

Hands-on Introduction to Deep Learning

Methodology

1. What **work** can be done to **improve** the **data** used for training?
2. How can a model be **evaluated**?
3. Is it possible to make the training more **robust**?
4. Can we **benefit** from an **already** trained model?
5. Bonus: Good **practices**

The chapter is divided in the following 5 sections :

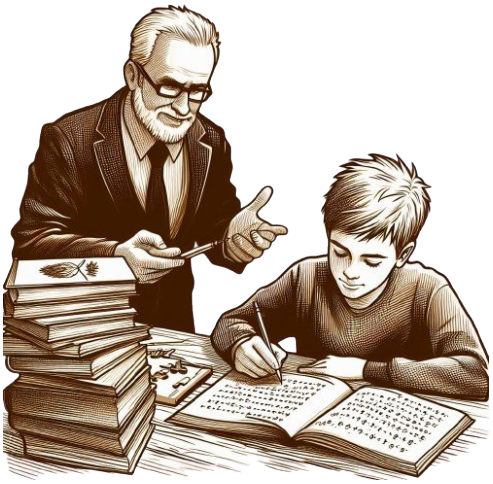
On data processing: do we have a good representation of our data (one that is relevant and has a direct impact on the result)? Do we have balanced data?

On model evaluation, particularly the choice of metrics: Which metrics are relevant, and how can we avoid falling into the trap of the ‘target metric’?

On training and methods to make it more robust

On model retraining, which avoids having to start training from scratch

And finally, some best practices to follow when working on deep learning projects



Learning from exercise with a teacher to guide us



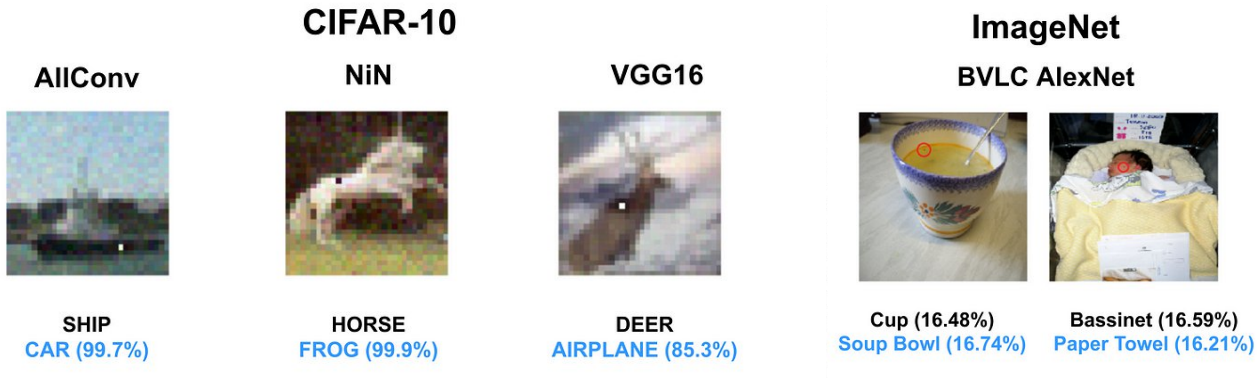
Applying what we learn to the real world

Training & Generalization

Learning is, first and foremost, a process of being guided to acquire knowledge and methods of generalisation. This knowledge and these methods are then put to the test on real-world data that has never been seen before.

It is this ability to generalise to unknown data that determines the quality of a model. If a model appears to perform well during training but fails when put to the test in real-world conditions, it means something has gone wrong.

https://arxiv.org/pdf/1710.08864.pdf



Generalization Lack Issue: One Pixel Attack

Here is an example of models that were initially considered robust (for their time) but which, in the end, proved extremely easy to fool using a ‘One Pixel Attack’.

The true class is shown in black; the prediction in blue. (For ImageNet: true class + percentage given by the model)

Subjetcs :

- 4 models: AllConv, NiN (Network in Network), VGG16 and AlexNet.
- 2 datasets: CIFAR-10 (10 classes) & ImageNet (20k+ classes)

Method:

- 1 modified pixel in the image

The classifier’s confidence in its incorrect answer (with the pixel) is **the same as for the correct answer** (without the pixel)

Explanations:

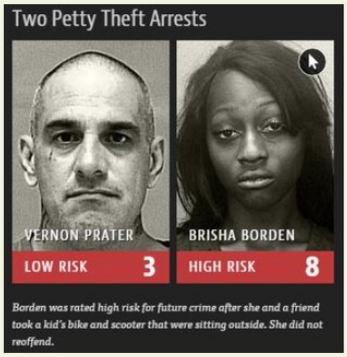
- During training, the model did not necessarily see examples with noise
- Robustness to noise can therefore be acquired

PROPUBLICA May 25 | 2016

MACHINE BIAS

What Algorithmic Injustice Looks Like in Real Life

A computer program rated defendants' risk of committing a future crime. These are the results.



Le Monde 25 août | 2026

PIXELS

Biais racistes de Gemini : Google « travaille à des améliorations »

« Je suis seule avec un Algérien », « Je suis seule avec un Italien » : deux questions simples, susceptibles de générer des réponses stéréotypées, voire racistes.

Par Nour Mohammedi

Les Echos October 13 | 2018

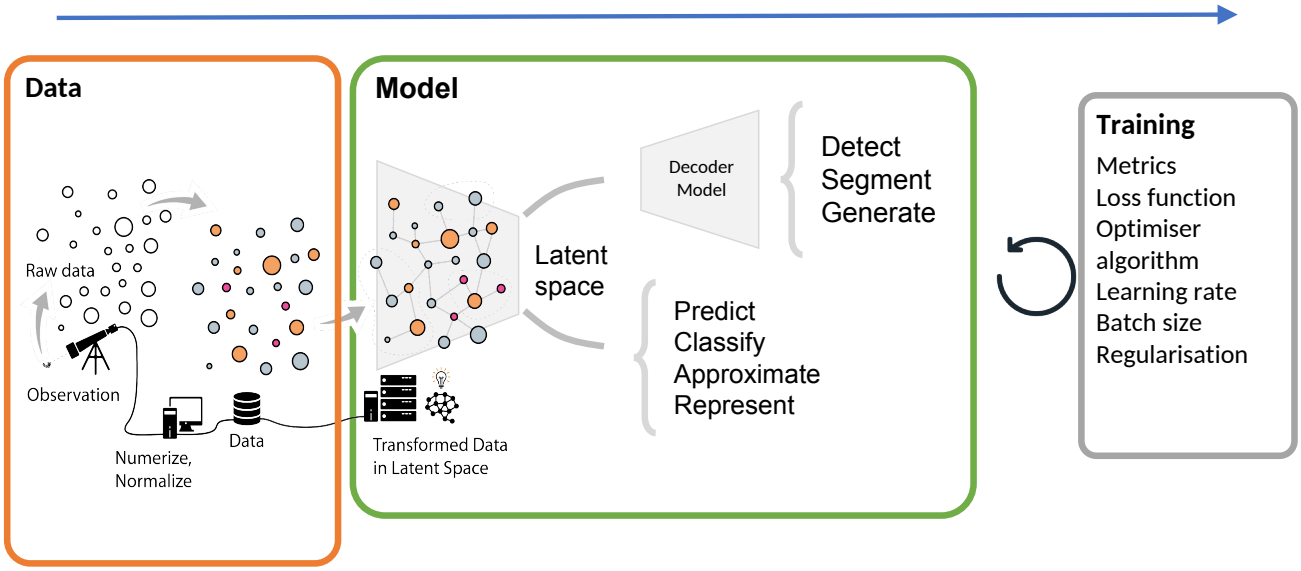
ENTREPRISES

Quand le logiciel de recrutement d'Amazon discrimine les femmes

En 2014, le géant du e-commerce a voulu confier ses candidatures à un algorithme, mais celui-ci a commencé à écarter les profils féminins.

Generalization Lack Issue : Discrimination

Lack of robustness can have real-world consequences .



Deep Learning Pipeline Principle

Pipeline for training our deep learning models:

1) Data loading and preparation :

Data, which may consist of images, text, graphs or physical quantities are the result of observations, aggregations or synthetisation. They usually go through a pre-processing step where experts (often computer-assisted) clean and curate them. The training pipeline contains dataloading steps to mitigate biases and enventually apply transformations to improve robustness.

Preparing data is a massive task and the most crucial. As the saying goes : « Garbage in, garbage out. »

2) Find a suitable model that takes our data as input and returns an output that corresponds to the solution

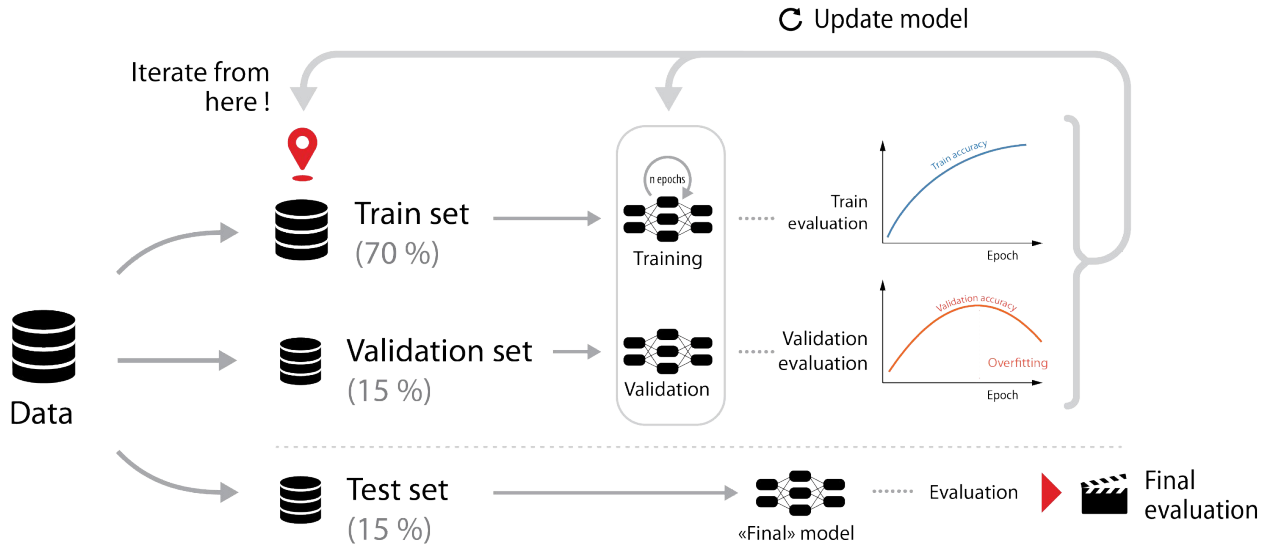
To summarize : The bulk of the model is here to find a **representation** of the data in what is called a **latent space**. This representation can then be utilized for various tasks, be it segmentation, visualisation,...

Finding the model is probably the easiest task.

3) Establish a training method that influences the way the model learns from the data.

This is the other big task : The training method has hyperparameters you can play with which will influence the training quality and efficiency.

We will see that all these elements play a part during training



Data splitting

A reminder of the importance of splitting the data into three subsets: **training** / **validation** / **test**

training: to evaluate the loss, calculate the gradient and then update the network's weights

validation: to tell us when to stop training in order to avoid overfitting

test: to evaluate the models' performance and compare them **once the model is done training**.

If once the model has been evaluated on the **test** set you go back and tweak it then restard a training, the test set has now been used for training and cannot be considered a truly reliable indicator.

If we have less data, we can do without the validation set.

Dataset is the culprit ?!



Augment It, Transform It !!!

Super Regularizator !!



Use his Super Powers !!

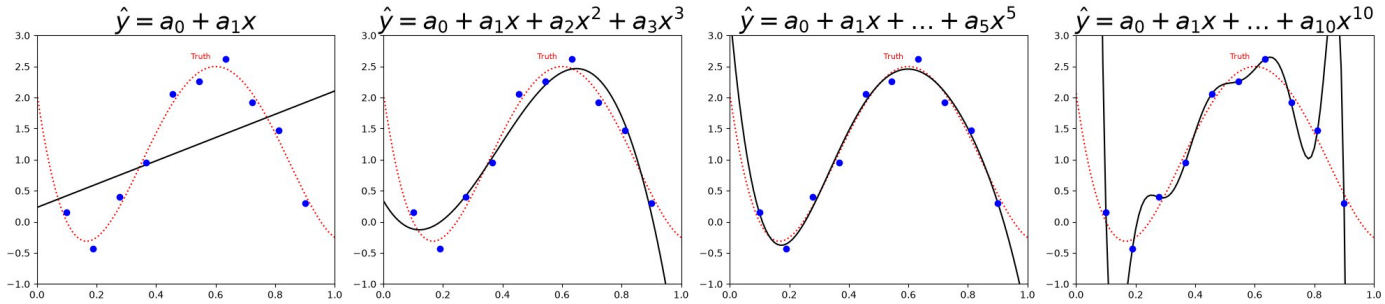
For a Better Generalization...

As previously mentioned, the ability to generalise, meaning to perform as well on new data as on training data, is synonimous with training a robust neural network.

It's therefore paramount that we are able to measure the efficacy of our training and whether we have learned enough to generalise. And two new questions arise :

- How can we make the training more robust with only the data we have ?
- What happens when the model encounters new data ?

Regularization is a set of techniques used to avoid **overfitting** and thus improve the model's performance on unseen data. Regularization is often mandatory and thus an automatic inclusion in most deep learning pipelines.



Underfitting and Overfitting

Lastly, we need to introduce the concepts of **underfitting** and **overfitting**.

Let's take a simple example of a model trying to fit a polynomial on a set of sample points. The **ground truth** is a degree 4 polynomial on which noised samples were taken.

We can easily see what happens when the degree of our model's polynomial is off.

WIth a lower degree the polynomial cannot correctly map the data : this is **underfitting**. *The performance is poor on both the training set and the testing set.*

With too high of a degree, the polynomial matches the data too closely, and ends up matching the noise of our data rather than its general tendency. *It will perform very well on the training set but poorely on the validation set.*

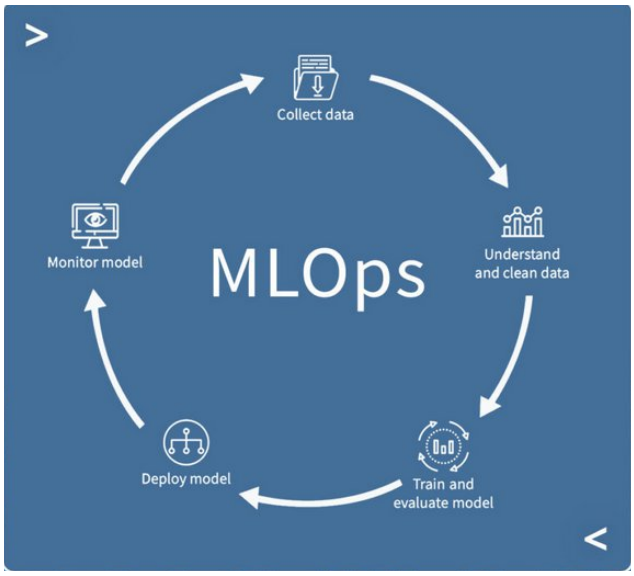
We'll see that a neural network can be subject to both **underfitting** and **overfitting**, although it is more complex than simply changing an polynomial's degree. This is were regularization comes into play.

With a simple, real-world example, it would be like training a model to classify apples versus other fruits. Because of bad luck/bad rigor, our dataset is composed in majority of apples of the golden variety. Thus, our model, instead of learning the general characteristics of an apple, will learn « yellow = apple » and classify a banana as an apple and a red apple as « not an apple ».

1. What **work** can be done to **improve** the **data** used for training?
2. How can a model be **evaluated**?
3. Is it possible to make the training more **robust**?
4. Can we **benefit** from an **already** trained model?
5. Bonus: Good **practices**

The first section will tackle the data processing : how it influences our model's final capabilities. We will see how to balance the dataset and ensure it gives us the best performances.

