Continual Learning

Lorenzo Leuzzi

2024

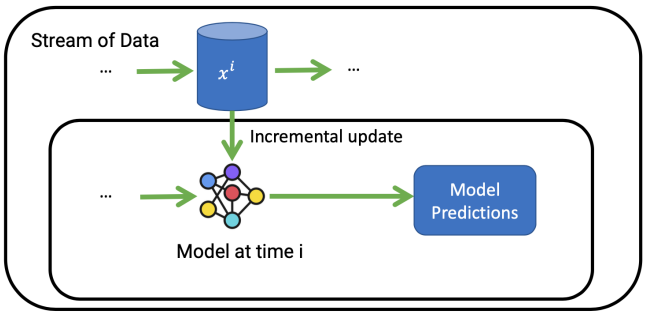
Contents

Online Machine Learning

1.1 Introduction

1.1.1 What is Online Machine Learning?

Online machine learning (OML) is a **dynamic approach** where models are continuously **updated as new data becomes available over time**. Unlike traditional batch learning methods where models are trained on fixed datasets, online machine learning algorithms process data in real-time or in mini-batches, making predictions or adjustments as each new piece of information arrives. This iterative process allows models to adapt to changing conditions and evolving patterns, making it particularly useful for scenarios with rapidly changing data streams or where immediate responses are required.
In OML data is generated continuously, received as a **stream** over time, and often in high frequency or volume, making it impractical to store.



Batch vs Online Learning

Batch learning involves training a machine learning model on a **fixed dataset** where all the data is available at once. It typically follows an i.i.d. (independent and identically distributed) sampling approach, meaning data points are randomly selected from the dataset for training. Batch learning methods do not face stringent computational constraints and can take their time to train since they have access to all the data upfront. They often have separate training and evaluation phases.
On the other hand, **online learning** is a **sequential** process where the model is **updated continuously** as **new data arrives**. Instead of having access to the entire dataset at once, online learning algorithms process data point by point or in small batches. Computational constraints are more critical in online learning, as models must adapt quickly to incoming data while managing resources efficiently. In online learning, training and evaluation are often interleaved, with the model being updated incrementally as it receives new information.

1.1.2 Evaluation

Prequential Evaluation

Prequential evaluation, short for predictive-sequential evaluation, is a method used in machine learning to assess the performance of predictive models over time as new data becomes available. Unlike

traditional evaluation techniques (e.g. holdout) that separate training and testing phases, prequential evaluation is an online evaluation method that continuously evaluates the model's performance on incoming data streams.
Prequential evaluation adopts an **interleaved-test-then-train** approach where each sample in a single stream is first used for testing before being used for training. The evaluation is based on the **aggregate prequential loss** over time and this can present challanges. Initially, the model may be underfitted, and as it improves, past errors are remembered, leading to an overestimation of the current model's holdout error. This issue becomes more pronounced in nonstationary distributions, as the desire is to evaluate the model on future data while the loss is computed on older data and models. A solution lies in controlled forgetting of past performance. Techniques such as Sliding Window, which computes accuracy based only on the last k elements, and Fading Factor, which maintains a running average of the loss, offer ways to mitigate these challenges.

Real-time machine learning encounters situations where targets are not immediately available but they are **delayed**. In forecasting scenarios, targets are inherently received in the future, while in supervised classification tasks, labels may be delayed, often due to manual offline labeling processes. To address this delayed evaluation, a strategy involves storing samples without targets in a buffer and training the model as soon as the targets become accessible. This approach aligns with prequential evaluation, which remains effective even with sparse targets, allowing for adaptive learning in real-time environments.

Ensemble Validation

There's different ways to **split** the stream if we want to train (and evaluate) an **ensemble**.

- **K-fold distributed cross-validation** involves randomly selecting one classifier for testing while utilizing the rest for training, adapting the offline cross-validation method. This approach optimally utilizes the data, ensuring that only one $1/k$ sample remains unused in each iteration, although it may introduce high redundancy.
- **K-fold distributed split-validation** randomly assigns each sample for training in one classifier and testing in all others, resulting in models trained on disjoint data but potentially underutilizing the dataset with each model only using $1/k$ data for training.
- **K-fold distributed bootstrap-validation** involves each sample being trained in roughly $2/3$ of the classifiers with varying weights and tested in all classifiers, a technique further explored in the ensembling lecture.

Metrics

Kappa statistic is a metric that assesses the agreement between a model's accuracy p and a baseline accuracy (p_{random}). The **baseline** can be a random classifier that selects classes with the same proportion as predicted by the model. It provides a measure of relative improvement compared to the baseline accuracy, with $k = 1$ indicating a perfect classifier and $k = 0$ indicating a random classifier. Kappa statistic is particularly advantageous for imbalanced scenarios, as it is quick to compute online compared to other measures like the AUROC.
When the proportion of classes predicted by the model differs from that of the stream, kappa with random baseline becomes an unreliable estimate. This discrepancy suggests potential issues such as the classifier being underfitted or a change in the input-output distribution. In such cases, a **persistent classifier**, which predicts the next label to be the same as the last seen label, serves as a better **baseline**. This approach effectively captures simple correlations within the stream, providing a more accurate benchmark for evaluating model performance.

$$k = \frac{p - p_{baseline}}{1 - p_{baseline}} \tag{1.1}$$

1.1.3 Benefits and Challenges

Online learning offers several benefits, including the use of efficient algorithms that can continuously update the model, allowing it to adapt to changing data streams. Additionally, online learning enables

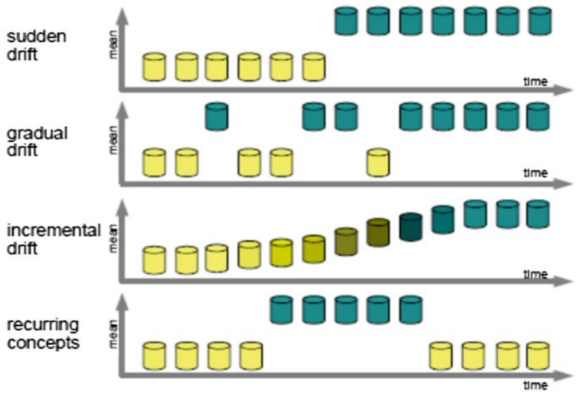
the model to **forget past information when it becomes irrelevant**, thus ensuring its relevance to current conditions. However, online learning also presents **challenges**, such as concept drift, which refers to the non-stationarity of data distributions over time, requiring the model to continuously adjust. Imbalance in the distribution of classes within the data stream can also pose a challenge, affecting the model's ability to generalize effectively. Furthermore, hyperparameter tuning remains a significant challenge in online learning, as optimizing parameters for dynamic data streams requires careful consideration of computational resources and model performance.

Transitioning from offline to online machine learning requires a paradigm shift with several key considerations. Efficiency becomes a primary focus in online settings, emphasizing the need for algorithms that can handle continuous data streams effectively. The assumption of independent and identically distributed (i.i.d.) data is often broken in online scenarios, necessitating models that can adapt to evolving distributions. Additionally, prequential evaluation, which involves interleaved testing and training on sequential data points, emerges as a crucial technique for assessing model performance in real-time environments.

1.2 Concept Drift

1.2.1 What is Concept Drift?

In online machine learning, **Concept Drift** (CD) refers to the phenomenon where there is a **change over time** in the real-world environment that subsequently impacts the distribution of input and/or output data. This change disrupts the model's ability to make accurate predictions over time. Concept drift is distinct from noise or outliers; it represents a fundamental shift in the underlying data-generating process, rather than random fluctuations. A Concept Drift is a change in the distribution that requires changing the model, while an **anomaly** is an example different the underlying distribution (outlier). CD poses a significant challenge for online learning algorithms, as they must continuously update their models to accommodate these evolving patterns and ensure accurate predictions over time.



Nomenclature

Concept drift encompasses various types of changes in data distribution over time.

- **Sudden changes**, often referred to as shifts, occur abruptly after a period of stability, resulting in a significantly different distribution.
- **Gradual changes**, on the other hand, involve small, incremental alterations that accumulate over time, eventually leading to noticeable shifts. These changes can be either global, affecting the entire item space, or partial, impacting only certain instances or attributes.
- **Recurrent concepts** involve distributions that reappear periodically, such as seasonal patterns or irregular events like mass gatherings affecting city traffic.

Causes of Shifts

Shifts in data can occur due to various reasons. One such cause is sampling bias, where the subset of the world we observe changes over time, leading to what is termed **virtual drift**. This type of bias can be seen in polling data or situations with limited observability. Another cause is non-stationary environments, where the world itself is changing continuously; this **real drift** is exemplified by changes in weather patterns or financial markets.

1.2.2 Probabilistic Definition

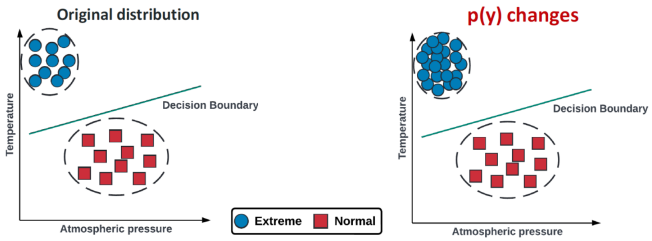
Given an input $x_1, x_2, ..., x_t$ of class y we can apply bayes theorem:

$$p(y|x_t) = \frac{p(y)p(x_t|y)}{p(x_t)} \tag{1.2}$$

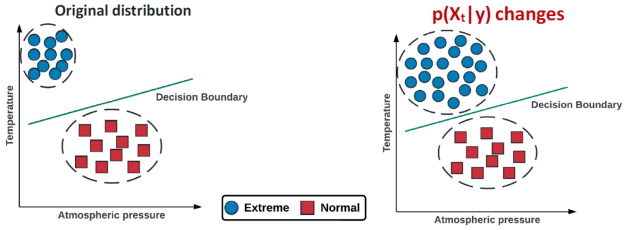
where:

- $p(y|x_t)$: the posterior probability of class y given the feature x_t. This is what we want to compute: the probability that the data point belongs to class y after observing x_t.
- $p(y)$: the prior probability of class y. This is our initial assumption about the probability of class y before observing x_t.
- $p(x_t|y)$: the likelihood, which is the probability of observing x_t given that the data point belongs to class y.
- $p(x_t)$: the marginal probability of observing x_t. This is the probability of observing x_t under all classes.

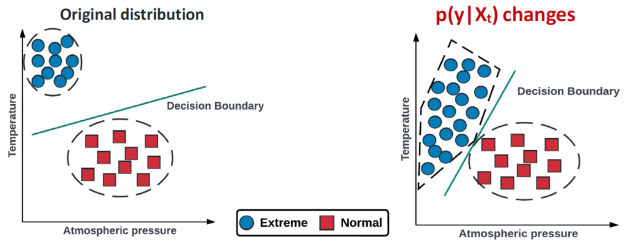
Let's consider an example of predicting extreme weather phenomena based on atmospheric pressure and temperature. Typically, extreme weather events occur under low atmospheric pressure and high temperature.
$p(y)$ **Concept Drift**: In the 20th century, the distribution of atmospheric pressure and temperature remained unchanged, but the frequency of extreme weather phenomena increased.



$p(X|y)$ **Concept Drift**: In the first two decades of the 21st century, the conditions of atmospheric pressure and temperature under which these phenomena occur began to change. However, these changes were not drastic enough to alter the decision boundary used for prediction.



$p(y|X)$ **Concept Drift**: Due to ongoing climate change, extreme weather phenomena now occur more frequently with higher atmospheric pressure and lower temperature. Consequently, we must update the decision boundary to maintain high predictive performance.



Dataset Shift Nomenclature

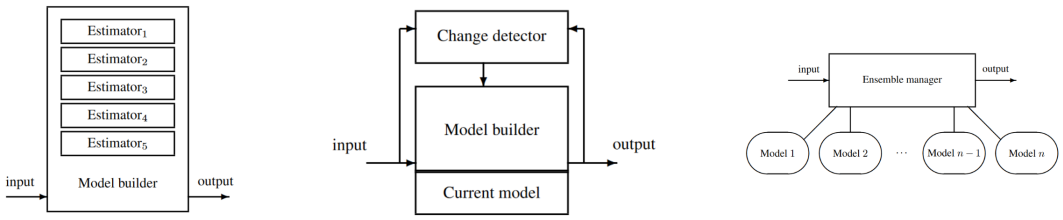
The nomenclature is based on causal assumptions. $x \rightarrow y$ problems: class label is causally determined by input (example: credit card fraud detection). $y \rightarrow x$ problems: class label determines input (example: medical diagnosis).

- **Dataset Shift**: $p_{tra}(x, y) \neq p_{tst}(x, y)$. Informally, any change in the joint distribution of inputs (x) and outputs (y) between the training and testing phases is considered a shift.
- **Covariate/Input Shift**: This happens in problems where the focus is on predicting y given x. The conditions are $p_{tra}(y|x) = p_{tst}(y|x)$ and $p_{tra}(x) \neq p_{tst}(x)$. Informally, the distribution of the input data x changes, but the conditional distribution of the output given the input (input-output relationship) remains the same.
- **Prior Probability Shift**: This happens in problems where the focus is on predicting x given y. The conditions are $p_{tra}(x|y) = p_{tst}(x|y)$ and $p_{tra}(y) \neq p_{tst}(y)$. Informally, the distribution of the output data (y) changes, but the conditional distribution of the input given the output (output-input relationship) remains the same.
- **Concept Shift**: This can occur in both $x \rightarrow y$ and $y \rightarrow x$ problems. For $x \rightarrow y$ problems, the conditions are $p_{tra}(y|x) \neq p_{tst}(y|x)$ and $p_{tra}(x) = p_{tst}(x)$. For $y \rightarrow x$ problems, the conditions are $p_{tra}(x|y) \neq p_{tst}(x|y)$ and $p_{tra}(y) = p_{tst}(y)$. Informally, the underlying relationship between the inputs and outputs changes. This means the concept or the definition of the class itself has changed.

1.2.3 Concept Drift Detection and Estimation

The requirements for effectively managing drifts include fast detection of change, robustness to noise and outliers, and low computational overhead. To address these needs, various families of strategies have been developed.

- **Estimator-based** approaches track statistics of the data stream, allowing algorithms to update the model's statistics incrementally.
- **Detector-based** methods actively monitor for concept drift and, upon detection, trigger an update to the model.
- **Ensembling strategies** maintain a dynamic population of models to ensure a more flexible and adaptive response to change.



Despite the sophistication of these strategies, a common limitation is their focus on single-dimensional or low-dimensional data, which constrains their applicability in complex, high-dimensional domains where concept drift may present itself in more subtle and multifaceted ways.

Estimation

Models need to continuously adapt to the changing data distribution. One way to achieve this is by updating the model's parameters or statistics to reflect the current data trend.
The expected value, often represented as θ_t, is a statistical measure that models need to estimate and update regularly. It is the expected value of a function $f(x)$ with respect to the probability distribution $p_t(x)$ at time t, which may be subject to drift. The expectation is defined as: $\theta_t = \mathbb{E}_{p_t(x)}[f(x)]$.
There's different ways to find outdated element and discard them from the computation:

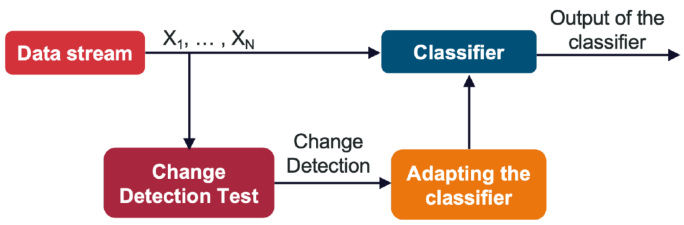
- **Window-based**: Linear Estimator Over a Sliding Window. Use only a recent window k to estimate the statistic and ignore older elements $\theta_t = \mathbb{E}_{p_t(x)}[f(x)] = 1/k \sum_{i=1}^{k} f(x_{t-i})$.
- **Exponentially Weighted Moving Average** (EWMA): memoryless estimator-based concept drift detection with exponential averaging. $A_t = \alpha x_t + (1-\alpha)A_{t-1}$, $A_1 = x_1$ and α is an exponential decay factor that controls the forgetting.

Detection

Models provide an alarm when a change is detected. There's different methods, using statistical tests monitoring the input distribution or monitoring the model's accuracy. In change detection, a **tradeoff** balance must be considered between prompt **detection** and avoiding **false positives**. If changes are detected too early, it can lead to unnecessary retraining of the model, wasting resources. Conversely, delayed detection results in the model making numerous errors. The sensitivity of detection depend on magnitude θ of the changes we want to detect. Several key metrics help in managing this tradeoff:

- **Mean Time Between False Alarms** (MTFA) how often we get false alarms when there is no change, with the False Alarm Rate (FAR) being its inverse.
- **Mean Time to Detection** (MTD) assesses the system's ability to detect and react to change when it occurs.
- **Missed Detection Rate** (MDR) captures the probability of not generating an alarm when there has been change.
- The **Average Run Length** (ARL) generalizes MTFA and MTD, indicates how long we have to wait before detecting a change after it occurs.

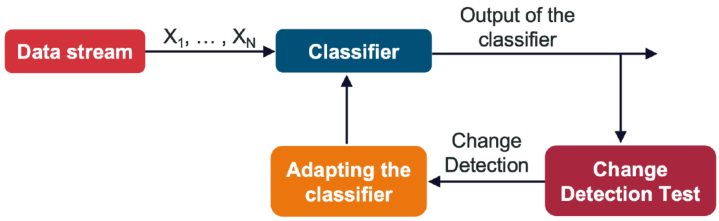
A class of CD detectors works by **monitoring** only the **input distribution**. The advantage of this approach is that it does not rely on labeled data and can utilize straightforward statistical tests for identifying changes. However, this method faces challenges in handling multivariate data streams or when the underlying data distribution is not well-defined. Additionally, this type of detector may not recognize changes that do not alter the observation distribution.



CUSUM Test (CUmulative SUm control chart) is a method to monitor the input: give an alarm when the **mean of the input data** significantly deviates from its previous value. It is an iterative process that starts at $g_0 = 0$ and at each iteration it calculates $g_t = max(0, g_{t-1} + z_t - k)$, where $z_t = (x_t - \mu)/\sigma$ is the standardized input and k and hyperparameter. When $g_t > h$ it declares a change and reset $g_t = 0$ and μ, σ.

Page-Hinkley is similar to CUSUM: give an alarm when the mean of the input data significantly deviates from its previous value. g_0 and g_t are the same but the condition to declare the change and to reset is if $g_t - G_t > h$, where $G_t = min\{g_t, G_{t-1}\}$.

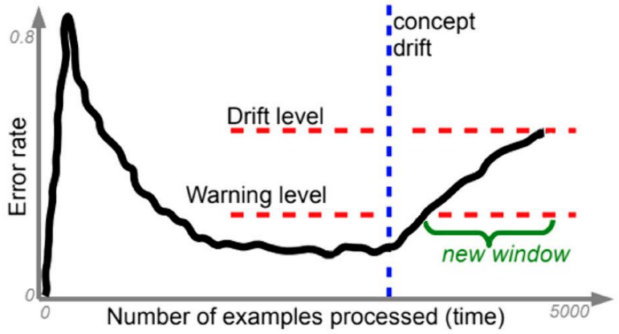
Monitoring the input distribution is simple but somewhat limited, instead we can **monitor the classification error** (output). Real-time error monitoring offers clear advantages, as low accuracy signals the need for retraining, ensuring straightforward feedback. Despite complex input, classification error remains a simple, one-dimensional time series, simplifying analysis. However, this approach relies solely on supervised signals to compute errors, limiting its applicability to scenarios where such signals are available.



The **Drift Detection Method** (DDM) is a drift detection based on model's accuracy. The idea behind DDM is that without drifts, error should decrease over time as more data is used. If the errors increase, then we have a drift. By modeling the distribution of errors over time, drift can be detected by identifying unexpected values within the error distribution, providing a means to monitor and adapt to changing data patterns. Given p_{min} the minimum error rate measured up to time t and s_{min} minimum standard deviation measured at time t ($s_t = \sqrt{p_t(1-p_t)/t}$):

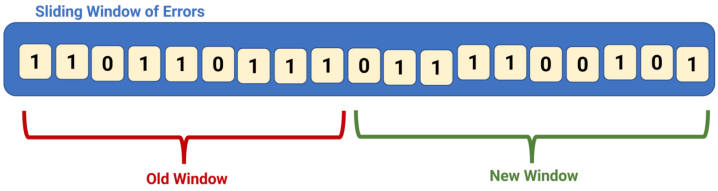
- if $p_t + s_t \geq p_{min} + 2s_{min} \rightarrow$ warning: start storing examples in the buffer to prepare for retraining.
- if $p_t + s_t \geq p_{min} + 3s_{min} \rightarrow$ drift: discard the previous model, train a new model using the buffer collected from the warning time and reset p_{min} and s_{min}.

This is a simple and general method. May be slow to changes since p_t is computed over all the examples since the last drift. Memory occupation depends on the distance between warning and drift. We can use EWMA to estimate errors.



Warning level: change is detected but is not high enough to be considered a drift. Drift level: the change is big enough to be considered a drift.

Early DDM is an approach that focuses on the distance between two errors classification instead of considering only the number of errors. As the learning process improves predictions, this distance typically increases. However, when drift occurs, the distance between errors decreases. By calculating the average distance between errors along with its standard deviation and identifying outliers in the tails, Early DDM effectively detects shifts in data distribution.



1: correct, 0: error, Sum: number of errors, Drift: large difference in number of errors.

Another approach is the **ADaptive sliding WINdow** (ADWIN) that involves comparing multiple sliding windows of different lengths. ADWIN utilizes Exponential Histograms to facilitate the computation of sums across sliding windows with different sizes, enhancing computational efficiency. It employs a statistical test to identify differences in error distribution, enabling the detection of drift. This test is applied to all potential sliding windows, allowing for comprehensive analysis and timely adaptation to changing data patterns.

1.3 Online Classification Models

1.3.1 Introduction

An online classification method can be of two kind:

- **Streaming Classification Models** designed to handle infinite data streams as input. Unlike traditional methods, these models cannot save the input data due to its continuous nature, and they must operate under constraints such as latency and computational resources.
- **Out-of-core methods** are employed for models too expensive to be trained on the entire dataset in one step, primarily addressing computational constraints. Notably, these methods typically do not encounter concept drifts, as the data remains static and can be shuffled.

1.3.2 Training Online

Online classification methods face constraints such as the inability to store the entire data stream in **memory** and the limitation of shuffling data. Despite the capacity to maintain a **small buffer**, retraining from scratch at each step is time consuming. Instead, these methods utilize **small mini-batches** to update the model incrementally. Algorithms must efficiently handle the previous model and a small batch of samples as input, producing a new model to adapt to the continuous data stream. In online algorithms, initial phases of training often exhibit susceptibility to larger changes or suffer from poor initializations. To mitigate these challenges, leveraging an initial model **pretrained** on static data is common practice. This approach, known as **Warm Start** or **Finetuning** depending on the field, allows for a smoother transition into online learning by using the pretrained model as a starting point. In the realm of Deep Learning, this method is widely adopted as a standard practice to enhance model stability and performance.

1.3.3 Inherently Online Methods

Naive Bayes

The Bayesian method, particularly Naive Bayes classification, operates picking the class that maximize the joint probability of classes and the input features $argmax_i p(x, y_i)$. Under the **conditional independence assumptions** (*the input features are statistically independent between themselves given the target*) we can write the joint probability as:

$$p(\mathbf{x}, y_k) = \underbrace{p(y_k)}_{prior} \prod_{i=1}^{D} \underbrace{p(x_i|y_k)}_{cond\_prob} \tag{1.3}$$

The training consist in estimating the conditional probabilities and priors. Bayesian models are online methods because the way to calculate this probabilities is not different from the offline version:

- posterior: N_k/N
- discrete conditional probabilities: $M^i_{j,k}/N_k$

where N is the number of examples in the dataset (in streams count how many samples seen up to now), N_k are samples of class k and M it's a table of counters that for each possible value of x_i counts its occurrence in the training set.
Estimating the posterior probability of the parameters is training for Bayes models.

$$p(\theta|\mathbf{X}) = \frac{p(\mathbf{X}|\theta)p(\theta)}{p(\mathbf{X})} \tag{1.4}$$

In online training the posterior becomes the prior for the next step.

$$p(\theta_t|\mathbf{X}_t) = \frac{p(\mathbf{X}_t|\theta_t)p(\theta_{t-1}|\mathbf{X}_{t-1})}{p(\mathbf{X}_t)} \tag{1.5}$$

This method has some limitations: the posterior is often approximated and the error multiplicated so it may have large errors and we are also ignoring robustness to drifts and computational limitations.

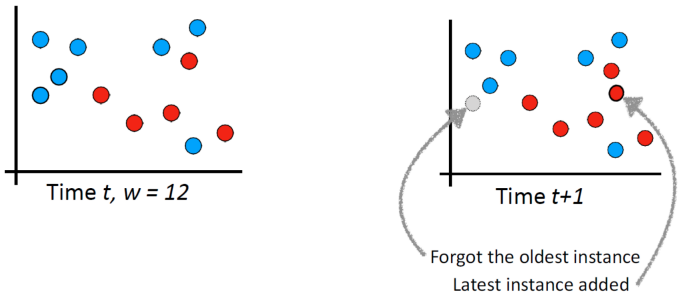
Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a parametric iterative algorithm that given a differentiable function f and loss function $\nabla\mathcal{L}(\theta, x)$ find a minimum θ_* s.t. $\nabla\mathcal{L}(\theta, x) = 0$. At each iteration a descent step is taken untill convergence $\theta_{i+1} = \theta_i + \lambda\nabla\mathcal{L}(\theta_i, x)$.
Incrementally updating in mini-batches aligns well with the characteristics of online learning, where models are continuously updated as new data becomes available. 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.

1.3.4 Adapting Offline Methods

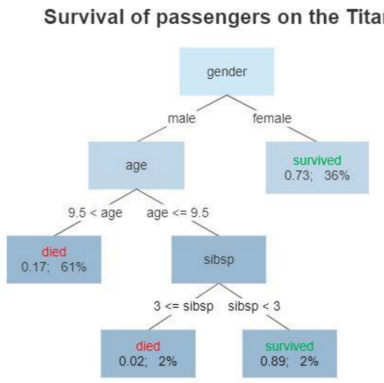
Online K-NN

K-NN is a non-parametric distance-based classifier that stores samples from the dataset and computes distances between old examples and new one to find k closest examples.
The problem is that the algorithm is designed for offline training: we cannot save the entire stream. A solution is to use use a **fixed sliding window**. If a concept drift occurs, with K-NN there is the risk that the instances saved into the window belong to the old concept. Using ADWIN to automatically set the size of the sliding window would solve this problem.



Hoeffding Decision Tree

A decision tree is a tree-like structure used for decision-making, where each internal node represents a feature, each branch represents a decision rule, and each leaf node represents a decision outcome. It recursively splits the dataset based on the features' values to classify or regress data.



Offline training consist in building the tree, so for each node we decide if it needs to be split, which feature to use for the split and how to do the split. Usually this is done using a greedy approach by comparing all the available attributes and choosing the best one (e.g. most discriminative) according to some heuristic measure (Information Gain). **Information Gain** (IG) is calculated as the difference between the entropy of the entire dataset $H(T)$ and the conditional entropy of the dataset given an attribute $H(T|a)$, $IG(T,a) = H(T) - H(T|a)$. It is effectively the reduction in entropy (or impurity) that results from splitting the data on attribute a. The attribute with the highest Information Gain is chosen for the split because it provides the most significant reduction in uncertainty.

In an offline setting, we can compute Information Gain and greedily split the node accordingly. However, in an **online** setting, we can only compute the IG based on past data and cannot predict how much it will change with future data. If the IG changes significantly, we may need to adjust the tree. To avoid constantly revising the split criterion, we must wait until we have enough data.
Our goal is to design a decision tree learner for extremely large (**potentially infinite**) datasets. This learner should require each example to be read at most once, and only a small constant time to process it. In order to find the best attribute to test at a given node, it may be sufficient to consider only a **small subset** of the training examples that pass through that node. Thus, given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate attributes there, and so on recursively (online). We solve the difficult problem of **deciding exactly how many examples are necessary at each node** by using a statistical result known as the **Hoeffding bound**. 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. Given enough samples, we can bound the change in the entropy! Now we have a criterion to decide when we have enough samples to do the split.
Consider a real-valued random variable r whose range is R. Suppose we have made n independent observations of this variable, and computed their mean $\bar{r}$. The **Hoeffding bound** states that, with probability $1-\delta$, the true mean of the variable is at least $\bar{r} - \epsilon$, where ϵ is:

$$\epsilon = \sqrt{\frac{R^2 log(1/\delta)}{2n}} \tag{1.6}$$

Let $G(X_i)$ be the heuristic measure (Information Gain) used to choose test attributes. Our goal is to ensure that, with high probability, the attribute chosen using n examples (where n is as small as possible) is the same that would be chosen using infinite examples. Assume G is to be maximized, and let X_a be the attribute with highest observed $\overline{G}$ after seeing n examples, and X_b be the second-best attribute. Let $\Delta\overline{G} = \overline{G}(X_a) - \overline{G}(X_b) \geq 0$ be the difference between their observed heuristic values. Then, given a desired δ, the Hoeffding bound guarantees that X_a is the correct choice with probability $1-\delta$ if n examples have been seen at this node and $\Delta\overline{G} > \epsilon$. In other words, if the observed $\Delta\overline{G} > \epsilon$ then the Hoeffding bound guarantees that the true $\Delta G \geq \Delta\overline{G} - \epsilon > 0$ with probability $1-\delta$, and therefore that X_a is indeed the best attribute with probability $1-\delta$. This is valid as long as the $\overline{G}$ value for a node can be viewed as an average of G values for the examples at that node, as is the case for the measures typically used. Thus a node needs to accumulate examples from the stream until ϵ becomes

smaller than $\Delta\overline{G}$ (notice that ϵ is a monotonically decreasing function of n). At this point the node can be split using the current best attribute, and succeeding examples will be passed to the new leaves. This leads to the **Hoeffding tree** algorithm. Hoeffding Tree is a very fast decision tree algorithm for streaming data. It strategically splits decisions based on the Hoeffding bound, allowing it to wait for enough instances to arrive before making a decision about a split. As the data grows sufficiently large the algorithm is proven to converge to the tree constructed by a batch learner, ensuring accuracy and efficiency in handling continuous streams of data. The Hoeffding Tree algorithm processes each new sample by finding its corresponding leaf and updating the table that keeps the statistics necessary to compute the split criterion. A split is made if sufficient samples have been collected at the node. This approach allows for incremental decision tree construction. The Hoeffding bound ensures that the greedy splits made do not need to be revisited.

The **Very Fast Decision Tree** (VFDT) is practical implementation of Hoeffding Tree with a few changes. It employs tie-breaking using the Hoeffding bound to decide when to split on attributes that have similar split gain. It computes G only after k updates to increase speed. And it saves memory by deactivating nodes with minimal promise. Additionally, it uses a warm start approach to improve initial model performance and hasten convergence.

The **Concept-Adapting VFDT** (CVFDT) algorithm updates VFDT for data streams that evolve over time by maintaining a decision tree in line with recent observations. The objective is to keep a DT model that is consistent with a sliding windows of w samples. It manages data in a sliding window, adding and removing data points to reflect the latest trends, and continuously checks and updates the decision tree's structure. While doing so, it trades off some of VFDT's theoretical guarantees for adaptability and requires tuning additional hyperparameters.

A **Hoeffding Adaptive Tree** (HAT) is Hoeffding Tree that uses ADWIN for concept drift detection. Unlike CVFDT, HAT creates a new tree as soon as a change is detected and switches to the new tree once it outperforms the old one. While CVFDT requires many hyperparameters related to the expected distance between drifts, HAT adapts to the scale of time change in the data automatically, without relying on prior guesses, thanks to ADWIN.

