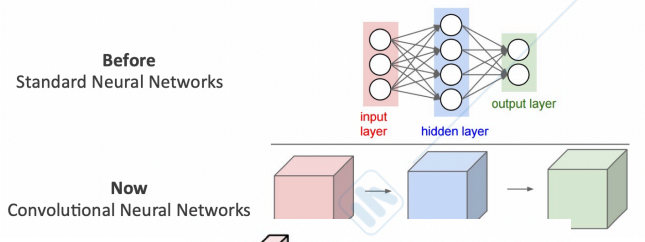
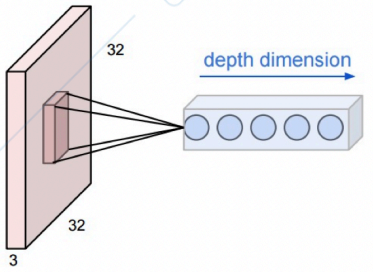
# Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are biologically-inspired architectures made up of neural networks stuff, there are the most used architectures in computer vision, starting from there they are adapted for other domain. CNNs stacks multiples stages of feature extractors and higher stages compute more global, more invariant features.




In standard neural network a bi-dimensional representation was sufficient in order to contain all the differente weights. In particular CNN are still NN but with some property that permits to decrease the number of parameters to the standard of NN.

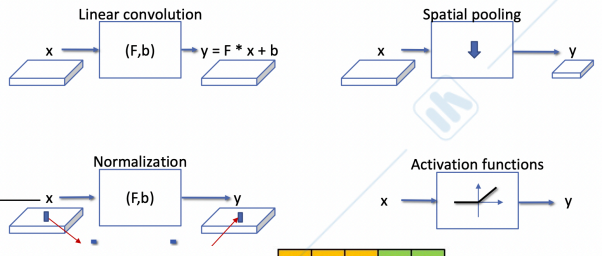



Each neuron is connected such as the connection is local in space (5x5 in 32x32 image, **local connectivity**) but is full in depth so its connect to all the tree deep channels simultaneously (All NN weights are arranged in 3 dimension).

Multiple neurons all looking at the same region of the input volume, stacked along depth. One single neuron just connect to small portion of our image.

**Weights sharing** means that the weights that are defined by this neuron to analyze this part of the image, are shared across all the image. So this neuron will analyze with the same weight the different parts of the image, different filter (neuron) different weights.

All the convolutional filter can be trained by standard back propagation (classification error is back propagated in order to value of the weights of the filter that we have) in respect of std NN have other special layers that are the *spatial pooling* and the *local response normalization,* the use of this layer is means to reduce the computational burden to increase the invariance and to make the optimization problem easier.

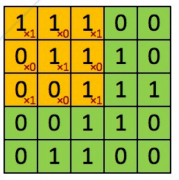


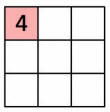

---

**Linear Convolution Layer:** the operation that is made is a *linear* one, we will need some non linearities after the application of this convolutional layer. Its *local* so each filter get connected with just one portion of image at the time. Its *translation invariant* the same filter will analyze different part of the image by using the same set of weights. Usually they are not using just a single filter but will deal with a *filter bank* in order to be able to form a richer representation of our data.

- Input $x = H \times W \times K$ array
- Filter bank $\mathrm{F} = H' \times W' \times K \times Q$ array
- Output $y = (H - H' + 1) \times (W - W' + 1) \times Q$ array

$$y_{ijq} = y_q + \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} \sum_{k=1}^{K} x_{u+i,v+j,k} F_{u,v,k,q}$$



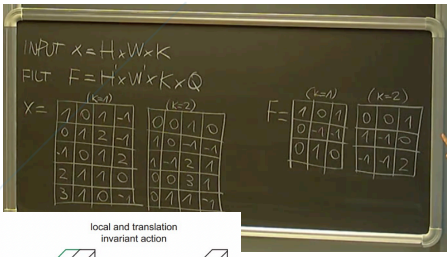




Input: K = 3 in previous example and could more than three

Filter bank: H' and W' the size of the kernel of the filter, Thant this filter have to be full in depth so K have to match the third dimension of our input. Q number of filters that are in our filter bank (in the convolutional layer or in NN represent the number of neuron in this convolutional layer).

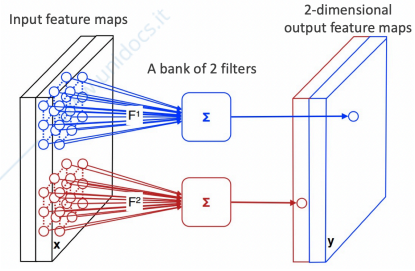

Out: that have a new dimension

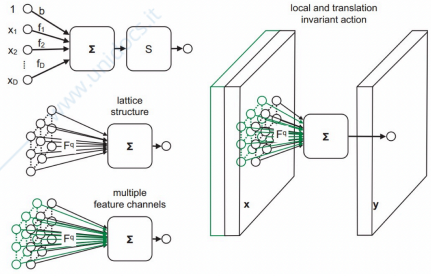
We also have *padding* that specify how many numbers we want to add outside out input in order to (done with zero usually). We end up with a h+1 and w+1 dimension so in the output we cancel the +1 in order to have the correct dimension.



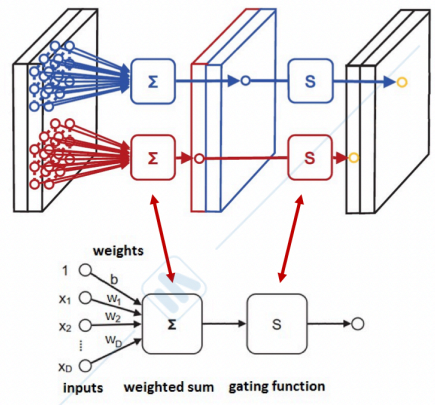

Stride = 1 padding = 1 cercare definizione **[11-17 lavagna]**

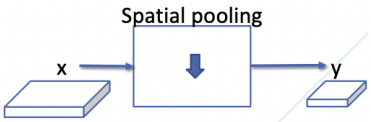
For each filter that we have in our filter bank we produce a new depth dimension in out output. Number of filter is the number of dimension that will have in output.

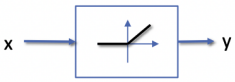


All this convolution is a linear combination of the value of the filters and the values in the image, if we take one convolution layer and we go in another convolution layer and other one, the fact that this is linear and have three of this is the same as have just a single one. In order to make this operation non-linear after a convolutional layer we have to include a non linearity also called **activation function**. This non-linearity is applied to each single value inside our volume (so each value in our output) very simple pertain in order to make non linearity).



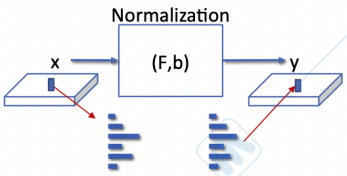

The **spatial pooling** layer is used to reduce the spatial dimension of out input, we have to tipe of pooling layer (max and average), this operate channel by channel in our input and they are define again with a size of the kernel that we want to apply and the stride. this pooling operation is used with the stride that is equal to the size of the kernel in order to be sure that we are reduce the dimension when we got inside this type of layer. Same kernel size, same reason of stride just average instead of maximum is the difference between the two type.

$$y_{ijk} = \max_{pq \in \Omega_{ij}} x_{pqk} \quad \text{Max pooling}$$

$$y_{ijk} = \operatorname{avg}_{pq \in \Omega_{ij}} x_{pqk} \quad \text{Average pooling}$$

**Local Response Normalization (LRN)** *not so used nowadays*

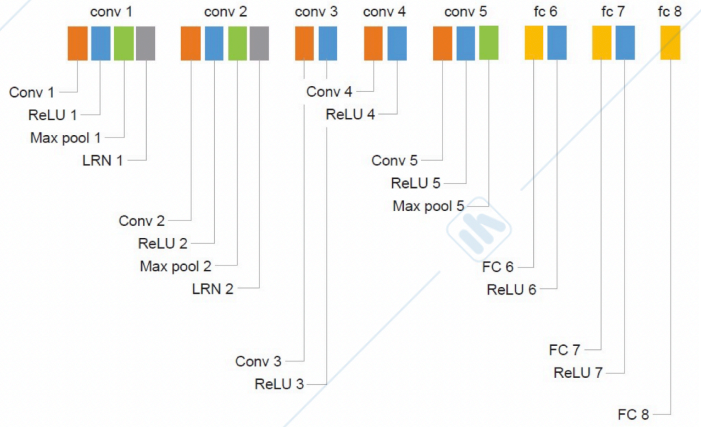
We want to build inside our architecture a sort of independence with the respect of the input condition. We want a output volume similar for the images even if they have different light condition and so on. They differ from how they operate on our input:




**Within channel:** operates independently on different feature channels; Rescales each input feature basing on a local neighborhood.

**Across channel:** operates independently at each spatial location and groups of channels; normalizes groups G(k) of feature channels and groups are usually defined in a sliding window manner.

# Training CNNs



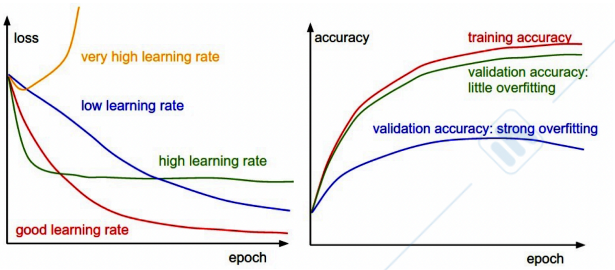

The general idea is that at first we have our convolution that try to get some information about the spatial information arrangement of our pixel in order to start to build a representation that has an higher level in our hierarchical representation. Then the spatial information is reduce going to the next one, pooling operation that reduce the spatial dimension of the output volume and in the same time the number of channel that we have in the different convolutional layers tends to increase.