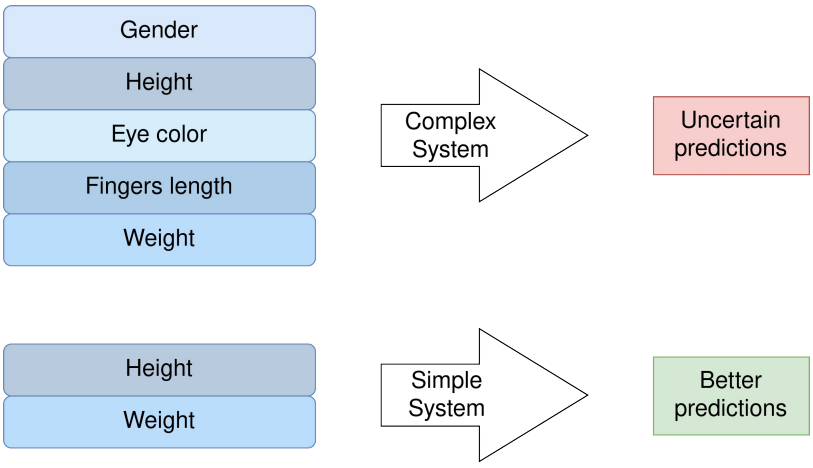
https://posit.co/blog/mlops-readiness/



50 to 80% time spent on data

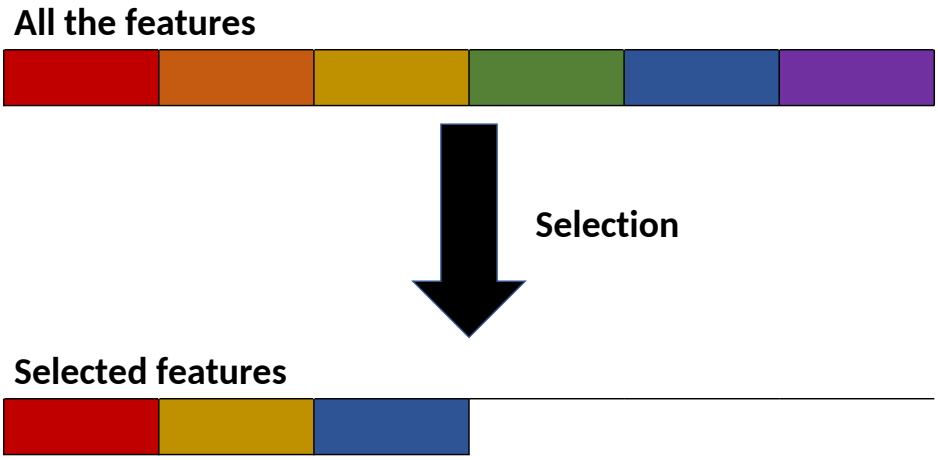
Model development

MLOps or ML Ops is a paradigm that aims to deploy and maintain machine learning models in production reliably and efficiently. About 50 to 80% of the time is spent on data collecting, curating and cleaning. It cannot be ignored.



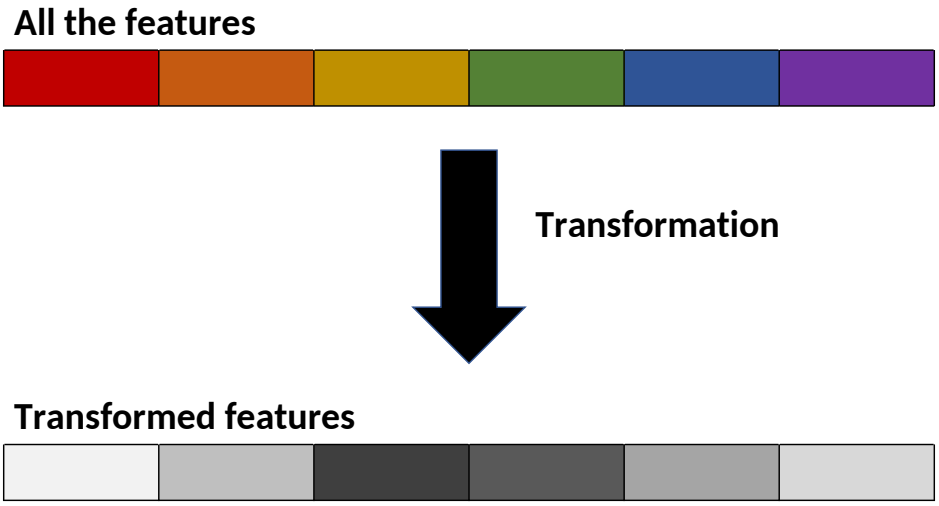
Features selection

Working with your data means, first and foremost, understanding how to feed it to your network.
For example, not all information is necessarily relevant, and some may even be detrimental to training.
So the simplest way to get round this is to remove any unnecessary or harmful features.



Features selection

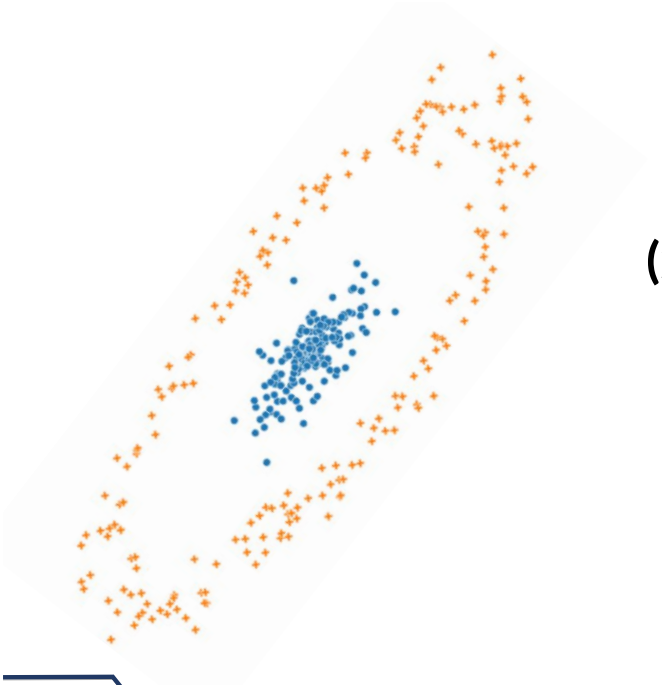
This removal of features is called « **Features selection** ». It doesn't remove samples from the dataset, but part of the information of each sample, keeping only the relevant ones.



Feature transformation

When every features is useful, but not evey information conveyed by them is, **Feature Transformation** is used to extract only necessary information from base features. One example of this is going from an RGB picture to greyscale when we only want to look at the brightness.

While it can sound similar to **Features Selection**, this technique is about extracting information, not substracting it. For example, **Feature Selection** on an RGB dataset would be removing the Red and Blue components, while **Feature Transformation** aggregates them into a new measurement.



(x , y) → **Cartesian coordinates: Complex relation between x and y**

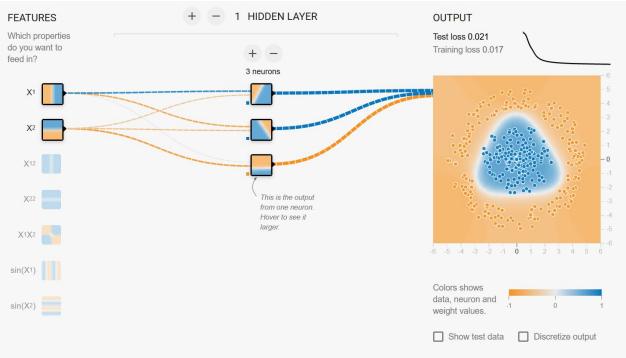
(r , θ) → **Polar coordinates: Simple relation with r (radius) and θ (angle)**

Feature transformation - Example

New example of feature transformation: changing coordinate systems – for example going from Cartesian to polar, where the coordinates are described with *r*, the distance from centre, and *theta*, the angle.

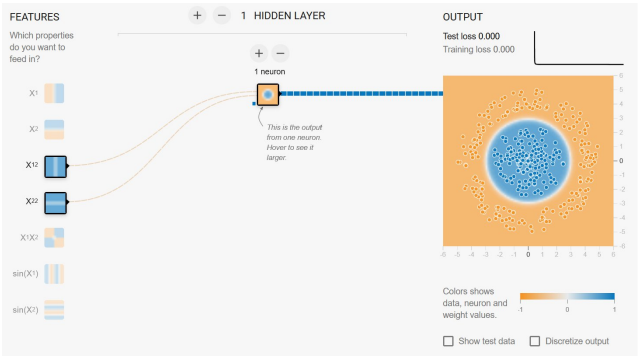
Here, it is very difficult to identify the centre points using X and Y coordinates, whereas simply setting a threshold on *r* is enough to distinguish them.

https://playground.tensorflow.org



Données de position (x, y)
Réseau à 3 neurones

Données transformées
Réseau à 1 neurone



Feature transformation - Example

Another example that was seen in a Practical : squaring our x and y features. Now we can get better predictions with a network that is 3 times smaller.

https://arize.com/blog-course/embeddings-meaning-examples-and-how-to-compute/



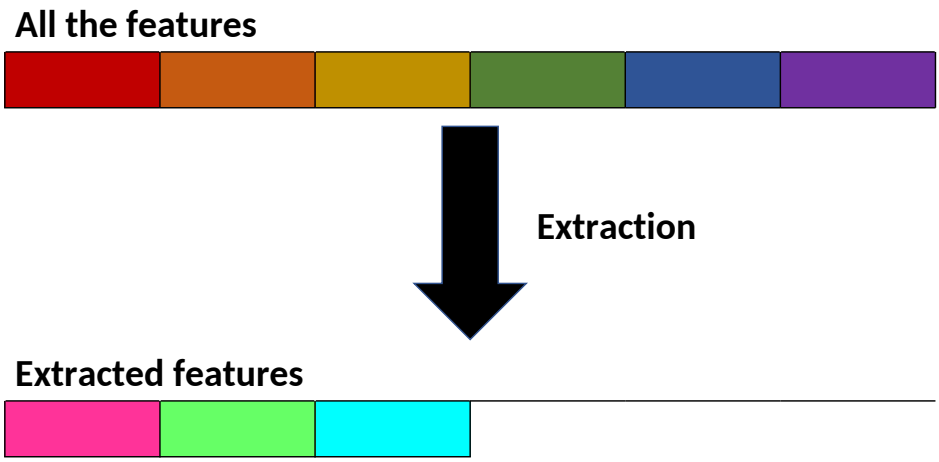
Features extraction

Finally, there is a third method which involves reducing the complexity of our data by transforming it into a smaller set of features – one that is therefore more compact but still distinguishing : **Feature Extraction**.

For example, by applying what is known as Principal Component Analysis to carry out dimensionality reduction and create new features that have no real physical counterpart, but still meaning.

Again, this technique looks and sounds similar to feature selection, but while the latter creates a subset of existing features, **Feature Extraction** creates a new set of features in a space of fewer dimensions. This is the concept behind Embedding : A 256x256 grayscale image has more than 65k dimensions, whereas its embedding in a latent space could be as low as 800, more meaningful dimensions.

In the embedding example above, we want to discriminate between traffic signs, but sometimes we may encounter a stop sign in a different language. Embedding allows us to find a representation space where all stop signs are close to each-other : Not all stop sign have « stop » written on them, but all of them are red and octagonal.



Features extraction

To sum up, **Features Extractions** results in fewer features – features that aren't necessarily physically interpretable, but which are more discriminating



All the features

Selection

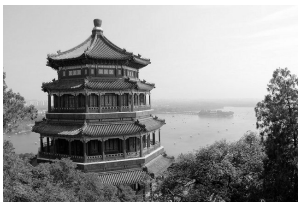
Selected features



All the features

Transformation

Transformed features



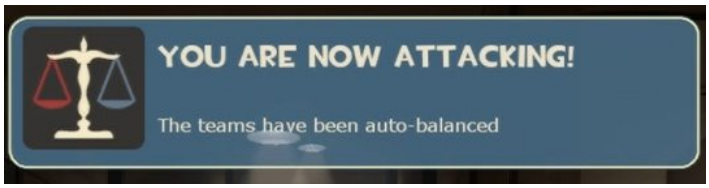
All the features

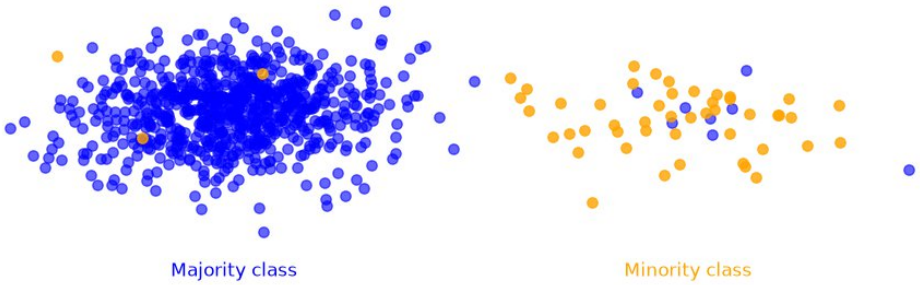
Extraction

Extracted features

```
[  6208.56  -160.27   304.41   661.01    52.87
    56.86  -125.47    21.15    39.49    34.21
                        ...
     5.89    19.53    -6.33    17.98    -0.35
    16.14     4.38     9.64     8.03    -7.29 ]
```

Features engineering - takeaway





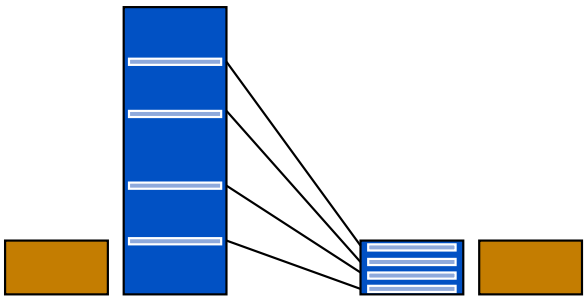
Balance the classes

The following problem is not related to our features but is intrinsic to the distribution of the dataset: the balance between classes.

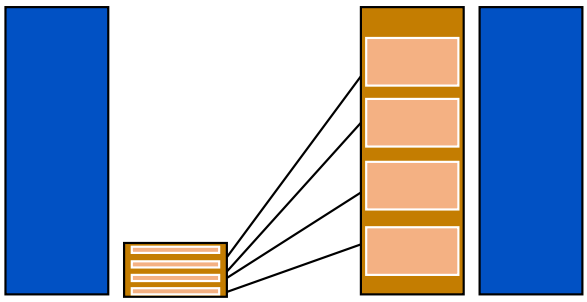
For example, when we have data labelled 1 and 0 (binary classification), it may happen that the positive class is far less numerous than the negative class.

The issues this raises are easy to see (as we'll see later: very high accuracy but very low precision); in any case, this will limit the robustness of the neural network if it does not see enough positive examples to learn to generalise.

Undersampling



Oversampling



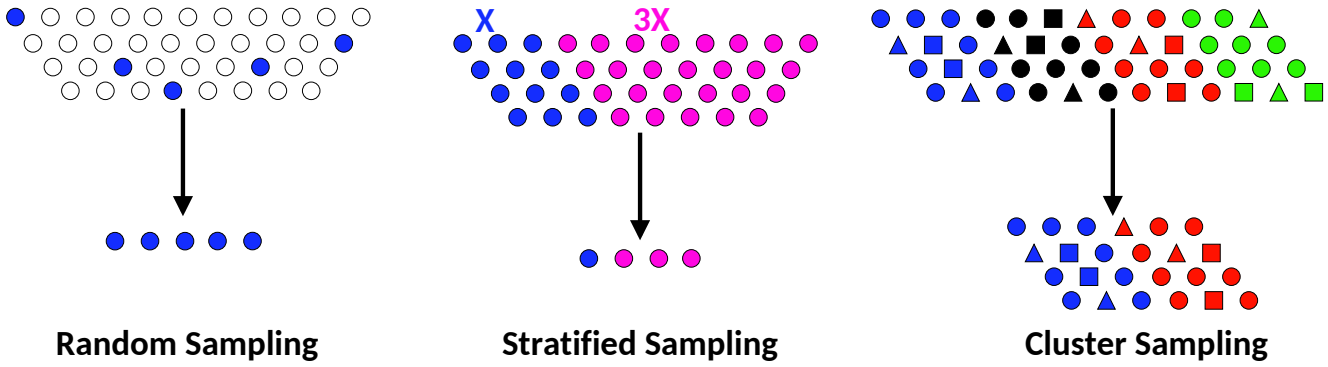
Balance the classes

Two types of class balancing :

– Undersampling (easyto execute but must be done properly): Sample a subset of the majority class.

– Oversampling (harder to execute properly): Expand the minority class by creating samples.

https://towardsdatascience.com/8-types-of-sampling-techniques-b21adcdd2124



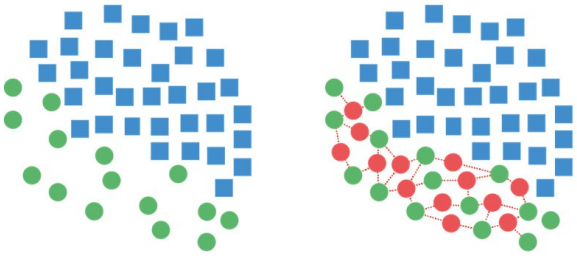
Under sampling

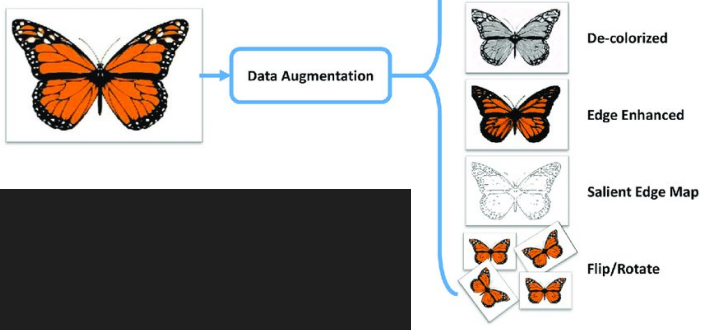
Subsampling techniques;

1) The ‘dumb’ method (there’s always a ‘dumb’ method) ==> Random subsampling
2) Stratified sampling (a more advanced method): the data is divided into homogeneous sub-populations (strata) and then random samples are drawn from within these sub-populations. This ensures that each data stratum is represented (where possible, the ratio of sizes between the strata is maintained)
3) Cluster sampling (a dsecond advanced method) : Create mutually homogeneous but internally heterogeneous clusters, then draw random clusters. This is less costly than stratified sampling, as you sample on clusters rather than individual data points.

https://pub.towardsai.net/maximizing-the-impact-of-data-augmentation-effective-techniques-and-best-practices-c4cad9cd16e4
https://medium.com/@parthdholakiya180/smote-synthetic-minority-over-sampling-technique-4d5a5d69d720
https://pytorch.org/vision/0.9/transforms.html

