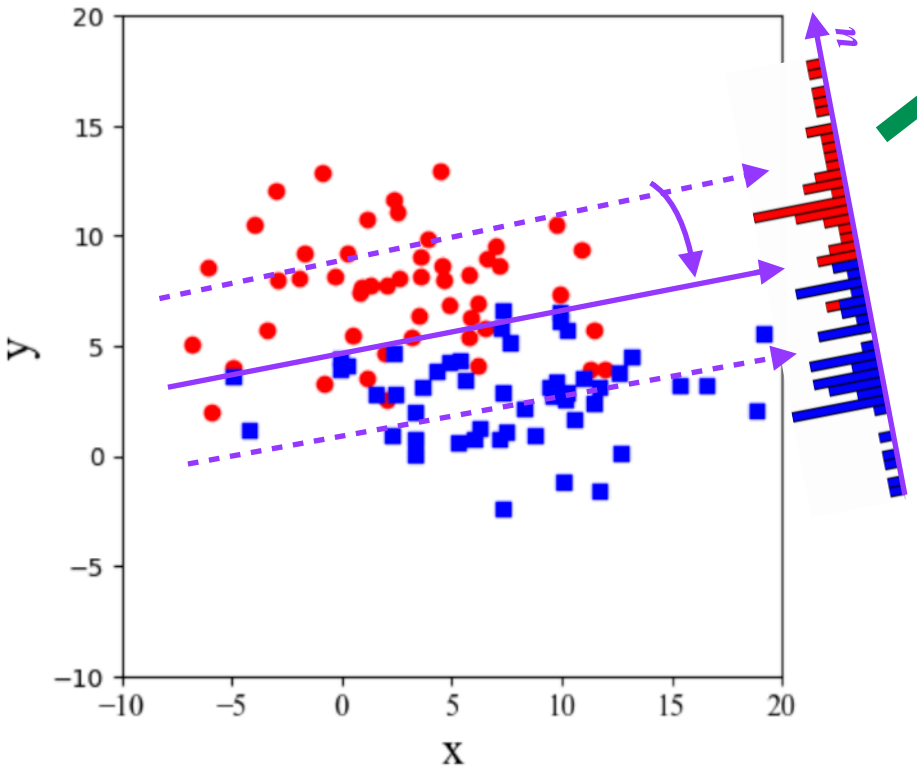
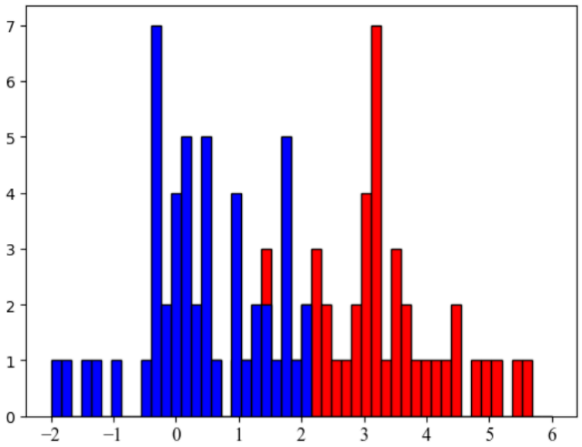
Lecture 14:

Artificial Neural Networks

- Fisher Discriminants & Perceptrons
- Universal Approximation
- Feed Forward, Propagate Backwards!
- 3 Useful Tools
- Playtime with PyTorch
- Stability & Quantifying Uncertainty

Recall the use of a Fisher discriminant to separate classes…





choose

$$w_x = \frac{\overline{x_1} - \overline{x_2}}{s_{x_1}^2 + s_{x_2}^2} \qquad w_y = \frac{\overline{y_1} - \overline{y_2}}{s_{y_1}^2 + s_{y_2}^2}$$

More Generally: $u = \vec{w}^T \vec{p}$

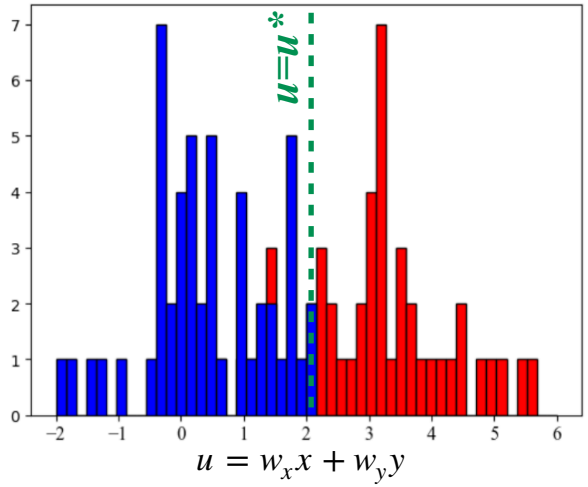
transpose vector of weights

vector of parameters

$$\vec{w} = (\Sigma_1 + \Sigma_2)^{-1}(\vec{\mu_1} - \vec{\mu_2})$$

class covariance matrices

vectors of class means



As with the Fisher Discriminant, the separation parameters are optimised ("trained") using a known data set (such as from simulation) and then applied elsewhere.

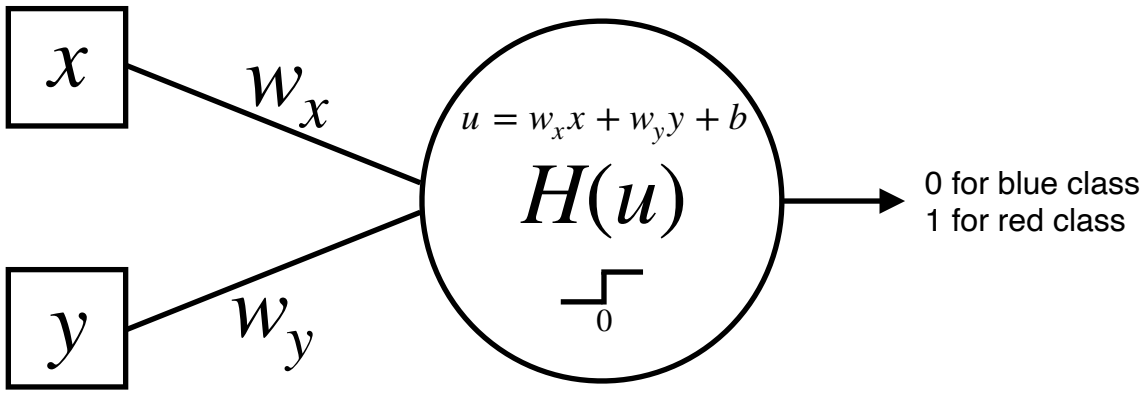
Another approach to finding w_x and w_y is to first take initial guesses for these, and then iterate their values (for example, using gradient decent) to find the best solution that allows a threshold cut value of $u > u^*$ that optimises the entropy or Gini index for separation etc.

Alternatively, defining $b = -u^*$, we want to perform a regression to optimise the parameters in the expression

$$u = w_x x + w_y y + b$$

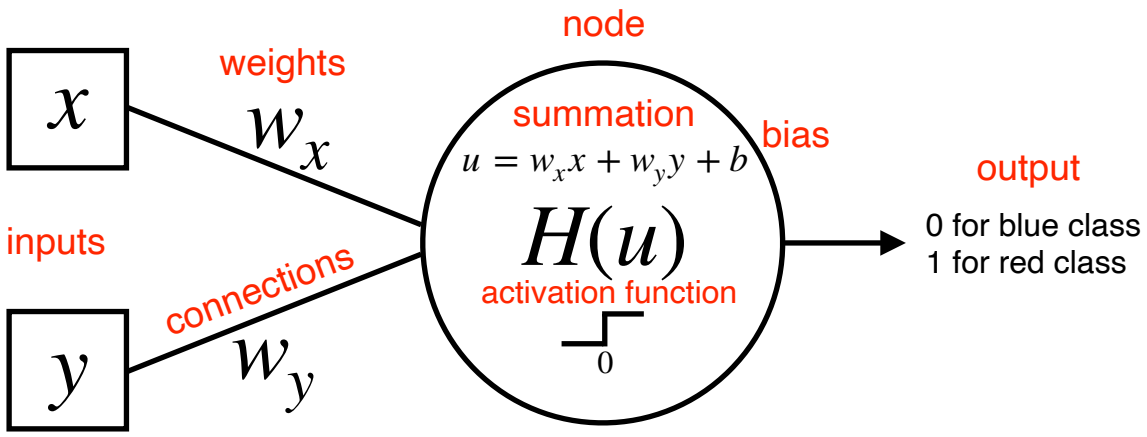
such that the cut u > 0 optimises the separation

This can all be diagrammatically represented as follows:



with parameters w_x , w_y and b to be optimised.

Perceptron

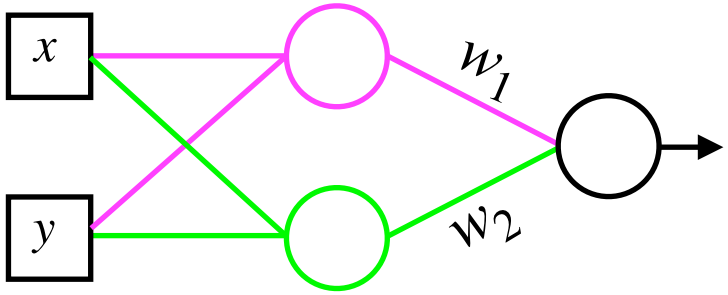


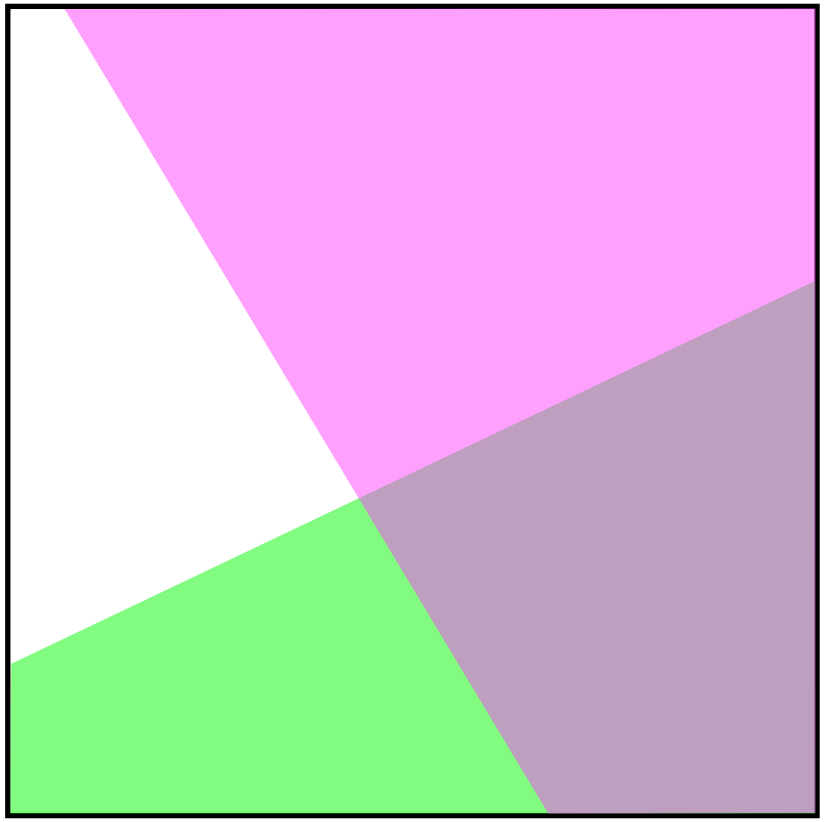
Can be used for simple **linear** discrimination
(independent of the form of the activation function)

Similar constructions with different activation functions can also be used to provide a continuous output instead of a simple binary classification.

