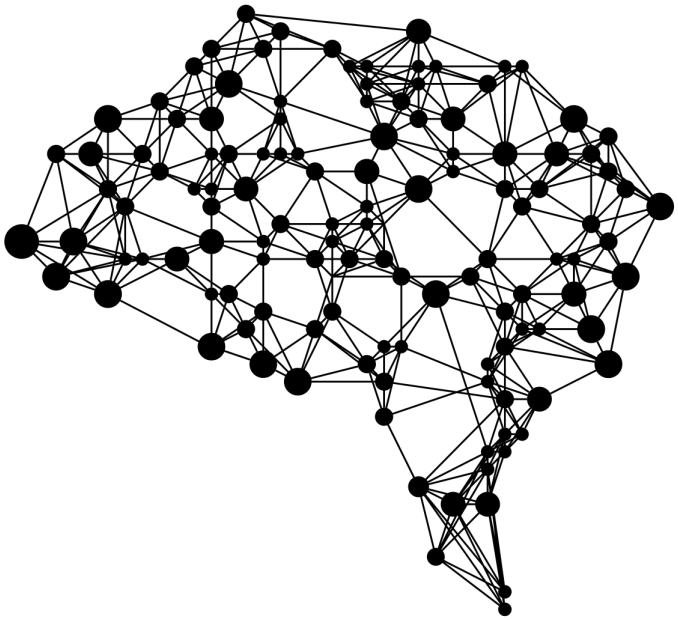
Deep Learning Notes

Andrea Mansi UniUD

I° Semester – 2021/2022



Contents

Info about the notes

Those Deep Learning notes are mainly based on the following bibliography:

- Lessons and slides from Prof. Giuseppe Serra's Deep Learning Course - Università degli Studi di Udine (http://ailab.uniud.it/).
- Coursera Online Course: Machine Learning by Andrew Ng - Stanford University click here;
- Machine Learning Glossary - click here.
- DeepLearningAI youtube CNN videos from Andrew Ng - link here.
- Andrew-NG-Notes by ashishpatel26 - link for github repository.
- Andrew-NG youtube RNN videos - link here.
- Shervine Amidi - Stanford University Notes CS229 and CS230.
- Other website sources: wikipedia, youtube videos and so on.

Notes contain, for some chapters, others links to videos and articles for further information/summaries about various topics.

Last edit: January 28, 2022

1 Introduction to Machine Learning

There are two famous definitions of Machine Learning. The first one, offered by Arthur Samuel, describe ML as:

> “The field of study that gives computers the ability to learn without being explicitly programmed.
>
> *Arthur Samuel* ”

A more modern definition, was provided by Tom Mitchell.

> “A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in T, as measured by P, improves with experience E.
>
> *Tom Mitchell* ”

For example, if we consider the game of checkers, E,T and P are the following:

- E = the experience of playing many games of checkers;
- T = the task of playing checkers;
- P = the probability that the program will win the next game.

In general, any ML algorithm problem can be assigned to one of the two broad classifications: **Supervised Learning** and **Unsupervised Learning**.

1.1 Supervised learning

In supervised learning, we are given a data set and we already know what our correct output should looks like, having the idea that there is a relationship between the input and the output. Supervised learning problems are categorized into **regression** and **classification** problems.

- In a **regression** problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. Note that some algorithms, such as logistic regression, have the name ”regression” in their names but they are not regression algorithms (example of regression algorithm: linear regression);
- In a **classification** problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories (example: logistic regression).

Examples:

- Regression problem: given a picture of a person, we have to predict their age on the basis of the given picture, age is a continuous numerical value, for exemple by fitting a linear model;
- Classification problem: given a patient with a tumor, we have to predict whether the tumor is malignant or benign, the tumor kind is a categorical value.

The **supervised learning** term refers to the fact that we use a dataset in which the right answers are given/known. We work on given examples, already "resolved" cases.

1.2 Unsupervised learning

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. The given data does not have the right answer included for each observation. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results.

Examples:

- Clustering problem: Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.
- Non-clustering problem: The "Cocktail Party Algorithm", allows you to find structure in a chaotic environment. (i.e. identifying individual voices and music from a mesh of sounds at a cocktail party).

1.3 Reinforcement Learning

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward. Although the designer sets the reward policy–that is, the rules of the game–he gives the model no hints or suggestions for how to solve the game. It's up to the model to figure out how to perform the task to maximize the reward, starting from totally random trials and finishing with sophisticated tactics and superhuman skills. By leveraging the power of search and many trials, reinforcement learning is currently the most effective way to hint machine's creativity. In contrast to human beings, artificial intelligence can gather experience from thousands of parallel gameplays if a reinforcement learning algorithm is run on a sufficiently powerful computer infrastructure.

1.4 Generative Learning

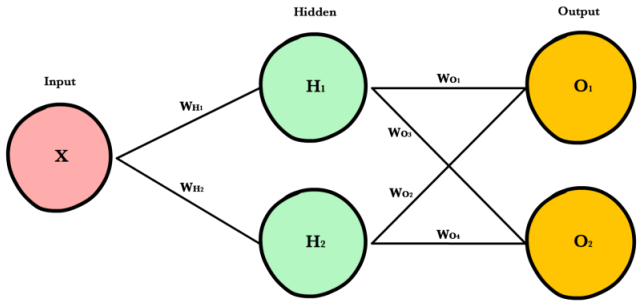
Generative learning is a learning theory that involves actively integrating new ideas with what the learner already knows. In other words, incorporating existing knowledge with new information based on open-mindedness and experimentation. For learners to understand what they learn, they have to construct meaning actively. Example: GANs "Generative Adversarial Networks". GANs are generative models: they create new data instances that resemble your training data.

2 Neural Networks

After the short summary about the concept of "machine-learning", given in the first chapter, we will now dive deep into the concept of neural networks. In this chapter main concepts are revised and some math basis are given. Following chapters will focus on some specific aspects, giving more details and a more clear explanations.

2.1 Introduction

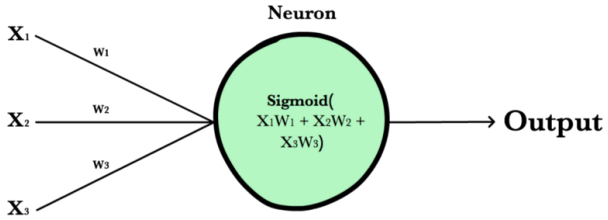
Neural networks are a class of machine learning algorithms used to model complex patterns in datasets using multiple hidden layers and non-linear activation functions. A neural network takes an input, passes it through multiple layers of hidden neurons (mini-functions with unique coefficients that must be learned), and outputs a prediction representing the combined input of all the neurons.



Neural networks are trained iteratively using optimization techniques like gradient descent. After each cycle of training, an error metric is calculated based on the difference between prediction and target. The derivatives of this error metric are calculated and propagated back through the network using a technique called backpropagation. Each neuron's coefficients (weights) are then adjusted relative to how much they contributed to the total error. This process is repeated iteratively until the network error drops below an acceptable threshold.

2.2 Main concepts and terminologies

Neuron: A neuron takes a group of weighted inputs, applies an activation function, and returns an output.



Inputs to a neuron can either be features from a training set or outputs from a previous layer's neurons. Weights are applied to the inputs as they travel along synapses to reach

the neuron. The neuron then applies an activation function to the "sum of weighted inputs" from each incoming synapse and passes the result on to all the neurons in the next layer.

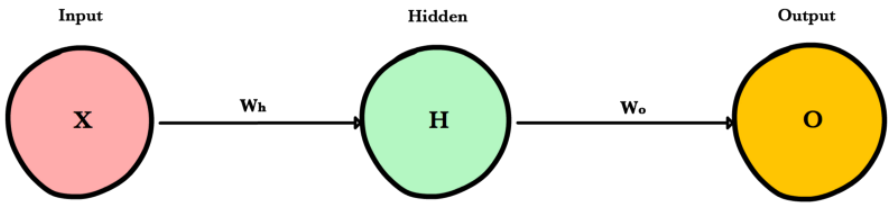
Synapse: Synapses are like roads in a neural network. They connect inputs to neurons, neurons to neurons, and neurons to outputs. In order to get from one neuron to another, you have to travel along the synapse paying the "toll" (weight) along the way. Each connection between two neurons has a unique synapse with a unique weight attached to it. When we talk about updating weights in a network, we're really talking about adjusting the weights on these synapses.

Weights: Weights are values that control the strength of the connection between two neurons. That is, inputs are typically multiplied by weights, and that defines how much influence the input will have on the output. In other words: when the inputs are transmitted between neurons, the weights are applied to the inputs along with an additional value (the bias).

Bias: Bias terms are additional constants attached to neurons and added to the weighted input before the activation function is applied. Bias terms help models represent patterns that do not necessarily pass through the origin. For example, if all your features were 0, would your output also be zero? Is it possible there is some base value upon which your features have an effect? Bias terms typically accompany weights and must also be learned by your model.

2.2.1 Layers:

There are three types of layers in a neural network:



Input Layer: Holds the data your model will train on. Each neuron in the input layer represents a unique attribute in your dataset (e.g. height, hair color, etc.).

Hidden Layer: Sits between the input and output layers and applies an activation function before passing on the results. There are often multiple hidden layers in a network. In traditional networks, hidden layers are typically fully connected layers; each neuron receives input from all the previous layer's neurons and sends its output to every neuron in the next layer. This contrasts with how convolutional layers work where the neurons send their output to only some of the neurons in the next layer.

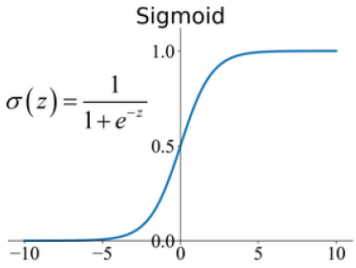
Output Layer: The final layer in a network. It receives input from the previous hidden layer, optionally applies an activation function, and returns an output representing your model's prediction.

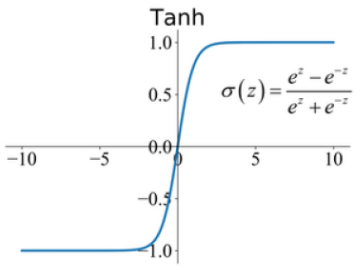
2.2.2 Activation Functions: Activation functions live inside neural network layers and modify the data they receive before passing it to the next layer. Activation functions give neural networks their power; allowing them to model complex non-linear relationships. By modifying inputs with non-linear functions neural networks can model highly complex relationships between features. Activation functions typically have the following properties:

- **Non-linear:** In linear regression we're limited to a prediction equation that looks like a straight line. This is nice for simple datasets with a one-to-one relationship between inputs and outputs. If the patterns in our dataset are non-linear, to model these relationships we need a non-linear prediction equation. Activation functions provide this non-linearity.
- **Continuously differentiable:** To improve our model with gradient descent, we need our output to have a nice slope so we can compute error derivatives with respect to weights. If our neuron instead outputted 0 or 1 (perceptron), we wouldn't know in which direction to update our weights to reduce our error.
- **Fixed Range:** Activation functions typically squash the input data into a narrow range that makes training the model more stable and efficient.

Two of the most historically used activation functions are: **Sigmoid Function** and **Tanh Function**.

Sigmoid function has seen frequent use historically, yet now it is rarely used, because it kills the gradient. It is commonly used in the last layer if you want a prediction between 0 and 1. The hyperbolic tangent activation function (Tanh) tipically performs better than the logistic sigmoid.

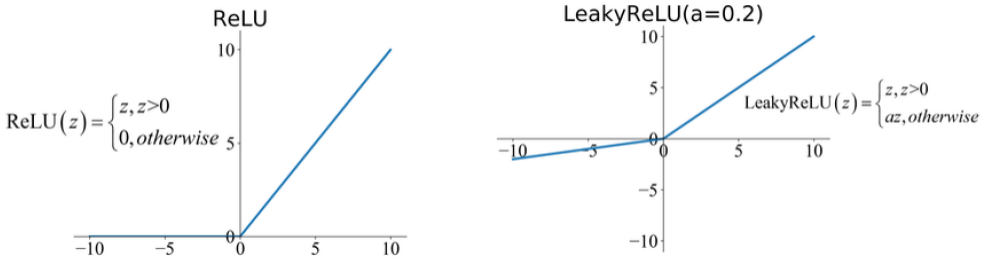




Another two commonly used activation functions are: **ReLU** and **LeakyReLU**.

ReLU has become the default activation function for many types of neural networks, because a model that uses it is easier to train and often achieves better performance. The derivative of the rectified linear function is also easy to calculate. The slope for negative values is 0.0 and the slope for positive values is 1.0. Derivative in zero is not defined (empirically is not a problem).

ReLU is affected by the Dying ReLU Problem: some ReLU Neurons essentially die for all inputs and remain inactive no matter what input is supplied, here no gradient flows (the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn - because the derivate is not defined). LeakyReLU fixs this problem. Empirically, the performance is similar to the ReLU.



2.2.3 Cost Functions: As already seen, a Cost function (or Loss function) is a wrapper around our model's predict function that tells us "how good" the model is at making predictions for a given set of parameters. The loss function has its own curve and its own derivatives. The slope of this curve tells us how to change our parameters to make the model more accurate. We use the model to make predictions. We use the cost function to update our parameters. Our cost function can take a variety of forms as there are many different cost functions available. There are a lot of popular generic loss/cost functions.

A very useful set of YouTube videos about "What is DL", by 3Blue1Brown can be found at this link.

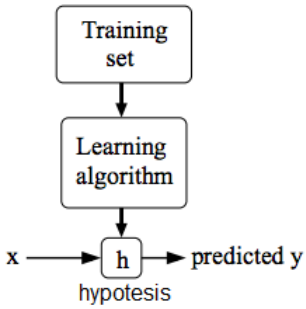
3 Model and Cost Function

3.1 Model Representation

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables, also called **input features** and $y^{(i)}$ to denote the "output" or **target variable** that we are trying to predict.

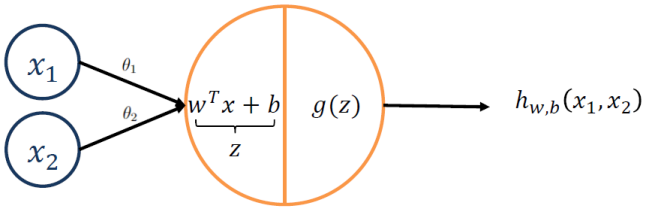
A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn (a list of m training examples) is called a **training set**. We will also use X to denote the space of input values, and Y to denote the space of output values.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : X \to Y$ so that $h(x)$ is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a **hypothesis**, but now is tipically called **model**. Seen pictorially, the process is therefore like this:



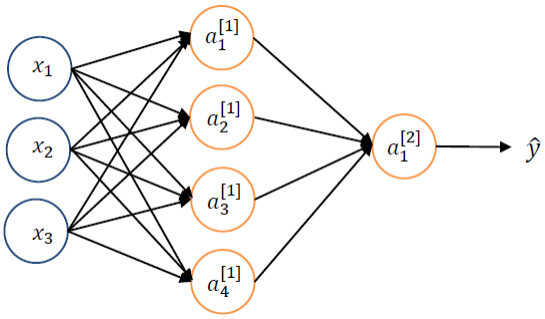
As already said, when the target variable that we're trying to predict is continuous, we call the learning problem a regression problem. When y can take on only a small number of discrete values, we call it a classification problem.

A graphical representation of a simple neural network can be the following one:



The represented network means that in order to compute the value of the decision function, we need to multiply the first input x_1 with the first weight θ_1, second input x_2 with the second weight θ_2 and then add the two values with b (bias), then apply the activation function (sigmoid function $g(z)$ in this example).

Consider now the following neural network:



We can recap how we denote values and parameters as follow:

- $W^{[j]}$ = matrix of weights controlling function mapping from layer $j-1$ to layer j;
- $b^{[j]}$ = bias, layer j;
- $a_k^{[j]}$ = activation of vector for layer j, node k;
- $g_k^{[j]}$ = activation function for layer j, node k.

3.1.1 Forward propagation - Vectorized Implementation: We will now see how forward propagation is performed in a network. A forward propagation means providing the input to the network and let it propagate through the layers until the output is given. In the given example we have that:

$$
\begin{aligned}
a_1^{[1]} &= g_1^{[1]}\left(W_{11}^{[1]}x_1 + W_{12}^{[1]}x_2 + W_{13}^{[1]}x_3\right)\\
a_2^{[1]} &= g_2^{[1]}\left(W_{21}^{[1]}x_1 + W_{22}^{[1]}x_2 + W_{23}^{[1]}x_3\right)\\
a_3^{[1]} &= g_3^{[1]}\left(W_{31}^{[1]}x_1 + W_{32}^{[1]}x_2 + W_{33}^{[1]}x_3\right)\\
a_4^{[1]} &= g_4^{[1]}\left(W_{41}^{[1]}x_1 + W_{42}^{[1]}x_2 + W_{43}^{[1]}x_3\right)
\end{aligned}
$$

and: $a_1^{[2]} = g_1^{[2]}(W_{11}^{[2]}a_1^{[1]} + W_{12}^{[2]}a_2^{[1]} + W_{13}^{[2]}a_3^{[1]} + W_{14}^{[2]}a_4^{[1]})$.

We'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{[j]}$ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

$$
\begin{aligned}
a_1^{[1]} &= g_1^{[1]}\left(z_1^{[1]}\right)\\
a_2^{[1]} &= g_2^{[1]}\left(z_2^{[1]}\right)\\
a_3^{[1]} &= g_3^{[1]}\left(z_3^{[1]}\right)\\
a_4^{[1]} &= g_4^{[1]}\left(z_4^{[1]}\right)
\end{aligned}
$$

Where for layer $j > 1$ and node k, the variable z will be:

$$z_k^{[j]} = W_{k,1}^{[j]} a_1^{[j-1]} + \cdots + W_{k,n}^{[j]} a_n^{[j-1]}$$

If $j = 1$ then $a^{[1]} = x$, so:

$$z_k^{[1]} = W_{k,1}^{[1]} x_1 + \cdots + W_{k,n}^{[1]} x_n$$

The vector representation of x and $z^{[j]}$ is:

$$x = \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix} z^{[j]} = \begin{bmatrix} z_1^{[j]} \\ z_2^{[j]} \\ \cdots \\ z_n^{[j]} \end{bmatrix}$$

We can get a vector of our activation nodes for layer j as follow:

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

Calculus examples: Remember that $z = w^T x + b$, given input X, we have some calculus example:

$$\mathrm{z}^{[1]} = W^{[1]} x + b^{[1]}$$

with those dimensionality:

- z = (4,1)
- $W^{[1]}$ = (4,3)
- x = (3,1)
- $b^{[1]}$ = (4,1)

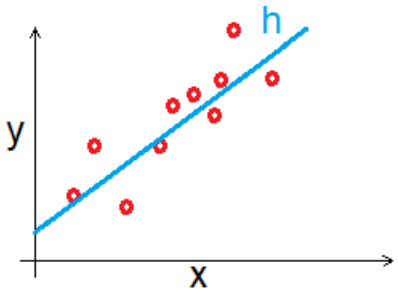
Another three examples:

- $\mathrm{a}^{[1]} = g^{[1]}\left(z^{[1]}\right)$
- $\mathrm{z}^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
- $\mathrm{a}^{[2]} = g^{[2]}\left(z^{[2]}\right)$

3.2 Cost Function

Let's start by introducing the linear regression model, which will be then used to describe the concepts behind the Cost/Loss function. In linear regression, we have a training set, like maybe the one in the image. What we want to do, is come up with values for the parameters θ_0 and θ_1 so that the straight line we get out of this, corresponds to a straight line that somehow fits the data well, like maybe that line over there.

$$h_\theta(x) = \theta_0 + \theta_1 x$$



The idea is that we get to choose our parameters θ_0, θ_1 so that $h_\theta(x)$, meaning the value we predict on input x, is at least close to the values y for the examples in our training set, for our training examples.

The objective is to minimize this value (so find the values of θ_0, θ_1 so that we minimize this value):

$$J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

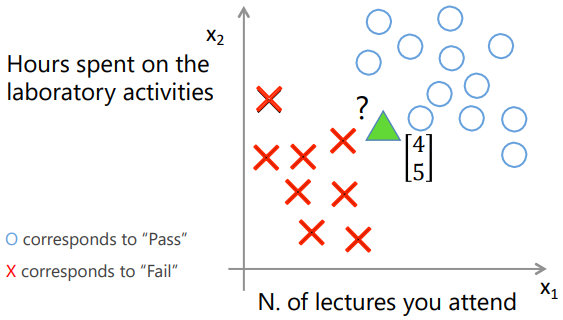
This function is otherwise called the **"Squared error function"** or **"Mean squared error"**.

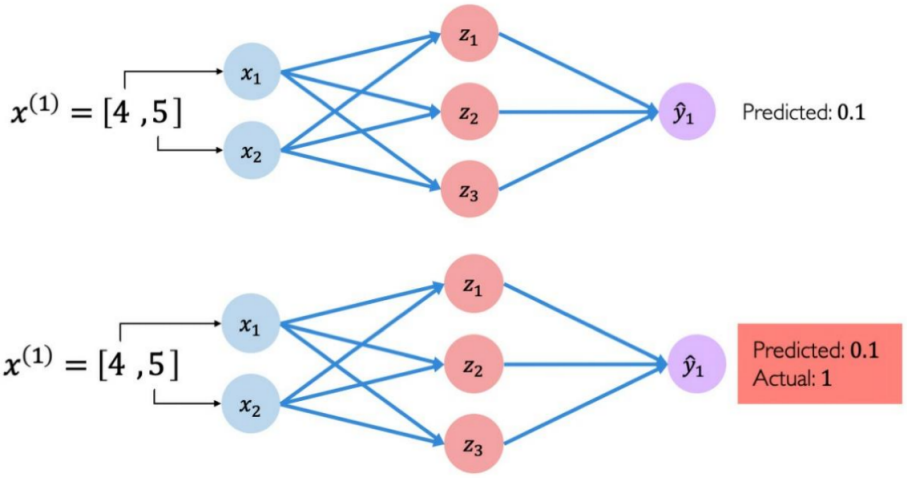
The cost function is what we use for evaluating "the performance of our model". It takes both predicted outputs by the model and actual outputs and calculates how much wrong the model was in its prediction. It outputs a higher number if our predictions differ a lot from the actual values.

As summary: we have an hypotesis function $h_\theta(x)$, with two parameters: θ_0, θ_1. Our goal is to find the values of those two paramets so that the cost function J is minimized.

> "In conclusion, what we want is to have software to find the value of θ_0, θ_1 that minimizes this cost function."

Let's now see the same concepts but in a classification problem, with some illustrations. We have a simple model with 2 features: x_1=number of lectures attended and x_2 hours spent on the lab. activities. We ask the model if the student in input will or will not pass the class.



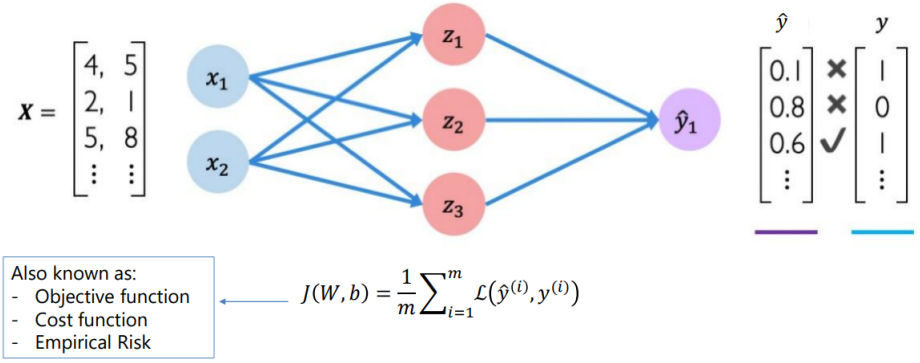


$$\mathcal{L}(\hat{y}, y)$$

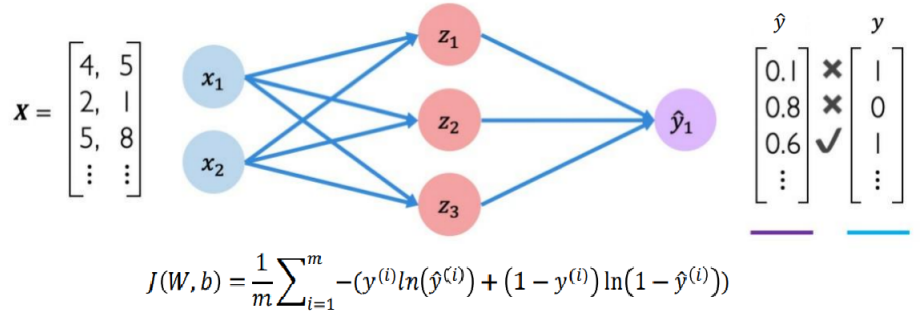
Predicted Actual

The Loss of our network measures the cost incurred from incorrect predictions

The **empirical Loss** measures the total loss over the entire dataset, as shown in the following image.



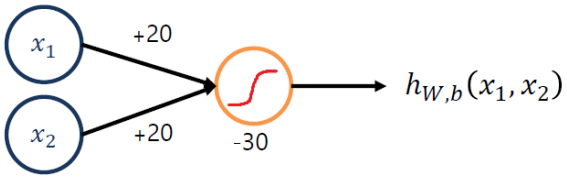
The **Binary Cross Entropy Loss** can be used with models that output a probability between 0 and 1.



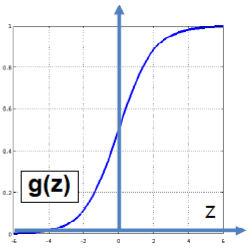
Example: Now, with more notions about model and cost function representation we can see a very simple practical example of a minimal neural network for the logic port AND.

$$y = x_1 \wedge x_2$$

$$h_{w,b}(x_1, x_2) = g(-30 + 20x_1 + 20x_2)$$



x_1	x_2	$h_{w,b}(x_1, x_2)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$



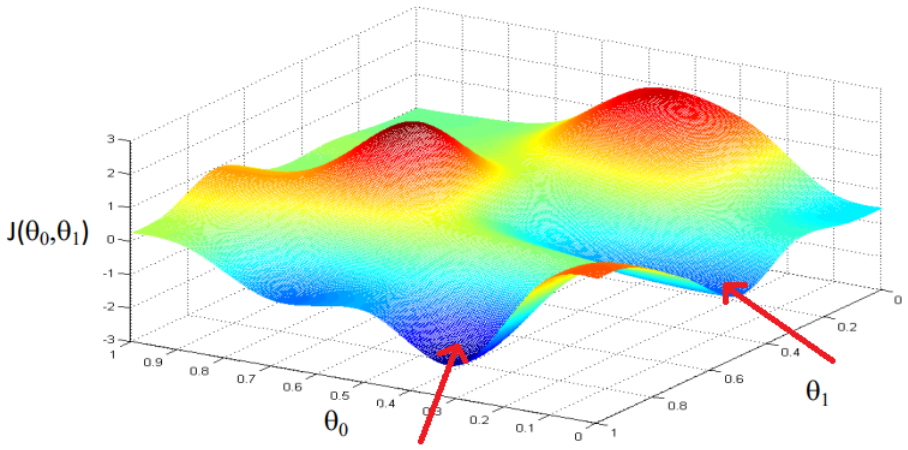
4 Gradient Descent

So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function. That's where gradient descent comes in, if the function is derivable.

IDEA: starts with some (random parameters) θ_0, θ_1, keep chaning them to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minium.

Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

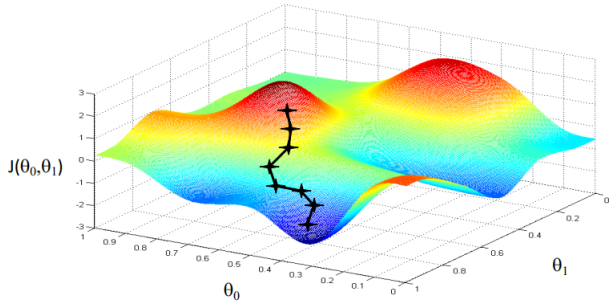
We put θ_0 on the x axis and θ_1 on the y axis with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. The graph below depicts such a setup.



We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum (local ones) points in the graph.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, which is called the **learning rate**.

Example: the distance between each "star" in the graph below represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step.



The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$. Depending on where one starts on the graph, one could end up at different points. The image above shows us two different starting points that end up in two different places.

The gradient descent algorithm is:

$$\text{repeat until convergence:} \quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

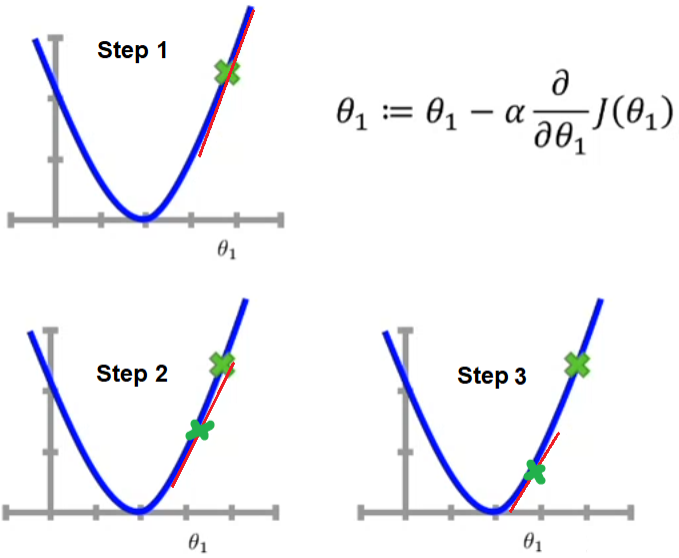
where := means variable assignment and with $j = 0, 1$ representing the features index.