Synthetic Minority Oversampling Technique






```
# Define the transformations
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.RandomResizedCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

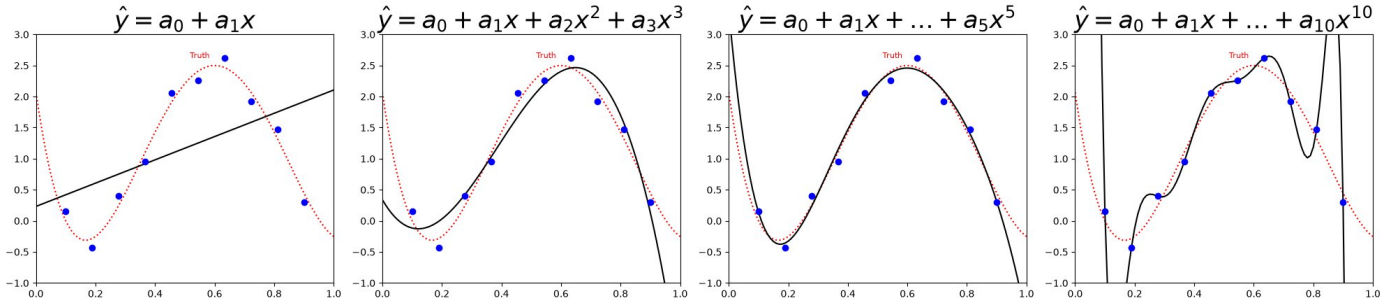
Over-Sampling/Data augmentation

Oversampling is more complicated, as it is difficult to generate new data: if we had more, we would use more. There are, however, a few techniques:

1) SMOT (Synthetic Minority Oversampling Technique) ==> By interpolating between two points in the minority class, a new point in that class is created.
2) Data augmentation: New data is created by slightly modifying existing data. This also has the advantage of making the model more robust in the face of noise.

1. What **work** can be done to **improve** the **data** used for training?
2. How can a model be **evaluated**?
3. Is it possible to make the training more **robust**?
4. Can we **benefit** from an **already** trained model?
5. Bonus: Good practices

The second section will focus on model evaluation, particularly the choice of metrics: Which metrics are relevant, and how can we avoid falling into the trap of the ‘target metric’?



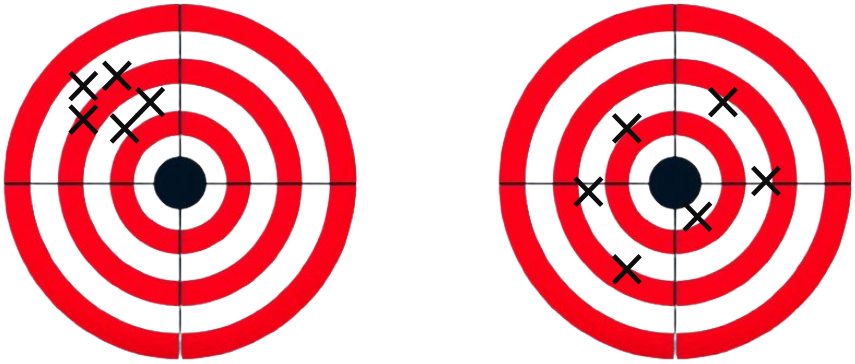
Bias variance trade-off

To do this, let's go back to our under and over-fitted polynomials in order to introduce the concept of biars-variance tradeoff.

As previously stated, when training a model, it is important not to go too far, whilst still doing enough to ensure it is relevant. In other words, our model must not be **biased**, but nor must it be too sensitive to the **variance** in our data.

And underfitted model is not sensitive enough and its results do not correlate with the data: this model **is too biased**.

Conversely, an overfitted model risks being influenced by noise in the training data. In other words, it is **too sensitive to variance**.



Noise or bias ?

In your opinion, which diagram represents a model dominated by bias, and which represents a model dominated by variance?



Bias



Noise

Noise or bias ?

There is no correlation between my model predictions and the label



Bias

My model says « It's a cat » no matter the image I give it.



Noise

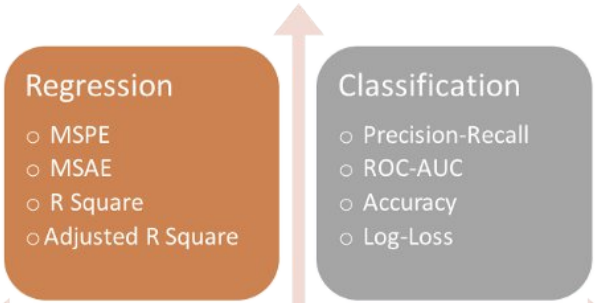
My model is not consistent in its answers and is mostly wrong.

Noise or bias ?

In both cases, the model's performance is poor.

The first one has a very regular aim, but doesn't aim at the right spot.

The second one is very poor at aiming but the noise is centered around the bullseye.



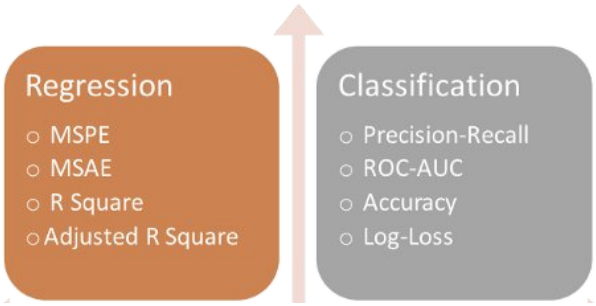
- **What is a metric?**
- **Is it different from a loss function?**

Metrics

Now that we've seen that: How do we measure our model's performance? By using metrics, which analyse our model's outputs to assign it a score.

There are a multitude of metrics, which are suited for different tasks (regressionvs classification) and different cases (binary classification). Because there isn't a single perfect metric, we will often use a combination of several metrics to accurately measure our model's capabilities.

So how does a metric this differ from a loss function?



- **What is a metric?**
- **Is it different from a loss function?**
  - **Differentiability**
  - **Training vs evaluation**
  - **Interpretability/Meaning**
  - **Number of samples**

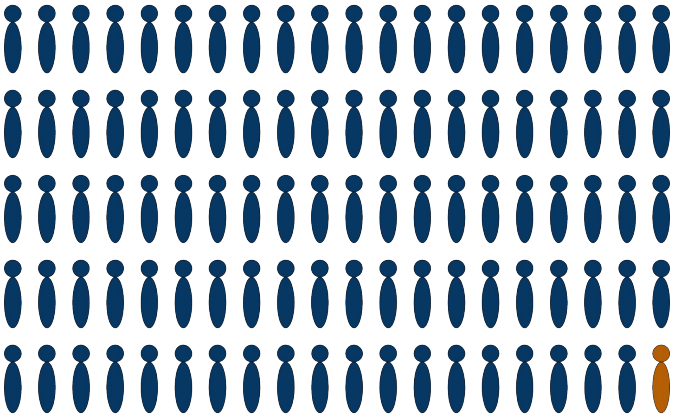
Metrics

The loss function is differentiable; it must be so in order to perform backward propagation.

The loss function is used solely during training, whereas a metric is used to evaluate the model at any given moment.

This is because the loss function has no real physical meaning: it serves to inform the network of the accuracy of its responses.

The amount of data used by each is different. The loss function can be calculated using just a handful of data points (in an extreme case: a single data point), whilst a metric requires a large volume of data to be representative and meaningful.





All negative : accuracy = 99% ✔ ?

New system for illness detection – the accuracy is not a good metric for this case

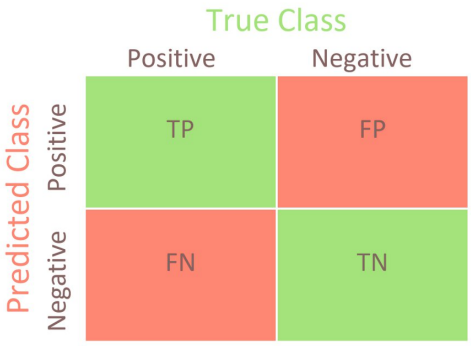
Metrics - Bad Choice

As with everything, there are good metrics and bad metrics:

The most obvious metric for classification tasks is **accuracy**: we measure how many of our model's predictions are correct.

But in the case of imbalanced classes (for example, a disease affecting 1 per cent of the population), a poorly trained network which always gives the same answer may still have good accuracy.

This problem cannot be solved by increasing the amount of data, as we want to evaluate the model using real data.



Confusion matrix

precision = $\frac{TP}{TP + FP}$

Above all positive prediction, how many are positive data

recall = $\frac{TP}{TP + FN}$

Above all positive data, how many have been predicted positive

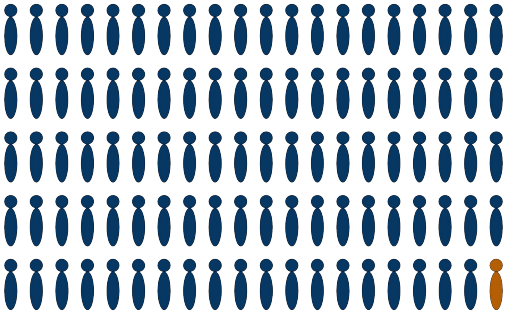
Metrics - Precision/Recall

It is therefore necessary to use several metrics to measure performance. Here, we can introduce two:

Precision = How many of our positive predictions are *actually* positive data points

Recall = How many of the *actually* positive data points were predicted as positive

These metrics are computed based on values found in what is called a « confusion matrix »





Predicted class \ True class	0	1
0	0	
1		99

$$Accuracy = \frac{99}{100} = 99\%$$

$$Precision = \frac{0}{0+0} = NaN$$

$$Recall = \frac{0}{0+1} = 0$$

Predicted Class \ True Class	Positive	Negative
Positive	TP	FP
Negative	FN	TN

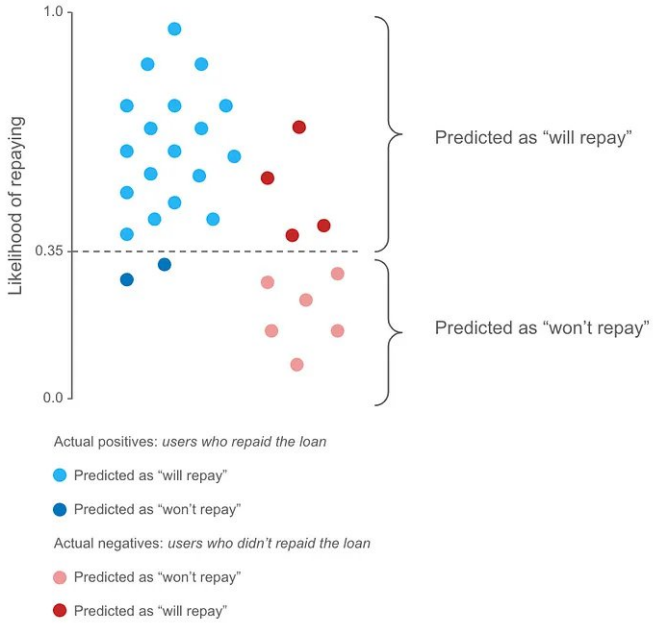
$$\text{precision} = \frac{TP}{TP+FP}$$

$$\text{recall} = \frac{TP}{TP+FN}$$

Metrics - Precision/Recall

So, whilst accuracy suggests our model as performing, precision and recall reveal its actual performance.

https://towardsdatascience.com/understanding-the-roc-curve-in-three-visual-steps-795b1399481c



Actual positives: *users who repaid the loan*

- Predicted as “will repay”
- Predicted as “won’t repay”

Actual negatives: *users who didn’t repaid the loan*

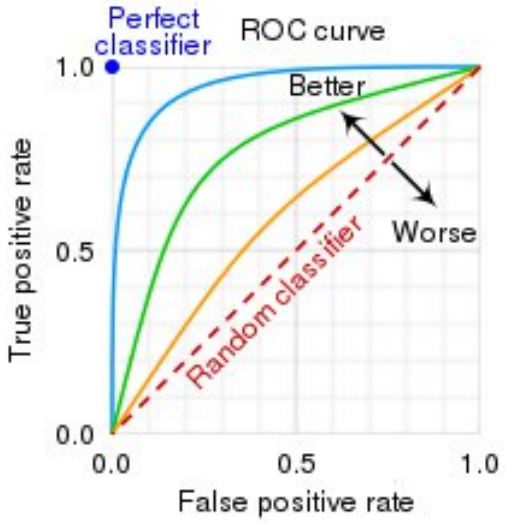
- Predicted as “won’t repay”
- Predicted as “will repay”

Metrics - ROC curve

Let’s now talk about an aggregated metric for binary classifiers.

Let’s say we have a model that must classify points between two categories : 0 and 1. The output of our model is therefore a number between 0 and 1 and we must choose a threshold above which a value is considered as 1, and 0 otherwise.

It is most likely that we can never get a perfect classifier and some points will be misclassified. How, then, can we find the threshold which offers the best compromise.



Variation of the acceptance threshold of a class to obtain the curve

$$TPR = \frac{TP}{TP + FN}$$

Above all positive data, how many have been predicted positive

$$FPR = \frac{FP}{FP + TN}$$

Above all negative data, how many have been predicted positive

NB : ROC = Receiver Operating Characteristics

Metrics - ROC curve

This aggregated metric is known as the ROC curve (Receiver Operating Characteristic).

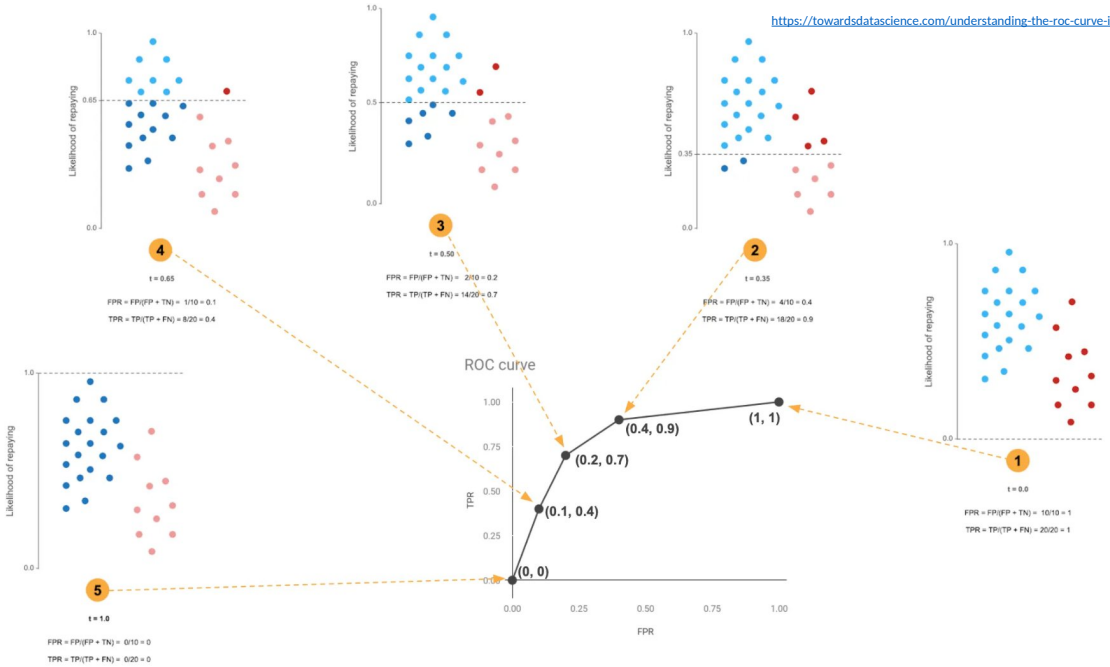
It involves measuring the True Positive Rate (TPR) – which is effectively the recall – and the False Positive Rate (FPR) based on the acceptance threshold for the positive class.

Those two metrics indicate how many of our data points have been correctly classified.

A perfect classifier would have a TPR of 1 and an FPR of 0.

We can see that the better our classifier is, the closer the area under the ROC curve will be to 1. It is this area we can use as metric. (But it is not necessarily the best one, as the ROC curve is used to measure a trade-off.)

https://towardsdatascience.com/understanding-the-roc-curve-in-three-visual-steps-795b1399481c



Metrics - ROC curve

This slide show how to build a ROC curve by choosing multiple decision threholds.

https://huggingface.co/docs/transformers/perplexity

Perplexity :
Is model surprised to see this text?

$$\mathrm{PPL}(X) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log p_\theta(x_i|x_{<i})\right\}$$

This person is innocent, she needs a lawyer

- “ssistance” - 1.5%
- “ lawyer” - 0.5%
- “ car” - 0.05%
- …
- “ sea” - 0.0001%

Metrics - Language models evaluation

Let’s now look at a metric specific to language models: perplexity:

For each word or part of a word in a text, we measure the probability that our model assigns to that part. Aggregating these probabilities gives us a score that measures how surprised our model is to see this text.

The lower the perplexity, the more our model’s predictions align with reality.

https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

⚠ Perplexity is not All you Need !!

T	Model	Average	ARC	HellaSwag	MMLU	TruthfulQA	Winogrande	GSM8K	Architecture	#Params (B)
	moreh/MoMo-70B-lora-1.8.6-DPO	77.29	70.14	86.03	77.4	69	84.37	76.8	LlamaForCausalLM	72.29
	moreh/MoMo-70B-lora-1.8.4-DPO	76.23	69.62	85.35	77.33	64.64	84.14	76.27	LlamaForCausalLM	72.29
	TomGrc/FusionNet_7Bx2_MoE_14B	75.91	73.55	88.84	64.68	69.6	88.16	70.66	MixtralForCausalLM	12.88
	Weyaxi/Helion-4x34B	75.48	69.71	85.28	77.33	63.91	84.37	72.25	MixtralForCausalLM	113.66
	one-man-army/UNA-34Beagles-32K-bf16-v1	75.41	73.55	85.93	76.45	73.55	82.95	60.05	LlamaForCausalLM	34.39
	Weyaxi/Cosmosis-3x34B	75.39	69.71	85.18	77.25	63.82	84.14	72.25	MixtralForCausalLM	87.24
	Weyaxi/Bagel-Hermes-2x34b	75.1	69.8	85.26	77.24	64.82	84.77	68.69	MixtralForCausalLM	60.81
	jondurbin/bagel-dpo-34b-v0.2	74.69	71.93	85.25	76.58	70.05	83.35	60.96	LlamaForCausalLM	34.39
	jondurbin/nontoxic-bagel-34b-v0.2	74.69	72.44	85.64	76.41	72.7	82.48	58.45	LlamaForCausalLM	34.39
	moreh/MoMo-70B-LoRA-V1.4	74.67	69.2	85.07	77.12	62.66	83.74	70.2	LlamaForCausalLM	72.29
	udkai/Turdus	74.66	73.38	88.56	64.52	67.11	86.66	67.7	MistralForCausalLM	7.24

Be sure to look at Language models evaluation benchmarks which suite your needs.

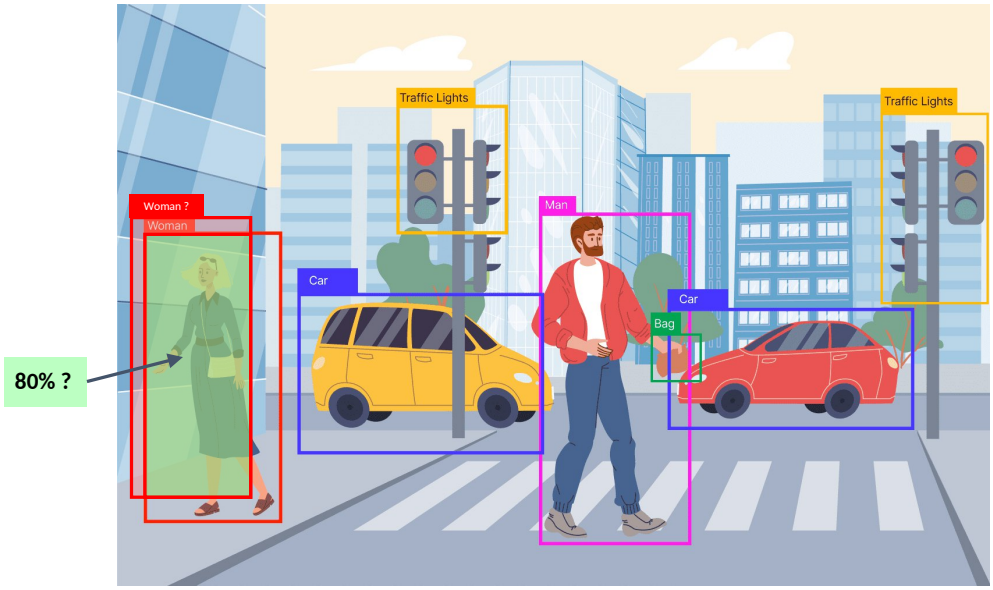
The evaluation is complicated, there are many different tasks
It can depends on the targeted application
It is hard to be fair/objective without knowing the training data



Metrics - Language models evaluation

But perplexity is not the be-all and end-all of language models. Whilst it allows us to assess whether our model speaks a language well (which is, after all, what we ask of it in the first place), it does not measure other aspects of the model's performance (such as its ability to solve puzzles, tackle maths problems or tell jokes, for example).

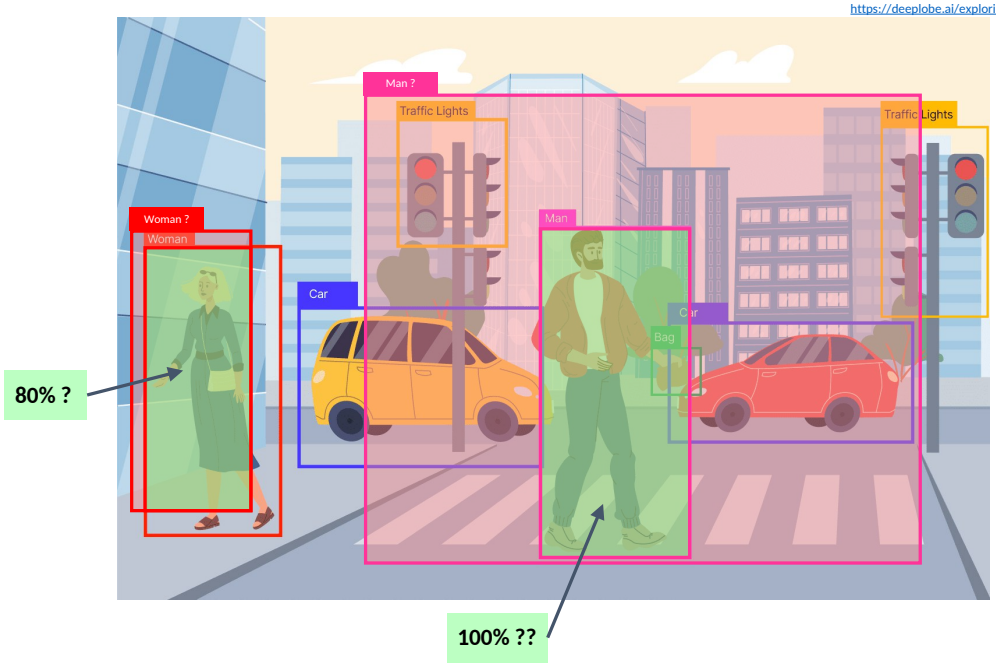
https://deeplobe.ai/exploring-object-detection-applications-and-benefits/



Metrics - Intersection over Union

Intersection over Union, or **IoU**, is a metric used for image segmentation task. It measures how much of our model's predicted zone is truly the zone of interest in our image.

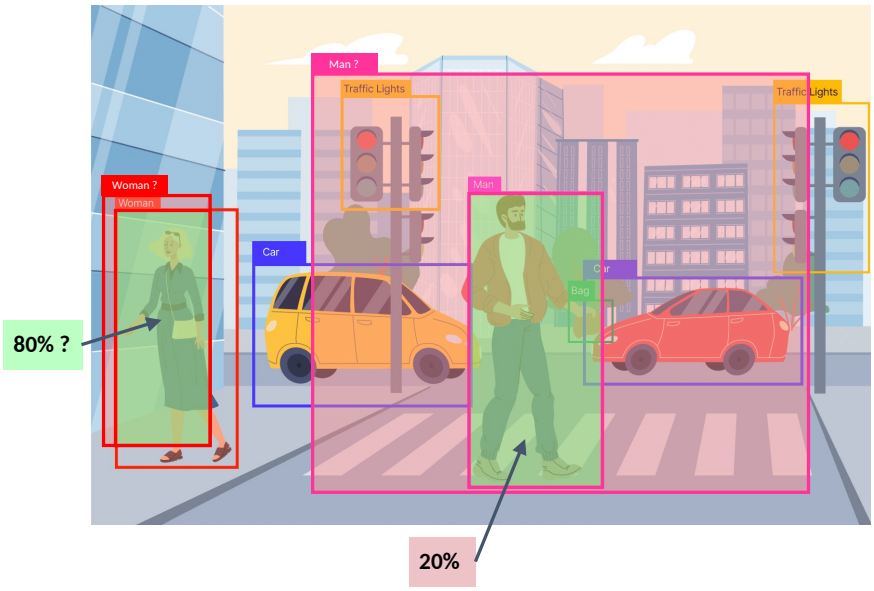
https://deeplobe.ai/exploring-object-detection-applications-and-benefits/

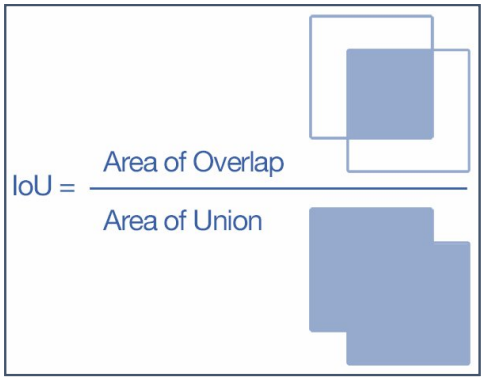


Metrics - Intersection over Union

If we were to simply measure how much of the truth our model predicted, models would be encouraged to take as wide a zone as possible to be sure to encapsulate the points of interest.

https://deeplobe.ai/exploring-object-detection-applications-and-benefits/





Intersection over Union

Metrics - Intersection over Union

IoU solves that by rewarding the **Intersection** of prediction and truth, and penalizing the **Union** of prediction and truth. Its is a value between **0** and **1** equal to the ratio of the **Intersection** over the **Union**.

If the model misses the points of interest, the intersection is null and so is **IoU**.

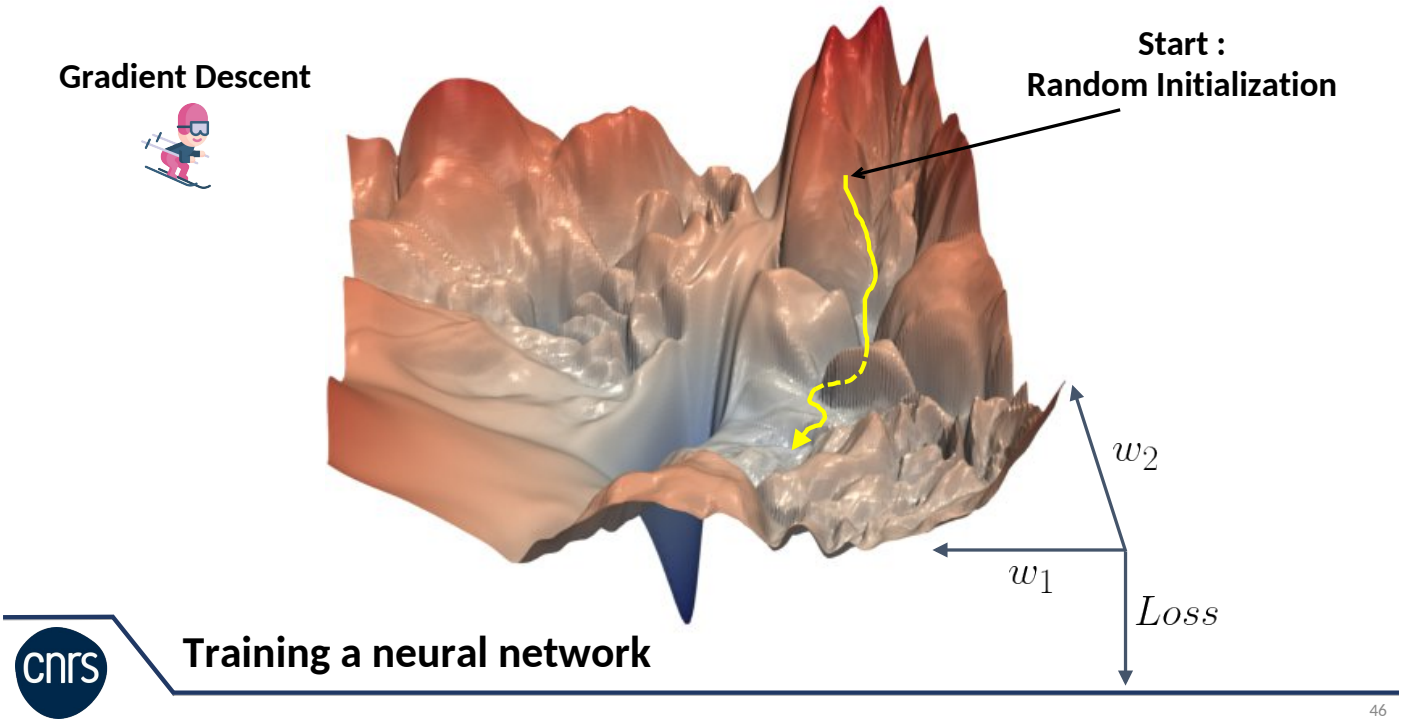
If the model takes too large of a zone, the union is too large and **IoU** is close to zero.

Only when the model's predictions closesly matches the truth is the **IoU** close to 1.

1. What **work** can be done to **improve** the **data** used for training?
2. How can a model be **evaluated**?
3. Is it possible to make the training more **robust**?
4. Can we **benefit** from an **already** trained model?
5. Bonus: Good practices

This third section will focus on techniques to render the training algorithm more robust, and mainly **regularization** techniques.

https://losslandscape.com/



What does a gradient descent run look like?

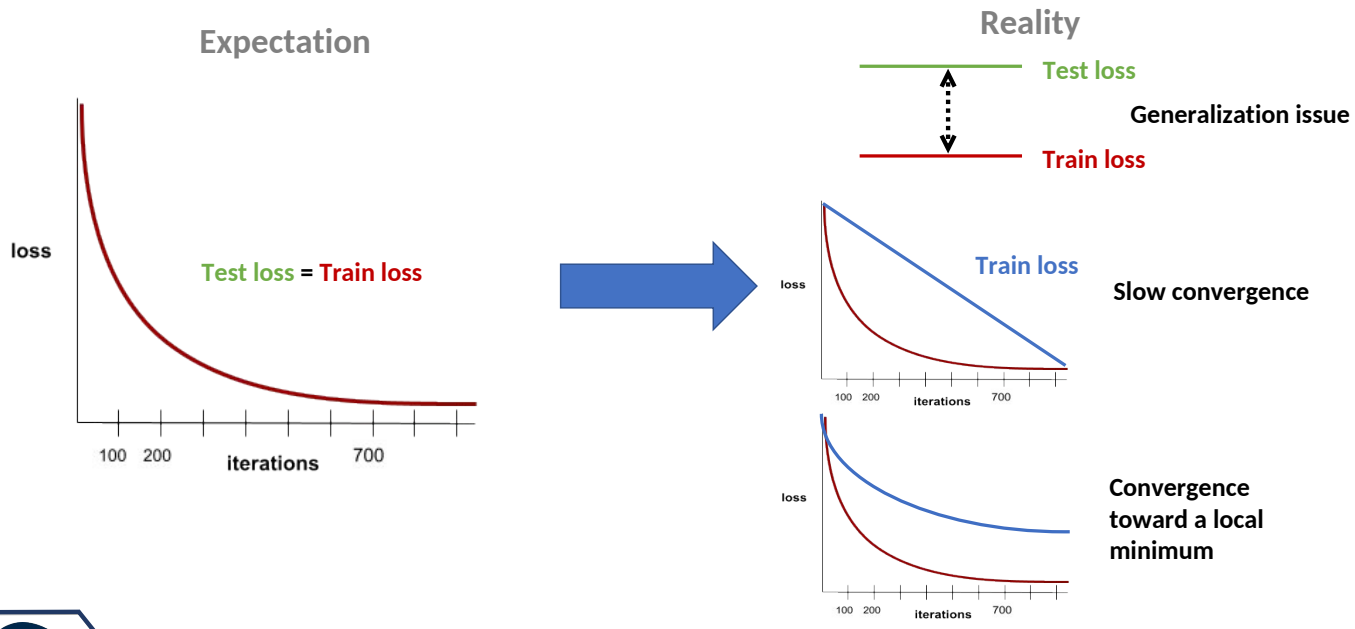
The image shown above is called a ‘loss landscape’. It is a representation of the loss value as a function of the model’s weights, or at least a projection of the model’s weights.

It is important to note that this landscape is, in reality, ‘inaccessible’. During training, we can only evaluate the loss based on the network’s current state and obtain information about the neighbourhood of that state by calculating the gradient. This gradient will indicate, locally, the direction in which we should move to reduce this loss.

What the loss landscape illustrate in a very visual way, though, is a « physical » analogy to gradient descent : it is analogous to litteraly sliding down a slope at a given speed.

Ideally, we would like to reach the global minimum of this function, but there is no guarantee of this; instead, we hope to reach a good local minimum.

The loss ladnscape can be retrieved through costly calculation as some papers have done. The website https://losslandscape.com/ contains useful rsources to understand the loss landscape and gradient descent.

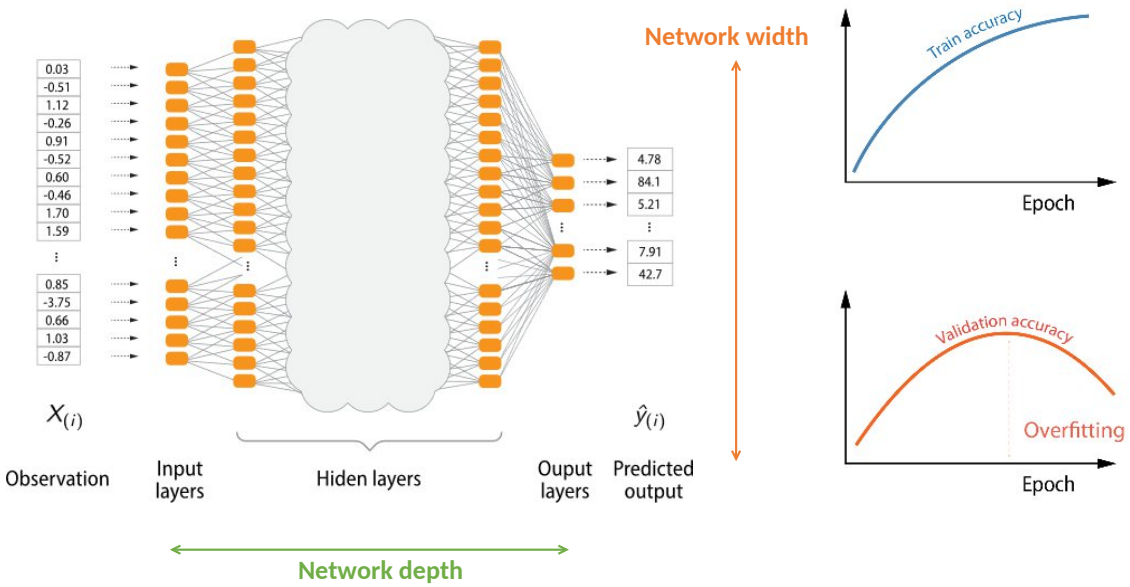


What happens during training ?

So, what happens during a ‘failed’ training session?

In an ideal world, we would expect a model to produce similar results on the training and validation sets at every stage of training. In practice, this is not the case, and it is precisely this difference in performance that measures the network’s ability to generalize.

Convergence might not be 1-to-1 between the training and validation sets,, and while a model might sometimes just be slow to converge (which is an issue cost-wise), it can also become trapped in a local, unsuitable minima and stop learning all-together.



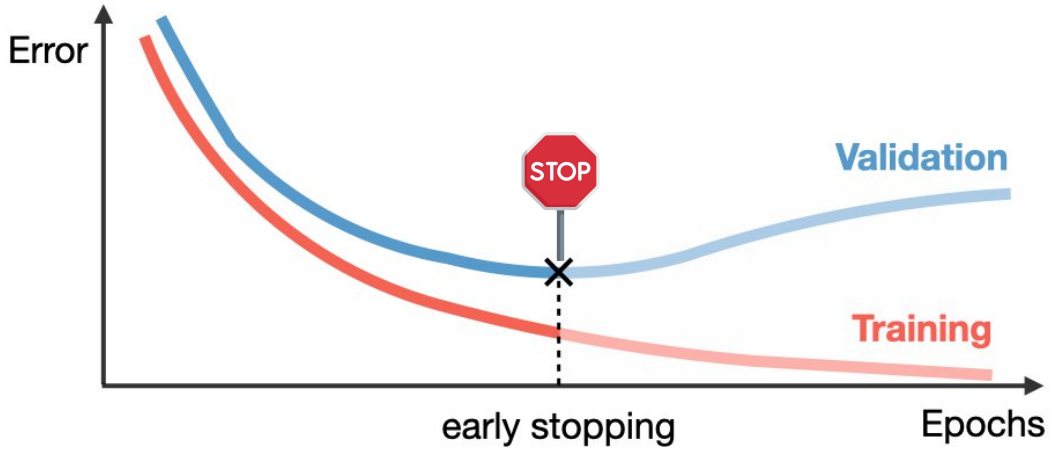
Model architecture and overfitting

Overfitting will be characterized by the validation accuracy collapsing after a certain point. Up to that threshold, the model was still able to generalize, but trainign further will only make the model learn the training set « by heart », so to say.

Often, overfitting will be caused by a model that is too complex. It was shown that if a model has too many parameters relative to the number of training sample, it will be prone to overfitting.

There are intuitive ways to there are intuitive ways to scale the architecture by considering the depth and width of the network; this yields good results. If we want to refine the architecture, we can use NAS (Network Architecture Search), which is costly.

https://wandb.ai/wandb_fc/french/reports/L-extraction-d-informations-partir-de-documents--Vmlldzo0OTI0OTU



Basic way : Early stopping

A common practice to avoid overfitting is to use early stopping. Either we detect when the validation loss starts diverging, or we save the current best model at each epoch, and at the end of training use that checkpointed model as our final one.

We will now look at other regularization methods.

$$\Theta_{t+1} = \Theta_t - \eta \nabla_\Theta \left[\mathcal{L}(\hat{y}_i, y_i) \right]$$

Updated weights = Weights before update − Learning rate * Gradient [Cost function (Prediction , Label)]

Weight update equation

Regularization - Loss function and weight update

This is the standard weight update algorithm for the Stochastic Gradient Descend.

Notice the first of what we call « hyperparameters » of the training : the learning rate (represented by the greek letter Eta or Alpha). Its role is to set the speed at which we update gradient weights at each iteration.

Hyper-parameters

- Learning rate
- Regularization
- Optimizer
- Model architecture
- Batch size
- …



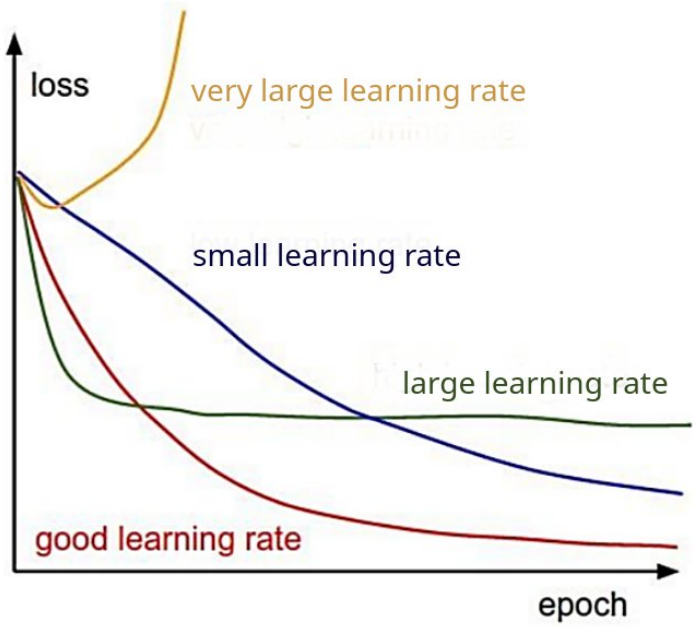
Hyperparameters and training

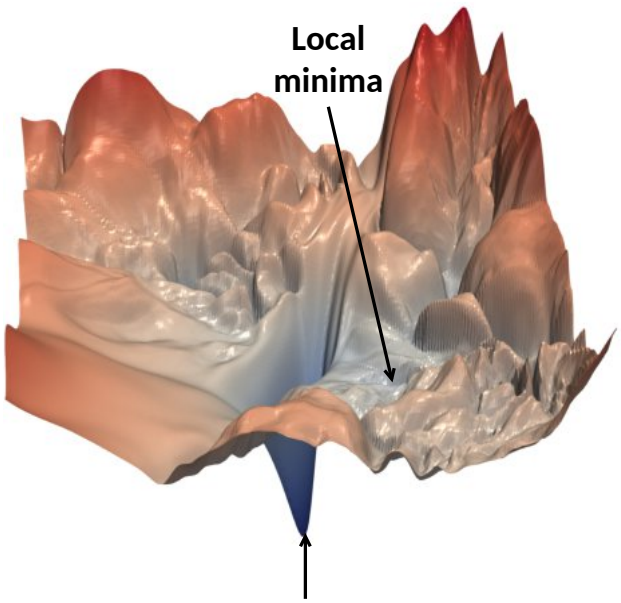
