Deep Learning

Personal summary by Gilberto Lem

Summer 2018

Contents

1 Basic Machine Learning Concepts

The course is based on a classifier of images. The classifier tries to label correctly the object that is shown in each image. It can be a cat, dog, horse, plane, boat, car and so on. We start with a Nearest Neighbor classifier as a reference, which simply takes the "distance" between two images with respect to a defined geometry and assigns the label which contains the labeled image with the shortest distance. Through the course we are going to increase the accuracy of that classifier using Deep Learning.

1.1 Training and Test Sets

Linear regression is a kind of supervised learning which consists on finding a linear model that explains a target y with respect to given inputs X. For getting this model, we have to have a data set consisting of inputs and their respective outputs. This data set is going to be splitted in two. The **training set** and the **test set.** As any kind of supervised machine learning algorithm, we take the training set and fit the model to it. After that, we use our fitted model to make predictions on the data of the test set, and we compare our predictions with the real label. Comparing our model to the test set is useful to know if our model is valid not only for our data set, but that it can generalize to new data as well.

1.2 Loss Function

For training our model we have to know how well or how bad our current model represents the data. This is achieved with the **loss function**. The loss function takes the values of our predictions and compares them to the real values, and assigns a scalar. The higher the loss function, the worse our model reflects the data. Naturally, what we want is to find the parameters that minimize our loss function. This is achieved via an optimization algorithm.

But how to define the loss function? A common method for doing it is with the **Maximum Likelihood Estimate**. This consists on estimating the parameters of the model by finding which values of the parameters maximize the likelihood of making the observations given the parameters. At the end, the loss functions obtained through a Maximum Likelihood approach are in the form of logarithms. This is because as we work with several input samples, this technique yields a product of probabilities. As it is more comfortable to work with sums, and logarithms are monotonic functions, we apply logarithms to transform the product into sums.

In the specific case of linear regression, defining the **least squares estimate** as the loss function, we can find an analytic solution to the optimization problem, that is given by the normal equations. On the other side, if we apply the Maximum Likelihood approach to linear regression, together with an assumption of normality, we get an equivalent loss function.

1.3 Classification

Supervised Learning can be divided in regression and classification. While regression predicts a continuous output value, classification predicts a discrete value. One way to transform linear regression, which outputs a value in $\mathbb{R}$, to a classifier is through the sigmoid function. The sigmoid function simply maps $\mathbb{R} \to [0, 1]$, and thus our output can be interpreted as a probability. This represents a binary classifier. Modeling this binary classifier with Bernoulli trials, together with the maximum likelihood estimate, yields a loss function called the **cross-entropy loss.** This loss function does not have a closed-form solution for its minimum, and thus to optimize it we must make use of iterative methods.

1.4 Optimization

The most basic iterative method for optimizing is **gradient descent**. However, this method is not very fast, and it is very prone to get stuck in local minima. It is also very sensible to the stepsize you use for making each step (in the context of Deep Learning, the stepsize is called **learning rate**).

There are several other more efficient methods to optimize used in Deep Learning. Many of them are gradient based methods, for which we need to have the gradient of the loss function. The computing of this gradient is talked about later.

1.5 Regularization

Sometimes our model gets trained in excess and we end up having a model that works too specifically for our training set, it kind of *memorizes* the training set. This is called **overfitting**. When this happens, our training error is low but our test errror gets high.

For overcoming this, we can divide our data set in one more part, called the **validation set**. This set will not be used for training the model (in this aspect it is similar to the test set), but instead it will be used to compare several trained models and choose the better, before getting the final test with the test set.

Regularization consists in any technique which intends to lower validation error and increase training error. Some regularization techniques are: L2 (which enforces the weights to have similar values), L1 (that enforces sparcity), Max norm, and dropout.

Regularization generally works with additional parameters. This parameters are independent of the model parameters, and they do not change through the training. They are called **hyperparameters**. They are generally design choices, as the number of layers in a net, or optimization parameters as the learning rate.

1.6 Neural Networks

A neural network is basically a nesting of functions. It is conformed by neurons. The name *neurons* was inspired in a similar behavior of the cerebrall cells. Each neuron consists on a linear operation and a nonlinear operation to the result. This pair of operations can be for example a linear function followed by a sigmoid function for a classifier. The linear operation contains the weights (or parameters) of the model, and after that the nonlinear operation generally contains only hyperparameters. More examples of the nonlinear operations can be hyperbolic tangent functions or $\max(0, x)$ functions. This nonlinearity functions are called **activation functions**.

The nonlinearity of the neurons makes the network able to learn much more complex patterns than just having linear functions.

A neural network has an input layer (which consists of the original features of our dataset), an output layer (which is the final result) and hidden layers in between.

1.7 Backpropagation

Each neuron is going to have its own set of weights, and each weight has to be optimized. This optimization is achieved through gradient-based solvers. For computing the gradients we can use a technique called **backpropagation.** This technique is based on the chain rule. It considers the neural net as being a computational graph, with values and operations, and starts getting the gradient of the loss function (which is just a scalar) with respect to the last variables, then computes the gradient of those last variables with respect to the second to last variables, and so on, accumulating gradients until reaching the desired variable of interest. This means that when training, each neuron has to do a forward pass (i.e. applying its functions), and then has to do a backward pass with the gradients.

2 Optimization

There are some well stablished techniques for optimization of linear systems, as LU, QR, Cholesky, Jacobi, Gauss-Seidel, etc. However neural nets are non-linear, so we need another set of techniques. This nonlinear techniques can be iterative algorithms or can have a totally different approach. We are going to cover first order and second order iterative algorithms. They are not the only ones though, there exist different approaches as genetic algorithms, montecarlo techniques, and constrained and convex solvers.

For convex functions, all the local minima are global minima, however NN's are non-convex. They have many different local minima and there is no practical way to ensure that a local minimum found is the global minimum. This means that the results we get with our algorithms are not necessarily the optimal solution.

2.1 First Order Approaches

2.1.1 Stochastic Gradient Descent

Recall the formula for gradient descent:

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta^k)$$

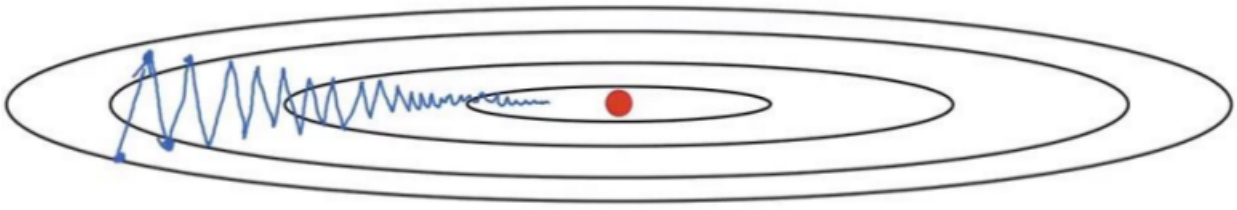
Gradient descent uses a fixed learning rate α which has to be carefully chosen. If it is too small, the optimization can be a very slow process, or it could get stuck in plateaus. On the other hand, if it was too big it could skip the minima, or it could even diverge.

Note that every training sample is going to provide a different gradient. When we want to consider more than one training sample in the optimization, we can take the average of the individual gradients. However, if we have for example 1 million labeled images, and our network had 500k parameters, this starts getting expensive to compute. To overcome this, as our gradient is an expectation value, we can approximate it with a small number of samples, called **minibatches.** We can modify the gradient descent algorithm to take a different random minibatch of the training data in each iteration, and taking the step with the gradient of just the minibatch. This is called **Stochastic Gradient Descent (SGD)**[1]. The minibatch size would be an additional hyperparameter.

With the introduction of the concept of minibatch, the concept of an **epoch** arises naturally. An epoch is just a complete pass through the training set, and it is achieved after many iterations of the optimization algorithm, because every iteration considers only a minibatch. For example, if our training set size is of 50k, and our minibatch size is 50, we would need 1k minibatches to complete an epoch.

