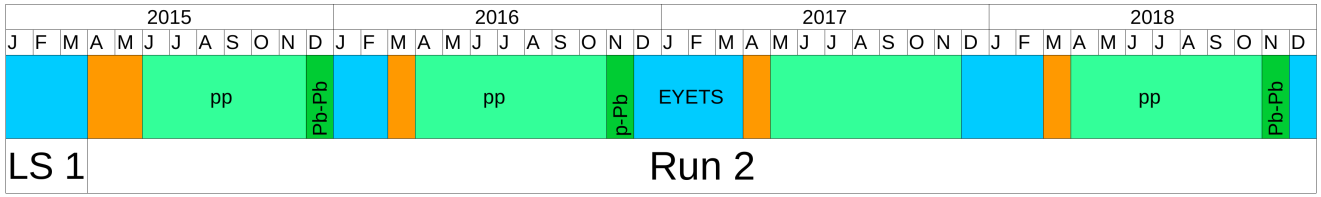
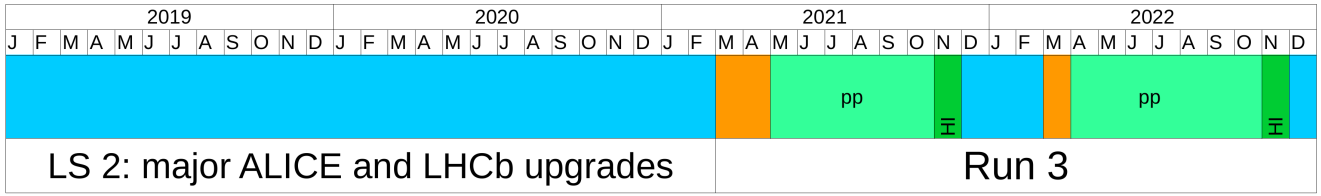
Exploring the Higgs boson with Machine Learning

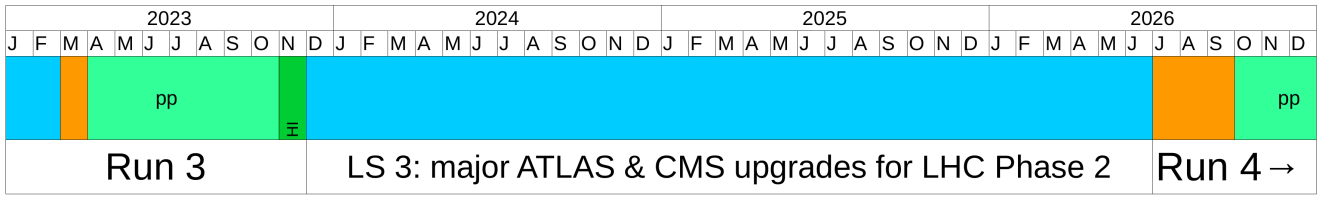
Felipe Ferreira de Freitas

Felipe F. Freitas, Charanjit K. Khosa, Verónica Sanz

Phys.Rev. D100 (2019) no.3, 035040 (2019-08-31) [arXiv:1902.05803]







- Higgs boson detected 👍👍.
- 750 GeV 😟 😟.
- No SUSY evidence 😟 😟.
- No DM evidence 😟 😟 😟.
- No new physics, yet.

Overview

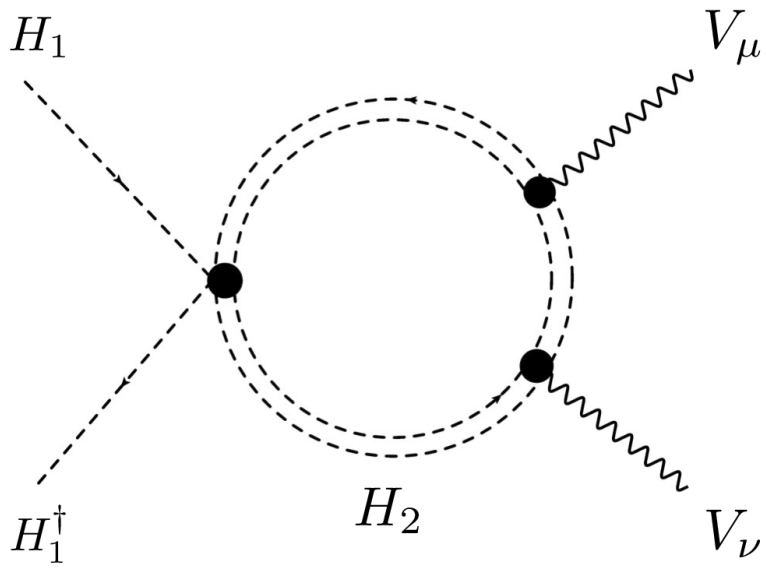
We are shifting our focus from the resonance searches into more subtle (indirect) effects of new physics. In a nutshell, the SMEFT is a consistent way of exploring new theories as deformations from the SM structures,with a large number of possible SM deviations taken into account.

As an example, in the SMEFT approach the Higgs couplings to vector bosons V=W,Z would be modified in the following way:

$$\eta_{\mu\nu}\, g m_V \Rightarrow \eta_{\mu\nu}\, g m_V - \frac{2\, g\, c_{HW}}{m_W}\, p_\mu^V\, p_\nu^V + \dots$$

which in terms of Lagrangian terms would be equivalent to adding to the SM Lagrangian new terms suppressed by a scale of new physics:

$$\mathcal{L}_{SM} \Rightarrow \mathcal{L}_{SM} + \frac{2igc_{HW}}{m_W^2}\left[D^\mu H^\dagger T_{2k} D^\nu H\right] W_{\mu\nu}^k + \dots$$



One could trace the ultimate origin of these deformations to many different types of new physics, just too heavy to be discovered directly at the LHC.

The deformation (aka Wilson coefficient) c_{HW} could be the manifestation of a new set of scalar particles, such as in 2HDMs, too heavy to be seen in direct production, but still felt via virtual effects

- As the LHC analyses, in the context of the SMEFT effects in the Higgs sector, moves from the κ formalism to the use of kinematic information, the information contained in multidimensional distributions becomes an important source to identifying even subtler effects.
- The need to quickly identify subtle effects in multidimensional distributions of information,clearly calls for artificial intelligence methods.
- The production of the Higgs in association with a massive vector boson, or VH, is already firmly established.

