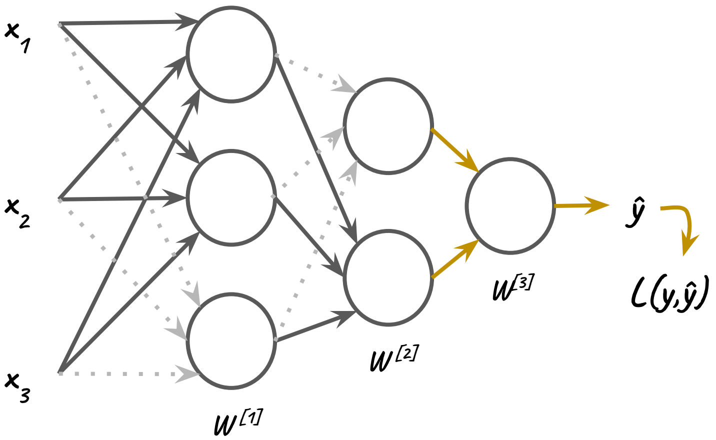
As we saw in the first day, it is important to keep a balance between the need to fit to the training data and the need have a model as generalizable as possible, avoiding overfitting. This is achieved by keeping the model as simple as possible with ***regularization***.
We saw that ***regularization*** forces the model weights to be close to (L_2) or exactly zero (L_1), under the assumption that small values for the model weight → simpler model, and it actually works fine.



L_2 and L_1 also apply for Neural Networks, being additional terms to add to the cost function J.
Early stopping, surprisingly, works also fine for Neural Networks, with the necessary caveat that you must always know what you're doing, so don't try this at home.
Another efficient technique is the *dropout regularization*.

The concept is: for each training step, do not update *ALL* the weights in the layers, but turn a random fraction (e.g. 20%) off, and update all the others.

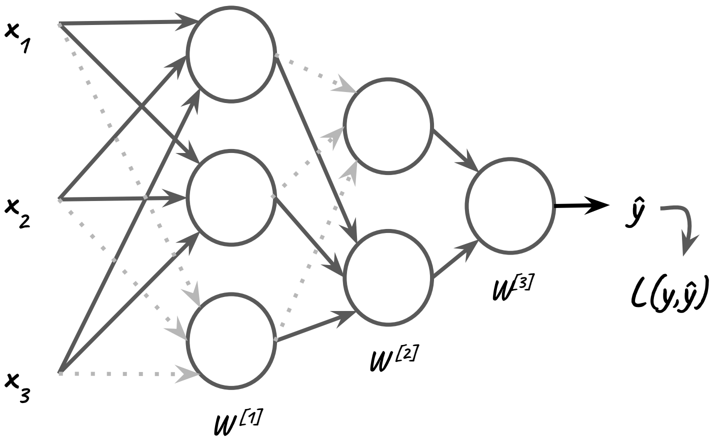
As we saw in the first day, it is important to keep a balance between the need to fit to the training data and the need have a model as generalizable as possible, avoiding overfitting. This is achieved by keeping the model as simple as possible with ***regularization***.
We saw that ***regularization*** forces the model weights to be close to (ℓ_2) or exactly zero (ℓ_1), under the assumption that small values for the model weight → simpler model, and it actually works fine.



ℓ_2 and ℓ_1 also apply for Neural Networks, being additional terms to add to the cost function J.
Early stopping, surprisingly, works also fine for Neural Networks, with the necessary caveat that you must always know what you're doing, so don't try this at home.
Another efficient technique is the *dropout regularization*.

The concept is: for each training step, do not update *ALL* the weights in the layers, but turn a random fraction (e.g. 20%) off, and update all the others.

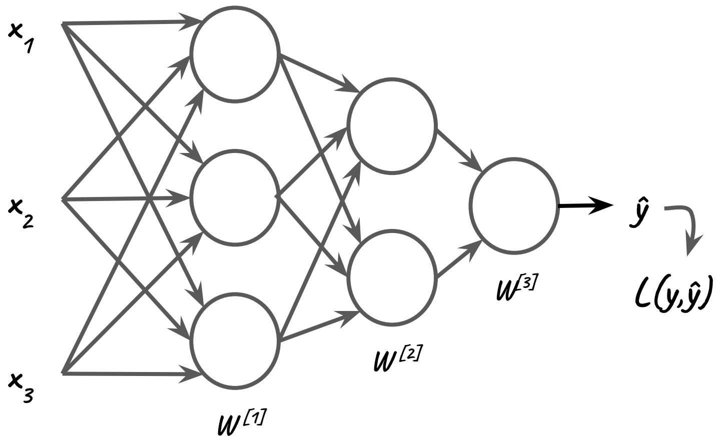
As we saw in the first day, it is important to keep a balance between the need to fit to the training data and the need have a model as generalizable as possible, avoiding overfitting. This is achieved by keeping the model as simple as possible with ***regularization***.
We saw that ***regularization*** forces the model weights to be close to (ℓ_2) or exactly zero (ℓ_1), under the assumption that small values for the model weight → simpler model, and it actually works fine.



ℓ_2 and ℓ_1 also apply for Neural Networks, being additional terms to add to the cost function J.
Early stopping, surprisingly, works also fine for Neural Networks, with the necessary caveat that you must always know what you're doing, so don't try this at home.
Another efficient technique is the ***dropout regularization***.

The concept is: for each training step, do not update *ALL* the weights in the layers, but turn a random fraction (e.g. 20%) off, and update all the others.
This works by forcing the Neural Network to spread the weights over the hidden units, shrinking the weights exactly as ℓ_2 would do.

As we saw in the first day, it is important to keep a balance between the need to fit to the training data and the need have a model as generalizable as possible, avoiding overfitting. This is achieved by keeping the model as simple as possible with *regularization*.
We saw that *regularization* forces the model weights to be close to (L_2) or exactly zero (L_1), under the assumption that small values for the model weight → simpler model, and it actually works fine.



L_2 and L_1 also apply for Neural Networks, being additional terms to add to the cost function J.
Early stopping, surprisingly, works also fine for Neural Networks, with the necessary caveat that you must always know what you're doing, so don't try this at home.
Another efficient technique is the *dropout regularization*.

The concept is: for each training step, do not update *ALL* the weights in the layers, but turn a random fraction (e.g. 20%) off, and update all the others.
This works by forcing the Neural Network to spread the weights over the hidden units, shrinking the weights exactly as L_2 would do.

A common problem found when training **DEEP** NN is the one of *vanishing/exploding gradients*, meaning that the gradient might become exponentially high or low, thus halting the whole training process.
The problem has been recently solved (2014, but the idea dates back to 1961) with *skip connections* and *residual blocks* (*ResNet*, more on later).
Another common practice to reduce the chance of *vanishing/exploding gradients* is weight initialization:

$W^{[l]}$ = [random initialization] * np.sqrt(1/$n^{[l-1]}$) "Xavier initialization"

the number of features entering the layer

*(np.sqrt(2/$n^{[l-1]}$) works better with *ReLU*

*(np.sqrt(2/($n^{[l-1]}$+$n^{[l]}$)) Bengio & co. initialization

TensorFlow

Tensorflow (by Google) is an open source library to build, train, evaluate, deploy in production, in general *work* in a ML environment.

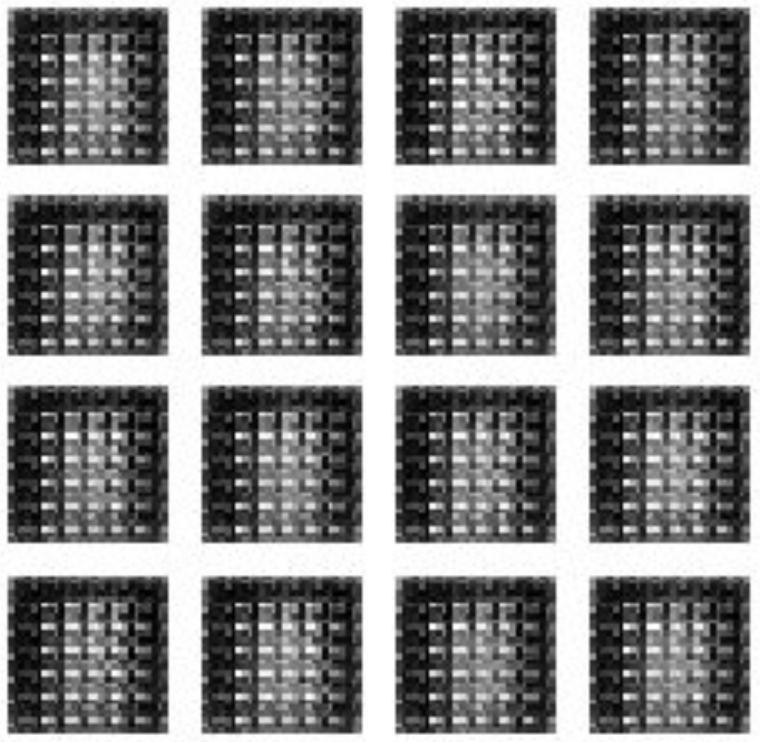
```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



https://www.tensorflow.org/

TensorFlow Keras

Tensorflow

Tensorflow (by Google) is an open source library to build, train, evaluate, deploy in production, in general *work* in a ML environment.
The Deep Learning Python API of *Tensorflow* is based on **Keras**.

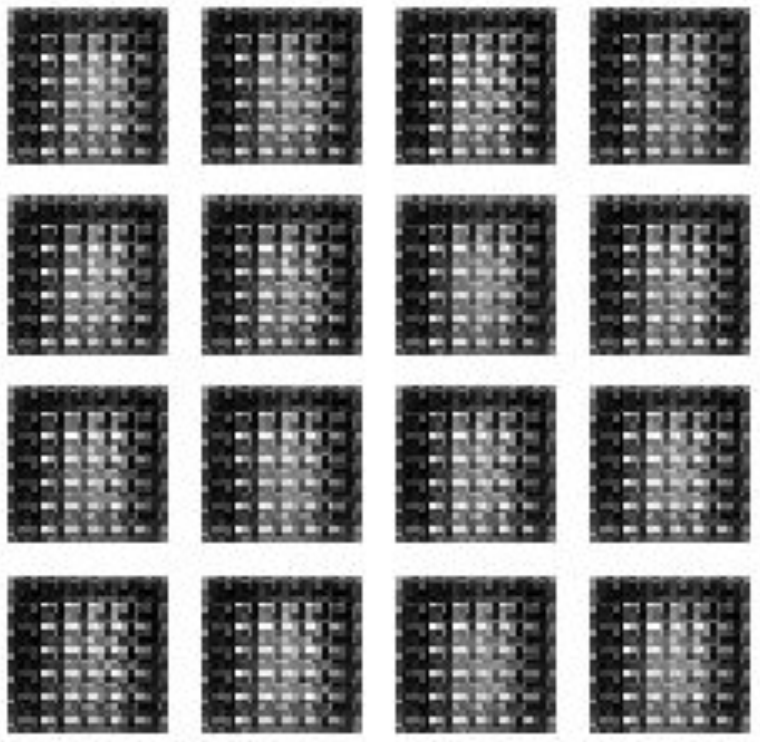
```python
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



https://www.tensorflow.org/ https://keras.io/examples/ https://keras.io/

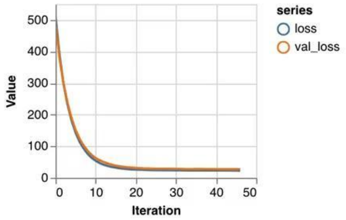
TensorFlow Keras *Tensorflow.js Demos*

STATUS

Starting training process...

Baseline loss (meanSquaredError) is 85.58

TRAINING PROGRESS



Epoch 47 of 200 completed.

Top 5 weights by magnitude

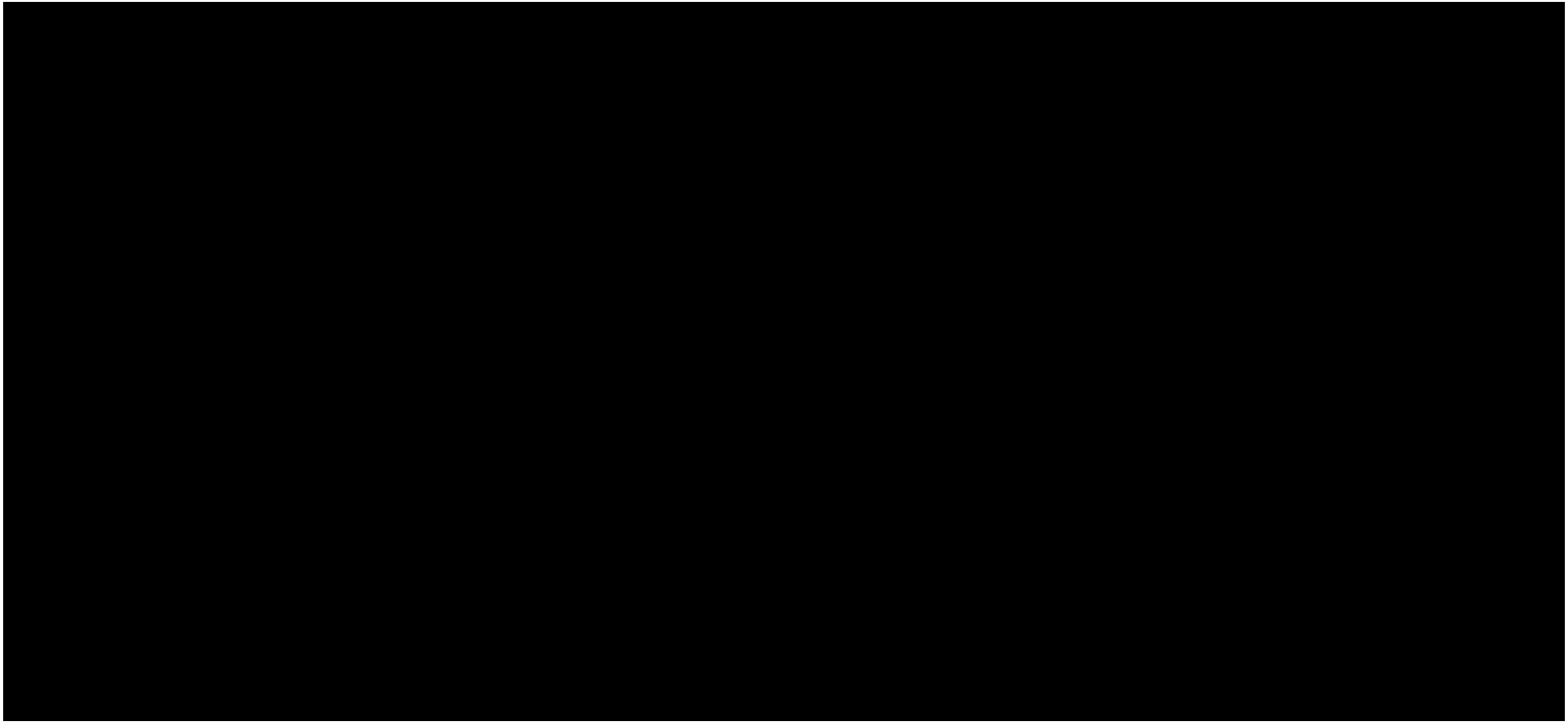
School drop-out rate	-3.9945
Number of rooms per house	2.6450
Distance to commute	-2.4197
School class size	-1.6939
Distance to highway	1.4261

Train Linear Regressor | Train Neural Network Regressor (1 hidden layer) | Train Neural Network Regressor (2 hidden layers)

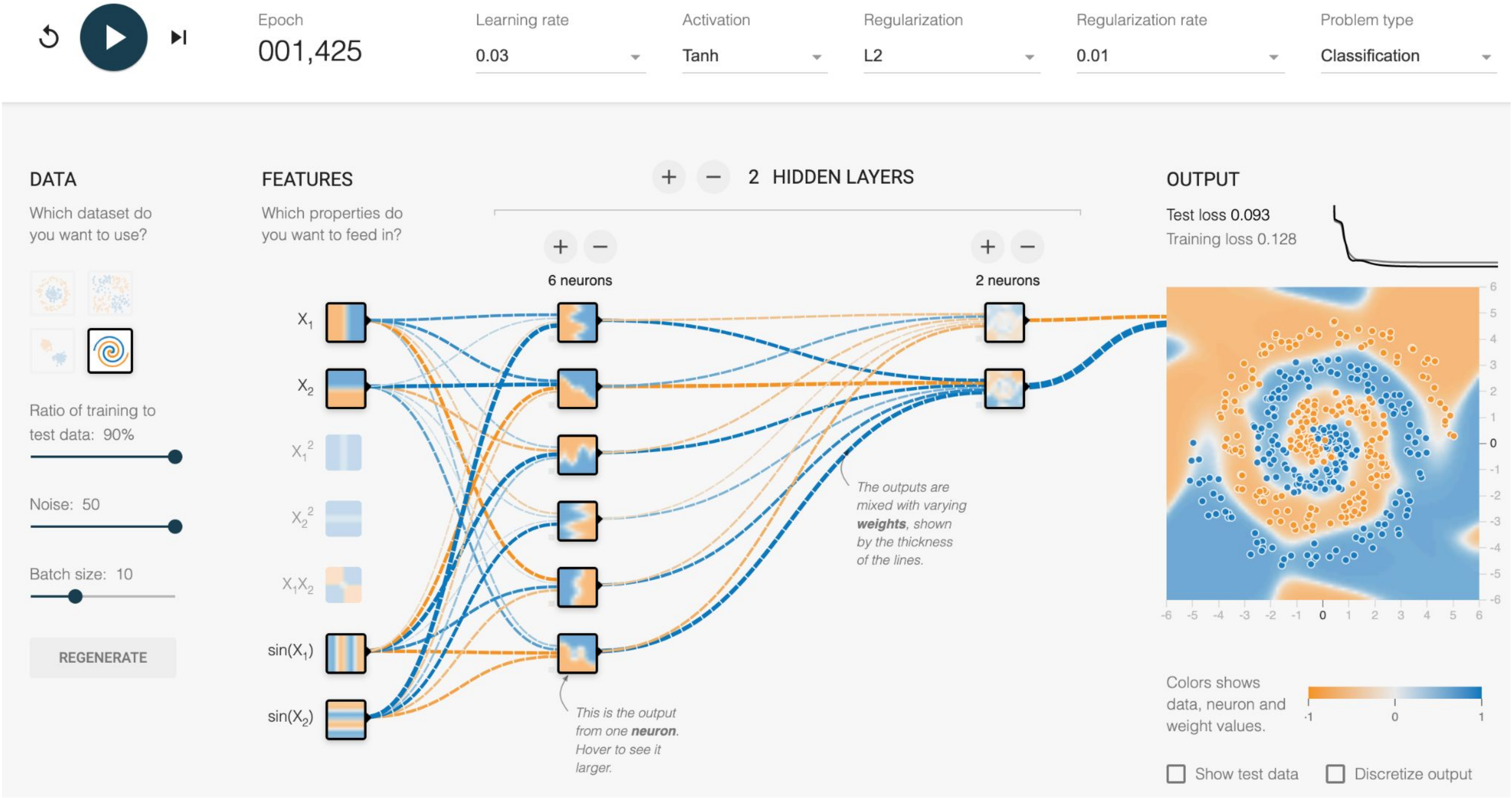
https://www.tensorflow.org/js/demos

Tensorflow Playground

Here: https://playground.tensorflow.org/ you will find a beautiful didactical playground to play with Neural Networks on different sets of typical classification/regression examples. You can choose the hidden units, layers, activation functions, features, whatever you want.



