BUILDING UNIVERSAL UNDERSTANDING

Deep Learning Meetup @Unbabel

Unbabel

Expectations

What this presentation **won’t** be

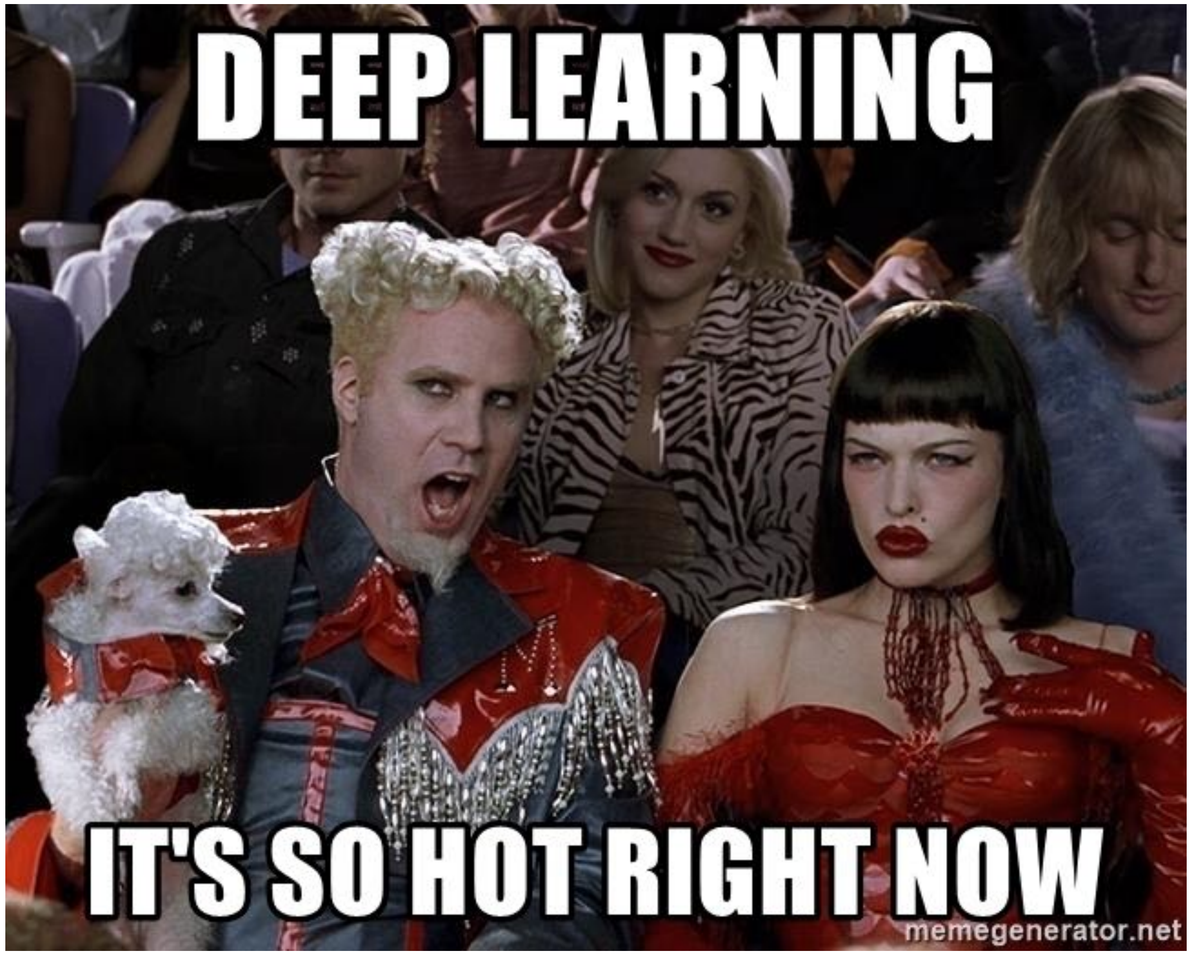
- A mathematics class
- A comprehensive overview of machine learning
- An intensive course in deep learning
- A deep dive into the presented techniques

What this presentation **will** be

- A brief overview of the field
- A review of some machine learning concepts with a step up to deep learning
- A light presentation with some pointers
- A practical meetup where you can get started with simple code examples

What this presentation **will** be

What is deep learning?


DEEP LEARNING
IT'S SO HOT RIGHT NOW
memegenerator.net

The AI/deep learning hype

Facebook kills AI that invented its own language because English was slow

Google AI platform Neural Machine Translation develops internal language

Artificial intelligence can now emulate human behaviors – soon it will be dangerously good

11.42am BST

OpenAI Writes So Well That Creators Won't Release It, Fearing Flood of Disinformation

What is deep learning

Artificial Intelligence

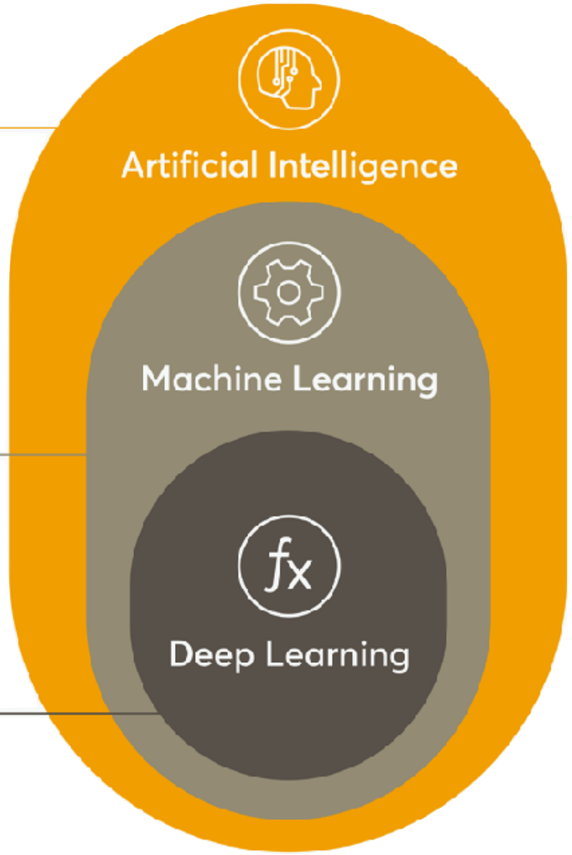
Any technique which enables computers to mimic human behaviour.

Machine Learning

Subset of AI techniques which use statistical methods to enable machines to improve with experiences.

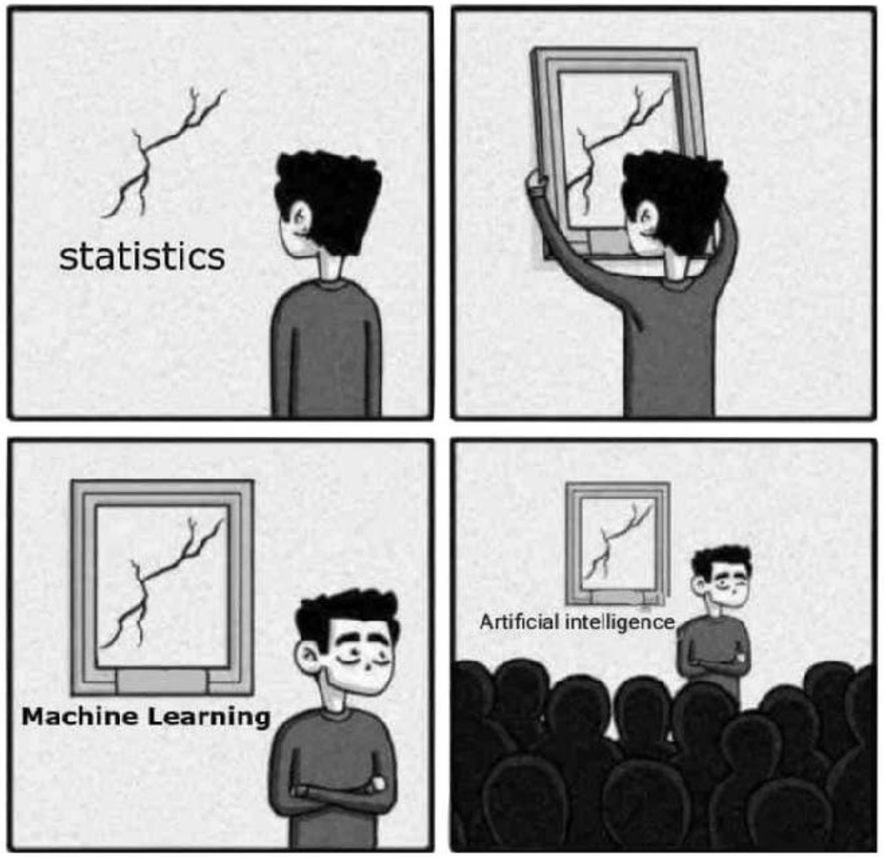
Deep Learning

Subset of ML which makes the computation of multi-layer neural networks feasible.



source: https://www.nastel.com/blog/simple-guide-ai-machine-learning-deep-learning/

Is machine learning just statistics?



source: https://towardsdatascience.com/no-machine-learning-is-not-just-glorified-statistics-26d3952234e3

- Most methods rooted in statistics, but it is a different set of techniques/algorithms
- Some methods more related than others
- At the end of the meetup, maybe you'll make up your own opinion about it