More complex functionality can then be achieved by joining units together into networks…

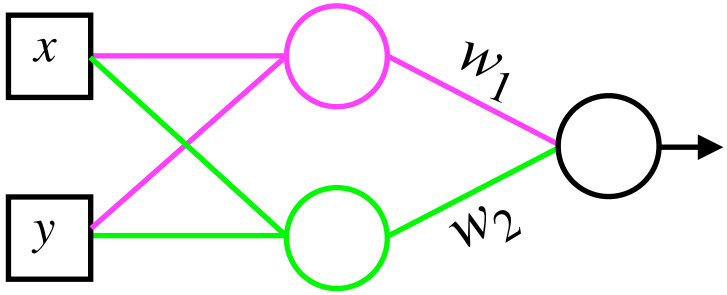
For example:





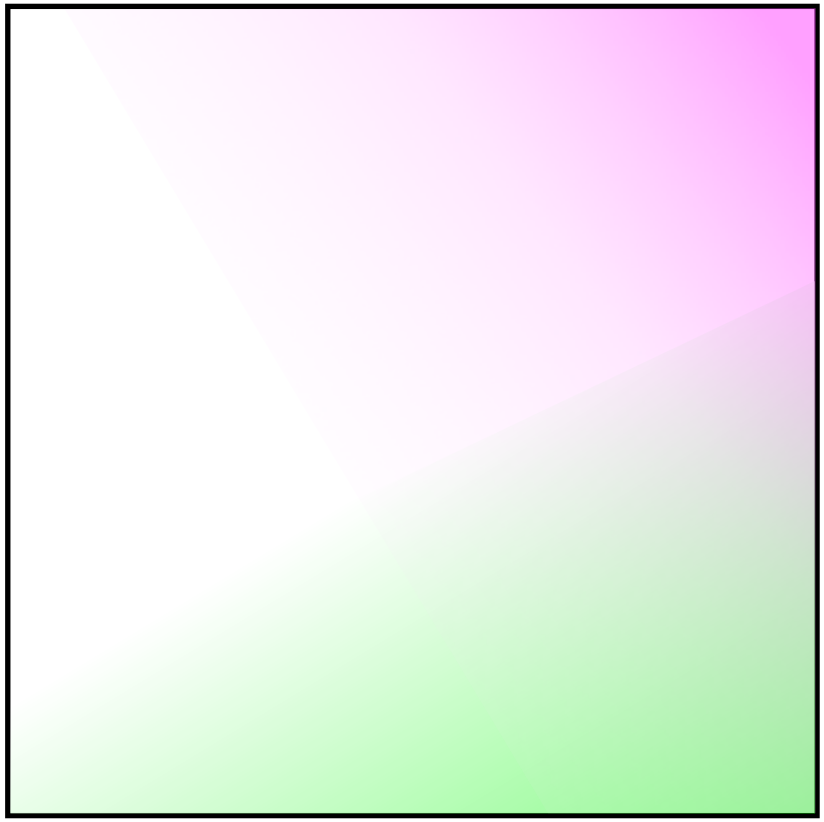
Can now select or exclude combinations of these 4 regions depending on how the values w_1, w_2 and final threshold function are chosen

For example:



Can now select or exclude combinations of these 4 regions depending on how the values w_1, w_2 and final threshold function are chosen

A more continuous activation function shades these more subtly, but keeps the same basic structure



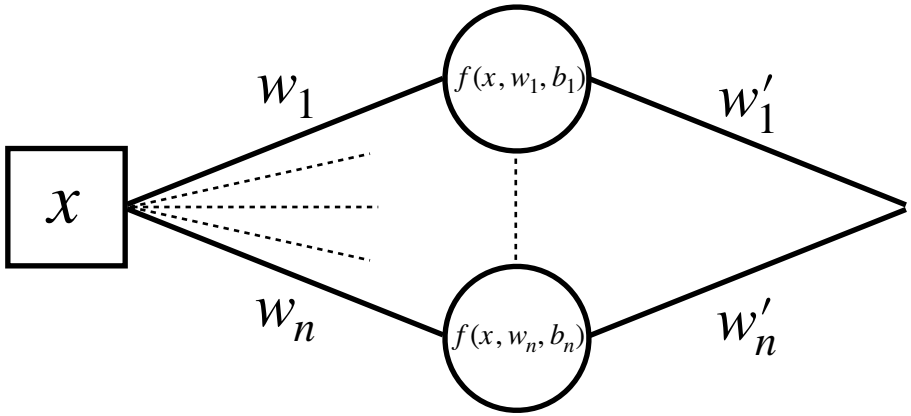
Universal Approximator

A central feature at the heart of ANNs is a construction that has the ability to approximate any arbitrary function, permitting a mapping:

$$(x_1, x_2 \ldots x_n) \longrightarrow (y_1, y_2 \ldots y_n)$$

arbitrary

The function is approximated in a piece-wise manner through the nodes by adjusting a finite set of weights and biases:



$$y(x) = \sum_{i=1}^{n} w_i' f(x, w_i, b_i)$$

So the output is simply a linear combination of activation function evaluations. This is where the main flexibility of the construction comes from, and the main purpose of the activation function is to insure independent equations for different values of x so as to permit weights to be tuned to yield arbitrarily different y values.

Practically speaking, this means that the activation function must not be capable of being couched as a linear expression in terms of some function of x alone. In other words:

$$f(x, w, b) \neq F(x)G(w, b) + g(w, b) + c$$

Otherwise, we could re-write the output as:

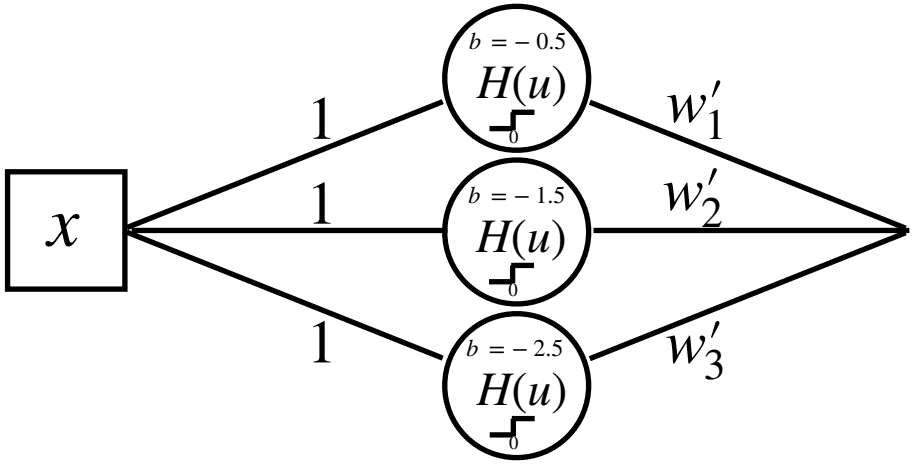
$$y(x) = F(x) \sum w_i' \, G(w_i, b_i) + \sum w_i' \, [g(w_i, b_i) + c]$$

The behaviour as a function of x would then no longer be determined by individual weights and biases, but simply by the collective sums in the above relationship. Degeneracies in these sums would then lead to equations for x and y that are not independent.

This not only means f must be non-linear but, for example, it also cannot be a power law, unless b is non-zero; nor can it be an exponential with non-zero b, unless weights are allowed to change.

So long as the activation function obeys the above inequality, it can have just about any form. However, some forms may be better at providing a smooth interpolation between nodal constraints.

A Simplified Toy Example

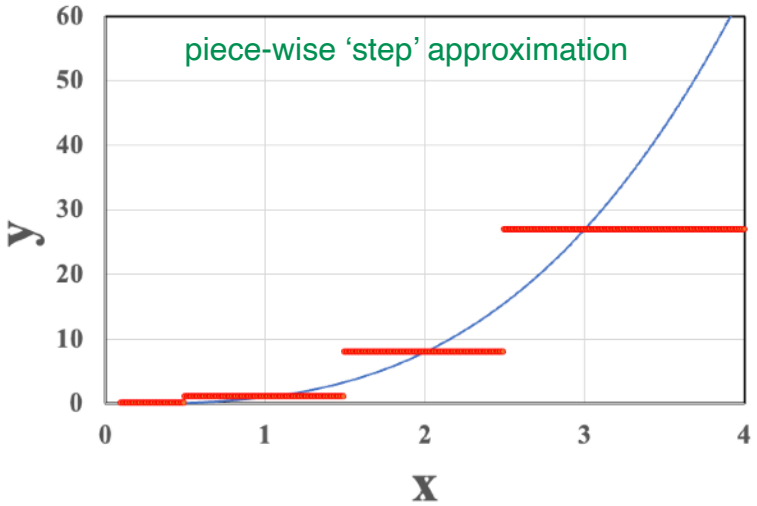


"train" on 3 data points

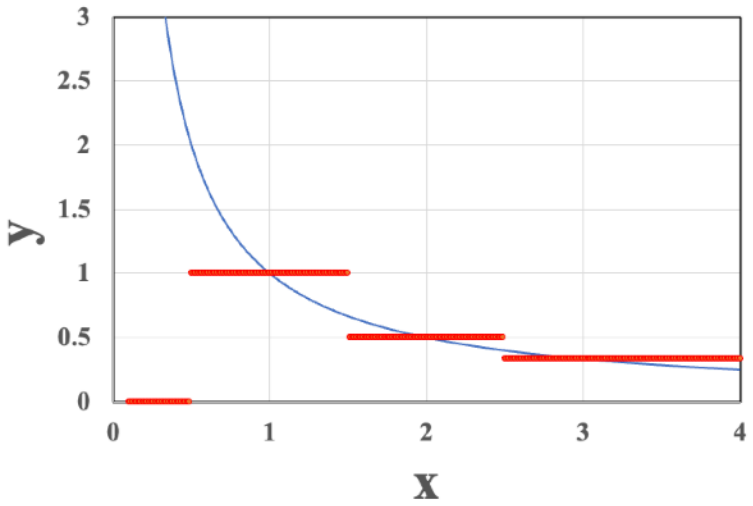
$$y(1) = w_1'$$
$$y(2) = w_1' + w_2'$$
$$y(3) = w_1' + w_2' + w_3'$$