Current status: limits on the SMEFT and the VH channel at the LHC:

The Wilson coefficient c_{HW} it is currently constrained to values in the range (individual constraint):

- $c_{HW} = 0.002 \pm 0.014$

The limits on SMEFT operators were obtained by perfoming a global fit including kinematic information on VH and electroweak WW production at LEP2 and LHC but only 40 fb^{-1} of data, half of the total Run2 dataset.

J. Ellis, C. W. Murphy, V. Sanz and T. You, JHEP1806(2018) 146 , M. Aaboudet al.[ATLAS Collaboration], Phys. Lett. B786(2018) 59

The observation of the Higgs decaying into two b-quarks has been done by combining a challenging set of channels collectively denoted by VH, which corresponds to the Higgs produced in association with a massive vector boson V= Z or $W^{\pm}$. The final states are classified as 0L ($Z \to \nu\nu$), 1L $\left(W \to l\nu\right)$ and 2L $\left(Z \to l^{+}l^{-}\right)$.

- $\mu_{VH}^{\mathrm{ATLAS}} = 1.01 + 0.12(\mathrm{stat})^{+0.6}_{0.15}$
- naively indicate a two-sigma exclusion for $|c_{HW}| < 0.02$.

in our analysis:

- we select the value $c_{HW} = 0.03$ as a benchmark point.
- we consider $c_{HW} \neq 0$ as our $signal$ and $c_{HW} = 0$ as $backgorund$.
- gather the most information as possible from the final state particles.

We consider the following observables as data features:

- $p_T^{b_1}$ transverse momentum of the leading b-jet.
- $p_T^{b_2}$ transverse momentum of the sub leading b-jet.
- p_T^{VH} transverse momentum of the VH pair.
- M_T^{VH} transverse mass of the VH pair.
- $p_T^{W/Z}$ transverse momentum of gauge boson.
- p_T^H transverse momentum of the reconstructed Higgs boson.
- η^H pseudo-rapidity of the reconstructed Higgs boson.
- ϕ^H azimuthal angle of the reconstructed Higgs boson.

0L channel:

- p_T^l transverse momentum of lepton
- missing transverse energy
- $\Delta\phi_{b_1 l}$ azimuthal angular separation between leading b-jet and lepton
- $\Delta\phi_{l\mathrm{MET}}$ azimuthal angular separation between lepton and MET

1L channel:

- M_T^W transverse mass of the $W^{\pm}$
- ΔR_{wl} separation between lepton and W boson in the $\eta - \phi$ plane
- missing transverse energy
- $\Delta\phi_{b_1\mathrm{MET}}$ azimuthal angular separation between leading b-jet and MET
- $\Delta\phi_{l\mathrm{MET}}$ azimuthal angular separation between lepton and MET

2L channel:

- $p_T^{l_1}$ transverse momentum of the leading lepton
- $p_T^{l_2}$ transverse momentum of sub-leading lepton
- ΔR_{ll} separation between two lepton in the $\eta - \phi$ plane
- $\Delta\phi_{b_1 l_1}$ azimuthal angular separation between leading b-jet and leading lepton
- $\Delta\phi_{b_2 l_1}$ azimuthal angular separation between sub-leading b-jet and leading lepton

How the data looks like:

In [14]:
```
zh_chw_0d01_df.head(5)
```

Out[14]:

	ptb1	ptb2	misset	pth	ptz	etah	phih	mtvh	ptvh	dphib1met	signal
0	125.407039	86.117322	205.932526	209.328455	205.932526	1.277426	1.929536	449.712567	4.603307	3.037227	1
1	131.907616	70.616237	187.837401	190.132443	187.837401	-0.563147	-0.758883	415.367124	3.029103	2.895890	1
2	142.205381	29.009372	121.233232	120.412595	121.233232	-2.029023	-1.629733	294.761943	1.941833	3.006520	1
3	106.761004	27.168427	119.788307	120.780482	119.788307	-0.135757	-2.948970	293.614796	0.997513	2.935436	1
4	94.397035	73.704720	132.123257	134.300456	132.123257	2.286593	-1.262787	315.572440	3.875016	2.604910	1

- The data is not ready yet for the Machine Learning analysis, we have to normalize the entries to be more Machine Learning friendly.
- We can use the Scikit-learn library, specifically the MinMaxScaler() function to do this task.

In [16]:
```
from sklearn import preprocessing

min_max_scaler = preprocessing.MinMaxScaler()
def scaleColumns(df, cols_to_scale):
    for col in cols_to_scale:
        df[col] = pd.DataFrame(min_max_scaler.fit_transform(pd.DataFrame(df[co
l])),columns=[col])
    return df
```

Before normalize the data, we first need to create our features input (x) and the targets (y):

In [17]:
```
X_cHW_0d01 = pd.concat([zh_chw_0d01_df, zh_sm], ignore_index=True)
X_cHW_0d03 = pd.concat([zh_chw_0d03_df, zh_sm], ignore_index=True)
X_cHW_0d1 = pd.concat([zh_chw_0d1_df, zh_sm], ignore_index=True)
X_cHW_1 = pd.concat([zh_chw_1_df, zh_sm], ignore_index=True)

y_cHW_0d01 = X_cHW_0d01['signal']
y_cHW_0d03 = X_cHW_0d03['signal']
y_cHW_0d1 = X_cHW_0d1['signal']
y_cHW_1 = X_cHW_1['signal']
```

- Only normalize your features after combine everything you are going to use. ☺

In [19]:
```
for i in [X_cHW_0d01, X_cHW_0d03, X_cHW_0d1, X_cHW_1]:
    i = scaleColumns(i,['ptb1', 'ptb2', 'misset',
                        'pth', 'ptz', 'etah',
                        'phih', 'mtvh', 'ptvh',
                        'dphib1met'])
```

After the normalization, the imputs looks like this:

In [20]:
```
X_cHW_1.head(5)
```

Out[20]:

	ptb1	ptb2	misset	pth	ptz	etah	phih	mtvh	ptvh	dphib1met	signal
0	0.086453	0.015394	0.069946	0.070658	0.069946	0.317102	0.782922	0.065425	0.520621	0.887508	1
1	0.040042	0.079937	0.060259	0.060270	0.060259	0.712868	0.544151	0.055591	0.386925	0.943983	1
2	0.188917	0.163342	0.224048	0.223358	0.224048	0.428109	0.001863	0.218536	0.270565	0.975580	1
3	0.012409	0.012235	0.008392	0.008747	0.008392	0.288645	0.147635	0.006867	0.141265	0.911109	1
4	0.037988	0.037252	0.036305	0.036593	0.036305	0.694466	0.051104	0.032546	0.248878	0.735323	1

In [35]:
```
for i in X_cHW_1.keys():
    print(i,':',X_cHW_1[i].mean(),X_cHW_1[i].std())
```

```
ptb1 : 0.06603380945984406 0.07807105493537096
ptb2 : 0.05276380572195325 0.06926308503087236
misset : 0.06716309934162445 0.08949834993587825
pth : 0.06718128565788557 0.08941226283848779
ptz : 0.06716309934314747 0.08949834993804219
etah : 0.5006507384165415 0.17312476022576348
phih : 0.5000665818066831 0.28900209027092005
mtvh : 0.06388180888051978 0.08844030270164628
ptvh : 0.28225733120272967 0.14680672893989266
dphib1met : 0.7684152266915593 0.19672374641410206
signal : 0.5 0.5000012500046875
```

Don't forget to drop the column $signal$ from your features input, otherwise the Machine will imeadiatly know what is $signal$ and what is BG ☺

In [36]:
```
for i in [X_cHW_0d01, X_cHW_0d03, X_cHW_0d1, X_cHW_1]:
    i.drop(['signal'], axis=1, inplace=True)
```

In [37]:
```
X_cHW_0d03.shape, y_cHW_0d03.shape
```

Out[37]:
```
((200000, 10), (200000,))
```

Now we can shuffle our inputs and split into training and test datasets:

- To avoid pain we can use Scikit-learn function train_test_split(), which automatically take cares of all we need.

In [38]:
```
X_train_cHW_0d01, X_test_cHW_0d01, y_train_cHW_0d01, y_test_cHW_0d01 = \
train_test_split(X_cHW_0d01, y_cHW_0d01, train_size=0.8, test_size=0.2)

X_train_cHW_0d03, X_test_cHW_0d03, y_train_cHW_0d03, y_test_cHW_0d03 = \
train_test_split(X_cHW_0d03, y_cHW_0d03, train_size=0.8, test_size=0.2)

X_train_cHW_0d1, X_test_cHW_0d1, y_train_cHW_0d1, y_test_cHW_0d1 = \
train_test_split(X_cHW_0d1, y_cHW_0d1, train_size=0.8, test_size=0.2)

X_train_cHW_1, X_test_cHW_1, y_train_cHW_1, y_test_cHW_1 = \
train_test_split(X_cHW_1, y_cHW_1, train_size=0.8, test_size=0.2)
```

- We can save our training and test datasets, so in the future we don't need to do everything again:

In [39]:
```
np.savez('../data/inclusive/X_train_0L_cHW_0d03.npz', x=X_train_cHW_0d03, y=y_t
rain_cHW_0d03)
np.savez('../data/inclusive/X_train_0L_cHW_0d1.npz', x=X_train_cHW_0d1, y=y_tra
in_cHW_0d1)
np.savez('../data/inclusive/X_train_0L_cHW_1.npz', x=X_train_cHW_1, y=y_train_c
HW_1)

np.savez('../data/inclusive/X_test_0L_cHW_0d03.npz', x=X_test_cHW_0d03, y=y_tes
t_cHW_0d03)
np.savez('../data/inclusive/X_test_0L_cHW_0d1.npz', x=X_test_cHW_0d1, y=y_test_
cHW_0d1)
np.savez('../data/inclusive/X_test_0L_cHW_1.npz', x=X_test_cHW_1, y=y_test_cHW_
1)
```