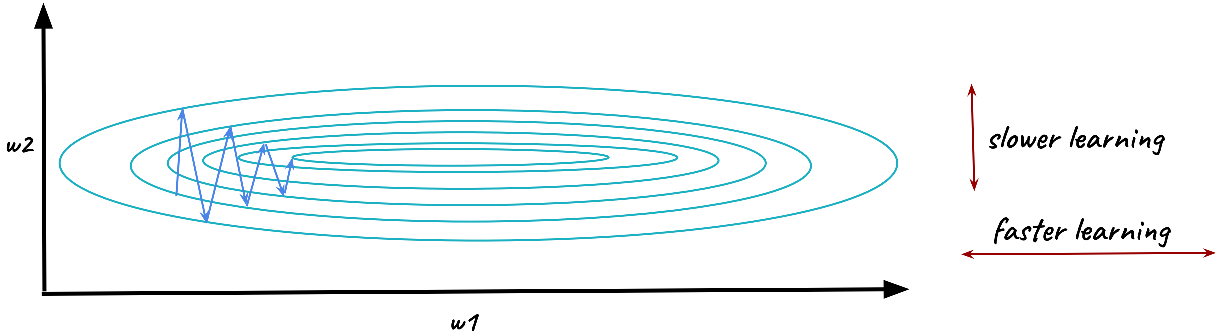
Here: https://playground.tensorflow.org/ you will find a beautiful didactical playground to play with Neural Networks on different sets of typical classification/regression examples. You can choose the hidden units, layers, activation functions, features, whatever you want.



Right now we only saw one possible optimization algorithm, that is the algorithm used to perform gradient descent and reach the minimum of the cost function *J*. It is the simplest possible form of gradient descent:

$$w := w - \alpha\, \mathrm{d}J(w)/\mathrm{d}w$$

which is fine, but as you can imagine there are other possible techniques to perform gradient descent *faster* and *safer* (meaning, avoiding local minima and vanishing/exploding gradients).



Right now we only saw one possible optimization algorithm, that is the algorithm used to perform gradient descent and reach the minimum of the cost function *J*. It is the simplest possible form of gradient descent:

$$w := w - \alpha\, \mathrm{d}J(w)/\mathrm{d}w$$

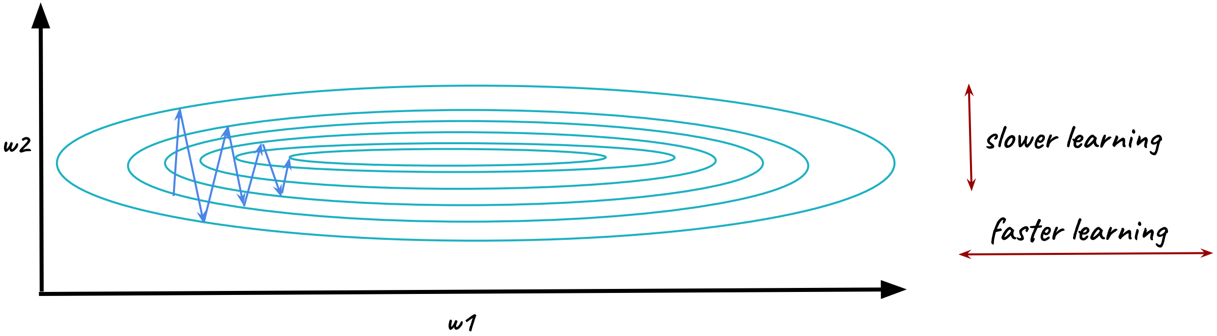
which is fine, but as you can imagine there are other possible techniques to perform gradient descent *faster* and *safer* (meaning, avoiding local minima and vanishing/exploding gradients).

Gradient descent with Momentum: evaluates an exponentially weighted average of the gradients, and use that gradient to update the weights
This smooths out the steps of gradient descent.
For each training iteration t:

1) compute dw on current batch/mini-batch
2) compute $V_{dw} = \beta V_{dw} + (1 - \beta)\, dw$
3) update the weights $w := w - \alpha\, V_{dw}$

the same for b

β controls the number of past gradient evaluations upon which evaluate the weighted average



https://distill.pub/2017/momentum/, for an overly detailed description of Momentum, with lots of visualizations

Right now we only saw one possible optimization algorithm, that is the algorithm used to perform gradient descent and reach the minimum of the cost function J. It is the simplest possible form of gradient descent:

$$w := w - \alpha\, \mathrm{d}J(w)/\mathrm{d}w$$

which is fine, but as you can imagine there are other possible techniques to perform gradient descent *faster* and *safer* (meaning, avoiding local minima and vanishing/exploding gradients).

Gradient descent with Momentum: evaluates an exponentially weighted average of the gradients, and use that gradient to update the weights
This smooths out the steps of gradient descent.
For each training iteration t:

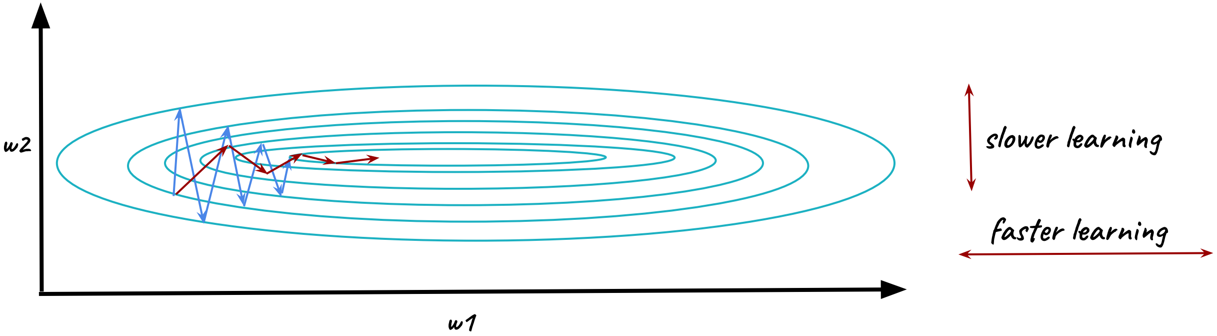
1) compute dw on current batch/mini-batch
2) compute $V_{dw} = \beta V_{dw} + (1 - \beta)\, dw$
3) update the weights $w := w - \alpha\, V_{dw}$

the same for b

β controls the number of past gradient evaluations upon which evaluate the weighted average

Two hyperparameters:

- α
- β (usually 0.9)



https://distill.pub/2017/momentum/, for an overly detailed description of Momentum, with lots of visualizations

Right now we only saw one possible optimization algorithm, that is the algorithm used to perform gradient descent and reach the minimum of the cost function *J*. It is the simplest possible form of gradient descent:

$$w := w - \alpha \, \mathrm{d}J(w)/\mathrm{d}w$$

which is fine, but as you can imagine there are other possible techniques to perform gradient descent *faster* and *safer* (meaning, avoiding local minima and vanishing/exploding gradients).

Root Mean Square prop. : same concept as momentum, you want to speed up learning for certain weights and slow down learning for other weights, but with some small tweaks to the algorithm

For each training iteration t:

1) compute dw on current batch/mini-batch
2) compute $S_{dw} = \beta S_{dw} + (1 - \beta)\, dw^2$
3) update the weights $w := w - \alpha\, dw * (1/\sqrt{S_{dw}})$

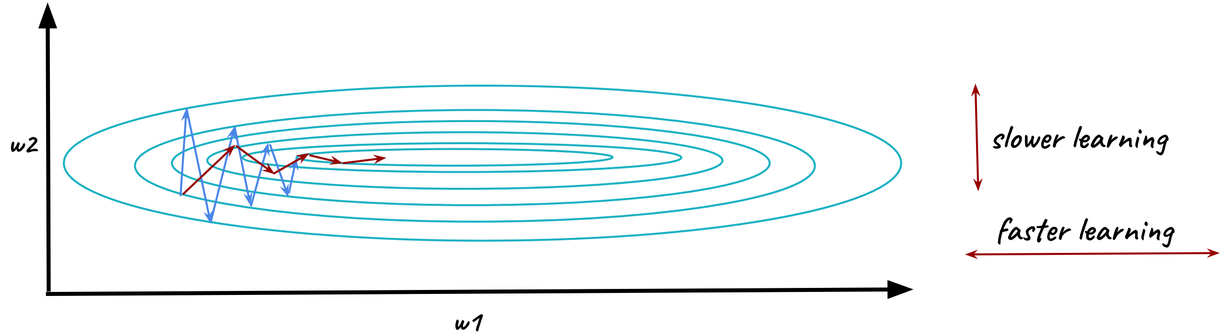
the same for b

To avoid dividing per zero, a small η (e.g. 10^{-8}) is added to the code

Two hyperparameters:

- α
- β (usually 0.9)

in this case you can even try a high value for α without risking overshooting



Right now we only saw one possible optimization algorithm, that is the algorithm used to perform gradient descent and reach the minimum of the cost function *J*. It is the simplest possible form of gradient descent:

$$w := w - \alpha\, \mathrm{d}J(w)/\mathrm{d}w$$

which is fine, but as you can imagine there are other possible techniques to perform gradient descent *faster* and *safer* (meaning, avoiding local minima and vanishing/exploding gradients).

Adam (which stands for ***Adaptive Moment estimator***) is one of those rare case of an algorithm that works well literally everywhere and for every application, and as such you will always see it used as ***the*** optimization algorithm of a NN. It puts together **Momentum** and **RMSProp**.
Adam starts by initializing $S_{dw} = 0$ and $V_{dw} = 0$
For each training iteration t:

1) compute dw on current mini-batch
2) compute:
$$V_{dw} = \beta_M V_{dw} + (1 - \beta_M)\, dw$$
$$S_{dw} = \beta_R S_{dw} + (1 - \beta_R)\, dw^2$$
3) compute bias correction:
$$V^{corr}_{dw} = V_{dw} / (1 - \beta^t_M)$$
$$S^{corr}_{dw} = S_{dw} / (1 - \beta^t_R)$$
4) update the weights $w := w - \alpha\, V_{dw} * (1/\sqrt{S_{dw}})$

α → needs to be tuned
β_M → 0.9
β_R → 0.999

as suggested by the Adam Authors

Right now we only saw one possible optimization algorithm, that is the algorithm used to perform gradient descent and reach the minimum of the cost function *J*. It is the simplest possible form of gradient descent:

$$w := w - \alpha\, \mathrm{d}J(w)/\mathrm{d}w$$

which is fine, but as you can imagine there are other possible techniques to perform gradient descent *faster* and *safer* (meaning, avoiding local minima and vanishing/exploding gradients).

Learning rate decay means slowing reduce α while training with time. It's not an optimization algorithm on its own, *learning rate decay* can be easily attached to *Momentum*, *RMSProp* or *Adam*.
There isn't a single way to do α-decay, various methods apply, e.g.:

$$\alpha = \frac{1}{1+\text{decay rate}+\text{epoch num}}\alpha_0$$

or a generic number < 1

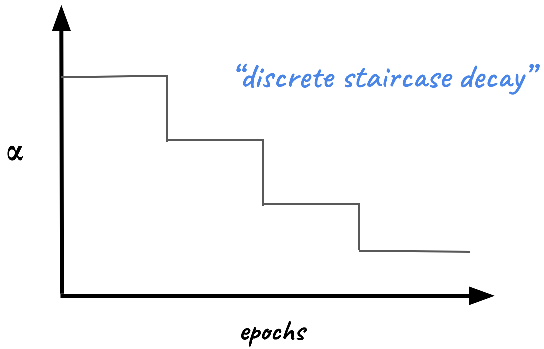
$$\alpha = 0.95^{\text{epoch num}}\alpha_0$$

"exponential decay"

$$\alpha = \frac{\text{const}}{\sqrt{\text{epoch num}}}\alpha_0$$

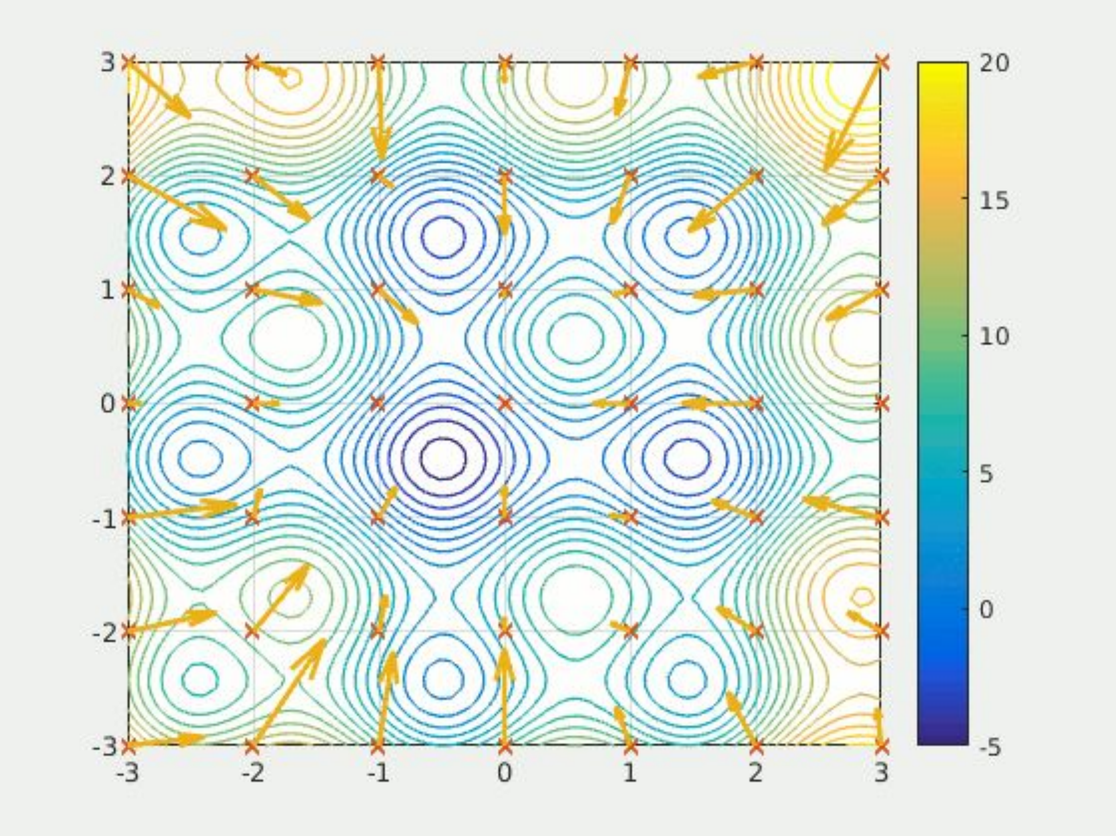
mini-batch number

$$\alpha = \frac{\text{const}}{\sqrt{t}}\alpha_0$$



Optimization algorithms - Particle Swarm Optimization

Particle Swarm Optimization is, for once, an idea taken from another field (*Animal Social Scie… Studies*) that actually applies on this one.



The idea of *PSO* is to emulate the social behaviour of birds and fishes, by initializing a *set* of candidate solutions to search for an optima.

So, it is not just a single particle searching for the minimum of the cost J, but a set (*swarm*) of particles.

The idea is similar to that of walkers in MonteCarlo sampling, with particles talking with each other and sharing their knowledges about the parameter space they're sampling.

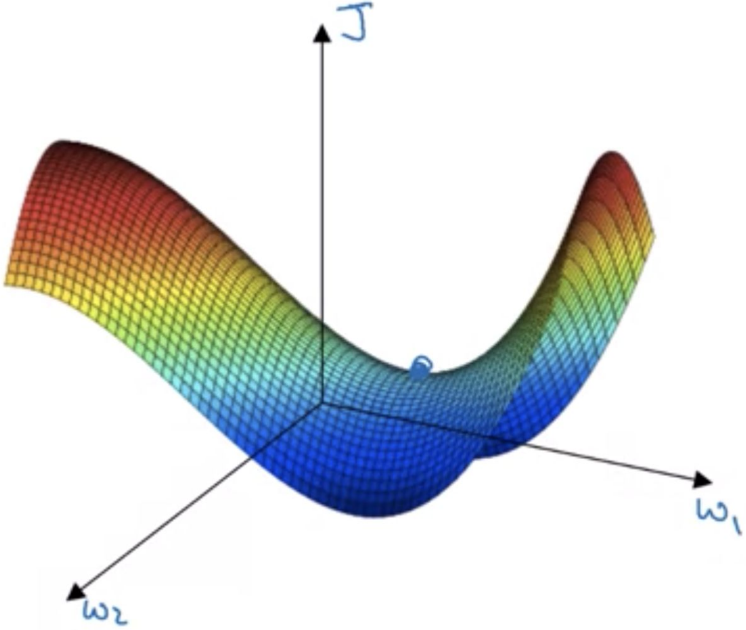
Particles are scattered around the search-space, and they move around it to find the position of the optima. Their movements are affected by:

(1) their cognitive desire to search individually

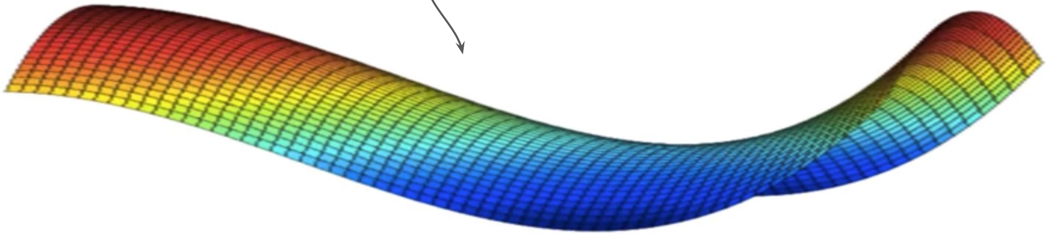
(2) the collective action of the group or its neighbors.

