Predictive Model for Optimizing Grid Utilization

Intern: Antonio Di Bello

Supervisor: Michael Kirby

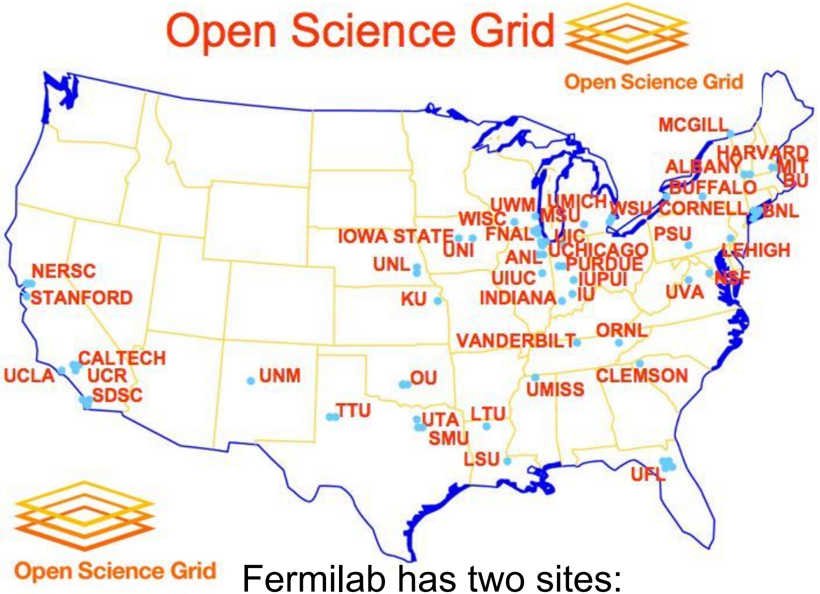
Open Science Grid and HTCondor

HTCondor is a scheduling and provisioning software that matches resources to computing workflow requirements.

Users submit their jobs to HTCondor, which collects jobs together and places them in a queue for matching with resources.

Once resources are available and based on "fair-share priority" ordering in the queue, jobs execute on a site on the OSG.

Each job belongs to an experiment (Virtual Organization), that has a specific allocation (Quota). User can exceed quota only when other VOs are not utilizing their quota.



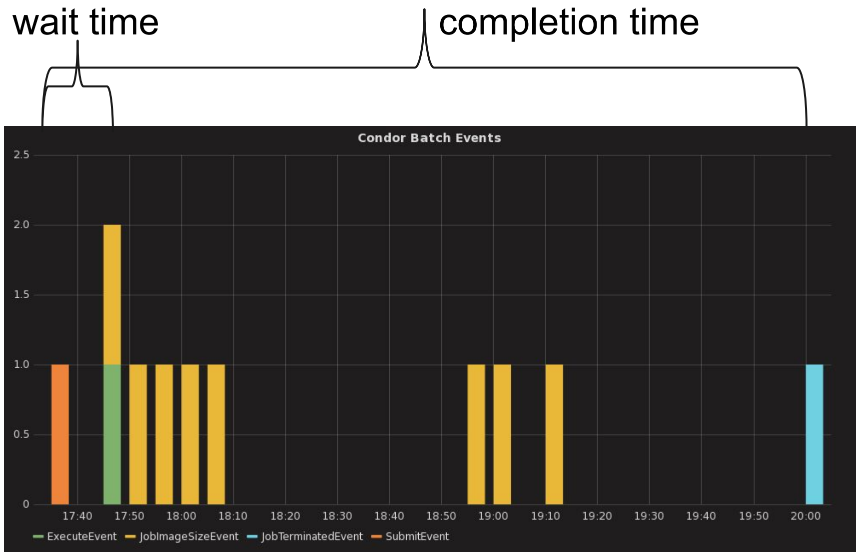
Fermilab has two sites:
FermiGrid and CMS Tier1

Goal :

We want predict the necessary time for one job (once it is submitted) for exiting the queue and starting its execution.

Once a user submits a workflow, it is difficult to predict when the jobs will start running and when they will complete.

This time depends from many parameters that are uncorrelated between them (and there isn't an analytic model), then we have decided to develop a Deep Learning model.