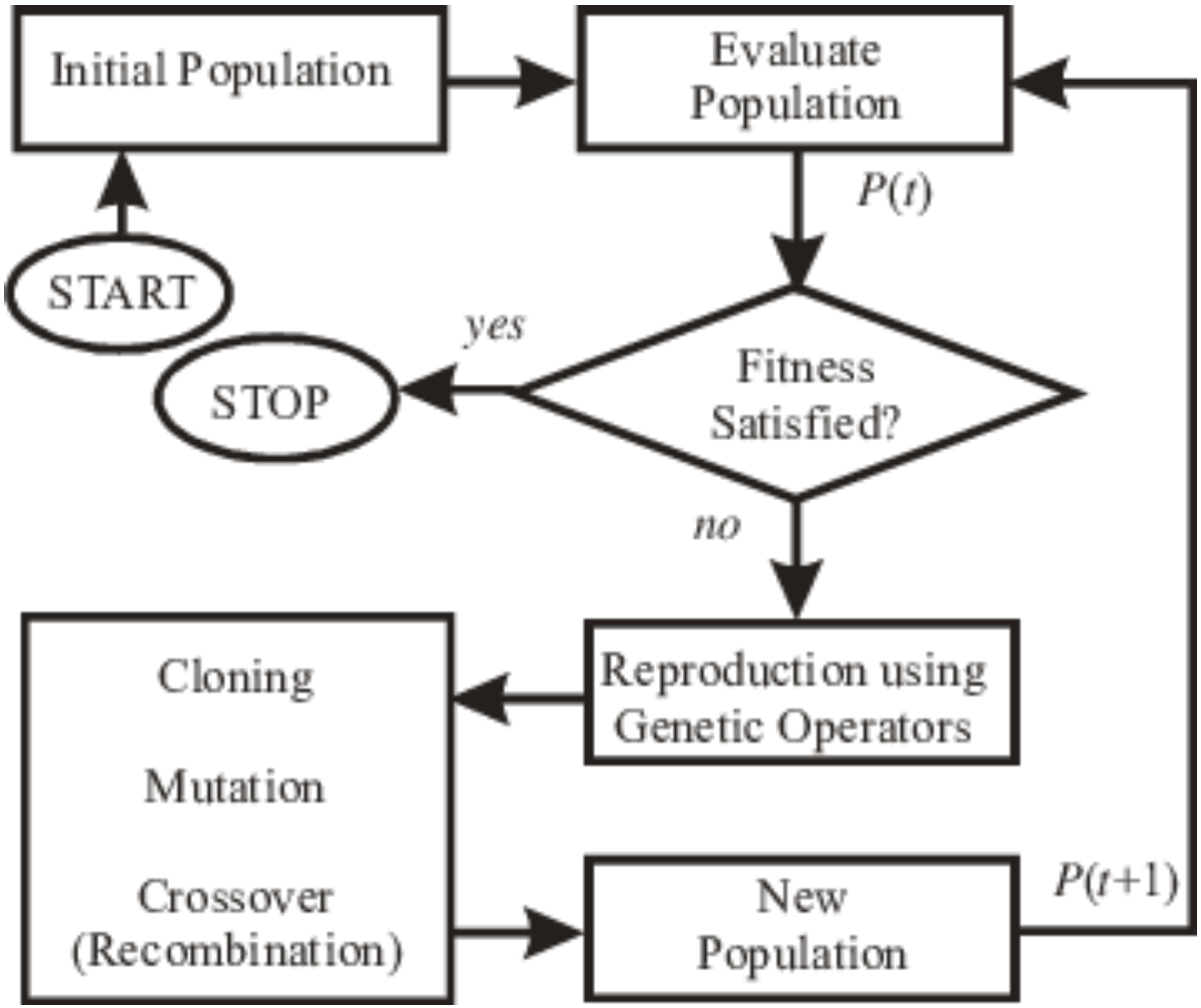
NN architectures: Deep or shalow 🤔

To extract the maximum amount of information from the kinematic features, one needs to combine multidimensional information. The objective is to maximise our ability to detect new phenomena, which in HEP means maximising the significance of an observation.

- Number of layers.
- Activation function of each layer.
- Regularizers: L1, L2, Batch normalization, drop out.
- Drop out probability.
- ...

To set the best combination of hyper-parameter I built a Evolutionary Algorithm to search through all combinations of parameters and select only the best ones.

Evolutionary search algorithm:



in order to do so we are going to need some very important pieces, the first one is a class to produce random archtectures with the parameters we want to inspect:

In [40]:
```
class Network():
    """Represent a network and let us operate on it.
    Currently only works for an MLP.
    """

    def __init__(self, nn_param_choices=None):
        """Initialize our network.
        Args:
            nn_param_choices (dict): Parameters for the network, includes:
                nb_neurons (list): [64, 128, 256]
                nb_layers (list): [1, 2, 3, 4]
                activation (list): ['relu', 'elu']
                optimizer (list): ['rmsprop', 'adam']
        """
        self.accuracy = 0.
        self.nn_param_choices = nn_param_choices
        self.network = {}  # (dic): represents MLP network parameters
```

In [41]:
```
    def create_random(self):
        """Create a random network."""
        for key in self.nn_param_choices:
            self.network[key] = random.choice(self.nn_param_choices[key])

    def create_set(self, network):
        """Set network properties.
        Args:
            network (dict): The network parameters
        """
        self.network = network
```

In [42]:
```
    def train(self, dataset):
        """Train the network and record the accuracy.
        Args:
            dataset (str): Name of dataset to use.
        """
        if self.accuracy == 0.:
            self.accuracy = train_and_score(self.network, dataset)

    def print_network(self):
        """Print out a network."""
        logging.info(self.network)
        logging.info("Network accuracy: %.2f%%" % (self.accuracy * 100))
```

We need a way to evaluate our population of NN, to do so we build an optimizer class:

In [43]:

```
class Optimizer():
    """Class that implements genetic algorithm for MLP optimization."""

    def __init__(self, nn_param_choices, retain=0.4,
                 random_select=0.1, mutate_chance=0.2):
        """Create an optimizer.
        Args:
            nn_param_choices (dict): Possible network paremters
            retain (float): Percentage of population to retain after
                each generation
            random_select (float): Probability of a rejected network
                remaining in the population
            mutate_chance (float): Probability a network will be
                randomly mutated
        """
        self.mutate_chance = mutate_chance
        self.random_select = random_select
        self.retain = retain
        self.nn_param_choices = nn_param_choices
```

A function to generate the population:

In [44]:

```
    def create_population(self, count):
        """Create a population of random networks.
        Args:
            count (int): Number of networks to generate, aka the
                size of the population
        Returns:
            (list): Population of network objects
        """
        pop = []
        for _ in range(0, count):
            # Create a random network.
            network = Network(self.nn_param_choices)
            network.create_random()

            # Add the network to our population.
            pop.append(network)

        return pop
```

fitness function (a.k.a "motivation"):

In [46]:

```
    @staticmethod
    def fitness(network):
        """Return the accuracy, which is our fitness function."""
        return network.accuracy

    def grade(self, pop):
        """Find average fitness for a population.
        Args:
            pop (list): The population of networks
        Returns:
            (float): The average accuracy of the population
        """
        summed = reduce(add, (self.fitness(network) for network in pop))
        return summed / float((len(pop)))
```

Produce new generation of NN 😏

In [47]:

```
    def breed(self, mother, father):
        """Make two children as parts of their parents.
        Args:
            mother (dict): Network parameters
            father (dict): Network parameters
        Returns:
            (list): Two network objects
        """
        children = []
        for _ in range(2):

            child = {}

            # Loop through the parameters and pick params for the kid.
            for param in self.nn_param_choices:
                child[param] = random.choice(
                    [mother.network[param], father.network[param]]
                )

            # Now create a network object.
            network = Network(self.nn_param_choices)
            network.create_set(child)

            # Randomly mutate some of the children.
            if self.mutate_chance > random.random():
                network = self.mutate(network)

            children.append(network)

        return children
```

introducing random mutation

In [ ]:

```
    def mutate(self, network):
        """Randomly mutate one part of the network.
        Args:
            network (dict): The network parameters to mutate
        Returns:
            (Network): A randomly mutated network object
        """
        # Choose a random key.
        mutation = random.choice(list(self.nn_param_choices.keys()))

        # Mutate one of the params.
        network.network[mutation] = random.choice(self.nn_param_choices[mutatio
n])

        return network
```