- α is called **learning rate**;
- $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ is called derivate term.

Implementation note: At each iteration j, one should simultaneously update the parameters θ_i. Updating a specific parameter prior to calculating another one yield to a wrong implementation.

Correct: Simultaneous update	Incorrect:
$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$	$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$
$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$	$\theta_0 := \text{temp0}$
$\theta_0 := \text{temp0}$	$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$
$\theta_1 := \text{temp1}$	$\theta_1 := \text{temp1}$

Example: Suppose you have a very simple cost function: $J(\theta_1)$. The intuition is that for each iteration we check the tangent of θ_1 (red lines) and we do a step of length multiplied by α.



$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

In the final step we will reach the local minimum value (convergence reached).

Choosing α: technically we can fall in one of two main cases:

- α too small: gradient descent can be very slow in convergence to the minimum value;
- α too big: we take huge steps for each iteration so there is the risk to overshoot the minimum. The algorithm may fail to converge, or even diverge.

Note that gradient descent can converge to a local minimum, even with the learning rate α fixed. As we approach a local minimum, gradient descent will automatically take smaller steps because the derivate term begins to be smaller (because it converge to 0 while moving in the direction of the local minimum).

4.1 Gradiend descent for Linear Regression

We're going to put together gradient descent algorithm with our linear regression cost function. This will give us an algorithm for linear regression or "putting a straight line to our data".

When specifically applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to:

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x_i \right) - y_i \right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\left(h_\theta \left(x_i \right) - y_i \right) x_i \right)$$

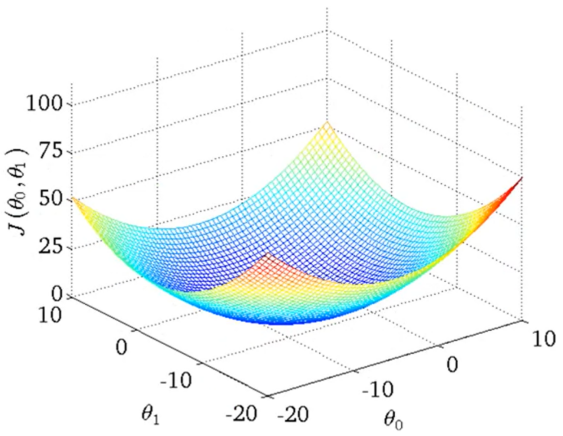
}

where m is the size of the training set, θ_0 is a constant that will be changing simultaneously with θ_1 and x_1, y_1 are values of the given training set.

Note that we have separated out the two cases for θ_0 and θ_1 into separate equations.

The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

Note also that the cost function for linear regression will always be a convex function like this:



so there is no risk to end up in two different local minimums like the previous example. In linear regression cases, gradient descent end up always in the global minimum.

"Batch" Gradient Descent: Each step of gradient descent uses all the training examples (sum from i to m), otherwise sub-sets are used, called mini-batches.

5 Logistic Regression

Remember that logistic regression is a classification problem, even if the name contains the word regression. When talking about classification problem, we can distinguish them by counting the number of variables (classes):

- $= 2$ variables: binary classification problem;
- ≥ 2 variables: multiclass classification problem.

Intuitively, it also doesn't make sense for the hypotesis function $h_{w,b}(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$. To do this, the form for our hypotheses $h_{w,b}(x)$ must satisfy $0 \leq h_{w,b}(x) \leq 1$. This is accomplished by plugging it into the **Logistic Function (Sigmoid Function)**.

$$\text{Hypotesis:} \quad h_{w,b}(x) = g\left(w^T x + b\right)$$

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

$$\text{Sigmoid function:} \quad g(z) = \frac{1}{1 + e^{-z}}$$

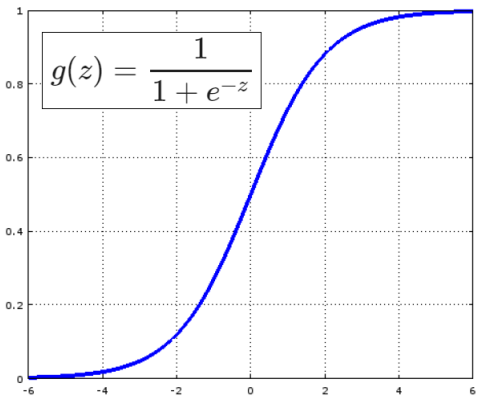


Figure 1: Sigmoid Function

The function $g(z)$, shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification. $h_{w,b}(x)$ will give us the probability that our output is 1.

So, with logistic regression, the goal is to find the parameters W, b of the hypotesis function.

5.1 Decision Boundaries

In order to get our discrete 0 or 1 classification, we can translate the output of the hypotesis function as follows:

$$h_{w,b}(x) \geq 0.5 \rightarrow y = 1$$
$$h_{w,b}(x) < 0.5 \rightarrow y = 0$$

The way our sigmoid (logistic) function g behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5:

$$g(z) \geq 0.5 \quad when \quad z \geq 0$$

Note that:

$$z = 0, e^0 = 1 \Rightarrow g(z) = 0.5$$
$$z \rightarrow \infty, e^{-\infty} \rightarrow 0 \Rightarrow g(z) = 1$$
$$z \rightarrow -\infty, e^{\infty} \rightarrow \infty \Rightarrow g(z) = 0$$

5.1.1 Example:

let's consider a 2D case in which:

$$h_{w,b}(x) = g(b + w_1 x_1 + w_2 x_2)$$

Using the logistic regression algorithm we will get, for example, those values:

$$b = 3 \text{ and } w = [1, 2]$$

Then the decision boundary is:

$$h_{w,b}(x) = 0.5 \mapsto w^T x + b = 0 \mapsto -3 + x_1 + 2x_2 = 0$$

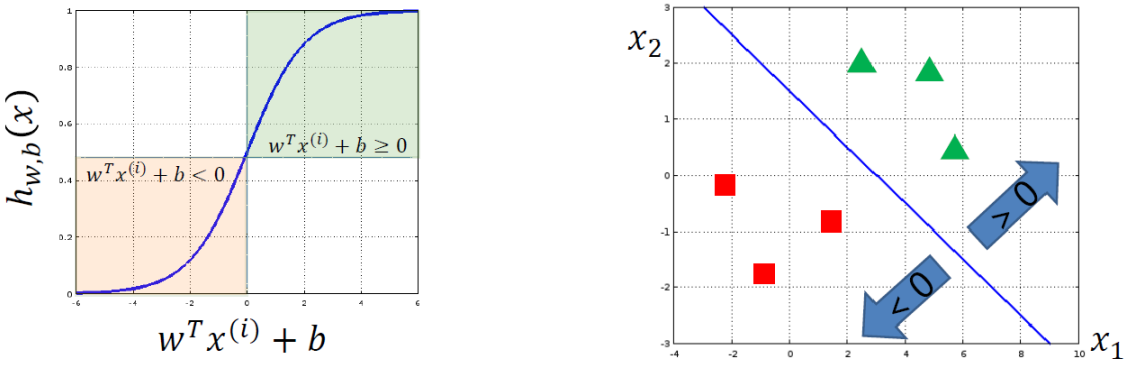


Figure 2: Left: sigmoid function where = 1 or = 0. Right: plot of the boundary example

In conclusion, if our input for the sigmoid function is $W, b^T X$, then that means:

$$h_{w,b}(x) = g\left(w^T x\right) \geq 0.5$$
$$\text{when} \quad w^T x \geq 0$$

From these statements we can now say:

$$w^T x \geq 0 \Rightarrow y = 1$$
$$w^T x < 0 \Rightarrow y = 0$$

The decision boundary is the line that separates the area where $y = 0$ and where $y = 1$. It is created by our hypotesis function.

5.2 Cost Function

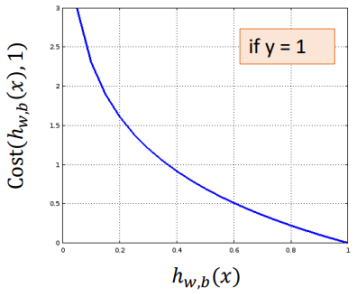
We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function. Instead, our cost function for logistic regression looks like:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}\left(h_{w,b}\left(x^{(i)}\right), y^{(i)}\right)$$

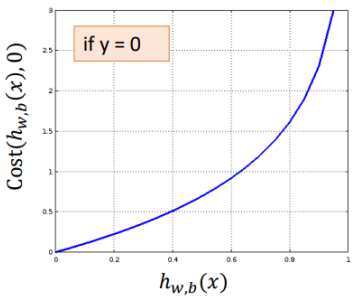
where Cost is equal to:

$$\text{Cost}\left(h_{w,b}(x), y\right) = -\log\left(h_{w,b}(x)\right) \qquad \text{if } y = 1$$
$$\text{Cost}\left(h_{w,b}(x), y\right) = -\log\left(1 - h_{w,b}(x)\right) \qquad \text{if } y = 0$$

When y=1, we get the following plot for $J(w,b)$ vs $h_{w,b}(x)$:



When y=0, we get the following plot for $J(w,b)$ vs $h_{w,b}(x)$:



We can summarize our cost function for the logistic regression as follow:

$$\begin{aligned}&\text{Cost}\left(h_\theta(x), y\right) = 0 \text{ if } h_\theta(x) = y\\ &\text{Cost}\left(h_\theta(x), y\right) \to \infty \text{ if } y = 0 \text{ and } h_\theta(x) \to 1\\ &\text{Cost}\left(h_\theta(x), y\right) \to \infty \text{ if } y = 1 \text{ and } h_\theta(x) \to 0\end{aligned}$$

5.3 Simplified (compressed) Logistic Regression cost function

We can compress our cost function of two conditional cases into one case (for the generic parameter θ which is (w, b) in our case):

$$\text{Cost}\left(h_\theta(x), y\right) = -y \log\left(h_\theta(x)\right) - (1 - y) \log\left(1 - h_\theta(x)\right)$$

Notice that:

- when y is equal to 1, then the second term $(1 - y) \log\left(1 - h_\theta(x)\right)$ will be zero and will not affect the result;
- if y is equal to 0, then the first term $-y \log\left(h_\theta(x)\right)$ will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right]$$

Now we need a procedure to minimize the cost function and get the parameteres optimal values: Gradient Descent.

5.4 Gradient Descent for Logistic Regression

Remember that the general form of gradient descent is:

$$\text{Repeat}\{\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)\}$$

By working out the derivative part using calculus we get:

$$\text{Repeat}\{\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}\}$$

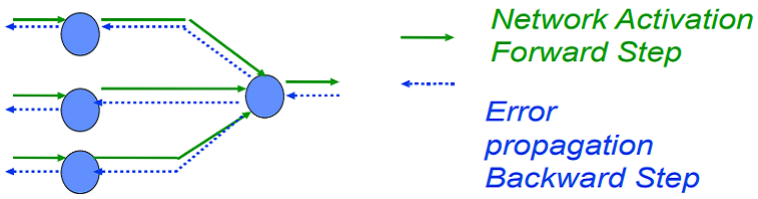
Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta. Obviously the algorithm is not the same because the hypotesis function is different.

6 Training a Neural Network

Neural networks are trained via a back-propagation algorithm. This algorithms consists of the repeadted application of the following two steps:

- **Forward step**: in this step the network is activated on one example and the error (of each neuron) of the output layer is computed.
- **Backward step**: in this step the network error is used for updating the weights. Starting at the output layer, the error is propagated backwards through the network, layer by layer. This is done by recursively computing the local gradient of each neuron.

A graphical representation of the back-propagation algorithm:



6.1 Gradient computation: Backpropagation Algorithm

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$$\min_{\Theta} J(\Theta)$$

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J(Θ):

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$

To do so, we use the following algorithm:

Given training set $\left\{\left(x^{(1)}, y^{(1)}\right) \cdots \left(x^{(m)}, y^{(m)}\right)\right\}$

set $\Delta_{i,j}^{(l)} := 0$ for all $(1, i, j)$, (hence you end up having a matrix full of zeros)

For training example $t = 1$ to m

(1): set $a^{(1)} := x^{(t)}$

(2): Perform forward propagation to compute $a^{(l)}$ for I $= 2, 3, \ldots,$ L

(3): Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$.

Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

(4): Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = \left(\left(\Theta^{(l)}\right)^T \delta^{(l+1)}\right) \cdot * a^{(l)} \cdot * \left(1 - a^{(l)}\right)$.

The delta values of layer I are calculated by multiplying the delta values in the next layer with the theta matrix of layer I. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$. The g-prime derivative terms can also be written out as: $g'\left(z^{(l)}\right) = a^{(l)} \cdot * \left(1 - a^{(l)}\right)$

(5): $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^T$ Hence we update our new Δ matrix.

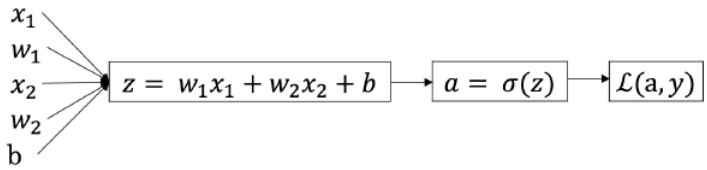
$$D_{i,j}^{(l)} := \frac{1}{m}\left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)}\right), \text{ if } j \neq 0$$

$$D_{i,j}^{(l)} := \frac{1}{m}\Delta_{i,j}^{(l)} \text{ if } j = 0$$

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

6.1.1 Computing Gradients on Logistic Regression

Let's start with a graphical rapresentation:



Remember that in Logistic Regression we have:

$z = w^T x + b$

$a = \sigma(z)$

$\text{Cost}(a, y) = \mathcal{L}(a, y) = -(y \ln(a) + (1 - y) \ln(1 - a))$

Derivatives terms are:

$\frac{d\mathcal{L}}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{da}\frac{da}{dz} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)(a(1-a)) = a - y$

$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{dw_1} = (a - y)x_1$

$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{dw_2} = (a-y)x_2$

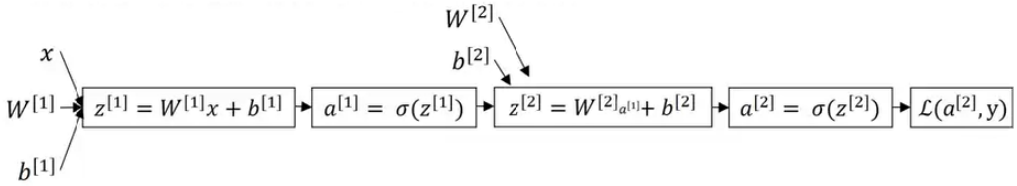
$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{da}\frac{da}{dz}\frac{dz}{db} = (a-y)$

And remember that, for Logistic Regression the Gradient Descent Algorithm is:

$$w_1 := w_1 - \alpha\frac{dJ(w,b)}{dw_1}$$
$$w_2 := w_2 - \alpha\frac{dJ(w,b)}{dw_2}$$
$$b := b - \alpha\frac{dJ(w,b)}{db}$$

6.1.2 Example: Neural Network for Binary Clasification

Let's suppose to design a Neural Network for Binary Classification:



Parameters: $W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}$

Cost Function: $\mathrm{J}\left(W^{[1]}, b^{[1]}W^{[2]}, b^{[2]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$

$\mathcal{L}(\hat{y}, y) = \mathcal{L}\left(a^{[2]}, y\right) = -\left(y\ln\left(a^{[2]}\right) + (1-y)\ln\left(1-a^{[2]}\right)\right)$

Forward propagation step:

$\mathrm{z}^{[1]} = W^{[1]}x + b^{[1]}$

$\mathrm{a}^{[1]} = g^{[1]}\left(z^{[1]}\right)$

$\mathrm{z}^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

$\mathrm{a}^{[2]} = g^{[2]}\left(z^{[2]}\right)$

Back propagation step:

$\frac{d\mathcal{L}}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$

$\frac{d\mathcal{L}}{dz^{[2]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}} = \left(-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right)\left(a^{[2]}\left(1-a^{[2]}\right)\right) = a^{[2]} - y$

$\frac{d\mathcal{L}}{dW^{[2]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{dW^{[2]}} = \left(a^{[2]} - y\right)a^{[1]T}$

$\frac{d\mathcal{L}}{db^{[2]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}dz^{[2]}}{dz^{[2]}}\frac{d^{[2]}}{db^{[2]}} = \left(a^{[2]} - y\right)$

$\frac{d\mathcal{L}}{da[1]} = \frac{d\mathcal{L}}{da[2]}\frac{da^{[2]}dz^{[2]}}{dz^{[2]}da[1]} = \left(a^{[2]} - y\right)W^{[2]T} = W^{[2]T}\left(a^{[2]} - y\right)$

$\frac{d\mathcal{L}}{dz^{[1]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}} = W^{[2]T}\left(a^{[2]} - y\right) * \left(a^{[1]} * \left(1 - a^{[1]}\right)\right)$

$\frac{d\mathcal{L}}{dW^{[1]}} = \frac{d\mathcal{L}}{da[2]}\frac{da^{[2]}dz^{[2]}da^{[1]}}{dz^{[2]}}\frac{dz^{[1]}}{da[1]}\frac{d\mathcal{L}}{dz^{[1]}}\frac{dW^{[1]}}{dz^{[1]}}x^T$

$\frac{d\mathcal{L}}{db^{[1]}} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}dz^{[1]}}{dz^{[1]}}\frac{dL}{db^{[1]}} = \frac{d\mathcal{L}}{dz^{[1]}}$

Gradient Descent Algorithm: $W^{[1]} := W^{[1]} - \alpha \frac{dJ\left(W^{[1]}, b^{[1]} W^{[2]}, b^{[2]}\right)}{dW^{[1]}}$

6.1.3 Random Initialization

When you're running an algorithm of gradient descent, we need to pick some initial value for the θ parameteres. Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ.

6.2 Putting all together: recap

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features $x^{(i)}$;
- Number of output units = number of classes;
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units);
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

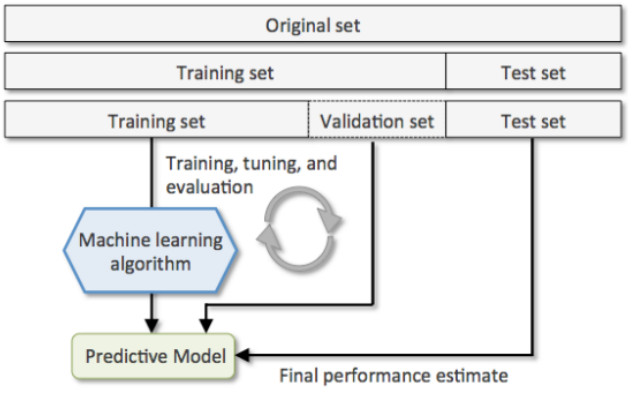
Steps for Training a Neural Network:

- Randomly initialize the weights;
- Implement forward propagation to get $h_\Theta\left(x^{(i)}\right)$ for any $x^{(i)}$;
- Implement the cost function;
- Implement backpropagation to compute partial derivatives;
- Use gradient checking to confirm that your backpropagation works. Then disable gradient checking;
- Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

6.3 Train-Validation-Test Set

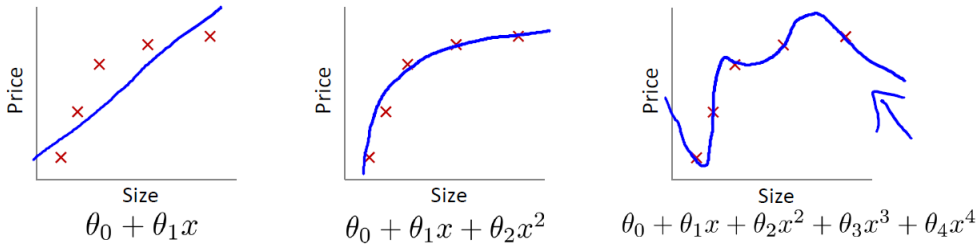
When training a neural network model, it is important to use original data in the correct way.

- **Train-set:** A training data set is a set of examples used during the learning process and it is used to fit the parameters (e.g. weights).
- **Validation-set:** The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration. The validation set is used to evaluate a given model, but this is for frequent evaluation. We, as machine learning engineers, use this data to fine-tune the model hyperparameters. Hence the model occasionally sees this data, but never does it "Learn" from this. We use the validation set results, and update higher level hyperparameters. So the validation set affects a model, but only indirectly. The validation set is also known as the Dev set or the Development set. This makes sense since this dataset helps during the "development" stage of the model.
- **Test-set:** The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset. The Test dataset provides the gold standard used to evaluate the model. It is only used once a model is completely trained (using the train and validation sets). The test set is generally what is used to evaluate competing models (For example on many Kaggle competitions, the validation set is released initially along with the training set and the actual test set is only released when the competition is about to close, and it is the result of the the model on the Test set that decides the winner). Many times the validation set is used as the test set, but it is not good practice. The test set is generally well curated. It contains carefully sampled data that spans the various classes that the model would face, when used in the real world.

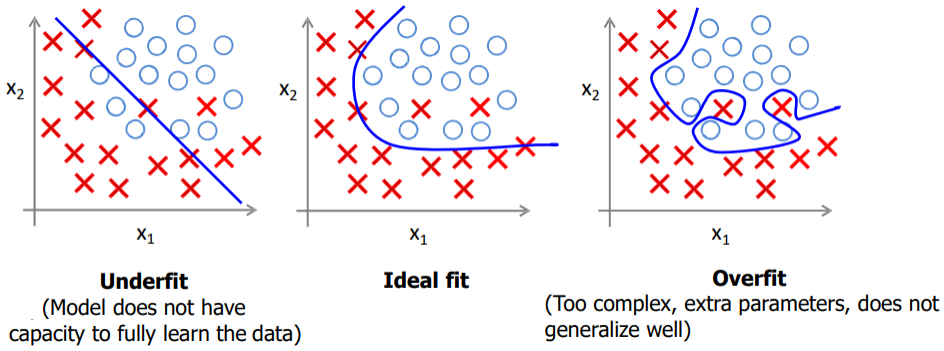


6.4 The problem of Overfitting and Underfitting

Consider the problem of predicting y from $x \in \mathrm{R}$. The leftmost figure below shows the result of fitting a $\mathrm{y} = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Generalizing the concept, the image show the three cases:



Train set error:	1%	15%	0.5%
Dev set error:	11%	16%	1%
	Overfitting	Underfitting	Good*

If we go back to our house-price example, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th} order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$.

We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of underfitting in which the data clearly shows structure not captured by the modeland the figure on the right is an example of overfitting.

Underfitting, or high bias, is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features.

At the other extreme, **overfitting**, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data. This terminology is applied to both linear and logistic regression.

There are two main options to address the issue of overfitting:

1) Reduce the number of features:

- Manually select which features to keep;
- Use a model selection algorithm.

2) Regularization:

- Keep all the features, but reduce the magnitude of parameters θ_j.
- Regularization works well when we have a lot of slightly useful features.

6.4.1 Regularization

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost. Say we wanted to make the following function more quadratic:

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

We'll want to eliminate the influence of $\theta_3 x^3$ and $\theta_4 x^4$. Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our cost function:

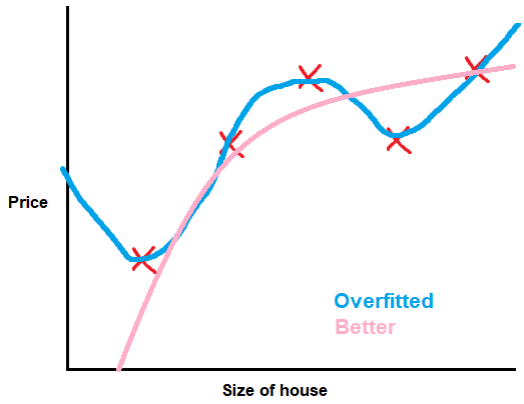
$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2$$

We've added two extra terms at the end to inflate the cost of θ_3 and θ_4. Now, in order for the cost function to get close to zero, we will have to reduce the values of θ_3 and θ_4 to near zero. This will in turn greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function. As a result, we see that the new hypothesis (depicted by the pink curve) looks like a quadratic function but fits the data better due to the extra small terms $\theta_3 x^3$ and $\theta_4 x^4$.

We could also regularize all of our theta parameters in a single summation as:

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2$$

The λ, or lambda, is the regularization parameter. It determines how much the costs of our theta parameters are inflated. Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.



6.4.2 Regularized Logistic Regression model

Minimization problem with regularization: $\min_{(w,b)} J(w,b)$

$$J(w,b) = \frac{1}{m}\left[\sum_{i=1}^{m} \text{cost}\left(h_{w,b}\left(x^{(i)}\right), y^{(i)}\right) + \frac{\lambda}{2}\sum_{j=1}^{n} w_j^2\right]$$

λ is called regularization parameter.

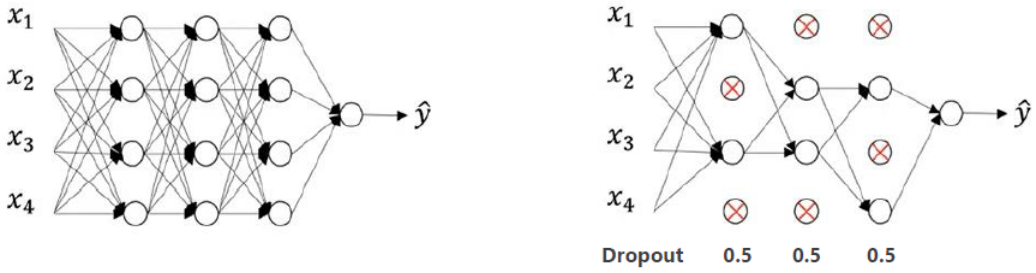
Usually b is ignored in the regularization process.

6.4.3 Dropout Regularization

During training, some number of layer outputs are randomly ignored or "dropped out."

Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the input. The idea is to train your model using a smaller neural networks.

Dropout is not used after training when making a prediction with the fit network.



6.4.4 Data Augmentation

Data augmentation in data analysis are techniques used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data.

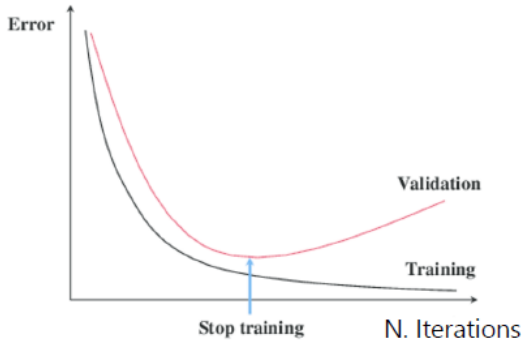
Typical transformations: geometric transformations, flipping, color modification, cropping, rotation, noise injection and random erasing. Some reseachers use videogames to try Deep Learning system (e.g. Self driving car domain)

6.4.5 Early Stopping

When training a large network, there will be a point during training when the model will stop generalizing and start learning the statistical noise in the training dataset.

This overfitting of the training dataset will result in an increase in generalization error, making the model less useful at making predictions on new data.

One approach to solving this problem is to treat the number of training epochs as a hyperparameter and train the model multiple times with different values, then select the number of epochs that result in the best performance on the train or a holdout test dataset.



6.5 Normalization

Normalization is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the numeric values in the dataset to use a common scale, without distorting differences in the ranges of values or losing information. A normalization phase should not affect in any way the output accuracy of the model, so why it's used? it improves the numerical stability of the model and it can only speed up the training process. The two most common types of normalization techniques are Mean and Min-Max normalization.

6.5.1 Mean Normalization (Standardization)

Mean Normalization (also called Z-Normalization or Standardization) transforms data to have a mean of zero and a standard deviation of 1. It's the preferred normalization for Neural Networks.

Given a feature x_i the mapped feature is $x' = \frac{x_i - \mu_i}{\sigma_i}$ where μ_i, σ_i are the mean and standard deviation computed on the training set.

Mean: $\mu_i = \frac{1}{m} \sum_{i=1}^{m} x_i$

Standard Deviation: $\sigma_i = \sqrt{\frac{1}{m-1} \sum_{i=1}^{m} (x - \mu_i)^2}$ where m is the number of samples of the training set

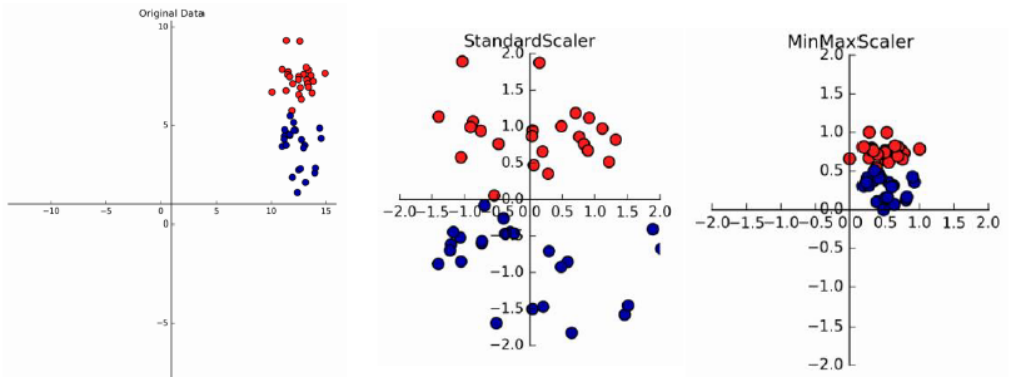
6.5.2 MinMax Normalization

The MinMax normalization shifts the data such that all features are between 0 and 1.