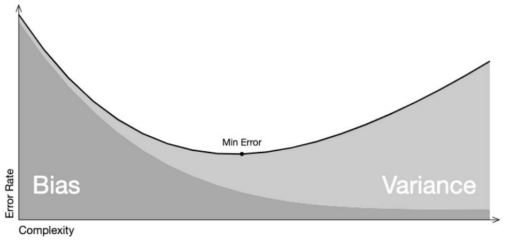
1.4 Ensemble Methods

1.4.1 Introduction

An ensemble model is a machine learning technique that **combines** the predictions of multiple individual models to produce a more accurate and robust prediction. Instead of relying on a single model, an ensemble model aggregates the predictions of several base models, often referred to as **weak learners**, to generate a final prediction. This approach leverages the diversity of the base models, such as using different algorithms or subsets of the training data, to improve overall performance, reduce overfitting, and enhance the generalization ability of the model. Common ensemble methods include bagging, boosting, and stacking.
In online machine learning, dealing with concept drifts involves retraining models, potentially leading to forgotten past concepts. For recurrent concepts, maintaining specialized knowledge is important, and can be achieved by employing separate models for each concept. However, selecting the right model at each step is challenging, highlighting the complexity of managing ensemble models in online learning.

Bias-Variance Tradeoff

The ideal model finds the optimal bias-variance **tradeoff**. Weak learners have high bias/low-variance. An ensemble of weak learners improves the overall performance. The bias is decreased if the errors are decorrelated. We have to ensure the weak learners are different. This means that each weak learner should focus on different aspects or subsets of the data, contributing unique insights to the ensemble. By diversifying the weak learners in this way, the ensemble can capture a wider range of patterns in the data and produce more accurate and robust predictions overall.



Ensemble Properties

An ensemble should have the following properties.

- **Diversity**: ensembles work best when the base models are different and have no correlated errors. This can be achieved thru horizontal partitioning (training in different chunks of data) or vertical partitioning (training with different subsets of features).
- **Combination**: different way to combine the models outputs. Flat: base models trained on input data, decision fusion via simple combination. Meta-learner: the combination function is a model itself. Hierarchical: structured organization of the base learners. Network: base models are nodes in a graph, the graph structure informs the combination scheme. There's also different **voting schema**: Majority vote: every classifier has the same weight. Weighted majority: gives a different weight to each classifier. Classifier selection: uses a dynamic criterion to select the best classifier for the current sample.
- **Adaptation** (for OML): how many base models? Fixed, the number of base learners cannot grow, or Dynamic, add classifiers on the fly.

1.4.2 Offline Ensemble

Offline ensemble methods involve training multiple individual models on the entire dataset and then combining their predictions to make a final decision. These methods are typically used when the entire dataset is available upfront and can be divided into training and validation sets. Three popular offline ensemble methods are Bagging, Random Forest, and Stacking.

- **Bagging** (Bootstrap Aggregating): involves training multiple base models independently on different bootstrapped samples of the training data, where each sample is created by randomly sampling the data with replacement. This results in a diverse set of models that have been exposed to different subsets of the data. The final prediction is made by averaging or voting over the predictions of all base models. Bagging helps reduce overfitting and variance in the predictions by leveraging the diversity of the base models.
- **Random Forest**: is an extension of bagging that adds randomness to the model-building process. In addition to training base models on bootstrapped samples, Random Forest randomly selects a subset of features at each split in the decision tree building process. This introduces further diversity among the base models and helps decorrelate their errors. The final prediction is made by averaging or voting over the predictions of all decision trees in the forest.
- **Stacking**: involves training multiple base models on the entire training dataset and then combining their predictions using a meta-model. The meta-model is trained on the predictions of the base models, effectively learning how to weigh or combine their outputs to make the final prediction. Stacking allows the models to learn from each other's strengths and weaknesses, potentially improving overall performance. It is a more complex ensemble method compared to Bagging and Random Forest but can yield higher predictive accuracy when properly tuned.

Bootstrapping

1.4.3 Online Ensembles

Accuracy-Weighted Ensemble

The idea is to have an ensemble of **models** that is **trained on chunks of the data stream**. As new data arrives, a new model is trained on the latest chunk and added to the ensemble. To maintain a constant ensemble size, the oldest model is removed. This strategy ensures the ensemble adapts to changes in the data stream over time but requires setting a fixed window size, which determines the number of models in the ensemble and the amount of data each model is trained on. Each classifier is weighted by the expected accuracy on the future data ($w_i = err_i - err_{rand}$) estimated using a separate test set with recent data. The final ensemble output is a linear combination of the weights and the single model outputs: $f(x) = sign(\sum_i w_i C_i(x))$.

Online Bagging

In the online settings, since data arrives sequentially, each sample is assigned a discrete weight as it is received. This weight determines how many times the sample is considered in training the ensemble of models, effectively simulating the replacement aspect of **bootstrap** sampling. The number of replicas for a specific example in a bootstrapped set follows a **Binomial** distribution. This is because bootstrapping involves performing n independent Bernoulli trials (where each trial is the selection of an example), each with two possible outcomes: either an example is selected (with probability $p = 1/n$) or it is not (with probability $1 - p$). For large n, the binomial can be approximated with a **Poisson** distribution: draw weights according to a Poisson with $\lambda = 1$.
ADWIN Bagging is an adaptation of the Bagging ensemble method designed to address the challenges posed by concept drift in streaming data. In ADWIN Bagging, a traditional Bagging ensemble is combined with the ADWIN algorithm for concept drift detection. The ensemble consists of multiple base models, each equipped with an ADWIN change detector to monitor errors over time. When a change in the data distribution is detected by an ADWIN detector, the worst-performing classifier is removed from the ensemble using a "replace the loser" strategy, and a new classifier is trained to replace it.

Hoeffding Option Tree

The **Hoeffding Option Tree** (HOT) is an online ensemble learning method that integrates multiple decision trees within a single tree structure. **Option Nodes** within the tree represent points where the tree can split into multiple subtrees, essentially creating an ensemble of decision paths within one tree. At these nodes multiple tests will be applied, implying that an example can travel down multiple paths of the decision tree, and arrive at multiple leaves. For inference, HOT evaluates all subtrees in parallel, and the outputs from the leaves are aggregated, often by weighted voting, to make a final decision. During training, if the splitting criterion (based on the Hoeffding bound) is similar for different attributes, the tree is branched into several options instead of choosing a single best attribute to split on, as is done in a standard Very Fast Decision Tree (VFDT). This branching

reduces instability and allows for earlier splitting, leading to a more robust model as it can capture more complex patterns in the data.

Streaming Random Patches

The Streaming Random Patches (SRP) method is an adaptation of the Random Forest algorithm tailored for online learning. SRP employs global subspace randomization, where each model in the ensemble is trained on a unique, randomly selected subset of features. This differs from the **Adaptive Random Forest** (ARF), which uses local subspace randomization to determine features for node splits within each tree.

Recurrent Concepts Drift

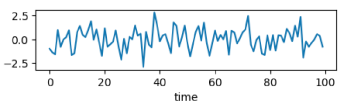
Recurrent Concepts Drift (RCD) refers to the phenomenon where the data distribution changes over time but eventually **previous concepts may reappear**. This can be common in scenarios with seasonal trends. A strategy to deal with RCD is to **maintain two sets of classifiers**: one with a library of classifiers for past concepts (currently inactive), and another set with an ensemble of active models that represent the current concept. The idea is that instead of learning from scratch every time a concept reoccurs, the system can reactivate appropriate models from the library that have already learned the concept.
For concept drift detection, we could use the Drift Detection Method (DDM), which monitors the model's error rate; a significant increase in error signals a potential drift. When a warning level is reached a new model begins training on a buffer of recent samples. If the drift is confirmed, this buffer is compared to stored buffers to determine if the new concept is indeed a recurrence of an old concept. If it is, the corresponding model from the library is reactivated, allowing the system to quickly adapt without the need to relearn the concept.

1.5 Time Series Analysis and Forecasting

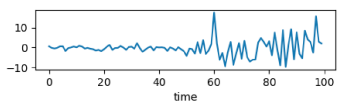
Time series analysis involves studying the patterns, trends, and dependencies present in the data over time. **Time series forecasting**, on the other hand, focuses on using historical data to make predictions about future values of the series.

1.5.1 Stochastic Processes and Stationarity

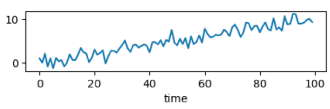
Strong stationarity refers to a property of a time series where all of its statistical properties remain constant over time. In other words, the probability distribution does not depend on time. Formally, given a stochastic process $\{X_t\}_{t\in T}$ and its cumulative distribution function $F_X(x_{1+\tau}, ..., x_{n+\tau}) = F_X(x_1, ..., x_n)$. Ensuring strong stationarity is challenging; therefore, **weak stationarity** is employed as an alternative. Weak stationarity is a property of a time series where the mean, variance, and auto-covariance are constant over time, but the distribution of the data may not necessarily be stationary. In other words, the average behavior of the series remains constant over time, but individual data points may still exhibit randomness. This property simplifies the analysis of time series data, allowing for easier analysis and predictability. We consider a time series with constant mean, variance and no seasonality stationary.



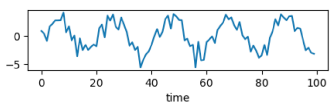
Stationary



Non-constant Variance



Non-constant mean (trend)



Seasonality

Sometimes, non-stationarity is obvious, it is possible to visually detect it. There's also more principled approach: statistical tests (ADF Test, KPSS Test).

1.5.2 Decomposition and Detrending

We can **decompose** time series on trend, seasonality and residual. $TS = T \odot S \odot R$ where we can have, for example, $+$ or $\cdot$ as the operator. Additive model assumes the seasonal is approximately constant over time. Multiplicative model is better when the seasonal component changes over time according to the general trend.

Trend Elimination

We want to model the trend only, ignoring the seasonal component. Differenciating $x_t^{new} = x_t - x_{t-1}$ is a simple method to remove the linear trends, if you have a polinomial trends (degree n) you need to difference n times.
Trend estimation involves fitting a trend model, such as a linear or polynomial function, to a time series data set to capture its underlying trend. Once the model parameters are estimated, the trend component can be removed from the original data to obtain a detrended series. For a linear trend, the model has the form $\bar{x}_t = at + b$, where a and b are parameters to be fitted, and the detrended series is given by $y_t = x_t - \bar{x}_t$, subtracting the estimated trend from the original time series (x_t). This detrending helps in analyzing the time series data without the influence of the trend.

Seasonality Elimination

Seasonal differencing with period d is a technique used when a time series exhibits a fixed and known seasonal pattern, such as daily, weekly, etc cycles. The method needs first to have the trend removed then subtracts the data point in one season from the data point in the same season in the previous cycle: $x_t^{new} = x_t - x_{t-d}$.
In conjunction with these, centered moving averages can be used to smooth out fluctuations and estimate the trend within a time series. This involves averaging data points within a window centered around each time point. For seasonality removal, the time series is first detrended, then divided into windows matching the seasonal period, and the mean of each position within the period is computed to identify and subtract the seasonal pattern. These methods are essential for preparing time series data for further analysis or forecasting by isolating irregular components from the seasonal and trend components.

1.5.3 Forecasting

Given a time series $x_1, ..., x_n$, we want to **predict** x_{n+k} (k steps ahead). There's two approaches: first, preprocess to make the TS stationary, then train the forecasting model or train the forecasting model directly. For evaluation, various metrics such as Mean Absolute Error (MAE), Mean Absolute Percent Error (MAPE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and others can be utilized. As a basic benchmark, predicting the new value x_{n+1} could involve methods like using the average, window-based average, or the last observed value.

1.5.4 Exponential Smoothing

Exponential smoothing methods use **weighted averages** of past observations, with the weights **decreasing exponentially** as the observations get older. This approach yields fast and reliable **forecasts**.

Simple Exponential Smoothing

In simple exponential smoothing the time series does not exhibit a trend or seasonality. It assumes that the time series is essentially a flat line (the level) with random noise around it. For $t = 1, ..., T$, the smoothed values, the forecasts at each time value are obtained recursively:

$$\hat{y}_{t+1|t} = \alpha y_t + (1 - \alpha)\hat{y}_{t|t-1} \tag{1.7}$$

Holt's Method

Holt's method can capture additive trend or multiplicative trend. This trend can vary adaptively over time.

$$\begin{aligned}
\text{Forecast equation: } & \hat{y}_{t+h|t} = \ell_t + hb_t \\
\text{Level equation: } & \ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\
\text{Trend equation: } & b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1}
\end{aligned}$$

where: b_t is the trend at time t, α is the level smoothing parameter and β^* is the trend smoothing parameter. The level equation updates the current level ℓ_t as a weighted average of the current observation y_t and the sum of the last level and trend $(\ell_{t-1} + b_{t-1})$. The trend equation updates the current trend b_t as a weighted average of the difference between the current and previous level $(\ell_t - \ell_{t-1})$ and the previous trend b_{t-1}.

Holt-Winter's Method

Holt-Winter's method can capture local additive or multiplicative trend and/or seasonality.
The addictive model is:

$$\begin{aligned}
\hat{y}_{t+h|t} &= \ell_t + hb_t + S_{t+h-m(k+1)} \\
\ell_t &= \alpha(y_t - S_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\
b_t &= \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1} \\
S_t &= \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)S_{t-m}
\end{aligned}$$

where $k = \text{floor}\left(\frac{h-1}{m}\right)$, seasonal component s_t with smoothing factor γ and m seasonality period.
And the multiplicative model is:

$$\begin{aligned}
\hat{y}_{t+h|t} &= (\ell_t + hb_t)S_{t+h-m(k+1)} \\
\ell_t &= \alpha\frac{y_t}{S_{t-m}} + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\
b_t &= \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1} \\
S_t &= \gamma\frac{y_t}{(\ell_{t-1} + b_{t-1})} + (1-\gamma)S_{t-m}
\end{aligned}$$

Knowledge Transfer and Adaptation

2.1 Introduction: Deep Learning Tips & Tricks

2.1.1 Deep Learning Concepts

Training a deep neural network enables the automation of feature extraction by **stacking multiple layers** sequentially. This approach empowers the network to learn increasingly complex representations of the data as it progresses through each layer, ultimately leading to more sophisticated and abstract features being extracted.
When learning multiple tasks we can consider the following points:

- **Sharing layers** can be beneficial in neural network architectures.
- It is possible to train a network on one task and reusing it for **another downstream task**.
- **Learning generic features** that are useful for a broad range of tasks is a consideration.

So, with DL a learned latent representation can also be reused, opening up many new applications.
Batch normalization is a technique in deep learning that helps accelerate the training of neural networks by reducing the internal covariate shift in order to standardizes the outputs of the hidden layers.
Dropout is a regularization technique used to prevent overfitting by randomly dropping out a fraction of neurons during training.
ReLU, or **Rectified Linear Unit**, is an activation function used in neural networks that introduces non-linearity by outputting the input directly if it is positive, and zero otherwise. It is helpful because it's cheap to calculate, it accelerates the convergence of stochastic gradient descent, mitigates the vanishing gradient problem, and introduces sparsity in the network, improving its generalization ability.
Pooling is a technique used in convolutional neural networks (CNNs) to reduce the spatial dimensions of feature maps by downsampling them.
A **residual connection** is a shortcut connection in a neural network that allows the input of a layer to bypass one or more layers and be added to the output of subsequent layers. Residual connections improve the gradient flow by allowing to skip layers.

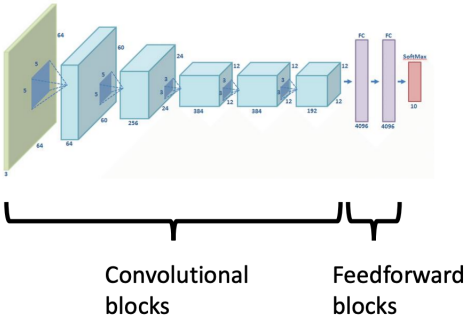


Figure 2.1: Convolution Neural Network

For a classification taks, **Logits**, which are the outputs from a model's final layer, serve as the input to the **Softmax** function that translates them into normalized probabilities. **Cross-entropy loss** then measures the discrepancy between these predicted probabilities and the true labels, guiding the model during training. Rather than calculating Softmax probabilities directly, modern practices often involve computing Cross-entropy from logits to improve numerical stability. During inference, identifying the highest probability class does not require explicit Softmax probabilities, as ordering is preserved. It's important to refer to the machine learning framework's documentation to determine the appropriate usage of logits or probabilities.

2.1.2 Tips and Tricks

Full-scale optimization is often impractical: not all hyperparameters significantly impact model performance, though some, like the learning rate, are crucial. There's also interplay between hyperparameters. The slides suggest consulting Google Research's DNN Tuning guidelines for practical advice `https://github.com/google-research/tuning_playbook`. It's useful to start with a proven model from literature as a baseline for improvement.
Proper model initialization is crucial for the early stability of neural network training. Networks often struggle to recover from poor initial starting weights, leading to suboptimal performance or lack of convergence. Overlooking weight initialization is a widespread mistake, manifesting as training instability.
In computer vision tasks, optimal performance is often achieved using Stochastic Gradient Descent (SGD) with momentum and a strategically scheduled learning rate. It's recommended to start training with a higher learning rate and decrease it gradually as training progresses.

2.2 Finetuning and Transferability

2.2.1 Transfer Learning Definition

Transfer learning is a machine learning technique where a model trained on one task is **adapted** or **transferred** to a related task, typically by leveraging the knowledge learned from the source task to improve performance on the target task.
In transfer learning, we address a **target task** T_b using a dataset D_b sampled from it. This technique leverages knowledge acquired from solving related **source task**(s) T_a to improve performance on T_b, even though the dataset D_a used for T_a is not available during transfer learning. Typically, D_a is large while D_b is small, and D_a might not be accessible, such as in the case of a **pretrained model** from a private company. In transfer learning, it is assumed that the tasks involved are related, implying that discriminative features learned for task T_a are beneficial for task T_b.
In **Multi-Task Learning** (MTL), data for all tasks is available simultaneously, and we train a model on multiple tasks that usually have similar size and complexity. **Transfer Learning** (TL), on the other hand, typically involves two tasks, with the second task having much less data than the first.

2.2.2 Warm Start

Here are some popular optional choices in transfer learning:

- **Epochs**: typically, fewer iterations/epochs are needed compared to training from scratch.
- **Learning rate**: a new, often smaller, learning rate (α).
- **Weight decay**: may be set to 0.
- **Freezing**: use a small learning rate or freeze early layers.
- **Reinitialization**: randomly reinitialize the last layers.

One in particular is very important, **Warm Start**: train only the last layer initially, then finetune the entire model. In this approach, we begin with a pretrained model θ_a and freeze all its parameters except for the classifier. The classifier is randomly initialized and then fine-tuned on the target task. Subsequently, we unfreeze all parameters and fine-tune the entire model. The rationale behind this

method lies in the fact that the randomly initialized classifier may produce large gradients, leading to significant changes in the deep neural network (DNN). Warm start aims to mitigate the "**forgetting**" of learned representations, although it may not always be the optimal choice.
It is well-established that **lower layers** of a neural network serve as **general feature extractors**, while **higher layers** tend to capture more **task-specific features**. However, if the domain is significantly altered, transfer learning may become ineffective.

2.3 Domain Adaptation

2.3.1 Definition

When considering a specific task, a domain is a subset of that task. For instance, when the task is image classification of animals, various domains can be defined. These domains may include different environments such as jungle or savannah, as well as variations in images such as distant views, close-ups, and images with varying resolutions.
We will use the following terminology:

- **Source Domain**: the data distribution on which the model is trained using labeled examples.
- **Target Domain**: the data distribution on which a model pretrained on a different domain is used to perform a similar task.
- **Domain Translation**: the problem of finding a meaningful correspondence between two domains.
- **Domain Shift**: a change in the statistical distribution of data between different domains.

Domain adaptation is a technique used to transfer knowledge from a source domain to a target domain, where the source and target domains have different distributions of data. The goal is to improve the performance of a model trained on the source domain when applied to the target domain, by reducing the distributional discrepancy between the two domains.
Domain adaptation can be seen as a specific case of transfer learning, and can be applied in situations where there is only a **covariate shift** between source and target data

2.3.2 Domain Adaptation Methods

The assumptions underlying the scenario involve distinct yet **closely related source and target domains**, where a single hypothesis, such as a model or deep neural network (DNN), demonstrates low error rates on data from both domains. Unlike transfer learning, where the source and target tasks may differ substantially, here the tasks remain closely aligned. The transition from the source to the target domain is considered a form of **virtual drift**.
An interesting example from the life sciences is training a classifier for predicting protein interactions in some species for which biologically validated interactions are known, and applying this classifier to other species, for which no such data validated interactions exist. In such case what we would like to do is make sure that our model **performs well on the target data, while still training on the source data**.

Data Re-weighting

In our domain adaptation problem, we have two joint distributions, namely the source distribution $p_S(x, y)$ and the target distribution $p_T(x, y)$, model trained on the source distribution ignores samples from the target distribution. In training on the empirical source distribution, we want optimize for performance on the target distribution. To do this, we consider the Error on source $\mathbb{E}_{p_S(x,y)}[\mathcal{L}(x, y, \theta)]$ and the Error on target $\mathbb{E}_{p_T(x,y)}[\mathcal{L}(x, y, \theta)]$, and apply the following trick for transferring knowledge

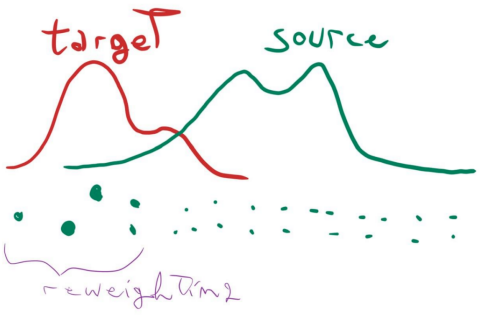
from our source domain to our target domain.

$$\begin{aligned}\mathbb{E}_{p_T(x,y)}[\mathcal{L}(x,y,\theta)] &= \int p_T(x,y)\mathcal{L}(x,y,\theta)\,dx\,dy \\ &= \int p_T(x,y)\frac{p_S(x,y)}{p_S(x,y)}\mathcal{L}(x,y,\theta)\,dx\,dy \\ &= \mathbb{E}_{p_S(x,y)}\left[\frac{p_T(x,y)}{p_S(x,y)}\mathcal{L}(x,y,\theta)\right]\end{aligned}$$

Thus,

$$\theta^* = \arg\min_\theta \mathbb{E}_{p_S(x,y)}\left[\frac{p_T(x,y)}{p_S(x,y)}\mathcal{L}(x,y,\theta)\right]$$

We can train on the source data while optimizing for performance on the target data: we want to minimize error on target domain by **weighing source data** by $\frac{p_T(x,y)}{p_S(x,y)}$. Remember that the joint probability $p(x,y)$ can be expressed as conditional probability $p(x,y) = p(y|x)p(x)$. In the context of domain adaptation and transfer learning, $p(y|x)$ is often considered domain-independent because it represents the relationship between the input features x and the output labels y. This relationship is determined by the underlying task or problem, which is assumed to remain the same across different domains. So we can ignore it, and the **weight** becomes $p_T(x)/p_S(x)$.



Importance sampling is a statistical technique used to estimate the expected value of a function under one distribution by sampling from another distribution. In this context, re-weighted empirical risk minimization is essentially equivalent to importance sampling, with the assumption that the conditional class probabilities between the source and target data are identical.
The problem with the above result, is that $p_T(x)$ and $p_S(x)$ are difficult to determine. We can apply Bayes rule to the importance sampling coefficient.

$$\frac{p_T(x)}{p_S(x)} = \frac{p(x|target)}{p(x|source)} = \frac{p(target|x)p(source)}{p(source|x)p(target)} \tag{2.1}$$

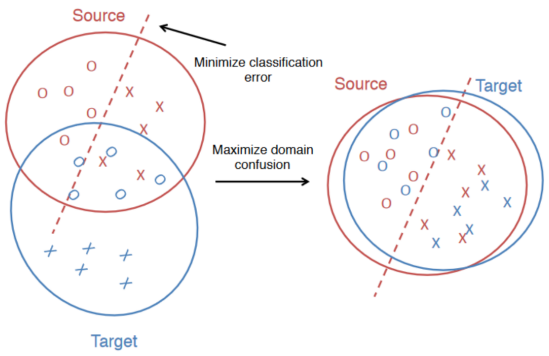
The term $p(source|x)$ functions as a binary domain classifier (whether x comes from the source or the target) and the constant term $p(source)/p(target)$ doesn't affect the optimal solution, thus can be ignored in calculations.
The training algorithm becomes:

1. Train domain classifier $p(source|x;\lambda)$ to classify source/target
2. Re-weight samples by $w_i = \frac{1-(p(source|x;\lambda)}{(p(source|x;\lambda)}$
3. Minimize $w_i\mathcal{L}(x_i, w_i, \lambda)$

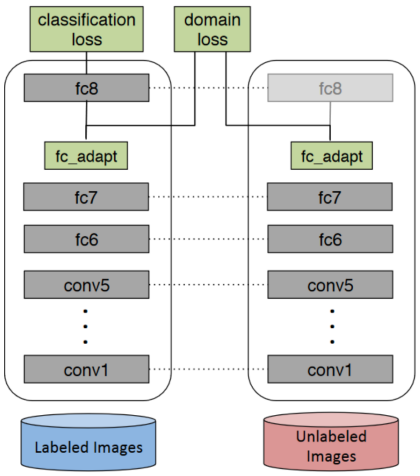
Informally, the source domain contains the target domain, approximately true if you go from a general domain (ImageNet) to a speficic one (birds classification). Probably false if you switch from one specialized domain to another.

Feature Alignment

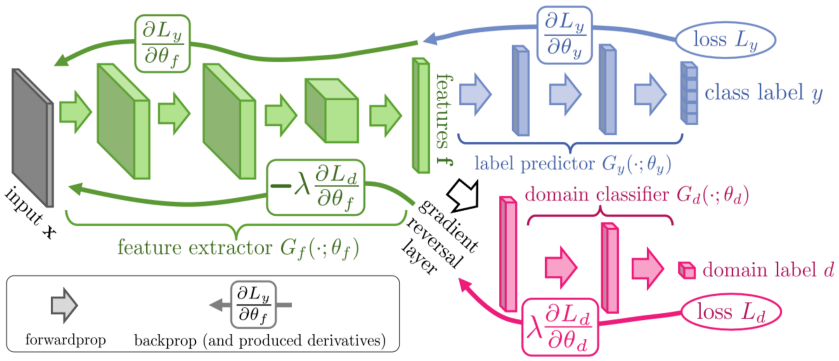
In cases where importance sampling cannot be utilized, an alternative approach involves **aligning** the **features** of the source and target datasets. The objective is to **transform** both source and target **features** into a **common aligned feature spac**e, enabling the **reuse** of the source classifier with the target data within this aligned feature space.
The model is split into feature extractor $f_\theta(x)$ and classifier $c_\theta(h)$. We want source and target features that have the same distributions, be **domain invariant** meaning that features should be invariant w.r.t. domain. If the features have the same distribution, a trained domain classifier should have a random accuracy. Can we train the feature extractor to "fool" the domain classifier?



A **Deep Domain Confusion Network** is composed by a shared CNN feature extractor and a domain adaptation layer. We want to learn a representation that is both semantically meaningful and domain invariant that. The loss consists in two parts: a domain confusion loss to minimize domain-distance and a classification loss.



A **Domain-Adversarial Neural Network** works in the following way. The classifier and feature extractor are trained to classify the source data, aiming to learn discriminative features. Also, a domain classifier is trained to predict the domain. The feature extractor is trained with a GAN-like objective to deceive the domain classifier. The networks is composed by three modules: G_f is the feature extractor, G_y is the label predictor and G_d is the domain classifier.



The domain classifier G_d is trained to guess the domain, the classifier G_y is trained to classify the source data (target is unlabeled). The feature extractor is optimized to improve the classification and to fool the domain classifier. The feature extractor and domain classifier optimize the same objective in opposite directions.

Domain Translation

CycleGAN: `https://github.com/lorenzoleuzzi1/unipi/blob/main/ispr/ispr-midterm4.pdf`

2.4 Multi-Task Learning

2.4.1 Definition

A **task** refers to a specific problem or objective in machine learning that involves processing data D, using a loss function $\mathcal{L}$ to guide the learning process, and producing a model f that solves the problem. Tasks can vary widely, they can include working with different kinds of objects or people, and can address different problems such as classification, detection, or segmentation. Each task can involve data from various sources, which might have different features, like distinct objects or backgrounds, and different domains. The nature of the task dictates the design choices in the modeling process, such as the architecture of the model and the techniques used to train it.

Multi-task learning (MTL) is a machine learning approach where multiple learning tasks are solved at the same time, with the goal of mutual benefit. The objective in MTL is to find an optimal set of model parameters θ that minimize the combined loss $\mathcal{L}_i$ over all tasks D_i. The key advantages of MTL are:

- **Concurrent Task Solving**: It allows the model to solve multiple tasks simultaneously, improving efficiency.
- **Knowledge Sharing**: The model can leverage shared knowledge between tasks, which often leads to better learning and generalization.
- **Task Relationship Exploitation**: By understanding the relationships between tasks, MTL can help the model to converge faster and potentially achieve better performance on each task.

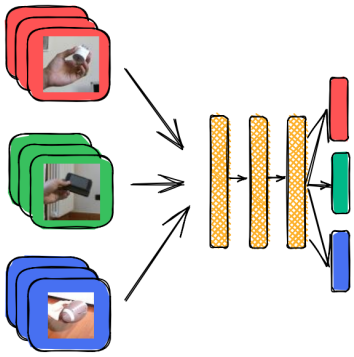
A critical assumption in MTL is that the tasks **share some common structure**, which assists in learning them jointly. However, this shared structure can also cause interference, potentially leading to difficulties if tasks are too dissimilar or compete with each other for model capacity.
Multi-task learning (MTL) problem types often include:

- **Multi-task classification**: the loss function $\mathcal{L}$ is shared among multiple tasks. This implies that while training a single model, it learns to classify across different but related classification problems simultaneously.
- **Multi-label learning**: the model predicts multiple outputs $p(x)$ for each input. Unlike multi-task classification, where each task may have its own set of classes, multi-label learning often involves predicting multiple labels for each instance within a single dataset.

Additionally, MTL may leverage a **task descriptor** z_i, which can be a simple identifier (like an integer) or a more complex representation (like a vector of task features), to inform the model about which task it is currently learning or predicting. This descriptor helps the model to switch contexts or apply specific parts of its learned knowledge to the task at hand.

The objectives of Multi-Task Learning (MTL) go beyond simply learning multiple tasks simultaneously. The key **goals** include:

- Fast Adaptation: MTL aims to learn tasks more quickly than if learned independently, with multi-task models converging faster than single-task models.
- Forward Transfer: MTL seeks to improve the generalization capabilities of the model, where the multi-task model should ideally achieve a lower loss across tasks compared to models trained on individual tasks, particularly in situations with limited data.
- Meta-Learning: Through MTL, models are designed to learn new tasks more efficiently using knowledge gained from learning previous tasks. This is especially effective when the model has been pretrained on a range of tasks and needs to adapt to new, related tasks.
- Few-Shot Learning: MTL can be particularly advantageous in few-shot learning scenarios, where the model must generalize from a very limited amount of data, often just a few examples.