Side tangent : What are hyperparameters ?

They are values that are set before training begins and which influence the training process, and are distinct from the model's parameters (aka its weights).

Because their value influence the training (cf the Learning Rate), finding the correct values is a task of its own. This is the engineering side of machine learning, where we'll need to try to identify the hyperparameters' impact on the robustness of the training process.

There can sometimes be benefits to choosing hyperparameters that make the training less robust but subsequently lead to better generalisation results.





Learning rate

We'll talk here about the learning rate value and introduce some concepts related to it.

So how do we choose a proper learning rate value. Let's first see how the range will generally affect training :

With a **small learning rate**, our weight updates will be small in turn and while we might reach a proper, performant state of our network, convergence is going to be slow, and we might get stuck in a local minimum.

While we want to speed-up convergence, taking a **very large** learning rate will be counter-productive, and training will become unstable and diverge.

Then let's reduce is to « simply » a **large** learning rate, which will converge in a stable manner but might bring our model closer to **overfitting**.

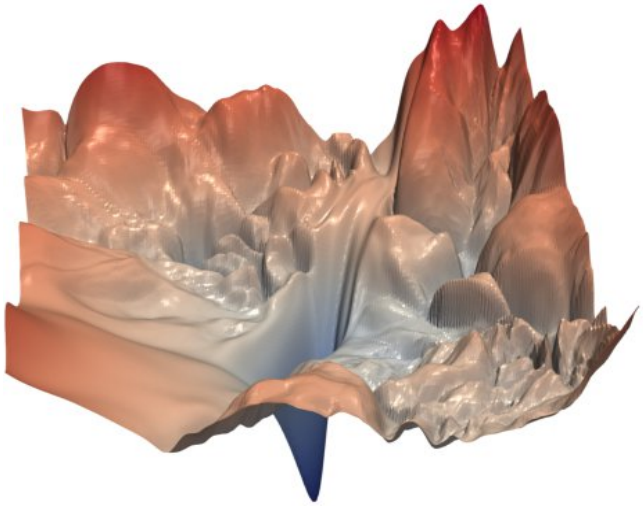
So what we want is a **good** learning rate, which will bring a fast enough convergence while keeping our model robust and able to generalize.

$$\underbrace{\theta}_{\text{Gradient}} = \theta - \underbrace{\eta}_{\text{LR}} \underbrace{\frac{\partial J(\theta)}{\partial \theta}}_{\text{Gradient}}$$

High Learning-rate



+ Larger steps (faster convergence)
+ Escape local minima
+ Good for early steps
- Risk skipping acceptable minima
- Risk "bouncing" over minima



Small Learning-rate

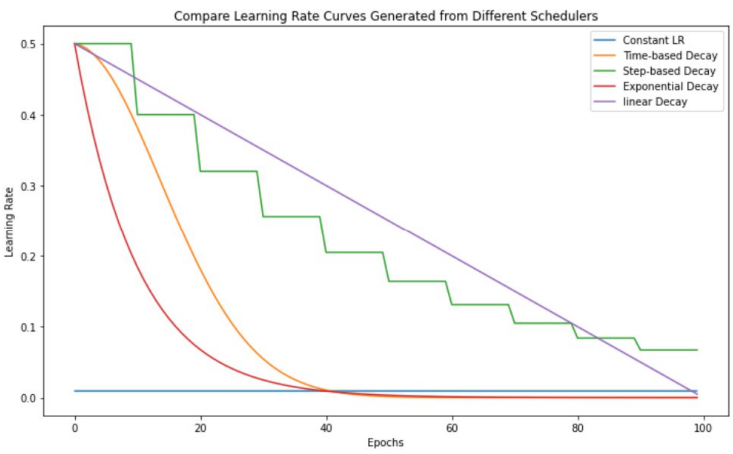


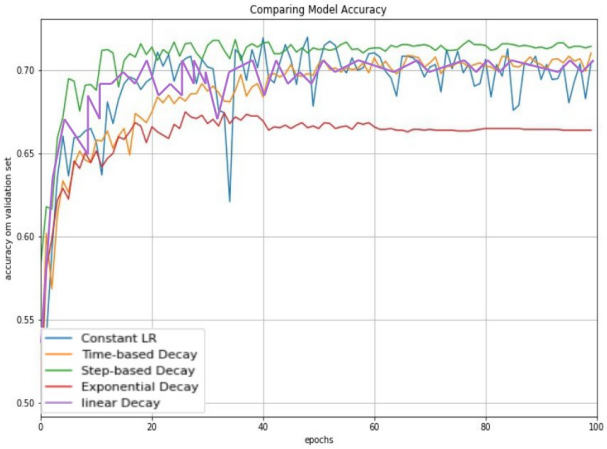
+ Don't skip local minima
+ Converge towards bottom of minima
+ Good for later steps
- Smaller steps (slower convergence)
- Risk getting stuck in local minima

Learning rate

However, the loss landscape (https://losslandscape.com/explorer) allows us to visualize that an initially high learning rate saves time by rapidly approaching the solution, whereas a lower learning rate allows for smaller steps and a more precise approach to the local minimum.

However, a learning rate that is too low at the start will result in very slow convergence, whilst a learning rate that is too high at the end risks taking us further away from the minimum.





Learning rate scheduler : decay

Fortunately, the learning rate does not have to remain constant throughout training. And based on what has just been said, we can choose to gradually decrease the learning rate during training to suit our needs.

There are several ways to do this:

Constant (blue) ==> Good performance but unstable throughout

Linear Decay (purple): The learning rate is reduced steadily with each epoch ==> Good performance, stable towards the end

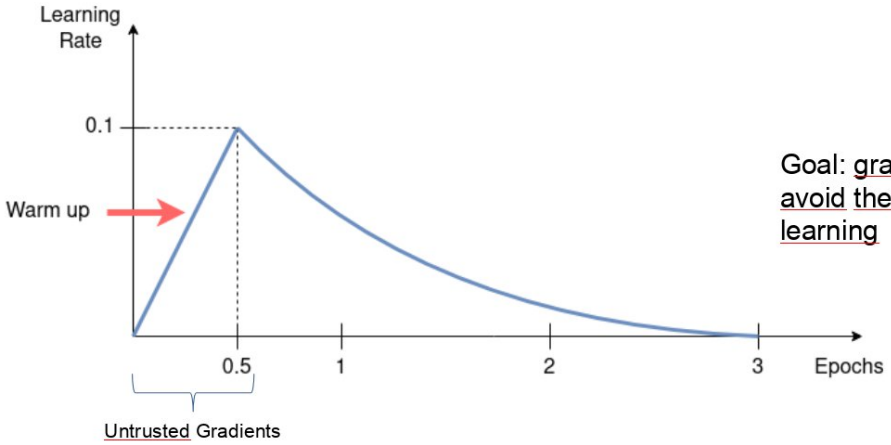
Step Decay (green): The learning rate is reduced by a fixed factor every x epochs ==> Quite similar to linear decay (in terms of profile and performance. Here, it converges more quickly to the optimum because it remains high for longer)

Time-based (orange): The learning rate is reduced by a factor inversely proportional to the number of epochs elapsed ==> The learning rate decreases rapidly, so it reaches the optimum more slowly than the others, but not too slowly to achieve the same performance

Exponential (red): The learning rate is reduced exponentially (quite simply)

$$\Theta_{t+1} = \Theta_t - \eta \nabla_\Theta [\mathcal{L}(\hat{y}_i, y_i)]$$

Problems : The first iterations have too much effect on the model (**significant losses**, high gradients, etc.)
A high learning rate can cause strong instability or divergence



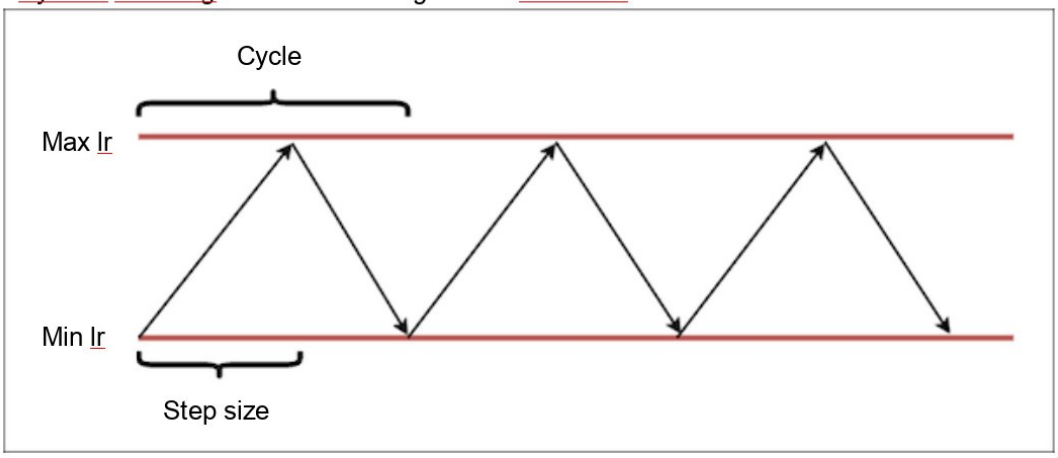
Goal: gradually increase the learning rate to avoid the risk of divergence at the start of learning

Learning rate scheduler : warmup

Another issue arised, where the initial iterations can still have too great an effect on the model's updates (high loss, steep gradients, bias, etc.).

To try to reduce the effects of these initial iterations, a warm-up phase (https://arxiv.org/pdf/2406.09405) can be added alongside the learning rate scheduler; this starts with a low learning rate and gradually increases it at the beginning of training.

Cyclical Learning Rates for Training Neural Networks - Leslie N. Smith 2017



Paramètres :

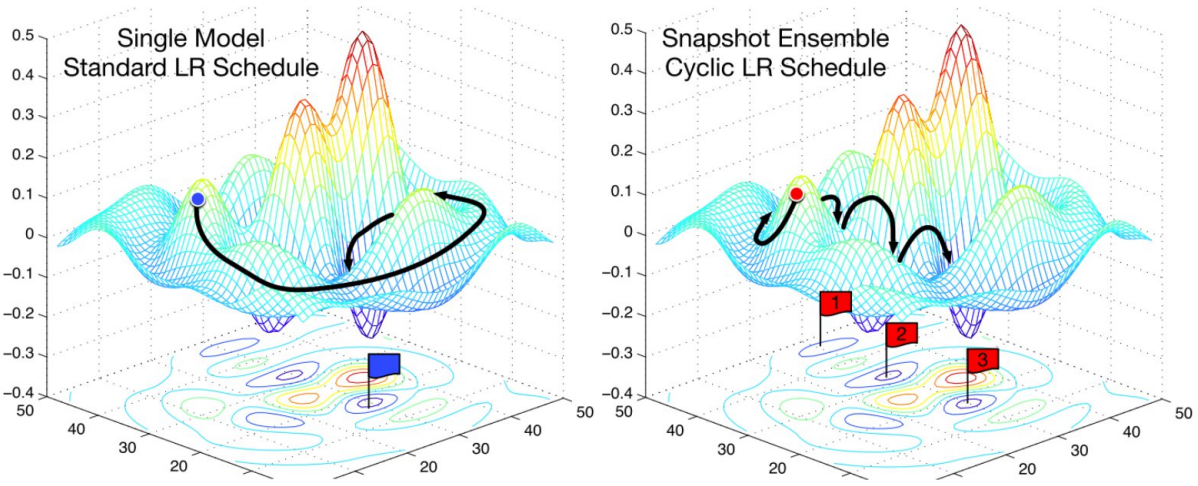
- Step_size = x * epoch (2 ≤ x ≤ 10)
- Base_lr -> min convergence value
- max_lr -> max convergence value

Succession of warmups and learning rate decays

Cyclic Learning Rate Scheduler

Building on the idea of a learning rate that is dynamically adjusted during training, Leslie N. Smith introduced the concept of the Cyclic Learning Rate Scheduler in 2017.

The principle involves alternating between warm-up phases and phases where the learning rate is reduced. The learning rate varies between a minimum and a maximum value and repeats these cycles, the duration of which is proportional to the number of iterations per epoch.



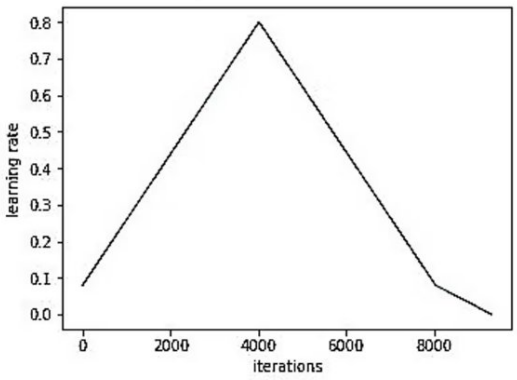
SNAPSHOT ENSEMBLES: TRAIN 1, GET M FOR FREE
Gao Huang, Yixuan Li, Geoff Pleiss

Cyclic Learning Rate

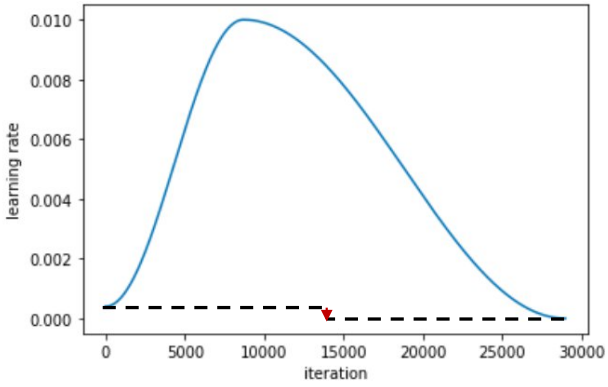
The main advantage of this technique is that it forces the model to easily escape from potential local minima in order to try and find a better one each time the learning rate increases. It also enables the model to escape from ‘saddle’ points with low gradients.

A disciplined approach to neural network hyper-parameters - *Leslie N. Smith*

Proposition initiale



cosine annealing : Recommandation par FastAI

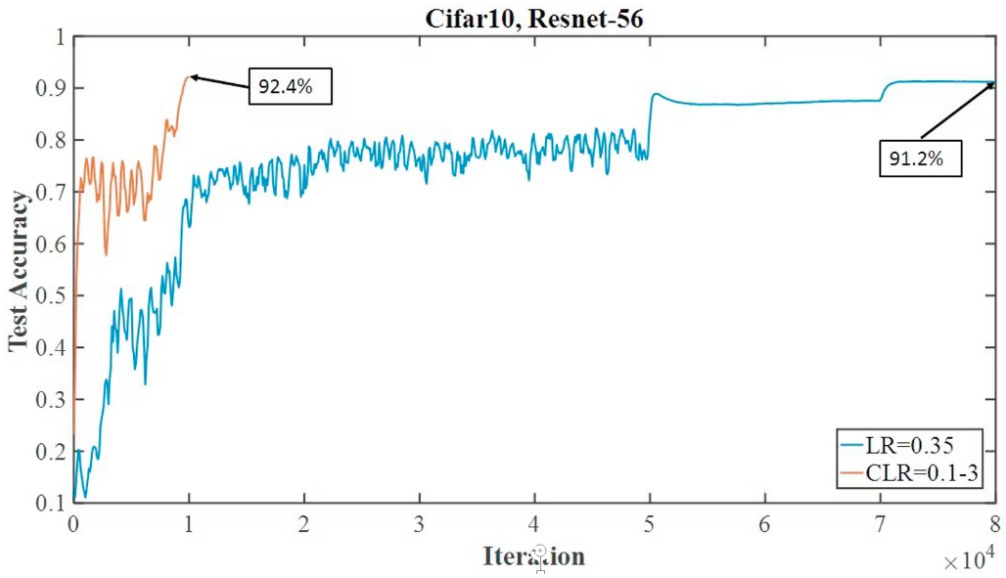


One Cycle is enough

In 2018, Leslie N. Smith realised that, rather than running several cycles, it is more efficient to run just one large cycle of increasing and decreasing the learning rate, known as the ‘One Cycle Learning Rate’. This leads to a ‘super-convergence’ effect, whereby the model is forced to converge rapidly with a **high learning rate** following a phase of gradual increase, followed by a phase of decrease towards **a learning rate lower than the initial LR**.

The original proposal involved a linear increase and decrease, but FastAI later proposed a more efficient variation using a sinusoidal curve. In both cases, the cycle ends with a significant reduction in the learning rate during the final iterations.

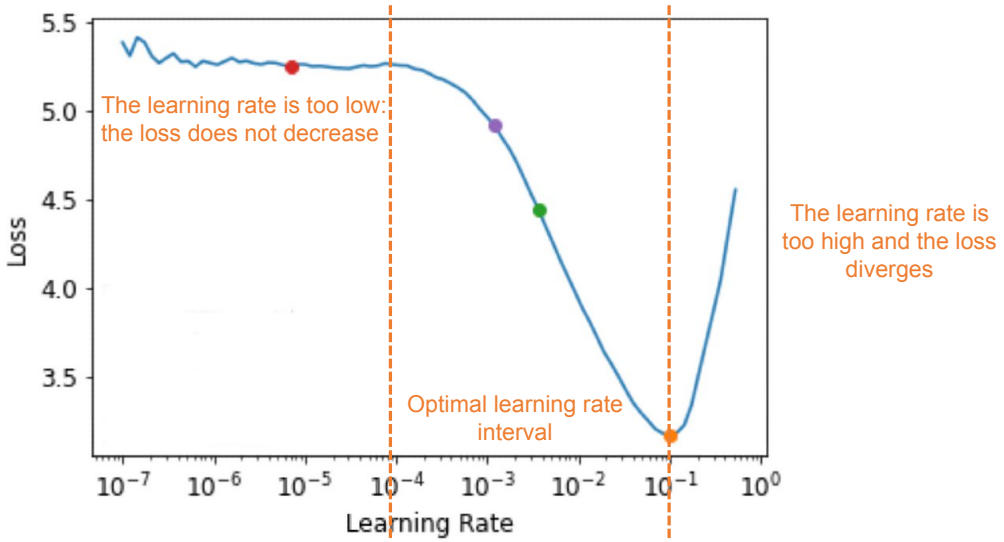
Faster convergence for equivalent final precision



Cyclic learning rate scheduler

In his paper, Leslie N. Smith demonstrates that, using a ResNet 56 network and the CIFAR-10 dataset, convergence is achieved more quickly with a cyclic learning rate than with a constant learning rate, whilst achieving a similar level of accuracy.

Goal: Find the **optimal learning rate** values for your model, particularly for **the maximum value** of a *cyclic scheduler*



Learning Rate Finder

To find the minimum and maximum values of the cyclic learning rate, we use the learning rate finder technique.

We will train a model for a few epochs, starting with a very low learning rate which will gradually increase until it reaches a very high value.

By plotting the loss curve for this short training run, we will be able to observe two things:

- Initially, the loss does not decrease until we reach a learning rate high enough to have a positive effect on the model. This value will be the minimum.

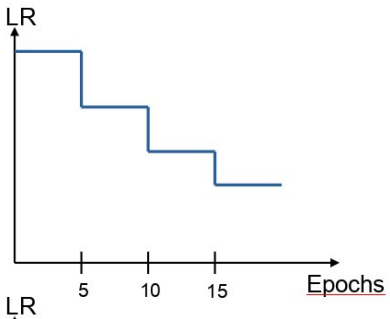
- The loss decreases fairly steadily until it begins to fluctuate significantly and diverge. We can therefore find the maximum learning rate that has a positive effect on the loss without causing it to diverge.

Each **scheduler** has *its own settings*