An introduction to machine learning

Supervised learning

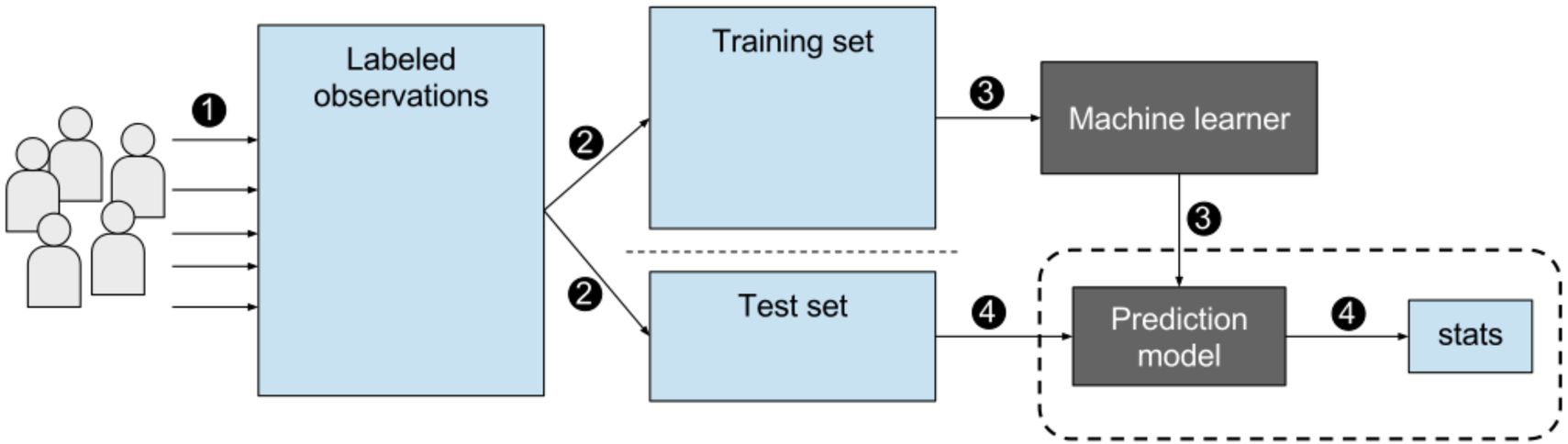
- **Supervised learning**: model learns from labeled data, this is, several examples where you have the desired output - the true answer

- **Unsupervised learning**: dataset does not have an explicit solution, and it is just a collection of example data with no label. The methods need to learn automatically some sort of structure in the data purely by analysing features

- **Reinforcement learning**: The model attempts some answer on a particular goal, and receives a reward signal depending how well it performs. Overall aim is to maximise this reward.

Supervised learning

- We will focus only on supervised learning: for our tasks we have labeled observations, with which we can train or model to learn to perform well in its predictions



source: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/

Simple tasks - regression

- Let’s say everytime you go out with your friends, you register the amount of money you spend, together with some information you consider useful, like the number of people that went out

- After a few times, you that maybe you can use this data to predict how much you’re going to spend on your next night out

Number of people	Money spent
4	20.4
6	13.5
3	24.13
6	14.12
…	…
8	13.12

Simple tasks - regression

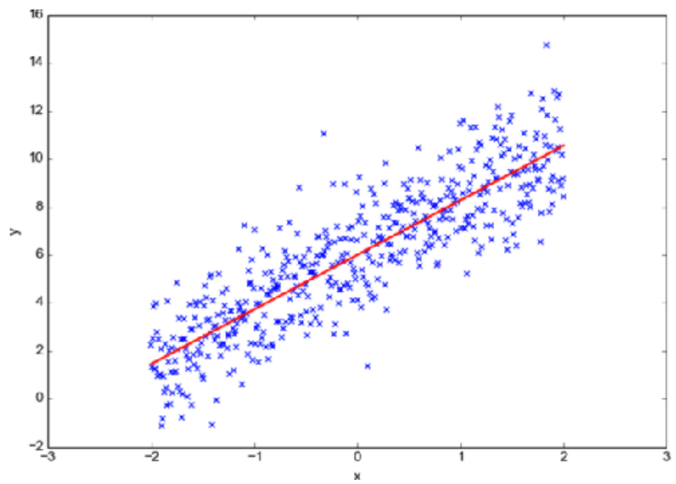
Or let’s say you did the same exercise for the money you spend on the supermarket and the number of days since you last went there

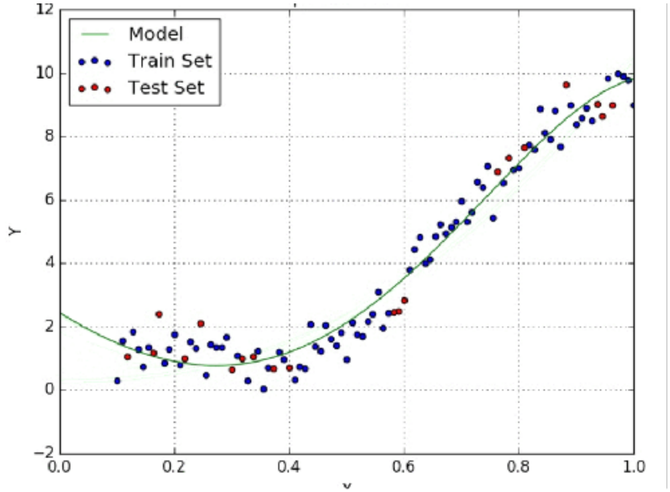
- You also want to predict how much you will spend on your next trip to the supermarket

Number of days	Money spent
2	10.1
1	6.50
6	24.13
6	14.12
…	…
5	45.02

Simple tasks - regression

- Both these tasks are simple **regression** tasks. **Regression predictive tasks** try to approximate a mapping from input variables **X** to a continue output variable **y**. You probably already heard at some point of this concept, for example in its simple form of linear regression or polynomial regression, the figures show below.





source: https://towardsdatascience.com/introduction-to-linear-regression-and-polynomial-regression-f8adc96f31cb

Start with linear models

We can define our regressions as a function a(x) that we want to model and that can be represented by a linear combination of our inputs **x**:(x_0, x_1, ... x_n), also called features, through weights w_0, w_1, ..., w_{n+1} :

$$a_w(x) = w_0 + w_1 x_1 + w_2 x_2$$

We can also write it in a vector form, as a dot product between features and weights.

$$a_w(x) = \sum_{i=1}^{n} w_i x_i = \overrightarrow{w}^T \overrightarrow{x}$$

Our problem becomes one of finding weights **w** for our model that generate good predictions

Measuring error

We need to measure how well predictions from the model a(x) fit our examples y.

For this purpose we define a measure of error between predictions and labels, which we call an error function, or loss function. For regression, it is common to use mean squared errors as this metric, defined below.

$$L(x, y, w) = \frac{1}{m} \sum_{i=1}^{m} (a_w(x(i)) - y(i))^2$$

How does it learn

For simple regression models we can often produce a solution that minimises our error through exact methods.

However, we'll focus on learning techniques that iteratively find better estimations of the model, converging into good approximations of these models

This iterative technique is called **Gradient Descent**

Gradient Descent

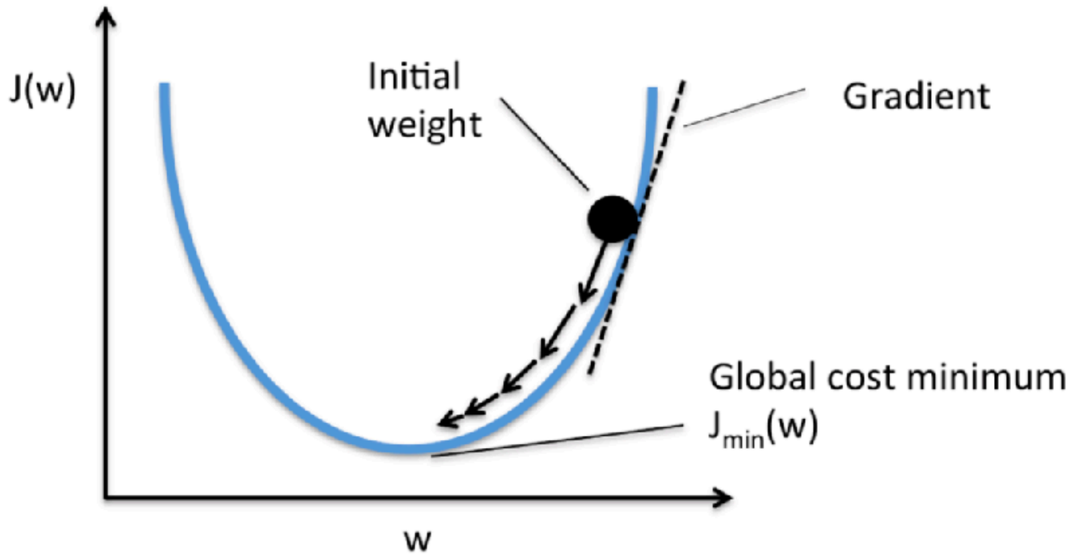
Gradient Descent makes use of the loss function, moving into the direction of error minimisation. For this purpose, it makes use of the derivative of the loss function.