We have a point where the spatial information is so small that we can destroy it and transform it in one dimensional vector.

Dropout is applied to the fully connected layers and not to the convolutional layers.

Data augmentation by jittering samples in the training set, and we cannot relabel the image you use this to simulate different type of light.

**Diagnose training**

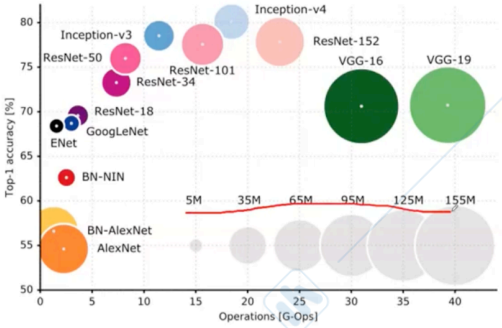



If we have a very big learning rate initially we gonna see that the loss is going to decrease and then it will even diverge (nan, infinity or big number), so the learning rate is too high for the problem.

If we have a learning rate that is too low we gonna need too much epoch to reach the minimum of the loss, decreasing every slow. A good one still decrescente at the end.

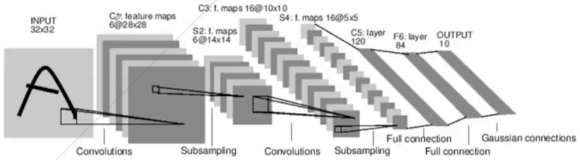
The ideal is that the performance that we have on validation set are closer to the one had in training: validation loss than accuracy means a little overfitting, it can be a little over the red line but is also fine, the true problem is when there is a huge variation between the curve, if they are too far we have a strong overfitting. It even append that start to increase with the training and than start to decrease, in this case means that the model it can be over parametrize in respect to the problem or the data, so need to include some strongest parameters to the regularization maybe increasing the weight decay or dropout or even do a more aggressive data augmentation, to permit to the model to actually memorize the training data and predict good on new data.

# Famous Architecture




**Imagenet:** is implemented the top 5 accuracy, in the competition from 2010/11 the performance of the methods submitted reached a plateaux in performance. The first CNN with 8 layers drop the error with AlexNEt, in 4 year the improvement was big, each method increase the number of layers. It start from shallow methods because they was just feature extraction with maybe intermediate representation and classifier.

**LeNet** (5) - LeCun and Bengio



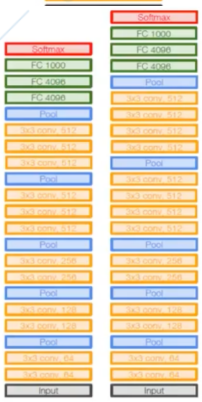

Create to recognize the adores and the postal code of the letters, so it can recognize letter and numbers. It was very small the input where 32x32. We have a first convolutional layer with 6 filters and we can se that the spatial dimension is reduce by an half, so we are dealing with a pooling operation, another convolution with 16 filter and then another reduce of the spatial. Then we go in a fully connected layer, so we destroy spatial information and we flatten the representation to go into this fully connected layers. At the end we have a output of 10.

**AlexNet**


We start from a big convolution filters with 96 channels, in input we have 3 channels, than this go in a 5x5 convolution increasing the number of channels… (5x5x256 parameters)

If we computer the number of parameters we can see that the vat majority of the layer is inside the Fully connected layer 4096x4096 and then.. this layer are very expensive

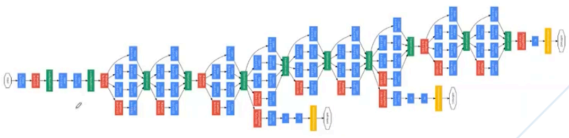
**From this we go in the VGG family architecture**



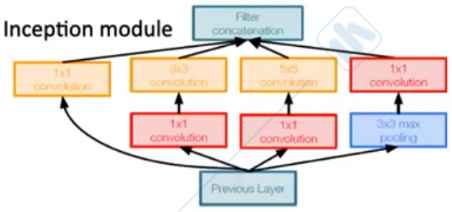
The main difference between the two architecture are that at the beginning in the AN we have some large convolutional filter that are substituted in VGG family by just using 3x3 convolutional filter. The reason behind this choice that stack multiple convolutional filters with 3x3 kernel size is equivalent to analyze the image with one single filter size 7x7 due to how the convolution are done. By decomposing the convolution in convolution with a smaller number of filters actually we have a saving in a total number of parameters, moreover after each convolution we put inside non linearities this permit to makes our CNN deeper, high number of layer and more non linearities that is important because is was permits to fit more diverse problem and data (so flexibility to predict each problem that we can face). We notice that the number of channel is increasing.

The main difference between VGG16/19 is that we have two more convolution layer, but the end of the three architecture are identical.

**GoogleNet (Inception) -** Sx input and dx the input



The idea is to create a good network topology that is repeated multiple time across our architecture, no more just convolutional layer one against the other, but one small network architecture local that is good enough to be repeated multiple times (network in a network).




They stack this block one on top of the other, the local topology is made up to be repeated multiples time. We go in parallel with different operation with different filters and than we concatenate the output of different branches to generate our output. From the output volume of the previous layers we apply the four part in the image.

The inception model is repeated 9 time stacked one on top of the other.

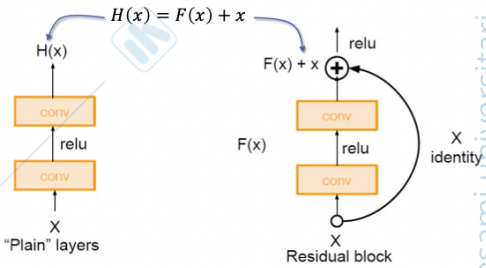
Than we have the real classifier in the end of our architecture but they have other two auxiliary classifiers in order to be able to inject additional gradient in the lower layer of our architecture.

When we start to back-propagate our gradient, from the output, since we have many multiplication due to the chain rule in order to compute our gradients in layers that are closed to the input, this information is reduced as we apply them multiple times this introduce the problem called *vanishing gradients* is effect is that at a certain point in our multiplication we end up with something very small close to zero and become not possibile to update the weight in the lower part of our architecture so we are not able to properly train all our

architecture. So the idea was to add additional classification layers, by create a loss that is the linear combination of the three losses of the classifier but giving more importance to the last one that actually what we need is how we are able to actually train our complete architecture. During the inference time we are not interesting into the additions classifier but we take just the out from the principal classifier.

We also have *stem network* that is the first part that process the input, one classifier on the end and we don't have the fully connected layer in order to reduce the total number of parameters.

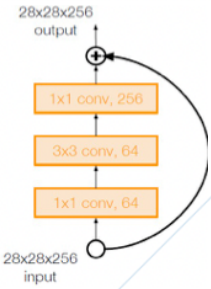
**ResNet**



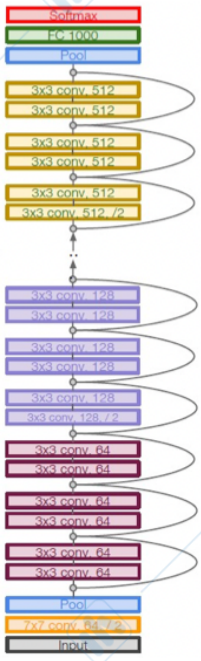

More layer we add and worst error we find ins espy to shallow methods, so its not so simple to go deeper. The problem is an optimization problem, deeper models are harder to optimize (vanishing gradients).

The first deep models came from the ResNet architecture, the idea is to use network layers to fit a residual mapping instead of directly tying to fit a desired underlying mapping. The information flow from the "plain" layers (previous ones) to the H(x).

The CNN try to fit a model so they are learning a representation that can be seen as a function, the idea is to slit this function H(x) in two parts an identity and F(x), so instead of learning the commute transformation we just learn F(x) since the identity is already learn (given for free) by our architecture. This is done by the old one plus a connection called the *residual connection* or *skip connection* that perform the identity.




In the standard *plain layers* in the backward (we go form the out to the in) you will have to compute the gradients and than with the chain rule we are going to compute the gradient with respect to the weights in the last one than multiply to for other gradient in order to obtain the weight in the other layer and so on. So multiply for something that is usually smaller than one, this gradient is reduced as we go back to the input. In the new one it happened the same thing, but we also have the connection this means that the gradient that we have at the end in one brach goes multiplied by the classical chain rule but we also have the connection: the derivative of the transformation (X) is 1, so the gradient can skip the inside layer and get in the part closer to the input unchanged.




So we made the optimization problem easier because we use need to learn the difference in respect to the identity transformation from one side, and on the o there side when we are back propagating the gradient we have a way to preserve the gradients that can go backward unchanged even in the first layer of our architecture.

ResNet is composed of stack of residual blocks, each one has two 3x3 convolutional layers. For deeper (ResNet 50+) we also have bottleneck layer to improve the efficiency, this are composed by 1x1 convolution that try to compress the information coming from the previous layers but maintain the same spatial information that we perform some padding operation to keep the size of out input unchanged than another expiation of dimension from 64 to 256. In this operation is important that the spatial size of our output is the same of the input one, that's because of the external connection to be able to sum the x with F(x) and for doing that they need to have the same size.