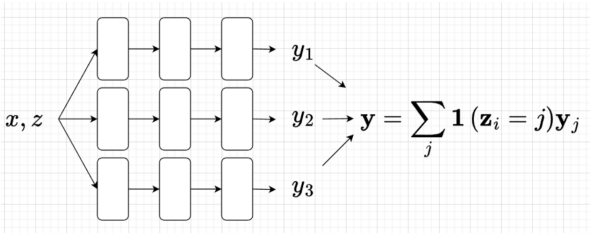


2.4.2 Design Choices

When deciding which **layers** to **share** among tasks, several factors should be considered. The similarity between tasks plays a crucial role; tasks that are more alike can share more layers, as they are likely to benefit from the same underlying feature representations. Also the **complexity** of each task should be taken into account; simpler tasks may require fewer specialized layers, whereas more complex tasks might necessitate additional dedicated parameters. Lastly, the **availability of data** for each task should be observed; tasks with limited data may benefit from sharing more layers to prevent overfitting and promote better generalization.

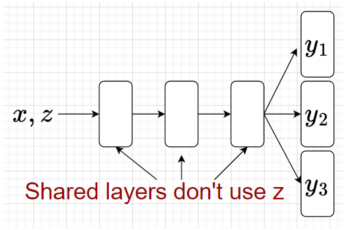
Weight Sharing – No Thank You

In **No Weight Sharing** MTL **independent networks** are used for each task without any shared components. This implies that for each specific task, a completely separate model with its own parameters is trained. The multiplicative gating mechanism is a way to choose which model should be applied to a given input. This mechanism works by assigning a task identifier z_i to each input and using it to select the corresponding model y_j whose output should be considered. If the task identifier z_i matches the model identifier j, then that model's output is used. This approach avoids any negative interference that might occur due to sharing weights between tasks that are not closely related or do not benefit from shared representations.



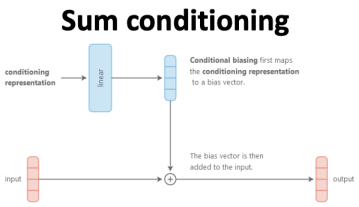
Weight Sharing – Multi-Head

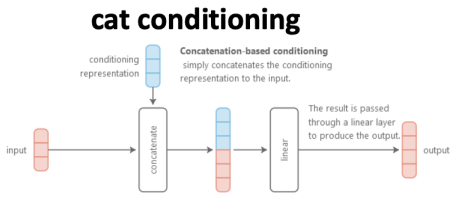
Multi-Head weight sharing is an architecture that utilizes a **shared feature extractor** and **multiple task-specific classifiers**, also known as "heads." In this setup, all tasks utilize a common set of layers to extract features. This shared feature extractor is beneficial because it allows the model to learn general features that are useful across different tasks. Then, for each specific task, a separate classifier head is trained. When an input is presented to the network, a gating mechanism determines which head (classifier) should be applied to the extracted features for that particular task. This gating mechanism is often based on task descriptors or some form of conditioning that enables the network to identify the task for which the input is intended. It's important because it ensures that while all tasks benefit from the shared representation learned by the common feature extraction layers, they still maintain task-specific decision boundaries learned by the individual classifier heads.

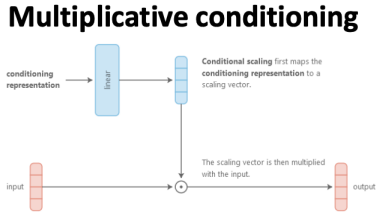


Weight Sharing - Task Conditioning

Task Conditioning refers to modifying the shared representation for each specific task. The conditioning can be done in various ways, such as adding a task-specific vector (addition), concatenating a task descriptor with the input (concatenation), or scaling the activations (multiplicative conditioning). The network's parameters are divided into shared (θ^{sh}) and task-specific (θ^i) parts. The shared parameters are used across all tasks, while the task-specific parameters are unique to each task. This allows the network to learn both general features and task-specific nuances. The objective of the MTL framework is to minimize the sum of the loss functions across all tasks ($\sum_{i=1}^{T} \mathcal{L}_i(\theta^{sh}, \theta^i, \mathcal{D}_i)$).







2.4.3 Transfer and Interference

The effects of transfer learning can be positive or negative in the context of MTL, where tasks are learned jointly, typically through sharing weights among neural network models.

- **Positive Transfer** refers to the benefits of joint training, where learning multiple tasks together leads to better performance on individual tasks. This is especially true for small tasks, where a model that is trained on multiple tasks is less likely to overfit to one specific task and instead develops a more generalized solution.
- **Negative Transfer** is a potential downside of multi-task learning, where the interference between tasks can actually worsen the performance. This can happen for several reasons, such as tasks interfering with each other (cross-task interference), tasks learning at different rates, or the model lacking sufficient representational capacity to handle multiple tasks. To mitigate this, MTL networks often need to have increased capacity (i.e., be larger) to accommodate the complexity of multiple tasks.

It is not yet possible to guess in advance whether multi-task training will results in positive/negative transfer.

2.5 Self-Supervised Learning

2.5.1 Definition

Supervised Learning, since its inception, has suffered from the lack of **availability of annotated data** and the huge costs associated with annotating new data. It is exactly this drawback of Supervised Learning that Self-Supervised Learning (SSL) aims to improve. Self-supervised learning is a form of predictive learning where a model is trained using a large **unlabeled dataset**. The **Pretext Task** (a form of pre-training), is the self-supervised task that defines a supervised loss given the unsupervised data. After the model is trained on the pretext task, it's then fine-tuned and evaluated on separate **downstream tasks**, which are the actual problems we aim to solve. The key point in SSL is that the pretext task is merely a method to pre-train the deep neural network (DNN) to learn useful representations; the real interest lies in the performance on the downstream tasks, not the pretext task itself.

An **example** of self-supervised learning is **BERT**, a language model trained through a self-supervised approach. In this method, the model's objective is to perform **Masked Language Modeling** (MLM). It takes an input sequence of tokens with some tokens intentionally masked, and the objective is to predict these masked tokens from their surrounding context. This process enables the model to learn rich contextual representations of language without requiring annotated data.

Different methods of self-supervised learning exist:

- **Augmentations**: exploit the properties of the domain (vision), such as invariances to transformations, to learn robust representations.
- **Contrastive learning**: learn representations by comparing and contrasting pairwise images.
- **Encoding/Autoregressive**: learn to reconstruct the input or predict a missing part of the input (not covered because *ISPR* and *HLT*).

2.5.2 Augmentations

Augmentation is a concept in self-supervised learning that aims at achieving **invariance** to input modifications through pretext tasks. The core idea is that applying **small changes** (like rotations or adding noise) to an image shouldn't alter its inherent class or characteristics. The pretext task in this context involves a batch of images that have undergone various augmentations, and the model needs to predict whether these modified images are variations of the same original image. The training process requires the model to focus on the essential features that define an image and to disregard

the alterations made by the augmentations. In other words, the model is being trained to understand that, despite superficial changes, the **underlying content remains constant**.

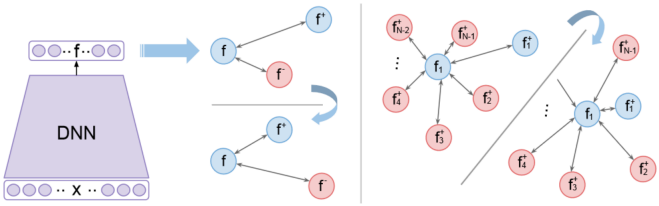
2.5.3 Contrastive Learning

Contrastive learning is an approach to learning that focuses on extracting meaningful representations (for downstream tasks) by **contrasting positive and negative pairs of instances**. It leverages the assumption that similar instances should be closer together in a learned embedding space, while dissimilar instances should be farther apart. By framing learning as a discrimination task, contrastive learning allows models to capture relevant features and similarities in the data.
The goal is to minimize the distance between the anchor and positive and maximize the distance between the anchor and negative using a **Triplet Loss**. Where anchor is the embedding representing the class, positive is the embedding of an image with the same label as the anchor and negative is the embedding of an image from a different class.

$$\mathcal{L}_{\text{triplet}}(\mathbf{x}^a, \mathbf{x}^+, \mathbf{x}^-) = \sum_{\mathbf{x} \in \mathcal{X}} \max\left(0, \|f(\mathbf{x}^a) - f(\mathbf{x}^+)\|_2^2 - \|f(\mathbf{x}^a) - f(\mathbf{x}^-)\|_2^2 + \epsilon\right) \tag{2.2}$$

Contrastive Learning

Triplet sampling ($< x_i^a, x_i^+, x_i^- >$) is crucial, ideally, we want to pick the most difficult examples e.g. the positives which are further from the anchor (hard positive) and the negative closer to the anchor (hard negative).
Triplet loss can converge slowly because it updates the model using only one positive and one negative for each anchor. **N-way loss**, also known as N-pair loss, is an extension of triplet loss. Instead of considering just one negative example, N-way loss compares the anchor against $N-1$ negative examples. In doing so, it simultaneously considers **multiple negative pairs for each positive pair**, which provides a richer signal for each update and can lead to faster convergence. In practical terms, when you use N-pair loss, for each anchor-positive pair, you also consider $N-1$ negative pairs, making it a more efficient approach (the same N example are reused for all the minibatch). It generalizes the triplet loss framework to go beyond the single negative example, which can be particularly useful when you have a large batch of data to train on. The computation for N-pair loss involves all the N pairs within a batch, while traditional triplet loss would require $(N+1) \times N$ computations, making N-pair loss computationally more efficient.



Negative mining is a process used in contrastive learning to identify the **most challenging negative samples** that will help the model learn more effectively. When implementing N-way loss, the goal is to select negative samples that are difficult for the model to distinguish from the positive samples, which can lead to a more robust learning process.
The steps for negative mining are:

- Evaluate Embedding Vectors: randomly select a large number of output classes. For each class, randomly choose a small number (typically one or two) of examples and extract their embedding vectors.
- Select Negative Classes: randomly select one class from the classes evaluated in step 1. Add new classes in a greedy fashion that are most likely to violate the triplet constraint with respect to the initially selected class. This means selecting classes whose examples are similar (close in embedding space) to the positive class but are actually different classes (negatives). If there is a tie in the selection process (meaning multiple classes are equally difficult), one is chosen at random.
- Finalize N-pair: from each selected class from step 2, draw two examples to create the N pairs for training.

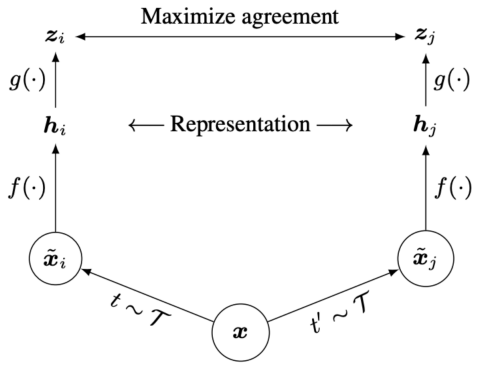
The similarity $f^T f^+$depends on the direction and norm of the embeddings. Unconstrained optimization leads to unbounded norm growth pushes embeddings further without changing their direction, which we want to avoid. In practice, penalizing the embedding norm $||\mathbf{f}||_2^2$ has shown effectiveness. This approach effectively prevents unbounded growth by imposing a penalty on the norm, yet it doesn't overly constrain the model's behavior.

SimCLR

SimCLR is a Simple framework for Contrastive Learning of visual Representations. It learns representations by maximizing agreement between differently augmented views of the same data example via a contrastive loss in the latent space. It consists of:

- **Augmentations**: a stochastic data augmentation module that transforms any given data example randomly resulting in two correlated views of the same example, denoted $\tilde{\mathbf{x}}_i$ and $\tilde{\mathbf{x}}_j$, which is considered a positive pair. SimCLR sequentially applies three simple augmentations: random cropping followed by resize back to the original size, random color distortions, and random Gaussian blur.
- **Base encode**: $\mathbf{h}_i = f(\cdot)$ a neural network base encoder that extracts representation vectors from augmented data examples. The framework allows various choices of the network architecture without any constraints.
- **Projection head**: a small neural network $g(\cdot)$ that maps representations to the space where contrastive loss is applied.

The **contrastive loss** is the softmax on similarity scores of the $\mathbf{z_i}, \mathbf{z_j}$ outputs.



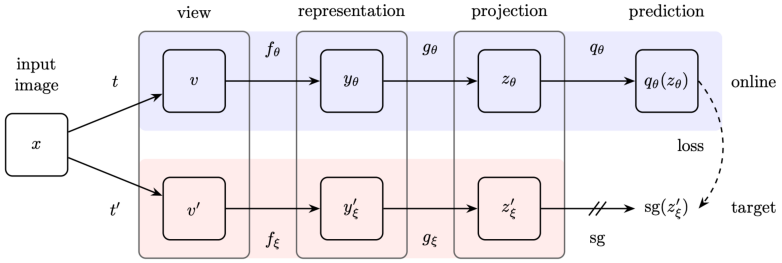
A minibatch of N examples is randomly sampled and the contrastive prediction task is defined on pairs of augmented examples derived from the minibatch, resulting in $2N$ data points. Negative examples are not sampled explicitly. Instead, given a positive pair, the other $2(N-1)$ augmented examples within a minibatch are treated as negative examples.

Bootstrap Your Own Latent

BYOL (Bootstrap Your Own Latent) is a new approach to self-supervised learning. BYOL's goal is to learn a representation y_θ which can then be used for downstream tasks. BYOL uses two convolutional neural networks to learn: the online and target networks. Before passing the input to the networks, it is agumented with two different distribution $t \sim \mathcal{T}, t' \sim \mathcal{T}'$ of image augmentations. Both networks include three modules: an encoder f, a projector g and a predictor q, the **online network** is defined by a set of weights θ and the **target network** uses a different set of weights ξ.
BYOL **minimizes a similarity loss** between $q_\theta(z_\theta)$ and z'_ξ. The Loss is expressed as $\mathcal{L}^{\text{BYOL}}_{\theta,\xi} = \mathcal{L}_{\theta,\xi} + \tilde{\mathcal{L}}_{\theta,\xi}$, where:

$$\mathcal{L}_{\theta,\xi} = \left\| \overline{q_\theta(z_\theta)} - \overline{z'_\xi} \right\|_2^2 = 2 - 2 \cdot \frac{\langle q_\theta(z_\theta), z'_\xi \rangle}{\|q_\theta(z_\theta)\|_2 \cdot \|z'_\xi\|_2}. \tag{2.3}$$

and $\tilde{\mathcal{L}}_{\theta,\xi}$ is the same as $\mathcal{L}$ but with v and v' switched.
BYOL does **not backpropagate** through the target network, the stop gradient sg truncates the backpropagation. This is because the ξ weights are updated using exponential moving average of θ: $\xi \leftarrow \tau\xi + (1-\tau)\theta$. The online network update rule is: $\theta \leftarrow \text{optimizer}\left(\theta, \nabla_\theta \mathcal{L}^{\text{BYOL}}_{\theta,\xi}, \eta\right)$
At the end of training, everything but f_θ is discarded, and y_θ is used as the image representation.



2.6 Meta-Learning

2.6.1 Meta-Learning

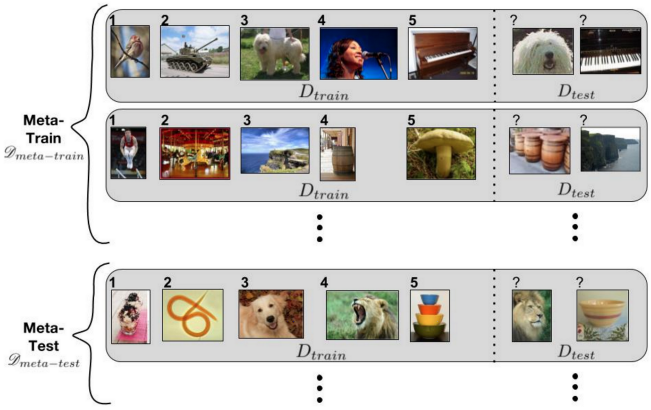
Meta-learning, often referred to as *learning to learn*, is a branch of machine learning where the focus is on the design of algorithms that can improve their learning efficiency through experience. Unlike classic learning, where a model is trained on a specific task using a single dataset, meta-learning involves training on a variety of tasks, each with its own dataset, to develop learning algorithms or models that can **generalize** from these experiences. This allows the model to perform well on new tasks with minimal additional training data.

In meta-learning the **input** is a set of datasets representing different tasks, and the goal is to optimize the learning algorithms or hyperparameters. The **output** is a learning algorithm or model that can quickly adapt to new tasks. This could manifest as a model that's been pre-trained to have a good initialization for fine-tuning, a model capable of few-shot learning (learning from a few examples), or an optimizer designed for rapid convergence on new types of data.

Few-shot Learning

Deep learning works very well but requires large dataset. In many case we only have a **small sample of data** available, but we might have lots of data of a similiar type of other tasks. Use plentiful prior task to meta-train the model that can **learn the new task quickly** with only a **few examples**. The model is trained on a variety of tasks to learn generalizable features. The *meta* part refers to

the ability of the model to adapt to a completely new task using the insights gained from the training tasks. In the test dataset for **meta-testing**, the classes have changed, which were not seen during the training phase. The model's ability to perform on this new classification task, with minimal additional training examples, demonstrates the meta-learning capability: it has learned how to learn from the training phase and can now apply this to classify new, unseen data efficiently.



Few-shot (meta) learning

Generic Learning vs Meta Learning

A **task** is defined by a distribution over inputs, a conditional distribution over outputs given an input, and a loss function $< p_i(x), p_i(y|x), \mathcal{L}_i >$. In practice, for each task, you have a dataset D_i consisting of pairs of inputs and outputs $< x, y >$ drawn from the distributions $p_i(x, y)$ and liked by the relation $f(x) \rightarrow y$ (supervised learning).
A **meta-task**, on the other hand, is a distribution of such tasks along with a loss function. This means we are not focusing on learning just one task but instead **learning a range of tasks** T_j sampled from a broader distribution $p(T)$. Meta-learning is training on a set of datasets, each sampled from a different task in T, with the aim of **learning generalizable knowledge** that can be applied to unseen tasks $D_i \sim \mathcal{T}_i, \mathcal{T}_i \sim p(\mathcal{T})$. **Supervised meta-learning**: $f(D^{tr}, x) \rightarrow y$
Thus a learner is a machine learning model and meta-learner is a parameterized learning algorithm. From the optimization viewpoint, the emphasis is on leveraging a collection of meta-datasets or tasks to identify a model or an optimization algorithm that is capable of **rapidly acquiring knowledge** about new tasks that are similar or related to the ones in the meta-dataset. Essentially, it seeks to **optimize** a model's or algorithm's **ability to learn efficiently when presented with new tasks**. Conversely, the probabilistic perspective is focused on using the given meta-datasets or tasks to **distill prior knowledge** about the structure or nature of these tasks. This extracted prior knowledge is then employed to infer the posterior distribution for new tasks. In this approach, there is a Bayesian flavor, where prior experience informs the belief about how new tasks might be structured, which in turn influences how to best approach and learn from these new tasks.

Terminology

Here's some common terminology:

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

Objective

The Objective of meta-learning is to find the optimal parameters θ^* that minimize the expected loss over the distribution of datasets. The objective function is written as $\theta^* = \arg\min_\theta \mathbb{E}_{D\sim p(D)}[\mathcal{L}_\theta(D)]$. This means we are trying to find the parameter set θ that, on average, yields the lowest loss across all datasets D sampled from a probability distribution $p(D)$.
Unlike traditional machine learning, which typically samples individual instances (like pictures or text snippets), meta-learning samples entire datasets to train on. This is because meta-learning is concerned with **learning across multiple tasks**, not just performing well on one. The datasets are assumed to be drawn from a probability distribution $p(D)$ that defines the family of tasks we are interested in. This distribution governs the variety and types of tasks that the model will encounter and needs to learn from. The learning process aims to find parameters θ that are not just suited for one specific task but generalize well over the entire range of tasks represented by the distribution $p(D)$.
A key assumption in meta-learning is that tasks have some shared underlying structure. This shared structure allows for transfer learning, where knowledge gained from one task can help improve performance on others.

Meta-Learning Families

In **model-based** meta-learning, the architecture of the model is crafted to enable quick adaptation to new tasks. This swift adaptability can be derived from the intrinsic design of the network or through the employment of a meta-learner. A key example of this approach is the use of **Memory-Augmented Neural Networks** (MANN), such as the Neural Turing Machines, which have been specifically tailored for meta-learning applications. Furthermore, these networks typically employ a meta-network structure that bifurcates the network's weights into two categories: fast and slow. The slow weights undergo gradual updates using methods like Stochastic Gradient Descent (SGD), whereas the fast weights are dynamically adjusted by the meta-learner to facilitate rapid learning and adaptation to new tasks.

In **optimization-based meta-learning**, a meta-learner is tasked with refining the model's parameters. This type of meta-learning utilizes, for example, an LSTM (Long Short-Term Memory) meta-learner to perform weight updates, functioning similarly to a learned optimizer, paralleling the update mechanics of Stochastic Gradient Descent (SGD). Another example of this approach is Model-Agnostic Meta-Learning (MAML), which focuses on learning a generalized initialization that enables the model to adapt rapidly to a variety of tasks, facilitating both fast adaptation and efficient learning from a limited number of examples (few-shot learning).

There's also **metric-based learning** (next).

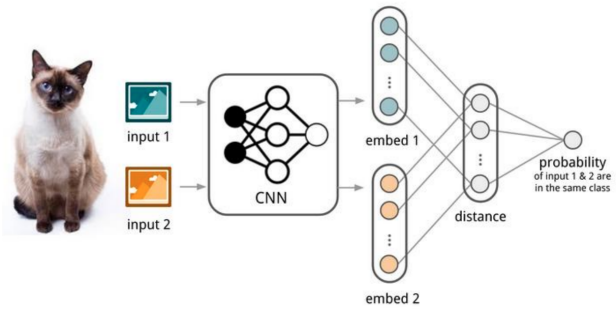
2.6.2 Metric-based Learning

Algorithms such as k-means clustering, DBSCAN, decision trees, kNN which are representative algorithms of data mining and machine learning operate based on distance functions, determining a **distance function** suitable for given data is critical in terms of their algorithm accuracy. However, among predefined distance functions, a distance function suitable for all data does not exist in reality. For this reason, **metric learning** directly creates a distance function suitable for data with a machine learning algorithm.
The purpose of the metric learning problem is to learn an embedding function that transforms the input data to be well distinguished according to each target value. In other words the objective is to **discriminate classes by computing distances in the embedding space**. Thus, given meta-training data, a collection of datasets:

- In the **Meta-Training stage**, the aim is to **develop a well-defined metric space**. This involves crafting a function that can effectively quantify the similarities and differences between data points, forming the foundation for comparison and classification.
- In **Meta-Testing** stage, this refined **metric space is employed** for image classification. The key here is to utilize the structural nuances and relational properties encapsulated by the metric space to accurately categorize new images. This process hinges on the metric's ability to translate raw data into insightful, actionable embeddings that inform the classification decisions.

Siamese Networks

Siamese networks consist in two **deep neural networks** that **share weights** and compute **embeddings**. In **meta-training** each network receives a **pair** of images as input, **calculate the distance between two embeddings**. The distance is then converted to a probability p by a linear feedforward layer and *sigmoid* and finally the output is binarized indicating whether the images belong to the same class or not. The loss is the cross-entropy calculated between pair of images. Once the network has been trained, it can be tested on new image pairs. In **meta-testing** the network processes images: **one from the test set** (query image) and **all the other from the training set** (support images). The network calculates the distance between the test image and each image in the support set to predict the class of the test image. The class assigned to the test image is that of the support image with which it has the smallest calculated distance, implying the highest similarity or the highest probability of being in the same class.
Siamese Networks assume that embeddings learned during the training phase can generalize to classes not seen during training. This implies that when new tasks are presented during the meta-test phase, the model can effectively utilize the distance metric learned without further updating the network. Note that during meta-training, the model employs binary classification while during meta-testing, the model performs n-way classification.



Matching Networks

Matching Networks are a type of neural network architecture designed for few-shot learning and they have the **same meta-train and meta-test conditions**, **k-way classification** during both. Unlike other approaches, Matching Networks do not attempt to learn a generalizable metric space or embeddings. Instead, they build a weighted classifier within the embedding space that is directly influenced by the support set.
Here's how a Matching Network works:

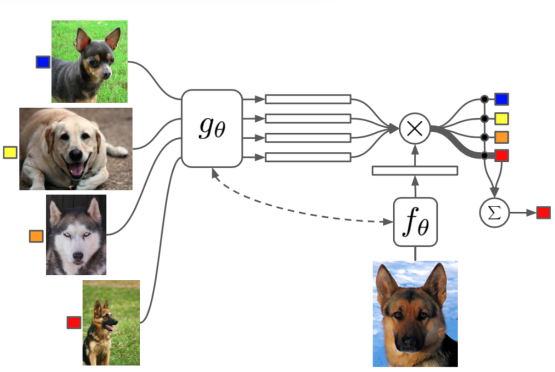
- **Embedding Functions**: Both the support set and the target set are passed through embedding functions g_θ and f_θ, which can be convolutional neural networks that convert input images into vector representations.
- **Attention Mechanism**: The network applies an attention mechanism over all of the support set. The attention is based on a comparison between the embedded representation of the single target example and the embeddings of the support examples. This is done through a softmax over **cosine similarities** in the embedding space.

$$a(\mathbf{x}, \mathbf{x_i}) = \frac{exp(cossim(f(\mathbf{x}), g(\mathbf{x_i})))}{\sum_j exp(cossim(f(\mathbf{x}), g(\mathbf{x_j})))}$$

- **Output**: The final output is a classifier that assigns to each target image the label of the support image to which it is most similar. This is based on a attention-weighted sum of the support set classes.

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

where $S = (\mathbf{x}_i, y_i)_{i=1}^k$ and k is the number of all classes.

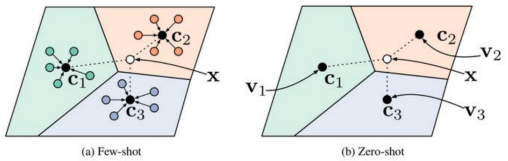


The embeddings can be context-dependent, meaning that the representation of an item from the support set can depend on the other items in the support set. This is sometimes achieved by using a bidirectional LSTM.
Matching networks offer the advantage of maintaining consistency in the task between meta-training and meta-testing phases. Additionally, they can leverage the relationships within the entire support set when utilizing full context embeddings.

Prototypical Networks

Prototypical Networks are another model within the family of metric-based meta learning. The key concept of Prototypical Networks is to represent each class by a **prototype**, which is essentially the **mean** of its examples in a learned metric space. Here's how they operate:

- **Embedding Function**: the network uses a neural network (often a CNN when dealing with images) to map inputs into a high-dimensional space. This function transforms both the support (train) and query (test) examples across various meta-training tasks. The **objective** is to minimize the cross-entropy loss, allowing the network to learn an embedding where **examples from the same class cluster around a common prototype**.
- **Prototypes**: for each class in the support set, the model computes the prototype, which is the **centroid** of the **embeddings** of the examples of that class. This means that for each class, you average the embedded points of all examples of that class to find its prototype.
- **Test**: once f_θ is trained to classify a new example (from the query set), the model calculates the distance (usually Euclidean) between the embedded query example and each of the class prototypes in the embedding space. It assigns a probability of that sample belonging to each class based on these distances. The new example is assigned to the class of the nearest prototype. The intuition is that if the embedding function is good, examples of the same class should cluster around their prototype, making this a good basis for classification.
- **Learning**: the network is trained by optimizing the distances between the examples and their corresponding class prototype. The goal is to minimize the distance of each example to its correct prototype while maximizing the distance to prototypes of other classes.



Prototypical Networks are effective because they simplify the classification problem in few-shot settings to finding the **nearest class prototype**, a much more manageable task when there are very few examples from which to learn. They're also attractive due to their simplicity and the intuition that classes can be represented by a single point that captures their "essence" in the learned metric space.

2.6.3 Optimization-Based Meta Learning

Optimization-Based Meta Learning refers to a subset of meta-learning methods that focus on optimizing the learning algorithm itself. The meta-learning algorithm ($\mathcal{A}_{META}$) takes multiple datasets ($D_1, D_2, ..., D_N$) and learns the best base learning algorithm ($\mathcal{A}^*$) along with its hyperparameters. It essentially learns how to learn.

$$\mathcal{A}^{META} : D_1, ..., D_N \rightarrow \mathcal{A}^*, \mathcal{A}^* : D_{N+1}^{train} \rightarrow \theta^{N+1}$$

The **learner** in this context is a model that can be **differentiated** and **optimized**, such as a deep Convolutional Neural Network (CNN). The **meta-learner**, on the other hand, is a learning algorithm that's **also differentiabl**e and parameterized. It is responsible for updating the learner's parameters in a way that's generalized across tasks.
This approach include:

- Learning an **optimizer** that can, for example, update the learner's weights in a more effective way than traditional optimizers like SGD (Stochastic Gradient Descent)
- Learning **hyperparameters**, such as the learning rate and schedule, which can adapt to different tasks more effectively than fixed hyperparameters.
- Learning a good **initialization**, which sets the starting parameters of the learner in such a way that they are well-suited for adaptation to a variety of tasks with minimal further training.

Model-Agnostic Meta Learning

It focuses to learn the **parameters optimal initialization** θ^*.

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{p(\mathcal{T})} \left[\mathcal{L}_{\mathcal{T}} \left(U_k(\theta, D_{\tau}^{\text{Supp}}), D_{\tau}^{\text{query}} \right) \right]$$

where $p(\mathcal{T})$ is the distribution over tasks, θ is the model's initialization, and U_k is a fast adaptation base learning algorithms (k steps) that is trained few-shot style on a (small) support set. It's a bilevel optimization problem: the **inner loop** (U) optimize on a new task starting from θ with algorithm U and the **outer loop** optimize initialization θ^* to improve generalization over the whole family of tasks.

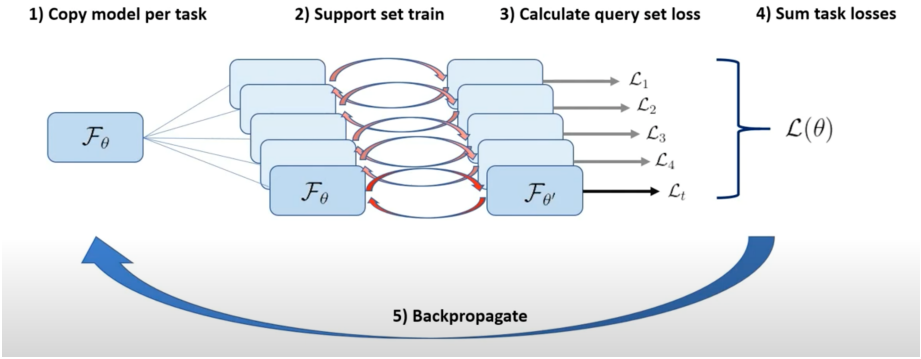
Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

The core idea is to find the best initial parameters θ^* for a model, which are optimized across a variety of tasks. It aims to find parameters from which the model can quickly adapt to new tasks with minimal additional training.
Meta-train:

- Inner loop: For each task, the model parameters θ are fine-tuned to produce a task-specific model. This is akin to short-term learning where the model quickly adapts to each specific task.
- Outer loop: The model parameters θ are then updated based on the performance across all tasks. This is a longer-term optimization that seeks to improve the model's ability to learn new tasks in the future. The update mechanism used in the outer loop should be differentiable, such as SGD, allowing for backpropagation through the updates.

The two-step optimization in MAML requires backpropagating through the inner optimization step to update the outer step parameters, which involves computing second-order derivatives (gradients of the gradients) and can be computationally intensive. **First-Order MAML** (FOMAML) offers a good approximation by omitting these second-order derivatives, significantly reducing the computational cost while still providing effective meta-learning performance.
After the model has been trained across various tasks (meta-training), it is then tested on new tasks (**meta-testing**). Here, the pre-trained and optimized parameters θ^* are used as a starting point, to quickly adapt to new tasks.