```
import torch.optim as opt

scheduler = opt.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

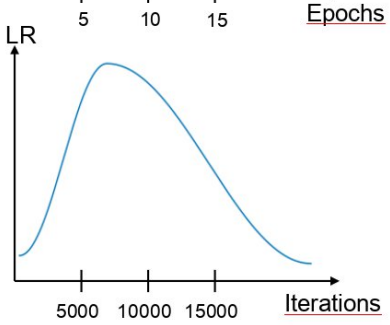
for epoch in range(100):
        train(...)
        validate(...)
        scheduler.step()
```




```
import torch.optim as opt

scheduler = opt.lr_scheduler.CyclicLR(optimizer, base_lr=0.01, max_lr=0.1)


for epoch in range(10):
        for batch in data_loader:
                train_batch(...)
                scheduler.step()
```

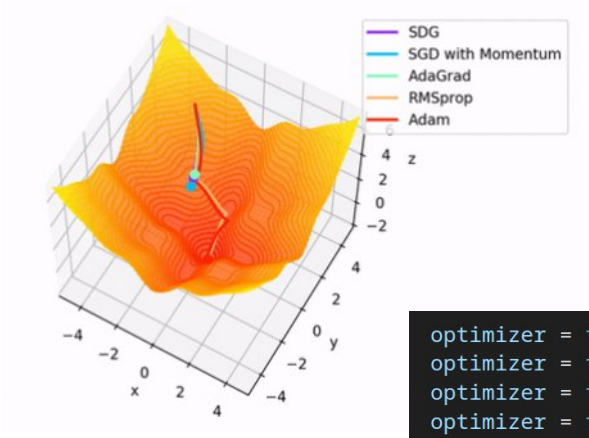


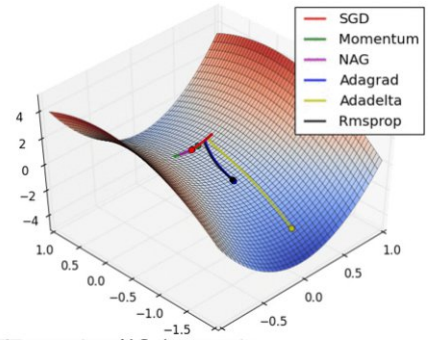
Learning rate scheduler implementation

All the types of learning rate schedulers we have looked at in this course are implemented in the PyTorch library.

It should be noted that cyclic learning rate schedulers update their values with every new batch, unlike the others, which update their values at the start of each epoch.

https://www.makerluis.com/content/images/2023/12/Advanced_Optimizations_Saddle_optim-1.gif
https://medium.com/@LayanSA/complete-guide-to-adam-optimization-1e5f29532c3d