These equations are all independent and we can trivially make these three y values anything we like by choosing appropriate weights

For example, we can try to approximate a cubic function by choosing $w_1' = 1,\ w_2' = 7,\ w_3' = 19$

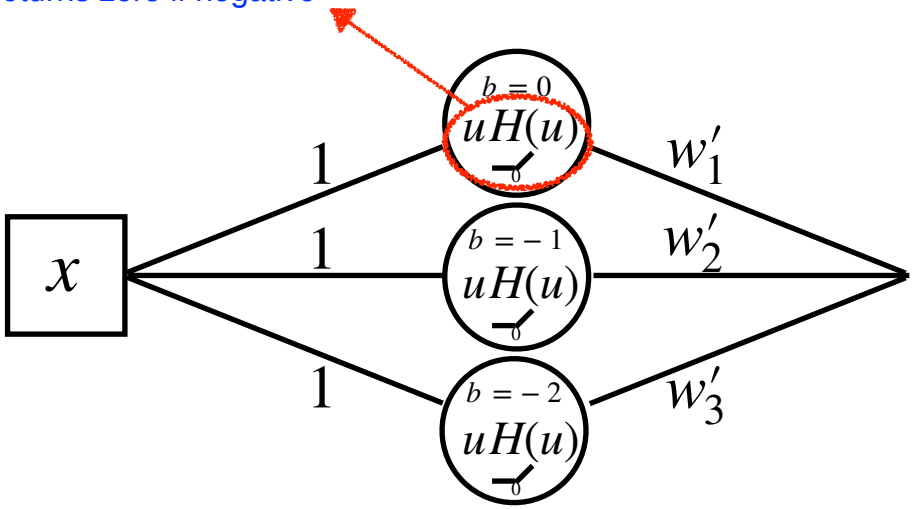


Or we can try to approximate the function 1/x by choosing $w_1' = 1,\ w_2' = -0.5,\ w_3' = -1/6$



A Simplified Toy Example

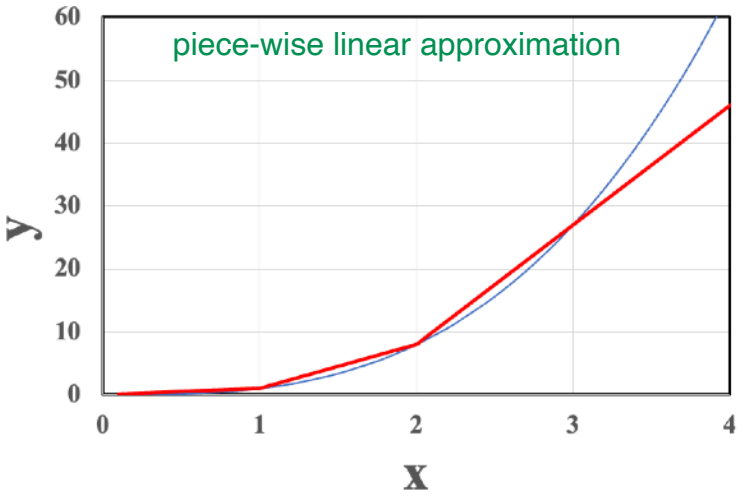
Rectified Linear Unit
returns u value if positive,
returns zero if negative



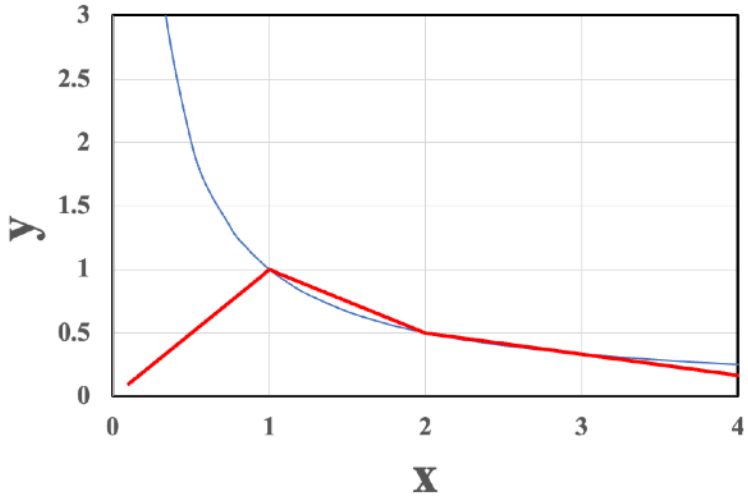
$$y(1) = w_1'$$
$$y(2) = 2w_1' + w_2'$$
$$y(3) = 3w_1' + 2w_2' + w_3'$$

These equations are all independent and we can trivially make these three y values anything we like by choosing appropriate weights

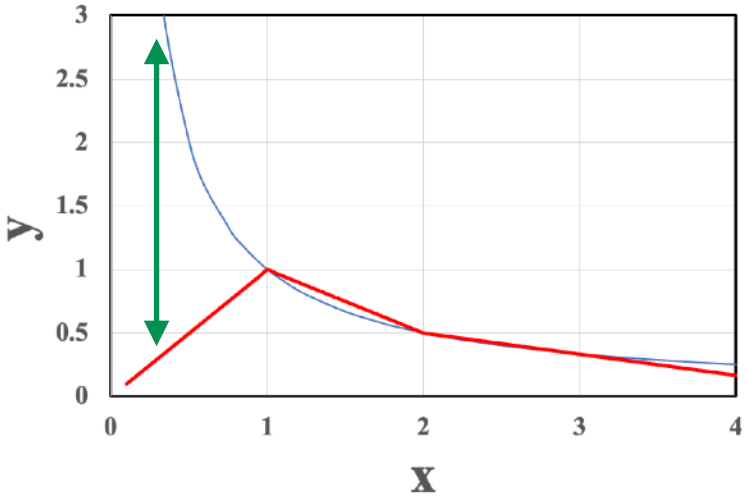
For example, we can try to approximate a cubic function by choosing $w_1' = 1,\ w_2' = 6,\ w_3' = 12$



Or we can try to approximate the function 1/x by choosing $w_1' = 1,\ w_2' = -1.5,\ w_3' = 1/3$



A reminder that this is not deducing an underlying model!! You are really just connecting the dots, so extrapolation beyond the training range should not be trusted in general.



In practice, a sufficient number of nodes are chosen for a given task and a training set is chosen to produce the best overall results for the relevant range of the data.

As the number of nodes gets larger, the accuracy in function approximations improves and the distinction between results using different activation functions vanishes.

Ultimately, the function you are trying to approximate is specified by the training target output for a given data input

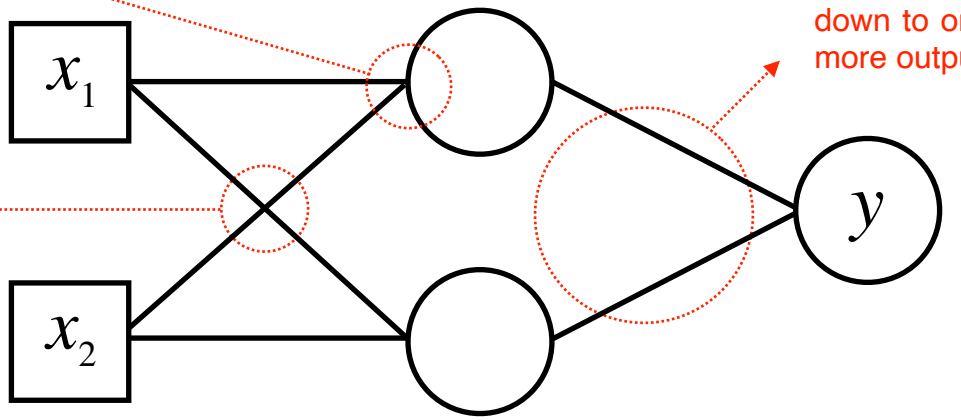
Networks & Training

A Simple “Feed-Forward” Network

Intersections (at nodes) between lines, planes, hyperplanes etc. in space of input parameters

Crossing connections don’t directly interact, but allow correlations between the inputs via the nodes

consolidation down to one or more outputs



 # nodes in a given layer: Defines the complexity of connections. It is the # geometric regions from the previous layer that can be linearly combined to form new regions. The overall # nodes is therefore related to how many connections are needed to sufficiently approximate the function that maps inputs to outputs.

 # layers: Defines the complexity of the base topologies. Each new layer uses the topologies defined by the previous layer as a basis to form more complex selection regions

A Simple “Feed-Forward” Network

 # nodes in a given layer: Defines the complexity of connections. It is the # geometric regions from the previous layer that can be linearly combined to form new regions. The overall # nodes is therefore related to how many connections are needed to sufficiently approximate the function that maps inputs to outputs.