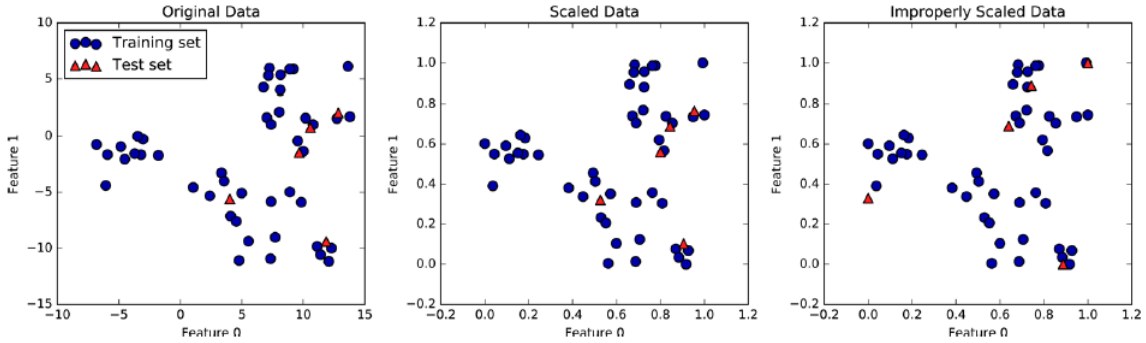
$$x_i' = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Where x_i is the original value, x_i' is the normalized value.

Figure shows a two-class classification dataset with two features and a comparison between Mean Normalization (StandardScaler) and MinMax Normalization (MinMaxScaler):



It is important to apply the same transformation to the training set and the test set for the supervised model to work on the test set. The figure illustrates what would happen if we were to use the wrong range for the transformations:



The third panel shows what would happen if we scaled the training set and test set separately. In this case, the minimum and maximum feature values for both the training and the test set are 0 and 1. But now the dataset looks different. The test points moved incongruously to the training set, as they were scaled differently. We changed the arrangement of the data in an arbitrary way. Clearly this is not what we want to do.

7 NLP & Word Embedding

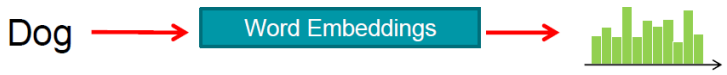
Let's start with the concept of natural language processing. NLP is a subfield of linguistics, computer science, and artificial intelligence; concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data.

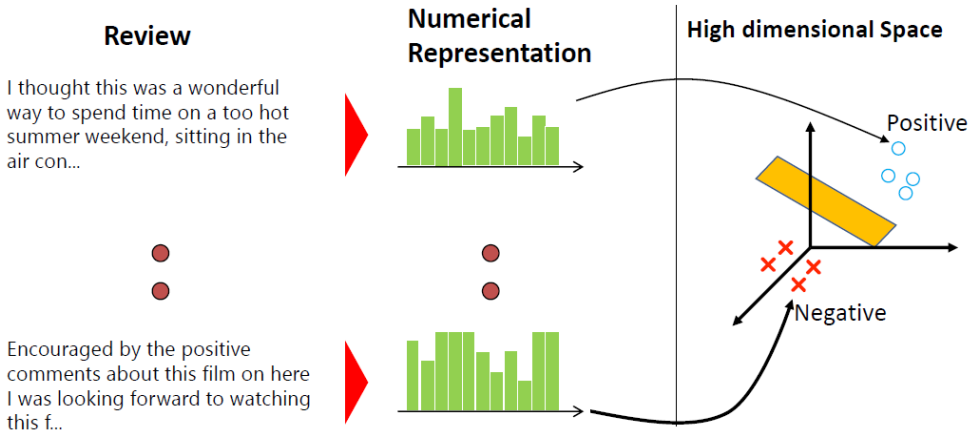
The goal is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them. The technology can then accurately extract information and insights contained in the documents as well as categorize and organize the documents themselves.

Challenges in natural language processing frequently involve speech recognition, natural language understanding, and natural language generation.

Examples of application are: email auto-responders, chatbots, video machine translators, social-media bots, question answering, document analysis, sentiment analysis on a given topic (for example covid-19 vaccine from tweets), fake news detection.

Concept of **Word embedding:** Word embeddings are a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning. Key to the approach is the idea of using a dense distributed representation for each word. Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding. The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag of words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

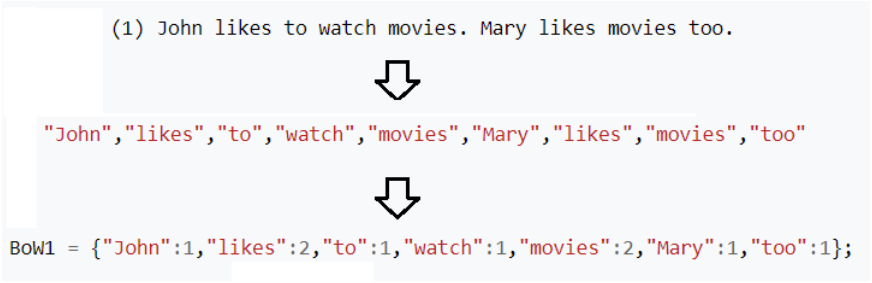




Word representation techniques represent words as feature vectors. Combined with Deep Learning these techniques are state of the art in NLP in almost all the tasks. We will now see some basic and more advanced techniques.

7.1 Models

The **bag-of-words** model is a simplifying global representation used in natural language processing to represent a text (such as a sentence or a document). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity. It's quite effective in document classification.



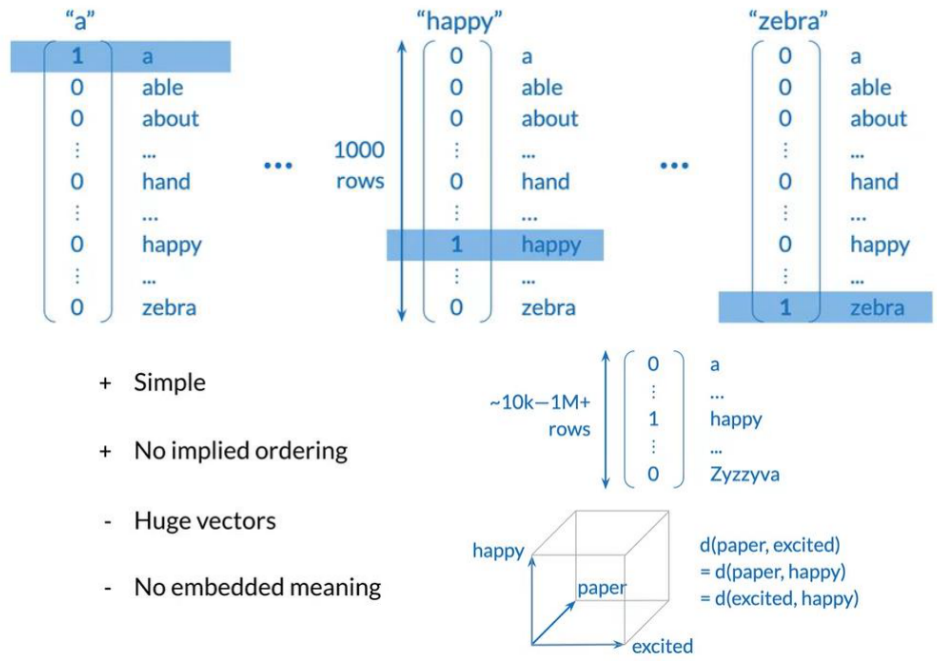
Each key is the word, and each value is the number of occurrences of that word in the given text document. The order of elements is free.

Anyway, the simplest way to represent words as number is for a given vocabulary to assign a unique integer to each word. The set of unique words used in the text corpus is referred to as the vocabulary.

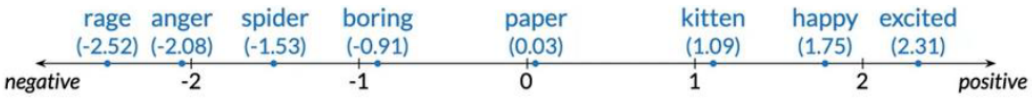
Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000

This numbering scheme is simple, but one of the problems is that we introduce an order of the words. For instance, alphabetical order doesn't make much sense from a semantic perspective.

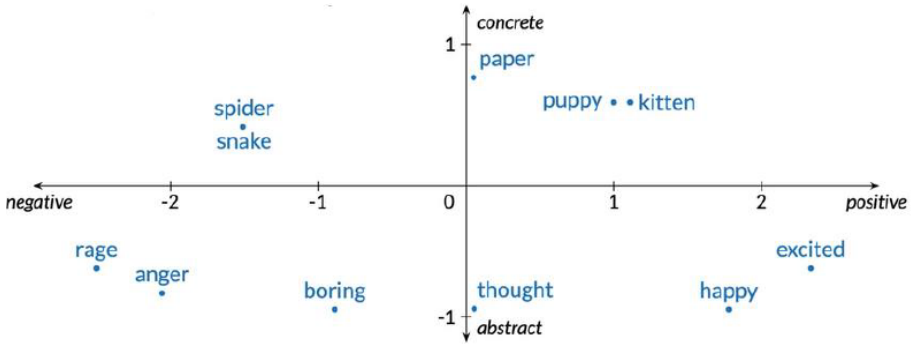
A valid alternative is the **One-hot encoding** representation technique. One-hot vectors represent the words using a column vector where each element corresponds to a word of the vocabulary.



Word embeddings are vectors that are carrying meaning. Consider an horizontal number line like this:

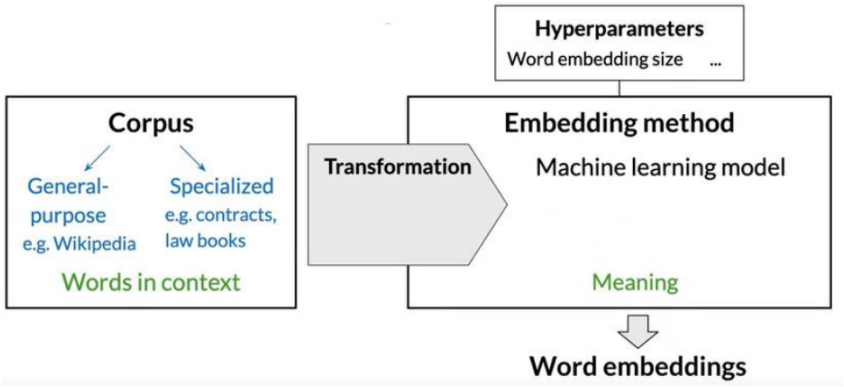


Words on the left are considered negative in some way and words on the right are considered positive in some. Now, you can say that "happy" and "excited" are more similar to each other compared to the word "paper". You can extend this representation by adding a vertical number line, words that are higher on this line are more concrete physical objects. Whereas lower on this line are more abstract ideas. What you've done here is to represent the vocabulary of words with a small vector of length 2.



This representation has a lower dimension and embed meaning (for example semantic distance).

Word Embedding Process: a scheme of the word embedding process:



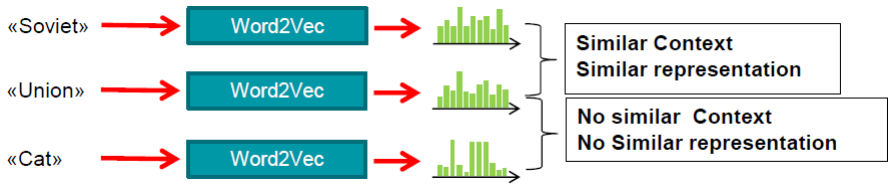
7.2 Word2Vec

Word2vec is a technique for natural language processing published in 2013. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between the vectors) indicates the level of semantic similarity between the words represented by those vectors.

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.

In summary: Word2Vec network is a technique for building a rich semantic word embedding space.

The key idea is that two words have similar word embedding representations if they have a similar contexts. For example:



There are two methods for learning representations of words:

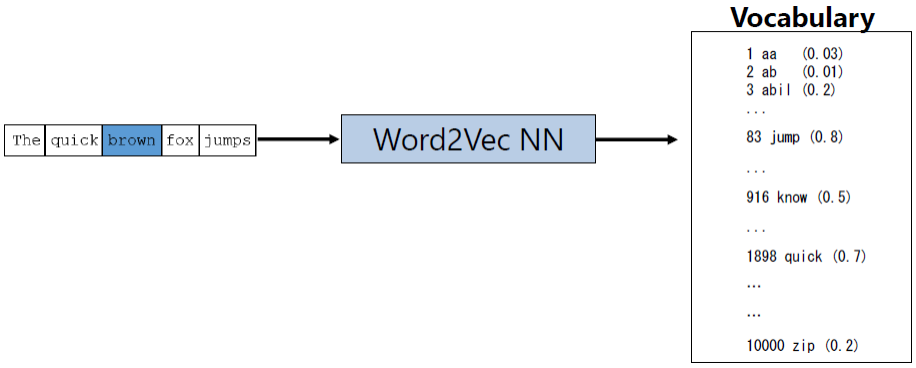
- **Continuous Bag-of-Words Model** which predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

- **Continuous Skip-gram Model** which predict words within a certain range before and after the current word in the same sentence.

We will now see the Word2Vec version based on Skip-Gram Neural Network Architecture. More in this link or in this one.

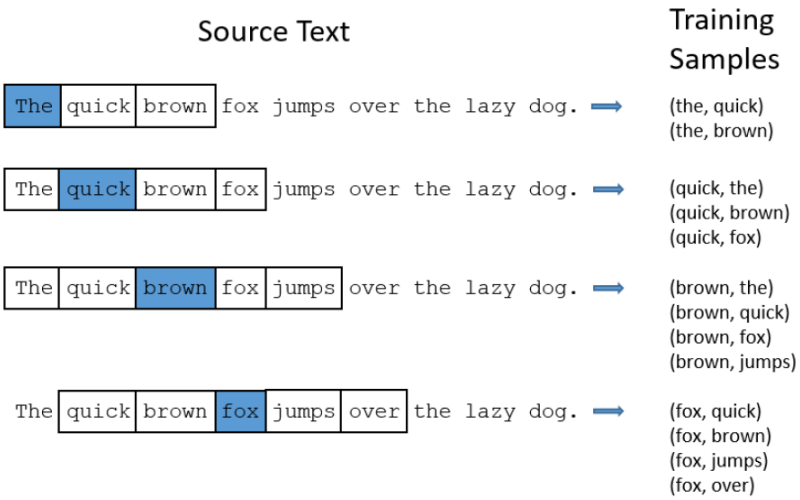
7.2.1 The Fake Task

Let's suppose we have a vocabulary. Given a specific word in the middle of a sentence (the input word), the network is going to tell us the probability for every word in our vocabulary of being the "nearby word" that we chose. When we say "nearby", there is a "window size" parameter to the algorithm. A typical windows size might be 5, meaning 5 words behind and 5 words ahead.



7.2.2 Training Procedure

We'll train the neural network to do this by feeding it with pairs of words found in our training documents. Here we can see a windows size = 2 example:

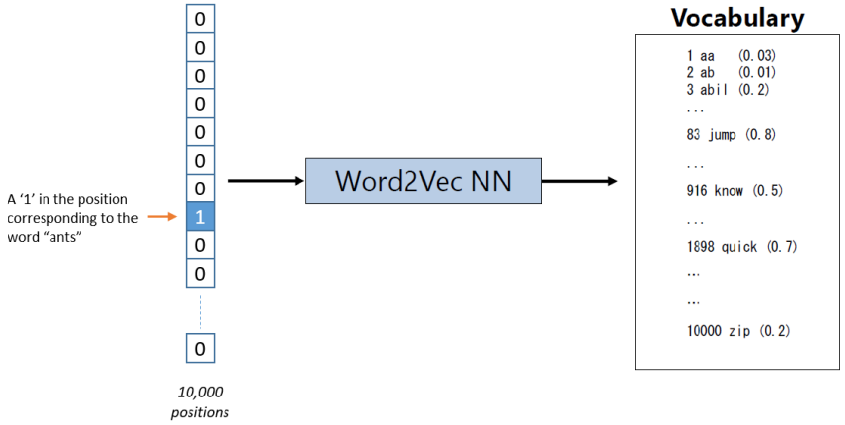


The first part "The" is the input word, the "quick" is the target word. The network is going to learn the statistics from the number of times each pairing shows up. The network

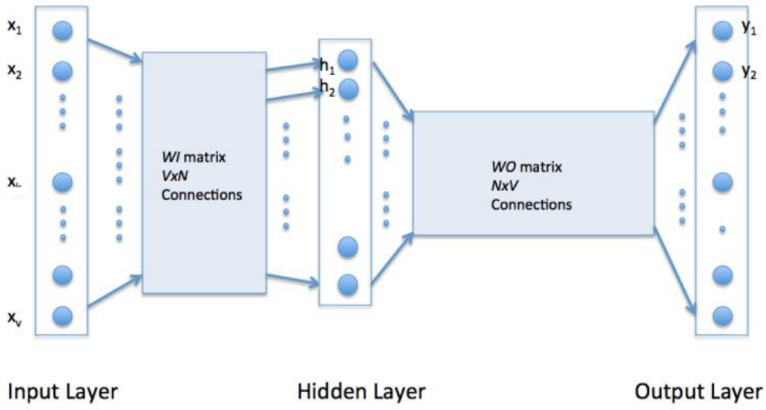
is probably going to get many more training samples of ("Soviet", "Union") instead of ("Soviet", "Cat"). When the training is finished, if you give it the word "Soviet" as input, then it will output a much higher probability for "Union" or "Russia" than it will for "Cat".

7.2.3 Model Details

Let's suppose we have a vocabulary of 10,000 unique words. One-hot vector for the input word: e.g. "ants".



The neural network is composed by an input layer, Hidden layer and Output layer. There is no activation function on the hidden layer neurons.



For our example, we're going to say that we're learning word vectors with 300 features. Hidden layer is represented by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for every hidden neuron). If you look at the rows of this weight matrix, these are actually what will be our word vectors! The output layer has the same number of neurons as the input.

Softmax Function is used in order to obtain probabilities for words in the output layer

$$\hat{y}_k = \Pr\left(\text{word}_k \mid \text{word}_{\text{context}}\right) = \frac{\exp\left(y_k\right)}{\sum_{n=1}^{V} \exp\left(y_k\right)}$$

More about the skipgram model on this website.

7.2.4 Simple example:

The Corpus is the following one:

"the cat climbed a tree, the dog saw a cat, the dog chased the cat"

The vocabulary is the following one:

Words	Index
a	1
cat	2
chased	3
climbed	4
dog	5
saw	6
the	7
tree	8

Let's assume dimension of Word2Vec repsentation equal to 3, then we have:

$WI =$

$$\begin{bmatrix} -0.094491 & -0.443977 & 0.313917 \\ -0.490796 & -0.229903 & 0.065460 \\ 0.072921 & 0.172246 & -0.357751 \\ 0.104514 & -0.463000 & 0.079367 \\ -0.226080 & -0.154659 & -0.038422 \\ 0.406115 & -0.192794 & -0.441992 \\ 0.181755 & 0.088268 & 0.277574 \\ -0.055334 & 0.491792 & 0.263102 \end{bmatrix}$$

$WO =$

$$\begin{bmatrix} 0.023074 & 0.479901 & 0.432148 & 0.375480 & -0.364732 & -0.119840 & 0.266070 & -0.351000 \\ -0.368008 & 0.424778 & -0.257104 & -0.148817 & 0.033922 & 0.353874 & -0.144942 & 0.130904 \\ 0.422434 & 0.364503 & 0.467865 & -0.020302 & -0.423890 & -0.438777 & 0.268529 & -0.446787 \end{bmatrix}$$

Training example:

The input word "cat" $X = [0\ 1\ 0\ 0\ 0\ 0\ 0\ 0]^T$

The target word "climbed" $= [0\ 0\ 0\ 1\ 0\ 0\ 0\ 0]^T$

The output of the hidden layer neurons:

$$H^T = X^T WI = [-0.490796\ -0.2299030\ 0.065460]$$

The activation vector for the output layer neurons:

$$H^T WO = [0.100934\ -0.309331\ -0.122361\ -0.151399\ \ 0.143463\ -0.051262\ -0.079686\ \ 0.112928]$$

Since we need probabilities for word in the output layer, we use the softmax function:

$$\hat{y}_k = \Pr\left(\text{word}_k \mid \text{word}_{\text{context}}\right) = \frac{\exp\left(y_k\right)}{\sum_{n=1}^{V} \exp\left(y_k\right)}$$

The probabilities for eight words in the corpus are:

$$0.143073\ \ 0.094925\ \ 0.114441\ \ 0.111166\ \ 0.149289\ \ 0.122874\ \ 0.119431\ \ 0.144800$$

The target word "climbed" $[0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0]^T$

$$Error = [0.143073\ \ 0.094925\ \ 0.114441\ \ -0.888834\ \ 0.149289\ \ 0.122874\ \ 0.119431\ \ 0.144800]$$

Updating weights matrices using backpropagation. If two different words have very similar "contexts" then our model needs to output very similar results for these two words. And one way for the network to output similar context predictions for these two words is if the word vectors are similar. So, if two words have similar contexts, then our network is motivated to learn similar word vectors for these two words!

7.2.5 Full Skip-Gram

Let's suppose we set a window of one single word. Considering the following training example:

- The input word "cat" $X = [0\ 1\ 0\ 0\ 0\ 0\ 0\ 0]^T$
- The target word1: "climbed" $T_{w1} = [0\ 0\ 0\ 1\ 0\ 0\ 0\ 0]^T$
- The target word2: "tree" $T_{w2} = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 1]^T$

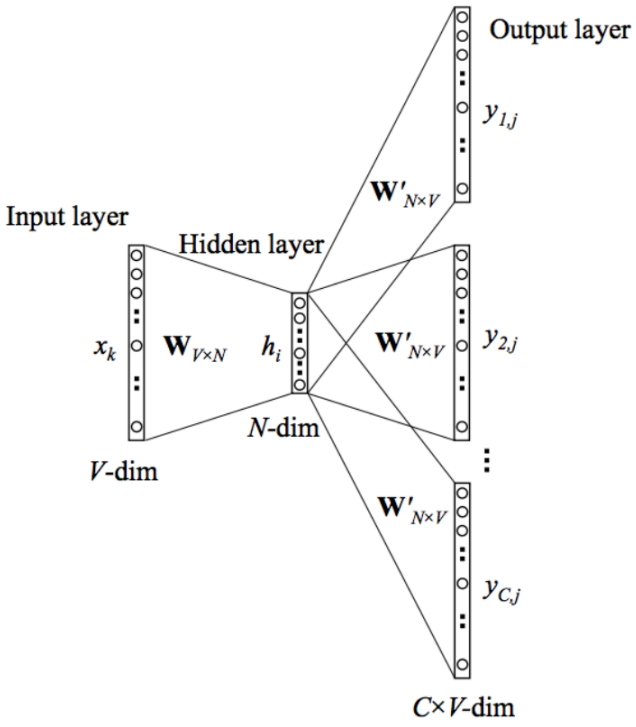
We have $y_{1,j} = y_{2,j}$, because they share the same weight matrices. $E_1 = y_{1,j} - T_{w1}$ and $E_2 = y_{2,j} - T_{w2}$ so:

$$E = E_1 + E_2$$

We update weights matrices using backpropagation.

The skip-gram model for Word2Vec is a large neural network. If we have word vectors with 300 components and a vocabulary of 10,000 words, each weight matrix will have $300 \times 10000 = 3$, million weights.

Running gradient descent on a neural network that large is going to be slow. It requires a huge amount of training data. More in this link.



7.3 Improvements for Word2Vec

There are three innovations introduced in this technique in order to improve Word2Vec:

- Treating common word pairs or phrases as single "words" in their model.
- Subsampling frequent words to decrease the number of training examples.
- Modifying the optimization objective with a technique called "Negative Sampling", which causes each training sample to update only a small percentage of the model's weights.

More on this website/article.

7.3.1 Word Pairs and Phrases

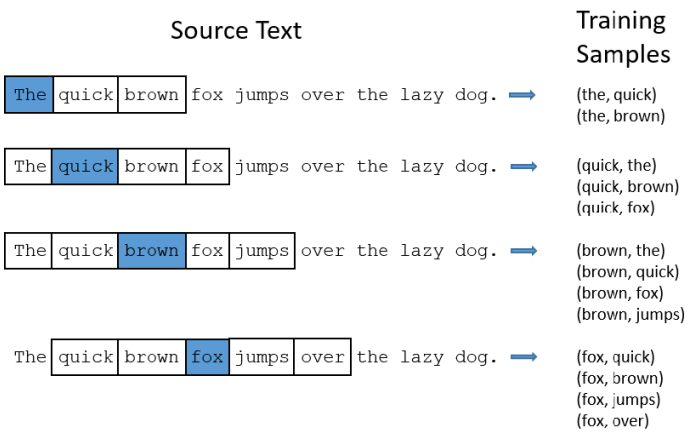
The authors pointed out that a word pair like "Boston Globe" (a newspaper) has a much different meaning than the individual words "Boston" and "Globe".

So it makes sense to treat "Boston Globe", wherever it occurs in the text, as a single word with its own word vector representation. Words that appear always together are merged.

7.3.2 Subsampling Frequent Words

There are two "problems" with common words like "the": ("fox", "the") doesn't tell us much about the meaning of "fox". We will have many more samples of ("the", ...). Frequent words are removed.

It's like/similar stopwords removal techniques from the Information Retrieval field.



7.3.3 Negative Sampling

Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In other words, each training sample will tweak all of the weights in the neural network.

As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here's how it works.

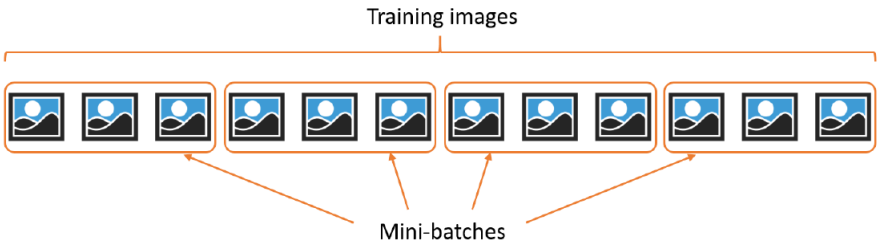
When training the network on the word pair ("fox", "quick"), recall that the "label" or "correct output" of the network is a one-hot vector. That is, for the output neuron corresponding to "quick" to output a 1, and for all of the other thousands of output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of "negative" words (let's say 5) to update the weights for. (In this context, a "negative" word is one for which we want the network to output a 0 for). We will also still update the weights for our "positive" word (which is the word "quick" in our current example).

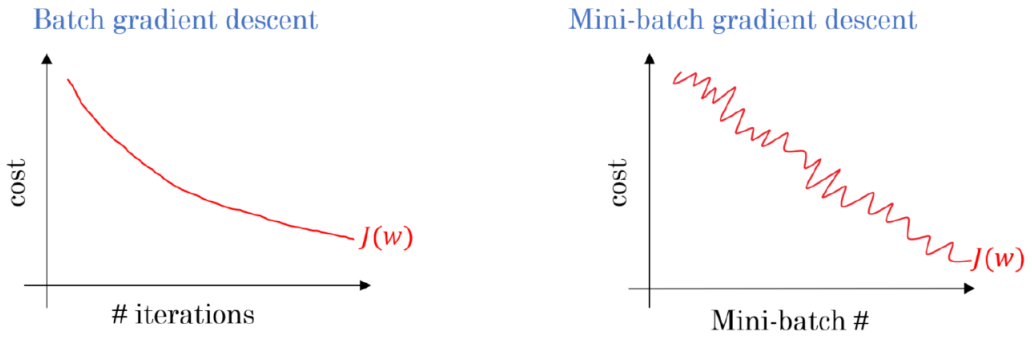
8 Network Optimization

8.1 Batch vs. Mini-Batch vs Stochastic Gradient Descent

Let's suppose you want to solve a Computer Vision task and you have the following training set:



When you are using the Gradient Descent Algorithm, the entire training set is used to compute the Cost function and the derivatives. However, it's prohibitive when you are dealing with very large scale datasets. Instead if you are using a Mini Batch Gradient you compute the Cost Function and the derivatives in each of the batches. At the end of this process you have done 1 Epoch through the training set. Usually we train the neural networks for more than 1 epoch.



The mini-batch behaviour is less smooth because some batch can be easier than others.

What about **Stochastic Gradient Descent**? SDG is when you use a single observation to calculate the cost function. While doing so, we pass a single observation at a time, calculate the cost and update the parameters. The procedure will take the first observation, then pass it through the neural network, calculate the error and then update the parameters. Then will take the second observation and perform similar steps with it. This step will be repeated until all observations have been passed through the network and the parameters have been updated. Then will take the second observation and perform similar steps with it. This step will be repeated until all observations have been passed through the network and the parameters have been updated.

More about Gradient Descent variants at this website.

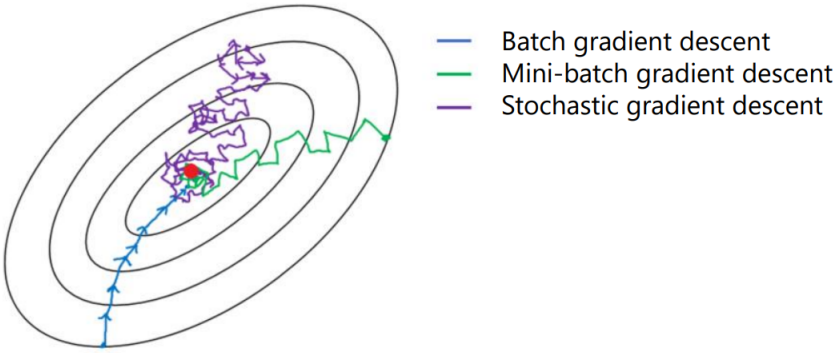
8.1.1 Choosing mini-batch size:

Given m = data set size:

- If mini batch size = m $\mapsto$ Gradient Descent
- If mini batch size = 1 $\mapsto$ Stochastic Gradient Descent

Consider that:

- Stochastic Gradient Descent does not use vectorization. Therefore it's slow.
- Batch Gradient Descent requires a huge memory space.
- Mini-batch gradient descent is an intermediated solution (it uses vectorization and does not require huge memory space).



