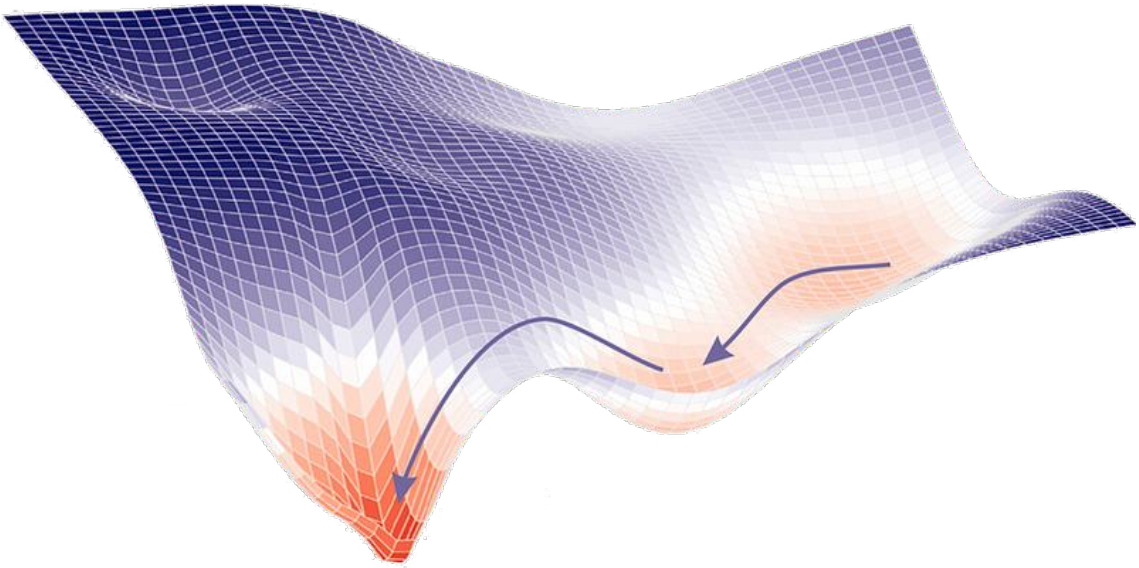
Deep Learning Notes

2024 class



By: Ludovico Comito
Contact: ludocomito@gmail.com

1 - Shallow and deep neural networks

Shallow neural networks

A shallow neural network can be seen as a function of the input $f[x, \phi] = y$ that maps the input x with parameters ϕ to an output y. Both input and output can be univariate, multivariate or a combination of this.

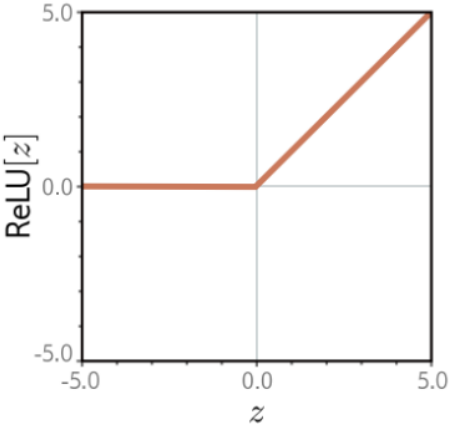
An example of this function can be:

$$y = f[x, \phi] = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11} x] + \phi_2 a[\theta_{20} + \theta_{21} x] + \phi_3 a[\theta_{30} + \theta_{31} x]$$

This function has a total of 10 weights. Notice that the weights ϕ_1, ϕ_2, ϕ_3 multiply a function a. This function is called activation function, and is tipically a non-linear function of a linear combination of the input. Why? Real world data is usually non-linear by nature, and using nonlinearities allow us to model better this kind of data.

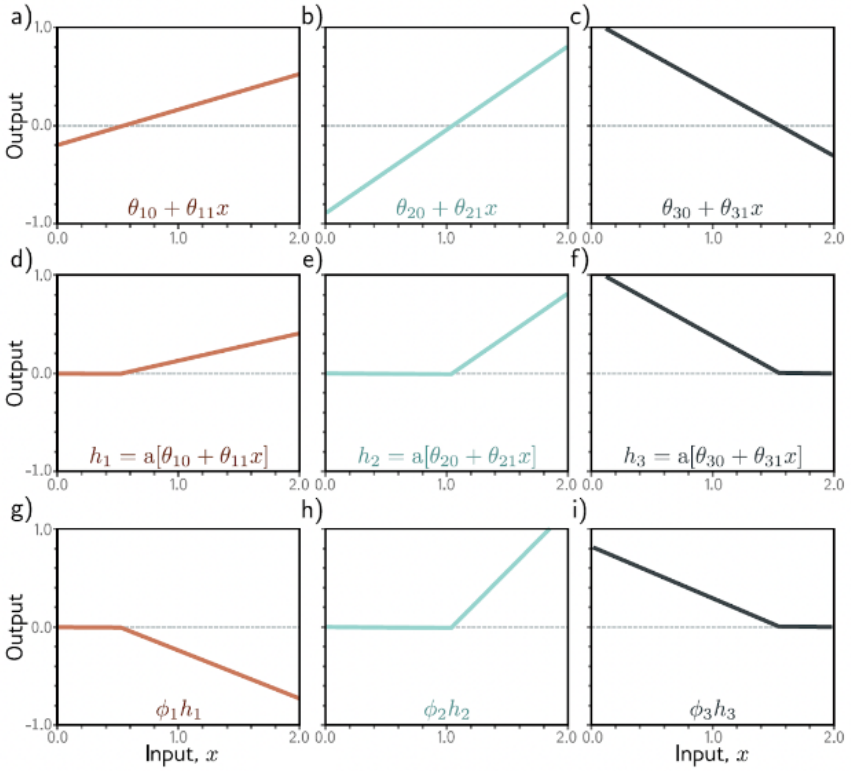
The most popular activation function is ReLU (Rectified Linear Unit), which is defined as:

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

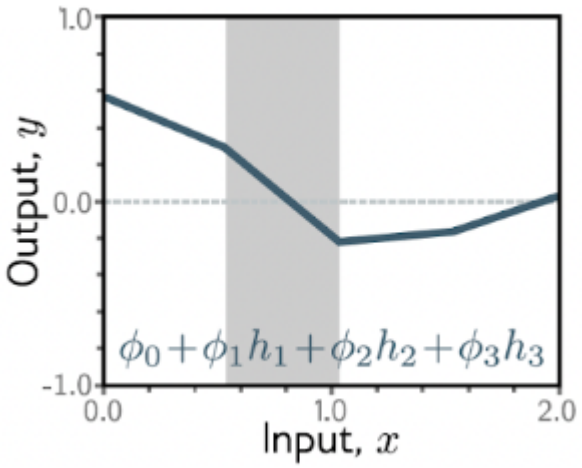


Basically it cuts out every value that is below zero, and acts as an identity function for every value higher or equal than zero.

We can see the effect of using an activation function by looking at how the function gets processed inside each hidden unit:



As you can see, using a ReLU allows us to treat a linear function as a piece-wise linear function. This allows us to create different regions of that function, and their composition in the end will give us a final function that will be split in regions:



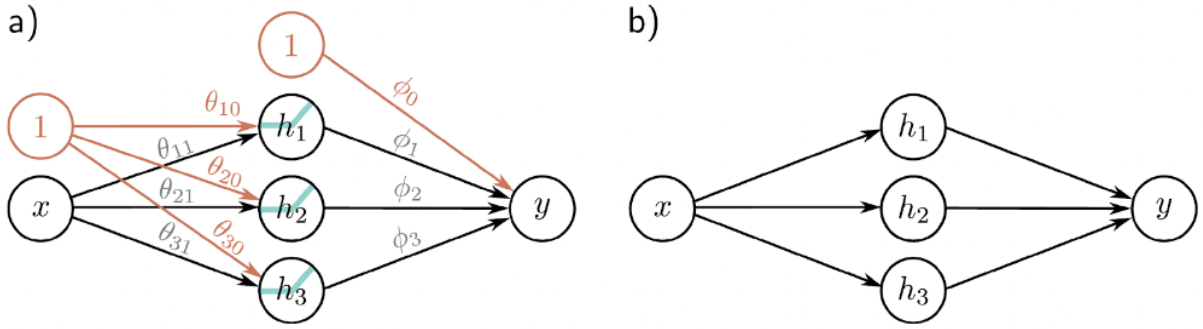
If we have n hidden units, we will obtain at most $n+1$ total regions.

The result of an activation function is called hidden unit h. We can write the previous expression as:

$$f[x,\phi] = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

What changes inside the hidden units are the weights θ, that will process the input differently for each hidden unit.

The resulting function will be a weighted composition of the hidden units plus a bias term ϕ_0:



The role of the bias here is to shift the function vertically. This allows us to represent negative values, or values that have a distribution way above zero.

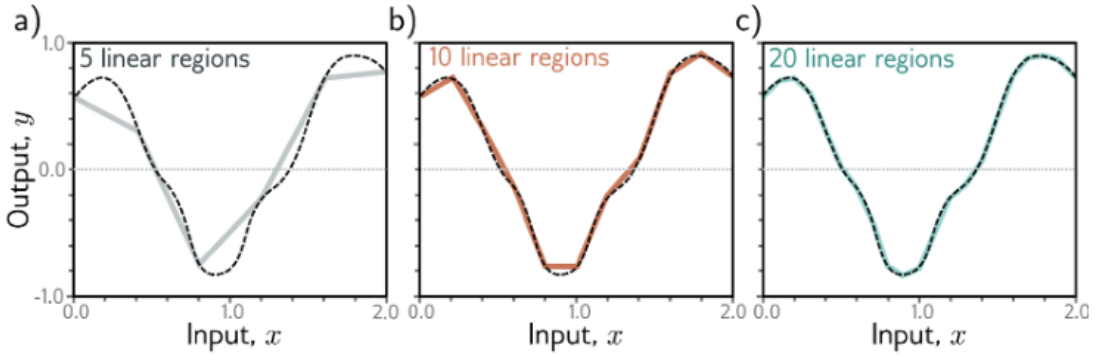
In a more compact notation, we can write the output as a weighted sum of functions of the input:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

Universal Approximation Theorem

Why do we care about the regions that we can obtain in the output function?

Well, if we zoom in enough, we can notice that even a very complex function can be approximated by a piece-wise linear function:



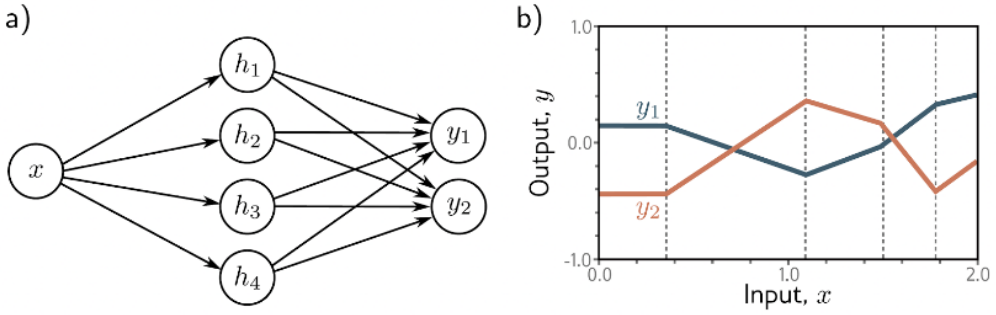
This intuition is proved and formalized by the universal approximation theorem, which tells us that:

> 💡 With enough hidden units, a shallow network can describe any continuous 1D function defined on a compact subset of the real line to arbitrary precision.

Multivariate outputs

A shallow network can have multivariate outputs (more than one output). We will see better that these outputs are not independents, since they share the same weights that get updated accordingly.

An example of shallow neural network with multivariate output is this:

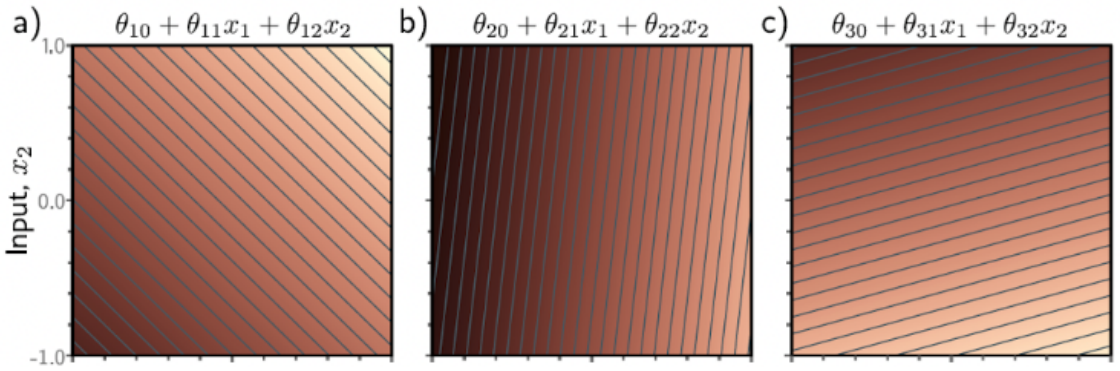


$$y_1 = \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4$$

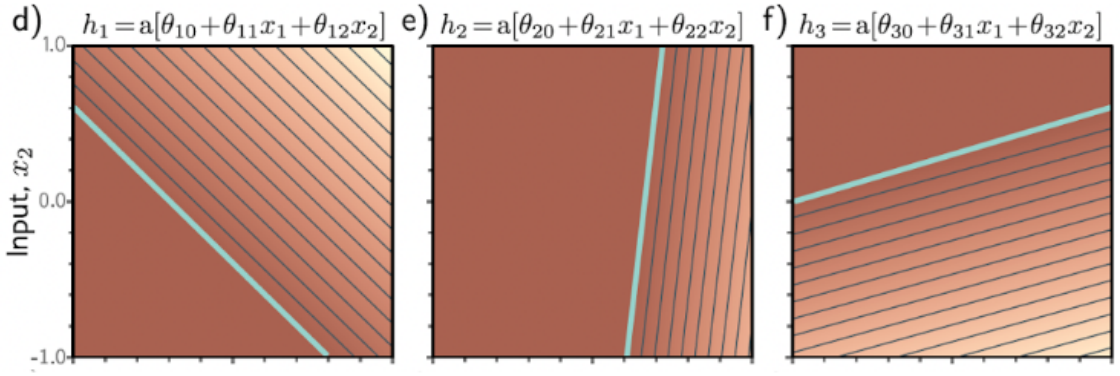
$$y_2 = \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4$$

Multivariate inputs

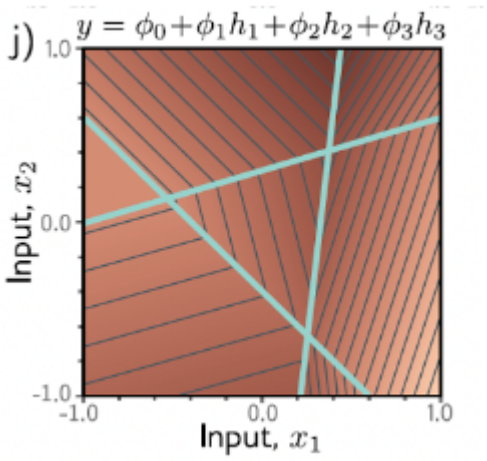
Moreover, a shallow nn can have more that one input. For example if we have two inputs x_1, x_2 the input of each hidden unit will be a linear function of the two inputs, which corresponds to an oriented plane:



The ReLU function will then clip the regions of these planes that are below zero:



The final result of the weighted sum will be a composition of planes that is split in regions:

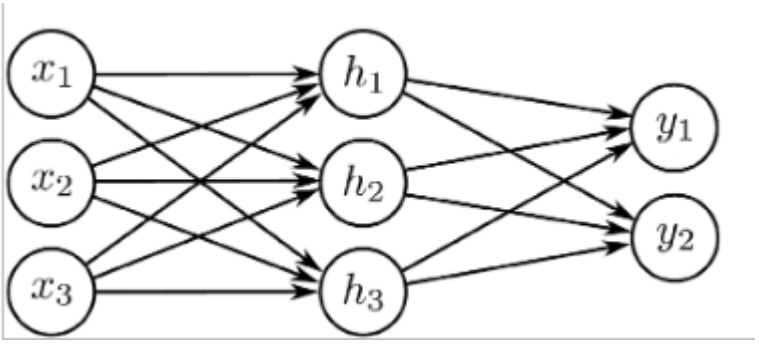


General case of shallow networks

In the general case, we can have multivariate inputs and outputs. We will say that f is a function mapping an input $x \in \mathbb{R}^{D_i}$ to an output $y \in \mathbb{R}^{D_o}$. The dimensionality of the hidden units will be $h \in \mathbb{R}^D$:

$$h_d = a[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i]$$

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d$$



Activation functions family

Besides ReLU, we have a family of different activation function that have developed in history. Some examples are:

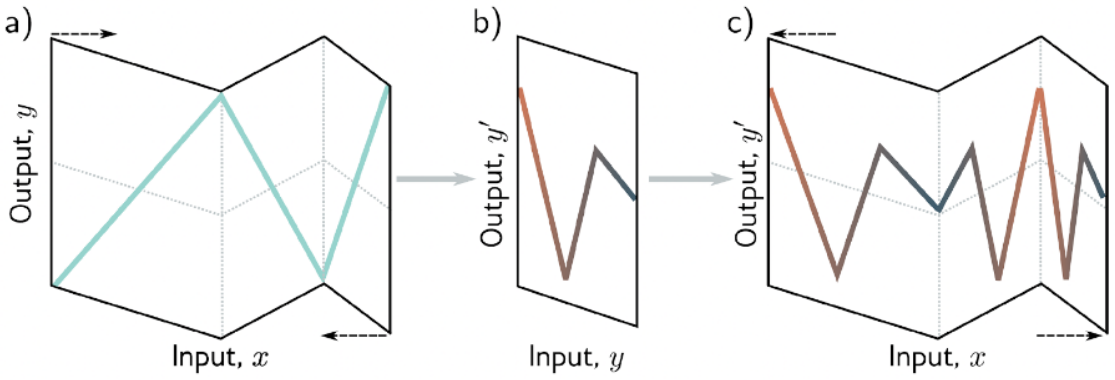
- Sigmoid
- Tanh

- LReLU → instead of clipping totally values below zero, just dampen them by a certain factor.
- PReLU → same concept as LReLU, but the slope of dampening before 0 is learned.

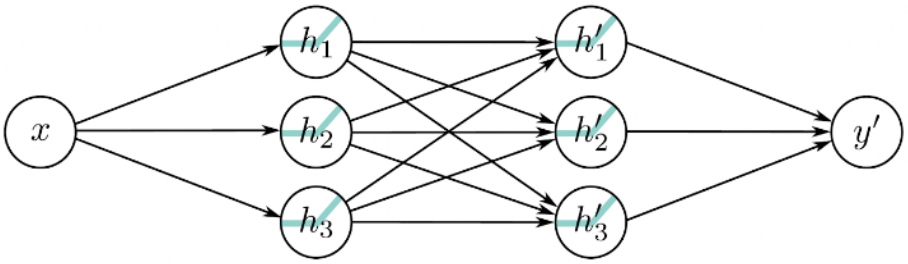
Deep Neural Networks

We have seen from the Universal Approximation Theorem that one hidden layer is enough to represent functions with arbitrary precision. The catch here is that the number of hidden units required for a single layer may grow up to infinity.

The power of deep neural networks is to stack different hidden layers one after the other. This is because when adding another hidden layer, each regions of the function obtained by the layer before is subsequently split in two. This gives us a huge advantage in representational power over a single shallow neural network, even when we use the same total number of hidden units.



When stacking two or more layers we obtain a Multi-Layer Perceptron (MLP):



To alleviate notation, we can express this function in matrix form:

$$h = a[\theta_0 + \theta x] \quad h' = a[\psi_0 + \Psi h] \quad y = \phi_0 + \phi h'$$

Deep neural networks have two fundamental hyperparameters:

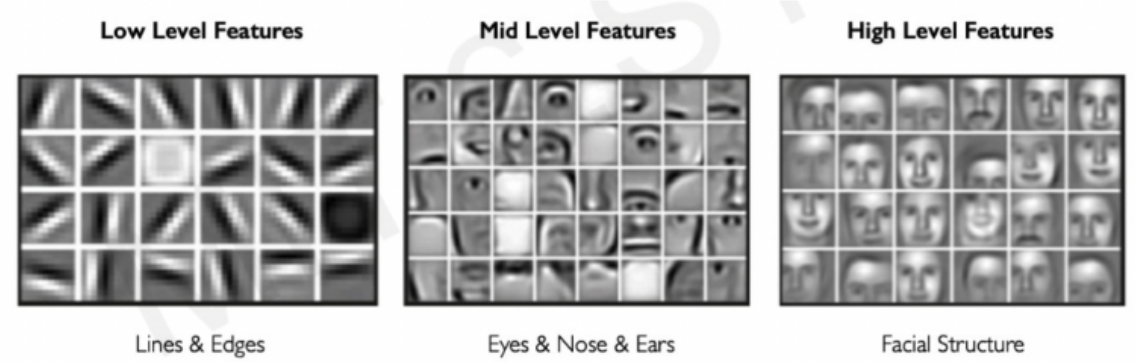
- Width → the number of hidden units for each hidden layer.
- Number of hidden layers.

We will see after that our goal will be to design neural network that exploit the intrinsic nature of our data to reduce as much as possible the number of parameters needed to reach good performances.

Why Deep Learning?

We have seen that one of the great advantages of using neural newtorks is that they can be considered as automatic feature extractors. The advantage of deep learning is that when stacking more layers, our network can learn a hierarchical representation of features, going from a low level representation in the first layer, to an abstract higher representation towards the final layers.