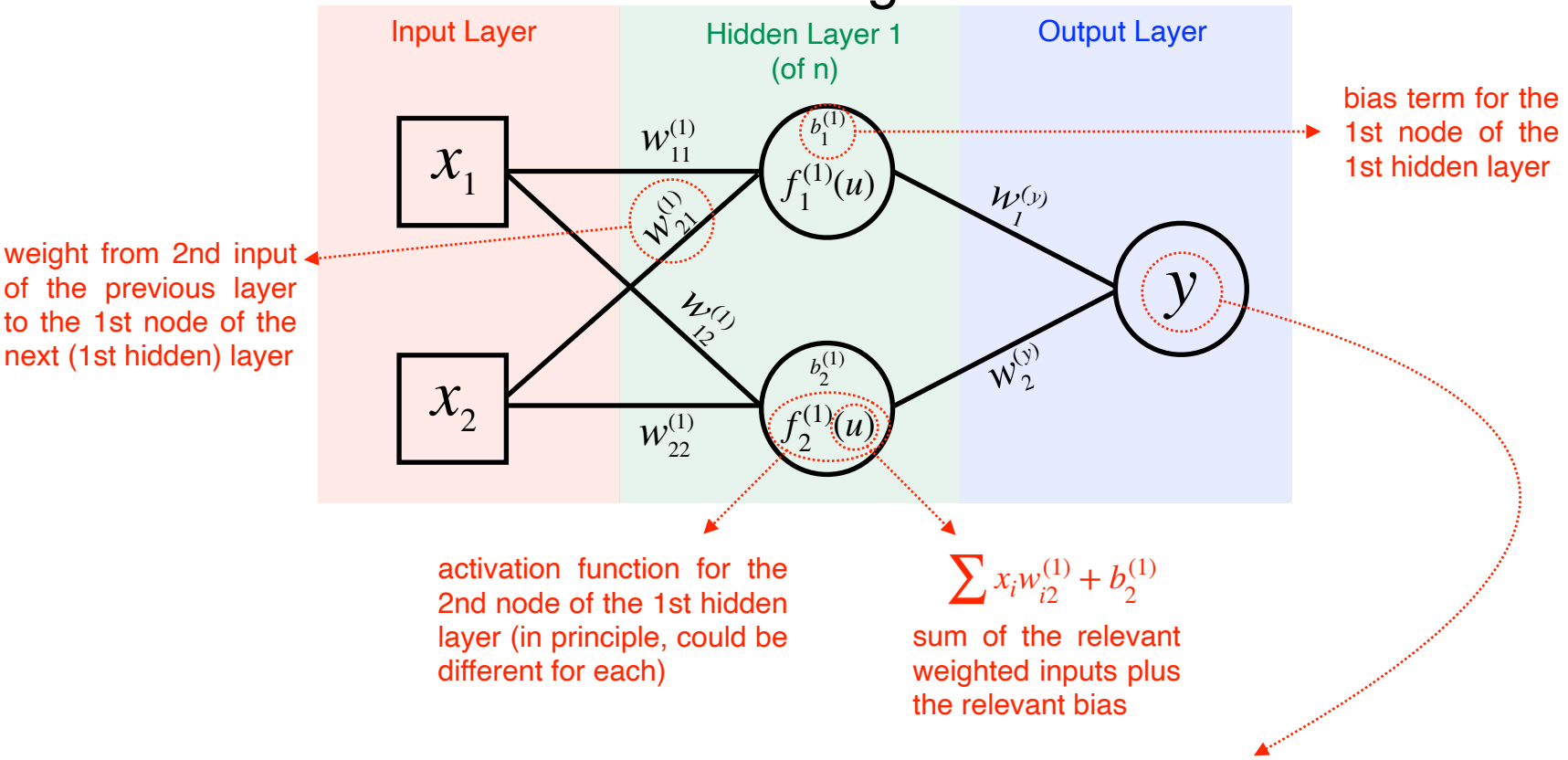
 # layers: Defines the complexity of the base topologies. Each new layer uses the topologies defined by the previous layer as a basis to form more complex selection regions

Typically used for **regression** (*i.e.* fitting best parameter values) or **classification** (discriminating categories)

Networks need not be that complex for most tasks - recall that just a single perceptron can do linear discrimination with multiple variables!

A reasonable strategy is to start simple, and then add complexity if it improves performance

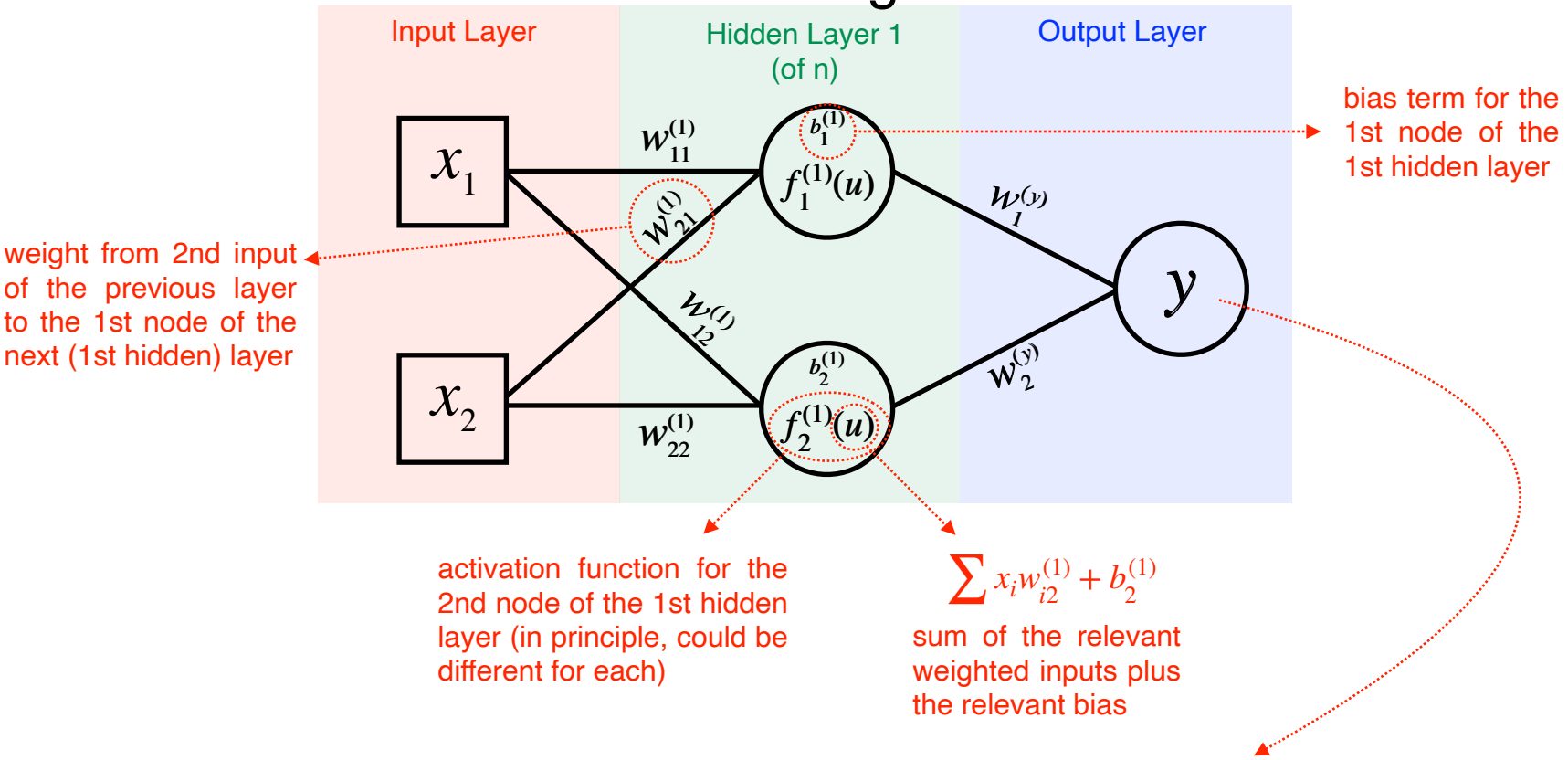
I will use the following nomenclature:



There may be multiple outputs y_n which, in general, are functions of the weighted inputs resulting from the last hidden layer that are chosen based on the particular application. This could just be the sum, or a threshold function for classification, or something else.

A given training run will typically start by ascribing random values (between -1 and 1) to the weights and biases, and then iterate…

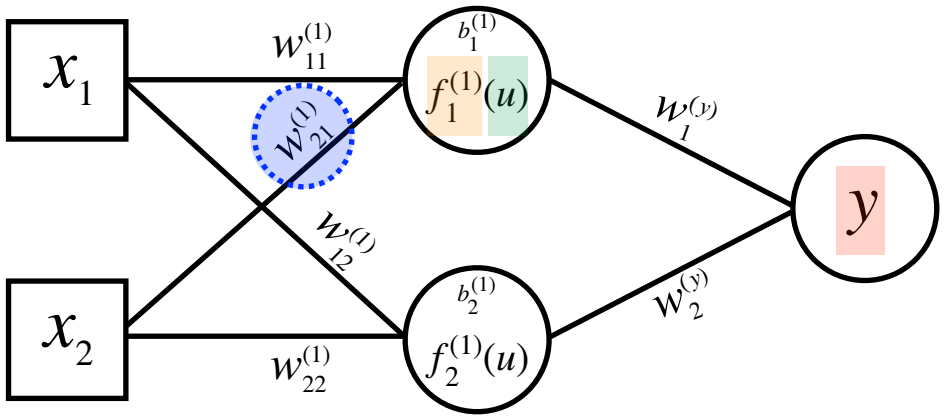
I will use the following nomenclature:



There may be multiple outputs y_n which, in general, are functions of the weighted inputs resulting from the last hidden layer that are chosen based on the particular application. This could just be the sum, or a threshold function for classification, or something else.

A given training run will typically start by ascribing random values (between -1 and 1) to the weights and biases, and then iterate…

Training via Gradient Descent & Back-Propagation



$\rightarrow L(y)$

Define a "loss function" that you would like to minimise. For example, this could be $\frac{1}{2}(y - y_0)^2$ or $-\ln \mathscr{L}$ etc.

Recall that, for gradient descent, we iterate according to: $\vec{x}_{n+1} = \vec{x}_n - \lambda \nabla f(\vec{x})\,|_{x=x_n}$

So we need gradients of the loss function for every parameter we want to tune!

But chain-rule comes to our rescue…

for example

$$\frac{\partial L(y)}{\partial w_{21}^{(1)}} = \left(\frac{\partial L(y)}{\partial y}\right)\left(\frac{\partial y}{\partial f_1^{(1)}}\right)\left(\frac{\partial f_1^{(1)}}{\partial u}\right)\left(\frac{\partial u}{\partial w_{21}^{(1)}}\right)$$

typically chosen to be simple functions

relevant unweighted input from previous layer

Start at the end and work backwards, passing on common factors to the next set of gradient calculations

Training is done after a fixed number of events ("batches") that are processed simultaneously, after which weights and biases are adjusted based on the averaged gradients for that batch

3 More
Useful Tools

Sigmoid Activation Function

Especially in a long chain of nodal interactions, it is useful to have an activation function that permits output as a continuous variable but contained within a standardised range.