Kennedy and Eberhart, "Particle Swarm Optimization," Proceedings of the IEEE International Joint Conference on Neural Networks, 1995, pp. 1942-1948.

However, keep in mind that in Deep Learning, the thing you should be worried the most is not getting stuck in local minimum, which almost never happen*, but on *saddle points*.
Especially *extremely elongated saddle points*, that create a local plateau where your gradient descent might vanish.



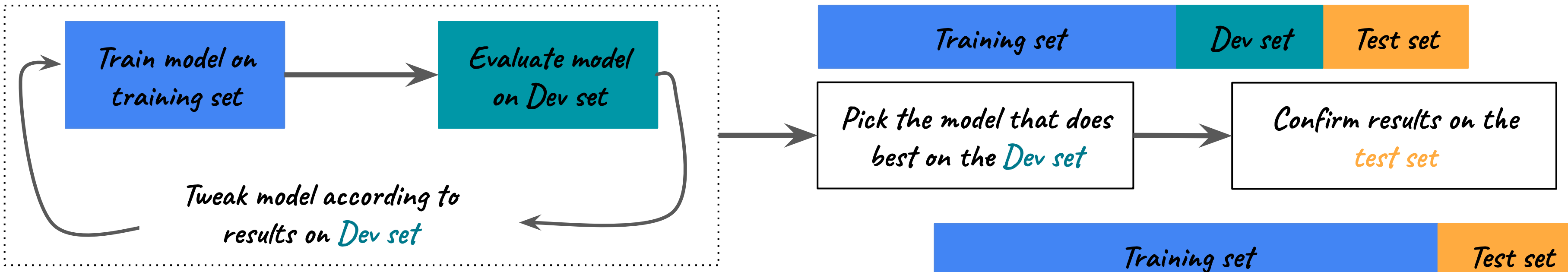
and that's why Adam is so effective and ubiquitous



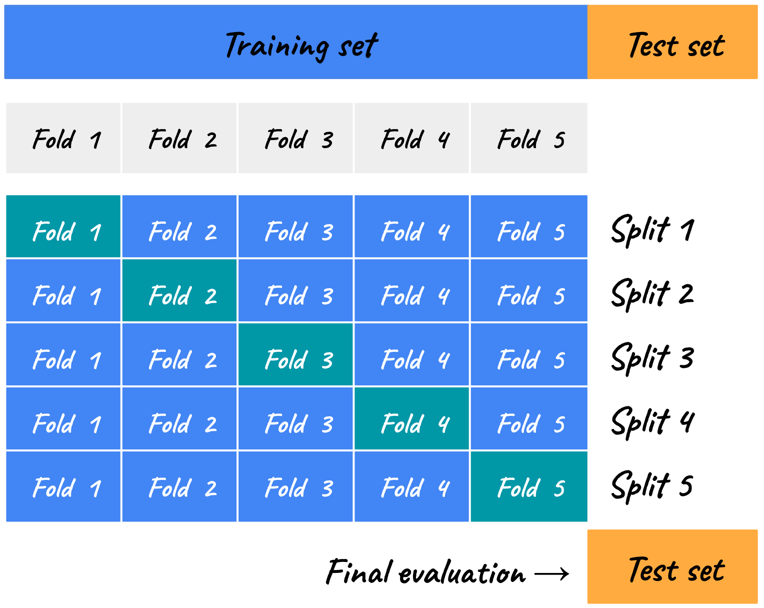
* there are tricks to smooth the Loss landscape, e.g. skip connections
or, use mini-batch gradient descent: for each mini-batch, the Loss landscape changes
what is a local minima for a mini-batch might not be a local minima for the next mini-batch

Hyperparameter Optimization

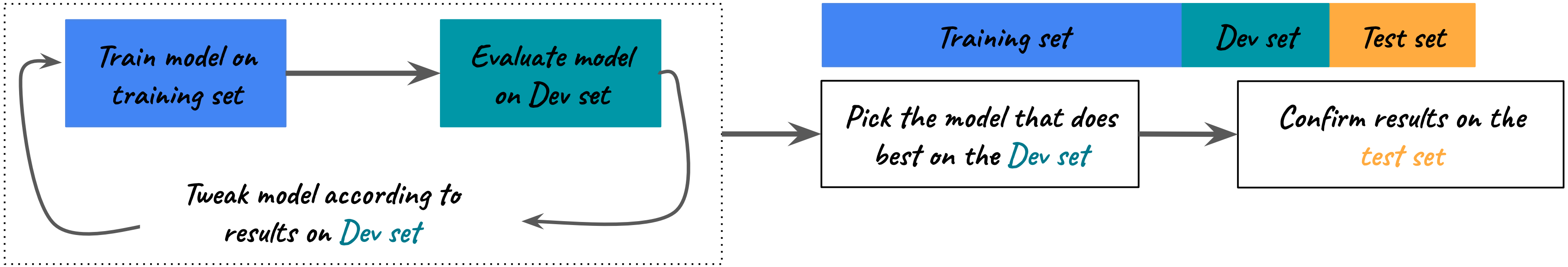
As we've already seen in the previous lectures, a typical ML algorithm has at least one major hyperparameter that needs optimal tuning to lead to the best possible performances. That's the reason why the typical workflow in ML splits the sets between *train*/*Dev*/*test*:



… and you can use, e.g., *nested k-folds cross-validation* technique to be as most general as possible →



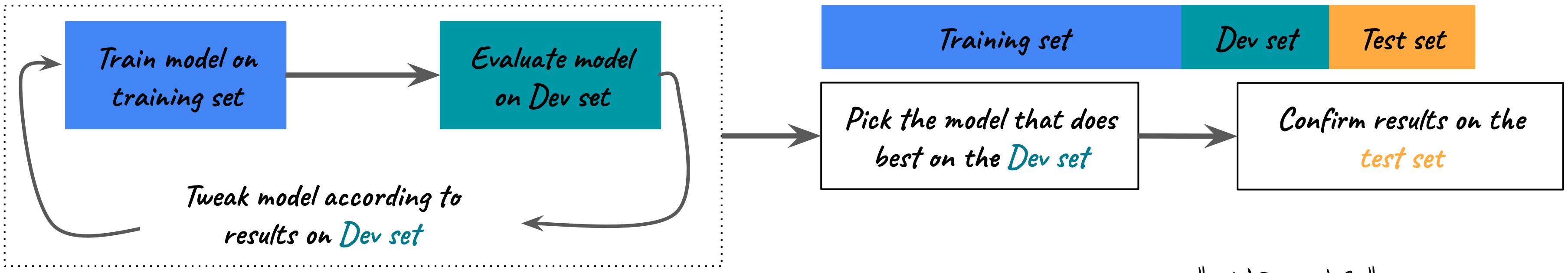
As we've already seen in the previous lectures, a typical ML algorithm has at least one major hyperparameter that needs optimal tuning to lead to the best possible performances. That's the reason why the typical workflow in ML splits the sets between *train*/*Dev*/*test*:



But this does not answer a question: when you have more than one important hyperparameter, how do you *actually* look for the best possible values?

scikit-learn has its own big module for model selection:
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

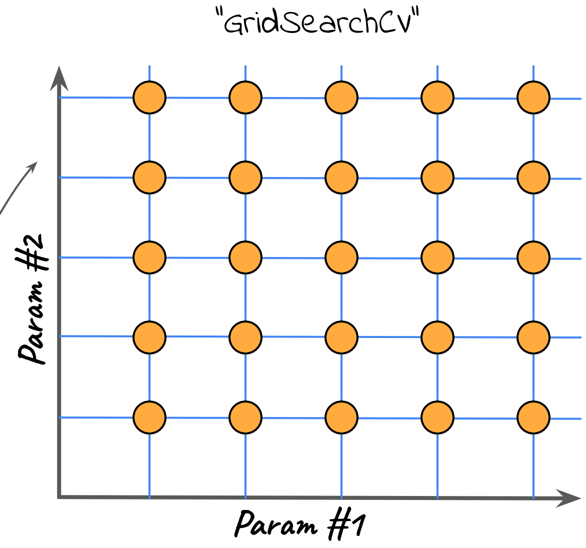
As we've already seen in the previous lectures, a typical ML algorithm has at least one major hyperparameter that needs optimal tuning to lead to the best possible performances. That's the reason why the typical workflow in ML splits the sets between *train*/*Dev*/*test*:



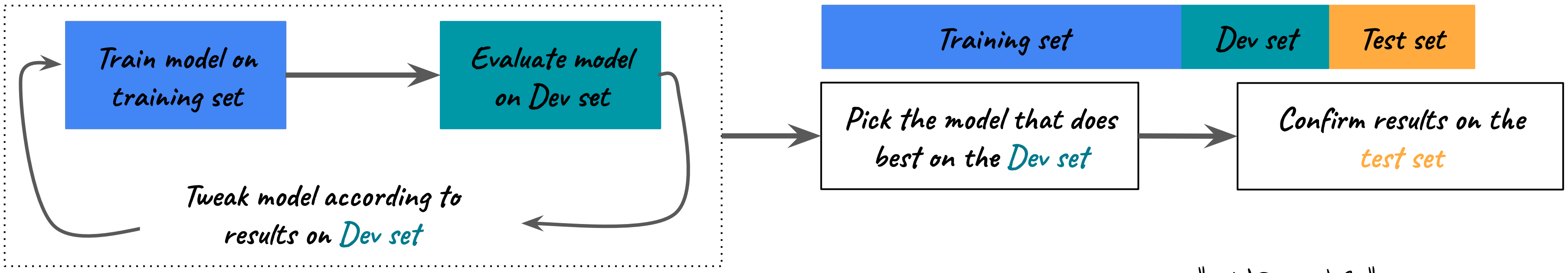
But this does not answer a question: when you have more than one important hyperparameter, how do you *actually* look for the best possible values?

scikit-learn has its own big module for model selection:
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

When the number of hyperparameters is small, the most commonly used is GridSearchCV, where you sample a regular grid of possible parameter values, looking for the ones with the best *Dev* loss.



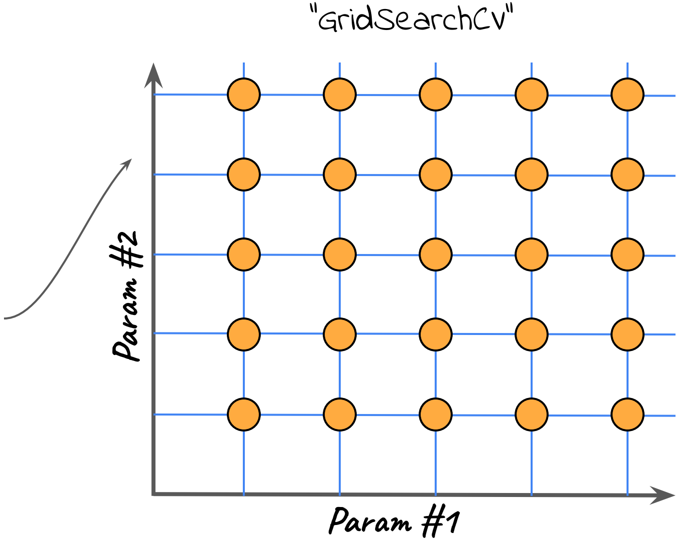
As we've already seen in the previous lectures, a typical ML algorithm has at least one major hyperparameter that needs optimal tuning to lead to the best possible performances. That's the reason why the typical workflow in ML splits the sets between *train*/*Dev*/*test*:



But this does not answer a question: when you have more than one important hyperparameter, how do you *actually* look for the best possible values?

scikit-learn has its own big module for model selection:
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

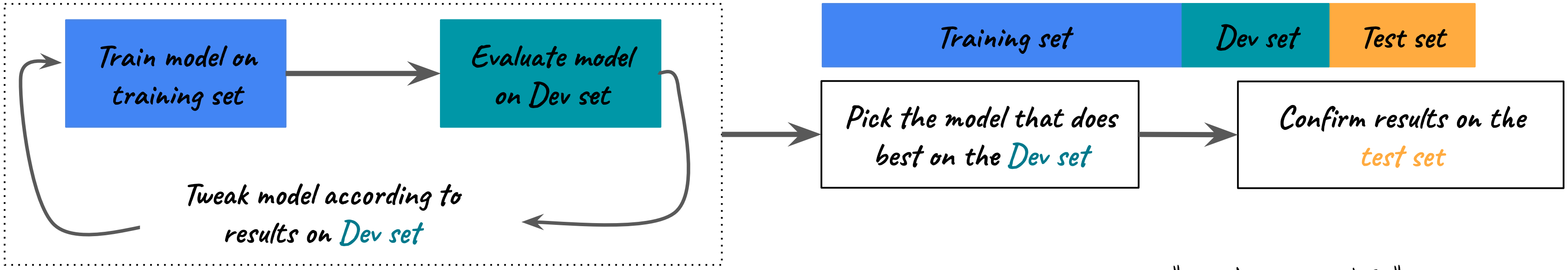
When the number of hyperparameters is small, the most commonly used is GridSearchCv, where you sample a regular grid of possible parameter values, looking for the ones with the best *Dev* loss.



If the number of hyperparameters is relatively small, GridSearchCv is computationally feasible and actually used.

Hyperparameter Optimization

As we've already seen in the previous lectures, a typical ML algorithm has at least one major hyperparameter that needs optimal tuning to lead to the best possible performances. That's the reason why the typical workflow in ML splits the sets between train/Dev/test:

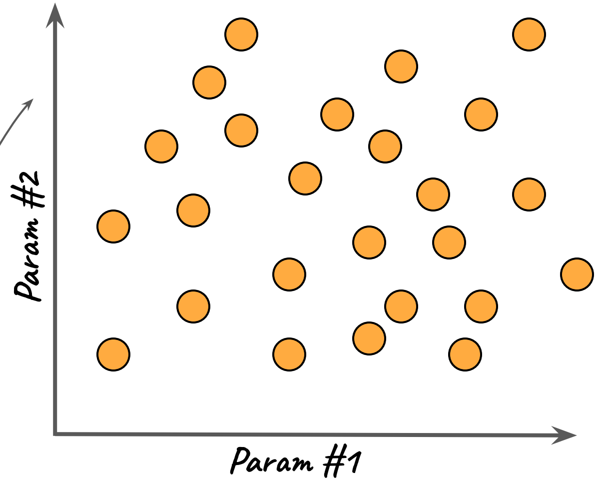


But this does not answer a question: when you have more than one important hyperparameter, how do you *actually* look for the best possible values?

scikit-learn has its own big module for model selection:
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

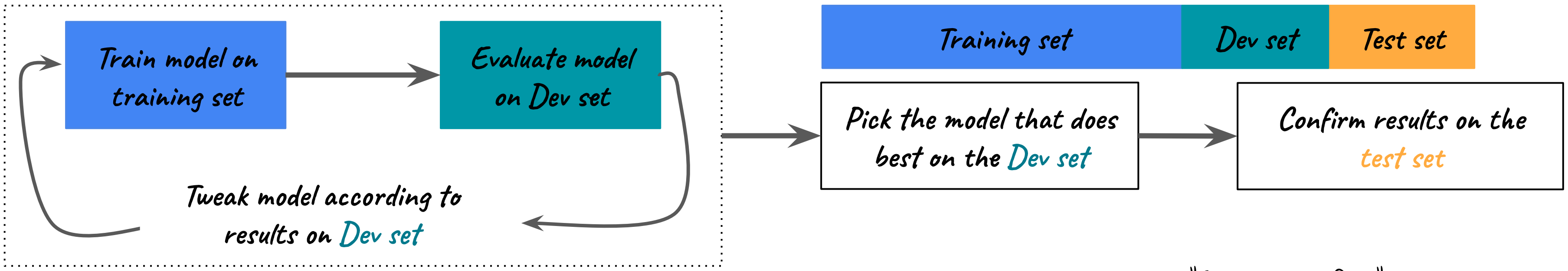
When the number of hyperparameters is relatively high, as in the case of Deep Learning (α, βs, #layers, #hidden units, α-decay, mini-batch size…), a grid search is unfeasible. So, it is better to use RandomSearchCV …

"RandomSearchCV"



Hyperparameter Optimization

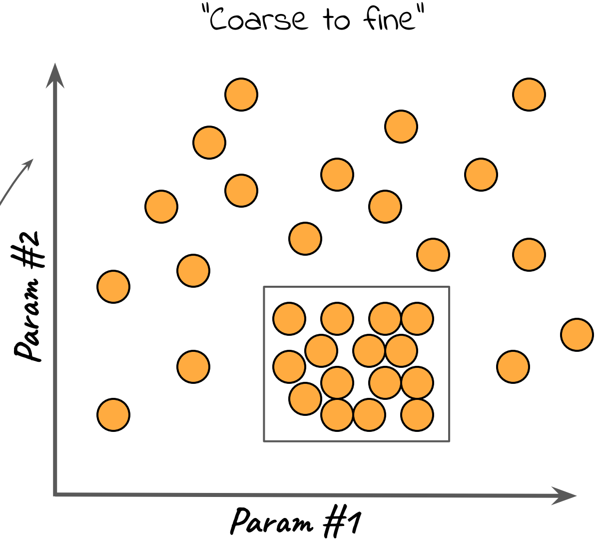
As we've already seen in the previous lectures, a typical ML algorithm has at least one major hyperparameter that needs optimal tuning to lead to the best possible performances. That's the reason why the typical workflow in ML splits the sets between *train*/*Dev*/*test*:



But this does not answer a question: when you have more than one important hyperparameter, how do you *actually* look for the best possible values?

scikit-learn has its own big module for model selection:
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

When the number of hyperparameters is relatively high, as in the case of Deep Learning (α, βs, #layers, #hidden units, α-decay, mini-batch size…), a grid search is unfeasible. So, it is better to use `RandomSearchCV`, or better, the *"coarse-to-fine"* sampling scheme: first sample at random, then sample more densely in a subregion of the hyperparameter space where you expect to find the best values.



Batch normalization makes the hyperaparameters search easier, and the whole network more robust.
It is an extension of feature scaling, applied to the $a^{[l]}$ values (well, actually to the z's that enter activation functions) in any hidden layer, so as to train the weights $W^{[l]}$, $b^{[l]}$ faster.

Ioffe and Szegedy, 2014, https://arxiv.org/abs/1502.03167

Batch normalization makes the hyperaparameters search easier, and the whole network more robust.
It is an extension of feature scaling, applied to the $a^{[l]}$ values (well, actually to the z's that enter activation functions) in any hidden layer, so as to train the weights $W^{[l]}$, $b^{[l]}$ faster.