The updates are done as follows:

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} L(x, y, w)$$

For our cost function:

$$\frac{\partial}{\partial w_j} L(x, y, w) = (a_w(\vec{x}) - y)x_j$$



Gradient Descent

$$\vec{w} = \vec{w}^0$$

for **k** in **n_iter**:

$$\Delta L(w) = X^T(X\vec{w} - y)$$

$$\vec{w} := \vec{w} - \alpha \Delta L(X, \vec{y}, \vec{w})$$

X - matrix of example features [n_samples, n_features]

y - vector of example labels [n_samples, 1]

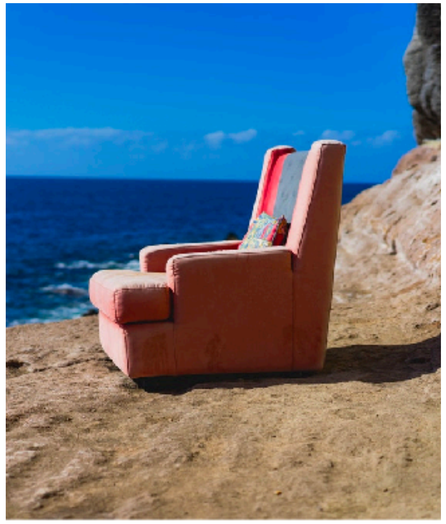
w - vector of weights [n_features, 1]

α - learning rate

Simple tasks - classification

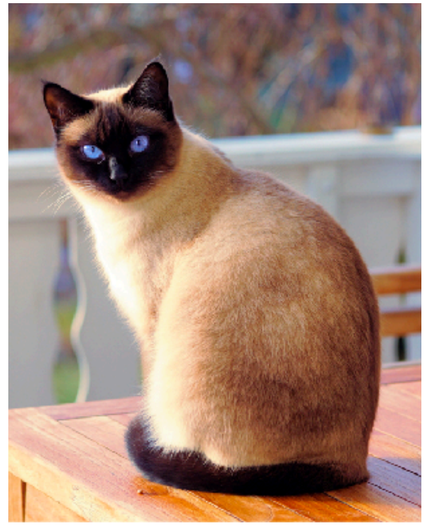
Now, let’s say that instead of predicting some value according to variables you collected, you want to split data into classes, for example identifying if a picture has a dog or not.

label: dog



label: not a dog

label: dog

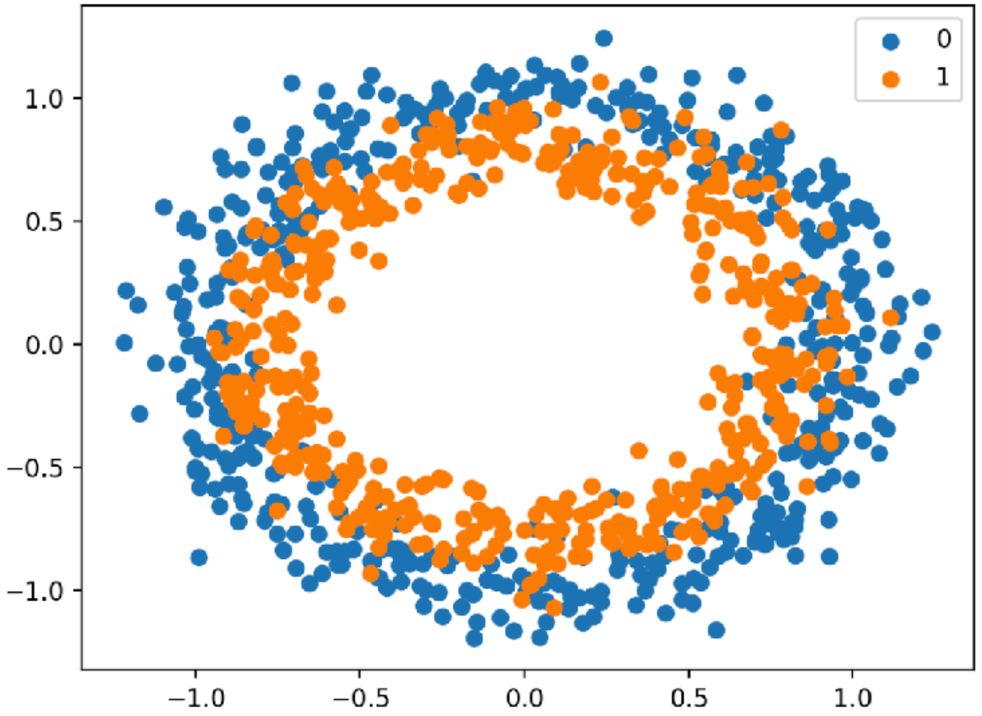


label: not a dog

label: not a dog

Simple tasks - classification

Or we want to do something slightly simpler, like finding a model to classify the data in the following picture



source: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Simple tasks - classification

- Both of these are **classification tasks**, in particular binary classification tasks, since we only have two classes. In these, what we want to predict is usually a class from a discrete set of options.

- In binary classification, you can tweak the problem as that of finding the probability of the example belonging to the default class

$$P(y = 1 \mid x; w) = a_w(x)$$

$$P(y = 0 \mid x; w) = 1 - a_w(x)$$

A linear model for classification

We still would like to use a linear model for this particular use case, however, if you define a(x) as follows:

$$a_w(x) = w_0 + w_1 x_1 + w_2 x_2$$

$$a(x) = \sum_{i=1}^{n} w_i x_i = \overrightarrow{w}^T \overrightarrow{x}$$

The output value does not translate to a probability. We thus use the **logistic function** - a function that will transform our output into a value that can be interpreted as a probability

$$a_w(x) = \sigma(\overrightarrow{w}^T \overrightarrow{x})$$

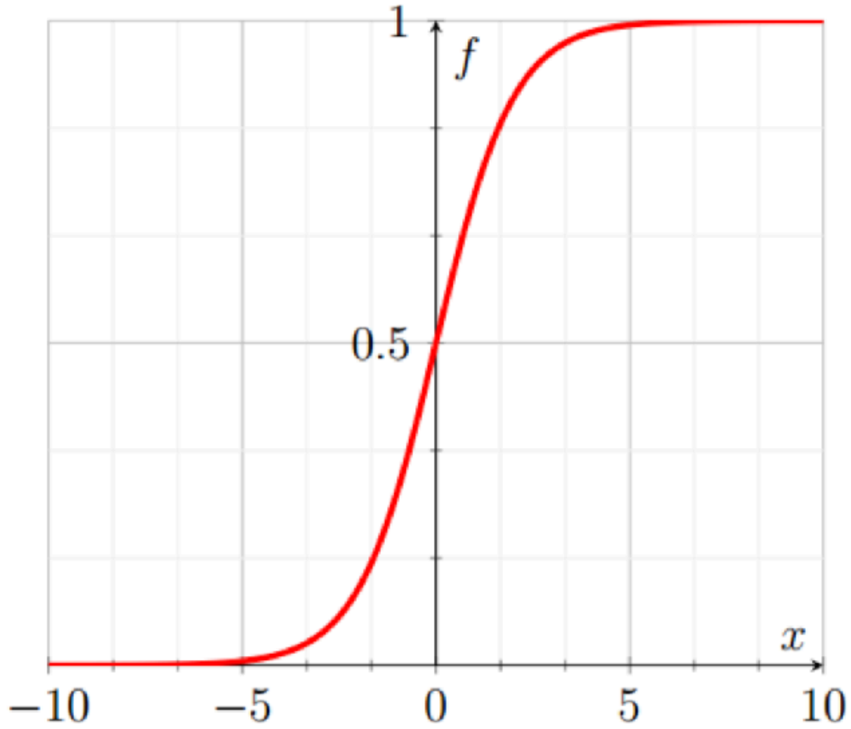
Our problem is still to find weights θ for our model

Logistic function

The logistic function is defined as follows

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

Which, as you can see, it transforms any input into a value between 0 and 1



Measuring error

For this particular task, we'll measure error through cross entropy. This is a measure of how well two distributions fit. It relies on that by comparing predictions with the true labels in the following way for a given sample i:

$$l(x_i, y_i, w) = -\left[y_i \cdot logP(y_i = 1 \mid x_i, w) + (1 - y_i) \cdot log(1 - P(y_i = 1 \mid x_i, w))\right]$$

And for many samples, we just average over the metric:

$$L(X, \overrightarrow{y}, w) = \frac{1}{\ell} \sum_{i=1}^{\ell} l(x_i, y_i, w)$$

Gradient Descent

Once again, we'll apply the gradient descent. When we find the derivative of the cross entropy with respect to the weights, we end up with a similar function to the one in regression.

$$\frac{\partial}{\partial w_j} L(x, y, w) = (a_w(\vec{x}) - y)x_j$$

The main difference being that a(x) contains the sigmoid calculation. However, the algorithm is still pretty similar

Gradient Descent

$$\vec{w} = \vec{w}^0$$

for **k** in **n_iter**:

$$\Delta L(w) = X^T(\sigma(X\vec{w}) - y)$$

$$\vec{w} := \vec{w} - \alpha\Delta L$$

X - matrix of example features [n_samples, n_features]

y - vector of example labels [n_samples, 1]

w - vector of weights [n_features, 1]

α - learning rate

The gradient descent method is quite useful, and it has many variations that improve over the initial algorithm. However, we don’t cover those.

We will, however, throughout the exercises, cover the concept of mini-batching, since it is one important aspect of Gradient Descent’s implementation and performance.