Batch Normalization is a technique that rescales neural network activations by normalizing the input to each layer for each mini-batch. This helps in stabilizing and speeding up the training process. In the training phase, Batch Normalization calculates the mean and variance of the activations within the current mini-batch. However, within MAML, each mini-batch typically contains samples from only a single task, which may lead to statistics that are not representative of the entire task distribution. During inference, MAML uses the Exponential Moving Average (EMA) of these mean and variance values observed during training. But since in MAML, the statistics are averaged over all tasks, the values during training (coming from a single task) might differ from those needed during inference across multiple tasks.
A key problem with Batch Normalization in multi-task and meta-learning settings is that it relies on the assumption that the samples are independently and identically distributed (i.i.d.), which may not hold true in these scenarios. This can be problematic, leading to less accurate models. As a result, MAML implementations might often forgo the typical training mode updates of Batch Normalization, and instead rely on other normalization strategies or disable Batch Normalization altogether.

To enhance the **stability** of MAML, it's recommended ("How to Train your MAML") to employ stable network architectures with skip connections, minimize loss at each step, start training with a first-order approximation before using full gradients, apply specific learning rates per step and layer, utilize cosine annealing for the learning rate, and adjust batch normalization statistics per step. These measures contribute to a smoother and more stable training process for MAML.

In optimization based meta learning we can have two possible goals: **rapid learning** (fast adaptation) or **feature reuse** (learn features that generalize across task). Sometimes, we don't need to adapt the whole network, perhaps only the final layer (partial adaptation).

Deep Continual Learning

3.1 Introduction to Deep Continual Learning

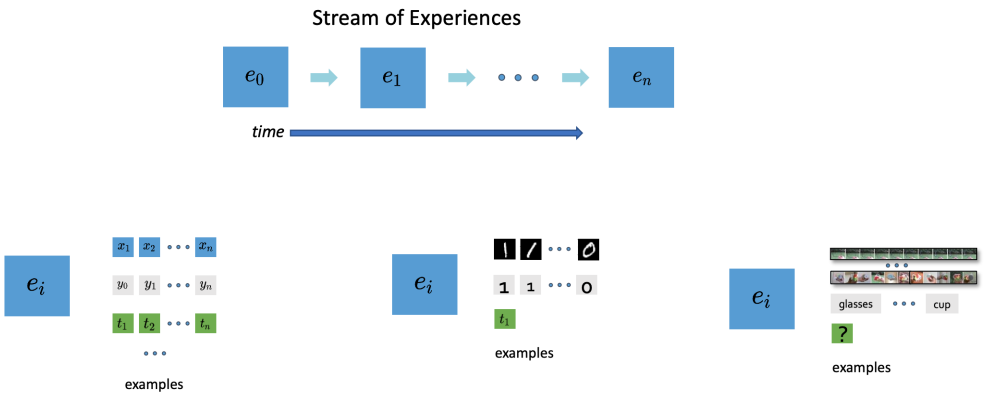
Challenges in offline learning include sustainability and efficiency concerns, especially in low-data scenarios. Privacy becomes a significant issue when data cannot be stored securely, and distributed learning across multiple devices poses additional challenges due to privacy and efficiency constraints. Deep Continual Learning is learning from a non-stationary stream with Deep Neural Networks.
The main goals of Continual Learning are:

- **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.

3.1.1 Catastrophic Forgetting

Catastrophic interference, a.k.a. catastrophic forgetting, is the tendency of deep neural networks to completely and abruptly forget previous tasks/domains when learning new ones. The CL loss consists of two components: $\mathcal{L}^{tot}(\theta) = \mathcal{L}^{old}(\theta) + \mathcal{L}^{new}(\theta)$. The first part comprises the loss on current data, which can be computed easily. However, the second part is the loss on previous data, which cannot be computed due to the unavailability of the data.
The **Stability-Plasticity Dilemma** is a fundamental challenge in continual learning that relates to the model's ability to acquire new knowledge (plasticity) while retaining previously learned information (stability). On one hand, the model needs to be plastic, or adaptable, to absorb and integrate new



Continual Learning examples

data as it becomes available; on the other hand, it also needs to be stable enough to preserve the old knowledge without it being overwritten or interfered with by the new information. If a model is too plastic, it can easily adapt to new tasks but may suffer from catastrophic forgetting, where learning new information leads to the loss of previously learned information. Conversely, if a model is too stable, it resists changing its parameters in light of new information, leading to an inability to learn new tasks efficiently. Finding a **balance** between these two aspects is key to developing effective continual learning systems. We can freeze the network to prevent forgetting or we can do a naive finetuning (or even randomly initialize) to have optimal plasticity.

3.2 Scenarios, Evaluation, and Metrics

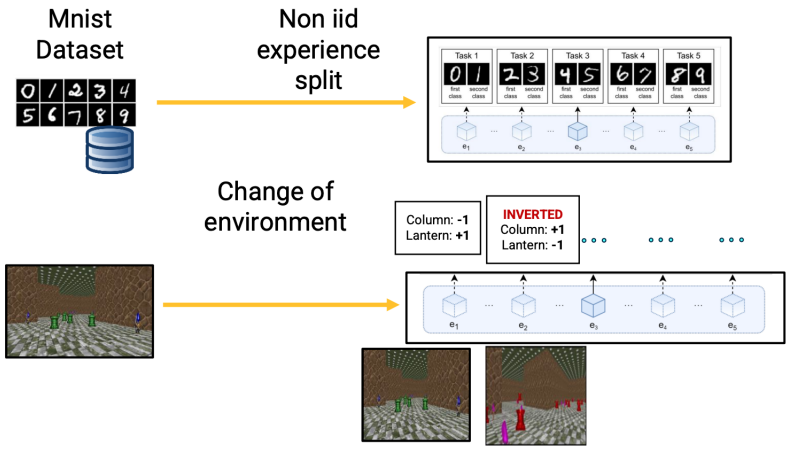
3.2.1 Nomenclature

A Continual Learning scenario is:

- A set of CL metrics that we want to optimize.
- A set of constraints that the learning algorithm must satisfy.
- A restricted form of access to the data through a sequential data stream.

Type of Shift

Under the non-stationarity assumptions we have two types of shifts in the context of machine learning and data: real shifts and virtual shifts.
Real shifts occur when there are actual changes in the world that impact the relationship between the input data x and the output y. This can happen suddenly (sharp), such as the societal changes during the COVID lockdowns, or gradually (blurry), as with changes in weather patterns or financial markets, leading to a change in the joint probability distribution $p(x, y)$. In some scenarios, these real shifts mean that models may need to forget past data because only current and future data are relevant.
On the other hand, **virtual shifts** do not involve changes in the real world but arise due to changes in the way data is sampled, leading to sample selection bias. In this case, the underlying real-world process does not change, meaning $p(y|x)$ remains constant, but the input distribution $p(x)$ shifts. Under these circumstances, it's important **not** to **forget** previous data because it may become relevant again in the future. The concept of virtual shifts aligns with a common assumption in continual learning (CL) where despite new data being encountered, the task relationships essentially stay the same.



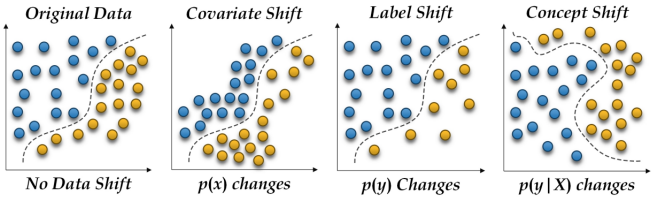
Real vs Virtual Shift

Dataset Shift

A **Dataset Shift** is, informally, any change in the distribution $p_{\text{tra}}(x, y) \neq p_{\text{tst}}(x, y)$.
A **Covariate shift** happens in $x \rightarrow y$ problems when $p_{\text{tra}}(y|x) = p_{\text{tst}}(y|x)$ and $p_{\text{tra}}(x) \neq p_{\text{tst}}(x)$. Informally the input distribution changes, the input-output relationship does not.
A **Prior probability shift** happen in $y \rightarrow x$ problems when $p_{\text{tra}}(x|y) = p_{\text{tst}}(x|y)$ and $p_{\text{tra}}(y) \neq p_{\text{tst}}(y)$. Informally output-input relationship is the same but the probability of each class is changed.
A **Concept shift** is:

- $p_{\text{tra}}(y|x) \neq p_{\text{tst}}(y|x)$ and $p_{\text{tra}}(x) = p_{\text{tst}}(x)$ in $x \rightarrow y$ problems.
- $p_{\text{tra}}(x|y) \neq p_{\text{tst}}(x|y)$ and $p_{\text{tra}}(y) = p_{\text{tst}}(y)$ in $y \rightarrow x$ problems.

Informally the "concept" (i.e. the class) changes.



Dataset Shifts

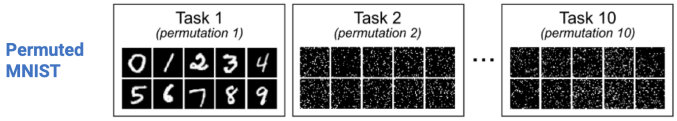
Common Characteristics

In CL, the common **assumptions** are:

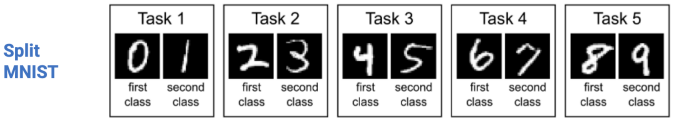
- **Shift is only virtual**: we do not want to forget, we need to accumulate knowledge.
- **No labeling errors**/conflicting information: targets are always correct (but possibly noisy).
- **Unbounded time**: No hard latency requirements. We may have computational constraints.
- **Data** in each experience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

The common types of **shifts** are:

- **New Instances**: each experience provides new instances for old classes. Old instances are never seen again (in the training stream). Example: permuted MNIST.



- **New Classes**: each experience provides new classes. Old classes are never revisited (in the training stream). Example Split MNITS.



Tasks

The CL setting can be categorized based on the **presence of task labels**. In task-aware the task labels are available during training and inference while in task-agnostic the task labels are not available. Task labels simplify the problem: we can use multi-task models that take task labels as an explicit argument.

Online vs Batch CL

Online Continual Learning (or Streaming Continual Learning) involves processing single examples or small minibatches sequentially. In contrast, **Batch** Continual Learning entails handling larger batches of data without constraints on the size of the experience. In a batch scenario the typical assumption is that each experience is the result of a distribution shift. In online settings knowledge of task boundaries is more useful (because the stream is much longer) but all the methods assume that they don't have access to it (more realistic).

Common Scenarios

A not exhaustive categorization of common scenarios is the following:

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

This table categorizes different continual learning scenarios by their characteristics:

- Task Labels: Indicates whether the learning scenario requires labels that identify the task to which the data belongs.
- Boundaries: Specifies whether there are clear distinctions or boundaries between tasks. A "Yes" implies that the model can tell when it is dealing with different tasks.
- Classes: Describes if the model is exposed to new classes as learning progresses or if it works with the same classes throughout. "New" indicates that new classes are introduced over time.
- Output: Refers to how the output space of the model is structured. "Shared" means the model outputs a common set of predictions for all tasks, while "Separate" implies different predictions for each task.

An alternative categorization is this one:

	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)	-	-	-

3.2.2 Evaluation

Average Stream Accuracy

The model is evaluated on the average accuracy on the entire (test) stream. It must remember how to classify data from old experiences. The objective of a CL algorithm is to minimize the loss $\mathcal{L}_S$ over the entire stream of data S:

$$\mathcal{L}_S(f_n^{CL}, n) = \frac{1}{\sum_{i=1}^{n} |D_{test}^i|} \sum_{i=1}^{n} \mathcal{L}_{exp}(f_n^{CL}, D_{test}^i)$$

$$\mathcal{L}_{exp}(f_n^{CL}, D_{test}^i) = \sum_{j=1}^{|D_{test}^i|} \mathcal{L}(f_n^{CL}(x_j^{(i)}), y_j^{(i)})$$

where the loss $\mathcal{L}(f_n^{CL}(x), y)$ is computed on a single sample (x, y), such as cross-entropy in classification problems.

Model Selection

Offline model selection involves training multiple models on a dedicated training data stream and evaluating their performance on a separate validation data stream. The best-performing model is then chosen based on its validation results for the final assessment on the test data stream. This approach assumes full access to each data stream and that all streams share the same distribution, although this can be an unrealistic assumption in real-world scenarios where data may not be fully available at once.
Early model selection is a process where a portion of the initial experiences, or data, is used to select a model before continuing training on subsequent data. Specifically, models are trained sequentially on the first part of the training data (up to a certain point k) and evaluated on a corresponding portion of the validation data. The model that performs best on this initial subset is then selected and further trained on the remainder of the training stream. While the selection is made offline, it occurs early in the training process, allowing for quicker adaptation and potentially less computational resource expenditure.

Continual Hyperparameter Selection

Continual hyperparameter selection involves **tweaking** the **learning configuration** to achieve two main goals: plasticity, which is the model's ability to learn and perform well on new data, and stability, which is the capacity to retain performance on previously learned data, thus minimizing forgetting. This optimization occurs without access to old data to evaluate stability, and often using a current validation set to measure plasticity. In scenarios where only two hyperparameters are available, one is adjusted to enhance plasticity (e.g. learning rate), and the other to maintain stability (e.g. regularization strength), striking a balance between acquiring new knowledge and preserving existing one.
For each new learning experience, optimal hyperparameters for plasticity are first determined to maximize current accuracy, potentially sacrificing stability. These parameters are then fixed, and the process moves to optimizing stability hyperparameters, beginning from a point of maximum stability. The stability parameters are gradually relaxed until a satisfactory balance between accuracy and stability is achieved. This constitutes the stability-plasticity tradeoff: stopping adjustments too early may lead to low adaptability to new data (low plasticity), whereas adjusting too far may result in excessive forgetting of previously learned information.

3.2.3 Deep Continual Learning vs Online Machine Learning

	Deep CL	**Online ML**
Batch size	(possibly) Large experiences	One sample at a time
Drift	Virtual	Real
Domains	Vision, NLP, speech	Time series data
Evaluation	Average accuracy on the full stream	Prequential accuracy

3.2.4 Metrics

In a CL setting the elements to monitor are: performance across current, past, and future experiences, resource consumption, model size growth, execution time, latency, and data efficiency, etc.

Accuracy

Let N be the stream length, T the current timestep, $R_{t,i}$ be the accuracy on the experience i at time t. In the context of model evaluation across a data stream, there are different strategies based on the **time** of the model and the data used. For the time dimension, one can either use the most recent model, $R_{T,T}$, or an average of the model's performance over time, $\frac{1}{T+1}\sum_{t=0}^{T} R_{t,i}$. Regarding the **data**,

the options are to evaluate based on the current experience $R_{t,t}$, an average over the data seen up to the current time $\frac{1}{T+1}\sum_{i=0}^{T} R_{T,i}$, or using the entire stream $\frac{1}{N}\sum_{i=0}^{N-1} R_{T,i}$. These approaches reflect different ways to summarize the model's performance over its learning lifecycle.

Forward Transfer Metric

The FWT metric measures if continual learning is improving the performance on **future experiences**.

$$FWT = \frac{1}{T-1}\sum_{i=2}^{T} R_{i-1,i} - \hat{b}_i \tag{3.1}$$

FWT compares the accuracy on future experience $i+k$ after training on experience i against the accuracy on experience i of a model trained with a random initialization ($\hat{b}_i$) averaged over time $i = 2, ..., T$. FWT assumes that the model can predict future experiences but most models can't predict unseen classes, it only makes sense for **new instances**, not new classes.
An alternative solution can be evaluating whether the latent representation helps learning unseen tasks via **Linear Probing** (updating only the last linear layer). A linear classifier is trained on top of the learned representation obtained from a pre-trained model. This technique involves comparing the performance of the linear classifier against a baseline model using random feature extraction and any previously trained models. This measures whether the learned features transfer to the new data

Backward Transfer Metric

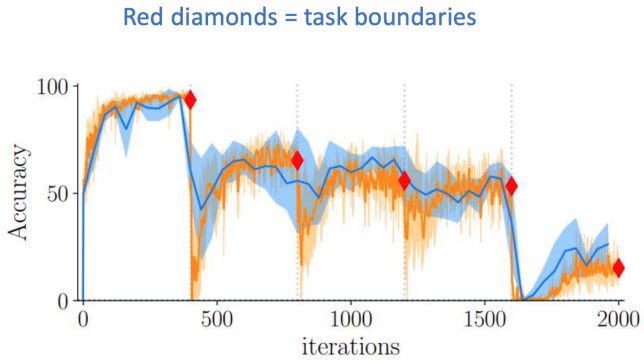
The BTM metric measures if continual learning is improving the performance on **old experiences**.

$$BWT = \frac{1}{T-1}\sum_{i=1}^{T-1} R_{T,i} - R_{i,i} \tag{3.2}$$

BWT compares the accuracy on experience i after training on experience T against accuracy on experience i after training on experience i, averaged over $i = 1, ..., T-1$. The forgetting can be interpreted as $-$BWT.

Desiderata

In online continual learning, the goals present unique challenges due to the ongoing and undefined nature of task boundaries within a data stream.



A CL goal is **Knowledge Accumulation**: the model should consistently improve, retaining a high level of accuracy over time, with the agility to quickly adapt to new information. A metric used for this purpose is the Average Anytime Accuracy, the accuracy along the entire curve. Do not confuse with the average accuracy at the end of training (final diamond) or the average at task boundaries (avg of diamonds).
Another goal is **Continual Stability**. It's essential for the model to maintain previously acquired knowledge without significant forgetting, assuming virtual drifts in the data distribution. To effectively measure continual stability, one should observe the model's accuracy trajectory during training,

specifically noting how it fluctuates between the established task boundaries (marked by diamonds in graphs). CL methods forget and re-learn old experiences during training but this phenomenon is masked when using typical metrics that measures only at boundaries (red diamonds).
An objective of CL is the **Representation Quality**. The model's latent representations should become progressively refined, facilitating better generalization and allowing for performance evaluation on data distributions that differ from those seen during training or through self-supervised methods.

Memory Usage

Model Size measures how much space the model occupy (MB, number of params, etc.). **Scalability over time** monitors the increment in space required for each new experience. And **Samples Storage Size** tells you ow much space do you require for additional information (replay buffer, past models, etc.).
The computational overhead during training and inference phases are measured with **Multiply and ACcumulate metric** (MAC). MAC which is a common operation in neural networks, representing the core computation in convolutional and fully connected layers. The running time on CPU or GPU platforms is also a critical consideration, which can affect the practical deployment of models. A crucial note is that evaluating GPU performance is complex and potentially misleading if only considering MAC operations. It requires considering the entire system, including data transfer between the CPU and GPU, synchronization overhead, and how effectively the computations are parallelized across the GPU cores.
Each application is different and multiple objectives may interfere with each other: computational constraints, privacy constraints, accuracy and forgetting. In general, multiple objectives...

3.2.5 Forgetting in CL Scenarios

Task-Incremental vs Class-Incremental

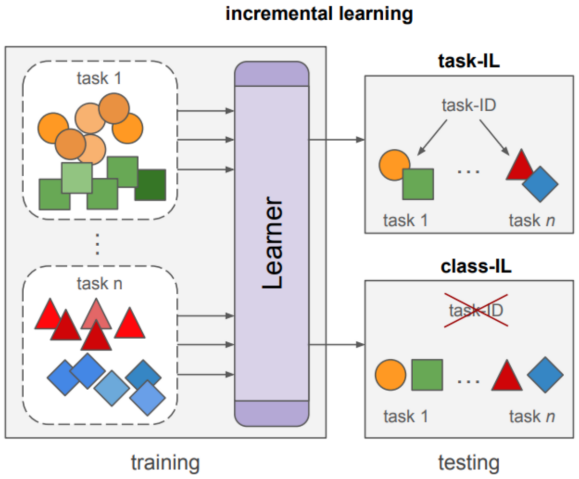
The terms Task-Incremental and Class-Incremental refer to different paradigms within the field of incremental learning, which is a subset of machine learning where the model is exposed to a continuous stream of data over time and must learn from this data without forgetting previous knowledge.
In **task-incremental learning**, the model learns a new task with each set of data it is exposed to. Here are some key points:

- The model knows when a new task begins.
- It learns to perform different tasks that are distinct from each other, with task boundaries being clear.
- The model is typically allowed to use task identifiers at inference time, allowing it to apply the specific knowledge it has learned for that task.
- The aim is not to mix the knowledge across tasks but rather to maintain separate expertise in each task.

Class-incremental learning focuses on learning new classes within the same task over time. Here are the key aspects:

- There is no explicit indication of when new classes will be introduced.
- The model must learn to differentiate between all classes seen so far, regardless of when they were introduced.
- There are no separate task identifiers used during inference; the model must decide from a single, growing pool of classes.
- The challenge here is to prevent "catastrophic forgetting" of old classes when new ones are introduced.

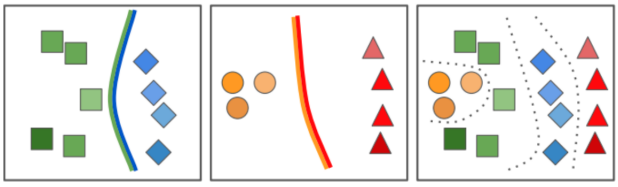
The primary difference between the two is that task-incremental learning maintains clear task boundaries and knowledge separation, while class-incremental learning continually integrates new classes into a single, unified model. Task-incremental learning often deals with completely new tasks with their own data distribution, while class-incremental learning adds new categories or labels to an existing task, which can make it more challenging to maintain old knowledge due to overlapping data distributions.



In incremental learning, disjoint tasks are learned sequentially. Task-IL has access to the task-ID during evaluation, while the more challenging setting of class-IL does not.

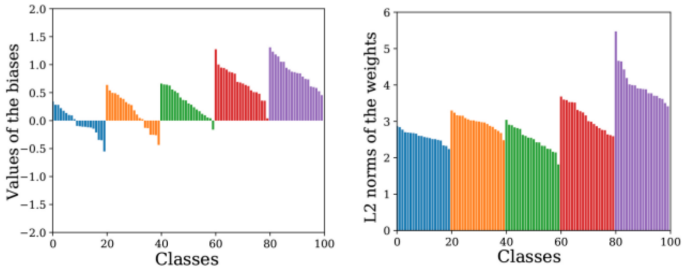
Issues

Inter-task confusion occurs when a model, trained in a continual learning scenario, fails to develop features that effectively distinguish between classes. This typically happens when the focus is more on differentiating tasks than the classes within them.



A network trained continually to discriminate between task 1 (left) and task 2 (middle) is unlikely to have learned features to discriminate between the four classes (right). We call this problem *inter-task confusion*.

Task recency bias describes the tendency of a model to have a stronger weighting, in terms of the magnitude of the weights, towards the most recently learned tasks. This can lead to the model performing better on these latest tasks while potentially forgetting previous ones.



Self-Supervised Models

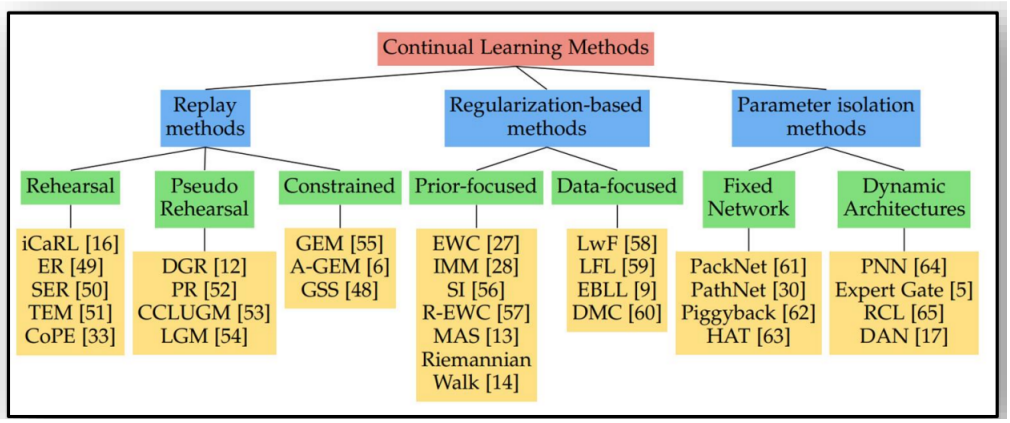
Self-Supervised Learning (SSL) methods are known for their robustness during continual training. The quality of the learned representations can be assessed using linear probing, where a linear classifier is trained on top of the frozen features from the SSL model. While SSL reduces the reliance on labeled data during training, labels are still necessary for fine-tuning in tasks like classification. However, SSL approaches typically have longer training times and demand extensive data to perform well.
Continual Pretraining in this context is pre-training the SSL incrementally over time. Then the last model can be used for Finetuning on the downstream tasks.

3.3 Continual Learning Strategies

A CL strategy is a learning method designed for Continual Learning. Typically a combination of naive finetuning plus some CL specific component. It can be categorized as:

- Replay: store sample and revisit them.
- Regularization: penalize forgetting.
- Parameter-Isolation/Architectural: separate task-specific parameters.

CL methods can be combined together, e.g. regularization + replay + architectural + bias correction (methods for output layer).



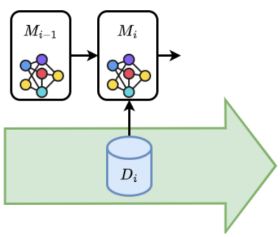
3.4 Baselines

In continual learning, a baseline refers to a standard or reference point used to **evaluate and compare the performance** of different algorithms or approaches. The baseline can be an existing, well-established method or a simple technique that serves as a point of comparison.

3.4.1 Baselines Types

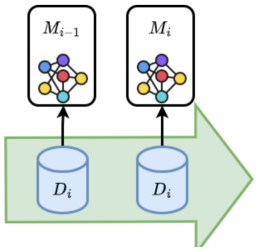
Naive Finetuning

During training, sequential Stochastic Gradient Descent (SGD) is employed, utilizing **only the current data** at each iteration. For inference, the last trained model is utilized. It's important to note that naive fine-tuning approaches often **lead to catastrophic forgetting**. Continual Learning (CL) methods should consistently outperform this naive baseline.



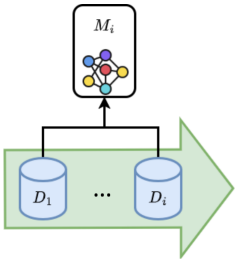
Ensemble

An ensemble-based approach can be used as baseline for continual learning. A **separate model** is trained **for each distinct experience**, with no overlap or shared learning between models. During inference, the model corresponding to the appropriate task or experience is used to make predictions. If task labels are not available, an oracle (a perfect predictor) is assumed to select the correct model. If each experience dataset is sufficiently large, this method could be tough to beat, even without task-specific labels.



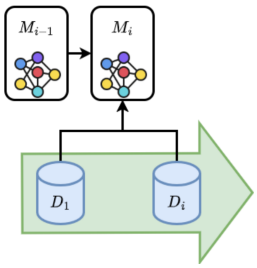
Joint Training

Joint Training, also known as Offline Training, is a baseline approach where all data across tasks is pooled together and the model is trained from scratch with a random initialization. This approach keeps task labels for the training process. Although sometimes incorrectly referred to as the upper bound of performance, Joint Training is a straightforward method that **does not account** for the incremental or **sequential nature of data** acquisition found in continual learning scenarios.



Cumulative

In the Cumulative baseline approach the model is re-trained with all available data at each new experience, using the previous version of the model as the starting point. This approach leverages the complete historical dataset for training, potentially achieving better performance if sufficient data is available, compared to training from scratch each time. It also usually results in faster convergence due to the model initializing from an already partially trained state.



3.4.2 Basic Design Choices

For the lower bound, naive finetuning can be used. As for the upper bound ensemble methods, joint and cumulative approaches can be considered.
Pretraining is consistently beneficial when accessible. Research indicates that the initial stages of training are pivotal. If the model is exposed to a limited set of closely correlated samples during the early epochs, it may struggle to achieve optimal performance later on.
Regarding the model architecture different choice can be made: CNN vs transformers, batch normalization, regularization (dropout), wide vs deep networks, etc.

3.5 Replay

3.5.1 Overview

In a typical **Class-Incremental** Learning scenario, a key challenge lies in addressing **virtual drift**. This occurs when new classes are continually introduced with each new experience, without any data repetition. This is a difficult problem, especially if there is no opportunity to revisit previous data. One strategy to mitigate this issue is **Replay**, which involves **storing a limited subset of samples from previous experiences** in a **reservoir** and using these samples for rehearsal. This process helps in retaining knowledge of the old classes by rehearsing them while learning new ones.

Replay brings **good news**: it's a straightforward, versatile, and effective strategy CL. It approximates an i.i.d. distribution and offers an approximation of cumulative training. Moreover, it's relatively inexpensive in terms of computations. However, there's also **bad news** to consider. Memory limitations or privacy constraints may pose challenges, while scaling can become an issue for long streams, requiring the storage of increasingly large buffers as memory requirements escalate over time.

Replay Algorithm

The Replay Algorithm operates with a set parameter known as memory size. During the training phase, which includes both finetuning and rehearsal, the algorithm follows these steps: first, it samples from the current data. Then, it samples from the buffer utilizing a specific sampling policy. Next, it performs a Stochastic Gradient Descent (SGD) **step on the concatenated mini-batch obtained from the current data and the buffer samples**. After each experience, there's a **buffer update phase**. Here, the algorithm employs an insertion policy to select data from the current experience. It then adds these selected examples to the buffer. Additionally, if the buffer size exceeds the predefined limit, a removal policy is applied to ensure the buffer does not become too large.

Replay Choices

Choose the **memory size.** In Growing Memory, each experience contributes k examples, leading to unbounded growth. On the other hand, Fixed Memory operates with a set maximum memory size k and necessitates the implementation of a removal policy to manage memory constraints.
Choose the **insert, remove, sample policies**. Insertion is identifying the most suitable examples for storage in the buffer. Removal involves identifying the least useful examples to remove from the buffer. Sampling is selecting the optimal examples to utilize for the current training iteration. Ideally, data should be balanced (approximated i.i.d.), common strategies include class/task/experience balancing. Replay data can be stored in different formats: Input Replay entails storing input images, while Latent Replay involves storing latent representations at intermediate layers.

3.5.2 Replay Techniques

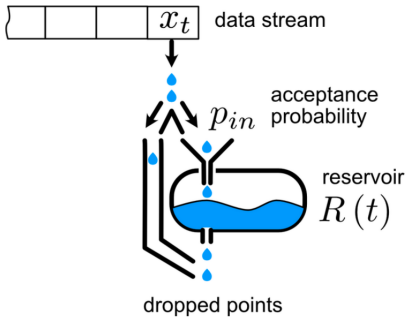
Random Replay

Random Replay is a basic form of replay, characterized by **random** insertion, deletion, and sampling of data. It operates with a parameter called memory size. During training, the algorithm goes thru the concatenated data (current batch + the memory) and performs a SGD step. After each experience, it randomly samples from the current experience data and populates a fixed Random Memory (RM). The number of examples inserted or removed is inversely proportional to the length of the data stream. In many cases, the experience data and the buffer may exhibit significant differences in size. Instead of simply concatenating the data, a more effective approach involves separate sampling from both sources. **Enhanced Sampling** involves randomly sampling from the current data and the buffer independently. Then, a SGD step is performed using the concatenated mini-batches obtained from these separate samples.
Random replay is easy to implement and effective in basic scenarios, but it does have its **limitations**. It struggles with determining the appropriate amount of data to keep, making it unsuitable for Online Continual Learning (OCL) where discrete sample selection is necessary. Additionally, it fails to address scenarios where the number of experiences vastly exceeds the memory size. It also overlooks potential imbalances in the data stream, allowing overrepresented classes to dominate the buffer. While it doesn't require task boundaries or labels, which can be advantageous, it still falls short in OCL scenarios. An enhancement could involve segmenting the buffer by task in a balanced manner, utilizing task labels for improved performance.