The following image shows this principle with images:



2 - Model Training

Loss Functions

When training, we have a dataset $\{x_i, y_i\}$ made of inputs and their corresponding outputs. How can we measure the discrepancy between our model's prediction and the actual label?
We use a loss function
$L[\phi]$, which can be considered a function purely dependant on the model's parameters (this is because our dataset is assumed to always be fixed).

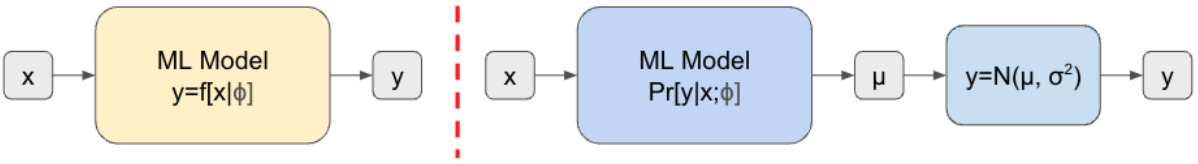
An interesting point of view is to consider the loss as a function that tells us how good our model is with the current version of parameters.

Maximum Likelihood

In general, when we measure a datapoint, we always assume a level of uncertainty behind that measurement, that is due to the fact that we can have some errors during the measurement.

This means that we can associate a probability of measuring a certain datapoint, which is centered around the measured value. So it makes sense to say that we want our prediction function to resemble that distribution. But at the moment we have a problem: our neural network's function $y = f[x|\phi]$ maps each value in input to a single value in output, not a probability $Pr[y|x;\phi]$.

The solution is to have our model output the parameters of that probability:



In this case we have the mean of a normal distribution.

Therefore, now we want to find the maximum likelihood for this probability:

$\hat{\phi} = \text{argmax}_{\phi}[\prod_{i=1}^{I} Pr(y_i|x_i)] = \text{argmax}_{\phi}[\prod_{i=1} Pr(y_i|\phi_i)] = \text{argmax}_{\phi}[\prod_{i=1} Pr(y_i|f[x_i, \phi_i])]$

The previous equation holds under the assumption that observation ad Independent and Identically Distributed (IID).

Since the logarithm of a product is equal to the sum of the logarithms of each elements, we can use this property to simplify the previous expression as a summation:

$$\hat{\phi} = \text{argmax}_{\phi}[\sum_{i=1}^{I} log(Pr(y_i|f[x_i; \phi])$$

The previous expression can be formulated as a minimization problem by adding a minus in front of the summation:

$$\hat{\phi} = \text{argmin}_{\phi}[-\sum_{i=1}^{I} log(Pr(y_i|f[x_i; \phi))] = \text{argmin}_{\phi}[L[\phi]]$$

This is also known as minimizing the negative log likelihood.

When performing inference, we usually return the maximum of that distribution:

$$\hat{y} = argmax_y(Pr(y|f[x; \phi])$$

Recipe to compute a loss function

In the previous chapter we have seen that there is a correspondence between the negative log likelihood and the loss function. As you can imagine, choosing the distribution that suits the task best will be crucial to define our loss function.

The generic steps for our loss are:

1. Choose the probability $Pr(y|\theta)$ that best suits the task.

2. Set the machine learning model $f[x;\theta]$ to output the parameters of this distribution:

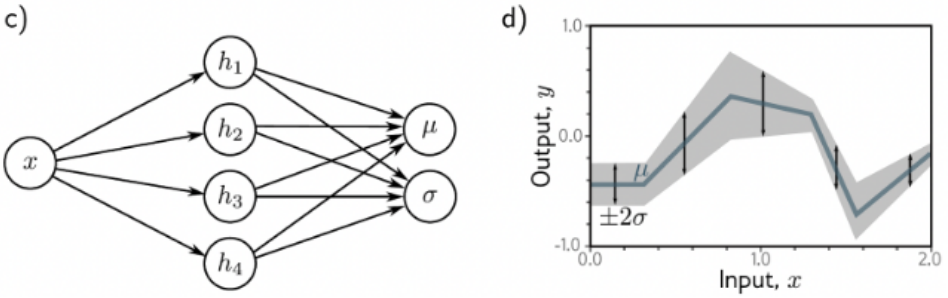
   $$Pr(y|\theta) = Pr(y|f[x,\phi]).$$

3. We use the negative log-likelihood as the loss function over the training dataset pairs $\{x_i, y_i\}$:

   $$\hat{\phi} = argmin_{\phi}[L[\phi]] = argmin_{\phi}[-\sum_{i=1}^{I} log(Pr(y_i, f[x_i,\phi]))]$$

4. To perform inference, return either the full distribution, or just its argmax.

Let's make an observation. Until now, we utilized our function f to just output the mean μ of the distribution. A more interesting approach is to make it output also the variance σ^2 for the distribution. In this way, will will also have a measure of uncertainty for our predictions:



This will allow us to properly work with generic probability distributions.

Distributions for Loss functions of different types:

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots, K]$	Plackett-Luce	ranking

Gradient descent algorithms

We have learned how create a Loss function. How do we use it to train our model?

In almost all cases, we use an algorithm called Backpropagation, which is used to update our model's weight based on the result of the loss function.

A fundamental notion to know is the gradient of a function:

> 💡 The gradient of a function indicates the direction of maximum increase of that function in a given point.

Since we are treating a minimization problem, we are interested in the negative direction of that gradient, that will point to the minimum of that function. What do we do in practice?

Given a set of function parameters: $\phi = [\phi_1, \phi_2, ..., \phi_N]$, we compute the derivative of our loss with respect to those parameters:

$$\frac{\partial \mathcal{L}}{\partial \phi_1} \quad \frac{\partial \mathcal{L}}{\partial \phi_2} \quad \frac{\partial \mathcal{L}}{\partial \phi_N}$$

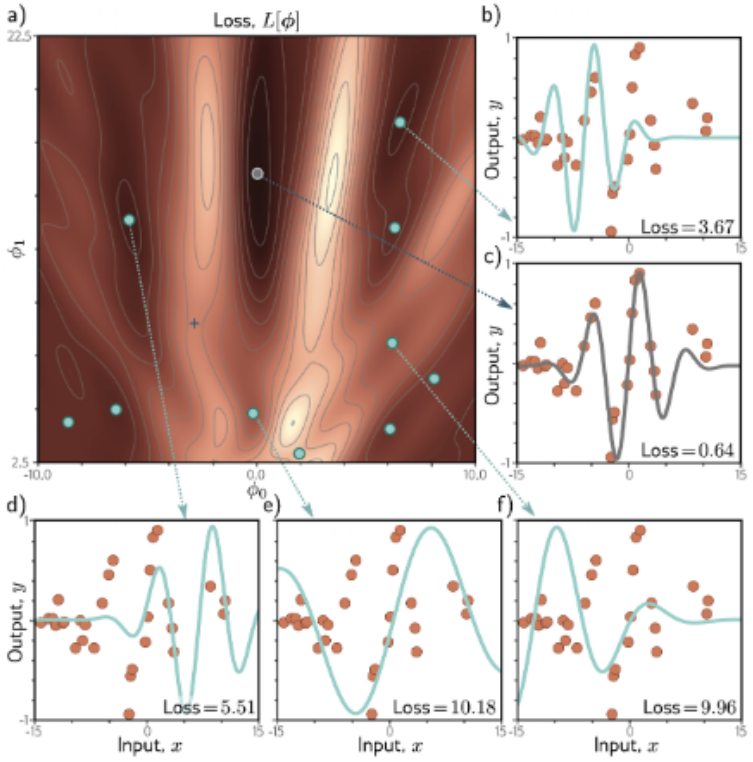
$$\frac{\partial L}{\partial \phi} = [\frac{\partial L}{\partial \phi_1} \ \ldots \ \frac{\partial L}{\partial \phi_N}]^T$$

In order to update the weights, we subtract to our weights the value of the gradient, rescaled by a factor α which is called learning rate:

$$\phi \leftarrow \phi - \alpha \frac{\partial L}{\partial \phi}$$

It's very important to use the learning rate to set the magnitude of the update. In fact, when the update is too large, we might risk to miss the actual minimum or to cause an oscillatory behavior around it that won't allow us to reach it.

When loss functions have a very complicated landscape (not convex), the basic behavior of gradient descent may still cause us the problems that we described before. In order to cope with these problems, there are a number of approaches that have been developed.

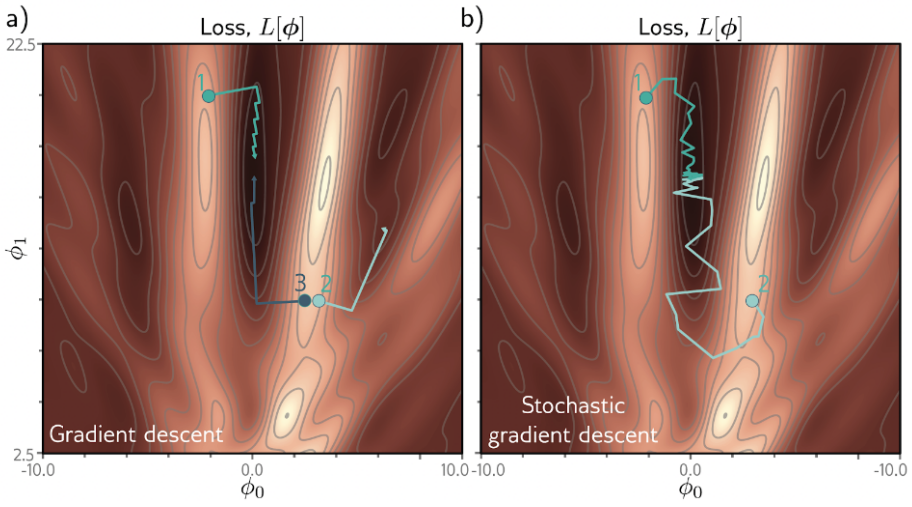


Example of complicated landscape (Gabor Model).

Stochastic Gradient Descent (SGD)

The main idea of SGD is to compute an update step with respect to a single element iteratively instead of using the whole dataset. Although we know that in

this way we are never moving towards the exact direction of the gradient, the positive aspect is that we have a more dynamic behavior. This in theory should allow us to explore more the loss landscape, and allow us to discover other minimums that we might not have explored.



As we can appreciate from the picture above, the path towards the minimum is dynamic, but also very noisy. An approach that tries to smoothen this path is SGD with minibatching. We can consider it as a middle point between using the whole dataset and one element per time. In fact, with minibatching we consider a subset of the training set (batch) for each iteration

Another important notion is to always shuffle our training data before. This allows us to avoid any possible simmetry that might cause our data to not respect the IID assumption.

Pros of SGD:

- Although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration. Hence, the updates tend to be sensible even if they are not optimal.
- Given that it draws training examples without replacement and iterates through the dataset, the training examples all still contribute equally.
- It is less computationally expensive to compute the gradient from just a subset of the training data.
- It can (in principle) escape local minima.
- It reduces the chances of getting stuck near saddle points; it is likely that at least some of the possible batches will have a significant gradient at any

point on the loss function.

Scheduled Learning Rate

In practice, SGD is often used with a learning rate scheduler. As we have seen, having a learning rate that is too large may cause our model to escape a minimum of have an oscillatory behavior around it. An intelligent approach is to modify the learning rate over the epochs. We start with a larger learning rate, that can make our model explore the landscape of peaks and valleys. As we go on with the epochs, we start decreasing the lr value, since we expect to have reached a minimum, that we will able to explore avoiding escaping from it.

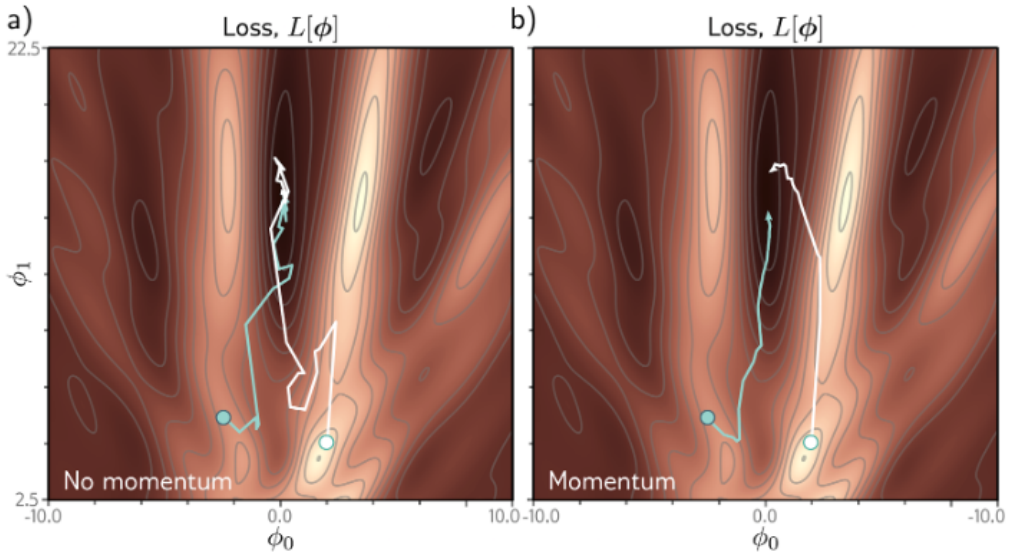
SGD with momentum

The idea here is to add a momentum term that, like in physics, makes our model trajectory at time t dependent from the previous steps. This results in a smoother behavior that avoids oscillations.

In practice, we compute the momentum m, rescaled by a factor $\beta \in [0, 1]$ as:

$$m_{t+1} \leftarrow \beta m_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\phi_t]}{\partial \phi}$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot m_{t+1}$$



From the picture above we can notice that we obtain a smoother behavior, but we're still subject to abruptive changes in direction. To cope with this, we can

use Nesterov momentum, where we compute the gradient of the current points discounted by the previous ones:

$$m_{t+1} \leftarrow \beta m_t + (1-\beta)\sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\phi_t - \alpha \cdot m_t]}{\partial \phi}$$

This discount factor has the effect of smoothing the trajectory even more.

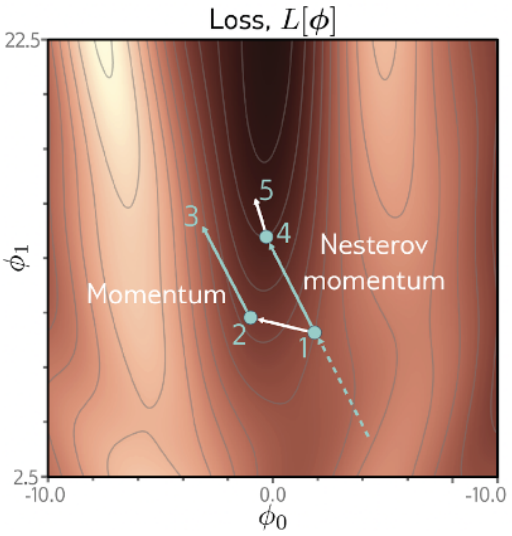


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

SGD with Adam

The Adam approach adjusts the learning rate adaptively for each parameter. This allows us to perform a kind of normalization that avoids that there are terms with large updates and others with smaller ones. It is like normalizing the value of the gradients:

$$m_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}$$

$$v_{t+1} \leftarrow \left(\frac{\partial L[\phi_t]}{\partial \phi}\right)^2$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon}$$

Adams further extends this notion by adding a momentum term.

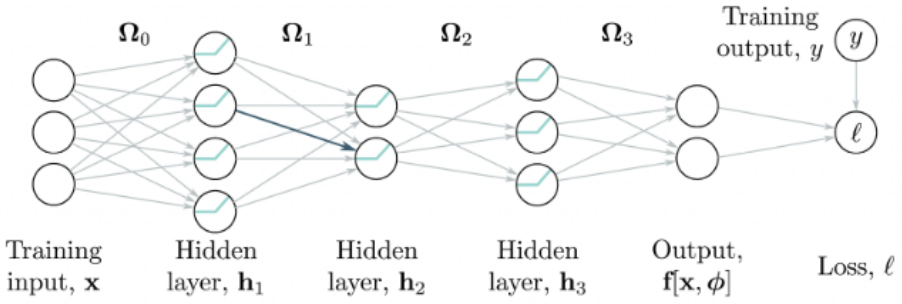
Backpropagation

Up until now we have described how the core principle of SGD is the gradient computation. This is easy when we have a single hidden layer, but with deep neural networks the complexity of this operation increases by a lot.

The algorithm we use to compute the gradient of the loss function with respect to the model's parameters is called backpropagation. This works by considering our network's function as a composition of functions. This algorithms works in two steps:

Forward pass

We start from the input x and pass it forwards to each layer, computing the activations at each hidden layer until we get to the output y.



Backward pass

Once the loss is computed, we compute go backwards (from the last layer to the first). At each step we compute the derivative of the previous function with respect to the next one. In this way, we simplify the computation by exploiting the fact that derivatives can be computed sequentially using the chain rule:

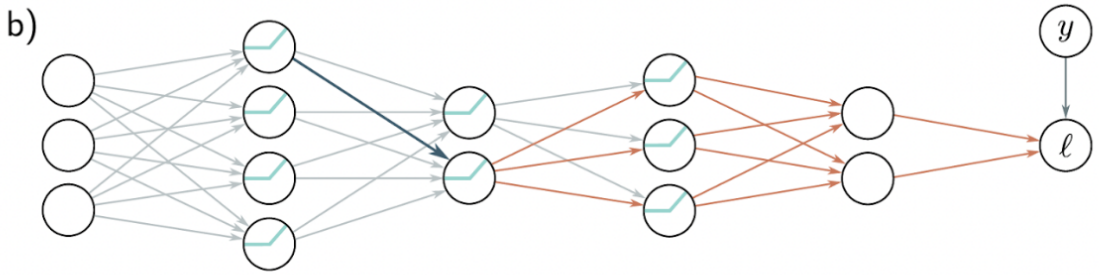


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

We continue in this way, computing the derivatives of the output with respect to these intermediate quantities (figure 7.4):

$$\begin{aligned}
\frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right).
\end{aligned} \tag{7.12}$$

Vanishing and exploding gradients

When performing backpropagation, we are computing the derivative of the loss function with respect to all the parameters of our model. The way we do it is sequential, by exploiting the chain rule we start from the last layer all the way to the first one.

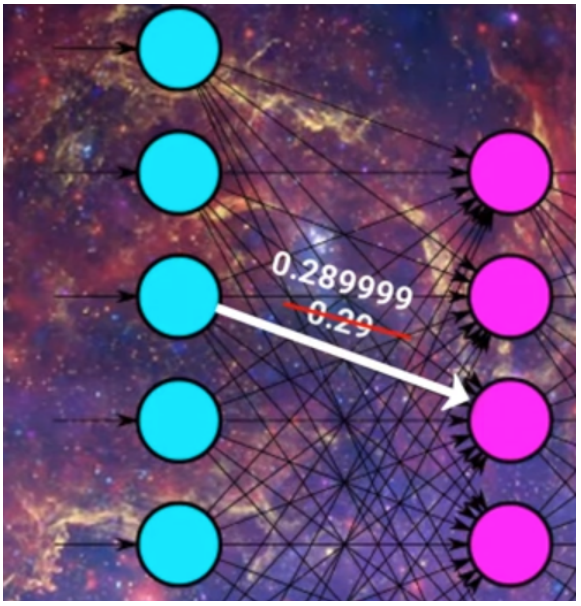
During this process two kinds of problem might arise due to the distribution of the values of our weights in the network:

Let's say that all the parameters in our model will start with a value smaller than one. The result propagating through our network will be increasingly smaller and smaller, as we are multiplying values below zero. As a result, when performing the backward pass, we will have the same behavior where gradients get smaller and smaller towards the earlier layers of our network. This

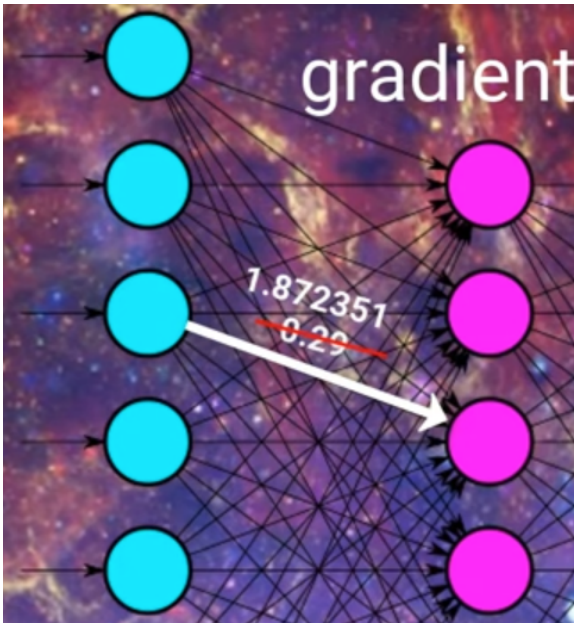
implies that we will start having updates that are negligible, resulting in the fact that the process of convergence becomes slower or unreachable. This phenomena is called *exploding gradients.*

In the inverse case, we have all the parameters that start with very high values. When performing the backward step in this case we will have that values of the gradient will be increasingly higher, resulting in updates that are too great. In this case we mights never converge due to the instability of the gradients. This phenomena is called exploding gradients.

We can deal with both problems by choosing a proper initialization strategy of our network. In this way we will make sure that the distribution of our parameter's values will be balanced.



Vanishing gradient example



Exploding gradient example

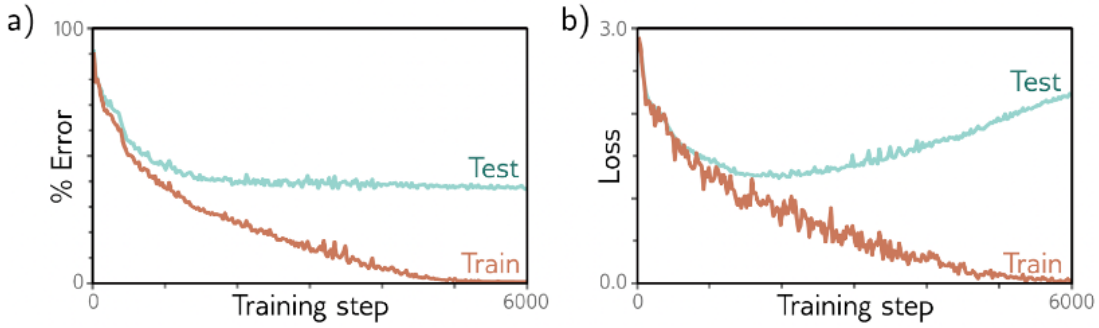
3 - Measuring performance

Dataset splits

Once we have our training figured out, we have to establish metrics to measure the performance of our model. These metrics will have to be quantified against splits of our dataset. Traditionally, we split our dataset in train, validation and test. During the training phase, we fit our model on the training data, and pick the best version of it according to the best performance on the validation set.

Why not using the set for validation and test? Because otherwise we would be fitting on our test data, but we want it to be totally unrelated to the training phase.

Learning curve



From the image above we can appreciate an interesting phenomena. The left graph shows the percentage of error over the epochs, while the right one the loss over the epochs.

Why is it that at a certain point the test error remains stable while test loss starts diverging?

As the neural network keeps fitting on the training data, it will start making predictions with more and more confidence. What happens is that from a

certain point we will still make the same number of errors, but more confidently. This increases the discrepancy between predictions and the desired output, causing an increase in the value of the loss.

Sources of error

When training a model there are various source of error that can arise. Two important ones are:

- **Noise** → real word data is noisy. This can be due to a number of reasons, starting from the intrinsic nature of the data generation process to mislabelling of data, noisy measurements etc. We have to take into account that our model will have to be as robust as possible to this kind of noise.
- **Bias** → this depends on the assumptions that we make while designing it. Wrong assumptions, like the number of parameters, variance in initialization etc. can hurt the predictions.

Mathematically, we can express the expected value of the loss function as being made up by three components:

$$E_{\mathcal{D}}[E_y[L[x]]] = \text{variance} + \text{bias} + \text{noise}$$

There is always a tradeoff between variance and bias: decreasing the variance will increase the bias and vice-versa. Noise instead is not dependent on the training data, but it's intrinsic to the data generation process.

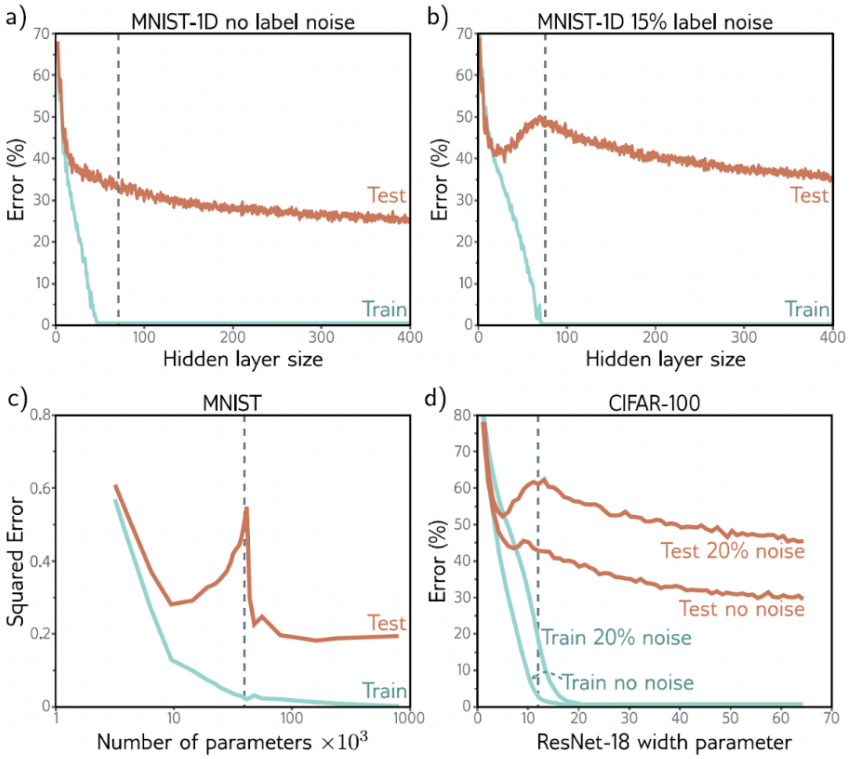
Reducing variance error

A way of reducing variance error without hurting the bias is to increase the size of our training dataset. This allows to average out the intrinsic noise and ensure that the input space is well sampled.

Reducing bias error

The bias of our model is due mainly to its inability to represent the true underlying function. We know that the representational power of a model is dependent on its number of parameters. By increasing our model's capacity, we can mitigate the bias by increasing the representation power of the model.

Double descent

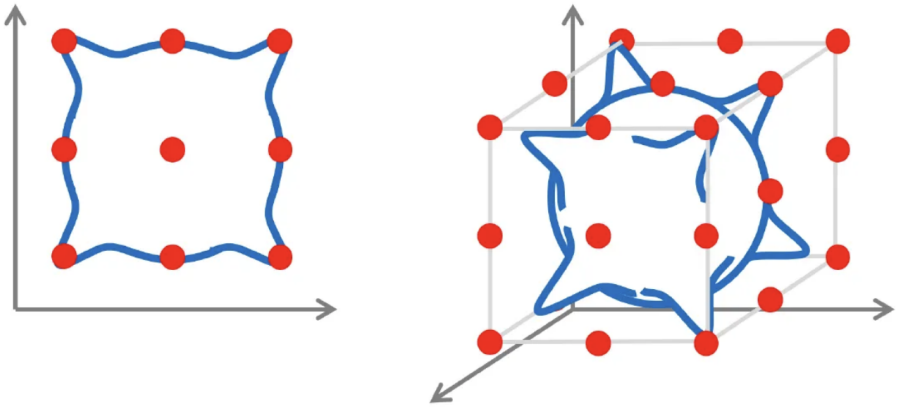


The phenomena of double descent happens when increasing the parameters of a model causes a temporary drop in the performance of our model, until a certain point where it starts improving again.

Curse of dimensionality

The curse of dimensionality problem can be formulated in this way:

As the number of dimensions of our input increase, the representation of datapoints will become increasingly sparse. Let's imagine tracking a point in the 2D space and one in 3D space. While in 2D the point is confined within a certain plane, in 3D we have many more possible position for it. And this is just by increasing one dimension.



Moving from a 2D space to a 3D representation

As the number of dimension increases, theoretically we would need an increasing amount of training data in order to face with this complexity. Yet, with neural networks we are often able to obtain great results with a limited amount of training data with respect to the input's dimensions.

This is because often real world phenomena behave with a certain pattern that can be described in a lower dimensional manifold. The great power of Neural Networks is precisely to work as automatic feature extractors, therefore being able to recognize these low dimensional patterns.

Regularization

As we have already seen, a common problem with machine learning models is overfitting, which happens when models focus more on memorizing training data rather than improving their generalization capabilities. Regularization techniques are used to deal with the problem of overfitting. The most popular ones are:

- **L1/L2 regularization →** we add a term to the loss function that penalizes the weights whose norm is too large. The difference lies within the method that we use to compute this norm: in L2 regularization we use a ball-shaped contour, while in L1 a diamond-shaped contour.
- **Early stopping →** we measure both validation and train loss. As soon as losses start diverging (val loss increases, and train loss decreases) we stop training, since we are approaching the overfitting region. Of course this technique requires model checkpointing.
- **Data augmentation →** we apply transformation to the original train dataset to increase the number of datapoints with the goal of making the model

more robust to noise and learn from more sources.

- **Dropout →** for each batch, we set weights of our model to 0 with a certain probability. This is like modifying the architecture of our model at different steps. Eventually this is like creating a mixture of models within a single model, where each model is specialized to a certain batch. Dropout doesn't work well when data is scarce.

4 - CNNs and ResNets

Introduction

We have seen that MLPs are theoretically universal function approximators. However this kinds of networks are not suitable for high dimensional data such as images, as we would need a huge amount of parameters to process these inputs.

Recalling the notion of inductive bias, we can exploit the intrinsic nature of our data in order to design models architectures that reach good performances using as few parameters as we can.

Invariance

How can we enforce the inductive bias in images for example? Well, we can notice that images are subject to different kinds of invariances:

- **Self-similarity** → if we take a patch from the image, pixels within this patch will tend to be similar one to the other. First intuition: we could share the same weights for multiple pixels.
- **Translational invariance** → we expect the representation of the same subject to be equal even if we translate it to a different part of the image.

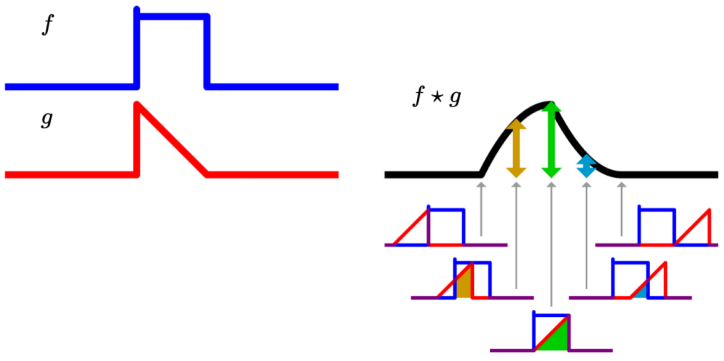
- **Deformation invariance** → even if we modify a bit the same subject, we want to be able to have a similar representation of the original one.
- **Hierarchy and compositionality** → tipically composing contiguous patches from the same image will tend to form a subject. Ideally we want an architecture that is able to combine patches in a hierarchical fasion.

Convolutional Neural Networks

Convolution in maths

In maths the convolution operator is used to extract features from a signal. It works by sliding a kernel function (think at this as a feature extractor) over the patches of the original signal.

$$\underbrace{(f \star g)(x)}_{\text{feature map}} = \int_{-\pi}^{\pi} f(t)\, \underbrace{g(x-t)}_{\text{kernel}}\, dt$$



A fundamental property of the convolution operator is that it is shift invariant. this means that applying the convolution before or after a shift will yield the same result.

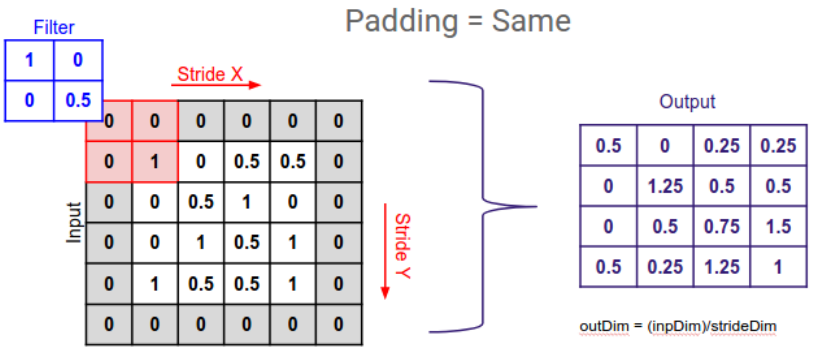
Ideally, we want to do the same thing with a Neural Network. In particular with images we want a discrete convolution, as pixels are a discrete quantity.

CNNs

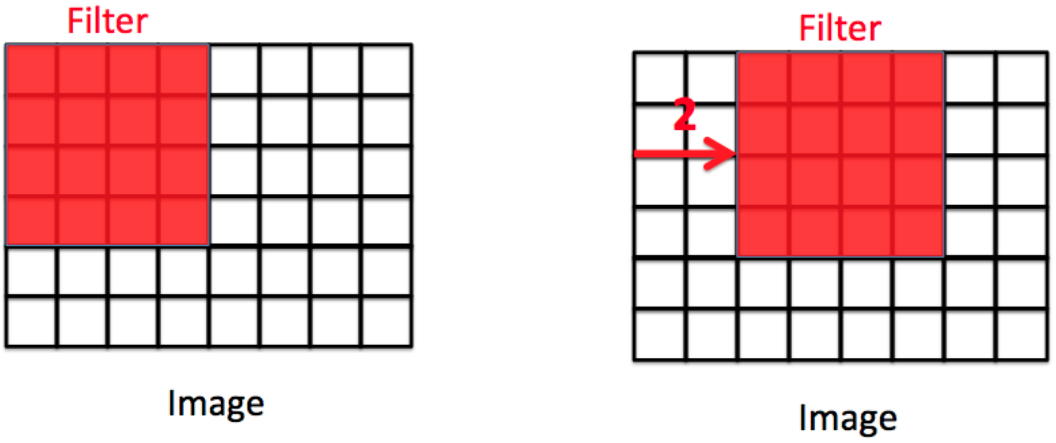
Convolutional Neural Networks use the convolution operation over images. The kernel function operates over patches of the image, and its values are learned as parameters of the network. This allows us the learn the feature extractor and using a limited number of parameters to represent the whole image.

Padding and stride

Padding consists in adding pixels around the original input image. This is used to modify the shape of the convolution's output. In particular if we use no padding, the shape of the input will be reduced. Using valid padding consist in contouring the image with one pixel. In this way the shape of the output will be the same as the input.



Strides consists in defining by how much the kernel moves within the image. A stride of two means that the kernel will slide by two pixels at a time instead of one.



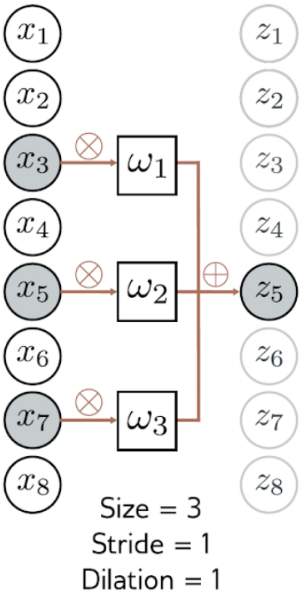
Convolutional layer

A typical convolutional layer works by applying a convolution of the input, and then passing this convolution to an activation function. The following is an example of a convolution layer with a kernel of size three

$$h_i = a[\beta + w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1}] = a[\beta + \sum_{j=1}^{3} w_j x_{i+j-2}]$$

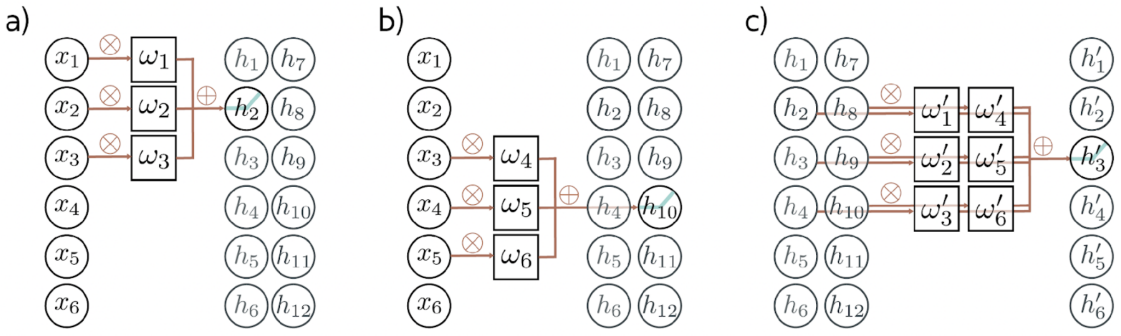
Dilation

The concept of dilation consists in skipping contiguous elements and performing the convolution on pixels with a certain distance. This allows to cover an higher portion of parameters with the same kernel.



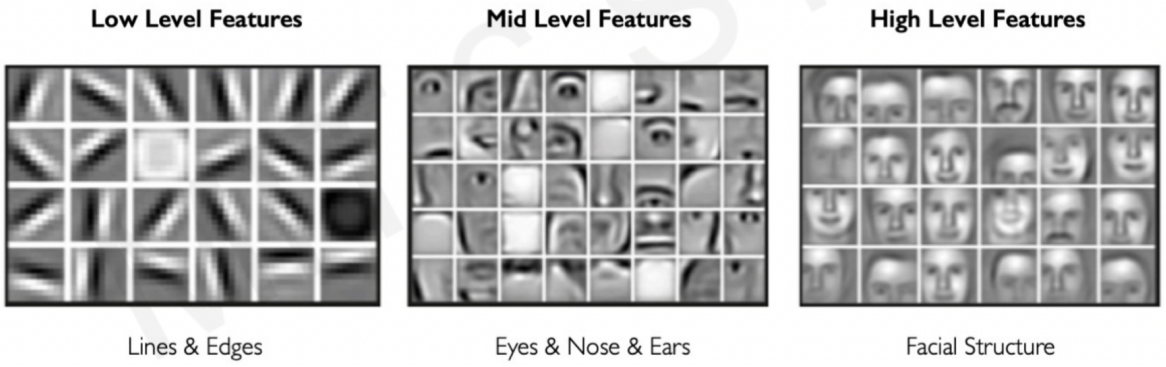
Channels

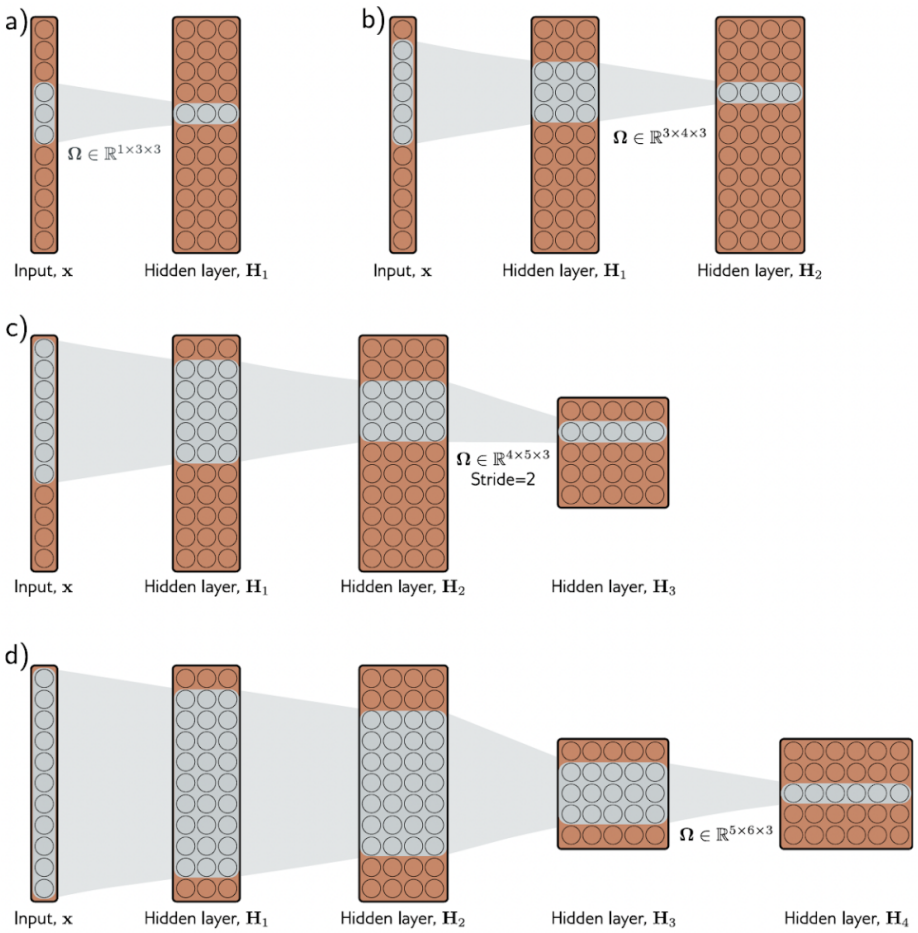
Usually we apply multiple kernels to the same input in order to extract multiple features from it. Each result of the convolution is called channel or feature map. Using multiple channels is particularly useful when working with multi-dimensional inputs.



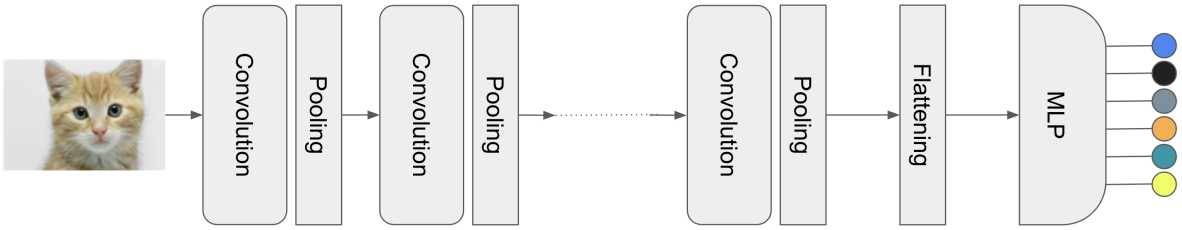
Receptive field

As we move on with the layers of a CNN, we are shrinking more and more the dimension of the input. This means that each time, one single element of the output will represent a bigger portion of the input. This means that we are exploiting the hierarchical structure of image by creating a hierarchical representation within the layers.





Typical architecture of a CNN

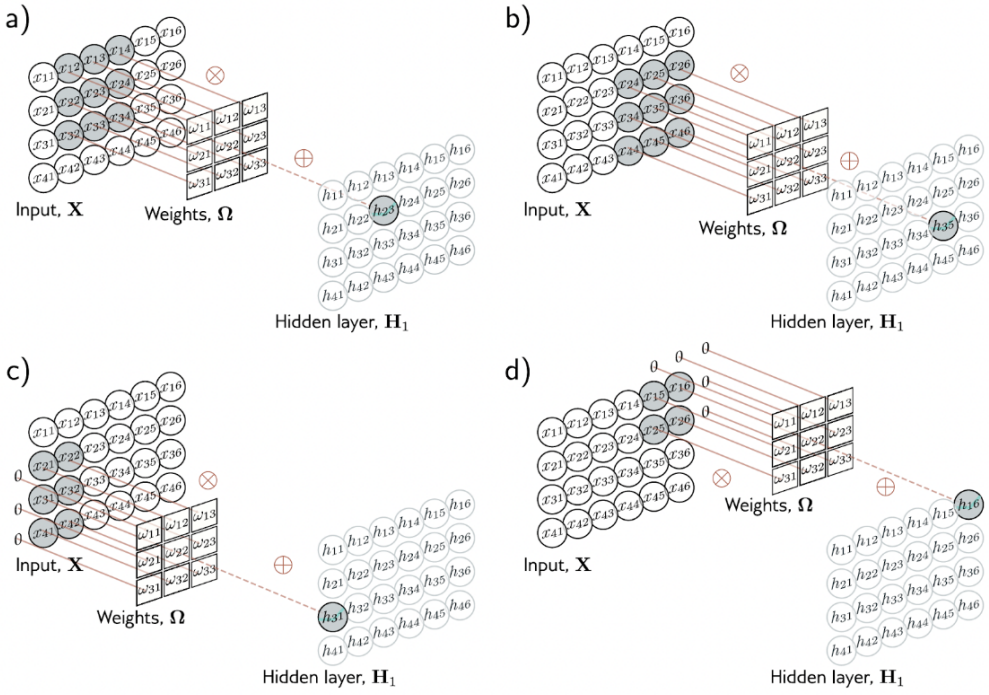


CNNs and 2D inputs

When we deal with 2D inputs, such as images, we will have to design a kernel as a 2D object. For example a 3X3 kernel applied to a 2D within a hidden unit will perform the following computation:

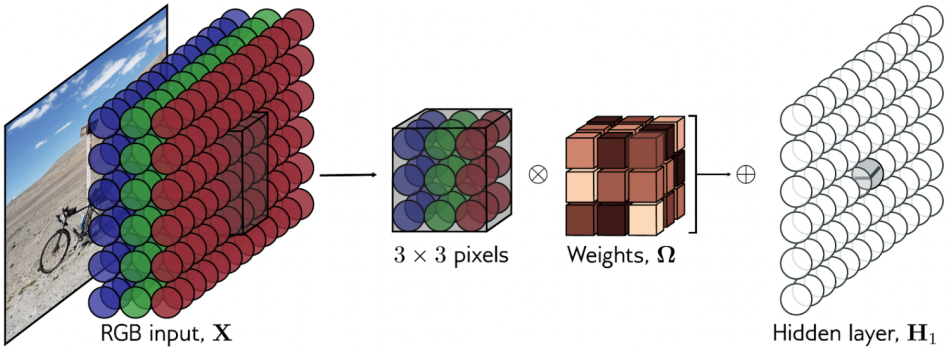
$$h_{i,j} = a[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3} w_{mn} x_{i+m-2,j+n-2}]$$

With w_{mn} being the entries of the convolutional kernel. The result will basically be the activation applied to a weighted sum.



What if we have multi-channel 2D inputs? An example of this are RGB images, which are 2D inputs with three channels. In this case we will utilize a multi-

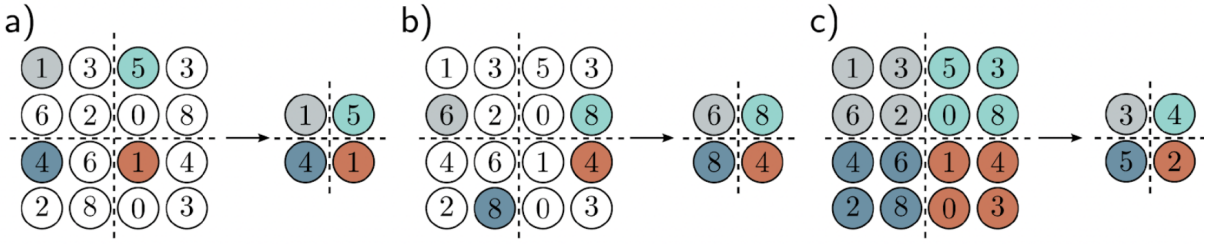
channel kernel. For example a 3×3 kernel will have: $n\_channels * witdth * height$ number of parameters.
In this case
$3 \cdot 3 \cdot 3 = 27$ parameters plius the bias term.



Downsampling

High dimensional inputs like images carry many informations, and typically a good amount of it is noise. Ideally, we want our CNN to be capable of filtering out noise as we progress with the layers. In order to achieve this, there are three main strategies that can be applied to the output of a hidden layer of a CNN:

- **Sub-sample** → divide the 2D output in quadrants and take always the element at the same position for each quadrant.
- **Max-pooling** → from each quadrant take the element with the maximum value (idea: always take the most significant element).
- **Average-pooling** → for each quadrant compute the average of the elements within.

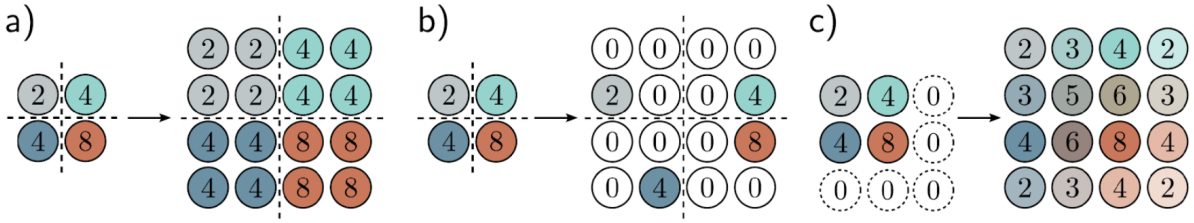


Upsampling

There are also cases in which we want to upsample the signal. An example of this are segmentation tasks, where we want to reconstruct an output of the

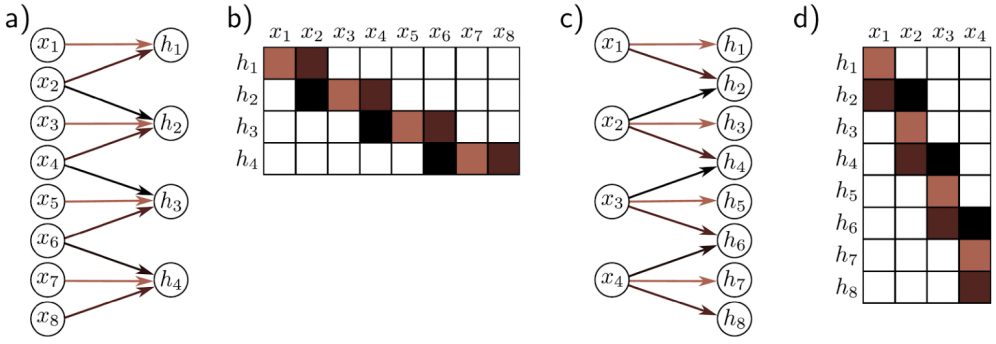
same dimension of the input, but with pixels labeled in a semantic way to represent different objects in a scene. We have three main upsampling techniques:

- **Simple** → take each element and duplicate it M times.
- **Max unpooling** → surround each element with zeros.
- **Bilinear interpolation** → add pixels with values equal to the average of the surrounding pixels.



Transposed convolution

Transposed convolution is one of the most effective upsampling techniques. In this case, each input is sent to multiple output channels, such that for one input we will obtain more than one output. Each output will be also influenced by multiple inputs.



1×1 convolutions → we can use a 1×1 kernel to change the number of channels of the input.

Applications of CNNs

Image classification

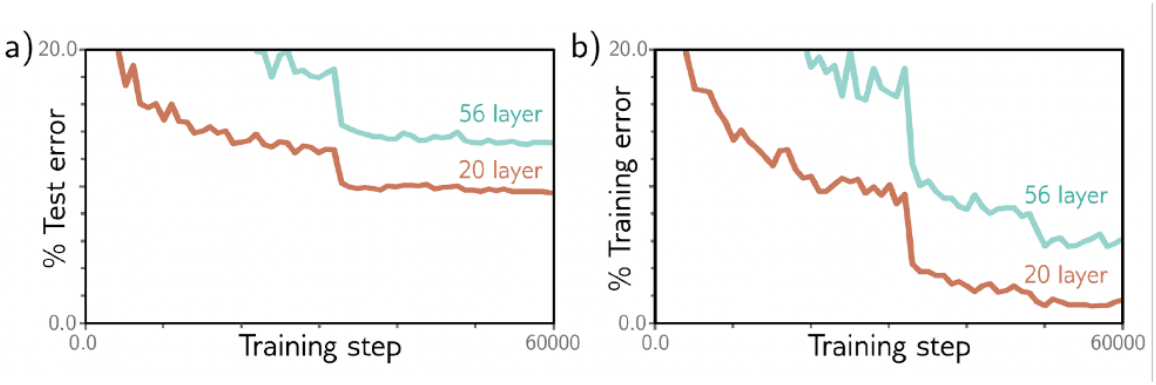
In image classification we want to assign the correct class to each input image. The most famous architectures are AlexNet (squeeze the input and than pass it to a series of FC layers) and VGG (first learn filters as outputs of the same input size, but more channels and then squeeze it).

Object detection

Semantic segmentation → label each input pixel with a color, such that the result is a semantic segmentation of the input. Typical architectures are based on downsampling the input to extract features, and than upsampling back to the original input size.

ResNets

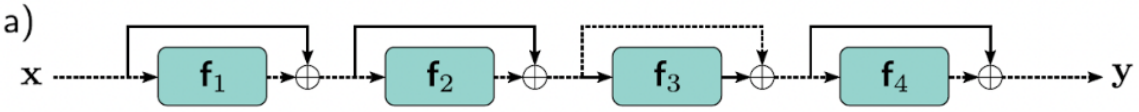
As we have seen, adding layers to our model is often beneficial. But commonly adding parameters is effective up to a certain point, after which we might experience actually a degradation in performances. One of the principle causes in these cases is the problem of vanishing gradients. As we have seen, with too many layers we might risk to have lower and lower values of the gradient during backpropagation.



ResNets were built to address this problem. The way they do it is by adding residual connections, that pass the input of the layer forward to its output, where it will get summed.

We can think about residual connections as channels that allow the gradient to flow unchanged.

$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\
\mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\
\mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\
\mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4]
\end{aligned}
$$



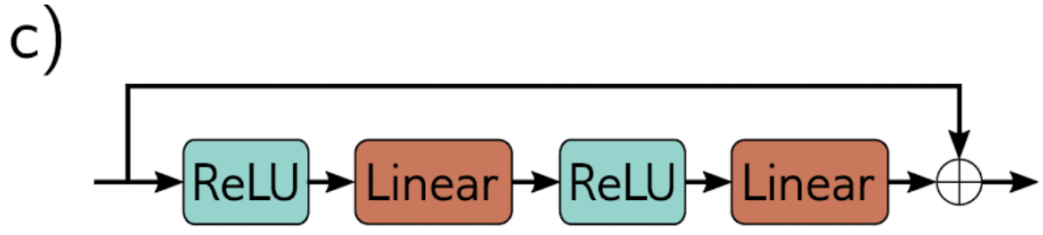
Notice how the final output will depend also on the outputs of all the previous layers:

$$
\begin{aligned}
\mathbf{y} = \mathbf{x} &+ \mathbf{f}_1[\mathbf{x}] \\
&+ \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right] \\
&+ \mathbf{f}_3\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right]\right] \\
&+ \mathbf{f}_4\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right] + \mathbf{f}_3\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right]\right]\right]
\end{aligned}
$$

The only downfall is that the resulting computational graph of ResNets will be more complicated with respect to other architectures.

Residual blocks

A single cell of a ResNet is also called residual block. Residual blocks are commonly composed of one ore more pairs of linear layers and activation. A usual setup is the following one:
ReLu→Linear→ReLu→Linear



Exploding gradients

We have seen that ResNets are able to cope with the vanishing exploding gradient problems, but they still can't solve the exploding gradients one by themselves. The solution to this problem is **batch normalization**.

Batch normalization is an adaptive reparametrization technique where given a minibatch of activations H of the layer to normalize, we normalize H by subtracting the mean of each unit μ and dividing by its standard deviation σ:

$$H' = \frac{H - \mu}{\sigma}$$

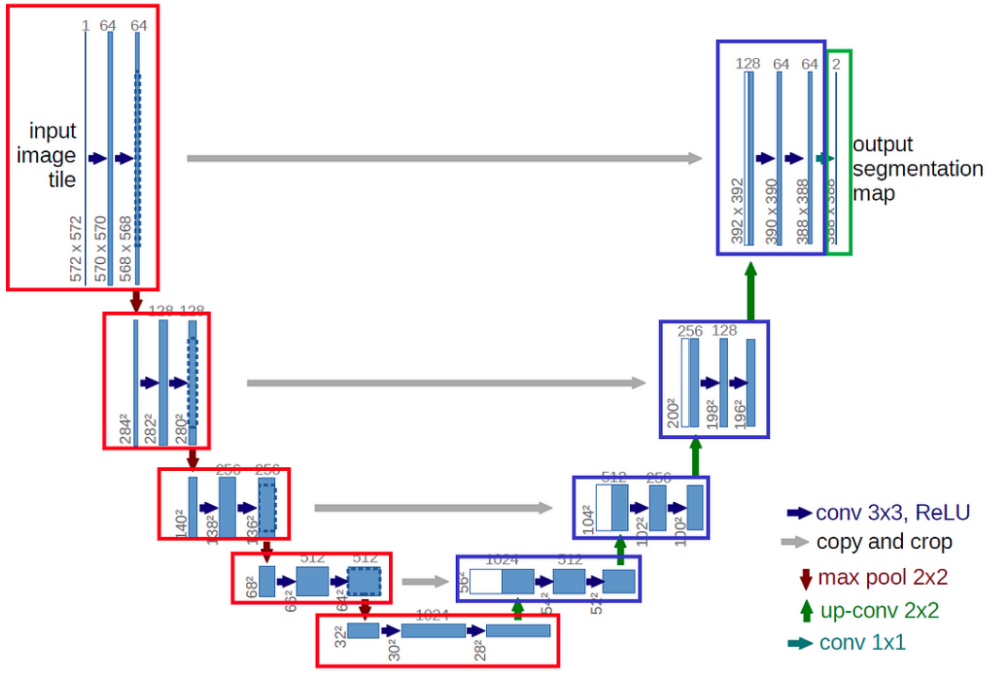
At training time, we don't want this reparametrization to be adaptive, so we will fix μ and σ to be the running averages of values that were collected at training time.

U-Net architecture

The base concept of a U-Net architecture is to perform downsampling of the input performed by an encoder, followed by an upsampling performed by a decoder. Within the encoder we have what's called a contraction path, where the input is squeezed to extract relevant features. The decoder then upsamples the squeezed input utilizing transposed convolutions.

The interesting thing about U-Nets is that level of the contraction part has a skip connection that sends that level to the corresponding level of the expansion in a ResNet fashion. In this way, the decoder will be able to take advantage of informations from the input if needed.

2


input image tile
572 x 572
570 x 570
568 x 568
output segmentation map
392 x 392
390 x 390
388 x 388
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

5 - Self Supervised Learning

Introduction

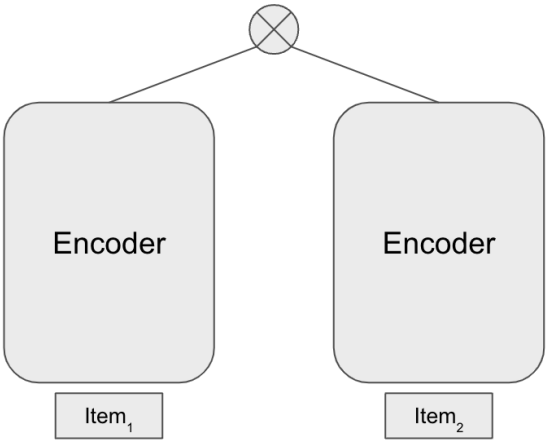
In Self-Supervised learning, our goal is to make our model learn on unlabelled data. There are a number of techniques and network developed around this approach. One of the core concepts is to learn representations effectively by comparing similar objects.

Siamese Neural Networks

With the Siamese Neural Network approach, we utilize two encoders that are internally the same. Both encoders get fed the same input or two very similar inputs, and the encoders are trained to produce the same representation in the latent space. The way training works is by designing a loss function that is based on a similarity metric, such that we can compute the distance between the representations and make the encoder learn to minimize it.

By doing this training, we are implicitly forcing the geometry of the latent space to obtain a meaningful representation of the input.

In particular, we say that we are performing a pretraining of the encoder. In this way we will be able to finetune the encoder on a downstream task, but starting from a network that already contains an implicit understanding of the input. These kind of pretraining tasks are clalled *proxy tasks*.



Contrastive learning

Contrastive learning is a very popular technique in the Self Supervised Learning landscape.

It is based on making our model learn similar representations for inputs of the same class, while maximizing the distance between representations of inputs of different classes in the embedding space.

Mathematically, the idea is to minimize the distance for x_i and x_k if $y_i = y_j$, and maximize the distance between the two to be at least ϵ if $y_i \neq y_j$. In this way we design the contrastive loss function as:

$$\mathcal{L}_{cont}(x_i, x_j, \theta) = I[y_i = y_j]||f_\theta(x_i) - f_\theta(x_j)||_2^2 + I[y_i \neq y_j]max(0, \epsilon - ||f_\theta(x_i) - f_\theta(x_j)||_2)^2$$

ϵ is a lower bound and is treated as an hyperparameter.

Triplet loss

In triplet loss, we have three inputs: one anchor x, one positive example x^+ and one negative example x^-. The objective of the loss is to minimize the distance between the anchor and the positive, while maximizing the distance between the anchor and the negative. The triplet loss is formulated like this:

$$\mathcal{L}_{triplet}(x, x^+, x^-) = \sum_{x \in X} max(0, ||f(x) - f(x^+)||_2^2 - ||f(x) - f(x^-)||_2^2 + \epsilon)$$

Ingredients for contrastive learning

Heavy data augmentation

As we have seen, contrastive learning techniques require a lot of data to work properly, since we have to produce negative and positive examples for each input we have. The solution often is to augment the initial dataset by generating new samples obtaining by applying transformations to the original data. This is done to create mainly positive examples.

Large batch size

Having a large batch size allows to have many negative examples at once. In this way the loss function can cover a diverse enough collection of negative examples.

Hard negative mining

In order to make the network learn properly, we need to challenge it proposing hard negatives.

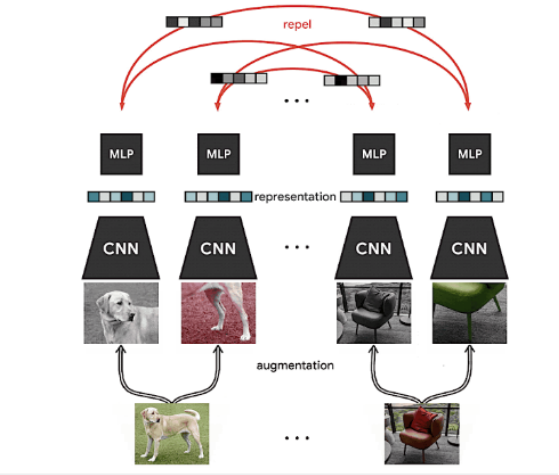
Image based SSL

A common workflow to pretrain a model is to train it on a variety of tasks with unlabelled images. We can make the model “play” different games such as aligning with distortions, rotations, patch reconstruction and jigsaw puzzles.

Popula SSL algorithms

Sim CLR (Similarity Contrastive Learning Representation)

We take to samples of different classes, apply two transformations each and feed the obtained transformed samples to 4 CNNs. This CNNs will project representations to a common embedding space. The objective then is to use triplet losses to attract representations of the same augmented object and repel representations of different ones.



BYOL (Bootstrap Your Own Latent)

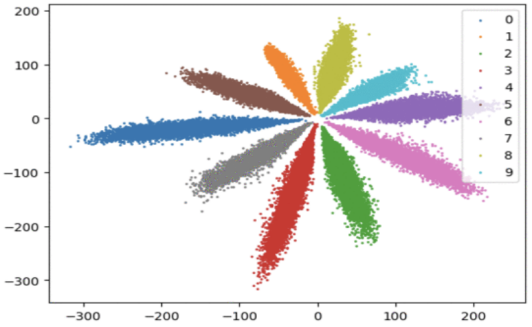
BYOL is not based on contrastive learning, since it does not utilize negatives. Instead, it feeds the same input to two networks: one network has to learn the parameters, the other one is used to adjust these parameters.

Barlow twins

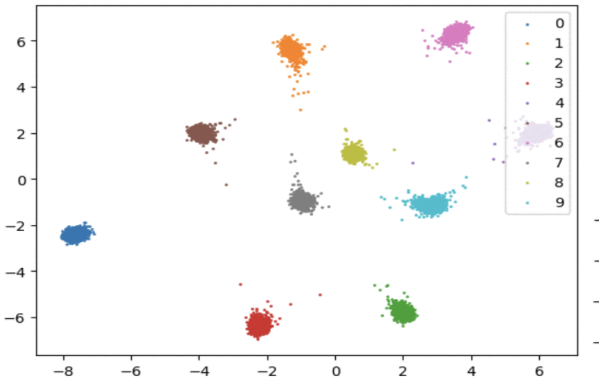
Here as well we don't need negatives. The Barlow Twins method is based on using to siamese networks to produce representations of different objects. From these representations we build a correlation matrix, and the loss will measure the difference between the cross correlation matrix and the identity.