2.1.2 First Moment

One of the problems with SGD is that the learning rate is the same in all directions, so it can be slower than necessary. Imagine the following case, where the blue line follows the path done by an SGD algorithm:



[1] Often called vanilla SGD

In this case, the vertical gradients are just averaging, and the real direction we are taking is the horizontal direction. We would be more efficient if we accumulated the gradients over time. This is achieved with an algorithm called **Gradient Descent with Momentum**, that has the following update rules:

$$m^{k+1} = \beta_1 m^k + (1 - \beta_1)\nabla_\theta L(\theta^k)$$
$$\theta^{k+1} = \theta^k - \alpha m^{k+1}$$

Here, v is called a velocity vector. Note that what this vector does is sum the previous velocities in every iteration with the current gradient, and thus it is accumulating gradients. So, instead of only one gradient, it takes something like the average of the gradients (the first moment). The steps will be largest when a sequence of gradients all point in the same direction. β_1 is the accumulation rate (called also momentum), another hyperparameter that is usually set to 0.9.

Nesterov's Momentum Nestorov's Momentum adds an additional step to the GD with Momentum algorithm. It defines an intermediate parameter set $\tilde{\theta}$, which includes the information of the velocity, and that's where the gradient is evaluated:

$$\tilde{\theta}^{k+1} = \theta^k + m^k$$

$$m^{k+1} = \beta_1 m^k + (1 - \beta_1)\nabla_\theta L(\tilde{\theta}^{k+1})$$

$$\theta^{k+1} = \theta^k - \alpha m^{k+1}$$

This is similar to Heun's method in ODE's. We are evaluating the derivative in an approximation of the next step, to increase the order of the method.

2.1.3 Second Moment

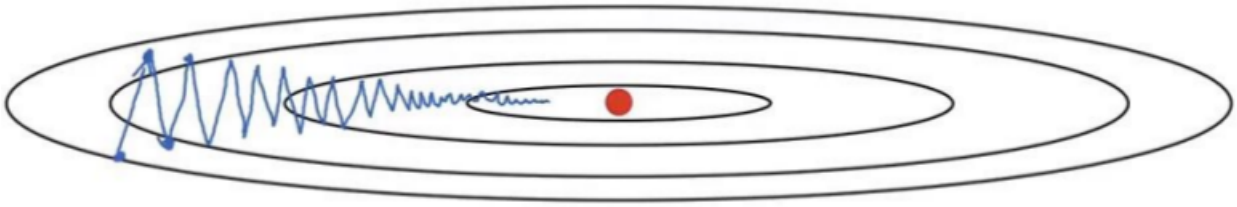
The idea behind another method called **Root Mean Squared Prop (RMSProp)** is to use instead of the first moment, the second moment, the variance. This would be:

$$s^{k+1} = \beta_2 s^k + (1 - \beta_2)\left[\nabla_\theta L \odot \nabla_\theta L\right]$$

This is accumulating the squares of the gradients with the ponderation β_2 (which is usually 0.999). This would be the uncentered variance of gradients. And then we divide our gradient by its standard deviation:

$$\theta^{k+1} = \theta^k - \alpha \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \epsilon}$$

ϵ is added only for numerical stability reasons, and is set usually to 1e-8.
In the same example:



This would help to damp the oscilations for the high-variance direction (y-axis), because the division by the variance in the y-direction would be large, and in the x-direction would be small. This enables us to use increased learning rates, because it is less likely to diverge.

2.1.4 Adaptive Moment Estimation (Adam)

This method combines the first momentum and the second momentum:

$$m^{k+1} = \beta_1 m^k + (1 - \beta_1)\nabla_\theta L(\theta^k)$$

$$s^{k+1} = \beta_2 s^k + (1 - \beta_2)\left[\nabla_\theta L(\theta^k) \odot \nabla_\theta L(\theta^k)\right]$$

$$\theta^{k+1} = \theta^k - \alpha \frac{m^{k+1}}{\sqrt{s^{k+1}} + \epsilon}$$

To correct for the bias of the estimators, we can use $\hat{m}^{k+1} = m^k/(1-\beta_1)$, and $\hat{s}^{k+1} = s^k/(1-\beta_2)$:

$$\theta^{k+1} = \theta^k - \alpha \frac{\hat{m}^{k+1}}{\sqrt{\hat{s}^{k+1}} + \epsilon},$$

which is statistically more correct. Here the standard values are: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$, and α is a hyperparameter.

Although there are a couple of other methods, **Adam is mostly the method of choice for neural networks**.

2.2 Second Order Approaches

Approximating our loss function by a second order Taylor expansion, and evaluating in the optimum point such that $\nabla_\theta L(\theta^*) = 0$, we could get the following expression:

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta L(\theta),$$

where H is the Hessian. The application of this method would be ideal, but it is extremely computationally expensive, taking into account that the number of parameters of a network can be millions, and thus the elements of the Hessian would be millions squared, and the computational complexity of inverting the Hessian every iteration would be millions cubed.

There are alternatives to the inversion (or solving of the linear system) of the Hessian, as Quasi-Newton methods BFGS, L-BFGS, Gauss-Newton and Levenberg-Marquardt, which approximate the Hessian and are cheaper.

However, all these Newton and Quasi-Newton methods, although have very fast convergence, work only for full batch updates, which practically never happens for Deep Learning.

2.3 Wrap-Up

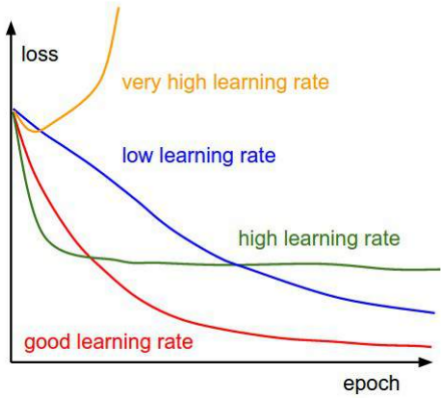
When you have the possibility of doing full batch updates, use Newton and quasi-Newton methods. If not, the standard optimization method in Deep Learning is Adam. If for any reason you cannot use Adam, you could use SGD with momentum, but vanilla SGD or GD are not good options at all.

3 Practical Aspects

3.1 Learning Rate Choosing

It is a common practice to start with a high learning rate and reduce it after every epoch, empirically a nice number for this reduction is 90-95%. There exists many schemes for this, but basically they all do the same: reduce the learning rate with time.

One way to see if the learning rate is adequate is to see a graph of loss with respect to iterations. A low learning rate would look like a linear decay, a good learning rate as a negative exponential, and a high learning rate as a negative logarithm:



3.2 Learning

Learning means to generalize to an unknown dataset. Means to hope that based on the training, the optimized parameters will give similar results on different data. This can be tested with different data-sets. We usually divide the data set in three:

- Training set, used to train
- Validation set, used for hyperparameter optimization and check the generalization progress
- Test set, which should not be touched during development or training, only for the very end

Tipically the relation is (60%,20%,20%) respectively, or (80%,10%,10%).

The graphs of error versus iterations for the training and the validation set are the most useful indicator of overfitting. The validation error is an estimator of our generalization error.

3.3 Hyperparameter Tuning

The hyperparameters are chosen looking at the validation errors. We would choose the hyperparameters which represent the lowest validation error. For getting this we could make a manual search, a grid search (going through all possible configurations in a space of hyperparameters made with different orders of magnitude) or random searchs.

3.4 Cross Validation

If the data set is extremely small and/or the method used is fast, we can use cross validation, which consists on not having a single validation set, but partitioning the data into k subsets, train on $k-1$ and evaluate performance on the remaining subset, and repeat this k times. Additionally, it would be possible to perform this process with different partitions and at the end average the results.

3.5 Bias and Variance

The difference between the human level error and the training set error would be called the **Bias**, which is the limit our model has to represent correctly the data set i.e. our model is not complicated enough to represent it completely i.e. underfitting.