We can guarantee that even in the firsts layers of our architecture we received gradients that have a great magnitude, so we ca properly train our architecture in all different layer.

We don't have pooling layers, just two near the in/out inside the architecture this process is done by using stride equal to 2, at a certain point in the architecture we double the number of filters that we are considering and at the same time we reduce by one half the spatial dimension of the data without using the pooling layer and without compute the maximum value that is quite expensive.

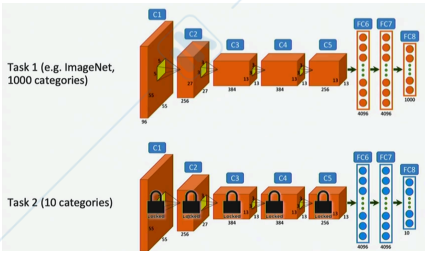
We can image this as it occupy alway the same quantity of memory since we have 64 different channel but we have a double of the spatial dimension than we double the number of filters and reduce the spatial dimension otherwise the quantity of memory that we are occupying it will grow exponential and in this type of architecture we are able to do it linearly.

# Transfer Learning

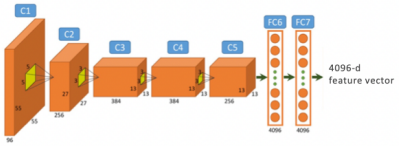
Sometime we don't have such amount of data but we can still use deep learning ?

Not so many people train an entire CNN from scratches because it is relatively rare to have a dataset of sufficient size, it's common to pre-train a CNN on a very large dataset and than using this CNN either for *fine tuning* or as a *fixed feature extractor* for the task of interest.

We cannot use the base architect train on 1000 categories if we have 10 categories to predict, the idea is to take the architecture that we have trained and cut it at certain point replace the old layers with new layers in particular the last one have the correct number of neuron. So we keep the weights in the first part of our architecture frozen and just train the weight in the layer that we have replaced, so we keep the representation learnt up to a certain level and from that point towards the higher level we replace the layer so they can lear a new representation.




There is a second possibility to use this as feature extraction, we take our pre-trained network, cut it at a certain point in the architecture and for each image we get a representation that is equal to the number of weights in the last layer that we maintain, so the output is a one dimensional vector with the dimension equal to the number of filters or neuron that are in the last layer.




So for each image we computed a descriptor so we can fed this descriptor together with the ground truth to any standard classifier, without using the DL to perform our complete inference. We just use it as it was a descriptor and then we use a standard classifier attached at this representation.

It was proved that if we train this architecture on imagenet and then using as feature extractor on different task the perform are good, so is a fast way to produce good result without study everything on that problem.

- New dataset is *small* and *similar* to original dataset: Train a linear classifier on CNN features from higher layers, since they are similar you can exploit the representation that the model has trains up to a very abstraction level. Since is small we are not able to actually train new layers of our architecture so the suggested way to use it is to extract the feature from some layer that is very close to the out and train an external classifier.
- New dataset is large and *similar* to original dataset: Fine-tune the CNN, since they are similar we can cut the architecture at very higher layer close to the out and replace some layer and fine tune the CCN, it can be done since we have a large amount of data. In this casa we can also think to train our architecture from scratch, but using the fine tuning we are starting from a warm state instead a cold one, completely random, train will be much faster.
- New dataset is *small* but very *different* from original dataset: Train a linear classifier on CNN features from lower layers, abstraction level very low so they are using a representation that could be used in different task even if they are very different.
- New dataset is *large* and very *different* from original dataset: Train CNN from scratch or fine-tune it, only case when is ok to train this can from scratch. Since they are very different even starting from a network that has learn something on a very different task does no represent an advantage starting point in order to speed up the training process because what to be learn is very different from the task.

So if the dataset in small the idea is to use our architecture as feature extractor but if they are similar we extract the feature from high level if they are different from very low layer.

# Model Compression

We just don't want a smaller size of the model, maybe we also want to keep the same accuracy of the original uncompressed model and maybe even improve it, for sure we want to speed up the inference. Smaller models means lower computational burden, lower power consumption and the possibility of running this time of model even on very cheap devices.

Transformer network are model with a number of parameters that is around 8.3 billion, almost no one are able to train it.

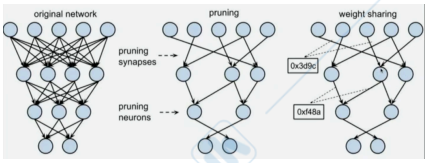
The importance of this model compression is also in order to be albe to use such big models on commonly available resources. So we can think about compressioni in two different way: one is to use standard CNN even if on cheap devices with very limited computational capabilities and in other hand make available such big model to the reach community in order to use it.

Main approaches: weight sharing, network pruning, low rank matrix and tensor decomposition, knowledge distillation, quantization and design low resource and efficient architectures. All the technics that we are studying can be applied to every type of neural architecture.

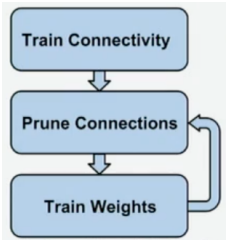
**Weight sharing**

The simplest form of network reduction involves sharing weights between layers or structures within layers (e.g filters in CNNs). Unlike other compression techniques standard weight sharing is carried out prior to training the original networks as opposed to compressing the model after training. Weight sharing reduces the network size and avoids sparsity, it is not always clear how many and what group of weights should be shared before there is an unacceptable performance degradation for a given network architecture and task.

**Network pruning**




Pruning weights is perhaps the most commonly used technique to reduce the number of parameters in a pre-trained DNN. We take the original network and we remove some connections in order to have less weights in our compressed model, we have the possibility to remove single connections or complete neurons (so incoming and out coming connection simultaneity removed). A further step can be also weighs sharing: maybe two connection had weights that are very similar we actually use the same weight for the two to save space.




The problem of this technic is that is iterative, we start with full connection in our model and we start to prune some synapses and neuron and we cannot expect that such a model could have the same accuracy of a complete model, even if some weight of some arches are very low they still contribute to the final estimate. So in order to avoid this problem after we removed some connection we do a train phase to let the remain weight to adapt to the fact that some of weight and neuron are not present anymore, this operation can be repeated several times.

This operation is a little bite expensive: original train, prune, train weights so we need multiple train to reach the size that we expected.

We can set a threshold that decides which weights or units (in this case, the absolute sum of magnitudes of incoming weights) are removed: threshold can both vary for each layer or be global for the whole network. We can have different approach like global threshold or adaptive one, so each layer could have a different one.

Instead of setting a threshold, we can set an upper-bound of number of neuron to get to be removed, maybe a percentage. So not a threshold but a target on how we want to reduce the size of our architecture.

The before mentioned criteria for pruning are all types of *magnitude-based pruning* (*MBP*), pruning is driven by the fact that we have different weight in our network so based on this information we select the one to be removed. MBP is the most commonly used in DNNs due to its simplicity and performs well for a wide class of machine learning models (including DNNs) on a diverse range of tasks. When we finished the first round of training we are ready to select which are the weight to be removed, so there are not other computation that have to be carried out.

In respect of the two method: set a threshold for each layer and one in the complete model, in general, global MBP tends to outperform layer-wise MBP because there is more flexibility on the amount of sparsity for each layer, allowing more salient layer to be more dense while less salient to contain more non-zero entries.

**Categorization of pruning techniques**:

1. *Magnitude-based pruning* where by the weights with the lowest absolute value of the weight are removed based on a set threshold or percentage, layer-wise or globally.
2. Methods that *penalize the objective with a regularization term* to force the model to learn a network with (e.g l1, l2 or lasso weight regularization) smaller weights and prune the smallest weights. This regularization force the architecture to learn weight that are more close to zero so we have more potentially candidate to be removed.

3. Methods that *compute the sensitivity of the loss function* in respect of the different weighs, when weights are removed and using this as a criterion for removing weights that result in the smallest change in loss. Give us more information than the previous one we remove weights that have lower magnitude we don't know what will be the impact on our final accuracy, here instead we compute what will be the effect of that neuron on final accuracy. So we know which one remove for sure and the effect on accuracy, so we remove just the one with lower impact on final accuracy.

4. *Search-based approaches* (e.g particle filters, evolutionary algorithms, reinforcement learning) in order to encode the connection structure and let the algorithm to automatically select which connection to keep and which to remove. All the connection can be represent by a one-dimensional with k entries vector and if we set one of this element to 0 means that this connection does not exist in the model, 1 otherwise. Coupling this information with other vector that contains the actual weights this permit to performe this evolutionally computation ignorer to understand which connection to keep and what weights to give to the connection that survive this evolutional algorithm.
   So simultaneously train both part, but is not feasible if the dataset is too big.

**Magnitute-based pruning: iterative pruning + retraining**

We train our complete network, we prune some connection on the basis of the threshold or percentage but we start from the connection that have a very low weight and than we performe again the training to adapt the weights to its new connection scheme. This tipe of pruning based approach is more effective in fully connected layers, in convolutional layers there is much information that can be removed. In fully connected layers we can remove 65% of our weight and still not have loss in finally accuracy, huge saving in term of computation and space for our model.

**Pruning using weight regularization**

We have see this penalization in order to regularize the network and try to not overfit the training data, the second effect that the penalization have is try to push your weight towards zero so is also easier to find some weights to be removed because many of them are pushed towards zero.