One popular choice is the sigmoid function:

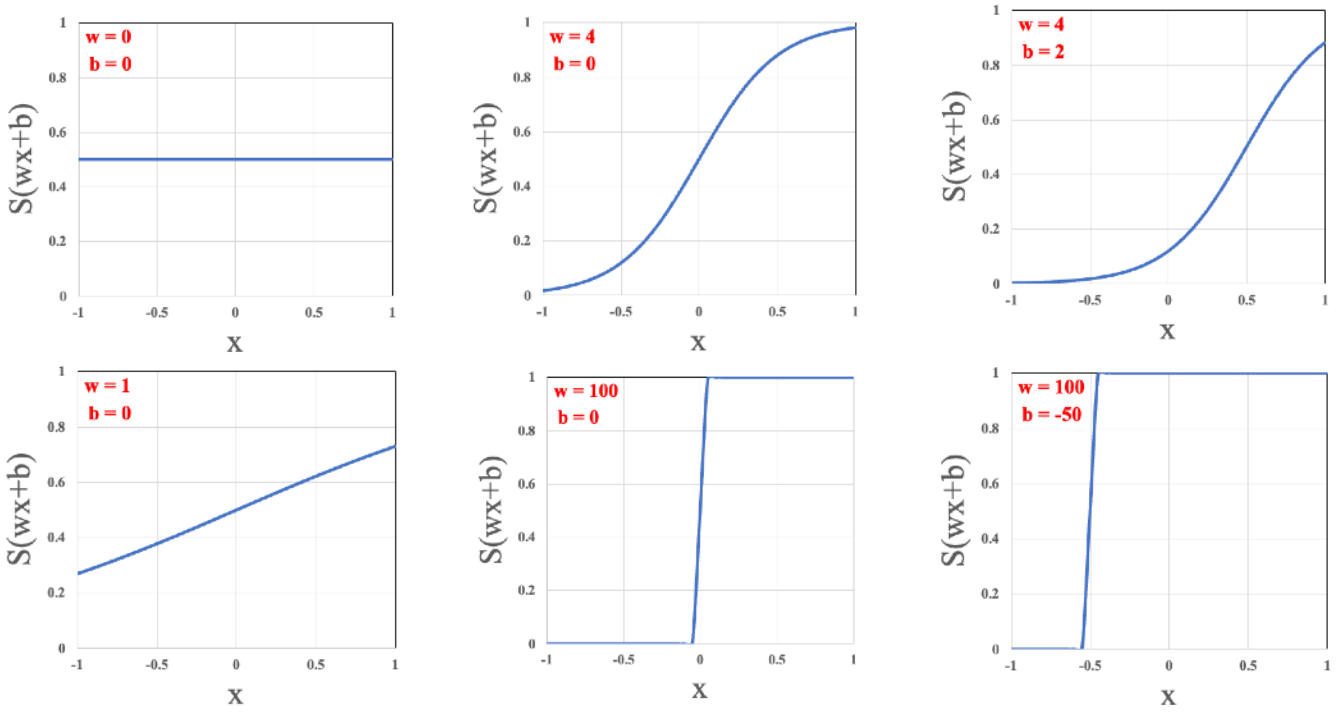
$$S(u) \equiv \frac{1}{1+e^{-u}}$$

where

$$u = \sum w_i x_i + b$$

and the sum is over all the connections coming into the node

A very flexible function!



Softmax (Boltzmann-like) Probabilities

(a common output function for classification/regression)

In general, weights and biases are allowed to take on any real number values, leading to outputs that span the range from -∞ to ∞, with more negative values typically indicating lower probabilities (*e.g.* less likely to have engaged an activation function).

Thus, if we wish to characterise such outputs as approximate probabilities, it seems reasonable to first exponentiate these to map -∞ to 0, and then normalise the sum of outputs to 1:

approximate probability associated with the ith output

relevant weighted input from previous layer

$$\tilde{p}_i \sim \frac{e^{z_i}}{\sum_j e^{z_j}} \equiv S(z_i)$$

sum over all weighted inputs

This is, in fact, analogous to the Boltzmann probability distribution. In ML it is known as "softmax," as it is a more nuanced number than a "hard" binary score.

Useful derivatives for back-propagation:

$$\frac{\partial S(z_i)}{\partial z_j} = \begin{cases} S(z_i)\left(1 - S(z_i)\right) & for\ i = j \\ -S(z_i)S(z_j) & for\ i \neq j \end{cases}$$

bonus exercise!

Cross Entropy

(a common loss function for classification/regression)

Say that we train a given ANN on a collection of N different events containing k known classifications that we wish the ANN to predict. The true number of events in each class is n_i, and the predictions for each class by the ANN for a given set of weights and biases are m_i.

The corresponding probabilities predicted for each class is then given by $q_i(w,b) = m_i/N$, and we can write the likelihood for the correct event classification given the model:

$$\mathscr{L}(\text{correct classes} \mid \mathbf{w}, \mathbf{b}) = \prod_{i=1}^{k} \left(q_i(\mathbf{w}, \mathbf{b})\right)^{n_i} \quad \Rightarrow \quad \ln \mathscr{L} = \sum_{i=1}^{k} n_i \ln q_i(\mathbf{w}, \mathbf{b})$$

Note that: $$\frac{1}{N} \ln \mathscr{L} = \sum_{i=1}^{k} \left(\frac{n_i}{N}\right) \ln q_i(\mathbf{w}, \mathbf{b}) = \sum_{i=1}^{k} p_i \ln q_i(\mathbf{w}, \mathbf{b})$$

"true" class probability, as N → large

Want to maximise the likelihood, or minimise →

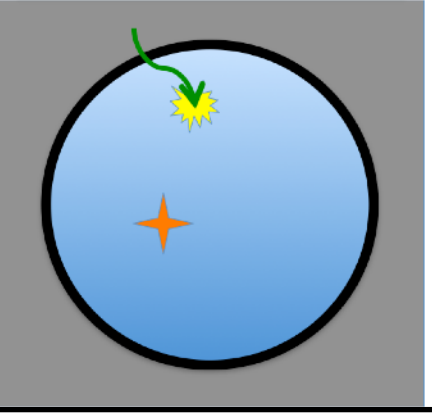
$$-\frac{1}{N} \ln \mathscr{L} = -\sum_{i=1}^{k} p_i \ln q_i(\mathbf{w}, \mathbf{b}) \equiv \mathscr{H}\left(p, q(\mathbf{w}, \mathbf{b})\right)$$

"Cross Entropy"

Useful derivative for back-propagation: if $q_i = S(z_i)$ $\quad \frac{\partial \mathscr{H}}{\partial z_i} = S(z_i) - p_i$

bonus exercise!


Let's
Play



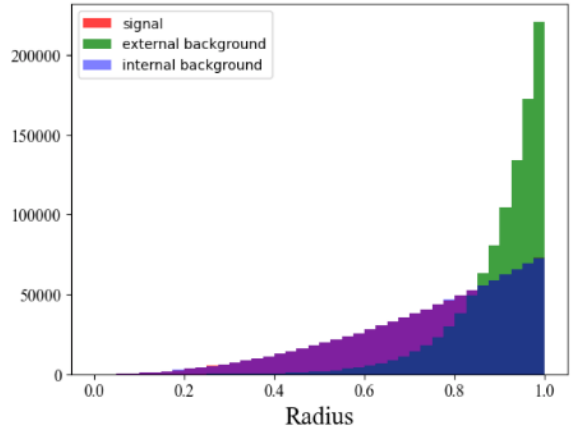
Signal
External background
Internal background

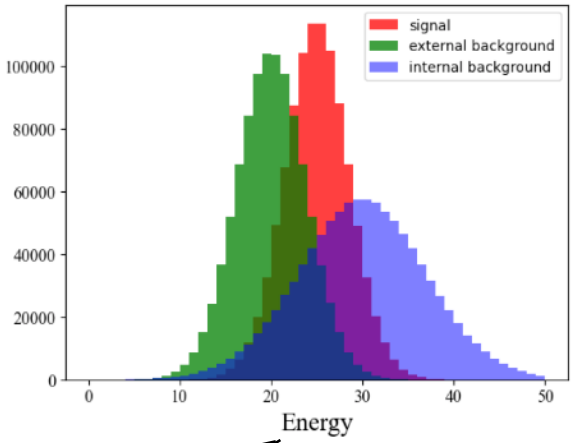
Model Data Set

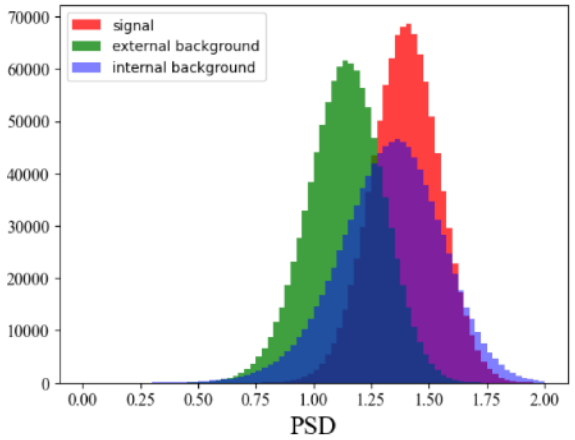
Position (radius)
Energy
Pulse Shape Discrimination

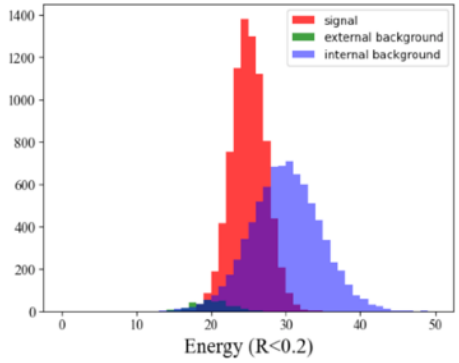
3 categories of events, each with 3 features

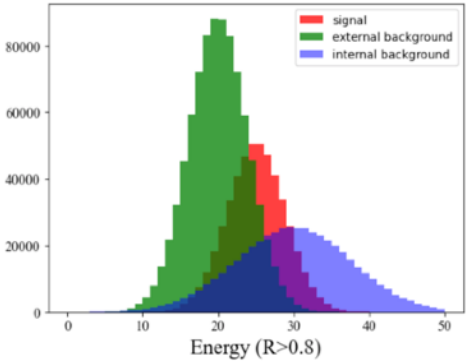
- Signal and internal backgrounds are uniform in detector volume [~R^3]
- External background falls exponentially from detector edge [~ exp((1-R)/0.1)]
- Energy resolution is twice as bad at the detector edge compared to the centre
- Pulse Shape Discrimination values scale linearly with sqrt of apparent energy