Issues with SSL

If we think about the result of the contrastive loss in the embedding space, we would imagine the result to be well separated clusters of classes. Actually the results is messier, with elongated cluster that are not very well separated:



An effective method that has been developed to improve clustering is to add a regularization term to the contrastive loss that makes elements of the same cluster to be closer to their centroid. In this way we obtain blobs of points that are much better separated:

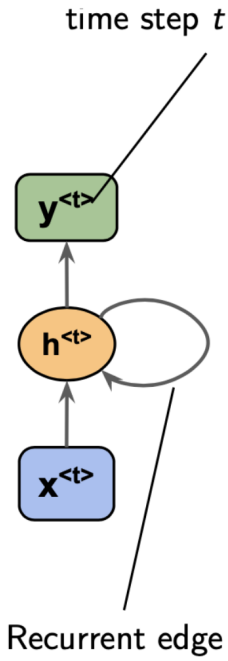

0
1
2
3
4
5
6
7
8
9
6
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

6 - RNNs and Transformers

Recurrent Neural Networks

In the real world we have a lot of examples of sequential data like text, videos, sounds etc.

Before the advent of transformers, RNNs were the most utilized networks to model sequences. The core idea of Recurrent Neural Networks is to add a self loop to the hidden state block. In this way, the output hidden state is passed as a new input for the next computation. If we think about it, this is a way to add make the network aware of the portion of the sequence that was already processed by means of an internal representation of the hidden state.



This resembles a lot the concept of feedback loop in theory systems: we have a state that is function of the current input and the previous state:

$$h^{<t>} = f(x^{<t>}, h^{<t-1>})$$

How do we backpropagate through these computations?

We have to unfold computation, which mean to consider the input, state and output at each timestep. The loss will be computed as the combination of the losses for each output:

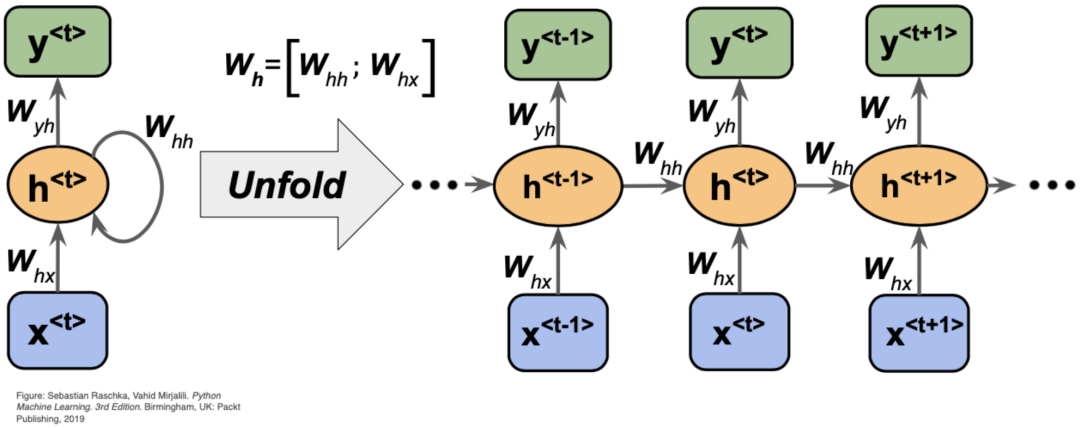


Figure: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition.* Birmingham, UK: Packt Publishing, 2019

This is called **backpropagation through time**.

How do we instantiate a vanilla RNN?

A possible implementation of an hidden layer is the following one:

$$h_t = tanh(W_{hh}h_{t-1} + W_{hx}x_t)$$
$$O_t = \text{softmax}(W_{ho}h_t)$$

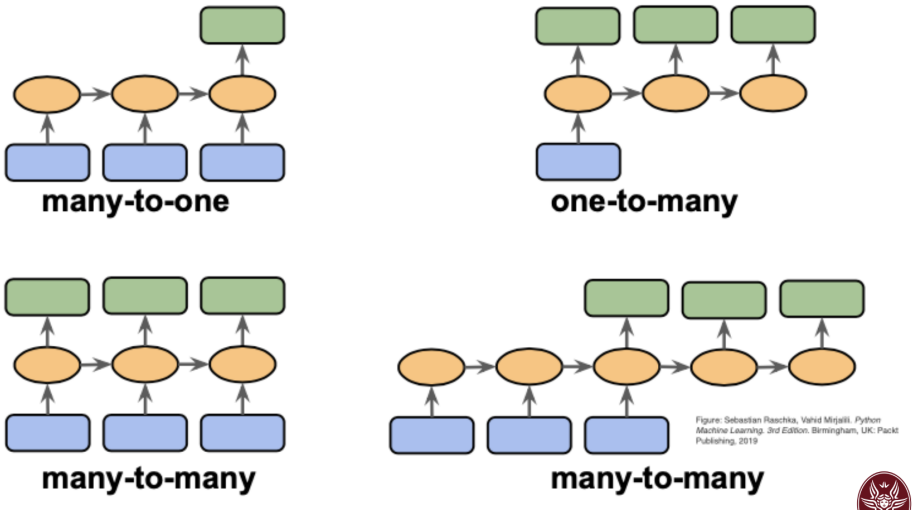
Notice how the hidden state at times t depends on the matrix W_{hh}, which is the one that weights the previous hidden state and the matrix W_{hx}, which is the one that weights the current input. Finally, the hidden state is passed through an hyperbolic tangent, which squeezes the values between -1 and +1. This allows to assign also negative values to the hidden state if it is less useful to compute the next output.

Sequence to sequence models

In sequence to sequence tasks we want to go from a sequence of objects to a sequence of other objects. It's important to notice that the input sequence and output sequence an be of different lengths.

We have three different kind of tasks:

- Many to one → from a sequence to one object (ex. sentiment classification).
- One to many → from one object to a sequence (ex. label to image).
- Many to many → from a sequence to another sequence (ex. translation).



Encoder-decoder paradigm

In the encoder-decoder paradigm, we use an encoder to produce the representation of the input, then a decoder takes this representation and produces the output.

Autoregressive models

In seq2seq tasks is often the case that the sequence that we have to generate is dependent on the previous portion of the sequence. In a standard encoder-decoder architecture, we produce an hidden representation of the input utilizing an encoder, and then we start producing the output utilizing that representation.

In autoregressive models the concept is to feed as an input at timestep t the original input plus the generated output up to timestep t-1. In this way we obtain richer representation of the sequence at each timestep.

The input for autoregressive models is created by encoding (tokenizing) the original input. This tokens are then embedded and fed to the network.

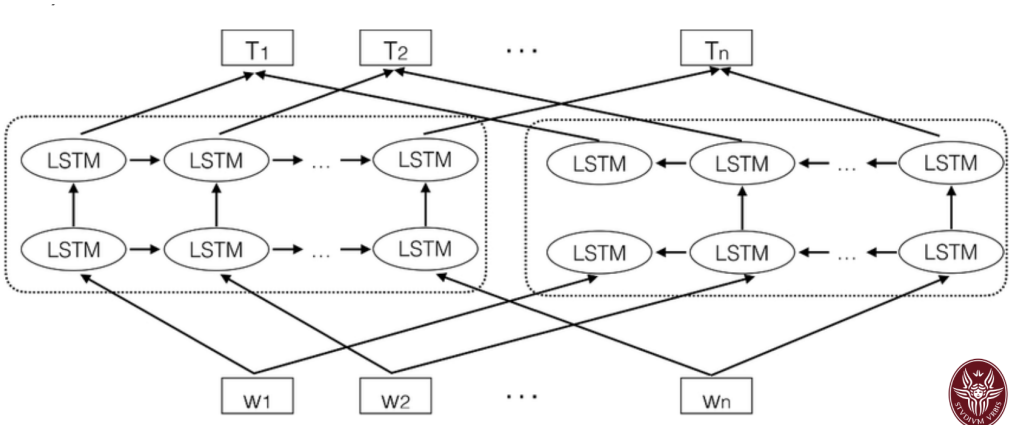
Teacher forcing

Since at training time we often know the ground truth of a sequence, we can choose to give as input token at time t, the ground truth token with probability p or the generated token with probability 1-p. In this way we help the model by reducing the number of mistakes. This technique is called teacher forcing.

ELMo

ELMo has been the first model to exploit the concept of contextualized embeddings. In fact, if we think about seq2seq tasks like translation, considering the context of a token is really important to model it in a meaningful way.

The way ELMo works is by utilizing two LSTM layers. the first one scans the sequence forwards, while the other one scans the sequence backwards. In this way we can merge the hidden representations of the two LSTMs to produce a final representations that is richer in context. Word embeddings will not be static but contextualized.

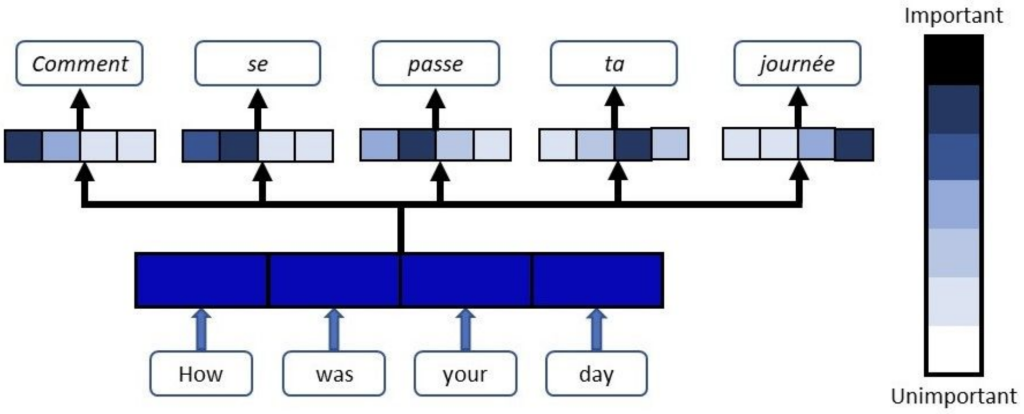


Transformers

Attention mechanism

If we think about it, when producing an output of a sequence that it depends only on small portions of the input. We'd like a way to focus adaptively on parts of the input sequence. The attention mechanism implements this intuition in the following way:

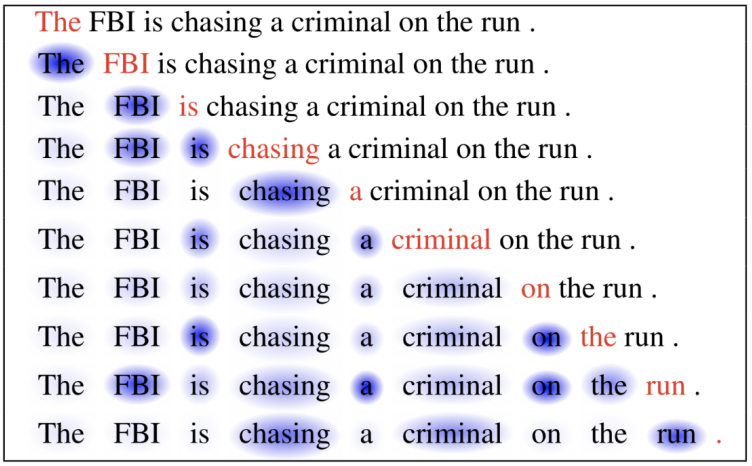
For each output we produce a vector as long as the input sequence. Each element of this vector will measure the relevance of the input for that output (attention score).



Self-attention

Self-attention is the particular mechanism that is utilized in transformers. the main intuition is the following: each token is treated as a key that we use to build a query. The query is sent to a memory, that returns the values (scores for

parts of the input) that are relevant for that specific key. In this mechanism, queries, keys and values are all the same, and they correspond to the input sequence, that's why it's called self attention.



The self-attention is implemented in the following way:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{n}})V$$

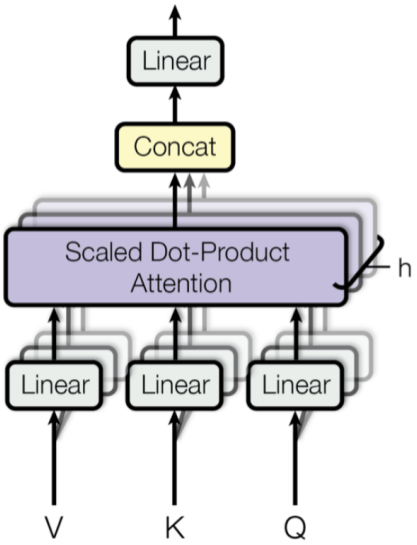
The $softmax(QK^T)$ returns scores for each element of the sequence given a specific query and value. The score for each word is computed by taking the dot product of the query vector with keys for all the words. Then we apply the softmax and sum.

Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}

Multi-Head attention

An advanced way to implement self attention is to use multiple attention heads. In this way, we aim at extracting different levels of semantics for the input sequence. The resulting attention scores are then concatenated and projected.

A huge advantage is that the various attention heads can be parallelized and computed independently o multiple GPUs.



Transformer architecture

Natively a Transformer is an encoder-decoder based seq2seq model. There relevant steps of computation within the transformer are the following.

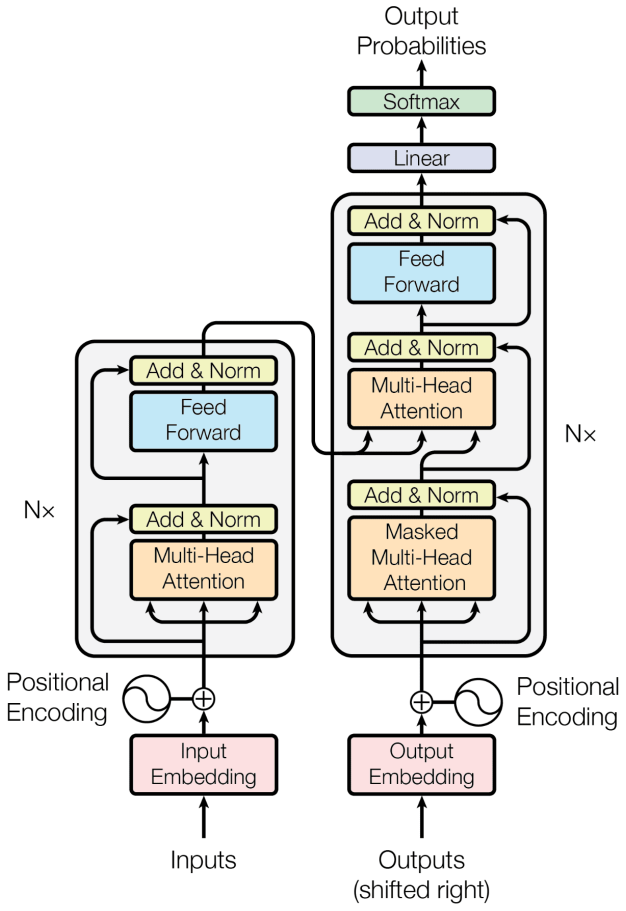
Encoder

1. Add to the tokenized input the Begin Of Sequence and End Of Sequence tokens. The tokenized input is embedded using an embedding layer.
2. In order to keep memory of the ordering of tokens, we add to the embeddings positional encodings.
3. Feed the obtained input to the multi-head attention block.
4. Normalize and add the residual input.
5. Project the result using a feed forward network.

Decoder

1. Take the encoded input and pass it to a multi-head attention block.
2. Nomalize and add.

3. Pass the output through a feed forward neural network and optionally a softmax.
4. The projected output goes back as input, passing to a masked multi-head attention block.



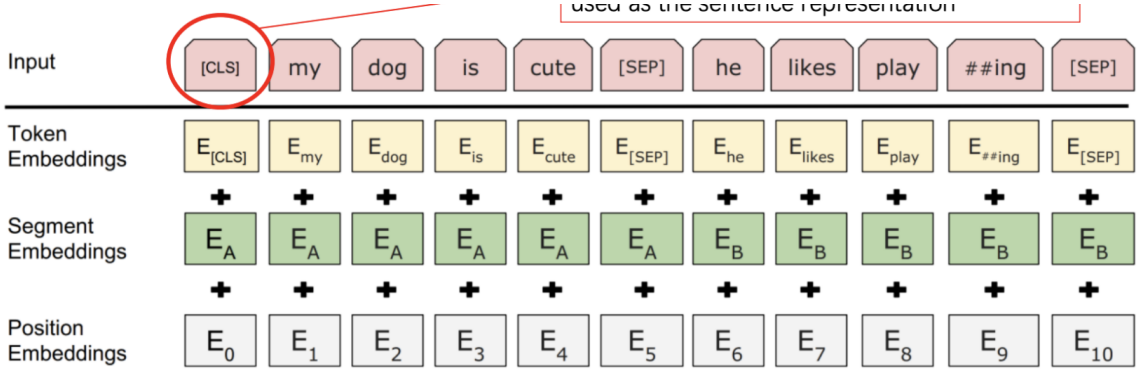
BERT

Bert is a multi-layer bidirectional transformer. It uses only the encoder part of the transformer to create representations for language. It inherits the same structure as ELMo, but with transformers instead of LSTMs.

One of the major pros of BERT is that it was pretrained on two very interesting taks:

- Masked Language Modelling → mask some words within a sequence and train the model to reconstruct the correct word. In this way, the model learns contextualized word embeddings. In particular with 80% probability a word is masked, with 10% is replaced with another random word, with 10% is not replaced at all.

- Next sequence prediction → Give two sequences, the model has to determine which sequence follows the previous one. In this task we add the [CLS] token, which aims at representing the semantics of the entire sequence.



7 - RAG

Introduction

NLP problems can be broadly divided in tasks that do not require specific domain knowledge and knowledge intensive tasks. With knowledge intensive tasks we mean tasks where we need access to an external knowledge to give a proper answer.

Classical autoregressive knowledge are trained on the next token prediction task, which has proven to perform incredibly well, but with some huge problems. One of the main problems are hallucinations, which are episodes where the model responds with nonsensical of non-factual answers. RAG tries to solve this problem by grounding answers to explicit knowledge resources.

Retrieval Augmented Generation basics

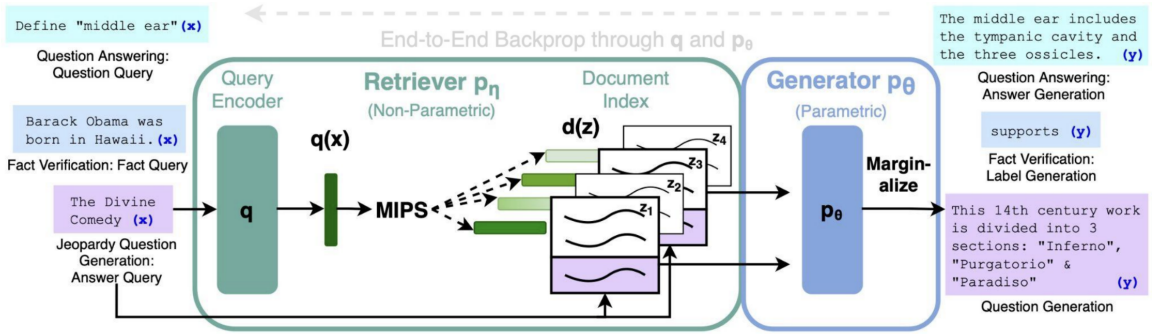
The basic architecture for RAG is the Retriever/Generator paradigm.

As a first step we have to generate embeddings for every single document we have to feed. The retriever model is trained by giving as input the embedding of all documents and the embedding of the query, and the model has to return the most important documents for that query. This is done by computing a similarity score between the embeddings. We can formalize this operation in the following way: we want to compute the probability of the most likely document z given a query x. In other words $p_\eta(z|x)$.

Once the document is retrieved, it is appended to the original input query. This new input is then passed to the generator (for example T5), which will have to generate the correct answer y given the query and the document: $p_\theta(y|z,x)$.

The joint probability will be:

$$p_{\text{rag-sequence}}(y|x) = \sum_{z \in \text{top-k}(p(\cdot|x))} p_\eta(z|x) p_\theta(y|z,x)$$

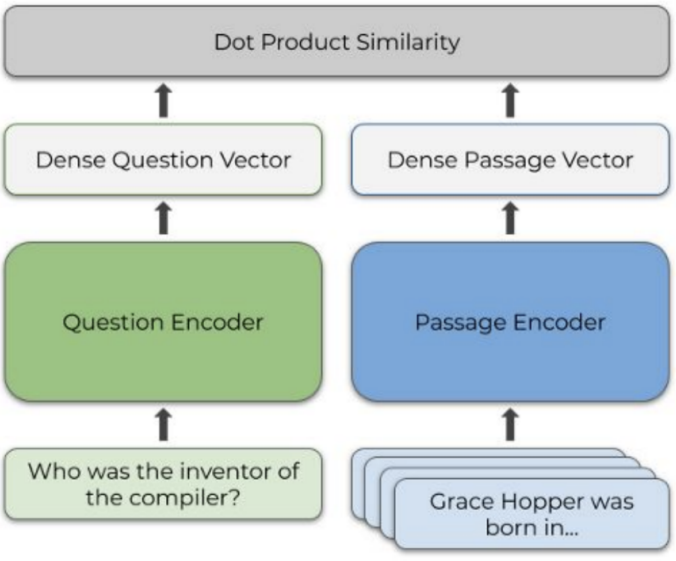


You don't need to train the full architecture, often the generator is partially trained.

Dense passage retrieval

In passage retrieval we don't want to look among many documents, but identify the most relevant passage (segment) within a document that answers our query.

In dense passage retrieval we use two encoders: the question encoder creates a vector for the input query, while the passage encoder encodes all the passages of the document. The model will then return the passage (or top-k passages) with the highest similarity score.



A possible approach to passage retrieval is to use the extractive method. We use a bert-based reader that is trained in predicting two particular labels: the start of the relevant sequence and the end of it. In this way we will be able to extract the passage without the need of generation of text.

Indexing

Indexing is the process of building a data structure for a fast lookup among documents. This is crucial where we have databases of millions of documents.

Metrics for RAG

- **Speed** → how long does it take to find relevant documents.
- **Memory usage** → how much RAM is consumed.
- **Accuracy** → relevance of the returned list measured on MAP, recall@k etc.
- **Index build time** → how much doe it take to rebuild the index.

RAG for contemporary LLMs

In modern approaches we exploit large language models that were pretrained on huge quantities of text. We fine-tune the model to answer correctly to the given queries. Recent relevant papers are:

- WebGPT → data is collected from online resources by human annotators. Given a query, the annotator picks the best answer from the web.
- Toolformer/Cohere RAG → the model learns to use other tools than just search (ex. calculator). It does this by understanding when to make API calls and incorporating their result within the input.

Challenges

- Bias in the knowledge sources (blackbox search engines, wikipedia ...).
- Source quality.
- Size of the context window (limited by the architecture).

8 - Generative modeling

Introduction

Machine learning models can be divided in two main categories: discriminative models and generative models. With discriminative models, the goal is to assign a certain label y to a given input x. For generative models instead, we are interesting in generating new samples from a distribution learned by the model. In mathematical terms we want to sample $x \approx P_\theta(X)$. For example if $P_\theta(X)$ is a distribution of images, then x will be an image.

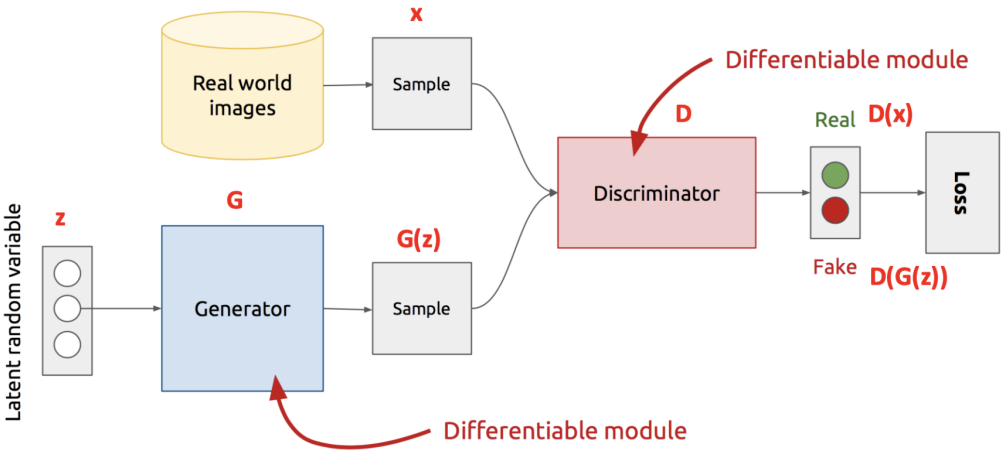
Generative Adversarial Networks (GANs)

Usually with machine learning models we are interested in computing the maximum likelihood for a function. GANs instead use an adversarial approach, which differentiates from the concept of maximum likelihood.

The GAN architecture has two main actors: the generator and the discriminator.

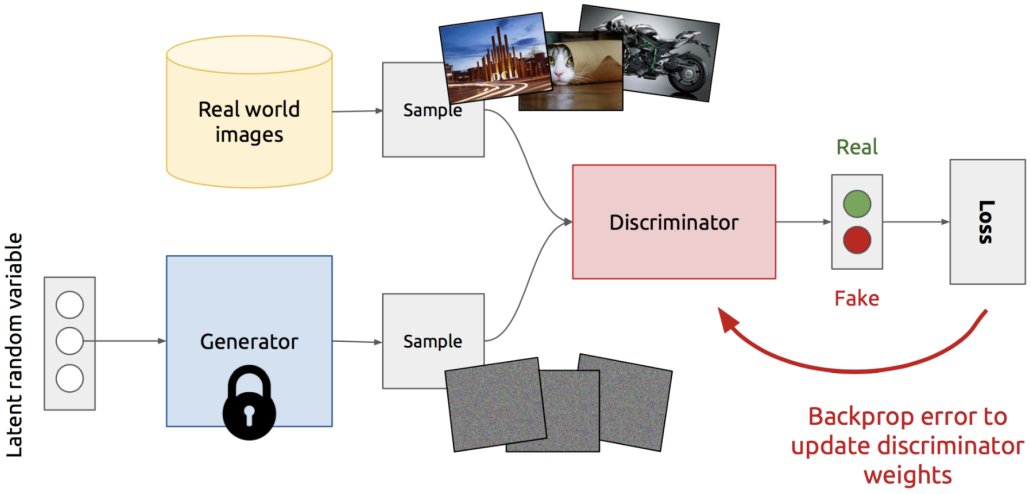
The **generator** creates samples from a latent random variable z. This random variable is described by a latent space, which contains an internal representation of the probability distribution we want to approximate. If we change z by a little bit, we expect to sample an object within the same distribution, but slightly different from the previous one.

The **discriminator** instead is basically a classifier: it's role is to discriminate fake vs real images.



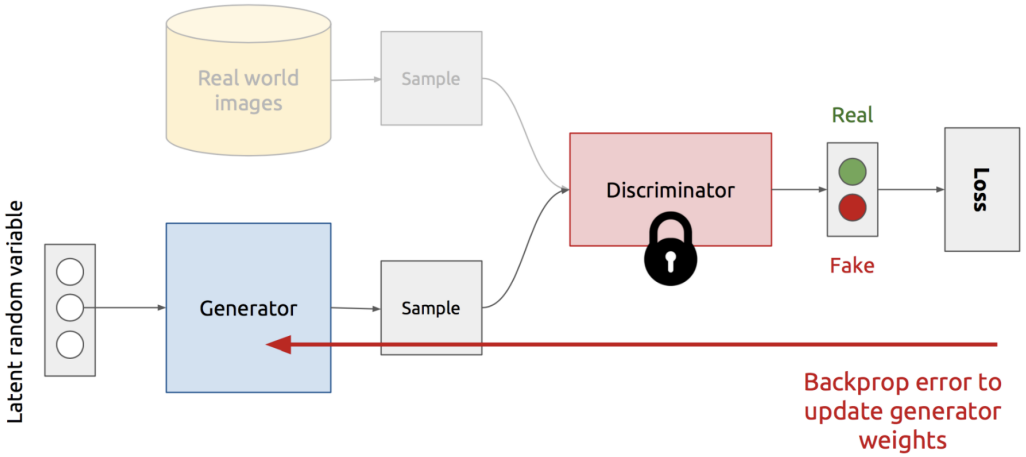
Training the discriminator

When training the discriminator, we keep the generator frozen. This means that only the discriminator's weights are updated. During training, we feed the discriminator with an input that sometimes is a real image, and sometimes is generated. The discriminator has to correctly label the input as real or fake. We expect $D(x)$ to be classified as real and $D(G(z))$ to be classified as fake.



Training the generator

When training the generator, we freeze the discriminator's weights. The generator will have to generate images and feed them to the discriminator. The loss will be computed based on samples that are able to fool or not the discriminator. In this way the generator learns to improve the generation of samples.



GAN formulation

We can formulate the functioning of a GAN as a min-max game. The discriminator wants to maximize its reward function, while the generator wants to minimize the reward of the generator. We can represent game with the following value function:

$$min_G\ max_D V(D,G)$$

$$V(D,G) = \mathbb{E}_{x \approx p(x)}[logD(x)] + \mathbb{E}_{z \approx q(z)}[log(1 - D(G(z)))]$$

When we feed a sample from the original distribution, if the discriminator is wrong and classifies the sample x as false (0), then the term of the loss explodes to infinity. Similarly we have a term that goes to infinity if the sample is from the fake distribution $q(z)$ and the discriminator classifies it as true (1).

For this kinds of min-max games, we are always guaranteed to have an equilibrium, called the Nash equilibrium. This point is reached when the distribution of the generator becomes the same as the original data, meaning that the discriminator will discriminate correctly with probability $\frac{1}{2}$.

The optimum for the discriminator will be to maximize the value function

$$D^*_G = argmax_D(V(D,G)).$$

The optimum for the generator conversely will be to minimize the previous equation:

$$G^* = argmin_G(V(D^*_G, G)).$$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

Discriminator updates

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

Generator updates

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

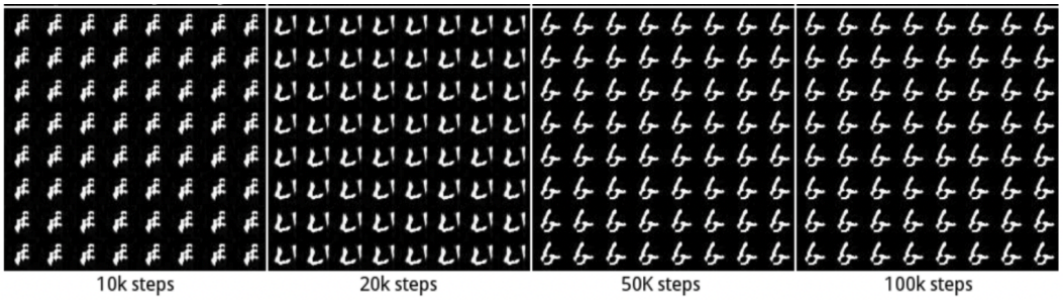
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Improving the stability of the loss

As we have seen, the loss for the generator is defined as $\mathbb{E}_{z \approx q(z)}[log(1 - D(G(z)))]$. A problem with this formulation is that at earlier stages, when the capabilities of the generator are really poor, the discriminator is very likely to do well its job. This means that we risk that the loss of the generator quickly saturates. A way to improve stability is to train the generator to maximize the following term: $log(D(G(Z))$. In this way we are guaranteed to have the same convergence, but with a much better stability.

Mode collapse

As we have seen before, the role of the generator during training is to fool the discriminator. One common issue that may arise from the current definition of GAN training is that if the generator learns to produce one sample so good that the discriminator is always fooled, then the generator might start producing only this output each time. This phenomena is called mode collapse.



Why does this happen?

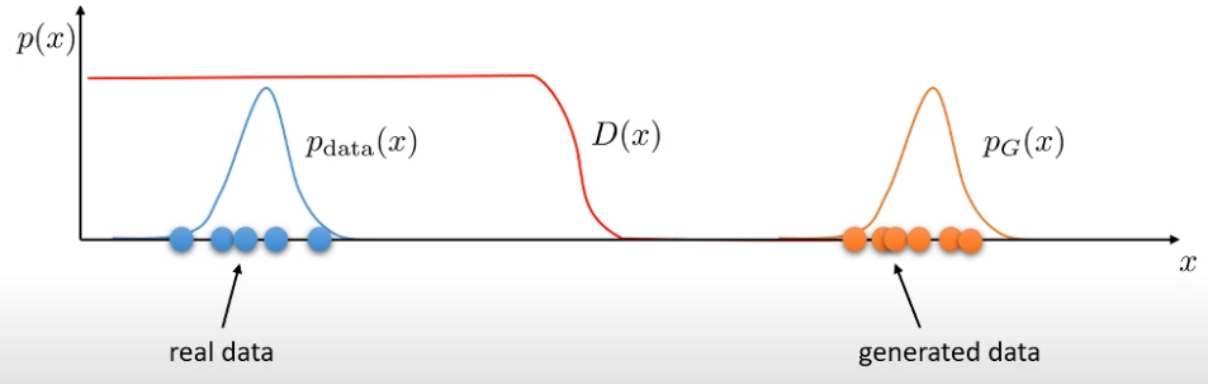
If we look at the formulation of the objective of the generator:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(z^{(i)})))$$

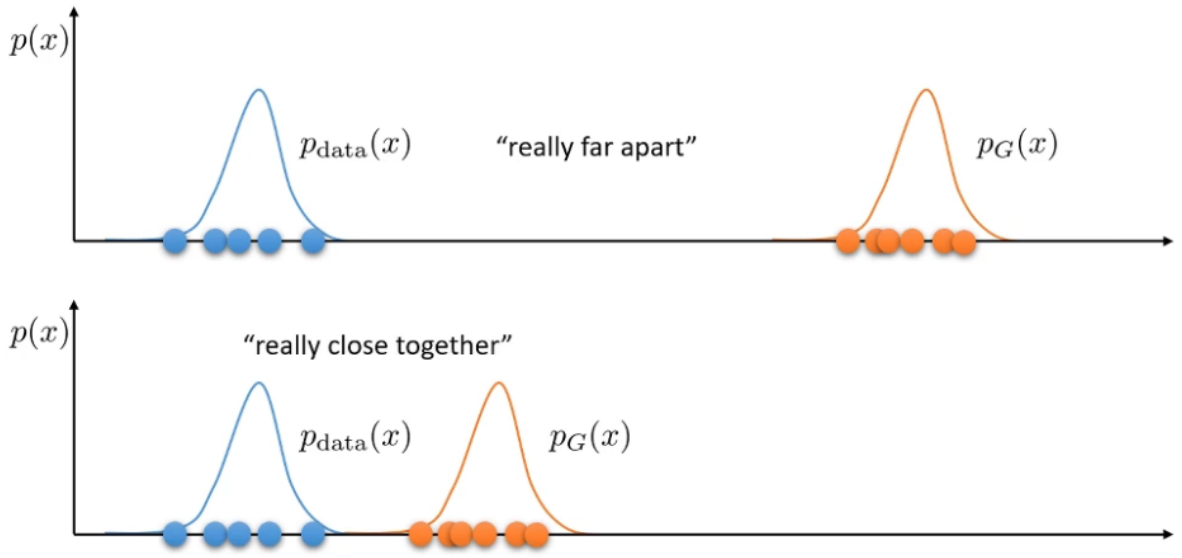
If the generator produces an image x^* that is so good to fool the discriminator, then this generated image will be enough to maximize the objective: $x^* = argmax_x D(x)$. In this way the loss will reach zero by just producing that term.

Wasserstein GANs

As we have seen, standard GANs come with two main problematics which are loss saturation at early stages and mode collapse. Both problems arise from the formulation of the GAN where the discriminator is only trained to output a 1 or 0 label to a sample in order to tell apart real and fake images. Here is a nice visualization:

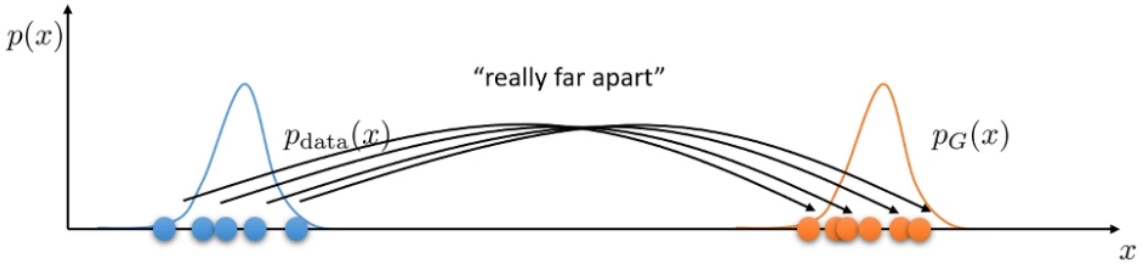


Another problem is that the JS divergence measure used in classical GANs does not take into account the notion of distance. For example the divergence of these two distributions below is almost the same, as there is no overlap in both cases.



We would like a divergence metric that tells the gradient the direction in which the two distribution can become closer.

Wasserstein's distance measures the total distance that we have to travel in order to move the various points from a distribution to another in order to make the first distribution identical to the second one. This notion implements a concept of distance in the Euclidean space.



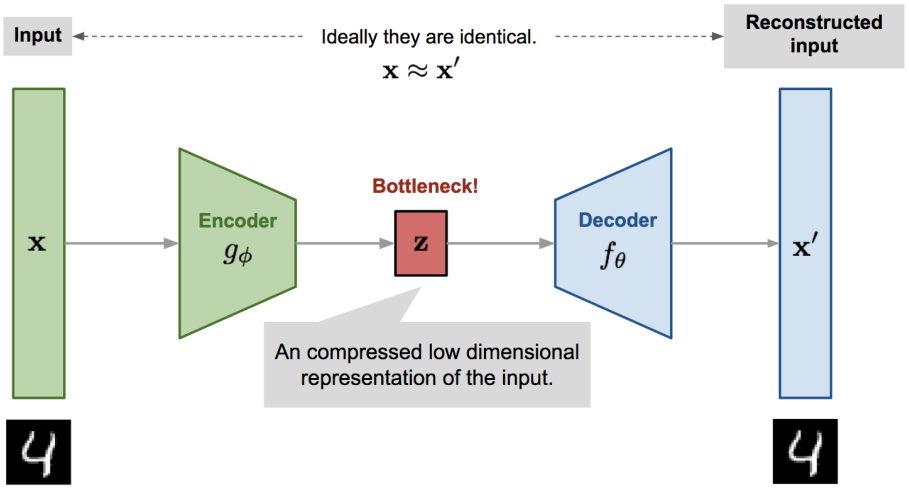
The idea then is to make the discriminator learn not the difference between real and fakes, but an actual continuous value that measures the distance between the two distributions. This is possible under the Kantorovich-Rubinstein duality, which demands a function to be K-Lipschitz continuous. This is done by clamping gradients.

The "discriminator" model does not play as a direct critic but a helper for estimating the Wasserstein metric between real and generated data distribution.

Autoencoders

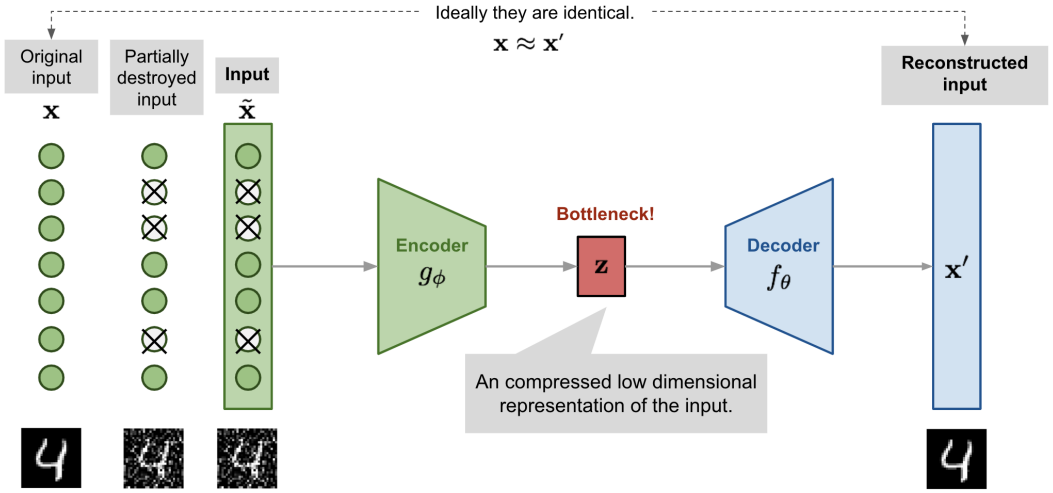
Introduction

Autoencoders are a family of encoder-decoder architectures. They work by using an encoder to compress the input x into a latent space z. The decoder then will have to use the latent representation to reconstruct the original input. In this way we expect to encode within the latent space a low dimensional representation of the input space, that we can utilize in different ways.



Denoising autoencoders

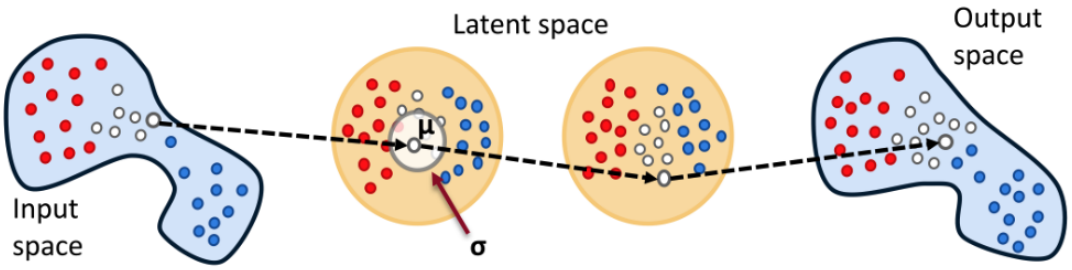
Denoisining autoencoders have the same basic architecture, but we train them in a different way. In particular, we create multiple corrupted versions of the same input, and we train the autoencoder to recreate the original one. This makes the model more robust to perturbations.



Variational Autoencoders (VAEs)

As we have seen, an autoencoder has the potential of recreating samples from a latent representation. Ideally, this could be used for generation of new samples, but the standard architecture isn't enough. This is because we don't have any constraints on the geometry that the latent space learns. In other words, representations of very different objects can be very close within the latent space geometry, meaning that we can't really use it in a controlled way.

The fundamental idea of VAEs is to map the input not into a single vector, but into a distribution $p_\theta(z)$.



The relationships between the input x and the latent representation z can be summarized by:

- $p_\theta(z)$ → the **prior**. This prior will define the shape of the latent space distribution.
- $p_\theta(x|z)$ → the **likelihood**. How do we move back from the latent space to the input space.
- $p_\theta(z|x)$ → the **posterior**. How to represent the latent space from the input space.

If we assume to have the optimal parameters θ^* that characterize the distribution, then the sampling operation will be the following:

1. We sample a $z^{(i)}$ from the prior $p_{\theta^*}(z)$.
2. We generate a $x^{(i)}$ from the likelihood conditioned on the latent sample $p_{\theta^*}(x|z = z^{(i)})$.