and evolve our population 🧬

In [48]:

```
    def evolve(self, pop):
        """Evolve a population of networks.
        Args:
            pop (list): A list of network parameters
        Returns:
            (list): The evolved population of networks
        """
        # Get scores for each network.
        graded = [(self.fitness(network), network) for network in pop]

        # Sort on the scores.
        graded = [x[1] for x in sorted(graded, key=lambda x: x[0], reverse=Tru
e)]

        # Get the number we want to keep for the next gen.
        retain_length = int(len(graded)*self.retain)

        # The parents are every network we want to keep.
        parents = graded[:retain_length]

        # For those we aren't keeping, randomly keep some anyway.
        for individual in graded[retain_length:]:
            if self.random_select > random.random():
                parents.append(individual)
```

In [ ]:

```
        # Now find out how many spots we have left to fill.
        parents_length = len(parents)
        desired_length = len(pop) - parents_length
        children = []

        # Add children, which are bred from two remaining networks.
        while len(children) < desired_length:

            # Get a random mom and dad.
            male = random.randint(0, parents_length-1)
            female = random.randint(0, parents_length-1)

            # Assuming they aren't the same network...
            if male != female:
                male = parents[male]
                female = parents[female]

                # Breed them.
                babies = self.breed(male, female)

                # Add the children one at a time.
                for baby in babies:
                    # Don't grow larger than desired length.
                    if len(children) < desired_length:
                        children.append(baby)

        parents.extend(children)

        return parents
```

We can use the Asimov significance as loss function and minimize $(1/Z_A)$

The Asimov significance is defined as follow:

$$Z_A = \left[2\left((s+b)\ln\left[\frac{(s+b)(b+\sigma_b^2)}{b^2+(s+b)\sigma_b^2}\right]\frac{b^2}{\sigma_b^2}\ln\left[1+\frac{\sigma_b^2 s}{b(b+\sigma_b^2)}\right]\right)\right]^{1/2}.$$

- $s =$ number of signal events
- $b =$ number of background events
- $\sigma_b^2 =$ systematic uncertanties

Adam Elwood, Dirk Krücker, Direct optimisation of the discovery significance when training neural networks to search for new physics in particle colliders, DESY-18-082 ,

In [ ]:

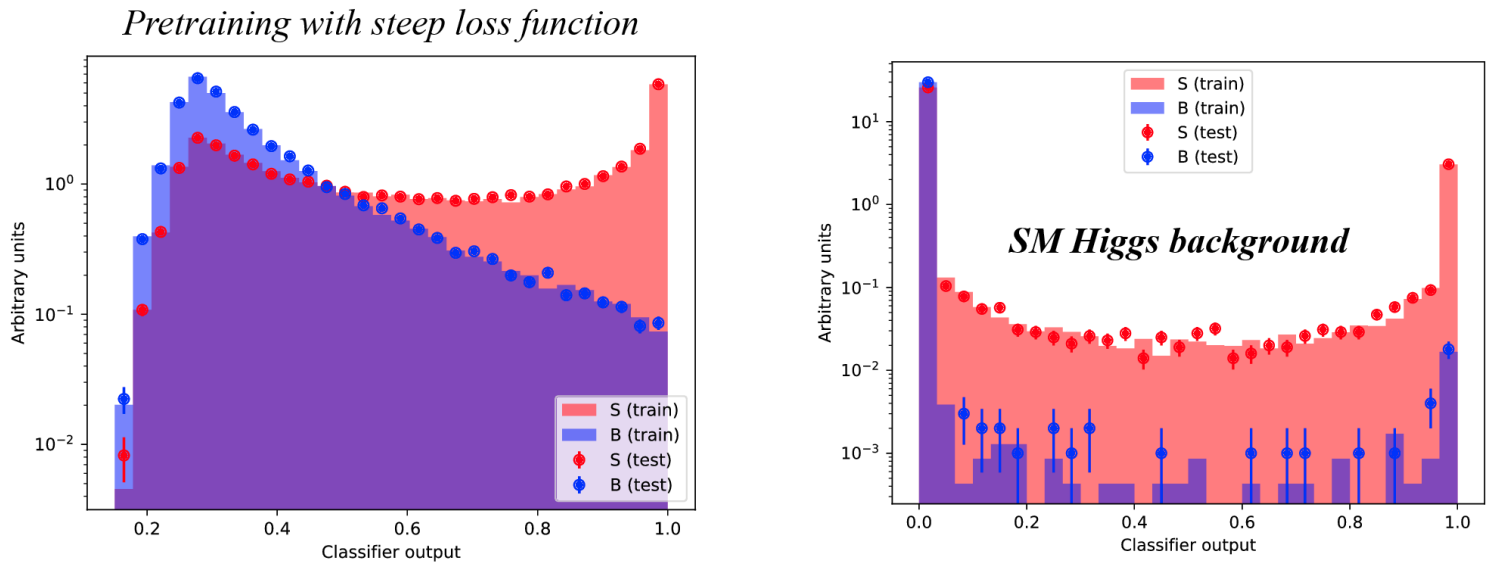
```
def asimovSignificanceLossInvert(expectedSignal,expectedBkgd,systematic):
    '''Define a loss function that calculates the significance based on fixed
    expected signal and expected background yields for a given batch size'''


    def asimovSigLossInvert(y_true,y_pred):
        #Continuous version:

        signalWeight=expectedSignal/K.sum(y_true)
        bkgdWeight=expectedBkgd/K.sum(1-y_true)

        s = signalWeight*K.sum(y_pred*y_true)
        b = bkgdWeight*K.sum(y_pred*(1-y_true))
        sigB=systematic*b

        return 1./(2*((s+b)*K.log((s+b)*(b+sigB*sigB)/(b*b+(s+b)*sigB*sigB+K.ep
silon())+K.epsilon())-b*b*K.log(1+sigB*sigB*s/(b*(b+sigB*sigB)+K.epsilon()))/(s
igB*sigB+K.epsilon()))) #Add the epsilon to avoid dividing by 0

    return asimovSigLossInvert
```