Project phases:

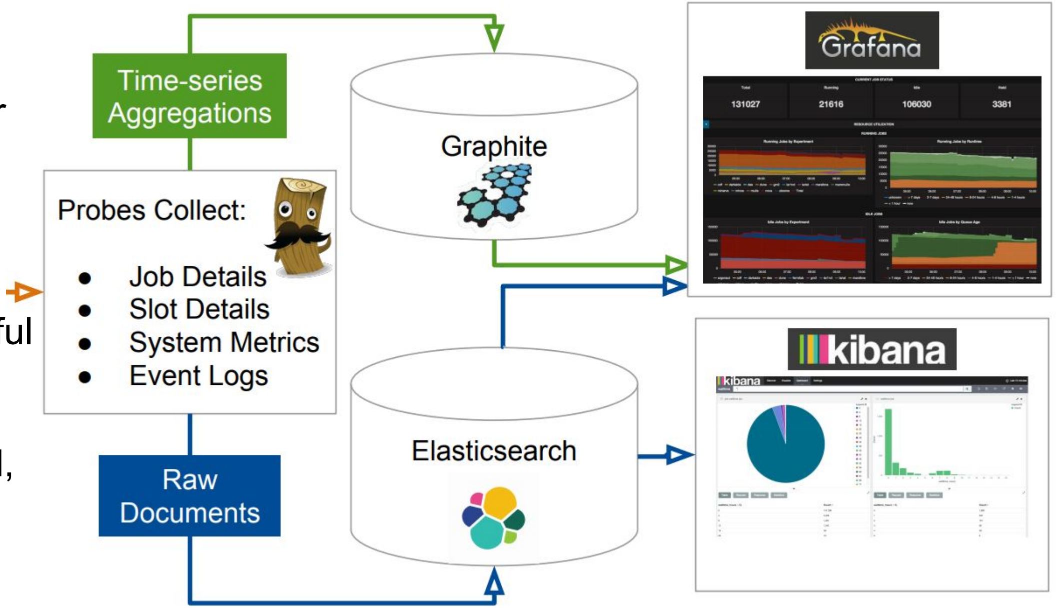
1. **Collect** historical job submission data from system logs.
2. **Design** a multivariate algorithm (MVA) to predict the start time of the workflow.
3. **Train** and **test** the MVA.
4. **Deploy** the application into production for users to have instant feedback about when their jobs will start.

Historical Open Science Grid Information

Two main different of monitoring:

- Time series - Which is useful for drawing graph.
  Example : running job by VO, quota, etc.

- Raw Documents - Which is useful for general purpose query.
  Example: job submission times, site status, resources requested, etc.



Collect the Job data

The information related to a single job(cluster) are:

- Time of submit
- Time of execution
- Time of terminate
- Required resources (Core,Memory,Disk,Time)
- Experiment (Virtual Organization)
- Site where executed

Used to calculate wait and delay time

We collect this info through a python script that performs an ElasticSearch query on Kibana.

```
19150155.0@fifebatch1.fnal.gov
sub: 2017-08-01T00:10:08   exe: 2017-08-01T00:10:25   end:2017-08-01T01:12:21
wait time : 17.0    delay time : 3733.0
19150156.0@fifebatch1.fnal.gov
sub: 2017-08-01T00:10:12   exe: 2017-08-01T00:10:34   end:2017-08-01T01:15:32
wait time : 22.0    delay time : 3920.0
```

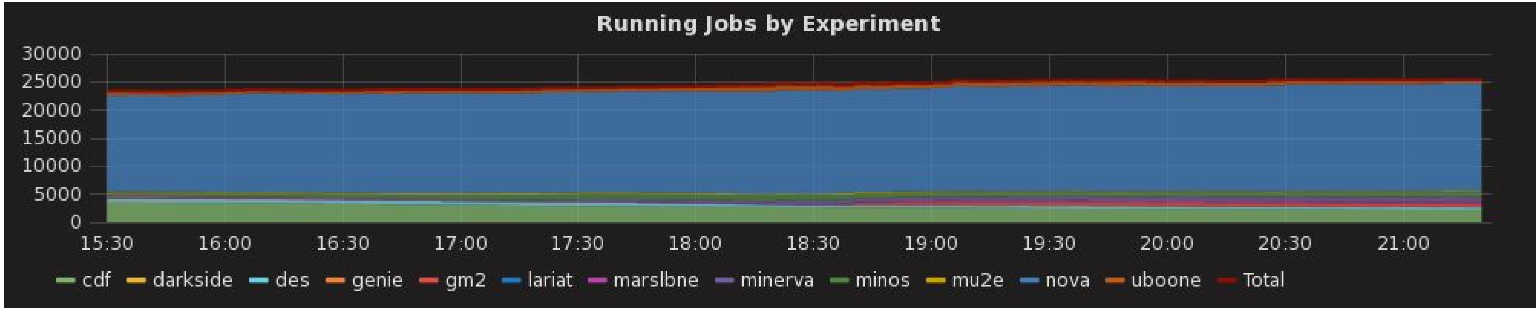
Collect the System data

The data about the system status is overwhelming and complex - difficult to define which is important or not, and create analytic prediction from “first principles”

In order to collect them, we directly query Graphana that tell us all the historic information.