```
optimizer = torch.optim.Adadelta
optimizer = torch.optim.Adagrad
optimizer = torch.optim.Adam
optimizer = torch.optim.AdamW
optimizer = torch.optim.Adamax
optimizer = torch.optim.ASGD
optimizer = torch.optim.LBFGS
optimizer = torch.optim.NAdam
optimizer = torch.optim.RAdam
optimizer = torch.optim.RMSprop
optimizer = torch.optim.Rprop
optimizer = torch.optim.SGD
```

Optimizers

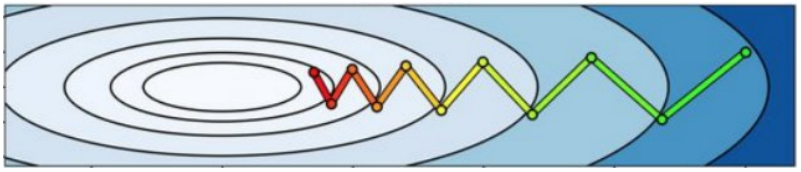
Now let's talk about another hyperparameter, which is not a numerical value: the optimizer.

The optimizer is the algorithm which dictates how we update our model weights. Known optimizer include the **Stochastic Gradient Descent (SGD)**, **Adam**, and **AdamW**.

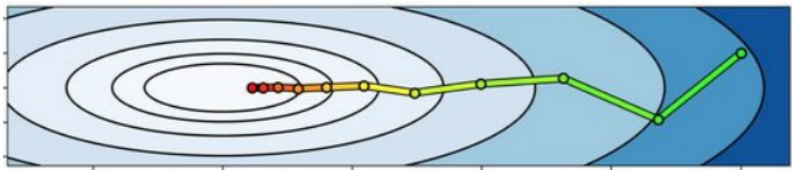
The latter is the most well known and generally the better choice no matter the task.

https://ai.stackexchange.com/questions/10013/what-is-a-bad-local-minimum-in-machine-learning
https://jermwatt.github.io/machine_learning_refined/notes/3_First_order_methods/3_8_Momentum.html

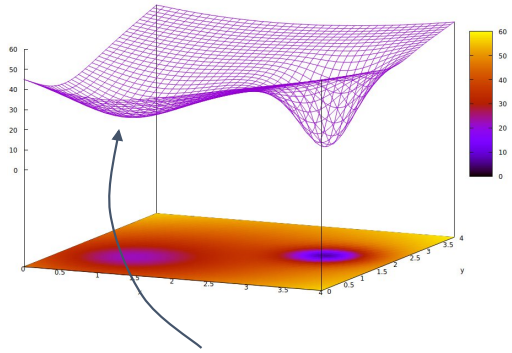
Convergence



Without momentum



With momentum



Local minimum

Optimizers : Momentum

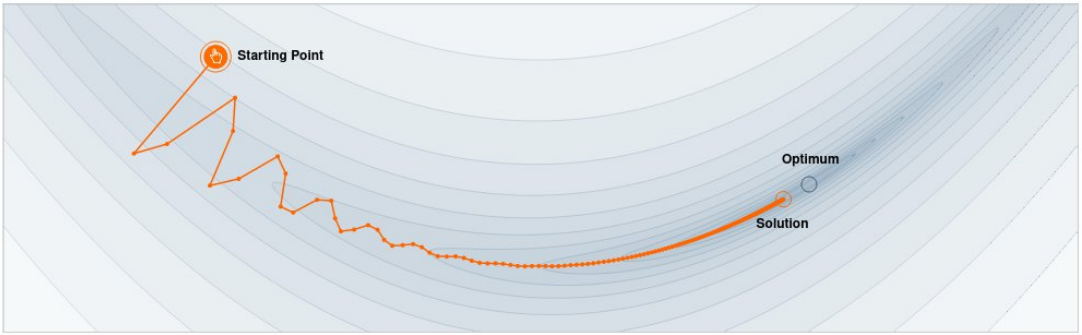
Optimizers will often bring thei own numerical hyperparameters, and one of them is the momentum.

Momentum is a value which keeps a « memory » of past gradients and applies them to the current weight update.

As its name suggests, it is analogous to the physical momentum (inertia), where an object sliding down a slope will keep its speed when reaching a bump or turn and not follow the slope exactly.

Momentum makes the gradient descent less sensitive to the non-convexity of the loss landscape and will generally cause a faster convergence, but might also make us overshoot our target.

https://distill.pub/2017/momentum/



Step-size α = 0.02

0 0.003 0.006

Momentum β = 0.99

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

Why momentum works ?

Website https://distill.pub/2017/momentum/ has ressources explaining and illustrating momentum in details.

$$\Theta_{t+1} = \Theta_t - \eta \nabla_\Theta \left[\mathcal{L}(\hat{y}_i, y_i) + \lambda R(\Theta_t) \right]$$

Updated weights = Weights before update − Learning rate * Gradient [Cost function (Prediction , Label) + Regularization rate * Regularization function (Weights before update)]

Weight update equation

- L1 Regularization
- L2 Regularization
- Max norm Regularization
- Regularization with the cost function
- Dropout

L1 : LASSO	L2 : Ridge
$\lvert\Theta\rvert$	Θ^2

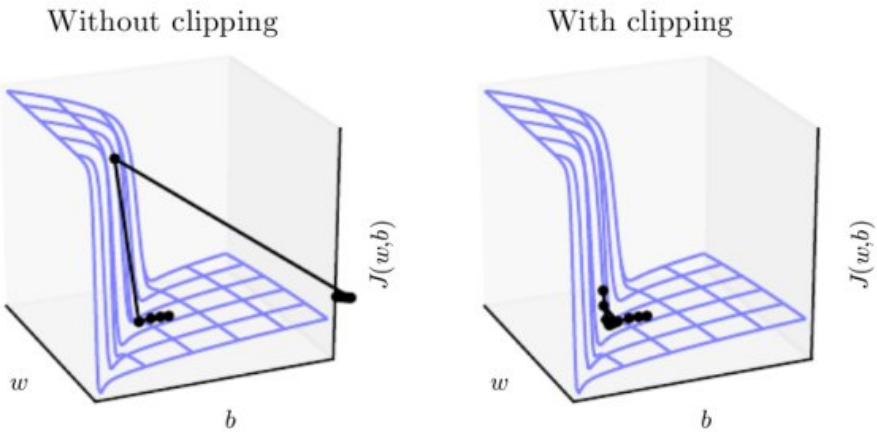
Regularization : L1 and L2

Here we will talk about more classical regularization methods which apply inside the weight update equation, as well as Deep Learning-specific ones.

– L1 and L2 regularization apply a numerical penalty to weights, based respectively on the **Absolute Values** or the **Squared Values** of weights.
  – L1 will bring weights closer to absolute 0 and encourages **sparsity** (only a few wieghts do the work, others are set to 0). It is used to select a subset of relevant predictors.
  – L2 will reduce weight values but not set them to 0. It will prevent overfitting but retain all predictors (weights).

Dropout is another regularization method for neural networks we will see in a moment.

https://neptune.ai/blog/understanding-gradient-clipping-and-how-it-can-fix-exploding-gradients-problem

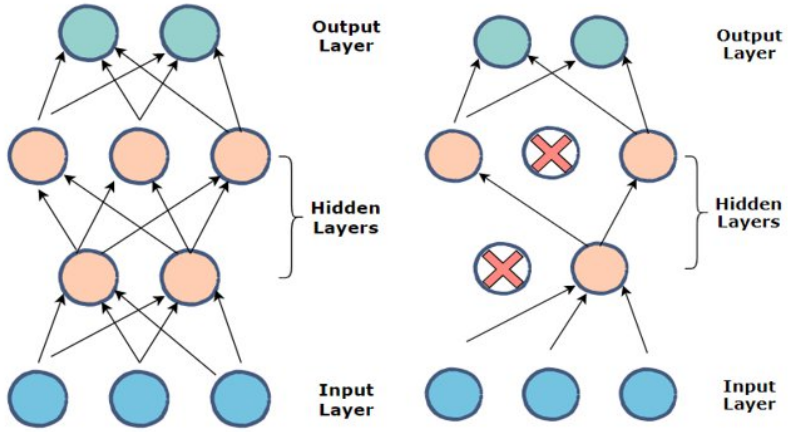


Regularization : Gradient clipping

Gradient clipping involves reducing overly high gradient values based on a predefined threshold.

Hgh gradients can cause instabilities during training. By scaling them down we prevent those instabilities and make our training more robust.

https://medium.com/unpackai/introduction-of-dropout-and-ensemble-model-in-the-history-of-deep-learning-a4c2a512dcca



Original network drop out networks

Regularization : Dropout

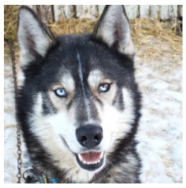
Dropout is a technique used to force the model to use all of its weights by randomly « cutting off » some of the neurons during a step.

Source :
pytorch.org
data-flair.training
Right for the Right Reason: Making Image Classification Robust





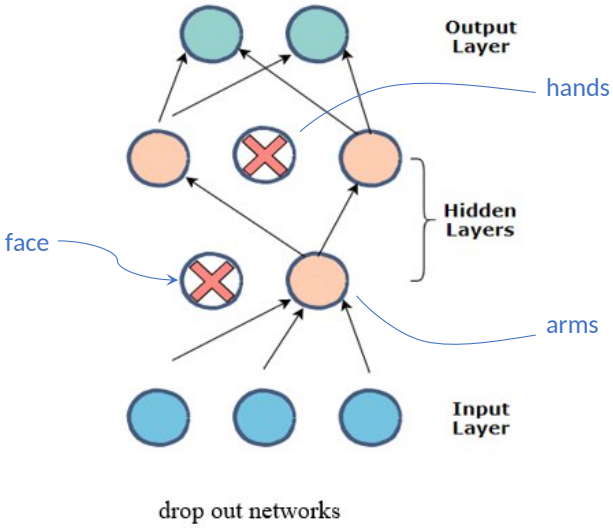
Human = face ?



(a) Husky classified as wolf



(b) Explanation



drop out networks

Regularization : Dropout

Why use dropout ?

While it's difficult to analyze why exactly a neural network predicts what it predicts, some people have done so and realized that, sometimes, a neural network will use only a handful of neurones for its prediction because what they specialized in was sufficient to get a good enough performance during training.

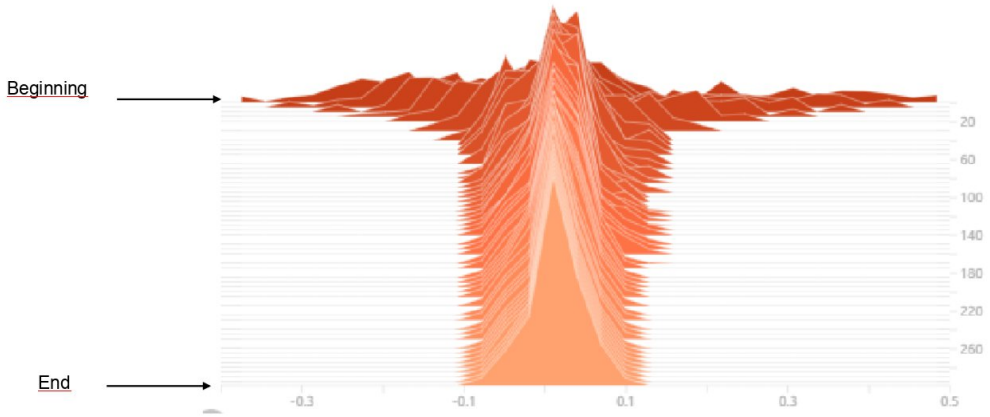
Let's take for example a model trained to recognize wolves and dogs. The model would work well in training, but when confronted to a picture of a husky, would classify it as a wolf.

This is because in the training set, pictures of wolves were almost always in snow, while dogs weren't. The model thus only relied on the « snow = wolves » relation and other weights where secondary in the decision process.

By including dropout in the training, the output of some neurones, including the « snow neurones » were randomly set to 0 meaning those are not updated, forcing the model to rely on the other weights.

A neural network that **converges and generalizes correctly*** generally has weights **that tend to 0**.
*(neither underfitting nor overfitting)

Distribution of weights during learning:



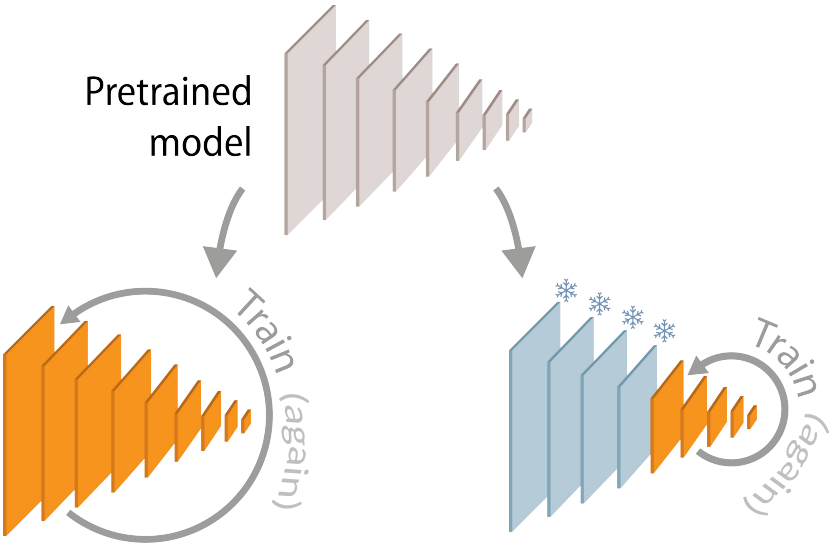
Weight Decay

$$\text{Loss} = L_\theta + \lambda \sum \theta^2$$

Finally, weight decay is a fancy deep-learning name for the **L2 Ridge-regression** regularization method.
We observed that models which performed well had the absolute value of their weights closer to 0.
By applying weight-decay, we penalize overly-high weights and scale them closer to 0.

1. What **work** can be done to **improve** the **data** used for training?
2. How can a model be **evaluated**?
3. Is it possible to make the training more **robust**?
4. Can we **benefit** from an **already** trained model?
5. Bonus: Good practices

In this section we will show you how you can benefit from a pre-trained model instead of training your from scratch.



Hugging Face

Transformers

Timm

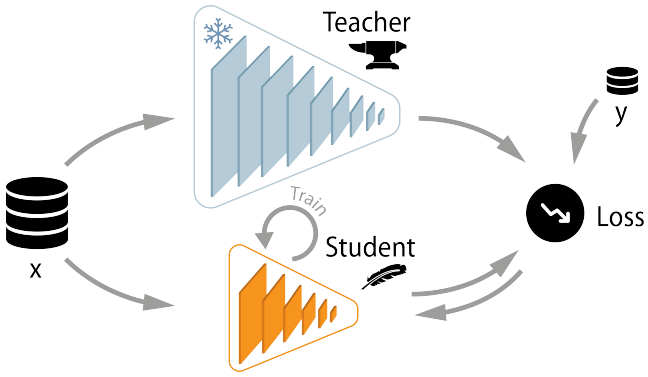
MMCV

Fine tuning & Transfer Learning

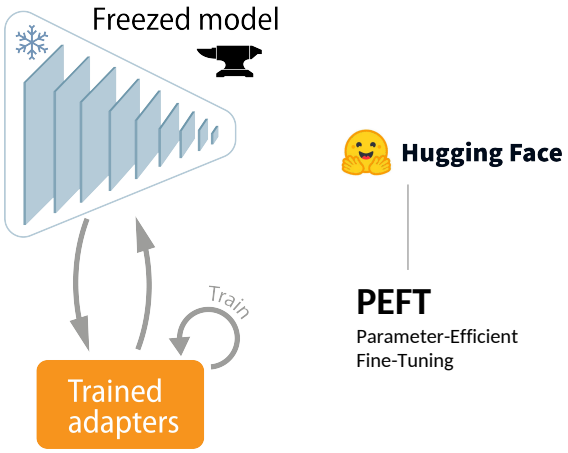
Let’s imagine we have a model already trained on a large dataset, like ImageNet, which is indeed a real-world example for common pre-trained models.

If we want to train a model to recognize specific species of kangaroos, we do not have to traing it from 0 : we can take a model which is already trained to recognize shapes and colours and animals, and simply retrain it entirely, aka **Fine Tuning**, or freeze most of the weights and retrain only the « head » of the model, which is where the classification happens (aka **Transfer Learning**).

https://huggingface.co/



Knowledge distillation



Adapters

Transfer learning

Other transfer learning methods include :

– Distillation : Instead of training a model by comparing its predictions to the ground truth, we also compare it to the predictions of a larger, pre-trained « teacher » model. Through this, we obtain a smaller « student » model with performance comparing to the larger teacher model but more size and compute-efficient.

1. What **work** can be done to **improve** the **data** used for training?
2. How can a model be **evaluated**?
3. Is it possible to make the training more **robust**?
4. Can we **benefit** from an **already** trained model?
5. Bonus: Good practices

Finally, we'll list some best practices for your deep learning projects.

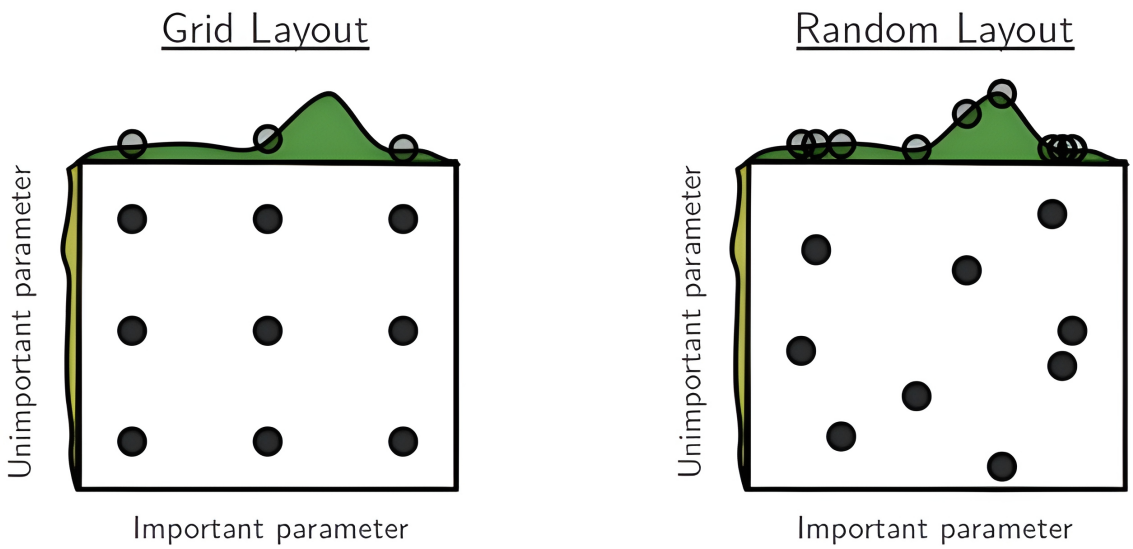
Hyper-parameters	Methods
• Learning rate • Regularization • Optimizer • Model architecture • Batch size • …	• Manual research • Grid search • Random research • Gradient • Evolutionary algorithms • …

HPO (Hyperparameter Optimization)

Find the good hyper-parameters

As previously mentioned, hyper-parameters are values which affect the quality of training. FInding the optimal values can be difficult and especially costly, since it will involve multiple training with different values, which is called « Hyperparameters Optimization » (or HPO).

https://dl.acm.org/doi/pdf/10.5555/2188385.2188395

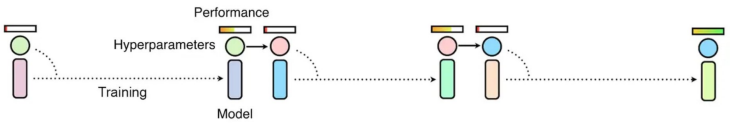


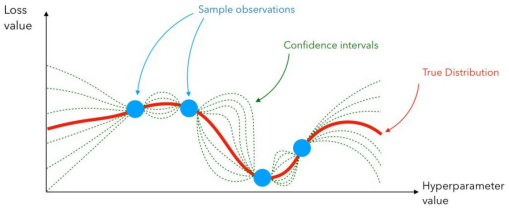
HPO : Grid & Random

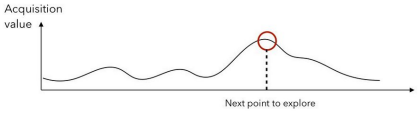
The naive way to do HPO is to brute-force it with a **Grid search**. In addition to being quite costly, it risks missing actually relevant places in the space of hyperparameters and sampling in irrelevant ones.

Instead, counter-intuitively, a **Random search** will perform better.

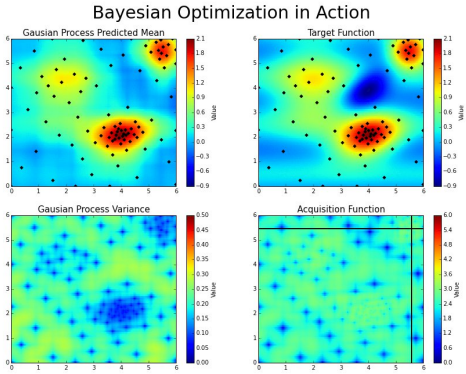
https://maelfabien.github.io/assets/images/bo.gif
https://maelfabien.github.io/machinelearning/Explorium4/#acquisition-function







- Tree Parzen Estimator / Gaussian Process
- Sequential but allows to quickly find the global optimum.
- Proposes a new set of hyper parameters based on the scores obtained by the previous ones tested.



HPO : Bayesian Optimization

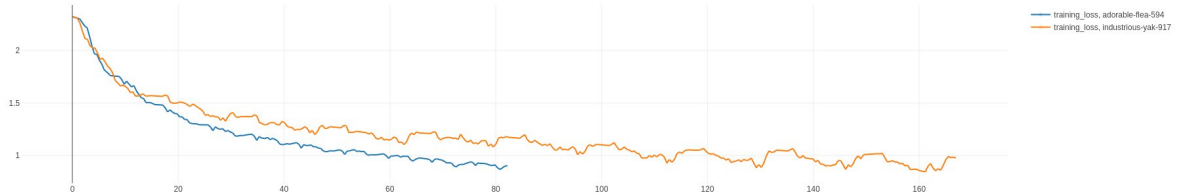
There are also methods that make use of training runs already carried out to restart training runs with more relevant values.

For our example here, we apply Bayesian optimisation using a Gaussian distribution, which will give us the points we are looking for.

Table view | Chart view | metrics.rmse < 1 and params.model = "tree" | Sort: Created | Columns | Refresh

Time created: All time | State: Active

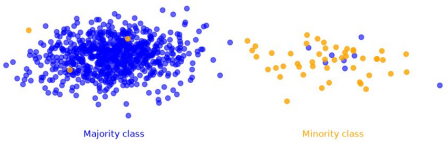
			Metrics			Parameters					
Run Name	Created	Duration	test_accuracy	training_loss	batch_size	epochs	learning_rate	momentum	weight_clampir	world_size
adorable-flea-594	1 day ago	1.5min	58.73	0.935	32	10	0.01	0.9	False	2
righteous-calf-205	1 day ago	1.5min	61.84	0.738	32	10	0.01	0.9	True	2
loud-dog-130	1 day ago	2.8min	55.87	1.303	32	10	0.01	0.9	True	1
industrious-yak-917	1 day ago	2.9min	57.57	0.915	32	10	0.01	0.9	False	1



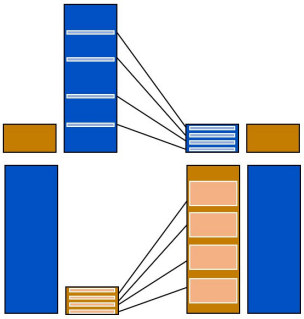
Collaboration & Reproducibility

There are tools developped to support collaborative work in deep learning, especially work involving comparative experiments such as HPO or architecture search. Tools such as **mlflow**, **weights&biases** or **tensorboard** provide easy-to-use graphboards with summaries of each run and comparison graphs to easily identify successful and unsuccessful runs.

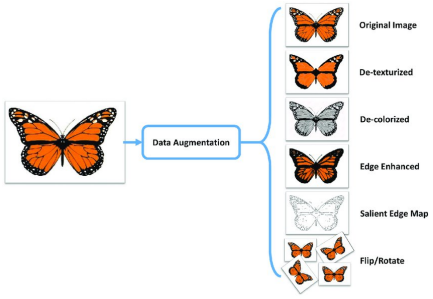
Data: Visualize them, understand them, curate them



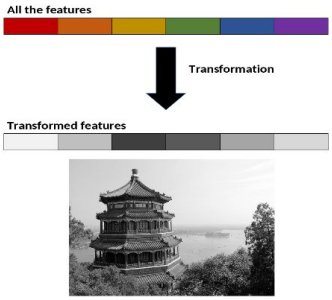
Be **aware** of irregularities (class imbalance, lack of diveristy)



Under/over-sampling



Data augmentation



Features Engineering

Use **data augmentation** techniques to solve them

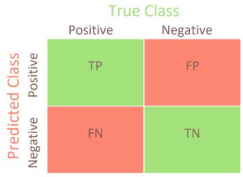
Conclusion and takeaway

Metrics: they should be chosen accordingly



All negative:
accuracy = 99%

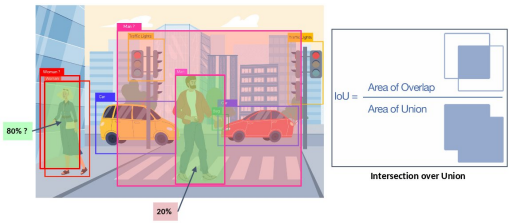
Avoid catch-all, non-representative metrics.



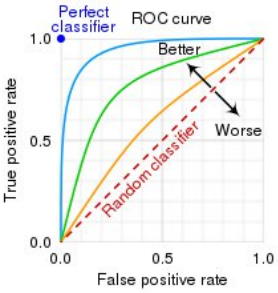
Confusion matrix

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$



Intersection over Union



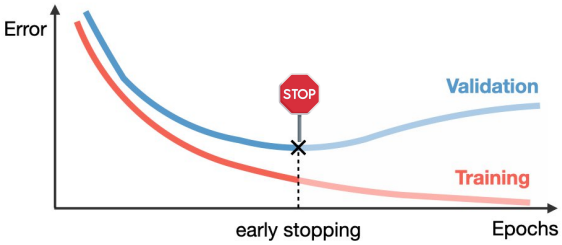
ROC Curve

Use **corresponding** metrics and **combinations** of metrics to identify where the model does and does not perform.

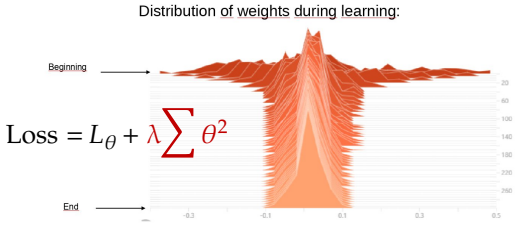
A **loss** is not a metric, and a metric is not a loss. A loss is differentiable and evaluated on a few sample for training purposes.
A metric is a non-differentiable tool for evaluation purposes.

Conclusion and takeaway

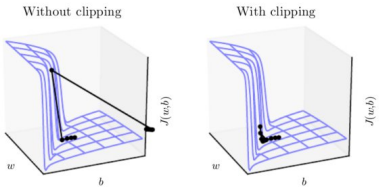
Overfitting : A model which performs well on the training set might actually not work under real-world conditions



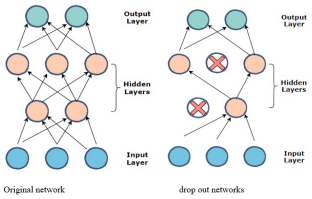
Identify when the model is starting to overfit and stop the training.



Weight decay



Gradient clipping



Dropout

Use **regularization** techniques to help prevent overfitting in the first place

Conclusion and takeaway