Lecture #1
Course Introduction and ML Refresher

This Lecture is a super-condensed refresher of Machine Learning, Neural Networks and Deep Learning.

1 Fundamentals of Machine Learning

Machine Learning is a field of artificial intelligence dealing with models and methods that allow computer to learn from data. Reinforcement Learning is perceived almost as a different entity, it sits in the crossway of a lot of different areas such as Economics, Psycology, Computer Science and so on.

If we want to look at it in the classic framework of machine learning, we have three main type of learning:

1. *Supervised Learning* We learn a map from an input, x, to another one, y. I need to give to the system an input and an expected behaviour.

2. *Unsupervised Learning* We want to give meaning to data: discover partitioning, dependencies or relationship, se only feed the system with an input x.

3. *Reinforcement Learning* Rather than learning to implement a function from x to y we're looking for a behaviour, a policy. Learning so we can see the direct effect of the choice, with a partial view on the world. The data model for RL is an information about the current state s, i operate an action a, i receive as a result of that action a reward r, which doesn't tell me in the action was correct but only an effect of the action.

1.1 Information Representation

ML is much about understanding data, so we must be aware of the types of data we must deal with. The data we have to deal with is usually differentiate in 3 main classes.

Vectorial Data is the classical representation. the i-th input sample x_i is a D-dimensional numerical vector which describes an individual of our world of

interest. The single dimensions d are called features and numerically represent an attribute of the individual. If x_i describes a patient, $x_i(d)$ can be his age.

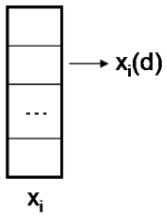


Figure 1: Vectorial data

Images are a minor generalization of the vectorial data, but still a really important one. Images are matrices of pixel intensity for every channel such as RGB etc

Structured Data is information comprising atomic elements that needs to be interpreted in the context of the surrounding elements. It can be a sequence, a tree, a graph and so on.

In this course we will be dealing with mainly sequential data. Note that each element of the sequence can be a vector and we're gonna use it both as input and as output.

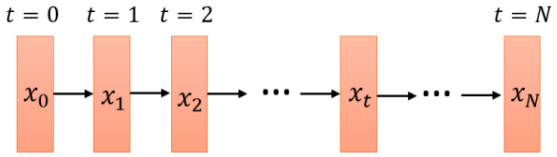


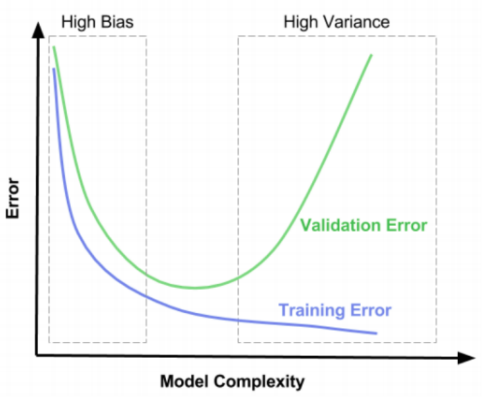
Figure 2: Sequential data

1.2 Fundamental Concepts

ML model is a computational model which encapsulate different *adaptive parameters* that can be changed automatically thanks to an algorithmic procedure that is gonna modify them in such a way that the ML model behave accordingly to some form of performance function.
A ML model also has *hyperparameters* and it's programmer responsability to set them. They may be something like the number of hidden layer, or the number of neuron in each layer, what is the learning rate and so on.

Training is the process through which model parameters are modified to adapt to training data by optimizing a cost function.

Generalization is what where seeking. We want our model to scale prediction to unseen data. That's the point of having a Validation Error. The following chart evaluate the error on the training data and the fresh one we're using for testing. If that's not happening there's something really bad in the data. The



less complexity, the more we generalize, but if we are little complex the overall error will be high. That's the bias-variance dilemma.

Overfitting is what happen when we get more and more complex: we behave excellently on training data while being poor on tests.

1.3 Model Selection

Is the set of techniques to measure generalization avoiding overfitting and reducing the effect of model bias. We can divide it in different phases:

1. Separate training phase and its data from the choice of model configuration (including hyperparameters), from model generalization assessment. Be aware that testing on validation is not real testing, and we must have different datas!