Collected data:

- Running jobs
- Idle jobs
- Run by V.O.
- Idle by V.O.
- V.O. Quota



How collect the data

1. Make only one query of aggregation type.
2. For each cluster:
   Verify if exist at least one process that has a Bad Event
3. Obtain cluster info (res required)
4. Get the last termination event
5. Get the related system information

Query on Graphite, resolution of 5 min

We make only 2 query plus 1 GET request.
We should avoid to make so many queries, otherwise we could overload the server.

We collect about 6 million of sample, about 1 GB of text data.

Next step : NN topology

Which topology is better for our purpose ?

CNN , RNN ...

Maybe we can test them and choose consequently.

Problem of Regression:

We want a continuous output, not a classification.

NN topology

Linear Model:

The output is a weighted sum of the input, plus a bias.
One layer.

Recurrent NN:

There is a directed cycle between neurons.
The net learns the dynamic temporal behavior.

Convolutional NN:

Made of three layers types:

- Convolution
- Pooling
- Dense

Possibly repeated many times.

Combination Linear + Deep:

A mix of linear and nonlinear approach

Long/Short Term Memory:

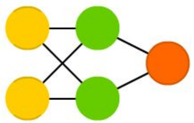
Like RNN, but the net can choose when “remember” and when “forget”.

. . .

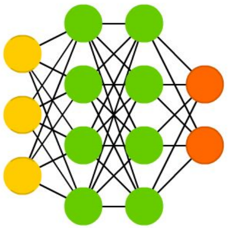
Graphics example

Different type of topology precedently illustrated :

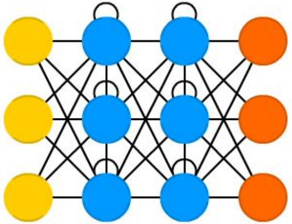
Feed Forward (FF)



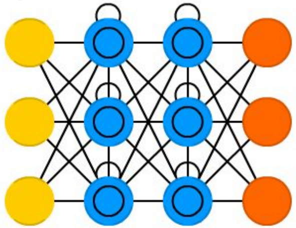
Deep Feed Forward (DFF)



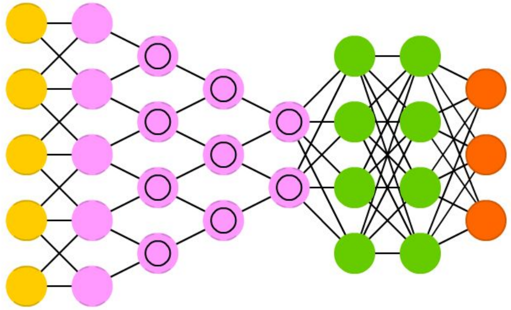
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)



Deep Convolutional Network (DCN)

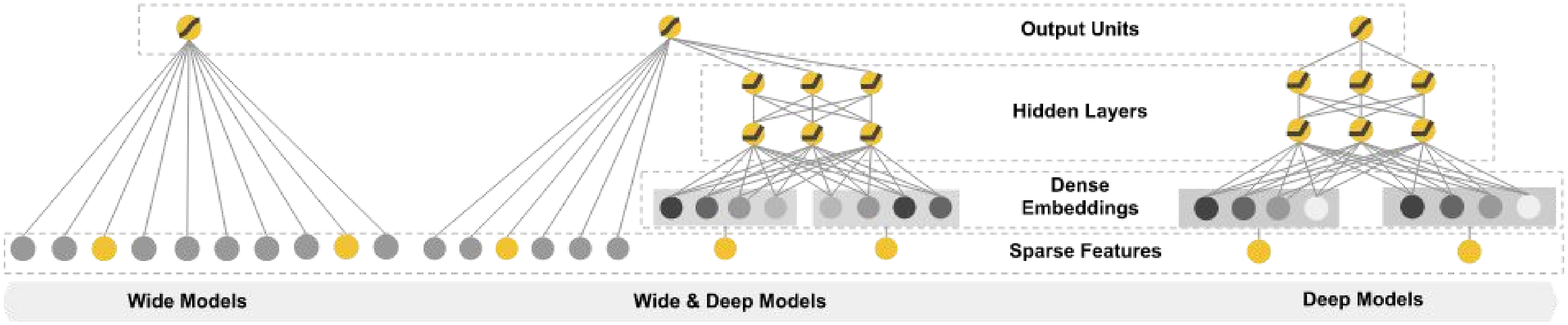


A first Idea: Wide and Deep Models

This approach combines the strengths of memorization and generalization.

Linear models with crossed features can memorize an “exception rules” effectively with fewer model parameter. (Very specific feature pairs)

Through dense embeddings, deep models can generalize better and make predictions on feature pairs that were previously unseen in the training data.