So, for a generic intermediate layer, *batch normalization* would compute the Z-score with values $z^{(1)} \dots z^{(n)} \rightarrow Z^{(i)}_{norm}$, but then instead of feeding the hidden units directly $Z^{(i)}_{norm}$, it gives them a slightly changed version:

$$\hat{Z}^{(i)} = \gamma Z^{(i)}_{\text{norm}} + \beta$$

unrelated to Adam's β parameter

each activation $a^{[l]}$ has its own (β, γ) couple

with β and γ learnable parameters of the model, allowing to give the normalized values whatever the mean or range you want it to be (e.g. for sigmoid activation functions).

Ioffe and Szegedy, 2014, https://arxiv.org/abs/1502.03167

Batch normalization makes the hyperaparameters search easier, and the whole network more robust.
It is an extension of feature scaling, applied to the $a^{[l]}$ values (well, actually to the z's that enter activation functions) in any hidden layer, so as to train the weights $W^{[l]}$, $b^{[l]}$ faster.

So, for a generic intermediate layer, *batch normalization* would compute the Z-score with values $z^{(1)} \dots z^{(n)} \rightarrow Z^{(i)}_{norm}$, but then instead of feeding the hidden units directly $Z^{(i)}_{norm}$, it gives them a slightly changed version:

$$\hat{Z}^{(i)} = \gamma Z^{(i)}_{\text{norm}} + \beta$$

unrelated to Adam's β parameter

each activation $a^{[l]}$ has its own (β, γ) couple

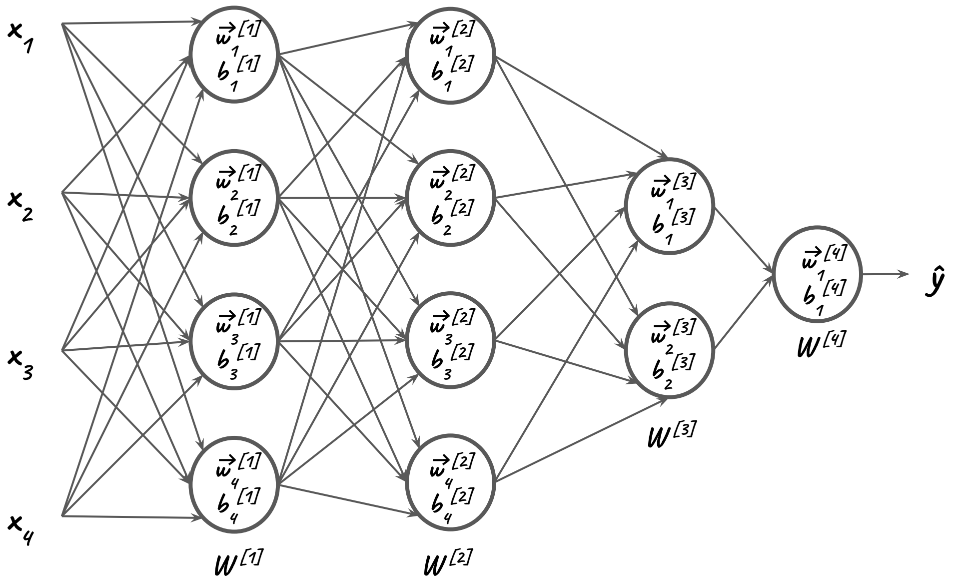
with β and γ learnable parameters of the model, allowing to give the normalized values whatever the mean or range you want it to be (e.g. for sigmoid activation functions).

Batch normalization makes the weights more robust, acting in a sense like a regularization scheme, with three great advantages:

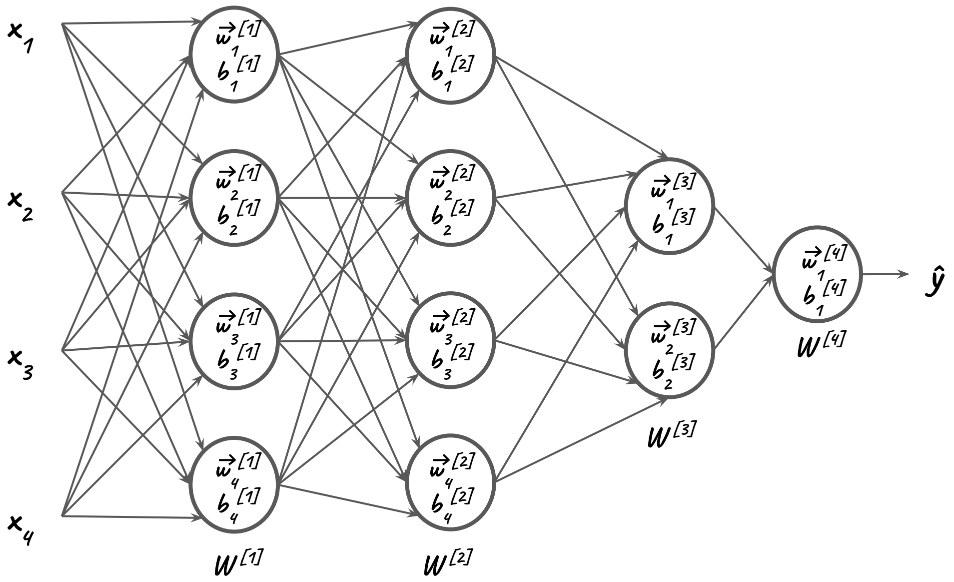
- *faster training*: although each iteration will be slower because of the extra normalization calculation during forth- and back-propagation, it should converge much more quickly
- allows to set a *higher learning rate* α, thereby increasing the speed of training.
- optimal *weight initialization*: batch normalization reduces the sensitivity to the initial starting weights.

Ioffe and Szegedy, 2014, https://arxiv.org/abs/1502.03167

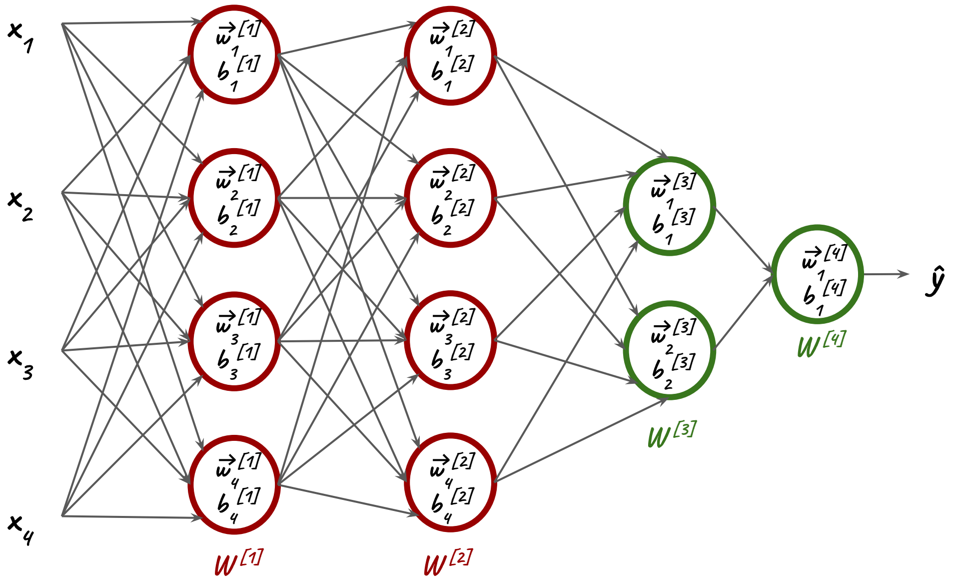
The power of Neural Networks does not only reside in their being low-bias machines that naturally adapts to GPUs with (lots) of libraries and softwares that takes the most out of the technique, but also in subtleties that makes them extremely versatile and able to adapt even to cases when you do not have that much training data in your hands. *Transfer Learning* is one of those.



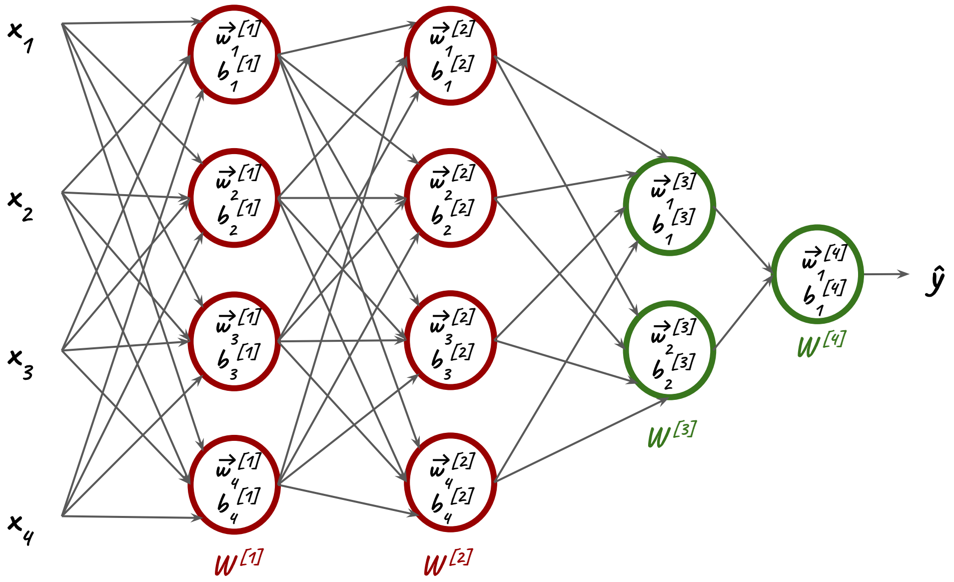
The power of Neural Networks does not only reside in their being low-bias machines that naturally adapts to GPUs with (lots) of libraries and softwares that takes the most out of the technique, but also in subtleties that makes them extremely versatile and able to adapt even to cases when you do not have that much training data in your hands. Transfer Learning is one of those.
The idea is: transfer the knowledge obtained by the training other people made on another Neural Network to your case, where *the knowledge*, in this case, are the network *weights*.
The application is even simpler: take the other model, freeze the *weights* of the first layers (*"pre-training"*) and train the *weights* of the final layers (or even just the output layer) on your training set.



The power of Neural Networks does not only reside in their being low-bias machines that naturally adapts to GPUs with (lots) of libraries and softwares that takes the most out of the technique, but also in subtleties that makes them extremely versatile and able to adapt even to cases when you do not have that much training data in your hands. *Transfer Learning* is one of those.
The idea is: transfer the knowledge obtained by the training other people made on another Neural Network to your case, where *the knowledge*, in this case, are the network *weights*.
The application is even simpler: take the other model, freeze the ***weights*** of the first layers (*"pre-training"*) and train the ***weights*** of the final layers (or even just the output layer) on your training set.

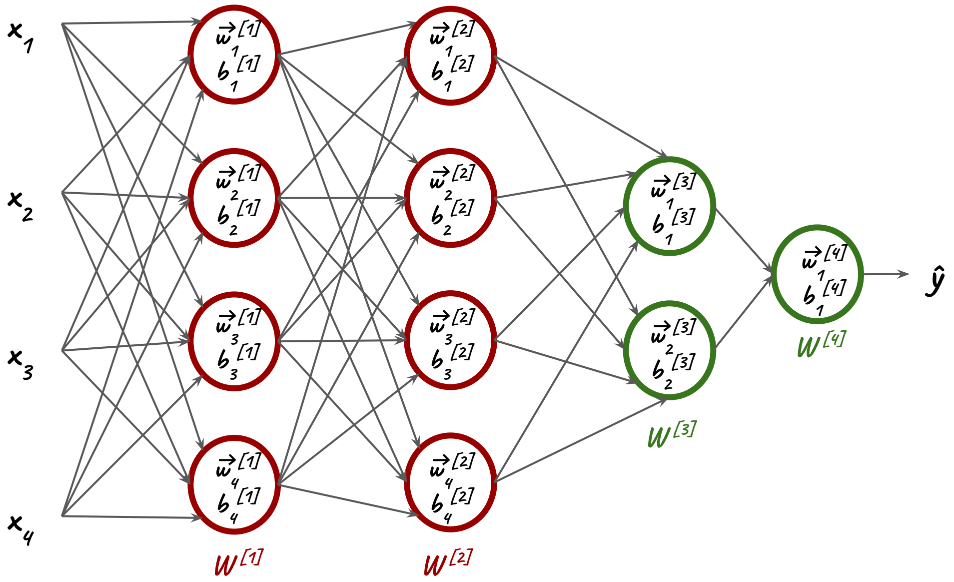


The power of Neural Networks does not only reside in their being low-bias machines that naturally adapts to GPUs with (lots) of libraries and softwares that takes the most out of the technique, but also in subtleties that makes them extremely versatile and able to adapt even to cases when you do not have that much training data in your hands. *Transfer Learning* is one of those.
The idea is: transfer the knowledge obtained by the training other people made on another Neural Network to your case, where *the knowledge*, in this case, are the network *weights*.
The application is even simpler: take the other model, freeze the ***weights*** of the first layers (*"pre-training"*) and train the ***weights*** of the final layers (or even just the output layer) on your training set.



There are multiple reasons to use *transfer learning*.
Let's suppose you want to train a *Convolutional Neural Network* to recognize your face in pictures. There's no need to train from scratch a *CNN* finding billions of pictures of human beings just to recognize your face; instead, you could take the weights of the first layers of a typical *CNN* doing face-recognition, and train only the final *softmax* layer to recognize your face out of all the possible people, with just a reduced training set of your pictures.

The power of Neural Networks does not only reside in their being low-bias machines that naturally adapts to GPUs with (lots) of libraries and softwares that takes the most out of the technique, but also in subtleties that makes them extremely versatile and able to adapt even to cases when you do not have that much training data in your hands. *Transfer Learning* is one of those.
The idea is: transfer the knowledge obtained by the training other people made on another Neural Network to your case, where *the knowledge*, in this case, are the network *weights*.
The application is even simpler: take the other model, freeze the *weights* of the first layers (*"pre-training"*) and train the *weights* of the final layers (or even just the output layer) on your training set.

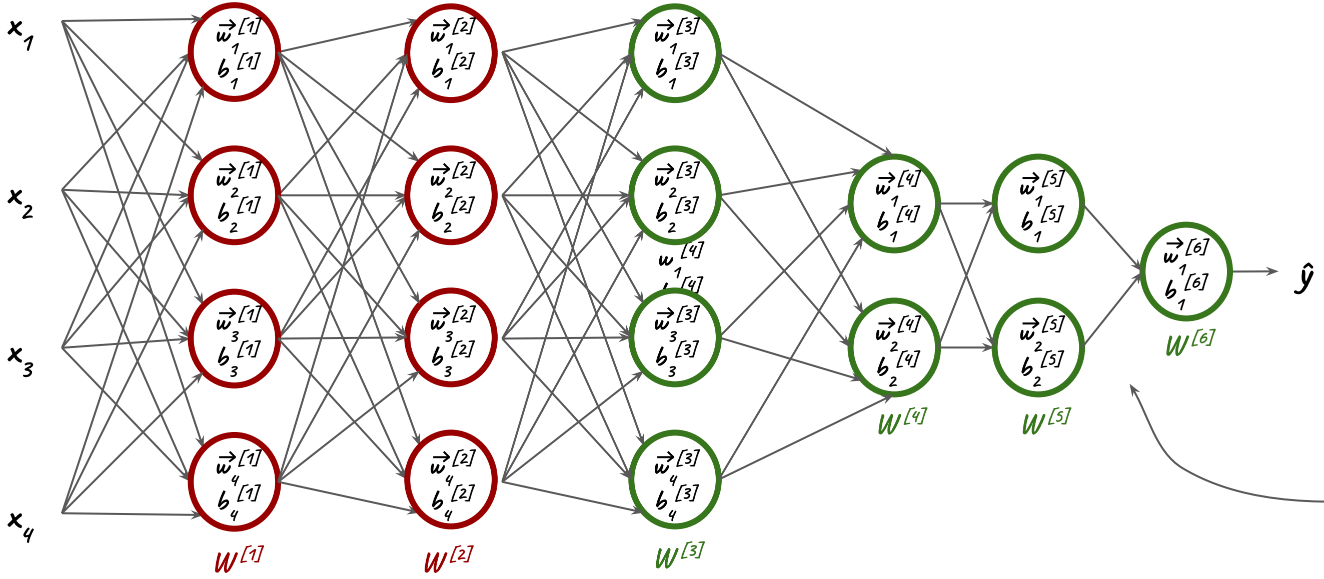


There are multiple reasons to use *transfer learning*.
Let's suppose you want to train a *Convolutional Neural Network* to recognize your face in pictures. There's no need to train from scratch a *CNN* finding billions of pictures of human beings just to recognize your face; instead, you could take the weights of the first layers of a typical *CNN* doing face-recognition, and train only the final *softmax* layer to recognize your face out of all the possible people, with just a reduced training set of your pictures.

Translate this into Astrophysics: you do not have to look for billions of pictures of galaxies to do morphological classification on your particular, specific case: there are already lots public models capable of at least recognize a galaxy and its feature out of noise, so take those models, and train the final layers only on your dataset.

Transfer Learning

The power of Neural Networks does not only reside in their being low-bias machines that naturally adapts to GPUs with (lots) of libraries and softwares that takes the most out of the technique, but also in subtleties that makes them extremely versatile and able to adapt even to cases when you do not have that much training data in your hands. *Transfer Learning* is one of those.
The idea is: transfer the knowledge obtained by the training other people made on another Neural Network to your case, where *the knowledge*, in this case, are the network *weights*.
The application is even simpler: take the other model, freeze the ***weights*** of the first layers (*"pre-training"*) and train the ***weights*** of the final layers (or even just the output layer) on your training set.



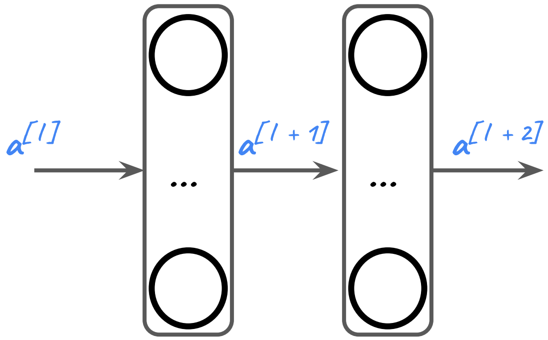
Transfer learning makes sense when:
- the other guys' network and yours have the same input
- there is a lot more available data for training their network than yours
- low-level features from the other guys model are actually helpful for learning your model

you can also change the NN architecture based on your particular application

Skip connections and Residual Blocks

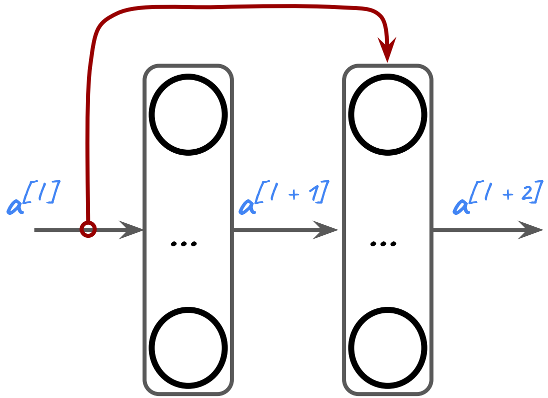
For years, very deep NN have been really hard to train because of vanishing/exploding gradients.