We found that:

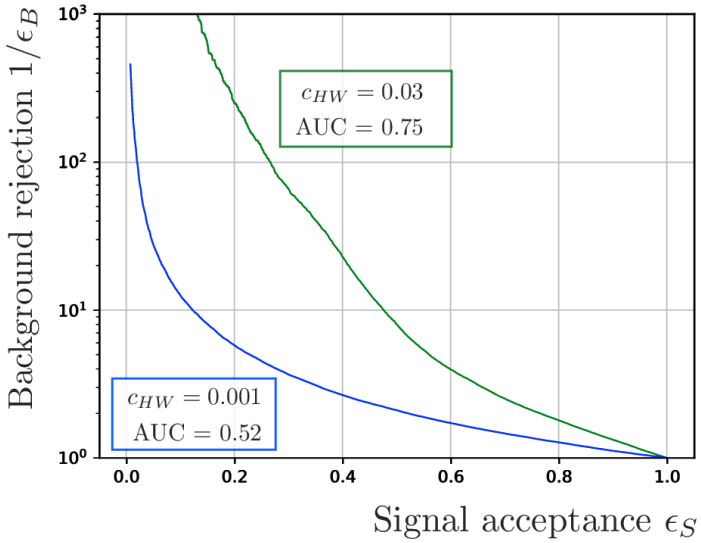
- 1 hidden layer with the same number of neurons as the number of inputs.
- ReLu as activation function.
- Adam as optimizer.
- $L1 = 0.003$
- Batch size of 4096.

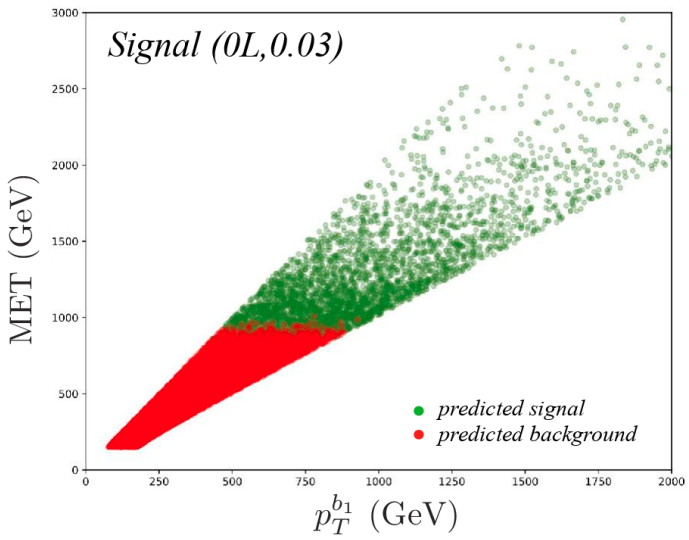
Important notes:

- The Asimov loss takes a little more effort to minimize, so we set a pre-training set of runs for 5 epochs using a steeper loss function.
- A longer run, with about 20 ~ 30 epochs is done after the pre-training.



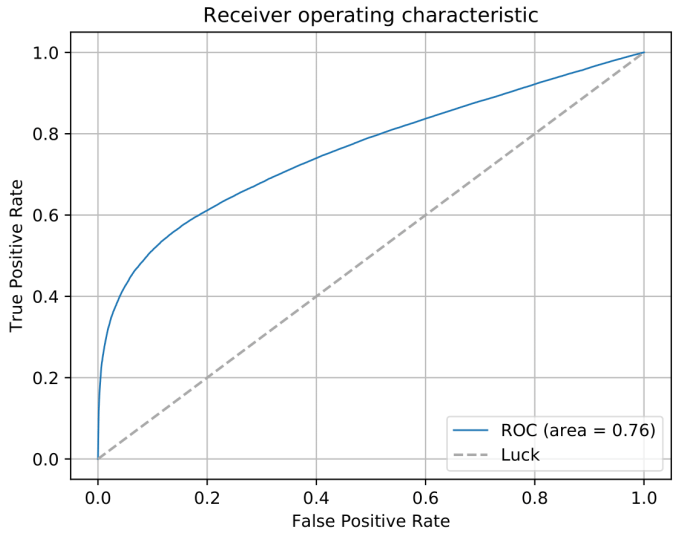
- Distribution of 0L signal (red) and background (blue) events as a function of the classifier output. The left plot is the outcome of performing an initial pretraining run with 5 epochs.

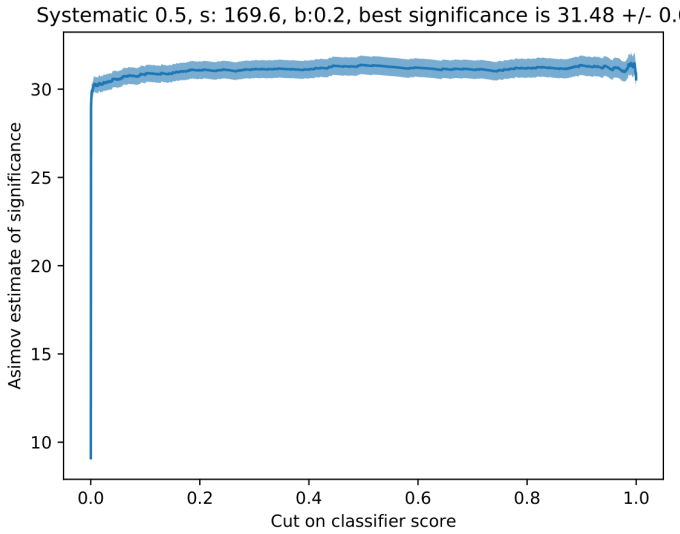




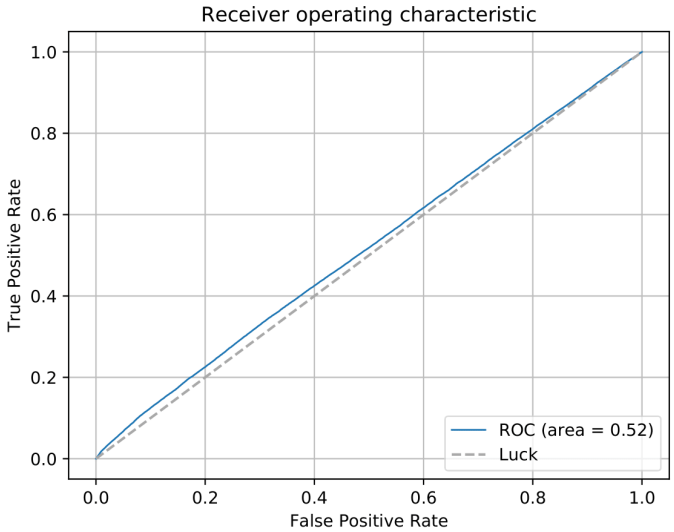
- Left: ROC curve for two values of the SMEFT coefficient in the 0L channel.
- Right: Classification of true signal events for $c_{HW} = 0.03$ and their mapping to kinematic features.