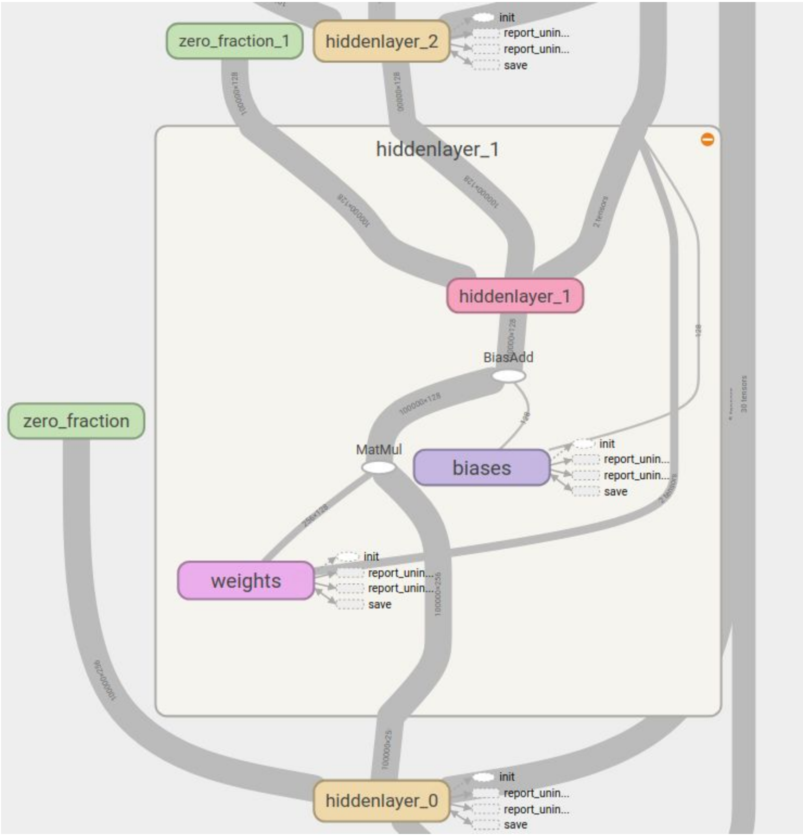
DNN Model

Parameter to define a DNN:

- Number of layer, Neuron for layer
- Input Features
- Optimizer
- Activation Function
- Dropout

Topology: many Dense Layers.

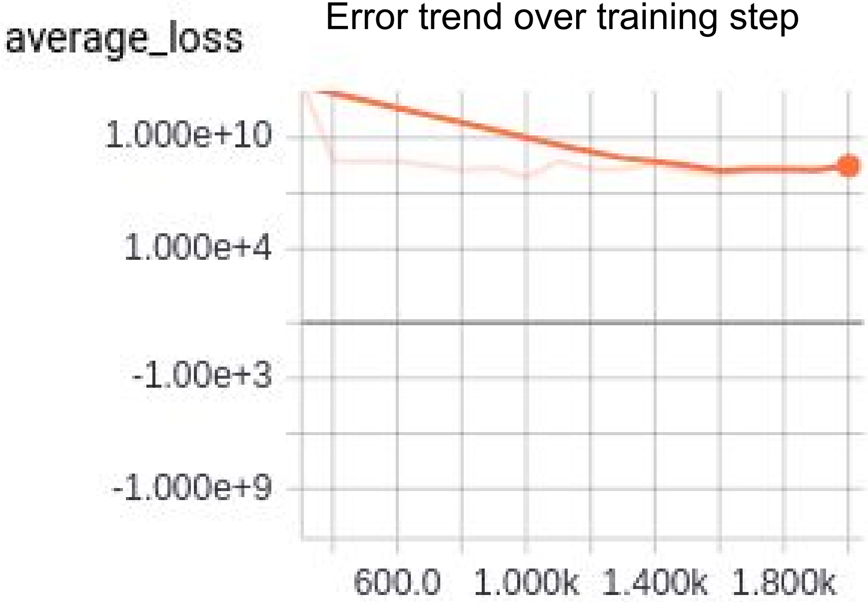
Layers where each neuron is fully connected with the others of the next layer.