If you have small training set (e.g. 2000 samples), just use batch gradient descent. If you have larger training set typical mini batch sizes are: 64, 128, 256, 512, 1024, ...

The main constraint is that the batch must be allocated in CPU/GPU memory. The max size usually depends on your applications and your hardware.

Size of mini batch can be defined as an hyperparameter. Thus, you need to test several values in order to find the correct one.

8.2 Optimization Algorithms

It is very important to tweak the weights of the model during the training process, to make our predictions as correct and optimized as possible. But how exactly do you do that? How do you change the parameters of your model, by how much, and when?

Best answer to all above question is the usage of **optimizers**. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights.

The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

So far, the theory told us to use Gradient Descent. It is still a powerful technique, but, in practice, this tecnique may encounter certain problems during the training phase. Problems can be saddle points, local minima or others.

On the other hand, even if we have gradients that are not close to zero, the values of these gradients calculated for different data samples from the training set may vary in value and direction. We say that the gradients are noisy or have a lot of variances. This leads to a zigzag movement towards the optimal weights and can make learning much slower.

There are different algorithms/optimizers, still based on Gradient Descent, but which extends the idea and are more effective.

A good source about this topic is this article.

8.2.1 Exponential Weighted Average EWA

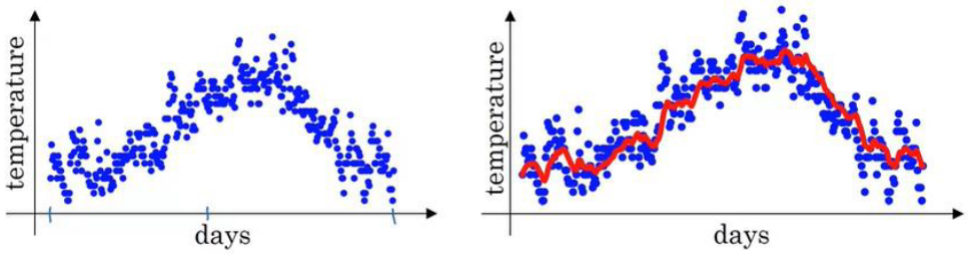
We will see, before actual optimizers, review the concept of exponential weighted average. Let's start with an example about temperatures in a year:

First day: $\theta_1 = 4°$
Second day: $\theta_2 = 9°$...
200° day: $\theta_{200} = 20°$...

The plot is very noisy. If you set:

$v_0 = 0$
$v_1 = 0.9v_0 + 0.1\theta_1$...

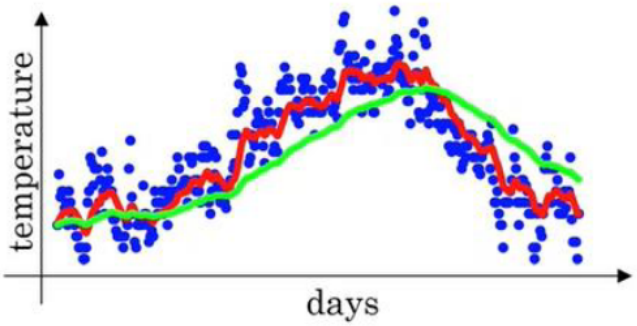
The red line is a more smoothed curve and it is called **Exponentially Weighted Average** of the daily temperature.



$$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$$

It can be proved that V_t is an approximation of $\frac{1}{1-\beta}$ days'temperatures.

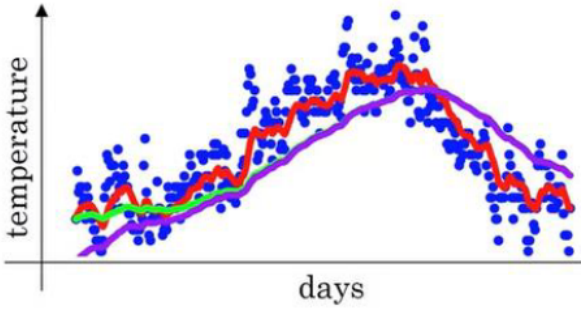
Red line: $\beta = 0.9$ (average of ≈ 10 days' temperature)
Green line: $\beta = 0.98$ (average of ≈ 50 days' temperature)



Bias Correction: If you implement this formula:

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

you will have the purple curve.



Let's see an example:

If you set $\beta = 0.98$ and $V_0 = 0$ you will have:
$\rightarrow \mathbf{v}_1 = 0.98V_0 + 0.02\theta_1 = \mathbf{V_1} = \mathbf{0.02\theta_1}$
$\rightarrow \mathbf{v}_2 = 0.98V_1 + 0.02\theta_2 = 0.98 * 0.02 * \theta_1 + 0.02\theta_2 = \mathbf{0.0196\theta_1 + 0.02\theta_2}$

Note: V_1 and V_2 are not very good estimations of the first two days of temperature of the year. Thus, we add a bias term. The revised formula is: $\tilde{V}_t = \frac{V_t}{1-\beta^t}$ (green curve).

$$\rightarrow t = 2 \rightarrow 1 - \beta^t = 1 - (0.98)^2 = 0.0396 \rightarrow \tilde{V}_2 = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

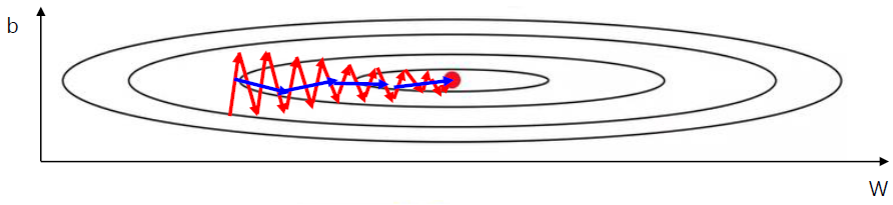
Note: when $t \rightarrow \infty$ the term $1 - \beta^t \rightarrow 1$

8.2.2 Gradient Descent with Momentum

Used in conjunction Stochastic Gradient Descent (SGD) or Mini-Batch Gradient Descent, **Momentum** takes into account past gradients to smooth out the update.

This is seen in variable v which is an exponentially weighted average of the gradient on previous steps. This results in minimizing oscillations and faster convergence.

Colors: Gradient Descent and Gradient Descent with Momentum.



Gradient descent with Momentum: For each iteration compute $\frac{d\mathcal{L}}{dW}$ and $\frac{d\mathcal{L}}{db}$.

$V_{dw} = \beta V_{dw} + (1-\beta)\frac{d\mathcal{L}}{dW}$; $\quad V_{db} = \beta V_{db} + (1-\beta)\frac{d\mathcal{L}}{db}$ $\quad$ good value of $\beta = 0.9$

$W := W - \alpha V_{dw}$; $\quad$ b $:= b - \alpha V_{db}$

with: W = weighted tensor, β = hyperparameter to be tuned, α = learning rate, v = the exponentially weighted average of past gradients, $\frac{d\mathcal{L}}{dW}$ = cost gradient with respect to current layer weight tensor.

When considering "Exponentially Weighted Averages" in your process, the average of b is "constant" (vertical direction), while W goes toward to right.

Gradient Descent with Momentum almost always works empirically faster than standard gradient descent algorithm.

8.2.3 RMSprop

Another adaptive learning rate optimization algorithm, **Root Mean Square Prop (RMSProp)** works by keeping an exponentially weighted average of the squares of past gradients. RMSProp then divides the learning rate by this average to speed up convergence.

RMSprop:
For each iteration compute $\frac{d\mathcal{L}}{dW}$ and $\frac{d\mathcal{L}}{db}$

$S_{dw} = \beta S_{dw} + (1-\beta)\left(\frac{d\mathcal{L}}{dW}\right)^2$; $\quad S_{db} = \beta S_{db} + (1-\beta)\left(\frac{d\mathcal{L}}{db}\right)^2$ $\quad$ good value of $\beta = 0.9$

$W := W - \alpha \frac{d\mathcal{L}}{dW} / \left(\sqrt{S_{dw}} + \epsilon\right)$; $\quad$ b $:= b - \alpha \frac{d\mathcal{L}}{dW} / \left(\sqrt{S_{db}} + \epsilon\right)$ $\quad$ good value $\epsilon = 10^{-8}$

with: W = weighted tensor, β = hyperparameter to be tuned, α = learning rate, v = the exponentially weighted average of past gradients, $\frac{d\mathcal{L}}{dW}$ = cost gradient with respect to current layer weight tensor, ϵ it's a small value just added to avoid division by zero.

In our example $\frac{d\mathcal{L}}{dW}$ are a small values, while $\frac{d\mathcal{L}}{db}$ are a large values. Therefore, when you update b you divide for a large number the derivative, so the oscillations will be reduced.

8.2.4 ADAM

Adaptive Moment Estimation (ADAM) combines ideas from both RMSProp and Momentum. It computes adaptive learning rates for each parameter and works as follows.

- First, it computes the exponentially weighted average of past gradients V_{db}.
- Second, it computes the exponentially weighted average of the squares of past gradients S_{dW}.

- Third, these averages have a bias towards zero and to counteract this a bias correction is applied $V_{dW}^{corrected}$, $S_{db}^{corrected}$.
- Lastly, the parameters are updated using the information from the calculated averages.

Adam:

$$V_{dW} = \beta_1 V_{dW} + (1-\beta_1)\frac{d\mathcal{L}}{dW} \qquad V_{db} = \beta_1 V_{db} + (1-\beta_1)\frac{d\mathcal{L}}{db}$$

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\left(\frac{d\mathcal{L}}{dW}\right)^2 \qquad S_{db} = \beta_2 S_{db} + (1-\beta_2)\left(\frac{d\mathcal{L}}{db}\right)^2$$

$$\tilde{V}_{dw} = V_{dW}^{\text{corrected}} = \frac{V_{dW}}{1-(\beta_1)^t} \qquad \tilde{V}_{db} = V_{db}^{\text{corrected}} = \frac{V_{db}}{1-(\beta_1)^t}$$

$$\tilde{S}_{dw} = S_{dW}^{\text{corrected}} = \frac{S_{dW}}{1-(\beta_2)^t} \qquad \tilde{S}_{db} = S_{db}^{\text{corrected}} = \frac{S_{db}}{1-(\beta_2)^t}$$

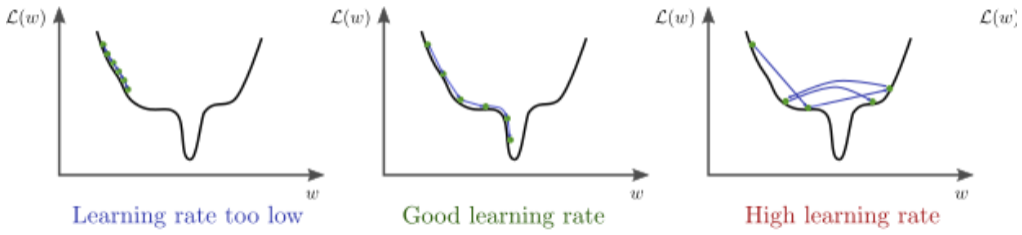
$$W := W - \alpha\frac{V_{dW}^{\text{corrected}}}{\sqrt{s_{dW}^{\text{corrected}}}+\varepsilon} \qquad b := b - \alpha\frac{V_{db}^{\text{corrected}}}{\sqrt{s_{db}^{\text{corrected}}}+\varepsilon}$$

Values and parameters:

- V_{dW} - the exponentially weighted average of past gradients
- S_{dW} - the exponentially weighted average of past squares of gradients
- β_1 - hyperparameter to be tuned
- β_2 - hyperparameter to be tuned
- $\frac{d\mathcal{L}}{dW}$ - cost gradient with respect to current layer
- W - the weight matrix (parameter to be updated)
- α - the learning rate
- ϵ - very small value to avoid dividing by zero

8.3 Learning Rate Decay Problem

As we saw, based on the learning rate we can have the following three situations:



So it is important to handle the learning rate as an hyperparameter, in order to use the most effective value. Not too high and not too low.

More at this article.

Learning rate should be adapted (usually reduced) during the training phase, for example with this formula:

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

Example:

$$\alpha_0 = 0.2; \text{ decay } - \text{ rate } = 1$$

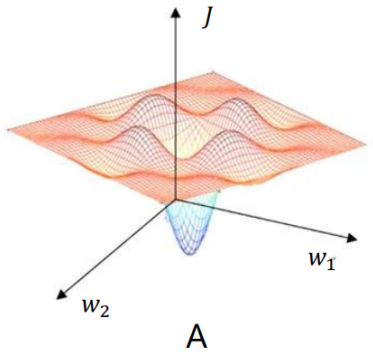
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
...	...

Other valid formulas can be:

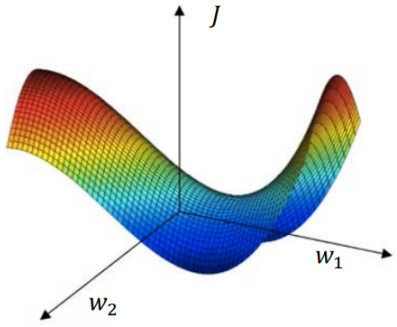
$\alpha = \beta^{\text{epoch-num}} \alpha_0$ with $\beta \in (0, 1)$ - Exponentially Decay s $\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \alpha_0$ with $k > 1$

8.4 Local Optima in Neural Networks

In Neural Network (we are working with high-dimensional space) the cost function likely has saddle points (figure B) instead of local point (figure A), because in high-dimensional space it's unlikely to have all the dimensions with convex shape.

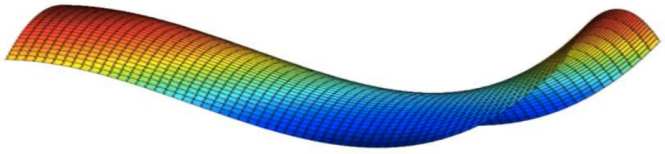


A



B

We can also have problem with "plateaus". In the scenario shown in the image below, Gradient Descent with Momentum, RMSprop or Adam work better then Gradient Descent.



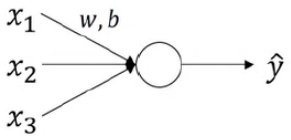
9 Normalizing inputs to speed up learning (BatchNorm)

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

BatchNorm accelerates convergence by reducing internal covariate shift inside each batch. If the individual observations in the batch are widely different, the gradient updates will be choppy and take longer to converge.

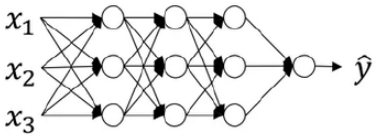
The batch norm layer normalizes the incoming activations and outputs a new batch where the mean equals 0 and standard deviation equals 1. It subtracts the mean and divides by the standard deviation of the batch.

In Logistic Regression we used feature normalization:

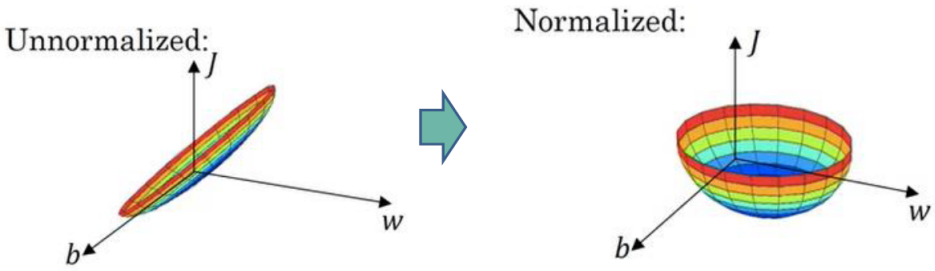


$x' = \frac{x_i - \mu_i}{\sigma_i}$ where μ_i, σ_i are the mean and standard deviation computed on the training set.

In Neural Networks:



Batch Normalization consists of normalizing intermediate layer values in order to improve the network stability.



9.1 Implementing Batch Norm

Given intermediate layer values of $z^{[l]}$ (values before the activation function) of a mini-batch B of size m of the entire training set.

The mean and variance of B could thus be calculated as:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)}$$

and

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z^{[l](i)} - \mu\right)^2$$

For each unit of the layer, its input is normalized (re-centered and rescaled) separately. Thus, the normalization for a d-dimensional layer can be calculated as:

$$\hat{z}_k^{[l](i)} = \frac{z_k^{[l(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

ϵ is added in the denominator for numerical stability and is an arbitrarily small constant (avoid division by 0).

However, the distribution should no be centered in zero, so:

$$\tilde{z}_k^{[l](i)} = \gamma_k^{[l]}\hat{z}_k^{[l](i)} + \beta_k^{[l]}$$

where the parameter $\beta^{[l]}$ and $\gamma^{[l]}$ are learned in the optimization process.

$$\text{if } \gamma^{[l]} = \sqrt{\sigma^2 + \epsilon} \text{ and } \beta^{[l]} = \mu \rightarrow \tilde{z}^{[l]} = \hat{z}^{[l]}$$

The parameter β and γ help the Neural network to center the date in the best place for improve the optimization process.

Note: when you are using Batch Norm in a layer the optimization of bias term is useless, because in the normalization process we subtract the mean and finally we add a β value.

9.1.1 BatchNorm at test time

When we are in a training process, we can divide the training set in mini-batches and compute Mean and Standard Deviation.

Recall, the mean and variance of the mini-batch B could be calculated as:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{[l](i)}; \quad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z^{[l](i)} - \mu_i\right)^2; \quad \hat{z}_k^{[l](i)} = \frac{z_k^{[l(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}; \quad \tilde{z}_k^{[l](i)} = \gamma_k^{[l]}\hat{z}_k^{[l](i)} + \beta_k^{[l]}$$

At the test time you will have just one sample, so the standard approach is to use the averages of μ and σ collected during training time.

Useful recap article about Batch-Norm: click here

10 Loss Functions

In this chapter, we will see some additional details about loss functions.

10.1 Binary Classification

For Binary Classification we often use the Binary Cross-Entropy Loss:

$$\mathrm{L}(\mathrm{W}, \mathrm{b}) = -\frac{1}{m}\sum_{i=1}^{m} -y^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - h_{W,b}\left(x^{(i)}\right)\right)$$

Where m is the number of examples in the training set (see cost Function of the Logistic Regression). We already seen this in this chapter.

For Regression (as seen in linear regression, for example) we often use the Mean Squared Error Loss:

$$\mathrm{L}(\mathrm{W}, \mathrm{b}) = \frac{1}{m}\sum_{i=1}^{m} \left(h_{W,b}\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Where m is the number of examples in the training set.

Note: in the regression setting the network ends with a single unit and no activation. Thus, the last layer is purely "linear" and the network is free to learn to predict values in any range.

10.2 Multiclass Classification

In multiclass-classification problems, we can (and usually) use one-hot representation for represent the classes:

Pedestrian	Car	Motorcycle	Truck
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

10.2.1 SoftMax Layer

In multiclass classification the last layer of a neural network often use a Softmax activation function. Softmax function calculates the probabilities distribution of each of the events over n different events.

In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later, the calculated probabilities will be helpful for determining the target class for the given inputs.

It means that the network will output a probability distribution over the classes.

The SoftMax function (SoftArgMax or Normalized Exponential Function) is a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers.

That is, prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval [0 , 1] , and the components will add up to 1, so that they can be interpreted as probabilities. The standard (unit) softmax function is defined by the formula:

Formaly, let's $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$

Softmax layer is:

$$\mathrm{a}^{[i]} = \frac{e^{\mathrm{z}^{[i]}}}{\sum_{J=1}^{N} e^{\mathrm{z}^{[i]}}}$$

where N is the number of classes.

Example:

$$Z^{[i]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \rightarrow a^{[i]} = \begin{bmatrix} \frac{e^5}{t} \\ \frac{e^2}{t} \\ \frac{e^{-1}}{t} \\ \frac{e^3}{t} \end{bmatrix} \text{ where } t = \left(e^5 + \mathrm{e}^2 + \mathrm{e}^{-1} + \mathrm{e}^3\right) \rightarrow a^{[i]} = \left.\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}\right\} \sum = 1$$

10.2.2 Cross-Entropy Loss

To train a neural network for a multiclass problem we can use the Cross-Entropy Loss. Cross-entropy is commonly used to quantify the difference between two probability distributions.

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log loss of 0.

Usually the "true" distribution (the one that your machine learning algorithm is trying to predict) is expressed in terms of a one-hot distribution.

$$\mathrm{L(W, b)} = -\sum_{i=1}^{m} y^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)\right)$$

Where m is the number of training samples (or in the mini-batch), $y^{(i)}$ is the ground truth output distribution and $h_{W,b}\left(x^{(i)}\right)$ is the predicted output distribution.

In practice, $y^{(i)}$ is a one-hot representation of the right label.

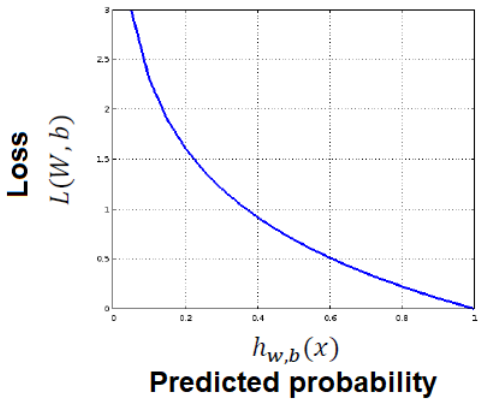
For Example:

$$y^{(i)} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T; \;\; h_{W,b}\left(x^{(i)}\right) = \begin{bmatrix} 0.228 & 0.619 & 0.153 \end{bmatrix}^T$$
$$L = -(0 * \ln(0.228) + 1 * \ln(0.619) + 0 * \ln(0.153)) = 0.479$$

The Cross-Entropy Loss focuses on the positive class, because the negative classes are potentially infinite.

Another Example:

Exactly right: $L = -(1 * \ln(1)) = 0$
50% Probability on correct target: $L = -(1 * \ln(0.5)) = 0.693$
25% Probability on correct target: $L = -(1 * \ln(0.25)) = 1.386$
0% Probability on correct target: $L = -(1 * \ln(0)) = \infty$

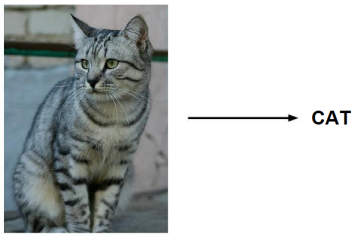


11 Computer Vision

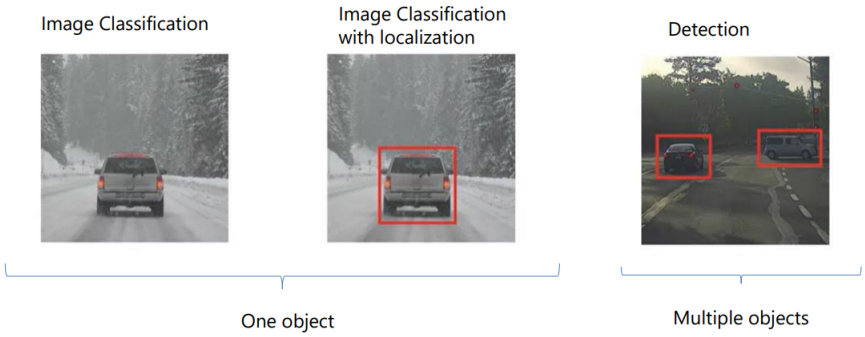
Computer vision is an interdisciplinary scientific field that deals with how computers can gain high-level understanding from digital images or videos. From the perspective of engineering, it seeks to understand and automate tasks that the human visual system can do.

There are many visual recognition problems that are related to image classification, such as object detection, image captioning, semantic segmentation, visual question answering, visual instruction navigation, scene graph generation etc.

Image classification is a core task in computer vision.



Classification, localization and detection:



11.1 Convolutional Neural Networks

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.

CNNs are regularized versions of multilayer perceptrons. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

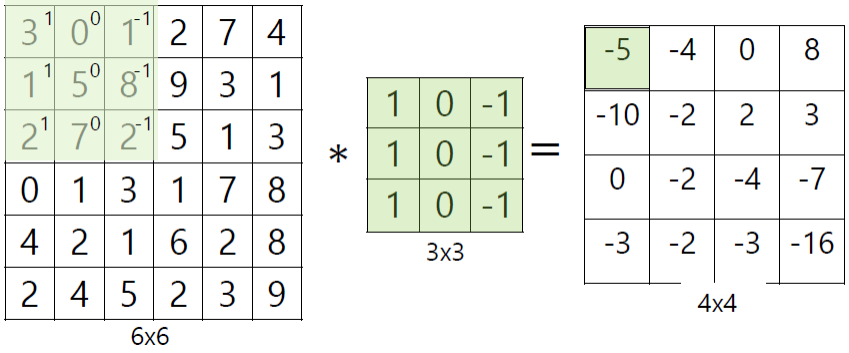
Convolutional networks were inspired by biological processes in which the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were

hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

11.2 Convolutions

The name "convolutional neural network" indicates that the network employs a mathematical operation called **convolution**.



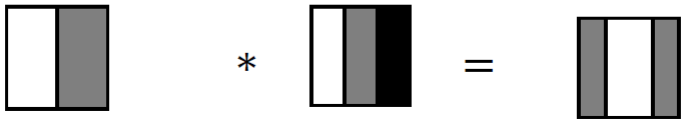
$$I(x,y) * h = \sum_{i=-a}^{a} \sum_{j=-b}^{b} I(x - i, y - j) \cdot h(i,j)$$

Inner product

In mathematics (in particular, functional analysis), convolution is a mathematical operation on two functions (f and g) that produces a third function ($f * g$) that expresses how the shape of one is modified by the other. The term convolution refers to both the result function and to the process of computing it. It is defined as the integral of the product of the two functions after one is reversed and shifted. And the integral is evaluated for all values of shift, producing the convolution function.

Convolutions can be used for many purposes, here an example of vertical edge detection:

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$



Other examples:

11.3 Learning to detect edges

Image processing and computer vision has many hadcrafted filters, like for example: Canny, Sobel, Gaussian Blur, Morphological filters, Gabor filters, etc.

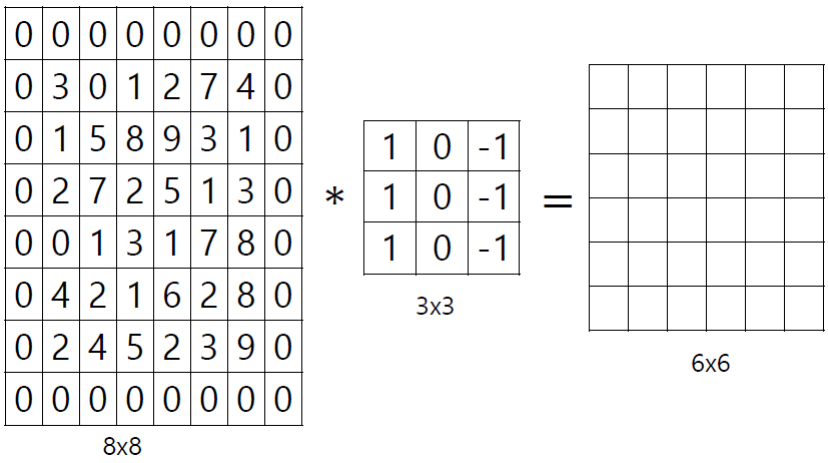
Now, with deep learning, we let the network learn them:

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

The values of the filter (kernel) are weights the networks has to learn.

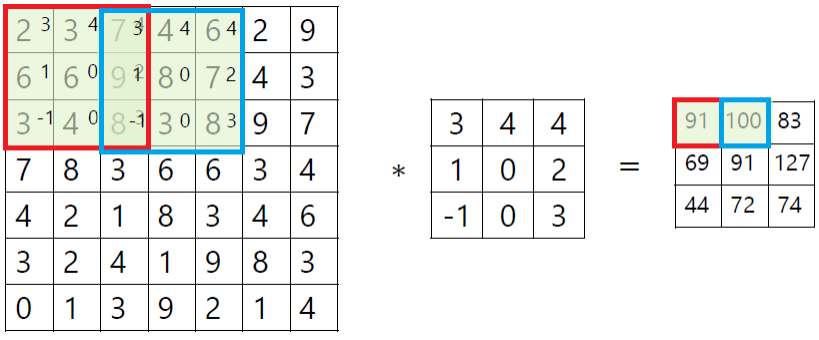
11.4 Padding

Sometimes it is convenient to pad the input with zeros on the border of the input volume. The size of this padding is a third hyperparameter. Padding provides control of the output volume spatial size. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume.



11.5 Strided Convolutions

Stride controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. When the stride is 2 then the filters jump 2 pixels at a time as they slide around. Similarly, for any integer $S > 0$, a stride of S causes the filter to be translated by S units at a time per output. The receptive fields overlap less and the resulting output volume has smaller spatial dimensions when stride length is increased.

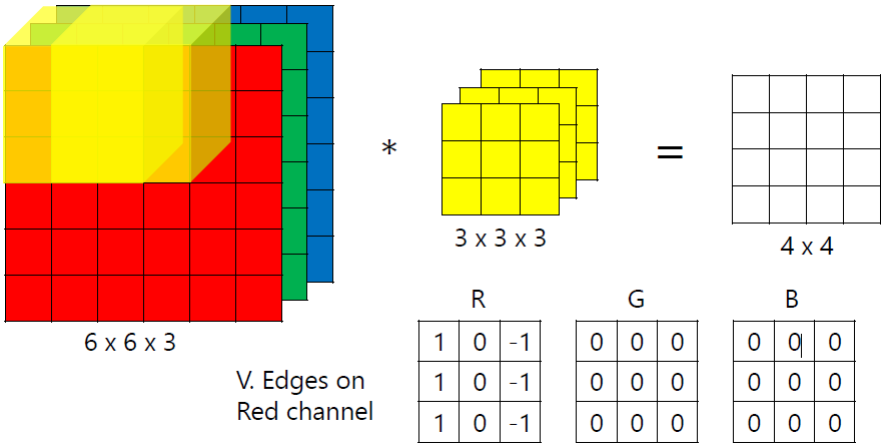


11.6 Convolutions on RGB image

If we want to detect a feature in an RGB image, we must work on a three-dimensional space. Instead of a $n \times n \times 1$ image, we would have a $n \times n \times 3$ image, where 3 corresponds to the 3 color channels RBG. We are now working with volumes.