The difference between the training error and the validation or test error is the **Variance**, which represents how much our model is trained specifically for the training set i.e. overfitting.

If the training error is high (**high Bias**) we would need a bigger model, a new architecture or just to train longer.

If the training error is low but the validation error is high (**high Variance**) we may need more data, regularization or a new model architecture.

3.6 Debugging

It is easy to get overwhelmed when our NN is not working, because of the big amount of things to consider. That is why it is recommended to start simple when one trains. First overfit to a single training sample (achieve 100% accuracy, let the net memorize the input), then overfit to several training samples, with simple architectures at first.

Also it is useful to estimate the timings for training (how long for each epoch?)

4 Loss and Activation Functions

Neural Networks have the following general structure:

input layer - hidden layers - output layer

The hidden and output layers have associated nonlinear functions. The type of nonlinear functions we choose in the output and the hidden layers is a crucial decision to make, and will afect the accuracy of our network.

The functions used for nonlinearity are called **Activation functions**.

4.1 Activation Function of the Output Layer

The output layer will be responsable of making our final prediction. One of the most used functions for the output layer is the sigmoid function. This is used for binary classification.

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

When we have more than one class, we can generalize the sigmoid function, to get the **softmax** function:

$$p(y_i|x, \theta) = \frac{e^{x\theta_i}}{\sum_j e^{x\theta_j}}$$

Note that this is just a normalized exponential.

4.2 Loss Function

After having a computed value for our prediction, we need to see how good our prediction is, i.e. we have to choose a loss function.

4.2.1 Naive Loss

The most natural way to compute the loss would be to take a norm of the difference between our prediction and the real results. For this we could choose an L2 or an L1 norm:

- **L2:** This is a sum of **squared** differences, which makes it sensible to outliers. It is a very efficient norm, because its gradient is easy to compute. Its optimum is the mean.
- **L1:** This is a sum of **absolute** differences, which makes it more robust but expensive to compute (the gradients are slightly more expensive). Its optimum is the median, and it it is not prone to outliers.

4.2.2 Cross-Entropy Loss

If we apply the Maximum Likelihood Estimate to the softmax function, we get the **cross-entropy loss,** which is a negative logarithm applied to the outputs of the softmax function. The negative logarithm will assign a very high value if the predicted value is not the real value, and will asign a low value if it is correct.

4.2.3 Hinge Loss (SVM)

Hinge Loss is given by the following expression:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

This means that for each training sample, it will take a look to the scores of the labels that are not the correct label. If the score of this label is greater than the score of the correct label (or if it is less than one less) it is going to assign a loss. The bigger the score of the incorrect label with respect to the score of the correct label, the bigger the loss. If it is less, it does not assign anything.

Note that when Hinge Loss achieves the level when the score of the correct label is the biggest, it stops trying, this is called saturation, while cross-entropy always wants to improve.

4.2.4 Considering weight decay

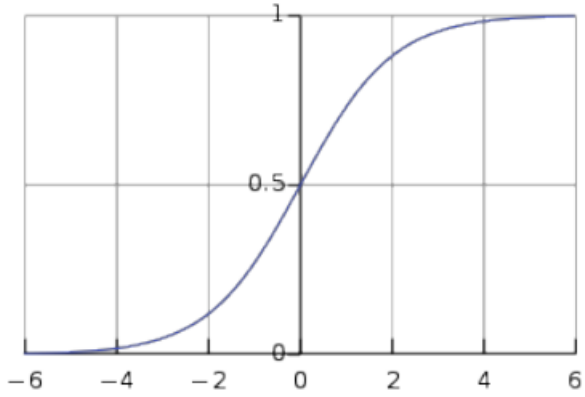
Note that any loss function, if convenient, can be complemented with a regularization L1 or L2 term. See the regularization section.

4.3 Activation Functions of the Hidden Layers

4.3.1 Sigmoid

The sigmoid function maps $\mathbb{R} \to [0, 1]$, and thus can be interpreted as a probability. Its mathematical form is

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



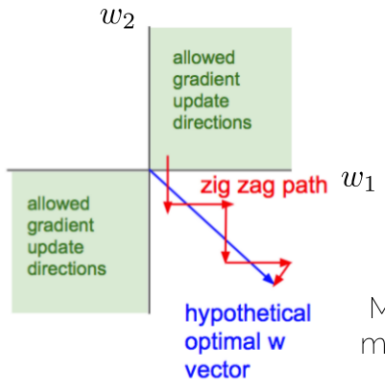
There are however some issues concerning its derivative and its always positive output.

- **Issues with the derivative:** When the neuron saturate (its input is very high or very low), the derivative of the sigmoid function goes to zero, and thus, the gradients vanish very quickly. As the gradients are computed with backpropagation, they are the result of lots of multiplications of individual gradients, and the vanishing gradients of the sigmoid would make the total gradients to vanish and our optimization could get stuck. The active region for the sigmoid is for example $[-2, 2]$ in the graph. This makes the Sigmoid very sensitive to weight initialization. If you are not in the region, the training is very hard/slow.

- **Issues with positive output:** As the output is always positive, when you want to compute the gradient with respect to the weights:

$$\frac{\partial \sigma}{\partial w} = \frac{\partial \sigma}{\partial s}\frac{\partial s}{\partial w}$$

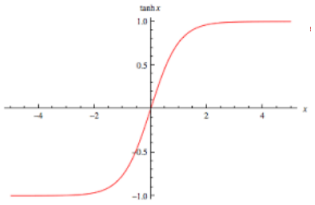
But $\frac{\partial \sigma}{\partial s} = \sigma(1-\sigma) > 0$, and thus $\frac{\partial \sigma}{\partial w}$ has the same sign as $\frac{\partial s}{\partial w}$. If the input was purely positive, this would mean that all the gradients will be either positive or negative, which could slow the process of optimization when we have the following case:



In the previous image, the optimal path to follow for the optimization would be to go to the right bottom corner. But as we can only go with both weights positive or both weights negative, we would have to follow the zig-zag path, which is the best it can do.

4.3.2 Tanh

An alternative for the sigmoid is the tanh function, which is a logistic function too, but zero-centered:



Note that although this function solves the positive output problem, it does not solve the vanishing gradients.

4.3.3 ReLU

A Rectified Liner Unit is the function $\text{ReLU}(s) = \max(0, s)$, and was proposed in 2012. This function provides fast convergence, does not saturate like the sigmoid and provides large and consistent gradients. However, if a ReLU outputs zero or negative, the gradients are zero and the neuron stops learning. This is called a **dead ReLU.** A trick for avoiding dead ReLU's is to initialize ReLU neurons with slightly positive biases (0.1), which makes it likely that they stay active for most imputs.

4.3.4 ReLU variants

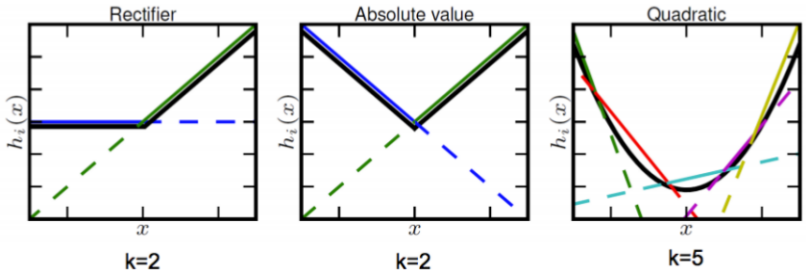
Another way to avoid dead ReLUs is to remove the deadly part. **Leaky ReLU** has the form $f(x) = \max(0.01x, x)$, which does not die. And a generalization is **parametric ReLU**, which has the form $f(x) = \max(\alpha x, x)$, however this introduces one more parameter to backprop into.

4.3.5 Maxout

Another option is the maxout, which is a generalization of ReLUs. This also consists on a max function, however the maximum is between a set of independent scores. For example for k scores:

$$maxout = \max(w_1^T x + b_1,\ w_2^T x + b_2, ...,\ w_k^T x + b_k)$$

Note that this implies k sets of weights for each neuron, and is equivalent to piecewise linear approximation of a convex function with k pieces.