For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with ***skip connections*** and ***residual blocks***, that almost single-handedly solved the problem.



He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

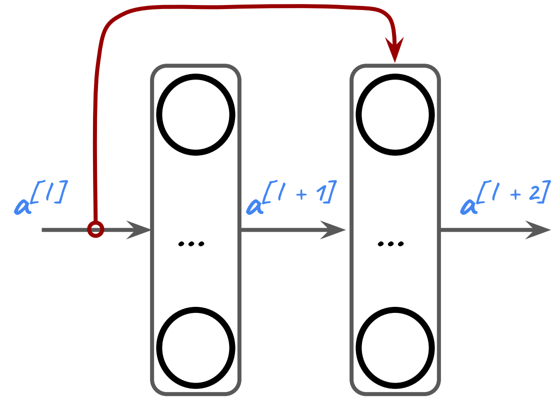
For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with *skip connections* and *residual blocks*, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer, as in the example below:



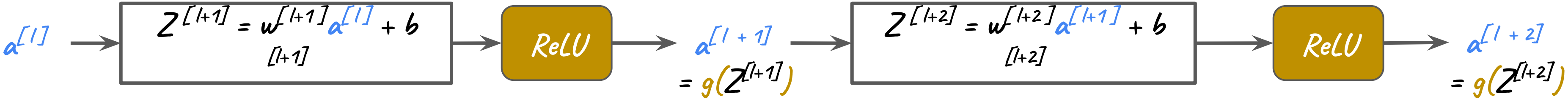
He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

Skip connections and Residual Blocks

For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with ***skip connections*** and ***residual blocks***, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer, as in the example below:

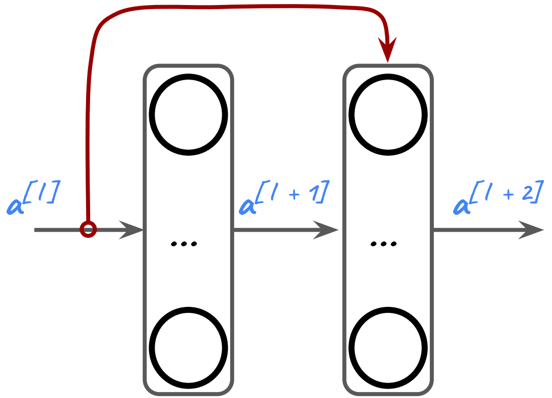


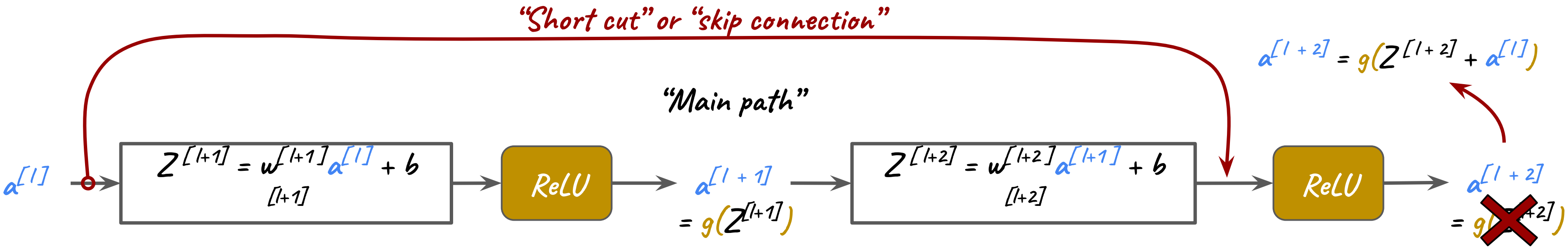
"Main path"



He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with ***skip connections*** and ***residual blocks***, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer, as in the example below:

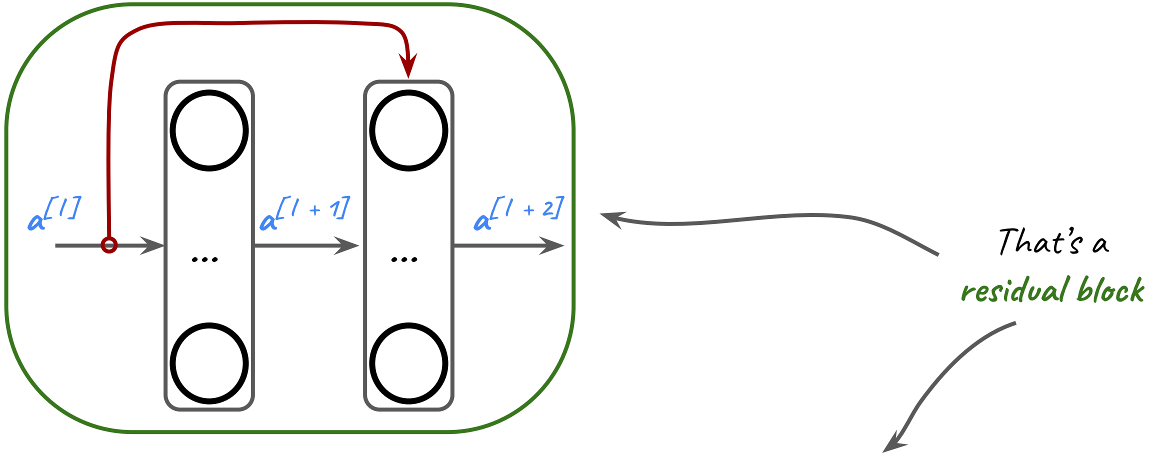


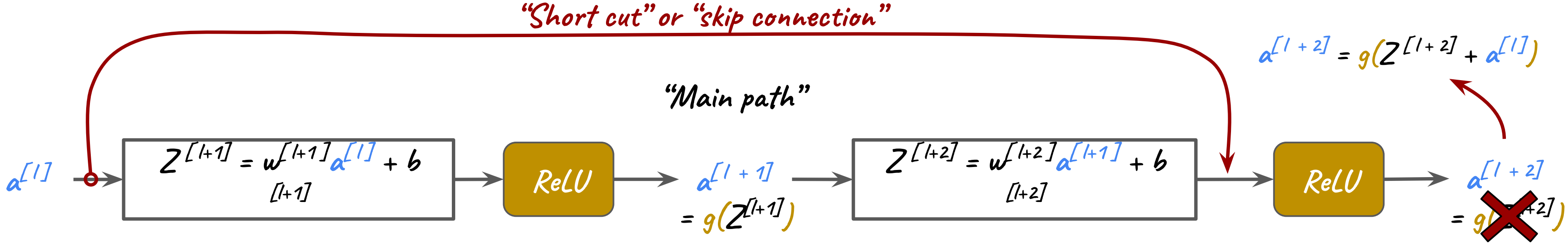


He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

Skip connections and Residual Blocks

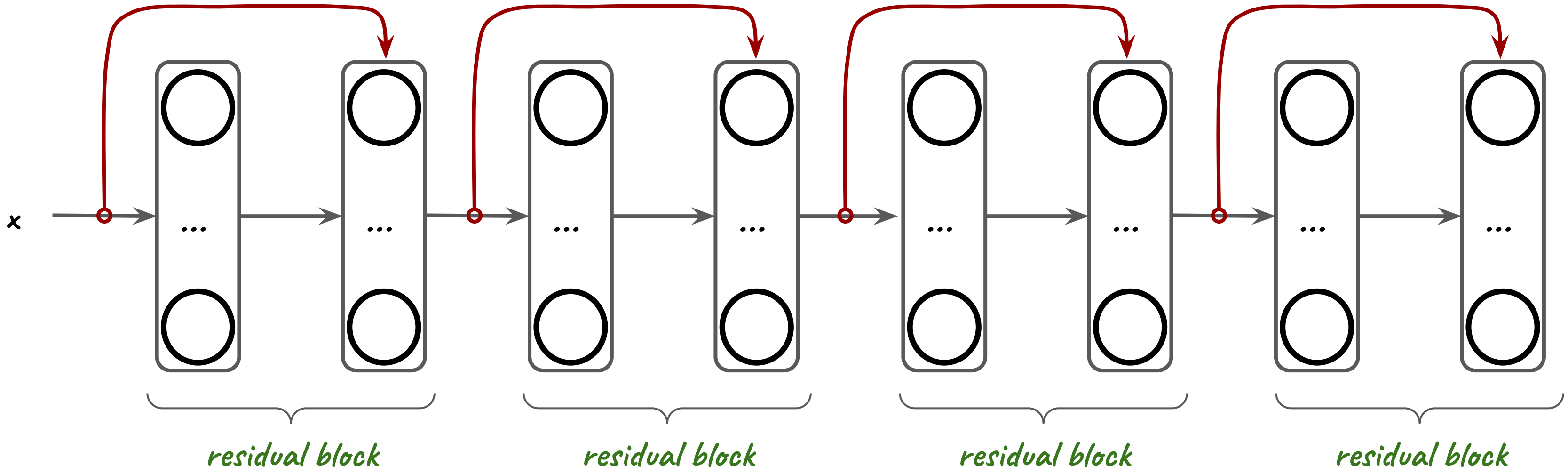
For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with ***skip connections*** and ***residual blocks***, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer, as in the example below:





He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

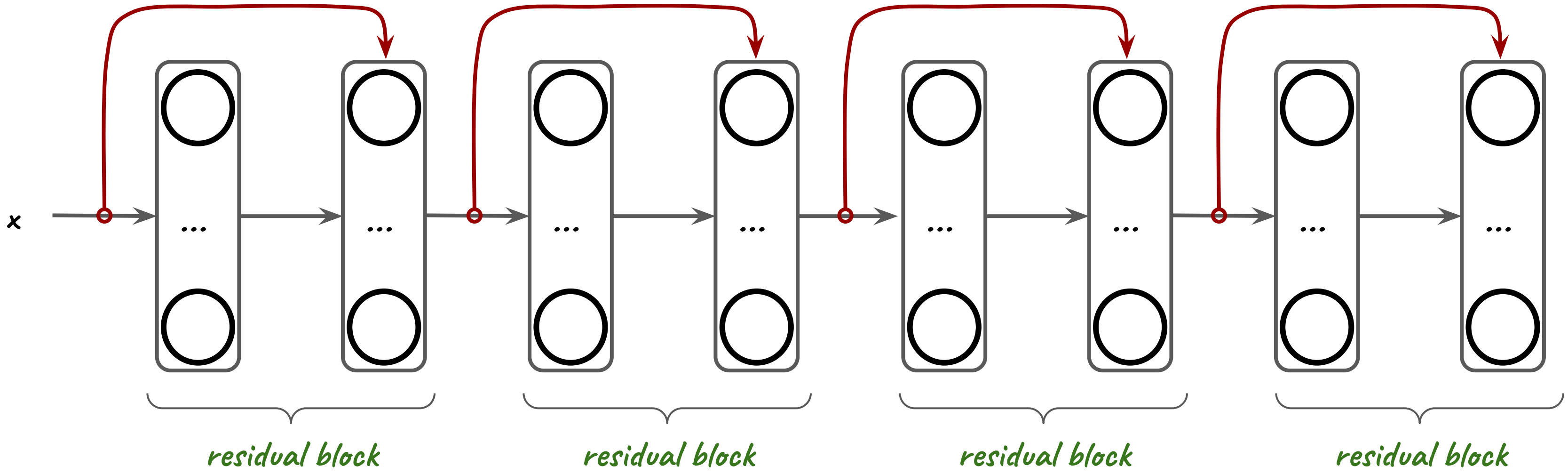
For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with *skip connections* and *residual blocks*, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer. How to turn a *plain network* into a *ResNet*?



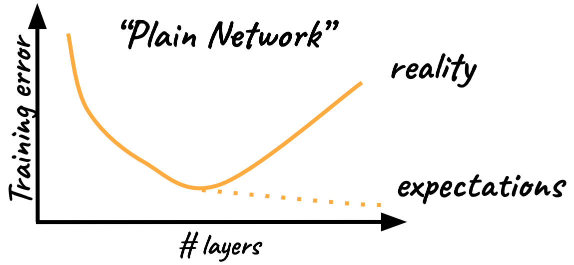
He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

Skip connections and Residual Blocks

For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with ***skip connections*** and ***residual blocks***, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer. How to turn a *plain network* into a *ResNet*?

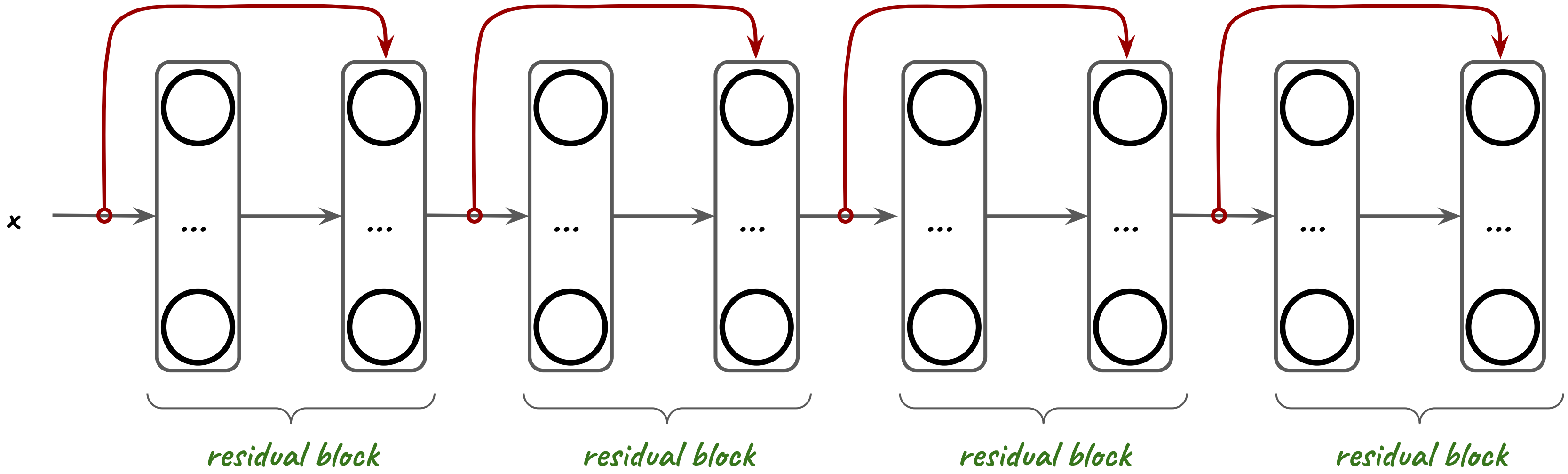


Using ***residual blocks*** allows to train much deeper NN (even *hundreds* of layers) →

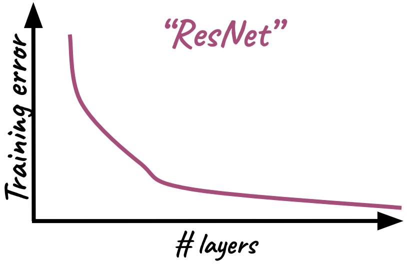


He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

For years, very deep NN have been really hard to train because of vanishing/exploding gradients.
Then, in 2015, someone came out with *skip connections* and *residual blocks*, that almost single-handedly solved the problem.
Skip connections take activations from a layer, and inputs it in a deeper layer. How to turn a *plain network* into a *ResNet*?

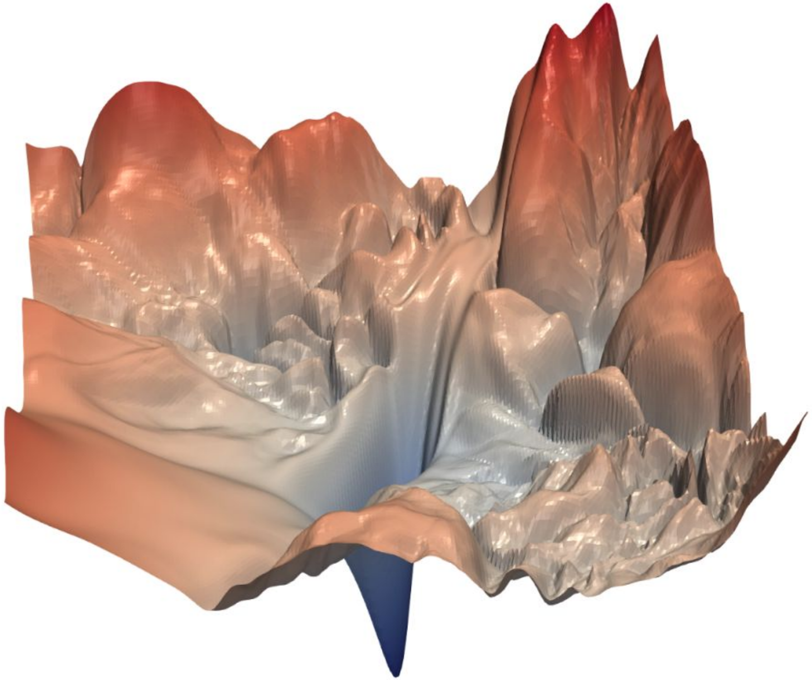


Using *residual blocks* allows to train much deeper NN (even *hundreds* of layers) →

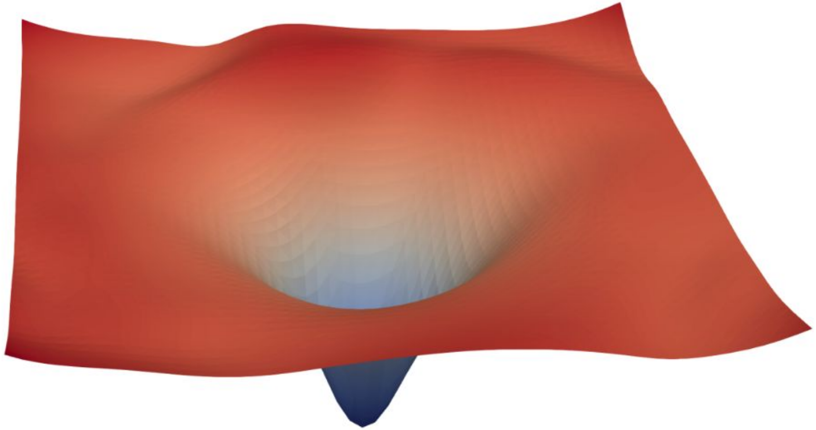


He et al., 2015, "Deep Residual Networks for Image Recognition", https://arxiv.org/abs/1512.03385.
This paper has almost **170K** citations, based on Google Scholar. ADS is more conservative, "only" **3.5k** citations. Lame.

ResNet work because they smooth the parameter space. They really, *really*, ***really***, ***<u>really</u>*** smooth it.



(a) without skip connections

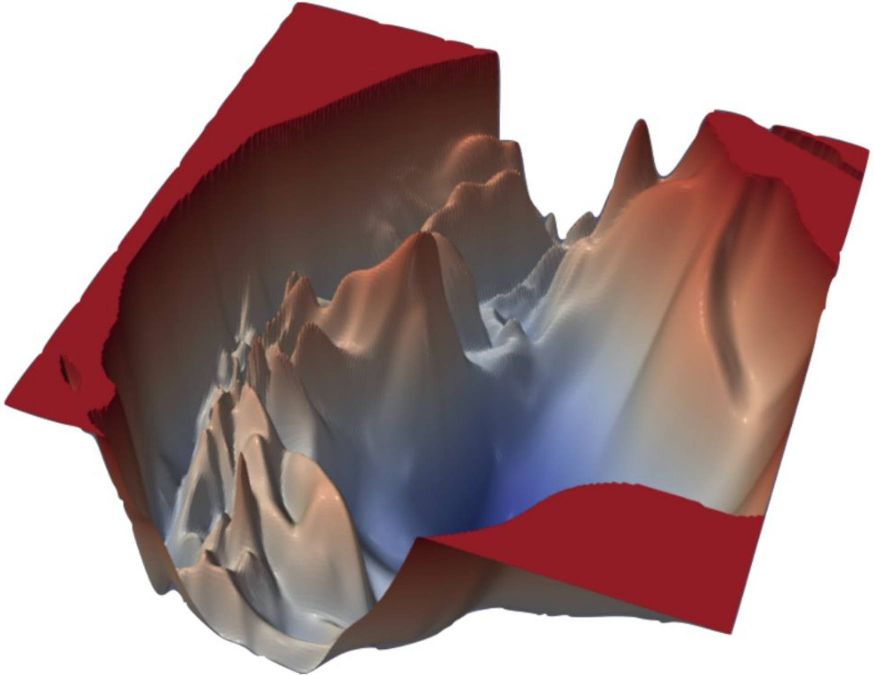


(b) with skip connections

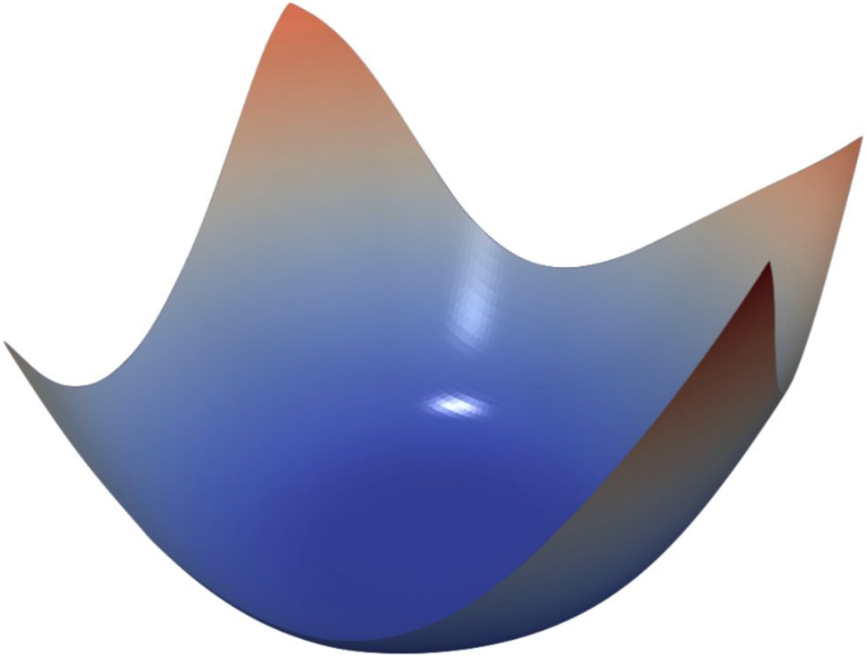
ResNet-56, with or without skip connections

Li et al., 2017, "[Visualizing the Loss Landscape of Neural Nets]"

ResNet work because they smooth the parameter space. They really, *really*, ***really***, ***really*** smooth it.



(a) ResNet-110, no skip connections

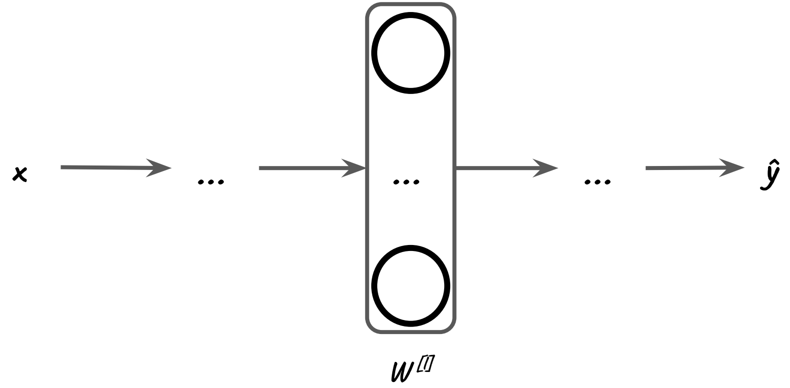


(b) DenseNet, 121 layers

Li et al., 2017, "Visualizing the Loss Landscape of Neural Nets"

Those are standard, widely used, *deterministic* Neural Networks: trained with a training set, cross-validated with a dev set, finally evaluated with a test set, then deployed for production.

Those are standard, widely used, *deterministic* Neural Networks: trained with a training set, cross-validated with a dev set, finally evaluated with a test set, then deployed for production.
For each new example you'll get a single *point estimate* prediction, with no information about the uncertainty of the model, nor the prediction
→ enters Bayesian Neural Networks : instead of learning a single, specific set of weights, try to learn the whole weight distributions, encoding the model uncertainty, from which we can sample to produce (still, *a single*) output for a given input.



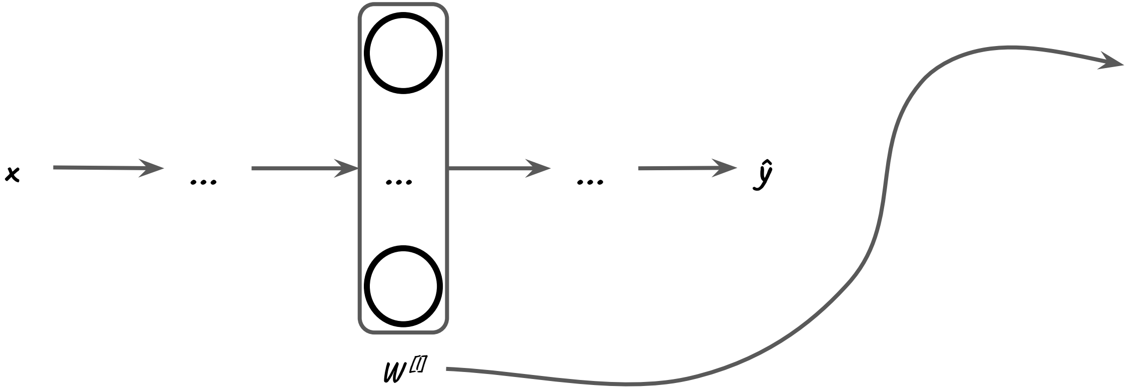
https://www.tensorflow.org/probability https://keras.io/examples/keras_recipes/bayesian_neural_networks/

Bayesian Neural Networks

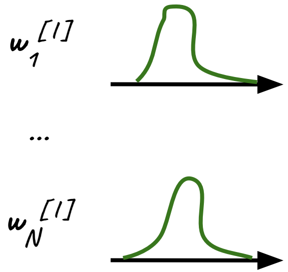
Those are standard, widely used, *deterministic* Neural Networks: trained with a *training set*, cross-validated with a *dev set*, finally evaluated with a *test set*, then deployed for production.

For each new example you'll get a single *point estimate* prediction, with no information about the uncertainty of the model, nor the prediction

→ enters *Bayesian Neural Networks* : instead of learning a single, specific set of *weights*, try to learn the whole *weight distributions*, encoding the model uncertainty, from which we can sample to produce (still, *a single*) output for a given input.



The weights are not single, fixed model parameters, but are sampled from a distribution $p(w_i^{[l]}|x)$

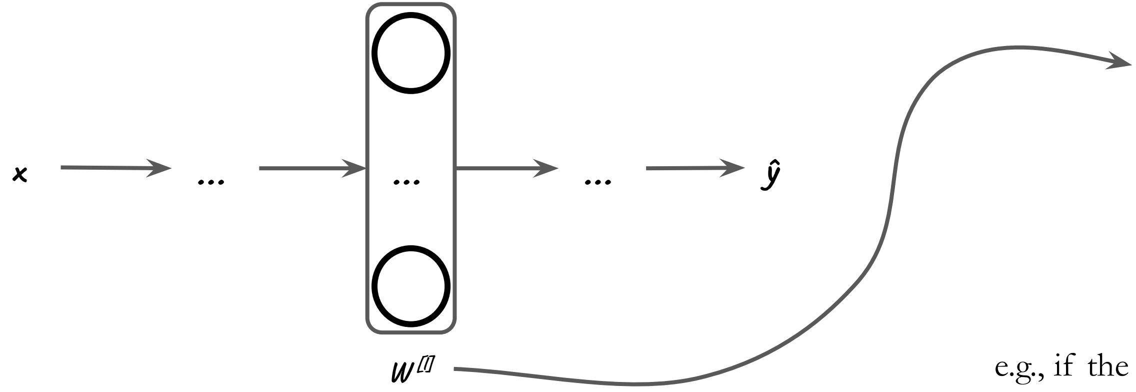


https://www.tensorflow.org/probability

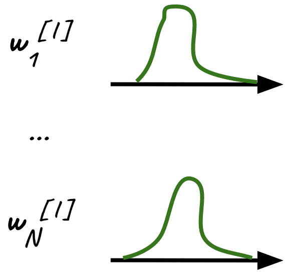
https://keras.io/examples/keras_recipes/bayesian_neural_networks/

Bayesian Neural Networks

Those are standard, widely used, *deterministic* Neural Networks: trained with a training set, cross-validated with a dev set, finally evaluated with a test set, then deployed for production.
For each new example you'll get a single *point estimate* prediction, with no information about the uncertainty of the model, nor the prediction → enters Bayesian Neural Networks : instead of learning a single, specific set of weights, try to learn the whole weight distributions, encoding the model uncertainty, from which we can sample to produce (still, *a single*) output for a given input.



The weights are not single, fixed model parameters, but are sampled from a distribution $p(w_i^{[l]}/x)$

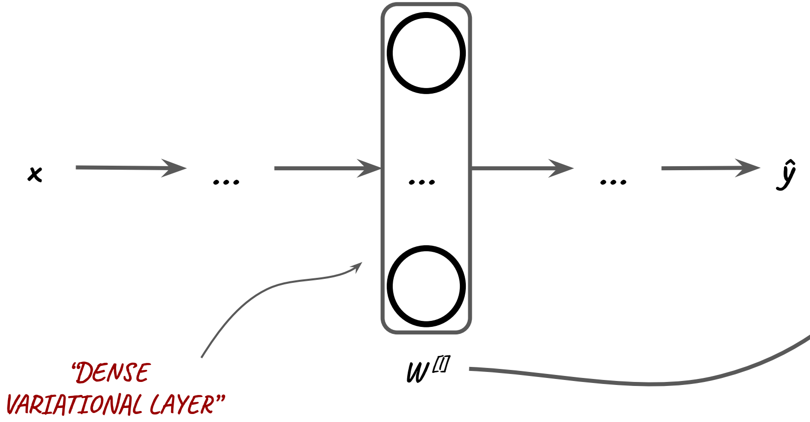


e.g., if the weights are modeled with *Normal* distributions, a Bayesian Neural Networks just need to find a mean **μ** and a variance **σ** to describe them

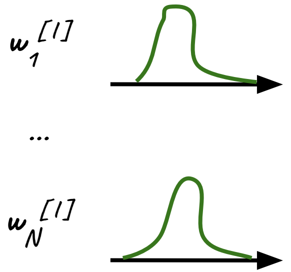
https://www.tensorflow.org/probability

https://keras.io/examples/keras_recipes/bayesian_neural_networks/

Those are standard, widely used, *deterministic* Neural Networks: trained with a *training set*, cross-validated with a *dev set*, finally evaluated with a *test set*, then deployed for production.
For each new example you'll get a single *point estimate* prediction, with no information about the uncertainty of the model, nor the prediction
→ enters *Bayesian Neural Networks* : instead of learning a single, specific set of *weights*, try to learn the whole *weight distributions*, encoding the model uncertainty, from which we can sample to produce (still, *a single*) output for a given input.



The weights are not single, fixed model parameters, but are sampled from a distribution $p(w_i^{[l]}|x)$



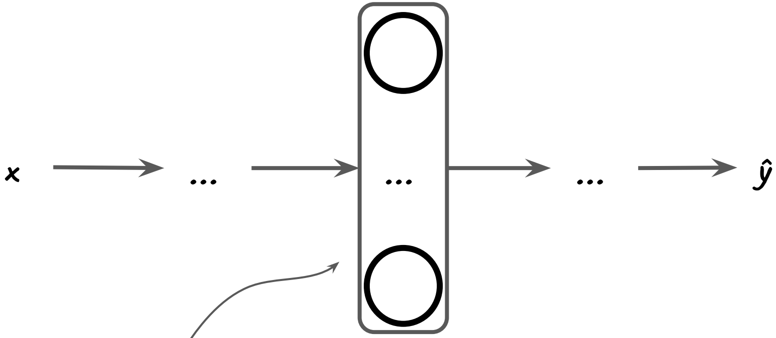
e.g., if the weights are modeled with *Normal* distributions, a *Bayesian Neural Networks* just need to find a mean **μ** and a variance **σ** to describe them

and, if you remind from the first lesson, ℓ_2 regularization **forces** the weights to be normally distributed with mean zero (and ℓ_1 actually forces some of those to be exactly zero)

https://keras.io/examples/keras_recipes/bayesian_neural_networks/

Bayesian Neural Networks

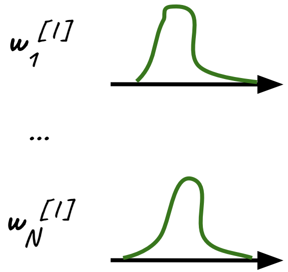
Those are standard, widely used, *deterministic* Neural Networks: trained with a *training set*, cross-validated with a *dev set*, finally evaluated with a *test set*, then deployed for production.
For each new example you'll get a single *point estimate* prediction, with no information about the uncertainty of the model, nor the prediction
→ enters *Bayesian Neural Networks* : instead of learning a single, specific set of *weights*, try to learn the whole *weight distributions*, encoding the model uncertainty, from which we can sample to produce (still, *a single*) output for a given input.



"DENSE VARIATIONAL LAYER"

$w^{[l]}$

The weights are not single, fixed model parameters, but are sampled from a distribution $p(w_i^{[l]}|x)$



but in principle that distribution might be whatever *PDF* you could think of, whose precise measurement is a complicated task

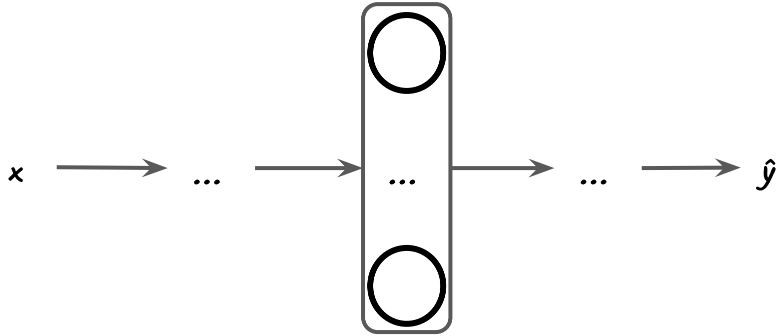
Sampling methods

Approximately compute the distribution by generating a finite set of weights $(w_1 \dots w_N)$ whose distribution matches $p(w|x)$ in the limit of large N.
The challenge is to relatively quickly produce a small number N of network samples that yield a decent approximation of the *PDF*.