In order to detect edges or some other feature in this image, we convolve it not with a 3×3 filter, but now with a 3–dimensional filter. That's gonna be a $3 \times 3 \times 3$, so the filter itself will also have three layers corresponding to red, green and blue channels.

Article CNN Convolutions on RGB Images.



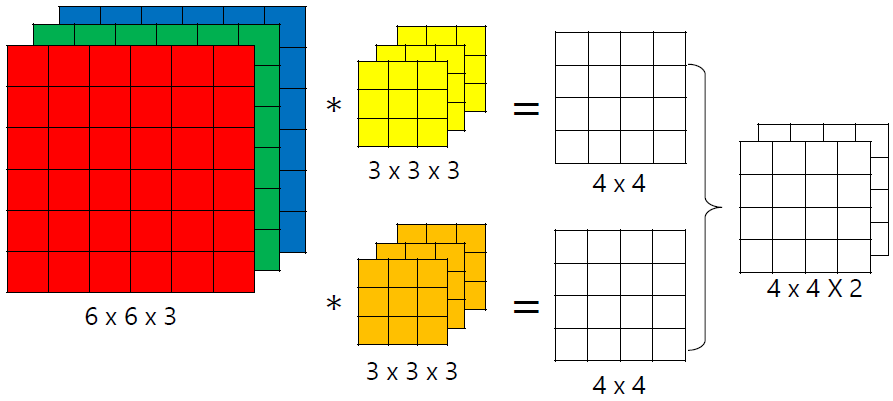
Here we can see the $6 \times 6 \times 3$ image and the $3 \times 3 \times 3$ filter. The last number is the number of channels and it matches between the image and the filter. To simplify the drawing the $3 \times 3 \times 3$ filter, we can draw it as a stack of three matrices. Sometimes, the filter is drawn as a threedimensional cube.

To compute the output of this convolution operation, we take the $3 \times 3 \times 3$ filter and first place it in that most upper left position. Notice that $3 \times 3 \times 3$ filter has 27 numbers. We take each of these 27 numbers and multiply them with the corresponding numbers from the red, green and blue channel. So, take the first nine numbers from red channel, then the three beneath it for the green channel, then three beneath it from the blue channel and multiply them with the corresponding 27 numbers covered by this yellow cube. Then, we add up all those numbers and this gives us the first number in the output. To compute the next output we take this cube and slide it over by one. Again we do the twenty-seven multiplications sum up 27 numbers and that gives us the next output.

We choose the first filter as 1,0,-1,1,0,-1,1,0,-1 (as we already did). This can be for a red color, for the green channel the values will be all zeros and for the blue filter as well. We stack these three matrices together to form our $3 \times 3 \times 3$ filter. Then, this would be a filter that detects vertical edges, but only in the red channel.

Knowing how to convolve on volumes is crucial for building convolutional neural networks. New question is, what if we want to detect vertical edges and horizontal edges and maybe even 45° or 70° as well. In other words, what if we want to use multiple filters at the same time?

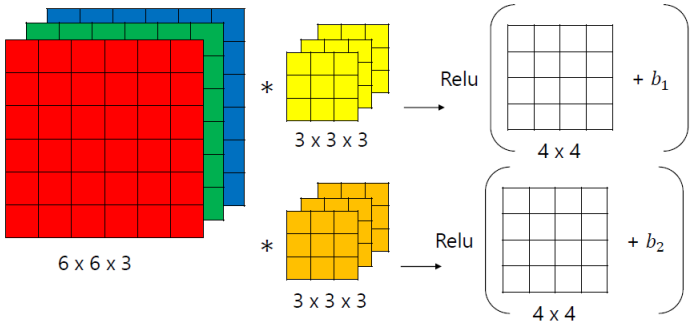
We can add a new second filter denoted by orange color, which could be a horizontal edge detector. Convolving an image with the filters gives us different 4×4 outputs. These two 4×4 outputs, can be stacked together obtaining a $4 \times 4 \times 2$ output volume. The volume can be drawn this as a box of a $4 \times 4 \times 2$ volume, where 2 denotes the fact that we used two different filters.



11.7 Convolution layer

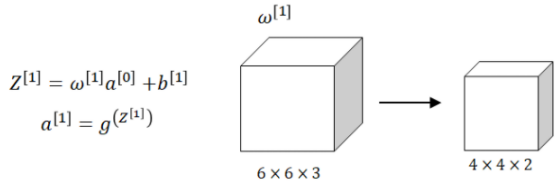
To turn what seen before into a convolutional neural network layer we need to add bias which is a scalar.

Then we will apply activation function, for example $ReLU$ activation function. The same we will do with the output we got by applying the second $3 \times 3 \times 3$ filter (kernel). So, once again we will add a different bias and then we will apply a $ReLU$ activation function. After adding a bias and after applying a $ReLU$ activation function dimensions of outputs remain the same, so we have two 4×4 matrices.



Next, we will repeat previous steps. Then we stack end up with a $4 \times 4 \times 2$ output. This computation have gone from $6 \times 6 \times 3$ to a $4 \times 4 \times 2$ and it represents one layer of a convolutional neural network.

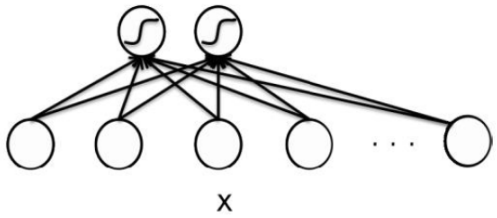
In neural networks one step of a forward propagation step was: $z^{[1]} = W^{[1]} \times a^{[0]} + b^{[1]}$ where $a^{[0]} = x$. Then we applied the non-linearity to get $a^{[1]} = g^{z^{[l]}}$. The same idea we will apply in a layer of the Convolutional Neural Network.



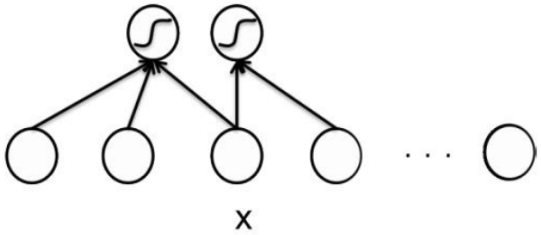
More at this link.

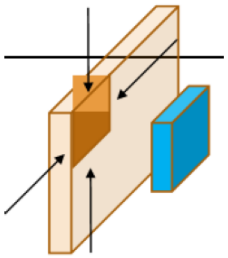
11.8 CNN Key idea

For Standard Neural Networks every neuron in the first hidden layer connects to all the neurons in the inputs:



In Convolutional Neural Networks, the connection patterns are that neurons can only look at adjacent input (for image inputs are pixels). In images the idea is to use the strong correlation among adjacent pixels.





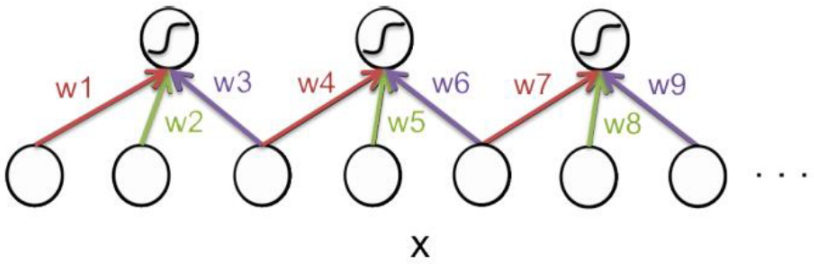
As we seen, a learned filter is used to scan an image: the same filters (the weights of the filter) are applied to different image locations.

It can be called weight sharing: some of the parameters in the model are constrained to be equal to each other.

For example, in the following layer of a network, we have the following constraints

$$w_1 = \mathrm{w}_4 = w_7;\ w_2 = w_5 = w_8 \text{ and } w_3 = w_6 = w_9$$

(edges that have the same color have the same weight):



11.9 Back Propagation Algorithm

Let's suppose we are using the same neural network architecture defined before. Then, backpropagation algoritm works as following:

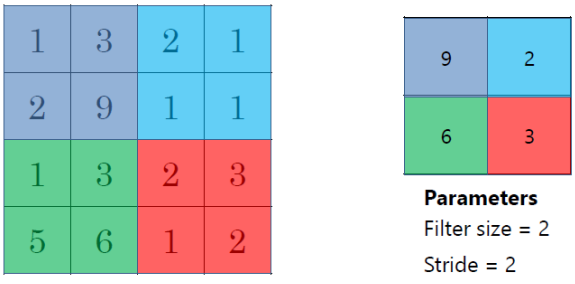
- In the forward pass, perform explicit computation with all the weights, $w_1, w_2, \ldots, w_9$ (some of them are the same);
- In the backward pass, compute the gradient $\frac{\partial J}{\partial w_1}, \ldots, \frac{\partial J}{\partial w_9}$, for all the weights;
- When updating the weights, use the average of the gradients from the shared weights, e.g., $w_1 = w_1 - \alpha \left(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7} \right), w_4 = w_4 - \alpha \left(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7} \right)$ and $w_7 = w_7 - \alpha \left(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7} \right)$.

11.10 Pooling Layer

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2×2. Global pooling acts on all the neurons of the convolutional layer. In addition, pooling may compute a max or an average. Max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Average pooling uses the average value from each of a cluster of neurons at the prior layer.

If you have multiple channels, pooling filter is applied at each channel independently.

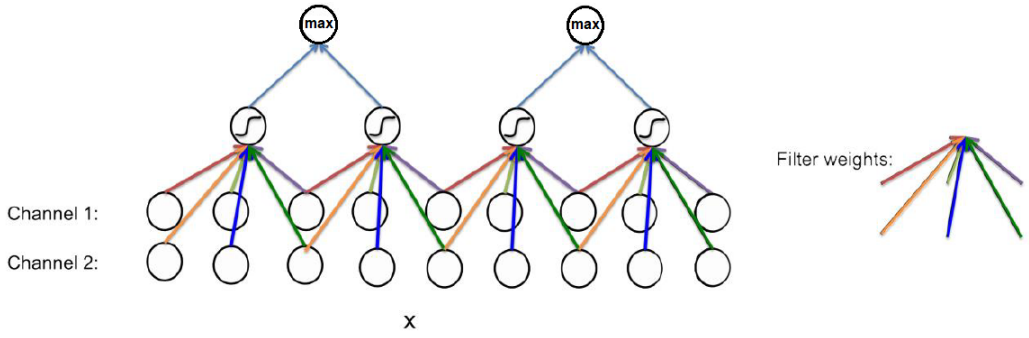
Example of **Max Pooling**:



The idea of max pooling is that if a particular feature is detected, then keep a high number. There are no weights to learn for this layer. If there are multiple channels, the pooling filter is applied independently for each of them.

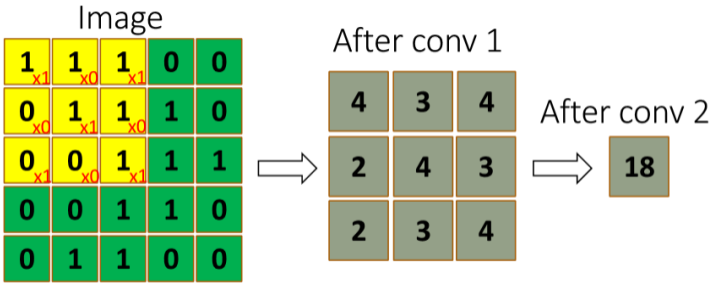
11.11 2D CNN with multi-channels inputs

Images typically have multiple channels (for example, RGB channels). In Convolutional Neural Netoworsk (CNN) 2D, the weights are often not shared across channels. The following figure demonstrates a convolutional architecture for an image with two channels:



11.12 Dimensionality reduction with convolutions

By applying convolutions to images (input or output of previous layers) the dimensionality get smaller and smaller. We should take care of this phenomenon, details are lost if the dimensionality get too small, recognition accuracy drop. For example, a 5x5 image, after a convolution with stride 1, padding 0, and kernel size 3x3 becomes 3x3, after another equal pass it becomes 1x1 (which is actually a single pixel, usually accuracy drops a lot).



As already seen, a lot of techniques can help to handle this problem, padding is one example.

11.13 Transfer Learning

Assume two datasets: T and S

Dataset S is:

- Fully annotated, plenty of images
- We can build a model h_s

Dataset T is:

- Small dataset of annotated images

We can use the model h_s to learn a better h_t. This is called **"Transfer Learning"**.

Even if our dataset T is not large, we can train a CNN for it without overfitting because we can pre-train a network on the dataset S. Then, there are two solutions: **Fine-tuning** and **CNN as features extractors**.

11.13.1 Fine-Tuning

Assume the parameters of S are already a good start near our final local optimum. Use them as the initial parameters for our new CNN for the target dataset. It is better to use when your source S is large and target T is medium. We can decide which layer we want to initialize.

Note: What layers to initialize and how?

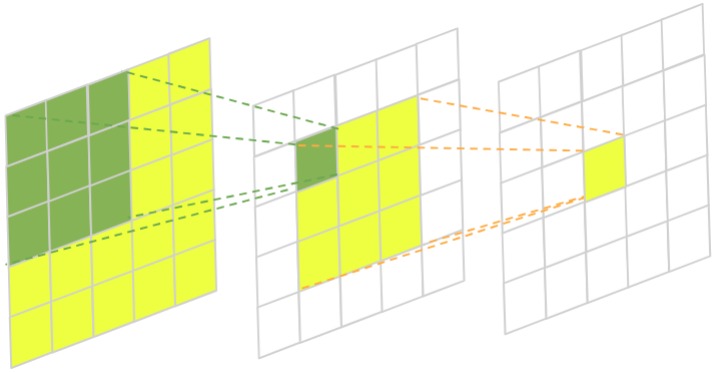
- When we have too few data, then fine-tune only last layers (Fully connected layers);
- When we have more data, then fine-tune also Convolutional Layers. Usually, the first layers capture low level information and does not change to much.

11.13.2 CNN as feature extractors

We use the trained CNNs as a feature extractor. The extracted features are used as input to a classifier (Deep or Traditional). This is useful when the target dataset T is very small.

11.14 Receptive Field

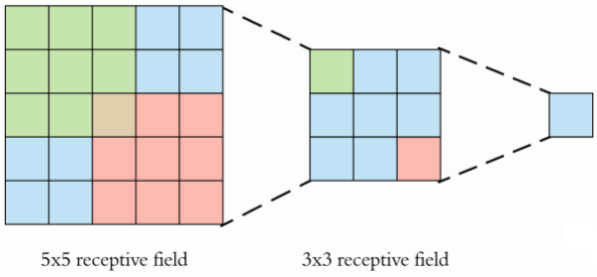
So what actually is the receptive field of a convolutional neural network? Formally, it is the region in the input space that a particular CNN's feature is affected by. More informally, it is the part of a tensor that after convolution results in a feature. So basically, it gives us an idea of where we're getting our results from as data flows through the layers of the network. To further illuminate the concept let's have a look at this illustration:



For example: Two 3x3 filters have the receptive field of one 5x5 (see figure). Three 3x3 filters have the receptive field of one 7x7.

Deeper stacks of smaller filters likely more powerful than single large filter. Three more nonlinearities for the same "size" of pattern learning.

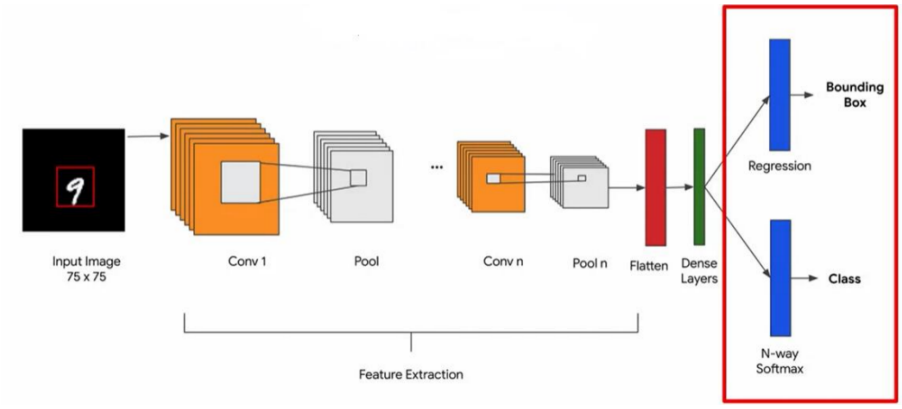
Fewer parameters: a stacks of three filters of 3x3 has a total of 27 parameters. One large 7x7 filter has 49 parameters. The receptive field is the same.

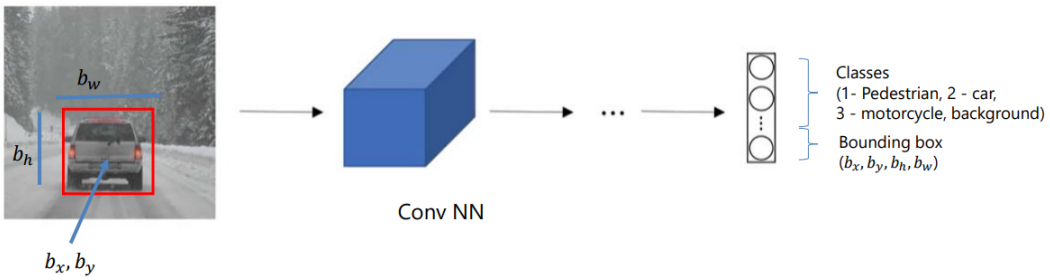


A detailed explanation with formulas can be found here.

11.15 Object Detection

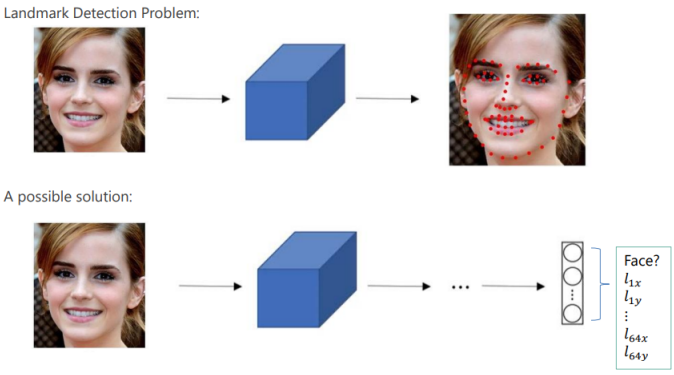
Object detection is a sub-field of computer vision. We want to detect an object (inside its bounding box) and its class. We want to classify and to localize the object.





How loss is handled: if there is an object in an image, loss will take care to all the output values, otherwise loss will take car just for the P_c value.

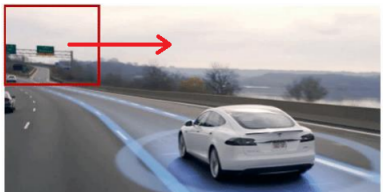
11.15.1 Landmark detection problem: In many computer vision applications, neural network often needs to recognize essential points of interest (than bounding box) in the input image. We refer to these points as landmarks. In such applications, we want the neural network to output coordinates (x, y) of landmark points than those of bounding boxes.



The same idea can also be applied to the entire full body.

11.15.2 How to perform object detection

The idea is to use a **sliding window technique** to detect the object. Each window is used as input of our trained Convolutional Neural Network. In order to deal with different scales we just scan the image with windows with different sizes.

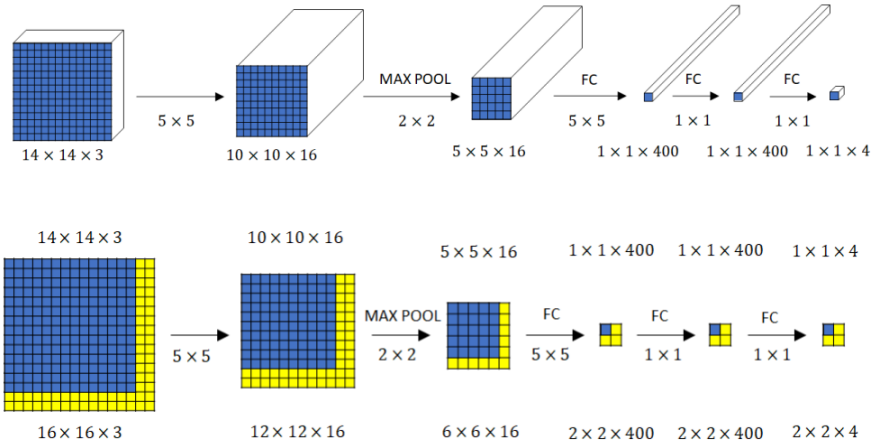


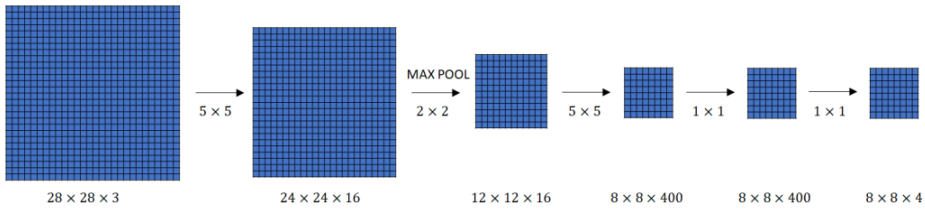
The idea is to feed the network with one portion of the image at at time until an object is detected (or not) and the entire image is scanned. So this algorithm is called Sliding Windows Detection because you take these windows, these square boxes, and slide them across the entire image and classify every square region with some stride as containing the searched object or not. There's a huge disadvantage of the Sliding Windows Detection algorithm, which is the computational cost. Because you're cropping out so many different square regions in the image and running each of them independently through a ConvNet and if you use a very coarse stride, a very big stride, a very big step size, then that will reduce the number of windows you need to pass through the cofinite, but that courser granularity may hurt performance. Whereas if you use a very fine granularity or a very small stride, then the huge number of all these little regions you're passing through the ConvNet means that means there is a very high computational cost. The solution is to perform the algorithm in a "convolutional" way.

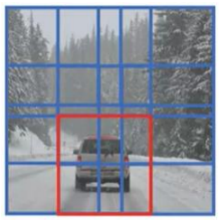
More about that here.

11.15.3 Convolution implementation of Sliding Windows

Instead of implementing a sliding window algorithm naively, we can do in a "convolutional" way. More at this link.







Problem: position of the bounding box is not so accurate

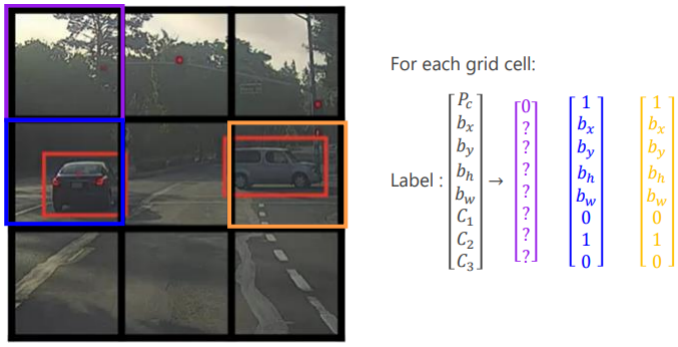
11.15.4 YOLO algorithm

YOLO algorithm is an algorithm based on regression, instead of selecting the interesting part of an Image, it predicts classes and bounding boxes for the whole image in one run of the Algorithm. To understand the YOLO algorithm, first we need to understand what is actually being predicted. Ultimately, we aim to predict a class of an object and the bounding box specifying object location. Each bounding box can be described using four descriptors:

- Center of the box (bx, by)
- Width (bw)
- Height (bh)
- Value c corresponding to the class of an object

Along with that we predict a real number pc, which is the probability that there is an object in the bounding box. YOLO doesn't search for interested regions in the input image that could contain an object, instead it splits the image into cells, Each cell is then responsible for predicting K bounding boxes.

Here in this example we are using a grid of 3x3, but you can use, for example, a grid of 19x19 (tipically used valued).

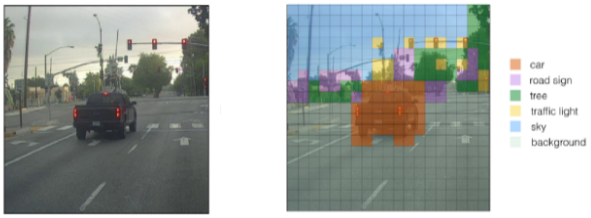


Therefore, for each grid cell the output is 8. Thus, the target output of the image is 3x3x8. Note each object is associated to a single grid cell. The location of the centroid object determinates the associated grid cell.

An object is considered to lie in a specific cell only if the center co-ordinates of the anchor box lie in that cell. Due to this property the center co-ordinates are always calculated relative to the cell whereas the height and width are calculated relative to the whole Image size.

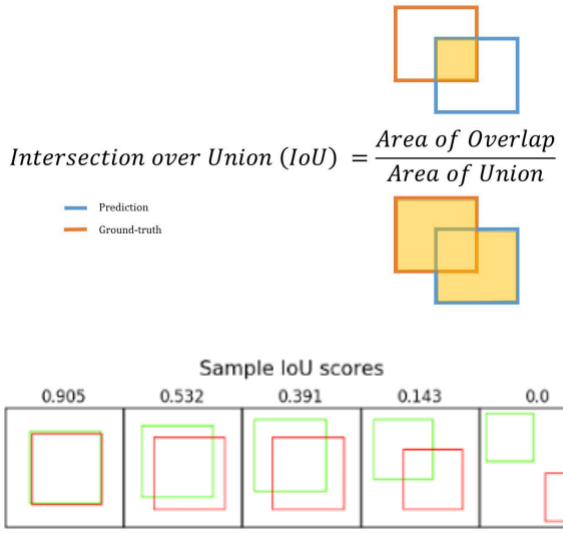
During the one pass of forwards propagation, YOLO determines the probability that the cell contains a certain class.

The class with the maximum probability is chosen and assigned to that particular grid cell. Similar process happens for all the grid cells present in the image. After computing the above class probabilities, the image may look like this:



This shows the before and after of predicting the class probabilities for each grid cell. After predicting the class probabilities, the next step is Non-max suppression, it helps the algorithm to get rid of the unnecessary anchor boxes, like you can see that in the figure below, there are numerous anchor boxes calculated based on the class probabilities.

To resolve this problem **Non-max suppression** eliminates the bounding boxes that are very close by preforming the IoU (Intersection over Union) with the one having the highest class probability among them.



It calculates the value of IoU for all the bounding boxes respective to the one having the highest class probability, it then rejects the bounding boxes whose value of IoU is greater than a threshold. It signifies that those two bounding boxes are covering the same object but the other one has a low probabilty for the same, thus it is eliminated. Once done, algorithm finds the bunding box with next highest class probabilities and does the same process, it is done until we are left with all the different bounding boxes.

After this, almost all of our work is done, the algorithm finally outputs the required vector showing the details of the bounding box of the respective class.

11.16 UpSampling

Upsampling (increase image size) is usually used to improve/enlarge images feeded as input to a CNN. The most used techniques are Nearest-techniques (copies values from nearest pixel to the new one) or Bilinear-techniques (to interpolate from nearby pixels to create a new one). Other advanced techniques do exist.

11.17 Residual Networks (ResNets)

A residual neural network (ResNet) is an artificial neural network that builds on constructs known from pyramidal cells in the cerebral cortex. Residual neural networks do this by utilizing **skip connections**, or **shortcuts** to jump over some layers.

Typical ResNet models are implemented with double or triple layer skips that contain nonlinearities (ReLU) and batch normalization in between. An additional weight matrix may be used to learn the skip weights; these models are known as HighwayNets. Models with several parallel skips are referred to as DenseNets.

In the context of residual neural networks, a non-residual network may be described as a plain network. There are two main reasons to add skip connections: to avoid the problem of **vanishing gradients**, or to mitigate the **degradation problem** (accuracy saturation); where adding more layers to a suitably deep model leads to higher training error.

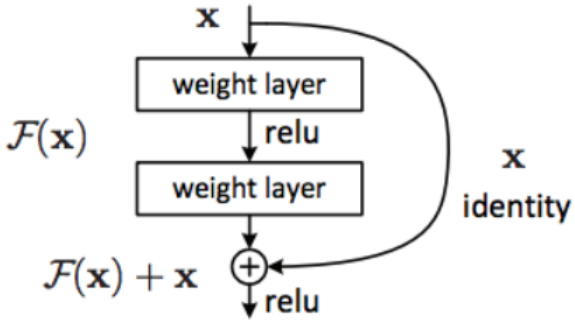
A good introduction about Res Nets can be found in this article.

11.17.1 The problem of Vanishing Gradients: the vanishing gradient problem is encountered when training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0,1], and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the early layers in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the early layers train very slowly. When activation functions are used whose derivatives can take on larger values, one risks encountering the related exploding gradient problem.

More about vanishing gradients in this video.

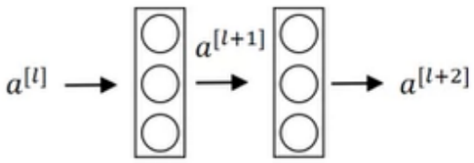
11.18 Residual Block

By using a Residual Networks you can design very deep Neural Network. The following image shows the building block of a ResNet: a **Residual Block**.



