UNIVERSITÀ DI PISA

Continual Learning

Notes by Gennaro Daniele Acciaro

Intro

- Chapter 1 : OML - Online Machine Learning

The main reference for this part is the MOA book: [Book – MOA]

Online Learning

What is Online Learning?

We live in a reality where data is generated continuously.
For this reason we don't have a single dataset but a stream of data received over time; often this data can not be store due to the high frequency or the high volume.

Moreover, having a stream, some changes can happen: in this way we could change our goals during the time (e.g. we could start from a Convolutional NN that detects dogs and cats, but during the stream other animals may join).

Many times we have stringent Quality of Service requirements: maybe we want a very fast reply (otherwise the price of bitcoin changes, for example).

Furthermore, data is changing over time: for example, 2020 mobility data is completely different from 2019 due to covid lockdowns.

It's worth noting that even when you have a static dataset you may want to use online algorithms due to memory/computation requirements, for example if we want to perform a PCA of a static dataset, if this dataset is huge we could use the incremental PCA which approximate the classical one.

Batch Learning VS Online Learning

Batch Learning and Online Learning are two different paradigms.

Batch learning involves **accessing all of the data** at once and **using i.i.d.sampling** to train the model. The training and evaluation phases are separated, and we have no stringent computational constraint

Online learning, on the other hand, involves training the model sequentially with access to **one data at a time**. The training and evaluation phases are interleaved and we have stringent computational constraints.

Holdout

The very first method used to evaluate the current model is **Holdout**.
Holdout involves testing the model on a separate set of data at regular intervals; which can be expensive (we perform the full evaluation at each step) and requires updating the test set when the data changes (nonstationarity).

Prequential Evaluation

Another method is called **prequential evaluation** (or interleaved-test-then-train).

This strategy involves **testing each sample in a single stream before training**, making it more efficient and not requiring a separate test set. In other words, this method consists of using each sample to test the model, which means to make a prediction, and then the same sample is used to train the model (partial fit). This way the model is always tested on samples that it hasn't seen yet.

But here a problem is happening: during the first few epochs the model may not perform very well (it improves over time) but the prequential error that we use to measure its performance also takes into account all the mistakes it made in the past.

The **prequential error overestimates the holdout error** of the current model.

To handle this problem, we can use controlled forgetting techniques: we do this with two common methods: **sliding window** and **fading factor**.

- Sliding window only considers the last k elements to compute accuracy, ignoring the earlier errors.

$$L = \sum_{i=0}^{k-1} L_{i-k}(\dots)$$

- Fading factor uses a **running average of the loss**, giving more weight to recent data and gradually forgetting the old data.

$$L_t = \alpha L_{t-1} + (1 - \alpha) L_t(\dots)$$

The reason we need to do this is because we want to evaluate the model's performance on future data, but our loss is computed on old data: especially in situations where the data distribution is constantly changing, we need to make sure that our evaluation accurately reflects the model's current capabilities.

Validation: we can distinguish several types of validation that used prequential evaluation.

1. *K-fold distributed cross-validation*;
   where **each sample is used for <u>testing</u> one classifier** selected randomly, and used for training on all the others. This method has a good use of the data since it uses only 1/k sample for testing, but can be redundant.

2. *K-fold distributed split-validation*;
   where **each sample is used for <u>training</u> one classifier** selected randomly, and for testing in all the other classifiers, resulting in underutilization of data.

3. *K-fold distributed bootstrap-validation*:
   where **each sample is used for training about ⅔ of the classifiers**, with a specific weight for each classifier and the testing phase is performed in all the classifiers.

Delayed evaluation: At the end, there are some contexts where we don't need the target immediately. Actually, in a forecasting scenario, we need them but in supervised learning scenarios, where we need to classify things, we may not need the labels right now. For this reason we can delay the evaluation of our models.
This technique is called "*delayed evaluation*": **it consists of storing the information without labels in a buffer until we have the target, then we train our models.**
The Prequential evaluation also works with sparse targets.

Evaluation Metric - Kappa statistic

Another problem of the OML setting is that **data may be unbalanced**, so the accuracy is not a good metric.
For this reason we introduce the **Kappa Statistic** which compare the accuracy p of a model against the random baseline accuracy:

$$k = \frac{p - p_{random}}{1 - p_{random}}$$

When $k = 1$ we have the perfect classifier while with $k = 0$ we have a random classifier.

The main advantage of using this metric is the computing speed, indeed, it is easier to compute compared to other measures used in imbalance scenarios (e.g. Area Under the ROC, AUROC).

Evaluation Metric - Kappa-Temporal statistic

If the proportion of classes predicted by the model is different from that of the stream, k is not a good estimate, indeed the classifier may be underfitted.

To fix this, we modify the Kappa statistic formula introducing the **persistent classifier** that is a better baseline, indeed, it predicts the next label is the same as the last seen label.

$$k = \frac{p - p_{per}}{1 - p_{per}}$$

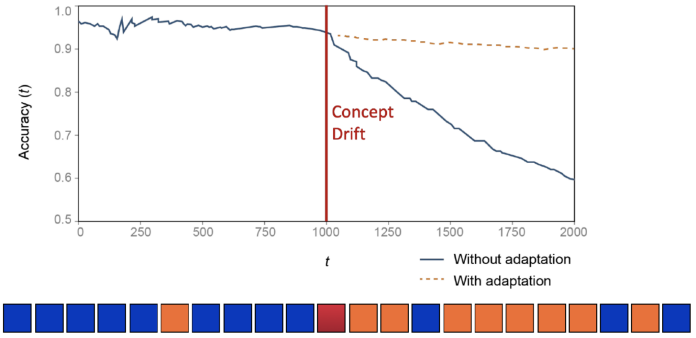
Concept Drifts

Concept Drift: informal definition

Machine Learning models are trained to predict the "normal" behavior of the data, but what happens when the "normal" behavior changes?

Concept drift refers to the phenomenon where the **statistical properties** of a target variable or concept **changes over time**, leading to a mismatch between the model's training distribution and the real-world distribution.
It's worth noting that **we are not talking about noise or outliers**.
A trivial way to deal with Concept Drifts is to retrain the whole model, but obviously this is quite expensive.

This is the usual behavior of the accuracy of a model after a CD appears, without any adaptation it will decrease over time.



Types of Concept Drift

There are different types of concept drift, each one needs a different way to handle it. This is very important because if we do not correctly recognize which CD we are working with, we may implement techniques that will give us bugs.

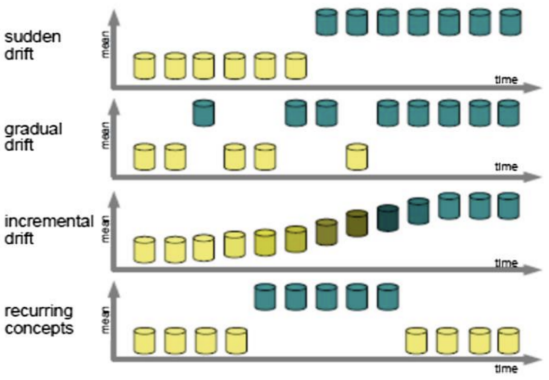


Fig. 3. Types of concept drift [99]

1. **Sudden (or shift)** concept drift
   When the distribution changes after a long period of time when there has been no change.

2. **Gradual (or incremental)** concept drift occurs when, for a long time, the distribution experiences at each time step a tiny, barely noticeable change, but these accumulated changes become significant over time.

3. **Global / partial** concept drift, depending on whether it affects all of the item space or just a part of it.

4. **Recurrent concepts** concept drift (*seasonality*), this occurs when distributions that have appeared in the past tend to reappear later.

Concept Drift vs Anomaly Detection

Concept drift refers to the situation where the statistical properties of the data being analyzed change over time, making it necessary to update the models or algorithms used to make predictions or decisions.

Anomaly detection, on the other hand, refers to the process of identifying patterns that are significantly different from the expected behavior. Anomalies may indicate errors or outliers in the data, or they may be indicative of important events or changes in the system being analyzed.

In other words, the Concept Drift answers the following question "*is yesterday's model capable of explaining today's data*?", while anomaly detection answers: "*Do these samples conform to the normal ones?*"

Concept Drift: probabilistic definition

Given an input $x_1 \dots x_t$ of the class y, we can apply the Bayes theorem:

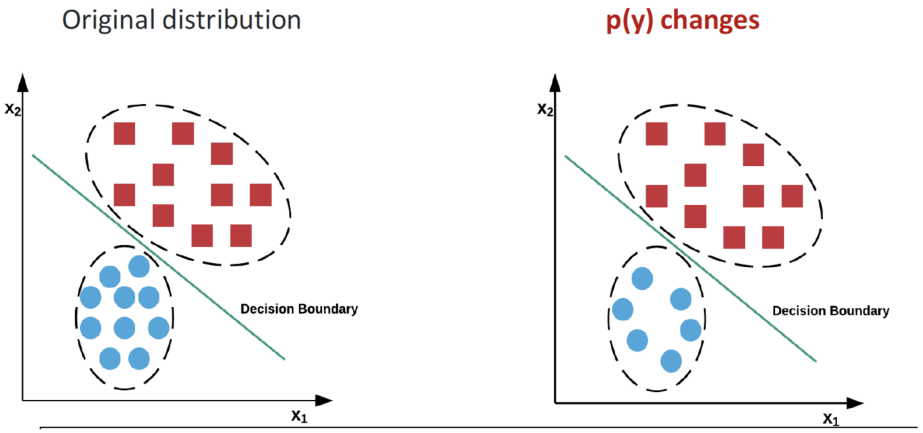
$$P(y|x_t) = \frac{P(y)\,P(x_t|y)}{P(x_t)}$$

All of these components can change with a Concept Drift, and each one can give us a different scenario.

The main information we want to understand is **when** we do need to **retrain**. Let's see these scenarios, using as example linear separable data, the retrain is needed (in this case) when we need to change the decision boundary:

1) when P(y) changes

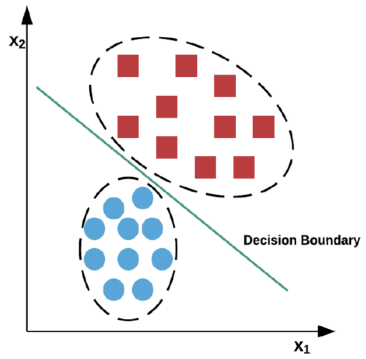
In this case the blue class is less frequent but the decision boundary is the same, so we don't need to retrain.



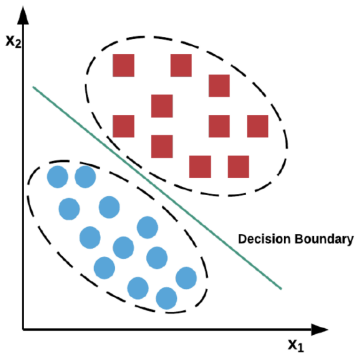
2) when $P(x_t|y)$ changes

In this case the blue class is more frequent but the decision boundary is the same, so we don't need to retrain.

Original distribution



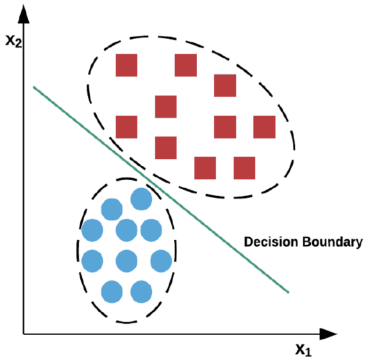
$p(X_t|y)$ changes



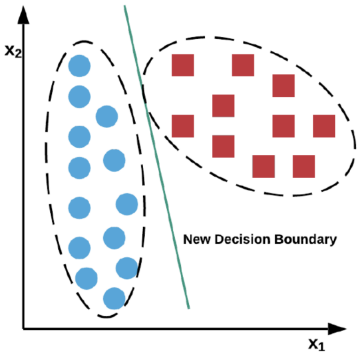
3) when $P(y|x_t)$ changes

In the end, in this case the datas change. In a more abstract sense this means that change the concept of "belonging" to a class, and so the decision boundary change. For example, the dots in the upper left corner used to be graded red, now they are blue.

Original distribution



$p(y|X_t)$ changes



Dataset shifts Nomenclature

We will now present a nomenclature of shifts applied to datasets. However, it is important to note that these definitions can also be applied to real-time data streams. So, we will talk about train/test distributions here, but in OML we will have past/present subsequences.

The notation is the usual: x are the input features, y are the classes and $P(x, y)$ is the joint distribution.

The nomenclature is based on **causal assumptions**:

- $x \rightarrow y$ problems: class label is causally determined by input.
  Example: credit card fraud detection
- $y \rightarrow x$ problems: class label determines input.
  Example: medical diagnosis

So, here we can distinguish four kind of shifts:

1. **Dataset shift**
   happens when the distribution of training data is different from the distribution of test data.
   $$p_{train}(x, y) \neq p_{test}(x, y)$$

2. **Covariate shift**
   this shift happen in $x \rightarrow y$ problems, when the posteriors of train and test are equals but the input distributions are different.
   $$p_{train}(y|x) \neq p_{test}(y|x) \text{ and } p_{train}(x) \neq p_{test}(x)$$

   informally: the input distribution changes, the input → output relationship does not

3. **Prior probability shift**
   this shift happen in $y \rightarrow x$ problems, when the likelihood of train and test are equals but the output distributions are different.
   $$p_{train}(x|y) \neq p_{test}(x|y) \text{ and } p_{train}(y) \neq p_{test}(y)$$

   Informally: output → input relationship is the same but the probability of each class is changed

4. **Concept shift**
   happens when the $x \rightarrow y$ relationship (aka. the concept) change.

   $x \rightarrow y$ problems: $p_{train}(y|x) \neq p_{test}(y|x)$ and $p_{train}(x) = p_{test}(x)$
   $y \rightarrow x$ problems: $p_{train}(x|y) \neq p_{test}(x|y)$ and $p_{train}(y) = p_{test}(y)$

Causes of shifts (Real vs Virtual Drifts)

We can distinguish two main causes of shifts:

1. **Sampling bias**
   In this case, the world is fixed, this means actually that data is not changing. But we are not able to see all the data together so we have to **sample** them.
   We will also call it **virtual drift**.

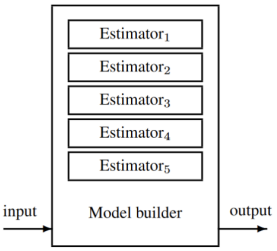
2. **Non-stationary environments**
   Another cause of shifts is when the data are actually changing.
   We will also call it **real drift**.

Families of algorithms to handle Concept Drifts

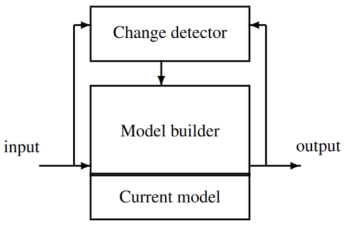
We have **different families** of algorithms that deal with the same Concept Drift problem. All of them share the same requirements: **fast detection** of changes, **robustness** to noise and outliers and a **low computational overhead**.

In particular we can define three families:

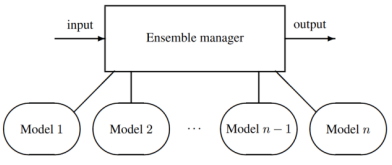
1. **estimators-based**: tracks stream statistics (e.g. the mean) and instead of retraining the model, we use algorithms to update model's statistics.



2. **detector-based**: use some algorithm to detect when the Concept Drift happens and then update the model.



3. **ensembling**: dynamic population of models.
   We need some policies to add models and select the current one. We will see ensembling methods in detail in the next sections.



The main limitations of these methods is that they work only for low-dimensional data, so we can not generalize for high-dimensional settings.

Introduction to Estimators-based CD algorithms

Let's focus on the first family of algorithms: the CD estimation ones. This kind of algorithms compute statistics on the input data, which may change over time. We concentrate on the case in which such a statistic is (or may be rewritten as) an expected value of the current distribution p_t of the data.

$$\theta_t = \mathbb{E}_{p_t(x)}[f(x)]$$

where $f(x)$ is our model.

Part of the problem is that, with the possibility of drift, it is difficult to be sure which past elements of the stream are still reliable as samples of the current distribution, and which are outdated.
We have two methods of finding "old" elements and ignoring them by storing samples in **memory** (with a buffer or windows), called windows-based, or by storing nothing (**memoryless** estimators)

Let us now see an example for both categories:

- Linear estimator over a Sliding Windows for windows-based
- EWMA for memoryless.

Window-based: Linear Estimator Over a Sliding Window

The simpler algorithm uses a window (a buffer); only the elements inside the window are used to estimate θ_t. Let k be the (fixed) windows size, so:

$$\theta_t = \mathbb{E}_{p_t(x)}[f(x)] \approx \frac{1}{k}\sum_{i=1}^{k} f(x_{t-i})$$

The main properties of this algorithm are that:

- it ignores older elements
- the windows size k is a fixed parameter, so we need to find a trade-off.

Memoryless: EWMA

This estimator instead of using a windows, we use an exponential moving average (indeed EWMA stands for Exponential Weighted Moving Average). This means that it updates the estimation of a variable by combining the most recent measurement of a variable with the EWMA of all previous measurements

The moving average A_t at time t is given by:

$$A_t = \alpha f(x_t) + (1 - \alpha)A_{t-1}$$
$$A_1 = x_1$$

Where $\alpha \in [0, 1]$ is an exponential decay factor parameter and it controls forgetting.

Introduction to CD detection algorithms

Let us turn to another family of algorithms that deal with Concept Drift: the detection ones. Our goal now is to provide an alarm when a change is detected.

Now we can distinguish two main methods:

1. We can **monitor the input distribution** through statistical tests
   Algorithms of this method:
   CUSUM, Page-Hinkley.

2. We can **monitor the model's accuracy**.
   Algorithms of this method:
   DDM, EDDM, ADWIN.

Tradeoff between detection and false positives

When we design a detection system, we have to deal with a tradeoff between detecting too early and getting false positives (in this case we waste a lot of time for retraining), or detecting too late and making a lot of errors.
This tradeoff is often a function of the minimum magnitude θ of the changes that the system is designed to detect. In particular we can define several metrics that depend on θ.

1. **Mean Time between False Alarms (MTFA)**, which measures the frequency of false alarms when there is no actual change occurring.

2. **False Alarm Rate (FAR)**, which is defined as the ratio of false alarms to the MTFA :
   $$FAR = 1/MTFA$$

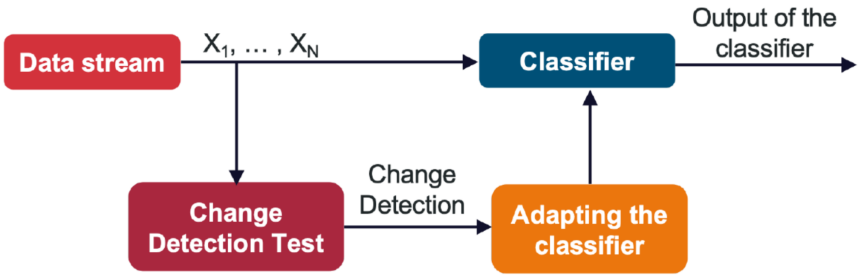
3. **Mean Time to Detection (MTD(θ))** is a measure of the system's capacity to detect and respond to changes when they occur.

4. **Missed Detection Rate (MDR(θ))** measures the probability that the system will fail to generate an alarm when a change has occurred.

5. **Average Run Length (ARL(θ))** is a metric that generalizes both the MTFA and the MTD, providing an estimate of how long the system will take to detect a change after it has occurred.
   In fact, the MTFA can be expressed as ARL(0), while the MTD(θ) is equal to ARL(θ) for $\theta > 0$

CD detection monitoring the input distribution

The first class of CD detectors is the one that works by monitoring only the input distribution. We can define some advantages and disadvantages compared to the class of CD detectors (that works by monitoring the model's accuracy).



Advantage:

1. It can work in an unsupervised scenario
2. We can use simple statistical tests to detect changes

Disadvantages:

1. Difficult to design detectors for multivariate streams or where the underlying distribution is unknown
2. It does not detect changes that do not affect the distribution of observations.

For this class we will see two algorithms: CUSUM and Page-Hinkly.

CUSUM

This algorithm is a CD detector which monitors the input distribution, in particular it gives an alarm when the mean of the input data significantly deviates from its previous value.

Let x_t be the input sequence, μ and σ are the mean and the standard deviation from the input sequence (we can get them a priori or we can estimate them), the $z_t = (x_t - \mu)/\sigma$ is the standardized input and at the end we have two hyperparameter: h, k.

So, this algorithm, works with the relative residual error:

$$g_t = max(0,\ g_{t-1} + z_t - k)$$

Then, when $g_t > h$, the threshold, we have a change; so we will reset g_t, μ and σ.

The first hyperparameter k is needed to balance the weight of z_t otherwise after some steps we will alarm even when there are no changes.

The main advantages are that we don't need a fixed windows, it has a constant time complexity for each item. The guidelines says us that we have to set k to half to the value of the change to be detected (in standard deviation) and set h to $\ln \frac{1}{\delta}$, where δ is an acceptable False Alarm Rate.

Page-Hinkly

The Page-Hinkly algorithm is a variation of the CUSUM algorithm.
As CUSUM, it gives an alarm when the mean of the input data significantly deviates from its previous value. Indeed, as before we have: $g_t = g_{t-1} + z_t - k$, with z_t, k, h as the same.

The main difference here is that we introduce another value $G_t = min\{ g_t, G_{t-1} \}$.

We declare a change when $g_t - G_t > h$.

The purpose of this change is to prevent the algorithm from accumulating too many false alarms over time, which can happen in the original CUSUM algorithm.

CD detection monitoring classification error

Monitoring the input distribution is a simple and effective method but it implies strong assumptions on the change of the data and it's not always possible to use that type of monitoring (e.g. it can not work with timeseries).

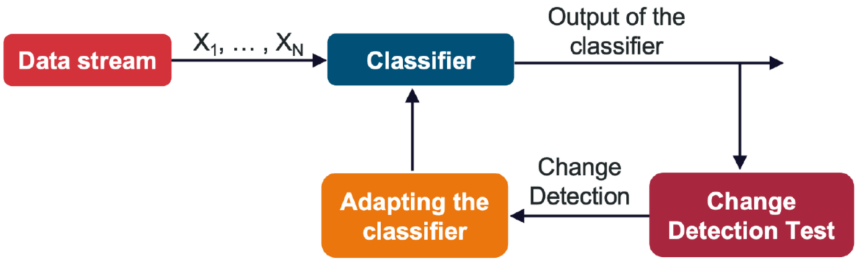
For this reason we introduce algorithms that monitor the classification error (often is the accuracy) during time.

Advantages:

- Straightforward: if the accuracy is low, we know we need to retrain
- Simple: the classification error is a one-dimensional time series even if the input is very complex

Disadvantages:

- We can do it only if we have supervised signals to compute the error, so we do need targets.



Usually, algorithms that belong to this class uses two thresholds: a **warning** threshold and a **drift** threshold. When we reach the warning level we start storing data for an eventual drift. The difference between the two levels causes a latency between the start of the concept drift and the detection time.

For this class we will see three algorithms: DDM, EDDM, ADWIN.

DDM - Drift Detection Method

This algorithm is the simpler one about the CD detection algorithm that monitors the number of errors produced by a model learned on the stream.

The idea on which it is based is the following:
Without drifts, error should decrease (or at least remain stable) over time as more data is used. When, instead, DDM observes that the prediction error increases, it takes this as evidence that change has occurred.
For this reason we can model the distribution of errors over time, in this way we can detect the drift by detecting unexpected values in the error distribution.

The error distribution is modeled in this way:
At instant t, the detection algorithm uses p_t, the error rate of the predictor at time t.
Since the number of errors in a sample of t examples is modeled by a binomial distribution, its standard deviation at time t is given by

$$s_t = \sqrt{p_t(1 - p_t)/t}$$

DDM stores the smallest value p_{min} of the error rates observed up to time t, and the standard deviation s_{min} at that point.

The conditions for entering the warning zone and detecting change are as follows:

- **Warning**

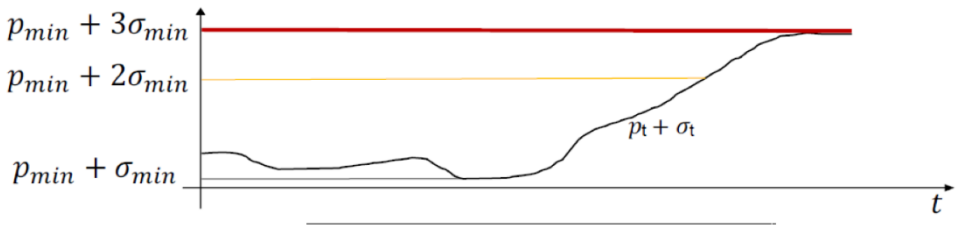
  $$p_t + s_t \geq p_{min} + 2 * s_{min}$$

  From now on, start storing examples in the buffer to prepare for retraining.

- **Drift**

  $$p_t + s_t \geq p_{min} + 3 * s_{min}$$

  Now we will discard the previous model, train a new model using the buffer collected from the warning time and reset p_{min} and s_{min}.



DDM is a simple and general method but it can be slow because p_t is computed over all the examples since the last drift, moreover the space complexity depends on the distance between warning and drift.

EDDM - Early DDM

Using this variation of DDM we try to mitigate the slowness of DDM.
This algorithm considers the distance between two error classification instead of considering only the number or errors.

While the learning method is learning, it will improve the predictions and the distance between two errors will increase.
Instead, when a drift occurs, the distance between two errors will decrease.

So, we compute the average distance between 2 errors and its std, and look for outliers in the tails of the distribution.

ADWIN – ADaptive sliding WINdow

The last method of CD detection algorithms that monitor the accuracy is called ADWIN (ADaptive sliding WINdow).
The idea is that we want a method that compares multiple sliding windows of different lengths. To do this efficiently, **exponential histograms** are used: we will not see this algorithm in detail.

Using exponential histograms we do not have too care about the tradeoff between reacting quickly to changes and having false alarms (as we did for the α parameter in EWMA). On the negative side, it is computationally more costly (in time and memory) than simple methods such as EWMA or CUSUM.

ADWIN efficiently keeps a variable-length window of recent items; such that it holds that there has been no change in the data distribution. This window is further divided into two sub-windows (W_0,W_1) used to determine if a change has happened.

So, the ADWIN algorithm uses:

1. an hyperparameter $\delta \in (0, 1)$ that is a confidence value.
2. a statistical test $T(W_0, W_1, \delta)$ over two sub-windows W_0 and W_1.
   This test compares averages of the two windows and decides if they come from the same distribution. Concept drift is detected if the distribution equality no longer holds, in particular:
   - If W_0 and W_1 are generated from the same distribution, then we declare (with a probability of $1 - \delta$) that there are no changes.
   - If W_0 and W_1 are not generated from the same distribution, then we declare (with a probability of δ) that there is a change.

Pseudo-code:

for each new element
    update the exponential histogram
    runs $b - 1$ statistical tests T using
    - W_0 oldest i buckets of the exponential histogram
    - W_1 newest $b - i$ buckets of the exponential histogram
    drop old buckets when a change is detected

where b is the number of buckets of the exponential histogram

Online Classification Models

We can distinguish two classes of Online Classification Models:

1. **Streaming classification models:** models that takes as input an infinite stream, where we can not save the input data (even if we can keep a small buffer) and moreover we are in a scenario where there are some latency and computational constraints.
   Since we cannot keep the entire stream in memory, we cannot shuffle the data. This is a limitation since shuffling helps us to find better models.

2. **Out-of-Core Methods**: when we have all the data, but the model we want to use is too expensive and can't be trained on the whole dataset at once and we can't keep the whole dataset in RAM, so in this case we only have computational constraints, we don't care about, for example, latency.
   In this case we don't have drifts, because precisely, a single, static dataset that is in the disk. For example, we can sample it randomly, we can shuffle the data, things that we definitely cannot do when we have streaming.

In both methods we can not retrain from scratch at each step but we can update the model using small mini-batches.Algorithms need to be able to take a pretrained model and a small batch of samples as input and return a new model.

- Online algorithms: $\theta_{t-1}, D \rightarrow \theta_t$
- Offline algorithms: $D \rightarrow \theta_t$

Warm start

Many online algorithms are susceptible to larger changes in the first phases of training, for this reason we usually use an initial model pretrained on some static data.

This process is called "Warm start" or "finetuning"; it helps us to avoid bad initializations and makes our online algorithms more stable.

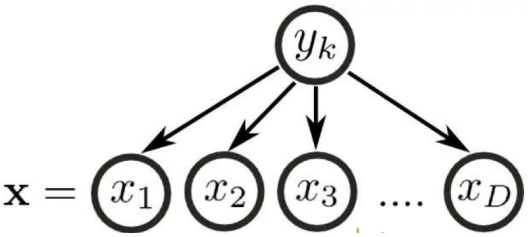
Online Methods

We will see two online methods: Naive Bayes and SGD.

Naive Bayes

This is a Bayesian model, where the classification works performing the $\arg\max_i p(x, y_i)$.

Here, there is a **strong** assumption on the input feature where we assume that they are all **statistically independent between themselves** giving the target.