Variational methods

Directly model the posterior $p(w|x)$ using a parametrized distribution $q_\phi(w)$ called the *approximate posterior*, then iteratively improve the approximation by solving a suitable optimization problem (***stochastic variational inference***), e.g. minimizing the ***Kullback-Liebler divergence***

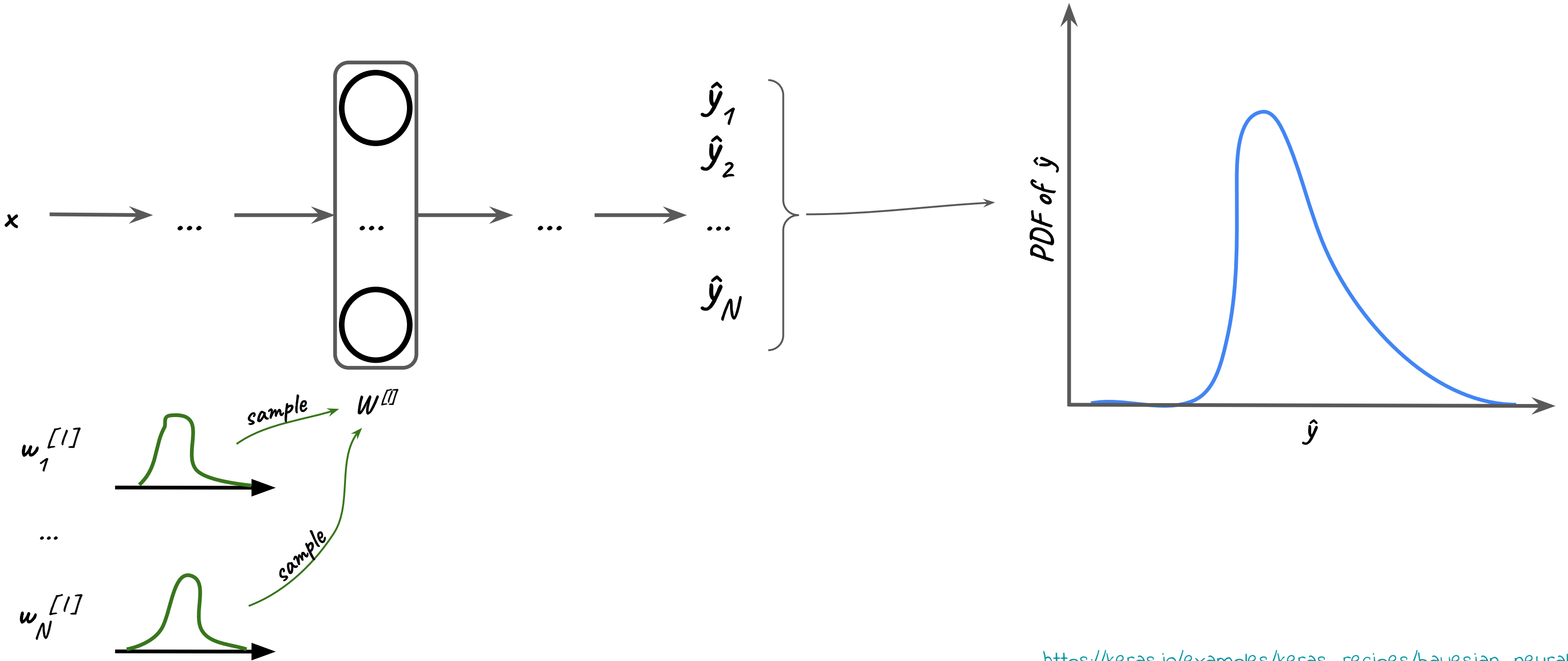
Once you have the weights distributions $p(w_i^{[l]}/x)$, the next step is to sample from the distribution each time to produce a ***distribution*** in output, instead of a single point estimate, for the predicted label $\hat{y}$.
In this way, the model will capture both the *epistemic* (model) and *aleatoric* (data) uncertainties, due to the stochastic process generating the ***weights*** distributions inevitably influenced by the irreducible noise in the data.



https://www.tensorflow.org/probability

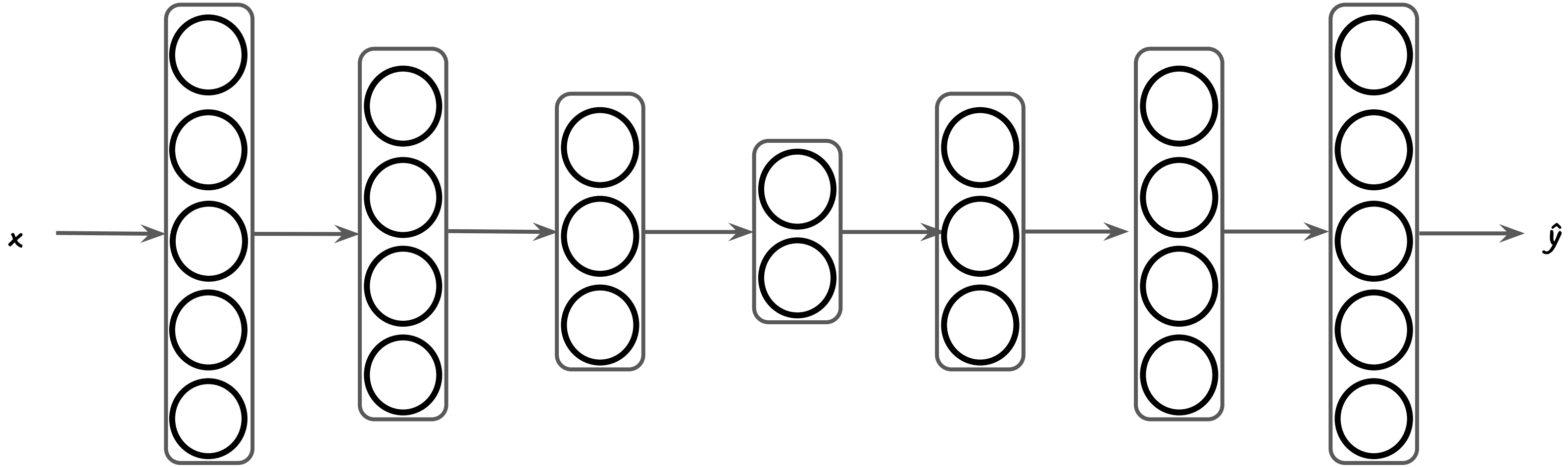
https://keras.io/examples/keras_recipes/bayesian_neural_networks/

Once you have the weights distributions $p(w_i^{[l]}/x)$, the next step is to sample from the distribution each time to produce a ***distribution*** in output, instead of a single point estimate, for the predicted label $\hat{y}$.
In this way, the model will capture both the *epistemic* (model) and *aleatoric* (data) uncertainties, due to the stochastic process generating the ***weights*** distributions inevitably influenced by the irreducible noise in the data.

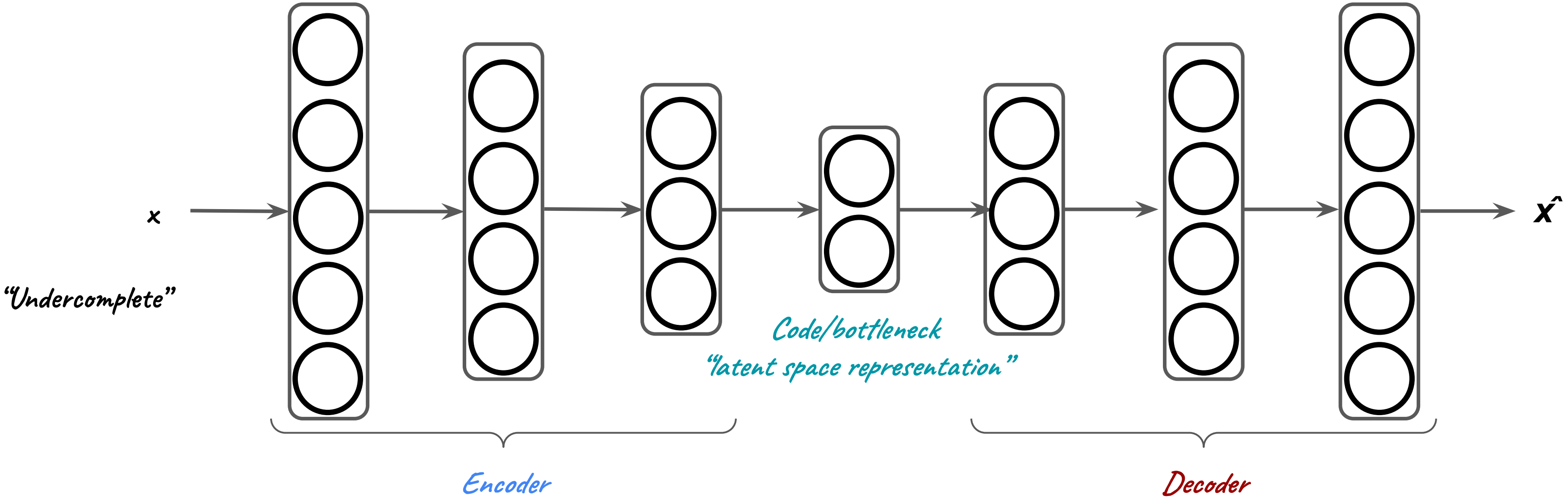


https://keras.io/examples/keras_recipes/bayesian_neural_networks/

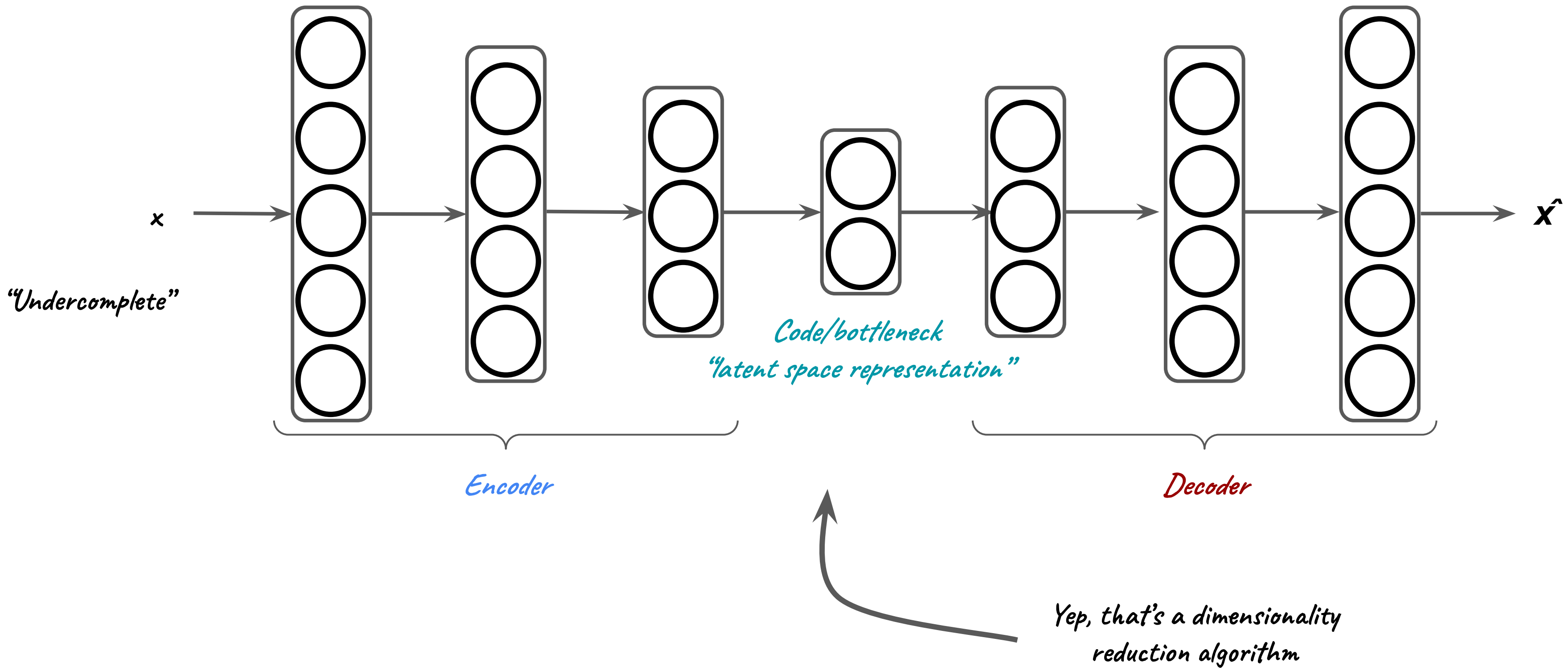
Take a look at this Neural Network. What do you think it is doing?



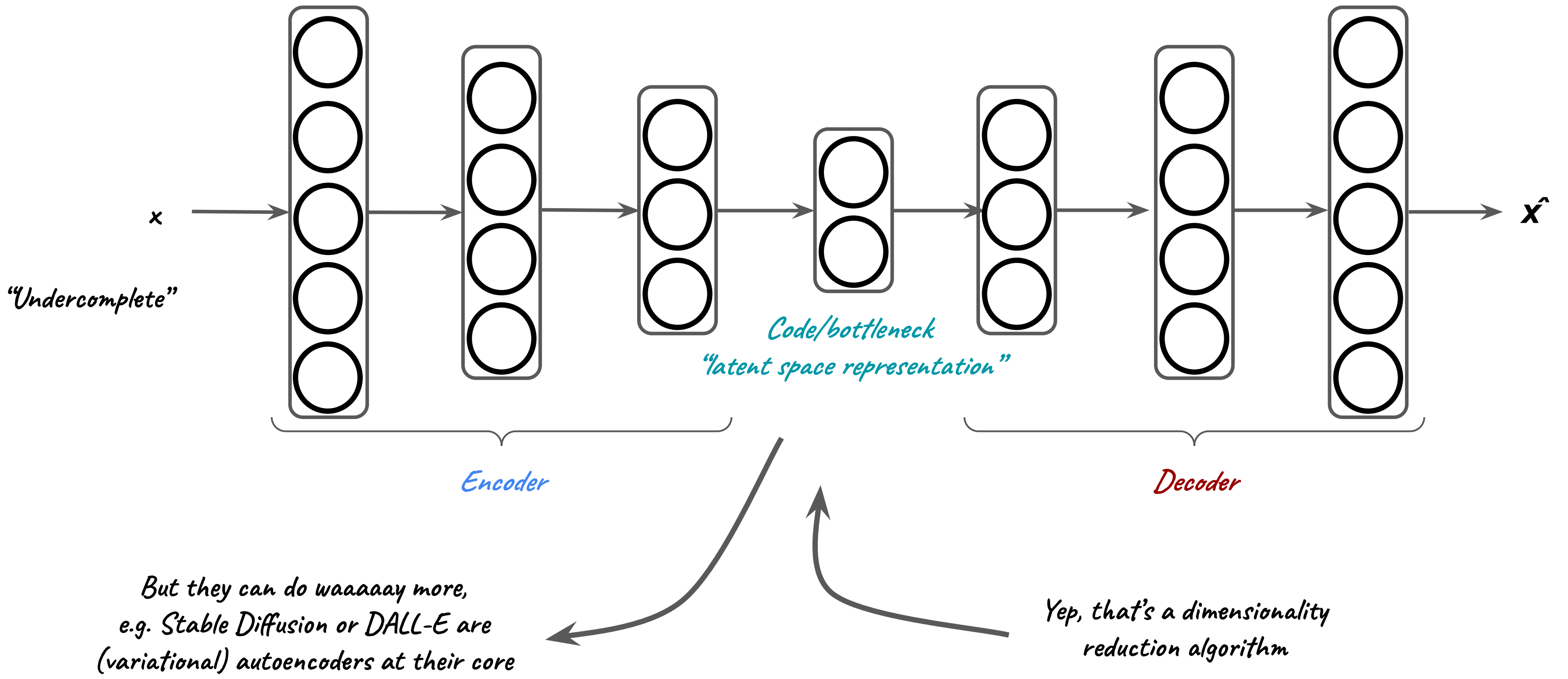
This is an *autoencoder*, a Neural Network trained to attempt to copy its input to its output.
In doing so, it passes through a bottleneck, called the *code* or *"latent space representation"*, which is a compressed representation of the input data. The first part is called the *encoder*, which learns how to compress $\mathbf{x}$ into the *code*, and the second part is the *decoder*, that learns how to *reconstruct* $\hat{\mathbf{x}}$ from the compressed representation in the bottleneck layer.



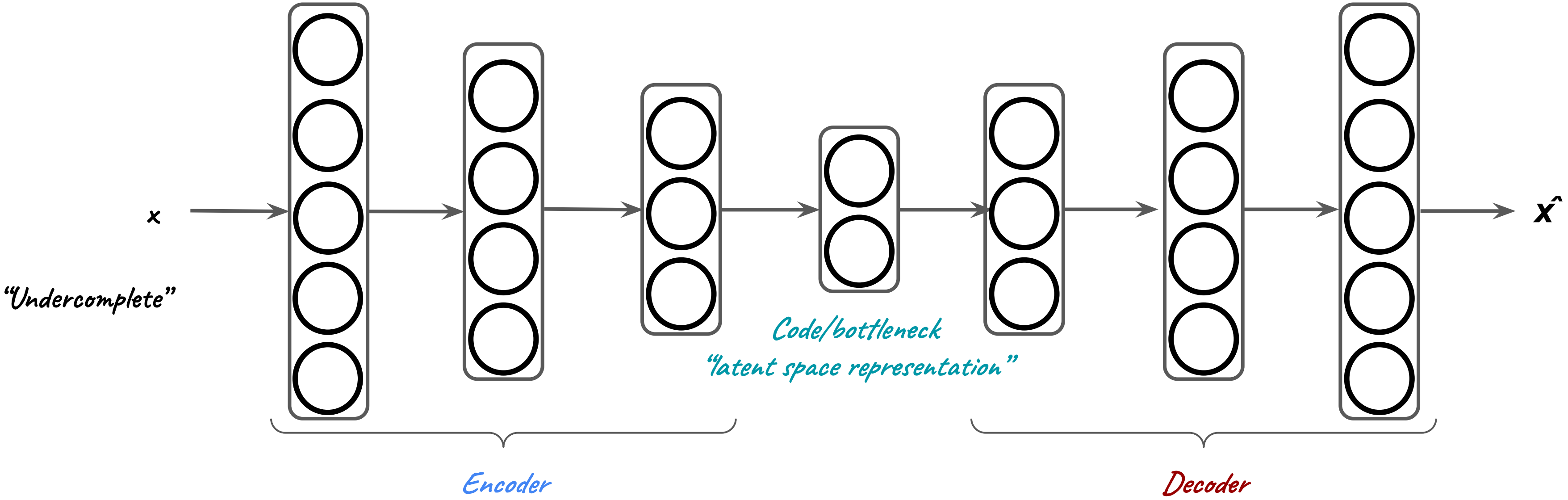
This is an *autoencoder*, a Neural Network trained to attempt to copy its input to its output.
In doing so, it passes through a bottleneck, called the *code* or *"latent space representation"*, which is a compressed representation of the input data. The first part is called the *encoder*, which learns how to compress $\mathbf{x}$ into the *code*, and the second part is the *decoder*, that learns how to *reconstruct* $\hat{\mathbf{x}}$ from the compressed representation in the bottleneck layer.



This is an *autoencoder*, a Neural Network trained to attempt to copy its input to its output.
In doing so, it passes through a bottleneck, called the *code* or *"latent space representation"*, which is a compressed representation of the input data. The first part is called the *encoder*, which learns how to compress x into the *code*, and the second part is the *decoder*, that learns how to *reconstruct* $\hat{x}$ from the compressed representation in the bottleneck layer.