Actually, we don't know θ^*, so how do we find it?

We might think to use the traditional way of maximizing the log likelihood of generating real data samples:

$$\theta^* = argmax_\theta \sum_{i=1}^{n} p_\theta(x^{(i)})$$

The problem here is that in order to compute $p_\theta(x^{(i)})$ we would need to compute this integral:

$$p_\theta(x^{(i)}) = \int p_\theta(x^{(i)}|z)p_\theta(z)dz$$

The problem is that this integral is intractable, meaning that it doesn't have a closed form solution. What we can do, is to find a new distribution $q_\phi(z|x)$ that approximates the original one with a tractable distribution (ex. a Gaussian). If we know this distribution, then we can revert it and use it to generate new samples.

$q_\phi(x|z)$ will be a generative model, known as probabilistic decoder.

$q_\phi(z|x)$ will be an approximation function, known as probabilistic decoder.

ELBO

As we said, we want the estimated posterior $q_\phi(z|x)$ to be really close to the original one $p_\theta(z|x)$. In order to measure this discrepancy, we can use the KL divergence to quantify the distance. Out goal will be to minimize $D_{KL}(q_\phi(z|x)||p_\theta(z|x))$ with respect to ϕ. This divergence is computed as:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \int q_\phi(z|x) log \frac{q_\phi(z|x)}{p_\theta(z|x)} dz$$

By means of algebraic manipulation, we can get to the following equation:

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \log p_\theta(x) + D_{KL}(q_\phi(z|x)||p_\theta(z)) - \mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z)$$

We can rearrange the equation in the following way:

$$\log p_\theta(x) - D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(x|z) - D_{KL}(q_\phi(z|x)||p_\theta(z))$$

Then we can see that the right term represents what we were looking for: maximizing the log likelihood of generating real data and minimizing the difference the real and estimated posterior distribution.

Finally, we can define the loss function for our VAE as:

$$L_{VAE}(\theta, \phi) = -\log p_\theta(x) + D_{KL}(q_\phi(z|x)||p_\theta(z|x))$$

Our aim will be to find $\theta^*, \phi^* = \arg\min_{\theta,\phi} L_{VAE}$

Finally, by reversing the sign of the loss we can see that:

$$-L_{VAE}(\theta, \phi) = \log p_\theta(x) - D_{KL}(q_\phi(z|x)||p_\theta(z|x)) \leq \log p_\theta(x)$$

Maximizing $\log p_\theta(x)$ implies minimizing the KL divergence, meaning that we are getting closer to the original distribution.

Reparametrization trick

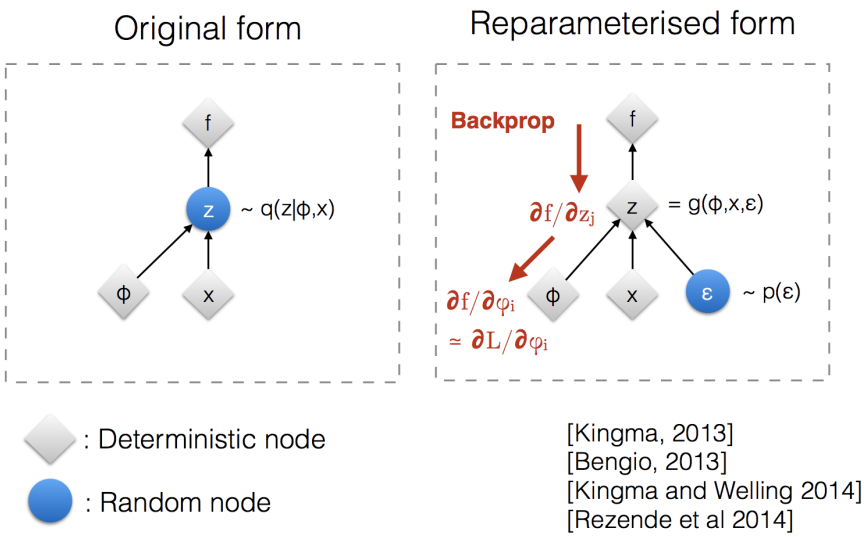
We have one last problem. The computation of the loss function requires to generate samples from $q_\phi(z|x)$. Since the sampling process is stochastic, we cannot perform backpropagation on sampling.

The reparametrization trick introduces a new random variable ϵ which is sampled from a normal distribution, while keeping the mean and standard deviation fixed. If the choice of the distribution is a multivariate Gaussian:

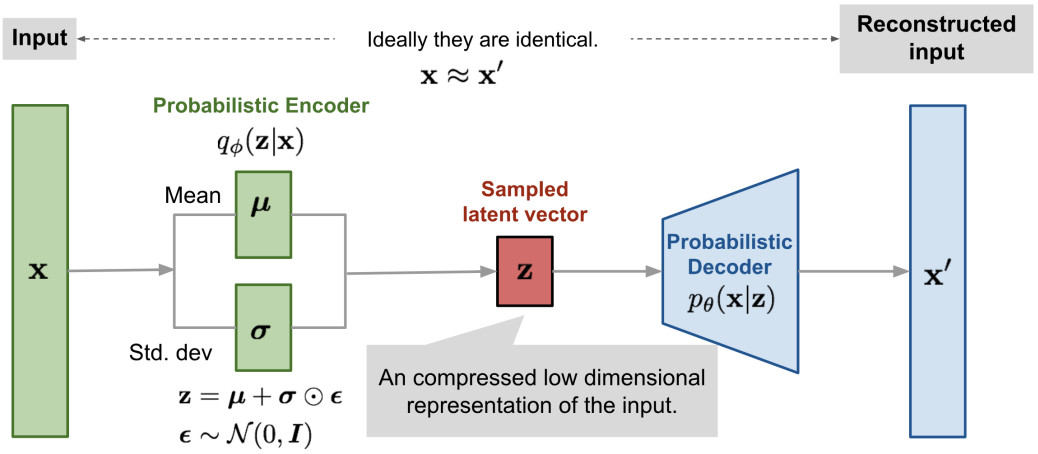
$$z \sim q_\phi(z|x^{(i)}) = \mathcal{N}(z; \mu^{(i)}, \sigma^{2(i)} I)$$

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

In this way, the stochasticity is only related to ϵ and we can backpropagate through the mean and std terms:



The final architecture of the VAE will be:



In order to generate a new sample, we can pick values for mean and standard deviation. The probabilistic encoder will then sample a certain vector from the latent space, and finally the probabilistic decoder will generate a new sample. Since this latent representation is continuous, we can move around the same distribution to generate similar objects, but different one another.

9 - Graph Neural Networks

Introduction

Graphs are a very powerful kind of structure that are able to represent many real world objects like molecules, social networks etc. In this context, Graph Neural Networks try to extract latent representations of these objects by exploiting the structure of the graph and the relationships between nodes.

GNN basics

We can define a graph G as a set of vertexes and edges $G = (V, E)$. A graph can be represented by means of an adjacency matrix. An adjacency matrix is an $N \times N$ matrix (where N is the number of nodes) where the i, j entrance is 1 if there is an edge between node i and j, 0 otherwise.

We want to learn a low dimensional vector representation (embeddings) $\{z_i\}_{i \in \{|V|\}}$ for noes in the graph $\{v_i\}_{i \in \{|V|\}}$ such that important graph properties like the structure are preserved in the embedding space. In this way we can create measures of similarity between nodes that have similar structures.

Nodes and edges can have attributes that we call features.

We want to learn a node embedding matrix $Z \in \mathbb{R}^{|V| \times D}$ and a mapping between the adjacency matrix W and Z.

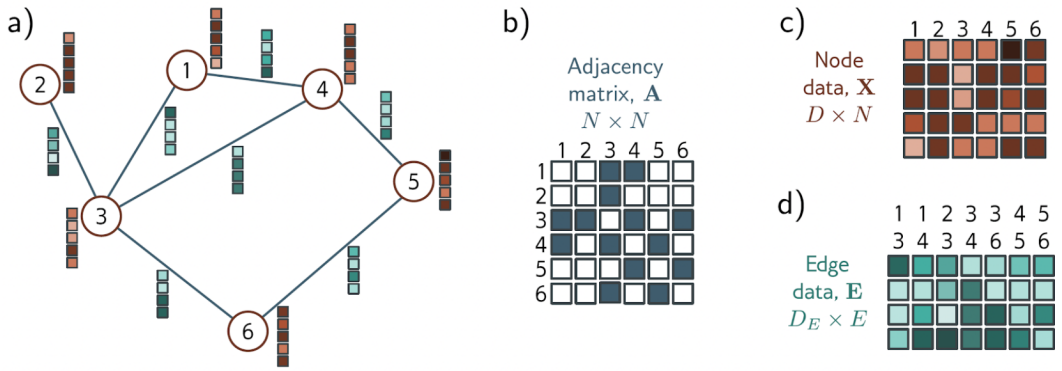


Figure 13.3 Graph representation. a) Example graph with six nodes and seven edges. Each node has an associated embedding of length five (brown vectors). Each edge has an associated embedding of length four (blue vectors). This graph can be represented by three matrices. b) The adjacency matrix is a binary matrix where element (m, n) is set to one if node m connects to node n. c) The node data matrix $\mathbf{X}$ contains the concatenated node embeddings. d) The edge data matrix $\mathbf{E}$ contains the edge embeddings.

In general, given the adjacency matrix A and a data matrix X, we will feed them through GNN layers to create hidden representations that will then be utilized for one of the following tasks:

- Graph-level tasks.
- Node-level tasks.
- Edge-level tasks.

Graph level tasks

In graph level tasks we want to assign a label or a number associated to the entire graph by exploiting the embeddings and the structure of the graph:

$$Pr(y = 1|X, A) = sig[\beta_k + 2_k H_k 1/N]$$

Where β_k are the biases, w_k is a $1 \times D$ vector of learned parameters and H_k is the embedding matrix for layer k.

Node level tasks

In node level tasks we want to perform regression or classification for the single node. In general we can formulate this for node n as:

$$Pr(y^n = 1|X, A) = sig[\beta_k + w_k h^n]$$

Edge level tasks

In edge level tasks we want to assign a probability of the existence of an edge between node n and node m:

$$Pr(y^{nm} = 1|X, A) = sig[h_k^{(m)T} h_k^{(n)}]$$

Graph convolutional networks

The operation of convolution in graphs comes in handy, as it exploits the inductive bias that we want to enforce of utilizing the structure of the graph and the relation between nodes. Each node will be represented by the aggregation of informations coming from its neighbors.

In general a GCN layer is a function $F[\cdot]$ with parameters ϕ that takes as input node embeddings and an adjacency matrix and outputs new node embeddings:

$$H_K = F[H_{k-1}, A, \phi_{k-1}]$$

For each node of the network, we apply an aggregation function to its neighbors (in this case a sum):

$$agg[n, k] = \sum_{m \in ne[n]} h_k^{(m)}$$

After the aggregation, we apply a linear transformation Ω_k to the aggregated embeddings and the embedding of the current node, and finally sum a bias term:

$$h_{k+1}^{(n)} = a[\beta_k + \Omega_k h_k^{(n)} + \Omega_k agg[n, k]]$$

In a more compact way, we can compute the aggregation for each node by means of matrix multiplication of the node embeddings and the adjacency matrix A:

$$H_{k+1} = a[\beta_k + \Omega_k H_k + \Omega_k H_k A] = a[\beta_k 1^T + \Omega_k H_k (A + I)]$$

For a final classification, we can aggregate embeddings by averaging them (avg pool), sum them to a bias and apply a sigmoid:

$$f[X, A, \phi] = sig[\beta_k + w_k H_k 1/N]$$

Batching

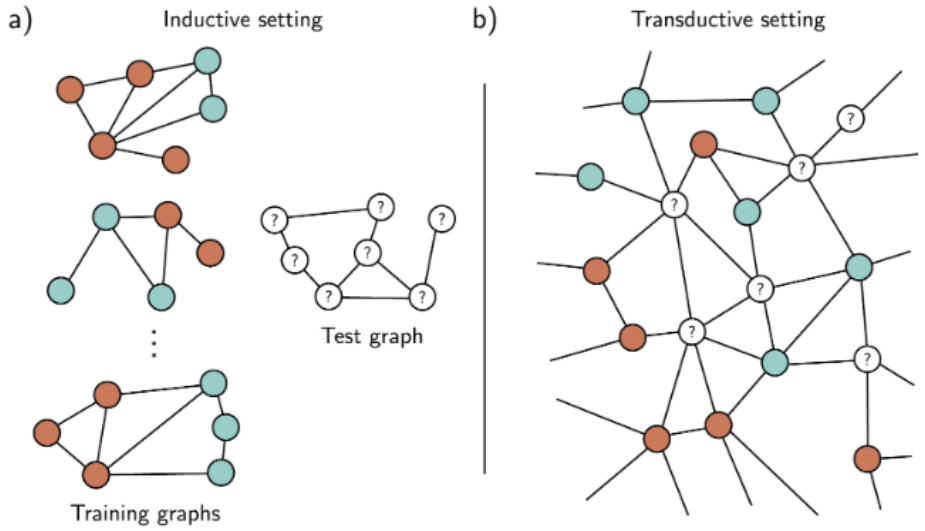
How do we perform batching if graphs have various shapes and sizes?

We can do something very similar to padding: we create a bigger adjacency matrix, and the original adjacency matrix will be a block matrix within it. When pooling we will have to remember to consider only the original matrix.

Inductive vs Transductive

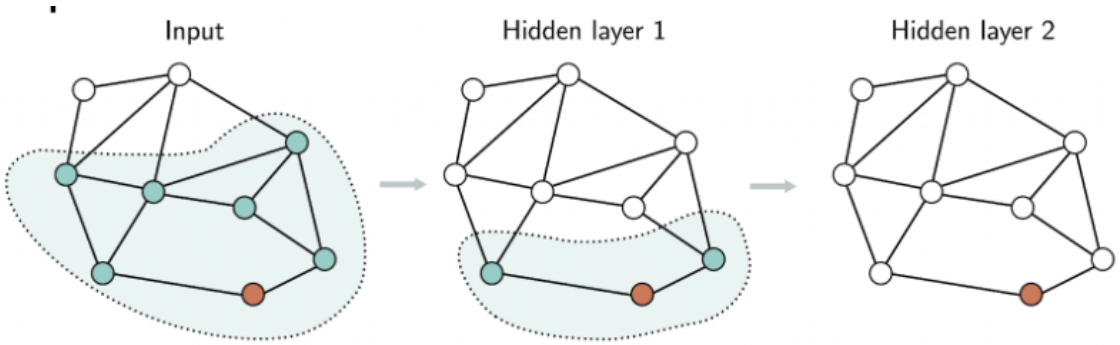
In the inductive setting we have multiple separated graphs, and for each graph we have train and validation nodes and a single test graph.

In the transductive setting instead we have one single big graph which contains train, validation and test nodes. The transductive setting is more expensive and not really feasible as is.



Receptive field in graphs

When performing graph convolution, we know that for each node we aggregate the information coming from its neighbors. As the number of layer increases each node will aggregate the information coming from the neighbors of its neighbors and so on. This is the equivalent of a receptive field. We also call it k-hop neighborhood.



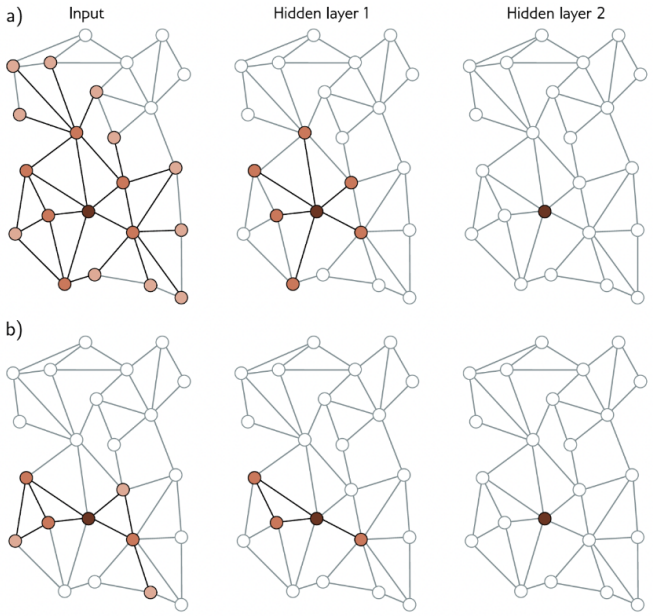
Issues with node classification

Taking into account an entire graph during training would require to store the embeddings for the nodes at each layer, and this may be really expensive when dealing with huge graphs.

Moreover, it's not obvious how to perform SGD if we have no batches, but a single graph.

We can form a batch by selecting a random subset of labeled nodes at each training step. As we have seen, at each layer we increase the receptive field, so we can use few nodes but still retain a lot of information. We will then perform gradient descent using the graph that forms the union of the k-hop neighborhoods of the batch nodes.

However, when we have a densely connected graph the solution before is still not efficient. A first approach can be performing neighborhood sampling: we only select a subset of neighbors for each node.



Another solution is graph partitioning: we can partition our graph in sub-graph and work with batches of sub-graphs.

Aggregation mechanisms

Until now we have seen sum as an aggregation mechanism for GCN. The following are alternative aggregation approaches.

Diagonal enhancement

We multiply each node by a factor $(1 + \epsilon_k)$ where ϵ_k is a parameter learned by the network for each layer:

$$H_{k+1} = a[\beta_k + \Omega_k H_k (A + (1 + \epsilon_k)I)]$$

Generalized diagonal enhancement

Like in diagonal enhancement, but we apply a generic linear transform ψ_k to each node:

$$H_{k+1} = a[\beta_k 1^T + \Omega_k H_k A + \psi_k H_k]$$

Residual connections

We can implement the concept of residual connection by propagating forward the embedding of the layer before:

$$H_{k+1} = \begin{bmatrix} a[\beta_k 1^t + \Omega_k H_k A] \\ H_k \end{bmatrix}$$

Mean aggregation

In some cases it makes more sense to aggregating using the average of the node's neighbors instead of the sum. In this case we use $agg[n, k] = \frac{1}{|ne[n]|} \sum_{m \in ne[n]} h_k^m$. In matrix form:

$$H_{k+1} = a[\beta_k 1^T + \Omega_k H_k (AD^{-1} + I)]$$

Where D is the diagonal degree matrix.

Kipf normalization

We down-weight the information coming from nodes that have a large number of neighbors. This is because these nodes aggregate too many information, and we would loose unique information:

$$agg[n] = \sum_{m \in ne[n]} \frac{h_m}{\sqrt{|ne[n]||ne[m]|}}$$

Max pooling aggregation

We only take the maximum value among the neighbors:

$$agg[n] = \max_{m \in ne[n]} [h_m]$$

Aggregation by attention

In Graph Attention Networks, the contribution of each node is weighted differently.

First we apply a linear transform to the current node embeddings: $H'_k = \beta 1^t + \Omega_k H$.

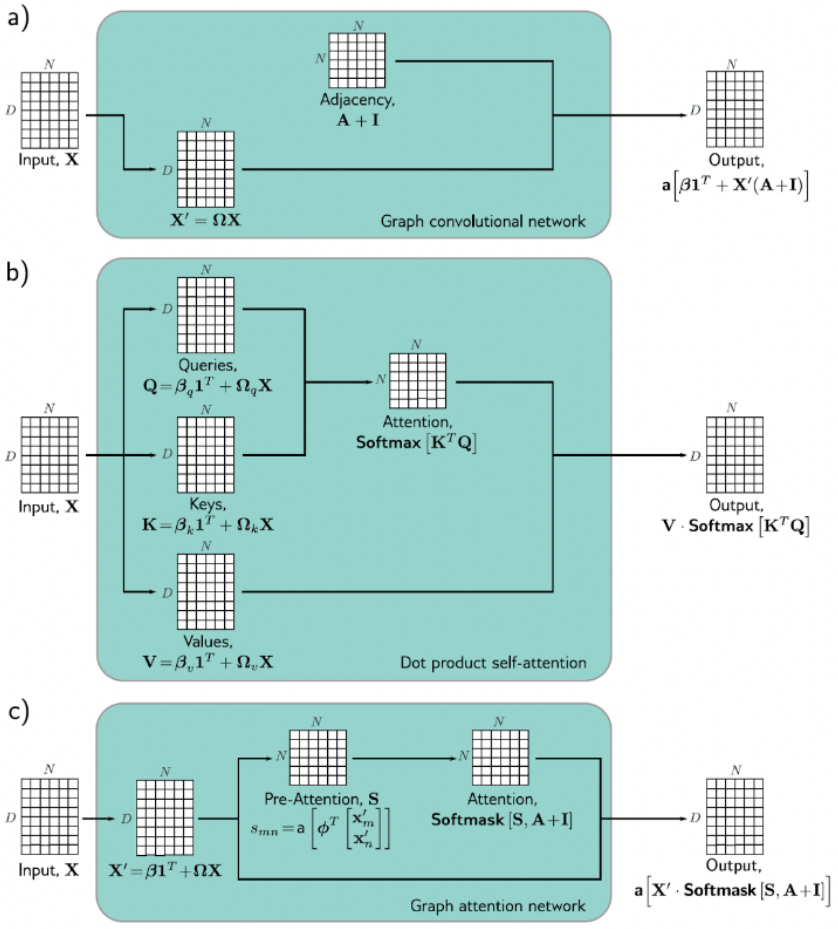
Then we compute the similarity s_{nm} between each n and m pair by concatenating their embedidngs h_n and h_m and taking the dot product with a column vector ϕ_k of learned parameters. Finally we apply an activation function to the result:

$$s_{nm} = a[\phi_k^T \begin{bmatrix} h'_n \\ h'_m \end{bmatrix}]$$

We store all the similarities in an $N \times N$ similarity matrix S, where each entry represents the similarity between each node and each other node.