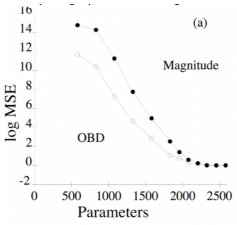
$$C(\boldsymbol{w}, \boldsymbol{v}) = \frac{\epsilon}{2}\Big(\sum_{m=1}^{h}\sum_{l=1}^{n} \boldsymbol{w}_{ml}^2 + \sum_{m=1}^{h}\sum_{p=1}^{C} \boldsymbol{v}_{pm}^2\Big)$$

**(Diagonal) Hessian-based pruning: Optimal Brain Damage**

The idea of model compression&speedup: traced by to 1990. Actually theoretically more "optimal" compared with the current state of the art, but much more computational inefficient because of the "saliency" and delete the parameters with the smallest. *Saliency* is defined as the effect of that neuron on the training error, we have to compute the salience of each parameter as the second derivative of each parameter with respect of our final error.

$$S_k = \frac{\partial^2 E}{\partial^2 u_k} u_k^2.$$

In order to apply this methods we have to train our architecture and during the train we still compute our first order derivative (gradients) in order to performe back-propagation tu adapt the weights, when we are arrived at a good solution we have to compute the second derivatives of every parameters in respect to the final error. This is the reason why this is not done usually, the number of derivatives that we have to compute grow quadratically. In order to try to keep this more treatable there is the idea of just using the diagonal, so compute the second derivative with respect of each signal variable alone without considering the mixed terms. After compute this saliencies for the different parameters, we sort them and we start to remove the ones with the lowest impact on our final accuracy, and we iterate so we adapt out network to the fact that we have less connection and repeat until maybe a final size of the model been reached.




Theoretically sounds good perhaps the best way to identify which connection to remove the problem is that computational unfeasible.

**Search-based approaches**

We have an external to our training process that select which are the connection to be kept and which one has to be switch off. So we have an external encoding that learns it and then we go into the training process with this new architecture scheme.

**Structured vs unstructured pruning**

Another important distinction to be made is that between structured and unstructured pruning techniques:

- *structured* aims to preserve network density for computational efficiency (faster computation at the expense of less flexibility) by removing groups of weights. It's faster to remove completely one neuron in terms of operation to perform intend of try to remove just single connection.

- *unstructured* is unconstrained to which weights or activations are removed but the sparsity means that the dimensionality of the layers does not change.

There is some penalization that we can use in order to induce the weights in different groups to go toward zero ,we are not forcing the weights in general in our architecture but we are computing this penalization terms on individual blocks of weights such that could be easier to remove one of them.

The main difference is that with the classical penalization in a given layer it may happen that we have one way in the first neuron that is close to zero than we have secondo one in other different neuron…this lead to inefficiency because when we want to remove one single neuron we have some connection that are important, by changing this penalization terms and using this group-wise L2 penalization we can induce all the weights in a single group to go towards zero so we can remove one single neuron because all his connection are used toward zero.

**Low Rank Matrix & Tensor Decompositions**

The idea starts from the observation that most weighs are in the fully connected layers and that FC layers are implemented with a single matrix multiplication, i.e. connections form a FC layer with N units into one with M units can be stored in a NxM matrix.

$$W = USV^T$$

$$W \in \mathbb{R}^{m\times k}, U \in \mathbb{R}^{m\times m}, S \in \mathbb{R}^{m\times k}, V^T \in \mathbb{R}^{k\times k}$$

$$\begin{pmatrix} & & \\ & W & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & U & \\ & & \end{pmatrix} \begin{pmatrix} w_1 & & \\ & w_2 & \\ & & w_3 \end{pmatrix} \begin{pmatrix} & & \\ & V^T & \\ & & \end{pmatrix}$$

The weight matrix W and we perform a singular value decomposition, so we end up expanding our matrix as multiplication of 3 matrix one of this contain the singular values that are reported in a decreasing order S.

The idea is to discard the singular values that are lower than a certain threshold or we just keep the first bigger singular values instead of considering the *k* original singular value we consider a number *t* lower than the original. So when we multiply we obtain something slightly different from our original matrix.

$$\widetilde{W} = \widetilde{U}\widetilde{S}\widetilde{V^T}$$

$$\widetilde{W} \in \mathbb{R}^{m\times k}, \widetilde{U} \in \mathbb{R}^{m\times t}, \widetilde{S} \in \mathbb{R}^{t\times t}, \widetilde{V^T} \in \mathbb{R}^{t\times k}$$

$$Rank(\widetilde{W}) = t < k = Rank(W)$$

We have a storage that is lower in respect what we needed before because by decomposing it in different matrix we can save parameters. S is diagonal, we have txt so we can just save t parameter instead of the all matrix.

Compression rate: $O(\frac{mk}{mt+t+tk})$

Theoretical error: $\left\|AW - A\widetilde{W}\right\|_F \leq w_{t+1}\|A\|_F$

We remove some singular value but we want to still obtain something that obtain some similar output of our original matrix. We have a theoretical bound of this error, if we have an activation A that have to be multiplied by our W in order to compute the activation of the next layer we know that the difference in terms of Frobenius norm is equal or less to the magnitude of first singular value that we have removed by the frobenius norm of the activation of our previous layer. If we had removed just the singular value that are very small that means that the different will be very small so we end up with activation that are very similar to our original ones. If we remove many singular value, so if we remove also the one with a magnitude that is not so small, the difference could start to increase. This theoretical is guided by the first removed value.

**Knowledge distillation**

This involves learning a smaller network from a large network using supervision from the larger network and minimizing some measure between the smaller one and the biggest one: entropy, distance or divergence between their probabilistic estimates. We don't want to cut some layer or connection or compress it, we start from scratches from another smaller network that already has the size that we intent to have in out final model and we learn this second model trying to replicate what the first network do in our network. So we use the supervision of the bigger model on the smallest model in the state of art is call *teacher and student.*

The idea of reducing model size by learning a student from a ensembles network, usually when train a network obtain a given configuration if train another time your model we end up with a different configuration and slightly different performance, if we repeat multiple times this operation we end up multiple different model. If we average the prediction of this enabled model you obtain something that is better than any of the single methods.

The idea is to have a smaller model that learn from an ensemble of bigger models in order to have something that this even better than individual bigger models, since they know this property they use a teacher network to label a large amount of unlabeled data and train a student network using supervision from the pseudo labels provided by the teacher. They find performance is close to the original ensemble with 1000 times smaller network.

Another approach (2015) train the smaller model by trying to imitate the activation of the bigger model (last layer) not just the prediction because they could be binary so there is not much information to back propagate into the smallest model. The bigger model have learn a very powerful representation of the data that is something the smallest model not be able to learn, but is able to emulate what the bigger can do.

**Quantization**

We are not removing some part of our architecture, we are chasing the representation of our weights.

Quantization is the process of representing values with a reduced number of bits. In neural networks, this corresponds to weights, activations and gradient values. Typically, when training on the GPU, values are stored in 32-bit floating point (FP) single precision. Half-precision for floating point (FP-16) and integer arithmetic (INT-16) are also commonly considered. The advantages of considering the last two type is a benchmark about speed of interference time on the standard GPU with ResNet architecture.

In FP-16, the result of a multiplication is accumulated into a FP-32 followed by a down- conversion to return to FP-16.

To speed up training, faster inference and reduce bandwidth memory requirements, ongoing research has focused on training and performing inference with lower- precision networks using integer precision (IP) as low as INT-8, INT-4, INT-2 or 1 bit representations.

There is another possibility in our architecture, we don't consider pure integer representation but we can represent mixed precision, maybe all the weights in our architecture are represented as integer but when we have to perform actual computation we have a floating point that multiples this weight such that we can have more flexibility in order to be able to represent a different portion of our data. If we need something that is outside our range we multiply by a floating point that is higher than what we have. This is a more flexible solution that permits to accommodate the range given by the representation that we are using in order to be able to adapt to the range of value that are needed for that particular layer. This flexibility comes because is possible to have different scaling value for different layers, in order to perform it's activation.

*Low resource and efficient architectures*

Design from the beginning low resource and efficient architectures, we start from a compressed model that doesn't need to be compressed because is already a small one and its already efficient in order to be able to obtain good performance in tasks.

**MobileNet**

Compression of convolutional neural networks for embedded and mobile vision applications using *depth-wise separable convolutions* (DSC) and use two hyper parameters that tradeoff latency and accuracy. DSCs factorize a standard convolution into a depth-wise convolution and 1x1 point-wise convolution. The idea is looking closing to what convolution performs and to separate it in order to have less parameters.

**SqueezeNet**

Reduce the number of input channels to 3x3 filters using *squeeze layers* and downsample later in the network to avoid the bottleneck of information through the network too early and in turn lead to better performance. Start to reduce the spatial dimension of input later in the architecture to avoid the bottleneck but trying to reduce the number of input channel by using the squeeze layer, so we can keep the number of parameters very low cause we constantly reduce the number of channel.

**ShuffleNet**

ShuffleNet uses *point-wise group convolutions,* i.e using a different set of convolution filter groups on the same input features, so we don't have a single filter who acts on previous layer but when we arrived at this layer we have multiples group of filter that actually operate on the same input. We can increase the inner parallelism in this architecture and obtain much reacher representation with a lower number of layers since in the same layer we have multiples groups that process our input in different ways. We are not taking one deep architecture but less deep architecture but wider, exploit parallelism to trying to obtain the same type of representation.