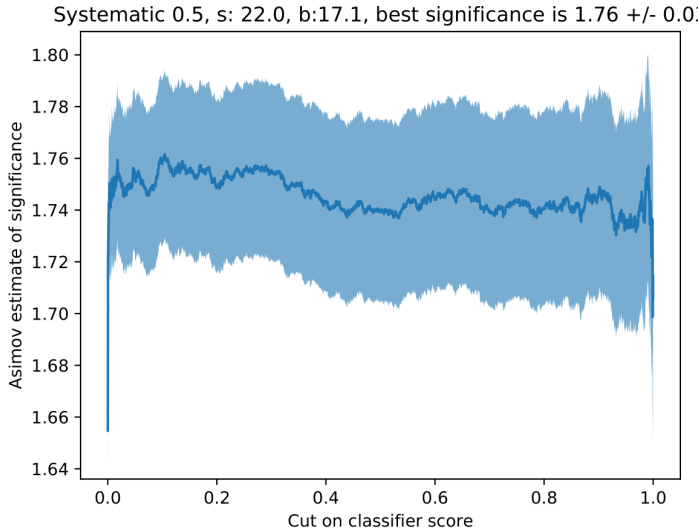
A more interesting way to evaluate our model





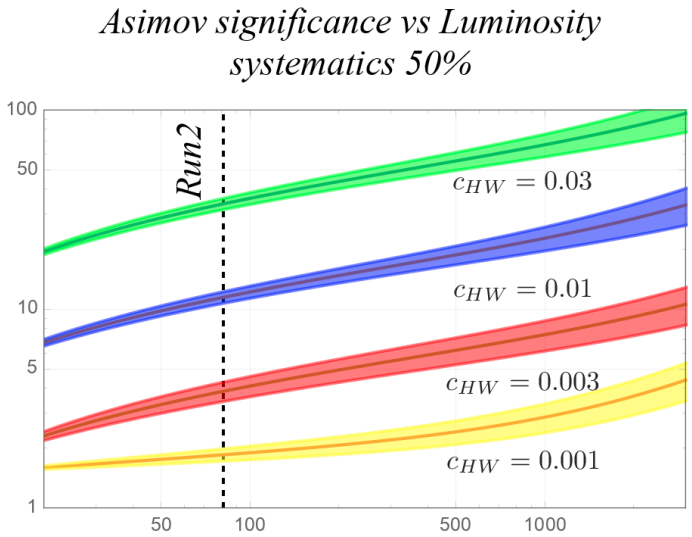
- $c_{HW} = 0.03$
- luminosity: 80 fb^{-1}
- AUC = 75%
- Asimov significance = 31.5, with 50% systematic error.

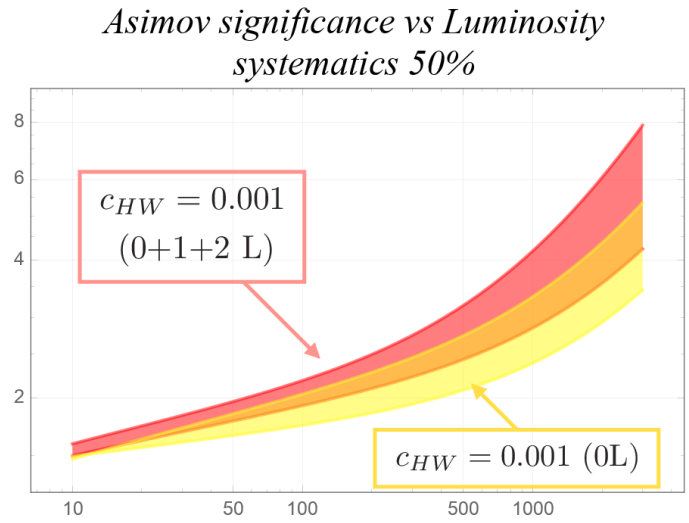




- $c_{HW} = 0.001$
- luminosity: 80 fb^{-1}
- AUC = 52%
- Asimov significance = 1.76, with 50% systematic error.

Project the significance to future luminosities:





- Left: Luminosity (fb^{-1}) versus Asimov significance for different values of c_{HW} in the 0-lepton channel and 50 % systematic uncertainty.
- Right: The effect of combination of VH channels for the limiting value $c_{HW} = 0.001$ and 50% systematic errors.

Summarize

- Create and deploy a state of art evolutionary algorithm to find the best hyper-parameters in a NN.
- Within the framework of our analysis, we found the 0L channel to be dominant.
- We obtained a limit in the SMEFT coefficient c_{HW} of 0.001, about 30 times better than the current constraint from global analysis with the Run2 data