Run the model

1. Train the model (with the train data)
2. Evaluate the model (with the test data)
3. Predict the result (with new data)

Error trend over training step

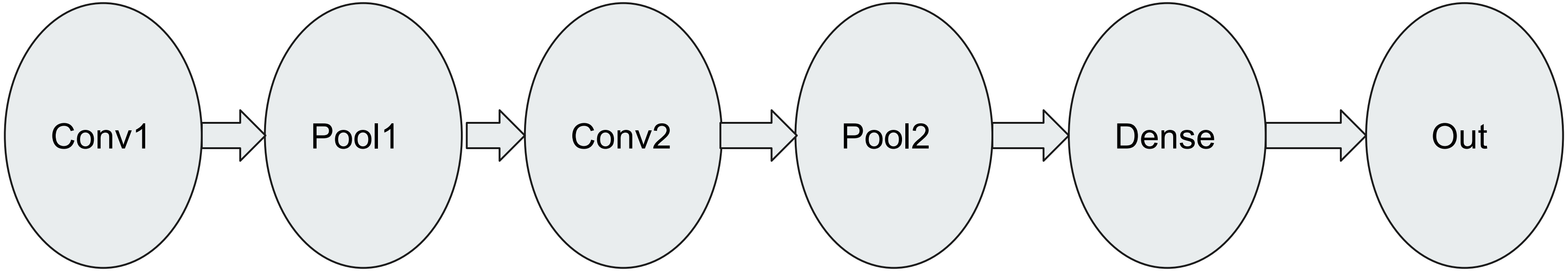


$$Loss_{average} = \frac{1}{n}\sum_{i}^{n}(y_i - \widehat{y_i})^2$$

$where\ y_i\ is\ the\ real\ wait\ time\ and\ \widehat{y_i}\ is\ the\ predicted\ wait\ time$

Custom Model : Convolutional NN

In order to get better results we can define our custom Neural Network.

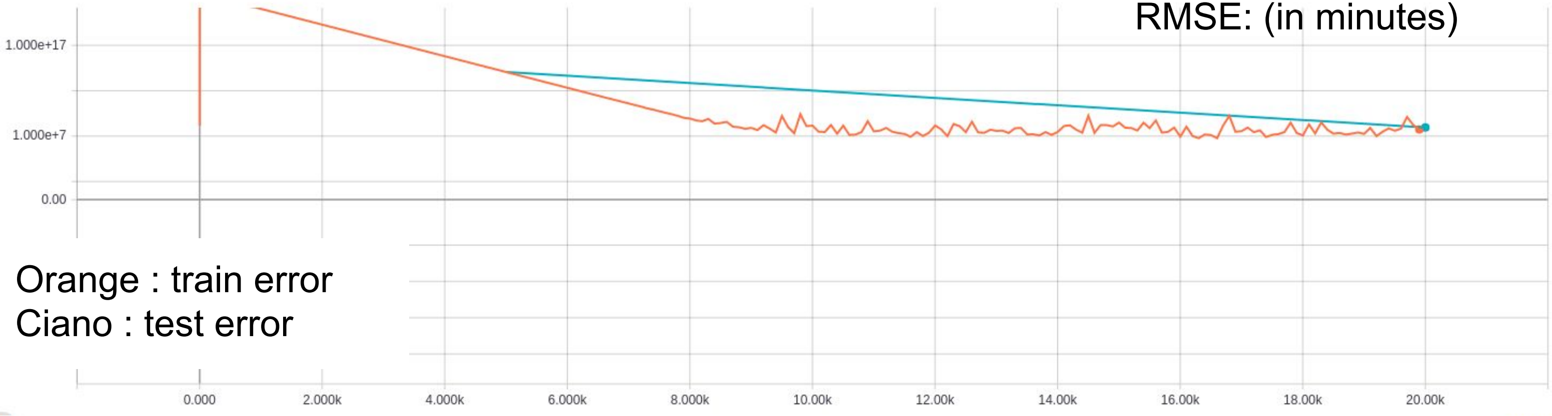


Result

Unfortunately in both cases, we have an error too bigger.
Indeed we have about on average one hour of error respect to the real value, and a root mean square error about of a couple of hours.

We evaluate with test data every 5k of training step.

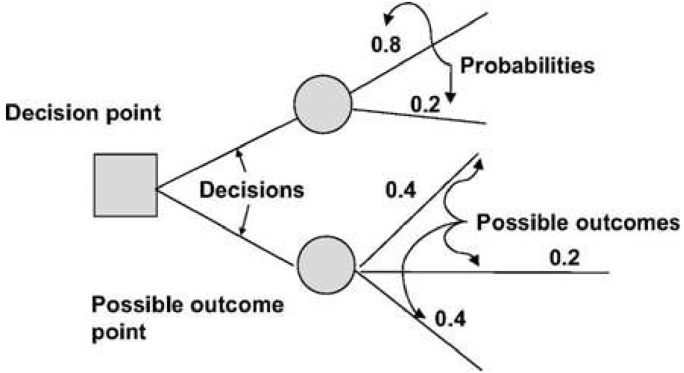
RMSE: (in minutes)



Orange : train error
Ciano : test error

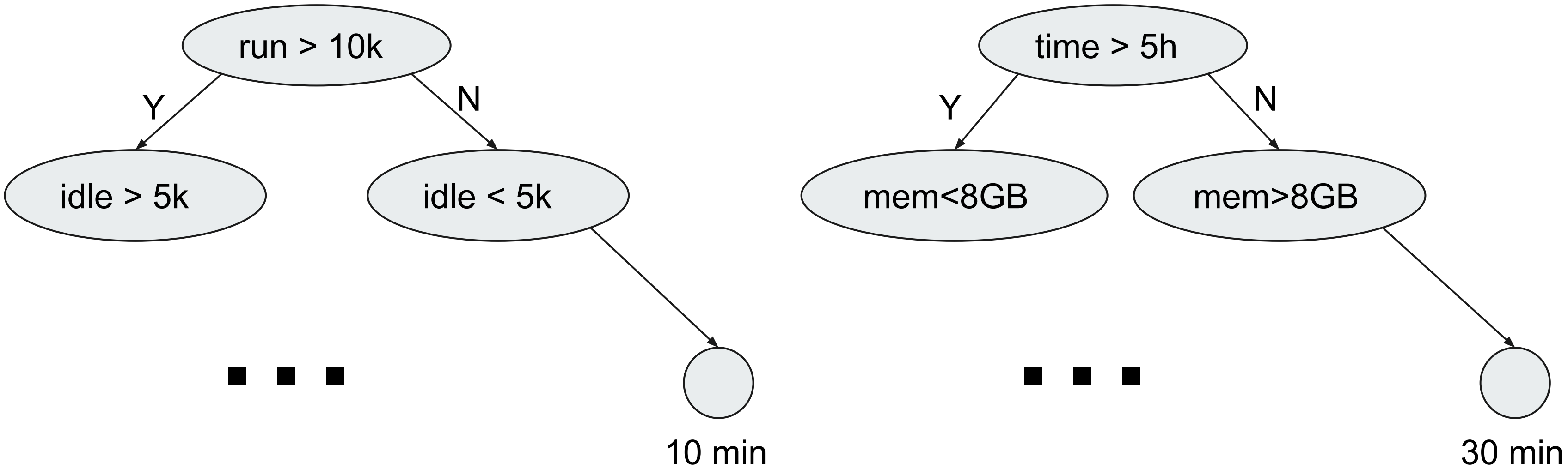
New approach: Forest of Decision Tree

Decision tree: a tree in which at each level it’s used one or more feature to make a decision of which branch take to arrive at the “right” leaf.



This is a weak learner, but if we have a large number of tree, we can combine their results (value of the chosen leaf) to obtain a good result !

General Idea



Generate many trees and combine in a weighted mode their predicted value.

XGBoost

So we first generate some random tree, so we expand them until some level and then prune them. So we test the quality of the result with the train data, decides which trees are better to add at the forest, and after we continue by make others trees and so on.

XGBoost is a library that can speed this algorithm, running it in parallel and doing multiple optimization.

Crossed Features

They can highlights some type of knowledge that otherwise our model could ignore.

To get better result we can create a new feature that represent for example the available resources in the exact experiment in which the job is submitted.

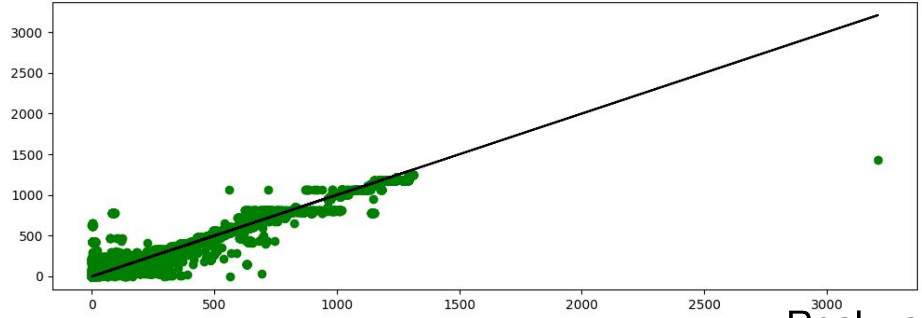
$$Avaibility = \frac{quota - run}{quota}$$

Crossed features are like artificial features where we have the total control when build they.

If we wisely put here some important knowledge, the model can better learn our sample.

Result

Predict wait (min)



Real wait (m)

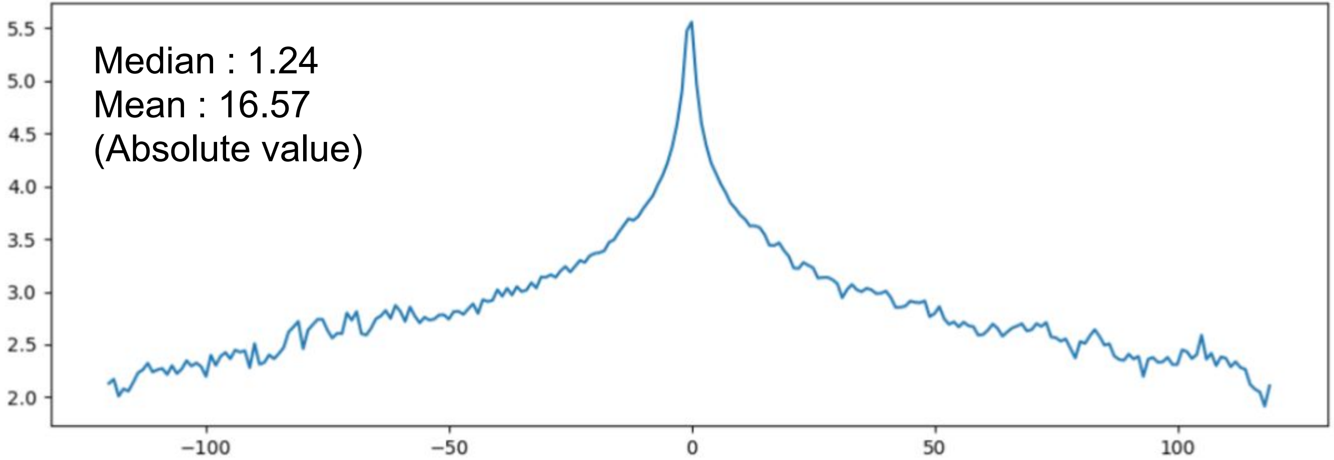
Visualization of test sample over a plan (true value / predicted value).

Ideally we want the point most near possible at the black line
(a hit without error).

We accept predictions with an error less than 15 minutes.

This is only a qualitative representation, not a quantitative.

Error Distribution



Median : 1.24
Mean : 16.57
(Absolute value)

The distribution is almost symmetric respect to the origin.
Mean(not absolute) : 0.079

Error distribution over a **logarithmic** base 10 scale.
On **y-axes** we have a logarithm number of occurrences of the error valuated on x variable.
On **x-axes** we have the error expressed in minutes.
Negative one indicates an underestimation respect to the real time,
positive one an overestimation.

Evaluation of error distribution

Abs error tolerated	N. of fails	Percentage of success
15 minutes	354,009	82.45 %
20 minutes	303,317	84.96 %
30 minutes	242,612	87.97 %
45 minutes	184,432	90.85 %
60 minutes	146,296	92.74 %
90 minutes	96,838	95.20 %

Distribution of errors over many value of tolerating error.

Train on 4096,690 sample. Test on 2017,774 sample.

If we ignore all the error of the prediction over the 8 hours,
we can increase the percentage of success (15 min) to 85.65%.

Almost all the errors (about 90%) are less than 45 minutes.

We have reached a good result !

Future Work

In order to get better results we can do some easy improvement :

- Increase the number of training data used (now Jan-Sept , 6M of records)
- Adding other features like priority of the user, other VO, others parameters.
- Others crossed features that explain other high level concept that otherwise the model could not consider.
- Consider the jobs that have been suspended (but not killed) for some reason.
- Tune better some specific parameter.
- increase the time of computation and explore more deeply some path early pruned in order to build a larger model

Summary

1. Now we have **collected** the useful Jan-Sept data from the system logs.
2. We have design a naive **topology** of the Neural Network. We tried also with custom one (CNN) .
3. We have applied the **Boosted Decision Tree**.
4. We will **deploy** the application into the system.

Thank you for the attention.

Question ?

Backup Slide

How collect the data - 1

1. Make only one query of aggregation type.

Group first by schedd, then by cluster.

We retrieve one entry per cluster with the submission time and execution time.

Submission Time : the earliest time between all the submission time of the process in the same cluster

How collect the data - 2

1. Make only one query of aggregation type.
2. For each cluster:
   Verify if exist at least one process that had a BadEvent

Search by cluster, schedd, BadEvent

We retrieve zero entry,
we are interested only at
the total number of hits.
(If is equal zero or not)

BadEvent example: If a job requires more resources than the expected, its execution is paused.(JobHeld)
We remove these clusters because concerned about bias in initial development.

How collect the data - 3

1. Make only one query of aggregation type.
2. For each cluster:
   Verify if exist at least one process that was helded
3. Obtain cluster info (res require)

Directly by a GET request on an index

We take the info about process zero, because inside the same cluster, all the process have the same requirement.