**DenseNet**

Gradients can vanish in very deep networks because the error becomes more difficult to back-propagate as the number of matrix multiplications increase. DenseNets address gradient vanishing connecting the feature maps of the previous layer to the inputs of the next layer, similar to ResNet skip connections. DenseNets can be made wider and shallower to become more memory efficient if required. It has some control parameter that permit to actually define the new architecture that is shallower and wider if needed to the task.

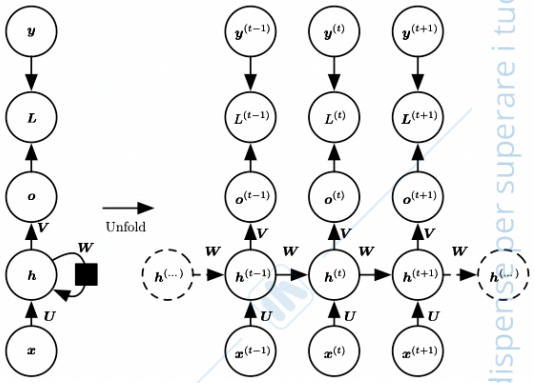
# Recurrent and Recursive Networks

Recurrent neural networks or RNNs are a family of neural networks for processing sequential data.. Much as a convolutional network is a neural network that is specialized for processing a grid of values X such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $x(1), \ldots, x(\tau)$, we don't have anymore a single input but a sequence of them.

Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. The RNN are able to process sequences of data without being limited to be able to process fixed sequence of data, we can use traditional NN or CNN to process sequential data, the problem will be able to just use a fixed length in our sequence.

Most recurrent networks can also process sequences of variable length, during training is better if the baches has the same size so we select from our data those sequence that have the same length and we put them together in the same batch for faster processing during training, but we are able to train them even without doing this.

We have a computational graph use to compute the training loss of a RNN use to map the input sequence of *x* value two the corresponding sequence of out values *o, y* is the ground truth and *L* is the loss that takes as argument our output and ground truth and compare them.




The RNN in the simplest version we have some connection that go from the input to our hidden layer that is parametrize by this matrix *U*, we have a connection from the hidden state to again the hidden state that is parametrize by a matrix *W* that is what forms the recurrent connection and the we have this connection that connect the hidden state with respect of output that now is parametrize by this weight matrix *V.*

The trainable parameters are some from input-to-hidden, recurrent connection from the hidden to itself and then from the hidden to output, then we can expand the representation across time. This is the way in which we can map some information from our previous tilmestep to our next times step. The *W* matrix is the same for each time-step, so with no change in our parameters this mean that all its behavior is just describe by a single *U/W/V* matrix. This means that if we are processing a sequence longer than the ones during training actually that can be handle our model having the same number of parameters, having to deal with this information among time does not add the number of parameters to be need to handle this type of data. The parameters are shared across time, when we are going to update this weights they are going to be updated to the respect to the batch that we are analyzing but than thy are the same for each timestamp.

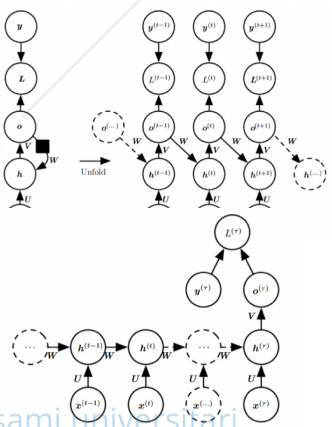
The basic idea behind RNNs it to make use of sequential information. In a traditional FNN it is assumed that all inputs (and outputs) are independent from each other, so there is not more information that we can extract from processing some inputs together. For the the sequential information this is not true because there is some more information int he sequence itself that can help us to understand and predict what could happened next.

For many taks the independence assumption among the input do not permit to capture the relation between the data.

RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations, if we don't consider the recurrent connection W we end up with a classical NN.

Another way to think about RNNs is that they have a «memory» that stores relevant information about, like previous computation. The hidden state h(t) can be considered as a sort of memory of the network and it capture the information about what happen in the previous timestamps. The output at step t $o(t)$ is computed only based on the memory at time *t,* the only way to go to from *h* to the *out* is to go trough the connection *W,* the out just depend from the memory on the current timestamp. Unlike traditional feedforward NN which use different parameters for each layer a recurrent share the same parameter across all time step.

This has different benefit: this reduce the number of parameters because we don't need to learn a new matrix for each timestep (long sequence more parameters) and this sharing of parameters also allow us to process sequence of different length cause there is not change in the architecture that is already ready to process sequences of a given length we just need to add a new connection and process the next input. The same task at every time-step with different input.



The architecture that we have been looking at so far consider that at each timestep we have an output.

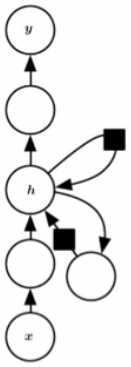
Some of the most relevant RNN architectures are:

- RNNs that produce an output at each time step and have recurrent connections between hidden units

- RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step.
- RNNs with recurrent connections between hidden units, that analyze an entire sequence and then produce a single output.

The computation in most RNNs can be decomposed into 3 blocks of parameters and associated transformations:

1. From the input to the hidden state
2. From the previous hidden state to the next hidden state
3. From the hidden state to the output

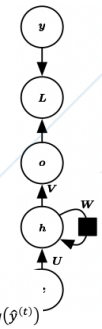



For the RNNs architectures seen so far each of the 3 blocks is associated with a single weight matrix, so there is just one single layer that connects this differs processing blocks.

Experimental evidence is in agreement with the idea that we need enough depth in order to perform the required mappings. We have some idea where to add depth: from one hidden state to the other, since the can be a problem because give more length to the information that flow from a hidden state to the next one, usually some skip connection are used. Or another possibility is to add more layer between hidden state and output, or from the input to hidden state.

**Train a RNN**

The train is similar to traditional NN, the concept of back-propagation is used in order to train our network but whit some differences because we have the time parameter to consider. Since the parameters are shared by all the time steps in the network, the gradient at each steps depends on both: the calculation of the current time step and of the previous time step. This is why the RNN training algorithm is called *Backpropagation Through Time* (*BPTT*).




In order to compute the hidden state at time $t$ $h^{(t)}$ supposing that we are using $tanh$ (non linear activation) this is computed as the sum of: the *U* matrix that multiplies the input at the current time step $x^{(t)}$ plus the matrix *W* that multiplies the hidden state at the previous time step $h^{(t-1)}$. Then the output *o* (in the figure) $\hat{y}$ (in the formula) is computed with some other non linearity applied to *V* multiplies the hidden state at the current time.

$$h^{(t)} = \tanh\left(Ux^{(t)} + Wh^{(t-1)}\right)$$
$$\hat{y}^{(t)} = softmax\left(Vh^{(t)}\right)$$
$$E^{(t)}\left(y^{(t)}, \hat{y}^{(t)}\right) = -y^{(t)} log\left(\hat{y}^{(t)}\right)$$
$$E(y, \hat{y}) = \sum_t E^{(t)}\left(y^{(t)}, \hat{y}^{(t)}\right) = -\sum_t y^{(t)} log\left(\hat{y}^{(t)}\right)$$

We also consider cross entropy loss as the loss (i.e. error) function we want to minimize: at each time step we have our euro measure between the out a this time step and the ground truth at this time step and in order to compute the complete loss over all sequence we have to sum the contribution of the loss competed at each time step summed over all the time step.

The goal of BPTT is to calculate the gradients of the error in respect of the weight matrix $U$, $V$, $W$ and then learn good values of these parameters using Stochastic Gradient Descent. Just like we sum up the errors, we also sum up the gradients at each time step for one training example.

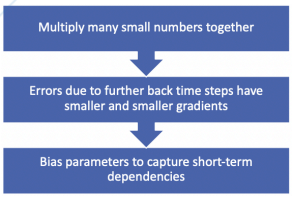
$$\frac{\partial E}{\partial U} = \sum_t \frac{\partial E^{(t)}}{\partial U} \text{ and } \frac{\partial E}{\partial V} = \sum_t \frac{\partial E^{(t)}}{\partial V} \text{ and } \frac{\partial E}{\partial W} = \sum_t \frac{\partial E^{(t)}}{\partial W}$$

Depends on the RNN architecture the computation maybe quite simple for partial derivative with respect of *V* because is the first one, because don't go inside the recurrent connection. The computation for the other two partial derivatives will be much more complicated to compute, because in order to compute this we need to back-propagate the gradients from the out through the network all the way back to the first instant that we are considering.

Computing the gradient with respect to $h^{(0)}$ involves many factors and repeated gradient computations. It may happen that we have many values $> 1$ we can have an exploding gradients, it just consider clipping the gradients to some maximum value to don't encoring in exploding gradient and they are very easy to find them.
It may happen that we have many values $< 1$: we can have vanishing gradients and solve with three different possibility:

1. *Activation function*, we choose properly the linear non activation function.
2. *Weight initialization*
3. *Network architecture* that try to limit the problem of this vanishing gradients.

The problem with vanishing gradient is that you cannot identify them because at a certain point in the architecture you don't know if the gradient is actually low or its actually vanishing from computation, so we have to print it because we are not able to be sure that is what is happening. Due to this problem we are not able to train the entire architecture but just the last part (close to the output).