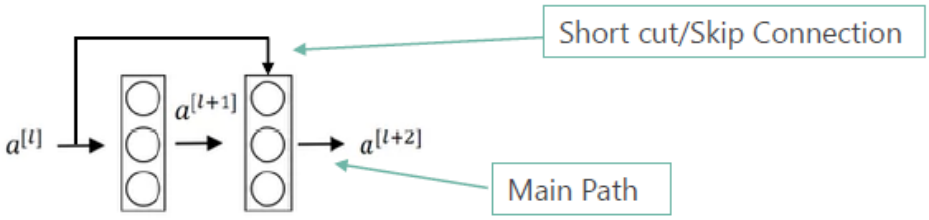
the most important modification to understand is the "Skip Connection", identity mapping. This identity mapping does not have any parameters and is just there to add the output from the previous layer to the layer ahead. However, sometimes x and F(x) will not have the same dimension. Recall that a convolution operation typically shrinks the spatial resolution of an image, e.g. a 3x3 convolution on a 32 x 32 image results in a 30 x 30 image. The identity mapping is multiplied by a linear projection W to expand the channels of shortcut to match the residual. This allows for the input x and F(x) to be combined as input to the next layer. The Skip Connections between layers add the outputs from previous layers to the outputs of stacked layers. This results in the ability to train much deeper networks than what was previously possible. A similar approach to ResNets is known as "highway networks". These networks also implement a skip connection, however, similar to an LSTM these skip connections are passed through parametric gates. These gates determine how much information passes through the skip connection.

For example, from this:



$$a^{l} \rightarrow \text{linear} \rightarrow \text{ReLU} \rightarrow a^{l+1} \rightarrow \text{linear} \rightarrow \text{ReLu} \rightarrow a^{l+2}$$

$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \rightarrow a^{[l+1]} = g\left(z^{[l+1]}\right) \rightarrow z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]} \rightarrow a^{[l+2]} = g\left(z^{[l+2]}\right)$$

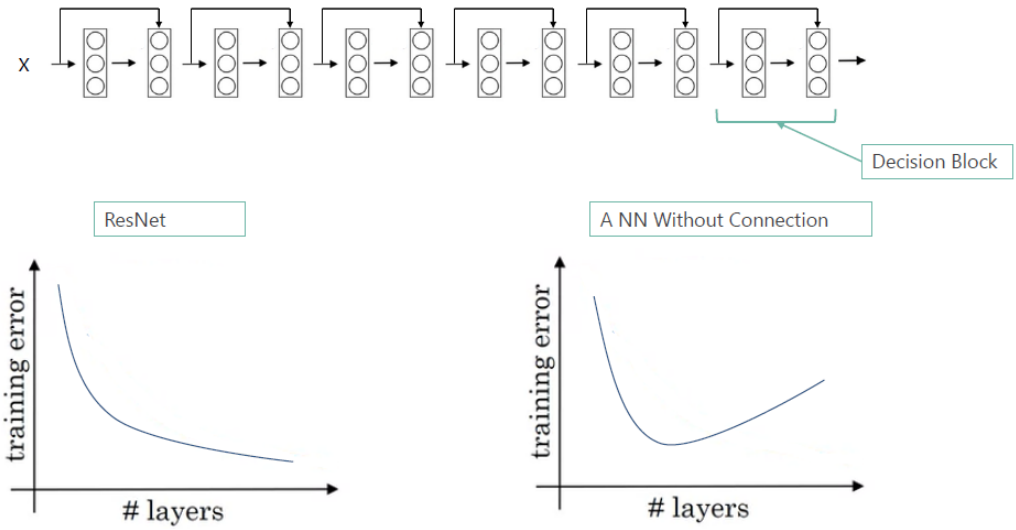


By adding a skip-connection it becomes:

$$z^{[l+1]} = W^{[l+1]}a^{[l]}+b^{[l+1]} \rightarrow a^{[l+1]} = g\left(z^{[l+1]}\right) \rightarrow z^{[l+2]} = W^{[l+2]}a^{[l+1]}+b^{[l+2]} \rightarrow a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$$

Note: $z^{[l+2]}$ and $a^{[l]}$ must have the same dimensionality. If not, you can add an extra weight matrix W_s (i.e $g\left(z^{[l+2]} + W_s a^{[l]}\right)$) in order to adjust the dimensionality issue.

Example of how to turn a "plain network" into a residual neural network.



If you train a res-net with standard optimization algorithms you find out that training error is better compared to networks without connection.

11.19 1x1 convolutions and their utility

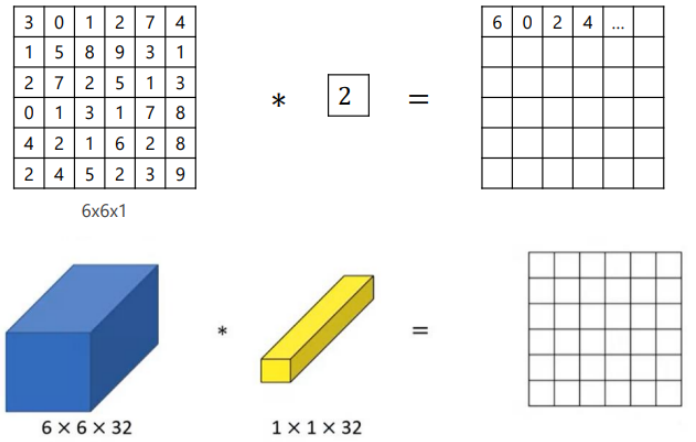
As the name suggests, the 1x1 convolution operation involves convolving the input with filters of size 1x1, usually with zero-padding and stride of 1. Taking an example, let us suppose we have a (general-purpose) convolutional layer which outputs a tensor of shape (B, K, H, W) where

- B represents the batch-size.
- K is the number of convolutional filters or kernels

- H, W are the spatial dimensions i.e. Height and Width.

In addition, we specify a filter size that we want to work with, which is a single number for a square filter i.e. size=3 implies a 3x3 filter. Feeding this tensor into our 1x1 convolution layer with F filters (zero-padding and stride 1), we will get an output of shape (B, F, H, W) changing our filter dimension from K to F. Sweet! Now depending on whether F is less or greater than K, we have either decreased or increased the dimensionality of our input in the filter space without applying any spatial transformation. All this using the 1x1 convolution operation.

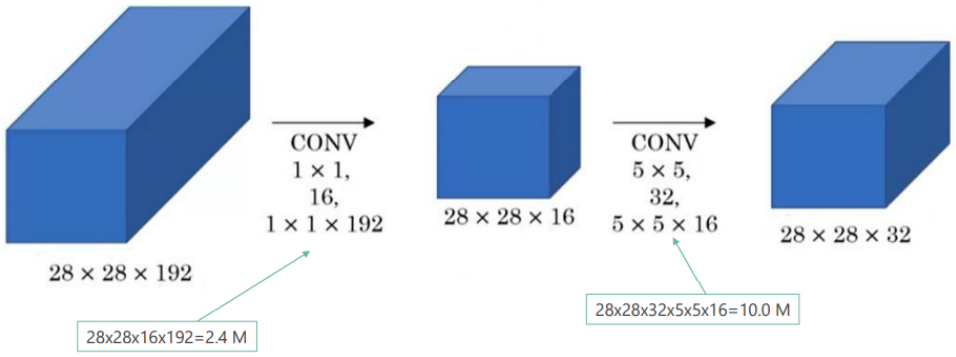
But how was this any different from a regular convolution operation? In a regular convolution operation, we usually have a larger filter size like a 3x3 or 5x5 (or even 7x7) kernels which then generally entail some kind of padding to the input which in turn transforms it's spatial dimensions of H x W to some H' x W'.



Benefits: In CNNs, we often use some kind of pooling operations to spatially downsample the activation maps of a convolution output, in order to keep things from becoming intractable. The danger of intractability is due to the number of generated activation maps which increases or rather blows up dramatically, proportional to the depth of a CNN.

That is, the deeper a network, the larger the number of activation maps it generates. The problem is further increased if the convolution operation is using large-sized filters like 5x5 or 7x7 filters, resulting in a significantly high number of parameters. Spatial downsampling (through pooling) is achieved by aggregating information along the spatial dimensions by reducing the height and width of the input at every step. While it maintains important spatial features to some extent, there does exist a trade-off between down-sampling and information loss. Bottom line: we can only apply pooling to a certain extent.

The 1x1 convolution can be used to address this issue by offering filter-wise pooling, acting as a projection layer that pools (or projects) information across channels and enables dimensionality reduction by reducing the number of filters whilst retaining important, feature-related information.



11.19.1 Inception Network

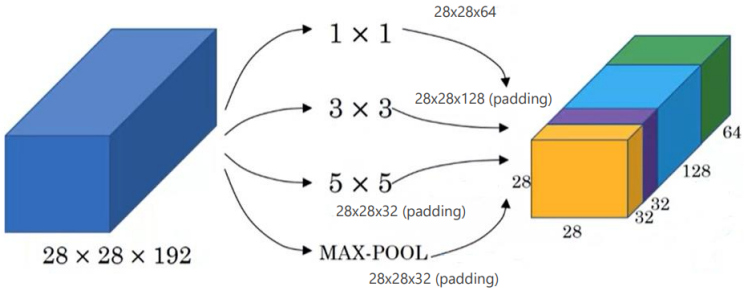
The Inception network was an important milestone in the development of CNN classifiers. Prior to its success, most popular CNNs just stacked convolution layers deeper and deeper, hoping to get better performance.

Premise: Salient parts in the image can have extremely large variation in size. Because of this huge variation in the location of the information, choosing the right kernel size for the convolution operation becomes tough. A larger kernel is preferred for information that is distributed more globally, and a smaller kernel is preferred for information that is distributed more locally.

Very deep networks are prone to overfitting. It also hard to pass gradient updates through the entire network.

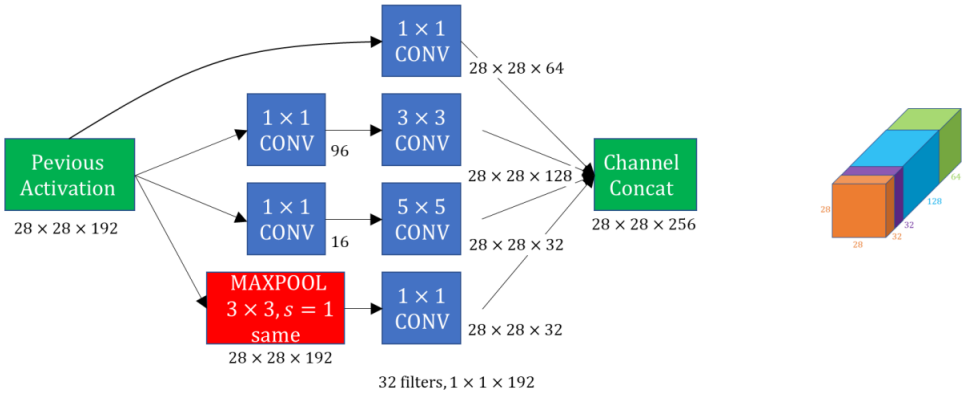
Naively stacking large convolution operations is computationally expensive.

Solution: Why not have filters with multiple sizes operate on the same level? The network essentially would get a bit "wider" rather than "deeper". The authors designed the inception module to reflect the same. The below image is the "naive" inception module. It performs convolution on an input, with 3 different sizes of filters (1x1, 3x3, 5x5). Additionally, max pooling is also performed. The outputs are concatenated and sent to the next inception module.

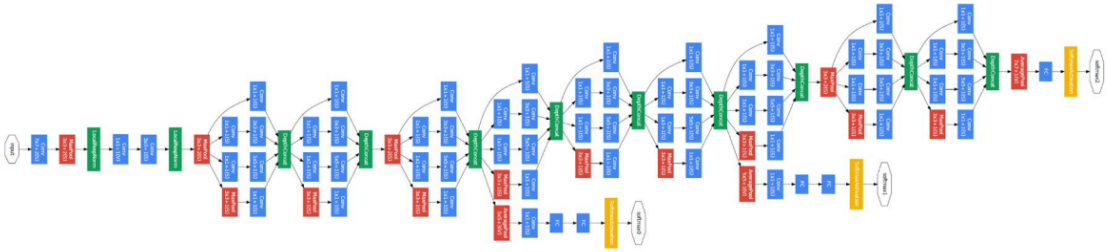


As stated before, deep neural networks are computationally expensive. To make it cheaper, the authors limit the number of input channels by adding an extra 1x1 convolution before the 3x3 and 5x5 convolutions, this is why we saw them before, they are a fundamental building block of the inception layer. Though adding an extra operation may seem counterintuitive, 1x1 convolutions are far more cheaper than 5x5 convolutions, and the

reduced number of input channels also help. Do note that however, the 1x1 convolution is introduced after the max pooling layer, rather than before.



Using the dimension reduced inception module, a neural network architecture was built. This was popularly known as GoogLeNet (Inception v1). The architecture is shown below:



Needless to say, it is a pretty deep classifier. As with any very deep network, it is subject to the vanishing gradient problem. To prevent the middle part of the network from "dying out", the authors introduced two auxiliary classifiers (first and second yellow boxes). They essentially applied softmax to the outputs of two of the inception modules, and computed an auxiliary loss over the same labels. The total loss function is a weighted sum of the auxiliary loss and the real loss. Weight value used in the paper was 0.3 for each auxiliary loss.

Inception Network v2: Inception v2 and Inception v3 were presented in the same paper. The authors proposed a number of upgrades which increased the accuracy and reduced the computational complexity. Inception v2 explores the following:

- Reduce representational bottleneck. The intuition was that, neural networks perform better when convolutions didn't alter the dimensions of the input drastically. Reducing the dimensions too much may cause loss of information, known as a "representational bottleneck"
- Using smart factorization methods, convolutions can be made more efficient in terms of computational complexity.

The proposed v2 solution factorize 5x5 convolution to two 3x3 convolution operations to improve computational speed. Although this may seem counterintuitive, a 5x5 convolution

is 2.78 times more expensive than a 3x3 convolution. So stacking two 3x3 convolutions infact leads to a boost in performance.

Moreover, they factorize convolutions of filter size nxn to a combination of 1xn and nx1 convolutions. For example, a 3x3 convolution is equivalent to first performing a 1x3 convolution, and then performing a 3x1 convolution on its output. They found this method to be 33% more cheaper than the single 3x3 convolution.

The filter banks in the module were expanded (made wider instead of deeper) to remove the representational bottleneck. If the module was made deeper instead, there would be excessive reduction in dimensions, and hence loss of information.

The above three principles were used to build three different types of inception modules (Let's call them modules A,B and C in the order they were introduced. These names are introduced for clarity, and not the official names).

Other performance and accuracy updates were introduced in v3 and v4. More about this in this link.

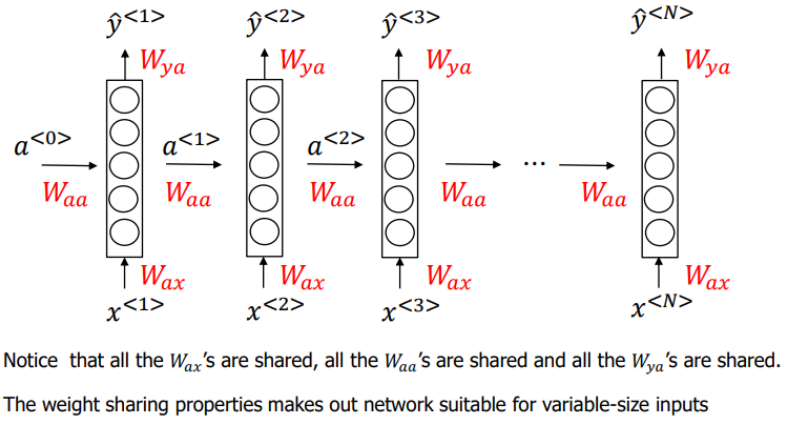
12 Recurrent Neural Networks

A Recurrent Neural Network is a neural network that is specialized for processing a **sequence of data** $x(t) = x(1), ..., x(T)$ with the time step index t ranging from 1 to T. For tasks that involve sequential inputs, such as speech and language, it is often better to use RNNs.

In a NLP problem, if you want to predict the next word in a sentence it is important to know the words before it.

RNNs are called **recurrent** because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.

A good intuition explanation can be found here. You can find more informations in this video and in this article. A good notation summary can be found here.



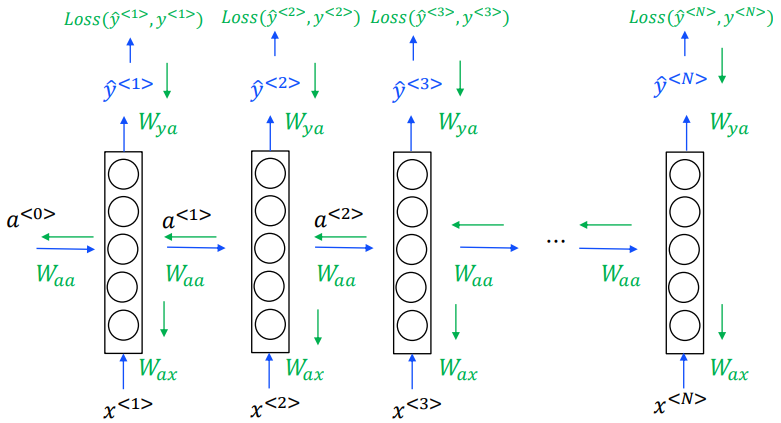
- Notice that all the W_{ax}'s are shared, all the W_{aa}'s are shared and all the W_{ya}'s are shared.
- The weight sharing properties makes out network suitable for variable-size inputs

12.1 Forward-propagation and Back-propagation

The gradient computation involves performing a forward propagation pass moving left to right through the network shown above followed by a backward propagation pass moving right to left through the network.

The runtime is $O(T)$ and cannot be reduced by parallelization because the forward propagation network is inherently sequential; each time-step may be computed only after the previous one. States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also O(T). The back-propagation algorithm applied to the unrolled network with O(T) cost is called back-propagation through time (BPTT).

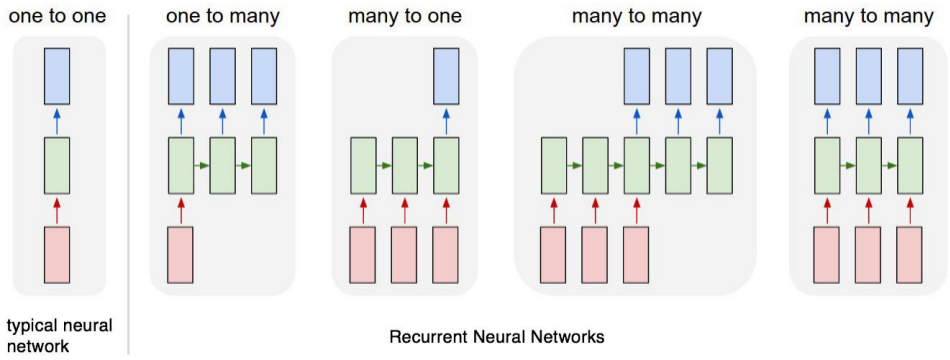
Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps.



Backpropagation Through Time (BTT)

12.2 Different RNN sequence types

Here's an image showing different RNN types:



12.2.1 Advantages and Drawbacks

Advantages and drawbacks can be summarized as follow:

Advantages:

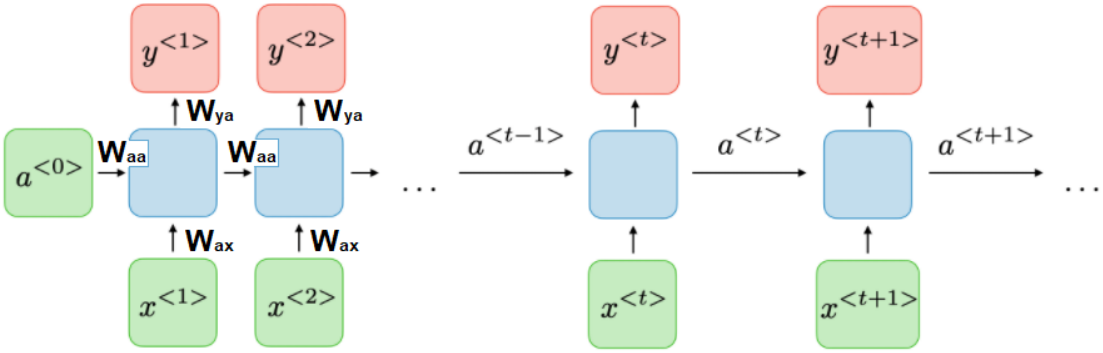
- Possibility of processing input of any length;
- Model size not increasing with size of the input;
- Computation takes into account historical information;
- Weights are shared across time.

Drawbacks:

- Computation being slow;
- Difficulty of accessing informations from a long time ago;
- Cannot consider any future input for the current state.

12.3 RNN notation and architecture

The tipical RNN structure is the following:



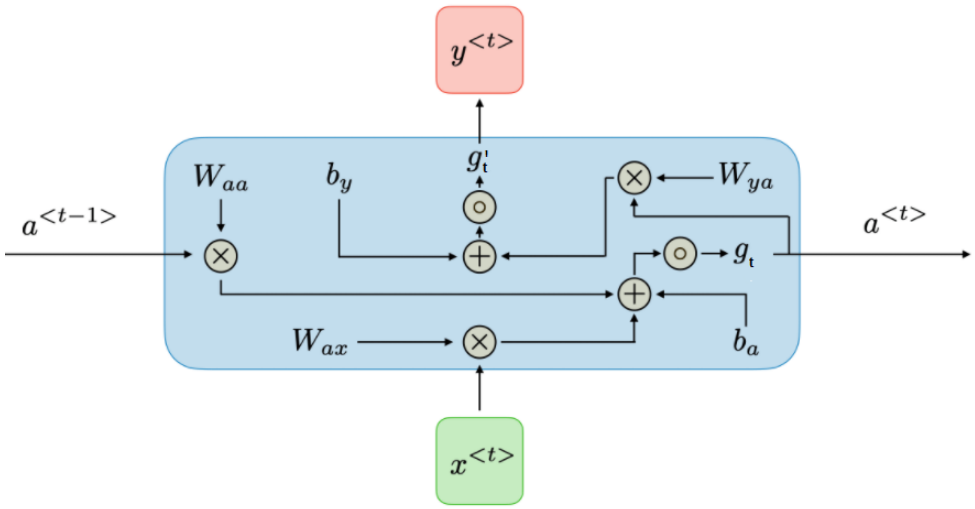
For each timestep t the activation $a^{<t>}$ and output $y^{<t>}$ are expressed as follow:

$$a^{<t>} = g_t\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) = g_t\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right)$$

$$\hat{y}^{<t>} = g'_t\left(W_{ya}a^{<t>} + b_y\right) = g'_t\left(W_y a^{<t>} + b_y\right)$$

Where g_t is usually a ReLU/Tanh function and g'_t is usually a Sigmoid function. $W_{ax}, W_{aa}, W_{ay}, b_a, b_y$ are weights which are shared temporally.

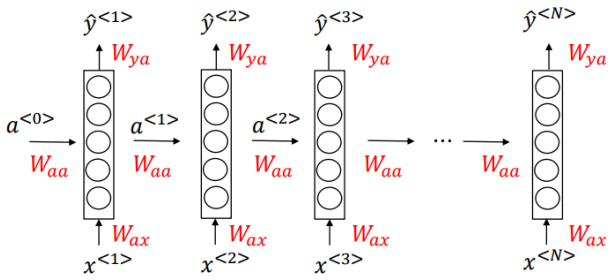
Inside the a layer, we have this structure:



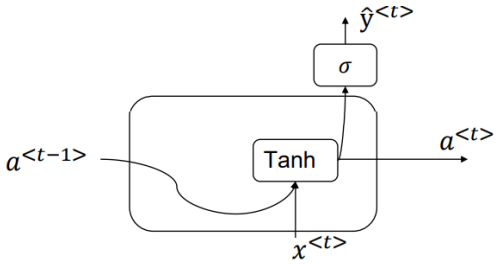
Where $W_a = [W_{aa}W_{ax}]$ and $[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$

In fact,

$$[W_{aa}W_{ax}]\begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} = W_{aa}a^{<t-1>} + W_{ax}x^{<t>}$$

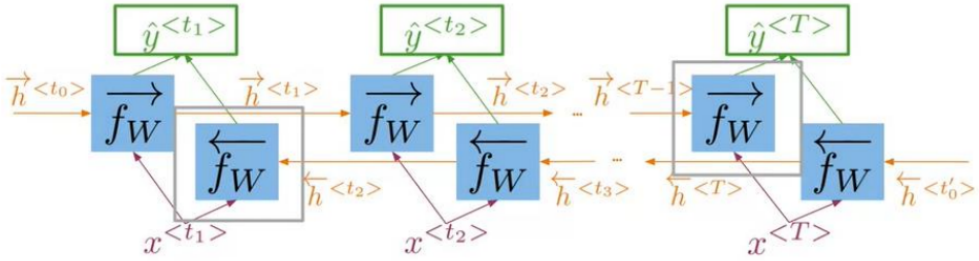


12.3.1 A Schematization example: $a^{<t>} = \tanh\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right)$ $\hat{y}^{<t>} = \sigma\left(W_y a^{<t>} + b_y\right)$



12.4 Bi-Directional RNNs

Bidirectional recurrent neural networks (BRNN) connect two hidden layers of opposite directions to the same output. With this form of generative deep learning, the output layer can get information from past (backwards) and future (forward) states simultaneously. BRNN are especially useful when the context of the input is needed. For example, in handwriting recognition, the performance can be enhanced by knowledge of the letters located before and after the current letter (only "before" is available in classic RNN).



$$\hat{y}^{<t>} = g(W_y[\overrightarrow{h}^{<t>}, \overleftarrow{h}^{<t>}] + b_y)$$

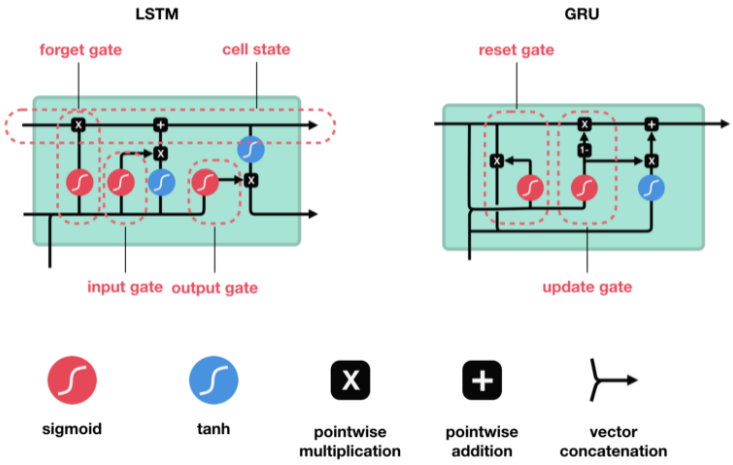
12.5 Gated Recurrent Unit (GRU) & LSTM

Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.

During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Remember that gradients are those values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.

So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory.

Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU.



LSTM 's and GRU's were created as the solution to short-term memory. They have internal mechanisms called **gates** that can regulate the flow of information.

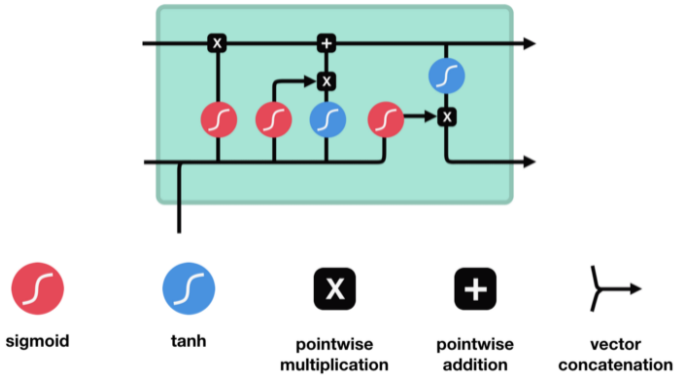
These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state of the art results based on recurrent neural networks are achieved with these two networks. LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation.

The special thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction.

Intuition: you read a review about a product, you will remember only important parts, not all the text! That is essentially what an LSTM or GRU does. It can learn to keep only relevant information to make predictions, and forget non relevant data.

12.6 LSTM

An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM's cells.



These operations are used to allow the LSTM to keep or forget information. Now looking at these operations can get a little overwhelming so we'll go over this step by step.

The core concept of LSTM's are the cell state, and it's various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make it's way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information get's added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.

First, we have the **forget gate**. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

To update the cell state, we have the **input gate**. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output

with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

Now we should have enough information to calculate the **cell state**. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.

To review, the Forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

12.7 GRU

So now we know how an LSTM work, let's briefly look at the GRU. The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.

The **update gate** acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.

The **reset gate** is another gate is used to decide how much past information to forget. And that's a GRU. GRU's has fewer tensor operations; therefore, they are a little speedier to train then LSTM's. There isn't a clear winner which one is better. Researchers and engineers usually try both to determine which one works better for their use case.

More informations about LSTMs in this article and in this video. Other details here.

12.8 RNN vs ResNet

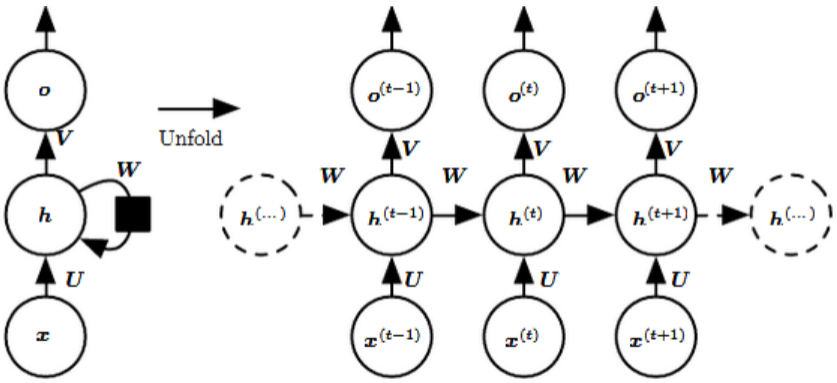
They can be seen as similar, but:

- **Residual Neural Networks** are very deep networks that implement 'shortcut' connections across multiple layers in order to preserve context as depth increases. Layers in a residual neural net have input from the layer before it and the optional, less processed data, from X layers higher. This prevents early or deep layers from 'dying' due to converging loss.
- **Recurrent Neural Networks** are networks that contain a directed cycle which when computed is 'unrolled' through time. Layers in recurrent neural network have input from the layer before it and the optional, time dependent extra input. This provides situational context for things like natural language processing.

More informations in this paper.

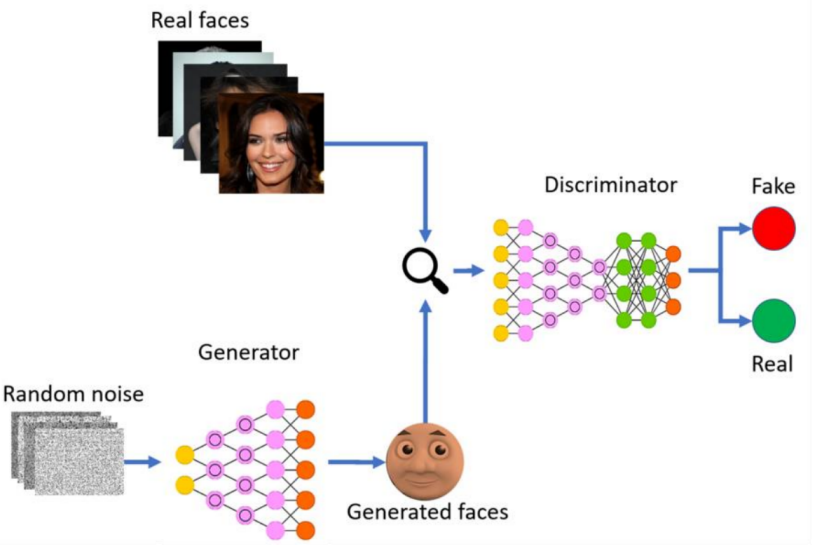
12.9 Unrolling a RNN

The left side of the above diagram shows a notation of an RNN and on the right side an RNN being unrolled (or unfolded) into a full network. By unrolling we mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 3 words, the network would be unrolled into a 3-layer neural network, one layer for each word.



13 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are an exciting recent innovation in Machine Learning. GANs are generative models: they create new data instances that resemble your training data. For example, GANs can create images that look like photographs of human faces, even though the faces don't belong to any real person. We can summarize a GAN network structure as follow:

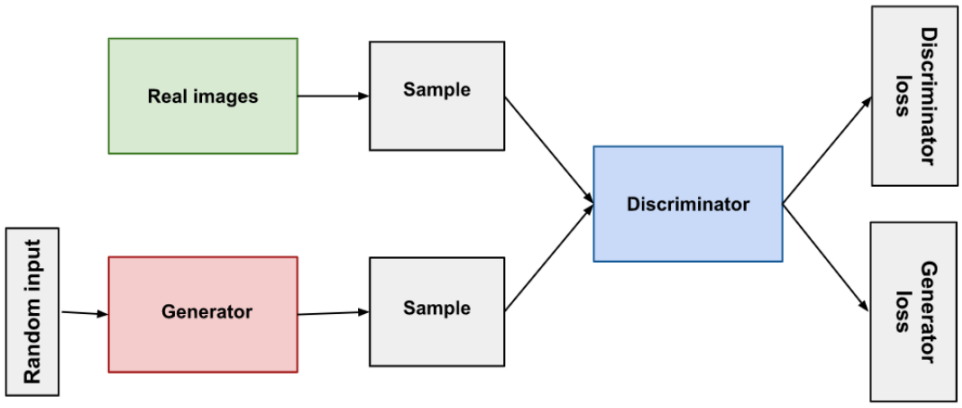


When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:

As training progresses, the generator gets closer to producing output that can fool the discriminator:

Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

A more formal scheme of a GAN network is the following one:



13.1 The Discriminator

The discriminator in a GAN is simply a classifier. It tries to distinguish real data from the data created by the generator. It could use any appropriated network architectures to the type of data it's classifying. The discriminator's training data comes from two sources:

- **Real data instances**, such as real pictures of people. The discriminator uses these instances as positive examples during training.
- **Fake data instances** created by the generator. The discriminator uses these instances as negative examples during training.

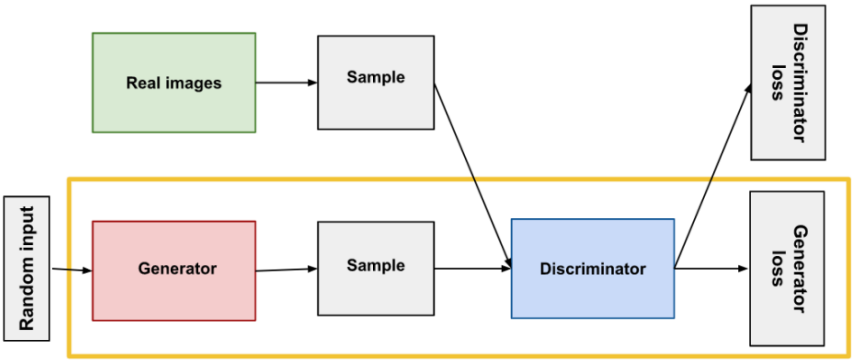
In Figure, the two "Sample" boxes represent these two data sources feeding into the discriminator. During discriminator training the generator does not train.

13.2 The Generator

The generator part of a GAN learns to create fake data by incorporating feedback from the discriminator. It learns to make the discriminator classify its output as real.

Generator training requires tighter integration between the generator and the discriminator than discriminator training requires. The portion of the GAN that trains the generator includes:

- random input;



- generator network, which transforms the random input into a data instance;
- discriminator network, which classifies the generated data;
- discriminator output;
- generator loss, which penalizes the generator for failing to fool the discriminator.

Neural networks need some form of input. Normally we input data that we want to do something with, like an instance that we want to classify or make a prediction about. But what do we use as input for a network that outputs entirely new data instances? In its most basic form, a GAN takes random noise as its input. The generator then transforms this noise into a meaningful output.

13.3 Training a GAN

To train a Neural Network, we alter the net's weights to reduce the error or loss of its output. In our GAN, however, the generator is not directly connected to the loss that we're trying to affect. The generator feeds into the discriminator network, and the discriminator produces the output we're trying to affect. The generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake.

Because a GAN contains two separately trained networks, its training algorithm must address two complications:

- GANs must juggle two different kinds of training (generator and discriminator);
- GAN convergence is hard to identify;

The generator and the discriminator have different training processes. So how do we train the GAN as a whole? GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs;
2. The generator trains for one or more epochs;

Repeat steps 1 and 2 to continue to train the generator and discriminator networks.

We keep the generator constant during the discriminator training phase. Similarly, we keep the discriminator constant during the generator training phase. It's this back and forth that allows GANs to tackle otherwise intractable generative problems.

Convergence: As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator flips a coin to make its prediction.

13.4 Loss function in GANs

A GAN can have two loss functions: one for generator training and one for discriminator training. How those two can work together? To better understand the Combined Loss proposed to train GANs, it is useful to recall the Cost Function in a Logistic Regression.

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)}),\}$

The goal is to find $\boldsymbol{\theta}$ so $\begin{cases} h_\theta(x) \geq 0.5 & if \;\; y = 1 \\ h_\theta(x) < 0.5 & if \;\; y = 0 \end{cases}$

We can define the following Cost Function:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}\,(h_\theta(x), y)$$

where $\text{Cost}\,(h_\theta(x), y) = -y\log\big(h_\theta(x)\big) - (1-y)\log\big(1 - h_\theta(x)\big)$

This cost function is convex and is derivable respect to θ, so we can use Gradient descent to solve the minimization problem.

Cost Function:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} -y\log\big(h_\theta(x)\big) - (1-y)\log\big(1 - h_\theta(x)\big)$$

Let's define positive and negative examples, the Cost Function will be:

$$J(\theta) = -\frac{1}{m_{pos} + m_{neg}}\left[\sum_{i=1}^{m_{pos}} \log\big(h_\theta(x)\big) + \sum_{i=1}^{m_{neg}} \log\big(1 - h_\theta(x)\big)\right]$$

This Cost function is also called Binary Cross-Entropy Loss!

Our objective is to maximize the chance to recognize real images as real and generated images as fake:

$$\max_D V(D) = \underbrace{\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})]}_{\text{recognize real images better}} + \underbrace{\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]}_{\text{recognize generated images better}}$$

- $\boldsymbol{D}(\boldsymbol{x})$ is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- $G(z)$ is the generator's output when given noise z.
- $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).

Our objective is to generate images with the highest possible value of D(x) to fool the discriminator:

$$\min_G V(G) = \underbrace{\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]}_{\text{Optimize G that can fool the discriminator the most.}}$$

13.4.1 Min-Max Loss: We often define GAN as a minimax game which G wants to minimize V while D wants to maximize it:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

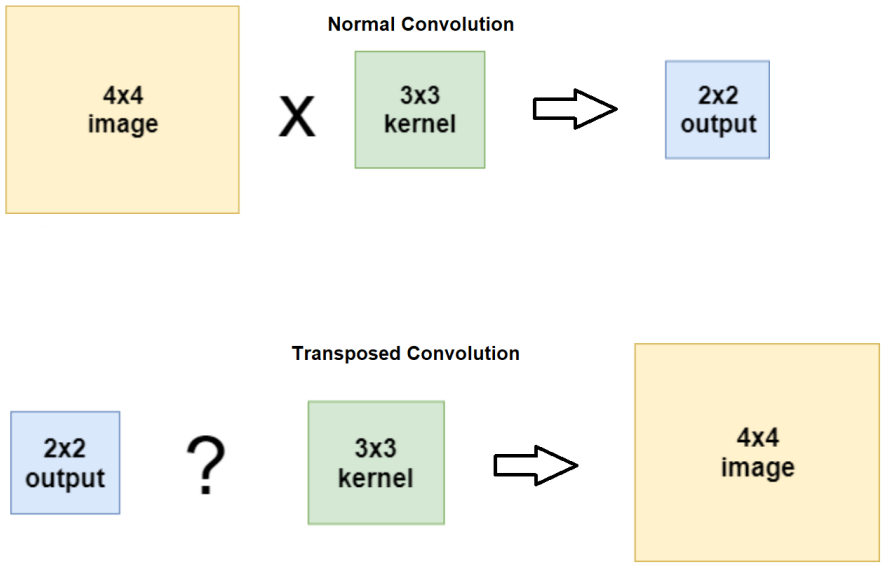
Once both objective functions are defined, they are learned jointly by the alternating gradient descent.

We fix the generator model's parameters and perform a single iteration of gradient descent on the discriminator using the real and the generated images.

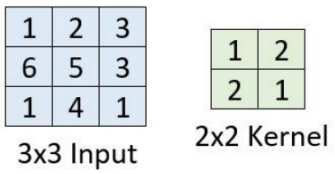
Then we switch sides. Fix the discriminator and train the generator for another single iteration. We train both networks in alternating steps until the generator produces good quality images.

13.5 Transposed Convolutions

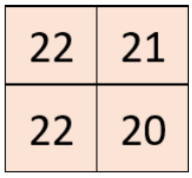
Transposed Convolution layer is a variation of classical Convolution layer, which can be used for upsampling images. To understand this variation, let's see the image below:

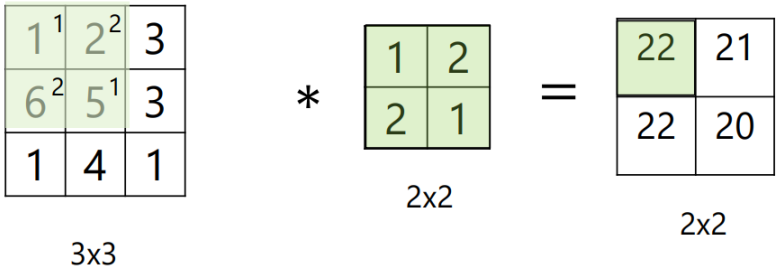


Let's now see how we can represent the normal convolution by means of a Convolution Matrix. Suppose that we're performing convolutions with a 2x2 kernel on a 3x3 input image, like this:



With our understanding of how regular convolutions work, it's not surprising to find that we'll end up with a 2x2 output or feature map:



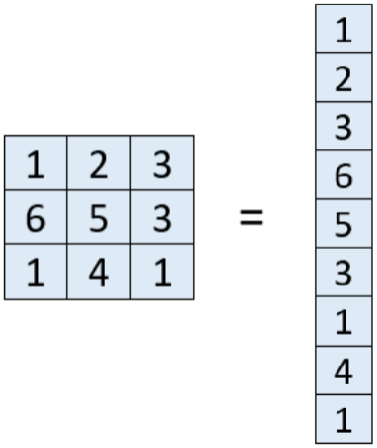


We can also represent this operation as a **Convolution Matrix**. It's a matrix which demonstrates all positions of the kernel on the original image, like this:

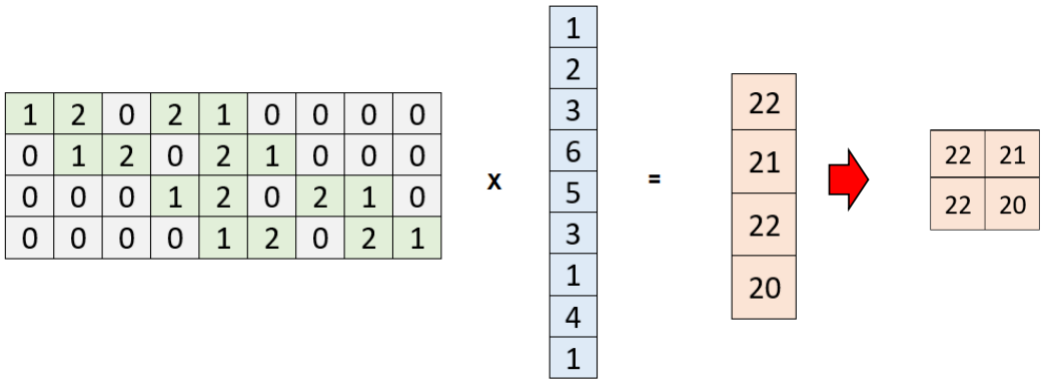
1	2	0	2	1	0	0	0	0
0	1	2	0	2	1	0	0	0
0	0	0	1	2	0	2	1	0
0	0	0	0	1	2	0	2	1

The 1, 2, 0 at the first row of the convolution matrix therefore represents the effect of the convolution at the first row of the input image. The 2, 1, 0 represents the effect of the convolution at the second row of the input image. Since at this point, the convolution is not applied in either the 3rd column or the 3rd row, either the third column value of the first and second row and all the third row values are 0.

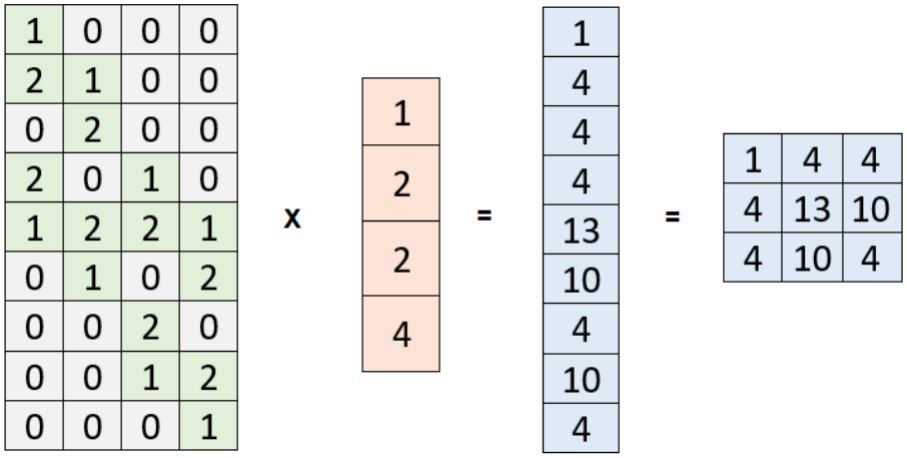
We can also reshape the input image into a (9x1) feature vector:



The good thing is that we can multiply this (9x1) matrix with the (4x9) convolution matrix and hence achieve a (4x9) x (9x1) = (4x1) output:



We find why we call the type of convolution transposed, as we can also represent this by means of the convolution matrix – although not the original one, but its transpose. Now we now have a (9x4) matrix and a (4x1) matrix, which we can multiply to arrive at a (9x1) matrix or, when broken apart, the (3x3) matrix we were looking for!



13.6 Common GANs problems

GANs have several common failure modes. All these common problems are areas of active research. While none of these problems have been completely solved.

Vanishing Gradients: Research has suggested that if your discriminator is too good, then generator training can fail due to vanishing gradients (Generator loss is close to zero).

In effect, an optimal discriminator doesn't provide enough information for the generator to make progress.

13.7 GANs Variants

The first variant we will see are the **conditional GANs** which train on a labeled dataset and let you specify the label for each generated instance. For example, an unconditional MNIST GAN would produce random digits, while a conditional MNIST GAN would let you specify which digit the GAN should generate.

Another variant are the **Image-to-Image translation GANs**. For example you can train an image-to-image GAN to take sketches of an object (for example a car) and turn them into photorealistic images of cars.

There are also **Text-to-Image GANs** which take a text as input and produce images that are plausible and described by the text. For example, to produce a flower image by feeding a text description to the GAN.

Super-Resolution GANs are networks which increase the resolution of input images, adding details where needed.

Face inpainting GANs have been used for semantic image inpainting task. Chunks of image are blacked out, and the system tries to fill in the missing chunks.

Text-to-Speech GANs produce syntethized speech from input text.

14 Attention Models

Attention models, or attention mechanisms, are input processing techniques for neural networks that allows the network to focus on specific aspects of a complex input, one at a time, until the entire dataset is categorized. The goal is to break down complicated tasks into smaller areas of attention that are processed sequentially.

Similar to how the human mind solves a new problem by dividing it into simpler tasks and solving them one by one. This is the underlying key intuition of attention models.

The attention mechanism emerged as an improvement over the encoder decoder-based neural machine translation system in natural language processing (NLP). Later, this mechanism, or its variants, was used in other applications, including computer vision, speech processing, etc.

Before the use of attention mechanism, neural machine translation was based on encoder-decoder RNNs/LSTMs. Both encoder and decoder are stacks of LSTM/RNN units. The encoder LSTM is used to process the entire input sentence and encode it into a context vector and the decoder LSTM or RNN units produce the words in a sentence one after another.

The main drawback of this approach is evident. If the encoder makes a bad summary, the translation will also be bad. And indeed it has been observed that the encoder creates a bad summary when it tries to understand longer sentences. It is called the long-range dependency problem of RNN/LSTMs.

RNNs cannot remember longer sentences and sequences due to the vanishing/exploding gradient problem. Although an LSTM is supposed to fix this problem and to capture the long-range dependency better than the RNN, it tends to become forgetful in specific cases. Another problem is that there is no way to give more importance to some of the input words compared to others while translating the sentence.

So is there any way we can keep all the relevant information in the input sentences intact while creating the context vector?

Bahdanau et al (2015) came up with a simple but elegant idea where they suggested that not only can all the input words be taken into account in the context vector, but relative importance should also be given to each one of them.

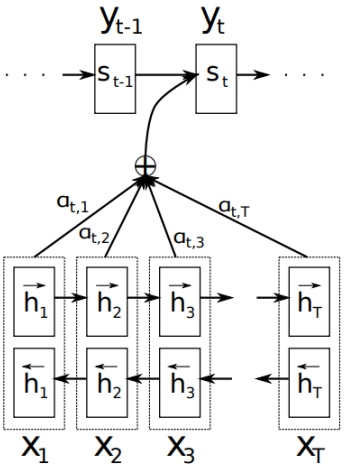
So, whenever the proposed model generates a sentence, it searches for a set of positions in the encoder hidden states where the most relevant information is available. This idea is called "Attention".

Attention is All You Need.
- from the Google paper.

More about attention models in this video and in this article. This chapter is strongly based on this comprehensive guide to the attention topic.

14.1 Attention mechanism

This is the diagram of the Attention model shown in Bahdanau's paper.



The symbol "alpha" in the picture above represent attention weights for each time-step of output vectors. There are several methods to compute these weights "alpha" such as using Dot Product, Neural Network Model with single hidden layer, etc.

These weights are multiplied by each of the words in the source and then this product is the fed to language model along with the output from the previous layer to get the output for present timestep. These alpha values determine how much importance should be given to each word in the source to determine the output sentence.

Here, a Bidirectional LSTM is used, which generates a sequence of annotations $h_1, h_2, ...$ for each input sentence. But Bahdanau et al put emphasis on embeddings of all the words in the input (represented by hidden states) while creating the context vector. They did this by simply taking a weighted sum of the hidden states.

Now, the question is how should the weights be calculated? Well, the weights are also learned by a feed-forward neural network and I've mentioned their mathematical equation below.

The context vector ci for the output word yi is generated using the weighted sum of the annotations:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

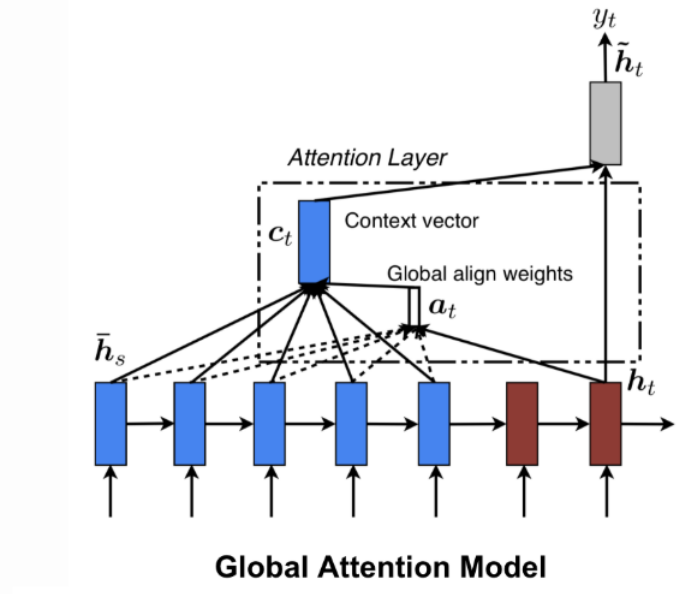
The weights αij are computed by a softmax function (or other alternatives are valid).

14.2 Types of attention models

We have discussed the most basic Attention mechanism where all the inputs have been given some importance. Let's now discuss about global vs local attention. The differentiation is that it considers all the hidden states of both the encoder LSTM and decoder LSTM to calculate a "variable-length context vector ct, whereas Bahdanau et al. used the previous

hidden state of the unidirectional decoder LSTM and all the hidden states of the encoder LSTM to calculate the context vector. When a "global" Attention layer is applied, a lot of computation is incurred. This is because all the hidden states must be taken into consideration, concatenated into a matrix, and multiplied with a weight matrix of correct dimensions to get the final layer of the feedforward connection. Can we reduce this in any way? Yes! Local Attention is the answer.Intuitively, when we try to infer something from any given information, our mind tends to intelligently reduce the search space further and further by taking only the most relevant inputs.

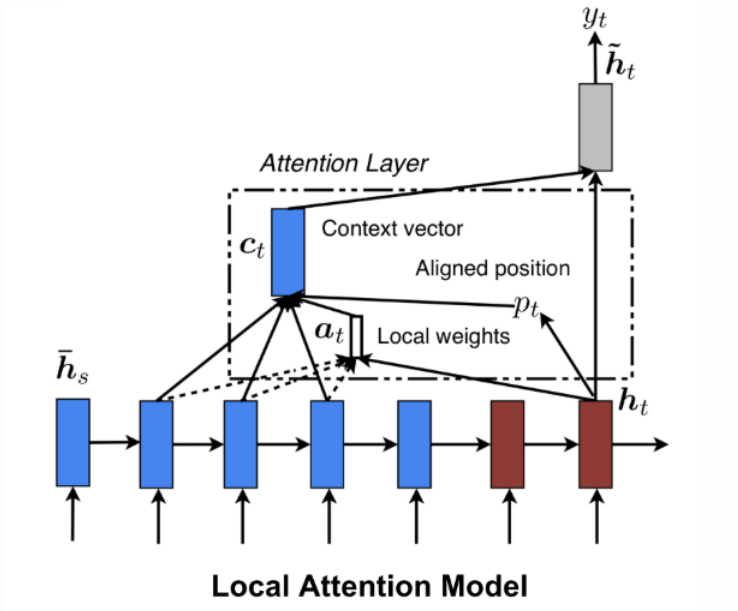
14.2.1 Global attention model: Input from every source state (encoder) and decoder states prior to the current state is taken into account to compute the output. Below is the diagram for the global attention model. $a < t >$ align weights or attention weights are calculated using each encoder step and $h < t >$ decoder previous step. Then using $a < t >$ context vector is calculated by taking the product of Global align weights and each encoder steps. It is then fed to RNN cell to find decoder output.



Global Attention Model

14.2.2 Local attention model: It is different from Global Attention Model in a way that in Local attention model only a few positions from source (encoder) is used to calculate the align weights $a < t >$. Below is the diagram for the Local attention model.

It can be referred from the picture, first single-aligned position $p < t >$ is found then a window of words from source (encoder) layer along with $h < t >$ is used to calculate align weights and context vector. Local Attention-It is of two types Monotonic alignment and Predictive alignment. In monotonic alignment, we simply set position $p < t >$ as "t" whereas in predictive alignment, position $p < t >$ instead of just assuming it as "t" it is predicted by the predictive model.



Local Attention Model

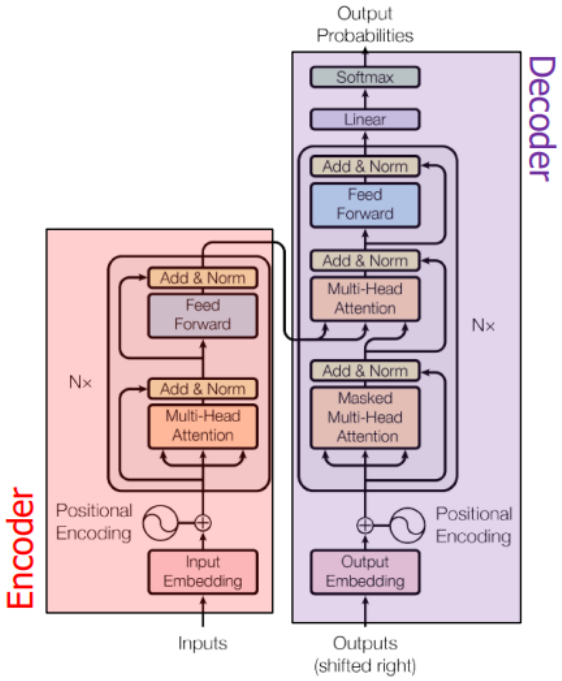
14.2.3 Hard and Soft Attention: Soft attention is almost the same as Global attention model. Foft attention is when we calculate the context vector as a weighted sum of the encoder hidden states as we had seen in the figures above. Hard attention is when, instead of weighted average of all hidden states, we use attention scores to select a single hidden state. Difference between hard and local attention model is that Local model is almost differential at every point whereas hard attention is not. Local attention is a blend of hard and soft attention.

14.2.4 Self-attention Model: Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.

14.3 Transformer Network

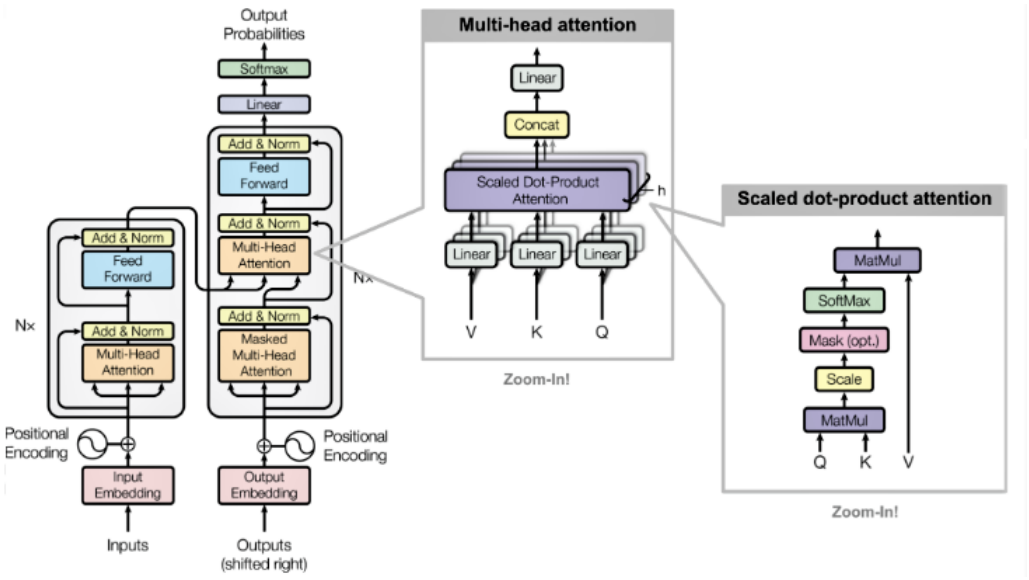
Transformer was developed for solving text translation problem. The Transformer architecture consists of an Encoder and a Decoder and the Encoder's output is an input to the Decoder.

Transformer network is entirely built upon self-attention mechanisms without using recurrent network architecture. The transformer is made using multi-head self-attention models.



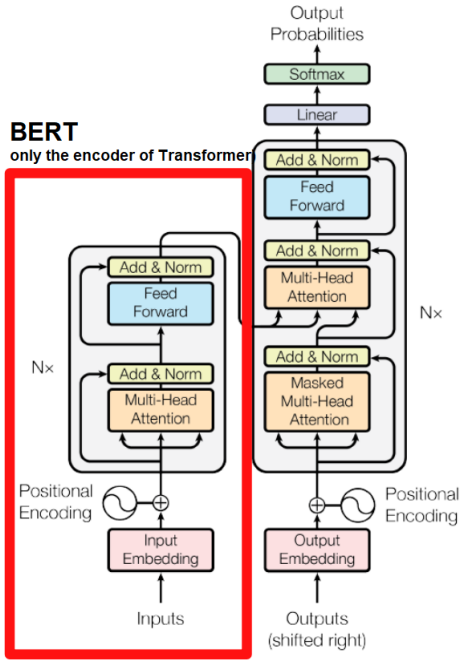
Encoder layer consists of two sub-layers, one is multi-head attention and the next one is a feed-forward neural network. The decoder is made by three sub-layers two multi-head attention network which is then fed to the feed-forward network. Decode attention network takes input from the encoder output and output from the previous layer of the decoder. The encoder, encodes the input text and transforms it in a machinable (embeddings) readable representation. The decoder takes the encored output and performs the required task.

More information about Transformer can be found in this stackoverflow topic.



14.4 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a recent paper published by researchers at Google AI Language. It has caused a stir in the Machine Learning community by presenting state-of-the-art results in a wide variety of NLP tasks: Question Answering (SQuAD v2.0), Named Entity Recognition, etc.



BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to language modelling. This is in contrast to previous efforts which looked at a text sequence either from left to right or combined left-to-right and right-to-left training. The paper's results show that a language model which is bidirectionally trained can

have a deeper sense of language context and flow than single-direction language models. In the paper, the researchers detail a novel technique named Masked LM (MLM) which allows bidirectional training in models in which it was previously impossible.

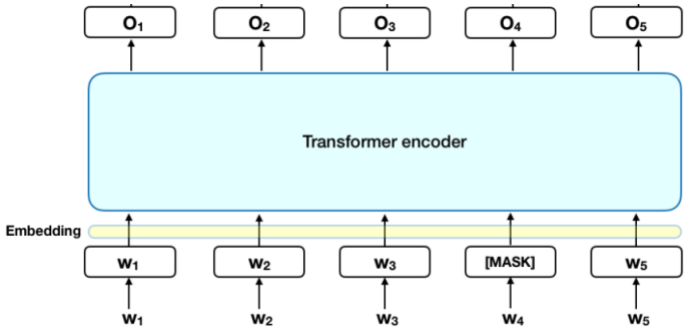
14.4.1 How BERT works: BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. In its vanilla form, Transformer includes two separate mechanisms — an encoder that reads the text input and a decoder that produces a prediction for the task. Since **BERT's goal is to generate a language model**, only the encoder mechanism is necessary. The detailed workings of Transformer are described in a paper by Google. As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional, though it would be more accurate to say that it's **non-directional**. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

The input is a sequence of tokens, which are first embedded into vectors and then processed in the neural network. The output is a sequence of vectors of size H, in which each vector corresponds to an input token with the same index. When training language models, there is a challenge of defining a prediction goal. Many models predict the next word in a sequence (e.g. "The child came home from ..."), a directional approach which inherently limits context learning. To overcome this challenge, BERT uses two training strategies:

- Masked LM (MLM)
- Next Sentence Prediction (NSP)

14.4.2 Transformer Encoder

The input is a sequence of tokens, which are first embedded into vectors and then processed in the neural network. The output is a sequence of vectors of size H, in which each vector corresponds to an input token with the same index.



14.4.3 Information Flow

The data flow through the BERT architecture as follows:

- The model represents each input token as a vector of emb_dim size. Tokenization, numericalization and word embeddings are produced.

- It then adds positional informations (positional encoding).
- The data goes through N encoder blocks (N in BERT was 12 or 24). A specific block is in charge of finding relationships between the input representations and encode them in its output. This iterative process through the blocks will help the neural network capture more complex relationships between words in the input sequence.

There is no weight sharing between different blocks. The Transformer uses Multi-Head Attention, which means it computes attention h different times with different weight matrices and then concatenates the results together.

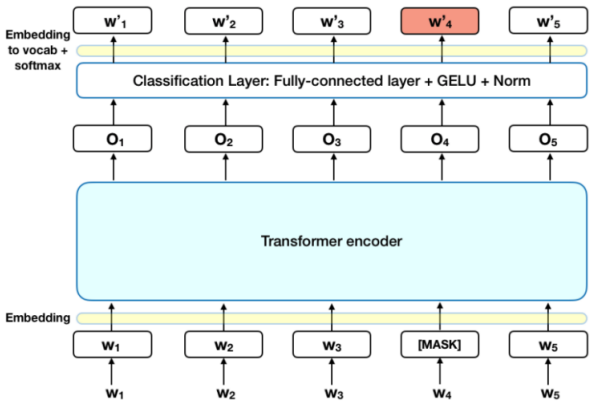
14.5 Training of BERT

Let's now see the two strategies cited before:

14.5.1 Masked Language Model (MLM)

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires:

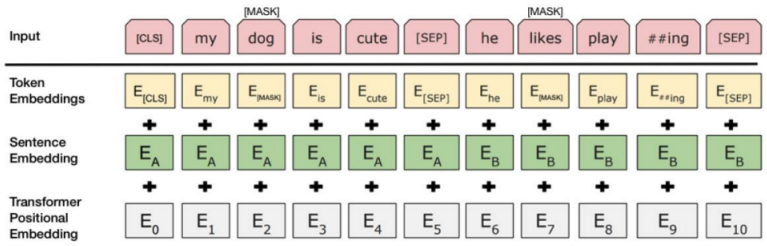
- Adding a classification layer on top of the encoder output.
- Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
- Calculating the probability of each word in the vocabulary with softmax.
- Note: In practice, the BERT implementation is slightly more elaborate and doesn't replace all of the 15% of the words with mask tokens, but does something more elaborated.



14.5.2 Next Sentence Prediction

This task consists on giving the model two sentences and asking it to predict if the second sentence follows the first in a corpus or not. To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

- A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- A sentence embedding indicating Sentence A or Sentence B is added to each token.
- A positional embedding is added to each token to indicate its position in the sequence.
- During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence.



To predict if the second sentence is indeed connected to the first, the following steps are performed:

- The entire input sequence goes through the Transformer model.
- The output of the [CLS] token is transformed into a 2x1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
- Calculating the probability of IsNextSequence with softmax.

More about BERT can be found in this article. The Google "Attention Is All You Need" BERT paper can be found for free at this link.

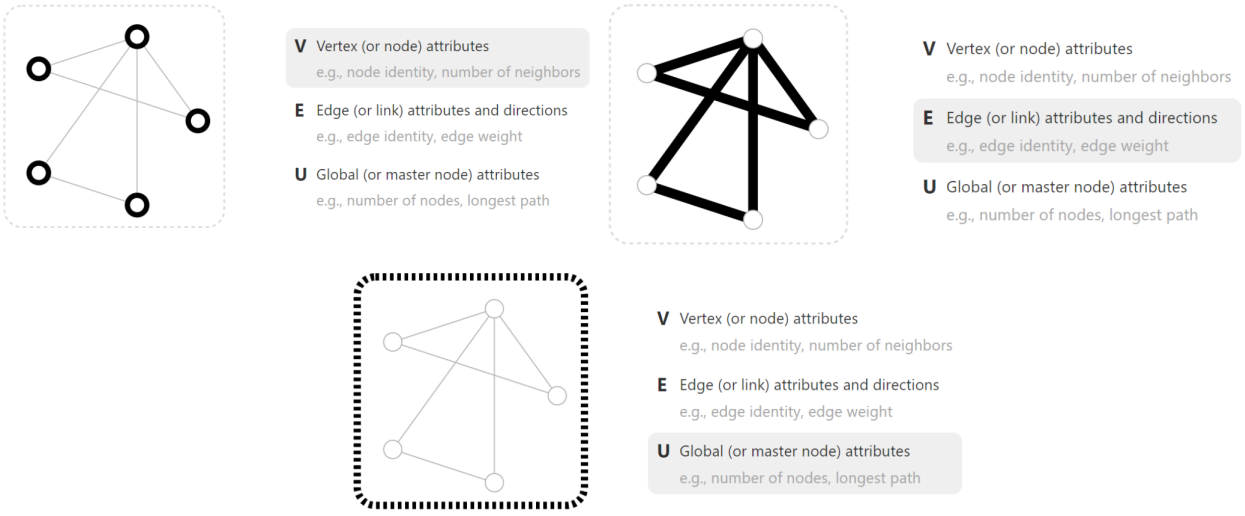
15 Graph Neural Networks

Recently, Graph Neural Network (GNN) has gained increasing popularity in various domains, including social network, knowledge graph, recommender system, and even life science. The power of GNN in modeling the dependencies between nodes in a graph enables the breakthrough in the research area related to graph analysis. Graph Neural Network is a type of Neural Network which directly operates on the Graph structure. A typical application of GNN is node classification. Essentially, every node in the graph is associated with a label, and we want to predict the label of the nodes without ground-truth.

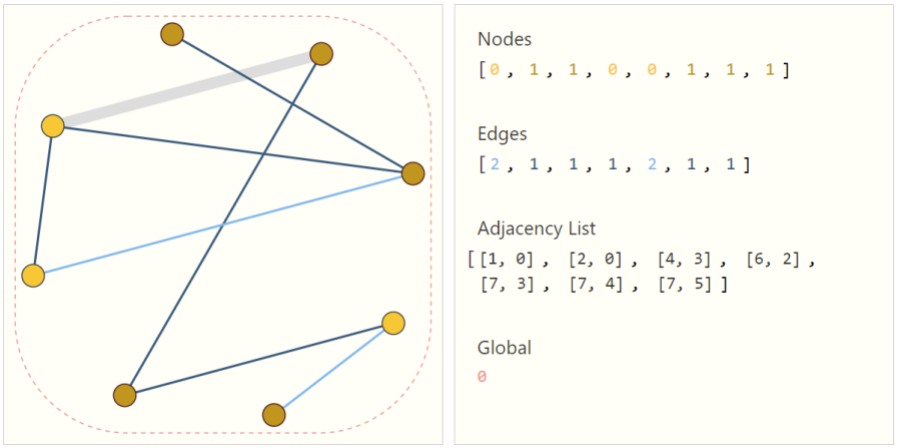
This chapter is a summary of this work which is a good introduction to the GNN topic. Other good summeries and informations can be found here and here.

15.1 Recap of Graph Theory

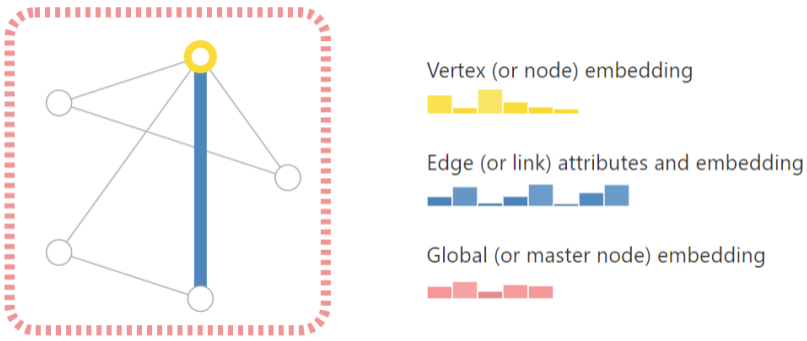
A graph represents the relations (edges) between a collection of entities (nodes).



Graphs can be represent as follow:



To further describe each node, edge or the entire graph, we can store information in each of these pieces of the graph.



15.2 Applications

We can use graphs to represent molecules, images, texts, social networks and so on.

We typically think of images as rectangular grids with image channels, representing them as arrays (e.g., 244x244x3 floats). Another way to think of images is as graphs with regular structure, where each pixel represents a node and is connected via an edge to adjacent pixels. Each non-border pixel has exactly 8 neighbours, and the information stored at each node is a 3-dimensional vector representing the RGB value of the pixel.

We can digitize text by associating indices to each character, word, or token, and representing text as a sequence of these indices. This creates a simple directed graph, where each character or index is a node and is connected via an edge to the node that follows it.

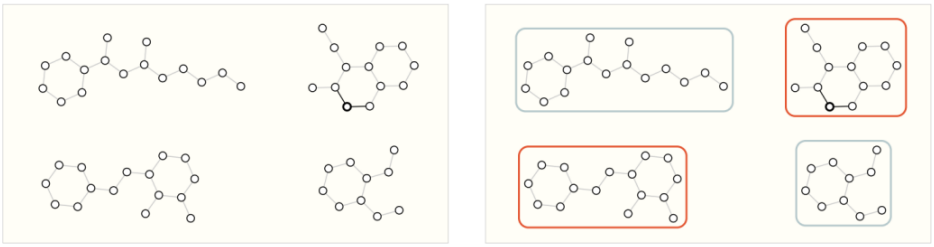
Social networks are tools to study patterns in collective behaviour of people, institutions and organizations. We can build a graph representing groups of people by modelling individuals as nodes, and their relationships as edges.

15.3 Prediction tasks on graphs

We have three types of prediction tasks on graphs.

15.3.1 Graph-level task

The goal is to predict the property of an entire graph. For example, for a molecule represented as a graph, we might want to predict what the molecule smells "pungent" or not. Or if a graph contains two rings:

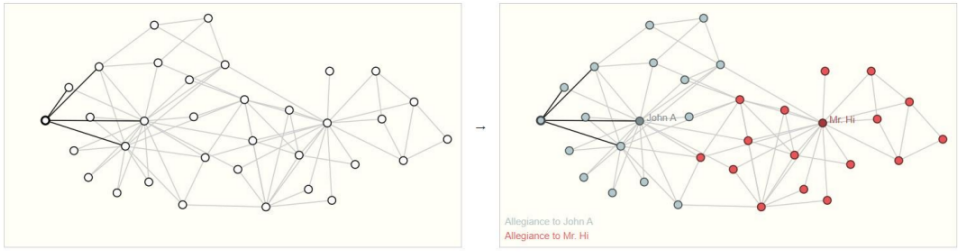


Input: graphs

Output: labels for each graph, (e.g., "does the graph contain two rings?")

15.3.2 Node-level task

Node-level tasks are concerned with predicting the identity or role of each node within a graph. A classic example of a node-level prediction problem is Zach's karate club. As the story goes, a feud between Mr. Hi (Instructor) and John H (Administrator) creates a schism in the karate club. The nodes represent individual karate practitioners, and the edges represent interactions between these members outside of karate. The prediction problem is to classify whether a given member becomes loyal to either Mr. Hi or John H, after the feud. In this case, distance between a node to either the Instructor or Administrator is highly correlated to this label.

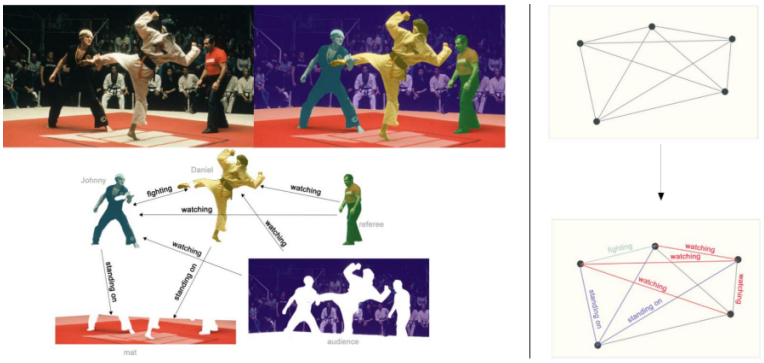


Input: graph with unlabled nodes

Output: graph node labels

15.3.3 Edge-level task

One example of edge-level inference is in image scene understanding. Beyond identifying objects in an image, deep learning models can be used to predict the relationship between them.



15.4 Graph Neural Networks

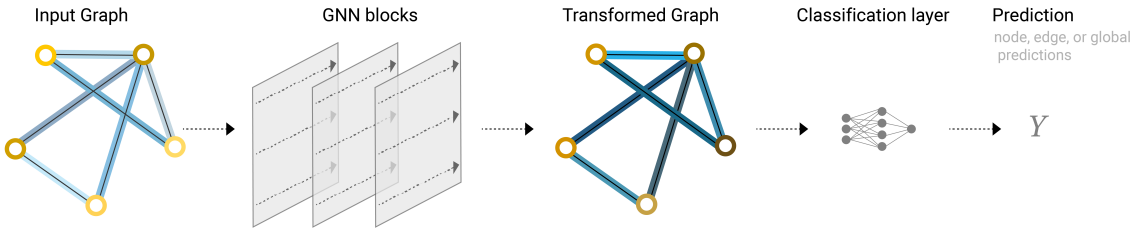
A GNN is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances).

We're going to build GNNs using the **"message passing neural network"** framework.

GNNs adopt a **"graph-in, graph-out" architecture** meaning that these model types accept a graph as input, with information loaded into its nodes, edges and global-context, and progressively transform these embeddings, without changing the connectivity of the input graph.

As is common with neural networks modules or layers, we can stack these GNN layers together.

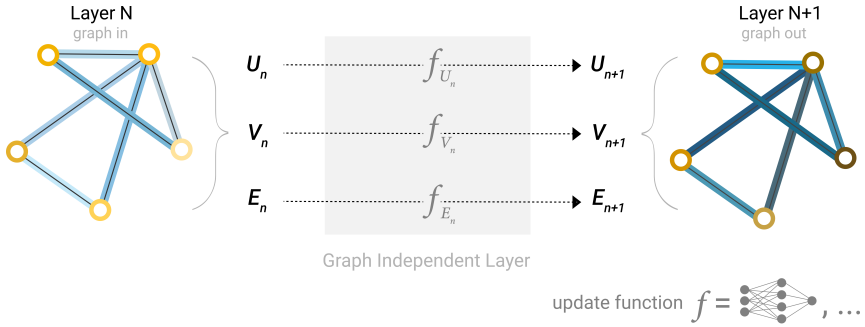
Here an example of an end-to-end prediction taks with a GNN Model:



We will start with the simplest GNN architecture, one where we learn new embeddings for all graph attributes (nodes, edges, global), but where we do not yet use the connectivity of the graph.

This GNN uses a separate multilayer perceptron (MLP) (or your favorite differentiable model) on each component of a graph; we call this a **GNN layer**.

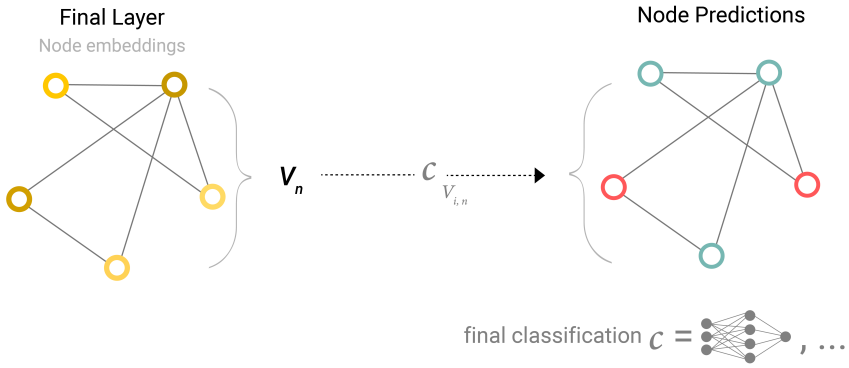
For each node vector, we apply the MLP and get back a learned node-vector. We do the same for each edge, learning a per-edge embedding, and also for the global-context vector, learning a single embedding for the entire graph.



GNNs does not update the connectivity of the input graph, so the output one has the same adiacency list but with updated embeddings.

15.4.1 GNN Predictions by Pooling Information

We have built a simple GNN, but how do we make predictions in any of the tasks we described above? If the task is to make binary predictions on nodes, and the graph already contains node information, the approach is straightforward: for each node embedding, apply a linear classifier.

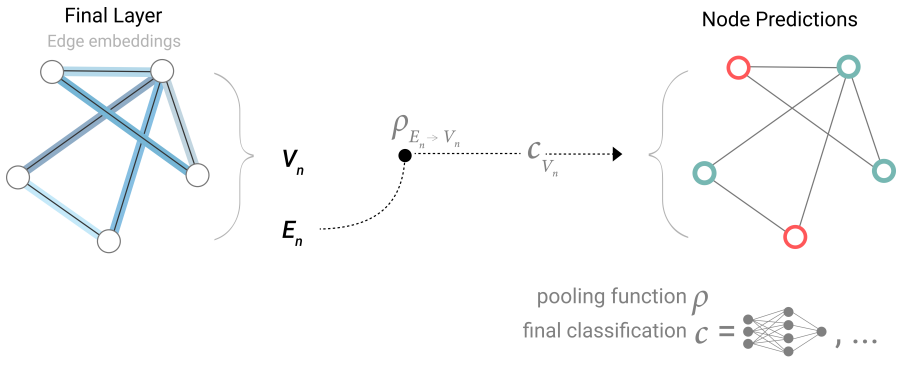


However, it is not always so simple. For instance, you might have information in the graph stored in edges, but no information in nodes, but still need to make predictions on nodes. We need a way to collect information from edges and give them to nodes for prediction. We can do this by **pooling**. Pooling proceeds in two steps:

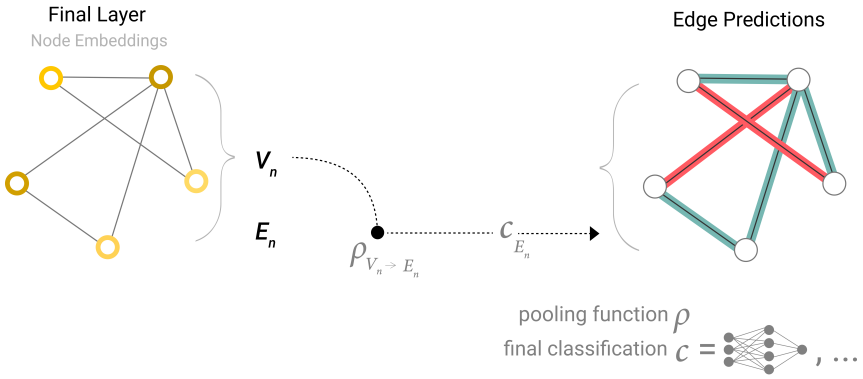
- For each item to be pooled, gather each of their embeddings and concatenate them into a matrix.
- The gathered embeddings are then aggregated, usually via a sum operation.

The pooling operation is represented by p. We use p to gather information from edges to nodes as $p_{E_n} \rightarrow V_n$.

So If we only have edge-level features, and are trying to predict binary node information, we can use pooling to route (or pass) information to where it needs to go. The model looks like this:



If we only have node-level features, and are trying to predict binary edge-level information, the model looks like this:



In our examples, the classification model c can easily be replaced with any differentiable model, or adapted to multi-class classification using a generalized linear model.

This pooling technique will serve as a building block for constructing more sophisticated GNN models. If we have new graph attributes, we just have to define how to pass information from one attribute to another.

Note that in this simplest GNN formulation, we're not using the connectivity of the graph at all inside the GNN layer. Each node is processed independently, as is each edge, as well as the global context. We only use connectivity when pooling information for prediction.

15.4.2 Passing Messages

We could make more sophisticated predictions by using pooling within the GNN layer, in order to make our learned embeddings aware of graph connectivity. We can do this using message passing, where neighboring nodes or edges exchange information and influence each other's updated embeddings.

Message passing works in three main steps:

1. For each node in the graph, gather all the neighboring node embeddings (or messages), which is g function described above.

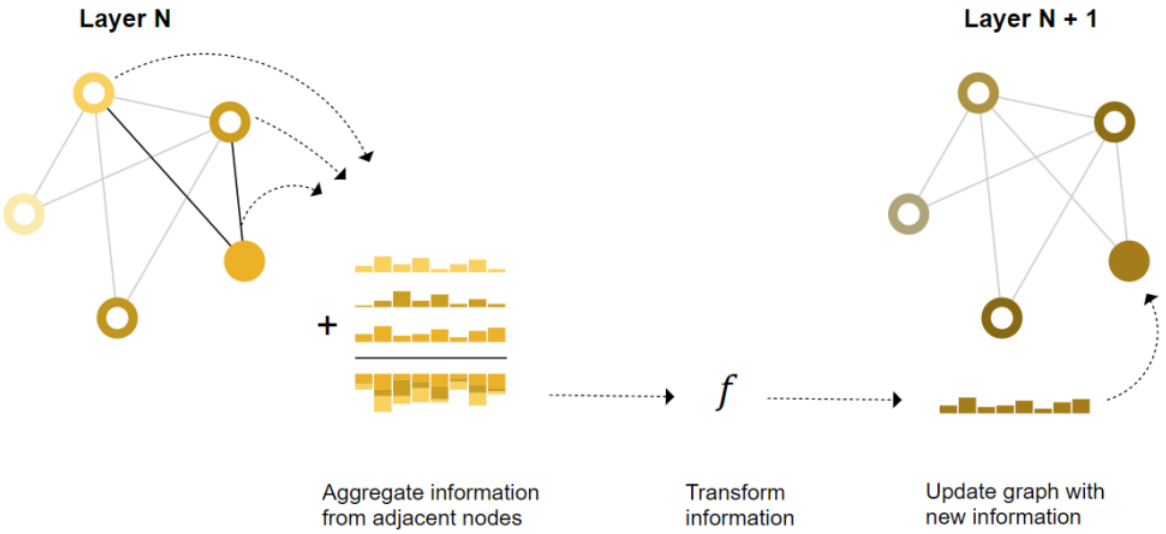
2. Aggregate all messages via an aggregate function (like sum).

3. All pooled messages are passed through an update function, usually a learned neural network.

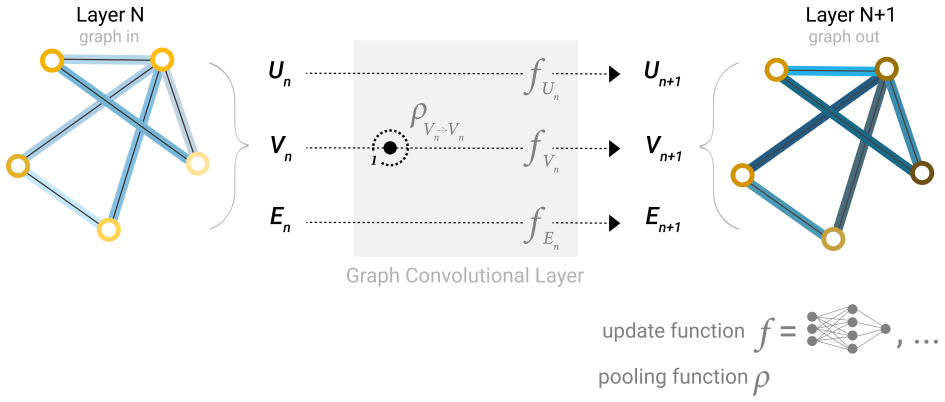
Just as pooling can be applied to either nodes or edges, message passing can occur between either nodes or edges.

Passing Messages and Convolutions

In essence, message passing and convolution are operations to aggregate and process the information of an element's neighbors in order to update the element's value. In graphs, the element is a node, and in images, the element is a pixel. However, the number of neighboring nodes in a graph can be variable, unlike in an image where each pixel has a set

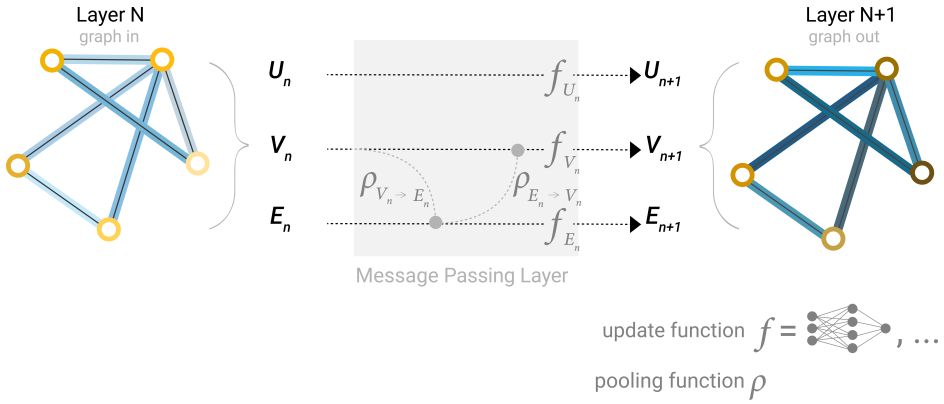


number of neighboring elements. By stacking message passing GNN layers together, a node can eventually incorporate information from across the entire graph: after three layers, a node has information about the nodes three steps away from it.



15.5 Learning Edge Representation

Our dataset does not always contain all types of information (node, edge, and global context). When we want to make a prediction on nodes, but our dataset only has edge information, we showed above how to use pooling to route information from edges to nodes, but only at the final prediction step of the model. We can share information between nodes and edges within the GNN layer using message passing. We can incorporate the information from neighboring edges in the same way we used neighboring node information earlier, by first pooling the edge information, transforming it with an update function, and storing it. However, the node and edge information stored in a graph are not necessarily the same size or shape. One way is to learn a linear mapping model from the space of them edges to the space of nodes, and vice versa.



A possible solution to update the global representation is to use a global context vector. This global context vector is connected to all other nodes and edges in the network, and can act as a bridge between them to pass information, building up a representation for the graph as a whole. This creates a richer and more complex representation of the graph than could have otherwise been learned.