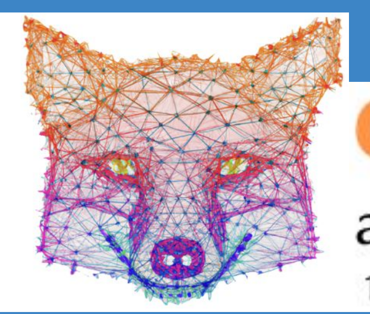
Validation and hyper-parameters in Machine Learning

Ana Peixoto (University of Washington)

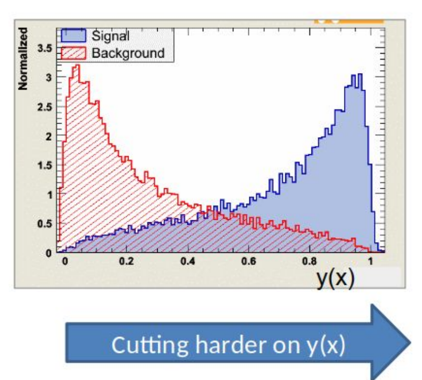


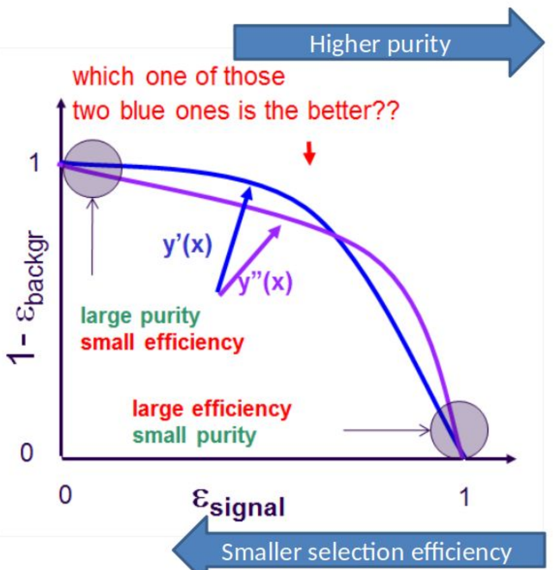
CHACAL 2024: Computing in HEP and Applications CNRS-Africa Lectures

15-27 Jan 2024, University of Witwatersrand, Johannesburg, South Africa

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 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!

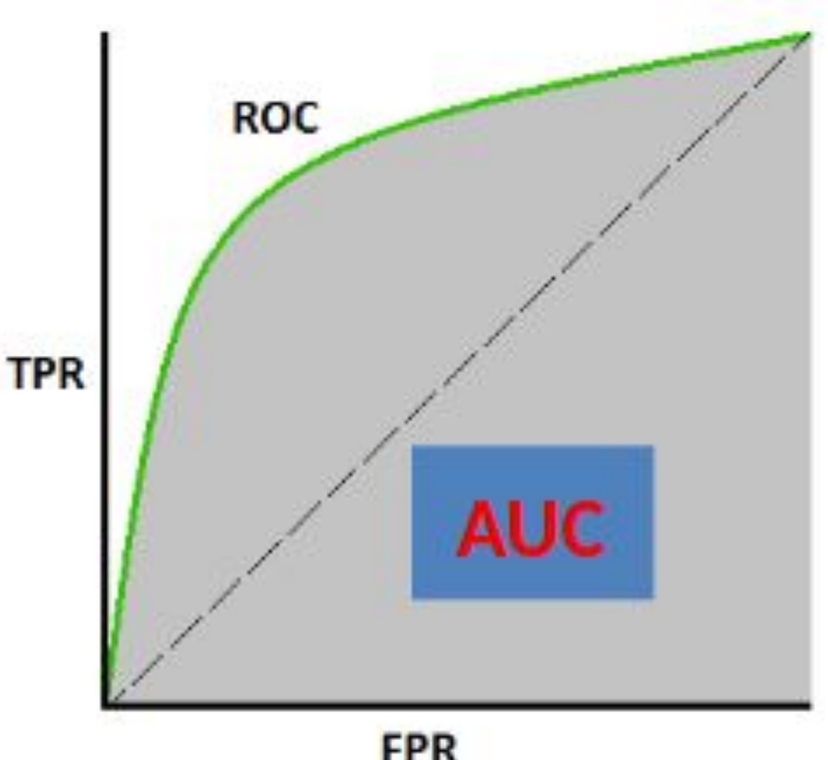
A simple criterion is to compute the area under the ROC curve (AUC):

- 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

NetChop C-term 3.0
TAP + ProteaSMM-i
ProteaSMM-i

True positive rate
False positive rate



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

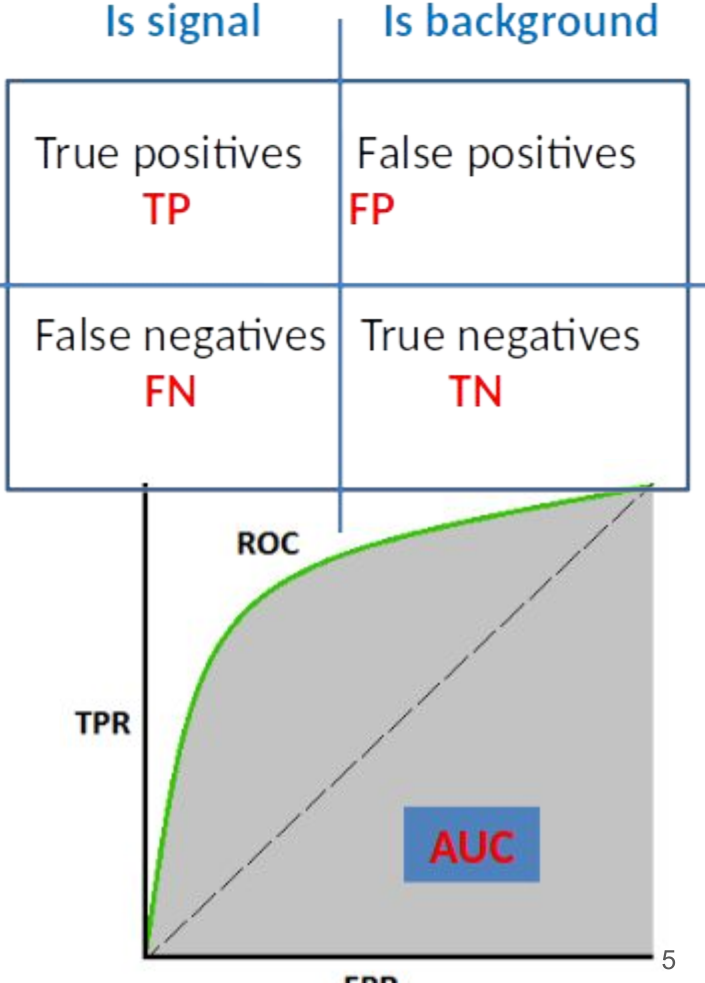
- Sensitivity: TP/(TP+FN)

think of it as "signal efficiency"

- Specificity: TN/(TN+FP)

"background rejected by selection"

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



Validation set

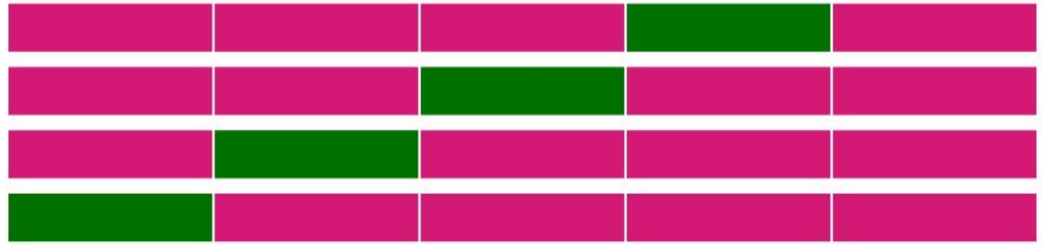
Original dataset

- 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

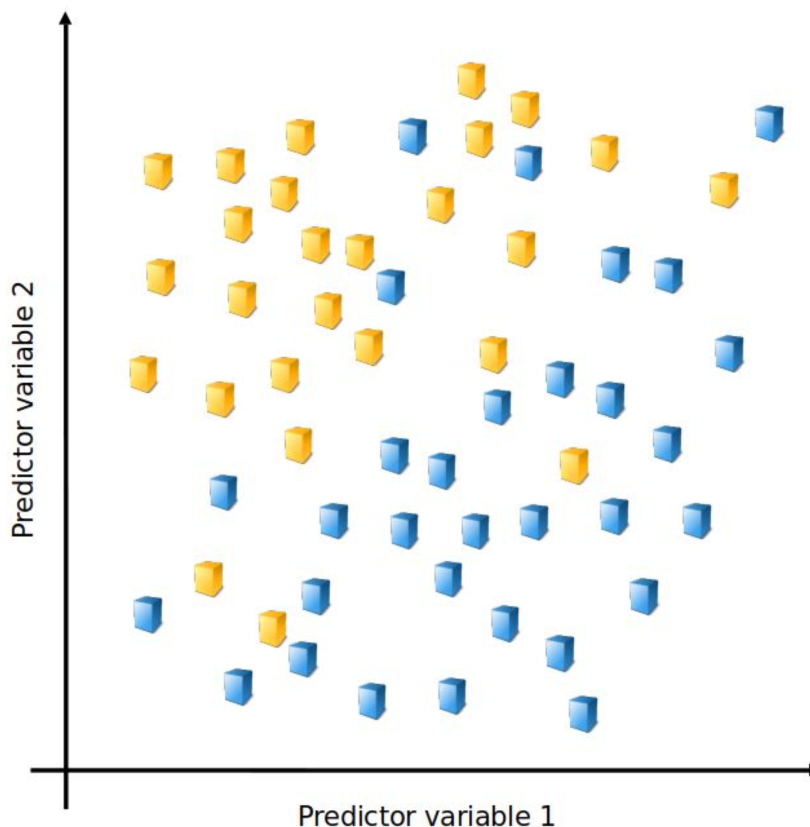
- Split the original dataset into k equal parts (e.g, $k = 5$)
- Train on the $k - 1$ parts and validate on the remaining one

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

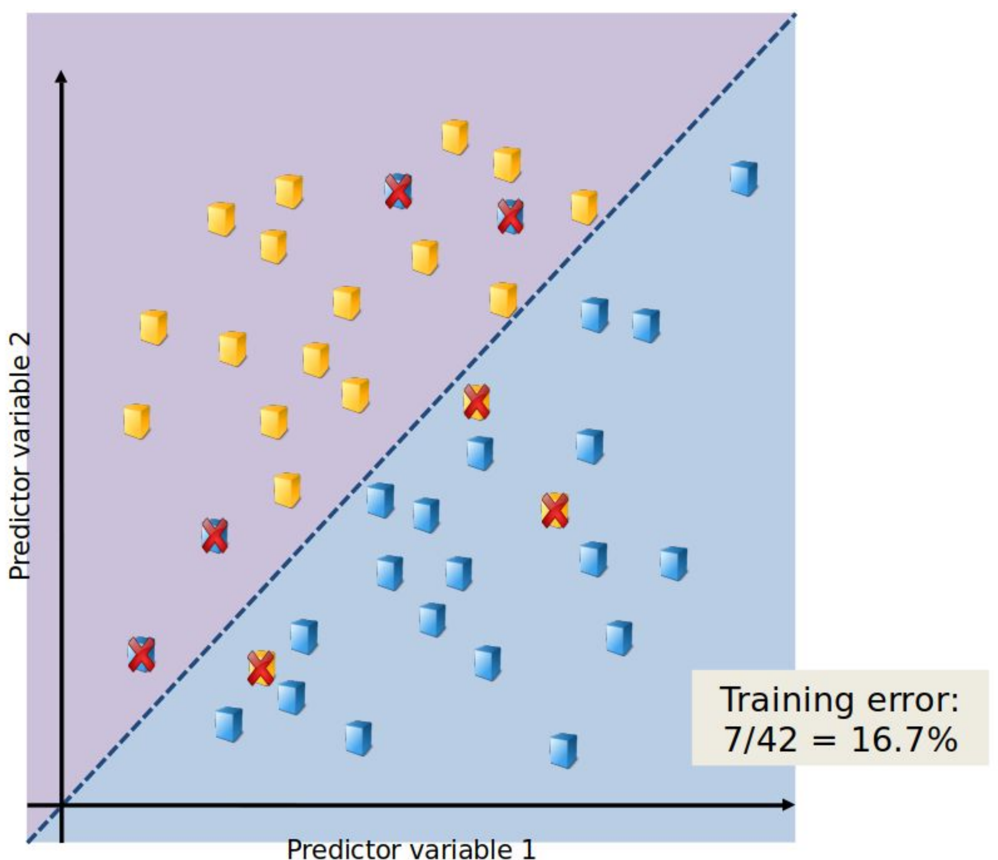


- Advantage: use all data as validation to improve the estimate of the test error, at the cost of more computation (k trainings)

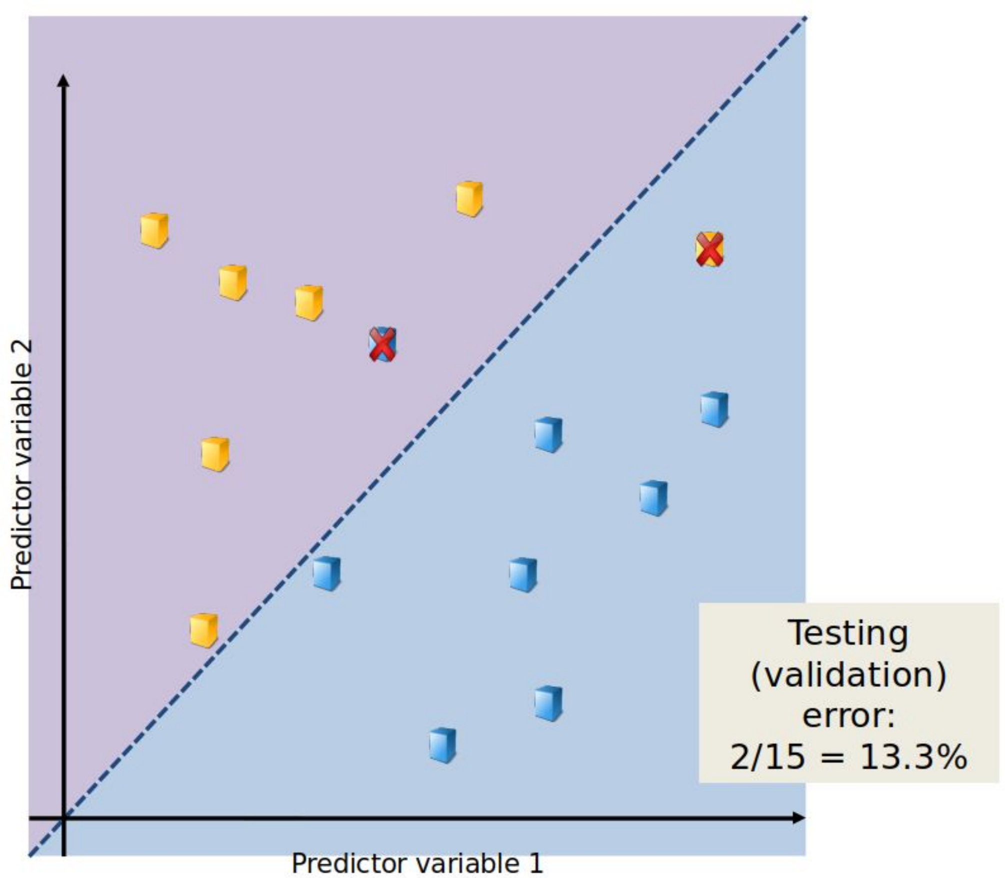
Training and testing



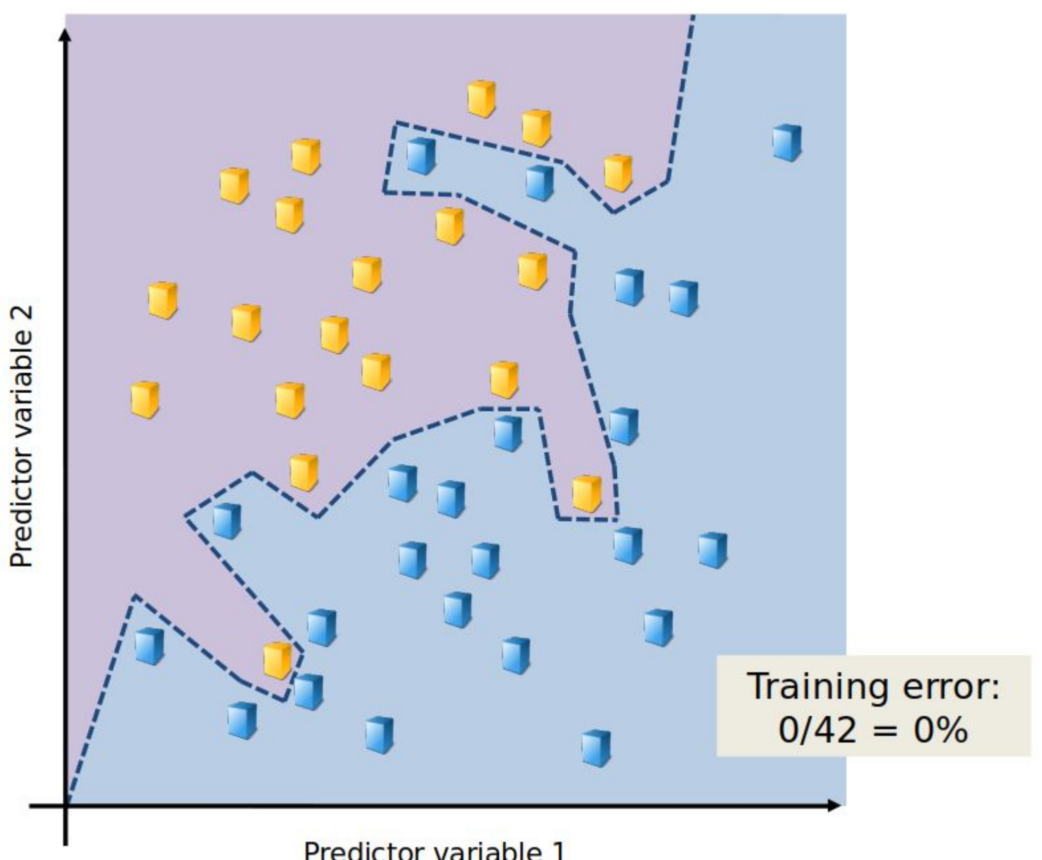
Training and testing



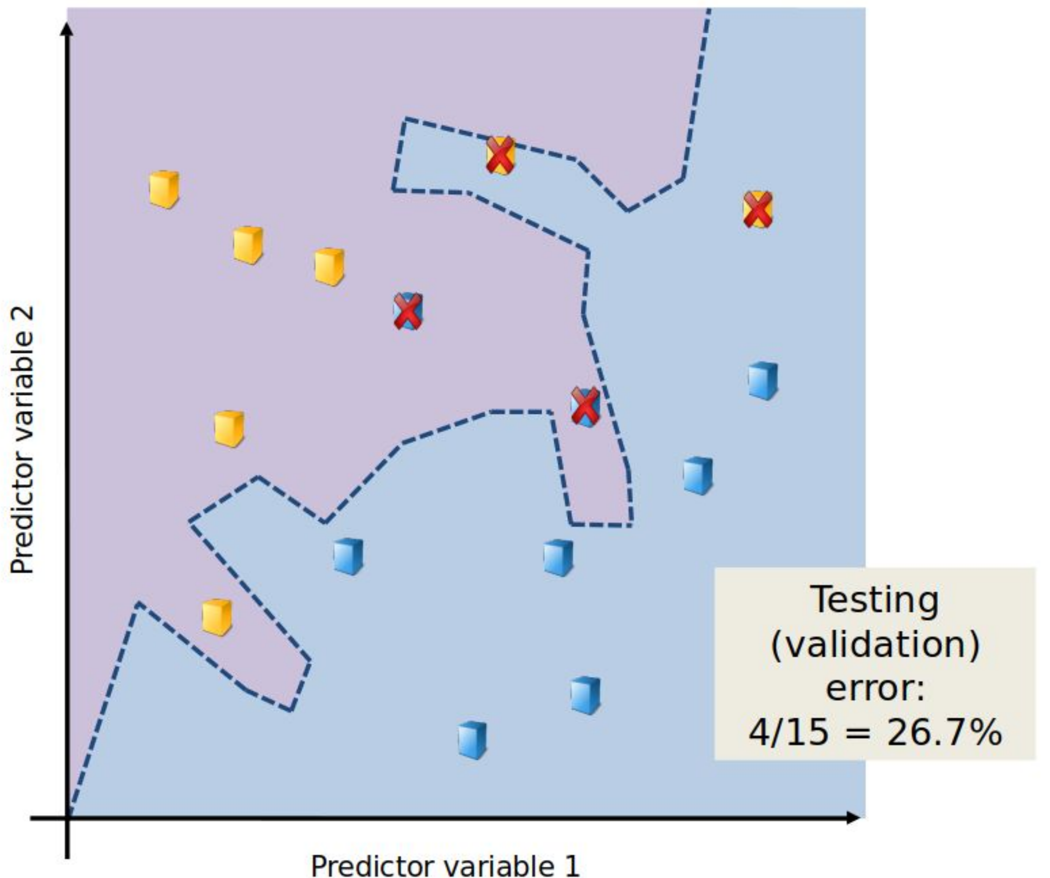
Training and testing



Training and testing



Training and testing



Training, Validating, Testing

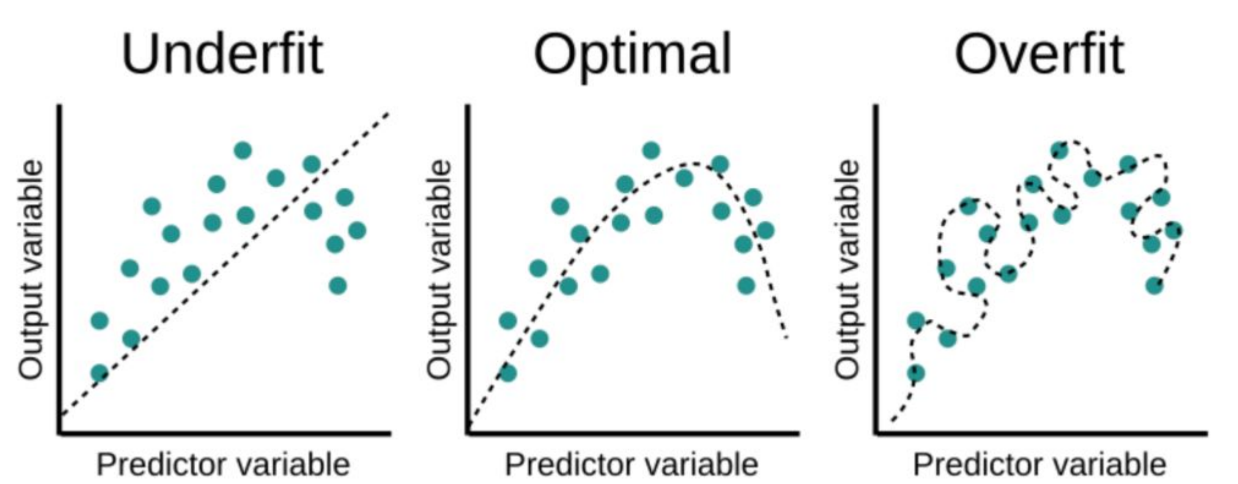
In order to construct and study a classifier built with a supervised algorithm and given a sample of signal events and background events, one usually separates these sets in three parts:

- **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 with “high bias” pages very little attention to the training data and over simpli es the model

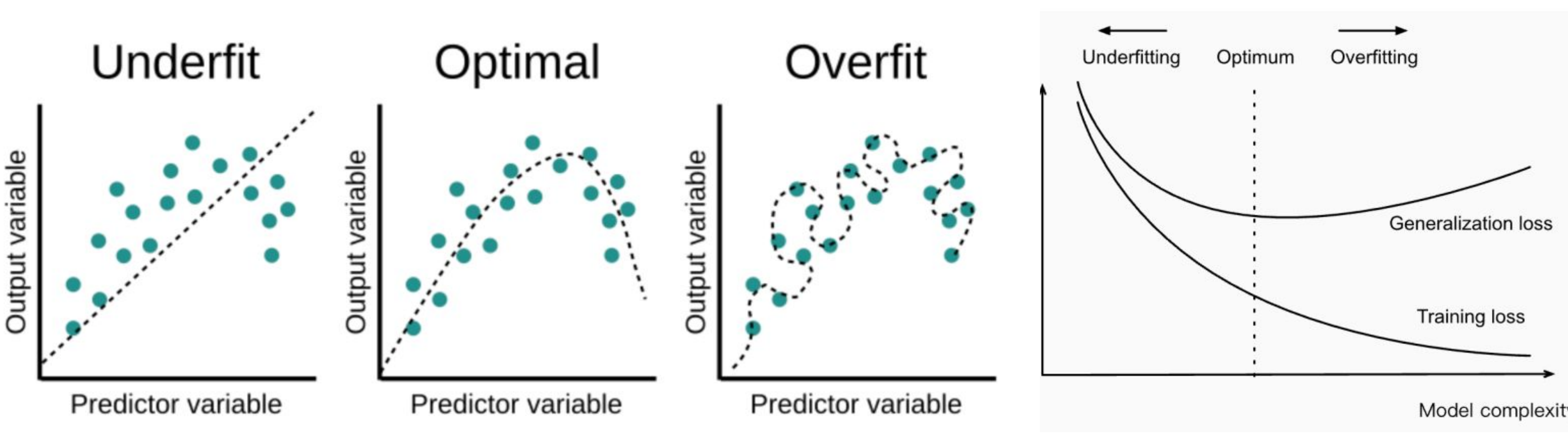
- Performs poorly on training as well as testing data ➡ Underfitting



- Increase model complexity
- Increase the number of features, performing feature engineering
- Remove noise from the data
- Increase the number of epochs or increase the duration of training to get better results

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 over tting. Data augmentation helps to increase the size of the dataset

Regularization: L1/L2

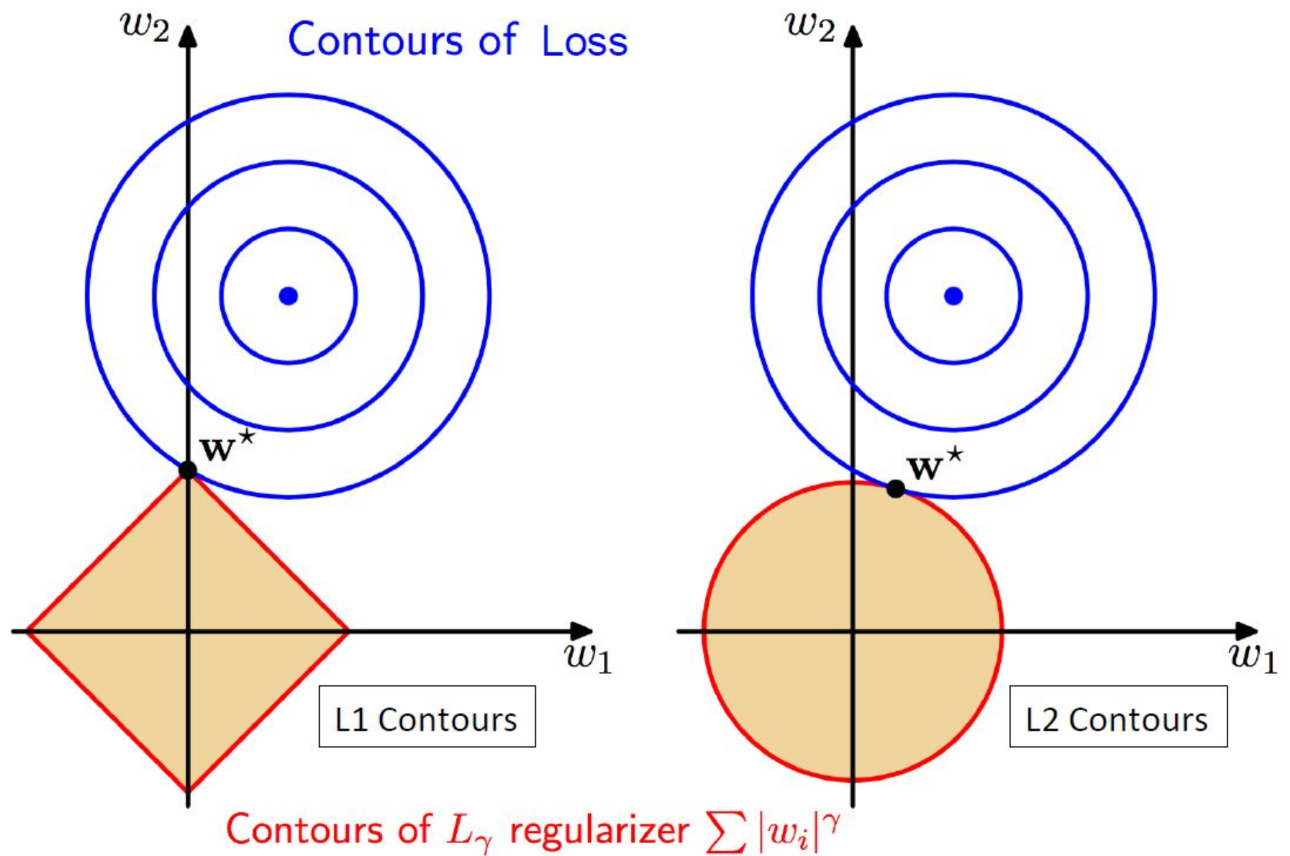
Regularizer on the NN weights (absolute values of weights / squared weights)

L2 regularization is perhaps the most common form of regularization
For every weight, w, in the network we add ½ λw^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:
½ $\lambda w^2 + \lambda\ |w|$

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

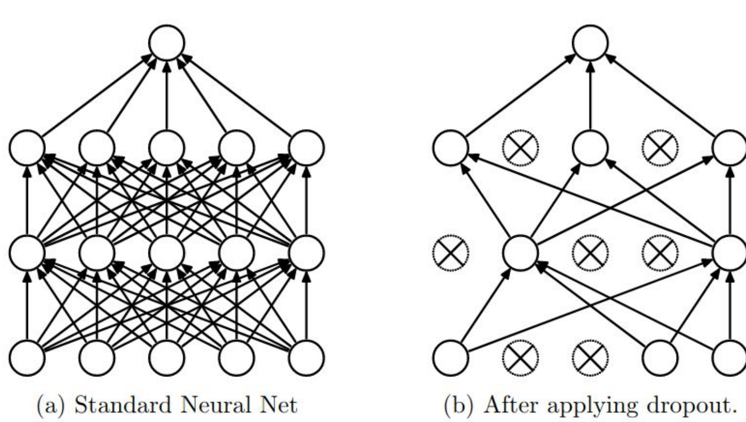


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

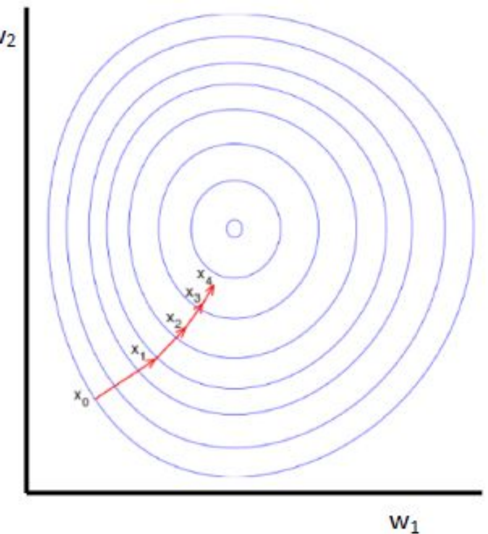
Loss minimization - Gradient descent

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:

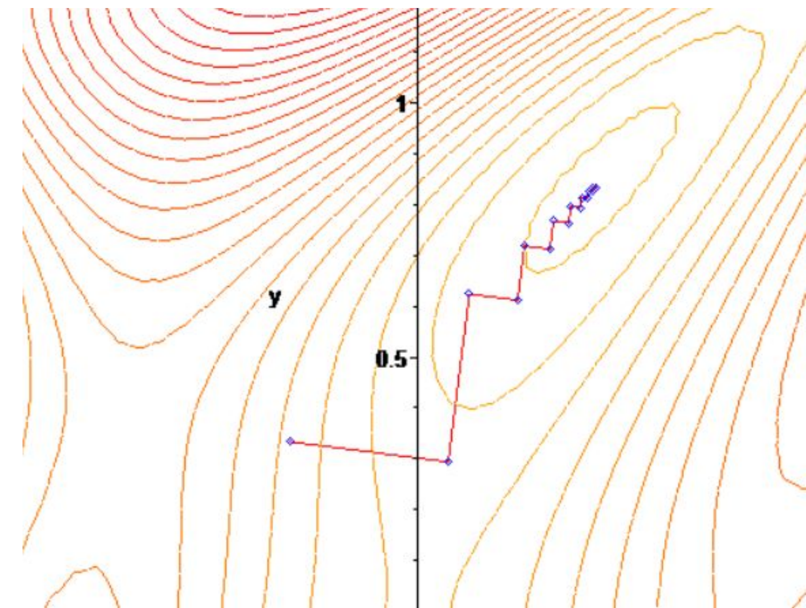
- CPU-intensive (must pass all dataset)
- large memory use, intractable if too large datasets
- does not allow updates on-the-fly (adding data online)
- Also, it can become ineffective, as risk of getting stuck in local minima is large in multi-dimensions

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 nal performance should be attainable using any batch size (see Shallue et al. 2018)

Hyper-parameters tuning

Choose the Choosing 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.

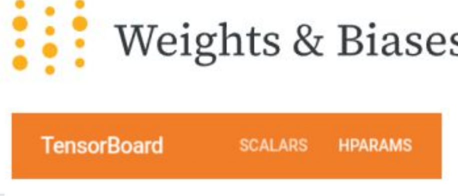
Choosing the number of training steps involves balancing the following tension:

- 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



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

- **Pick the one you like!**

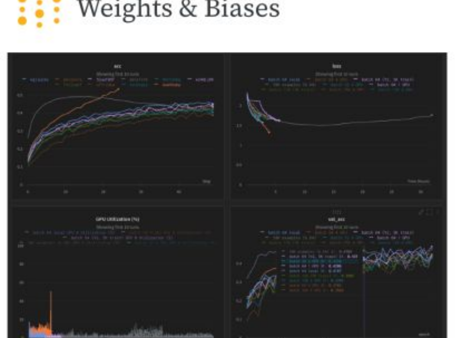
Tools for ML experiments visualization

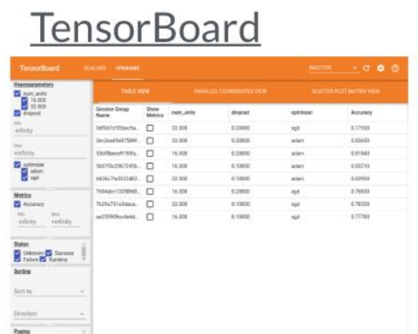
You need to do some or a lot of experimenting with model improvement ideas

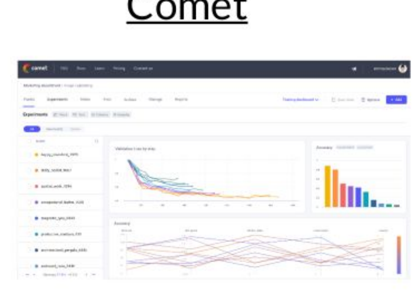
- Visualizing differences between various ML experiments becomes crucial

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

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







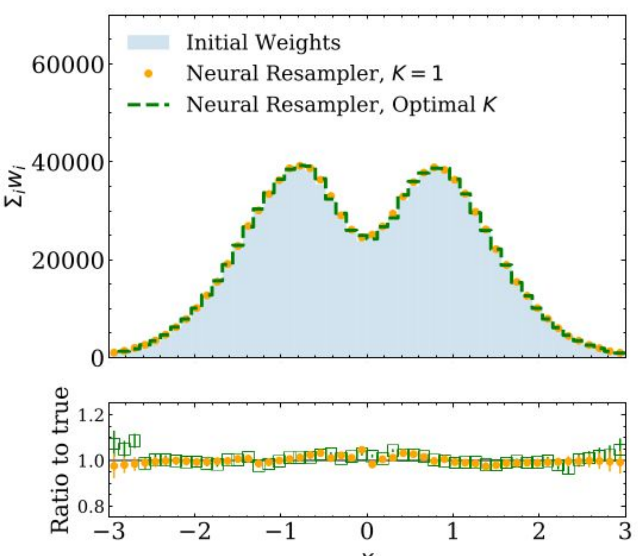
for managing the end-to-end machine learning lifecycle

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



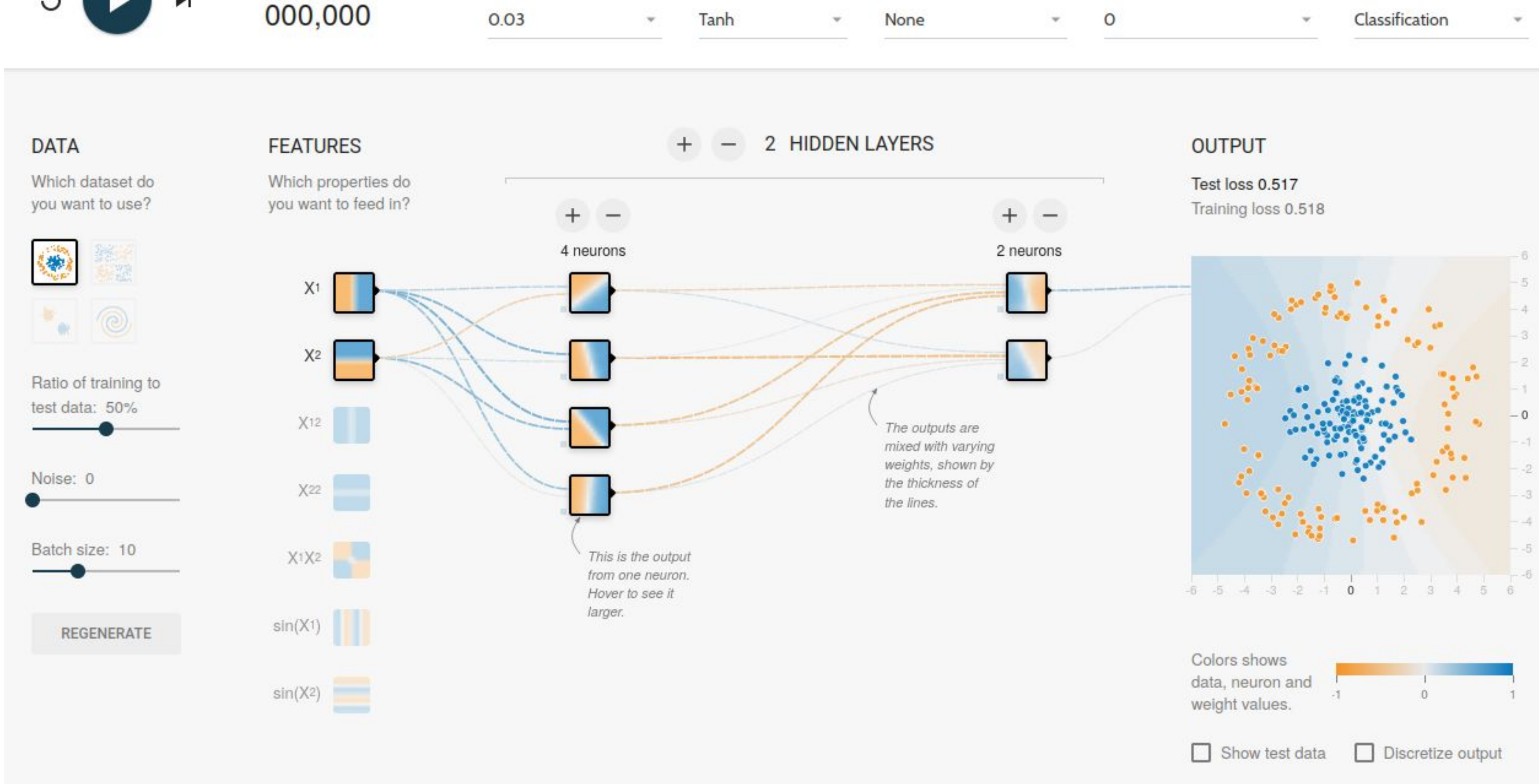
Hyper-parameters optimization

Today's focus will be on 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

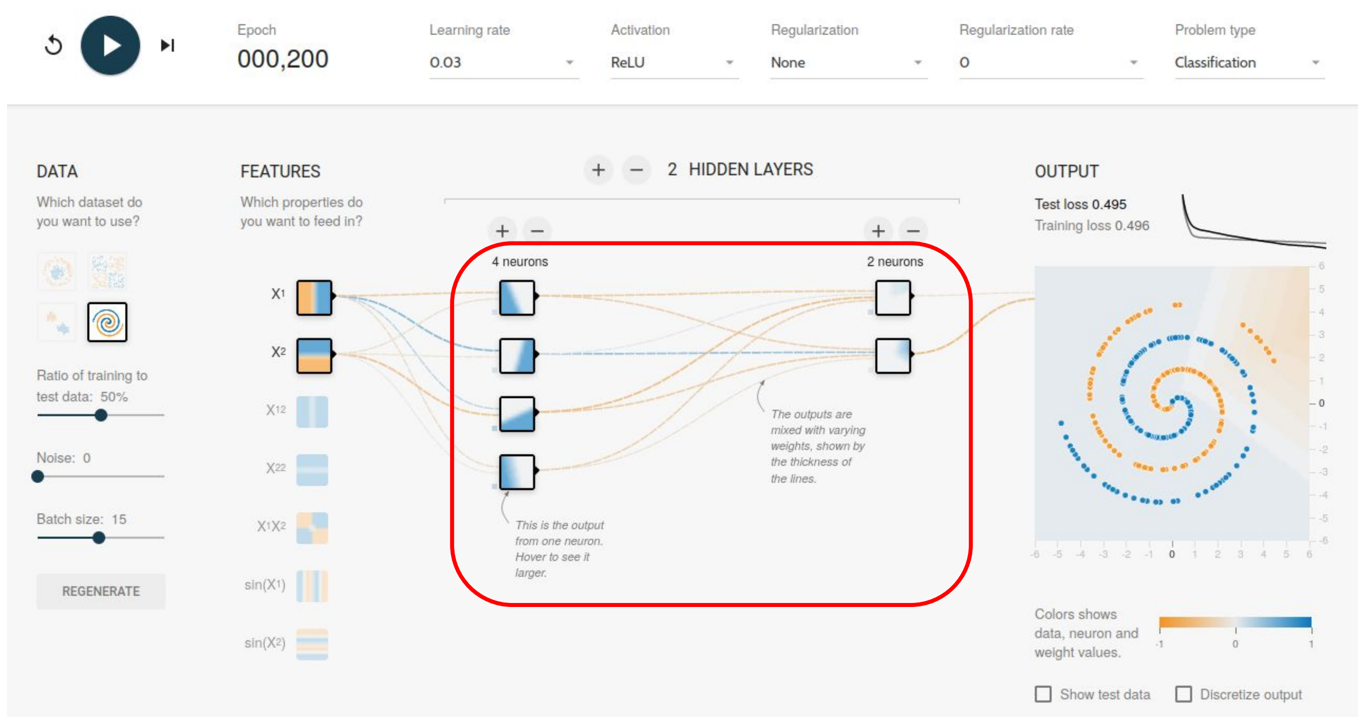
Let's play with NNs!

http://playground.tensorflow.org



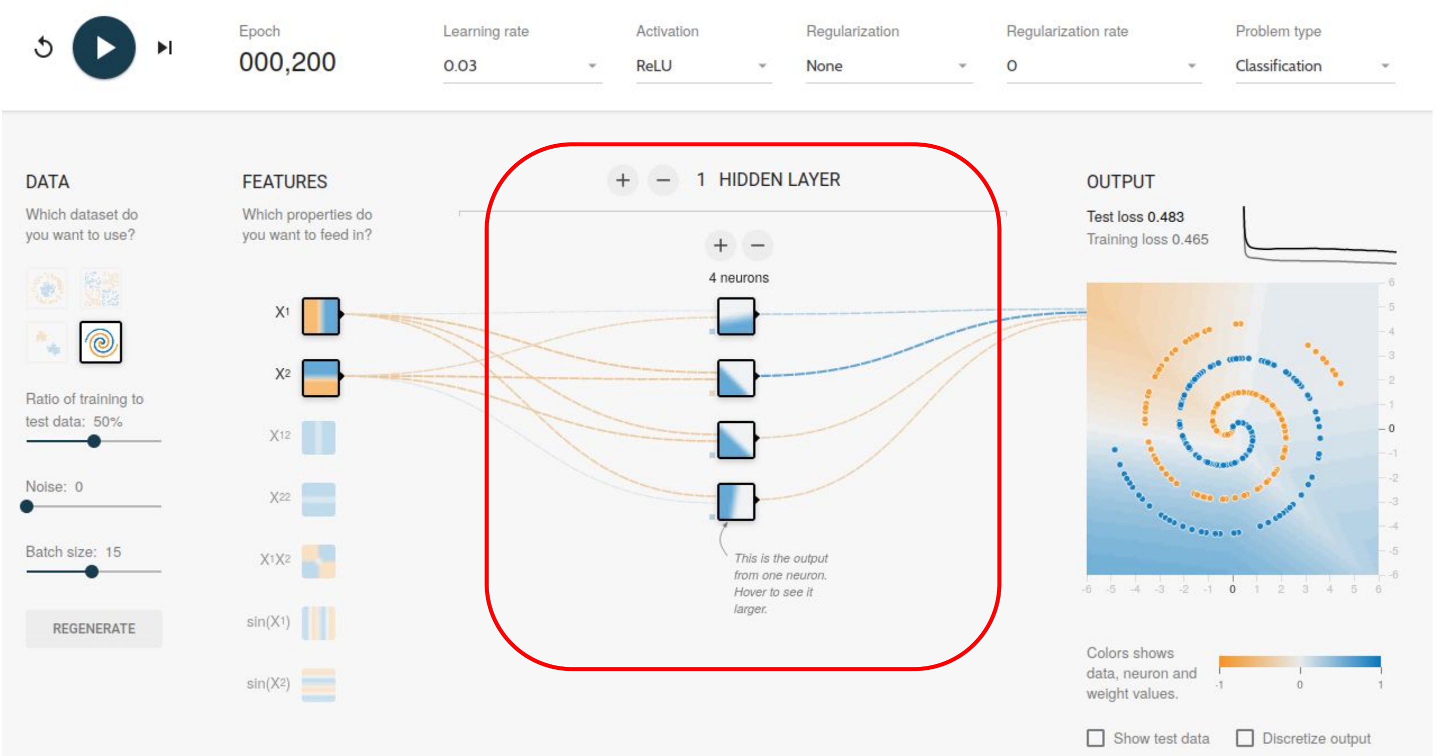
Width

- Number of neurons in a layer



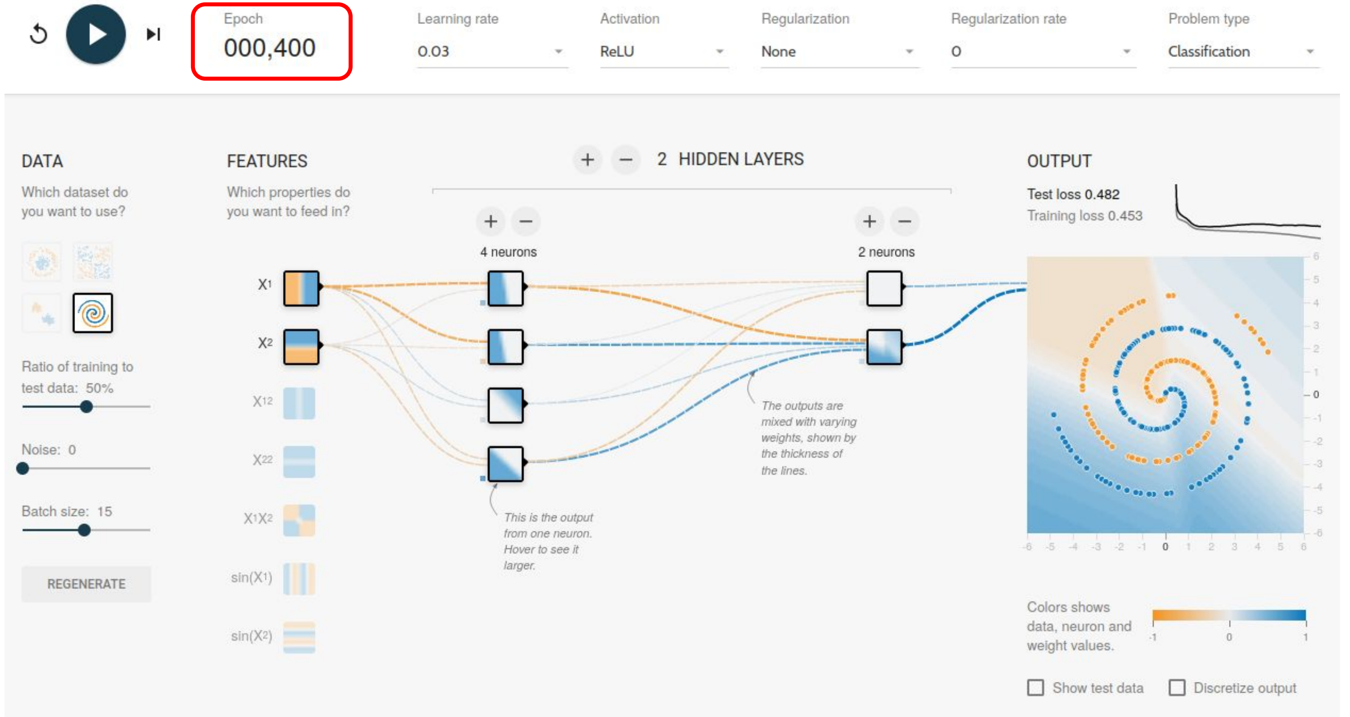
Depth

- Number of layers



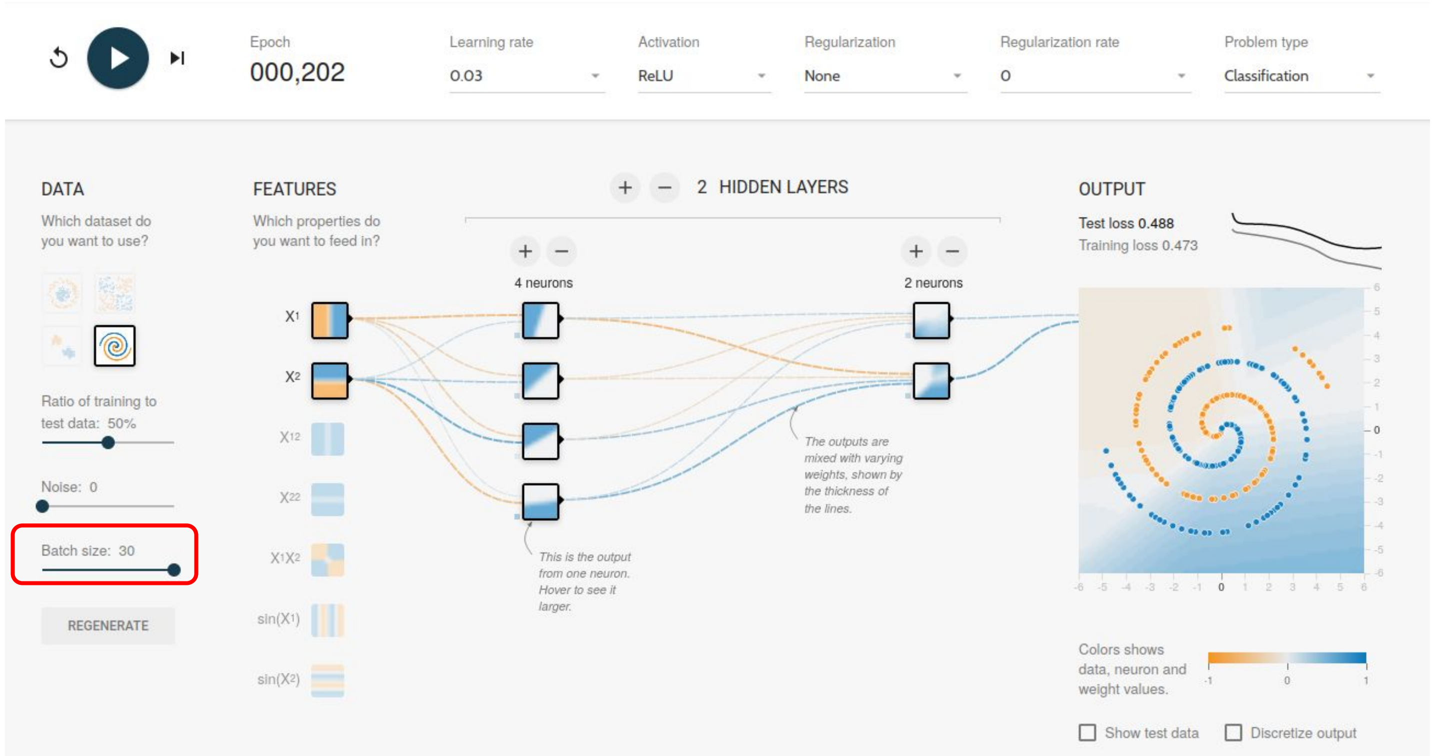
Epochs

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



Batch size

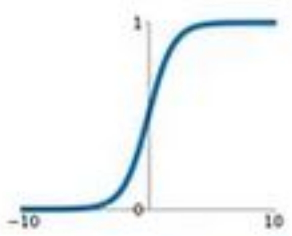
- Number of training instances in the batch
- batch_size=128 means that there are 128 training instances in each batch



Activation functions

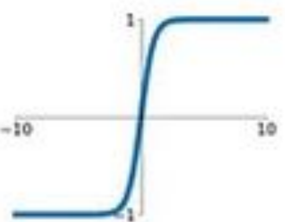
Sigmoid

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



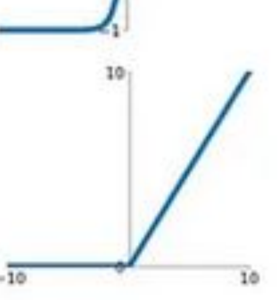
tanh

$\tanh(x)$



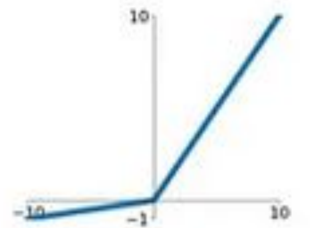
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

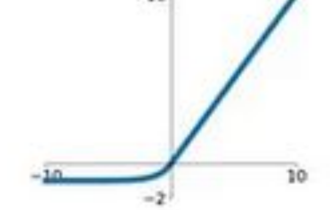


Maxout

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

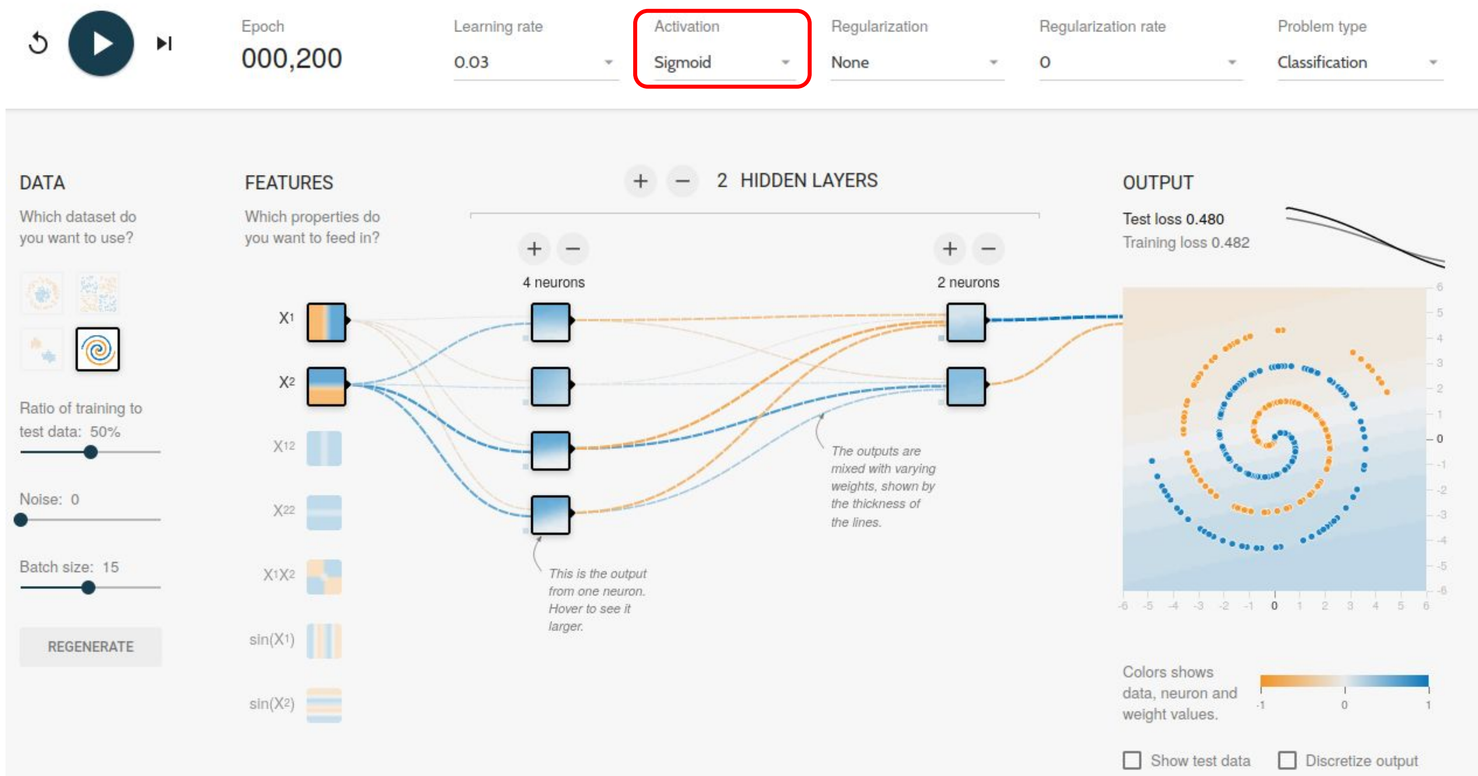
ELU

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



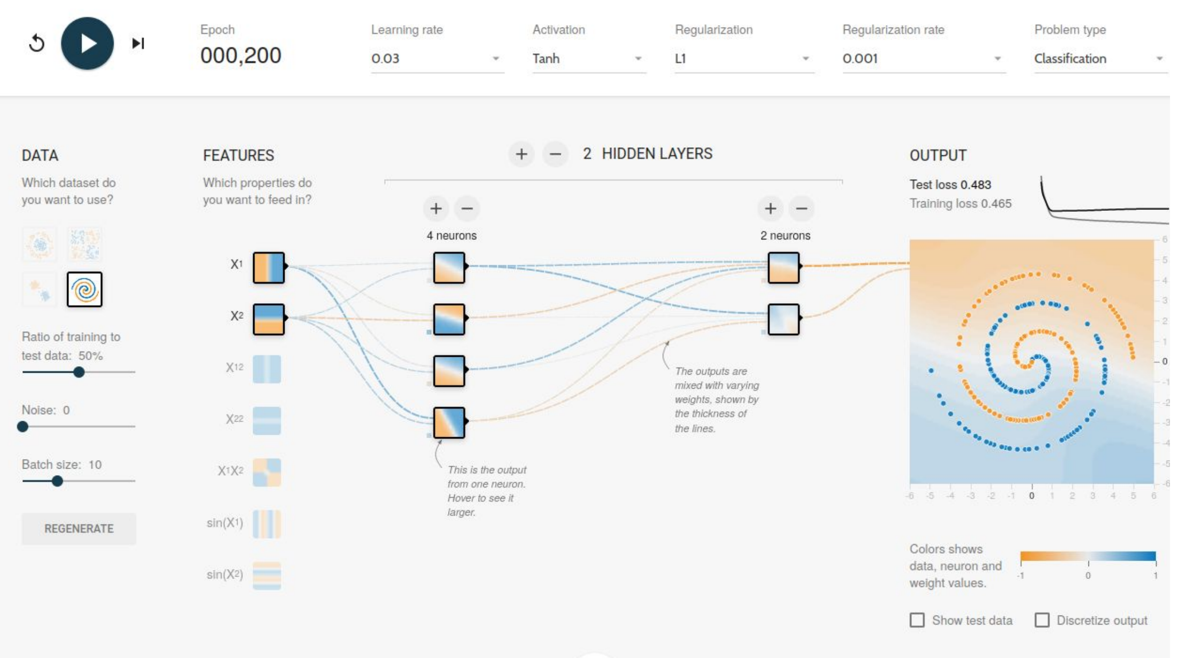
Activation

- Start with Relu/LeakyRelu/Sigmoid



Optimiser and learning rate

- Start with Adam (momentum + learning rate manipulation)
- Adam typically requires a smaller learning rate: start at 0.001, then increase/decrease as you see fit



Dropout

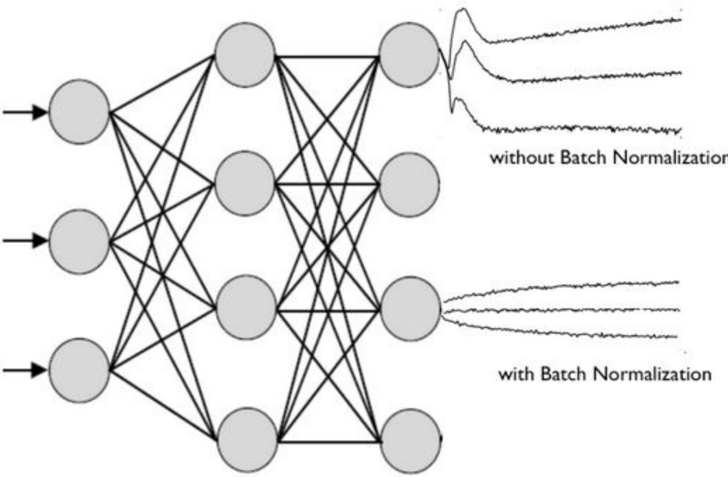
- Probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no outputs from the layer
- A good value for dropout in a hidden layer is between 0.5 and 0.8
- Input layers use a larger dropout rate, such as of 0.8

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**

“Dropout: A Simple Way to Prevent Neural Networks from Overfitting”

Batch normalization

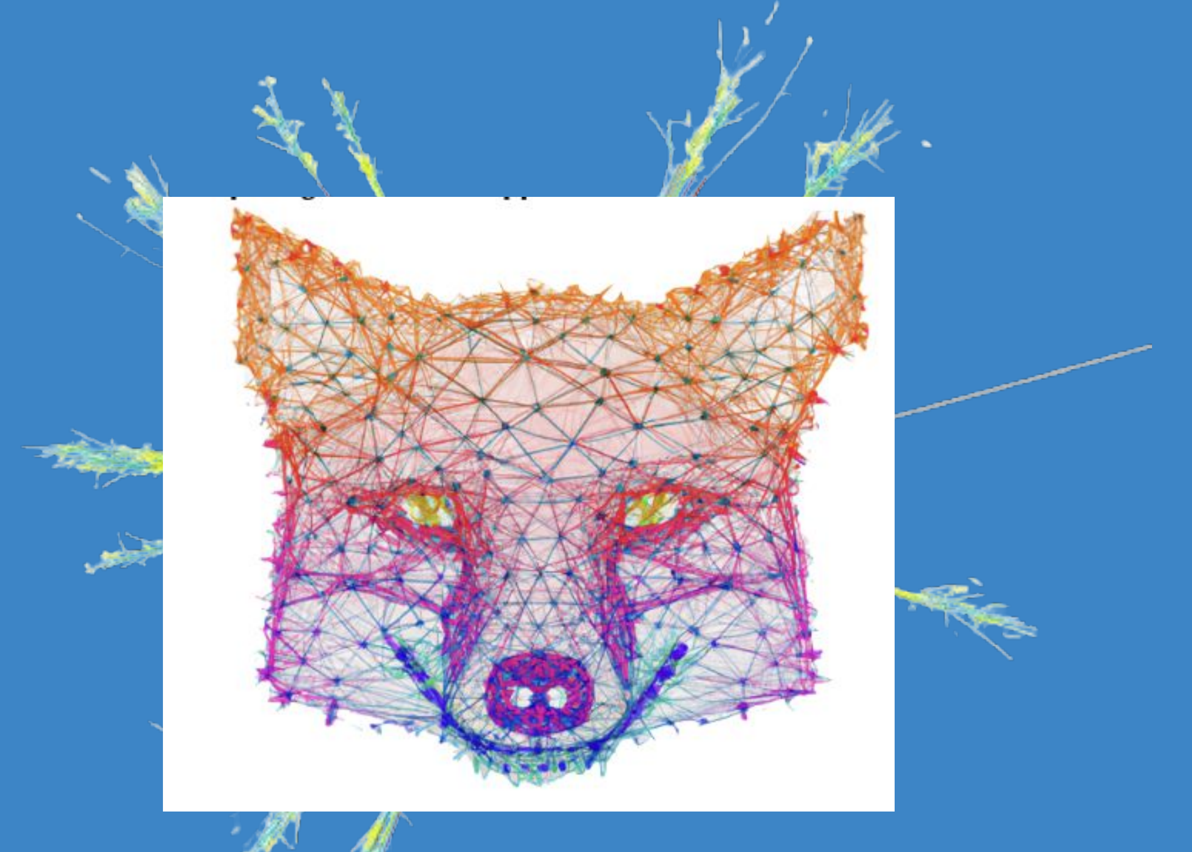
- Normalization technique done between the layers of a Neural Network instead of in the raw data
- Done along mini-batches instead of the full data set
- Serves to speed up training and use higher learning rates, making learning easier



node outputs, with and without Batch Normalization

Let's play with BDTs!

BDT playground



Thanks for the attention!