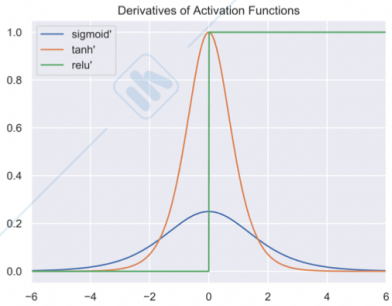



In this domain (RNN) we have other type of problem due to vanishing gradient: multiply many small numbers together errors due to further back time steps have smaller and smaller gradients, this mean that

some information that is very close to the initial time set is not able to be properly explored. So the vanishing gradient bias parameters to capture short-term dependencies. Since gradient is vanishing through the time we are just able to exploit information from very recent input but we are not able to train very long term dependencies, the state at those steps does not contribute to what you are learning.

Vanishing gradients also happen in deep Feedforward Neural Networks (recall the introduction of ResNets) but RNNs tend to be very deep (as deep as the sequence length), since backpropagation work through time have multiple step of our sequence increasing the depth of our architecture and this could be very deep which makes the problem much more frequent than in CNN.

**Solution 1: Activation function**




Properly choosing activation function, we look the derivatives because the vanishing is related to gradients. Sigmoid and tanh tends to shrink the value of the gradient unless in a restricted portion. Relu prevent the gradient to shrink when greater than zero.

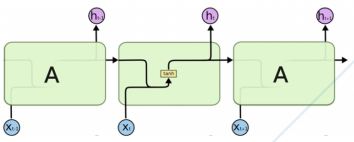
**Solution 2: Parameter initialization**

Initialize weights to identity matrix and initialize biases to zero. This helps preventing the weigths from shrinking to zero.

**Solution 3: Network architecture**

Use a more complex recurrent unit with $I_n = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$ gates that are able to control the amount of information that is passed through the next hidden units. The most useful solution

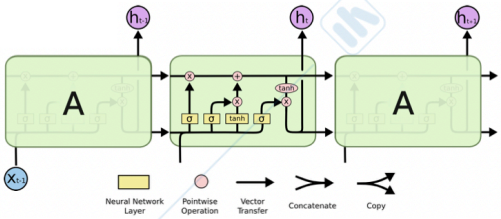
The most effective sequence models used in practical applications are so called *Gated RNNs*. They include both *Long Short-Term Memory* (LSTM) or *Gated Recurrent Unit* (GRU). The are based on the idea of creating paths through time that have derivatives that neither vanish nor explode, similar to ResNet, it's an extension we can have some gradient that can flow backward in time without modification. To achieve this goal they use connection weigths that may change at every time step.



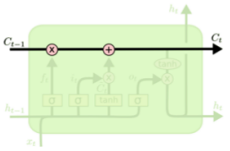

In standard RNNs, repeating modules contain a simple computation node.

The LSTM model is organized in cells which include several operations and it has an *internal state variable*, which is passed from one cell to another and modified by the following Operation Gates: forget gate, input gate and output gate.

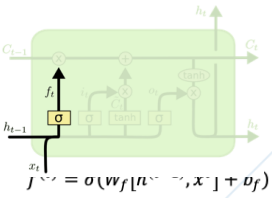
In the bottom part the input goes throw some computation taking into account also what we had in the hidden state of our previous step.




First part is called the *cell state,* maintains a vector $C_t$ that has the same dimensionality as the hidden state $h_t$. Some information can be added or deleted from this state vector via the forget gate or the input gate, so we have soothing here that have the same dimensionality as the hidden state that flows in the upper part of our node and some information that can be added or removed from this cell state considering what are called the *forget gate* or the *input gate*.

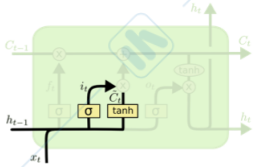


It is a sigmoid layer that takes the hidden state at time-step $h_{t-1}$ and the current input at time $x_t$, concatenates them and applies a linear transformation followed by a sigmoid and out this $f^{(t)}$.



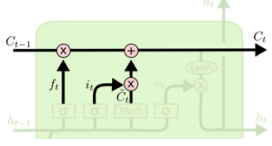

Because of the sigmoid, the output of the Forget Gate is between 0 and 1: If $f^{(t)} = 0$ then the previous internal state, if completely forgotten and if $f^{(t)} = 1$ then the previous internal state will be passed through unaltered and all the value between to select what amount of information should be remember and how much forgot.

The *Input Gate* determine which entries in the cell state to update by computing 0-1 sigmoid output $i^{(t)}$ then determine what amount to add/subtract from these entries by computing a tanh output (valued –1 to 1) function of the input and hidden state $\tilde{C}^{(t)}$. So it takes the previous hidden state and current input in order to compute this new information.

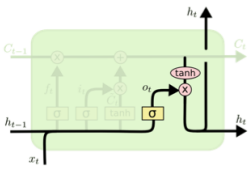



$$i^{(t)} = \sigma(W_i[h^{(t-1)}, x^t] + b_i)$$
$$\tilde{C}^{(t)} = \tanh(W_C[h^{(t-1)}, x^t] + b_C)$$

The previous state is multiplied by the forget gate and then added to the fraction of the new candidate allowed by the output gate $C^{(t)}$. In other words, cell state is updated by using component-wise vector multiplication to «forget» and vector addition to «include» new information.



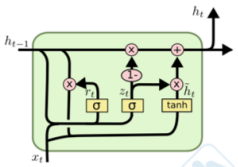
Hidden state is updated based on a «filtered» version of the cell state, scaled to –1 to 1 using tanh. Output gate computes a sigmoid function of the input and previous hidden state to determine which elements of the cell state to «output».



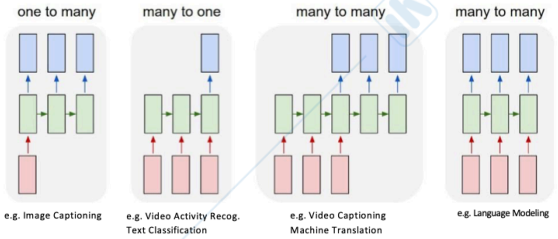
Key concepts, use gates to control the flow of information: Forget gate gets rid of irrelevant information, store relevant information from current input, selectively update cell state and output gate returns a filtered version of the cell state. Backpropagation through time with uninterrupted gradient flow.

*Gated Recurrent Unit* (GRU) alternative RNN to LSTM that uses fewer gates, combines forget and input gates into «update» gate, eliminates cell state vector. GRU has significant less parameters than LSTM and trains faster.

$$o^{(t)} = \sigma(W_o[h^{(t-1)}, x^t] + b_o)$$




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$




Each type of RNNs has its problem on which they perform so there is not a clear winner.

# Federated Learning

The standard setting in Machine Learning considers a centralized dataset processed in a tightly integrated system, but in the real world data is often decentralized across many parties, so we have a shift of paradigm.

If we centralize the data: sending the data may be too costly, maybe generate several TBs of data a day and some wireless devices have limited bandwidth/power. Also data may be considered too sensitive: we see a growing public awareness and regulations on data privacy and keeping control of data can give a competitive advantage in business and research.

If just learn a different model on each party (device): the local dataset may be too small and so we could have sub-par predictive performance, due to overfitting, or non-statistically significant results. Also the local dataset may be biased and be not representative of the target distribution.

*Federated Learning* (FL) aims to collaboratively train a ML model while keeping the data decentralized. We are not sending the data over the internet, the data are remains on the given divide that are acquire them but we want to design some strategy in order to be able to train such a ML model without having to send the data across the internet.

So let's suppose that we have a centralized party and in this case four different parties that are decentralize, the first thing to do it's to initialize the model and send this model to the different parties. This is important such that each one of the party will start the same model as the centralize one, this is important because after we can easily merge then when they are send to our central server.

Each parties makes an update using its local dataset, than each one send the updated model for aggregation in the central server. When the server has aggregated the updates, send the aggregated version of the model back to the parties. At this point each parties can update their copy of the model and iterate the process.

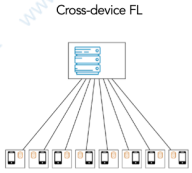
We would like the final model to be as good as the centralized solution (ideally), or at least better than what each party can learn on its own.

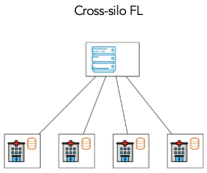
In *distributed learning*, data is centrally stored (e.g., in a data center): the main goal is just to train faster. We control how data is distributed across workers: usually, it is distributed uniformly at random across workers, cause of the capability of the single workers we could send more to a more capable one, in order to have a synchronized update step and not wait for the slower one to finish the task in order to update.

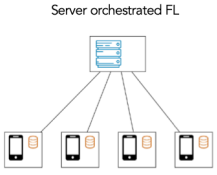
In *FL*, data is naturally distributed and generated locally, data is not independent and identically distributed (non-i.i.d.), and it is imbalanced. Additional challenges that arise in FL: enforcing privacy constraint, dealing with the possibly limited reliability/availability of participants and achieving robustness against malicious parties.

In FL we could have other distinction depending on the type of devices that we are considering:

- *Cross-device FL*: massive number of parties (up to 10^10), small dataset per party (could be size 1), limited availability and reliability, some parties may be malicious and so we have to provide some robustness against them.
- *Cross-silo FL*: 2-100 parties, medium to large dataset per party, reliable parties since are certified, almost always available and parties are typically honest.
- *Server orchestrated FL*: server-client communication, global coordination, global aggregation. Server is a single point of failure and may become a bottleneck, this also guarantee all the release of this model.
- *Fully decentralized FL:* Device-to-device communication, no global coordination, local aggregation and naturally scales to a large number of devices.





### FedAvg (local SGD)

We consider a set of *K* parties (clients), each party *k* holds a dataset $D_k$ of $n_k$ points. In total we will have a global dataset $D = D_1 \ldots \cup \ldots \cup D_k$ that is composed by the different dataset that belong to the different parties and we will have a total number of points (data) $n = \sum_k n_k$ that present the submission over the different number of data each client actually has.

We want to solve problems of the form $\min_{\theta \in \mathbb{R}^p} F(\theta; D)$, $\theta$ is the model parameters in general real space with p dimension) with respect all the data that we have. This covers a broad class of ML problems formulated as empirical risk minimization. Our solution will be combination weighted by how many points each dataset actually has of each model optimize for one of the different dataset that our client actually have. This is the only feasible solution that we have, each parties can train only each own dataset since the dataset is not shared, the solution will try to merge in someway the different model, weighted by how many data each dataset is actually composed of.

$$F(\theta; \mathcal{D}) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\theta; \mathcal{D}_k)$$

and

$$F_k(\theta; \mathcal{D}_k) = \sum_{d \in \mathcal{D}_k} f(\theta; \mathcal{D})$$

We have two different part (server/client side), on server side: we are not considering at each iteration each party that we have but we random sample some $\rho$ of the party that we have. We initialize the parameter of our model and we select a random set of our clients $\rho$ and for each client in parallel we perform an update of the model parameters and then when each client has return the update vision then we going to average the parameters with the respect of the cordiality of the local dataset.

**Algorithm** FedAvg (server-side)
**Parameters:** client sampling rate $\rho$

initialize $\theta$
**for** each round $t = 0, 1, \dots$ **do**
  $\mathcal{S}_t \leftarrow$ random set of $m = \lceil \rho K \rceil$ clients
  **for** each client $k \in \mathcal{S}_t$ in parallel **do**
    $\theta_k \leftarrow \text{ClientUpdate}(k, \theta)$
  $\theta \leftarrow \sum_{k \in \mathcal{S}_t} \frac{n_k}{n} \theta_k$

**Algorithm** ClientUpdate$(k, \theta)$
**Parameters:** batch size $B$, number of local steps $L$, learning rate $\eta$

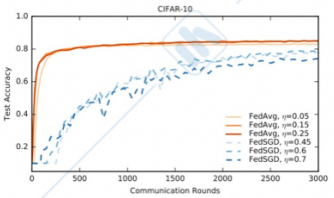
**for** each local step $1, \dots, L$ **do**
  $\mathcal{B} \leftarrow$ mini-batch of $B$ examples from $\mathcal{D}_k$
  $\theta \leftarrow \theta - \frac{n_k}{B} \eta \sum_{d \in \mathcal{B}} \nabla f(\theta; d)$
send $\theta$ to server

On the client update side: we have to set the batch size and how many iteration set are we performing on the client side before giving our update model to our centralized server. Classical SGD but on the data that we have on our client for a total number of step equal to L. When we have finished we send our updated parameters to the server that will average them in order to update our central model. This model is updated and send back to the different client until we have finished the total number of rounds that we want to perform.

We have some special cases:

- For $L = 1$ and $\rho = 1$, it is equivalent to classic parallel SGD: updates are aggregated and the model synchronized at each step
- For $L > 1$: each client performs multiple local SGD steps before communicating back to the server.

FedAvg with $L > 1$ allows to reduce the number of communication rounds, which is often the bottleneck in FL (especially in the cross-device setting). It empirically achieves better generalization than parallel SGD with large mini-batch and convergence to the optimal model can be guaranteed for i.i.d. data [Stich, 2019] [Woodworth et al., 2020] but issues arise in strongly non-i.i.d. case so is an uppercase because we are not in this case. The data are not i.i.d in our case.




Another way can be learn some personalized models: learning from non-i.i.d. data is difficult/slow because each party wants the model to go in a particular direction. If data distributions are very different, learning a single model which performs well for all parties may require a very large number of parameters. Another direction to deal with non-i.i.d. data is thus to lift the requirement that the learned model should be the same for all parties ("one size fits all"). Instead, we can allow each party *k* to learn a (potentially simpler) personalized model $\theta_k$ but design the objective so as to enforce some kind of collaboration.

So the local model could be too much biased towards the data that we are capturing but never the less we don't want to create a global model, because of the too many parameters and it will be too slow since the data is very unbalanced. We can learn some local model but using a loss function that exploit some formal collaboration.

One solution is to regularize personalized model to their mean, using a penalization terms on how different the local weights are with the respect of the average of the weight of all the different models.

$$F(\theta_1, \dots, \theta_K; \mathcal{D}) = \frac{1}{K} \sum_{k=1}^{K} F_k(\theta_k; \mathcal{D}_k) + \frac{\lambda}{2K} \sum_{k=1}^{K} \left\| \theta_k - \frac{1}{K} \sum_{l=1}^{K} \theta_l \right\|^2$$

Or something that came from the meta-learning so a global model which easily adapts to each party, not a single model, but different for each party but each one of this local models can be generated by the central one with very few iteration.

$$F(\theta; \mathcal{D}) = \frac{1}{K} \sum_{k=1}^{K} F_k(\theta - \alpha \nabla F_k(\theta); \mathcal{D}_k)$$

# Generative Adversarial Networks

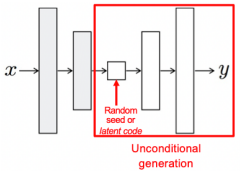
Starting from representation we want to go back to our image.

Generation (from scratch): the task of learn to sample from the distribution represented by the training set. We want to be able to generate meaningful example that are likely to belong to our training set. We will start with some noise vector and we want starting from this random vector to generate something that is in the distribution of the training set.

Image-to-image translation: we want to traslate from image labels to realistic road scene, revert form black and white to colors.

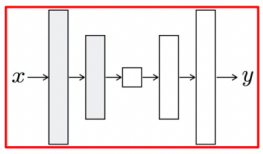
**Designing a network for generative tasks**

We need an architecture that can generate an image, something similar to this we we talk about auto encoder. The second part is the generative part which starting from something that is small actually ends up with generate something that is bigger. When you have train the auto encoder you could just cut the encoding part and by generating some random seed of the proper dimension we can end up with generating something, you have no control of what you are going to generate, so it's unconditional generation but this could work.

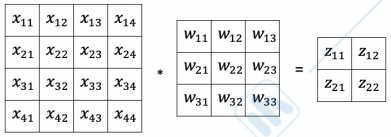



In the case a image to image translation, it's something very similar to an auto-encoder because starting from an image we want to end up with something that is again an image.

We need an architecture who generate data and also have to design the right loss function in order to be able to drive properly this process.

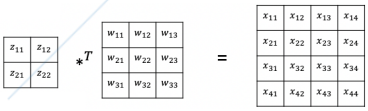



We pass from regular convolution to transposed convolution that is able to take our input (2x2) performe the transpose convolution with the 3x3 filter in order to obtain a 4x4 output. Before we have a 4x16 and now a 16x4 that we multiplies for a 4x1 to end up with a 16x1.



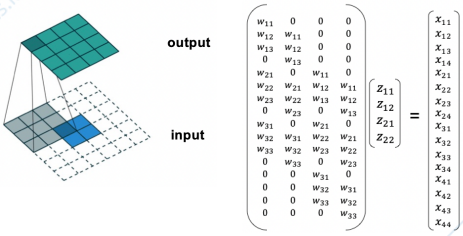
So he compute a coalition for every position in which we could have at least one element of our filter that overlaps one element in our input, the first position in which we perform this convolution is the one where just one single location in our filter overlaps the input and this correspond to our first output.


This traspose convolution is not the inverse of the original convolution operation, simply reverse dimension change.



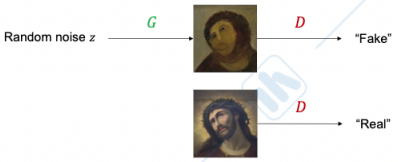
In order to achieve this task we need more than one network, in particular we have two networks which have opposing objective:

- *Generator*: learns to generate samples. Take some random vector as input and it generate our image.
- *Discriminator*: learns to distinguish between generated and real samples.




In this way we are not using a loss that says we are generate something that is good or not, but we are training a second network that is a discriminator that learns to distinguish between the generated and the real samples.

We want the generator as good as possible in order to foolish the discriminator, so is not able anymore to distinguish the sample from the generated ones. In the oder side we have this discriminator that we want to be better and better to find which are the image to be generated and which are the one instead that are real ones.




We have this to adversarial part and this is what permits us to train a generator to actually create some realistic data.

The discriminator $D(x)$ should output the probability that the sample x is real, we want $D(x)$ to be close to 1 for real data and close to 0 for fake. We could compute the expected conditions log likelihood for real and generated data, since the x is generated it means that x actually is something generated by our generator staring from our input noise z $G(z)$.

$$\mathbb{E}_{x\sim p_{\text{data}}} \log D(x) + \mathbb{E}_{x\sim p_{\text{gen}}} \log(1 - D(x))$$
$$= \mathbb{E}_{x\sim p_{\text{data}}} \log D(x) + \mathbb{E}_{z\sim p} \log(1 - D(G(z)))$$

This mean that our distribution is not computed on the images generated but is computed on our possible random noise vector that we have as input, from this distribution we generate our data.