We’ll apply these concepts now in a few exercises

Practice time!

https://github.com/CatarinaSilva/meetup-deeplearning

Neural Networks

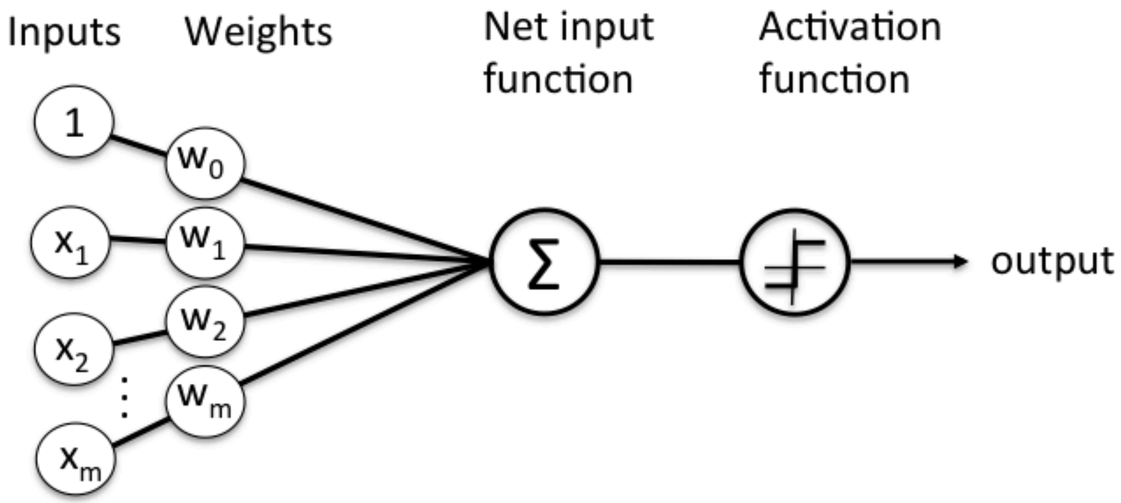
What’s next?

In the first part, you’ve seen how to use the iterative method of gradient descent to adjust the weights of our model

However, these models are quite simple and even to be able to model non-linear data they require a lot of feature engineering

We’ll now see a different type of models - neural networks - that are much more powerful to solve these types of tasks.

Rosenblatt’s single layer perceptron



soure: https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

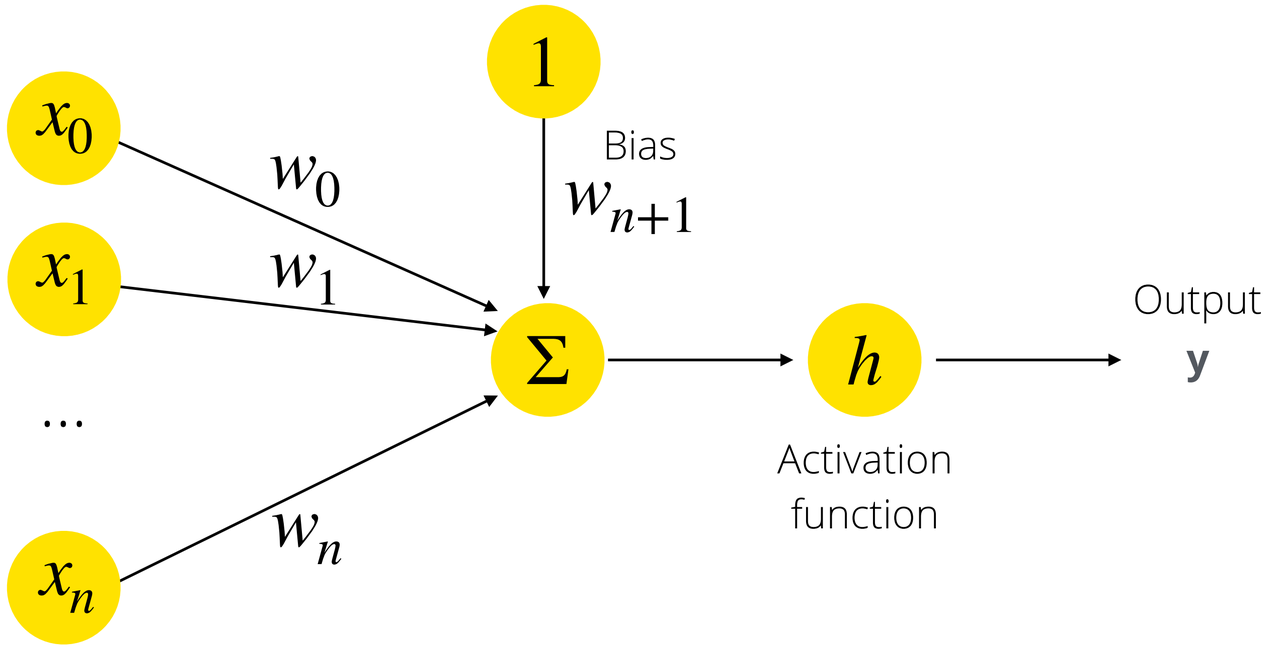
Very similar to what we’ve done before for classification, although it uses the sign function

Can handle only linearly separable data, or data that uses expanded non linear features

It is the basic unit for neural networks

Basic unit - Neuron

The simplest basic units for neural networks - neurons - look very similar to the previous perceptron. However, in place of the sign function, you might different activation functions, such as the sigmoid or the hyperbolic tangent



Sigmoid

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

Hyperbolic Tangent

$$h(z) = tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

Linear constraint

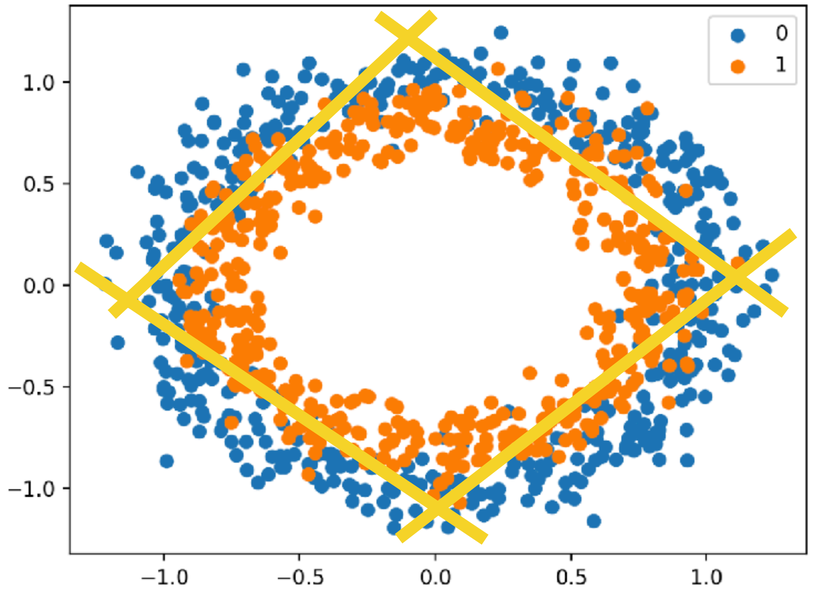
We've seen that the models so far only split linearly separable data

But what if we could find several boundaries

$$z_1(x) = \sigma(w_1^{z1}x_1 + w_2^{z1}x_2 + w_3^{z1})$$
$$z_2(x) = \sigma(w_1^{z2}x_1 + w_2^{z2}x_2 + w_3^{z2})$$
$$z_3(x) = \sigma(w_1^{z3}x_1 + w_2^{z3}x_2 + w_3^{z3})$$
$$z_4(x) = \sigma(w_1^{z4}x_1 + w_2^{z4}x_2 + w_3^{z4})$$

And combine them in a linear way:

$$a(x) = \sigma(w_1^a z_1 + w_2^a z_2 + w_3^a z_3 + w_4^a z_4 + w_5^a)$$

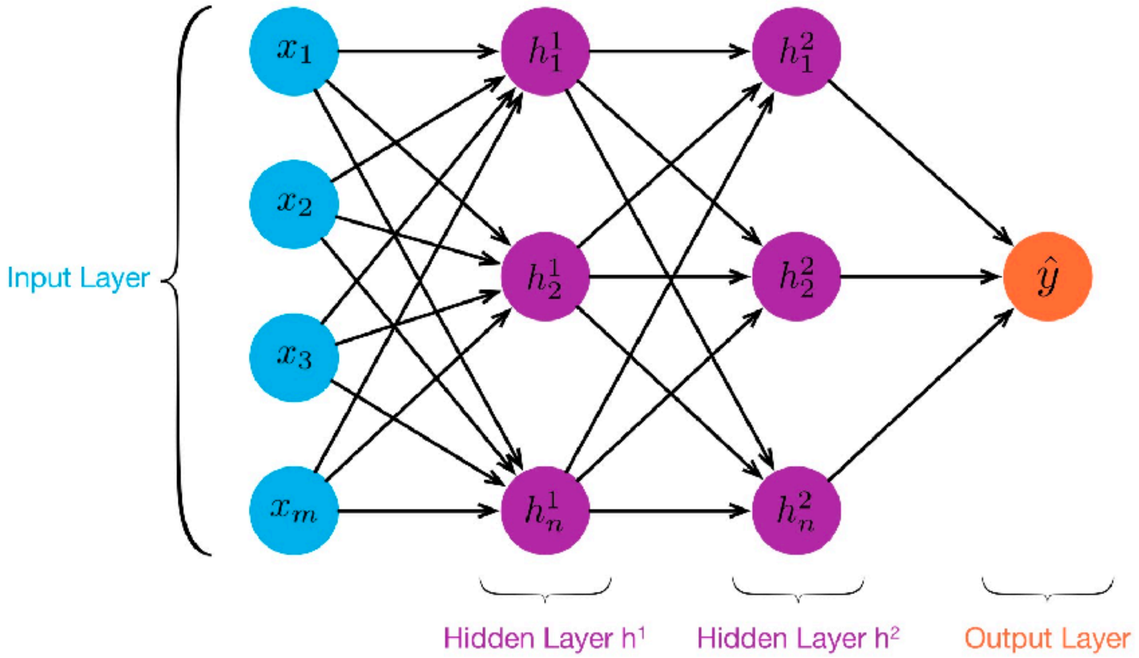


source: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Multi Layer Perceptron

The previous example can be represented as an interaction of different neurons, and we could even add more complexity to our models

This combination is what we call a neural network, and we normally distinguish the input layer and ouput layers from the hidden layers

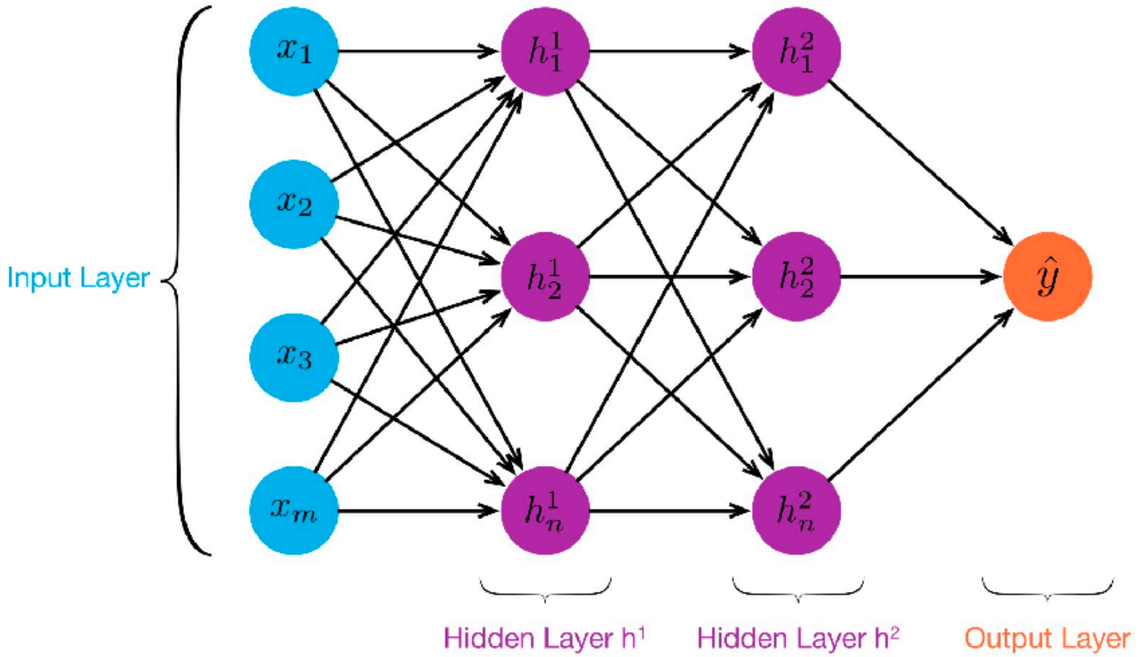


source: https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

Multi Layer Perceptron

This scheme in particular is called multi layers perceptron - it is a feedforward network with with fully connected layers

So how do we train it with the previous methods we've seen?



source: https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

Gradient

We can use the same algorithm - gradient descent, but now we'll have much more parameters and derivatives we'll need to compute.

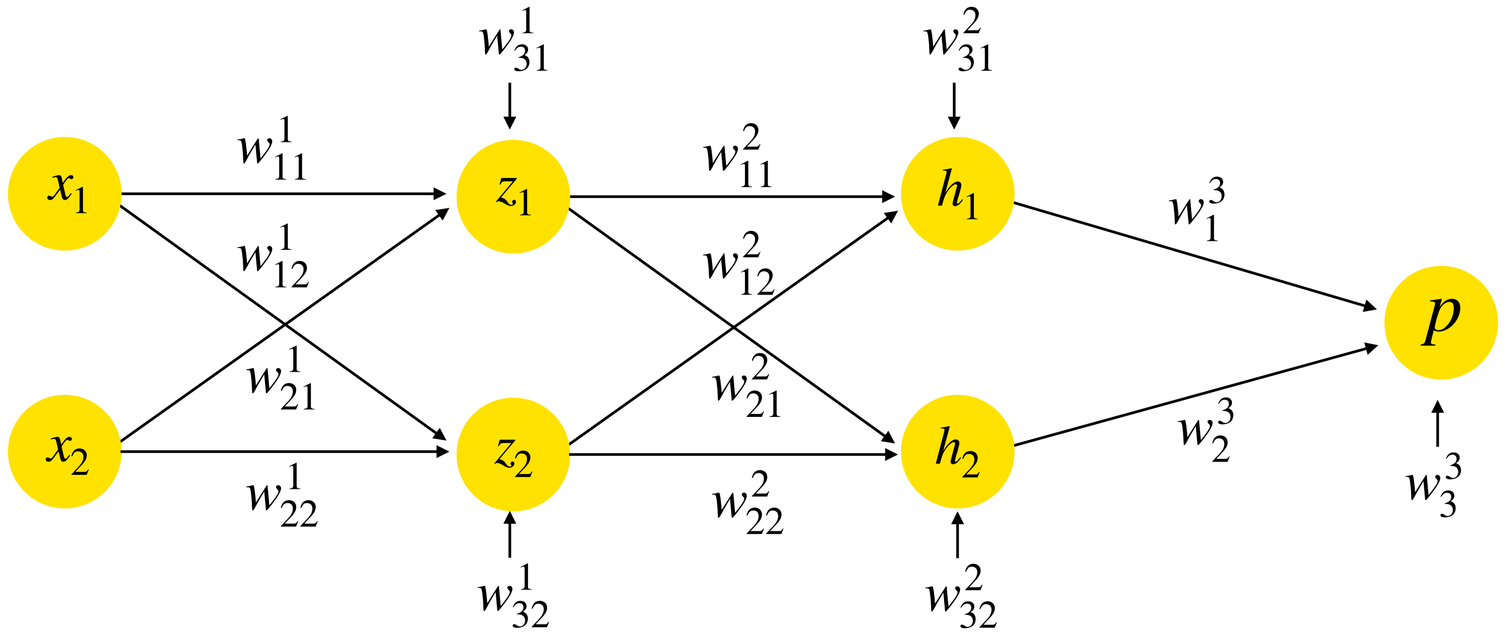
$$\overrightarrow{w} := \overrightarrow{w} - \alpha \Delta_w L(x, y, w)$$

But for that, we need to know 2 things:

- How to generate predictions with our model - we call this the **feedforward** pass
- How to update the weights with the gradients - we call this **backpropagation** (you'll see why)

Multi Layer Perceptron - example

Suppose we have the following network:



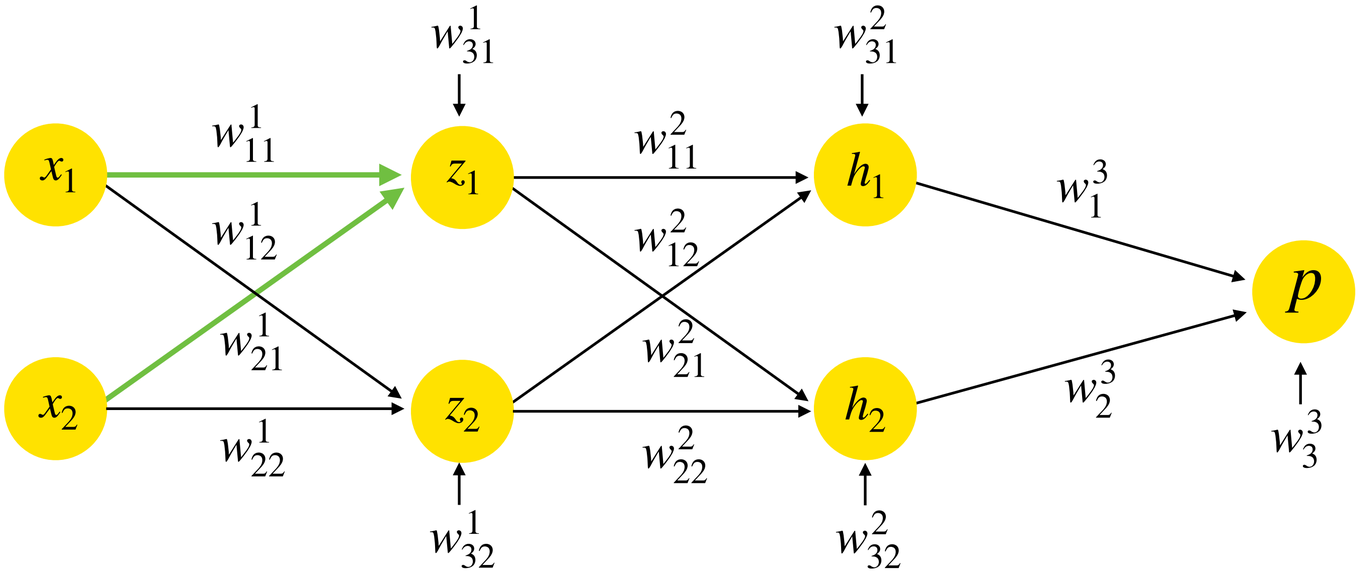
Where each node (except from the input layer) is given by a linear combination of inputs and an activation function g

Feedforward

For each node, we can compute first the linear combination and then the activation function:

$$z_1^{lin} = w_{11}^1 x_1 + w_{21}^1 x_2 + w_{31}^1 \qquad z_1 = g(z_1^{lin})$$

$$z_2^{lin} = w_{12}^1 x_1 + w_{22}^1 x_2 + w_{32}^1 \qquad z_2 = g(z_2^{lin})$$



Which can be rewritten as:

$$z^{lin} = X^T w \qquad z = g(z^{lin})$$

X: [input_size + 1, 1] - input vector with a unit term for the bias computation

w: [input_size + 1, output_size] - weight matrix that transform the input of the layer into the output of the layer

Feedforward

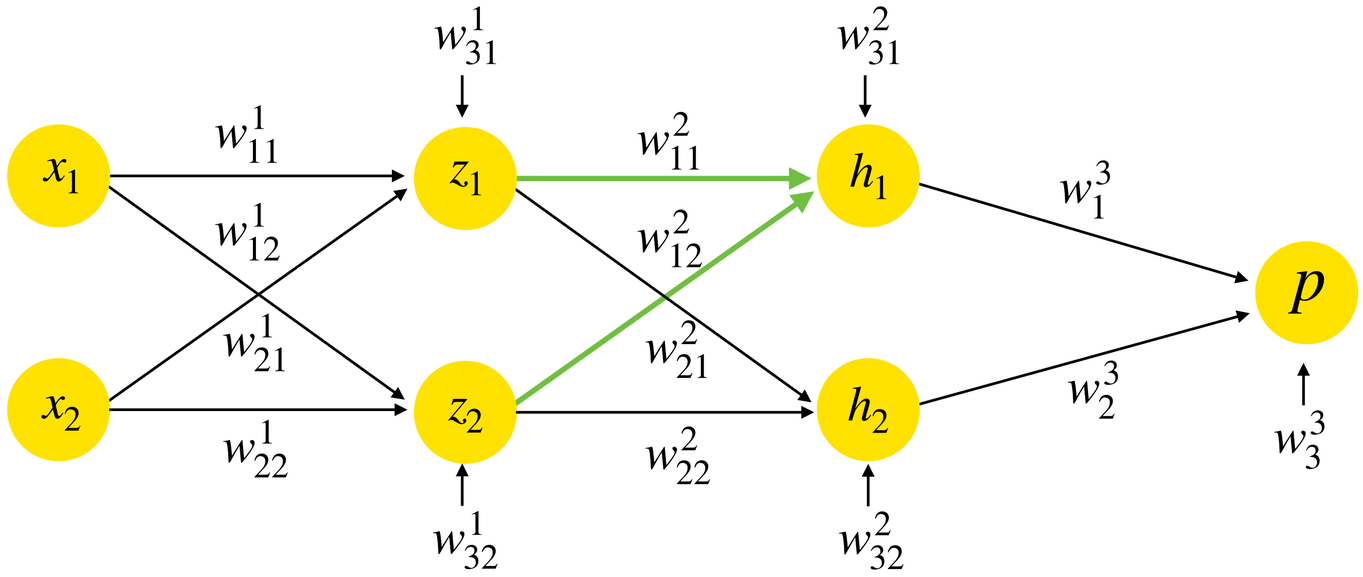
For each node, we can compute first the linear combination and then the activation function:

$$h_1^{lin} = w_{11}^2 z_1 + w_{21}^2 z_2 + w_{31}^1 \qquad h_1 = g(h_1^{lin})$$

$$h_2^{lin} = w_{12}^2 z_1 + w_{22}^2 z_2 + w_{32}^2 \qquad h_2 = g(h_2^{lin})$$

Which can be rewritten as:

$$h^{lin} = Z^T w \qquad h = g(h^{lin})$$



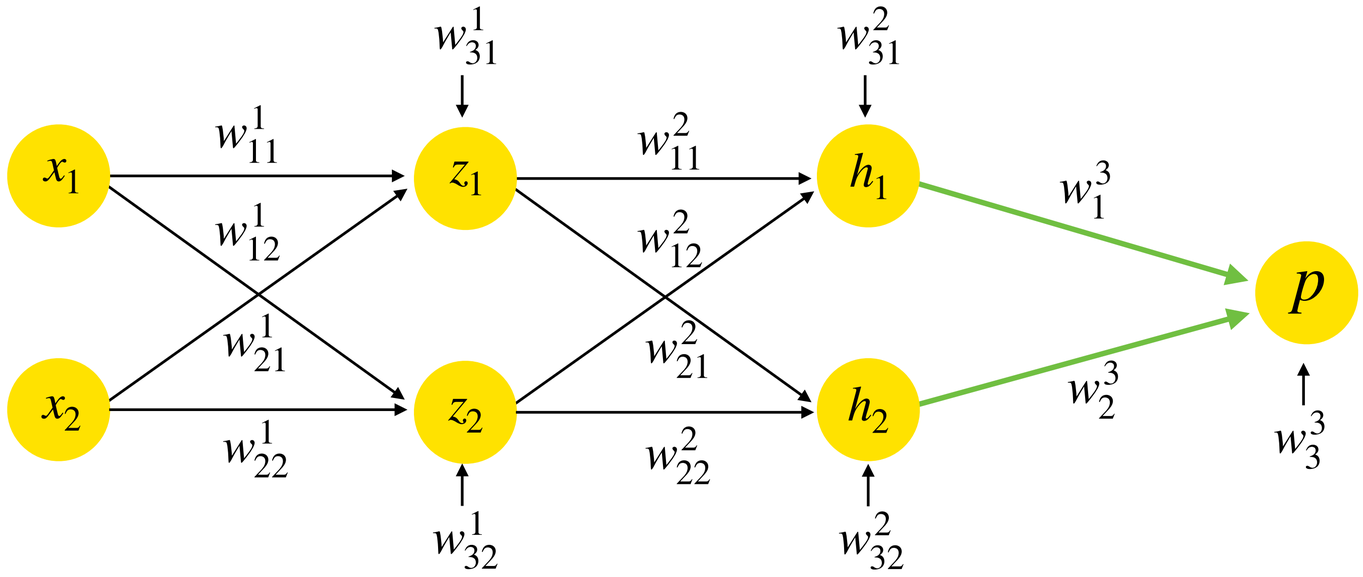
Feedforward

For each node, we can compute first the linear combination and then the activation function:

$$p^{lin} = w_1^3 h_1 + w_2^3 h_2 + w_3^3 \qquad p = g(p^{lin})$$

Which can be rewritten as:

$$\overrightarrow{p}^{lin} = \overrightarrow{h}^T w \quad p = g(p^{lin})$$



Feedforward

In conclusion, for every layer, we defined:

x [input_size + 1] - layer input
W [input_size + 1, output_size] - weight matrix
$\mathbf{z^{lin}}$ [output_size] - linear intermediate output matrix
z [output_size] - output matrix

$$\vec{z}^{lin} = \vec{x}^{T} W$$
$$\vec{z} = g(\vec{z}^{lin})$$

We can go layer by layer and use the previous output as the new input

Gradient

We already defined the feedforward pass, where for each layer we use the inputs to compute and store the linear combination in the node, and the output after the activation.

Now we need to understand the key part - how to compute the gradients so that we can update them with the gradient descent rule

Chain rule

Every node can be seen as a function of previous functions, so we can use the chain rule:

$$z(x) = z(y(x)) \qquad \frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

For example, if we're looking for the derivative of the final node with respect to one of the inputs, we just apply the chain rule:

$$p(x_1, x_2) = f(h_1(x_1, x_2), h_2(x_1, x_2))$$

$$h_1(x_1, x_2) = g_1(z_1(x_1, x_2), z_2(x_1, x_2))$$

$$h_2(x_1, x_2) = g_2(z_1(x_1, x_2), z_2(x_1, x_2))$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2}\frac{\partial h_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1}\left(\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2}\frac{\partial z_2}{\partial x_1}\right) + \frac{\partial p}{\partial h_2}\left(\frac{\partial h_2}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2}\frac{\partial z_2}{\partial x_1}\right)$$

Backpropagation

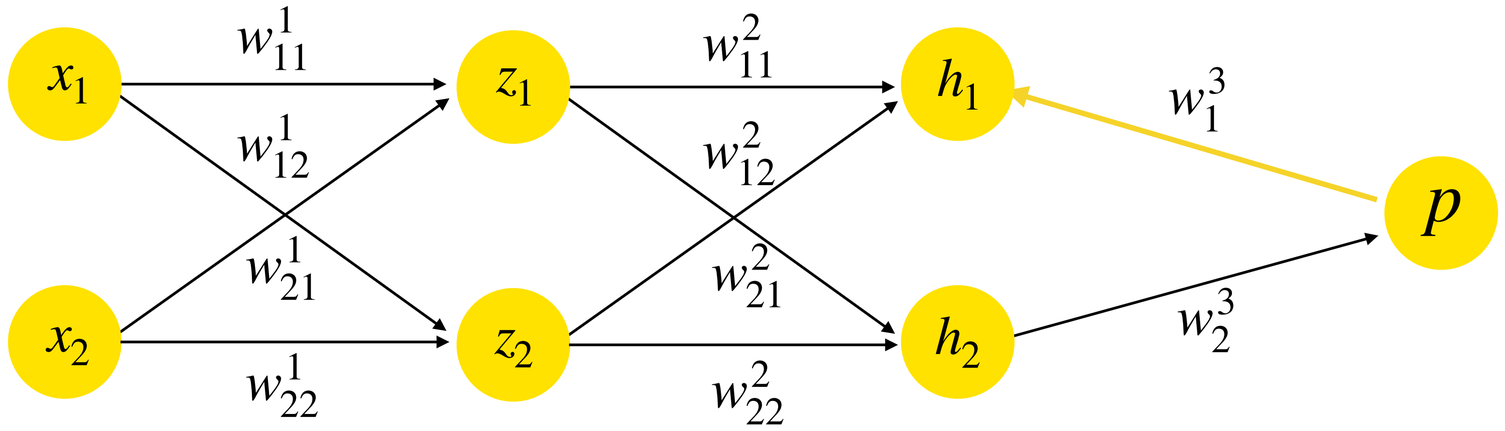
Now let's apply the chain rule to get equivalent derivatives of the loss function w.r.t the weights. We'll use cross-entropy as the loss function, where you can see the loss function as a composed of the output layer and the label y.

$$L(\vec{x}, \vec{w}, y) = L(p(\vec{x}, \vec{w}), y)$$

We can apply the chain rule in a similar manner to get derivatives with respect to the different weights, and with respect to the different nodes

We can apply the chain rule in a similar manner to get derivatives with respect to the different weights, and with respect to the different nodes

Backpropagation

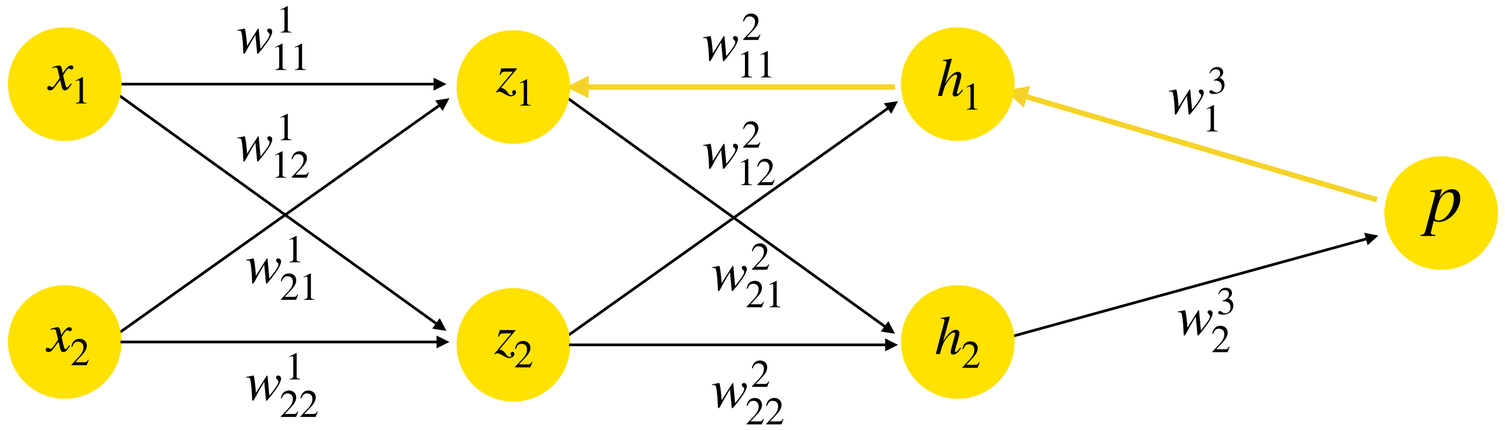


$$p(x_1, x_2) = f(h_1(x_1, x_2), h_2(x_1, x_2))$$

$$\frac{\partial L}{\partial w_1^3} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial w_1^3}$$

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial h_1}$$

Backpropagation



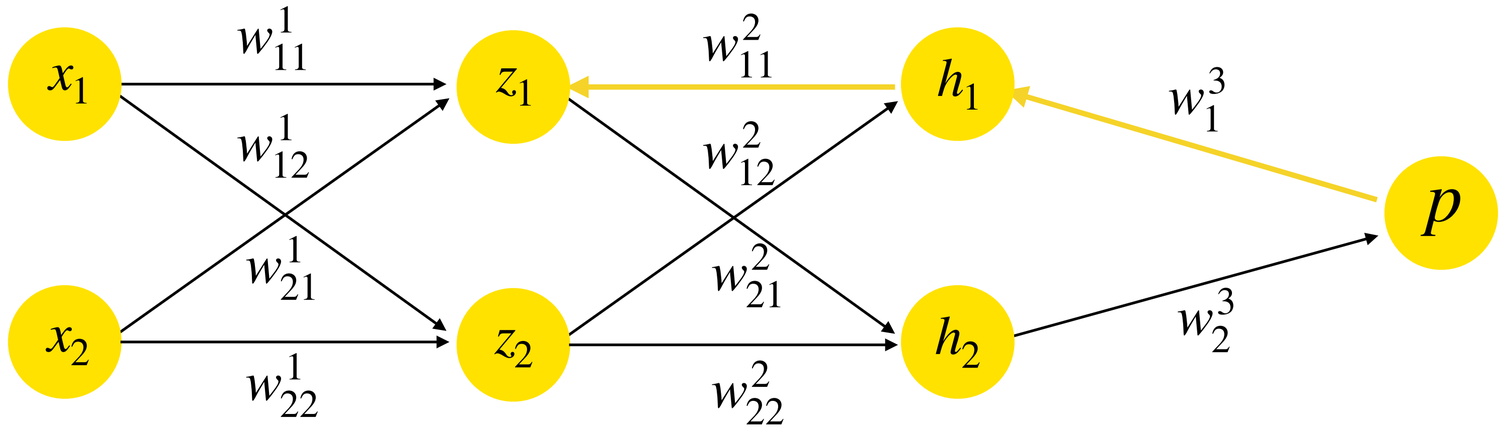
$$p(x_1, x_2) = f(h_1(x_1, x_2), h_2(x_1, x_2))$$

$$h_1(x_1, x_2) = f(z_1(x_1, x_2), z_2(x_1, x_2))$$

$$\frac{\partial L}{\partial w_{11}^2} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial w_{11}^2} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial w_{11}^2}$$

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_1} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial z_1}$$

Backpropagation



$$p(x_1, x_2) = f(h_1(x_1, x_2), h_2(x_1, x_2))$$

$$h_1(x_1, x_2) = f(z_1(x_1, x_2), z_2(x_1, x_2))$$

$$z_1(x_1, x_2) = f(x_1, x_2)$$

$$\frac{\partial L}{\partial w_{11}^1} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial h_1}\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial w_{11}^1} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial w_{11}^1}$$

Backpropagation

In conclusion, for every layer, going from the last to the first, we compute:

- A derivative with respect to the weights that feed that layer $\frac{\partial L}{\partial w^l}$
- A derivative with respect to the nodes that feed that layer, this is, the inputs $\frac{\partial L}{\partial h^l}$

We can then reuse the derivative w.r.t the input nodes to compute both in the next step:

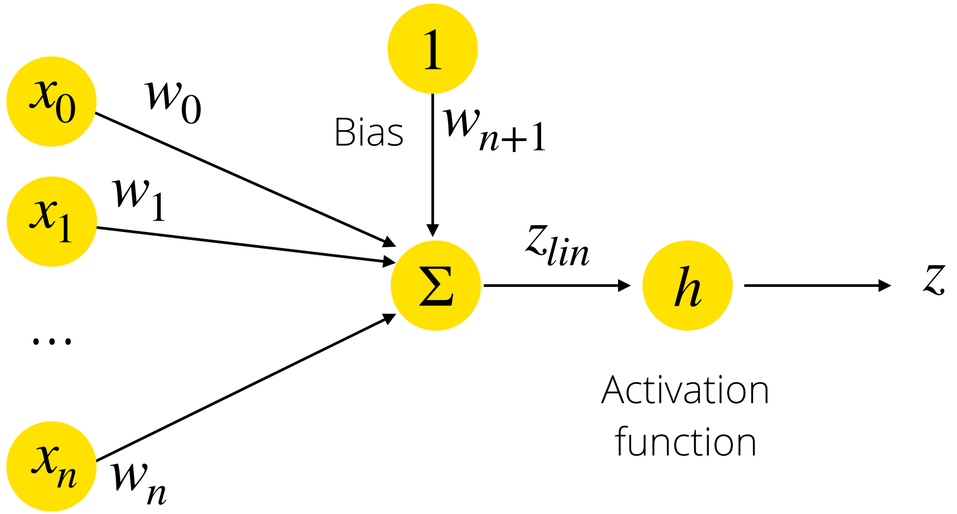
$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial w_{ij}}$$

$$\frac{\partial L}{\partial h_i} = \frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial h_i}$$

Backpropagation

$$\frac{\partial h_j}{\partial w^{ij}} = \frac{\partial z}{\partial w^{ij}} = \frac{\partial z}{\partial z_{lin}} \frac{\partial z_{lin}}{\partial w_{ij}}$$

$$\frac{\partial h_j}{\partial h_i} = \frac{\partial z}{\partial h_i} = \frac{\partial z}{\partial z_{lin}} \frac{\partial z_{lin}}{\partial h_i}$$



$$\frac{\partial z_{lin}}{\partial w_{ij}} = x_i \qquad \frac{\partial z_{lin}}{\partial h_i} = w_{ij}$$

$$\frac{\partial z}{\partial z_{lin}} = \sigma(z_{lin})(1 - \sigma(z_{lin}))$$

Depends on the activation function (Ex: sigmoid)

Backpropagation

$$\frac{\partial L}{\partial h_i} = \sum_j \left(\frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial h_i}\right) = \sum_j \left(\frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial z_{lin_j}}\frac{\partial z_{lin_j}}{\partial h_i}\right) \longrightarrow \frac{\partial L}{\partial h_i} = \sum_j \left(\frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial h_i}\right) = \sum_j \left(\frac{\partial L}{\partial z_{lin_j}} w_{ij}\right)$$

$$\frac{\partial L}{\partial z_{lin_i}} = \sigma(z_{lin_i})(1-\sigma(z_{lin_i}))\frac{\partial L}{\partial h_i} \longrightarrow \frac{\partial L}{\partial z_{lin_i}} = \sigma(z_{lin_i})(1-\sigma(z_{lin_i}))\sum_j \left(\frac{\partial L}{\partial z_{lin_j}} w_{ij}\right)$$

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial w_{ij}} = \frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial z_{lin_j}}\frac{\partial z_{lin_j}}{\partial w_{ij}} \longrightarrow \frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z_{lin_j}} x_i$$

We can only focus on computing these quantities

Backpropagation - loss function

For the two loss functions we've seen

- Mean Squared Errors $\frac{\partial L}{\partial p} = 2(y - p)$
- Cross entropy $\frac{\partial L}{\partial p} = \frac{y - p}{p(1 - p)}$

Backpropagation - loss function

For the two loss functions we've seen

- Mean Squared Errors $\frac{\partial L}{\partial p} = 2(y - p)$
- Cross entropy $\frac{\partial L}{\partial p} = \frac{p - y}{p(1 - p)}$

$$\frac{\partial L}{\partial p_{lin}} = \sigma(p_{lin})(1 - \sigma(p_{lin}))2(y - p) = 2p(1 - p)(y - p)$$

$$\frac{\partial L}{\partial p_{lin}} = \sigma(p_{lin})(1 - \sigma(p_{lin}))\frac{p - y}{p(1 - p)} = p(1 - p)\frac{p - y}{p(1 - p)} = p - y$$

Putting everything together

In conclusion, for every layer, in a reverse order, we can run (using the sigmoid as and cross entropy as an example):

Output layer:

$$\frac{\partial L}{\partial p_{lin_i}} = (y - p)$$

Hidden layers:

$$\frac{\partial L}{\partial z_{lin_i}} = \sigma(z_{lin_i})(1 - \sigma(z_{lin_i})) \sum_j (\frac{\partial L}{\partial z_{lin_j}} w_{ij})$$

Then, for all layers:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z_{lin_j}} x_i$$

x [layer_input_size + 1] - layer input
W [layer_input_size + 1, layer_output_size] - weight matrix
$\mathbf{z_{lin}}$ [layer_output_size + 1] - layer linear combination
z [layer_output_size + 1] - layer output

NOTE: remember we already had **z** and $\mathbf{z_{lin}}$ from the feedforward step

Putting everything together

1 - Run feedforward to obtain nodes linear combination and output:

For all layers:

$$\vec{z}^{lin} = \vec{x}^T W$$

$$\vec{z} = g(\vec{z}^{lin})$$

2 - Run backpropagation to get gradients

Output layer:

$$\frac{\partial L}{\partial p_{lin_i}} = (y - p)$$

Hidden layers:

$$\frac{\partial L}{\partial z_{lin_i}} = \sigma(z_{lin_i})(1 - \sigma(z_{lin_i})) \sum_j (\frac{\partial L}{\partial z_{lin_j}} w_{ij})$$

Then, for all layers:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z_{lin_j}} x_i$$

Practice time!

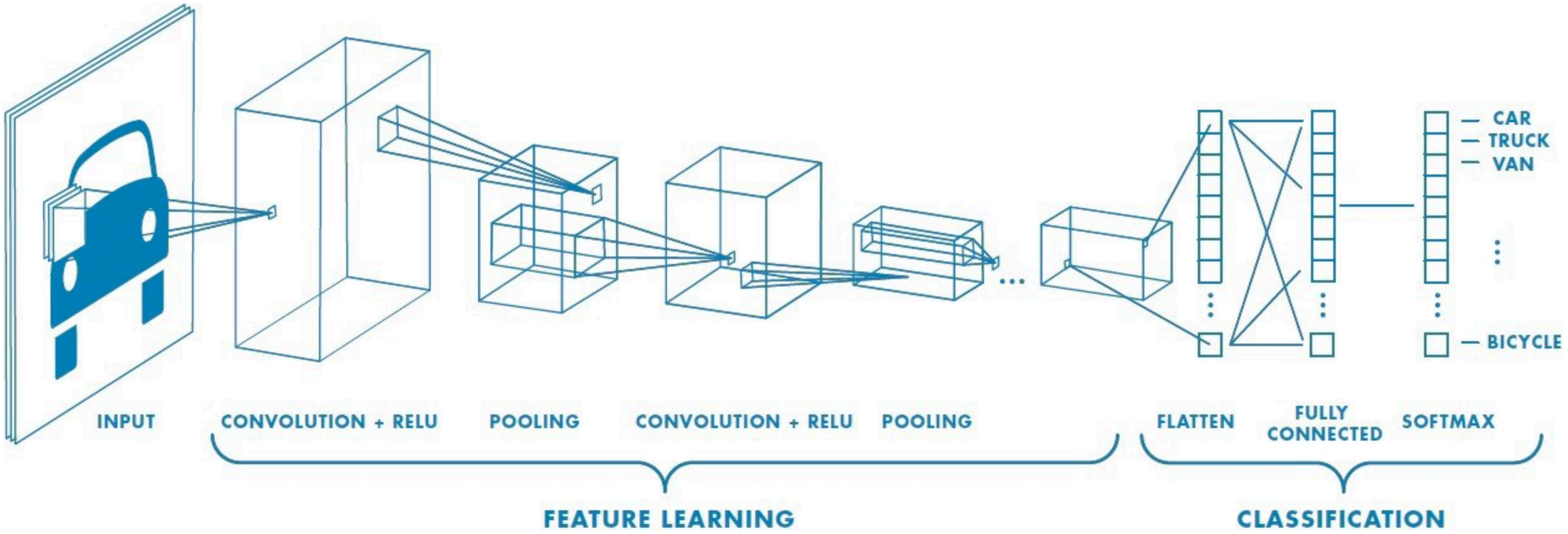
https://github.com/CatarinaSilva/meetup-deeplearning

Other topics not covered

- Batching
- Overfitting
- Validation sets
- Weight initialisation
- Regularization

Complex neural networks

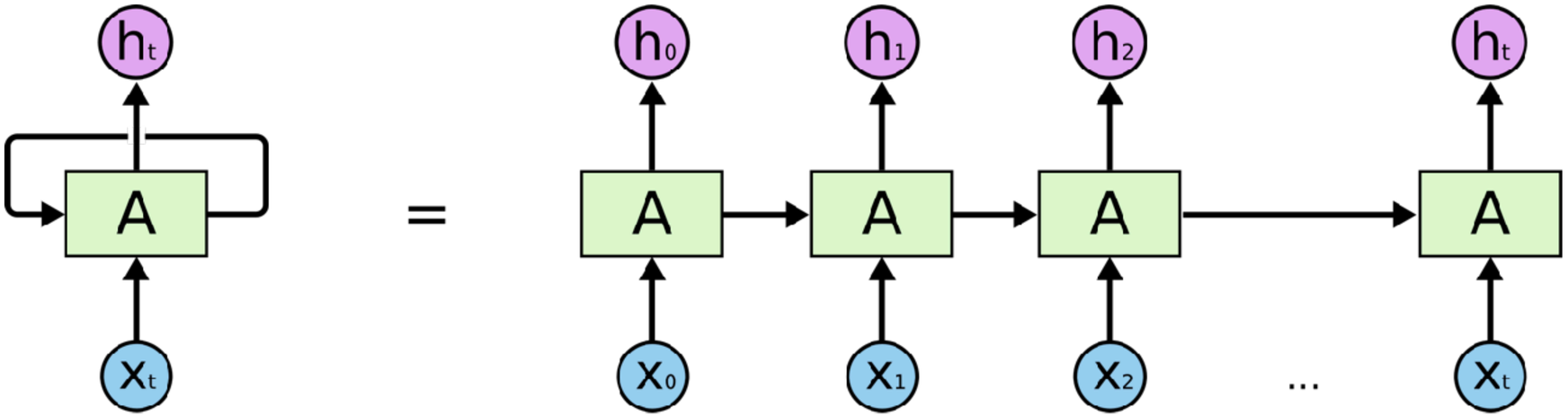
CNNs



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

CNNs

RNNs



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Some final considerations