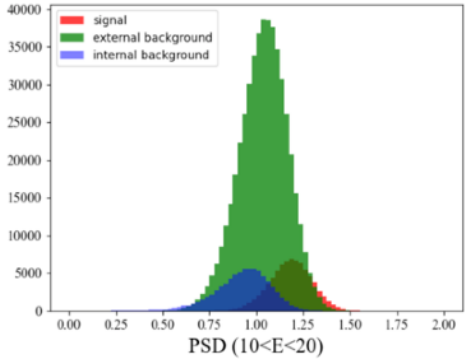


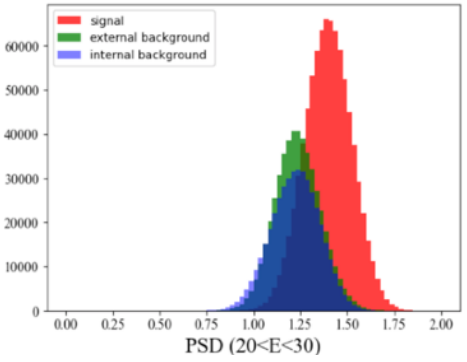






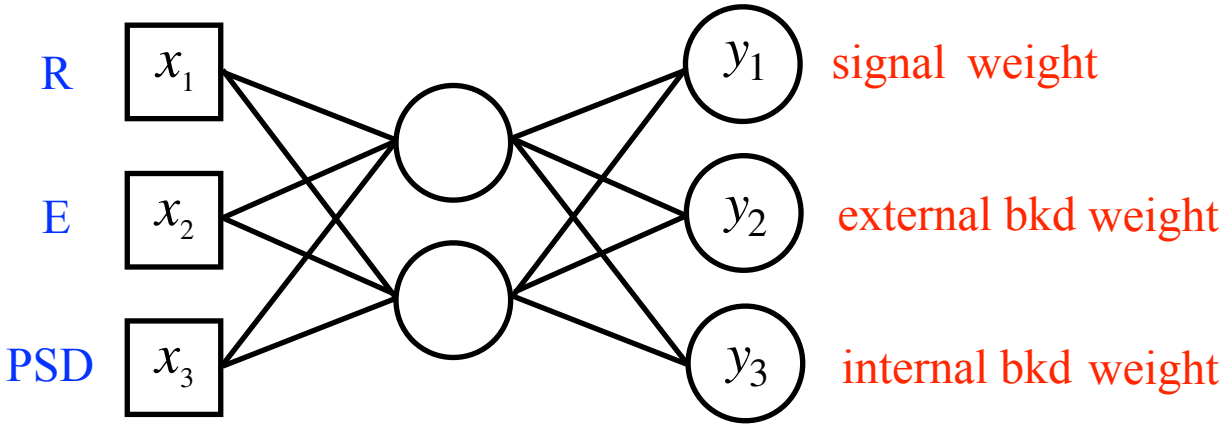






Defining the Network (PyTorch)

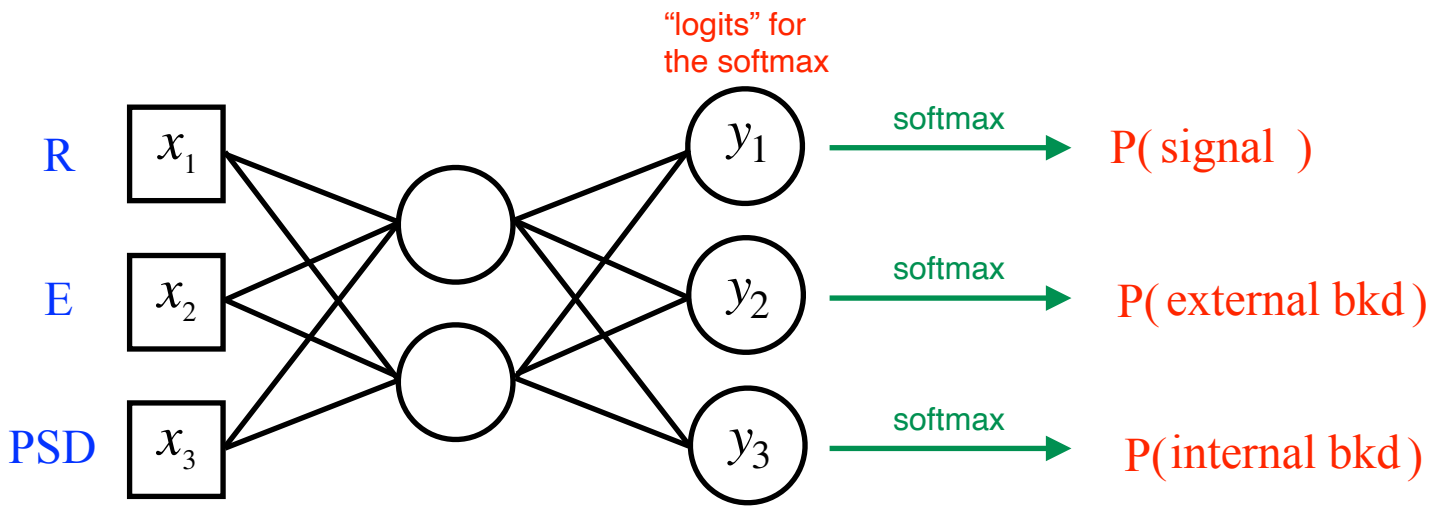
```python
import torch
import torch.nn as nn

# Create a model class that inherits nn.Module
class NeuralNet(nn.Module):
    # Input layer (3 features) -> hidden layer 1 (2 nodes) -> output (3 categories)
    def __init__(self,input=3,h1=2, out=3):
        super(NeuralNet,self).__init__() # Call constructor of parent class
        self.sigmoid = nn.Sigmoid()      # Use sigmoid activation function
        self.h1 = nn.Linear(input,h1)    # Fully connected linear links from inputs to hidden layer
        self.out = nn.Linear(h1,out)     # Linear output from hidden layer (also fully connected)

# Define the sequence of forward flow through the network
    def forward(self, x):
        out = self.h1(x)                 # Move inputs via fully connected links to hidden layer
        out = self.sigmoid(out)          # Next, apply sigmoid activation function
        out = self.out(out)              # And then advance to output
        return out
```



Selection regions are largely defined by the intersection of 2 planes in the space of the input parameters

Defining the Network (PyTorch)

```python
import torch
import torch.nn as nn

# Create a model class that inherits nn.Module
class NeuralNet(nn.Module):
    # Input layer (3 features) -> hidden layer 1 (2 nodes) -> output (3 categories)
    def __init__(self,input=3,h1=2, out=3):
        super(NeuralNet,self).__init__() # Call constructor of parent class
        self.sigmoid = nn.Sigmoid()      # Use sigmoid activation function
        self.h1 = nn.Linear(input,h1)    # Fully connected linear links from inputs to hidden layer
        self.out = nn.Linear(h1,out)     # Linear output from hidden layer (also fully connected)

# Define the sequence of forward flow through the network
    def forward(self, x):
        out = self.h1(x)                 # Move inputs via fully connected links to hidden layer
        out = self.sigmoid(out)          # Next, apply sigmoid activation function
        out = self.out(out)              # And then advance to output
        return out
```



Selection regions are largely defined by the intersection of 2 planes in the space of the input parameters

Note: This very simple network still has 14 free parameters!

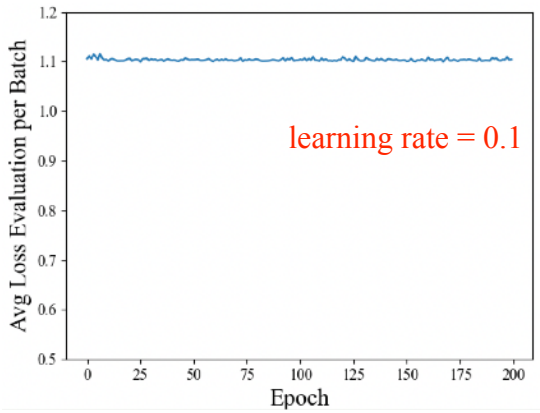
Training the Network

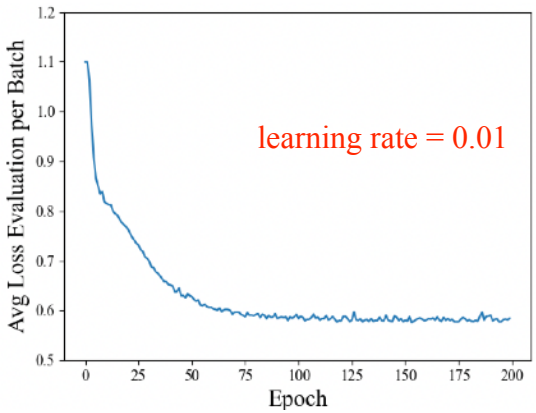
```
from torch.utils.data import DataLoader, TensorDataset

model=NeuralNet()       # Instantiate neural net

# Separate training set data frame into inputs and outputs, then turn these into tensors
X = Training_Set.drop('class',axis=1)
y = Training_Set['class']
X = torch.FloatTensor(X.values)
y = torch.LongTensor(y.values)

# Define loss function. Note: CrossEntropyLoss automatically assumes softmax for back propagation
criterion = nn.CrossEntropyLoss()

# Define optimizer to be used in back propagation and set the learning rate
optimizer = torch.optim.Adam(model.parameters(),lr=0.01)

N_EPOCHS = 100         # Number of times to run through training set ('epochs')
BATCH_SIZE = 16       # Number of events to evaluate simultaneously before updating weights
accumulation_steps=5 # Trick to increase effective batch size when memory limited by averaging gradients
losses = []           # Will keep record of loss function evaluations

# Create a DataLoader to create randomised batches
dataset = TensorDataset(X, y)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

for epoch in range(N_EPOCHS):                                    # Loop over epochs
    sumloss=0
    for id_batch, (x_batch, y_batch) in enumerate(dataloader):  # Loop over batches
        y_batch_pred = model(x_batch)                            # Predicted outputs from current network
        loss = criterion(y_batch_pred, y_batch)                  # Evaluate loss function
        sumloss=sumloss+loss                                     # Sum the loss evaluations for the epoch
        loss.backward()                                          # Back-propagate
        if not id_batch%accumulation_steps:                      # Implement gradient accumulation
            optimizer.step()                                     # Update the weights
            optimizer.zero_grad()                                # Zero the gradient to start again
    losses.append(sumloss.detach().numpy()/len(X))               # Keep track of loss evaluations per event

# Show progress in 10% steps and give average loss per event for the end of the current epoch
    if (epoch+1)%(N_EPOCHS/10) == 0: print(f'Epoch: {epoch+1} and loss: {sumloss/len(X)}')

plt.plot(losses)
plt.show()
```

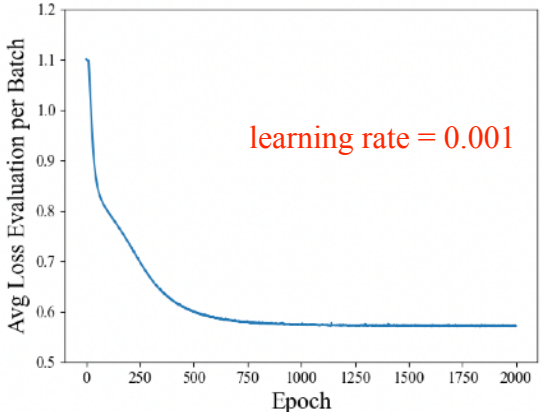
Saving and loading previously trained network weights:

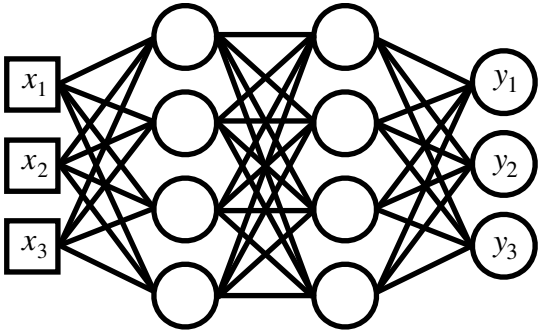
```
torch.save(x, 'tensor.pt')                    # Save to file

torch.load('tensors.pt', weights_only=True)   # Load from file
```

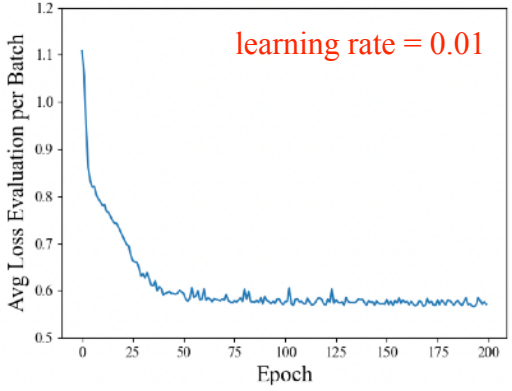
Training with 1000 events each of signal, external and internal backgrounds:











No need for a more complex network: a slightly noisier ride to the same average loss!

```python
import torch
import torch.nn as nn

# Create a model class that inherits nn.Module
class NeuralNet(nn.Module):
    # Input layer (3 features) -> hidden layer 1 (4 nodes) -> hidden layer 2 (4 nodes) -> output (3 categories)
    def __init__(self,input=3,h1=4,h2=4,out=3):
        super(NeuralNet,self).__init__() # Call constructor of parent class
        self.sigmoid = nn.Sigmoid()      # Use sigmoid activation function
        self.h1 = nn.Linear(input,h1)    # Fully connected linear links from inputs to hidden layer
        self.h2 = nn.Linear(h1,h2)       # Fully connected linear links from 1st to 2nd hidden layer
        self.out = nn.Linear(h2,out)     # Linear output from 2nd hidden layer (also fully connected)

# Define the sequence of forward flow through the network
    def forward(self, x):
        out = self.h1(x)                 # Move inputs via fully connected links to 1st hidden layer
        out = self.sigmoid(out)          # Next, apply sigmoid activation function
        out = self.h2(x)                 # Move via fully connected links to 2nd hidden layer
        out = self.sigmoid(out)          # Apply sigmoid activation function again
        out = self.out(out)              # And then advance to output
        return out
```

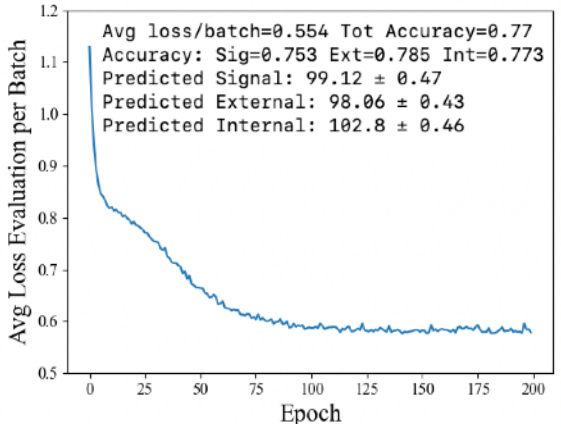
Can tune ‘hyperparameters’ such as learning rate and network construction to try to achieve the very best results for a given training set, then use a separate ‘validation’ training set with these parameters fixed to perform an unbiased training (since ‘overtraining’ on fluctuations is likely to have occurred when tuning)

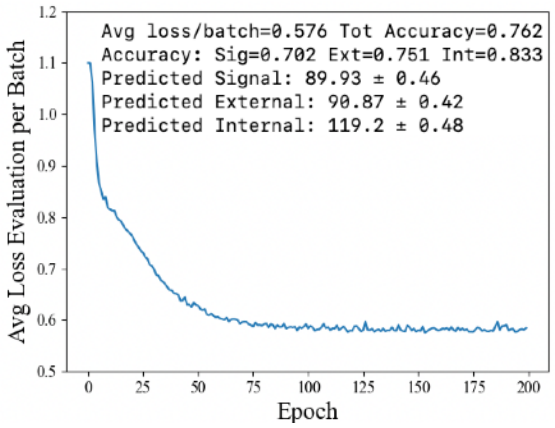
Evaluation

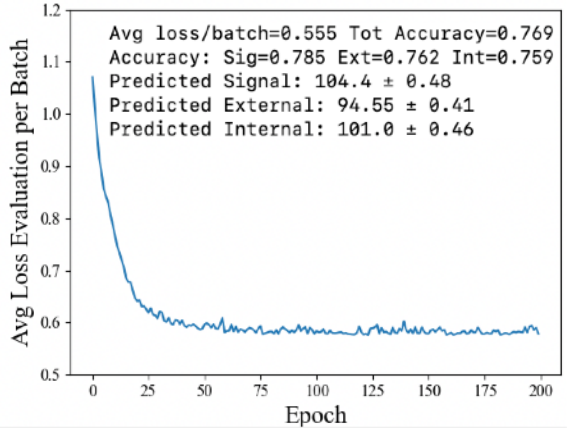
```
def softmax(x):                                 # Define softmax function to convert weights to probabilities
    x_norm = np.sum(np.exp(x), axis=0)
    for i in range(len(x)):
        x[i]=np.exp(x[i])/x_norm
    return x

sumavg=[0]*3
sum2avg=[0]*3
miss=0
for i in range(10):                             # Loop over 10 test data sets stored as data frame list
    X = Test[i].drop('class',axis=1)
    y = Test[i]['class']
    X = torch.FloatTensor(X.values)
    y = torch.LongTensor(y.values)

    dataset = TensorDataset(X, y)
    dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=False)

    sumloss=0
    sum_out=[0]*3
    for id_batch, (x_batch, y_batch) in enumerate(dataloader):
        ym=model(x_batch)
        sumloss = sumloss+ criterion(ym, y_batch)
        y_batch=y_batch.detach().numpy()
        ym=ym.detach().numpy()
        for j in range(len(x_batch)):
            if ym[j,y_batch[j]] != np.max(ym[j]): miss=miss+1  # If max probability isn't for true class, call miss
            softmax(ym[j])
            sum_out=sum_out+ym[j]           # Sum probabilities for each class
    sumavg=sumavg+sum_out/10                # Compute the average sum over the 10 data sets
    sum2avg=sum2avg+(sum_out**2)/10         # Compute the variance

print(f"Avg loss per event: {sumloss.detach().numpy()/len(X):.3}")
print(f"Fraction of misses: {miss/(10*len(X)):.3}")
print(f"Signal: {sumavg[0]:.4} ± {np.sqrt(sum2avg[0]-sumavg[0]**2)/np.sqrt(10):.2}")
print(f"External: {sumavg[1]:.4} ± {np.sqrt(sum2avg[1]-sumavg[1]**2)/np.sqrt(10):.2}")
print(f"Internal: {sumavg[2]:.4} ± {np.sqrt(sum2avg[2]-sumavg[2]**2)/np.sqrt(10):.2}")
```

Stability of Results & Quantifying Uncertainty

Apply trained network to 100 tests data sets each containing 100 events of each class:

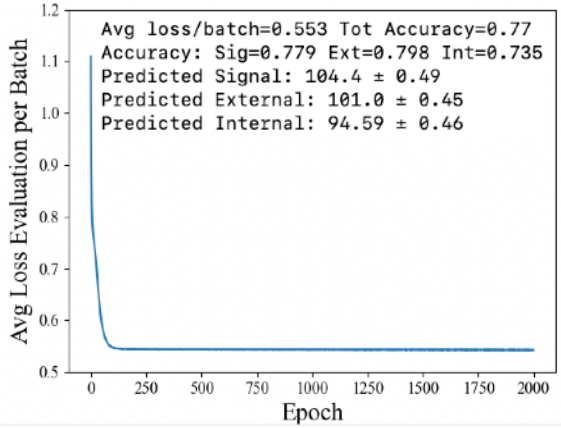


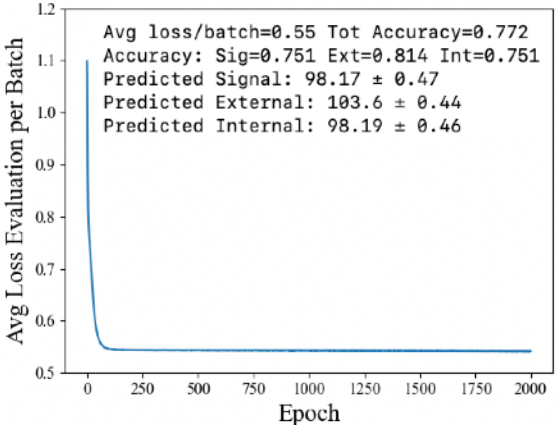


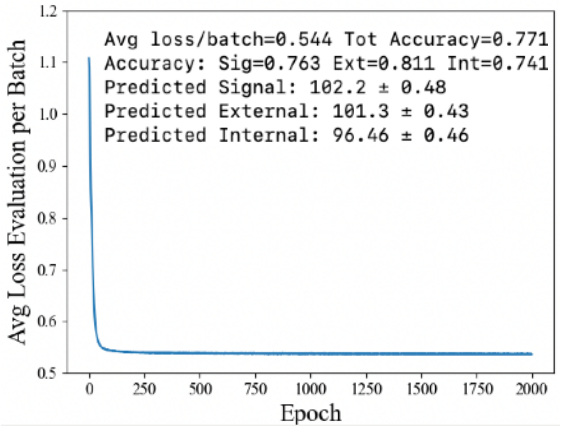


In the right ballpark, but systematic uncertainties that are not reflected by the variances resulting from the trained network. Plus, re-training the network, even on the same training data but starting with a different random seed, leads to different learning curves and slightly different results outside of variances when applied to the same test data!

Try 10 times larger training set, 10 times lower learning rate, 10 times more epochs:







The issue persists! **This is a well-known stability problem of neural nets, resulting from an approach that is under-constrained, resulting in many local minima!**

The stability problem is one of the central limitations of ANNs and is a significant area of study. Proposed approaches to mitigate this include:

- **k-Fold Cross-Validation:** Divide training set up into k smaller pieces, alternately treating one of these as a "test set" and the rest as the "training set" to characterise the performance of a particular network.
- **"Bayesian" Networks:** Rather than fixed weights, sample each network weight from an assumed prior distribution (typically a Gaussian) so that multiple evaluations provide a distribution of results that can be used to assess uncertainty.
- **Adversarial Examples:** These involve methods to produce training or test data sets that specifically target net vulnerabilities. For example, the Fast Gradient Sign Method (FGSM) produced adversarial examples by perturbing the inputs in the direction of the gradient*: $adv_x = x + \epsilon \times sign\left[\nabla_x L(y)\right]$, where ε is chosen to be some (arbitrary) small fraction of the input range. These can then be used to assess robustness
- **Deep Ensembles:** Train a small ensemble of multiple networks, apply these to the same test data set, and evaluate the results based on the mean and variance of the ensemble. These can also be combined with adversarial examples**. Tends to be comparable or better than Bayesian networks.

All of these suffer from some degree of computational costs, logistical limitations and arbitrariness

*Goodfellow, Shlens & Szegedey, arXiv:1412.6572 [stat.ML]

**Lakshminarayanan, Pritzel, & Blundell, arXiv:1612.01474 [stat.ML]

Another issue…

ANN trained on equal mix of classes (10000 each), but then applied to data sets containing: 50 signal, 200 external background, 150 internal background:

```
Avg loss/batch: 0.531   Tot Accuracy: 0.79
Accuracy: Sig=0.718 Ext=0.857 Int=0.725
Predicted Signal: 89.8 ± 0.48
Predicted External: 179.0 ± 0.44
Predicted Internal: 131.2 ± 0.47
```

Disaster! For cases where there are event-by-event ambiguities, the estimated relative probabilities depend on the assumed prior distributions for the classes in the training set.


```
criterion = nn.CrossEntropyLoss()
```

better…

```
# define the weights for each class
class_weights = torch.tensor([0.5, 2.0, 1.5])

criterion = nn.CrossEntropyLoss(weight=class_weights)
```

```
Avg loss/batch: 0.478   Tot Accuracy: 0.815
Accuracy: Sig=0.473 Ext=0.932 Int=0.774
Predicted Signal: 47.47 ± 0.34
Predicted External: 212.3 ± 0.45
Predicted Internal: 140.2 ± 0.45
```

Although,

Poor signal accuracy!

Network has learned to guess at the signal less often

Can get here by iteration: re-train with extracted class proportions as new input class weights, and keep going around until the output class proportions match

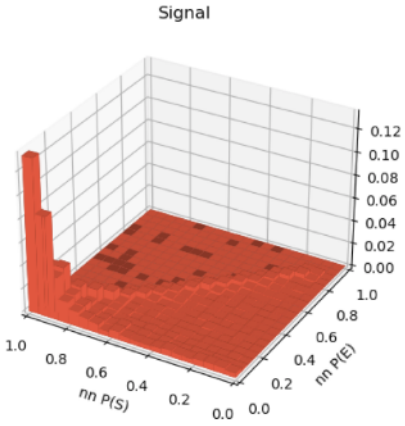
TRAP!! If you train an ANN as a discriminant and verify it with a test set that has a different composition from the data, you could get things wrong!

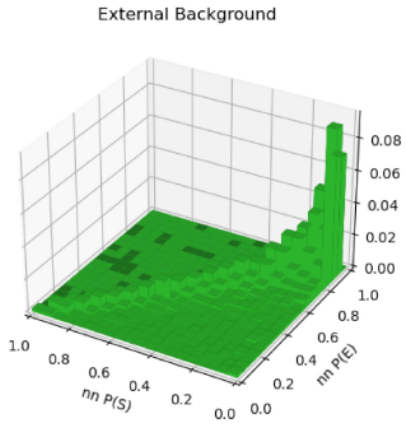
An Alternative Approach

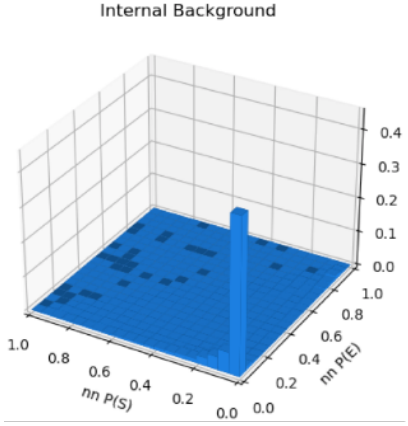
(at least for classification tasks)

Treat output from ANNs as a statistic: calibrate it, look at its distribution, derive PDFs for different classes in terms of it, and then apply standard statistical techniques, such as likelihood

In the present case, only 2 of the 3 ANN outputs are independent, so form PDFs in terms of the softmax outputs for signal and external background trained with signal=ext.=int.=1000

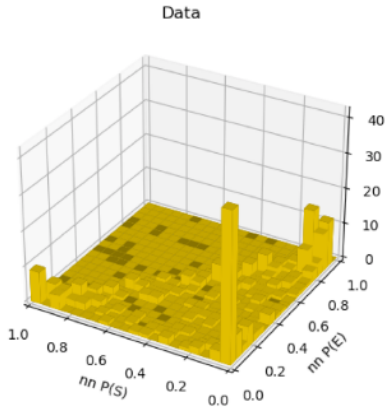




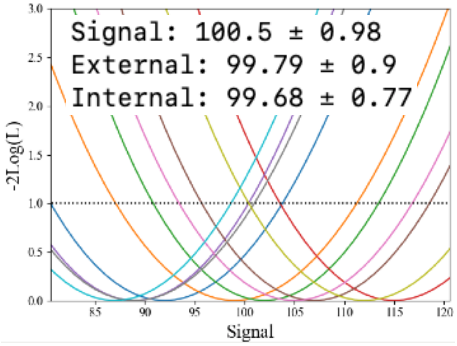


$$-2\log\mathscr{L} = -2\sum_{i=1}^{nbins} NData_i \log\left|\left(\frac{N_{sig}}{N_{tot}}\right)PDF_S(nn(S_i,E_i)) + \left(\frac{N_{ext}}{N_{tot}}\right)PDF_E(n(S_i,E_i)) + \left(\frac{N_{tot}-N_{sig}-N_{ext}}{N_{tot}}\right)PDF_In(S_i,E_i))\right.$$

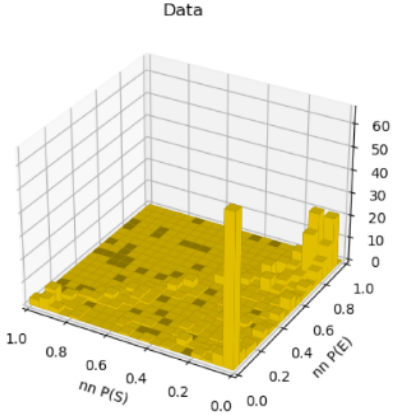
signal=100, external bkd = 100, internal bkd = 100:



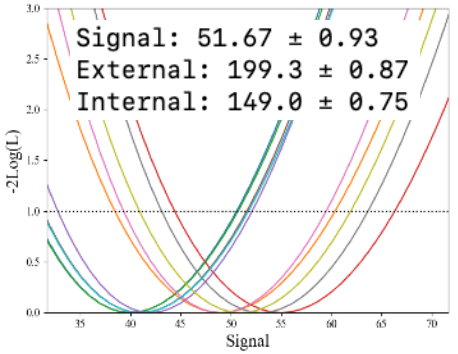
profile likelihood:



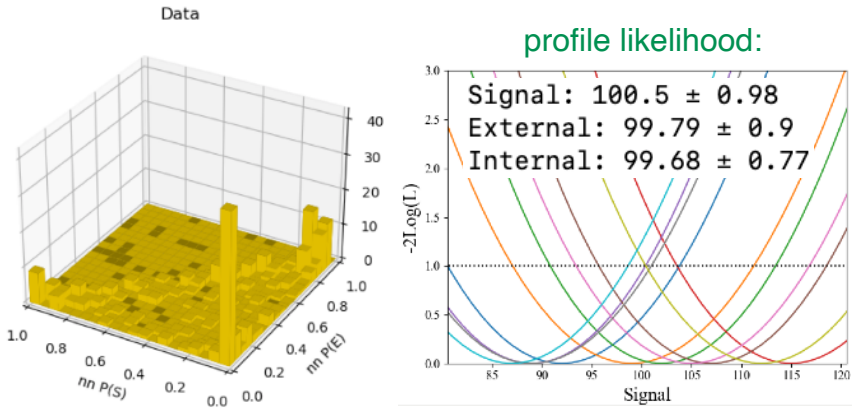
signal=50, external bkd = 200, internal bkd = 150:



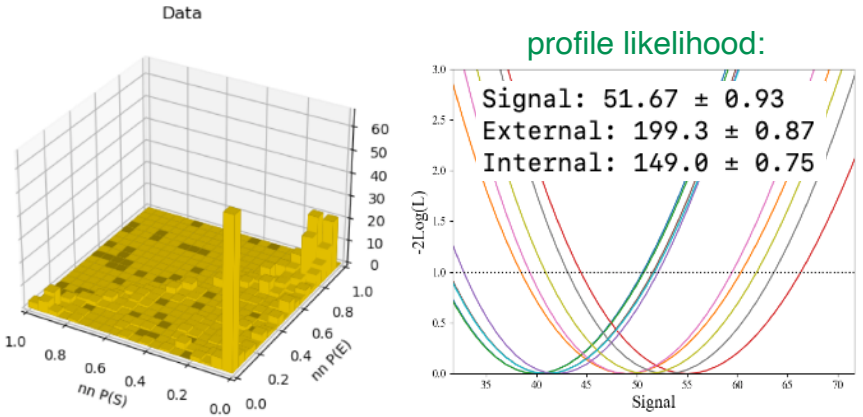
profile likelihood:



signal=100, external bkd = 100, internal bkd = 100:



signal=50, external bkd = 200, internal bkd = 150:



More accurate class predictions, robust to class composition, variances correctly reflect accuracy, likelihood gives reliable event-by-event uncertainties!

Similarly for BDTs:

Can characterise distributions of functional BDT evaluations for each class, then use these as PDFs in a likelihood evaluation of the class normalisations for any given set of data!

Advantages:

- In many cases, can notably reduce the dimensionality of more standard likelihood formulations (*e.g.* could add a great many more features as ANN inputs without changing the dimensionality of the PDFs, which are based on a small number of outputs).
- Takes full advantage of ANN ability to characterise relevant parameter dependencies that may be otherwise difficult to formulate.
- Provides a correct statistical description of any trained ANN
- Allows for the proper construction of confidence and credibility intervals
- Permits the full relevant information content to be conveyed to others via the likelihood

The lesson is that you shouldn't trust probability distributions that emerge from ANNs, but instead directly derive these yourself from simulations and calibration data.

But this **IS** how we generally do things anyway for fitters, classifiers, analysis cuts etc.!

Final Note: Because intuition can sometimes be obscured with ANNs, it's always useful to directly compare performance levels with more conventional methods where possible. If they are vastly different, then it's worth digging further to understand the reasons more specifically and check whether some crucial piece of information was missed by the conventional analysis or if the ANN might be anomalously tuning on some unintended feature.