The graphical model on the right shows the conditional dependency among the data.

Let our dataset be $x = \{x_1, \dots x_D\}$, using the assumption we can factorize this:

$$P(x, y_k) = P(x_1|y_k) * P(x_2|y_k) * \dots * P(x_D|y_k)\, P(y_k) = P(y_k) \prod_{i=1}^{D} P(x_i|y_k)$$

where:

- $P(x, y_k)$: the **joint** probability distribution of a feature vector x and a class label y_k represents the probability that a data point with feature vector x belongs to class y_k.
- $P(x_i|y_k)$: the **conditional** probability distribution of a feature x_i given a class label y_k represents the probability of observing feature x_i given that the data point belongs to class y_k.
- $P(y_k)$: the **prior** probability of a class is the probability of observing that class in the absence of any other information, based on our prior belief or knowledge.

From offline to online scenario

The training phase of this model **estimates the conditional probabilities and the prior**. How can we adapt this to an online model?

Assuming a discrete scenario, in order to estimate the prior $P(y_k)$ we will have two counters: N and N_k. The first one is the number of example in the whole dataset, while the latter one is the number of sample of the class k.

In a stream context, we count how many samples we have seen up to now.

In the offline case, the prior is calculated by performing N_k/N and, actually, even in the online case we have the *same way* to calculate it: N_k/N.

While, for the conditional probabilities, $P(x_i|y_k)$, this is a table of counters where for each possible value of x_i, counts its occurrence in the training set.
Even here, both in offline and online scenario, we have the same way to calculate this table: $M^i_{j,k} / N_k$, where $M^i_{j,k}$ is the counter of the occurrences for the class k on the attribute i for the value j $(= x_i)$.

So, the only change we need to do to adapt the Naive Bayes algorithm to an online scenario is change the prior with the posterior of the previous step, indeed in the offline training we have:

$$\underset{\text{Posterior}}{p(\theta|D)} = \frac{\overset{\text{Likelihood}}{p(D|\theta)} \quad \overset{\text{Prior}}{p(\theta)}}{\underset{\text{Evidence}}{p(D)}}$$

while in the online case:

$$\underset{\text{Posterior}}{p(\theta_t|D)} = \frac{\overset{\text{Likelihood}}{p(D_t|\theta_t)} \quad \overset{\text{Precedent posterior}}{p(\theta_{t-1}|D_{t-1})}}{\underset{\text{Evidence}}{p(D_t)}}$$

Limitations of online Naive Bayes

Often the posterior is approximated, so it will have some error. Not only that, because these errors are multiplied over time by accumulating.
The counters seen above do not take into account any drifts.

So this method is fine if we have no drifts, otherwise if we do, we need to implement some technique that allows us to "forget" the posterior.

Stochastic Gradient Descent

SGD is an optimization algorithm used to minimize the loss function of the neural network. SGD iteratively updates the network weights so as to minimize the loss function.

In particular, let f be a differentiable function and θ its parameters, we want to find the minimum θ *; actually we are satisfied when we find a local minimum such that $\nabla L(\theta, x) = 0$.

We do this applying the gradient descent step:

$\theta_{i+1} = \theta_i - \lambda \nabla L(\theta_i, x)$ with λ learning rate.

The choice of x in the descent step is different if we are in a streaming scenario or out-of-core scenario. In the former case we take the current input x_t (so we do not choose it), in the latter case we perform an iid sampling.

Stochastic Gradient Descent: why minibatches are useful

In practice, instead of using a single element it is better to update through mini-batches because:

1. **Noise tradeoff**: the smaller the batch size, the more the noise will get (recall that too much noise can slow convergence). But noise can help to escape local minima, so a little noise can be useful.

2. **Computational tradeoff**: using several GPU parallelization over batch size is trivial, so it is good for us to have more batches in parallel.

3. **Fast convergence**: at the end, having mini-batches help us to have less number of steps to perform.

Even if we need to consider the **Latency tradeoff** (in the streaming scenario), indeed, if we want larger mini batches we need to wait more.
In the training phase this is not a problem but for inference can block us to have larger mini batches.

Stochastic Gradient Descent: advantages

One thing to keep in mind is that SGD only guarantees local convergence. Despite this, SGD is still widely used because it is a fast and simple method. Moreover, there are many libraries available that can perform auto differentiation, making it easy for users to implement this method.
And, in the end, in out-of-core settings can be useful because it scales to huge datasets.

Stochastic Gradient Descent: i.i.d. assumption

SGD searches a local minimum for $L(\theta, x)$ assuming that x are i.i.d.

In presence of drifts, it will soon adapt to the new examples, «forgetting» the previous ones. We don't even need a drift detector, SGD will adapt quickly.

This is good if we want to forget the first classes but it is catastrophic if we want to keep memory of first classes plus the new ones.

Adapting Offline Methods

Now we will see how to adapt some offline methods (like kNN and Decision Tree) into an online setting.

Online k Nearest Neighbors

The kNN is a non-parametric distance-based classifier.
It stores **(all)** the samples from the dataset and computes distances between old examples and the new input. For classification, the output is a majority voting of the closest k samples.

So, we have two main hyperparameters:

- k: how many neighbors to use
- the distance metric *(in order to choose a good distance metric, we need to know the domain)*.

In an offline setting, there is not train, we simply store the entire dataset, while the inference is performed computing distance.
When we want to use this algorithm in an online setting we have a problem, indeed, the algorithm is designed for offline training: **we cannot save the entire stream**.

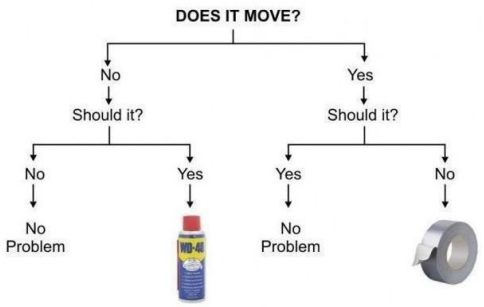
The solution relies on the use of a fixed sliding window: for each timestep, we add the latest instance and forget the oldest instances.
The inference phase does not change and the main advantage is that it is natively adaptive with respect to concept drifts, instead the main cons are that it is slow to adapt and it can not work on large datasets since it limits the generalization capability.

Online k Nearest Neighbors + ADWIN

A possible optimization is based on the implementation of ADWIN inside the online kNN because we want to discard examples only then it is necessary (when a CD happen), indeed, if a concept drift occurs, with KNN there is the risk that the instances saved into the window belong to the old concept.

For this reason we use ADWIN to automatically set the size of the sliding window to save the instances.

Online Decision Tree (Hoeffding Tree)



Now we will talk about another easy classification model for the offline setting: Decision Tree.

This model relies on a tree that represents the criteria to split samples where each sample is assigned to a leaf and internal nodes are the splitting criteria.

The (offline) inference is performed in this way: once the tree is constructed, it can be used for classification, starting from the root node, each feature is evaluated based on the learned decision rules until a leaf node is reached, providing the predicted class label for the input data.

The (offline) training is performed building the tree, where for each node we have to decide:

- if it need to be splitted
- which feature to use for the split

So, it is critical to define the splitting criterion.

In order to understand which feature to use for the split we have to find the most discriminative attribute, for this reason we use the Gini index or the Information Gain.

In a specific way, Information Gain is a measure to determine the usefulness of a feature in predicting the target variable.
In an offline setting, the Information Gain is perfect since we can compute it and greedily split the node but in an online setting we can only compute the Information Gain on the pasta data and we do not know how much the Information Gain will change with future data, indeed, if there are too much changes we need to change the tree.

The simpler idea to solve this problem is to wait for enough data, but how much data do we need before deciding the split?

This is where **Hoeffding bound** comes into play. Hoeffding's inequality provides an upper bound on the probability that the sum of bounded independent random variables deviates from its expected value by more than a certain amount. In other words, it states that as the number of samples increases, the probability of making an incorrect split decreases exponentially.
Now we have a criterion to decide when we have enough samples to do the split.

This leads us to Hoeffding trees: this algorithm is a fast decision tree algorithm for streaming data. It splits the nodes based on the Hoeffding bound, waiting for enough instances to arrive before splitting.

The Hoeffding bound ϵ relies on:

- R is the range of the random variable,
- δ is the desired probability of the estimate not being within ϵ of its expected value,
- n is the number of examples collected at the node

HOEFFDINGTREE$(Stream, \delta)$
Input: a stream of labeled examples, confidence parameter δ

```
1  let HT be a tree with a single leaf (root)
2  init counts n_ijk at root
3  for each example (x, y) in Stream
4       do HTGROW((x, y), HT, δ)
```

HTGROW$((x, y), HT, \delta)$

```
1  sort (x, y) to leaf l using HT
2  update counts n_ijk at leaf l
3  if examples seen so far at l are not all of the same class
4     then
5          compute G for each attribute
6          if G(best attribute) - G(second best) > sqrt(R^2 ln 1/δ / 2n)
7            then
8                 split leaf on best attribute
9                 for each branch
10                     do start new leaf and initialize counts
```

Inside every node we keep the statistic necessary to compute the split criterion, in practice, we keep a table where each row contains three elements: $< x_i, v_j, c >$, where x_i is the attribute, v_j the value of the attribute and c is the counter (this is the same information as needed for computing the Naive Bayes predictions). The memory needed depends on the number of leaves of the tree, not on the length of the data stream.
The DT construction is incremental and Hoeffding bound ensures that the greedy splits must not be revisited.

In particular, for each new sample we find its corresponding leaf, then we update the statistic inside its table and then we split the node if the difference between the value of the split gain (G) of the best attribute and the value of the split gain of the second best attribute is grater than the Hoeffding bound.

$$if \quad G(best\ attribute) - G(second\ best\ attribute) > bound:$$
$$split\ the\ node$$

In the end, the **very fast decision tree (VFDT)** model is an improved Hoeffding Tree algorithm where:

- it computes G every k updates
- it uses the Warm start
- it splits the nodes when two attributes have similar split gain G and Hoeffding's bound is lower than a certain threshold parameter τ.
    - The Hoeffding bound tells you that the G estimates are close to the real value and G are similar, so, you can split because the difference between the two is unlikely to change with more data.

Moreover, there are two more implementations: Concept-Adapting VDFT and HAT (Hoeffding Adaptive Tree). The first one uses a sliding window approach to adapt to new data, while HAT incorporates ADWIN for concept drift detection.

Combined Algorithm Selection and Hyperparameter Optimization (CASH problem)

The process of selecting the best algorithm to use for a given task and optimizing the parameters for that algorithm to achieve the best possible performance is called CASH (Combined Algorithm Selection and Hyperparameter Optimization).

In the offline case, AutoML processes automate the data mining pipeline (data cleaning, feature engineering, algorithm selection and hyperparameter tuning).

In the online case, on the other hand, it is not possible to use a similar approach because parameter adaptation in an evolving data stream with prequential evaluation is not considered.

The only thing we can do, although it is computationally costly and requires a large number of parallel trainings, is to perform the AutoMl training on the first portion of the data stream and re-perform the training from scratch for each concept drift.

In 2022, EvoAutoML comes into play: it's a library to adapt a set of algorithms on the stream without expensive training.

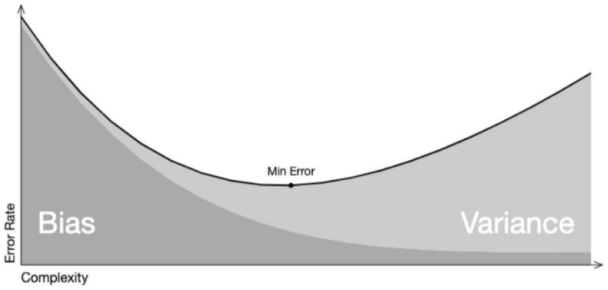
Introduction to Ensemble Methods

Informally, an ensemble is the set of several models combined together with a mechanism that can compute the output. Obviously this generates several non trivial questions, such as: how do we train the individual models? with what data? How do we "aggregate" them?

We will see several offline ensemble methods, which can also be used in the online version, in fact, in the presence of drift the models could be updated on the new data, forgetting the old ones. The problem occurs in case of recurring concepts (seasonality), as we do not want to forget old concepts. We can solve this by using separate models for each concept and selecting the correct model for each step.

Bias-Variance Tradeoff

The bias-variance tradeoff describes the relationship between the complexity of a model and its ability to accurately predict new data.



Bias refers to the error that is introduced by approximating a real-life problem with a simpler model. A model with high bias tends to oversimplify the problem and make assumptions that may not be true.

Variance refers to the error that is introduced by the model's sensitivity to fluctuations in the training data. A model with high variance is highly dependent on the specific data it was trained on and is likely to overfit, meaning it captures noise in the training data and is less able to generalize to new data.

Ensemble Definition

A weak learner is a model that performs slightly better than random guessing. Typically, a weak learner has high bias and low variance, meaning that it makes many simplifying assumptions about the problem and is not very sensitive to variations in the data.
On the other hand, a strong learner is a model that can achieve high accuracy in making predictions. It has low bias and low variance, meaning that it can capture complex patterns in the data and generalize well to new, unseen data.

Definition:

An ensemble can be described as a **composition of multiple weak learned** to form a **strong learner**.

Weak learners and Ensembling

We already write that weak learners have high-bias and low-variance; and an ensemble of weak learners can improve the overall performance.

Actually, in order to obtain this behavior, we need to ensure that the weak learners are different because the bias is decreased if the **errors are decorrelated**.

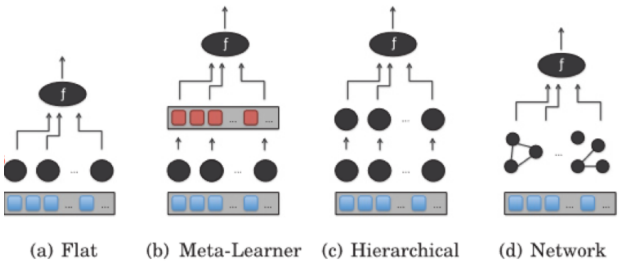
Properties of Ensembling

The properties we want from an ensemble (both offline and online) method are:

- **Diversity:**
  Ensembles work better when base models are different and have decorrelated errors.

  We can induce the diversity in two ways: **horizontal partitioning** and **vertical partitioning**. In the first case (horizontal) we train the models in different chunks of data, so, we split the data set using all the features. While in the second case (vertical) we train the models using different subsets of features.

- **Combination**
  There are several ways to combine base models into an ensemble one.



(a) Flat (b) Meta-Learner (c) Hierarchical (d) Network

  Flat ensemble: multiple models of the **same type**[1] are trained independently and their predictions are combined to produce a final prediction. This is done by taking the average, weighted average or majority vote of the predictions made by individual models.

  Meta-learner ensemble: the "meta-learner" is another model trained on the outputs of the base models.

  Hierarchical ensemble: The models at each level of the hierarchy make predictions based on the outputs of the models at the lower levels. The predictions are then combined at the top level of the hierarchy to produce a final prediction.

  Network ensemble: The outputs of each model are used as input to other models in the network, and the final prediction is produced by combining the outputs of all models in the network.

[1] This means that each model must have the same type of architecture, the same number of parameters, and the same set of hyperparameters.

While for online methods we must add:

- **Adaptation**
  In an online scenario, we could add and remove models on the fly. This is called **dynamic adaptation**; in this case we still need to limit the memory growth.
  Instead, in an offline scenario, the number of base models cannot grow (**fixed adaptation**).

At the end we must describe some of the most important **voting schemes**:

- **Majority vote**: every model has the same weight
- **Weighted majority vote**: gives a different eright to each model
- **Classifier selection**: uses a dynamic criterion to select the best model for the current sample.

Offline Ensemble Methods

In this section we will see some ensemble methods used in an **offline** scenario.

Bagging

The Bagging technique is one of the most famous ensemble techniques. The main idea of Bagging is that it **increases the diversity by training on different subsets of data**; it's a perfect example of horizontal partitioning.

It uses the concept of *Bootstrap* which means "sample with replacement" during the training process, indeed, assuming to have an original data set of dimension n, and m different models we have to sample m sub-dataset of dimension n in an i.i.d. way. So we will have some repetition of sub-datasets.

The inference phase combines the outputs using average or majority voting.
The main properties of Bagging are:

1. it reduce the variance
2. base models should have high variance (to have more diversity)

Random Forests

Random Forest is an algorithm that combines multiple decision trees to create a more accurate and stable prediction model. It's an example of Bagging, indeed, it trains m decision trees, each one on their own bootstrapped data set. This algorithm uses both the horizontal and vertical partitioning.

Stacking

At the end, we have the Stacking technique which relies on fitting a meta-learner to combine the outputs of several base models. Often this meta-learner is very simple (e.g. a linear combination of the outputs).

Online Ensemble Methods

Now let's focus on some Ensemble methods that can work in an online setting.

Accuracy-Weighted Ensemble

This ensemble technique relies on training a set of (fixed) number of offline classifiers C_i, adapting the whole set by removing the oldest models.

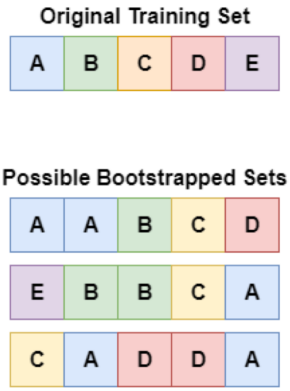
Each classifier is weighted by the expected accuracy on the future data, if we don't have it we can estimate it.
In particular the weight for the classifier C_i is $w_i = err_r - err_i$, where err_i is error of classifier i and err_r is the error of the random classifier.
The output of this ensembling method is $sign(\Sigma_i w_i C_i(x))$ where x is the input.

The stream is processed in chunks, when a new chunk is ready we train a new model to add to the set removing the oldest one to keep always a fixed number of models in the set. Ideally, we would estimate the performance using a separate test set with recent data but realistically we estimate it using the last chunk of data from the stream.

The fixed number of models is a strong limitation, moreover if the chunk is too large it could happen a drift inside it and so its performance will be very low.

Online Bagging

Recall that bootstrap means "sample with replacement".
In an offline scenario we simply draw n random i.i.d samples uniformly with replacement.



In the online setting, it initially seems difficult to extract a sample with replacement from the stream. However, the following property allows bootstrapping to be simulated:

Let n be the number of examples of the training set and k be the number of copies for a specific example, **the number of copies in the training set follows a binomial distribution:**

$$P(K=k) = \binom{n}{k} p^k (1-p)^{n-k} = \binom{n}{k} \frac{1}{n}^k \left(1 - \frac{1}{n}\right)^{n-k}.$$

This means that if we perform $n$independent experiments we have the probability $p = 1/n$ to pick the current example and $1 - p$ to pick another example.
For large values of n, this binomial distribution tends to a $Poisson(\lambda = 1)$ distribution, so we can use this distribution of our Online Bagging.

ONLINE BAGGING(*Stream*, M)

Input: a stream of pairs (x, y), parameter M = ensemble size
Output: a stream of predictions $\hat{y}$ for each x

```
1  initialize base models h_m for all m ∈ {1, 2, ..., M}
2  for each example (x, y) in Stream
3      do predict ŷ ← arg max_{y∈Y} Σ_{t=1}^{T} I(h_t(x) = y)
4         for m = 1, 2, ..., M
5             do w ← Poisson(1)
6                update h_m with example (x, y) and weight w
```

Note that the choice of $\hat{y}$ is performed with a majority vote.

ADWIN Bagging

A problem with the approach above is that it is not particularly designed to react to changes in the stream. For this reason we introduce a variant of the Online Bagging algorithm which supports the Concept Drift detector ADWIN.

It uses M instances of ADWIN (so we have M different CD detectors, one for each base model) to monitor the error rates of the base classifiers. When any one detects a change, the worst classifier in the ensemble is removed and a new classifier is added to it. This strategy is sometimes called "*replace the loser*."

Leveraging Bagging

An experimental observation when using online bagging is that adding more randomness to the input seems to improve performance.

This can be controlled by changing the parameter λ of the poisson distribution: since the Poisson distribution has variance (and mean) λ , this does increase the variance in the bootstrap samples with respect to regular Online Bagging.

This is performed in the Leveraging Bagging which implement the ADWIN Bagging with $\lambda \geq 1$. In addition to increasing the randomness to the input, Leveraging Bagging also adds randomization at the output using "output codes".
This means that multi-class problems map a set of classes into two sets {0,1}.

For example: if we have 4 classes {1,2,3,4}, the first model will perform a binary classification of {1,2},{3,4} while a second model will perform the binary classification over {2,3},{1,4} and so on.
In this way we increase the diversity since each model is learning a different function.

Hoeffding Option Tree (HOT)

Hoeffding Option Trees (HOT) represent ensembles of trees implicitly in a single tree structure. This algorithm is based on Hoeffding Trees. A HOT contains, besides regular nodes that test one attribute, **option nodes** that apply no test and simply branch into several subtrees.

Having several subtrees is important since during inference each subtree is evaluated in parallel (so, each option derived from an option node is performed in parallel) and the output from the leaves is then aggregated (by using a weighted voting).
Instead, during training, we split the tree into several options when we have a similar splitting criterion for different attributes (and we have enough data).

Strategies to limit tree growth are required, as option trees have a tendency to grow very rapidly if not controlled.

Random Forest for Online Learning

We have two main implementation of Random Forest in a Online Learning scenario:

1. **Adaptive Random Forest (ARF)**, uses **local subspace randomization**.
   With local subspace randomization, a random subset of features is selected independently at each node of a decision tree. This subset is used to determine the best split for that particular node.

2. **Streaming Random Patches (SRP)**, uses **global subspace randomization**.
   Each base model is trained on a randomly selected subset of features, so this technique can be applied to any kind of model.

How to deal with seasonality (Recurrent Concept Drift)

Recall that in the presence of concept drifts, previous concepts can reoccur (Recurrent Concepts or "seasonality"), and of course we don't want to learn from scratch every time.

We can deal this problem by keeping two sets of classifiers:
1. Current ensemble of active models (current concepts)
2. A library of available classifiers, currently inactive (past concepts)

We just need a policy to move the modes from one set to another.
So we can use DDM (Drift Detection Method) for CD detection; when we train the classifier c_a we store the training sample into a buffer b_a.

At warning level, we start training a new model c_n storing the samples in another buffer b_n. If the drift is confirmed we compare b_n with all the store buffers to check if the drift is a new drift or an old concept.

Time Series Analysis and Forecasting

In this section we talk only about **time series**.
Important: here there is a change of scenario! We assume to have the entire time series at once and we want to detect, model or remove the non-stationarity.

The problems we will handle are: **Time Series Analysis** (explaining the past, understanding seasonality, finding patterns etc..) and **Time Series Forecasting** (predicting future values).

Stochastic Processes and Stationarity

Definition of stochastic process:

> A **stochastic process** is a collection of random variables (RV) indexed by some set.
> Notation:
> $$\{X_t\}_{t \in T}$$

In the case of Time Series Analysis this set is the time.

We can define two kind of stationarity: **strong** and **weak** stationarity.

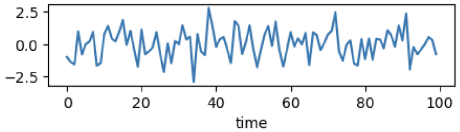
Definition of **strong** stationarity:

> Strong stationarity means that in a stochastic process the probability distribution of the random variable (RV) tossed in each time instant is exactly the same along time, and that the joint probability distribution of RVs in different time instants is invariant to time shifting (this joint probability is usually evaluated with correlation or covariance).
>
> Intuitively, is when the probability distribution does not depend on time.

Definition of **weak** stationarity:

> In a weak stationary stochastic process only the mean and the correlation and covariance of the RV are invariant to time shift (e.g. the higher order (>2) moments of the RV eventually change with time).

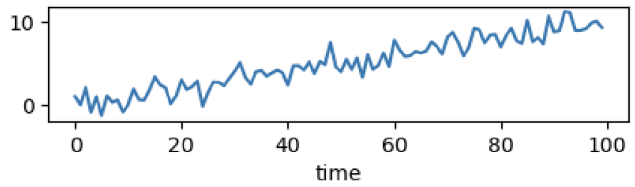
Example of stationarity time series:



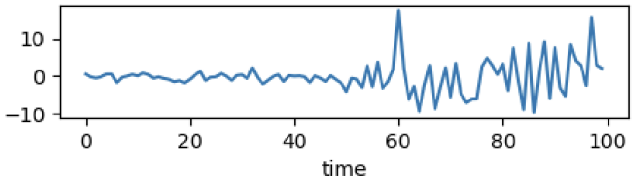
Many forecasting models assume stationarity, if they are not then they transform the time series into stationary via preprocessing because "stationarity" means "predictable".

Let us see some non-stationarity timeseries:

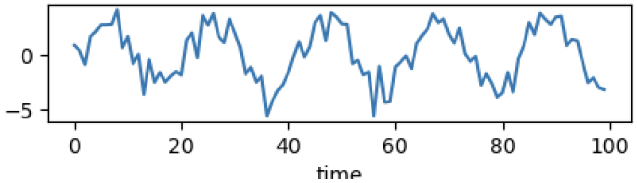
1. Non-constant mean (the trend grown linearly)



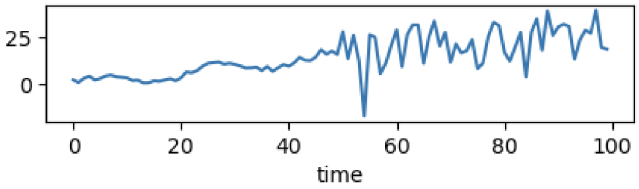
2. Non-constant variance



3. Seasonality



4. Non-constant mean + Non-constant variance + Seasonality



Testing for Non-stationarity

In order to check if a timeseries is stationary we can just plot the time series and look for some trends or seasonality. Otherwise we can compare mean and variance for different chunks of the time series or apply some statistical test.

Time Series Decomposition and Detrending

Now we talk about some ways to remove the non-stationarity from a time series (TS); indeed, we can model the different forms of nonstationarity (mean, variance, seasonality) and then remove nonstationarity from the raw time series via preprocessing.

Time Series components

We can state that a (raw) time series has three main components:

1. **trend (T)**: persistent and long-term change
2. **seasonal (S)**: periodic fluctuations
3. **residual (R)**: stationary component

Additive and multiplicative models

We will define two models (**additive** and **multiplicative**) to combine these components.
In particular, let m_t be the trend component, s_t the seasonal component and Y_t be the residual component.

The **additive model** can be expressed as $X_t = m_t + s_t + Y_t$; this model assumes the seasonal is approximately constant over time.
While the **multiplicative model** can be expressed as $X_t = m_t s_t Y_t$; this is better when the seasonal component changes over time according to the general trend.

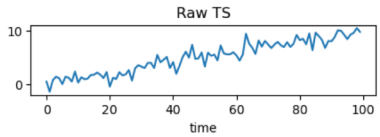
Trend elimination and trend estimation

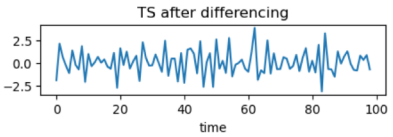
Getting started, we ignore the seasonal component, then we want to model the trend only; in order to do this we will have two strategies:

1. **Trend elimination via differencing**
   This is a simple method to remove trends; we transform a timeseries considering the difference of each couple of consecutive elements.

   $$Y_t = x_t - x_{t-1}$$

   this method remove only linear trends, so if we have a polynomial trends (of degree n) we need to difference n times.





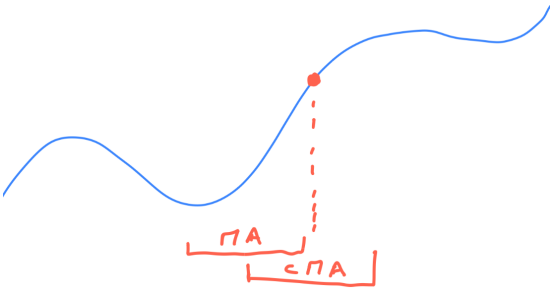
2. **Trend Estimation**
   This method relies on the training of a model which is able to recognize the trend which can be linear or even polynomial. Once we fit the model with the data, it recognizes the trend and then it can remove it.
   Let's see an example for the linear trend scenario: let x_t be the original timeseries, Y_t be the detrended timeseries and $\overline{x_t}$ the trend component.
   Our model will be something like $\overline{x_t} = at + b$, where a, b are the two parameters to be fitted, which means finding the best values of a and b that make the model closely match the original data. Then, we can remove the trend in this way: $Y_t = x_t - \overline{x_t}$.

Seasonal Differencing

The differencing technique can be applied even to seasonality, if we assume to have a fixed and know period d. This is plausible because often the periodic behavior follows the calendar. Hence, we can remove the tren by applying differencing with a period d:

$$Y_t = x_t - x_{t-d}$$

Estimate trends with (Centered) Moving Average



Another way to estimate trends is by using the **moving average** or the **centered moving average**.
In details, the moving average estimates the average at time t by using the last q elements, where q is the size of the window.

While the centered moving average estimates the average at time t by using a windows centered around t. With this method, the first and the last elements are ignored.

Seasonality with Moving Average

Assuming to know the period d, we can deal with seasonality with Moving Average.
In particular, after we detrended the timeseries, we can divide it in windows of length d. Then, we can compute $w_k \mid k = 1 \dots d$ as the average of all the windows in position k.
For example, suppose to have a dataset where each timeseries is one year, we have $d = 12\, months$ and we can compute $w_{1 \dots 12}$ as the average of values of January, February, ecc..
At the end, when we have all the w_k we can estimate the seasonality component by subtracting the average:

$$\hat{s}_k = w_k - \frac{1}{d}\sum_{j=1}^{d} w_j \qquad \forall\, k \in [1 \dots d]$$

Forecasting

Forecasting a time series involves predicting future values based on historical data, so, given a timeseries $x_1, \dots, x_t$ we want to predict x_{t+k}. We can predict trends, seasonal components or residual (short-term variations).

We have two approaches, indeed, we can preprocess the TS in order to make it stationary and then train the forecasting model or we can just train the forecasting model directly without any preprocessing.

The metrics used for this task are:

1. Mean Absolute Error (MAE): $\Sigma_i |Y_i - \hat{Y}_i|/n$
2. Mean Absolute Percent Error (MAPE): $(100/n)\,\Sigma_i |Y_i - \hat{Y}_i|/n$
3. Mean Square Error (MSE): $\Sigma_i (Y_i - \hat{Y}_i)^2/n$
4. Root Mean Square Error (RMSE): $\sqrt{MSE}$

where Y_i are the targets and $\hat{Y}_i$ the predictions.

While the baselines are very simple:

- **average** $\hat{y}_{t+1} = \frac{1}{n}\Sigma_i y_i$
- **windows-based** $\hat{y}_{t+1} = \frac{1}{k}\sum_{i=0}^{k-1} y_{t-i}$
- **last value** $\hat{y}_{t+1} = y_t$

Exponential Smoothing

Exponential smoothing is a family of methods for forecasting time series data. All these methods are weighted averages of past observations, with the weights decaying exponentially.

These models are easy to implement and they produce accurate forecasts quickly.

We will see different Exponential Smoothing methods but all of them don't need a preliminary timeseries decomposition.

Exponential Smoothing : Simple Exponential Smoothing (SES)

Simple Exponential Smoothing (SES) is the most basic form of exponential smoothing. It is used for forecasting time series data without any trends or seasonal patterns. SES assigns exponentially decreasing weights to past observations, giving more importance to recent data points.

The formula for Simple Exponential Smoothing is as follows:

$$\hat{y}_{T+1|T} = \alpha\, y_T + (1 - \alpha)\hat{y}_{T|T-1}$$

where:

- $\hat{y}_{T+1|T}$ is the forecasted value for the next time period
- y_T is the actual value at time period t
- $\hat{y}_{T|T-1}$ is the forecasted value at time period t
- α is the smoothing parameter. It determines the weight given to the most recent observation. The value of α lies between 0 and 1.
  A smaller α places more emphasis on past observations, while a larger α gives more weight to recent data points.

We can generalize the forecasting to predict the value of the time series after h steps (and not just the next one). SES may not be suitable for all forecasting scenarios due to its inability to handle trends and seasonality.

Exponential Smoothing : Holt's linear trend method

Holt's linear trend method is an extension of Simple Exponential Smoothing that takes into account the presence of a trend in the time series data.

Here (and in the next method) we can define the level equation, trend equation, and forecast equation. These equations help determine the level, trend, and forecasted values for a given time series:

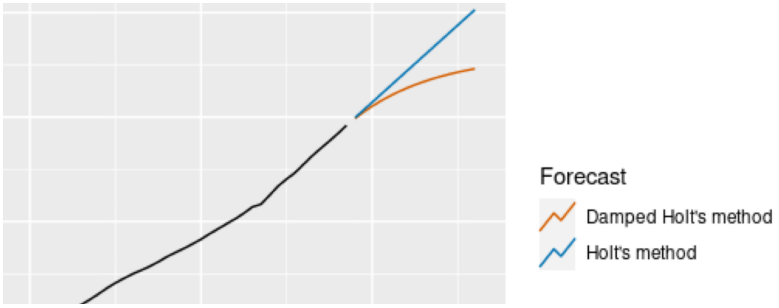
- The forecast equation predicts the future values of the time series based on the level and trend equations.
- The trend equation calculates the trend component of the time series, representing the rate of increase or decrease over time.
- The level equation calculates the level component of the time series. It represents the average value of the series at a specific time period.

For the Holt's linear trend method these equation are the following:

- **forecast equation** (for h steps ahead):
$$\hat{y}_{t+h|t} = l_t + hb_t$$
- **level equation**: we add the trend component
$$l_t = \alpha y_t + (1 - \alpha)(l_{t-1} + b_{t-1})$$
- **trend equation**
$$b_t = \beta(l_t - l_{t-1}) + (1 - \beta)b_{t-1}$$

Holt's method assumes a trend constant goes on indefinitely, overestimating the real trend. If we want a better model we can apply the **trend dampening**, it represents the rate at which the trend diminishes over time.
We will add a parameter $\phi \in [0, 1]$ which controls the dampering factor in this way:

- **forecast equation** (for h steps ahead):
$$\hat{y}_{t+h|t} = l_t + (\phi^1 + \phi^2 + \ldots + \phi^h)b_t$$
- **level equation**: we add the trend component
$$l_t = \alpha y_t + (1 - \alpha)(l_{t-1} + \phi\, b_{t-1})$$
- **trend equation**
$$b_t = \beta(l_t - l_{t-1}) + (1 - \beta)\phi\, b_{t-1}$$



Exponential Smoothing : Holt-Winters method

At the end we see the Holt-Winters method is an extension of Holt's linear trend method that additionally captures seasonal patterns in the time series data.

The seasonal component s_t is added with a smoothing factor γ over a seasonality period m. Here we can find the two methods:

1. **additive:** assuming roughly constant seasonal variations
2. **multiplicative**: seasonal variations are proportional to the level

Let $k = floor\,(\frac{h-1}{m})$.

Additive method:

$$\hat{y}_{t+h\,|t} = l_t + hb_t + s_{t+h-m(k+1)}$$
$$l_t = \alpha(y_t - s_{t-m}) + (1 - \alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1 - \beta)b_{t-1}$$
$$s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1 - \gamma)s_{t-m}$$

Multiplicative method:

$$\hat{y}_{t+h\,|t} = (l_t + hb_t)s_{t+h-m(k+1)}$$
$$l_t = \alpha(\frac{y_t}{s_{t-m}}) + (1 - \alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1 - \beta)b_{t-1}$$
$$s_t = \gamma(\frac{y_t}{l_{t-1} - b_{t-1}}) + (1 - \gamma)s_{t-m}$$

In conclusion even here we can add the dampened coefficient ϕ to the multiplicative method (additive is also possible):

$$\hat{y}_{t+h\,|t} = (l_t + (\phi^1 + \phi^2 + \ldots + \phi^h)b_t)s_{t+h-m(k+1)}$$
$$l_t = \alpha(\frac{y_t}{s_{t-m}}) + (1 - \alpha)(l_{t-1} + \phi\, b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1 - \beta)\phi b_{t-1}$$
$$s_t = \gamma(\frac{y_t}{l_{t-1} - \phi\, b_{t-1}}) + (1 - \gamma)s_{t-m}$$

– Chapter 2 : Knowledge Transfer and Adaptation

Introduction to Deep Learning: Classical ML Models require manual preprocessing to extract discriminative features. We might learn the feature extractor; this is the main aim of Deep Neural Networks (DNN).

DNNs are able to automate the feature extraction by stacking multiple layers sequentially. Low layers capture low-level knowledge (e.g. texture), while high layers high-level knowledge (e.g. complex shapes).

The algorithm used to learn a DNN is the Stochastic Gradient Descent, where we have the usual step:

1. Forward (compute the output for the current input)
2. Compute the Loss
3. Backward (compute gradients)
4. Descent step

Deep Neural Network for image classification

A common example of DNN is applied to the image classification task.

ResNet

One popular example of a Computer Vision (CV) model is the **ResNet** architecture.

The primary components of ResNet are residual blocks, which are composed of convolutional layers and **shortcut** connections. These shortcut connections allow information to bypass one or more convolutional layers, which helps to prevent the vanishing gradient problem that can occur in very deep networks.

In general, most networks have three main components: Convolutional blocks, Feedforward layer and the Classifier.

The Convolutional blocks contain different operations like: Convolution, Batch Normalization layers, ReLU activation function and so on.. Let's see them one-by-one.

Convolution

The basic processing block for images is the convolutional neural network (CNN) layer. It takes an input tensor of shape <C,W,H> where C represents the number of channels, and W and H represent the width and height of the image, respectively. The output tensor also has the same shape <C,W,H>.

The CNN layer has several parameters that can be set to modify its behavior, including padding, stride, and kernel size. Padding adds zeros to the edges of the input tensor to preserve its spatial dimensions during convolution. Stride determines the amount of movement of the convolutional kernel across the input tensor. Kernel size defines the dimensions of the filter used for convolution.

The number of dimensions in the input tensor depends on the domain. For example, for sequences such as text or audio, the input tensor may be 1D. For videos, the input tensor may be 3D to represent the additional temporal dimension.

As the neural network gets deeper, the number of channels in the output tensor may increase to capture more complex features of the image. This technique is called feature extraction and is used to detect more abstract and high-level features of the input image

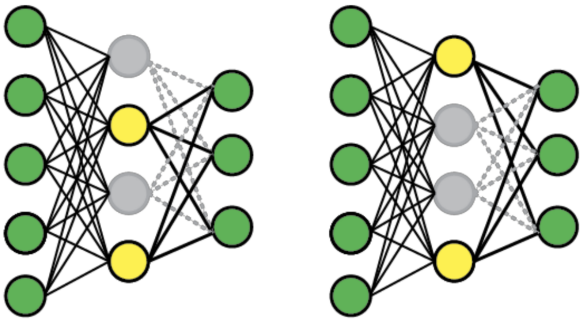
Batch Normalization

Batch normalization is a technique commonly used in deep learning to standardize the output of hidden layers. This is done by subtracting the mean and dividing by the standard deviation of the activations of the layer, which results in a distribution with zero mean and unit variance.

The parameters gamma and beta are learned during training through backpropagation and used to scale and shift the standardized output.

During training, as we saw, it removes mean and std computed on the minibatch.
Instead, during inference, it removes mean and std computed using a running average among the inputs.

Dropout



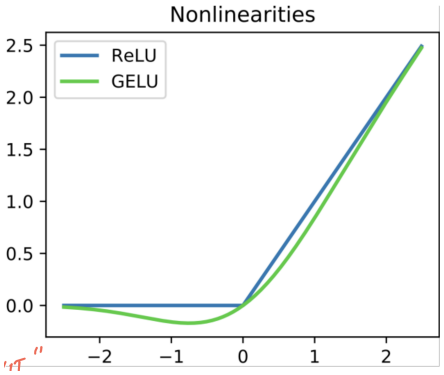
Dropout is a regularization method for Deep Neural Networks (DNNs) that helps prevent overfitting. The intuition behind dropout is to approximate the effect of ensembling multiple neural networks by training a single neural network with randomly dropping out, or "masking," some of its units during each iteration of training.

During training, for each layer of the neural network, a random mask is generated where each unit is masked (i.e., set to zero) with a probability parameter called the dropout rate, typically denoted by *p*. This means that during each iteration of

training, a different subset of units is masked out, forcing the network to learn redundant representations that generalize better.

During inference, the dropout is turned off, and the weights of the remaining units are scaled by p, which ensures that the total input to each unit remains the same as during training. This scaling is necessary to ensure that the expected value of each unit's output remains the same during training and inference, preventing the model from being dependent on the dropout process.

ReLU and GeLU



Sigmoid and tanh activation functions saturate the gradients, which can cause the vanishing gradient problem in deep neural networks. ReLU addresses this issue by providing a better gradient flow and avoiding saturation, leading to faster and more stable convergence during training.

GeLU (Gaussian Error Linear Unit) is another type of activation function commonly used in neural networks. It is a smooth approximation of the Rectified Linear Unit (ReLU) activation function and is based on the Gaussian error function.

Compared to ReLU, GeLU has a smoother curve and can help improve the performance of neural networks in certain tasks. However, it is more computationally expensive than ReLU and may not always lead to better results.

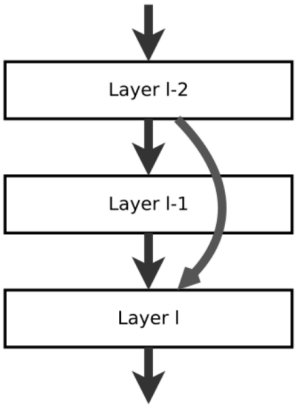
Pooling

Pooling is a common technique used in convolutional neural networks (CNNs) for downsampling the feature maps that are output by the convolutional layers.

During pooling, a sliding window is applied over the feature maps, and the values within the window are aggregated using a specific function, such as max or mean.

Residual Connection

During the backward pass, the gradient flows through each layer of the deep neural network in order to update the weights of the network. However, the problem with gradient flow in DNNs is that it can suffer from vanishing or exploding gradients, which can make it difficult for the network to learn and converge to a good solution.



Residual connections, also known as skip connections, can help to improve the gradient flow in deep neural networks by allowing information to bypass some of the layers.
This is achieved by adding a **shortcut** connection from the input of a layer to its output, allowing the gradients to flow directly through the shortcut instead of being multiplied by small values at each layer.

Softmax and Cross-entropy

In general, we define as "**logits**" the output of the penultimate layer.
In a CNN, logits are the input for the softmax layer, where the **softmax** is a smooth and differentiable argmax function, whose output is between 0 and 1 (so it can be seen as a probability).

Cross-entropy is a commonly used loss function in classification problems. It measures the dissimilarity between the predicted probability distribution and the true probability distribution.

When computing the cross-entropy loss during training, **it is not necessary to compute the softmax explicitly.** Instead, we can directly compute the cross-entropy using the logits (the pre-softmax outputs) of the network, which has better numerical stability; instead, during inference, we typically only need to find the class with the highest probability (i.e., the argmax of the logits). The softmax function is not needed since it only normalizes the probabilities but does not change their ranking.

Tips and useful tools for DNN

In this part we will see some useful practical aspect for learning a DNN

Hyperparameter Optimization

Perform a full hyperparameter research is often infeasible. We can notice that not always the hyperparameters have the same importance, indeed, most of them don't have a big impact on the final results.
On the contrary, some parameter (e.g. *learning rate*) is able to change a lot of the results, so we can focus mainly on these.

As a general rule: start from the best model in the literature and start to improve on it.

A nice guideline can be this Google Book:
https://github.com/google-research/tuning_playbook

Model Initialization

Weight initialization is an important aspect to take into consideration; indeed, usually a model cannot recover itself after a bad initialization. A common error is to ignore the weights initialization.
Some symptoms of this error is that the training is not stable or the network is not able to converge.

Learning Rate Scheduling

Another common general rule is the decay of the learning rate during time. Indeed, the best results in computer vision (up to now) are a combination of SGD and Momentum with learning rate decay. There are several ways to decrease it.

Early Stopping and Model Checkpointing

Other techniques used to stabilize the training are early stopping and model checkpoint.

- **Early stopping**:
  We have to evaluate the model on the validation set periodically. If we do not improve the metrics for "patience" epochs (where patience is a hyperparameter), we force the stop of the training. Using this technique we can set a huge number of epochs and we just stop when we stop learning.

- **Model checkpointing**:
  Even here we evaluate the model on the validation set periodically. If we have a better result with respect to the previous one we save a model checkpoint.
  After the training, we load the best checkpoint and we use it for inference.

Useful Tools

- *timm* and *torchvision* are two PyTorch libraries for computer vision with pretrained models and state-of-the-art architectures.
- Data augmentations:
  - *Almunmentations*
  - *Kornia*
- Faster data loader : *ffcv* this can reduce the training time a lot.
- Training frameworks:
  - *fastai*
  - *Pytorch lightning*
  - *avalanche*
- Logging and Visualization
  - *clearml*
  - *tensorboard*
  - *hiplot*

Transfer Learning, Fine-tuning and Domain Adaptation

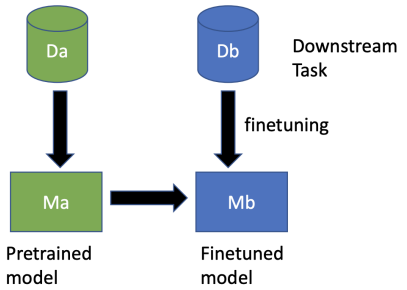
In this section we will talk about how to reuse pretrained model for new tasks.

Transfer Learning (TL)

Let T_a, T_b be two tasks (e.g. image classification of plants 🌿), let D_a,D_b be the datasets of these tasks and θ_a,θ_b be the parameters of a Deep NN after training on the respective datasets.

> **Definition of Transfer Learning (TL)**
>
> Solve the target task T_b after solving the source task T_a by transferring knowledge learned from T_a; we cannot access the dataset D_a .

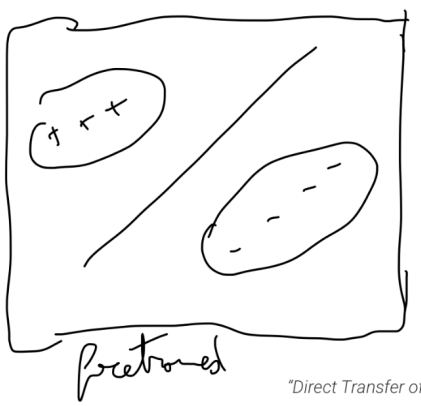
Note that the source task, actually, may be more than one; and you can solve the Multi-Task Learning (next section) with TL methods but not vice versa.
The typical setting of TL is having the pretrained dataset D_a very large and the dataset for new task D_b very small. Moreover, we don't care about solving T_a and T_b jointly.

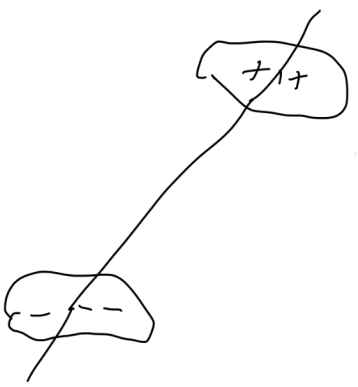


Transfer Learning is different from Multi-Task Learning because in MLT we have all the data for all the tasks at the same time and we want to learn all them jointly, while in TL we just have a pretrained model and a new task over a little dataset.

The assumption about TL

We need to do an important assumption about T_a and T_b: we need that these **tasks must be related**, this means that discriminative features learned from T_a must be helpful for T_b. In order to understand this assumption well, let's suppose to have an SVM model as a pretrained model and two classes : a positive one and a negative one (see the figure on the left). If the new dataset D_b is orthogonal to the positions of these two classes, the old model is useless! (see the figure on the right).





"Direct Transfer of L

Fine-tuning

The reuse of latent features is the key of TL, indeed they can be still useful to solve related tasks.
This means that in a DNN, since first layers are more "generic" we can change only the last ones.

The finetuning process is quite simple, we can perform the SGD on the new data D_b starting from the source model with its parameter θ_a.
The descent step is:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta, D_b)$$

Design Choices (about Fine-Tuning) and Warm Start

These are some popular choices from literature:

1. **epochs:** usually the number of epochs required for finetuning is less than the ones required for training from scratch.
   This means that we have a **fast adaptation** property to similar tasks (fast convergence) and since the number of epochs is small, we can **avoid overfitting** over small datasets.

2. **learning rate:** the new learning rate is often smaller

3. **weight decay:** the weight decay may be set to zero, because the weight decay is used to regularization, so if we are using a smaller learning rate and a smaller number of epochs perhaps no more regularization is needed.
   Moreover, weight decay bring the weights to zero but our goal is have a new model (θ_b) simal to the source one, so bringing the weights to zero is harmful.

4. **reinit:** for some scenarios it's useful to have a random reinitialization for the last layers (e.g. when we have a classifier at the end).

5. **freezing:** it is often not necessary to train the whole model, so we can freeze the first few layers. This is very efficient because we can also precompute all the activations for the freed layers, without recomputing them.

6. **Warm start:** this is a technique that relies on training only the last layer (usually, a classifier) and then finetuning everything.

   The idea behind this is that the randomly initialized classifier may have large gradients which result in large changes in the DNN;
   This technique helps to reduce the "forgetting" of the representations, even if not always is the best choice.

The pseudo-algorithm is the following:

a. Start from the pretrained model
b. Freeze all the model except the classifier (last layer)
c. randomly initialize the classifier
d. finetune only the classifier
e. unfreeze all the model
f. finetune the whole model

The steps (d)...(f) are needed to avoid a big change in the internal representations.

All this paragraph is saying also that we cannot simply copy the hyperparameters used in the pretraining part.

Difference between Self-transfer and Transfer

Suppose to have two ImageNet (1600 classes) splits called A and B.

We define "**self-transfer**" when we train the network on A and then we finetune it on A. Then, we define the "transfer" when we train the network on A and then we finetune it on B.

Problems about fine-tuning

The TL relies on the fact that low-level features are useful for different tasks, but this is not always the case, indeed if you change the domain too much, the transfer may not work anymore.

Another problem about TL applied on CNN (ImageNet) is about the texture bias.

At the end, we can have some spurious correlations that hurt the TL performance. For example, the difference between house dogs and sled dogs is evident for humans but a DNN may answer "the snow".

Definition of task and domain

A **task** is a tuple containing a probability distribution on the input, a probability distribution of the outputs given the input, and a Loss to be optimized.
Each task can have different probability distributions of the outputs and Loss.

$$T_i = \{p_i(x),\ p_i(y|x),\ L_i\}$$

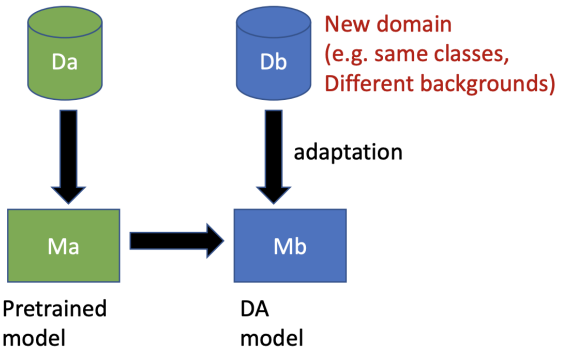
Example: image classification of animals.

A **domain** is a subgroup of a task, indeed it is defined in essentially the same way, but the Loss and probability distributions of the outputs are in common.

$$d_i = \{p_i(x),\ p(y|x),\ L\}$$

Example: image classification of animals with a jungle background, image classification of animals with a city background etc..

Domain Adaptation



Domain Adaptation is the process of **transferring knowledge between different domains**, while Transfer Learning focuses on using knowledge learned from one task to improve performance on another task.

An example of domain adaptation problems is when we have a finite (and fixed) number of classes and new images for this classes arrive but they have a different background.

In DA scenarios we have a **source domain** (the data distribution on which the model is trained using labeled examples), a **target domain** (the data distribution on which a model pretrained on a different domain is used to perform a similar task).

The term "**Domain translation**" refers to the problem of finding a meaningful correspondence between two domains, while for "**Domain Shift**" we mean a change in the statistical distribution of data between different domains. So, we define the **domain adaptation** as a transfer learning problem where we have access to target domain data during training.

Actually, there are several subtypes of domain adaptation:

- **Unsupervised DA**: unlabeled target domain data
- **Semi-supervised DA**: unlabeled target domain data with a small labeled subset
- **Supervised DA**: labeled target domain data

Domain Adaptation Methods

From now on we will assume that:

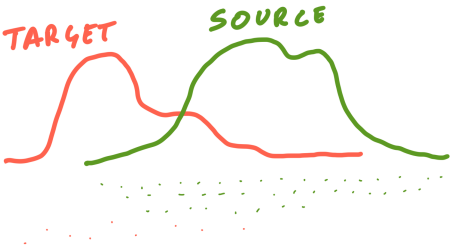
1. source and target are different domains but closely related
2. there exists a single model with low error on both source and target data
3. the shift from source to target is a form of virtual drift

We will see three different methods for performing Domain Adaptation:

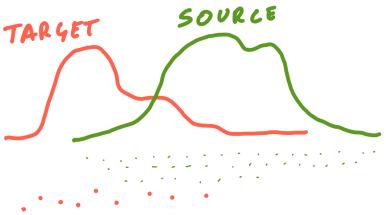
1. **Data reweighting**
2. **Feature Alignment**
3. **Domain Translation**

Domain Adaptation Methods: Data reweighting (Importance Sampling)

If we train a model on the source distribution $p_s(x, y)$,
it will ignore samples from the target distribution $p_t(x, y)$.
This is a classical example of "*imbalance problem*".



We can handle this problem by weighing more samples with high target probability (and low source probability).



So, we just need to find these **weights**; this technique is called **Importance Sampling**.
Recall that x is our input, y is our output, p_S is the source distribution and p_T is the target distribution.
Mathematically, we can use the Empirical Risk Minimization, this means that we will reweight data and then we will optimize the loss with respect to the reweighted data.

The error on source is the expectation over the source distribution of the loss:

$$\mathbb{E}_{p_s(x,y)}[L(x, y, \theta)]$$

The error on target is quite similar:

$$\mathbb{E}_{p_T(x,y)}[L(x, y, \theta)]$$

From the definition of expected value we can derive:

$$\mathbb{E}_{p_T(x,y)}[L(x, y, \theta)] = \int p_T(x, y)\, L(x, y, \theta)\, dxdy$$

$$= \int p_T(x, y) \frac{p_s(x,y)}{p_s(x,y)} L(x, y, \theta)\, dxdy$$

then we can swap p_s and p_T, obtain the definition of expected value on p_s:

$$= \mathbb{E}_{p_s(x,y)}[\frac{p_T(x,y)}{p_s(x,y)} L(x, y, \theta)]$$

The term we want to estimate, so, is $\frac{p_t(x,y)}{p_s(x,y)}$. We might try to estimate them directly and then divide but it is difficult to estimate accurately.

Instead we can do this, by definition we can say:

$$\frac{p_t(x,y)}{p_s(x,y)} = \frac{p(x|domain = target)}{p(x|domain = source)}$$

By applying Bayes rules we can get:

$$\frac{p_t(x,y)}{p_s(x,y)} = \frac{p(x|target)}{p(x|source)} = \frac{p(target|x)}{p(source|x)} \frac{p(source)}{p(target)}$$

The term $\frac{p(source)}{p(target)}$ is a constant and doesn't depend from the data, so we can remove it without changing the optimal solution - recall that we are under the ERM setting, so it is like a constant inside the loss function!

Then we can estimate $\frac{p(target|x)}{p(source|x)}$ by training a binary domain classifier to understand if a data belongs to the source or to the target domain; the probability will be $p(source|x_i; \theta)$ to belongs to the source domain and $1 - p(source|x_i; \theta)$ to the target domain.

Hence, $\frac{p(target|x)}{p(source|x)}$ can be estimated by the result of this binary classifier $\frac{1 - p(source|x_i;\theta)}{p(source|x_i;\theta)}$; this value for the item x_i will became the weight.
In conclusion, we can optimize the loss after applying the weight to the data.

The main problem of the Importance Sampling algorithm is that **it assumes that the source domain contains the target domain**. This is true if we start from a general domain to a specific one (e.g. from ImageNet to cars classification), but it can be false if we switch from a specialized domain to another specialized domain.

Domain Adaptation Methods: Feature Alignment

If we cannot apply the importance sampling, we can align the source feature with the target feature. In this way **we can reuse the source classifier with the target data** in the aligned feature space.

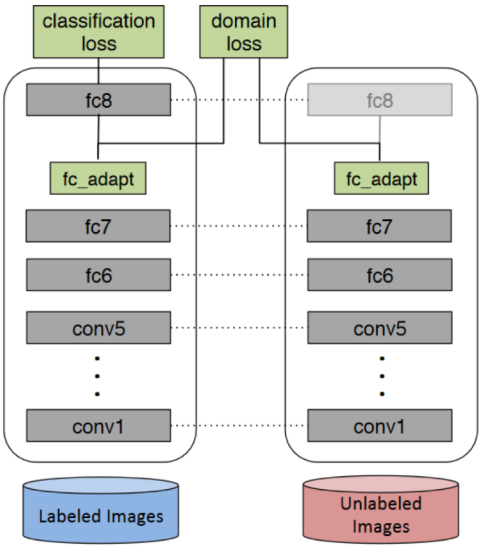
In general, the model is splitted into two parts:

- the feature extractor: $f_\theta(h)$
- the classifier: $c_\theta(h)$

We want to achieve the **domain invariance**: this means that our features, extracted by the feature extractor, should be invariant with respect to the domain, in other words, we want that the features from source and target have the same distribution.
Let's see two models based on this idea: *Deep Domain Confusion* and *DANN*.

Feature Alignment : Deep Domain Confusion

The first model is called **Deep Domain Confusion**.
This model is a shared CNN between labeled images from the source distribution and unlabeled images from the target distribution, so we are in an unsupervised setting.



Here we have two goals:

1. We want to learn features that are discriminative
   - **classification loss**: the usual cross entropy.
2. We want to learn features that are domain invariant
   - **domain loss**

The domain loss is also called *Maximum Mean Discrepancy* and given a minibatch of data from the source and from the target (X_s, X_T) it calculates first the means of the embedding for the source and for the target and then it calculates the norm of the difference of the means:

$$MMD(X_s, X_T) = \left\| \frac{1}{|X_S|} \Sigma_{x_s \in X_s} \phi(x_s) - \frac{1}{|X_T|} \Sigma_{x_t \in X_T} \phi(x_t) \right\|$$

In this way, the domain invariance is obtained because we are forcing the embeddings of source and target to be similar, and, if the embeddings are similar we have an alignment of the features.

So, the total loss we want to minimize is the combination of the classification loss and the MMD:

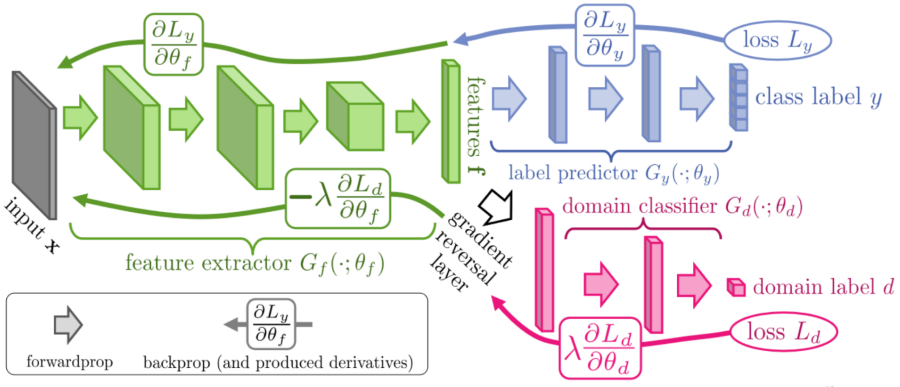
$$L = L_C(X_L, y) + \lambda MMD^2(X_S, X_T)$$

Feature Alignment : Domain-Adversarial Neural Network (DANN)

We have seen that Deep Domain Confusion performs feature alignment by aligning embedding statistics, now let's see a different approach called **Domain-Adversarial Neural Network (DANN)**. This method relies on adversarial training to force the feature alignment in order to obtain the domain invariance.
Even in this case we are in an unsupervised setting, so we have no labels for the target.

Moreover even here we have the same two goals: we want to learn **discriminative features** and obtain the **domain invariance**.

The first goal is obtained by training the classifier and the feature extractor to classify source data.
The second goal is obtained by training the domain classifier to guess the domain and, at the same time, by training the feature extractor to "fool" the domain classifier, this is an usually GAN-like objective.



The architecture is based on three components:

- the feature extractor G_f (green)
- the label predictor G_y (blu)
- the domain classifier G_d (red)

Moreover there are two losses to be optimized: the prediction loss L_y and the domain loss L_d. The domain classifier G_d is trained to guess the domain:

$$L_d = -\mathbb{E}_{x \sim Ps}[log\, G_d(G_f(x))] - \mathbb{E}_{x \sim Pt}[1 - log\, G_d(G_f(x)]$$

The classifier G_y is trained to classify the source data.

At the end, the feature extractor is optimized to improve both the classification and to fool the domain classifier:

$$\min_{\theta, \theta g} \mathbb{E}_{(x,y) \sim Ps}[L(G_y(G_f(x), y)] - \lambda L_d$$

Note that the $-\lambda L_d$ is the gradient reversal.

In other words, the feature extractor and the domain classifier optimize the same objective in opposite directions.

Domain Adaptation Methods: Domain Translation (CycleGANs)

The last way to perform Domain Adaptation is via **CycleGANs**.

The align feature techniques can be hard to apply; for this reason now we see the Domain Translation technique.

Suppose to have two function which are able to translate source to target and target to source:

$$F: S \rightarrow T; \quad G: T \rightarrow S$$

We could use these function to translate source data to the target domain, train the classifier with the translated source data and then use the classifier on the target domain. It also works in the other direction.

These functions can be implemented with a **CycleGAN**. CycleGAN is a deep learning model that can translate images from one domain to another by using a cycle consistency loss.

Note: CycleGAN are an implementation of GANs, from now on we will use $X \rightarrow Y$ and viceversa and no more S and T.

The idea behind the CycleGAN is called **Translation Consistency** (also called *cycle consistency*): this means that when an image is translated from domain A to domain B and then back to domain A, it should ideally keep its original characteristics and be similar to the original image.

So, for our goals, we want to learn the (source → target) mapping ($G: X \rightarrow Y$) with GANs but this is problematic because we have no constraints; this means that it is very unlikely that our target images have the same structure (e.g. a dataset where the zebras and horses are exactly in the same position, with the same light, etc..).

To solve this problem we learn also the **inverse** mapping, so we learn both $G: X \rightarrow Y$ and $F: Y \rightarrow X$. In this way we can force the cycle consistency between an input x and $F(G(x))$: this is performed that by optimizing the following loss function that compare the distance between x and $F(G(x))$ and the distance between y and $G(F(y))$:

$$L_{cyc}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}[\, ||\, F(G(x)) - x\, ||_1\,] + \mathbb{E}_{y \sim p_{data}(y)}[\, ||\, G(F(y)) - y\, ||_1\,]$$

It is worth noting that this loss is quite similar to the GAN loss but here the inputs are real images and not random noise: so in the GANs the discriminator is trained to distinguish real images from fake images, here we are training it to distinguish source images from target images.

The final loss function combines $L_{cyc}(G, F)$ to the other two GAN losses (which represents the translation and the inverse translation) in this way:

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F)$$

where L_{GAN} is the usual GAN loss, as example:

$$L_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{data}(y)}[log\, D_Y(y)] + \mathbb{E}_{x \sim p_{data}(x)}[log\,(1 - D_Y(G(x)))]$$

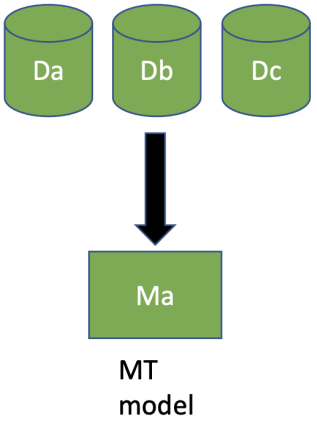
Multi-task Learning

References for this section: [Stanford Slides](#) | [Video Lecture](#)

Let's start recalling the definition of task:

> A task is a tuple like:
> $T_i = \{p_i(x),\ p_i(y|x),\ L_i\}$
> where L_i is the loss used for evaluation (not necessarily the training loss).

Multi-task learning is an approach to machine learning where a model is trained to perform multiple related tasks simultaneously, resulting in better performance than training separate models for each task.



In this context, we will deal with problems like:

- *few-shot learning* : when we have just few example for each class
  - we could also have a *zero*-shot scenario!
- *long-tail problems* : where the majority of the data consists of rare or infrequent examples.

An assumption we do in MLT problems is to have access to the task descriptor z_i, which is usually an integer (or even a more complicated vector) for distinguishing the task.

MTL Problems types

We can distinguish between two main problem types in MultiTask Learning:

1. **Multi-task classification**: where we have a common loss shared among all the tasks. For example, if you are building a model for animal classification, you might want to classify animals according to their species, habitat, and diet. In this case, multi-task classification allows you to use a single neural network to classify animals according to all three categories.

2. **Multi-label learning**: In this case, we have a common loss shared among all the tasks, moreover we also have the same input distribution between all the tasks. Our aim, so, is to assign multiple labels to each input, since each task labels in a different way the same input.

MTL Objectives

In a Multi-Task learning scenario, we solve all the tasks concurrently, **sharing the knowledge** between them. This allows a faster convergence and a better generalization. Mathematically speaking, we want to find the parameters that minimize the sum of the different Losses over their dataset:

$$\min_{\theta} \sum_{i=1}^{T} L_i(\theta, D_i)$$

This equation is called "*Vanilla MTL Objective*".

In other words, MTL is not just "combine" more models together, but we have this aims:

- **fast adaptation**: learn all the tasks more quickly, converging faster than the single task models.
- **forward transfer**: it improves the generalization, having a lower loss than the single tasks models
- **meta learning**: given a MT pretrained model, learning a new tasks it's not a big deal (we will come back to this aspect)
- **few-shot learning**: we want to use a MT model even in a low-data setting.

Of course we have a big assumption: **the tasks must share some common structure**.

At the current state-of-the-art we cannot guess in advance whether multi-task training will perform better or worse with respect to independent models for each task.

Design Choices about Multi-Task Learning

Let z, an integer that identifies uniquely a task, be the **task label**.
In this section we assume to always know the labels. Indeed, knowing z we can perform task-specific computation and use it to select some parameters.

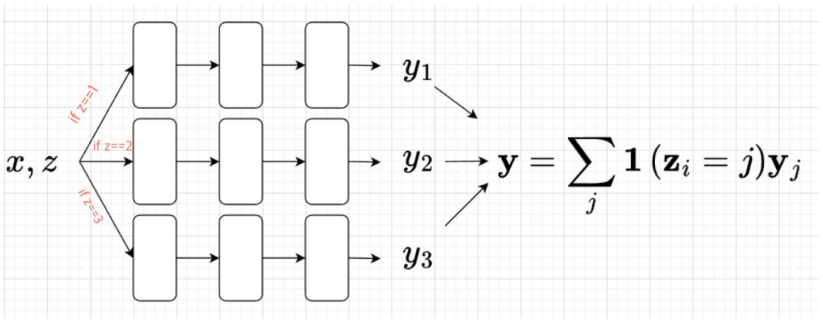
We do have a problem: **how do we choose which layer to share?** We have three main ways:

1. Do not share any layer
2. Multi-head architectures
3. Task Conditioning

Let's see them in detail.

Weight Sharing – No thanks

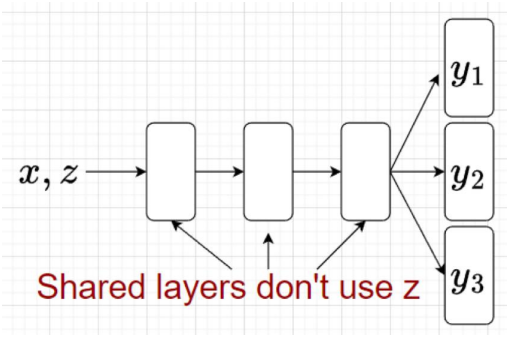
In this case we have an independent network, so there are no shared layers. We have a **gate mechanism** to select the correct model for each input.



Weight Sharing – Multi-head architectures

A Multi-head architecture must have three components:

1. a **shared** feature extractor
2. a **separate**, parallel, **processing path for each task**, called "head".
    a. each head could learn a particular feature of the input.
3. a **multiplicative gating** to select the correct head for each example.



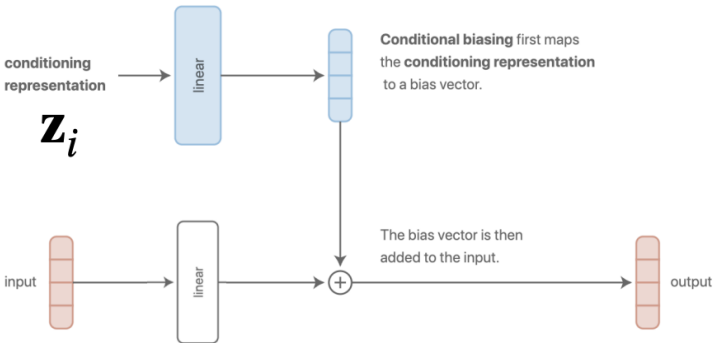
Weight Sharing – Task conditioning

Recall that we can identify each task with its z, which can be an embedding of the task label. We can split θ, the parameters, into **shared parameters** θ^{sh} and **task-specific parameters** θ^{i}.
In this way out objective function will become:

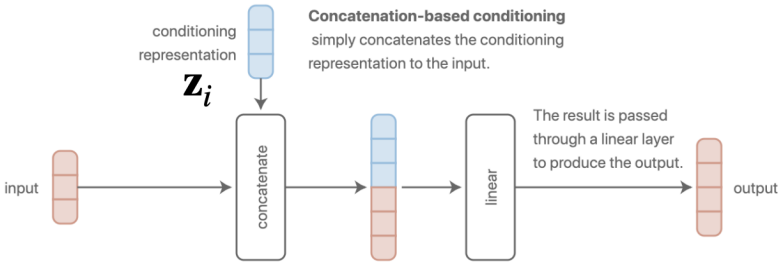
$$\min_{\theta^{sh},\theta^{1},\ldots,\theta^{T}} \sum_{i=1}^{T} L_i(\{\theta^{sh}, \theta^{i}\}, D_i)$$

The shared layers have to merge, with a certain way, the task label z with the input. We can do it in different ways but the common ones are sum, concatenation and multiplication.

Sum:
In this case the task label z_i is mapped into a bias vector that is summed to the input.

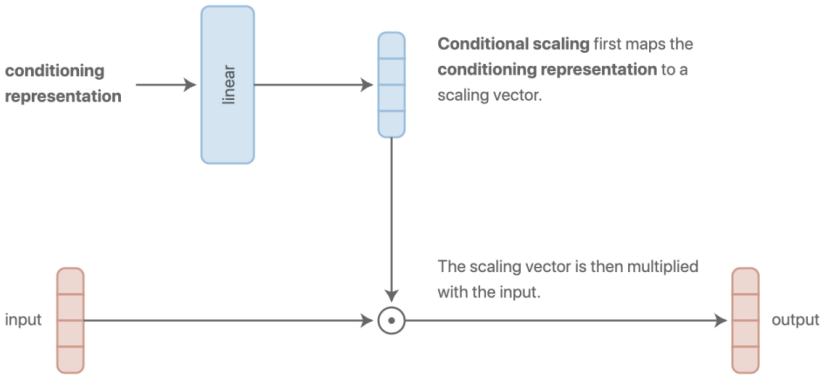


Concatenate:
In this case we simply concatenate the input with the condition representation. The result is passed through a linear layer to produce the output.



This method and the previous one are actually equivalent.

Multiplication:
At the end, the multiplication case maps the representation into a scaling vector that is multiplied with the input.
This case is **more expressive** and can be used to gate activations, so can help to generalize independent networks and independent heads.



MTL Objective

We already saw that we want to find the parameters that minimize the sum of the different Losses over their dataset:

$$\min_{\theta} \sum_{i=1}^{T} L_i(\theta, D_i)$$

Actually, we can weigh the different losses in order to give a different importance to each task; we can update the previous equation to:

$$\min_{\theta} \sum_{i=1}^{T} w_i L_i(\theta, D_i)$$

There are different heuristics to choose the weights: we can **manually based on importance** or priority, or we can **dynamically adjust throughout training**: for example we could **optimize** the model **for the worst task or** we could **normalize the gradients**, which means modifying the magnitude values of gradients during training to balance the importance of different tasks (this algorithm is called GradNorm).

A naive way to implement SGD in a MultiTask Scenario

A naive way to implement MT-SGD could be:

- sample tasks
- sample examples for that task
- apply the usual SGD step: forward -> backward -> descent step

Since we are controlling the samplings, we are implicitly balancing over the tasks instead over samples, so this ensures that tasks are sampled uniformly regardless of the data quantities.

Positive and negative transfer

In a MultiTask scenario we could have two main behaviors:

- Training tasks jointly **improves** the performance of the single tasks.
- Training tasks jointly **doesn't improve** the performance on the single tasks.

The first case is called **positive transfer** and implies that the weights sharing works, a common case is when the tasks are small then the joint solution is more robust and less prone to overfitting.

The second case is called **negative transfer** and implies that there was some kind of inferences among tasks, maybe because we have tasks with different rates of learning.

Self-Supervised Learning

In the previous section we saw that often the training can be splitted into pretraining and finetuning, and we saw how to perform finetuning.
In this section we will see how to do the first part (the pretraining); we will see the Self-Supervised Learning.

In particular, during the pretraining we want to learn discriminative features for a classification problem, but we don't have labels (since we will have them in the finetuning part).
In other words, we want features that are transferable to many downstream tasks[2].

So let's start with some definition:

- **Self-supervised learning (SSL)** is predictive learning.
  Given a (large) unlabeled dataset, the system is trained on a task where the labels are self-generated.
- **pretext task**: the self-supervised task that defines a supervised loss given the unsupervised data. In other words, pretext tasks are auxiliary tasks used in self-supervised learning to train a model to learn useful representations of data without explicit human annotation by requiring the model to perform a specific prediction task on the input data.
- **evaluation of a SSL**: we have a set of downstream tasks. We finetune the pretrained model on each task separately and evaluate the performance.
  We have two ways to do this:
  - finetuning on all layers
  - finetune only the final classifier (linear probing)

A very popular example of this is BERT model, that is trained via self-supervised learning using a masked language modeling (MLM) as objective function, since this objective doesn't require expensive labels and a model with MLM can transfer knowledge very well to downstream tasks in NLP.

Self-Supervised Learning: Methods

Now we will see three (actually two) methods to apply the Self-Supervised Learning in the field of Computer Vision:

1. **Augmentations**: exploit the properties of the domain such as invariance to transformation in order to learn robust representations.
2. **Contrastive learning:** learn representations by comparing and contrasting pairwise images.
3. **Encoding/Autoregressing**: learn to reconstruct the input (don't covered in this section)

[2] Downstream tasks are defined as all those tasks that can be executed starting from a pretrained model.

Augmentations

Augmentations are techniques used in computer vision and image processing to create variations of an existing image dataset by applying a set of image transformations to the original images. Example: flipping, rotating, changing brightness, adding noise or blurring the image, cropping, resizing, and so on..

We want a model that is invariant to augmentations, extracting salient features of the images to recognize them.

ExemplarCNN

ExemplarCNN is a CNN architecture that uses data augmentation techniques during training to increase the size of the training. Specifically, **each image in the dataset is augmented before training**, the pretext task in this case is to assign each augmented image to the corresponding original image.

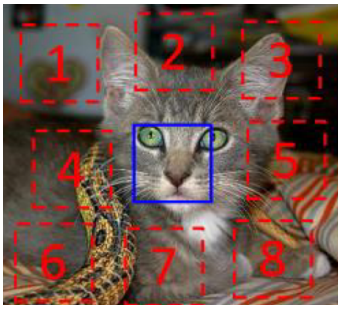
Learning from Image Patches

Another pretext task we might use is to **predict the relative position of two random patches from the same image**, where a patch is a portion of the image.

The distance between the patches is a tradeoff:

- if patches are too far, their relative position may be unpredictable
- if patches are neighbors, the model can track lines and edges to align the patches

When designing a pretext task, care must be taken to ensure that the task forces the network to extract high-level information, without taking "trivial" shortcuts, in order to have a better generalization.



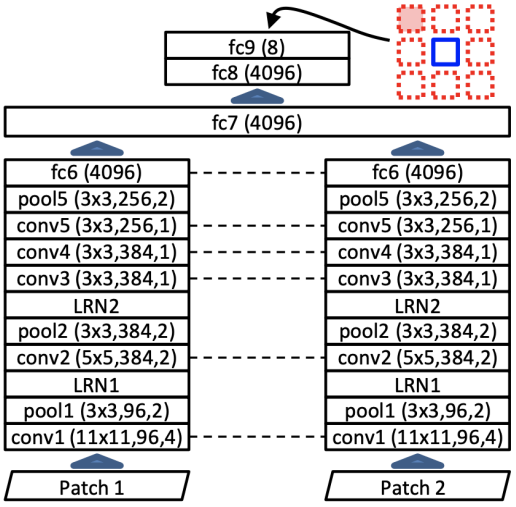
In this case, low-level cues like boundary patterns between patches could be a shortcut. Hence, for the relative prediction task, it was important to include a **gap** between patches.

Moreover, in order to make the task more difficult and encouraging learning non-trivial features (since we want better generalization of the model), we can randomly sample the central patch and randomly pick one neighbor from the 3x3 grid (so we have to choose one of the eight possible patches) centered around the first patch adding some **jitters** (a random oscillation).

Despite the gaps and jitters, it is possible to find another problem that makes the task more trivial: **Chromatic aberration**, a type of distortion of the image.

To handle this Chromatic aberration problem we can apply color transformations (e.g. shifting green and magenta toward gray).

The architecture proposed in the paper is a pair of **AlexNet**-style architectures that process each patch separately, until a depth analogous to fc6 in AlexNet, after which point the representations are fused, in order to predict the relative position of the two random patches.



Dotted lines indicate shared weights. ‘conv’ stands for a convolution layer, ‘fc’ stands for a fully-connected one, ‘pool’ is a max-pooling layer, and ‘LRN’ is a local response normalization layer. Numbers in parentheses are kernel size, number of outputs, and stride (fc layers have only a number of outputs) .

Contrastive Learning

Contrastive learning is a machine learning technique used to learn the general features of a dataset without labels by teaching the model which data points are similar or different.

The goal of Contrastive Learning is to learn robust features for downstream tasks, where by "robust features" we mean that different classes are linearly separable. Therefore, in order to separate classes, we need to bring similar images closer together and push different ones away.

The compare and contrast approach used during the training of a Contrastive Learning model is similar to what happens in Supervised Learning using a softmax with cross entropy. In that case the logits of the positive class are pushed up, and the logits of the negative classes are pushed down. But recall that in Contrastive Learning we do not have labels.

Representation Collapse

A legitimate question that could be asked at this time is "why do we need to push away negative examples?" The answer is that if we don't, everything is collapsed into the same representation.
This problem is called "Representation Collapse."

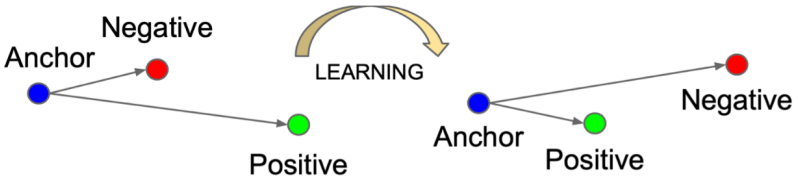
Triplet Loss

The compare and contrast approach ("bring similar images closer together and push different ones away") requires a particular Loss function called *Triplets Loss.*

This loss is made up by three components:

1. **anchor**: it's the embedding that represent the class
2. **positive**: it's the embedding of an image with the same class (same distribution) of the anchor
3. **negative**: it's the embedding of an image from another class (another distribution).

In a more formal way, the Contrastive learning model tries to minimize the distance between the anchor and positive samples in the latent space, and at the same time maximize the distance between the anchor and the negative samples.



So, this loss is :

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

where:

- x is the anchor
- x^+ is the positive
- x^- is the negative
- the first norm is the distance with the positive
- the second norm is the distance with the negative
- ϵ is a margin between positive and negative (hyperparameter)

It's worth specifying that we have just one example for the anchor, one example for the positive and one example for the negative.

Triplet Sampling

We have a problem with the tripless loss: the convergence is slow since it depends a lot from selection of the triples. Ideally, we would pick the **most difficult examples**:

- the positive as far away as possible

$$\arg \max_{x^+} || f(x) - f(x^+) ||^2$$

- the negative as close as possible

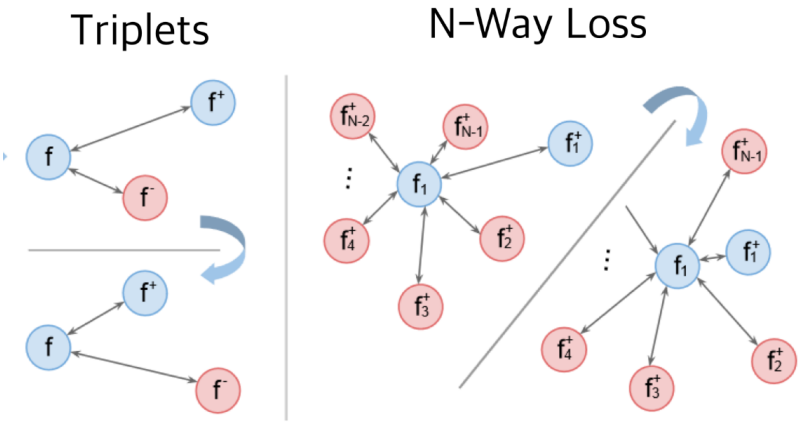
$$\arg \min_{x^-} || f(x) - f(x^-) ||^2$$

Having these, and moving the positive closer and the negative farther away, we gain as much as possible from the training.

The problem is that it is very expensive to calculate, so in practice we use very large mini batches.

N-way Loss

The triplet loss is slow to converge because it looks at a single <pos,neg> pair at each step. The solution is to compare **more negative examples** together.



This is done with the **N-way loss** that generalizes the Triplett loss using $n - 1$ negative examples.

In this way we reduce the complexity by using n pairs of example, instead of $(n - 1) \times n$ ones, indeed we can reuse the same n examples for all the mini-batch.

The formula is:

$$L(\{x, x^+, \{x\}_i^n; f) = \log\left(1 + \sum_{i=1}^{n-1} \exp(f^T f_i - f^T f^+)\right)$$

Negative Mining

The selection of negative samples for the N-way Loss is named 'Negative Mining'. As for the Triplet loss we want the hard samples (the positive as far away as possible, the negative as close as possible, both to the anchor).

1. We choose a very large number of elements, larger than the mini batch. This set will contain a large number of classes that we denote with C.
2. From that set we randomly extract one or two examples of each class.
3. We randomly select the first class inside the set.
4. The next classes we greedy select by looking for the classes that violate the constraints of the triplet (the one in which the negative is closest to the positive).
5. We Iterate this process until we have as many classes as we want in our mini-batch.
6. Finally, we sample the two examples for each class.

Embedding Regularization

Without any kind of regularization, the norm of embeddings grows unboundedly.
And for example, it has been seen that in classical ways of embedding (e.g. Word2Vec) the norm of an embedding of a word is linearly dependent on the frequency of that word in the corpus (so the more frequent it is the larger norm it will have).

This is a problem because the similarity $f^T f^+$ depends on the direction and the norm of the embeddings, so we want to avoid an unbounded norm growth.

We have two ways to solve this problem are:

1. Normalization
    a. we force all embedding vectors to have a certain value (e.g., all norms must equal one).
    b. this is too restrictive
2. Penalizing the norm
    a. we add a regularization term on the norm (higher the norm, the higher the penalty), encouraging the model to learn embeddings with smaller norms.
    b. is a good trade off between norm growth and training slowdown.

This is a common problem among all embedding-based models.

SimCLR

SimCLR is a simple framework that uses Contrastive Learning in a Computer Vision scenario.
The key idea of SimCLR is to bring together pairs of similar examples in representation space, while pushing apart pairs of dissimilar examples.

In order to achieve this goal, it creates **two different versions of the same image**, using two different augmentation functions, and uses these distorted images to learn useful visual features that are invariant to certain types of augmentations, without requiring labeled data.

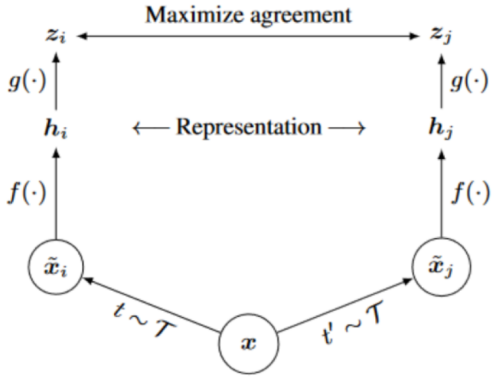
It introduced a **learnable nonlinear transformation** between the representation and the contrastive loss. The authors of this framework showed that this improves the quality of learned representations.
Since it uses the Contrastive Learning idea, it benefits from **larger batch size** and more training steps.

The components used in SimCLR are:

1. **augmentations**: mainly random crop and color distortion
2. **a base encoder**: $h_i = f(\tilde{x})$, where $\tilde{x}$ is the augmented version of an input x.
   This encoder is a ResNet without the linear classifier

3. **the projection head**: the contrastive loss is not applied to the embeddings h_i,h_j (called, also, representation) but to a projected space. We obtain this projection using a 1-layer MLP $g(\cdot)$.

Hence, we can summarize the framework with the following schema:



Note that "*maximizing agreement*" refers to the objective of bringing together pairs of similar examples in representation space, while pushing apart pairs of dissimilar examples.

Moreover note that this algorithm doesn't use negative mining since it uses very big batches.

The Contrastive loss function used in SimCLR is called **NT-Xent** which takes the normalized temperature-scaled cross entropy between the representations of two augmented versions of the same image and compares it to the representations of other images in the batch

The temperature parameter τ is used to smooth the distribution of similarities, making the model more robust to noise and outliers.

The NT-Xent loss formula is the following:

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(\mathbf{z}_i,\mathbf{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]} \exp(\text{sim}(\mathbf{z}_i,\mathbf{z}_k)/\tau)}$$

Algorithm 1 SimCLR's main learning algorithm.

input: batch size N, constant τ, structure of f, g, $\mathcal{T}$.
for sampled minibatch $\{\boldsymbol{x}_k\}_{k=1}^{N}$ **do**
  for all $k \in \{1,\dots,N\}$ **do**
    draw two augmentation functions $t \sim \mathcal{T}$, $t' \sim \mathcal{T}$
    # the first augmentation
    $\tilde{\boldsymbol{x}}_{2k-1} = t(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k-1} = f(\tilde{\boldsymbol{x}}_{2k-1})$ # representation
    $\boldsymbol{z}_{2k-1} = g(\boldsymbol{h}_{2k-1})$ # projection
    # the second augmentation
    $\tilde{\boldsymbol{x}}_{2k} = t'(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k} = f(\tilde{\boldsymbol{x}}_{2k})$ # representation
    $\boldsymbol{z}_{2k} = g(\boldsymbol{h}_{2k})$ # projection
  end for
  for all $i \in \{1,\dots,2N\}$ and $j \in \{1,\dots,2N\}$ **do**
    $s_{i,j} = \boldsymbol{z}_i^\top \boldsymbol{z}_j / (\|\boldsymbol{z}_i\| \|\boldsymbol{z}_j\|)$ # pairwise similarity
  end for
  define $\ell(i,j)$ **as** $\ell(i,j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]} \exp(s_{i,k}/\tau)}$
  $\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1,2k) + \ell(2k,2k-1)]$
  update networks f and g to minimize $\mathcal{L}$
end for
return encoder network $f(\cdot)$, and throw away $g(\cdot)$

..while there is the pseudo-algorithm on the right.

Performance of Self-Supervised Learning

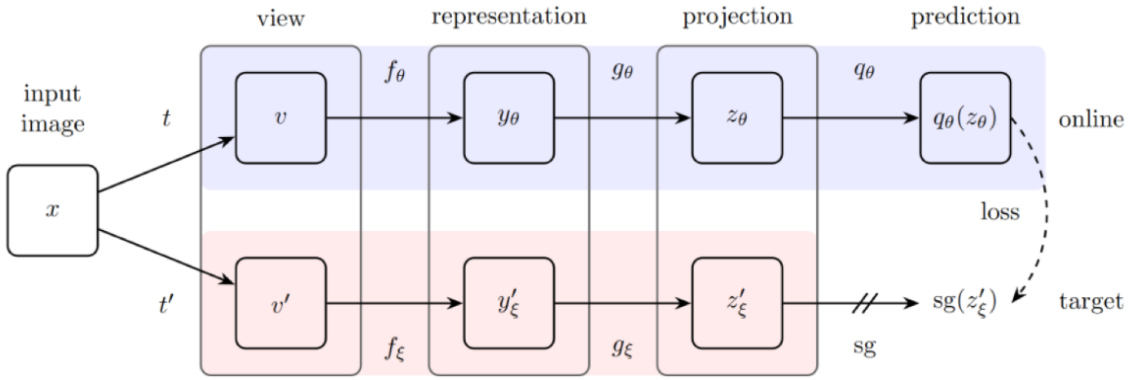
In NLP, self-supervised learning is the current state-of-the-art standard. GPT-3, BERT and Transformers are only a few examples using this technique.

In the field of computer vision, however, SSL, in the actual state of the art, performs slightly worse than supervised models.

BYOL - Bootstrap Your Own Latent

BYOL (*Bootstrap Your Own Latent*) is a training technique (a learning framework) that can be applied to several tasks; in this context we will see it applied to computer vision tasks.

The key idea is to use another network to provide targets for the SSL model.



In practice, we have two Deep NN with the same architecture, called:

- online (θ)
- target (ξ)

We have two different distributions of augmentations (T, T') for the two networks; given the input we extract two different views[3] v, v' applying two different augmentation $t \sim T$, $t' \sim T'$.

Both have three modules: *encoder f, projector g and predictor q.*

- **Encoder (f)**: The encoder module extracts high-dimensional representations from the input data, capturing essential features and patterns.
- **Projector (g)**: The projector module transforms the encoded representations into a different latent space, encouraging the network to learn more meaningful and invariant representations.
- **Predictor (q):** The predictor module operates on the projected representations, providing additional learning signals and regularization to enforce the network's capture of useful and discriminative information.

[3] a "view" is the augmented input

The online network is updated using the mean squared error between the normalized predictions and target projections as loss function:

$$\mathcal{L}_{\theta,\xi} \triangleq \left\|\overline{q_\theta}(z_\theta) - \overline{z}'_\xi\right\|_2^2 = 2 - 2 \cdot \frac{\left\langle q_\theta(z_\theta), z'_\xi \right\rangle}{\|q_\theta(z_\theta)\|_2 \cdot \left\|z'_\xi\right\|_2}$$

In other words, this loss function is just a similarity function between the target representations and online representations.

$$\mathcal{L}_{\theta,\xi}^{\text{BYOL}} = \mathcal{L}_{\theta,\xi} + \widetilde{\mathcal{L}}_{\theta,\xi}$$

where $\bar{z}$ are normalized vectors and $\widetilde{L}$ is equal to L with v and v' switched.

This loss is used to update the online network parameter, in this way:

$$\theta \leftarrow \text{optimizer}\left(\theta, \nabla_\theta \mathcal{L}_{\theta,\xi}^{\text{BYOL}}, \eta\right)$$

The target network is not updated with backpropagation but with a Exponential Moving Average of the online network, with a decay rate with a parameter τ :

$$\xi \leftarrow \tau\xi + (1 - \tau)\theta$$

Target network is initialized as a copy of the online network and the main objective of the target network is to provide a stable reference point for the online network during training.

So, to conclude, the target network provides targets to the online network; this works even if we use a randomly initialized network as target net (obvs, with a very low accuracy).

Conclusions about Self-Supervising Learning

In conclusion, let's recap some information:
Self-Supervised methods learn robust representation without any supervision.

Many methods require large batch size and more iterations and, in general, we cannot use the hyperparameters of a supervised method setting in a Self-Supervised learning.

In order to learn robust representations

- Augmentations are a fundamental tool even if they are expensive
- Contrastive Learning is the standard

In NLP Self-supervised learning is state-of-the-art but not in the computer vision field.

Meta-Learning, Metric Learning and Few-Shot Learning

In this section we will see the Meta-Learning, a field which has the goal of developing algorithms that can **learn how to learn**.

Meta-Learning

The learning process of a generic Machine Learning algorithm has the aim of optimizing a model, given a dataset; inputs are one or more samples from the dataset and the final output of the learning process is the trained model.

Instead, a meta-learning model is quite different from a "classic" learning process, indeed: given a set of datasets (different tasks), the aim here is to **optimize learning algorithms** (or hyperparameters).

The goal, in a meta-learning algorithm, is not only to train a model on a specific set of data, but also to teach the model how to learn and how to generalize its knowledge to new problems.

Example: Suppose we want to train a humanoid robot to walk; this robot has been trained to walk in different scenarios in the laboratory (this will be defined as '*meta-train set* '). Then, we deploy the robot in the real world ('*meta-test set*'), and we want it to be able to walk on any kind of terrain, using the knowledge of 'walking' learned in the lab. The ideal solution is that the robot takes a few steps, stumbles a couple of times and then adapts itself to the new environment (A/N: like a child).

Now, let's define everything in a more formal way:

We already saw the definition of task as a tuple $< p_i(x), p_i(y|x), L_i >$ where we assumed to have a data set D_i used during the training.
Instead, we can define a **meta-task** as a distribution of tasks $p(T)$ with a common loss function. During the meta-training, we can sample different tasks from the distribution ($T_1 ... T_n \sim p(T)$) and then we can retrieve the datasets of each task (we will perform the training on these datasets).

Then we defined as a learner a generic machine learning model, and now we can define as a **meta-learner** a parametrized learning algorithm (a learned optimizer, a learned initialization, a learned hyperparameter search etc..).:

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

Note that we sample datasets (and not instances) from $P(D)$ which is the data set distribution defined by our family of tasks (we assume that all the tasks have some shared structure). So, we minimize the parameters θ over all the entire family of tasks.

Typically, a meta-learning algorithm is something like this:

$for \ \ T_i \in p(T)$: //outer loop (meta-learning)

$for \ mb_i \in D_i$: //inner loop (learning)

** *learning* **

$L_{inner} = \ldots$

$L_{outer} = \ldots$

Some terminology:

- **support set**: task training set D_i^{tr}
- **query set**: task test set D_i^{test}
- **meta-training**: training process over the meta-train tasks
- **meta-test**: learning a new task given its support set

A common data set used for meta-learning benchmarks is called Omniglot, which can be seen as a meta-learning version of MNSIT; it contains 50 alphabets splitted into a background set of 30 alphabets and an evaluation set of 20 alphabets.
Usually, we use the background set to learn the general knowledge of characters and use the evaluation set for one-show learning.

To conclude, we can see the meta-learning in two different points-of-views:

1. **optimization view**
   Given a set of tasks, we want to find a model (an optimization algorithm) that is able to quickly learn new related tasks.

2. **probabilistic view**
   Given a set of tasks, we want to extract prior knowledge about tasks and use it to infer posterior for new tasks.

Meta-Learning Families

There are three main meta-learning families:

1. Model-based
2. Optimization-based
3. Metric-based

Meta-Learning Families : Model-based

This family of meta-learning algorithms is based on designing some model architecture that has a fast adaptation on new tasks.

Here we can find two main subfamilies:

1. **Memory-Augmented Neural Networks (MANN)** are NNs with some external memory (also called Neural Turing Machines).

2. **Meta-networks**, here the weights of the network is splitted into:
    - fast weight, updated via a meta-learner
        - these parameters represent the "actual" knowledge of the downstream task we are taking into consideration.
    - slow weight, updated by SGD
        - these parameters represent the "general" knowledge, something that doesn't change with respect to the downstream task

    This is done in order to have better generalization properties.

Meta-Learning Families : Optimization-based

This family of meta-learning algorithms is based on adapting the parameters of the meta-learner for learning new tasks.

Also here we can find two subfamilies:

1. **LSTM meta-learner**, since the LSTM cell update is very similar to the SGD update formula, we can update the weights using a LSTM used as an optimizer.

2. **Model-Agnostic Meta-Learning (MAML)**, here we learn an initialization that generalizes over more tasks; ideally, we would use this initialization to converge to the new task with just one epoch.

Meta-Learning Families : Metric-based

This family of meta-learning algorithms is based on learning a metric over the input space; a classic example is a KNN where the distance function is learned using a DNN. The metric is learned during the meta-training and used during the meta-test.

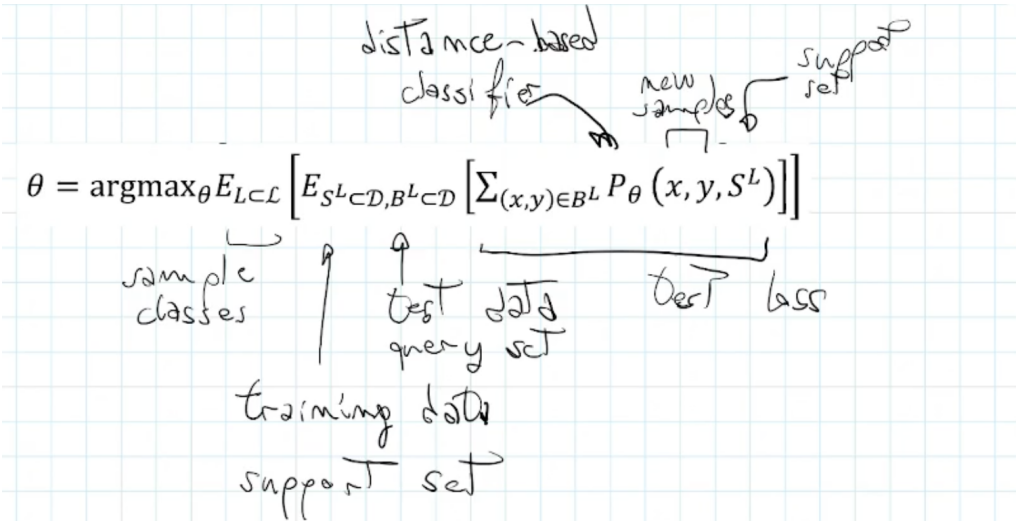
We will see some Metric-based algorithms:

- Siamese Networks
- Matching Networks
- Prototypical Networks

Few-Shot Learning

An example of Meta-Learning is the Few-Shot Learning.

The Few-Shot learning is an approach to learn from a limited number of examples.
The meta-training loss function for Few-Shot meta-learning is:



Note the dependency of the classification model P_θ to the support set S^L.
During the meta-test we receive a new support set for training the new task.

In extreme scenarios we could have even the **one-shot** and the **zero-shot** learning.
The zero-shot learning can be performed on metadata

KNN in a Few-Shot Learning scenario

A good choice for few-shot learning (and low-data scenarios, in general) is the KNN model since it is a non-parametric model. The train process is just "store the dataset" and the inference is just an average of the k closest distances.

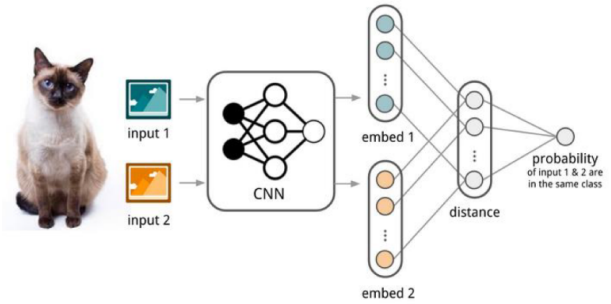
KNN model is distance metric based model, so, in general, in the meta-train we will learn the distance metric.

We can learn how to calculate the embeddings for the instances of our tasks. Recall that an embedding map instances in an high dimensional space encoding relationship between them, the semantic relationship are encoded as distances in the embedding space.

We already saw the embedding in a NLP scenario (e.g. Word2Vec), now we will see how to use embedding with images. The general plan for the next paragraph is to, assuming to have a meta-training data (aka. a set of datasets) we will perform the meta-train (learning a good metric space) and the meta-test (using the learned metric space to classify images).

Siamese Networks

A Siamese Network is a model composed of two CNNs, with weights sharing, that compute embeddings.



The input to a Siamese Network is typically a pair of images, and the model learns to output a similarity score that indicates how similar the two images are.

This score will be used as a metric in the KNN seen before.

So, during the meta-train the siamese network is trained to predict if two input images are from the same class, while during the meta-test the siamese network processes all the image pairs between a test image and every image in the support set.

In particular, the meta-train works in this way:

1. the Siamese network learns to encode two images into embeddings
2. it performs the **L1-distance between the two embeddings** (or any differentiable distance function)
3. **the distance is then converted into a probability** p using a linear feedforward layer and a sigmoid function, this value represents the probability that the two images are drawn from the same class:
$$p(x_i, x_j) = \sigma(W|f_\theta(x_i) - f_\theta(x_j)|)$$
4. then, we calculate the **cross-entropy loss between all pair of images** over the training batch B:
$$\mathcal{L}(B) = \sum_{(\mathbf{x_i},\mathbf{x_j},y_i,y_j)\in B} \mathbf{1}_{\mathbf{y_i}=\mathbf{y_j}} \log p\left(\mathbf{x_i},\mathbf{x_j}\right) + \left(1 - \mathbf{1}_{\mathbf{y_i}=\mathbf{y_j}}\right) \log\left(1 - p\left(\mathbf{x_i},\mathbf{x_j}\right)\right)$$

Moreover, in the original paper, the training batch is augmented with distortion.

Regarding the meta-test, instead, given the support set S and a test image x, the final predicted class is the label corresponding to the example with the higher probability.

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

where $\hat{c}(x)$ is the class label of an image and $\hat{c}(\cdot)$ is the predicted label.

Note that we don't update the Siamese network (aka, the distance metric) during the meta-test.

In the end, it's important to highlight that there is a little but substantial difference between meta-test and meta-train: during meta-train we perform **binary**

classification, while during meta-test we perform **n-way** classification because we use all the classes in the task since we are using the whole support set.

Important: experiments show us that this shift from a binary to n-way classification slightly degrades performance. For this reason we introduce Matching Networks.

Matching Networks

Matching Networks (MN) are another architecture, which like Siamese Networks return a score that can be used as a metric.

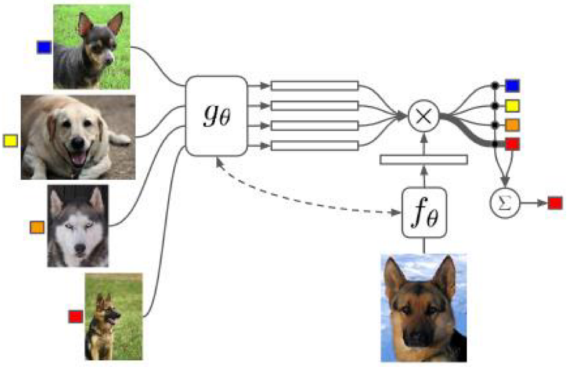


Figure 1: Matching Networks architecture

The idea behind MNs is identical to Siamese Networks but in this case we want to use the entire support set (instead of doing binary classification during the meta-train).

So there are two embedding functions, f and g, the first one embeds the test sample (the image we want to classify), the second one embeds the support set (the training data on the new task).

To use the entire support set, we use the **attention** mechanism.
In general, attention is calculated using a vector of queries and a vector of keys, and the similarity between the various key-values is calculated using a function (usually cosine similarity).

In our case, we then use f to embed the test sample, after which we use g to embed the support set.
Having done this, we can compute the attention between all the examples of the support set (x_i[4]) and the test sample (x) in the following way (using a softmax):

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

After calculating all alpha coefficient values, these represent how similar a new example is to each of the examples in the support set.
These coefficients are then used to calculate the output of the network representing the probability of a class y given the test sample x and the support set S.

$$P(y \mid x, S) = \ldots \; wait \;\; for \;\; it \ldots$$

[4] represents the i-th example of the support set

This probability is equal to the weighted sum of the labels of each example in the support set multiplied by the coefficient of attention between the test sample and the example itself.

$$P(y \mid x, S) = \sum_{i=1}^{k} \alpha(x, x_i)\, y_i$$

This probability is the output that will be returned by the network.

As for how embeddings are calculated, we have two ways:

1. **simple embedding**
   In this case, each test sample is calculated independently

2. **full context embeddings**
   the idea of full context embeddings is that if we calculate the embeddings for the whole support set jointly, we may be able to extract relations that are not evident if calculated independently (this is done via a bidirectional LSTM).

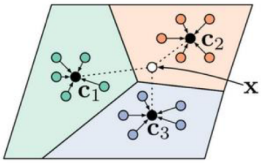
In the case of Matching Networks, by "meta-train" we mean to train f and g (the two embedding functions) on the k-way classification on the meta-train task, while in the "meta-test" (after fixing f and g) we will embed the support set and then do the k-way classification on the unseen data.

To conclude, the main advantages of this method are :

1. There is no longer the difference that there was before between meta-train and meta-test
2. Using full context embeddings we also have the possibility to explore relationships within the support set

Prototypical Networks

Prototypical Networks have been shown to be effective for few-shot learning tasks, this approach computes the class prototypes for classification, it's a simple and effective method and it can be used for zero-shot learning.



(a) Few-shot

Prototypical networks compute embeddings with an embedding function f_θ (think it as a generic CNN) and compute prototypes for each class using the support set.

For "*prototype*" we mean a vector in the space that indicates the class, once we learn all the prototypes for all the classes we can perform classification of a new item x by looking at the most similar prototype.

In particular we define the function f_θ that encodes the input in the embeddings space, then all the prototypes are computed as the average of all the embeddings of the same class in the support set.

$$\mathbf{v_c} = \frac{1}{|S_c|}\sum_{(\mathbf{x_i},y_i)\in S_c} f_\theta(\mathbf{x_i})$$

The output is given by a softmax over a differentiable distance between the prototype of the class v_c and encoding of the input $f_\theta(x)$.

$$P(y = c \mid \mathbf{x}) = \text{softmax}\left(-d_\varphi\left(f_\theta(\mathbf{x}), \mathbf{v}_c\right)\right)$$

Hence, here during the meta-train we train the f_θ on the meta-train tasks, while during the meta-test compute the prototypes using the support set and compute $P(y = c|x)$ for the test set (computing the distances between the new example and the prototypes).

Algorithm 1 Training episode loss computation for prototypical networks. N is the number of examples in the training set, K is the number of classes in the training set, $N_C \leq K$ is the number of classes per episode, N_S is the number of support examples per class, N_Q is the number of query examples per class. RANDOMSAMPLE(S, N) denotes a set of N elements chosen uniformly at random from set S, without replacement.

Input: Training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where each $y_i \in \{1, \ldots, K\}$. $\mathcal{D}_k$ denotes the subset of $\mathcal{D}$ containing all elements $(\mathbf{x}_i, y_i)$ such that $y_i = k$.
Output: The loss J for a randomly generated training episode.

$V \leftarrow$ RANDOMSAMPLE$(\{1, \ldots, K\}, N_C)$ ▷ Select class indices for episode
for k in $\{1, \ldots, N_C\}$ **do**
  $S_k \leftarrow$ RANDOMSAMPLE$(\mathcal{D}_{V_k}, N_S)$ ▷ Select support examples
  $Q_k \leftarrow$ RANDOMSAMPLE$(\mathcal{D}_{V_k} \setminus S_k, N_Q)$ ▷ Select query examples
  $\mathbf{c}_k \leftarrow \frac{1}{N_C} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i)$ ▷ Compute prototype from support examples
end for
$J \leftarrow 0$ ▷ Initialize loss
for k in $\{1, \ldots, N_C\}$ **do**
  for $(\mathbf{x}, y)$ in Q_k **do**
    $J \leftarrow J + \frac{1}{N_C N_Q}\left[d(f_\phi(\mathbf{x}), \mathbf{c}_k)) + \log \sum_{k'} \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))\right]$ ▷ Update loss
  end for
end for

Prototypical Networks are effective even for **zero-shot** settings where we have no labeled sample. In that case we have (given) some meta-data (an embedding) for each class.
On these meta-data we train the embedding function (the CNN) that gives us the embedding of the input image, this embedding is putted into a linear mapping, in order to match the input embedding and the given embedding on the meta-data.

- Chapter 3 : Continual Learning

Introduction to Continual Learning

Offline learning refers to a type of machine learning where the model is trained on a fixed dataset that is available in advance, without the need for continuous data streaming during training.

Offline learning has several limitations:

1. it can not learn in low data scenarios
2. it can not learn when we cannot store the data (e.g. for privacy constraints)
3. it can not be used in a distributed scenario

For these reasons we introduce **Continual Learning**, which can be summarized as "*Learning from a non-stationary stream with Deep Neural Networks without forgetting old tasks.*".

We have to revisit the previous problems in a more challenging setting, indeed, the Multi-Task learning will become Task-Incremental learning, while Online with prequential evaluation will become evaluation on the entire stream.

Formal definition of Continual Learning

Let $S = e_1 \dots e_n$ be the sequence of learning experiences. For a supervised classification problem an experience e_i is just a batch of data D^i, where each data is a tuple $< x_i, y_i >$.
Moreover, let t_i bet the (optional) task label which can be used to identify the correct data distribution, and let M_i be the buffer of past knowledge.
Then, a continual learning algorithm A^{CL} can be defined as a function:

$$A^{CL}: \ < f_{i-1}^{CL}, \ D_i^{train}, \ \ M_{i-1}, \ t_i > \ \rightarrow \ < f_i^{CL}, \ M_i >$$

where f_{i-1}^{CL} is the previous model and D_i^{train} are the new data

The objective of a CL algorithm is to minimize the loss over the entire stream S, often we cannot minimize this loss directly, so we have to estimate it with little data.

Continual Learning goals

- **Lifelong timescales**: learning incrementally on long (lifelong) timescales is a fundamental property of intelligent systems.
- **Limited memory**: No (or limited) access to previously encountered data.
- **Limited computation**: Computational cost is limited over time and it doesn't grow too much.
- **Incremental learning**: the model continually improves over time.

Efficiency + Scalability = Sustainability

Challenges in Continual Learning

Catastrophic Forgetting

The **catastrophic forgetting** (or catastrophic interference) is the tendency of deep neural networks to completely and abruptly forget previous tasks or domains when learning a new one.
Catastrophic forgetting occurs because the weights and connections in the neural network are updated during training to minimize the error on the current task, which can cause the network to overwrite previously learned representations.

One of the main problems is to approximate the loss on an entire stream without having access to the previous data (or at most to a very limited buffer of it).

We can then see the loss on the entire stream divided into two parts:

1. The loss on current data (which can be computed).
2. The loss on previous data that cannot be computed because we do not have the data.

To solve Catastrophic Forgetting we must then **approximate** the loss on the previous data or apply some kind of **regularization** that limits forgetting (e.g., EWC).

the Stability-Plasticity Trade-off

In a continual setting there are two conflicting objectives, so we will be dealing with a tradeoff, often handled by a hyperparameter.

1. **Stability**: the ability to avoid forgetting of old tasks/domains, we can maximize it by freezing the parameters.

2. **Plasticity**: the ability to learn new tasks, we can maximize it by finetuning the previous model on the new task. This can lead to some interferences so in some cases we can maximize it by retraining the whole model.

These solutions are diametrically opposed so we need to find a trade-off.

CL Scenarios

A **Scenario** is a common nomenclature that **identifies** a set of Continual Learning challenges, moreover every scenario has its own relevant **constraints** and handles a type of drift.

In particular, therefore, we can differentiate each scenario based on several parameters:

- whether we have the **task labels** or not,
- whether we know the **task** and **shift boundaries**,
- whether each example contains **new classes** or not
- whether the **output space** is shared or separate, and more.

Combining these parameters we can distinguish the following scenarios:

Name	Task Labels	Boundaries	Classes	Output
Class-Incremental	No	Yes	New	Shared
Task-Incremental	Yes	Yes	New	Separate
Domain-Incremental	No	Yes	Same	Shared
(Online) Task-Free	No	No	Any	Shared

CL Common Assumptions

Brief recap about Real and Virtual shifts:
In a **real shift** $p(x, y)$ changes, this means that the world changes. This can happen abruptly (e.g. the covid lockdown) or continuously (e.g. the weather). In this case we care only about the present and the future, so we can **forget the past**. In this way we avoid that old data interferes with new data.

In a **virtual shift** the world doesn't change; the shift is due to a sample selection bias. So, the data changes because the prior $p(x)$ changes while $p(y|x)$ is fixed. Here we **cannot forget the past** because we can encounter the old data again in the future.

Usually, in any CL scenario we have the same common assumptions:

- **Shift is only virtual**: we do not want to forget, we need to accumulate knowledge.
- **No labeling errors/conflicting information**: targets are always correct (but possibly noisy).
- **Unbounded time**: No hard latency requirements. We may have computational constraints.
- **Data in each experience can be freely processed**: you can shuffle them, process them multiple times, etc. like you would do during offline training

Mainly, therefore, there are two types of shifts:

1. **New instances**: the classes are fixed and each experience provides new instances for those classes; old instances are never seen again in the training stream.
2. **New classes** : each experience provides new classes, old classes will never be revisited in the training stream.
   In this setting we usually have a strong bias: the newest classes will "weigh" more with respect to the older ones.

Questa differenza nel contenuto dell'esperienza può essere utilizzata per categorizzare gli scenari da un nuovo punto di vista, valutando anche la presenza di etichette:

This differentiation can be used to see the scenarios from another point of view:

	New Instances (NI)	New Classes (NC)	New Instance and Classes (NIC)
Multi-Task (MT)	-	Task Incremental	-
Single-Incremental Task (SIT)	Domain-incremental	Class-incremental	Data-incremental
Multiple-Incremental-Task (MIT)	-	-	-

CL Tasks

We can differentiate two main scenarios: the **task-aware** and the **task-agnostic**.
We talk about "task-aware" when the task labels are available during training and inference time, on the other side, we talk about "task-agnostic" when the task labels are not available.

In general, task labels simplify our problems since we can use task labels as explicit arguments of the multi-task models. Moreover the output space is smaller (e.g. if we have 100 classes divided in 10 tasks, we have a 10-way classification, while if we have a single task scenario, we have a 100-way classification).

Another distinction to make in the CL world is between **Online (or streaming) CL** and **Batch CL**. In the Online CL we work with a single example or at least a very small mini-batch, while in Batch CL we have no constraint on the size of the experience, so we may have very large batches.

We define a "**task-free**" scenario when the model doesn't know when the shift happens, while there is no common term for defining the opposite.

In the batch scenario we assume that each experience is the result of a shift while in the online scenario we don't have this assumption; an idea is that to check the drift for each timestep but it's quite expensive.

Shifts can be classified based on their occurrence: abrupt drifts are referred to as **sharp shifts**, while slow drifts are referred to as **blurry** (or gradual) **shifts**; with blurry shifts we cannot perform the check drift at each timestep.

How to perform Model Selection in CL

The main metric used in a CL scenario is the Average Stream Accuracy, which consists of evaluating the model on the average accuracy on the entire test stream. We will see more metrics in the next sections.

We have three main ways to perform model selection:

- Offline Model Selection
- Early Model Selection
- Continual Hyperparameter Selection

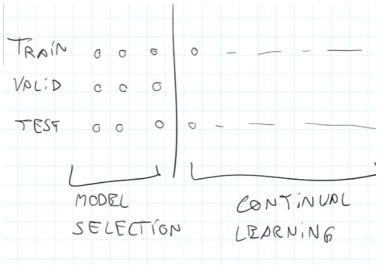
Offline Model Selection

The first way we have to perform Model Selection is to have three parallel streams (Train/Validation/Test streams). Given a generic time t, all these streams must follow the same distribution.

Assuming these streams, we can apply the Offline Model Selection, training different models on the train stream and selecting the best one considering the entire validation stream.
At this point we will carry out the final evaluation on the test stream.

This technique to perform Model Selection is very simple to implement but it is unrealistic for a real scenario in fact we assume that we have access to the entire stream at the end of the training.

Early Model Selection



Another approach to performing model selection is through **Early Model Selection**.
This technique is based on two steps:

- We initially use the first k experiences of the training stream to perform model selection, and after k experiences, we select the best model on the validation stream.
- At this point we continue training the best model on the rest of the training stream.

This model selection technique is still offline, but it is only offline for the first k elements of the stream.

Continual Hyperparameter Selection

The last approach we study to perform Model Selection in the continual domain is called Continual Hyperparameter Selection. This method does not want to access old data.
To achieve this, it tries to find the trade-off between plasticity and stability: specifically, we want to maximize plasticity (i.e., we want to learn as much as possible from new experiences) while minimizing forgetting old experiences (which we do not have access to).
In other words, we want to maximize the accuracy of the current data without changing the accuracy of past experiences.

We have to handle two hyperparameters: one to control plasticity (learning rate) and one for stability (regularization coefficient).

This algorithm relies on two steps:
Step 1: Finding the optimal value to maximize plasticity.
In this step, we ignore the old data; in this way, we get the highest accuracy we can get with the new data.

Step 2: Find the optimal value for stability.
In this step, we start with very high stability parameters (e.g., a very high regularization) and decrease it until we have high accuracy.

This is the stability-plasticity tradeoff: If you stop too soon you have low plasticity. If you stop too late, you have too much forgetting.

Algorithm 1. Continual Hyperparameter Selection Framework

input $\mathcal{H}$ hyperparameter set, $\alpha \in [0, 1]$ decaying factor, $p \in [0, 1]$ accuracy drop margin, D^{t+1} new task data, Ψ coarse learning rate grid
require θ^t previous task model parameters
require CLM continual learning method
`//Maximal Plasticity Search`
$A^* = 0$
for $\eta \in \Psi$ **do**
  $A \leftarrow \text{Finetune}(D^{t+1}, \eta; \theta^t)$ ▷ Finetuning accuracy
  if $A > A^*$ **then**
    $A^*, \eta^* \leftarrow A, \eta$ ▷ Update best values
`//Stability Decay`
do
  $A \leftarrow CLM(D^{t+1}, \eta^*; \theta^t)$
  if $A < (1 - p)A^*$ **then**
    $\mathcal{H} \leftarrow \alpha \cdot \mathcal{H}$ ▷ Hyperparameter decay
while $A < (1 - p)A^*$

CL vs OML

Let's recap the difference between (Deep) Continual Learning and Online Machine Learning:

- In an OML setting we have one sample at time and real drift. The domain is time series data and the evaluation is done with prequential accuracy.
- Instead, in a Deep CL setting, we can have very large experiences and only virtual drift. There are many domains (NLP, vision, speech, etc..) and the evaluation is performed by an average accuracy on the full stream.

CL Metrics

There are several metrics that we can take into account during training. Let us look at some of them.
Let N be the stream length, T be the current timestep amd $R_{t,i}$ be the accuracy on the experience i at time t.

1. **Accuracy**
   The accuracy might be a good candidate as metric but we need to choose two things:
   a. which model to use
   b. which data to use

   We can choose the **last model** (by picking $R_{T,T}$) or an **average over the time** of all the accuracies: $\frac{1}{T+1} \sum_{t=0}^{T} \sum_{i=0}^{t} R_{t,i}$.

   Regarding the data to use we have three choices:
   - **the current experience** $R_{t,t}$
   - **the data seen up to now** $\frac{1}{T+1} \sum_{t=0}^{T} R_{T,i}$
   - **the full stream** $\frac{1}{N} \sum_{n=0}^{N-1} R_{T,i}$
2. **Forward Transfer (FWT)**
   Another metric could be the FWT which measures the influence that learning a task has on the performance of **future** tasks.
   It's defined as :

   $$FWT = \frac{1}{T-1} \sum_{i=2}^{T} R_{i-1,i} - \bar{b}_i$$

   where $\bar{b}_i$ is the accuracy on the experience i of a model trained with a random initialization.
   So, FWT is the accuracy on future experience after training on experience i.

3. **Linear Probing**
   Forward Transfer (FWT) assumes that the model is capable of predicting future experiences. However, in reality, most models are unable to predict unseen classes. Thus, FWT only makes sense for new instances and not for new classes.

   An alternative solution is to evaluate whether the **latent representation** of the model can help in learning unseen tasks. This way, the model can still learn from new data even if it doesn't predict future experiences accurately.

   As we have already seen in the contrastive models, we can evaluate the representation with **Linear Probing**.

   In this case, we train a linear classifier from the learned representations and compare it with a random feature extractor and measure whether the representations learned from the trailed model are better than those from the random model.

   The Linear Probing measures if the learned features transfer to the new data.

4. **Backward Transfer (BWT)**
   Backward Transfer (BWT) measures the influence that learning a task has on the performance on **previous** tasks.
   In other words this metric is the accuracy on the experience i after training on experience T minus the accuracy on the experience i after training on experience i, average over all the $T - 1$ experiences.

$$BWT = \frac{1}{T-1} \sum_{i=1}^{T-1} R_{T,i} - R_{i,i}$$

   This metric is useful if we want to know if the model is improving the performance over the old experiences.

5. **Memory Usage**
   Sometimes we can simply use the number of parameters or megabytes to compare models.

6. **Computation**
   The last metric could be the computational overhead during the training and inference, which can be the number of operations, the CPU/GPU time etc.

In general we have multiple, often conflicting goals.

CL Baselines

A baseline is the starting point of a Machine Learning model used to compare the performance of other models. In this section we will look at the baselines taken into account in the Continual Learning setting.

Finetuning

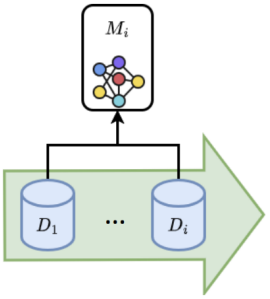
The very first baseline to take into account is the finetuning since naive finetuning often results in catastrophic forgetting. So, CL methods should always beat this Naive baseline and for this reason this represents a sort of "lower bound" of our learning.

Briefly, the finetuning works in this way: during the train it performs the sequential SGT, each time using only the current data to update the current model; during the inference it uses just the last model.

Ensemble

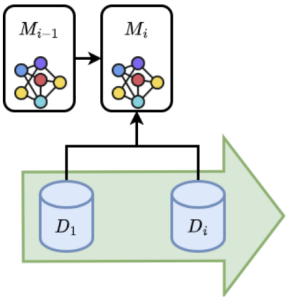
Another baseline is the Ensemble technique.
During the train, we train one model for each experience, so each model is completely independent. During the inference, we compute output using the correct model. Note that if the task labels (which are optionals) are not available we need an oracle.
This method can be used in a Task-aware scenario.

Joint Training



The Joint Training refers to the baseline that merges all the data (keeping task labels) and trains starting from a random initialization (or if we are using pretrained models, we start from the same initialization).
This method can be used in a Task-agnostic scenario.

Cumulative



The last baseline is the Cumulative one; it's similar to the Joint Training but in this case we don't start from a random initialization; accumulating all the data available up to now.
If we have enough data, starting from scratch (the Joint Training method) achieves a slightly higher performance than starting from the previous model.
This happens because in the other case the cumulative technique starts with low data and because of that then, he is unable to recover accuracy during training, but starting from the previous model achieves **faster convergence** than training from scratch.

This method can be used in a Task-agnostic scenario.

CL Strategies to fight catastrophic forgetting

Recall that catastrophic forgetting is the tendency of deep neural networks to completely and abruptly forget previous tasks or domains when learning a new one.

In order to deal the catastrophic forgetting so we have three main weapons:

1. **Buffer Replay**: store samples and revisit them.
2. **Regularization**: penalize forgetting.
3. **Parameter-Isolation/Architectural:** separate task-specific parameters

These strategies can be combined together.

CL Buffer Replay

The first and the easier strategy that we can use to fight the catastrophic forgetting is the **buffer replay.**

This strategy relies on storing a limited set of samples from the previous experiences and using them for rehearsal.

The good news is that this technique is simple, general and it works even in a CL scenario, because it approximates an i.i.d. distribution, and moreover it's relatively cheap in terms of computations.
The bad news is that it's not obvious that we can use it in every context, as we might have privacy or memory constraints. Furthermore, we have scaling problems as the longer the stream is, the more we need a larger buffer.

The pseudo-algorithm is really simple and it has only one parameter: the memory size.

1. <u>After each experience, we update the buffer.</u>
   The buffer update is performed by using an **insertion policy** to choose data from the current experience, add this experience to the buffer and then we use a **removal policy** if the buffer is too big.

2. <u>During training, finetuning and rehearsal</u>
   We sample from the current data, and sample from the buffer using a **sampling policy**. Then, we perform the SGD step on the concatenated mini-batch.

Regarding the policies, the insertion one finds the best example to store in the buffer, the remove one finds the least useful example to remove them from the buffer and the sample one finds the best examples to use for the current training iteration.

These policies might even be a random policy, but ideally, we want that the data inside the buffer should approximate an i.i.d. distribution.

CL Buffer Replay: Growing vs Fixed Memory

The first distinction we can make to classify different types of Buffer Replay is the type of memory.
In fact we can distinguish Buffer Replay with growing memory and fixed memory.

Growing Memory: in this case we assume that we have an unbounded growth of memory and therefore with each experience we add a number of instances in the buffer. Therefore the memory grows linearly with respect to time. In this case we do not need a *removal policy*.

Fixed Memory: In this case we instead assume a memory space limit M. We need a *removal policy*.

CL Buffer Replay: Real vs Synthetic Samples

Another distinction can be made by going to check the type of data that are saved in the buffer. We can, in fact, save real or synthetic data.

Real examples: in this case we save in the buffer data taken directly from the original dataset.
Synthetic examples: in this case values taken from a generative model are saved in the buffer.

This concept is also plausible from a neurological point of view but can be complex to implement.

CL Buffer Replay: Input vs Latent Replay

Finally, we can distinguish input replay and latent replay buffers.

In **latent replay**, instead of saving the original examples in the buffer, we are going to save a latent representation of an intermediate layer.
This is very useful since the original examples might be very heavy while the internal representations are much lighter. We will discuss Latent Replay in more detail later.

Random Replay

The first real buffering algorithm we study is Random Replay. In this algorithm all three policies (insertion, deletion, and sampling) are random.

After each experience we do sampling from the current experience, filling the buffer (of fixed size). The number of examples inserted and removed in the buffer is inversely proportional to the length of the stream.

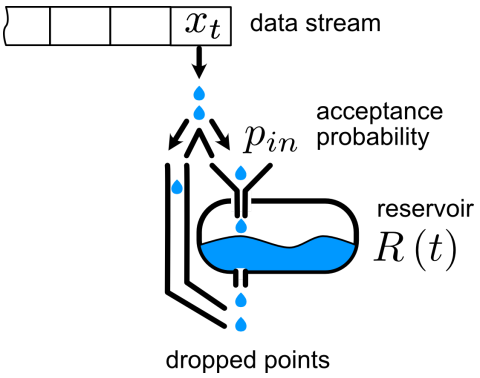
During the training, we concatenate the experience and buffer data, and on this concatenation, we perform sampling and then SGD.

However, this concatenation can create problems for us because if the sizes of the buffer and experience data are very different, we will no longer get an iid distribution. Because of this, we can improve this by sampling separately before concatenation.

This algorithm is very simple to implement and has very good performance in simple settings. Nevertheless, it has several problems:

1. At each step, we select a fixed number of data items to be added to the buffer. This does not work in a continuous scenario, since in a continuous scenario we usually have a very small mini-batch, sometimes as small as one item.
   In that case we have to choose:
   - we do not add that element to the buffer: but in this way the buffer is never updated.
   - we add that element to the buffer: but this way the buffer will become unbalanced toward the latest experience.
2. Ignore stream imbalance, if a class is over represented in the stream it will also dominate the buffer.
3. Does not require knowledge of task boundaries, which can be a problem if we need to know them.
4. Does not use task labels (we can solve this by dividing the buffer by task in a balanced way).

Reservoir Sampling



In order to fix the issues of the Random Sampling, we can use the **Reservoir Sampling**.
This sampling technique is a random sampling without replacement of N items from a stream of sample of length S.

It works in this two steps:

1. First, we have to fill the reservoir array R with the first k items of the stream.
2. Then, once we have the buffer filled, we replace items with respect to an **acceptance probability**.

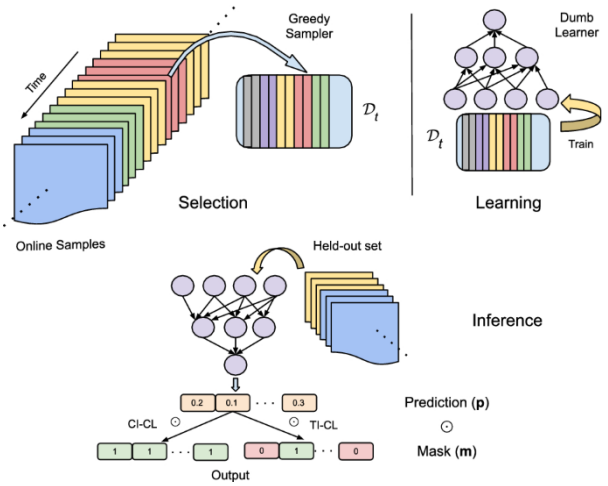
In this way it can work perfectly in a OCL setting but

1. It still ignores imbalance in the stream (If one class is much more frequent, it will be over-represented in the buffer).
   This can be still fixed by balancing the buffer capacity by class.
2. It doesn't need the task boundaries.

GDumb

This is another simple algorithm, often used as a baseline. It is made of three components:

1. Greedy Sampler
2. Dumb Larner
3. Masking.



During the training phase the greedy sampler store data inside the buffer balancing the classes, then a dumb learner is trained from scratch using only the data in the buffer.

Instead , during the inference phase we have a mask m. This algorithm classifies on the subset of labels provided by the mask.

In a class incremental scenario we mask only the unseen classes, while in a task incremental scenario we mask classes outside of the current task.

The greedy balancing sampler works in this way

1. When we have a new label:
    a. If the buffer is not full:
        i. Add the current sample to the buffer.

    b. If the buffer is full:
        i. We have to free some space by deleting a random sample from the largest class.

Despite its simplicity, this method is very competitive in the class-incremental setting.

This algorithm trains the model from scratch at each step, so there is zero knowledge transfer, for this reason the main limit of this method is the **latency**.

Moreover, since the model is trained only on a number of samples equivalent to the buffer size we have a **limited use of the data.**

MIR – Maximally Interfered Retrieval

Until now, the policy of sampling data from the buffer was random. This can be optimized with respect to the current data to minimize forgetting.
For this reason we introduce the **MIR algorithm** (Maximally Interfered Retrieval).
The MIR, instead of randomly selecting buffer elements, selects those that are negatively impacted by the current update, i.e., those that have higher interference.
The interference is calculated by comparing the loss obtained using the old parameters with the loss obtained with the updated parameters.

$$S_{MI-1}(x) = L(f_{\theta^v}(x), y) - L(f_{\theta}(x), y)$$

where, $S_{MI-1}(x)$ is the inference criterion, $L(f_{\theta^v}(x), y)$ is the loss calculated with the new weights and $L(f_{\theta}(x), y)$ is the loss calculated with the old weights.

In particular, θ^v are the model weights after a gradient descent step on the current data batch (X_t, Y_t).

$$\theta^v = \theta - \alpha \nabla L\,(f_{\theta}(X_t), Y_t)$$

The main limitation of the MIR strategy is the computational cost since it requires an additional forward pass to find the maximally inferred examples.

Latent Replay

As we already saw, perform the replay in the input space is inefficient with respect to perform the replay on the latent activations; for example, saving an image is very expensive rather than saving its latent activation.
The choice of which activation layer need to be picked is not obvious because there is the trade-off between the accuracy and the memory usage. In fact, the higher the layer, the more information is encoded, but the more memory is required to store the activations.

The algorithm is the following:
- We choose a layer.
- We store the latent representation of that layer in a buffer.
- We obtain the latent representation of the current batch, by passing the batch through the network up to the chosen layer.
- Concatenate the latent representation of the buffer with the latent representation of the current batch.
- Pass the concatenated latent representation through the network up to the output layer.

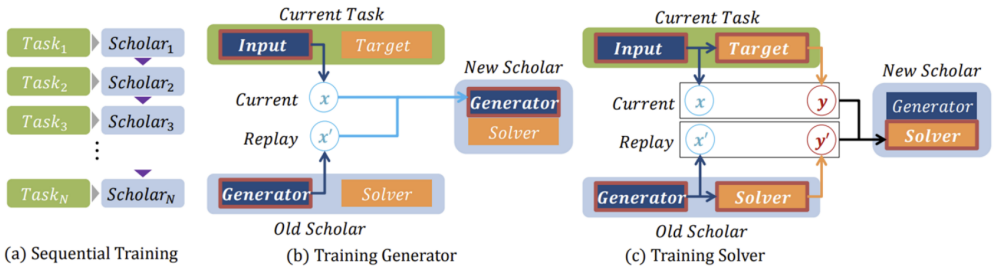
Low layers are trained early during training, they don't change much after a while; this means that we can freeze them at some point; this is called Freezing.

The Freezing technique is used to improve the latent replay, indeed, if we don't freeze the latent representations in the buffer, they will become obsolete over time.

Generative Replay

In order to solve the memory constraints of buffer methods we can use the Generative Replay technique. Using a generative model, we can just store the parameters of the model instead of the whole dataset and use the model to generate all the data we need. Moreover this is even biologically plausible, since the brain is able to generate data from memory.

The algorithm relies on the Generator and on the Solver, as any adversarial framework. We need to train both of them.
First, the new generator receives current task input x and generated inputs x' from the previous task. Real and replayed samples are mixed together and the generator learns to reconstruct cumulative input space, while the solver is trained to couple the inputs and targets drawn from the same mix of real and replayed data.
Here, the replayed target is the past solver's response to buffered input.



Actually, there are no algorithms that are able to train generative models continually, since generating data is noisy and this noise is accumulated during time.

An alternative could be to use the Knowledge Distillation technique with generative Samples.

Regularization

The buffer idea is a good and simple way to handle catastrophic forgetting but sometimes we cannot store the data for replay. And even if we can, the size of the buffer is limited, so it cannot contain all the information about the previous experiences.

For this reason we introduce **Regularization**, which answer two main questions:

1. Can we approximate the loss on the previous experiences without using the old data?
2. Can we model forgetting mathematically in order to prevent it?

We can distinguish three main categories of regularization:

1. **Constrained**
   These methods try to define forgetting as an optimization constraint.

2. **Prior-focused** (or prior-based)
   These methods use the previous model as a prior in order to calculate the posterior limiting the forgetting.

3. **Functional/Data Regularization**
   In the end we have these methods that replicate the behavior of the old model on the previous tasks while learning the new ones.

Regularization: Classic Regularization

In addition to the three main categories of regularization (Constrained, prior-based and Functional Regularization) we can still apply 'classical' way to regularize a model.

Early-Stopping, L1, L2 & Dropout

In order to mitigate forgetting we can still used Early-Stopping, L1/L2 weight decay or Dropout. This works because **the more we regularize, the less the model change, and so the model has less forgetting.**

Of course these regularizations alone are not enough to prevent forgetting and sometimes can make the situation worse.

In general, custom learning rate can work well and it is worth noting that we can recognize drifts using the value of the current loss. Indeed, a huge loss means that there will be a "huge" update and this is a symptom of catastrophic forgetting.

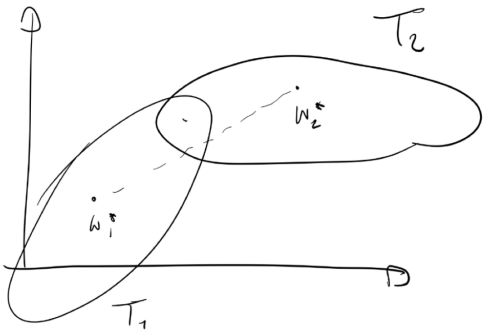
CL Loss Approximation

In offline training the model optimizes a single loss $L\,(D, \theta)$ that is constant during the entire training loop.
In a replay-free CL scenario (without a buffer replay), the model optimizes the actual loss ($L\,(D_t, \theta)$) but the true objective to minimize is over the whole stream $\Sigma_t\, L\,(D_d, \theta)$.

We need to ensure that the new minima remains a good solution for the old experiences.

This means that the first experiences need to learn a stable solution while later experiences must perform an incremental step that does not destroy the previously learned knowledge.
The two requirements are quite different, so the choice of optimizer, learning rate, batch size etc.. are critical.

Recall that by the term "*loss landscape*" we mean the graphical representation of the loss function (or objective function) in a multidimensional space.



The geometry of the loss landscape matters.
Let w_1 * be the solution after the first experience and w_2^* be the solution after the second experience; and $\Delta w = w_2^* - w_1^*$.
If the new solution is close enough we can linearly approximate the loss on the previous experiences.
The wider the curvature of the first task, the less the forgetting; but this depends on the eigenvalues of the Hessian at w_1^*, indeed, the forgetting can be measured as the difference between the losses of w_2^* and w_1 * , which can be approximated by the Hessian $\nabla^2 L_1(w_1 *)$ which has an upper bound that depends from the maximum eigenvalue λ_1^{max}.

$$F_1 = L_1(w_2 *) - L_1(w_1 *) \approx \frac{1}{2}\Delta w^T \nabla^2 L_1(w_1 *)\,\Delta w \leq \frac{1}{2}\lambda_1^{max} ||\,\Delta w||^2$$

So, we can follow three main ideas:

1. We can try to **change the model as little as possible**, but this is a strong assumption
2. We can try to **learn as much as possible in the very first task;** this means finding a point where we have to move around a little to learn new tasks.
3. We can try to **guide the optimization trajectory towards flatter minima**, because these are the points where the model can move the furthest without increasing the loss (precisely because they are flat!).

General rules for lr, batch size and scheduling

The general rule for choosing learning rate and batch size is the following:

- start with a high initial learning rate for the first task to obtain a wide and stable minima.
- for each following task decrease the learning rate but decrease the batch size.

Moreover we should prefer SGD to Adam.

Using Sparsity to prevent Forgetting

Sparse representations are useful for continual learning because they have several helpful properties.
One of these properties is the ability to separate tasks into independent subnetworks.
This means that subnetworks can work without interfering with each other, preventing forgetting.
Sparse representations are also very efficient in terms of memory, as they can store many networks with limited memory.
There are different types of sparsity:

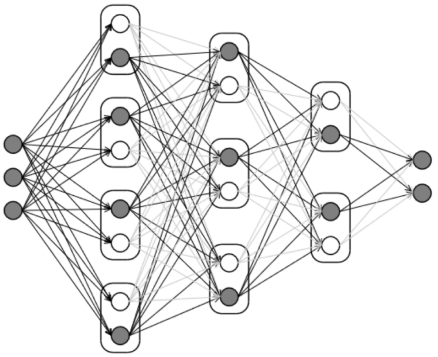
- weight sparsity
- activation sparsity
- gradient sparsity

which all contribute to the benefits of sparse representations for continual learning.

We can define two approaches to sparsity: Implicit and Explicit.
The first method is called implicit, which means using a regularization technique to encourage sparsity without explicitly building it into the network.
For example: Using L1, we encourage sparsity by making the absolute value of the weights small, while using L2 we encourage small weights without forcing sparsity.

Instead, the second approach, the explicit one, consists in architectural methods that build spare subnetworks (we will see these methods later).

LWTA : Local Winner Takes All



Local Winner Take All (LWTA) is a competition mechanism where only the highest activated neurons are selected and the others are masked to zero, in this way the "losers" have no impact. In this way we encourage sparsity.

The "L" in LWTA stands for "Local" meaning that each layer is split into separated blocks, ideally we want that each block corresponds to a single task, so in practice these blocks are very large.

Regularization: Functional Regularization

Another way to do regularization is called **Functional Regularization**.
This method relies on the fact that we have access to the previous model ${f_{i-1}}^{CL}$ that learned on previous experiences $S_{train}[1: i-1]$. This means that we can replicate the old model behavior and update it only on the new examples.

LwF: Learning without Forgetting (and Knowledge Distillation)

The Learning without Forgetting algorithm implements the Functional Regularization using the Knowledge Distillation as objective function.

Knowledge Distillation: The Knowledge distillation is a (general) technique where a small model (called student) is trained to reproduce the behavior of a larger model (called teacher) that is already trained. The small model is usually easier to train and faster to run. This method can help us in different scenarios, for example we can use it to reduce the size of a model.

The process of knowledge distillation can be splitted into the following steps:
- Train the teacher model on a large dataset.
- Train the student model on a smaller dataset.
- Calculate the soft targets for the student model.
- Train the student model to minimize the loss between its predictions and the soft targets.

The soft targets are calculated by taking the output of the teacher model and applying a softmax function to it. This converts the output into a probability distribution over the classes. The student model is then trained to minimize the loss between its predictions and the soft targets.

KL-Divergence: The KL objective function is the KL-Divergence between the teacher and the student, which is a similarity between two probability distributions. The KL-Divergence is defined as:

$$E(x|t) = -\Sigma \hat{y}_i(x|t)\, log\, y_i(x|t)$$

where $\hat{y}$ is the teacher and y student.
In alternative, we can use the MSE between the logits (often this is more robust in CL):

$$||\hat{y} - y||^2_2$$

Learning without Forgetting: Once we talked about KD, we can start with the Learning without Forgetting algorithm. LwF is a method that implements the distillation in a CL scenario (and uses the current data) to obtain the Functional Regularization.

This is an easy-to-implement and efficient method (Since it requires only an additional forward pass).

In detail, LwF was created for a Task-Incremental scenario (recall: that's the correspective Multi-Task learning of the non-Continual world), then it was extended to the Class-Incremental scenario.

LwF in Task-Incremental scenario: Starting from a pre-trained model, here we want to update the feature extractor (so, we will perform the finetuning of it) but we don't clearly want to forget anything. Recall that since it is a task-incremental scenario we will have a head for each task, and we are adding a new head for the new task.

During the new task training, the finetuning is applied everywhere, to the head just added, the feature extractor and even to each previous head.
Obviously the new head is trained for predict the class of the new data, but here the magic happens: the update of the previous heads is made in order to replicate the behavior (the values of the activations) of the previous tasks, this means that we are not forgetting the old knowledge.

This is achieved by using a distillation loss term that encourages the student network's predictions on the original task to match the predictions of the teacher network.

The final loss is made up by two terms:

- The Cross-Entropy computed on the new data using the new data (L_{new}), in order to understand how much we are learning on the new data.

$$L_{new}(y_n, \hat{y}_n) = -y_n \cdot log\, \hat{y}_n$$

- The Knowledge Distillation is computed on the old heads using the new data

$$L_{old}(y_o, \hat{y}_o) = -H(y_o', \hat{y}_o) = -\sum_{i=1}^{l} y_o'^{(i)} \, log\, \hat{y}_o^{(i)}$$

Then the two sub-loss are composed with a weight λ.

$$\lambda L_{old}(y_o, \hat{y}_o) + L_{new}(y_n, \hat{y}_n)$$

LwF in Class-Incremental scenario: here we changed the scenario, at each step we may add a new class to the single head.
This opens a problem: the classifier grows with time, and so we can't use the KL-divergence to compare two output probability distributions that have different sizes (since we are just adding a new class).

We solve this by computing the Knowledge Distillation over the old units using the new data (L_{old}) and the Cross Entropy over all the units using the new data (L_{new}); of course we need a way to distinguish between the **new** and **old units**.

Last considerations about LwF : the main limit of the LwF is that it need the **task boundaries** to know when to store the previous model.
Moreover, the KD is applied on the new data but the activations we want to replicate are calculated on the old data - this means that we will have some kind of drift because we want to replicate the activation of the i-th head on the i-th dataset but, after, we have no more the that dataset, so we can't handle that drift.

At the end, we have to state that Augmentation helps the KD, since we are interested just to replicate the old activations, so we can apply very strong augmentations.

Regularization: Prior-based Methods

The other family of regularization algorithms that we will study are called **Prior-based**.

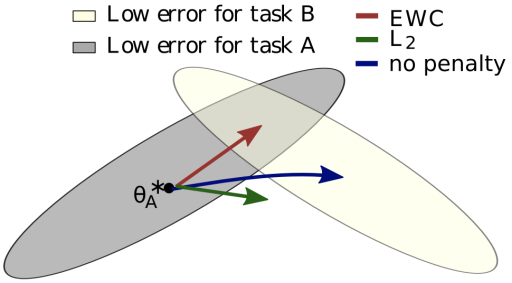
These methods use the previous model as a prior in order to calculate the posterior limiting the forgetting; the main idea is that **important weights should not change**, we can just change only the unimportant weights. The reason why is a strong assumption: deep NN are overparameterized but each task has a small number of important weights. Now the question is "*how do we find the important weights?*"
This question can be replied with the *Fisher Information*.

Fisher Information: Given a probability distribution $f(\theta)$, the Fisher Information $I(\theta)$ measures how sensitive the distribution is with respect to changes to each parameter. It is easy to compute since we only need the gradients, indeed, a high gradient value for a parameter means its significant impact.

$$I(\theta) = \mathbb{E}\left[\left(\frac{\partial}{\partial \theta}\ln f(x(\theta)\right)^2\right]$$

Prior-based Methods: EWC

Suppose to have two tasks (A and B) and we have a model already trained to solve task A, this method ensures task A is remembered whilst training on task B.



The **Elastic Weights Consolidation** (EWC) is a method that uses the Fisher Information as importance value in its loss function that will be minimized (the Fisher coefficients are used as weights for each parameter).

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^*_{A,i})^2$$

Where $\theta^*_{A,i}$ are the parameters after learning the first task, $L_B(\theta)$ is the loss for task B only, λ sets how important the old task is compared to the new one and labels each parameter are denoted with the i.

If we take gradient steps according to task B alone (blue arrow), we will minimize the loss of task B but destroy what we have learnt for task A.
On the other hand, if we constrain each weight with the same coefficient (green arrow) the restriction imposed is too severe and we can only remember task A at the expense of not learning task B. Instead, EWC, finds a solution for task B without incurring a significant loss on task A (red arrow) by explicitly computing how important weights are for task A.

In the original version of the paper, a matrix containing Fisher's information was stored for each task. Obviously, saving all these matrices has a high cost that grows linearly with the number of tasks, this is called “separate mode”.
An alternative version allows the Fisher to be estimated using only the previous one, updating it with an average or exponential moving average, “online mode”. In other words, `*separate*` keeps a separate penalty for each previous experience, while `*online*` keeps a single penalty summed with a decay factor over all previous tasks [ref].

Limits of EWC: the main limitations of EWC models is that it requires a trained model to estimate the Fisher Information, moreover it requires the task boundaries and it is only applicable to batch scenarios, so we have no ways to implement it into an online setting.

Prior-based Methods: Synaptic Intelligence (SI)

An alternative algorithm to EWC that is able to work even in the online setting is called **Synaptic Intelligence (SI)**. This algorithm is similar to EWC but it uses different importance values.

The idea behind this algorithm is that the importance of a parameter is proportional to its contribution to the loss change for an infinitesimal step.

The problem remains how to measure this importance: let $\theta(t)$ be the weights at time t, $\delta(t)$ the small change in weights and $g_k(t)$ the gradient for the weight k; we know that the change in the loss is approximated, for small steps, using the gradient:

$$L(\theta(t) + \delta(t)) - L(\theta(t)) \approx \Sigma_k \, g_k(t) \, \delta_k(t)$$

the total contribution to the loss change is by the sum of weight changes, weighted by the gradient, over the entire training curve:

$$\int_{t^{\mu-1}}^{t^{\mu}} g(\theta(t)) \cdot \theta'_k(t) \quad dt = -\sum_k w_k^{\mu}$$

where μ is the new task, and w_k^{μ} is the importance of the weight k for the task μ (Note that the original paper uses infinitesimal steps, for this reason we used integrals).

The importance w_k^{μ} means how much the parameter k contribute to the decrease in the loss for the task μ.

The Loss function used here is quite similar to the one used in EWC but here we have different importance values:

$$\tilde{L}_{\mu} = L_{\mu} + c \, \Sigma_k (\Omega_k^{\mu} (\tilde{\theta}_k - \theta_k)^2)$$

where c is an hyperparameter which regulates the strength and Ω_k^{μ} is the normalized coefficient.

In particular, Ω_k^{μ} is calculated using the values of importance of the weight k on the previous tasks $v < \mu$ normalized using $\Delta_k^{v} = \theta_k(t^{v}) - \theta_k(t^{v-1})$ that is the value that means how far θ_k moved during training of the task v, (t means the time).

$$\Omega_k^{\mu} = \Sigma_{v<\mu} \frac{w_k^{v}}{(\Delta_k^{v})^2 + \xi}$$

where ξ is just a little value to avoid zero in the denominator.

At the end, SI needs task boundaries to compute Δ_k^{v} and $\tilde{\theta}_k$.

Regularization: Constraints (Gradient Episodic Memory)

The last category of regularization methods is the method that relies on constraints: these methods try to model the interference mathematically, in this way we can modify our learning step in order to not interfere with the previous knowledge.

We will use the gradient-based definition of **interference**, based on the cosine similarity (that is we will compute the gradient for the new and old tasks and then we compute the cosine similarity between that gradients) :

- **negative cosine similarity** means that there will be an **interference**, so the Loss on the old task will increase;
- **positive cosine similarity** means **positive backward transfert** and so the Loss on the old task will not increase.
- **cosine similarity == zero** means that are orthogonal tasks, without interference.

The only method we will see for this section is called **Gradient Episodic Memory (GEM)**.

This method works in the online continual learning setting (so, even here no boundaries are needed) and this method uses a balanced buffer (by task) — M_t. Actually, the buffer is needed just to calculate the gradients.

Using the definition of gradient-based of interference, this algorithm first calculates the gradient g on the new data, then we want to solve the following **constrained optimization problem**, whose constraint is the non-interference:

$$min_{\tilde{g}}[\ 1/2\ \ ||g - \tilde{g}||_2^2\ ] \text{ subject to } < \tilde{g}, g_k > \geq 0$$

note: $1..k$ are all the previous gradients

In other words, we want to find the nearest update to g that have no interface to all the previous tasks.
The algorithm is the following, the $project()$ method resolves the constraint problem.

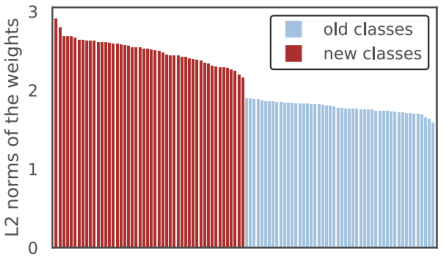
procedure TRAIN(f_θ, Continuum$_{\text{train}}$, Continuum$_{\text{test}}$)
  $\mathcal{M}_t \leftarrow \{\}$ for all $t = 1, \ldots, T$.
  $R \leftarrow 0 \in \mathbb{R}^{T \times T}$.
  for $t = 1, \ldots, T$ **do**:
    for (x, y) in Continuum$_{\text{train}}(t)$ **do**
      $\mathcal{M}_t \leftarrow \mathcal{M}_t \cup (x, y)$
      $g \leftarrow \nabla_\theta \ell(f_\theta(x, t), y)$
      $g_k \leftarrow \nabla_\theta \ell(f_\theta, \mathcal{M}_k)$ for all $k < t$
      $\tilde{g} \leftarrow$ PROJECT$(g, g_1, \ldots, g_{t-1})$, see (11).
      $\theta \leftarrow \theta - \alpha\tilde{g}$.
    end for
    $R_{t,:} \leftarrow$ EVALUATE$(f_\theta$, Continuum$_{\text{test}})$
  end for
  return f_θ, R
end procedure

Classifier Bias

In an offline learning scenario we can observe that low layers learn early and learn low-level features and they do not change much after the first epochs. Instead, the high levels tend to be more task specific.

This behavior leads us, into the continual learning world, to the fact that **most forgetting happens in the higher layers**. Indeed, by looking for the similarity between layers we can see that, after the train on a second task, the higher layers change significantly while the lower layers don't change much.

The classifier bias is the tendency of a (class-incremental) algorithm to favor new classes over others when learning from new data in an incremental manner; in practice this happens because new classes have weights with larger norms.



If we don't have task labels and we encounter new classes over time, and we don't have replay, the DNN suffers from classifier bias that is the biggest source of forgetting in a naive method.

We can deal this bias with:

- Replay methods (already presented)
- Training the classifier with algorithms that have less classifier bias (LDA)
- Classifier Normalization (LUCIR)
- Classifier finetuning/adaptation (CWR)

(The acronyms are the methods we will see in this section).

A simple trick to mitigate the forgetting

A simple trick to mitigate the forgetting consists of:

1. use a pretrained model
2. freeze lower layers after the first learning experience
3. finetune the classifier, preferably with a buffer replay

Deep SLDA

Deep SLDA [ref] is a model for efficiently training a CNN in an online scenario.
It relies on extracting from a fixed pretrained model G (e.g. ImageNet) its output layer, denoted by F. This model will train only the output layer F by using Streaming Linear Discriminant Analysis (SLDA), that is an online extension of LDA.
To make predictions, SLDA assigns the label to an input of the closest Gaussian computed using the **running class means** and **covariance matrix**.

SLDA is resistant to catastrophic forgetting because its running means for each class are independent, which directly avoids the stability-plasticity dilemma.
While the covariance matrix can change over time and is sensitive to class ordering, the changes to it result in, at most, gradual forgetting.
So, the whole model can be seen as:

$$F(G(\mathbf{X}_t)) = \mathbf{W}\mathbf{z}_t + \mathbf{b}$$

where $z_t = G(X_t)$, G is the fixed pretrained part and F is just the last fully-connected layer.

Let K is the total number of categories and d is the dimensionality of the data,SLDA stores one mean vector per class $\mu_k \in R^d$ with an associated count $c_k \in R$ and a single shared covariance matrix $\Sigma \in R^{d \times d}$.
When a new datapoint (z_t, y) arrives, the mean vector for the class y at time t and associated y-th counter are updated as:

$$\boldsymbol{\mu}_{(k=y,t+1)} \leftarrow \frac{c_{(k=y,t)}\boldsymbol{\mu}_{(k=y,t)} + \mathbf{z}_t}{c_{(k=y,t)} + 1}$$

$$c_{(k=y,t+1)} = c_{(k=y,t)} + 1 \ ,$$

Note that the mean vector uses the result of the fixed pretrained model z_t and the previous mean.
After this, the model need to estimate the covariance matrix $\Sigma \in R^{d \times d}$, this because calculate the whole full covariance matrix is too expensive (n^2 parameters).

$$\boldsymbol{\Sigma}_{t+1} = \frac{t\boldsymbol{\Sigma}_t + \Delta_t}{t+1} \qquad \Delta_t = \frac{t\left(\mathbf{z}_t - \boldsymbol{\mu}_{(k=y,t)}\right)\left(\mathbf{z}_t - \boldsymbol{\mu}_{(k=y,t)}\right)^T}{t+1}$$

To compute the predictions, the model use a linear classifier whose weights are computed by the inverse of the covariance matrix (called the precision matrix, Λ) and the running means in this way:

$$w_k = \Lambda\mu_k$$

$$F(G(\mathbf{X}_t)) = \mathbf{W}\mathbf{z}_t + \mathbf{b}$$

In conclusion, this algorithm helps to fight the classifier bias because working with means and covariance it is independent from the order.

Classifier Normalization (LUCIR)

Another method to handle the classifier bias is the Classifier Normalization introduced into the *Learning a Unified Classifier Incrementally via Rebalancing* paper.

In this work, to mitigate the bias the authors normalize the weights of the classifier using the cosine normalization in the last layer:

$$p_i(x) = \frac{\exp(\eta\langle\bar{\theta}_i, \bar{f}(x)\rangle)}{\sum_j \exp(\eta\langle\bar{\theta}_j, \bar{f}(x)\rangle)},$$

where p_i is the probability for the class i.

Copy Weights with re-init (CWR)

The last method to fight the classifier bias is called Copy Weights with Re-init (CWR). This method relies on the "fast" and "slow" weights system: in practice we have two sets of weights:

- **consolidated weights** *cw*
- **temporary weights** *tw*

The consolidated weights are used for inference while the temporary weights are used for training, after each learning step the temporary weights are resetted to zero.

After each learning experience, the method renormalizes the consolidated weights with a weighted sum of all the learners weights.

Architectural Methods

Another way to deal the catastrophic forgetting is by modifying the architecture of the model at runtime.
In this section we will see the Progressive Neural Networks and the Expert Gate.

Architectural Methods: Progressive Neural Networks (PNN)

The Progressive Neural Networks (PNN) is a model where catastrophic forgetting is prevented by instantiating a new column for each task. This method assumes to know the task labels.

In order to obtain transfer of knowledge, the new columns are connected to all the previous ones via adapters, which are needed because they enable the network to leverage the learned representations from earlier layers.
Adapters serve as a mechanism to adapt the features of the current layer to the features of the previous layers, allowing the network to transfer knowledge and leverage the existing representations.

After training the column is frozen.

During inference, the model uses the task labels to activate the correct columns.

The advantage of this model is that there is a **good forward transfer** since each task can re-use the previous columns and it **inhibits forgetting** by freezing columns.
The bad things is that it requires the task labels and moreover it has a poor scaling in memory size due to the quadratic nature of the adapters since each column is connected with all the previous ones.

Architectural Methods: Expert Gate

The previous model relies on the fact that we had the task labels; but we can remove that need by inferring the task labels with a classifier.
Often predicting the task label is easier than predicting the class, for example: identifying a language (task inference) is easier than predicting the next word of an incomplete sentence (solving the task).

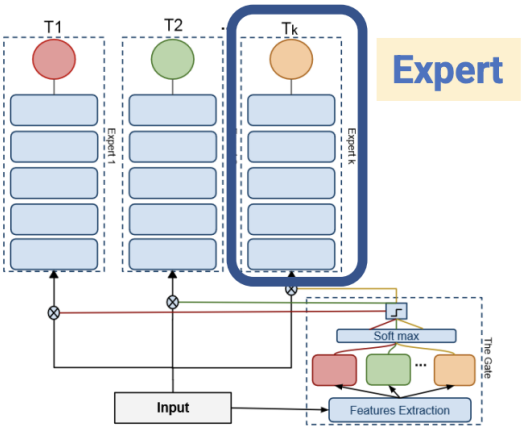


Figure 1. The architecture of our Expert Gate system.

A model that uses the task inference is the **Expert Gate**, this is made up by three components:

1. the **modular network**
2. the **task predictor**
3. the **gates**

The modular network is composed by a set of **experts** which are single modules trained on a single task.

The gate is a component that is able to enable and disable an expert of the network; this component is made up of a set of undercomplete auto encoders, one for each task.

Each autoencoder is trained along with the corresponding expert model and maps the training data to its own lower dimensional subspace.

During the inference, the model uses the expert associated with the most confident autoencoder by looking for the reconstruction error for each task, this is crucial because here we don't need more the task labels.

During the train, autoencoders can also be used to evaluate task relatedness at training time, which in turn can be used to determine which prior model is more relevant to the new current task.
Expert Gate can decide which expert to transfer knowledge from when learning a new task and whether to use fine-tuning or learning-without-forgetting.

Algorithm 1 Expert Gate

Training Phase input: expert-models $(E_1, ., E_j)$, tasks-autoencoders $(A_1, ., A_j)$, new task (T_k), data (D_k) ; output: E_k

1: A_k =train-task-autoencoder (D_k)
2: (*rel,rel-val*)=select-most-related-task(D_k,A_k,{A})
3: **if** *rel-val* >*rel-th* **then**
4: E_k=LwF(E_{rel}, D_k)
5: **else**
6: E_k=fine-tune(E_{rel}, D_k)
7: **end if**

Test Phase input: x ; output: prediction

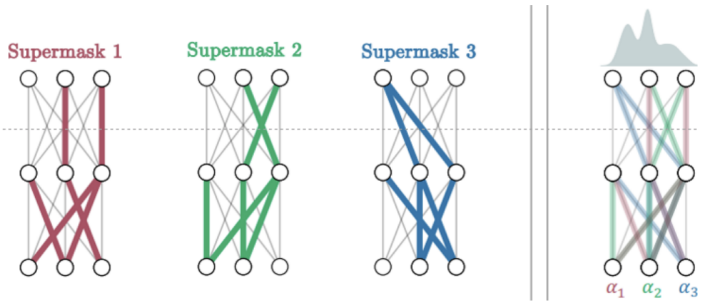
8: i=select-expert($\{A\}, x$)
9: prediction = activate-expert(E_i, x)

The whole model can be seen as a pure ensemble, and this means no knowledge transfer!
Other limitations are:

1. an pretrained autoencoder is required
2. the reconstruction error is not always a good metric for a task predictor, indeed if the tasks are similar the reconstruction error can be very low.

Masking

We know that deep neural networks are overparameterized.
We can deal with this problem with the **masking** technique.



This technique consist of using a large fixed network and select a subset of units for each task.

The main advantage is that we can achieve similar results to the modular networks but in a less expensive way.
Moreover this technique induces sparsity.

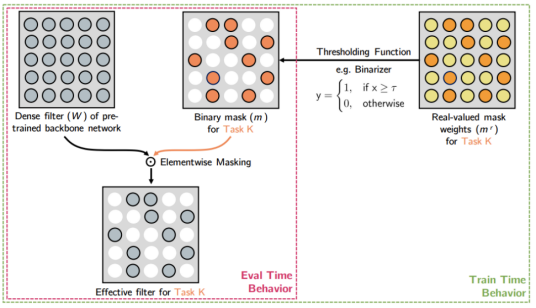
This technique relies on the **lottery ticket hypothesis**, which is just an hypothesis not a formal theorem, that states:

> dense, randomly-initialized, feed-forward networks contain subnetworks (*winning tickets*) that—when trained in isolation–reach test accuracy comparable to the original network in a similar number of iterations

The mask are implemented with binary masks which are easy to compress but we can not compute gradients on them, so we can't apply the SGD directly.
In the algorithm we will see in this section we assume to have some optimization algorithm that is able to be trained with binary masks.

Masking: Piggyback

Piggyback is our first masking method, that is a task-aware method. It relies on a fixed pre-trained model (called "backbone") which remains freezed.



For each task the model trains a binary mask, where "train" mean train as usual (with float values) and then binarizes the weights with a thresholding function, after this we apply the SGD.

Since each binary mask is independent, we have zero forgetting but also zero knowledge transfer.

Masking with Pruning: PackNet

A different method to find a mask is by pruning; the **Magnitude Pruning** technique is the following:

1) starting from a dense network
2) sort the weights layer-per-layer by their absolute magnitude
3) cut the lowest $p\%$

A little variation of this technique is called **Iterative Magnitude Pruning** where the process is repeated multiple times.

A model that implements this technique is called **PackNet** which is a task-aware method where the training starts from a pretrained model (the backbone) and for each task it finetunes the weights of the dense network, then prune a certain fraction of the weights an reapply the training on half of the epochs, at the end the task parameters are frozen.

Instead, for the inference time, it uses the task labels to choose the mask.
This is $1.5x$ more expensive than finetuning but, compared to Piggyback, here, we can have forward trasfert since the weights are trained (while in Piggyback the are kept fixed).

Masking: HAT

The *Hard Attention to the Task* (HAT) model is another task-aware method where we mask neurons instead of weights.

In particular, **it learns a soft attention mask over the units for each task**; during forward it used the mask to mask units and in backward it uses them to mask gradient.

The soft masks are a way to weigh the unit with a value between zero and one,
- please, note that Piggyback and Packnet use hard mask (which are binary).

As usual, masks are created to prevent large updates to the weights that were important for previous tasks.

The attention coefficient is based on a scaling parameter s which regulates between uniform weights and hard gates (the higher is s the stronger is the binarization).

In the backward step,there is a **cumulative attention vector** which keeps track of how much each unit is used by the previous task.
Used units should be protected from large changes, so there is the **gradient masking** which scales down the gradient for already used units.

Masking with Pruning: SupSup

SuperMasks in Superposition (SupSup) is the last method we will see. It combined hard masks with fixed backbone with random weights (relying on the Winning Ticket Hypothesis).
SupSup add two main contributions:

1. Using random weights we don't even need to store them, we can just save the seed.
2. Instead of using a single mask, we can use a weighted sum of them (called "*superpositions*").

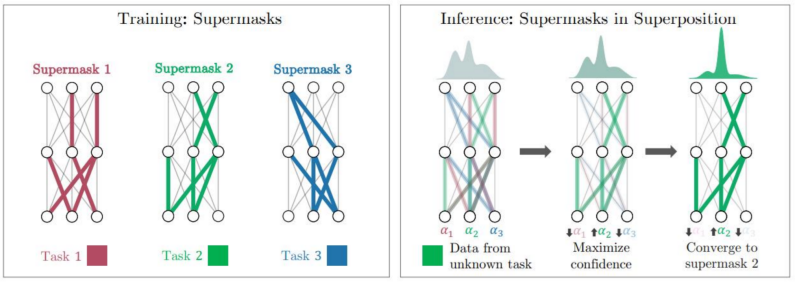


Figure 1: **(left)** During training SupSup learns a separate supermask (subnetwork) for each task. **(right)** At inference time, SupSup can infer task identity by superimposing all supermasks, each weighted by an α_i, and using gradients to maximize confidence.

After the training of the masks [5], we have one mask (that is one model) for each task. Now, in a task-agnostic setting, we can choose the task label selecting the most confident model by looking at the mask which outputs (through the network) the smallest entropy, which, in general, measures the confidence.

This is a naive way to implement the mask, indeed, the computational cost is a forward pass for each mask, but we want an algorithm that scales better with the number of tasks.

For this reason the authors introduced the superposition concepts: a weighted sum of all the masks. This is an approximation of the ensemble output which requires only a single forward pass.

The algorithm for the one-shot task inference is:

1) compute the output of the model $p(\alpha)$, where $\alpha \in [0 \dots 1]$ that is a vector for each mask that can be interpreted as the "belief" that supermask M_i is the correct mask .
2) compute the gradient with respect to the entropy
3) update α_i with SGD
4) choose the mask such that $arg \max_i \left(\frac{\partial H(p(\alpha))}{\partial \alpha_i}\right)$

[5] We will not see the algorithm for training them

In a task-agnostic scenario we don't have boundaries between tasks, this means that we need a mechanism to deduce when to start the training of a new mask.
Here again we can reuse the concepts of superpositions and entropy.

If all the coefficients α are uniform, the task is new, otherwise if some coefficients dominate all the other we can assume it is an old task.

Conclusion of masking

Transfer: how a learned task affects learning other tasks (can be forward/backward, positive/negative)

- **Fixed weights + independent masks** (Piggyback, SupSup)
  - Zero forward transfer
  - Zero forgetting
  - A pure ensemble of models (very efficient in space occupation)
- **Trainable weights + hard masks** (Packnet)
  - Forward transfer because new masks can reuse units **learned** on the previous tasks (in Piggyback and SupSup the backbone is fixed at initialization)
  - Zero forgetting
  - Similar to PNN, but more efficient in space
- **Trainable weights + soft mask** (HAT)
  - Forward transfer and possibly backward transfer since previously used units are not frozen.
  - Forgetting is possible
  - Soft gates may be harder to train (see HAT tricks about gradient flow)
  - Requires more space than hard gates (32 bit vs 1 bit before compression)

- Chapter 4 : Frontiers of CL

Introduction to Distributed and Federated learning

Every computational machine can be divided in three categories:

1. **Cloud**: main giant servers with thousands of nodes and massive computational resources, always viable and with high bandwidth and low latency.
2. **Fog**: intermediate servers, with a low usage.
3. **Edge**: our devices, billions of nodes with low computational resource. They are a prolific data source.

Here we can define two approaches to machine learning that involve training models across multiple devices: distributed learning and federated learning.

Distributed learning refers to the process of training a model using data stored on multiple devices. In this approach, each device processes a subset of the available data locally and exchanges information with other devices to collaboratively train a shared model. The key idea is to leverage the computational power of multiple devices while keeping the data decentralized.

In federated learning, the training process takes place directly on the user's device, eliminating the need to transmit raw data to a central server. The central server only coordinates the learning process by aggregating and updating the models based on the locally computed updates from the devices.

Distributed learning

The motivation behind the spread of Distributed Learning is very simple: models are getting bigger and bigger. The best GPU in the world, the NVIDIA A100, will take 3 millions of hours to train the whole GPT-3, due to its 175 billion parameters.

We can define two way to perform Distributed learning:

1. Data parallelism
2. Model Parallelism

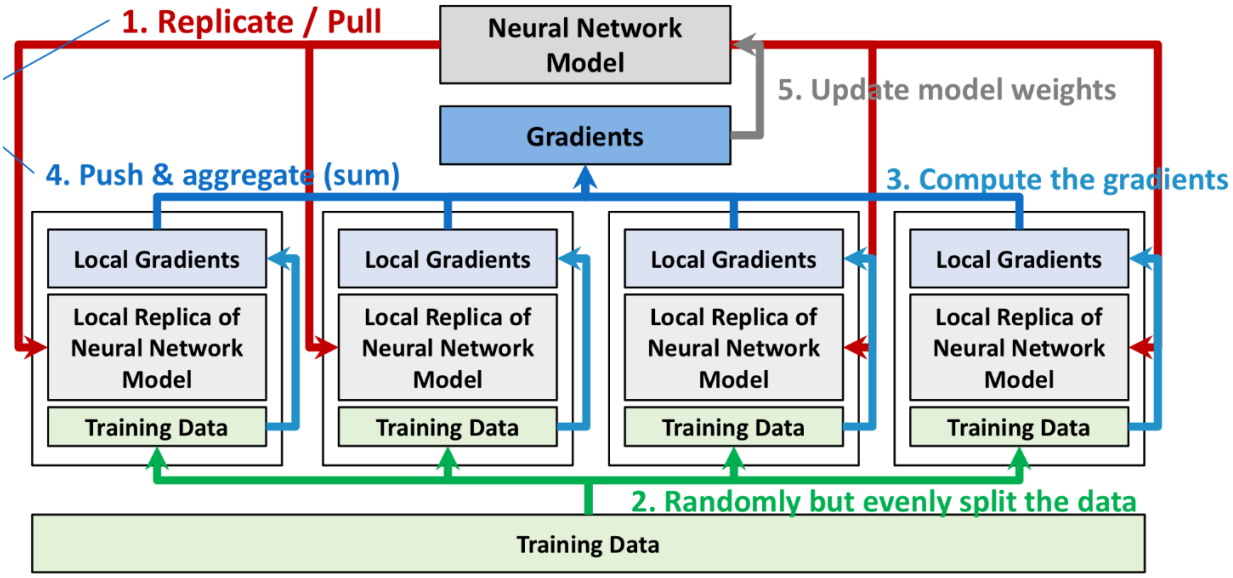
We will first talk about Data parallelism.

A first way to perform Data parallelism is replicating the model across all the devices. Another way is to split the dataset into n chunks and perform the train independently of each chunk on a different machine.

Both of these methods replicate the model having different training processes, but we want to perform the training of a single model.
The solutions for Data parallelism are: **Parameter Server Strategy** and **Mirroring Strategy**.

Distributed learning: Parameter Server Strategy

In this strategy the server is the central controller of the whole training process. It receives the gradients from workers, updates the (global) neural network and then sends back the aggregated results. On the other side, the workers compute the (local) gradients using a splitted dataset and a (local) replica of the neural network.
The local replica of the NN of the workers replicates the model of the server.



The main problem here is the **overload**, the bandwidth required is too much, we need a way to delete the centralized server. For this reason we introduce the Mirroring Strategies.

Distributed learning: Mirroring Strategy

In order to deal with the overload problem, we have the Mirroring Strategies.
We have different ways to implement it:

Naive All-Reduce: In this strategy, each worker sends its local gradients to all other workers, and each worker sums up the received gradients to compute the updated global gradients. The time complexity for this approach is O(N) since each worker communicates with N-1 other workers, and the bandwidth required is also O(N).

Ring All-Reduce: In this strategy, workers are organized in a ring topology. Each worker sends its gradients to the next worker in the ring, and the gradients circulate around the ring until they reach the original sender. Each worker updates the received gradients and passes them along to the next worker. The time complexity for this approach is O(N) since the gradients traverse the ring N times, and the bandwidth required is O(1) since each worker only communicates with its neighbors.

Parallel All-Reduce: In this strategy, the workers are divided into multiple groups, and each group performs All-Reduce independently. The gradients from each group

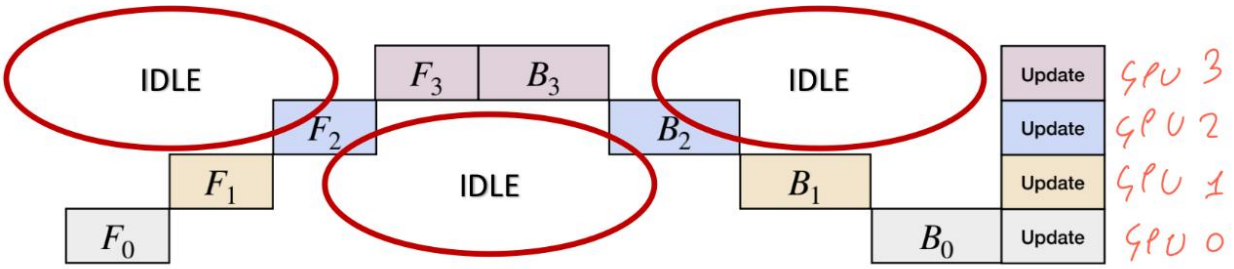
are then combined to compute the updated global gradients. The time complexity for this approach is O(1) since each group performs All-Reduce in parallel, but the bandwidth required is $O(N^2)$ since each worker needs to communicate with all workers in other groups.

The best of Parallel and Ring All-Reduce: In this strategy combines the advantages of parallel and ring All-Reduce methods by using a tree-based topology. Within each subtree, a ring-like pattern is followed. This approach achieves an optimal balance between time complexity (O(log N)) and bandwidth requirements (O(1)) since each worker communicates only with its parent and children in the tree structure.

Distributed learning: Model Parallel Distribution Learning

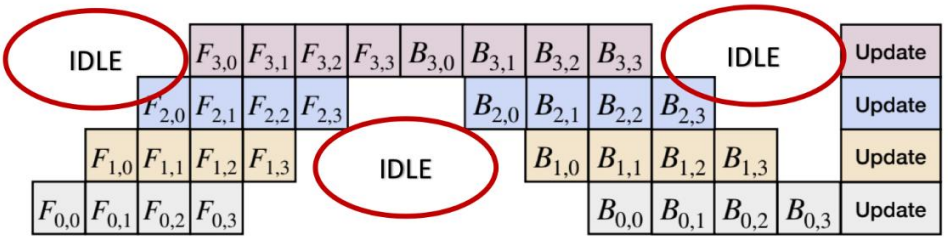
Using data parallelism strategies we have a better device utilization but one GPU is still one GPU, and for large models is not useful, because for example GPT-3 is 350Gb and the NVIDIA A100 is 80Gb: this means that even the best GPU cannot fit the whole model into memory.

For this reason we need to split the model between different GPUs, this technique is called **Model parallelism**, and a possible way to implement it is to use a GPU for each layer.
Assuming to have a 4-layer NN which is deployed across four different GPUs with model parallelism. Hence, we will have F_0 .... F_3 forward passes and B_0 .... B_3 backward passes.If we perform F_1 after F_0 (and so on...) naively, we have a lot of idle time.



Training Timeline

For this reason, we split the data into mini-batches and feed them in a stream.



However, utilization **improves to 57%**

We can do this because the model parameters do not change during the computation of a single batch. However the size and the number of the mini-batches are crucial parameters: if we increase them we have a higher throughput while potentially increasing the memory requirements and communication overhead.

Distributed learning: the problem of communication

Distributed learning relies on communication: we need a way to send and receive data (mainly gradients) among the computational resources and this is an overhead that needs to be taken into account.

In order to optimize the communication of gradients we can perform two methods: Pruning or Quantization.

Gradient Pruning: the idea is quite simple, here we send only the gradients with a high magnitude, setting to zero the other gradients, in this way we increase the sparsity. Before setting to zero the "little" gradients we store them into another matrix and we accumulate them.

In simple scenarios (MNIST, for example) this technique can speed up the training even by a 1.5x, but for modern models we have a loss in performance.
This happens because of the momentum; indeed, the accumulation change the direction of the update. The solution is to calculate the momentum on the previous step.

Gradient Quantization: here instead, we split the gradient with respect to the sign of each element of the matrix, in particular from the matrix g we obtain another matrix $g^{quantize}$ where it is a binary mask of the sign plus a quantization error matrix Δ_t.
A better version is achieved using two thresholds $-\tau, \tau$: in practice we split the gradient matrix in three intervals: $g_i <- \tau$; $-\tau < g_i < \tau$; $g_i > \tau$.

Federated learning

Federated learning is a machine learning setting where multiple entities (called clients) collaborate in solving a machine learning problem. **Each client's raw data is stored locally** and not transferred; instead, focused updates intended for immediate aggregation are used to achieve the learning objective.
The motivation behind this approach is **privacy**.

We cannot send even the gradients because in a certain way they contain informations about the data (and of course that's a lot of data).

Federated learning comes out with four challenges:

1. **Statistical Heterogeneity** : we cannot make assumptions on how the data are distributed across clients because each client has its own dataset, hence different models has very different gradients.

2. **Communication Efficiency** : we are leveraging the resources over the whole infrastructure, this means that an efficient communication is crucial. This can

be done with pruning or quantization (we cannot send raw gradients!)

3. **System Heterogeneity** : we cannot make assumptions even on the resources of the devices which join the infrastructure (it can be a very old Android phone, for example); so we need to handle the different calculation time of each device.

4. **Privacy and security** : data anonymization is the explicit goal of FL.Problem formulation of Federated learning

"Learning a global model" consists in minimizing global loss between each client:

$$F(x) = \mathbb{E}_{i \sim P}[F_i(x)] \qquad \text{(global loss)}$$

where P is the client distribution but due to the *System Heterogeneity* it is unobservable. On the other hand, the single local loss $F_i(x)$ relies on the local data distribution which is unobservable due to *Statistical Heterogeneity.*

$$F_i(x) = \mathbb{E}_{\xi \sim D_i}[f_i(x, \xi)]$$

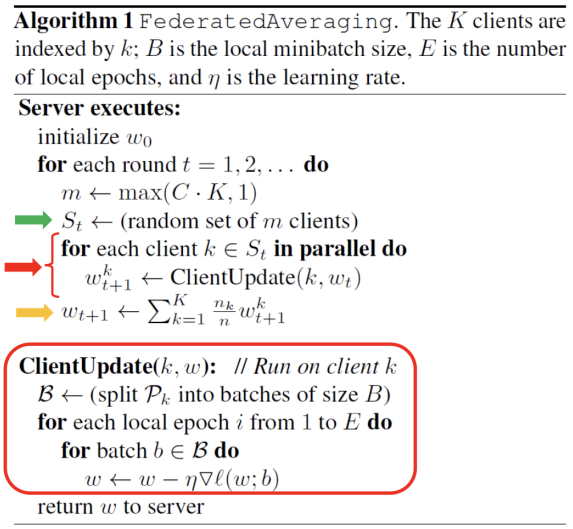
So, since we cannot observe them we try to approximate the learning problem by Empirical Risk Minimization.

Federated Averaging

The baseline of each federated algorithm is the FederatedAveraging which works in this way:

1. for each round of training, the server randomly select m clients
2. the server push to each client (in parallel) the current weight of the model w_t
3. each client receives the weights and locally updates its weights
4. when each client terminates the training loop, sends its weight to the server
5. here there is the magic: assuming to have a lot of device this algorithm is able to approximate the system and statistical heterogeneity with a simple average like:

$$w_{t+1} = \Sigma_{k=1} \frac{n_k}{n} w^k_{t+1}$$



A general (optimized) framework for Federated Learning

The idea introduced by the Federated Averaging paper led to a generalization of it: the FedOpt framework. This algorithm is a quite general framework which relies on four components:

1. **Client Sampling (1)** : how to sample the subset of clients
2. **ClientOPT (2) :** how to perform the local update

3. **Aggregation and ServerOPT (3)**: once received the gradients, the server need a way to aggregate them and perform the update of the global model
4. **Communication efficiency (4)**: how to broadcast the input to all clients and how to manage the receivement of the gradients of each client from the server.

Each component can be implemented with different algorithms, let's see some of them.

Algorithm 1: FedOpt

Input: Initial model $\mathbf{x}_0$; ClientOpt, ServerOpt with learning rates η, η_s

for $t \in \{0, 1, \ldots, T-1\}$ **do**

  Sample a subset $\mathcal{S}^t$ from clients $\mathcal{C}^t$; } 1

  Broadcast $\mathbf{x}^t$ to all clients $i \in \mathcal{C}^t$; }

  for client $i \in \mathcal{S}_t$ **in parallel do**

    Initialize local model $\mathbf{x}_i^{t,0} = \mathbf{x}^t$;

    for $k \in \{0, \ldots, \tau_i - 1\}$ **do**

      Compute local stochastic gradient $g_i(\mathbf{x}_i^{t,k})$;

      Perform local update $\mathbf{x}_i^{t,k+1} = \text{ClientOpt}(\mathbf{x}_i^{t,k}, g_i(\mathbf{x}_i^{t,k}), \eta, t)$; } 2

    end

    Compute local model changes $\Delta_i^t = \mathbf{x}_i^{t,\tau_i} - \mathbf{x}_i^{t,0}$;

    Send Δ_i^t to server; } 4

  end

  Aggregate local changes $\Delta^t = \sum_{i \in \mathcal{S}^t} p_i \Delta_i^t$;

  Update global model $\mathbf{x}^{t+1} = \text{ServerOpt}(\mathbf{x}^t, -\Delta^t, \eta_s, t)$ } 3

end

Client Sampling: the only algorithm we will see for Client Sampling is called **FedCS** (Federated Client Sampling) which deals with the system heterogeneity by solving an optimization problem.

ClientOpt: here we will see two algorithms: **FedProx** and **SCAFFOLD**.
The first one, FedProx, in order to deal with statistical heterogeneity tries to avoid the divergence of clients by introducing a proximal term in the local objective function which penalizes the loss as it moves away from the original weights (which are the global weights). Informally, it avoids that local models move "too far" from the global model.
Instead, SCAFFOLD works in a different way: it tries to correct the local update by following the global weights.

Aggregation and ServerOpt: instead here we will see just the **FedDF** algorithm.
It works by aggregating one single client at a time with the global model.

Communication efficiency: this can be implemented with *quantization* or *pruning*.

Cross-Silo and Vertical Federated Learning

In the cross-silo FL, the system heterogeneity is not a problem since the clients are few, the problem here is that all the clients have the same samples but different features (problem called "Vertical FL").

Take home messages for Distributed and Federated learning

DISTRIBUTED LEARNING	FEDERATED LEARNING
• Objective: scale up and train your models faster. • **When to use it**: whenever you have huge amounts of data, a huge model, or you have a time budget to respect	• Objective: make your ML process privacy-compliant, and offload the cloud. • When to use it: whenever you or the collaborating organizations have

• Two approaches:
  • **Data Parallelism**: replicate your model N times and train them in parallel
  • **Model Parallelism**: deploy a single instance across multiple devices
	• Two main categories:
	  • Cross-device FL: the process happens through the edge, communication efficiency is relevant due to low resources and massive distribution
	  • Cross-Silo FL: the process happens in the cloud and the problem may be different (Vertical FL)

• Keep in mind that communication may be a huge bottleneck	• Personalization may be useful in certain cases

Introduction to Active and Curriculum learning

Up to now, we assumed the data was given to us, without any control, but this is not true in many applications. Indeed, usually, we collect data over time (and we don't throw away the old one) and we annotate data over time. Then, we can create a stream (curriculum) from a large dataset to improve training.
Each of this steps can be improved if we can identify the best data to collect/annotate or identify the best data the model should use for training.

For these reasons we now introduce Active Learning and Curriculum Learning.
Active Learning: given a large unlabeled dataset (and sometimes a small labeled dataset) choose the next sample to annotate.
Curriculum Learning: given a labeled dataset, find the optimal sampling order to learn from it (we want a stream of increasing-difficulty of the samples).

Let's now define some concepts useful for next sections.
forgetting events: a forgetting event happen when a sample x was classified correctly at iteration t and incorrectly at iteration $t + k$.

hard sample: a sample forgotten with a high frequency.
easy sample: a sample forgotten with a low frequency, they are property of the data not the specific model; a significant fraction of easy sample can be omitted from the training data without hurting the final accuracy.

Active learning

We are in this scenario: we have a large set of unlabeled data and a small dataset of labeled data with which we perform the training of a model.
Them, we want to improve the mode: **which sample should we annotate**?

After the new data annotation we retrain with all the data and the annotation can be made by a human or by a machine, we assume that it is perfect.

We will see three different methods to perform active learning:

- **Version space reduction**: represent the entire subspace of hypothesis consistent with the actual date and refine it with new data.
- **Uncertainty and heuristics**: design measures to pick interesting samples.
- **Coresets**: model the data distribution and maximize coverage and diversity.

Active learning: Version space reduction

The version space is the set of hypotheses consistent with the training data.
A method to perform active learning is to choose the samples that reduce the size of the version space.

This can be very easy to implement but of course it works only with simple models and no noise is allowed.

Active learning: Uncertainty and heuristics

This other method relies on this pseudo algorithm:

- Starting from an initial classifier
- While we can label data
    - apply the current classifier to each unlabeled data
    - find the b examples for which the classifier is least certain
    - use an oracle to label a subset of the b samples
    - train a new classifier

In other words we will pick the samples with the larger entropy among the classes output (e.g. if we have three classes, an low entropy sample will output something like $[0.1, 0.1, 0.8]$ -which is not uniform- while an high entropy sample will output something like $[0.3, 0.4, 0.3]$, very uniform).

The most uncertain samples are hard sample (good for us, we want to label them), noise sample (useless for us) and outlier (sometimes may be good, sometimes may be shit, probability not the best choice). So, the best samples are hard enough but not too hard that may be noise/outliers.

Entropy may be problematic with a large number of classes due to how much the low probability classes weigh on the total entropy.

For this reason we introduce an alternative measure of uncertainty: **Best-versus-Second-Best** that is the difference between highest probability and second best.

Active learning: Coresets

We need to notice that uncertainty estimates are not robust against outliers and noise and they may be inaccurate at the beginning of training.
For these reasons we introduce the **Coresets** method.

K-center Greedy

This method relies on the famous combinatorial optimization NP-Hard problem where given some cities, we want to build some warehouses minimizing the maximum distance between a city and a warehouse. This is a purely unsupervised problem based only on distances.

From our point of view, we want to find the most distant example given the current pool of samples. Once found we will add it to the pool.

VAAL - Variation Adversarial Active Learning

This is a pool-based semi-supervised active learning algorithm that implicitly learns this sampling mechanism in an adversarial manner. It learns a latent space using a variational autoencoder (VAE) and an adversarial network trained to discriminate between unlabeled and labeled data.

Actually it is a β-VAE, which is a VAE which regulates the prior as a function of a scalar β.

The VAE Loss is composed by the loss of unlabeled data plus the loss of loss of labeled data. The notation used is the following: (X_L, Y_L) is the labeled set; X_U is the unlabeled pool; q_ϕ is the encoder; p_θ is the dencoder; $p(z)$ is the prior.

The total loss is defined as

$$L_{VAE} = \lambda_1 L_{TRD} + \lambda_2 L_{ADV}$$

where:

$$L_{TRD} = \mathbb{E}[\log p_\theta(x_L|z_L)] - \beta D_{KL}(q_\phi(z_L|x_L) \;||\; p(z)) \qquad \textit{(labeled loss)}$$
$$+ \mathbb{E}[\log p_\theta(x_U|z_U)] - \beta D_{KL}(q_\phi(z_U|x_U) \;||\; p(z)) \qquad \textit{(unlabeled loss)}$$
$$L_{ADV} = -\mathbb{E}[log(D(q_\phi(z_L|x_L))] - \mathbb{E}[log(D(q_\phi(z_U|x_U)))]$$

Being an adversarial scenario, there is also a discriminator D which learn to discriminate between labeled data (the left part of the next formula) and the unlabeled part (right part):

$$L_D = -\mathbb{E}[log(D(q_\phi(z_L|x_L))] - \mathbb{E}[log(1 - D(q_\phi(z_U|x_U)))]$$

So the whole training loop is defined by following algorithm:

Algorithm 1 Variational Adversarial Active Learning

Input: Labeled pool (X_L, Y_L), Unlabeled pool (X_U), Initialized models for θ_T, θ_{VAE}, and θ_D
Input: Hyperparameters: epochs, λ_1, λ_2, α_1, α_2, α_3

for $e = 1$ to epochs **do**
  sample $(x_L, y_L) \sim (X_L, Y_L)$
  sample $x_U \sim X_U$
  Compute $\mathcal{L}_{\text{VAE}}^{trd}$ by using Eq. 1
  Compute $\mathcal{L}_{\text{VAE}}^{adv}$ by using Eq. 2
  $\mathcal{L}_{\text{VAE}} \leftarrow \lambda_1 \mathcal{L}_{\text{VAE}}^{trd} + \lambda_2 \mathcal{L}_{\text{VAE}}^{adv}$
  Update VAE by descending stochastic gradients:
  $\theta'_{VAE} \leftarrow \theta_{VAE} - \alpha_1 \nabla \mathcal{L}_{\text{VAE}}$
  Compute $\mathcal{L}_{\text{D}}$ by using Eq. 3
  Update D by descending its stochastic gradient:
  $\theta'_D \leftarrow \theta_D - \alpha_2 \nabla \mathcal{L}_{\text{D}}$
  Train and update T:
  $\theta'_T \leftarrow \theta_T - \alpha_3 \nabla \mathcal{L}_{\text{T}}$
end for
return Trained $\theta_T, \theta_{VAE}, \theta_D$

After the training we can use the probabilities givens by the discriminator as a score to collect the b number of samples as unlabeled samples with the lowest confidence to be sent to the oracle.

Curriculum Learning

Humans learn in curricula: we start from easier tasks and we will go the harder tasks. The question is: *can a DNN benefit from this kind of learning?* and the answer is "*yes, sometimes.*". Sometimes the anti-curriculum (hard-to-easy) is better and as always the optimal solution depends on the domain. The idea behind the Curriculum Learning is to **order data in experiences by difficulty in order to improve the final accuracy and the convergence speed.**

Formerly, the definition of Curriculum learning is the following:

> A curriculum is a sequence of training criteria over T training steps.
>
> Each criterion is a reweighting of the target training distribution.

Behind, we have three conditions to assurance:

1. The entropy of distributions gradually increases
    - this means that diversity and information should gradually increase and so we introduce harder concepts gradually.
2. The weight for any example increases
    - this means that the training data grows over time, while old samples remain always available
3. The final weighting corresponds to the real distribution

A curriculum learning algorithm is made up by two components:

1. **Difficulty Measures**: it decides the difficulty of each sample.
2. **Training Scheduler:** it decides the sequence of data subsets.

In general we can define two families of Curriculum Learning methods:

1. **Predefined Curriculum Learning**: In this approach, both the Difficulty Measurer and Training Scheduler are designed by human prior knowledge without using data-driven algorithms.
2. **Automatic Curriculum Learning**: In this approach, either the Difficulty Measurer or the Training Scheduler (or both) is learned using data-driven models or algorithms.

For the Predefined Curriculum the algorithm which requires a human is called "Baby-Step".

For the Automatic Curriculum, instead, we will see the **Self-paced-learning (SPL)**, even if there are several other implementations using Reinforcement Learning, for example. The Self-paced-learning algorithm relies on the idea that at each step we train the model only on the $p\%$ easiest examples. It measures the difficulty using the loss for each sample and it increases the percentage of data over time. The difficulty is measured directly by the student.

Introduction to OOD Detection

So far, we've assumed that the samples we have in the test distribution always belong to the training data. In practice this is not true.
We would like the model to be able to estimate the uncertainty (of the data and its predictions) correctly.

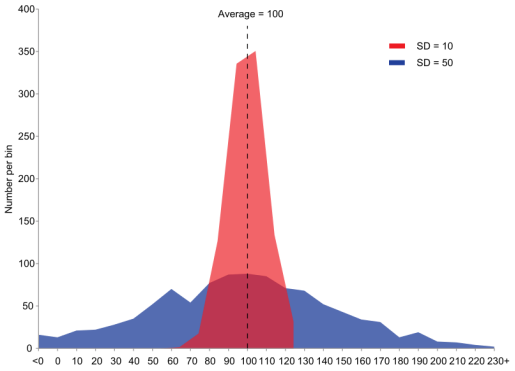
The samples that don't belong to the training data are called out-of-distribution (OOD) samples, and the recognition of this sample is a fundamental issue for many application such as the safety-critical one.

Today it is still an open question, we will see some techniques but they are not exhaustive.

A nice comparison can be made between OOD detection and Anomaly detection.
The latter one distinguishes between the normal and abnormal behavior and it is usually modeled as an highly imbalanced binary classification problem.
The first one, instead, relies on detecting a drift and rejecting uncertain outputs.

OOD Detection using Statistic (ODIN) + Ensembles

In statistics, dispersion is a property of a distribution measuring how *«stretched»* it is. For example, with reference to the figure, the blue distribution is much more dispersed than the red even though they have the same mean.



Another fundamental aspect is that there are two types of uncertainties to be considered that must be resolved differently:

- **Aleatoric uncertainty**: This comes from the data, it's irreducible noise but nevertheless we want to model it in our model.
- **Epistemic uncertainty**: this comes from the model (how certain the model is in its predictions), we can reduce it by adding more data.

A first try: A first observation we can make is that "Correctly classified examples tend to have greater maximum softmax probabilities than erroneously classified and out-of-distribution examples".
This means that we could use a threshold to separate OOD data from in-distribution data (ID).

But this simplistic solution only work in very simple setting because there will be some OOD example where the model is still very confident.

ODIN: So, we can make a second observation: "temperature scaling and small perturbations to the input can separate the softmax score distributions between in- and out- of-distribution images".

Indeed, we expect OOD samples to be more sensitive to small perturbations (because in-distribution samples are more robust to the noise) and temperature scales changing the softmax output distribution towards more uniform (high temperature) or more peaked (low temperature) distributions.

In particular:
The noise injected into the inputs is not random noise but "**adversarial noise**" that is, noise that goes in the direction that decreases the probability:

$$\tilde{x} = x - \epsilon\, sign(-\nabla_x log\, S_{\hat{y}}(x, T))$$

where $S_{\hat{y}}(x, T)$ is the softmax result on the input x scaled by the temperature T.
In other words, we calculate the gradient with respect to the input of the log-likelihood, we calculate the sign of the gradient and at the end we do a small step in that direction. We expect this perturbation to decrease more OOD examples.

Then we take the maximum score from each prediction l (obtained by the softmax) and at the end we use a threshold δ on that score to discriminate ID/OOD samples.

The values T and δ are obtained with a model selection process; the main advantage of this method is that it can be used to any pretrained supervised model.

Limits of Uncertainty Estimation: the main limitation is that DNN are overconfident, so we cannot trust their confidence. For example, a model trained on CIFAR10 can assign higher probability to a sample of another dataset with respect to CIFAR10!

Moreover, empirical observations shows that ODIN often fails!

Ensembles: a simpler solution for our problem could be the ensemble methods which use a proper scoring rule as training criterion and use the adversarial training to smooth the predictive distribution to be more robust to noise for the in-distribution example.

Empirical observations show that it can work but we have to pay the increasing computational cost (both for training and inference).

Introduction to the open world

So far, all the methods we have seen assume a closed world, even when we had a train or test drift. For example, in meta-learning we know that we have a new task and we get a small training set for the new task.

What happens when the model encounters unseen classes during testing?
What happens if the model doesn't even know how the unseen classes look like?

The open world problem

We start from two assumptions: the closed and open world assumptions.

- **Closed world assumption**: the model «knows what it needs to know» such as which classes are present in the data distribution.
- **Open world assumption**: the model may encounter new data at test time.
    - We don't expect the model to generalize to unseen classes
    - Therefore, we would like to model to be able to recognize what it doesn't know

These assumptions are the same as the Formal Logic ones.

Then we have to define three classes of samples:

1. **Known**: in-distribution samples and predicted correctly with high confidence (correct predictions)
2. **Known Unknowns**: low-confidence samples, such as successfully recognized anomalies (low confidence)
3. **Unknown Unknowns**: out-of-distribution samples with highly confident predictions (the model shouldn't have high confidence here)
    a. In general, if the test distribution drifts, it may become something completely new. Effectively, the model doesn't know what it doesn't know

In order to let the model to be able to recognize what it doesn't know we can use out-of-distribution technique already seen or apply other two strategies:

1. Prior knowledge
2. Open Set Recognition

Prior knowledge (the background technique)

A prior knowledge technique relies on training the model to recognize a background class. This is useful when we have access to a large set of unknown examples.

In this case we can model all the unknown example as a background class, which represents what the model doesn't know because we are just interested in a small subset of classes (already labeled).
In this way we convert an open world problem into a close world one.

Objectosphere: one way to implement this technique is called *Objectosphere* which consist of the train with a background class.
The idea behind is that magnitudes of features for unknown samples are often lower than those of known samples, so the *Objectosphere Loss* explicitly optimizes this objective: known samples should have a magnitude above a specified minimum, while background samples should have magnitudes close to zero.

Limitations of prior knowledge methods: using the background class we can train the model to recognize the unknown. But this work only when we can define the background class!

Moreover, adding the background class is not so easy because we cannot have a lot of data as background and few data for the labeled class.

Open Set Recognition

All the classification models that we used have unbounded decision boundaries. To understand this, imagine a binary linear model. It splits the space into two regions: one is positive, the other is negative.
The distance from the boundary is not taken into account, this is a problem because we could have an OOD sample which is still classified as negative or positive even if very far from the boundary.

So, in an open world setting, we need to allow a reject option.

For this reason we want to reject samples that are too far, this can be done by using a distance-based classifier and a threshold for the maximum distance.
The distance used can be the **Mahalanobis Distance** which compares the standard deviations of the current sample and the mean of standard deviation items of the selected class.

Another way to implement Open Set Recognition is to use the activation vectors.

Large Language Models

Large Language Models provides very general knowledge about large domains (vision, language), we can adapt them to solve new tasks with few (or zero) examples.
In particular we can define:

- **prompting**: just at inference time (no weights are updated!) we give a description of the task and optionally a sequence of examples.
- **few-shot**: the model is given a few samples of the task as conditioning
- **one-shot**: the model is given a single sample of the task as conditioning
- **zero-shot:** only the task description is given.

The challenge here is the evaluation of the zero-shot scenario because we cannot be sure that the task provided is really unseen. Companies are not sharing many details about the models!

Vision Transformers

Brief recall of the Transformers equations:

- Q query, K keys, V values
- $MultiHead(Q, K, V) = Concat(head_1, ..., head_k)\, W^0$
- $head_i = Attention(Q\, W^Q_i,\, K\, W^K_i, V\, W^V_i)$
    - Transformers use multiple heads, each transformed by different linear projections.
- $Attention(Q, K, V) = softmax(QK^T / \sqrt{d_k})\, V$

Vision Transformers, also known as ViTs, are a class of deep learning models that apply the transformer architecture to visual tasks, such as image classification and object detection.

The image is splitted in 16×16 patches, which are linearly embedded and plugged into a positional embedding needed because attention is invariant to the patch position.

After this preprocessing process we now have a sequence of vectors which can be used as input for the standard Transformer encoder, like we do for NLP tasks.

Usually ViTs work well only on huge datasets.

Vision-Language Models

A Vision-Language Model is a model that can understand and generate meaningful connections between visual information (images) and textual information (language).
The language can encode a high-level description of an image (e.g. a caption of an image is an "abstract" way to describe it).

This means that having lots of captioned images we can provide much more information than using the image alone, but we need to combine vision and language models.

One implementation of a Vision-Language Model is CLIP. It works, mainly, in two steps: the first one is the **pre-training task**, where it learns to map text captions with visual images. The goal is to understand which images and text belong together.
After the pretraining, CLIP performs the **zero-shot transfer**, where natural language is used to reference learned visual concepts (or describe new ones)

In particular, the pretraining task is performed using **contrastive training**. In this case we cannot use supervised training to learn from <caption, image> pairs, because there is too much diversity in the possible images and captions and this means that it is very difficult to predict the exact image/captions of the current pair.

Instead, the contrastive training is effective because it does not require exact matching. Given a batch of <caption, image> pairs it predicts which captions map which images, learning a multi-modal embedding space. In other words, during pretraining, CLIP learns to associate images and text by maximizing the similarity between positive pairs (images and their associated text) and minimizing the similarity between negative pairs (randomly paired images and text).

Then the **zero-shot transfer** using natural language is performed. An example of this transfer is the *zero-shot image classification*.
Given a set of classes {"dog", "cat", ...} we create prompts like "*this is an image of [class]*". Then for each image we find the closest prompt.
We can even use multiple prompts for the same class, this will create a kind of Ensembling.

Papers results say that the Zero-Shot CLIP is more robust to dataset shift with respect to the usual techniques like ImageNet with a ResNet.

Efficient Finetuning

Here we have some random ideas we can implement to perform an efficient Finetuning.

1. We can use large models to create synthetic dataset to train the smaller models.
2. For classification problems, the prompts are fixed embedding vectors. This means that we can learn them using *Learning2Prompt* algorithm.
3. Low Rank Adaptation (LoRA): We can finetune only a small part of the model by using a contractive autoencoder which takes into account only a subset of the input, the remaining part of the input are used with pretrained weights
4. Texqual Inversion: given few (3-5) images representing a concept (e.g. a banana), we can encode the concept in a text token (which can be even random words for us), but one we have that text token we can use it with some nice prompts in order to obtain custom results.

Scaling laws

Training large models is very expensive, we would like to predict the results in advance. We cannot even perform model selection, since these models are huge.

The only thing we can do is to recover the previous checkpoint when the model starts to be unstable.

There is an empirical law to determine the loss given the number of parameters of the model N and the number of tokens D in the datasets, this loss is called **Chinchilla Scaling Law**.

$$L(N, D) = \frac{406.4}{N^{0.34}} + \frac{410.7}{D^{0.28}} + 1.69$$

The last term is the irreducible loss.