We can represent this as a function of both generator and the discriminator. $V(G,D) = \mathbb{E}_{x\sim p_{\text{data}}} \log D(x) + \mathbb{E}_{z\sim p} \log(1 - D(G(z)))$
Substantially we wants the discriminator to correctly distinguish between real and fake samples, so the discriminator what to maximize our function $D^* = argmax_D V(G,D)$, to achieve a configuration D* which maximize over all the possible weights that the discriminator could have in order to maximize our function.

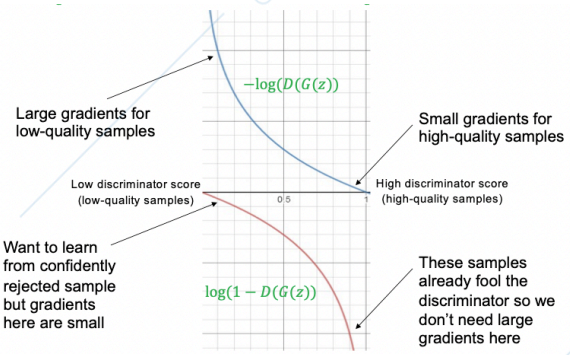
On the other side we want that the generator fool the discriminator, so generate some sample that are so good that the discriminator is not able to say if It is a fake $G^* = argmin_G V(G,D)$.

So we train generator and discriminator jointly in a minimax game, so an interaction between the two.

$\min_{w_G} \mathbb{E}_{z\sim p} \log(1 - D(G(z)))$ vs. $\max_{w_G} \mathbb{E}_{z\sim p} \log(D(G(z)))$

Assuming unlimited capacity for generator and discriminator and unlimited training data:

- The objective $min_g max_D V(G,D)$ is equivalent to Jensen-Shannon divergence between the distribution of our data $p_{data}$ and the distribution of our generated data $p_{gen}$, we can reach the global optimum (Nash equilibrium) when the two different distribution are the same $p_{data} = p_{gen}$ that is exactly what we want to obtain. So being able to generate data that have the same distribution of our training data.
- If at each step, $D$ is allowed to reach its optimum keeping the $G$ fixed, and $G$ is updated to decrease $V(G,D)$, there is the demonstration that our $p_{gen}$ actually converge to $p_{data}$.




We actually performe do gradient ascent on the generator in order to maximize the log probability of discriminator being wrong, we are performing both step of gradient ascent on both the different part that we are alternating .

We update the discriminator: Repeat for $k$ steps: Sample mini-batch of noise samples $z_1, \ldots, z_m$ and mini-batch of real samples $x_1, \ldots, x_m$. So we will take our noise sample pass them to our generator in order to create some generated examples. So we gonna have some generated sample that have the same cardinality of real samples and so we can update the parameters of our discriminator in order to disambiguate the generated and the real samples.

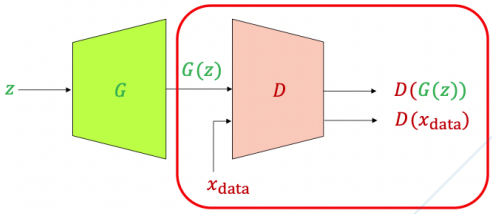
$$\frac{1}{m}\sum_m [\log D(x_m) + \log(1 - D(G(z_m)))]$$

Than after we have performe this k steps of update of discriminator we can pass to update the generator in particular we sample again a mini batch of noise sample and we can update the parameters of our generator by performing the stochastic gradient ascent of what we can achieve.

$$\frac{1}{m}\sum_m \log D(G(z_m))$$

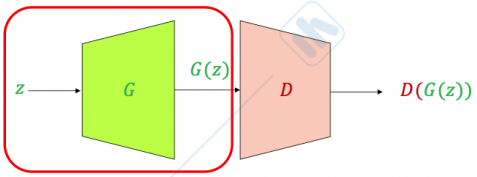
We exploit the gradient flowing in our discriminator in order to get better with our generator, repeat everything util we are satisfied of what is generated.

When we update the discriminator we want to push the $D(x_{data})$ close to 1 and $D(G(z))$ close to 0, so better to discriminate between real and generated samples.




The generator is a "black box" to the discriminator because don't see what happen in G but see just the output $G(z)$.

The we want to update the generator, so increase the value given by the discriminator on the sample generated by the generator $D(G(z))$, in order to fool as much as possible the discriminator. Requires back-propagating through the composed generator-discriminator network (i.e., the discriminator cannot be a black box).




The generator is exposed to real data only via the output of the discriminator (and its gradients).

At test time we don't need discriminator anymore so we just use it as lost function to train our generator but at test time we just want to be able to select a random noise vector given as input of the generator to generate some new sample. It could happen that the discriminator is used but usually it is used for a different task, is smoothing that I useful as starting point for a fine tuning or to extract some feature in order to perform other classification. Usually are just dropped and generator is used.

During training the first thing that we have to do is to update the discriminator such that it's output on real data will be close to one and it's output for generated data will be close to zero. In this part the generator is a black box with the respect of the discriminator since in order to update the weights that are inside the discriminator the gradients don't flow across the generator. The discriminator use the sample of the generator and the real data and then outputs its score. So in order to updated the gradients just flow throw the discriminator.

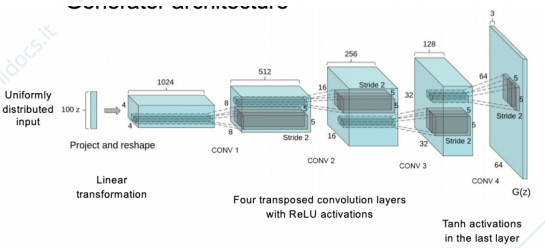
Second step, now we want to updated the generator we want to increase the score that the discriminator gives to the sample generated by the generator this means that we are keeping fixed the discriminator we want the generator become better and better generating our sample so that we could fool our discriminator. We keep fixed our discriminator and in order to increase the score that the discriminator gives to samples generated by the generator we have to flow backwards our gradients throw the discriminator and throw the generator in order to updated it, but discriminator is not updated it just permits us to propagate backwards the gradient the only part that will be updated in this second step will be the generator itself.

So first update the discriminator than create this sequence of two models where the second one is not learnable to update the weights inside our generator.

Then a test time we can remove our discriminator, since is not what we need for our task, and just keep our generator that taking any noise vector as input will generate some realistic sample.

**DCGAN (deep convolutional)**

From a random noise vector we have some convolutional layer that tent to increase the spatial resolution resolution of our output, as we go through our architecture reduce the depth information in order to enlarge the spatial resolution. We go in the opposite direction of the convolutional network.




The main ingredients here in convolutional layers is the transpose convolution.

We are learning to sample from the training distribution, for every random vector that we can generate we are able to generate one sample that is taken from the distribution of our training set. We select one random vector and we obtain one image. We select two random vector, we obtain images and now we have the possibility interpolating between this two random vector that we have generated and we can obtain a morphing effect in the visual result, meaning that we have a visual interpolation from the content that was generated using the first random vector up to the content that we have generated with the second random vector (like a transition between them). This is possible because we have learnt to associate to each random vector a new image taken from the distribution of our training set. We also have some vector arithmetic, learning to generate sample give us also the possibility to creating some relationship in this random space such that some information is preserved.

**Problem with GAN training**

Problem with stability: we have parameters that can oscillate or diverge and the generator loss does not correlate withe the sample quality, can happened that an image that have a higher loss in the generator actually are visually better than the one having a lower loss. This is because the loss that we are using does not correlate well with the sample quality.

Another problem that we have is that the behavior is very sensitive to hyper-parameter selection, maybe you have design some configuration, you are trying to optimize it and you cannot obtain good result, probably you need to experiment with different configuration of hyper parameters to find the best spot to generate something interesting.

Mode Collapse: supposed that we are generating number from minst so we have 10 different output that we want to be able to generate, it may happened that a given point the generator specialize just to generate one of the different output that you could have and all the other sample are not generated.

In order to reduce this problem we can make sure that the discriminator provides good gradients to the generator even when it can confidently reject all generator samples, in this case if we are using a non-saturating generator loss we are able to give some informative gradients in order to improve our generation. This is also achieved using smooth discriminator targets for "real" data (es. Target value of 0.9 instead of 1 when training discriminator (label smoothing). Another thing that could help to improve the result into use class labels in the discriminator if they are available.

**Wasserstein GAN**

The main thing is that we have a new type of loss such that could have more correlation with the human judgment on the quality of the generated samples. We have better gradient in the order of the loss that we are using in order to provide a more stable training. So reduce this loss give us result that are visually better.

All the improve in GANs regard its stability and the possibility of creating much better images or sample.

**How evaluate GANs**

Showing pictures of samples is not enough, we cannot even directly compare the likelihood of high dimensional samples because we font have such a way to say one is better that the other. Also some GAN approaches claim mainly to just improve stability which is hard to evaluate.

The first way that we have is to use some human studies in particular perform some Turing test, we present real and generated images and the task is too choose with are face or not.

Another idea is using *inception score,* so the idea is to use some architecture that has been train to recognize real objects and compute some statics for the images that are generated and actually real image. Extract some feature in different layers of our architecture, VGGnet trained on imagenet, generate one sample and other one that's real, then we compute the activation in different layers in our VGG architecture and we measure some statistic of the activation of different layers.