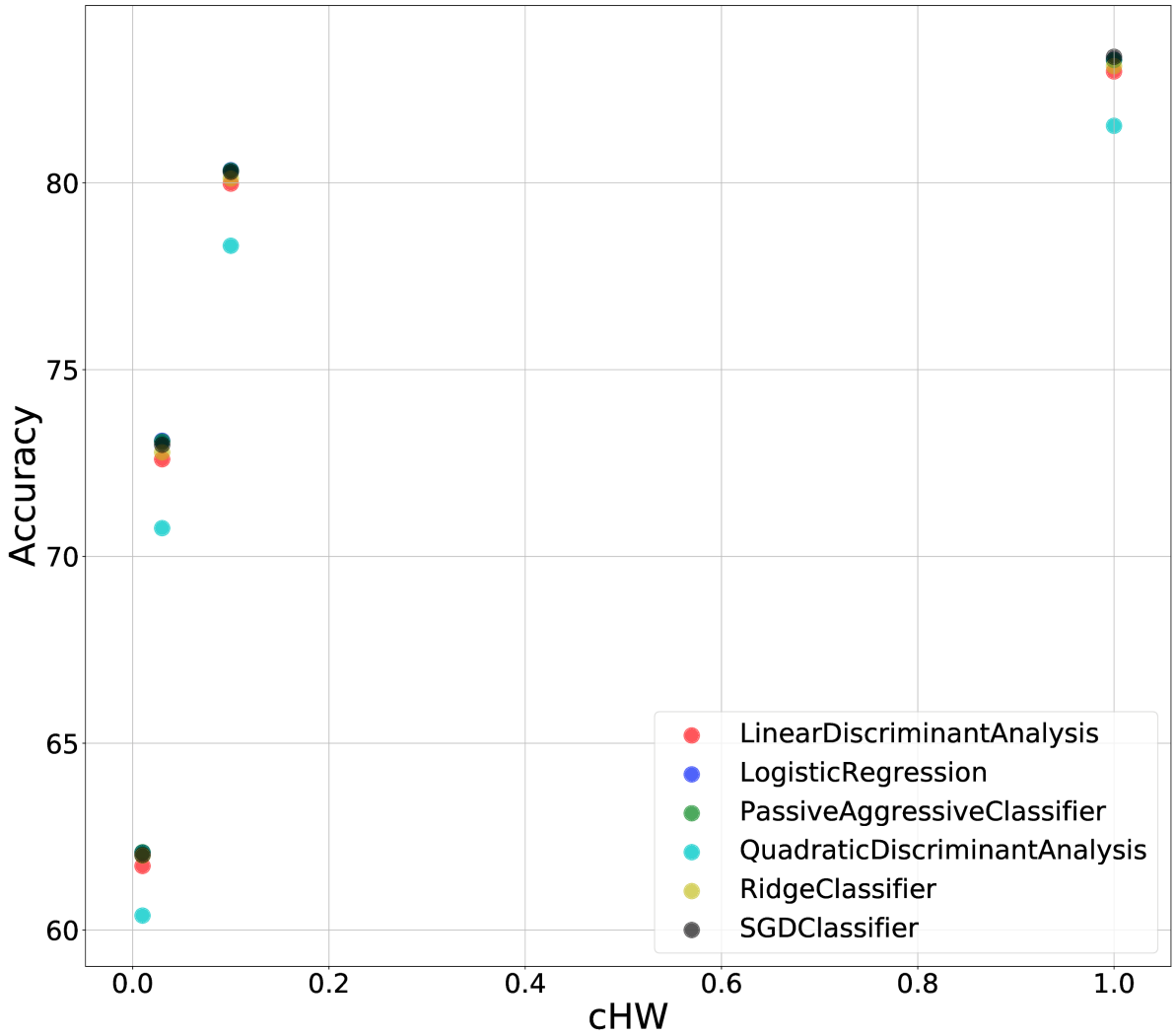
Our analysis could be improved in a number of ways:

- A more realistic simulation could be performed, including NLO SMEFT effects.
- Although we found that deep layers led to overfitting, and a shallow NN was more suitable, new algorithms could be explored to increase sensitivity.
- we should understand the effect of switching on more than one wilson coefficient.

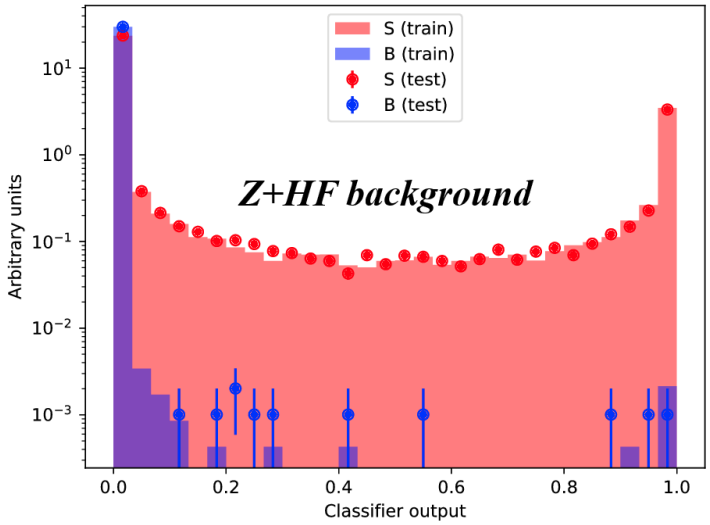
github.com/FFFreitas/Exploring-SMEFT-in-VH-with-Machine-Learning

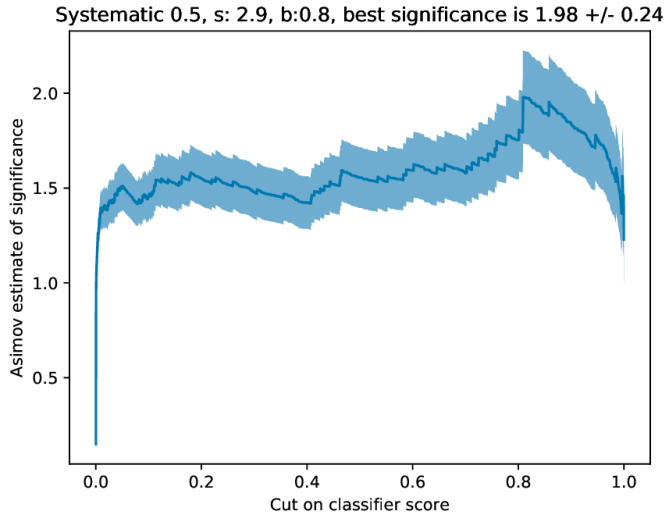
Obrigado (Thanks)

Backup:



- Classification performance for different linear models tested with ES.





- $c_{HW} = 0.001$
- luminosity: 80 fb^{-1}
- Vector boson + heavy flavour object as $background$.

In [ ]: