



Basics of Machine Learning



Ana Peixoto (**University of Washington**)
HSF-India HEP Software Workshop
University of Hyderabad
16th January 2025



About me

Natural from Paredes de Coura, Portugal

2015: Joined the ATLAS Experiment working on top FCNC

2016: Finished Master in Experimental Physics with the Universidade do Minho (Portugal)

2021: Finished PhD on the search for new interactions in the top quark sector in UM (Portugal) - Won one of the [ATLAS PhD Grant](#)

2021-2023: Postdoc in the Laboratoire de Physique Subatomique et de Cosmologie (France)

2023-Present: Postdoc with the University of Washington (United States of America)

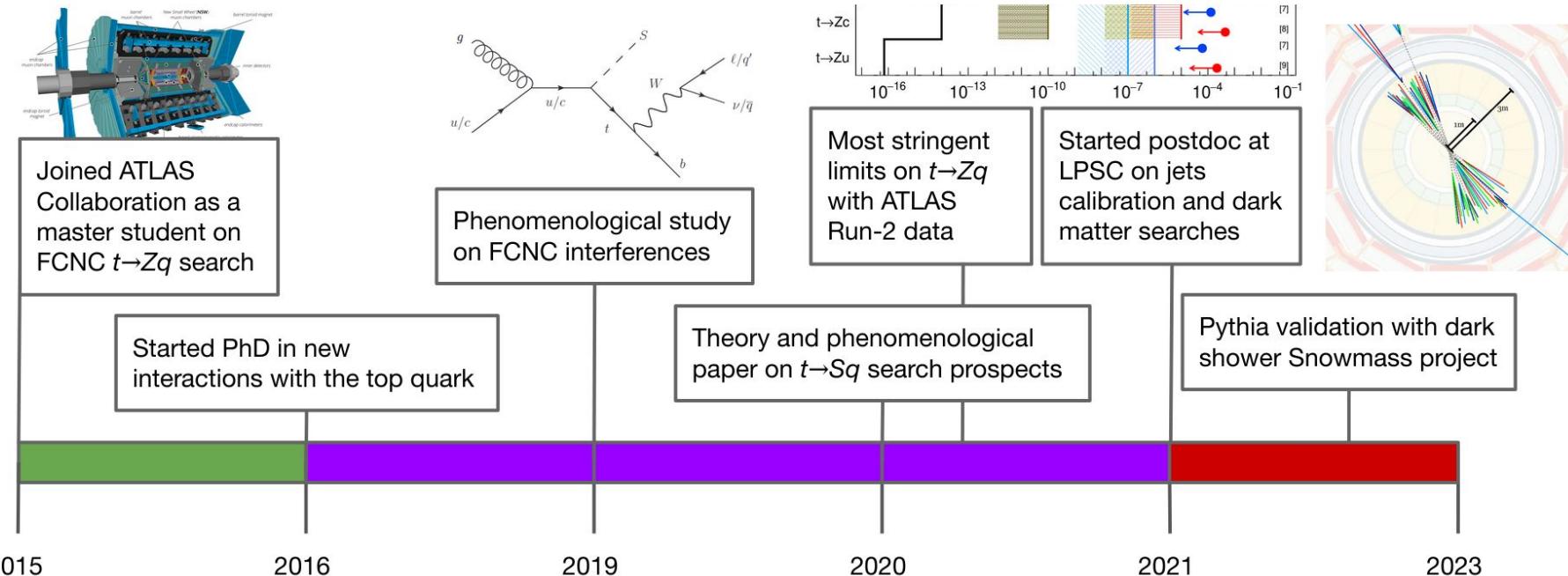


Email:

ana.peixoto@cern.ch

LinkedIn:[ana-peixoto-hep](https://www.linkedin.com/in/ana-peixoto-hep)

About me



Besides Physics, I also love listening to Jazz, dancing, practicing Martial arts and reading

Machine Learning

Without Machine Learning



With Machine Learning

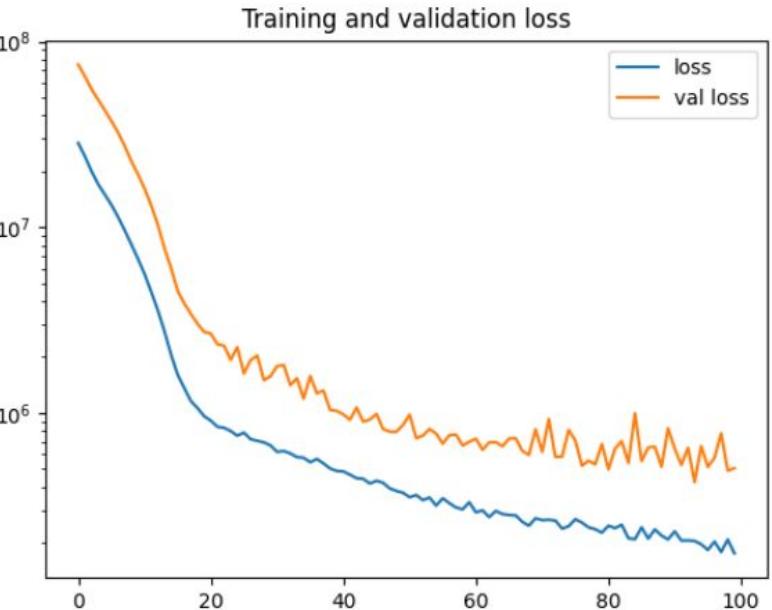
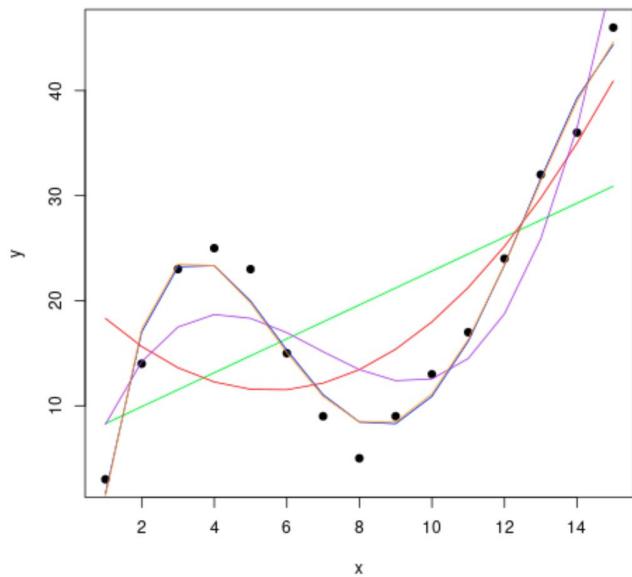
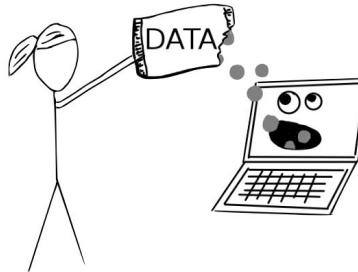


Machine Learning

Without Machine Learning



With Machine Learning



Machine Learning

Without Machine Learning

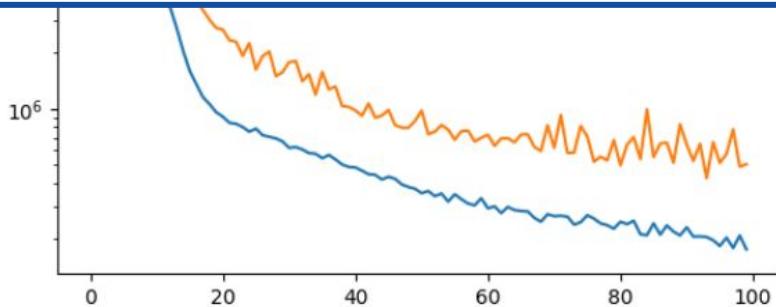
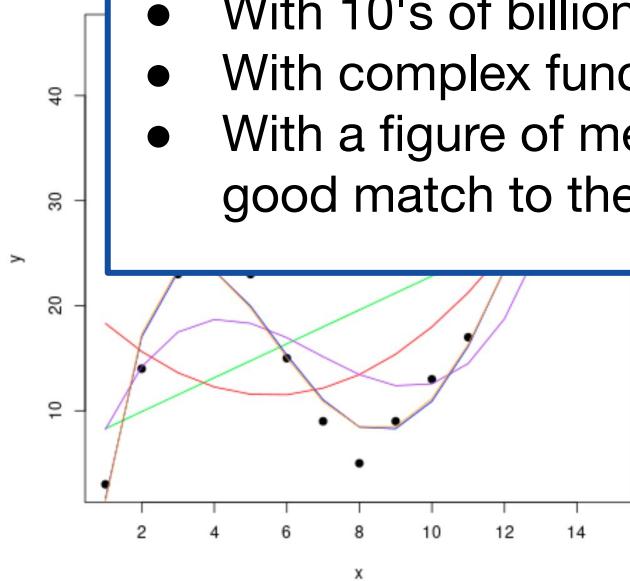


With Machine Learning



Machine Learning is just a function fit, in the extreme!

- With 10's of billions of data points
- With complex functions with millions of parameters
- With a figure of merit that tells you when the function is making a good match to the data



Inference

Fit of a function

$$y_i = f(x_i)$$

Where

y_i = A prediction for the i th row (0 – Background and 1- Signal),

x_i = The i the row of our feature data

$f(x_i; \Theta)$ = Our NN (that can be fairly complicated)

- Some function with parameters
- Straight line: $ax + b$
- Much more complex with millions of parameters!
- All the parameters are usually denoted Θ

Parameters definition

How do we fit a straight line?

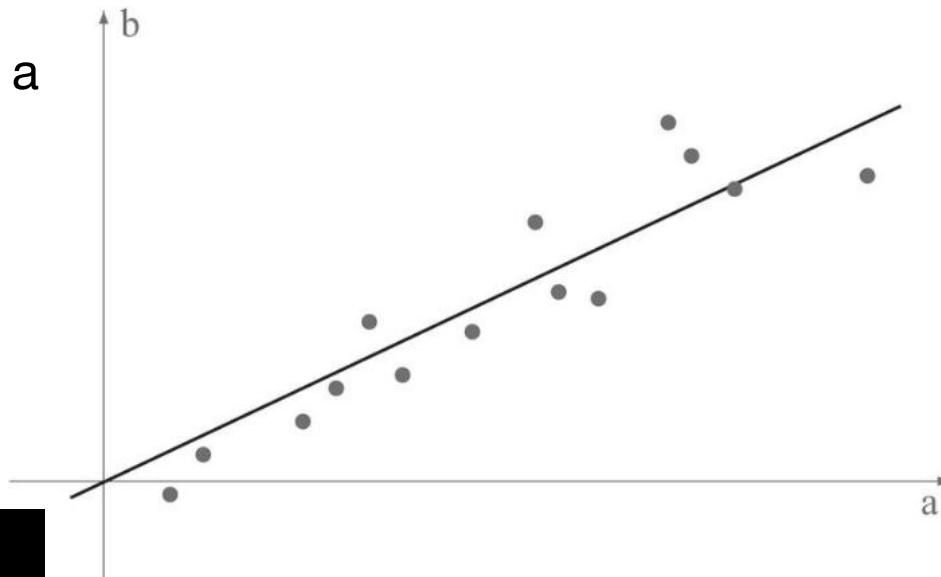
1. Define a mathematical criteria for a good fit:

$$r = \sum (x_i - y_i)^2$$

(the **Loss** function)

2. Minimize r (find Θ such that):

$$\frac{dr}{d\Theta} = \frac{d}{d\Theta} \left(\sum (x_i - y_i)^2 \right) = 0$$



Loss function choice

What do you want to optimize?

- Signal and background separation
- Measured mass
- Decorrelation of two outputs that separate signal and background
- And more!

What must the loss function do?

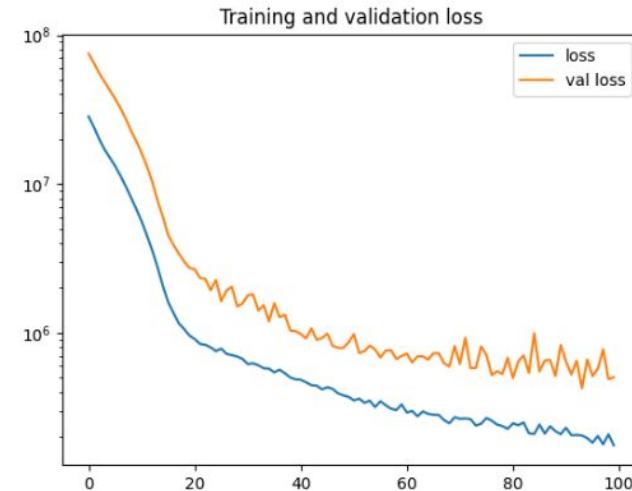
- Return a value – “figure of merit”
- Some “distance” between perfect fit and the current function

Loss function choice

Mean squared loss

- Works very well for regression problems

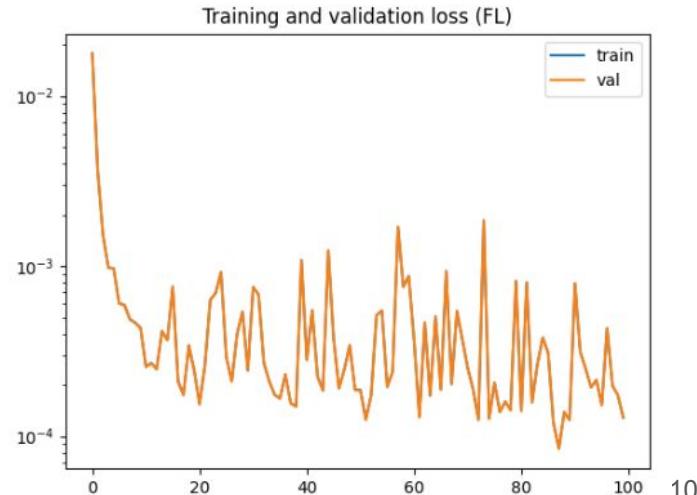
$$r = \sum (x_i - y_i)^2$$



Cross entropy loss

- Works better for classification!

$$r = \frac{1}{N} \sum (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$



Problem types

Regression:

- Continuous output
- Jet Energy Calibration
- Mass of the Higgs

Classification:

- 1 or 0 type output: cat vs dog, etc
- Is it signal or background?



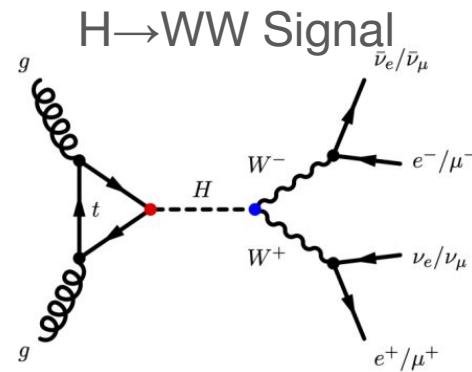
Problem types

Regression:

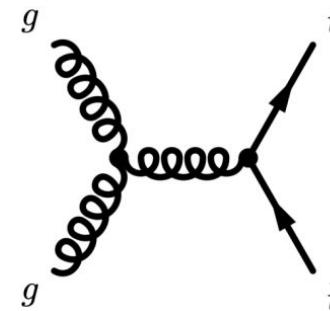
- Continuous output
- Jet Energy Calibration
- Mass of the Higgs

Classification:

- 1 or 0 type output
- Is it signal or background?
- Is it signal, QCD background,
or Beam Induced Background (as an example!)



Background



Neural Network

A Neural Network is a single neuron!

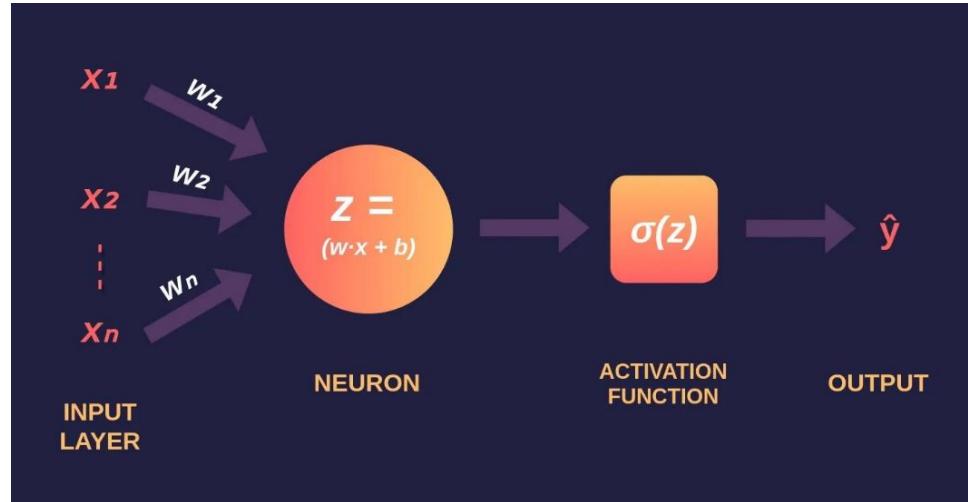
$$\hat{y} = \sigma(W \cdot x + b)$$

σ = activation function

W = weights

x = inputs

b = offsets



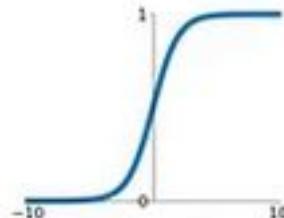
Activation functions

Form	Name	Pros	Cons
$\sigma(z) = \frac{1}{1 + e^{-z}}$	Softmax	Differentiable, finite range	Vanishing derivative at $\pm\infty$
$\sigma(z) = \max(0, z)$	Rectified Linear Unit (ReLU)	Computationally Efficient, does not saturate on one side	Dead Neuron, unbounded on positive side
$\sigma(z) = \ln(1 + e^z)$	Softplus	Vanishing derivative is better handled	Computationally expensive

Activation Functions

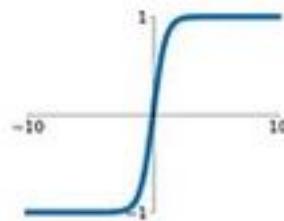
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



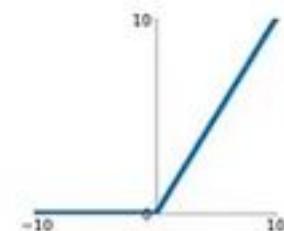
tanh

$$\tanh(x)$$



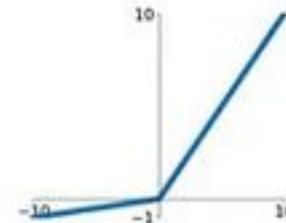
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

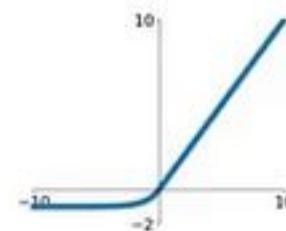


Maxout

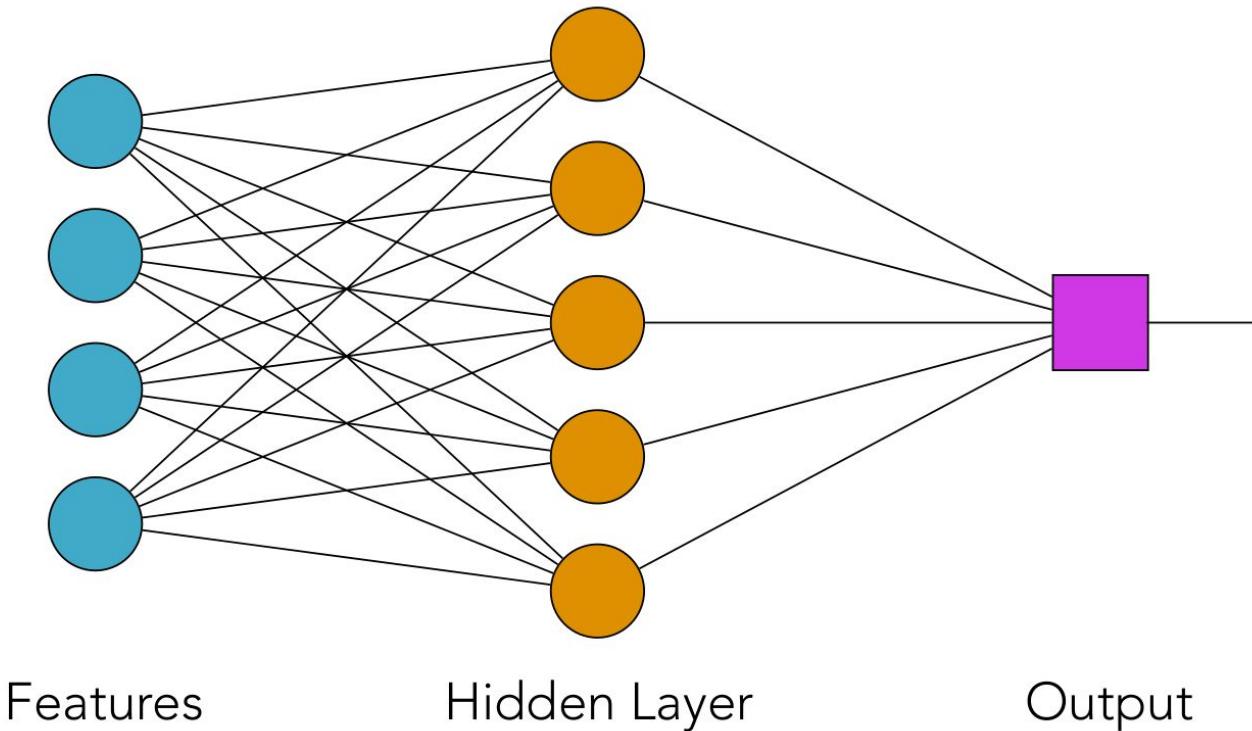
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural Network



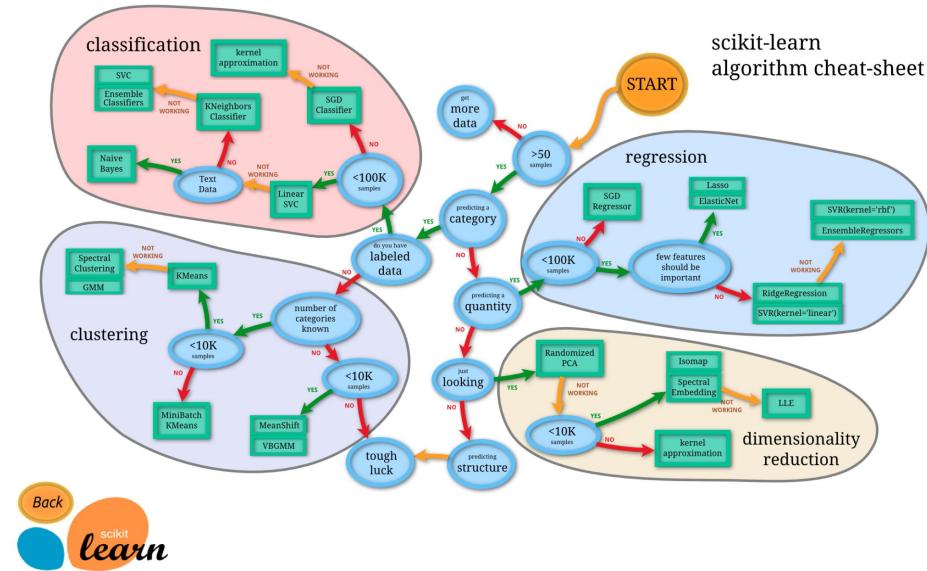
Neural Network is not always the answer!

Non-Neural Network forms of Machine Learning:

- Great for limited number of inputs (features)
- Great for high level features (angles between jets)
- Great for small number of features (> 100)

Neural Network forms of Machine Learning:

- Great for large numbers of inputs
- Great for patterns in detector data
- Great for low-level data
- Great at Geometrical or Variable length inputs



Neural Network software

All the scientific frameworks for Deep Learning Machine Learning are written in Python:

TensorFlow

- Developed by Google, managed as open source.
- Not used as much internally but has one of the most active user communities.
- API is most friendly to new users.

PyTorch

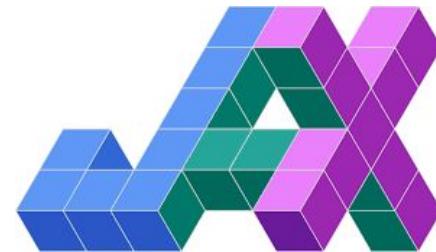
- Developed by Facebook, actively used.
- Faster than TF
- Is also a framework, but not quite as easy to use for a beginner.

JAX

- Developed in Google's DeepMind, used for most (all?) of their research
- Great when you want to do something unique or break open the box.

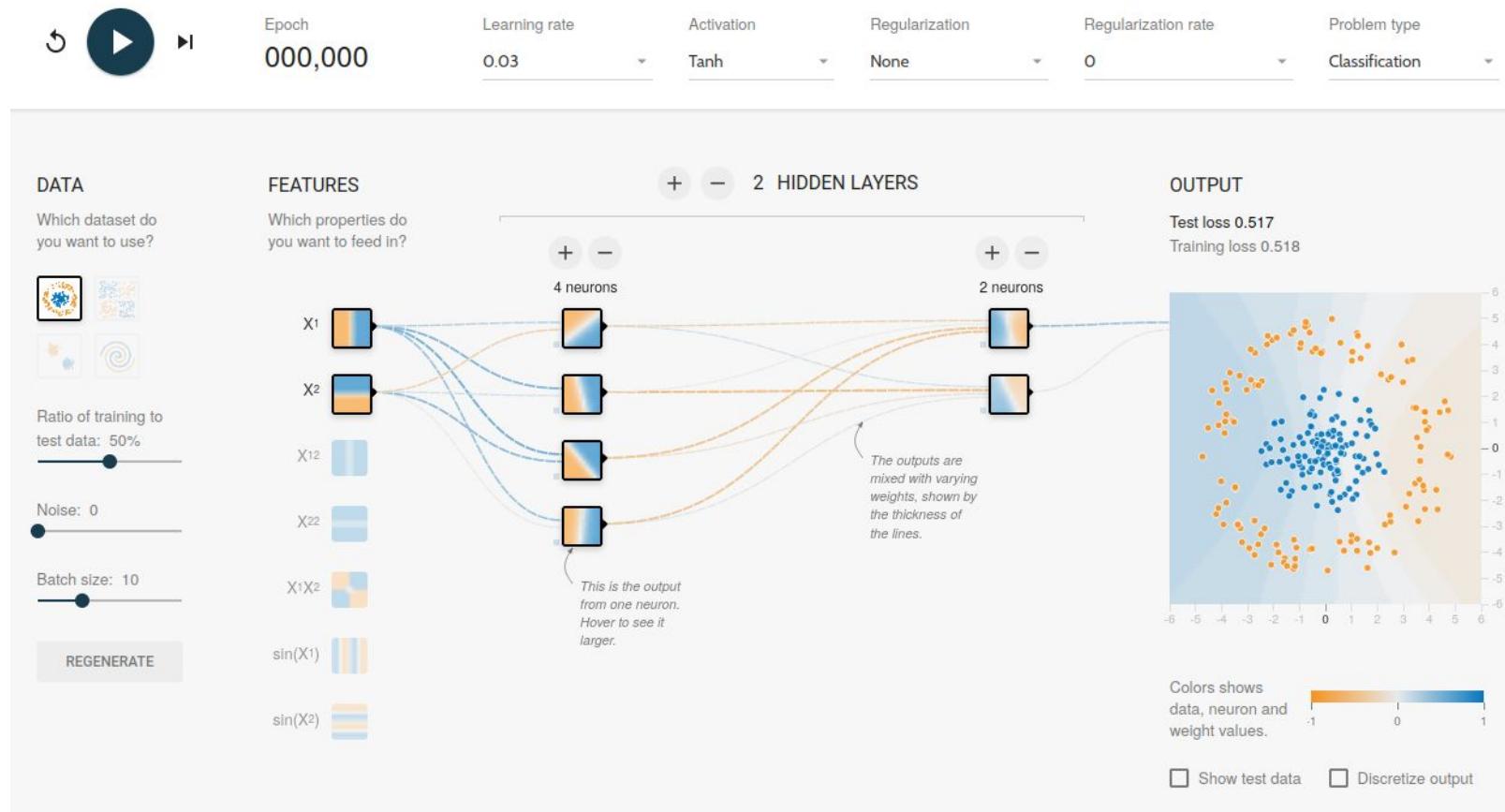


TensorFlow



Let's play with NNs!

<http://playground.tensorflow.org>



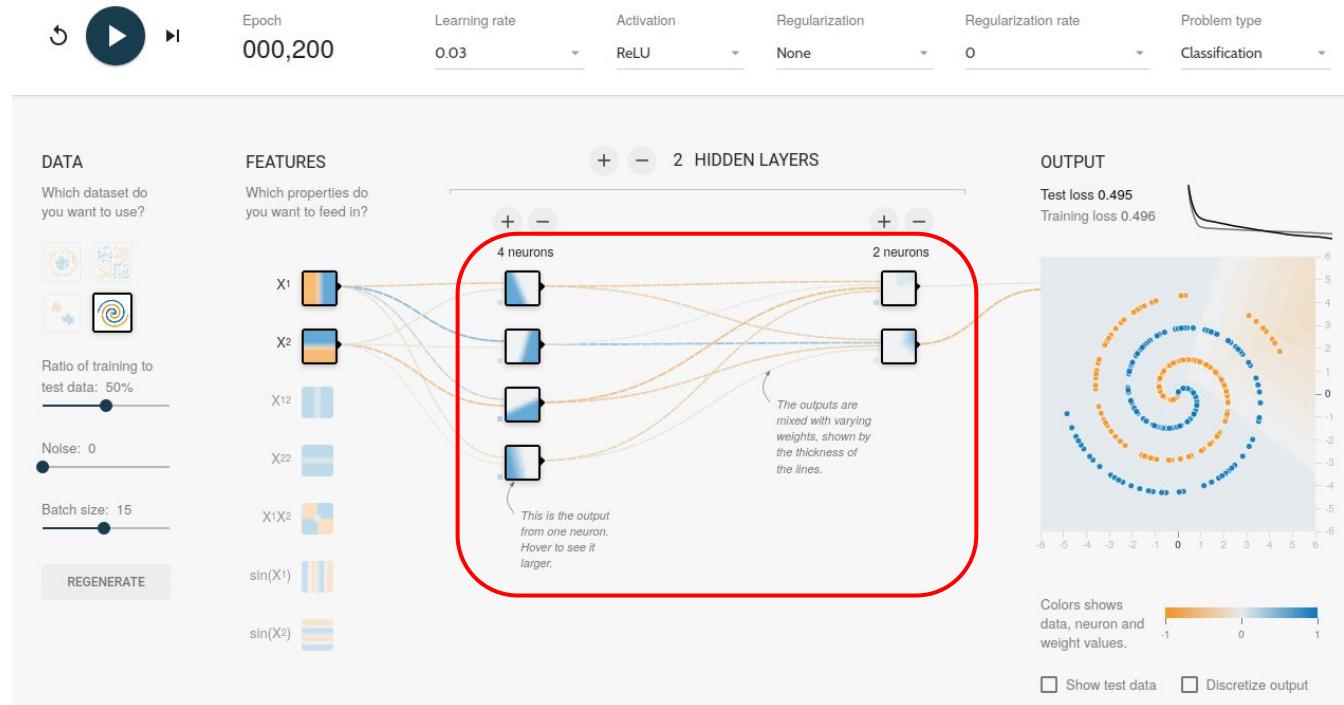
Hyper-parameters optimization

Hyper-parameters for dense neural networks:

- Width (Number of neurons in a layer)
- Depth (Number of layers)
- Epochs (Number of cycles)
- Batch Size (Number of training instances in the batch)
- Activation (Decides whether a neuron should be activated or not)
- Early Stopping
- Optimiser (Change on weights and learning rate in order to reduce the losses)
- Learning Rate (Regulates the weights of our neural network concerning the loss gradient)
- Dropout (Practice of disregarding certain nodes randomly during training)
- Batch Normalization

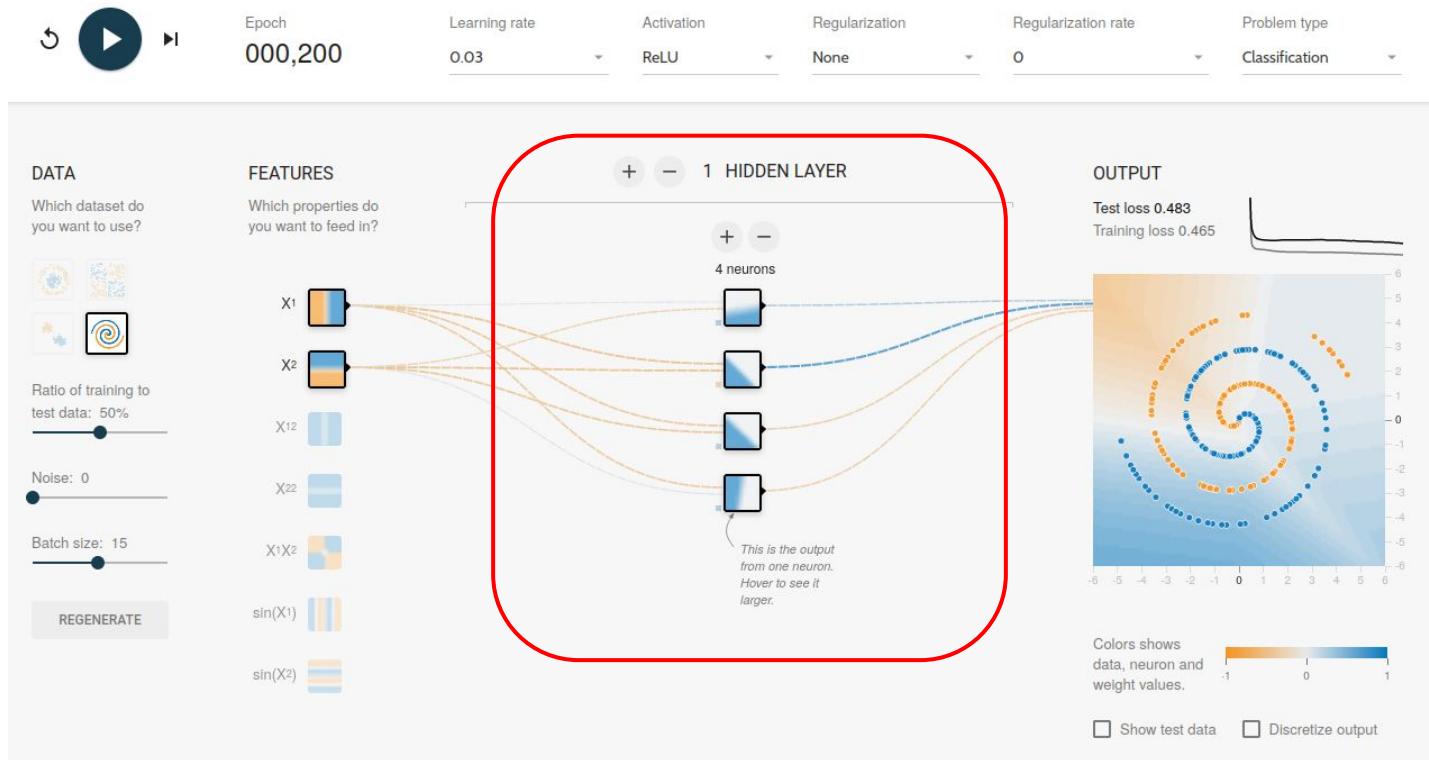
Width

- Number of neurons in a layer



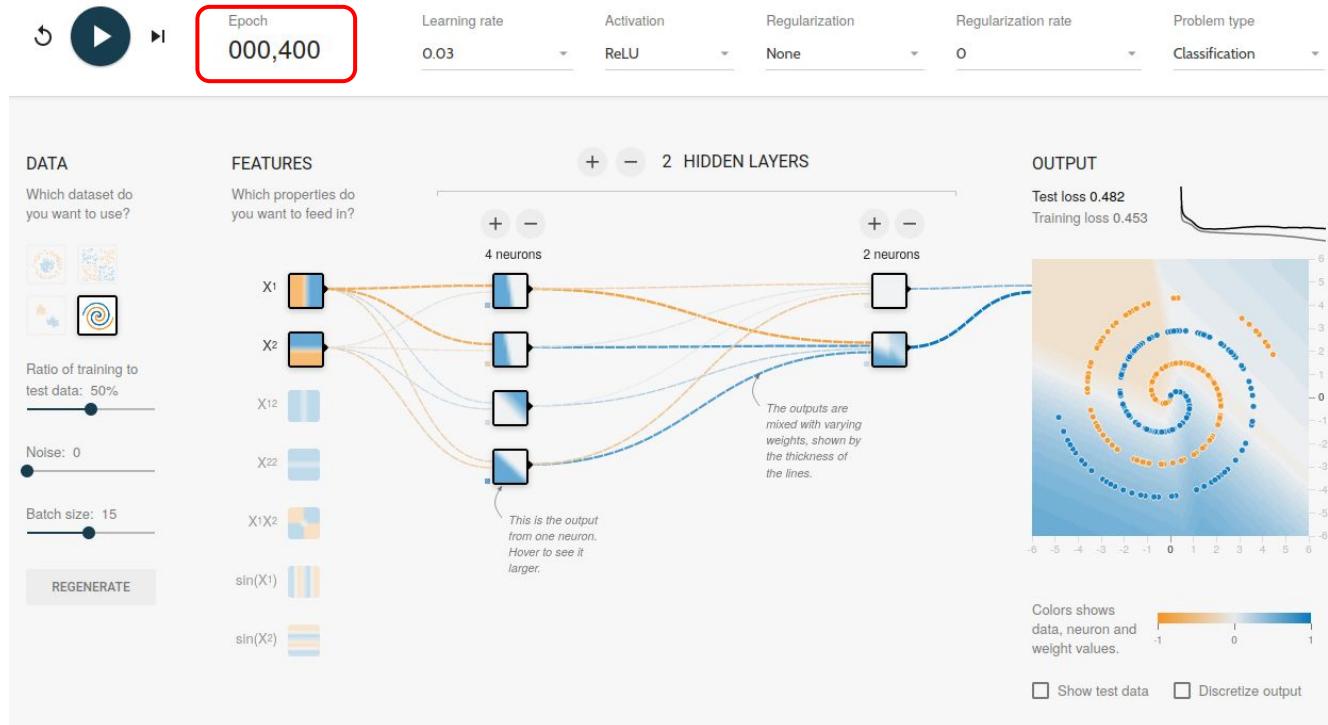
Depth

- Number of layers



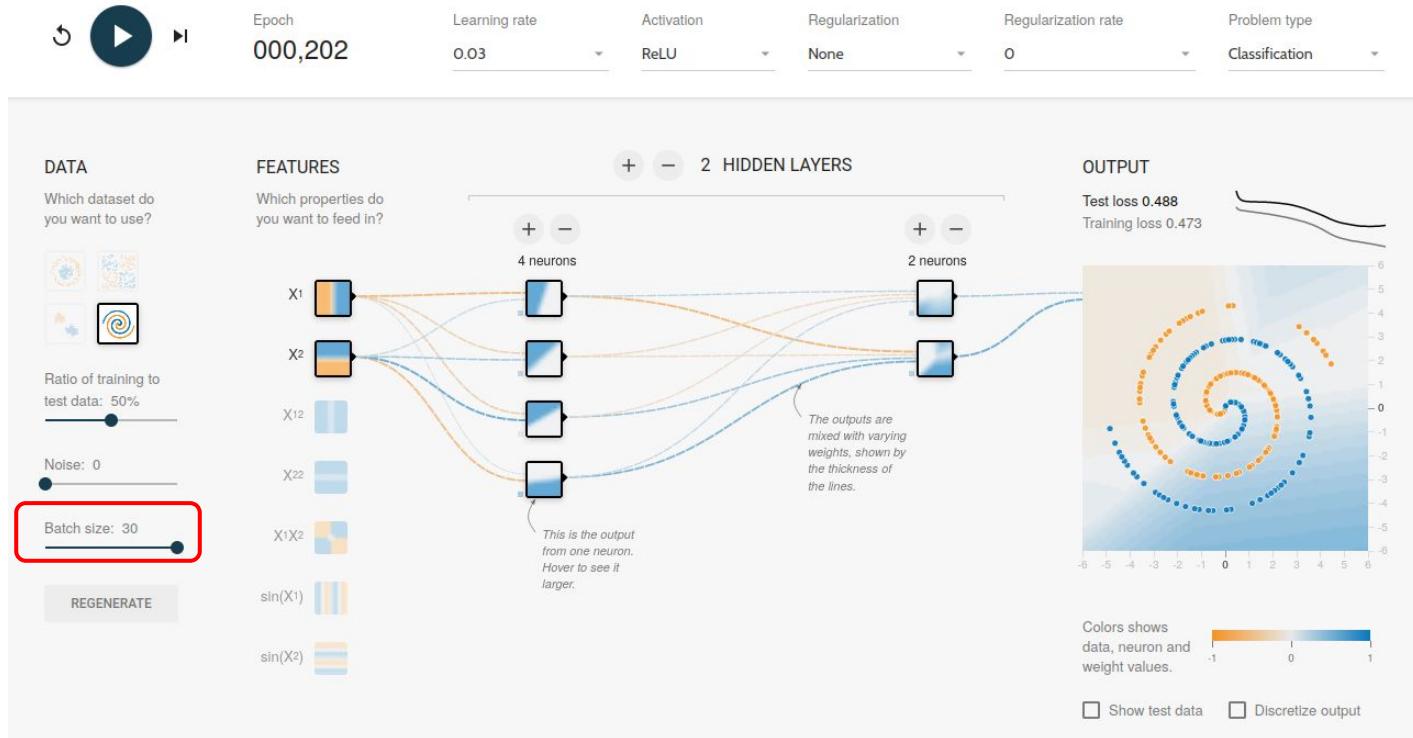
Epochs

- Track “validation loss” to decide this but often the significance might improve even though the loss does not



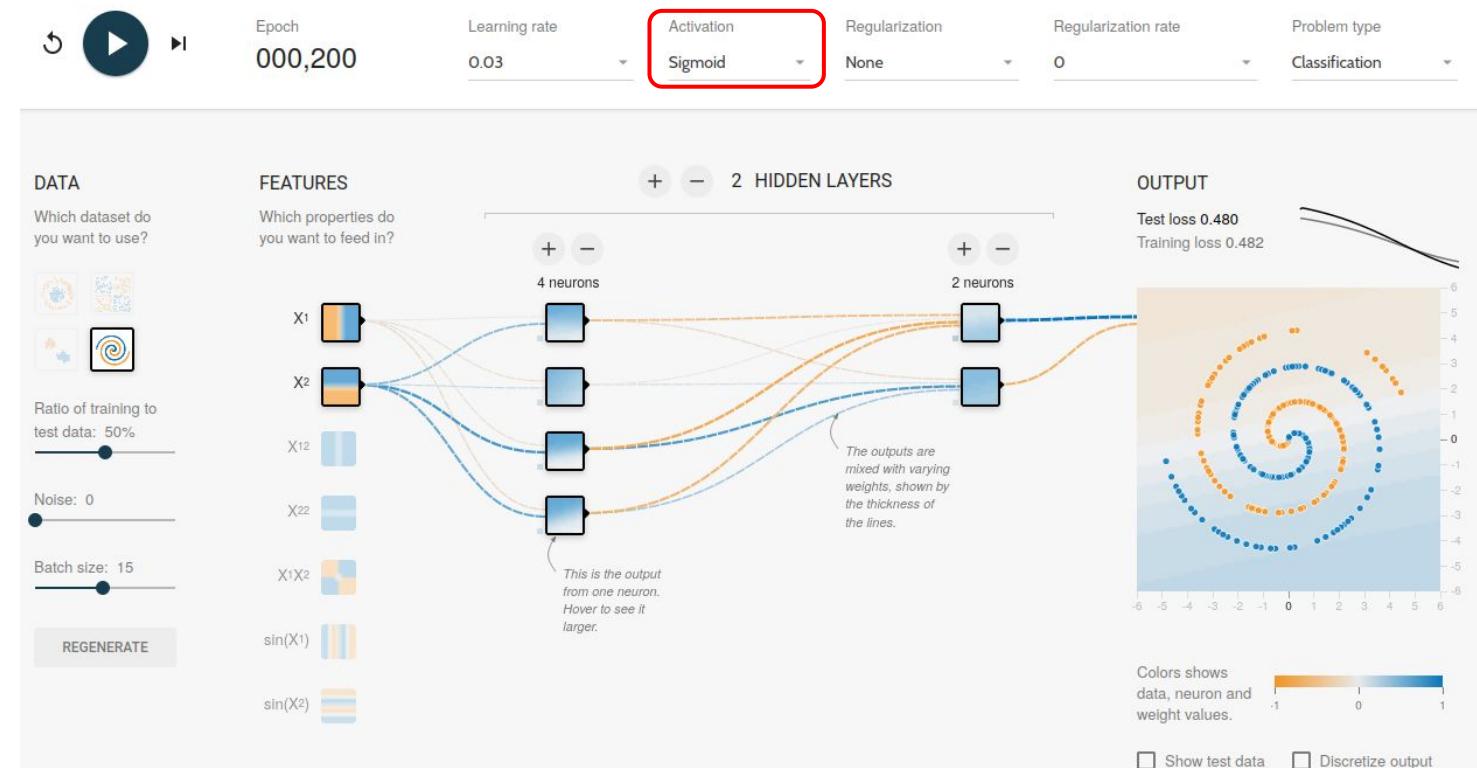
Batch size

- Number of training instances in the batch
- `batch_size=128` means that there are 128 training instances in each batch



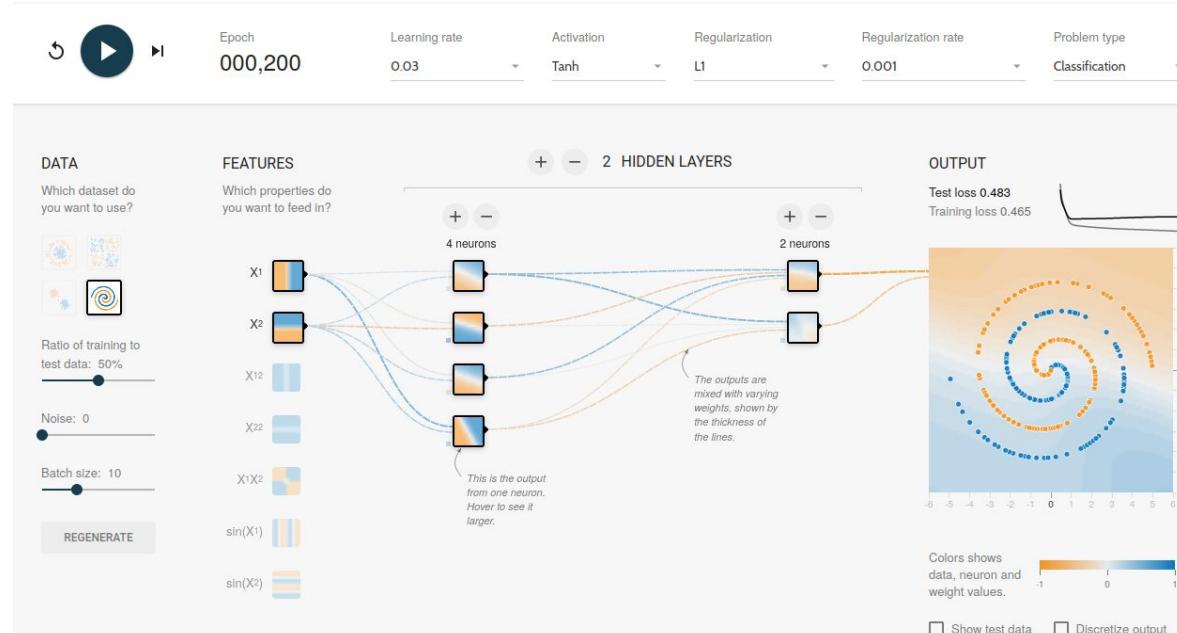
Activation

- Start with Relu/LeakyRelu/Sigmoid



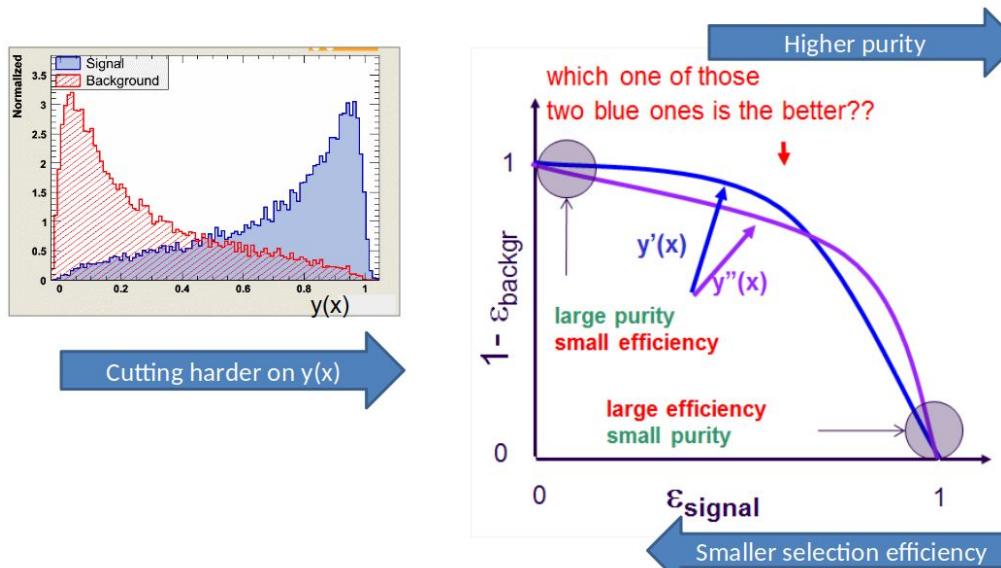
Regularization and its rate

- Start with L1 as an example
- L1/L2 typically requires a smaller rate: start at 0.001, then increase/decrease as you see fit



The ROC curve

- The ROC curve (Receiver Operating Characteristic) is a way to summarize how well you are doing with your estimated $y(x)$ in the classification problem
- How to quantify? Look at rejection at fixed efficiency, compute Area Under Curve (AUC) and many other metrics available



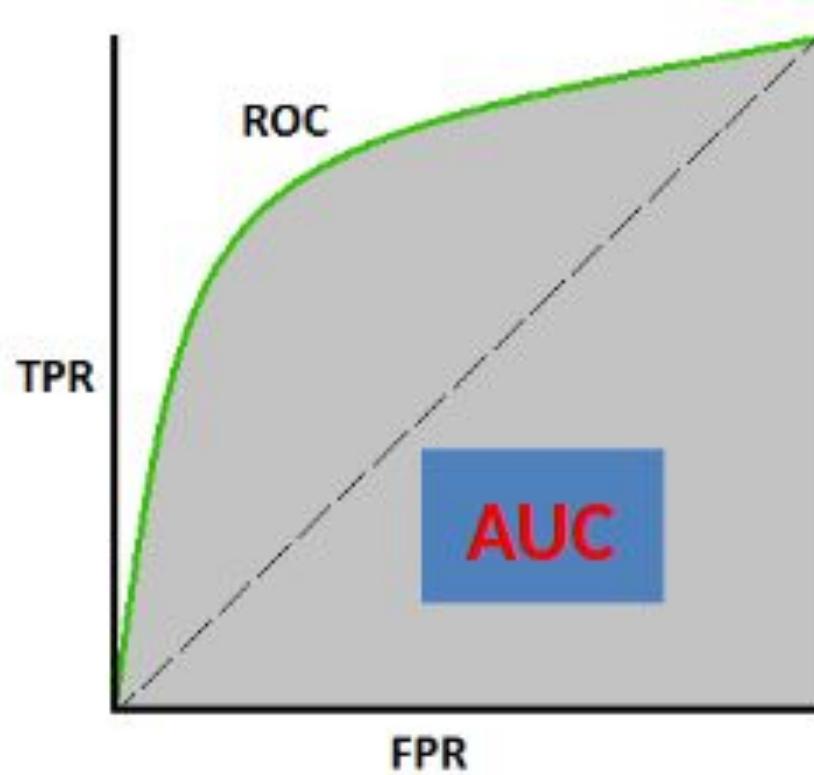
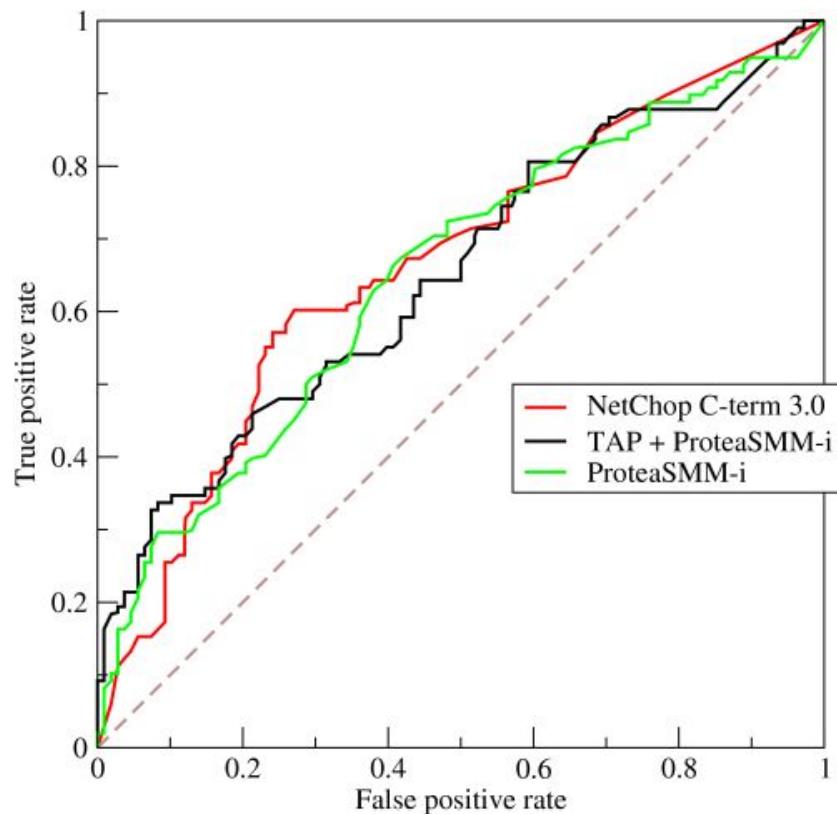
The AUC metric

- In some cases it is not trivial to rank classifiers by their performance.
- In particular, if one does not (yet) have an estimate of the proportion of signal and background (class probabilities), one cannot decide!

A simple criterion is to compute the area under the ROC curve (AUC):

- The AUC has a clean statistical interpretation: taken two events at random, one from each of the two classes, AUC is the probability that the signal event has higher score than the background event.
- AUC is a coherent measure of predictive power of $y(x)$ if we have no information on the relative misclassification cost of the two classes – i.e. if we do not know the operating point. The more we know of that, the less useful AUC is.

The AUC metric



Classifier confusion matrix

Let's consider binary classification:

- Two classes 0 and 1
- Confusion matrix is a 2x2 table

Actual Values: True/False

Predicted values: Positive/Negative

- Sensitivity: $TP/(TP+FN)$

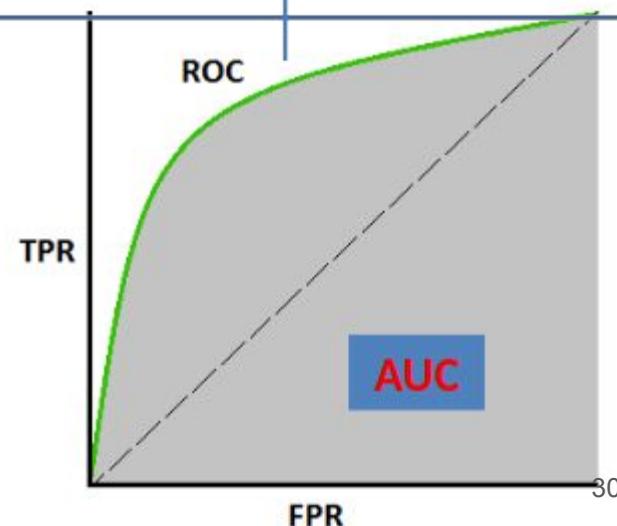
think of it as "signal efficiency"

- Specificity: $TN/(TN+FP)$

"background rejected by selection"

- Purity: $TP / (TP+FP)$ (a relative measure)

		Is signal	Is background
pass cut (positive)	True positives	False positives	
	TP	FP	
fail cut (negative)	False negatives	True negatives	
	FN	TN	



Validation set



Original dataset

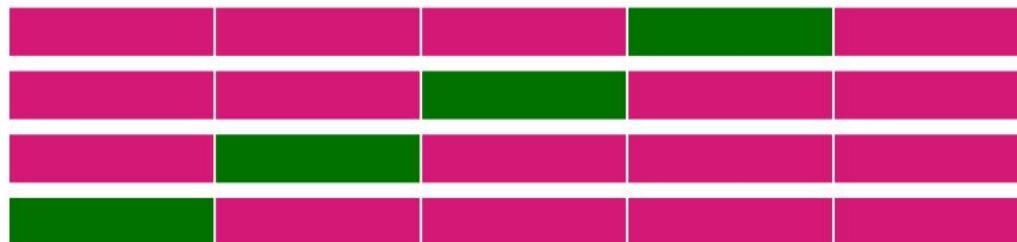
- Split the original dataset into a **training** and **validation** set:
 - ▶ Train model on the training set
 - ▶ Evaluate on the validation set to estimate the test error
 - ▶ Select the model class that gives the lowest estimated error
 - ▶ Optionally, re-train the selected model class on the whole dataset (training + validation)
- Issue: we would like both training and validation sets to be as large as possible (so that the estimate is better), but they must not overlap!

k-Fold Cross Validation

- ▶ Split the original dataset into k equal parts (e.g, $k = 5$)
- ▶ Train on the $k - 1$ parts and validate on the remaining one

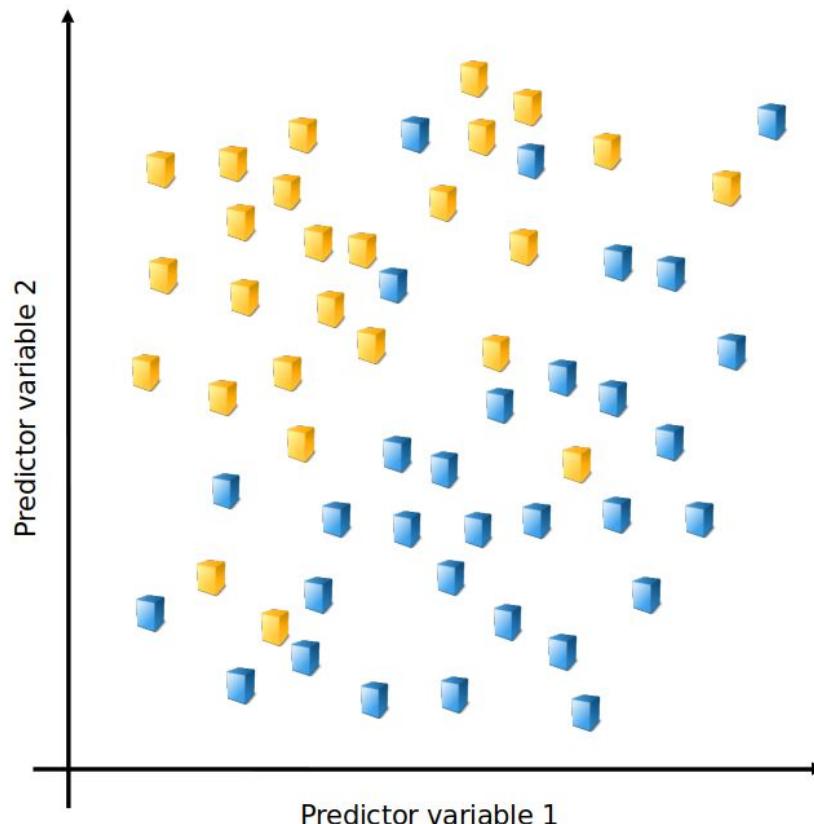


- ▶ Repeat for every choice of the $k - 1$ parts and average the validation errors

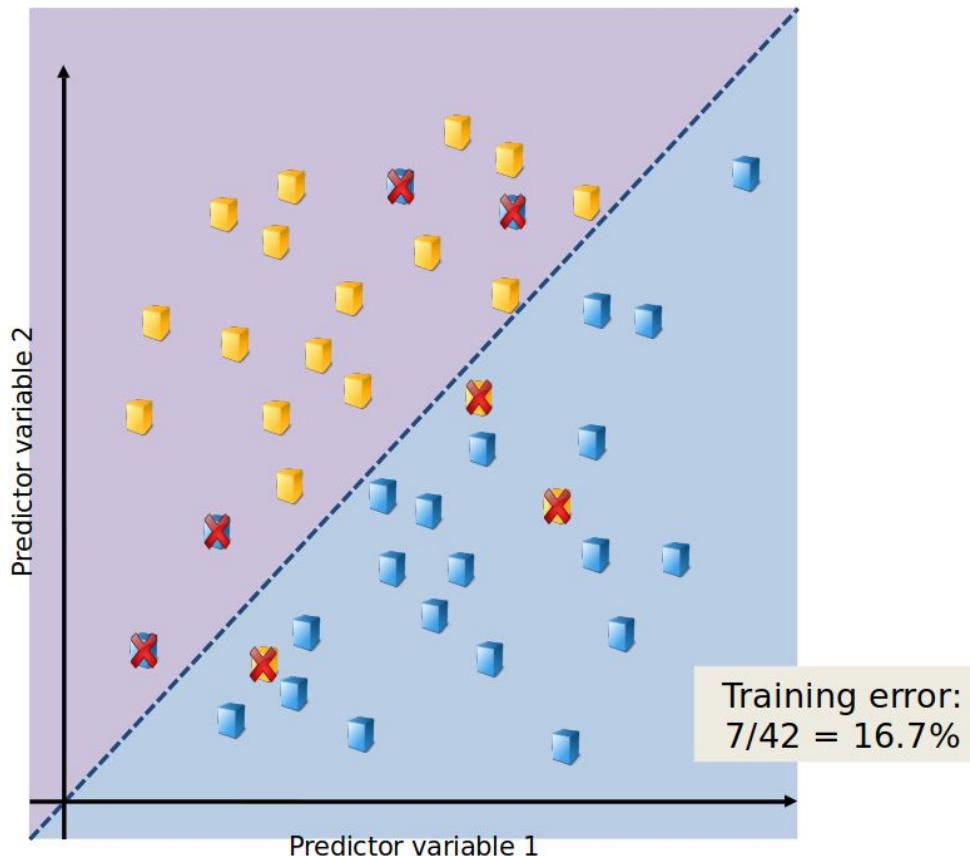


- ▶ Advantage: use all data as validation to improve the estimate of the test error, at the cost of more computation (k trainings)

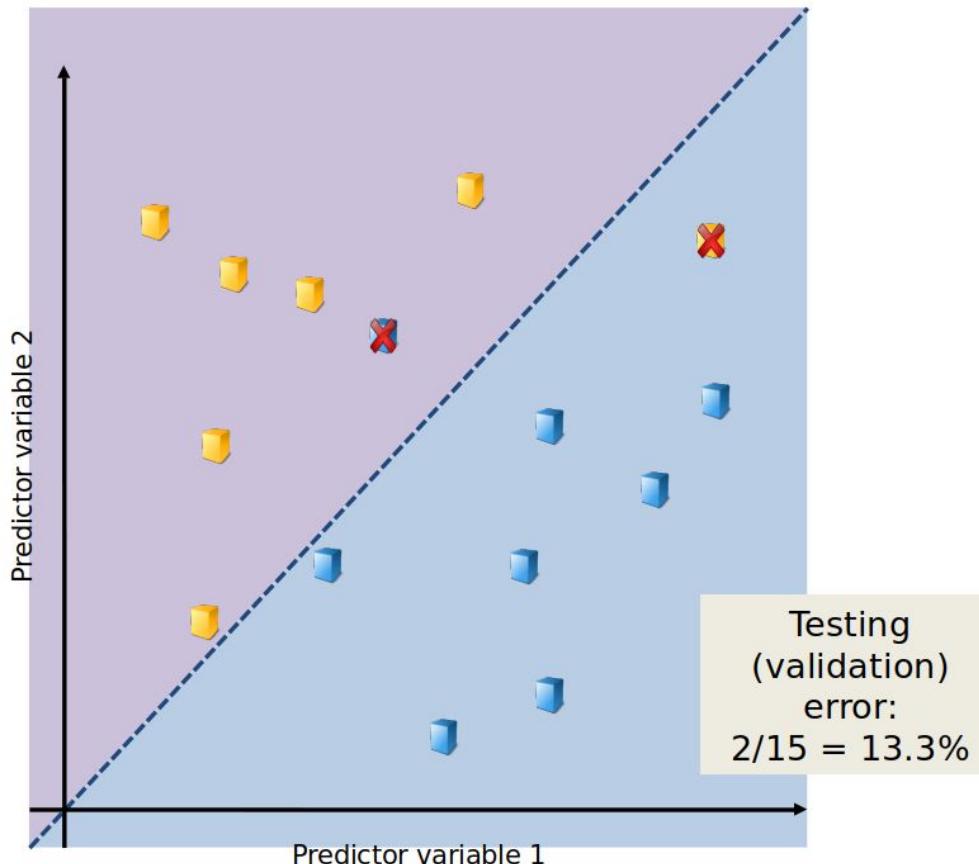
Training and testing



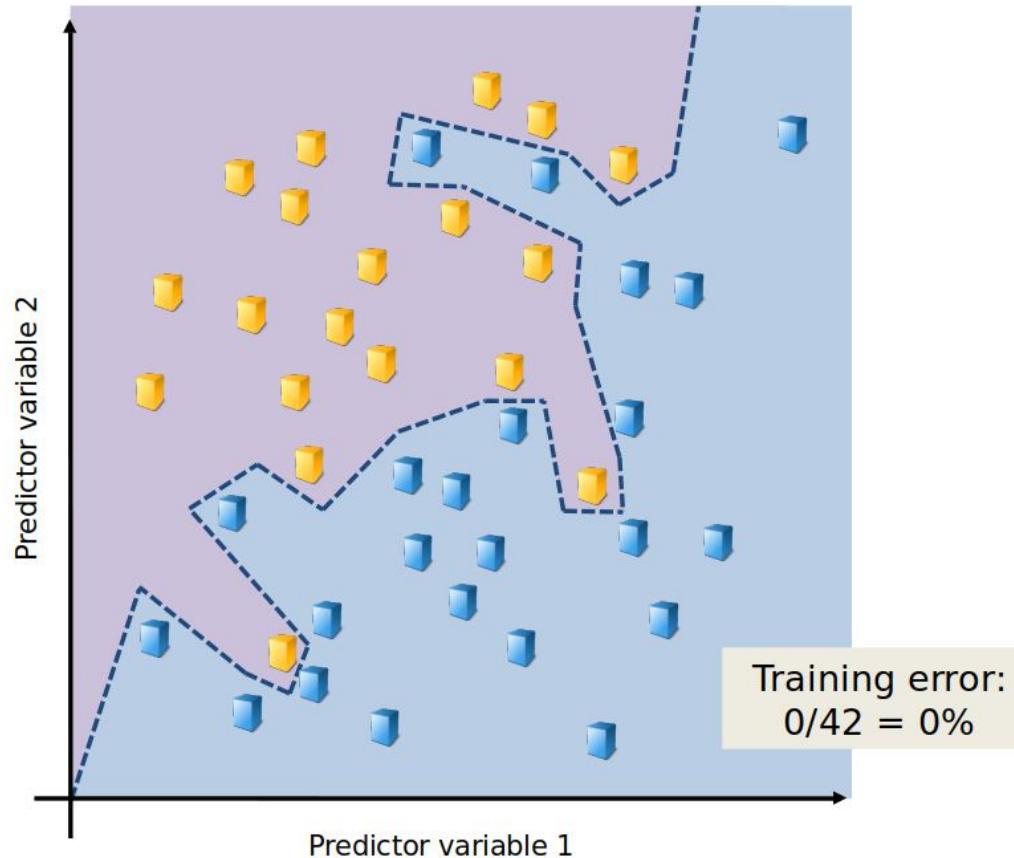
Training and testing



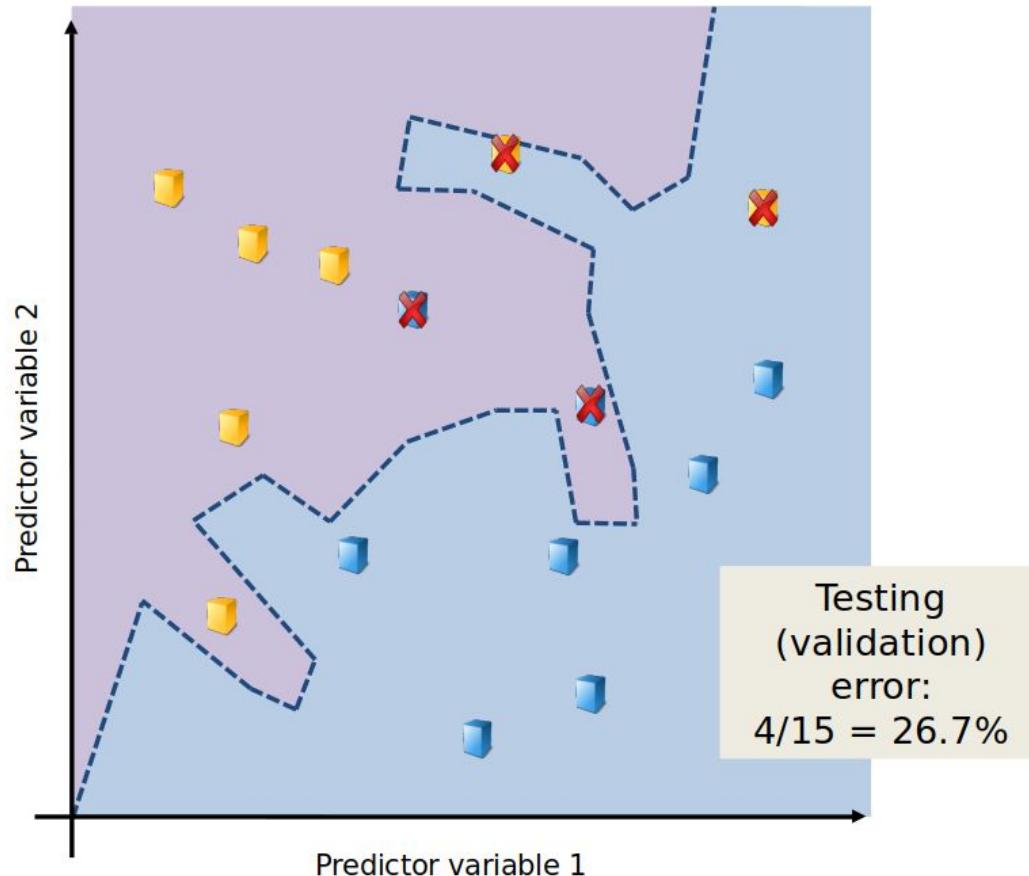
Training and testing



Training and testing



Training and testing



Training, Validating, Testing

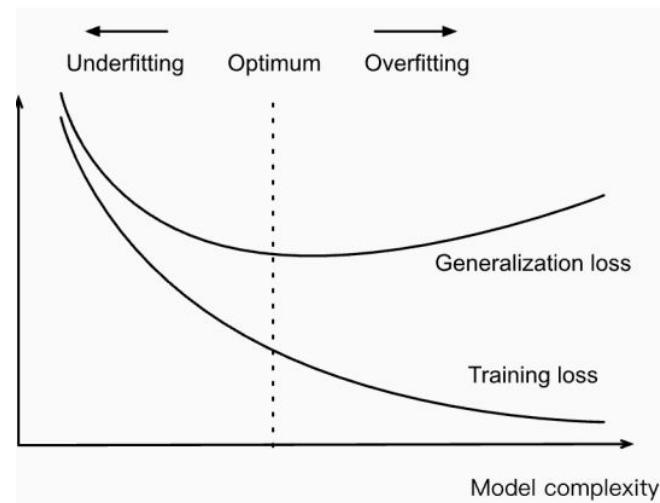
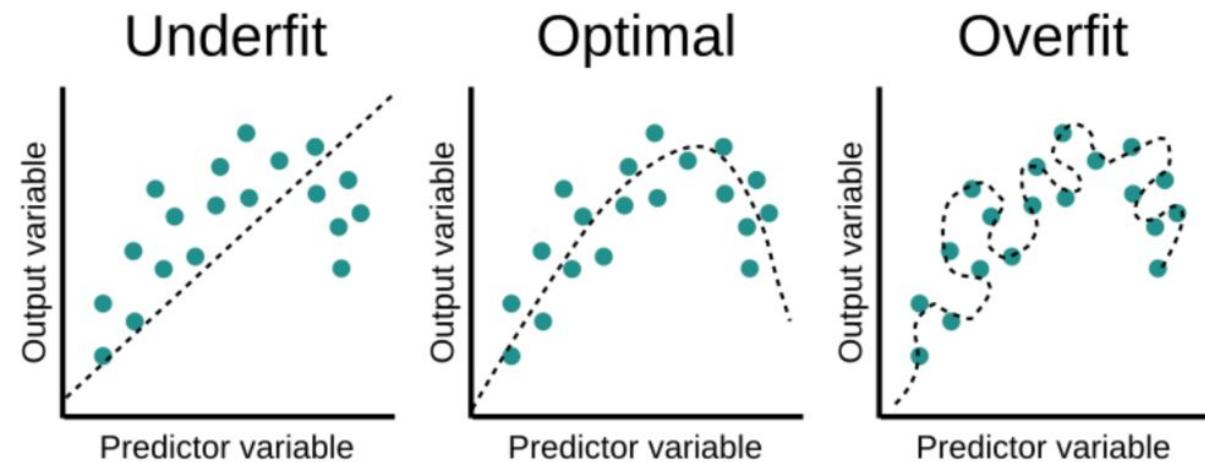
In order to construct and study a classifier built with a supervised algorithm and given a sample of signal events and background events, one usually separates these sets in three parts:

- **Training set:** events used to build the classifier. The algorithm employs them to estimate the prior densities of S and B, or directly the likelihood ratio or a monotonous function of it
- **Validation set:** this is used to understand whether the training was too aggressive (overfitting), and to tune the algorithm parameters for best results
- **Test set:** this sample is totally independent from the former two, and it is used to obtain a unbiased estimate of the final performance of the model, previously learned, validated and optimized.

Underfitting and overfitting

A model fits training data so well that it leaves little or no room for generalization over new data

We say that the model has “high variance” → Overfitting



Why overfitting?

A model fits training data so well that it leaves little or no room for generalization over new data

We say that the model has “high variance” ➔ Overfitting

- **Small training data:** Does not contain enough data samples to accurately represent all possible input data values
- **Noisy data:** The training data contains large amounts of irrelevant information
- **Long training:** The model trains for too long on a single sample set of data
- **High model complexity:** It learns the noise within the training data

How to Prevent Overfitting?

These are some popular techniques:

- **Early Stopping:** Pauses the training phase before the machine learning model learns the noise in the data
- **Reduce the network's capacity:** By removing layers or reducing the number of elements in the hidden layers
- **Regularization:** Collection of training/optimization techniques that try to eliminate factors that do not impact the prediction outcomes
- **Data Augmentation:** Large dataset will reduce overfitting. Data augmentation helps to increase the size of the dataset

Regularization: L1/L2

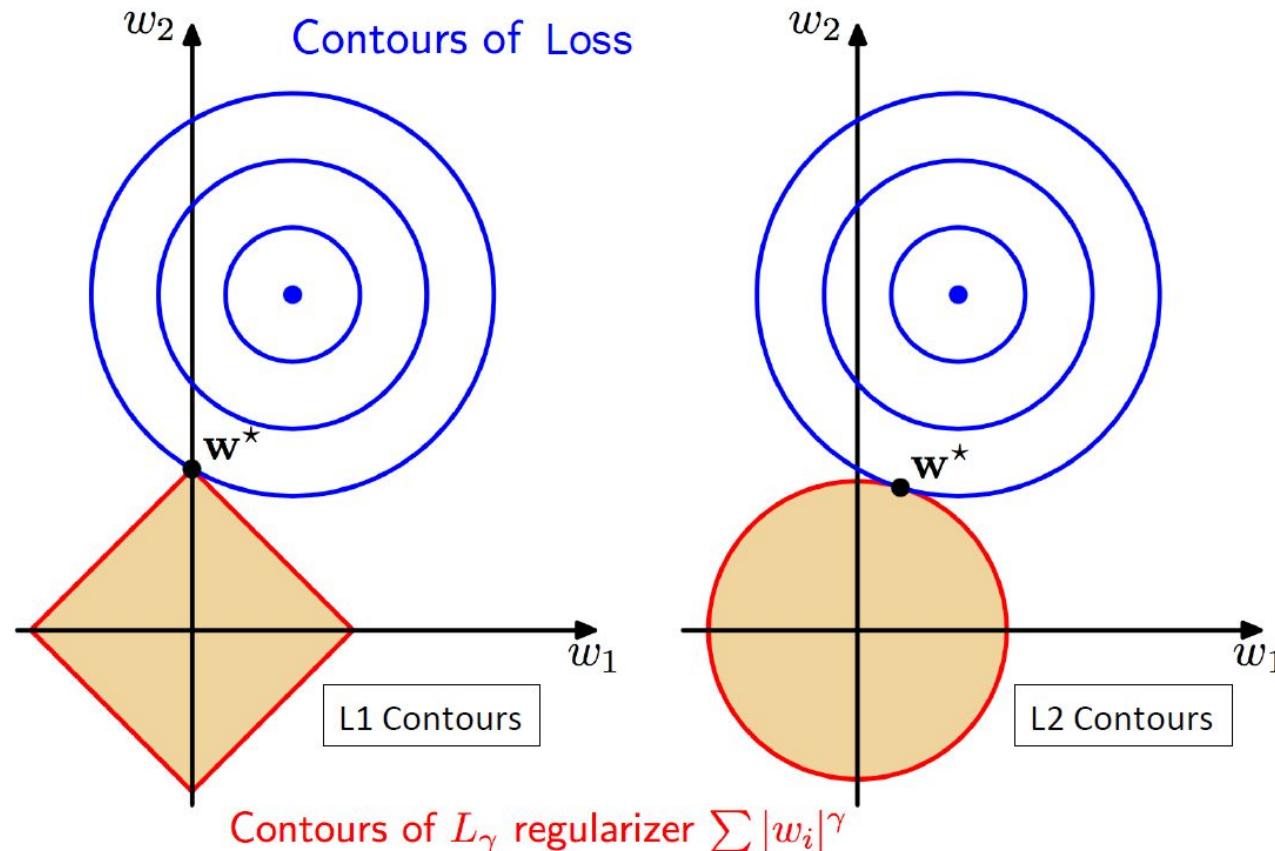
Regularizer on the NN weights (absolute values of weights / squared weights)

L2 regularization is perhaps the most common form of regularization
For every weight, w , in the network we add $\frac{1}{2} \lambda w^2$ to the objective where λ is the regularization strength

L1 regularization is another relatively common form of regularization, where for each weight we add the term $\lambda |w|$

It is possible to combine the L1 regularization with the L2 regularization:
 $\frac{1}{2} \lambda w^2 + \lambda |w|$

Regularization: L1/L2



Regularization: Dropout

Dropout is an extremely effective, simple and recently introduced regularization technique by Srivastava et al. in:

[“Dropout: A Simple Way to Prevent Neural Networks from Overfitting”](#)

that complements the other methods

During Training:

Dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise

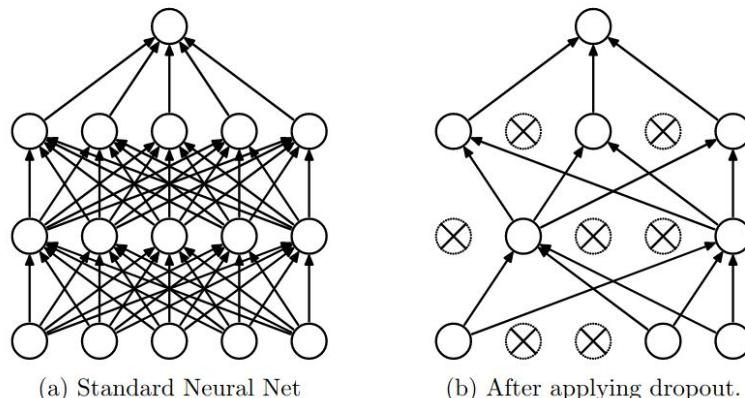


Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

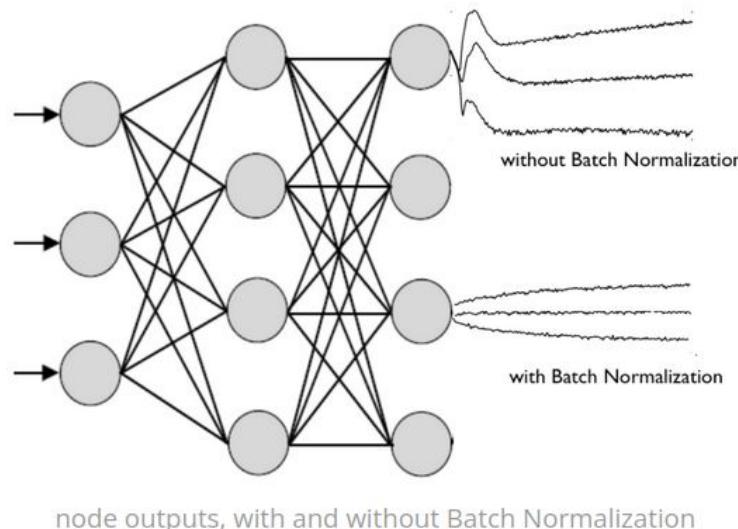
Dropout

- Probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no outputs from the layer
- A good value for dropout in a hidden layer is between 0.5 and 0.8
- Input layers use a larger dropout rate, such as of 0.8

Method	Test Classification error %
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	1.05

Batch normalization

- Normalization technique done between the layers of a Neural Network instead of in the raw data
- Done along mini-batches instead of the full data set
- Serves to speed up training and use higher learning rates, making learning easier



Loss minimization - Gradient descent

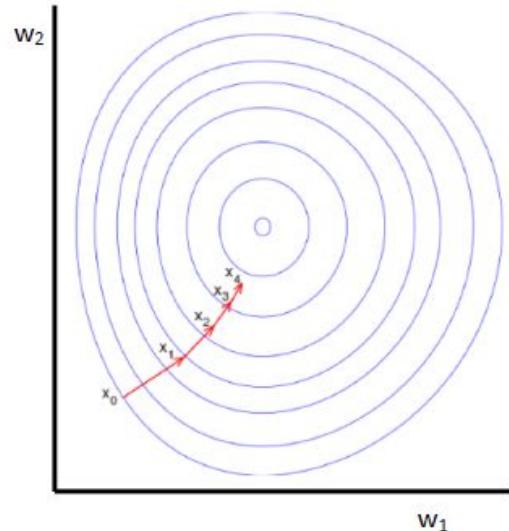
To minimize $-\log L$ and find optimal value of model parameters, in the absence of an analytical description, we "descend" toward the minimum by approximating the shortest route with local information:

- find gradient of L w.r.t. parameters w : $\frac{\partial L(w)}{\partial w}$
- update parameters:

$$w' \leftarrow w - \eta \frac{\partial L(w)}{\partial w}$$

and iterate.

- The success depends on how fast you descend, moduled by "learning rate" η .



Loss minimization - Stochastic Gradient Descent

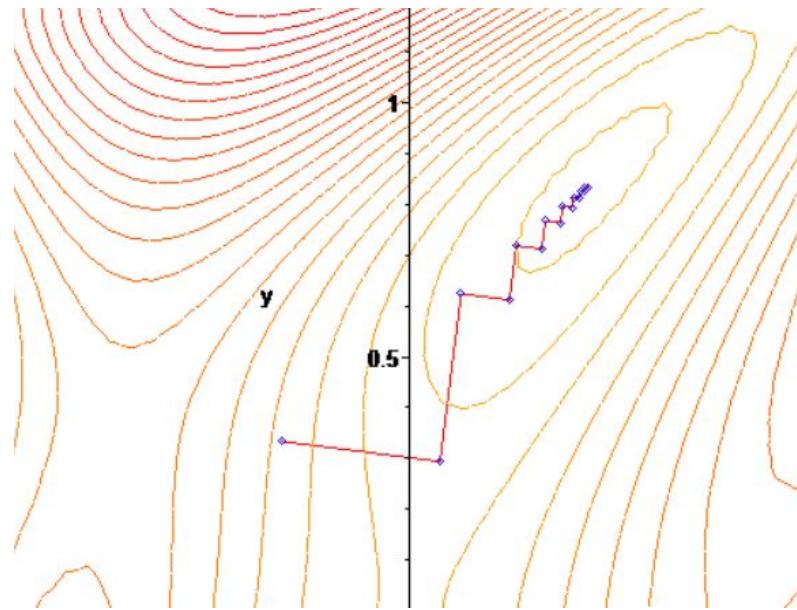
Computing the gradient over the whole training set at each step is sub-optimal:

- CPU-intensive (must pass all dataset)
- large memory use, intractable if too large datasets
- does not allow updates on-the-fly (adding data online)
- Also, it can become ineffective, as risk of getting stuck in local minima is large in multi-dimensions

Most modern deep ML methods employ "stochastic" techniques to find the optimal working point / parameter values

Loss minimization - Stochastic Gradient Descent

- This relies on the possibility to decompose the loss function into the sum of per-example losses
- Stochastic Gradient Descent updates parameters on a per-event basis objective function becomes noisy, but this has merits (can jump out of local minima)



Hyper-parameters tuning

[Hyper-parameter playbook](#)

Choose the model architecture

Summary: When starting a new project, try to reuse a model that already works.

- Choose a well established, commonly used model architecture to get working first
- Try to find a paper that tackles something as close as possible to the problem at hand

Choosing the optimizer

Summary: Start with the most popular optimizer for the type of problem at hand.

- Stick with well-established, popular optimizers, especially when starting a new project

Well-established optimizers that we like include (but are not limited to):

- [SGD with momentum](#)
- [Adam and NAdam](#), which are more general than SGD with momentum.
- Note that Adam has 4 tunable hyperparameters and [they can all matter!](#)

Hyper-parameters tuning

Choose the batch size

Summary: The batch size governs the training speed and shouldn't be used to directly tune the validation set performance. Often, the ideal batch size will be the largest batch size supported by the available hardware

- The batch size is a key factor in determining the training time and computing resource consumption
- Increasing the batch size will often reduce the training time
→ Allows hyperparameters to be tuned more thoroughly within a fixed time interval
- The batch size should not be treated as a tunable hyperparameter for validation set performance
- For an optimized network, the same final performance should be attainable using any batch size (see [Shallue et al. 2018](#))

Hyper-parameters tuning

Choose the initial configuration

Summary: quickly determine the starting points with manual exploration then do a more through check

- Before beginning hyperparameter tuning we must determine the starting point like
 1. The model configuration (e.g. number of layers)
 2. The optimizer hyperparameters (e.g. learning rate)
 3. The number of training steps

Determining this initial configuration will require some manually configured training runs and trial-and-error.

Choosing the number of training steps involves balancing the following tension:

- Training for more steps can improve performance and makes hyperparameter tuning easier (see [Shallue et al. 2018](#))
- Training for fewer steps means that each training run is faster, allowing more experiments to be run in parallel.

Hyper-parameters tuning

Several tools allow you to do hyper parameter scans and hyperparameter optimization:

- [Ray-tune](#)
- [Weights & Bias Sweep](#)
- [TensorBoard HParams](#)
- [Keras Tuner](#)
- [Scikit-Optimize](#)
- [Optuna](#)



RAY



Weights & Biases



TensorBoard SCALARS HPARAMS




OPTUNA

All of these tools have grid search,random search and Bayesian Optimization implemented

- **Pick the one you like!**

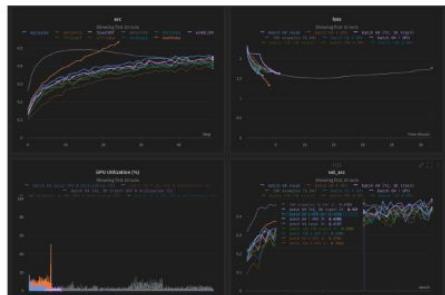
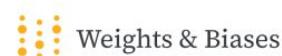
Tools for ML experiments visualization

You need to do some or a lot of experimenting with model improvement ideas

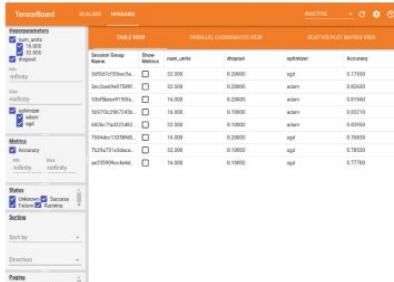
- Visualizing differences between various ML experiments becomes crucial

There are several popular tools: Weights & Biases, TensorBoard, Comet, MLflow, etc

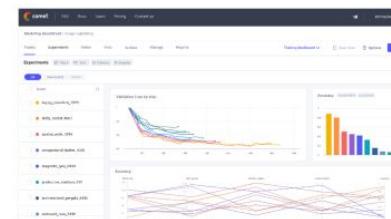
- Tracking and visualizing metrics such as loss and accuracy
- Monitor learning curves
- Visualize CPU/GPU utilization



TensorBoard



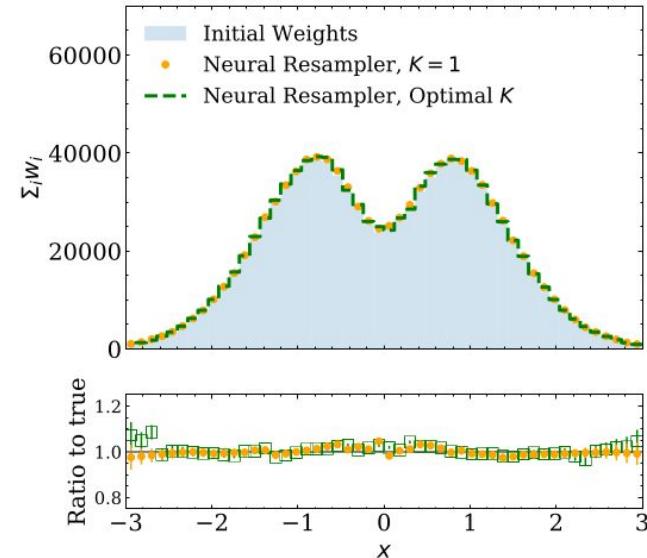
Comet



for managing the end-to-end machine learning lifecycle

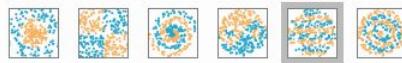
Negative Event Weight

- Only certain BDT packages can handle negative weighted events
- For NNs, negative weights make logical sense, loss is multiplied by a negative weight and everything works as you would expect. So use your negative sample weights!
- The more challenging problem: very large variance of weights (by orders of magnitude)
- Often, you can even re-weight them with ML to get rid of negative weights!
Neural Resampler, Unweighting with generative models

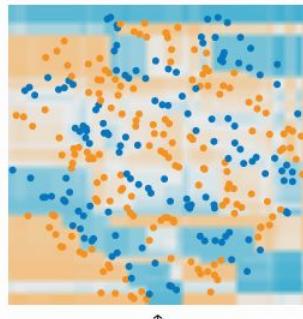


You can also play with Boosted Decision Trees!

Dataset to classify:

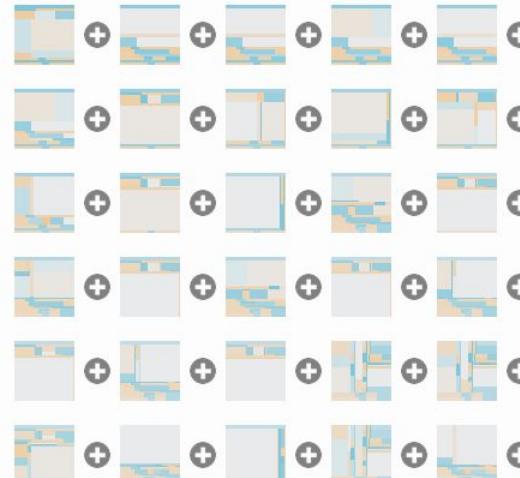


Prediction:



train loss: 0.351 test loss: 0.604

Decision functions of first 30 trees



tree depth: 8 learning rate: 0.2 rotate dataset:

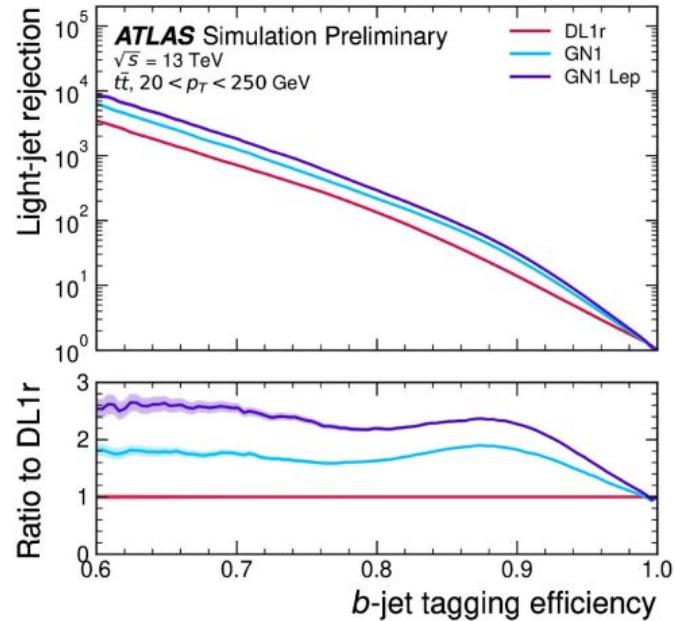
subsample: 100% # trees: 50

rotate trees show gradients on hover use Newton-Raphson update

[BDT playground](#)

Machine Learning in HEP

- ML has been in HEP for ages: Signal to background separation and flavor tagging
- What's new:
 - Significant increase in compute power (GPUs) available for ML
 - Dramatic increase in ML architectures for different applications: transformers, large-language models (LLMs), graph neural networks (GNNs), convolutional neural networks (CNNs), etc
- ML advancements have already affected HEP, e.g., drastic improvement in flavor tagging (GN1).

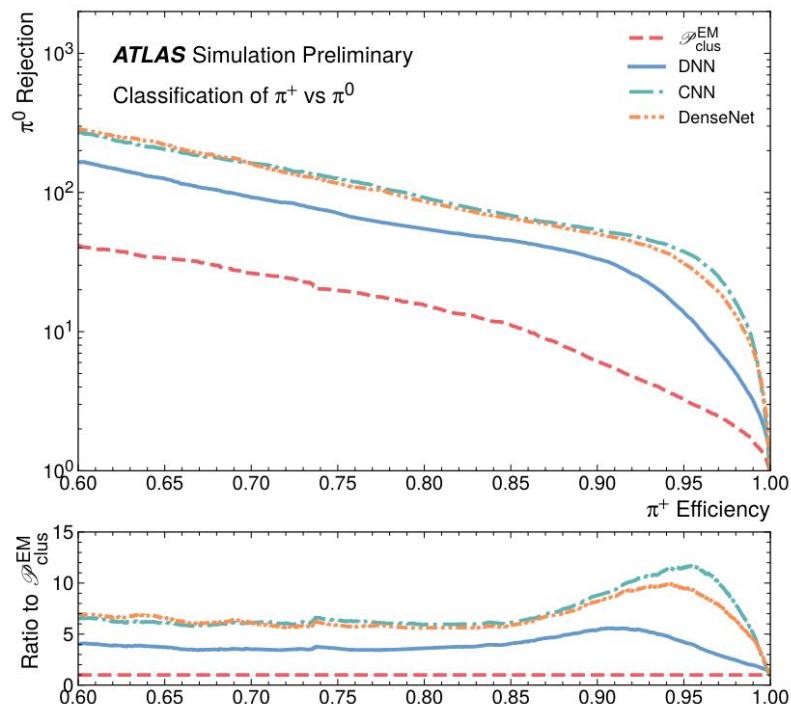
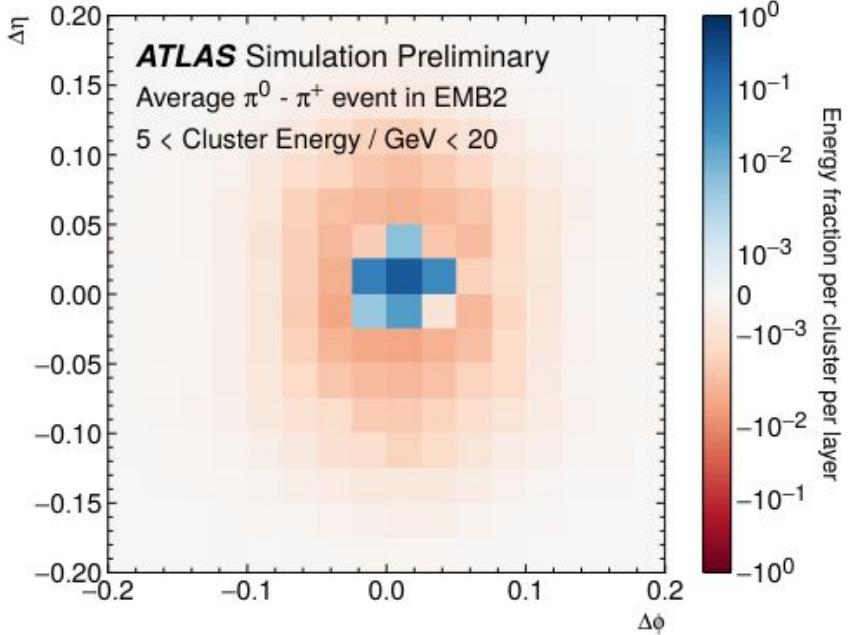


MVAs used since Run 1 (MV1)

Machine Learning in HEP

- Increasing complexity, decreasing interpretability
- Compatibility with calibration techniques
 - Can you derive scale factors, systematics, etc?
- Systematics do not kill your gains
 - Study them early on!
 - Even a flat uncertainty can give you a rough idea
- ML enables portability (ability to run on different types of hardware, e.g., CPUs, GPUs, FPGAs)

- Pixelated calorimeter images: Convolutional NN (CNN)
- The ML techniques all do an excellent job of distinguishing π^0 from π^\pm showers



Why use Machine Learning?

- ML to exploit high-dimensional correlations
 - Our multidimensional distributions are rarely rectangular
 - Rectangular cuts won't maximize signal efficiency and background rejection
 - Maximizing our performance is essential in luminosity era
- ML as a surrogate model: fast and/or can run many types of hardware:
e.g., ML-based track reconstruction, AtlFastSim3
- ML for non-standard data: data that is less confined than our physics objects (e.g., operational data), variable length input, etc

High Luminosity - LHC

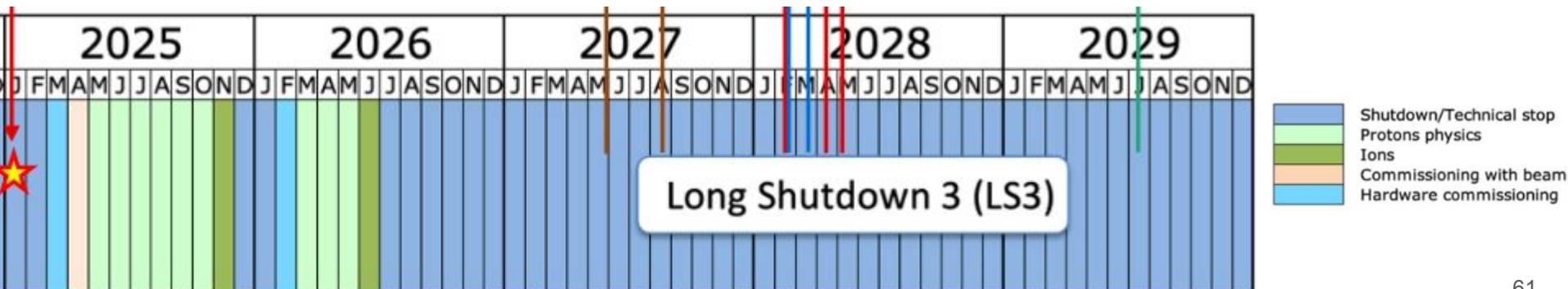
Trigger & data acquisition challenge

- Luminosity: $2 \rightarrow 7.5 \cdot 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$
- Pileup: $60 \rightarrow 200$
 - more time consuming

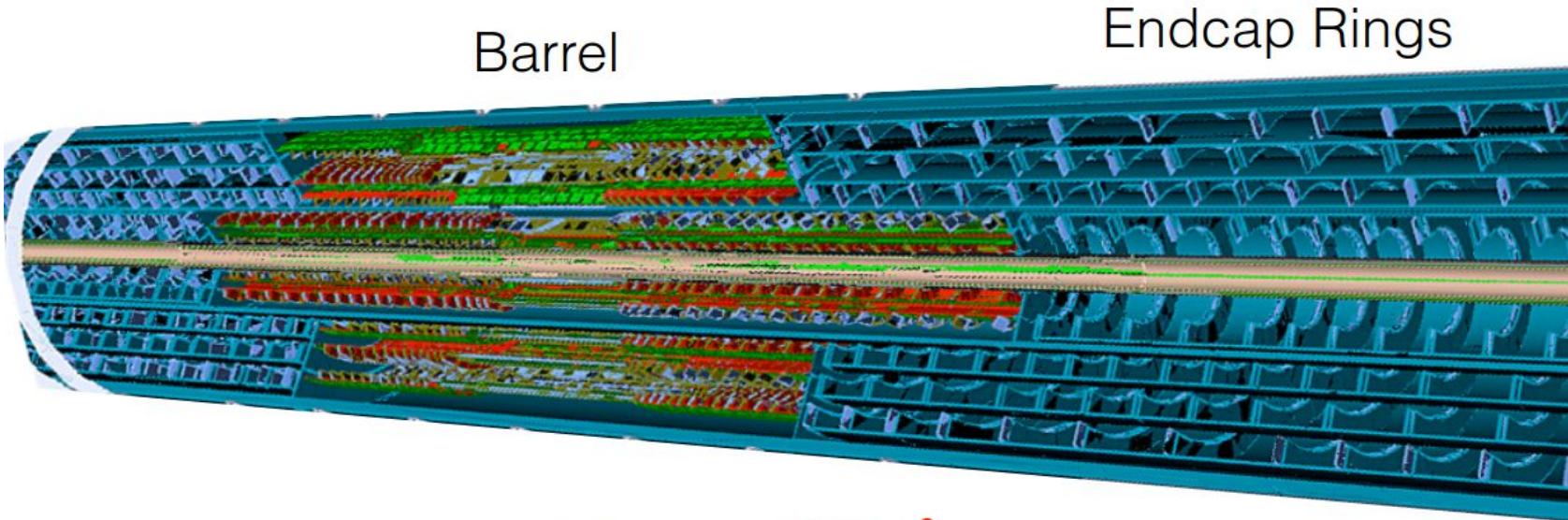


ATLAS detector upgrade

- New Tracker, new Timing Detector, additional muon chambers, new Tile electronics, ...



Why use Machine Learning?



Active area: **12.7 m²**

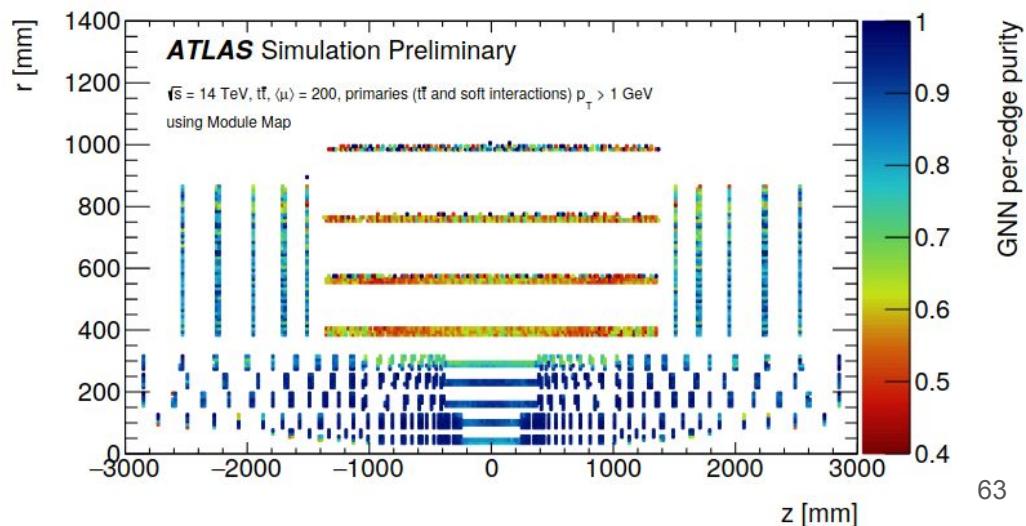
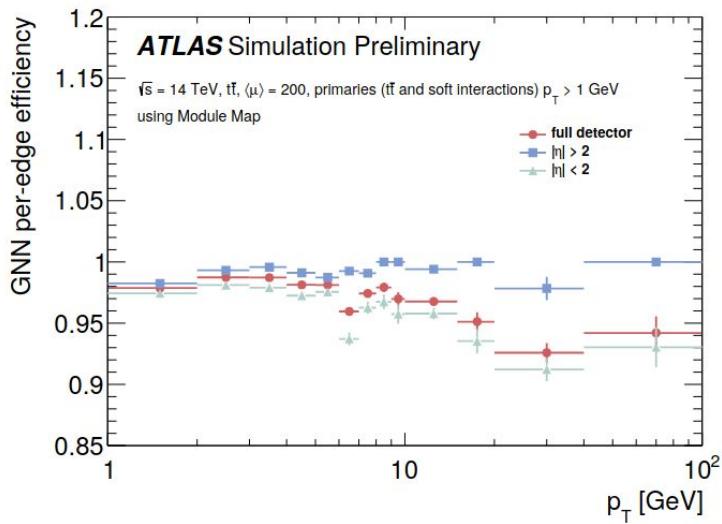
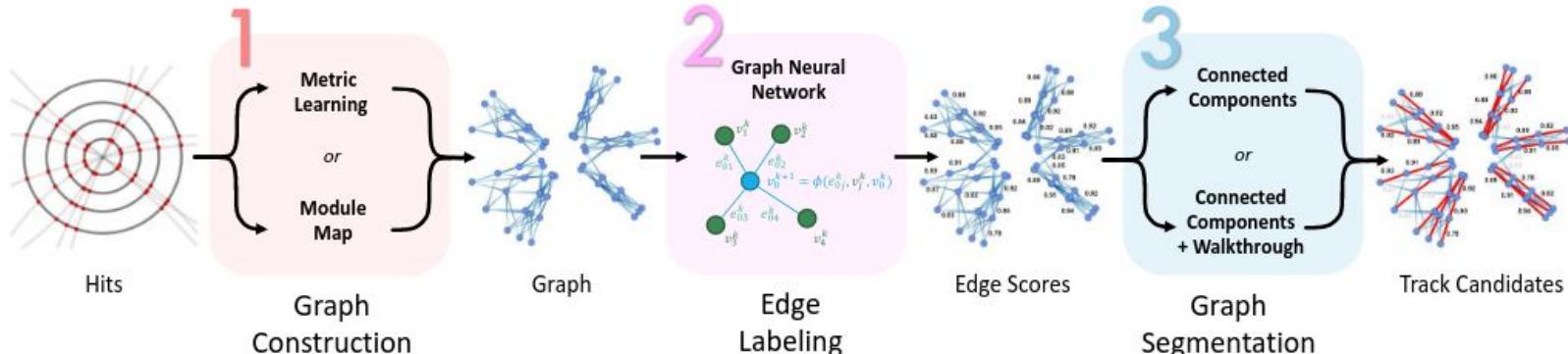
Pixel size: 50x50 (or 25x100) μm^2

of modules: 10276

of FE chips: 33184

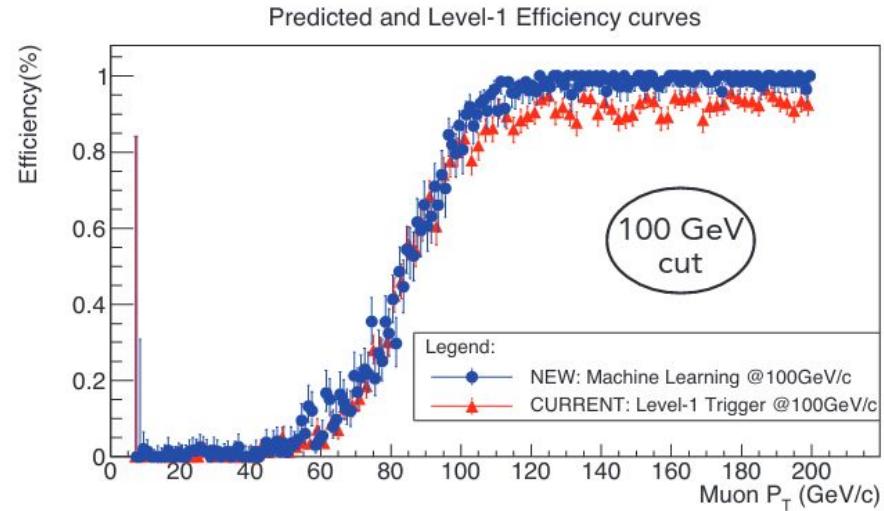
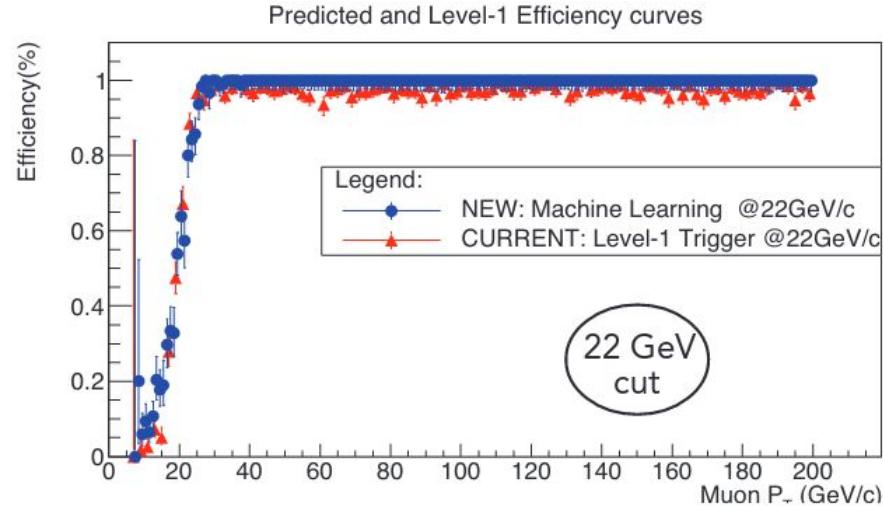
of channels: **$\sim 5 \times 10^9$**

Why use Machine Learning?



Real-time triggers

- Machine Learning based muon trigger algorithms for the Phase-2 upgrade of the CMS detector



At high values of p_T , the performances of the model predictions begin to decrease probably due to a low resolution for small bending muons.

Some ideas for the future

- Can we also improve our reconstruction for new sub-detectors as HGTD or New Small Wheel?
- Can we use reinforcement learning for automatic data quality monitoring in HEP experiments?
- Can we also have an electron/photon identification with a convolutional neural network similarly to the jets?
- Can we try to tag dark matter particles with ML? Or search for them?
- Can we study the systematic effects in Jet Tagging Performance?
- Can we use transformers for Particle Track Reconstruction and Hit Clustering?
- Can we improve the knowledge on heavy ions collisions by studying topological separation of dielectron signals?



Thanks for the attention!