2. Iterate the process changing data to obtain robust performance estimates. There is many way to divide datas and one of them is the *k-fold validation* that take all your data and spit it in k partitions.

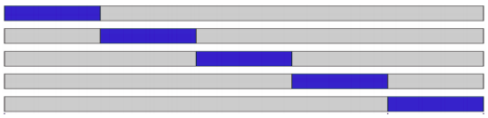


Figure 3: k-fold validation

Let's see an example in Figure 3.

We're splitting the data in 5 partitions. We iterate the process the process of training and validation by taking the first 4 partition and using the gray part as training data and the blue one as validation data. Then pick 4 different partition and do it again, in the end you can mediate the 5 results you got to obtain a more accurate validation error.
If you want to extract also the testing data you have to do a nested k-fold partition in which you have an external k-fold separtion extracting the training data in which you're gonna do another k-fold validation as described.

1.4 Model Definitions

Whenever we're dealing with probabilities in ML we'll be talking about at least 2 types of random variable:

- Observable random variables X, associated with data. It's usually the input variable.
- Hidden random variables Z, which are there to describe the model and needed to explain relationships in the data but we don't have data to observe

In addiction to that a general probabilistic learning model comprises also

- Model Parameters θ

On those terms we'll be resoning in terms of conditional probabilities given θ.

$$\overset{\text{Posterior}}{P(Z|X,\theta)} = \frac{\overset{\text{Likelihood}}{P(X|Z,\theta)}\overset{\text{Prior}}{P(Z|\theta)}}{\underset{\text{Marginal}}{P(X|\theta)}} = \frac{P(X|Z,\theta)P(Z|\theta)}{\int P(X|Z,\theta)P(Z|\theta)dz} \tag{1}$$

That's the ML interpretation of the Bayes Rule. We'll be using the name written in the equation, so we have:

- Posterior, it refer to the probability distribution of a hidden variable given observable data and model parameters
- Likelihood, it refers to the probability of data given a specific hipotesis on hidden random variables

- Prior, it refers to the prior knowledge about your hidden variables
- Marginal, is the probability of data, which we don't know because it's so complex (that's the assumption on ML) and i cannot design a mechanistic solution and i have to infer

The Marginal can be expanded by marginalization, and still it's not known how to analitically solve the integral.

1.5 Maximum Likelihood Learning

When you have a probabilistic model and maybe a Neural Network you will often be performing a Maximum Likelihood Learning:

$$\theta* = \underset{\theta}{argmax} P(X|\theta) = \underset{\theta}{argmax} \int P(X|Z,\theta)P(Z|\theta)dz \tag{2}$$

Since we don't know the Marginal we expand it by introducing some random variables and we're gonna find the parameters of the model that maximizing, using the *Expectation-Maximization Algorithm*. It works in 2 steps:

1. **E** It fixes the current model parameters and then it estimates a Q function computing the expectation (or, in other words, it makes an hypotesis) on the unobservable random variable.

$$Q(\theta|\theta^k) = \mathbb{E}_{Z|X,\theta^k}[\log P(X,Z|\theta^k)] \tag{3}$$

2. **M** Given the current posterior expectation (so the hypotesis we have estimated) it uses it to find new values for the model parameters.

$$\theta^{k+1} = \underset{\theta}{argmax} Q(\theta|\theta^k) \tag{4}$$

And you repeat it.

1.6 Evidence Lower Bound

It's something that may be helpful to maximizing the function $logP(X|\theta$ because it may be difficult and the integral it's too complex. So ELBO gives us a lower bound that it's more treatable and easier to maximize.

$$logP(X|\theta) \geq \underbrace{\mathbb{E}_{Q(Z|\lambda)}[logP(X,Z|\theta)]}_{\text{Expectation of complete likelihood}} \underbrace{- \mathbb{E}_{Q(Z|\lambda}[logQ(Z|\lambda)]}_{\text{Entropy}} = \underbrace{\mathcal{L}(X,\theta,\lambda)}_{\text{ELBO}} \tag{5}$$

It does so by introducing an auxiliary function Q which is parameter function simple enough to be computed, in place of something i cannot handle. By learning you will be optimizing your model both respect to the X parameters and the variational λ parameter.

What happens is that if you maximize the lower bound you will be also maximizing the *log*. There is another results that tells us that the equality

holds whenever you pick your Q to be the Posterior, so if $Q(Z|\lambda) = P(Z|\theta)$. If you're able to compute the posterior you better go for it. If you cannot compute it, change it with a generic function as similar as possibile to the posterior or alternatively you push, by some optimization algorithm, those ELBO as high as you can.

1.7 Sampling Approximations

Another way by which you can deal with things you cannot compute is approximating through sampling. You draw examples of your process from an empirical distribution and then you estimate the actual probability through the empirical distribution, by observation of the sample you have drawn.
If you go to the limit you will be getting the actual distribution:

$$\lim_{L\to\infty} \frac{1}{L} \sum_{l=1}^{L} \mathbb{I}[x^l = i] = p(x = i) \tag{6}$$

There are several ways in which you can do something like this such as

1. Ancestral sampling
2. Gibbs sampling
3. Markov Chain Monte Carlo Methods
4. Importance sampling (particle filtering)

But we're not gonna explain them now.

2 Fundamentals of Neural Network

Neural network architectural design influences deeply

- The type of tasks it can solve
- The type of data it can handle
- The quality of generalization of its results

Architectural choices (those are the hyperparameters)

- Topology and weight sharing
- Activation functions
- Regularization strategies
- Loss functions

2.1 Logistic Neuron

The main ingredient of a neural network is the Neuron (Figure 4). The Neuron

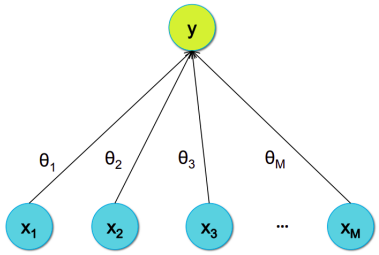


Figure 4: Neuron Structure

is just performing the operation

$$y = h_{\boldsymbol{\theta}}((\boldsymbol{x})) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x}) \tag{7}$$

It's just a weighted submission of the inputs, where σ introduce some sort of non linearity. The θ_i are the parameters updated by learning.
The activation functions may be very different and they may be step functions, identity functions, hyperbolic tangent, ReLU, sigmoid and so on.

2.2 Multilayer Perceptron

That's the main structure. This NN is learning a map between input and output by projecting the inputs in to a space defined by the activations of the hidden layer. You can have also multiple outputs, and you may often find a Softmax output map, a layer in which the output compete with each other and the results are normalized.

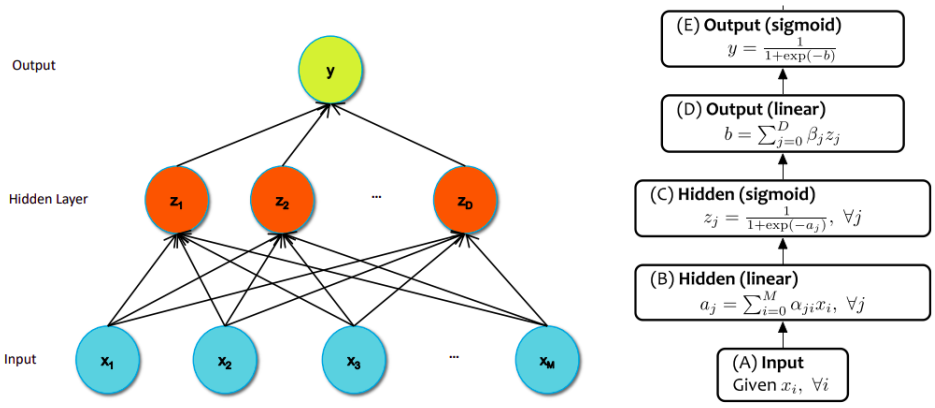


Figure 5: Multilayer Structure

2.3 Training NNs

NNs are trained by gradient descent meaning that you compute the error on the output layer and then create a new value for the weights correcting the old values in the opposite direction of the gradient of the loss function:

$$w_i^{'} = w_i - \alpha \frac{\partial I}{\partial w_i} \tag{8}$$

In order to do that in a complex NN with multiple layer you can compute the gradient only on the output layer and then you will have to backpropagate by the chain rule, to compute the contribution to the error from every layer.

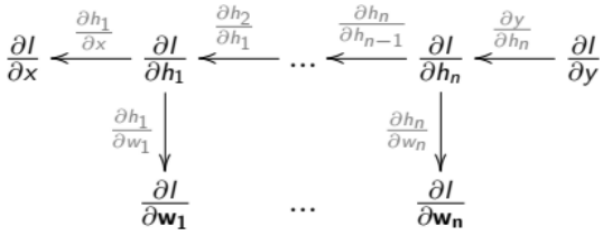


Figure 6: Back Propagation scheme

2.4 Loss Functions

There are plenty of possibilities on choosing the loss functions but usually we have two main possibilities based on the problem you're assessing.

Regression A problem where you predict a real-value quantity.

1. Output Layer: One node with a linear activation unit.
2. Loss Function: Quadratic Loss (Mean Squared Error (MSE))

Classification Classify an example as belonging to one of K classes

1. Output Layer: : One node with a sigmoid activation unit ($K = 2$) or K output nodes in a softmax layer ($K > 2$)
2. Loss function: Cross-entropy (i.e. negative log likelihood)

The difference is that usually the Cross Entropy has the steepest walls in gradient direction.
We never have the chance to work with convex function so the main problem is being able to find the global maximum avoiding all the local minima.

2.5 Optimization Algorithms

To solve the previous problem we may use different algorithms, listed here in increasing order of complexity and refining.

Standard Stochastic Gradient Descent (SGD)
Easy and efficient but difficult to pick up the best learning rate. Often used with momentum (exponentially weighted history of previous weights changes)

RMSprop
Adaptive learning rate method (reduces it using a moving average of the squared gradient). Fastens convergence by having quicker gradients when necessary.

Adagrad
Like RMSprop with element-wise scaling of the gradient.

ADAM
Like Adagrad but adds an exponentially decaying average of past gradients like momentum. Here we have different learning rate for different parameters.

2.6 Learning fashions

There are several ways, and we can divide them in three main categories:

Sequential Mode basically you update the weights after each pattern is presented to your network, also called as online, stochastic or per pattern. High speed of convergence but coupled with instability.

Batch Mode you save each computed error and in the end you average your errors, so in other words you average your gradients, and then you update the parameters, aslo called off-line or per-epoch. High stability but easier to get trapped in a local minima.

Minibatch mode it blends the previous two. It splits the data in mini batch and averages the gradient in it.

2.7 Convergence Criteria

We need to decide when to stop. To do so we define the *epoch* as one swipe over your training data.
We may use different criteria such as Euclidean norm of the gradient vector reaches a sufficiently small value, absolute rate of change in the average squared error per epoch is sufficiently small or stop when generalization performance reaches a peak.

Let's say we're evaluating the validation error and we want to stop when it starts to rise. In order to avoid an *early stopping* we better save the best result and keep training for some time, because maybe we're stuck in a local minima. We have to define the patience parameter to decide for how long we want to keep training after we reached a minimum.

Another good idea is the *regularization*. It constrains the learning model to avoid overfitting and help improving generalization. It adds penalization terms to the loss function that punish the model for excessive use of resources limiting the amount of weights that is used to learn a task and the total activation of neurons in the network.

$$J^{'} = J(y, y^{*}) + \lambda R(\cdot) \tag{9}$$

where $R(W_\theta)$ is a penalty on parameters and $R(Z)$ the penalty on activations. Common penalty terms are usually 1-norm or 2-norm e.g. $R(W_\theta) = ||W_\theta||_2^2$.
λ is again an hyperparameters.

In some cases we may use *dropouts regularization* it randomly disconnect units from the network during training. We're faking the network to be smaller while training so the neurons will try to learn things different ways.

3 Fundamental Deep Learning Models

A deepNN is a NN with different hidden layers. We saw the first deepNN only in 2005, that's because Backpropagation through many layers has numerical problems that makes learning not straightforward (Gradient Vanish/Esplosion).
We say that DeepNN is more than just a NN with a lot of hidden layers. It's about representation. The hidden layer of the layer learn a suitable and informative representation of the input data that is excellent for the task in exam. So

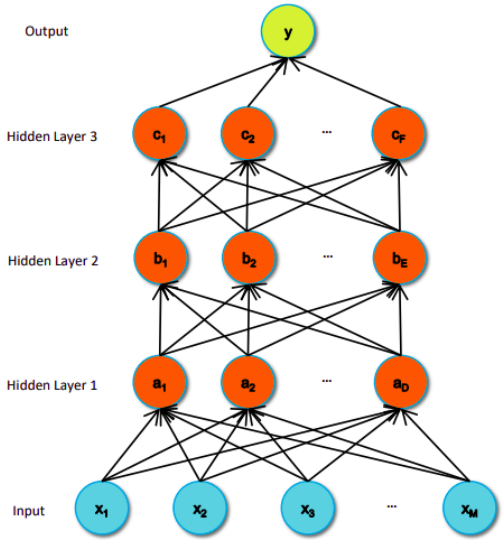


Figure 7: DeepNN structure

by DeepLearning we can learn abstract representation of images starting from raw pixels as input.

3.1 Autoencoders

There are a couple of notable models, and one of them is the autoencorders.

Basic Autoencoders (AE) It's receiving an input x and recostructing it. Only some information are passing through a form of information bottleneck.

I tipically use a classic loss function with a penalization on the activations or in high difference in results when low difference input are given like in (10) and (11):

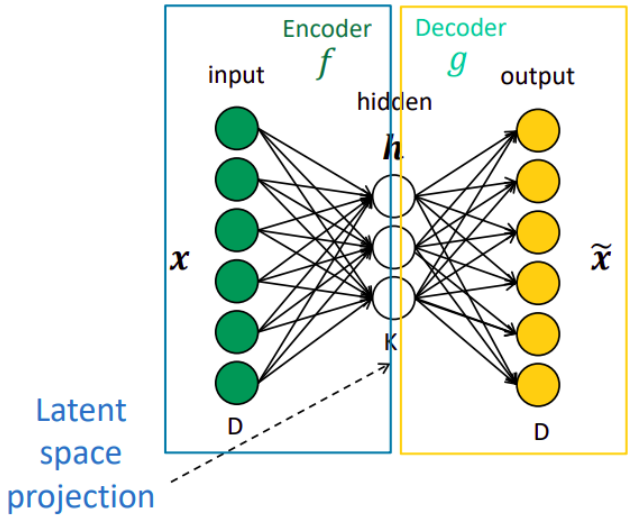


Figure 8: AE structure

$$\Omega(\boldsymbol{h}) = \Omega(f(\boldsymbol{x})) = \sum_j |h_j(\boldsymbol{x})| \tag{10}$$

$$\Omega(\boldsymbol{h}) = \Omega(f(\boldsymbol{x})) = \left|\left| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right|\right|_F^2 \tag{11}$$

Deep Autoencoders They have more hidden layers and can be trained in an incremental way (2 at the times). It extract more complex feature of the data and it may make sense to place on top of that a supervised learning training method.

Variational Autoencoder (VAE) It's a generalization in which instead of learning a deterministc way of mapping x in hidden layer and then a way of map it back, i learn a probabilistic mapping from x to z and viceversa. To obtain this i place before the decoder the reparameterization trick, in which i sample data from a random noise of some sort and adding that to the encoder to generate the stochastic behaviour.

3.2 Convolutional Neural Networks

Another type of NN models are the CNNs, and they're usually used in image processing.

We may have convolutional layers, based on the concept of adaptive convolution operator, which like every neuron is performing a weighted submission of its input by using the same kernel on each pixel, performing always the same operation and using that value as activation of the neurons. Look at computerphile video on youtube to get that if that's not clear.
As a result we'll get a new image that can be passed through a non-linearity σ to obtain a transformed feature map.

We may also have pooling layers, which subsample the image to make sure that the next layer will work on more abstract information. It gets it in different ways, e.g. the max pooling methods store the maximum value stored in a certain area instead of the whole area.

The bigger picture is that you superimpose a number of convolutional and pooling layer until you place some standard hidden layers and then the output.

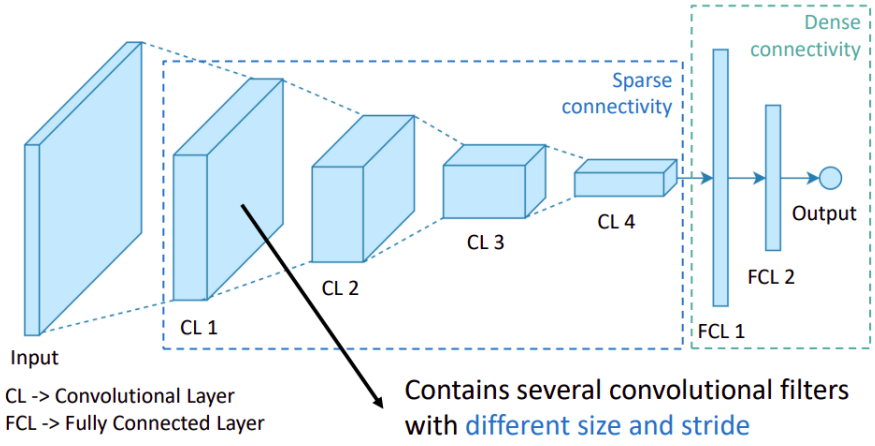


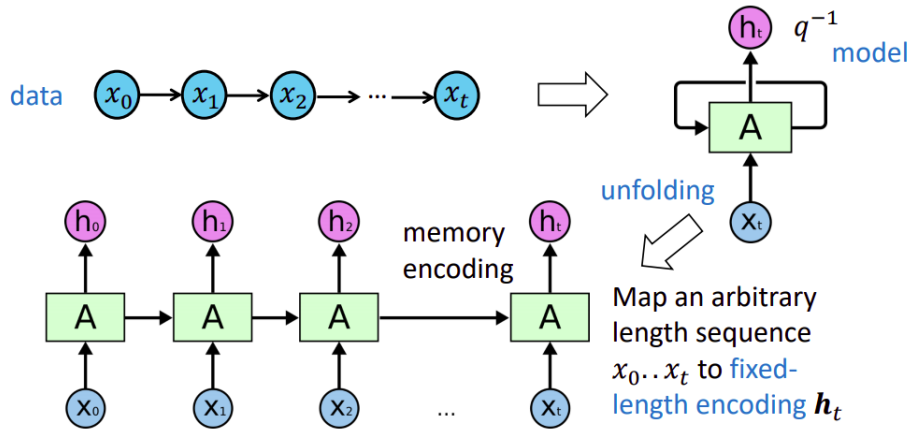
Figure 9: Bigger Picture

A lateral concept we may find in dealing with Deep Networks is *residual blocks*. The residual blocks are connections that bypass the structure to connect to not just the following layer. This simplify the convolutional problems and when backpropagating the gradient flows in full through these bypass connections.

Another typical idea is to use a already existent CNN already written and trained by big companies, cutting it at some points and attaching our new classifier and train it, and maybe with little backpropagation on the last layers. That's called Fine Tuning or Transfer Learning.

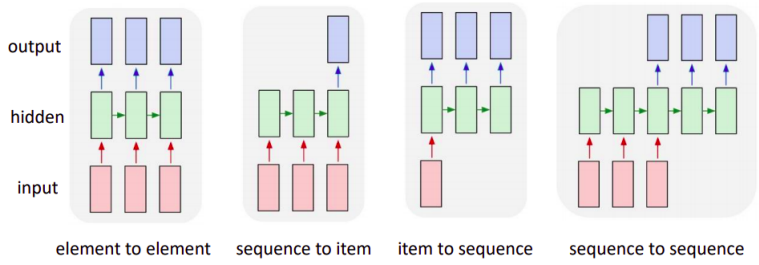
3.3 Recurrent Neural Networks

It's how we deal with sequential data. It's critical in RL because in RL we have to deal with time as variables and that's what sequential data is.
To understand how this all works let's take a look at the whole slide.



Here we can see how data is represented and how the RNN output result is based on the actual input and on the activation values at the step before. In order to make that dependence evident we can *unfold* the RNN. Once you get to the end you have your output, you confront it with the ground trouth, generate an error and try to backpropagate to train the model. "Try" is the actual keyword because it creates a lot of issues with the backpropagation because we have really long chains and we'll have gradient vanishing.

The output structure can be of varius type, i may want a result in real time for each argument of the input data, a single output at the end of a sequence input, a sequence output over a single input and so on as in figure.



A RNN are networks in which hidden neurons compute the activation based on a weighted submission of the inputs and a weighted submission of the ouput of the neuron (or of all the neurons at the same level) at the precedent time-step. This allows the system to have memory encoding what happened in the past.

Whenever the time gap is huge when we backpropagate the problems gets bigger, so there are two main standard tools that complicate the structure of a single neuron adding gates to control acces to the recurrent part of the neuron to prevent the gradient vanishing issues: Long-Short Term Memory and Gated Recurrent Unit.

Even here we can have the Deep ones with different hidden layer, working with more abstract data (e.g. character - word - sentence). There are also bidirectional RNNs when we can use both the past and the future results.

In this case encoding-decoding architectures may be used to translate text from one language to another. *Attention* is a mechanism that allows us not to use the last result of the RNN but to weight each of the output at different time-stamps and it may be useful on those tasks.

A Probability Refresher

Here we usually reason in term of random variables, tipically represented as X and other capital letters. Those are functions (and not variables) that assign value to the outcome of a process, usually represented with the same letter but lower case x.
The probability distribution of X is $P(X)$ is a function that measure the probability of every specific event. It cover all the space of event S.

Clearly we can say that:

$$P(S) = 1 \tag{12}$$

and also, in what we call *marginalization*, wheter X is discrete or continuos:

$$\sum_{x \in S} P(X = x) = 1 \tag{13}$$

$$\int_{-\infty}^{\infty} P(y)dy = 1 \tag{14}$$

Another important concept is *expectation*, and it is defined as it follows:

$$\mathbb{E}[Z] = \sum_{z \in S} z \cdot P(z) \tag{15}$$

And it is easily generalizable for custom f function as:

$$\mathbb{E}[f(Z)] = \sum_{z \in S} f(z) \cdot P(z) \tag{16}$$

What it does is recombining the outcomes based on their probability. That's very important because in RL we often reason in term of expected future rewards.

An additional concpet fundamental on RL is *conditional probability*.
We'll refer to $P(X, Y)$ as the joint probability of both X and Y happening and $P(X|Y = y)$ as the probability of X happening knowing that Y has occured. Those two concept are really different but held together by the following relationship:

$$P(X, Y) = P(X|Y)P(Y) \tag{17}$$

And this one can be generalized to different random variable, and it's known as *chain rule*.

What we'll may be interested in is something like $P(X_1, X_2|Y)$ and a big property related to this is *conditional indipendence*. We say that X_1 and X_2 are conditional independent given Y if:

$$X_1, X_2 \perp Y \quad \Rightarrow \quad P(X_1, X_2|Y) = P(X_1|Y)P(X_2|Y) \tag{18}$$

Take note that this is not true in general and it's different from the *marginal indipendence* or *unconditional* which tells us:

$$X_1, X_2 \quad \Rightarrow \quad P(X_1, X_2) = P(X_1)P(X_2) \tag{19}$$

Often when we're speaking about probabilities we represent probabilistic relationships and models with a graphical scheme called *bayesian network*

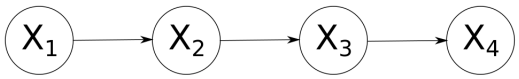


Figure 10: Example of a Bayesian Network

In Figure 4 we can see an example, and under some circumstances that can be a *Markov Chain*, that's what we're interested in for reasons that will become clear going on.

A Markov Chain is a graphical representation that explicitly represent what are the conditional relationships in my model. Every node is representing the joint probability $P(X_1, X_2, X_3, X_4)$ and the presence of the arrow, let's say from X_2 and X_3, it tells me that knowing X_2 the probability of X_3 can be completely defined without consider both X_1 and X_4.
This allows me to write the following relationship in terms of conditional distributions:

$$P(X_1, X_2, X_3, X_4) = P(X_1) \cdot P(X_2|X_1) \cdot P(X_3|X_2) \cdot P(X_4|X_3) \tag{20}$$