Maxout works on linear regimes which do not die and do not saturate, however, it increases the number of parameters k times!

4.3.6 Wrapup

Sigmoid is not really used anymore. ReLU is the standard choice, and the second choice is the variants of ReLU or the Maxout.

5 Weight Initialization

Preprocessing is to try to reduce the amount of work to do by the neural network. Tipically the most common practice is to zero-center and normalize the data.

As neural networks do not have a robust optimization method, and it is not guaranteed to achieve a global minimum, we have to make sure to have a good initialization. So, how do we initialize the weights?

If we initialized all the weights to the same value (e.g. zero), there would not be symmetry breaking, all the weights would have the same gradients, and there would be no learning.

One way to change this is to take the values for the weights sampling a gaussian distribution with zero mean and standard deviation of 0.01. Making this experiment in a network, the forward pass results in having the output of the layers going to zero mean and zero standard deviation as they go deeper in the net.

If the activation function were a sigmoid, with small weights we would be in the active part, and we would have a nice gradient **with respect to the score**. However the gradient **with respect to the weights** would vanish, because the inputs are getting smaller and smaller with each layer. If the initial numbers were very big, we would have the opposite effect, going to the saturating region and after that vanishing the gradients, but this time because of the derivative with respect to the score.

As solution to theses problems, we have two options. Work on the initialization itself or make a post-computing treatment in each layer.

5.1 Working on the Initialization Itself

Xavier Glorot in 2010 studied which initial standard deviation for the weights would be good. Assuming independence of the inputs to the weights, he got an expression for a good variance:

$$\mathrm{Var}(s) = (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

Which means that the variance would depend on our number of inputs n. To ensure that the variance of the input is the same as the variance of the output, one would need $\mathrm{Var}(w) = 1/n$. This mitigates the effect of the activations going to zero. Just by changing the variance for $1/n$.

Specifically for ReLU, as ReLUs kill half of the data, this would be $\mathrm{Var}(w) = 2/n$. Seems silly but makes a huge difference.

5.2 Batch Normalization

Another way to mitigate this vanishing of our scores is to make a treatment after each layer. **Batch Normalization** does this by working on minibatches. For each feature we normalize the outputs with the mean and standard deviation of the minibatch. However, maybe our network does not need to learn a normalization, so we allow the network to undo the process, learning an additional mean and standard deviation. This process is very useful, because it allows to be way more flexible on the weight initialization. And it is not only that. A much larger range of hyperparameters works similarly when using BN. This brings stability to the training procedure.

One could think that it is unnecessary to normalize and then reshape. One could only reshape and save one step. However if the network is in a zero-gradient zone, it won't be able to normalize anything. So with normalizing it, we ensure it will not be in a zero-gradient zone.

A nice thing is that you can set all biases of the layers before BN to zero because they will be cancelled out by BN anyway.

It is possible to compute mean and variance of the mini-batch at training time. However, when we are testing the network, maybe with only one sample at a time, we cannot get a mean or variance. For overcoming this, we start computing a general mean and variance at training time, and use those values when testing.

Emprically, convergence is faster with batch normalization, but we should not place it everywhere, only in some layers (this introduces another hyperparameter). Also, these batch normalization layers are placed after a FC or convolution layer, and **before the nonlinearity**.

6 Regularization

Regularization consists on any strategy that lowers validation error and increases training error, and its purpose is to generalize our neural network to future inputs.

There exist several techniques for regularization, as Weight decay, Data augmentation, Early stopping, bagging and ensemble methods and dropout.

6.1 Weight decay

The simplest case of regularization is **weight decay,** which includes L2 regularization and L1 regularization.

- L2 regularization promotes distribution of weights. With L2 regularization it is better to have 2 weights of $\theta/2$ than to having one of 0 and one of θ, because the decision is taken by both weights and with that it generalizes better.
- L1 regularization promotes sparcity. With L1 regularization it is better to have one weight of 0 and one of θ.

6.2 Data augmentation

When classifying images, the classifier has to be invariant to a wide variety of transformations, as rotations, translations, different illuminations, sizes, etc. One way to help the classifier with this is to generate fake data that simulates these plausible transformations (flipping, cropping, random brightness and contrast changes).

This is a kind of manual process, because for each case we have to see which kind of transformation makes sense.

Note that when comparing two networks they have to have the same data augmentation. The data augmentation is a part of the network design.

6.3 Early stopping

This technique consists in seeing when the validation starts going up again, and after running a lot of epochs and overfitting, you take the epoch before it starting raising as your result. Or just stop when it starts raising, to save time.

6.4 Dropout

As neural networks are very sensitive to initial conditions, one way to go is to train several models and average their results. Also you could make the models different by using different training sets. This is called **Bagging and ensemble methods.**

This is automized by **dropout**. Here we want to fakely create different models in one single net. We do not want to train two models, but instead train one model that contains 2 models. In each iteration, in each layer we will disable a fraction of the weights (typically 50%), chosen randomly. You make the optimization step with this "smaller" network. This forces the remaining neurons to still perform well even when the weights for a certain characteristic are not functioning. The remaining weights have to find a way to compensate the missing weights. We are reducing the co-adaptation between neurons.

In the brain it happens something similar. There are several paths in the brain that lead to the same conclusion. If a path for making a certain activity is destroyed, we can still make that activity, relying in other related paths.

At test time, we turn on **all the neurons**. This means that the weights are now bigger than they were supposed to be, and thus we are going to predict values that are bigger. For overcoming this

we normalize the result. If we have a dropout probability of 0.5, we would get a factor of $1/2$ in the training predictions.

Dropout is very efficient. However as we are reducing the capacity of the model, if our model is small initially, we would need to enlarge it before applying dropout.

7 Convolutional Neural Networks

Having the theory stablished before, one could think that adding more and more layers would allow the network to learn more and more complex patterns and with that it would perform better. However, adding arbitrarily large numbers of layers increases the difficulty of the optimization, and yields to performance plateaus or even drops.

7.1 Why Convolutions

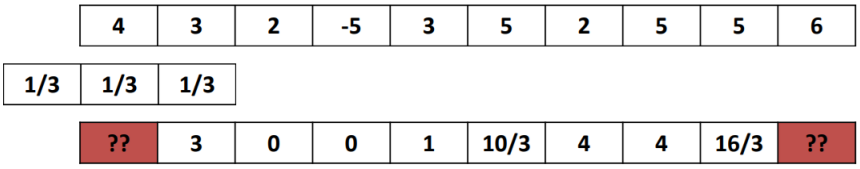
Imagine we are dealing with images, say of a standard size of 1000×1000 pixels (with 3 channels for RGB). If we used one Fully Connected Layer (classical layers with linear operations multiplying every weight with every input) would mean having with 3e6 weights per neuron. Making a standard layer of 1000 neurons, this becomes very impractical, having 3e9 weights for just one layer. It is natural then to think in restricting those degrees of freedom. An idea for doing this is using the same weights for different parts of the image. Convolutions can help with this.

A **convolution** consists in the application of a filter to a function. It is expressed mathematically as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\mathrm{d}\tau$$

We have a function $f(\tau)$, and we will get the result function $(f * g)(t)$ (note that this is dependent on t) as a sum of infinite multiplications of the whole function $f(\tau)$ with another function $g(t)$, but this sum will be displaced by τ in each multiplication. This function g that is displaced is typically called the **filter** or the **kernel**. Tipically both f and g have a finite $\mathrm{supp}(\tau)$.