Reservoir Sampling

Reservoir Sampling is a technique used for randomly sampling K items from a stream of unknown size S in a way that each item has an **equal probability of being selected**.
The process is as follows: initially, the first K items of the stream are stored directly in the reservoir. For each subsequent item x_t in the stream (from the $(K+1)^{th}$ item to the N^{th}, where N is the length of

the stream at that time), it may replace an item in the reservoir. The probability of x_t being included in the reservoir is K/N, ensuring that at any time N, each item in the stream up to that point has an equal chance of being in the reservoir. If x_t is chosen to be included, it replaces a randomly selected item in the reservoir.



The data stream concept and how a reservoir sampling method is used to select certain data points for the replay. The dropped points represent data that are not included in the reservoir, while p_{in} indicates the probability of a data point being accepted into the reservoir.

This method ensures that the reservoir **represents a random sample of the stream data without having to store the entire stream**. Reservoir sampling offers improvements over Random Replay by addressing certain challenges:

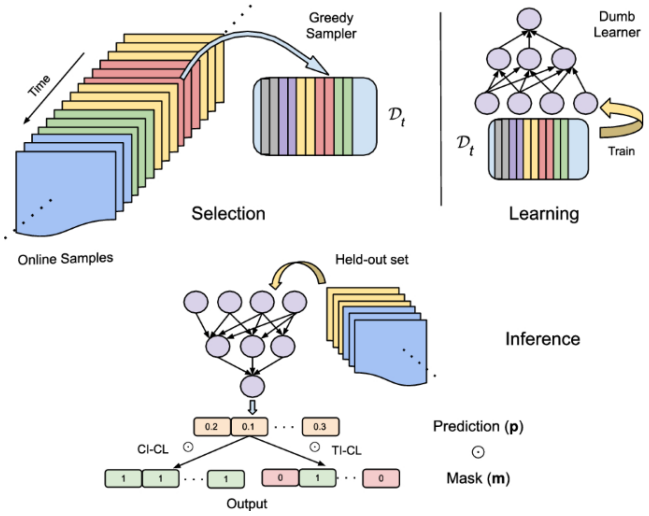
- Suitable for OCL: performs well in determining the amount of data, making it suitable for OCL scenarios.
- Imbalance resolution: it disregards stream imbalances, potentially leading to overrepresentation of frequent classes in the buffer. But this issue can be resolved by allocating buffer capacity per class, ensuring balance across reservoirs.
- Task boundaries: while it can address class imbalances, detecting imbalances at other levels (e.g. domain) would require task labels, which may not always be available. Reservoir sampling doesn't rely on task boundaries, making it a robust OCL baseline.

Maintaining a well-covered buffer of past data distribution is essential for continual learning systems. **Diversity** within the buffer is key to **mitigating forgetting** and preventing overfitting. Augmentations serve as crucial implementation details, especially in online scenarios with either multiple passes or constrained buffer capacities. These strategies collectively contribute to the robustness and effectiveness of continual learning models.

GDumb

GDumb is a dumb but popular replay baseline composed by three parts:

- **Greedy Sampler**: the sampler greedily stores samples while balancing the classes.
- **Dumb Learner**: before inference, the learner trains a network from scratch on memory D_t provided by the sampler.
- **Masking**: if a mask m is given at inference, GDumb classifies on the subset of labels provided by the mask; class-incremental mask only unseen classes, task-incremental mask classes outside of current task.



Gdumb adopts a strategy of training the model from scratch at each step, resulting in zero knowledge transfer as the network is consistently trained from scratch on buffer data. Despite its straightforward approach, it proves to be highly competitive in the class-incremental setting. However, its primary limitation lies in its lack of consideration for the constraints typically encountered in online machine learning (OML) methods.
Gdumb has several **limitations**. It lacks the ability to transfer knowledge between tasks or experiences. Training occurs solely on a subset of samples equal to the buffer size. Training the DNN from scratch before inference is computationally expensive, particularly for continuous evaluations. Despite these drawbacks, Gdumb serves as a straightforward **baseline** for **replay methods**.

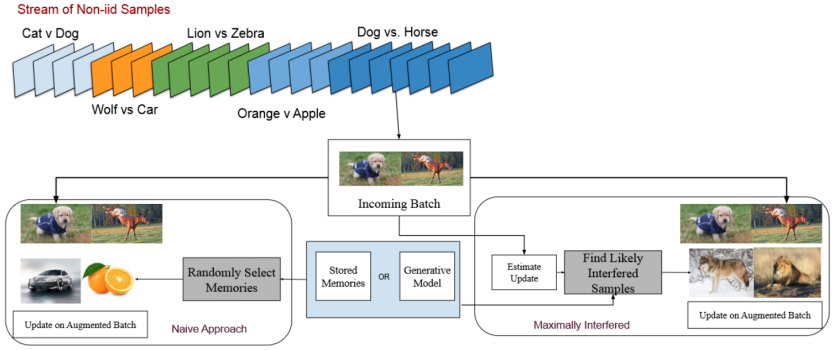
Maximally Interfered Retrieval

We would like to have a sample selection strategy that involves determining the most suitable examples from the buffer. Typically, sampling is random, often balanced across classes. However, there's room for improvement in this process.
Maximally Interfered Retrieval (MIR) idea is to **select examples** that are more **negatively impacted by the weights update**. The high-level algorithm involves the following steps:

1. Sampling from the current data.
2. Estimating the weight update based on the sampled data.
3. Estimating the loss of buffer samples using the new weights.
4. Selecting samples with the largest drop in loss: find the likely interfered samples.

$$s_{MI-1} = loss(f_{\theta^{new}}(x), y) - loss(f_{\theta^{old}}(x), y) \tag{3.3}$$

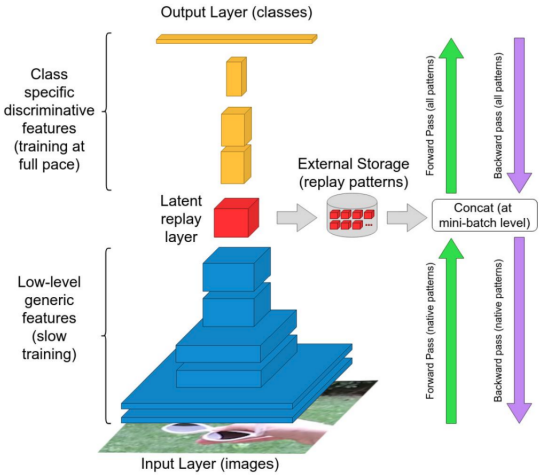
5. Performing an SGD step on the concatenated minibatch, which includes both the current data samples and the selected buffer samples.



MIR has some **limitations**: the method is computationally expensive compared to the actual improvement in accuracy gained over random selection. It requires an extra forward pass on a significant subset of the buffer examples to identify the most influential ones.

Latent Replay

Traditional replay techniques store **raw input data**, which can be **inefficient** and not analogous to biological learning. The proposed solution is to **store latent activations** (the intermediate representations from deeper within the model) which offers a **better balance** between accuracy, memory usage, and computational demands. The challenge lies in choosing the optimal layer from which to replay these activations since middle layers of neural networks can be very large. However, if the model allows for lossy compression, these activations can be reduced significantly in size.
The Latent Replay **algorithm** works as follows. Store the latent activations of past data. When new data arrives, process it through the network up to the layer where activations are stored. Concatenate these new activations with the stored ones. Continue the forward pass from the latent replay layer to the output layer to make predictions.



Architectural diagram of latent replay. The diagram illustrates a model architecture where specific layers are designated for latent replay and external storage, indicating the flow of data during the forward and backward passes. This allows the model to "rehearse" previous experiences and mitigate forgetting while learning new information.

Lower layers are typically trained in the initial stages of training and remain relatively stable thereafter. Freezing them at a certain point can be beneficial. This practice enhances latent replay, as failing to freeze the latent representation could result in it becoming outdated over time.

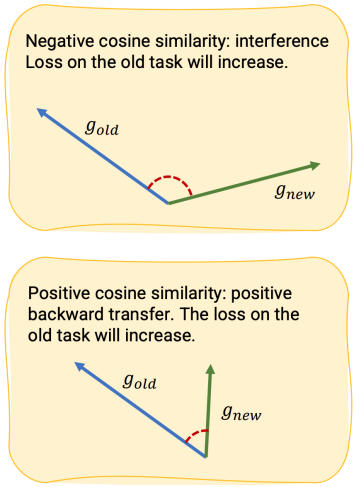
Generative Replay

The buffer size is often constrained by **memory limitations**. Utilizing **generative models** allows us to **store** a **finite number of parameters** while being able to **generate an unlimited number of examples**. This approach aligns with biological plausibility. Currently, there isn't an efficient algorithm for training generative models continuously. One approach is to train a separate model for each task or experience, which comes with a linear increase in memory usage. Another option is to employ Knowledge Distillation, where generative samples are used. For classification losses, the examples should closely resemble the target class, while methods like knowledge distillation might allow for the use of more distorted samples.

3.5.3 Constraints

Constraints methods find a mathematical definition of **forgetting as an optimization constraint**. Calculating interference explicitly with a mathematical definition. **Interference** refers to the **negative**

impact that **learning new information** can have on the **retention of old information**. To combat interference, an approach is to adjust the learning steps within the training algorithm. The idea is to modify the way the model learns in order to minimize the interference and thus prevent the "forgetting" of previously learned tasks. Interference is measured using a gradient-based definition that relies on **cosine similarit**y. Gradients are the directions in which the model's parameters should be adjusted to minimize the loss for a given task.



So the method involves computing gradient for the old and new task in order to compute cosine similaritiy between the gradients. If the **cosine similarity** between the **old task gradient** (g_{old}) and the **new task gradient** (g_{new}) is negative, this implies that the updates for the new task are in opposition to the updates for the old task. As a result, learning the new task may increase the loss on the old task due to interference. Conversely, a positive cosine similarity suggests that the updates for the new task are in the same direction as for the old task. This could lead to a positive backward transfer, meaning the learning of the new task could actually improve performance on the old task (potentially loss decrease). If the cosine similarity is 0 means that the task are orthogonal (without interference).
In estimating gradient interference we are in the setting where tasks are introduced sequentially with clear boundaries. The model employs **episodic memory** to maintain a balanced mix of data from past tasks. This allows for the reactivation of old knowledge by recalculating gradients based on this stored information.
Given the interference as the cosine similarity between gradients of the new task and old tasks we want to **remove** the **component of the new gradient** that **interferes** with previous tasks. To address this, a solution is to adjust the new gradient by projecting it such that it minimizes interference with the previous tasks. This is done by **ensuring** that the **cosine similarity constraint is satisfied** for all tasks, effectively reducing the negative impact of new learning on old knowledge. This optimization problem is solved using quadratic programming, a type of mathematical optimization that can be handled by established algorithms. The goal of this approach is to keep the useful components of the new gradient while discarding aspects that would negatively interfere with what the model has already learned.

Gradient Episodic Memory

Gradient Episodic Memory (GEM) model includes an episodic memory that stores a subset of the observed examples from task to mitigate the problem of catastrophic forgetting.
Here's how GEM works:

1. **Episodic memory update**: at every training iteration, GEM updates an episodic memory that stores a subset of the data from previously encountered tasks. This memory acts as a reference for the knowledge the model has acquired in the past.

2. **Sampling from memory**: it randomly samples a batch of data from the episodic memory.

3. **Gradient computation for old and new tasks**: GEM calculates the gradients of the model's parameters with respect to the loss on the old tasks using the sampled data. It also computes the gradient for the new task using the current data.

4. **Gradient projection**: GEM then projects the gradient for the new task onto the space that does not interfere with the old tasks. This means adjusting the gradient for the new task so that, if followed, it will not increase the loss on the old tasks.

5. **Model update**: finally, using the projected gradient, GEM updates the model. This way, the model learns the new task while retaining its performance on the previous tasks.

In essence, GEM tries to find a balance between acquiring new knowledge and retaining old knowledge by ensuring that the update for the new knowledge does not negatively impact what has been learned before. It leverages episodic memory to reference previous tasks and uses gradient projection to navigate the trade-off between retaining old knowledge and accommodating new information.
The **pros** are: we have a constraint formulation of forgetting and assuming the gradient estimate is correct (we only use few samples to estimate it) we can completely remove interference. While the **cons** are: the algorithm is slow due to QP problem at every iteration, it requires samples and the QP problem may not have a solution.
Interference can be seen as a **meta-objective** in continual learning strategies. Methods like GEM optimize it directly, while others aim to implicitly reduce gradient interference. Meta continual learning approaches either minimize task interference explicitly through loss functions or implicitly via MAML-like loops. Online approximations of GEM constraints provide alternative solutions in this domain.

3.5.4 Still Challenges

Do replay techniques fully address continual learning **challenges**? **Not quite**.

- Disparity with offline training remains significant.
- Accuracy gains relative to memory size often follow a logarithmic trend.
- Large buffer sizes, approaching a cumulative strategy, come with high costs. For instance, storing 50 images per class from ImageNet requires about 7 GB of memory.
- Additional computations are needed for repeated passes over the same examples.
- Balancing over tasks can quickly escalate to a linear cost over time.

A lot of exampes are needed to recover the joint training performance.

In real-world scenarios, **class repetitions** can **naturally** occur within the data stream (Natural Repetitions). Even in the absence of a dedicated replay buffer, rehearsal occurs organically. However, this form of rehearsal may be less effective due to potential issues such as missing classes, unbalanced distributions, or biases. One approach to approximate this natural repetition is by generating a more representative stream with repeated occurrences.

3.6 Regularization - Prior-focused

3.6.1 Proxy Losses

The Continual Learning objective is to minimize a loss given from the new data and old data.

$$\mathcal{L}^{tot}(\theta) = \mathcal{L}^{old}(\theta) + \mathcal{L}^{new}(\theta) \tag{3.4}$$

We estimate $\mathcal{L}^{new}(\theta)$ easily from the new data. Since we don't have access to the old data, we need to approximate $\mathcal{L}^{old}(\theta)$.
There's two distinct approaches to updating a machine learning model as new data arrives over time: the Bayesian approach and the optimization approach.
In the **Bayesian approach**, the model update is framed as a Bayesian inference problem. The current model parameters θ_t are updated using the posterior distribution $p_t(\theta)$ from the previous experience

as the new prior. Then, a new posterior $p_{t+1}(\theta)$ is computed as new data becomes available. However, this method involves approximating the posterior, which can introduce errors over time.
On the other hand, the **optimization approach** seeks to find an approximation $\hat{\mathcal{L}}^{old}(\theta)$ of the previous loss function $\mathcal{L}^{old}(\theta)$ and combines it with the new loss $\mathcal{L}^{new}(\theta)$ to optimize the model. This approximation uses local information about the loss at the current solution θ_t, which represents a minimum in relation to the previous experiences. This method allows the model to incorporate new information while trying to maintain performance on previously learned data.

Regularization as Importance

An approach to **approximate the loss** for previously learned data without having to store that data approach involves using a **Taylor series expansion**. It approximates the old loss function, $\mathcal{L}^{old}$, around the parameters θ_t from the previous learning experience.

$$\mathcal{L}^{old}(\theta) \approx \mathcal{L}^{old}(\theta_t) + \nabla\mathcal{L}^{old}(\theta_t)^T(\theta - \theta_t) + \frac{1}{2}(\theta - \theta_t)^T Hf(\theta_t)(\theta - \theta_t) \tag{3.5}$$

Where θ_t represents the parameters from the previously trained model and $\mathcal{L}^{old}(\theta_t)$ is a constant that we can ignore.
This allows for the construction of a **surrogate loss function**, which captures the **curvature of the original loss landscape** around the previous solution without needing the original data. The loss function for the old experiences is thus represented as a quadratic function, which can be optimized using current data. This method aids in retaining knowledge from old tasks when new data is being introduced to the model, addressing the challenge of catastrophic forgetting.
In the optimization process, we're looking to **minimize** the surrogate loss, which approximates the loss for previous experiences without needing to revisit the old data. Since θ_t is a point where the previous loss was minimized, we assume that the gradient of the loss with respect to θ_t is approximately zero ($\nabla\mathcal{L}^{old}(\theta_t) \approx 0$). This makes the linear term of the Taylor expansion negligible. The **quadratic term** is the only one kept and involves the Hessian $Hf(\theta_t)$ of the loss function. This is the second-order term that provides information about the **curvature** of the loss function. This term adjusts the linear approximation by considering how the **function bends**. However, to simplify the computation, a diagonal approximation of the Hessian is used. The resulting loss is a summation of individual quadratic terms for each parameter: $\mathcal{L}^{old}(\theta) \approx \sum_i h_i(\theta^i - \theta_t^i)^2$.
This can be seen as a **regularization technique**, which helps a model preserve the knowledge of previously learned tasks while learning new ones. The model uses a weighted quadratic loss $\sum_i h_i(\theta^i - \theta_t^i)^2$, where the **weights represent the importance of the parameters** in keeping old knowledge. This technique ensures that parameters crucial for past learning are not drastically altered. The regularization is **centered around** the **parameter** values from the **previous solution**, using a non-uniform distribution of importance across parameters. The coefficients h_i are the curvature of the loss around θ_t^i, informally, we can think of h_i as the **importance** of parameter i. Keep in mind that this is only a **local approximation**.
DNNs are characterized by overparameterization, where for each task, only a few parameters are crucial (high h_i). The loss function penalizes alterations in these important parameters (high curvature), while allowing the unimportant ones to adjust freely to accommodate the new task.

Regularization as Distance between Solutions

A distance measure of the distance between the old and new solution can serve as regularization. In this case what is a good distance measure?

$$\mathcal{L}(\theta) = \mathcal{L}^{new}(\theta) + D(\theta, \theta^{old}) \tag{3.6}$$

Bayesian Sequential Update

Updating a model's knowledge base (its posterior) in a Bayesian framework during continual learning is challenging. While it's impractical to compute the true posterior with deep neural networks (DNNs), we can attempt a rough approximation. Such an approximation, although it will lead to some degree of forgetting, might be the best available strategy. If we have an approximate posterior, we can employ

a sequential Bayesian update, where the new posterior is computed by combining the likelihood of the new data given the current model, the prior (which is the posterior from the previous update), and normalizing by the evidence of the new data. This update is shown as a logarithmic equation to facilitate computation.

$$\log p(\theta|D) = \log p(D_{\text{new}}|\theta) + \log p(\theta|D_{old}) - \log p(D_{new}) \tag{3.7}$$

Here, $p(\theta|D)$ represents the posterior probability of the parameters θ given the data D, $p(D_{\text{new}}|\theta)$ is the likelihood of the new data given the parameters, $p(\theta|D_{\text{old}})$ is the prior probability of the parameters given the old data, and $p(D_{new})$ is the evidence or the probability of observing the new data irrespective of the parameters.

Recap

Our aim is to accurately **approximate the loss for past data**, which necessitates several key pieces of information. These include: understanding the curvature of the loss function and identifying an appropriate minimum, establishing a metric to measure the distance between solutions, determining the importance of the model's parameters, and obtaining the posterior distribution of the weights. However, the challenge lies in finding effective approximations for these quantities which is crucial in minimizing the continual learning loss.

3.6.2 Curvature Approximation

Distance between solutions

We said that the term $\sum h_i(\theta^i - \theta_t^i)^2$ can be considered as a measure of distance between two models, θ and θ_t. The term h_i reflects the importance of parameter i, likely related to the curvature of the loss function at θ_t, indicating how changes in that parameter affect the loss. Is this a good measure of distance? It could be, if h_i is a good approximation of the loss curvature.
Can we find other meaningful and computationally efficient importance measures? A possibility can be using the simpler L2 distance, $\sum_i(\theta^i - \theta_t^i)^2$, but what truly matters in model comparison is not the parameter values themselves, but the distance between the probability distributions these parameters define. Essentially, the goal is to quantify how much the predictions of the model have changed, not just the parameters.

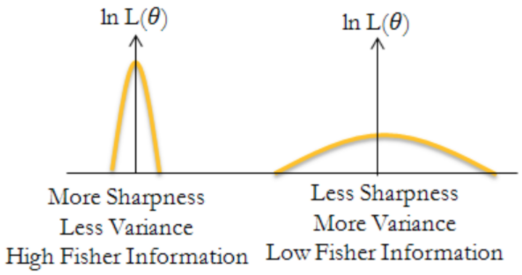
Kullback Leibler (KL) Divergence measures the distance between two probability distributions. It can be computed in closed-form for simple distributions; otherwise, we can estimate it using the data. KL is not symmetric.
We would like to meet two main objectives. First, we aim to consider the "local geometry" during SGD, which involves understanding the structure of the loss landscape in the vicinity of the current parameter configuration. Second, we try to change only the parameters that won't increase the KL distance too much, w.r.t. the previous model.

Fisher Information

Fisher information quantifies the amount of information that an observable random variable X provides about an unknown parameter θ upon which the probability of X is conditioned (how much information an observation carries about a parameter).

$$FI(\theta^*) = \mathbb{E}\left[(\frac{\partial}{\partial\theta}\log f(X;\theta^*))^2\right] \tag{3.8}$$

Fisher information measures how much the probability density function $f(X;\theta)$ for a random variable X "peaks" with respect to the unknown parameter θ. When f is sharply peaked, small changes in θ lead to large changes in the probability of observing a particular outcome of X, which means that X carries a lot of information about θ. Conversely, if f is flat, X provides less information about θ. The score $\frac{\partial}{\partial\theta}\log f(X;\theta)$ indicates the sensitivity (slope) of the likelihood $f(X;\theta)$ to changes in the parameter θ.



Fisher information is the variance of the score. When the Fisher information is high (Left), the parameter θ carries a lot of information about the data, meaning small changes in θ significantly affect the function. Conversely, when the Fisher information is low (Right), θ is not very meaningful; changing θ has little impact on the function.

Fisher's theorem essentially states that if we know how the likelihood (or equivalently, the log-likelihood) changes as we tweak our parameter θ, we can determine how much information about θ our random variable X contains. This information is used in estimation procedures to find the best estimate of the parameter θ. Fisher Information essentially measures the expected amount of information that would be lost if the true parameter θ^* were to be replaced with an alternative parameter θ. In other words it measures how **sensitive the distribution is w.r.t. changes to each parameter**. The Fisher information is equivalent to the **curvature** of the $KL(\theta|\theta^*)$. It gives a sense of the "shape" of the divergence around the true parameters θ^*. The Fisher information can be interpreted as the amount of "curvature" or "sensitivity" of the statistical model to changes in its parameters, thus can be used as an **approximation of the Hessian**.
The KL is not the loss function, but it's a good distance measure and often close to the loss function (e.g. crossentropy). It is easy to compute since we only need the gradients.

Different Fisher Information

There are variants of Fisher Information:

- **True Fisher Information**: this is a theoretical construct which measures how much information a random variable X provides about an unknown parameter θ under the true distribution $p_\theta(Y|X)$. It is not computable in practice because we do not have access to the true distribution $p(X)$.
- **Fisher Information**: this is what is typically computed in machine learning. It uses the model's output distribution $p_\theta(Y|X_n)$, where X is sampled from the dataset. It provides an expectation of the gradient's outer product, giving an approximation of the curvature of the likelihood landscape at θ.
- **Empirical Fisher Information**: this is a practical approximation that is easier to compute. It uses the gradient of the log-likelihood with respect to the parameters, evaluated at data points sampled from the dataset. It is not considered a good approximation for the curvature of the KL divergence (which is related to the true Fisher Information), but is often used due to its computational simplicity.

Application of the Fisher Information in Information Geometry

Information geometry uses concepts from differential geometry to study the properties of probability distributions. Fisher information is a key concept in information geometry, providing insight into the local structure or "shape" of the space of probability distributions (known as a **statistical manifold**) that a statistical model can represent. **Parameters** that are **more sensitive to small changes** can be identified through **Fisher information**, and the **sensitivity** can be **quantified** using the **KL divergence**. The KL divergence measures the difference between two probability distributions, providing a way to understand how a small change in parameters can lead to changes in the model's output distribution.

Fisher information can enhance the learning process by providing valuable insights into parameter importance and enabling training methodologies that respect the underlying structure of the model's parameter space. We can use this information for:

- Speed up training with **Natural Gradient Descent**.
- Prevent forgetting with **Elastic Weight Consolidation**

3.6.3 Natural Gradient Descent - Fast and stable SGD

When we do a descent step, the **learning rate is the same for every parameter**. However, with the Fisher information we know that some parameters are more sensitive than others (in the KL distance). *Can we account for the KL distance when we do a SGD step*?

Stochastic Gradient Descend with Kullback-Leibler Distance

We can **incorporates the KL divergence to SGD** reframing it as a constrained optimization problem.

$$\begin{aligned} \Delta\theta^* &= \underset{\Delta\theta}{\operatorname{argmin}}\ \mathcal{L}(\theta + \Delta\theta) \\ \text{s.t. } & D_{KL}(p_\theta \mid\mid p_{\theta+\Delta\theta}) = c \end{aligned}$$

The goal is to find the parameter update, $\Delta\theta^*$, that minimizes the loss function $\mathcal{L}(\theta + \Delta\theta)$ under a certain constraint. The **constraint is that the KL divergence** between the probability distribution of the model parameters before the update, p_θ, and after the update, $p_{\theta+\Delta\theta}$, **remains constant** at some value c. This means that while we want to update the parameters to minimize the loss, we want to do so in a way that **doesn't drastically change the distribution of our model's predictions**. By doing this, we are making sure that the steps taken in SGD do not lead to large deviations in terms of the distribution characterized by our model parameters. It is an effort to make sure that our model updates are significant enough to **improve performance** (by reducing loss) while still **being close to the previous parameter's distribution** (thus not moving too far in the parameter space).
This approach is equivalent to regular SGD if we replace the KL divergence with the $L2$ norm. This means that regular SGD can be seen as performing this constrained optimization with an $L2$ norm constraint on the size of the parameter updates instead of using the KL divergence. In essence, this is a method to regularize or constrain the learning process to ensure more controlled updates, potentially leading to more stable convergence during training.
To solve the constrained optimization problem we use the Lagrangian approximating both the loss and the KL with their Taylor expansion:

$$\Delta\theta^* = \arg\min_{\Delta\theta} \left[\mathcal{L}(\theta) + \nabla_\theta \mathcal{L}(\theta)\Delta\theta + \frac{1}{2}\lambda(\Delta\theta)^T Hf(\theta)\Delta\theta - \lambda c \right] \tag{3.9}$$

Here, $\mathcal{L}(\theta)$ is the loss function, $\nabla_\theta\mathcal{L}(\theta)$ is the gradient of the loss function with respect to θ, λ is a regularization parameter, and c is a constant. The **hessian** $Hf(\theta)$ mentioned is **equivalent to the Fisher information** $FI(\theta)$, a concept from information geometry that quantifies how much information the likelihood function of the parameters carries about the data. To find the minimum with respect to $\Delta\theta$, we need to compute the gradient of the objective function inside the argmin with respect to $\Delta\theta$ and set it to zero. Let's denote the objective function by $\mathcal{J}(\Delta\theta)$:

$$\mathcal{J}(\Delta\theta) = \mathcal{L}(\theta) + \nabla_\theta \mathcal{L}(\theta)\Delta\theta + \frac{1}{2}\lambda(\Delta\theta)^T FI(\theta)\Delta\theta - \lambda c \tag{3.10}$$

Now, compute the gradient of $\mathcal{J}(\Delta\theta)$ with respect to $\Delta\theta$.

- The term $\mathcal{L}(\theta)$ is constant with respect to $\Delta\theta$, so its gradient is zero.
- The gradient of $\nabla_\theta\mathcal{L}(\theta)\Delta\theta$ with respect to $\Delta\theta$ is $\nabla_\theta\mathcal{L}(\theta)$.
- The gradient of $\frac{1}{2}\lambda(\Delta\theta)^T FI(\theta)\Delta\theta$ with respect to $\Delta\theta$ is $\lambda FI(\theta)\Delta\theta$.

Thus, the gradient of $\mathcal{J}(\Delta\theta)$ w.r.t. $\Delta\theta$ is set to 0 :

$$\nabla_{\Delta\theta}\mathcal{J}(\Delta\theta) = \nabla_\theta\mathcal{L}(\theta) + \lambda FI(\theta)\Delta\theta = 0 \tag{3.11}$$

Solving for $\Delta\theta$:

$$\lambda FI(\theta)\Delta\theta = -\nabla_\theta\mathcal{L}(\theta) \tag{3.12}$$

Therefore, the optimal $\Delta\theta^*$ is:

$$\Delta\theta^* = \frac{1}{\lambda}FI(\theta)^{-1}\nabla_\theta\mathcal{L}(\theta) \tag{3.13}$$

This results in an equation for $\Delta\theta^*$ that includes the inverse of the Fisher information matrix and the gradient of the loss function. The factor $1/\lambda$ from the Lagrange multiplier is suggested to be analogous to the learning rate in the optimization process.
The **Natural Gradient Descent** update is:

$$\theta^* = \theta - \eta FI(\theta)^{-1}\nabla_\theta\mathcal{L}(\theta) \tag{3.14}$$

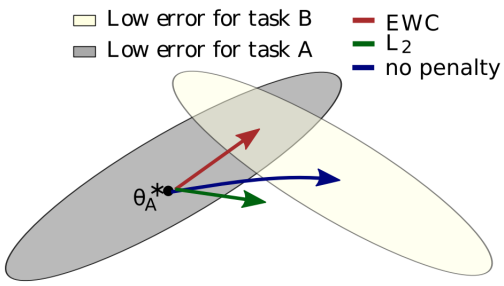
where η is the learning rate. The update steps considers the local curvature of the KL. Intuitively, more sensitive parameters are moved less, while less sensitive parameters are moved more (we multiply by the inverse). NGD is more stable.

Recap

The Fisher Information provide a local distance measure for our model. We can exploit it to make the gradient descent faster and more stable. NGD and the Fisher information appear everywhere in ML models: many methods employ some form of NGD under the hood, others using the natural parameters, a parametrization where we don't need the Fisher info (unfortunately, we don't have this luxury with DNNs).

3.6.4 Elastic Weight Consolidation

Importance-based methods are **prior-focused**, meaning they **incorporate knowledge from previous tasks** to inform the current learning process. This is done by modifying the model's loss function to include surrogate losses that represent previous tasks. The core idea is that **"important" weights** in the model (those that have been determined to be crucial for the performance on previous tasks) **should not change much** as new tasks are learned. Conversely, **"unimportant" weights** can be **adjusted more freely** to acquire new knowledge. This is predicated on the fact that deep neural networks are often overparameterized, having more parameters than strictly necessary, which should theoretically provide the capacity to learn multiple tasks without interference.
The problem that arises is how to identify which weights are important. This is where **Fisher Information** comes into play. It can be used to **measure the importance of weights** based on how changes in the weights affect the performance of the model on the data distribution. Weights with high Fisher Information are considered more important because they have a greater impact on the model's output.
Elastic Weight Consolidation (EWC) is a method for continual learning that uses the Fisher Information to prevent forgetting. In a EWC setting we are typically dealing with large batches, we know task boundaries, we have enough data to find a good model (i.e. the model at the boundaries is optimal for the current task). It is designed for task labels, but can work without them. The key idea of EWC is to protect the knowledge gained from previous tasks by penalizing changes to the most important parameters when learning new tasks. The importance of each parameter is determined by the Fisher Information, which quantifies how sensitive the prediction error is to changes in that parameter.
After training on a task (A), the model's parameters (denoted as θ_{i-1}) and the corresponding Fisher Information ($FI_{i-1}(\theta_{i-1})$) are stored. The Fisher Information is computed using the data from the task just completed (D_{i-1}). During the training of subsequent tasks (B), an additional loss term (EWC loss) is added to the total loss function. This EWC loss is a sum over all parameters, where each term is the squared difference between the current parameter value (θ_i) and its value after training on previous task A, ($\theta^*_{A,i}$), weighted by $\lambda/2$ of the Fisher Information for that parameter. This creates a



EWC ensures task A is remembered whilst training on task B. Training trajectories are illustrated in a schematic parameter space, with parameter regions leading to good performance on task A (gray) and on task B (cream). After learning the first task, the parameters are at θ^*_A. 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. EWC, conversely, 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.

quadratic penalty for moving away from the previous parameters, which represents a trade-off between learning new tasks B and retaining knowledge from previous ones. Formally the loss is:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} FI_i(\theta_i - \theta^*_{A,i})^2 \tag{3.15}$$

where $\mathcal{L}_B(\theta)$ is the loss for task B only, λ sets how important the old task is compared to the new one and i labels each parameter.
It is **Elastic** because weights are pulled towards the previous solution (θ^{i-1} as the center of the loss), weighted by their importance. How many importance values do we need to keep?