Resource requested : The resources (cpu,memory,disk,time) that an user expect to use for the cluster (job) - this request determines what resources can run the workflow.

How collect the data - 4

1. Make only one query of aggregation type.
2. For each cluster:
   Verify if exist at least one process that was helded
3. Obtain cluster info (res require)
4. Get the last termination event

Sort by desc, take the first result

We need from the log the last termination event in the cluster in order to calculate the execution time.

Execution time : time between the first start execution (of any process) and the last terminate execution (of any process) , in the same cluster.

How collect the data - 5

1. Make only one query of aggregation type.
2. For each cluster:
   Verify if exist at least one process that was helded
3. Obtain cluster info (res require)
4. Get the last termination event
5. Get the related system information

Query on Graphite, resolution of 5 min

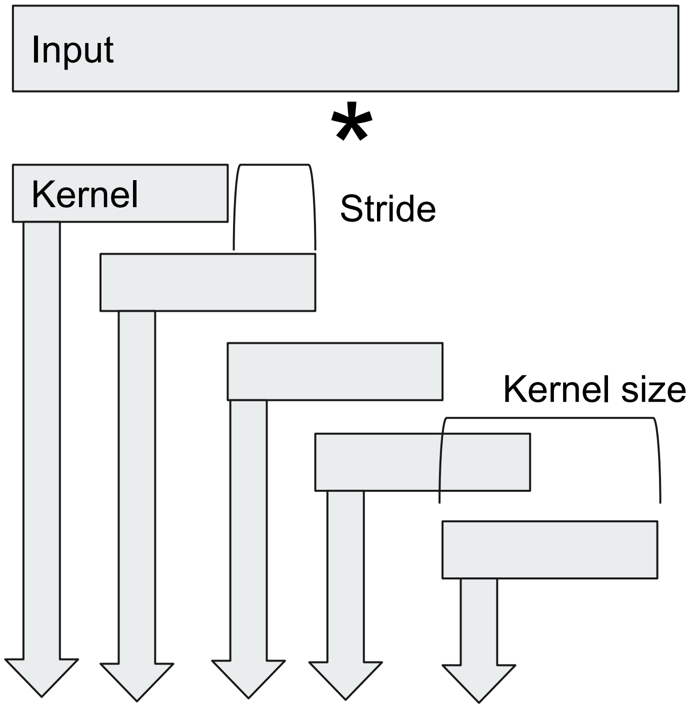
We take the information from Graphite because here are already grouped by time.

System information : cumulative info about the system, like total running job, total idle job, also grouped by experiment. maybe comment on quotas?

Convolutional Layer

We define the convolutional (1 dimension) layer specifying the number of features, the kernel size, and optionally the stride.

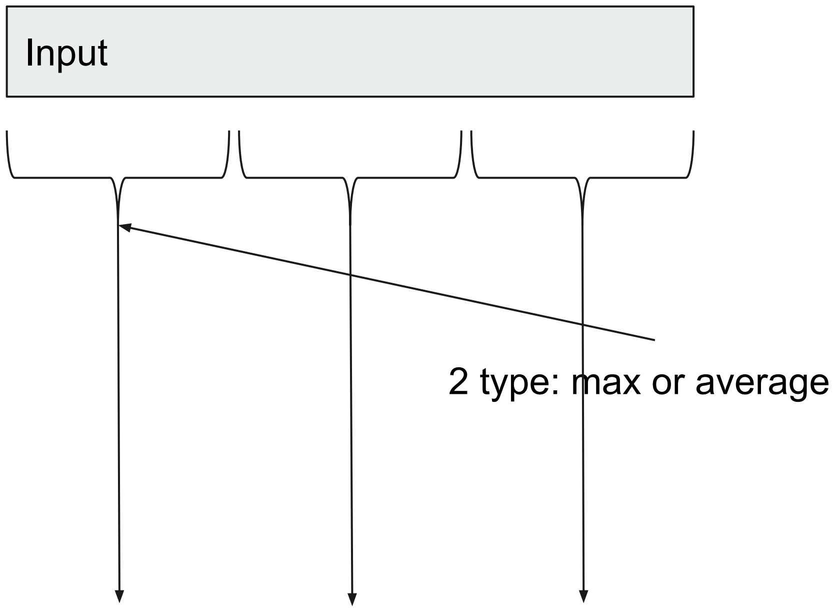
Example for one features :



Pooling Layer

In this layer we reduce the space by doing a reduction operation like maximum or average.
We can choose pool size and stride.

In this case the stride have the same size of the pool.



Dense Layer

This layer is a fully connected layer where exists a link between every pair of neuron.

We can choose the number of unit.

Also in this layer we can define an activation function,
a function that “normalize” the output in order to get a values more uniformly possible.