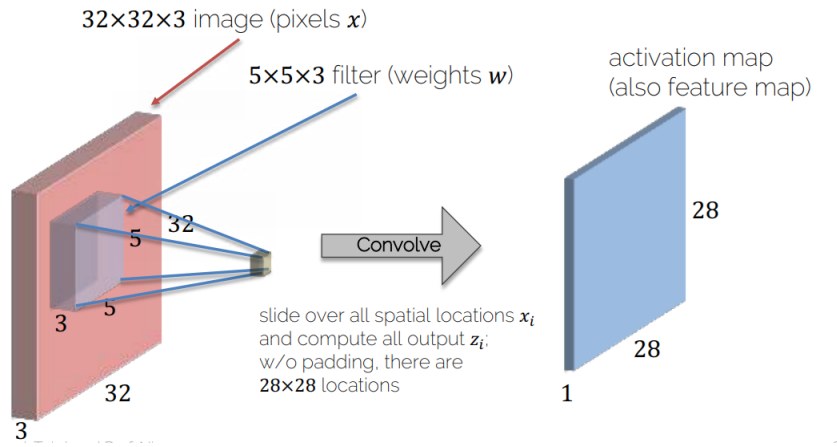
For the discrete case, f is a big array, g a small array. One slides the kernel at each position and computes a single value in the output data. Like sliding a dot product. In the following image, first array represents f, the small aray g, and the last array the result of the convolution.



The first 3 of the result would be gotten by the dot product of the first 3 elements of f and the kernel: $4 \cdot 1/3 + 3 \cdot 1/3 + 2 \cdot 1/3$. The next element in the result, 0, would be gotten by the dot product of the next three elements and the kernel: $3 \cdot 1/3 + 3 \cdot 1/3 - 5 \cdot 1/3$, and so on. Note that this operation outputs a result with less elements than f. We have in red the elements that dissapear. One could accept this *shrinkage*, or pad the array f with zeros to the sides to ensure that the output has the same size as f.

For the case of images, when applying a convolution, each kernel represents a different image filter. With engineering techniques (including trial and error) some filters were discovered way before the development of deep learning. The filters have certain effect in images, as edge detection, sharpening or blurring. The intention in deep learning is to learn kernels which allow the net to do their required task, e.g. classification, segmentation, etc.

Note that the depth of the kernel has to match with the channels of the image. For example, for an RGB image of size 32x32x3, we could use a kernel of 5x5x3, and each dot product of a chunk 5x5x3 of the image with the kernel would output one scalar. For this example, and considering the shrinkage effect, we would get an output of 28x28x1. This output is called **activation map**, or **feature map**. And contains the information (the features) that the kernel is getting from the image. For example, if it was an edge detection kernel, it would contain the edges.



Note that this weight sharing allows the extraction of the same features independent of their location within the image. Here instead of having a weight for each pixel and channel of the image as in FC layers, we would only have weights of the size of our kernel!

It is worth noting that apart from the weights of the kernel, convolutions include a bias term.

7.2 Convolution Layers

One filter (or kernel) would be the equivalent of a neuron, so a **Convolution Layer** would be formed with several filters. Each convolution layer is tipically followed by a ReLU. To keep with the example of the $32 \times 32 \times 3$ image, if we had 10 different filters, each one of size $5 \times 5 \times 3$ (note that each filter has different weights), we would end up with an output of $28 \times 28 \times 10$.

Thus, one basic Convolution Layer would be specified with 3 numbers: filter width, filter height and the number of filters. The depth of the filter (e.g. 3 for RGB images) has to match the input, so it is implicitly given. Each one of these filters would capture a different image characteristic, maybe horizontal edges, vertical edges, circles, squares, etc.

Stride Until now we have been assuming that we are *sliding* the dot product one pixel at a time, but that is not necessarily the case. One could choose to apply the filter just every n-th spatial location. This variable is called **stride.** Note that not every stride chosen would fit the image, for example, an image of 5 pixels, with a kernel of 2, with stride of 2 would not work, we would have one pixel remaining. These strides are just forbidden by convention. Generalizing, if we have an input of $N \times N$, and a filter of $F \times F$, with a stride of S, our output would be of size $\left(\frac{N-F}{S} + 1\right) \times \left(\frac{N-F}{S} + 1\right)$, where having non-integer $\frac{N-F}{S}$ would be illegal.

Padding As we have said, if not taken care of, our output is shrinked. When we have several convolution layers, every layer would shrink their input, and we would end up having very shrinked outputs. Furthermore, the corner pixel of the initial image would be used only once. For overcoming this one could pad the images. The most common padding is with zeros. With the example of the last paragraph, if we introduced a padding of P, we would end up with an output of $\left(\frac{N+2P-F}{S} + 1\right) \times \left(\frac{N+2P-F}{S} + 1\right)$. With this we introduce two types of convolutions:

- **Valid** convolution: using no padding and getting a shrinked output
- **Same** convolution: ensuring that the output size is the same as the input size. This needs computing the necessary padding

The number of weights per layer would then be $F \cdot F \cdot D \cdot K + K$, where D is the number of channels of the input of the layer, and K the number of filters in the layer. The additional K accounts for the bias term in each filter.

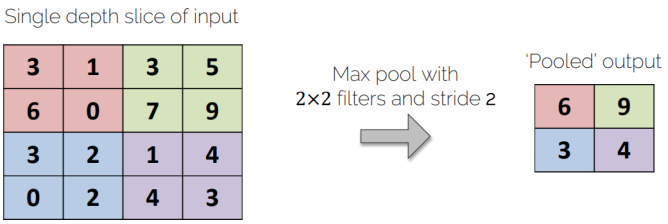
To put it in numbers: in the example of the RGB image of $32 \times 32 \times 3$ with kernel of 5×5, if we applied a convolution layer of ten filters, we would have $5 \cdot 5 \cdot 3 \cdot 10 + 10 = 760$ weights, while with a FC layer we would have $32 \cdot 32 \cdot 3 \cdot 10 \approx 31k$ weights!

It is worth noting that for each layer, F, K, P, and S are hyperparameters.

7.3 Convolutional Neural Networks (CNN's)

A CNN would be the concatenation of convolution layers and activation functions. Generally, the first layers learn low-level features, as general shapes and borders, and advancing through the net, the weights learn more specific features, maybe eyes, peaks, or other specific patterns.

In CNN architectures, it is common to include after some convolution layers an additional layer called the **pool layer.** While the convolution layers compute features, the pooling layers pick the strongest activation in a region. This pooling operation is a fixed function, as a max(...) operation or an average[2]. These layers however, do introduce their own hyperparameters F and S for their filter size and stride. Having for example a max Pool Layer over a matrix of 4×4, with $F = 2$ and $S = 2$, would yield a *pooled* output of 2×2, in which the first element of the output is the maximal value of the first 2×2 block of the matrix, and so on.



At the end of the Convolution and Pooling layers, it is often used to have one or two FC layers to make the final decision with the extracted features from the convolutions.

A prototype for a CNN could thus be:

(Conv - ReLU - Conv - ReLU) - Pool - (Conv - ReLU - Conv - ReLU) - Pool - FC - Softmax

Backpropagation

Although backpropagation can seem complicated to model mathematically, it is possible to assemble a matrix C, of size $O \times I$, where O is the output size and I the input size. The matrix contains the kernel weights in an ingenious setting. Doing the backward pass would be simply myltiplying the gradients by C^T.

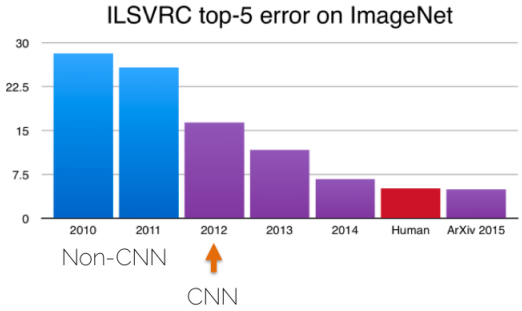
It is worth noting that the backpropagation is done only in the cells that have had an effect in the output. If there is a Max pooling, the backpropagation is going to be computed only for the cells that were maximums. Note that which cells are maximums is going to depend on the input, so with every input the backpropagation would be done with a different path.

7.4 Classic Architectures

Here we are going to compare some famous architectures for image classification. All these architectures were tested using the Image Net Dataset, a huge database that is classically used to evaluate

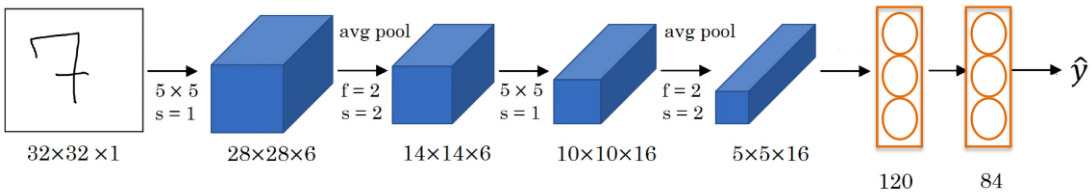
[2]The average pooling is generally used deeper in the network.

performance of architectures. This evaluation is done in a competition called ImageNet Large Scale Visual Recognition Competition (ILSVRC). We can see in the following figure the errors of classification in the competition for CNN's compared to Non-CNN architectures:



7.4.1 LeNet

LeNet is an old CNN proposed in 1998 for digit recognition, i.e. identifying a written number from 0 to 9 (there was not ImageNet at that time). It had the following structure:



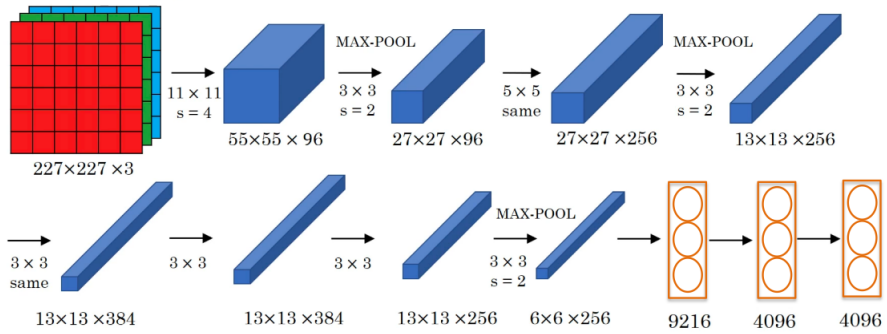
The last 2 layers are Fully Connected Layers. The convolution layers used *valid* convolutions, and thus we observe a shrinkage, in almost every layer. It is a common and recommended practice to **increase the number of filters as we go deeper in the network**, to mantain the same level of complexity.

Other design choices were using average pooling and sigmoid/tanh functions for the FC layers. Those two options are not that common nowadays. It is more common to use Max pooling and ReLUs.

This net has in total **60k parameters**.

7.4.2 AlexNet

AlexNet was proposed in 2012. It has the following structure:



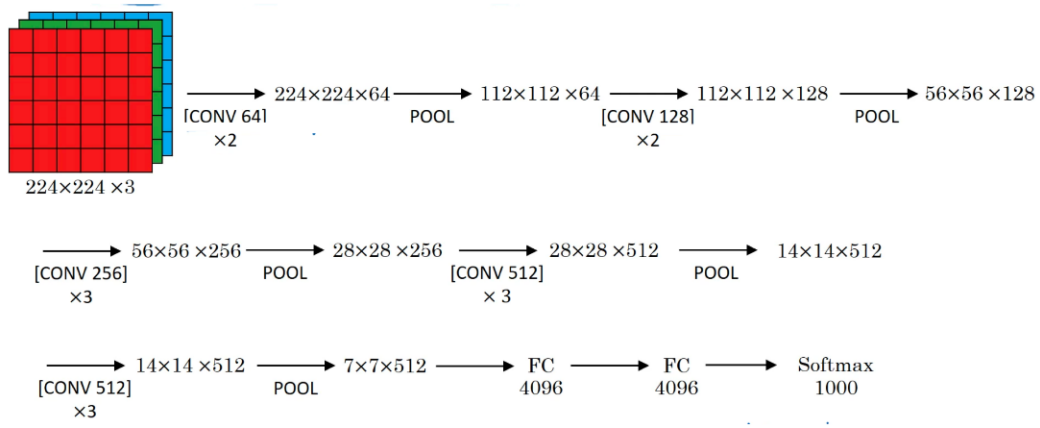
As we can see, since the beginning it uses a high stride to reduce the size, with a high number of filters. After that, it just uses *same* convolutions to preserve the size, although increasing the number of filters to compensate for shrinkage from the pooling. At the end it has three FC layers, and applies a Softmax for 1000 classes. This net uses the same ideas as LeNet, but in bigger structure, and using maxPooling and ReLUs.

It has in total **60M parameters**. It performed classifying images in ILSVRC in 2012 with an error of 16.4%.

7.4.3 VGGNet

Both of the previous architectures work well, however, their design choices seem arbitrary. For example, why do they use in AlexNet 1 convolution with 256 filters, then 2 with 384 and then another one with 256?

For not having this immense number of possibilities to choose from, VGGNet strives for simplicty. It defines convolution layers as always being *same* convolutions of size 3×3, with stride 1, and the pooling layers always being always 2×2 Max pools with stride 2. Also, the number of filters will double after every pooling. Here is an example:

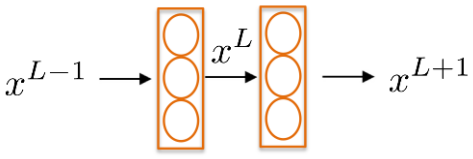


This net is called VGG-16 because it has 16 layers with weights. In total it has **138M parameters**. It is larger, but its simplicity makes it appealing. It performed in ILSVRC in 2014 with an error of 7.3%.

7.4.4 ResNet

Residual blocks

In the previous architectures, we have been seing a tendency to add more and more layers. However, with this the training becomes harder, because for training we need to compute gradients with the chain rule i.e. multiplying them. If the net is too deep, there are lots of multiplications, and the resulting gradients either go to zero or to infinity, i.e. they either vanish or explode. **Residual blocks** try to overcome that problem. Imagine we have the following two layers:



where x^{L-1} would be the input to our layer L, and the output of that layer would be x^L. After the first layer, we have a result $x^L = f\left(W^L x^{L-1} + b^L\right)$, where f is the nonlinear function applied. Similarily after the layer $L+1$, we have $x^{L+1} = f\left(W^{L+1} x^L + b^{L+1}\right)$.

The idea of the residual block is include the input of the block in the last nonlinearity, so we would have:

$$x^{L+1} = f\left(W^{L+1} x^L + b^{L+1} + x^{L-1}\right)$$

This connection between the two layers is called a **skip connection**.

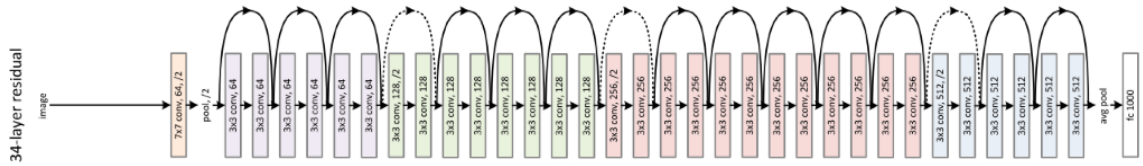
Why is this useful? Imagine that our net learns W^{L+1} and b^{L+1} to be zero. Then we would have $x^{L+1} = f(x^{L-1})$, that would be just the application of a nonlinearity to x^{L-1}. If x^{L-1} was the result of another layer, it had already a nonlinearity. A ReLU applied to a ReLU is still the same result, so we would have the same result as before the residual block. Having the weights to be zero then would be like having an **identity** function, because nothing changes with the block.

So the NN can now learn to have the same output as the input if it is convenient for its performance. One can assume that this identity is easy to learn, one could even put an L2 regularization to force the weights to bea smaller and smaller. This implies a guarantee that introducing the residual block will not hurt the performance - it can only get better!

Note that this requires x^{L-1} to have same the same dimensions as x^{L+1}, and thus we have to use either *same* convolutions or apply some transformation to match their sizes, that could be zero padding, or applying a matrix with learned weights.

ResNet

ResNet is an architecture proposed in 2015 that uses the concept of residual blocks. It starts with a convolution and pooling, and then it is followed by 14 residual blocks, a pooling, and finally a FC layer:



If any of the residual blocks is useless, at least it does not hurt performance.

Without ResNet, the exploding and vanishing gradients make that when you increase the number of layers you decrease the accuracy of your classifier. But with ResNet, the opposite effect is observed, the more layers, the more accuracy!

7.4.5 GoogLeNet

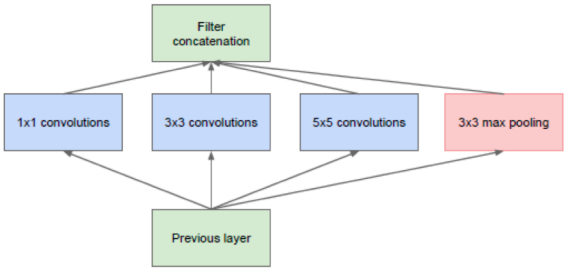
1×1 Convolutions

A 1×1 convolution implies a kernel that is just a number. For one filter, it would just scale the input, keeping the same dimensions. Note that each channel of the input would be scaled with a different number and then they would all be summed. The convolution is then followed by a nonlinear function, generally a ReLU.

At the end, having an input of $N \times N \times D$, after the 1×1 convolution we would have an output of $N \times N \times 1$. That's why one of the most common uses of these convolutions is to shrink the number of channels to the number of filters you want. This could be useful deep in the net when you have lots of channels/filters. Note that applying K 1×1 convolutions would be the same as applying one $D \times K$ FC layer.

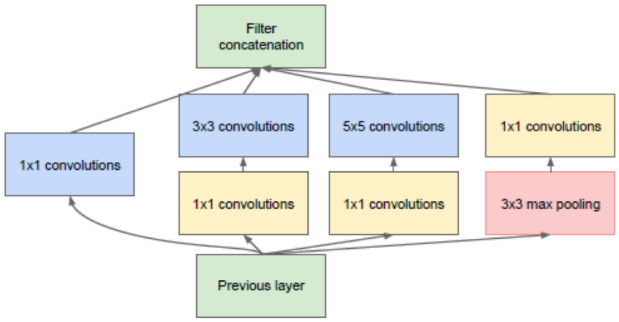
Inception Layer

An inception layer is a layer proposed by Google, that attacks the problem of having to choose a kernel size. Instead of considering a fixed kernel size considers all the kernel sizes at a time. It has the following structure:



The structure consists on taking kernels of size 1, 3, and 5, and also a 3×3 Max pooling, and after that concatenate all the results as an output. That way you are considering all the information in just one layer.

Note that all the convolutions have to be *same* convolutions in order to concatenate them, and also, the pooling is made with a stride of 1 for the same reason. The main problem with this is that the number of operations needed by this layer is extremely expensive, so they propose an alternative formulation including 1×1 convolutions:

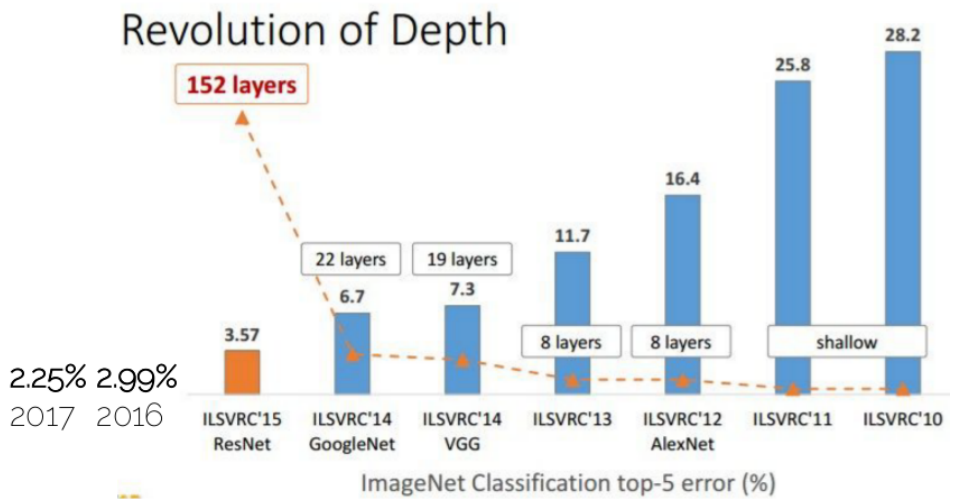


The 1x1 convolutions are used to shrink the number of channels of the input, which allows the 3×3 and 5×5 convolutions to be way lighter (10 times lighter for the case of 5×5!). They are also used to shrink in a similar way the output of the pooling.

GoogLeNet is an architecture consisting on the concatenation of a lot of inception layers. It is huge, but Google does have the computational power to train it.

7.4.6 Summary

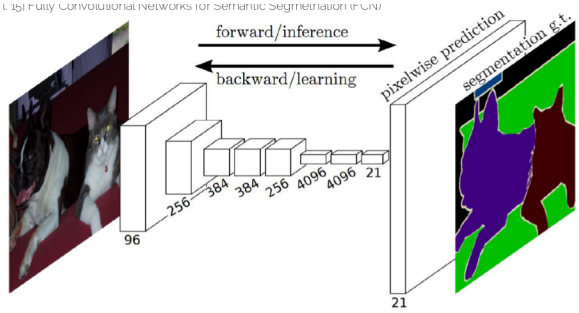
Here's a graph of the performance of the architectures, and their number of layers.



It is worth noting that these numbers were not acheived by just defining the architecture. There are a lot of details to train them right. For example for ResNet, some more design choices were crucial in its performance: using Xavier/2 initialization, SGD+Momentum (β =0.9), mini-batches of 256, weight decay of 1e-5 with no dropout, and a learning rate scheme of 0.1, divided by 10 when in a plateau.

7.5 Fully Convolutional Networks (FCN's)

As we have mentioned, applying K 1×1 convolutions would be the same as applying one $D \times K$ FC layer. We could then substitute our final FC layers that we have been seeing in our CNN's by 1×1 convolutions to get networks that are made completely from convolutions, called **Fully Convolutional Networks**. The advantage of doing this is that the output will have the same size as the output. This could be useful for example in **semantic segmentation**, where each pixel of the picture is assigned with a class, to identify which objects are in the picture in each zone. Also, the size of the FC layers is always fixed, and if the original inputs (images) can be of different sizes, FC would not be able to manage it. However, with 1×1 convolutions we could have an input of whatever size. The weights (weight) are the same for all the pixels! Here is an image of semantic segmentation:

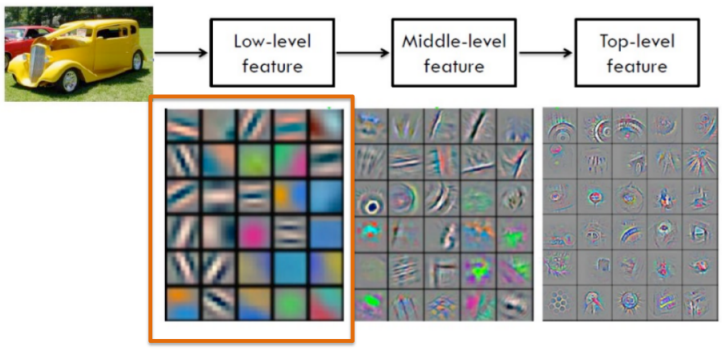


7.6 Transfer Learning

The architectures mentioned before are really big, and hard to train, so generally for using them you take weights already trained as a starting point to make your training. This is called **Transfer Learning**.

The idea of this is to use the weights that the nets have learned with a large data to do some task, and then train other weights with an available data set to learn to do some other task.

For doing this we could use the idea of the different level of features that the nets learn. As we have mentioned, the depth within the net tells you the level of features it has learned to identify. The first layers could identify edges, the middle ones smple geometrical shapes, and the last ones parts of an object.



For example, if you wanted to identify cars you could reuse a lot of things from a net that has learned to identify cats. At least the low-level features, as borders.

The amount of data you would reuse would depend on the amount of data you have available for training for your new task. If you had a very little data set for your new task, you could get pretrained weights from a large net, **freeze** all the weights, and then train only the last FC layer. On the other hand, if the data set was big enough, you could train more layers with a low learning rate.

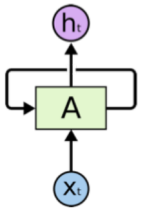
For doing transfer learning, we have some conditions to satisfy:

- Both tasks have to have the same type of input (e.g. RGB images)
- Have more data for the pretrained task than for the new task
- The low-level features of both tasks must be useful for both cases (e.g. images of cats are not helpful for MRI images)[3].

[3]One solution for this is to learn a type of network which learns to recreate your input images to learn the low-level features. These are called auto-encoder architectures.

8 Recurrent Neural Networks

Until now, the Neural Networks described do not consider the information already outputed from the net for making a decision. For example, if our classifier labeled one image as a cat, the label it gives to the next image will be independent of that. This kind of behavior is not useful for some problems, as image captioning (describing what is in an image), language recognition or machine translation. Recurrent Neural Networks (RNN's) handle this problem. Now the output of a network is considered in the next state of the network. This makes possible to include the notion of *time* or *sequence* in the net.

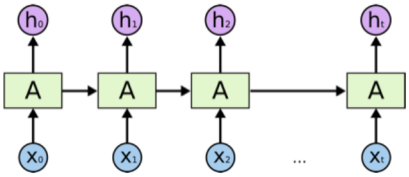


In the previous figure, x_t is the input, A is the *state* that in this case would be the network, and h_t is the output of the net. Mathematically:

$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

$$h_t = \theta_t A_t$$

Note that the weights θ are constant through the time/sequence. We could express the same net in an unrolled way as:



which would be equivalent as a feedforward net (a not RNN) with repeated weights.

As a comment, at the moment of backpropagation, when you want to compute the derivative with respect to one specific weight, you would have to add the derivatives with respect to that weight at the different times.

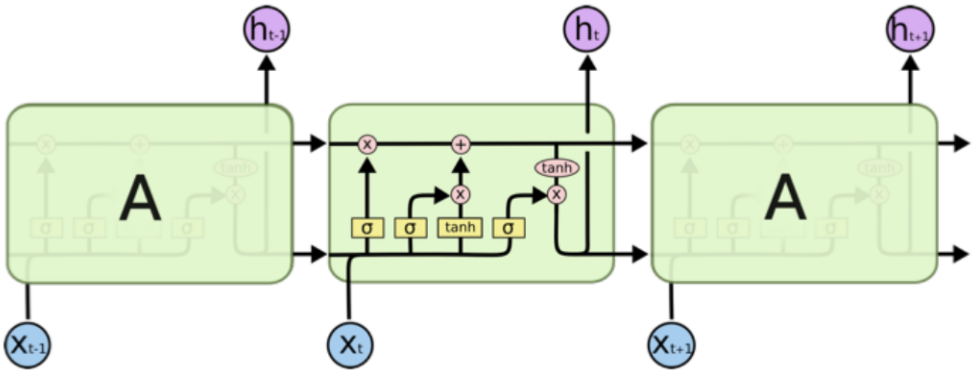
In theory, these networks could take into account the initial information provided to the net even in the long term, independently of how much time it has passed. However, this is not possible computationally. Nesting all the iterations, we could express the state t in terms of the initial state:

$$A_t \sim \theta^t A_0$$

If the eigenvalues of θ are less than one, or more than one, our results will vanish or explode! Ideally, one would need to have eigenvalues of 1 to be able to mantain the state.

8.1 Long Short Term Memory

Having eigenvalues of 1 would solve the problems of accumulating the magnitudes of the weights, however, another source of problems is the vanishing gradients. This is a problem because Classical RNN's use tanh functions when outputing results. Long Short Term Memory Units (LSTM's) attack this problem. For this we have to introduce the concept of **cell**. In the previous section, we represented the network as A, and it received inputs and outputs. Now we change A to be a **unit** of networks and operations:

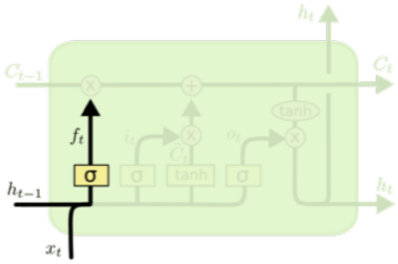


The little yellow squares represent networks, which are called **gates**, and the pink circles/elipses are pointwise operations.

The upper horizontal line which has the pointwise multiplication and sum is like a conveyor belt of information. The information it c arries is an auxiliary structure of information called a **cell**. The belt enters our unit with the cell of the previous time, and gets out of the unit with the new cell. It is worth noting that the cell is not the same as the output of the unit. The cells are just auxiliary structures that are used to compute the outputs, and that allow the flow of gradients to be smoother. Cells will be notated in the following figures with C_t and the outputs as h_t.

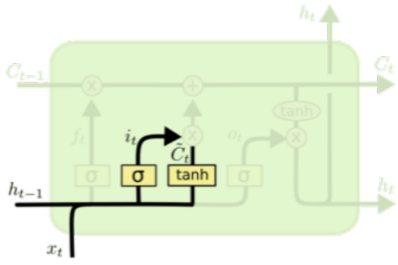
The role of gates is to remove or add information to the cell state. They remove the information that is not important anymore, and they add the relevant information. Let's walk through the process of the LSTM's step by step:

First, for removing information, we have the **forget gate**:



This gate consists on a network with sigmoid output, which is between 0 and 1. This gate would take the new input and the past output, and for each element in the cell, it would define a number between 0 and 1 that defines if the information of that element is going to be kept or not (0 is completely forgotten, 1 is completely kept). Note that after that, the result of the gate is multiplied pointwise with the past cell C_{t-1}.

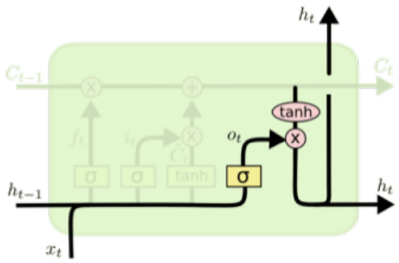
Then, for adding information, we have two gates: the **input gate** and the **new cell gate:**



The new cell gate is the one that has a tanh. This is a network that will output a potential new cell state, the equivalent of having a RNN without LSTM. This new cell state will be then multiplied pointwise with the result of the input gate. The input gate ends outputs a sigmoid, and similar to the forget gate, will decide which values to keep and which values to discard, but this time for the new potential cell state.

Each of the previous two steps had an output. The forgetting phase outputed a modified old cell state, input phase a modified new potential cell state. After that we sum both outputs to get the **new cell.**

As we have mentioned the cell is not the output, it is just used to compute the output. For getting the output we have to do an additional procedure. We will apply a tanh to the new cell state, and then we will call the **output gate**, which similarily to the forget and the input gates, will decide how what information from the new cell to keep:



Now we have the output of the unit. Note that the flow of the gradients is through the cell line, which includes not only the result of one sigmoid or tanh. This reduces significantly the chance of having vanishing gradients.

8.2 RNN's in Computer Vision

In computer vision, RNN's are used for example in caption generation. An input image is treated with a CNN to extract its features, and then the features are fed to an LSTM RNN to generate word by word a description of the image.

Another example is in instance segmentation. Now the goal is to identify and separate objects of the same type, for example, leafs of a tree. This is achieved with a FCN and after that a convolutional LSTM to have spatial inhibition and know what instances you have already identified.