It's important to notice that we are only interested in the attention scores for the neighborhood of each node. We use a softmask function, which allows to compute the softmax of only the neighbors by masking out the other nodes:

$$H_{k+1} = a[H'_k \cdot \text{softmask}[S, A + I]]$$



Graph convolutional framework

The graph convolutional framework can be summarized in two steps:

1. Initialize node embeddings using the input features: $H_0 = X \in \mathbb{R}^{|V| \times D}$
2. Update node embeddings using multiple graph convolutions.

Graph convolutions

We specify the shape of the convolution, which will regulate which nodes will interact with which other nodes. We can instantiate multiple filters (like in CNN

kernels). For each layer l we will have K matrices $(\Theta_1^l, ..., \Theta_K^l)$.

At each layer, hidden representations H' will be convolved with every patch using the convolution filter weights:

$$m_k^{l+1} = f_k(W, H^l)H^l\Theta_k^l \quad \text{for} \quad 1 \leq k \leq K$$

Finally we will merge the representation from the multiple filters into a single one:

$$H^{l+1} = h(m_1^{l+1}, ..., m_K^{l+1})$$

h can be any aggregation strategy we choose.

11 - Model compression

Introduction

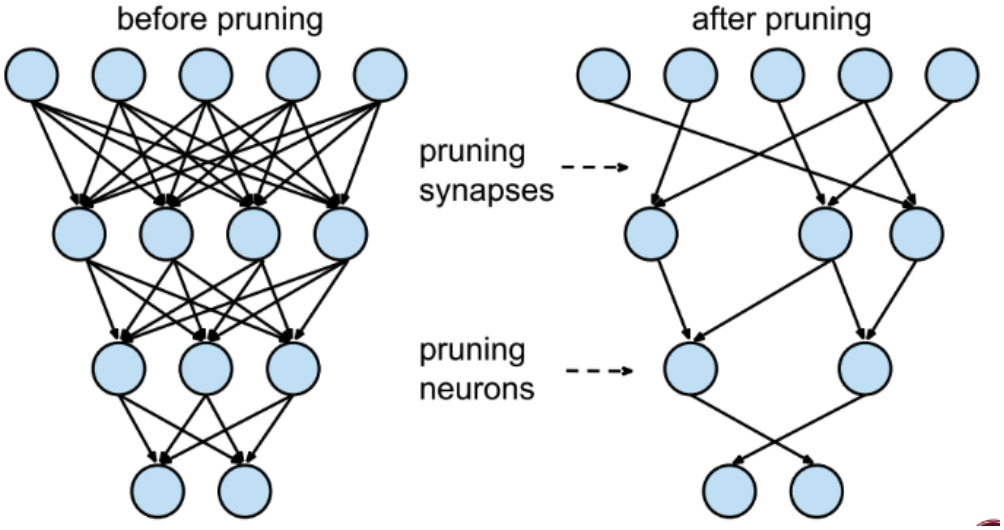
In the modern landscape of Deep Learning, we have an increasingly amount of large models (1B+) being trained and deployed. Although reaching generally greater performances this models are incredibly expensive to train (thousands to millions of dollars) and have a non negligible carbon footprint.

Moreover, these models require a lot of storage and computational power to be deployed, meaning that it is difficult to serve them in consumer devices like phones, cars etc.

Model compression is a field of Deep Learning that tries to address the aforementioned problems by means of a series of techniques.

Pruning neural networks

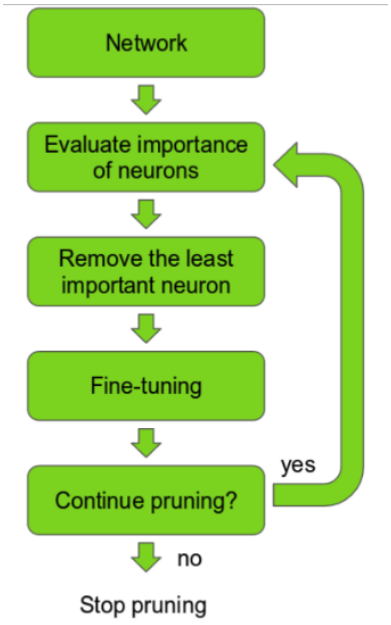
The operation of pruning neural networks essentially consists in eliminating parts of neurons or connections from an initial neural network:



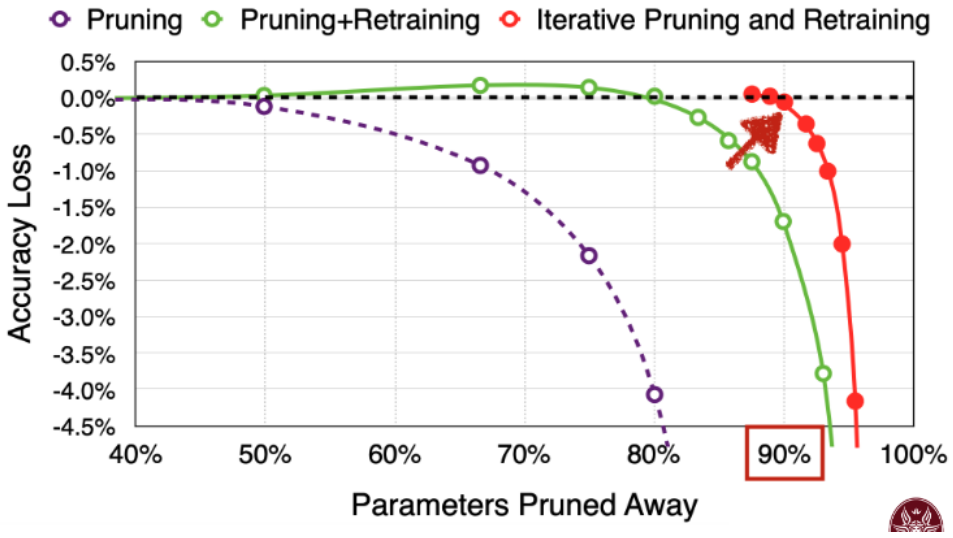
We will see that this can lead to very interesting results, but we have to keep in mind that by merely remove connections, we can save on storage but we might not save on training time as well. This is because GPUs are really good at matrix multiplication, but by pruning we are creating sparse matrices, and operations with these matrices cannot usually be parallelized.

Usually a pruning cycle works like this: we start from the original network and evaluate the importance of neurons. In order to make this evaluation, we have to establish a criterion. A very simple, yet effective one, is to prune weights below a certain threshold. We do this because they represent the weights that contribute the less in the final sum. During training we can think that the network learns to detect connections that are not useful, and will assign them smaller values. This is why we use this criterion.

After evaluating the importance, we proceed with pruning and fine-tune the new pruned network. Finally, we choose whether continue pruning or not.



Studies have shown that with pruning+retraining we can shrink the model by 90% and still achieve acceptable results.



Lottery ticket hypothesis

The lottery ticket hypothesis states the following:

A randomly initialized dense neural network contains a sub-network that is initialized such that -when trained in isolation- it can match the test accuracy of the original network for at most the same number of iterations.

A first important thing to notice is that this theorem implies that the random weight initialization is done once and for all. This means that the sub-network will be initialized with the same weights as the original one. The result is that we are picking from a fixed number of tickets.

The expected results is that we can find a sub-models that generalizes as well as the original one.

In order to prune weights, we do not remove connections, but mask the weights using a mask matrix of zeroes and ones: $m \in \{0,1\}^{|\theta|}$. The sub network will be $f(x; m \odot \theta)$.

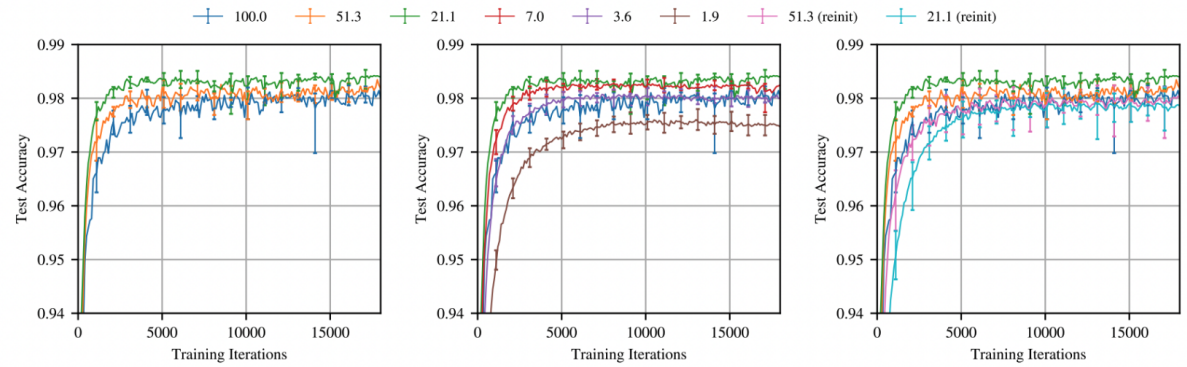
When optimizing the original network with SGD, we obtain minimum validation loss l at iteration j with test accuracy a.

When optimizing the masked network on the same training dataset with SGD, we reach minimum validation loss l_0 at iteration j_0 and test accuracy a_0.

The lottery ticket hypothesis states that $\exists m$ such that $j_0 \leq j$ (commesurate training time), $a_0 \geq a$ (commesurate accuracy) and $||m_0|| << |\theta|$ (fewer parameters).

So we are searching for the pruning matrix that satisfies the lottery ticket when applied to the same initial parameters as the original model. We still prune by

weight magnitude.



Weight quantization

Nowadays, the most effective model compression technique is weight quantization. This means to store the weights utilizing lower precision data types. For example passing from FP32 to FP16 or even INT4. This allows to both accelerate training and scale down storage reuired. It had been shown that quantization influences minimally the performance of the model.

Quantization aware training

We can utilize floating point values for weights during training, but during inference we are only allowed to utilize integers.

12 - Meta Learning

Introduction

The field of Meta Learning draws inspiration on how the human brain work. In order to understand a new task, often we as humans need just a few examples to be able to learn it. This happens because we inherit and store an intrinsic experience within the human brain, that adapts us to quickly adapt.

Can we design a model that learns to learn in a similar way?

While classical ML is focused on generalizing over new data, Meta Learning is interested in generalizing over new tasks. For example a good meta learning model that was trained on classifying cats, should be able also to classify dogs without these being in the train dataset.

In this setting, meta learning is fundamentally different from fine-tuning because we don't modify the model's weights after training.

Instead of sampling data points, in meta learning we sample datasets (tasks). Our goal will be to optimize the parameters over these tasks:

$$\theta^* = \arg\min_\theta E_{D \approx p(D)}[L_\theta(D)]$$

Few-shot learning

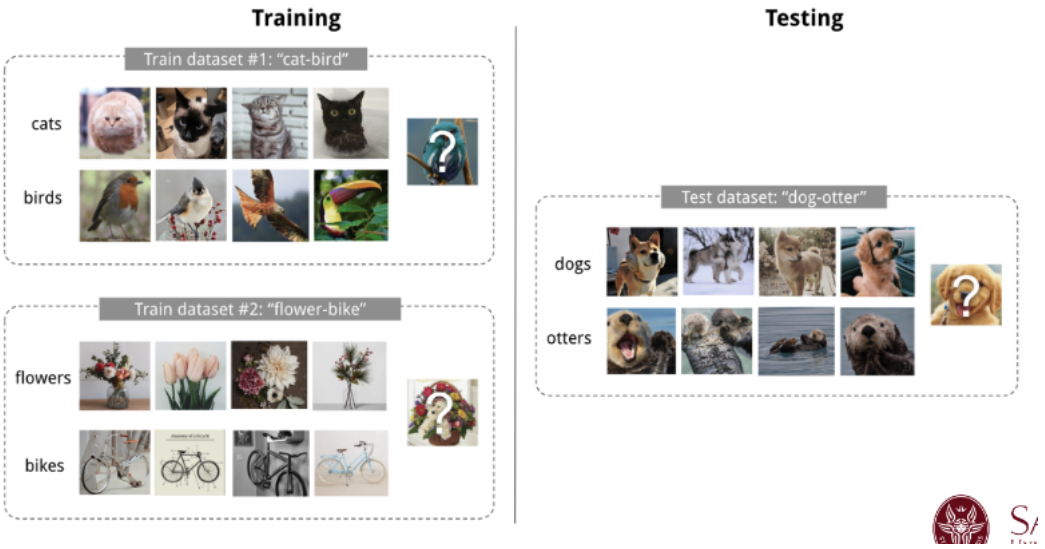
Few-shot learning is a particular instantiation of meta learning in the field of supervised learning.

In this setting, a dataset is split in two: we have a support set S and a prediction set B such that $D =< S, B >$. The role of the support set is to support the decision that the model has to make by providing some examples. This means

that when deciding over a new input, the model will also look at the data in the support set. The prediction set instead is made of data that the model should guess.

We will have K examples and N classes, and we talk about K-shot N-class classification task.

The test set will follow the same structure, but on a totally different task.



In meta learning we first sample a task (dataset), form this task we sample a support and prediction set and compute the probability of y given x using the support set:

$$\theta = \arg\max_{\theta} E_{L \subset \mathcal{L}}[E_{S^L \subset D, B^L \subset D}[\sum_{(x,y) \in B^L} P_\theta(x, y, S^L)]]$$

Metric based Meta Learning

The core idea of the metric based approach is very similar to how KNN works: we will use data from our support set to compare them to a new sample using a learned similarity function.

The predicted probability over a set of known labels y is a weighted sum of support set samples:

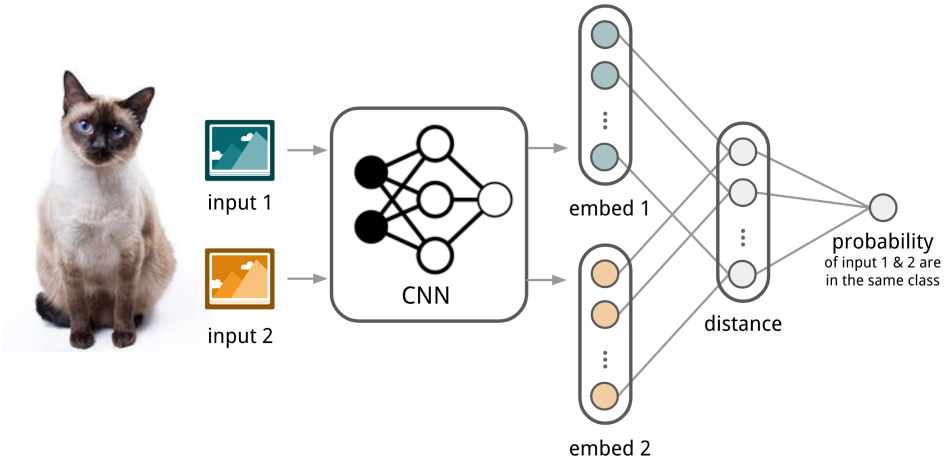
$$P_\theta(y|x, S) = \sum_{(x,y_i) \in S} k_\theta(x, x_i) y_i$$

The kernel function k_θ is responsible to assign the weights. To learn a good kernel is crucial to properly learn the similarity between data samples. This leads to the drawback that we might need many datapoints to learn a successful similarity metric.

Convolutional siamese neural network

We have seen that siamese neural networks are a good way to learn similarities. We can repurpose this concept for meta learning in the following way:

During training, the siamese networks are trained to tell whether two input images belong to the same class. At test time, the siamese network will process all the image pairs between the current input and every image in the support set, compute the distances and assign a probability for each class:



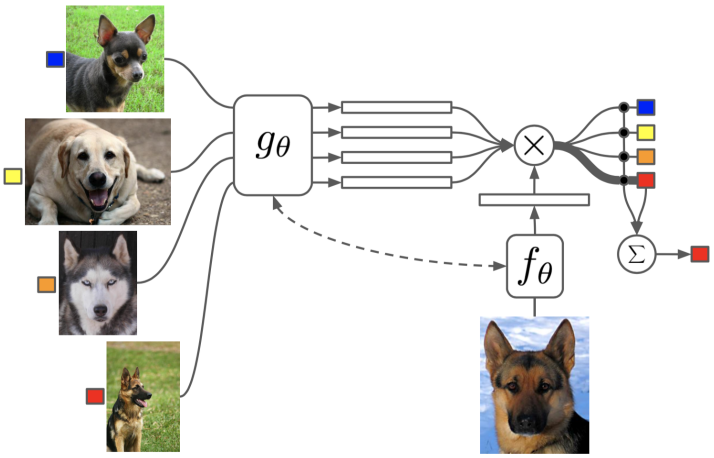
Given a support set S and a test image x, the final predicted class will be:

$$\hat{c}_s(x) = c(\arg\max_{x_i \in S} P(x, x_i))$$

Matching networks

In matching networks we learn a classifier that takes as input the whole support set and a new datapoint, and should outputs a probability distribution over the labels for that new datapoint. The core concept here is that we move from the single comparison with each sample to a system that should learn how to relate all the information in the support set to the input, such that we can dynamically choose which feature to relate for each sample.

Notice: we use two different functions g_θ and f_θ to embed the support set and the new sample.



This mechanism is basically very similar to attention, in fact we say that we learn an attention kernel.

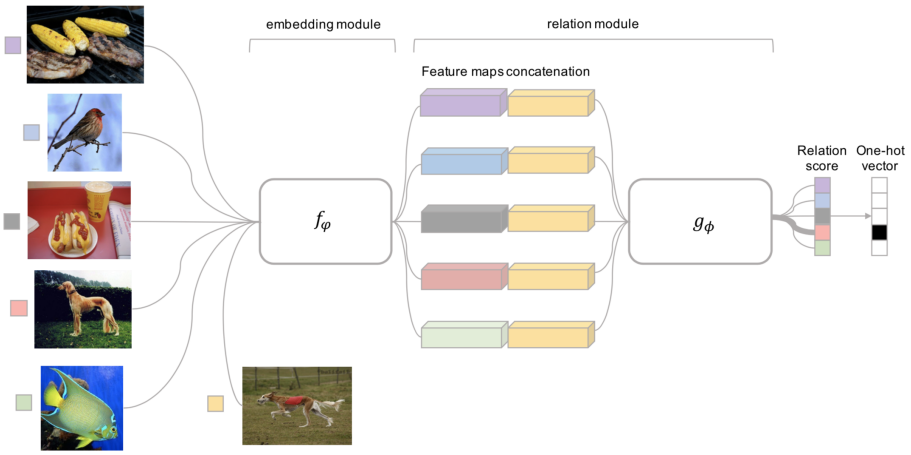
$$c_s(x) = P(y|x, S) = \sum_{i=1}^{k} a(x, x_i) y_i$$

Where:

$$a(x, x_i) = \frac{\exp(cos(f(x), g(x_i)))}{\sum_{j=1}^{k} \exp(cos(f(x), g(x_i))}$$

Relation network

In relation networks we use a single encoder f_φ to obtain the embeddings for both the support set and the new datapoint. After this, we concatenate the same embedding of the new datapoint to each embedding of the support set and feed this to a function g_ϕ that will output a relation score. The relation score will be treated as a similarity that will be used to compute a probability over the labels.



The advantage of this network is that we can train end-to-end both the embedding module and the relation module. In this way we will be able to learn similarity measures that better exploit the input features.

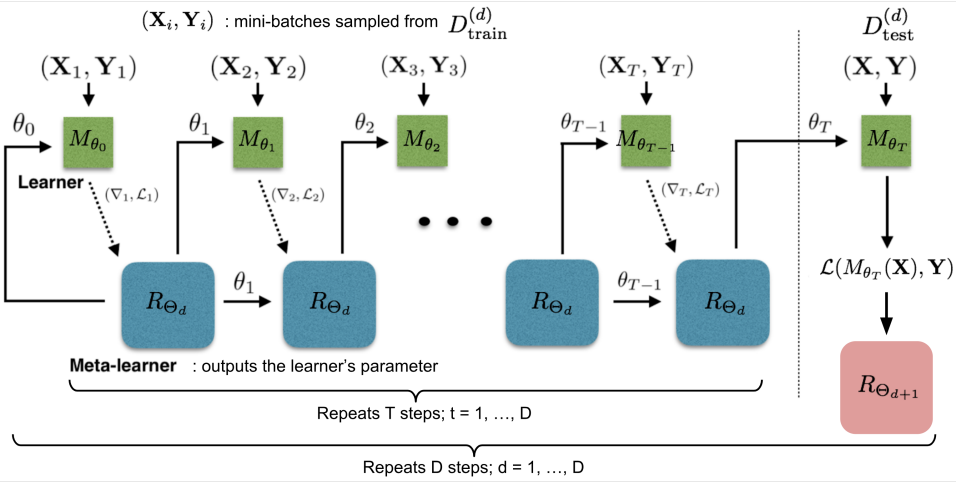
Optimization based methods

Optimization based methods try to learn new methods of optimization to train our network.

In the seminal paper, the goal was to learn a new gradient descent mechanism that is learned on the various tasks. The idea is to learn a function that takes as input the gradient of the model and outputs the update for the new weights:

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)$$

We learn the parameters ϕ of the function g by gradient descent. The way this system is implemented is by using an LSTM that take as input a sequence of previous gradients and predicts the next set of parameters:



Model Agnostic Meta Learner (MAML)

Until now we have seen that each update step of the model is based on its performances on a single task at a time. The core concept of MAML is to optimize the model with respect to a number of different tasks at each update steps. We do this by computing different losses at a time for different batches. Then this losses are utilized to compute updates, and the final update to the model will be done as a sum of these updates.

In practice this method will make our model more robust to different tasks.