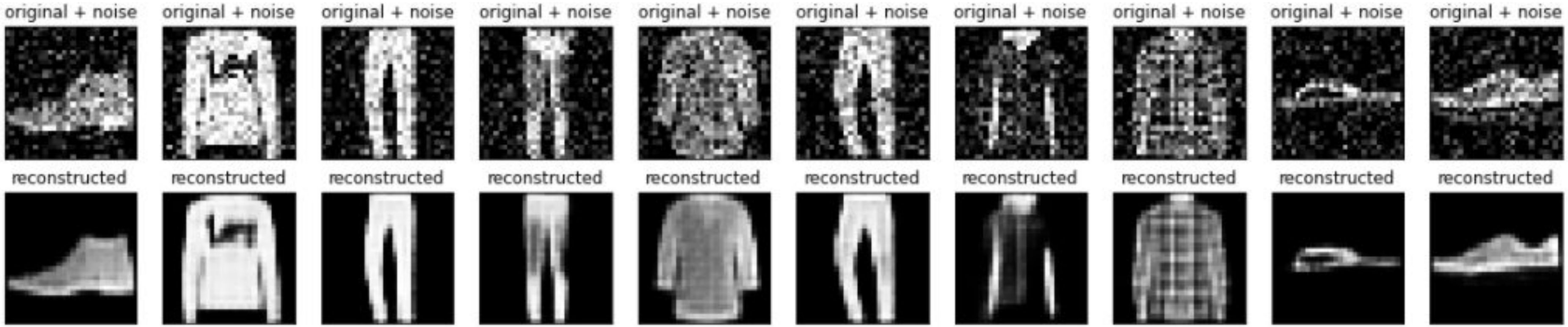


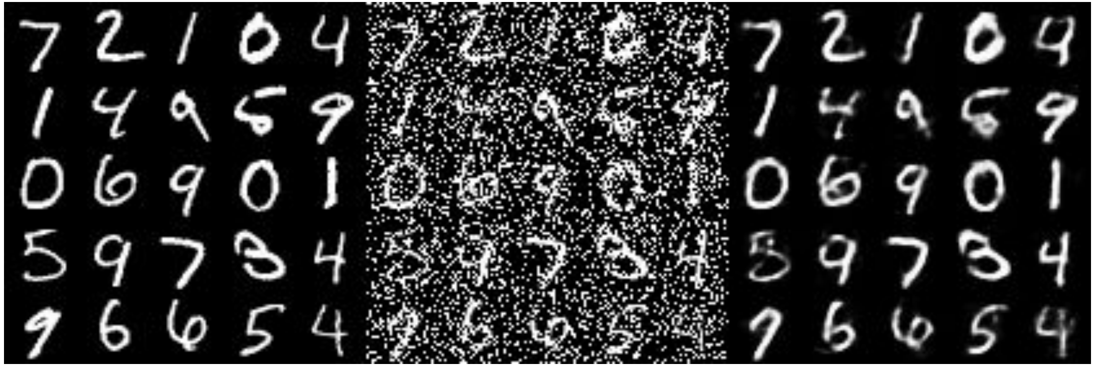
This is an *autoencoder*, a Neural Network trained to attempt to copy its input to its output.
In doing so, it passes through a bottleneck, called the *code* or *"latent space representation"*, which is a compressed representation of the input data. The first part is called the *encoder*, which learns how to compress x into the *code*, and the second part is the *decoder*, that learns how to *reconstruct* $\hat{x}$ from the compressed representation in the bottleneck layer.



In practice, an autoencoder is trying to learn a function $f_{W,b}(x) = h$ which *encodes* the data information, and a function $g_{W,b}(h) = g_{W,b}(f_{W,b}(x)) = \hat{x}$ *decoding* the data from the *latent space representation*, while minimizing the reconstruction error.
Of course, learning the identity function is useless, so these are designed to be unable to learn to copy perfectly. By placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data.

An *autoencoder* can be used to, e.g., doing dimensionality reduction and for visualization, for anomaly detection, but also for image denoising, or to generate new data that resembles the input data.

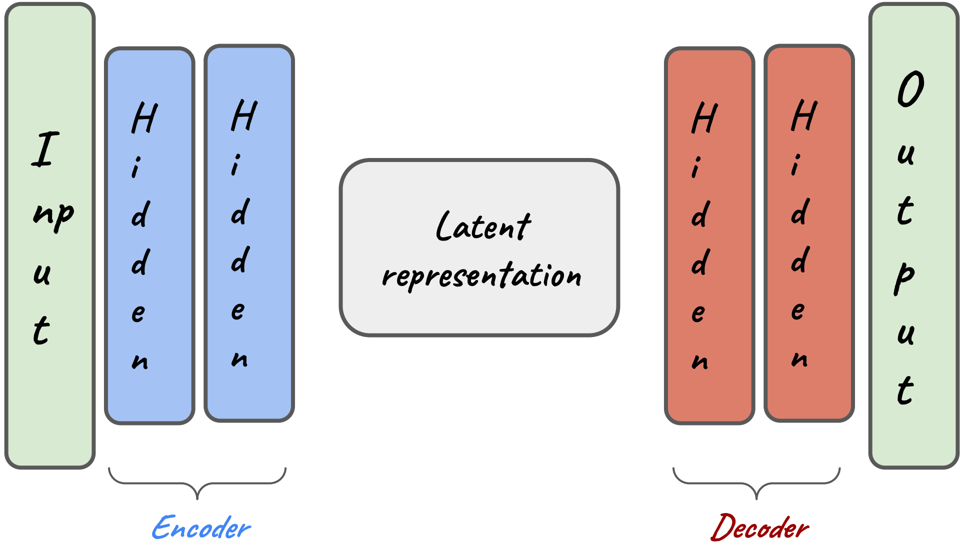




http://www.opendeep.org/v0.0.5/docs/tutorial-your-first-model

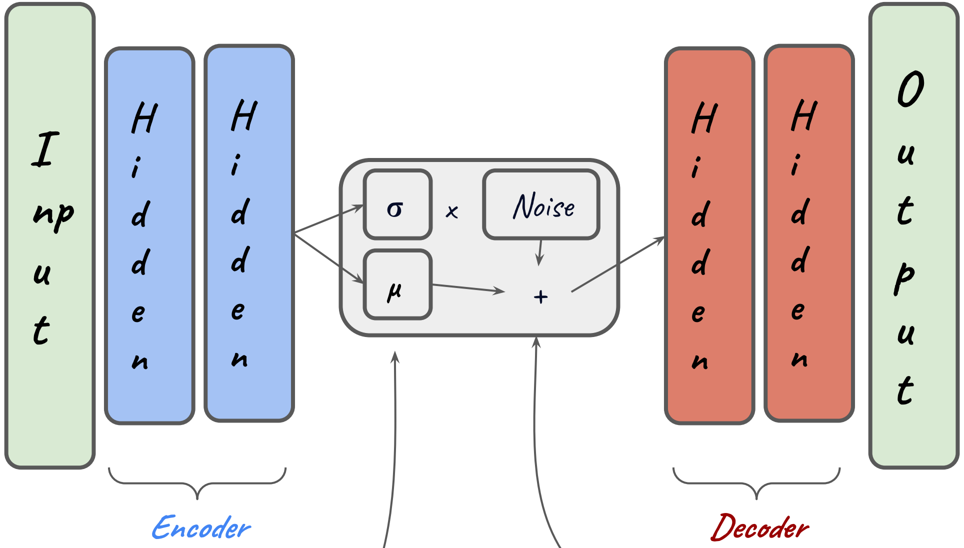
https://www.tensorflow.org/tutorials/generative/autoencoder

Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now *Stable Diffusion* and *DALL-E* work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.



Kingma and Welling, 2013, Auto-Encoding Variational Bayes, https://arxiv.org/abs/1312.6114

Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now *Stable Diffusion* and *DALL-E* work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.

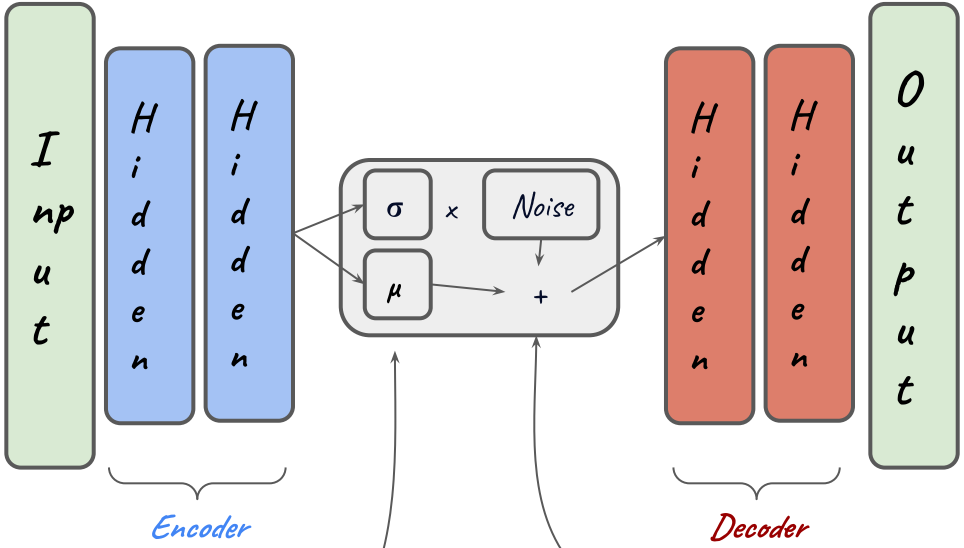


We take two outputs from the *encoder*: a mean encoding **μ** and a variance **σ** of the encoding

We also generate noise using a Gaussian distribution, so that the actual encoding will be sampled from the distribution using the mean encoding **μ** and the noise multiplied encoding variance **σ**.

Kingma and Welling, 2013, *Auto-Encoding Variational Bayes*, https://arxiv.org/abs/1312.6114

Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now *Stable Diffusion* and *DALL-E* work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.



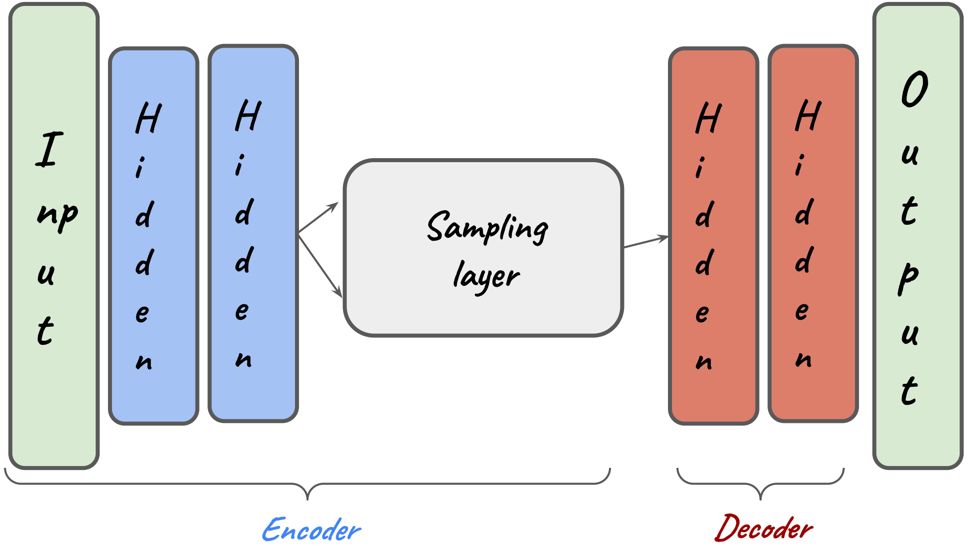
Doing this, a *VAE* is actually generating something completely new, but that it resemble enough the input data

We take two outputs from the *encoder*: a mean encoding μ and a variance σ of the encoding

We also generate noise using a Gaussian distribution, so that the actual encoding will be sampled from the distribution using the mean encoding μ and the noise multiplied encoding variance σ.

Kingma and Welling, 2013, Auto-Encoding Variational Bayes, https://arxiv.org/abs/1312.6114

Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now *Stable Diffusion* and *DALL-E* work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.



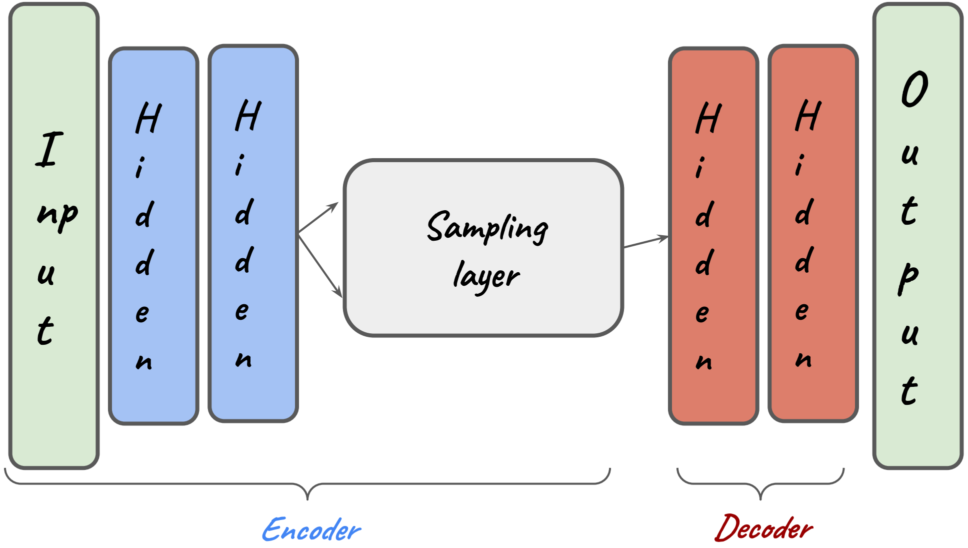
Doing this, a *VAE* is actually generating something completely new, but that it resemble enough the input data

We take two outputs from the *encoder*: a mean encoding μ and a variance σ of the encoding

We also generate noise using a Gaussian distribution, so that the actual encoding will be sampled from the distribution using the mean encoding μ and the noise multiplied encoding variance σ.

Kingma and Welling, 2013, *Auto-Encoding Variational Bayes*, https://arxiv.org/abs/1312.6114

Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now *Stable Diffusion* and *DALL-E* work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.



Doing this, a *VAE* is actually generating something completely new, but that it resemble enough the input data

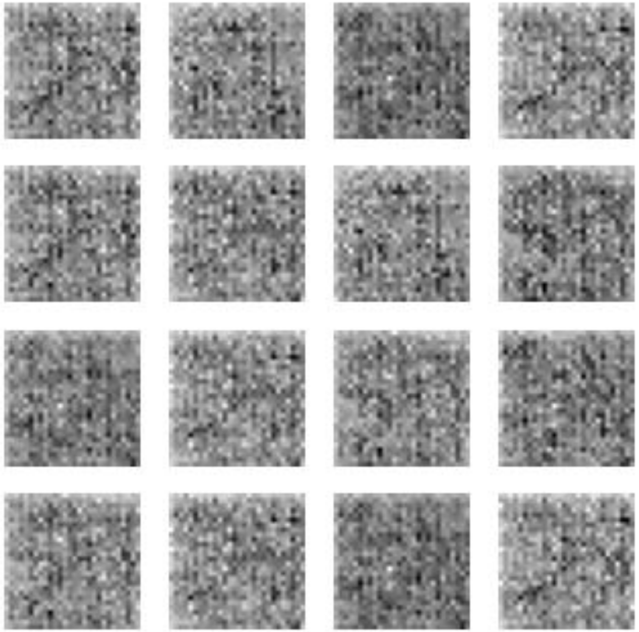
The loss function is usually a *Kullback-Liebler* loss.

We take two outputs from the *encoder*: a mean encoding μ and a variance σ of the encoding

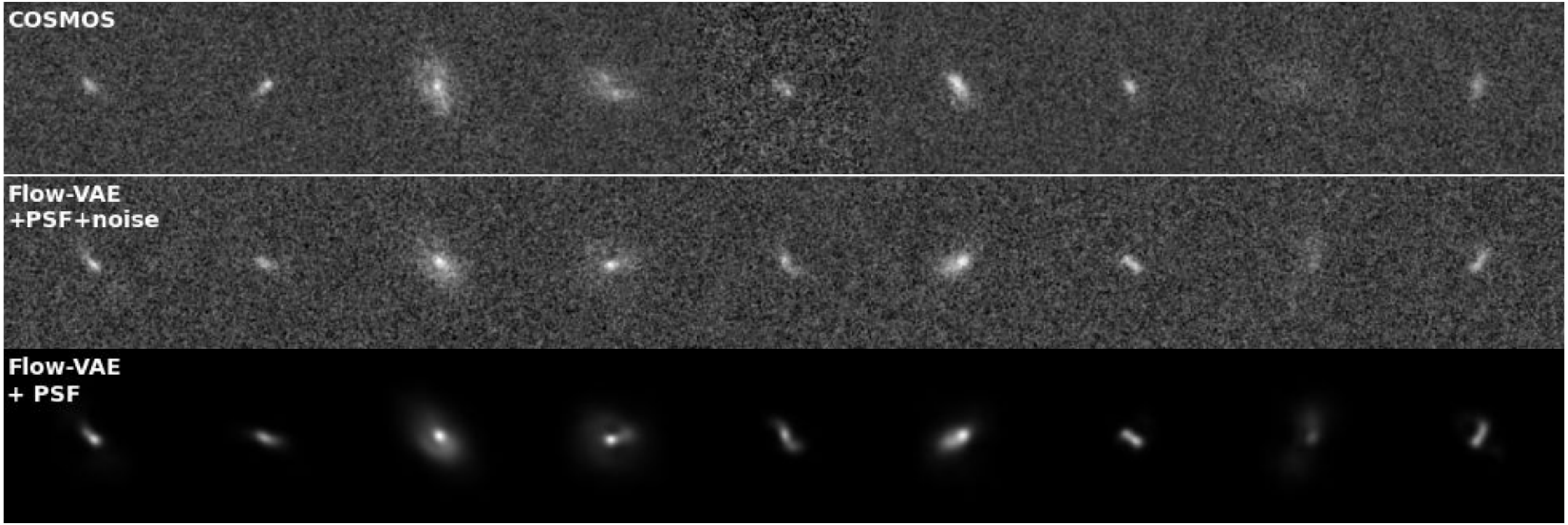
We also generate noise using a Gaussian distribution, so that the actual encoding will be sampled from the distribution using the mean encoding μ and the noise multiplied encoding variance σ.

Kingma and Welling, 2013, Auto-Encoding Variational Bayes, https://arxiv.org/abs/1312.6114

Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now ***Stable Diffusion*** and ***DALL-E*** work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.



Funny thing: until 2013 *autoencoders* were seen as the cute, didactic algorithm with absolute no practical application in real world. Now *Stable Diffusion* and *DALL-E* work on autoencoders, or better, on *variational autoencoders*.
In *VAE* the latent representation will be probabilistic, so the later output is partially determined by chance, so *VAE* can create new instances of the data that look and feel like they came from the training set.



Lanusse et al. 2021, "Deep generative models for galaxy image simulations" MNRAS, 504, 5543