- Separate Method: for each task, a tuple of the model parameters θ_{i-1} and the corresponding Fisher Information FI_{i-1} is stored after training. This method implies a linear cost because it's equivalent to saving a completely separate model for every experience, leading to a significant increase in storage requirement as the number of tasks grows.

- Online Method: instead of storing a separate Fisher Information for each task, an online estimate is maintained. After each task, the Fisher Information is updated using an average or an Exponential Moving Average (EMA) of the Fisher values. This approach is more storage efficient and adapts to new tasks without needing to keep separate Fisher Information for each past task. It is akin to performing a sequential Bayesian update.

The system is sensible to the regularization strength λ, which balances between learning new tasks and retaining knowledge of previous ones. The higher the λ, the more the model will try to conserve previous knowledge, potentially at the cost of learning new tasks effectively.
Importance weights follow a logarithmic distribution, this is a good news for us because having few important parameters means that we can use a strong regularization coefficient for them while having enough free capacity to learn new tasks.

Recap

EWC provides a good proxy loss for past experiences, computing the Fisher requires only the gradients (computed at boundaries). The problems are that needs task boundaries and large batches, which makes it useless in online settings. In separate mode it has a linear cost in the number of tasks (one Fisher and one model stored for each task).

3.6.5 Synaptic Intelligence

We need an **online importance estimate** because EWC requires a trained model to estimate the Fisher Information, it needs task boundaries, lots of data and it is only applicable to batch scenarios. **Synaptic Intelligence** (SI) is a method for calculating the importance of neural network parameters based on their contribution to the decrease in loss throughout the training process. The key concept is that a **parameter's importance** is **determined by how much it has helped reduce the loss** function over the training period. This is calculated by looking at the parameter's weight changes $\delta(t)$ and how these changes affect the loss. The change in the loss $\mathcal{L}(\theta(t) + \delta(t)) - \mathcal{L}(\theta(t))$ is approximated using the gradient (for small steps). The total contribution to the loss change is the sum of weight changes, weighted by the gradient, over the entire training curve.

$$\mathcal{L}(\theta(t) + \delta(t)) - \mathcal{L}(\theta(t)) \approx \sum_k g_k(t)\delta_k(t) \tag{3.16}$$

This technique aims to capture the "synaptic" contribution of each weight to the learned knowledge, assigning greater "importance" to weights that have a more significant effect on reducing the loss, and thus potentially contributing more to what the network has learned. In other words a parameter importance is proportional to its contribution to the loss decrease over the entire training trajectory. It is an efficient online computation but, as with EWC, hyperparameters may be difficult to calibrate.

3.7 Regularization - Data-focused

In continual learning, data-focused regularization methods aim to preserve the performance of a model on previous tasks as it learns new tasks. These methods operate by mimicking the output of the model for the previous tasks, often by **keeping a subset of the old data** (also called rehearsal or replay data). When new data arrives for a new task, the model is trained **not only on this new data but also on the stored old data**, so the model continues to perform well on both old and new tasks.
This approach helps to counteract catastrophic forgetting, which is the tendency of a neural network to completely forget previously learned information upon learning new information. By **rehearsing** the old data, the network's weights are regularly adjusted to maintain performance on the older tasks, effectively preserving the knowledge across the spectrum of learned tasks. This also allows the model to maintain a form of stability in its predictions over time, which is critical for applications where models must be deployed in changing environments without losing accuracy on the core competencies they have already mastered.

Classic hyperparameters like early stopping, L1/L2 weight decay, and dropout serve to **alleviate the issue of forgetting** in machine learning models. By applying regularization techniques, such as these, we effectively minimize the extent of model adjustments during training. Consequently, this reduced variability translates into lesser forgetting within the model. However, it's important to note that relying solely on these hyperparameters **may not be sufficient** to entirely prevent forgetting in practice.
In offline training, the model optimizes a single loss $\mathcal{L}(D, \theta)$ that is constant during the entire training loop. In replay-free CL, the model optimizes $\mathcal{L}(D_t, \theta)$, while the true objective is $\sum_t \mathcal{L}(D_t, \theta)$ we need to ensure that the new minima remains a good solution for the old experiences.

3.7.1 Continual Learning Loss

The **Taylor expansion** helps in understanding how the loss function behaves near a particular point θ_t by using both the gradient (first derivative) and curvature (second derivative). This approximation is valuable for optimization algorithms that rely on local information to update the parameters in a direction that minimizes the loss.

$$\mathcal{L}^{old}(\theta) \approx \mathcal{L}^{old}(\theta_t) + \nabla\mathcal{L}^{old}(\theta_t)^T(\theta - \theta_t) + \frac{1}{2}(\theta - \theta_t)^T Hf(\theta_t)(\theta - \theta_t) \tag{3.17}$$

This concept can be applied to approximate the loss function for previous tasks (denoted as $\mathcal{L}^{old}$) around the point θ_t, which represents the parameters of the model at time t. The first term in the equation, $\mathcal{L}^{old}(\theta_t)$, is the value of the loss at θ_t. The second term, $\nabla\mathcal{L}^{old}(\theta_t)^T(\theta - \theta_t)$, represents

the first-order approximation, which involves the gradient of the loss at θ_t. The third term, $\frac{1}{2}(\theta - \theta_t)^T H_{f(\theta_t)}(\theta - \theta_t)$, represents the second-order approximation, incorporating the curvature of the loss landscape via the Hessian matrix at θ_t.
The **curvature** described by the Hessian matrix gives us insight into the **local geometry of the loss function's landscape** at the minimum point θ_t. Since DNNs typically have numerous local minima, each with its own curvature properties, using the Hessian allows us to better understand and predict the loss associated with small changes to the parameters. By focusing on the curvature, we can effectively model the behavior of the loss function for past data around the solution. This approach is crucial in continual learning, where we want to maintain performance on previous tasks while minimizing interference as the network learns new tasks.
The idea is to find a **minima with a flat curvature**: even without any regularization, **flat minima** should **mitigate forgetting**.
The geometry of the loss landscape matters:

- If a new solution (set of parameters) is similar enough to the previous one, the loss for prior tasks can be approximated using a linear method. Idea 1: **Minimizing changes** to the model to **protect the performance on previous tasks**. This means making only necessary updates to the parameters to maintain what has been learned previously. Idea 2: **encouraging maximum learning** from the initial task. By solidifying the learning from the first task, the model may become more robust to future changes.

- The wider the curvature of the first task, the less the forgetting. Idea 3: steering the optimization process towards "flatter" minima. **Flatter areas in the loss landscape** indicate regions where the model's performance is **less sensitive to parameter changes**, which could **reduce forgetting** when new tasks are introduced.

The equation F_1 represents the change in loss moving from one set of weights w_1 to another w_2.

$$F_1 = L_1(w_2^*) - L_1(w_1^*) \approx \frac{1}{2}\Delta w^T \nabla^2 L_1(w_1^*)\Delta w \leq \frac{1}{2}\lambda_{\max}\|\Delta w\|^2 \tag{3.18}$$

It's approximated by the second order Taylor expansion, where Δw is the change in weights, and $\nabla^2 L_1(w_1^*)$ represents the Hessian matrix of the loss at the optimal weights w_1^* for the first task. The inequality expresses that this change in loss is bounded above by a term involving the largest eigenvalue of the Hessian (λ_{max}) and the square of the weight change (Δw). This captures the idea that changes to the weights should be controlled in magnitude to prevent large increases in the loss for previous tasks.

Through careful adjustment of SGD hyperparameters, we can steer the algorithm towards optimal solutions, reflecting an inherent bias in SGD. A general rule suggests starting with a high initial learning rate for the first task to secure a broad and stable minima. Subsequently, for each new task, it's advisable to marginally reduce the learning rate alongside decreasing the batch size. Additionally, a preference for SGD over Adam optimizer is recommended.

3.7.2 Functional Regularization

Functional Regularization for CL, avoids forgetting a previous task by **constructing and memorising an approximate posterior belief** over the underlying task-specific function.
There is access to a model f_{i-1}^{CL} which has been trained on the previous $i-1$ tasks. The central idea proposed is to **replicate the behavior** of the **old model on previously learned tasks**, ensuring that it remains consistent with what was learned before, while allowing it to update based on new examples for the current task. Two primary challenges are raised: what is the goal that we are optimizing while training the model on the new task? What dataset do we utilize for training? The model must remember old tasks without having access to the old training data, which is a challenge in CL known as catastrophic forgetting.

The equations in a multi-task learning scenario is:

$$\begin{aligned} f_i^{CL}(x,k) &= f_{i-1}^{CL}(x,k), && \forall x \in \mathbb{R}^N, k \neq i \\ f_i^{CL}(x,i) &= f_i^{Exp}(x), && \forall x \in \mathbb{R}^N. \end{aligned}$$

The first equation states that for any input x and task label k that is not equal to the current task i, the model's output should match the output of the model trained up to the previous task (f_{i-1}^{CL}). The second equation indicates that for the current task labeled i, the model's output should be determined by a new model, f^{Exp}, trained specifically for this task. In other words the first equation implies that the model should replicate the CL model up to task $i-1$ for all previous tasks, while the second equation suggests that the model should adopt the behavior of a new *expert* model for the new task. This approach helps to prevent the model from forgetting how to perform on tasks it was trained on previously, which is a common issue when sequentially training models on new tasks.

Knowledge Distillation

Knowledge Distillation is an offline training method used to **transfer knowledge** from a larger, pre-trained model (referred to as the "teacher") to a smaller, new model (referred to as the "student"). The **teacher** is the pre-existing, typically larger or more complex model that has been pre-trained on a specific task and whose outputs are used to guide the training of the student model. The teacher model's parameters are kept fixed during this process. The **student** is the new model that is being trained. The goal is for the **student** model to **learn to replicate** the outputs of the **teacher** model despite potentially having fewer parameters or less complexity.
KD is not limited to continual learning scenarios. It is a versatile approach that can be used in various situations, such as model compression. KD works well because it provides the student model with "soft targets" from the teacher model, which can be more informative than "hard targets" (the actual class labels). Soft targets can capture the similarity between classes and carry additional information about the data distribution, sometimes referred to as "dark knowledge". This process enables the student model to perform better or similar to the teacher model despite being smaller or less complex, as it leverages the nuanced insights the teacher model has gained from its training.
The **teacher student difference** can be measure with the **KL** divergence or classic **MSE**.

Learning without Forgetting

Learning without Forgetting (LwF) employs functional regularization with the objective of **knowledge distillation** while leveraging **current data**. It is straightforward implementation of KD in CL. It demonstrates efficiency, necessitating only an additional forward pass with the teacher model and it is known for its simplicity in implementation and widespread usage.

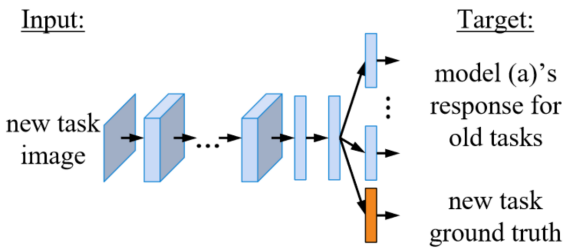
LwF in the context of **Task-Incremental** involves a **multi-head** mechanism. In this setup, there's a separate output layer (or "head") for each task in a neural network. This allows the network to give task-specific responses.
Given a CNN with shared parameters θ_s and task-specific parameters θ_o the goal is to add task-specific parameters θ_n for a new task and to learn parameters that work well on old and new tasks. This is achieved using only images and labels from the new task, without using old data, by transferring the learned knowledge through distillation.

LEARNINGWITHOUTFORGETTING:
Start with:
θ_s: shared parameters
θ_o: task specific parameters for each old task
X_n, Y_n: training data and ground truth on the new task
Initialize:
$Y_o \leftarrow \text{CNN}(X_n, \theta_s, \theta_o)$ // *compute output of old tasks for new data*
$\theta_n \leftarrow \text{RANDINIT}(|\theta_n|)$ // *randomly initialize new parameters*
Train:
Define $\hat{Y}_o \equiv \text{CNN}(X_n, \hat{\theta}_s, \hat{\theta}_o)$ // *old task output*
Define $\hat{Y}_n \equiv \text{CNN}(X_n, \hat{\theta}_s, \hat{\theta}_n)$ // *new task output*
$\theta_s^*,\ \theta_o^*,\ \theta_n^* \leftarrow \underset{\hat{\theta}_s, \hat{\theta}_o, \hat{\theta}_n}{\text{argmin}} \left(\lambda_o \mathcal{L}_{old}(Y_o, \hat{Y}_o) + \mathcal{L}_{new}(Y_n, \hat{Y}_n) + \mathcal{R}(\hat{\theta}_s, \hat{\theta}_o, \hat{\theta}_n) \right)$

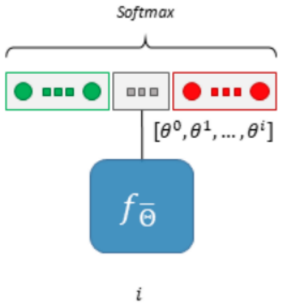
The overall objective is to train the network on a new task (using the new head) without forgetting the previous tasks (using the old head). The losses from KD and cross-entropy are combined to form a total loss function:

$$\mathcal{L}_{total} = \lambda_o \mathcal{L}_{old}(y_o, \hat{y}_o) + \mathcal{L}_{new}(y_n, \hat{y}_n) \tag{3.19}$$

Here, λ_o is a hyperparameter that balances the importance of retaining old knowledge against learning new information. y_o and $\hat{y}_o$ represent the true (old predicted, KD) and predicted outputs for the old tasks, while y_n and $\hat{y}_n$ represent the true and predicted outputs for the new task. The presented equations for $\mathcal{L}_{old}$ and $\mathcal{L}_{new}$ are cross-entropy loss functions, with $\mathcal{L}_{old}$ being applied to the outputs from the old tasks and $\mathcal{L}_{new}$ being applied to the output from the new task. This method aims to retain learned responses for old tasks while effectively learning the new task.



In contrast with the previous method where each task had its own output layer (multihead), in Learning without Forgetting (LwF) within **Class-Incremental** we have only one output layer (**single head**) that handles all the classes from both old and new tasks. The output units (neurons) in the single head are differentiated into "old units" for previously learned classes and "new units" for newly introduced classes. The model needs to maintain performance on the old units while learning the new ones. KD is applied to the old units to retain the knowledge when training on new data. This is achieved by comparing the predictions of the old units from the current model to those of the previously trained model and penalizing discrepancies. While KD focuses on old units, the cross-entropy loss is calculated over all units for the new data to ensure the model learns the new classes.
Maintaining the performance of the old units while introducing new ones is a challenge addressed by LwF. This is crucial in scenarios where you can't store all the data from old tasks due to memory constraints, or you want to avoid catastrophic forgetting.



Knowledge Distillation is a **function matching** problem where the desired output for any given input is precisely known. Unlike conventional ML problems, the emphasis lies less on the dataset itself and more on achieving the desired output. While diverse data related to the domain of interest can be beneficial, it's not imperative. KD can accommodate various inputs with differing convergence rates, including the teacher training data, out-of-domain data with augmentations, and even data from a new task, as seen in methods like Learning without Forgetting (LwF).

Weight vs Functional Regularization

There's contrasts between the two types of regularization strategies used in CL: Weight Regularization and Functional Regularization.

- **Prior-based methods (Weight Regularization)**: these methods constrain the weights of a new model such that they do not deviate significantly from the solutions of previous models. It

implies a conservative approach where the new model parameters are kept close to the parameters of the old model to prevent forgetting. However, this approach can be overly restrictive and typically requires either a large model capacity or a pretrained model to work effectively. It's based on an approximation of the real objective and usually employs penalties to enforce the similarity.

- **Functional Regularization**: this approach is less restrictive. Rather than focusing on keeping the weights similar, functional regularization focuses on preserving the output function of the model for previous tasks. The idea is to allow the weights to change freely as long as the outputs for the previous units (tasks) remain consistent. In other words, it cares more about what the model does (its function) rather than what the model is (its weights). This approach is particularly useful in continual learning scenarios where the model is expected to learn new tasks without forgetting the old ones.

Knowledge Distillation Miscellanea

We need **task boundaries** to know when to store the previous model. The data that we use for KD (new data) is different from the one we used to train the teacher (old data). **Augmentations** help KD, even when they distort the image.
KD can be integrated with **Experience Replay**, referred to as *Dark experience*, where tuples of inputs and logits are stored in the buffer using class-balanced reservoir sampling. During rehearsal, KD is applied with MSE loss on logits, eliminating the need for a teacher. This method involves saving the logits of the current teacher without updating them. Alternatively, labels can also be saved and used for rehearsal with a combination of KD loss and crossentropy loss. This approach has shown competitive results and offers low computational cost.
KD is traditionally used to transfer knowledge from a larger, trained teacher model to a smaller student model. In continual learning, KD is used to help the model retain previously learned information while it learns from new data. Combining Self Supervised Learning and KD is an approach that combines SSL loss on new data with KD loss on embeddings from the old model. The SSL loss helps the model learn from new, unlabeled or partially labeled data, while the KD loss ensures that the new model's output remains consistent with the previous model's output for previously learned tasks.

3.8 Classifier Bias

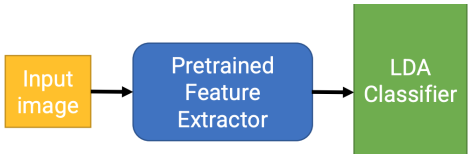
Studies find that in neural networks learning, **higher layers contribute significantly more to catastrophic forgetting compared to lower layers**. Analyzing representational similarity scores between layers before and after training, it's observed that **lower layers remain** relatively **stable** after few epochs while **higher layers change** significantly.
Effective strategies to mitigate forgetting include freezing lower layers after initial learning, leveraging pretrained models, and finetuning the classifier with the aid of a replay buffer to prevent memory loss.

3.8.1 Bias preventing Techniques

Deep Streaming Linear Discriminant Analysis

Deep Streaming Linear Discriminant Analysis (Deep SLDA), is a technique for training classifiers in a continual learning context. The technique starts with a deep neural network that has been pre-trained on a large dataset, such as ImageNet. This pre-trained model is used as a fixed feature extractor. Instead of using traditional neural network approaches for the classification layer, Deep SLDA utilizes Streaming Linear Discriminant Analysis. SLDA is a variant of **LDA** (Linear Discriminant Analysis), which is a classical machine learning method used for finding the **linear combinations of features that best separate two or more classes** of objects or events. The **streaming** aspect refers to the capability of the model to **learn incrementally** from a stream of incoming data, making it suitable for online learning scenarios. SLDA is particularly noted for its computational **efficiency** compared to methods that require backpropagation through the entire model for each new piece of data. By using a pre-trained feature extractor and **updating only the classifier layer**, it significantly reduces the computational burden. Deep SLDA can be applied to online learning scenarios,

where data arrives in a sequence over time, without the need to store or replay old data. This is beneficial when storage is limited or when privacy concerns prevent keeping data long-term.
In short, Deep SLDA leverages the strengths of deep neural networks for feature extraction and the efficiency of SLDA for classification to create a learning system that is both powerful and suitable for environments where data is continually updated.



SLDA trains a linear decoder $F(\cdot)$ to work with the features extracted by a pretrained model $G(\cdot)$. The relationship is defined by the equation:

$$F(G(X_t)) = Wz_t + b \tag{3.20}$$

where z_t is the feature vector produced by G for input X_t, W is a weight matrix, and b is a bias vector. These parameters are updated as new data comes in.
SLDA models **each class** as a **gaussian**. It maintains a mean vector μ_k for each class k and a count c_k of how many times that class has been seen, along with a single shared covariance matrix Σ. When a new data point (z_t, y) is observed, the mean vector for class y and the counter c_k are updated. The updated mean vector is calculated as a weighted average of the current mean and the new feature vector, and the count for the class is incremented.

$$\mu_{(k=y,t+1)} \leftarrow \frac{C_{(k=y,t)}\mu_{(k=y,t)} + z_t}{C_{(k=y,t)} + 1}$$
$$c_{(k=y,t+1)} = c_{(k=y,t)} + 1$$

The covariance matrix is updated using a shrinkage regularization approach, which adjusts the precision matrix Λ by blending the current covariance matrix Σ with the identity matrix I, scaled by a shrinkage parameter ε.

$$\Lambda = [(1-\varepsilon)\Sigma + \varepsilon I]^{-1} \tag{3.21}$$

For the SLDA variant that updates the covariance matrix online, the update is:

$$\Sigma_{t+1} = \frac{t\Sigma_t + \Delta_t}{t+1} \tag{3.22}$$

where Δ_t is calculated based on the deviation of the current observation from the mean:

$$\Delta_t = \frac{t(z_t - \mu_{(k=y,t)})(z_t - \mu_{(k=y,t)})^T}{t+1} \tag{3.23}$$

The prediction weights w_k for each class are then calculated using the precision matrix and the class means:

$$w_k = \Lambda\mu_k \tag{3.24}$$

And the bias b_k is determined by:

$$b_k = -\frac{1}{2}(\mu_k \cdot \Lambda\mu_k) \tag{3.25}$$

These equations show how SLDA can adaptively learn from a stream of data by updating its model parameters in real-time, a process that's essential for continual learning scenarios where data is available sequentially.

Multi-Head Classifier

A multi-head classifier architecture partitions a classifier into **multiple heads**, each responsible for a **different task**. Each head can specialize in the behavior for its designated task, which is particularly beneficial when tasks are clearly defined and distinct from each other. By limiting the number of classes

each head has to classify, the learning process is simplified because each head deals with a smaller, more manageable subset of the overall classification problem. There's a distinction between shared parameters, which are used across all tasks (typically found in the feature extractor), and task-specific parameters that are unique to each classification head. However, this architecture isn't universally applicable. It requires explicit task labels during training so the model knows which head to activate. Furthermore, at inference time, the correct task label is needed unless a mechanism is in place to infer it from the input.

Single Head in Class-Incremental

In class-incremental learning, the model needs to be able to learn new classes over time without forgetting previously learned ones. A **single-head classifier** model distinguishes **between old, current, and future classes** ("units"). This distinction allows the model to apply different strategies for preserving old knowledge and learning new information. As an example, in the LwF framework, Cross-Entropy (CE) loss is used for learning from both new and old units, whereas Knowledge Distillation (KD) is applied specifically for old units. This way, the model maintains its performance on old tasks while learning the new ones.
Masking Technique is a strategy for dealing with classes that are no longer relevant. By masking these classes, we effectively give them a zero probability, which helps prevent the model from making incorrect predictions about these classes.

3.8.2 Classifier Bias Issues

In the last layer of a DNN in a class incremental setting we see that **new classes** have **weights with a larger norm** (recency bias). If we don't have task labels, we encounter new classes over time, and we don't have replay, the DNN suffers from classifier bias. Classifier bias is the biggest source of forgetting in a naive method.

Mitigation

A **Cosine Classifier** can be used to reduce the bias in a classification task. This is done by normalizing the weights of the classifier's last layer, which can be seen as class embeddings θ_j. Normalization is applied to both the class embeddings and the extracted features before classification. The classification is then based on computing the cosine similarity between the normalized embeddings and features. Cosine similarity provides a measure of how similar the feature vectors are to the class weight vectors in terms of angle, regardless of their magnitude. This approach is particularly useful in situations where the magnitude of the vectors should not influence the classification decision, focusing instead on the directionality of the data.
Copy Weights with Re-Initialization (CWR) is another strategy used to mitigate the classifier bias. The core idea is to maintain two sets of weights: consolidated weights (cw) that are used for inference, and temporary weights (tw) that are used during the training phase. Additionally, a counter for the number of patterns seen for each class (past) and a scaling term (wpast) are employed to adjust the weights. In CWR the consolidated weights are updated with a normalized sum of all the learned weights after each learning experience, ensuring knowledge from previous tasks is retained. The temporary weights are reset to zero after each learning step, allowing for flexibility in learning new tasks without being hindered by old information. The method is designed to work across various continual learning scenarios, such as new instance, new class, and new instance and class, and is robust enough for online settings where data comes in a stream. This strategy can be seen as a dual learning system, where the model rapidly adapts to new information while also slowly integrating that information into a long-term memory, much like the short-term and long-term memory distinction in human cognitive processes.

Recap

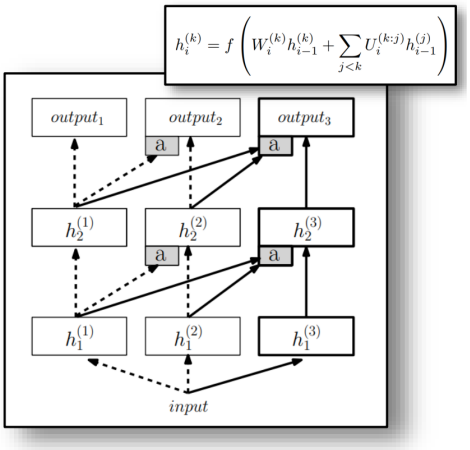
If we don't have task labels and we encounter new classes over time, and we don't have replay, the DNN suffers from classifier bias. To **mitigate this bias**, we could: train the classifier with algorithms that have **less classifier bias** (LDA), **normalize the weights** (Cosine Normalization) or **finetune/adapt the classifier** (CWR).

3.9 Architectural Methods

The **Modular Architecture approach** involves dividing networks into distinct modules. This division allows for modules to be connected in ways that facilitate **knowledge transfer between tasks**. To **prevent forgetting** what was learned in previous tasks, **some modules can be frozen or masked** during the training of new tasks.
Opportunities: Modular architectures enable clear differentiation between components that are task-specific and those that are shared across tasks. This can potentially eliminate the issue of catastrophic forgetting, where a network forgets previously learned information upon learning new information.
Challenges: These include controlling the increase in memory usage due to additional modules, the need for task labels which might not always be available, and ensuring forward transfer, which is the ability of a model to leverage knowledge from previous tasks to improve performance on future tasks. A successful modular approach needs to strike a balance between various factors (**Conflicting requirements**). It should manage memory usage effectively, prevent forgetting past information while also promoting the transfer of knowledge to assist in learning future tasks. This balance is crucial for the practical and efficient application of such models in continual learning scenarios.

3.9.1 Progressive Neural Networks

Progressive Neural Networks (PNNs) are a type of modular architecture designed for continual learning tasks, where the model needs to adapt to new tasks over time without forgetting previously learned information.
For every new task, a PNN **adds** a new set of neurons (or a **column**) to the network. This column has its own feature representations which are unique to the new task and do not alter the features learned for previous tasks. These **columns are not isolated**; they are **connected** to previous columns **via adapters**. Adapters allow the columns to communicate and transfer knowledge, which means the new task can benefit from the features learned by the network on previous tasks. The adapter operates by taking a weighted concatenation of the activations from all previous columns. This means that it combines the feature representations (activations) of all previous tasks into a single vector, and these combined features are weighted to give certain features more importance based on the new task's requirements. When the network is making predictions (**inference**), it uses task labels to determine which column(s) to activate. This ensures that the model uses the correct features for a given task.

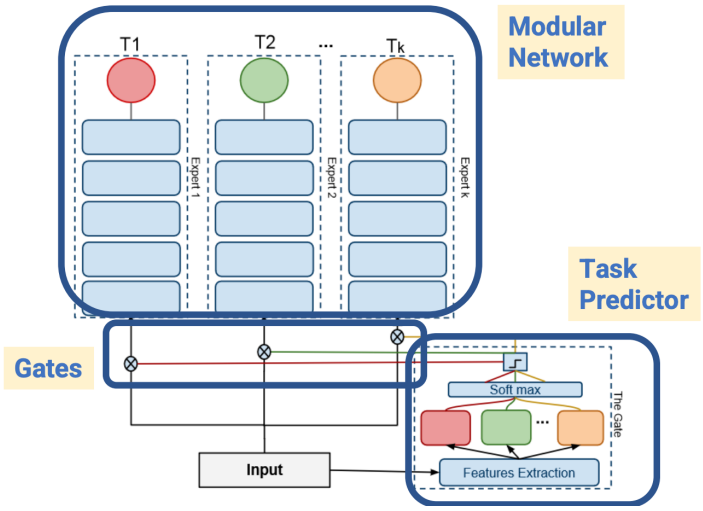


The main **advantage** of PNNs is their ability to preserve learned knowledge (avoiding catastrophic forgetting) by keeping the weights for previous tasks fixed. New tasks can leverage existing features through the adapters but do not overwrite or interfere with them. This architecture provides a structured way to manage continual learning, where tasks are learned sequentially without the need for the model to retain all the training data from past tasks.
PNNs exhibit **efficient forward transfer abilities**, enabling each new task to leverage knowledge from previous ones through re-using "columns" of neural network layers dedicated to those tasks. This

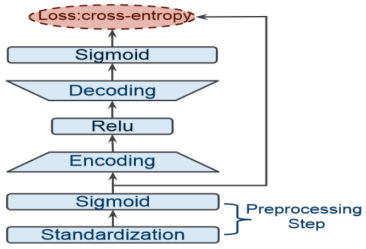
architecture also prevents catastrophic forgetting by "freezing" columns related to previous tasks, thereby preserving the learned representations.
However, a **significant drawback** of PNNs is their **poor scalability** regarding memory usage, with memory requirements growing quadratically due to the need for adapters connecting new and previous columns. Furthermore, PNNs necessitate **task labels** for operation, as these labels determine which columns should be. A good news is that most of the capacity of a PNN is not used. We can reduce the size of new columns over time (limiting memory growth) by compressing them after training.

3.9.2 Gated Ensemble

On top of a modular architecture we could add a **task inference classifier to remove need for task labels**, a classifier that given an input predicts the task label. 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). By predicting the labels, the system becomes more autonomous, reducing the dependency on external information and allowing for a more flexible and scalable learning system.
The **Gated Ensemble** approach within a modular neural network architecture introduces the concept of a gate. A gate is a mechanism for **enabling or disabling different modules** within the network. These modules are organized to correspond to specific tasks or functions. In scenarios where task labels are not provided, these **gates** can be **controlled by a task label predictor**. This means that for a given input, the network can predict which task is being performed and subsequently activate only the relevant module or modules needed for that task, optimizing the network's performance for that specific task and avoiding interference from other modules. This system's advantages include specialization where each module (**Expert**) can become highly competent at its respective task, and a reduction in catastrophic forgetting since modules for previous tasks can be left unchanged as new tasks are learned. However, this architecture needs to maintain competition or normalization between the gates to manage resource allocation effectively and ensure that the correct task module is activated at the right time.



The **Gate** controller is an autoencoder that takes as input a the output of the last convolutional layer of a pretrained model (e.g. AlexNet pretrained on ImageNet). The input is standardized and passed through an undercomplete autoencoder for each task. Inference is done using the expert model associated with the most confident autoencoder.



Regarding the **Experts** the primary idea is to develop specialized *expert modules* that are adept at handling specific tasks. The input for training an expert is the same as that for the gate (output of a CNN).

When a **new task** is introduced, the system doesn't start from scratch. Instead, it **selects the most related expert** based on a *Task Relatedness Measure* and uses this expert as the initial setup. This measure is not symmetric and if the new task is quite similar to a previous one (above a certain threshold), it uses LwF, which is a strategy designed to update the model for a new task while retaining performance on previous tasks. If the tasks are not sufficiently related, the system uses finetuning, where the model is adjusted specifically for the new task at the potential expense of older tasks (which is not very related).
The task relatedness is based on the difference in reconstruction errors from an autoencoder for the new task versus the old task. The closer this value is to 1, the more related the tasks are considered to be. In essence, the Expert Gate system aims to leverage previously learned representations and expertise to efficiently learn new tasks. By initializing new modules with related existing ones and selectively training with LwF or finetuning, the system navigates the trade-off between adapting to new tasks and retaining knowledge of old ones.

The Expert Gate presents a modular architecture designed to be **agnostic to specific tasks**, offering simple mechanisms for **inferring** tasks. It provides valuable insights into the relationship between transfer learning and task relevance. Despite its strengths, this approach has limitations. It relies on a pretrained network, which may not always be available or suitable for the intended task. Additionally, the reconstruction error used as a task predictor might not consistently yield accurate results, as autoencoders excel at reconstructing unseen data but may struggle with task inference.

3.9.3 Masking and Architectural Sparsity

There is a concern about increasing memory occupation over time as more tasks are learned. Deep neural networks often have more parameters than necessary for a given task, leading to **overparameterization**. This means that there are redundant or non-essential elements within the model that do not contribute to performance on the task.
To address this, a solution is to utilize a large, fixed-size neural network and **selectively activate a subset of units** (neurons) for each task through **masking**. This is achieved by applying *masks* that effectively turn on certain pathways within the network while turning others off. This approach is said to be similar to modular networks where different modules are responsible for different tasks, but masking is less resource-intensive. The binary nature of the masks (where units are either on or off) makes them easy to store and compress, reducing the memory footprint. Binary masks are discrete parameters so we can't compute gradients and cannot use a straightforward SGD. We will assume that optimization algorithms are available (oracles) and solve the problem of learning masks in CL. Masking inherently induces **sparsity** in the network, which can lead to more efficient computation and potentially reduce the risk of overfitting.
Lottery Ticket Hypothesis: within a randomly-initialized, dense neural network, there are smaller subnetworks (referred to as *winning tickets*) that can be trained to achieve test accuracy comparable to the entire network in a similar number of training iterations, but using fewer resources. The Lottery Ticket Hypothesis is not a formal theorem, but rather an observation.

Sparsity in CL

Sparse representations offer valuable advantages for CL. They enable the **separation of tasks** into distinct, **independent subnetworks**, which in turn **prevents interference** and **eliminates forgetting**. Moreover, their high level of sparsity facilitates the storage of numerous networks even within memory constraints. The objective is to activate diverse subnetworks for each task, minimizing interference. This approach aligns with biological plausibility, given the vast number of neurons and synapses in the brain.
A **subnetwork** is the set of units that activate for a particular task/class. Similar digits have similar subnetworks and subnetwork are less similar after training than before meaning that competition helps learning specialized and separate subnetworks.
There are various types of sparsity in continual learning:

- Weight Sparsity: utilizing small sparse networks for each task facilitates the creation of cost-effective ensemble models. However, employing a single weight-sparse network for all tasks does not offer the same benefits.
- Activation Sparsity: sparse representations should exhibit increased orthogonality, reducing interference between tasks.
- Gradient/Update Sparsity: sparse gradients should demonstrate enhanced orthogonality, minimizing interference during updates.

Note that sparsity considerations also involve stability rather than just plasticity. Often, sparse methods tend to decrease forward transfer.
There are two primary approaches to achieving sparsity:

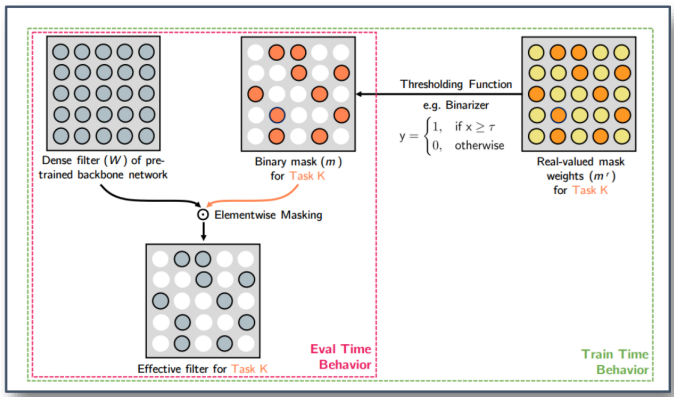
- **Implicit Approach**: utilizes regularization methods that implicitly encourage sparsity. L1 encourages sparsity $|\theta|$ and L2 encourages small weights (not sparsity) $|\theta|^2$.
- **Explicit Approach**: involves architectural methods that explicitly construct sparse subnetworks.

Local Winner-Take-All (LWTA) is a competition mechanism in neural networks. It refers to a process where, in a given competition, **only the highest activated neuron's signal is allowed to propagate forward** through the network. The other neurons, which are not the winners of this competition, have their activations suppressed to zero. The competition is localized within the network. This means that the network's layers are segmented into separate blocks, and the competition for which neuron's activation will propagate forward is carried out within these individual blocks. Because of the WTA rule, sparsity is induced in the network. This sparsity can lead to more efficient and possibly more interpretable neural networks, as fewer neurons are active at a time. The LWTA model is inspired by how biological neural systems are believed to operate, with competition among neurons and selective activation patterns making it a biologically plausible mechanism.

Weight Mask (Piggyback)

The approach starts with a **fixed pre-trained model** (e.g. ResNet18) which has been previously trained on a dataset (e.g. ImageNet). This model serves as the **backbone** or base network. For each new task binary masks are trained. These masks effectively turns certain weights of the pre-trained model on or off when applied (element-wise multiplication).
Initially, the mask is trained with floating-point values and is subsequently binarized to create a binary mask. By using this mask, the original pre-trained weights of the model are not modified, which ensures that there is no forgetting of the previous tasks the model has learned. While this method prevents forgetting, it also means that there's no transfer of knowledge between tasks since the masks are task-specific and no weights are shared or modified across tasks.
The method is memory efficient because each task's binary mask requires only a small amount of memory (1 bit per weight per task). Thus, only a handful of kilobytes are needed for each new task. The system requires knowledge of which task is being addressed at any given time in order to apply the correct mask.

In **PiggyBack**, a unique **binary mask** is associated with **each layer** of the neural network and each element of the mask corresponds to a weight in the neural network's layer. When a forward pass is conducted (i.e., when input data is passed through the network to get a prediction), the weights of the network are element-wise multiplied by the binary mask: $y = (W \odot m)x$, where W represents the weights of the layer, m represents the binary mask, and x represents the input to the layer.

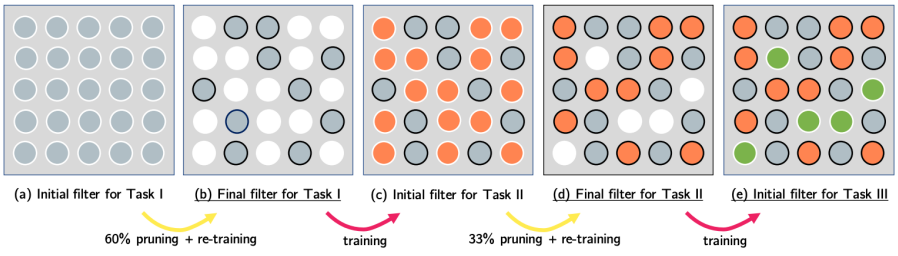


The training step consists in: binarizing the learned floting-point mask m^r with the thresholding function. Computing the forward pass with the masked layer, then calculating the gradient of the mask and finally execute a SGD step. Even though the hard thresholding function is nondifferentiable (can't backpropagate thru that), the gradients of the thresholded mask values m serve as a noisy estimator of the gradients of the real-valued mask weights m^r

Masking with Pruning Methods

We can use pruning methods to find a mask. **Magnitude Pruning** involves training a network and cut the lowest $p\%$ weights in a layer sorted by their absolute magnitude. A variation is Iterative Magnitude Pruning (IMP), where the process is repeated multiple times, each time pruning $p\%$ and retraining.
PackNet is a method for neural network optimization that uses magnitude pruning in a task-aware manner. This is a technique to reduce the size of a neural network by **removing weights** (setting them to zero) that have the smallest absolute value and are deemed **least important**. PackNet uses for **inference** the task labels to choose the respective mask. The output of a layer is given by $y = (W \odot m)x$, where W represents the weights, m is a binary mask, and x is the input to the layer.
Training Process starts from a pretrained model and for each new task:

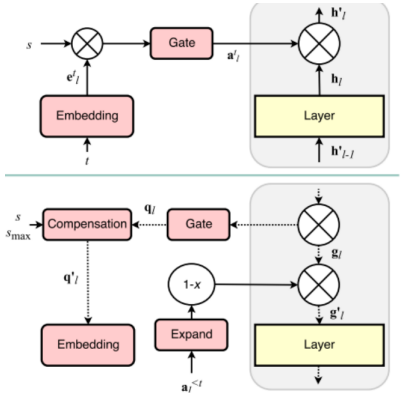
- Fine-tune: the network fine-tunes the weights without a mask (all weights are available for updates).
- Pruning: after fine-tuning, a fraction of the weights (the least important ones) are pruned.
- Retrain: the network is then retrained to regain any lost accuracy due to pruning.
- Freeze: the task-specific parameters (masks) are frozen to preserve knowledge for that task while the pruned weights can be used to learn a new task.



The method is 1.5 times more expensive than simple fine-tuning, presumably due to the additional steps of pruning and retraining. PackNet allows for forward transfer, which means knowledge from previous tasks can positively influence learning new tasks because the weights are actively trained and then selectively pruned. This contrasts with PiggyBack, where the weights are not trained but rather selected through masks and remain fixed, potentially leading to less adaptability and transfer learning capability.
In essence, PackNet is designed to selectively use parts of the network for different tasks by carefully pruning and retraining weights, aiming to achieve compact models without forgetting previous knowledge while still allowing the benefits of transfer learning.

Hard Attention to the Task

The concept of **Hard Attention to the Task** (HAT) involves learning a soft attention mask for the units of a neural network for each specific task. Instead of using a fixed binary mask like some other methods, HAT proposes a learnable **soft attention** mechanism that **determines** which units (neurons) of the network should be **active** for a given task. This mask effectively scales the activations of the units during the forward pass, enabling task-specific responses from the network. The method is **task-aware**, meaning the masks are specific to each task and the network is aware of the task being performed at any given time.
Contrary to masking weights directly, **HAT focuses on masking neurons** (units). Each neuron's activation is modulated by the **learned soft attention mask**, effectively **gating** (scaling) **the neuron's contribution to the next layer**. The soft attention masks are also applied during backpropagation. This means that gradients are modulated by the mask, which can be seen as a form of weight sharing where the mask influences which weights receive updates during training. This process helps in selectively updating parts of the network relevant to the current task, while minimizing interference with what has been learned for previous tasks.



Forward (top) and backward (bottom)

Forward Pass (mask units):

- The embedding e_l^t for a given task t is input to the gate, along with a scaling factor s.
- This gate computes attention a_t which is applied element-wise to the output h_l from the previous layer l.
- The result is a modulated signal that selectively activates certain neurons based on the task, passing this task-specific signal to the next layer.

Backward Pass (mask gradients):

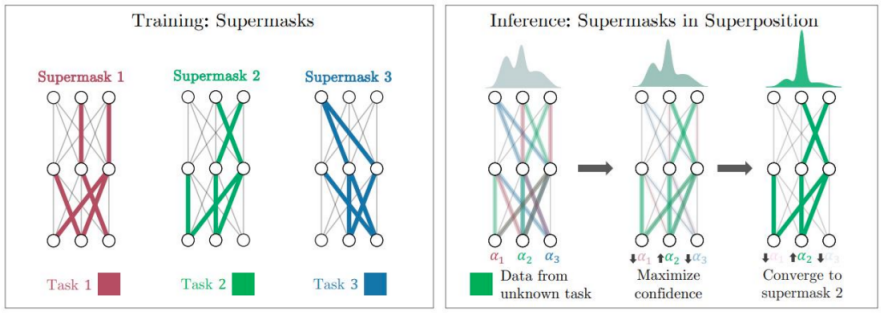
- The gate's parameters are modified following the gradients.
- During the backward pass, the gradient g_l is modulated to preserve the information learned in previous tasks upon learning a new task, we condition the gradients according to the cumulative attention from all the previous task. The cumulative attention vector is expanded to match the

dimensionality needed for gating. The cumulative attention vector keeps track of how much each unit is used by the previous task. Used units should be protected from large changes.

In both passes, the network uses task-specific embeddings to modulate the activations and gradients, which helps the model to focus on different parts of the network for different tasks. This selective focusing prevents interference between tasks and promotes better retention of previously learned tasks while learning new ones.

Supermasks in Superposition

Supermasks in Superposition (SupSup) is a method that deals with training and inference in neural networks, leveraging the concepts of **fixed network** backbones and **binary masks**, along with the principle that we can find good binary masks even from random weights (Lottery Ticket Hypothesis). SupSup has the ability to use **random weights** without the need to store them explicitly. Instead, a **random seed can be saved**, and the weights can be regenerated as needed. This saves on storage and computation. For inference, instead of a single mask for each task, **SupSup utilizes a weighted sum of multiple masks** (superposition). This means that for any given input, the network can apply a combination of masks to produce the output.
This approach can be advantageous. The network **does not need task labels** during training or inference. It can perform task inference, i.e., the network can predict which task it should perform based on the input data. The system is potentially more memory-efficient, as it uses a base model and dynamically generates weights, avoiding the need to store them.



The diagrams show the process of training different supermasks (left) for different tasks and how these supermasks can be combined at inference time to handle data from unknown tasks or to optimize for task identity. This is done by maximizing the confidence of the output through gradient ascent on the weighted sum of the supermasks (right).

Following training using the Edge-Popup algorithm (not covered in the course), **we obtain a distinct mask or model for each task**. In a scenario where tasks are not specified beforehand, the challenge lies in determining the appropriate task label for inference.

A supermask in **superposition** is a weighted sum of all the masks. This refers to combining all the binary masks associated with different tasks into a single mask by calculating a weighted sum. This superimposed mask is then applied to the network's weights.
The SupSup procedure for **task identification** inference is as follows: first we associate each of the k learned supermasks M_i with a coefficient $\alpha_i \in [0, 1]$, initially set to $1/k$. Each α_i can be interpreted as the "belief" that supermask M_i is the correct mask (equivalently the belief that the current unknown task is task i). The model's output is then be computed with a weighted superposition of all learned masks.

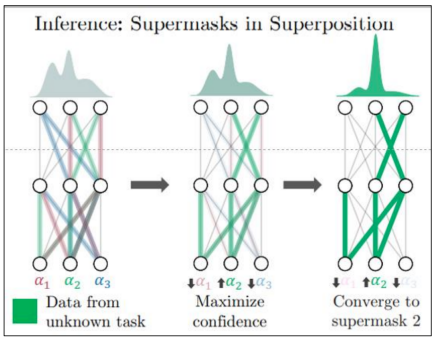
$$\mathbf{p}(\alpha) = f\left(\mathbf{x}, \mathbf{W} \odot \left(\sum_{i=1}^{k} \alpha_i M^i\right)\right) \tag{3.26}$$

This process approximates what an ensemble of models would output, but it's done in a single forward pass through the network, which is computationally efficient. The correct mask M_j should produce a confident, low-entropy output. Therefore, to recover the correct mask we find the coefficients α which

minimize the output entropy of $\mathbf{p}(\alpha)$. The task is inferred using a single gradient, **One-shot Task Inference**. Specifically, the inferred task is given by:

$$\arg\max_i \left(-\frac{\partial \mathcal{H}\left(\mathbf{p}(\boldsymbol{\alpha})\right)}{\partial \alpha_i} \right) \tag{3.27}$$

In other words it selects the mask that maximizes the confidence of the output, effectively optimizing the mask to select for a given task. More gradient steps can be performed but one step is usually sufficient.



Initially, there's uncertainty about the task (1). Through gradient optimization, the model maximizes confidence in a task prediction (2). Finally, the model converges to a supermask associated with a specific task, effectively selecting the correct task to perform for a given input (3).

In Task-agnostic **training** the task boundaries are not known beforehand, we need a mechanism to decide when to start training a new mask. The model leverages the superposition of masks and entropy to decide when to start training a new mask. Based on α of the superposition, if certain coefficients dominate, it suggests the data likely belongs to a previously learned task. Conversely, if the coefficients are uniform or non-dominant, it may indicate a new task.
The algorithm includes the following steps:

- Compute $v = softmax(-\nabla_\alpha H(p(\alpha)))$, where $\nabla_\alpha H(p(\alpha))$ is the gradient of the entropy with respect to α, and $softmax$ is applied to these gradients. This step essentially ranks the masks based on their relevance to the incoming data.
- Check if the maximum v_i is below a certain threshold $\frac{1+\epsilon}{k}$ where ϵ is a small constant and k is the total number of tasks. If so, create a new mask for a new task.
- Otherwise, continue using the mask with the maximum v_i, which corresponds to the most relevant existing task.

Recap

The following is an **overview** of different methods for task transfer, detailing how learning one task can influence the learning of subsequent tasks, positively or negatively, highlighting their potential for forward and backward transfer, risk of forgetting, and their efficiency in terms of space occupation.

There are methods using **fixed weights and independent masks**, such as Piggyback and SupSup. These approaches do not facilitate forward transfer the ability to leverage previous learning to improve performance on subsequent tasks. However, they prevent forgetting, meaning that learning new tasks does not interfere with the performance on previously learned tasks. They are highly space-efficient as they act like an ensemble of models, where each model is lightweight and specific to a task.

There's an approach using **trainable weights with hard masks**, exemplified by Packnet. This allows for forward transfer since the new masks can incorporate units that were important in previous tasks. The backbone of the network is fixed at initialization, leading to no forgetting of old tasks. This method is more space-efficient than Progressive Neural Networks (PNN) but operates on similar principles.

Methods using **trainable weights with soft masks**, like Hard Attention to the Task (HAT), enable both forward and potential backward transfer (where learning new tasks can interfere with previous knowledge). However, there is a risk of forgetting as previously used units are not frozen. Soft masks are generally more challenging to train and require more space (e.g. 32 bits versus 1 bit for hard gates before compression).

Applications, Frontiers, Research in Continual Learning

4.1 Active Learning and Curriculum Learning

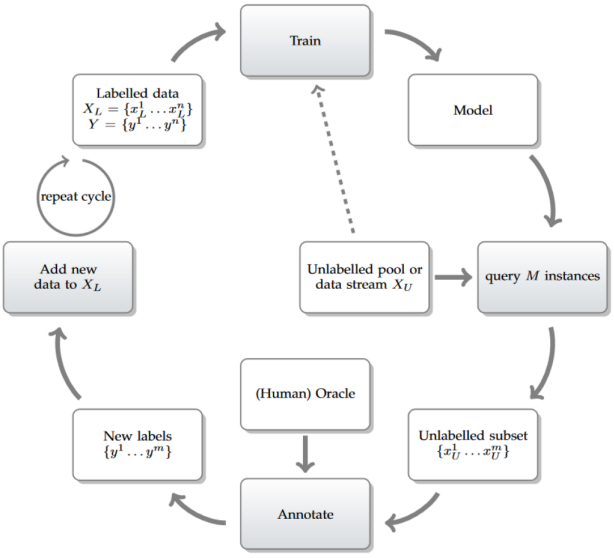
Active learning involves selecting for annotation the next sample from a large unlabeled dataset, whereas **curriculum learning** focuses on determining the optimal sampling order within a dataset to facilitate effective learning.
Until now, our assumption has been that we receive data passively, without any control, but this is often not the case in practical applications. Data collection and **annotation** are ongoing processes, and old data is retained rather than discarded. We have the opportunity to construct a stream or curriculum from a vast dataset to **enhance** training progressively. At each stage, improvement is possible if we can **identify** the **most beneficial data** to collect or annotate, as well as the most suitable data for model training.
Not all samples hold equal importance: some are frequently forgotten (hard samples), while others remain retained (easy samples). Easy samples demonstrate generalizability across various neural architectures, representing a characteristic of the data rather than the specific model. Interestingly, a notable portion of easy examples can be excluded from the training dataset without adversely affecting the final accuracy.

4.1.1 Active Learning

Given a **vast collection of unlabeled data** alongside a **smaller portion labeled**, we train the model with the labeled data and the objective is to enhance the existing model. Which samples are more helpful when annotated? Following **annotation** of new data, the model undergoes retraining using both the freshly labeled samples and existing data. The assumptions made for active learning are the following:

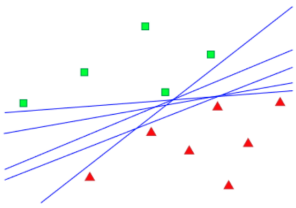
- We have the entire (unlabeled) dataset available from the beginning.
- We ignore computational cost, unlike continual learning methods, we reuse all the data seen up to now. It is a cumulative training on all the data.
- The process involves a perfect oracle (annotator). In real-world scenarios, human annotators are imperfect, making errors. Moreover, the samples that would benefit the most from annotation might also be the most challenging (and thus costly) to annotate.

The **acquisition function** is the one responsible for selecting and labeling samples. It operates based on inputs such as the current model, the pool of unlabeled samples, and the current set of labeled samples. The output is the chosen samples from the unlabeled dataset.



Version Space Reduction

This method involves representing the entire subspace of hypothesis consistent with your data (version space) and refine it with the new data. Learning with Version Spaces is restricting the space to solutions consistent with the training data. Picking the samples with more disagreement allows to find a smaller version space.



Version space examples for a linear classifier. All hypothesis are consistent with the data but each represent a different model in the version space

Heuristics

These methods design good measures to pick interesting samples.

Uncertainty sampling works in the following way: after having created an initial classifier:

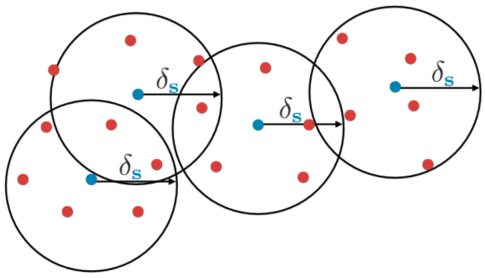
1. apply the current classifier to each unlabeled example;
2. find the example for which the classifier is less certain;
3. have the teacher label those samples;
4. train a new classifier on all labeled examples.

So this methods queries the samples with largest entropy (low entropy: $[0.1, 0.9]$, high entropy: $[0.4, 0.6]$).
In considering which samples to select for annotation, we encounter a **heuristic tradeoff**. The goal is to identify the most uncertain samples, which typically fall into several categories: hard samples, which are beneficial for labeling; noise, which is uninformative; and outliers, which may have some utility but are often not the optimal choice. The concept suggests that the **ideal samples** for annotation strike a balance: they should be **challenging** enough to provide valuable information **but not so difficult** that they are likely to be noise or outliers. This approach embodies an exploration/exploitation

tradeoff, recognizing that an excessive focus on novelty might not always yield the best results, as the most novel data may not accurately represent the expected data.
A model is **calibrated** if its probabilities are calibrated. That is, if a model predicts a class with probability p, then the prediction is correct $\%p$ of the time. Entropy-based heuristics assume that the output probabilities are correct and calibrated but in general, DNNs are not calibrated: during training, we push the probabilities to 0 or 1, without any care about calibration, the difference between $p = 0.02$ and $p = 0.1$ is probably not significant, but it makes a big difference in entropy. Entropy is sensitive to large number of classes and very low probabilities: uncalibrated models with a large number of classes will have unreliable probability values for most classes.
Best-versus-Second-Best in an alternative measure of uncertainty. It measures the difference between highest probability and second highest. It is more robust to high number of classes and smaller values for unimportant classes.

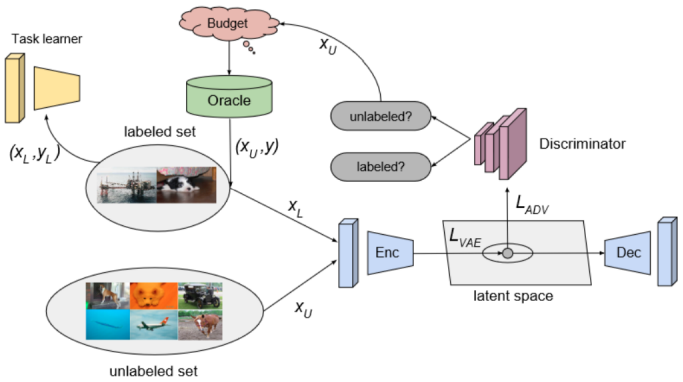
Coresets

Uncertainty estimates are not robust against outliers, noise and may be inaccurate at the beginning of training. A different approach is try to annotate diverse samples.
A **Coreset** refers to a **smaller, distilled representation of a dataset** that ideally **captures the essential patterns or distribution** of the larger dataset. The goal is to select samples (points) such that the maximal distance between any unselected sample and the nearest selected sample (center) is minimized. The concept of Coresets in Active Learning, focuses on selecting a representative subset of samples from a larger dataset for training machine learning models, in a way that is both **unsupervised and efficient**. This approach is unsupervised, meaning it relies solely on the spatial relationships (distances) between data points, without using any labels or additional information. This makes it particularly useful for scenarios where labeled data is scarce or labeling is costly.
This is similar to a common problem in operations research known as the p-center problem, where the objective is to place a given number of facilities (or warehouses) in such a way that the maximum distance from any city (or data point) to the nearest facility is minimized. This ensures that all cities are adequately serviced by the nearest facility.
It is an NP-hard problem but an approximate solution with a greedy algorithm can be found: given the current pool, find the most distant point and add it to the pool. Repeat until you have enough samples.



Variation Adversarial Active Learning

Active learning aims to develop **label-efficient algorithms** by sampling the most representative queries to be labeled by an oracle. VAAL is a pool-based semi-supervised active learning algorithm that implicitly learns this sampling mechanism in an adversarial manner. The method learns a **latent space** using a **variational autoencoder** (VAE) and an **adversarial network** trained to **discriminate** between **unlabeled and labeled data**. The minimax game between the VAE and the adversarial network is played such that while the VAE tries to trick the adversarial network into predicting that all data points are from the labeled pool, the adversarial network learns how to discriminate between dissimilarities in the latent space.



The model learns the distribution of labeled data in a latent space using a VAE optimized using both reconstruction and adversarial losses. A binary classifier predicts unlabeled examples and sends them to an oracle for annotations. The VAE is trained to fool the adversarial network to believe that all the examples are from the labeled data while the adversarial classifier is trained to differentiate labeled from unlabeled samples.

The Variation Adversarial Active Learning (VAAL) loss function incorporates components from Variational Autoencoders (VAEs) and adversarial training to effectively utilize both labeled and unlabeled data.
The VAE learns to encode both labeled and unlabeled data in the same latent space. The **VAE Loss** component is designed to minimize the difference between the reconstructed data and the actual data while also encouraging the encoded representations to approximate a prior distribution.

$$\begin{aligned}\mathcal{L}_{VAE}^{trd} = \mathbb{E}[\log p_\theta(x_L|z_L)] - \beta D_{KL}(q_\phi(z_L|x_L)\|p(z)) \\ +\mathbb{E}[\log p_\theta(x_U|z_U)] - \beta D_{KL}(q_\phi(z_U|x_U)\|p(z))\end{aligned}$$

where q_ϕ and p_θ are respectively the encoder and the decoder and $p(z)$ is the prior. This is a reconstruction loss that ensures that the data can be effectively reconstructed from the latent variables. It is represented as the expected log-likelihood of observing the data given the latent variables. And a KL divergence, a regularization term that measures the difference between the learned distribution of the latent variables and the prior distribution, typically assumed to follow a Gaussian (we don't want a distribution too much different from a Gaussian).
The **VAE Adversarial Loss** uses a discriminator that tries to distinguish between the latent representations of labeled and unlabeled data. The VAE tries to "fool" the discriminator by making these representations indistinguishable. The Generator's Adversarial Loss encourages the encoder to produce latent variables that the discriminator cannot easily classify as coming from labeled or unlabeled data

$$\mathcal{L}_{VAE}^{adv} = -\mathbb{E}[\log D(q_\phi(z_L|x_L))] - \mathbb{E}[\log D(q_\phi(z_U|x_U))] \tag{4.1}$$

The total VAE loss combines the past two equation, typically with weighting factors to balance their influence.

$$\mathcal{L}_{VAE} = \lambda_1 \mathcal{L}_{VAE}^{trd} + \lambda_2 \mathcal{L}_{VAE}^{adv} \tag{4.2}$$

Finally the **Discriminator Loss** ($\mathcal{L}_D$) trains the discriminator to correctly identify whether a latent variable comes from the labeled or unlabeled dataset.

$$\mathcal{L}_D = -\mathbb{E}[\log D(q_\phi(z_L|x_L))] - \mathbb{E}[\log(1 - D(q_\phi(z_U|x_U)))] \tag{4.3}$$

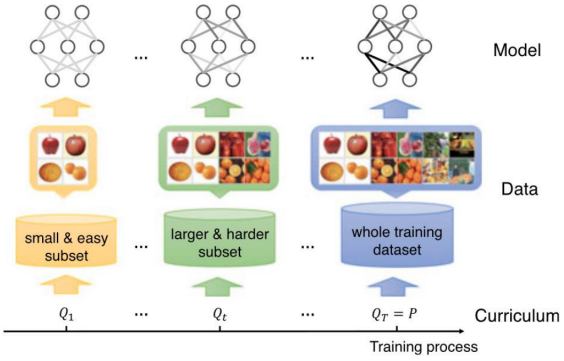
The **sampling** strategy used in Variational Adversarial Active Learning (VAAL) efficiently select samples from an unlabeled dataset for labeling (annotation) by an oracle. The process utilizes a discriminator that assesses the difficulty of samples based on how well it can distinguish between labeled and unlabeled data. VAAL aims to choose samples from the unlabeled pool that are most informative for training, based on the assumption that the most uncertain samples (as judged by the discriminator) will provide the most value when labeled. The steps involved are the following:

1. From the unlabeled dataset, identify a subset of samples that have the lowest confidence scores from the discriminator. These scores reflect the discriminator's uncertainty about whether samples are from the labeled or unlabeled datasets.
2. The selected samples are then presented to an oracle (an expert or automated system) to obtain the true labels for these samples.
3. Incorporate the newly labeled samples into the labeled dataset, effectively expanding it with additional data points that are likely to be informative for model training.
4. The samples that have now been labeled are removed from the unlabeled pool to prevent them from being selected again in future iterations.
5. Output the updated sets of labeled and unlabeled data, ready for further processing or additional rounds of sampling and labeling.

This procedure helps in refining the training dataset by adding only those samples that are deemed most valuable for improving model performance, thus optimizing the use of labeling resources.

4.1.2 Curriculum Learning

Humans typically learn in curricula, **gradually progressing from easier to more challenging material**. This prompts the question: can deep neural networks (DNNs) similarly benefit from a curated learning progression? Yes, but not always. The concept of curriculum learning involves **arranging data into experiences based on their difficulty levels**. There's two primary objectives: to improve the final accuracy of the model and to expedite the convergence process. By structuring the learning process in this manner, DNNs can potentially achieve more effective and efficient learning outcomes.



Definition

A curriculum is a sequence of training criteria over T training steps: $\mathcal{C} = \langle Q_1, ..., Q_t, ..., Q_T \rangle$. Each criterion Q_t adjusts the weighting of the training samples based on a function of their inherent characteristics. This function, represented as $Q_t(z) \propto W_t(z)P(z)$, emphasizes different subsets of the data at different training stages by reweighting the underlying distribution $P(z)$ of the training dataset D. The entropy of these distributions is intended to increase over time $H(Q_t) < H(Q_{t+1})$. This means that the training starts with less diverse or simpler examples, and gradually introduces more complex or diverse examples. The increase in entropy corresponds to a gradual introduction of more challenging or varied training data.
The weight assigned to each sample z increases over time $W_t(z) < W_{t+1}(z)$, indicating that the training set effectively grows, incorporating more of the dataset as the training progresses. Early stages might focus on a subset of the data, while later stages expand to include a broader range of examples.
The process culminates with $Q_T(z) = P(z)$, where the final weighting aligns with the true distribution of the dataset. This implies that the training eventually encompasses the full complexity and variability of the data.

Design

In curriculum learning, a crucial component is the **Difficulty Measurer**, responsible for determining the difficulty level of each sample. Unlike data-driven decisions, this aspect is manually crafted, assessing various factors like structural complexity and diversity within the samples. For instance, sentence length can indicate complexity, while the presence of rare words or high entropy signifies diversity. Furthermore, noise estimation plays a role, where cleaner data tends to be easier to learn from. An example of this is how different data collection methods may yield varying levels of noise within the dataset.
Another component is the **Training Scheduler** that decides the sequence of data subsets. It can be discrete: update data after a fixed number of epochs or convergence on the current subset; or continuous: update data after each epoch.

Self-Paced-Learning

Self-Paced-Learning is an **automatic curriculum** where the students measures the **difficulty according to its loss**. It operates on the concept of gradually increasing the difficulty level of training examples over time. At each step, the model focuses solely on the $p\%$ easiest samples, determined by their loss values. As training progresses, the percentage of data used for training increases incrementally. This process is automatic: the model itself evaluates the difficulty of each sample, embedding the curriculum design directly into the learning objective.
Self-Paced Learning is a training strategy that modifies the learning process based on the complexity or difficulty of training samples. It uses a weighted sum of losses, where each sample's contribution to the overall training loss is adjusted by a weight that reflects its relative ease or difficulty for the model at that stage in training. The overall learning objective is to minimize a combination of the weighted losses and the regularizer, formally:

$$\min_{w, v \in [0,1]^N} \mathbb{E}(w, v; \lambda) \left(\sum_{i=1}^{N} v_i l_i + g(v; \lambda) \right) \tag{4.4}$$

The Loss per Sample (l_i) represents the loss for each individual sample, indicating how well or poorly the model is currently performing on that specific example. The weights (v) are vectors for each sample that adjusts the impact of each sample's loss on the overall training process. The weights are determined by the self-paced regularizer.
The Self-Paced Regularizer ($g(v; \lambda)$) is a function that adjusts the weights based on a parameter λ (often referred to as the age coefficient). It is designed to give higher weights to "easier" samples (those with a loss less than λ) and zero weight to "harder" samples (those with a loss greater than λ). The function is given by:

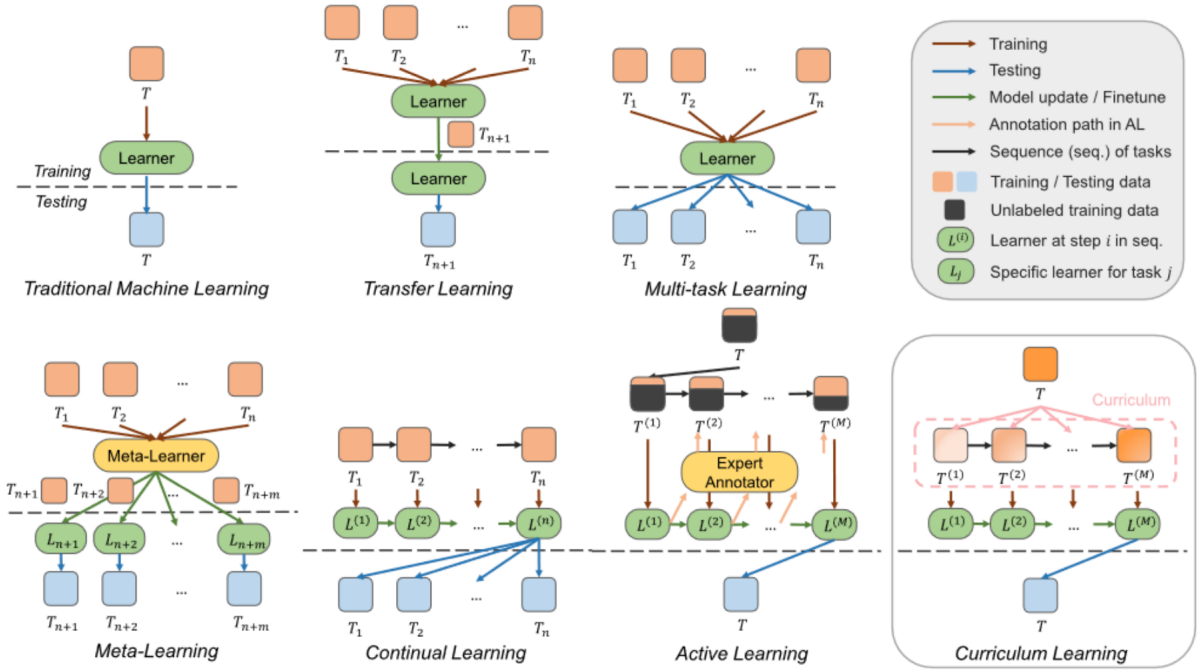
$$g(v; \lambda) = -\lambda \sum_{i=1}^{N} v_i \tag{4.5}$$

The training involves an alternated optimization of weights v and model parameters w. The weights v are optimized keeping w fixed using the criterion $v_i^* = \text{argmin}_{v_i \in [0,1]} v_i l_i + g(v_i; \lambda)$. This part of the process is convex, allowing for an optimal solution where v_i is set to 1 if $l_i < \lambda$ (easy samples) and 0 otherwise (hard samples). After updating v, the model parameters w are updated (typically using SGD) keeping v fixed $w^* = \text{argmin}_w \sum v_i l_i$.

Miscellanea

Sometimes, the anti-curriculum (hard-to-easy) is better.
Other automatic curriculum methods include Transfer Teacher, where a robust pretrained teacher model evaluates the difficulty of samples. Another method is RL Teacher, which formulates curriculum design as a reinforcement learning challenge, utilizing feedback from the student model as the reward signal.

4.2 Overview



4.3 Out of Distribution Detection and Open World

4.3.1 Out of Distribution Detection

In real-world scenarios, it's common to encounter training data that doesn't fully represent the entire test distribution. Consequently, we can't always rely on the model's predictions for **out-of-distribution** (OOD) samples. Therefore, it's crucial for the model to **accurately assess the uncertainty associated with both the data and its predictions**. This is even more important in safety-critical domains: a self-driving car should (safely) stop in uncertain conditions and give control to the passenger, medical systems should give uncertainty estimate so that an expert can determine the correct course of action...

Anomaly Detection involves discerning between normal and abnormal behaviors, such as distinguishing between legitimate emails and spam or identifying potential network intrusions. Typically, this is framed as a highly imbalanced binary classification task.
On the other hand, **OOD Detection** focuses on identifying shifts between training and testing data and rejecting uncertain predictions, even when the nature of uncertainty is not precisely known.

Open Set Recognition is a domain of machine learning that contrasts the traditional closed-world scenarios like multi-class classification, face verification, and detection. In closed-world settings, all possible classes are known during the training phase, such as identifying predefined classes or verifying known identities. Open Set Recognition, however, deals with scenarios where the model might encounter completely unseen classes during testing, which is a common situation in real-world applications. This necessitates models that can identify when a sample does not belong to any trained category and handle such unknowns effectively. It is important to develop systems that are robust and adaptable to new and unforeseen data, moving beyond the limitations of traditional models that operate under the assumption that all test scenarios are known beforehand.

4.3.2 Uncertainty

In statistics, dispersion is a property of a distribution measuring how "stretched" it is, some measures are variance, standard deviation, IQR, etc.

Aleatoric Uncertainty pertains to the inherent uncertainty in the data generating process, including factors like the distribution of the data and noise in the measurements. It is considered irreducible, meaning it cannot be eliminated even with more data or a better model. In contrast, **Epistemic Uncertainty** relates to uncertainty in the model itself, indicating how uncertain the model is in making predictions. This uncertainty can be reduced with additional data, implying that it is more dependent on the model's capacity and the quality of the training data.

ODIN

The approach to detecting out-of-distribution (OOD) examples involves observing a pattern: correctly classified examples typically exhibit higher maximum softmax probabilities compared to erroneously classified and out-of-distribution examples. A basic solution is sorting the data based on their maximum softmax probabilities and then employing a threshold to distinguish between in-distribution (ID) and out-of-distribution (OOD) examples. However, it's important to note that this method is effective only in relatively simple scenarios.

ODIN is a method for enhancing the identification of in-distribution (ID) versus out-of-distribution (OOD) samples in neural networks. ODIN leverages temperature scaling and small perturbations to the input to effectively distinguish between ID and OOD images. The idea is that **temperature scaling** and **small perturbations** to the input can separate the softmax score distributions between in and out of distribution images making easier to tell them apart (amplify the differnce). Increasing the temperature results in a more uniform distribution of softmax scores (less confident), while decreasing it makes the distribution more peaked (more confident). Introducing slight changes to the input image can significantly affect the softmax scores. For inputs that are well within the model's learned distribution ("flat regions" of the input space), these perturbations should not heavily alter the softmax probability, maintaining a higher confidence level. However, for OOD samples which are likely atypical to what the model has seen, even slight perturbations can cause large changes in softmax scores, resulting in lower confidence predictions.

ODIN is straightforward to implement as it can be added to any pretrained supervised model without needing substantial modifications. Its simplicity and the use of existing model structures without requiring retraining or additional annotations make it a practical choice for improving model robustness against OOD samples. These technique improve the model's ability to discern and correctly classify new, unseen data types that it was not explicitly trained on, addressing the challenge of open set recognition where the model needs to handle unknown classes during testing.

The input image x is **perturbed** slightly to generate a new input $\tilde{x}$. This perturbation is directed by the gradient of the negative log of the softmax score $S_y(x;T)$ with respect to the input x. The perturbation aims to **push the input slightly away from the high-probability** regions of the input space, making the model's output more sensitive to outliers or novel inputs. This is formally given by:

$$\tilde{x} = x - \epsilon \cdot \text{sign}\left(-\nabla_x \log S_y(x;T)\right) \tag{4.6}$$

where ϵ is a small scalar that controls the magnitude of the perturbation.

The softmax scores are adjusted using a **temperature** scaling factor T, which modifies the sharpness of the distribution. A high value of T makes the output distribution flatter (more uniform), while a low T makes it sharper (more peaked). This can help in distinguishing between in-distribution and OOD samples by altering the confidence level of the predictions. The softmax function with temperature scaling is defined as:

$$S_i(x;T) = \frac{\exp(f_i(x)/T)}{\sum_{j=1}^{N} \exp(f_j(x)/T)} \tag{4.7}$$

where $f_i(x)$ are the logits output by the model for class i, and N is the total number of classes.

To determine whether an input is in-distribution or out-of-distribution, a threshold δ is applied to the maximum softmax score after perturbation and temperature scaling (**Detector**):

$$g(x;\delta,T,\epsilon) = \begin{cases} 1 & \text{if } S_{\hat{y}}(\tilde{x};T) \leq \delta \\ 0 & \text{if } S_{\hat{y}}(\tilde{x};T) > \delta \end{cases} \tag{4.8}$$

Here, $S_{\hat{y}}(x;T)$ represents the maximum softmax score across all classes for input x at temperature T. If the score is below a predefined threshold δ, the input is classified as OOD; otherwise, it is considered in-distribution.

Deep Ensembles

Deep neural networks (DNNs) often exhibit overconfidence in their predictions, a trait shared by both supervised and generative models. While Bayesian models offer a promising approach for estimating uncertainty, they are typically computationally expensive and require heavy approximations. A simpler alternative is the use of **ensembles**, where multiple models are trained independently and their predictions are combined to provide a more reliable estimation of uncertainty.
The ensemble utilizes a proper **scoring rule** as the training criterion. This means that each model in the ensemble is trained to optimize a metric that accurately reflects the quality of the probability distributions it outputs, ensuring that the model's predictions are not only accurate but also well-calibrated. To further enhance the robustness of the models, **adversarial training** is employed. This involves modifying the input data slightly to create adversarial examples, which are then used in training. This method helps in smoothing the predictive distribution, making the models less sensitive to small perturbations in the input data, thus improving generalization across unseen data. **Multiple models** are trained independently. This **diversifies** the perspectives of the models on the data, since each model starts with different initial weights and may train on slightly different variations of the data due to random shuffling. After training, the **predictions** of all models are **combined**, typically by averaging, to produce a **final output**. This combination helps in reducing variance and provides a more reliable and stable prediction.
The **training algorithm** is outlined as follows:

- Initialize each model with random parameters.
- For each model, sample a data point and generate its adversarial example.
- Update the model by minimizing the loss on both the actual data point and its adversarial counterpart.
- Repeat this for all models, typically in parallel to speed up the process.

4.3.3 Open World

Knowing in an Open World

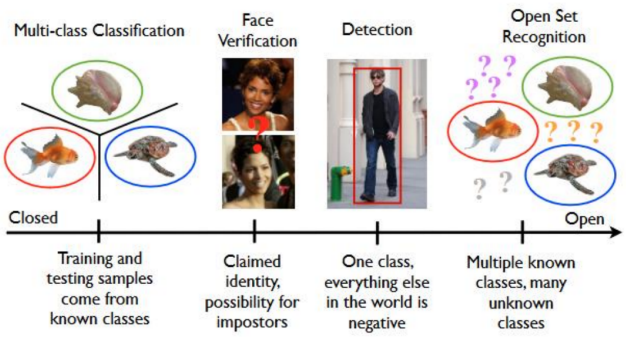
Understanding uncertainty in an open world scenario involves categorizing samples based on their confidence levels and whether they belong to known or unknown distributions.

- **Known**: in-distribution samples that are predicted correctly with high confidence (correct predictions).
- **Known Unknowns**: low-confidence samples, such as successfully recognized anomalies (low confidence).
- **Unknown Unknowns**: out-of-distribution samples with highly confident predictions but the model shouldn't really have high confidence here. In general, when the test distribution drifts, the model may encounter entirely new scenarios. Effectively, the model doesn't know what it doesn't know.

Up until now we didn't expect the model to recognize unknown unknowns, this was consistent with the closed-world assumption applied to both continual learning and metalearning. A new problem arises: how do we identify unknown unknowns?

Prior Knowledge

In the **Prior Knowledge** approach the idea is train the model to recognize a **background** class. In the context of an open-world scenario, a background class is a category designed to encompass all examples that do not belong to any known classes or categories. It acts as a **catch-all** for instances that are not explicitly represented in the model's training data or known classes. By training the model to recognize this background class, it can learn to distinguish between familiar categories and any novel or unexpected inputs that fall outside of those known categories. This converts an open world problem into a closed world one.

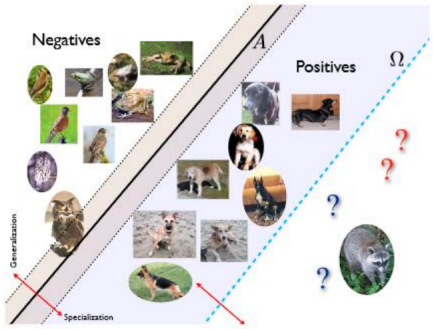


Vision problems arranged in order of *openness*. For some problems, we do not have knowledge of the entire set of possible classes during training and must account for unknowns during testing. In this article, we develop a deeper understanding of those open cases

The **magnitudes of features** for **unknown samples** in deep feature space is **lower** than those of **known samples**. To address this, methods like the Objectosphere loss are employed, which explicitly optimize this objective. In practice, known samples are expected to have feature magnitudes above a specified minimum, indicating high confidence in their classification. In contrast, background samples, representing unknown or novel instances, are expected to have feature magnitudes close to zero, reflecting the uncertainty surrounding their classification status. This approach helps the model differentiate between known and unknown samples based on the magnitudes of their features in deep feature space.
The limitations here is that we can train the model to recognize the unknown only because it is known: we have the background class. Unfortunately, this is not always the case.

Open Set Recognition

All the classification models that we used have unbounded decision boundaries: all points on one side of a linear decision boundary are positive, the others negative regardless of how far they are from the boundary or the training data distribution. In an open world setting, we need to allow a reject option.



A solution might be to use **distance-based rejection** method: reject samples that are too far based on a metric. This involves changing the softmax with a distance-based classifier and threshold the maximum distance. Mahalanobis Distance can be used, a measure of how many standard deviations of distance between the point and the class mean.

4.4 Distributed Learning

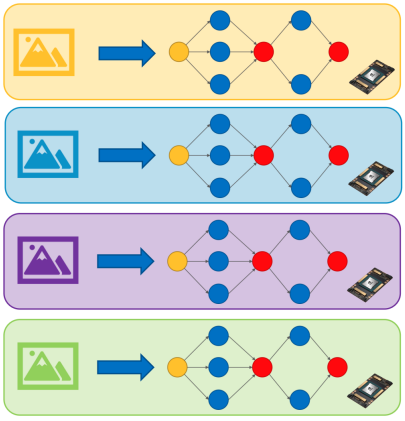
OpenAI's GPT-3 has 175 billion parameters, and training it on its vast dataset would required 3 million GPU hours. The objective of distributed learning is to **scale up and go faster**.

4.4.1 Data Parallel Distributed Learning

Data Parallelism includes:

- Replicate the model across all the available devices.
- Split the data evenly.
- Replica of the model trains on the local data independently

The problem here is that we don't want multiple replicas as output of the training process, but a single model.

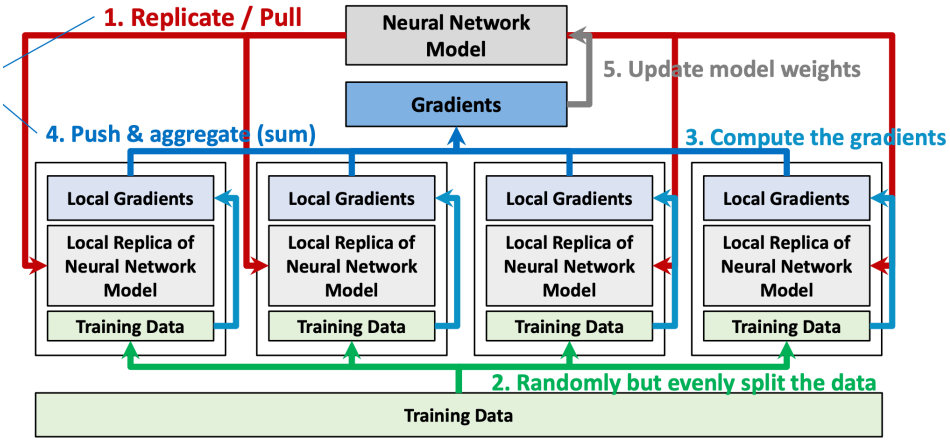


Parameter Server Strategy

Parameter Server strategy requires the nodes to be split in two main roles:

- Parameter Server: the central controller of the whole training process, receives the gradients from workers and sends back the aggregated results.
- Worker Nodes: the hardware accelerators and dataset storage, compute the gradients using splitted dataset and send to parameter server

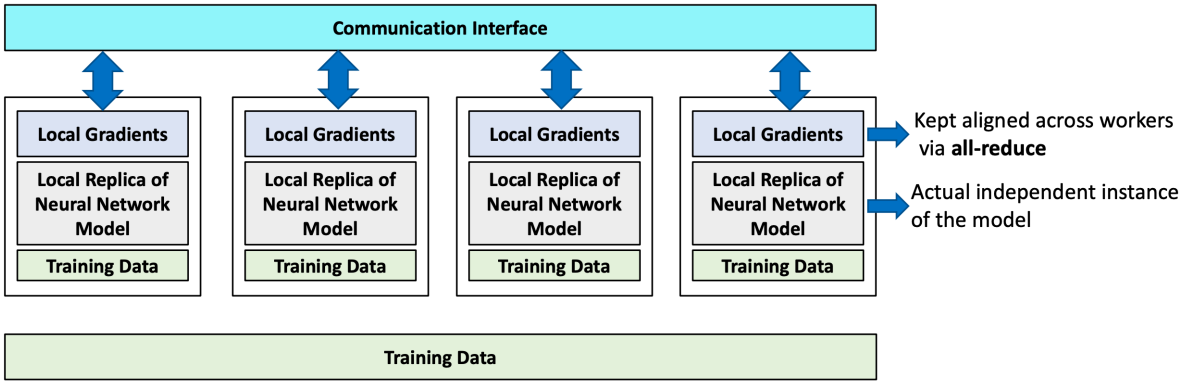
The workflow is as follows:



The problem is that the synchronization steps occur between steps 1 and 4, meaning a single central server handles this load, leading to overload. We need to get rid of the centralized server.

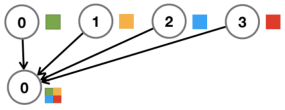
Mirroring Strategy

The infrastructure is this one:

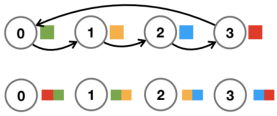


Mirroring strategy in different way:

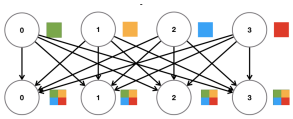
- Naive All-Reduce: each step performs a single reduce operation (Time $O(N)$, Bandwidth $O(N)$).



- Ring All-Reduce: each step performs a single send and merge (Time $O(N)$, Bandwidth $O(1)$).



- Parallel All-Reduce: perform all reduce operations simultaneously (Time $O(1)$, Bandwidth $O(N^2)$).



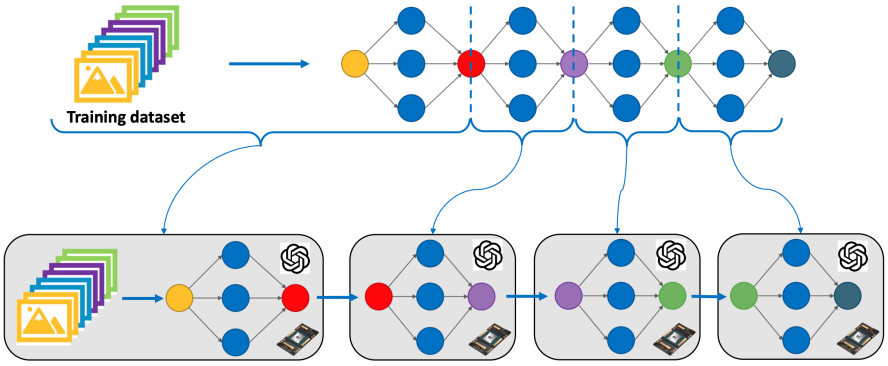
The best of parallel and ring all-reduce: recursive halving all-reduce (Time $O(logN)$, Bandwidth $O(1)$).

Data Parallelism and Huge Models

Though data parallelism provides a better device utilization, even the best GPU cannot fit the model into memory! One GPU is still one GPU.

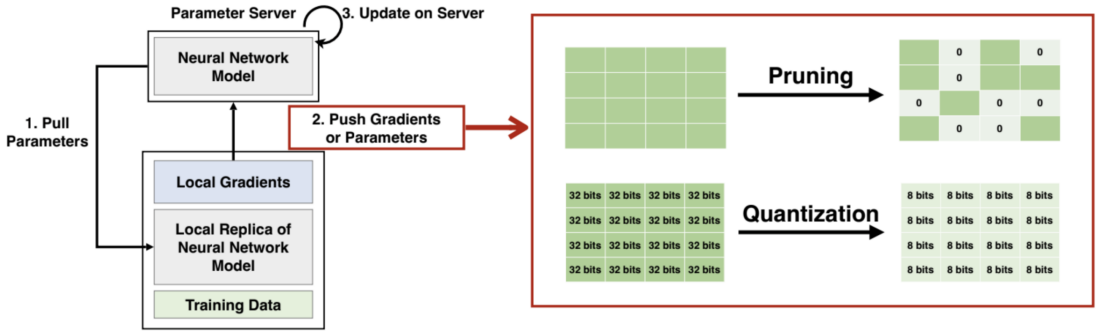
4.4.2 Model Parallel Distributed Learning

It means deploying models across multiple devices: split the model and move activations through devices. Here we have a single copy of the model. It is hard to parallelize because of load balancing issues.



For better utilization we split the data in mini-batches and feed them in stream (pipeline style). Model parameters are not changed during computation within a batch, thus we can pipeline computation and communication.

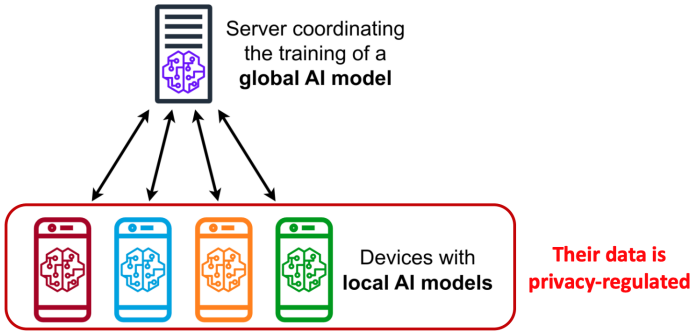
4.4.3 Improving Communication in Distributed Learning



Gradient Pruning: involves selectively removing less important gradients (i.e., small or redundant updates) during model training to reduce communication overhead and computational cost.
Quantization: is the process of reducing the precision of the model's parameters and gradients, typically converting them from 32-bit floating-point to lower-bit representations (e.g., 8-bit integers), to decrease memory usage and accelerate computations without significantly compromising accuracy.

4.5 Federated Learning

Federated learning is a machine learning setting where multiple entities collaborate in solving a machine learning problem. Each client's raw data is stored locally and not exchanged or transferred; instead, focused updates intended for immediate aggregation are used to achieve the learning objective.



The four pillars of federated learning are the following:

- Statistical Heterogeneity: we cannot make assumptions on how the data are distributed across clients.
- System Heterogeneity: we cannot make assumptions on the resources of the devices which join the federation.
- Communication Efficiency: we are leveraging resources which are deployed across the whole Cloud-Edge Continuum. Efficient communication is crucial.
- Privacy and Security: this represents the explicit goal of Federated Learning. Some algorithms may disclose data, so they require additional mechanisms to comply with this point.

4.5.1 Problem Formualtion

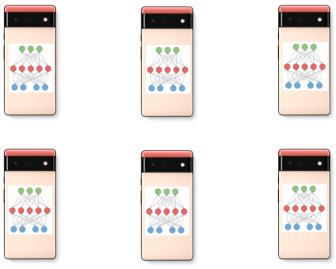
The goal is to minimize a function $F(\mathbf{x})$, defined as the expectation over client distributions $\mathbb{E}_{i\sim\mathcal{P}}\left[F_i(\mathbf{x})\right]$, where $F_i(\mathbf{x})$ represents each client's local objective, itself an expectation over the local data distribution $\mathbb{E}_{\xi\sim\mathcal{D}_i}\left[f_i(\mathbf{x},\xi)\right]$. Client distribution $\mathcal{P}$ denotes the availability and resources of clients, reflecting system heterogeneity, while local data distribution $\mathcal{D}_i$ represents data variation across clients, reflecting statistical heterogeneity. Since these distributions are unobservable, direct minimization of $F(\mathbf{x})$ is infeasible. To approximate the learning problem, Empirical Risk Minimization (ERM) is introduced, defining the objective $F^{ERM}(\mathbf{x})$ as the weighted sum of local ERM objectives $F_i^{ERM}(\mathbf{x})$, with weights p_i reflecting the importance or size of each client's data. $F_i^{ERM}(\mathbf{x})$ is the empirical average of local loss functions. No data points are directly fed to $F^{ERM}(\mathbf{x})$, underscoring the challenge of indirect global model evaluation in federated learning due to data privacy and distribution constraints.

$$F^{ERM}(\mathbf{x}) = \sum_{i=1}^{|C|} p_i F_i^{ERM}(\mathbf{x})$$

$$F_i^{ERM}(\mathbf{x}) = \frac{1}{|D_i|} \sum_{\xi\in\mathcal{D}_i} f_i(\mathbf{x},\xi)$$

4.5.2 Cross-device Federated Learning

Cross-device Federated Learning is a type of data parallelism: replicate the execution model across multiple machines with their own local dataset (map) and aggregate information from their local learning processes (reduce). This results in an insanely prolific data source and computational resource which enables massively parallel learning processes. This method has its drawback: limited bandwidth and computation, occasional availability, data heterogeneity across devices and privacy constraints on local data.



Methods

In **Federated Averaging** (FedAvg) a central server initializes a global model and iteratively improves it through multiple rounds. In each round, a subset of clients is selected to update the model locally using their own data. These local updates are then averaged by the server to create a new global model. This process preserves data privacy as clients do not share their data, only their model updates.

FedOpt is a federated optimization framework where the server coordinates the training of a global model by sampling clients, broadcasting the current model, and aggregating updates. Clients compute local model updates using their data and a specified optimization method (ClientOpt). The server then aggregates these updates and applies its own optimization (ServerOpt) to update the global model, enhancing communication efficiency and ensuring robust client-server optimization.

FedCS (Federated Client Selection) is a method designed to mitigate system heterogeneity in federated learning by optimizing client selection. It maximizes the number of clients $|\mathcal{S}|$ participating in each round while ensuring the total time for a round, including computation, communication, and aggregation, does not exceed a specified deadline. This approach enhances resource utilization and improves overall learning efficiency by dynamically selecting clients based on their availability and capabilities.

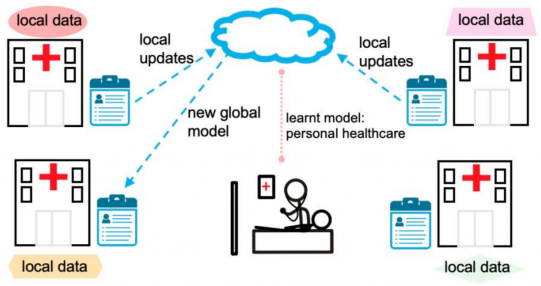
FedProx is a federated optimization framework designed to handle statistical heterogeneity among clients by introducing a proximal term in the local objective. This proximal term helps to prevent local models from diverging too far from the global model. In each training round, a subset of clients is selected to update the global model. Each selected client optimizes its local model by solving a modified objective that includes both the local loss and a penalty term proportional to the distance from the global model. These local updates are then sent back to the server and aggregated to form the new global model. This approach ensures more stable and coordinated updates across diverse client data distributions.

SCAFFOLD is a federated learning optimization method designed to correct the direction of updates of local models by including a control variate term. This term helps mitigate client drift, where local model updates diverge from the global objective due to data heterogeneity. By incorporating correction terms in the update rule, SCAFFOLD improves convergence and alignment of local models with the global model, leading to more efficient and accurate federated learning.

FedDF (Federated Distillation and Fusion) is a federated learning method that enhances model fusion using knowledge distillation. It involves initializing a model fusion process, where an ensemble of client models is used to update a server model iteratively. The server model is refined by minimizing the divergence between the predictions of the client ensemble and the server model on unlabeled data. This approach leverages the collective knowledge of client models to produce a robust and well-generalized global model.

4.5.3 Cross-Silo Federated Learning

Multiple organization collaborate to produce a model without disclosing their data. The key different is that system heterogeneity is a less relevant problem. The participating nodes are orders of dozens.



Vertical Federated Learning

Vertical Federated Learning (VFL) is a collaborative machine learning technique where different parties, each holding different features of the same dataset (i.e., same samples but different attributes), jointly train a model without sharing their data. The process involves:

- Entity Alignment: finding common sample IDs across different parties without revealing other non-intersecting samples.

- Privacy-Preserving Training: Each party computes intermediate representations H_i from their local features. These intermediate representations are sent to an active party. The active party computes a global model update using these representations and sends the updates back to the respective parties. Each party then updates their local model parameters accordingly.

This method ensures data privacy while leveraging the combined information from different feature sets to build a more comprehensive model.