Lecture 12: Neural networks and deep learning

Nils Olovsson
nils.olofsson@igp.uu.se

Introduction to data science and Python programming for medical research
Uppsala University

2025

UPPSALA
UNIVERSITET

Content

Introduction

Introduction

Introduction: Problem

Have successfully used sklearn for classification and regression problems of various sizes and complexities.

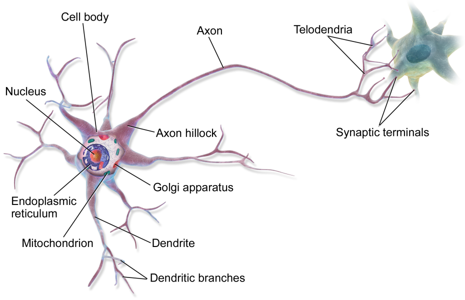
We can choose which method in sklearn to use but we don't have much saying on how it operates.

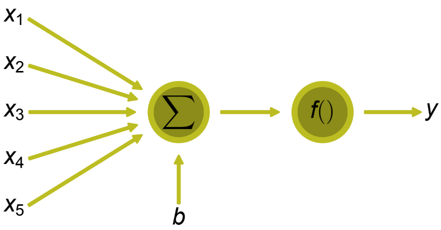
Also quite limited in doing complex tasks such as performing operations on *e.g.* images.

Pytorch is a deep learning package that provides a lot of flexibility.

Introduction: Artificial neuron

A neuron aggregates stimuli from multiple incoming connections.
When the sum of stimuli exceed some threshold the neuron is activated.
It then sends a signal of its own.
Mathematically modeled as a single node that sums weighted incoming values.
The activation can be modeled using a non-linear activation function.





https://en.wikipedia.org/wiki/Neuron

Introduction: Perceptron neural network (1950s)

The artificial neuron was introduced in the 1950s by Rosenblatt *et al.*, inspired by contemporary neuroscience.

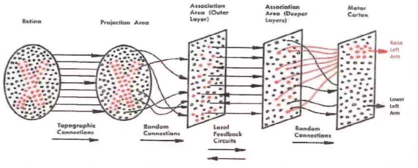


FIG. 1 — Organization of a biological brain. (Red areas indicate active cells, responding to the letter X.)

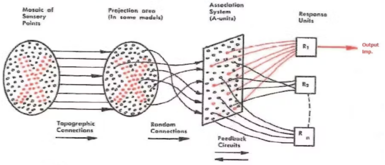
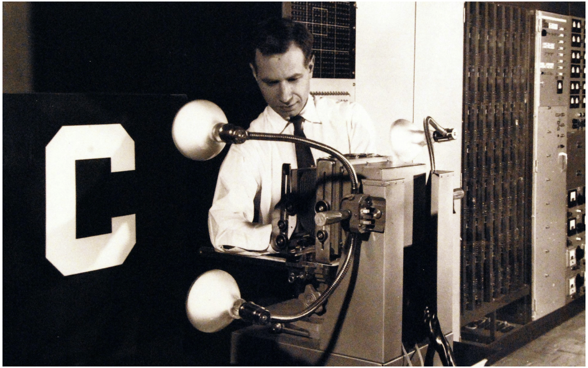


FIG. 2 — Organization of a perceptron.

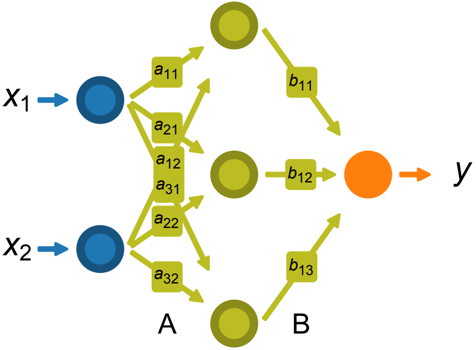


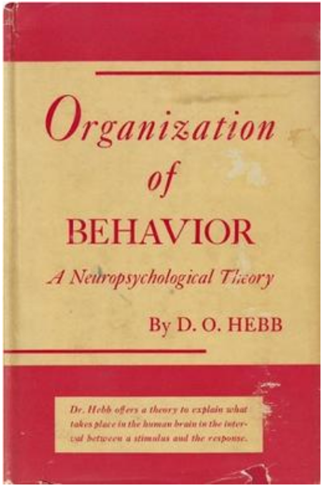
https://theconversation.com/how-psychologists-kick-started-ai-by-studying-the-human-mind-24854
https://americanhistory.si.edu/collections/object/nmah_334414

Introduction: Hebbian learning (1949)

How do we set the weights in a network of several *neurons*?

"Neurons that fire together, wire together."





https://en.wikipedia.org/wiki/Hebbian_theory
https://en.wikipedia.org/wiki/Organization_of_Behavior

Linear regression

Linear regression

Linear regression: Single neuron

Same problem as in lecture on linear regression.

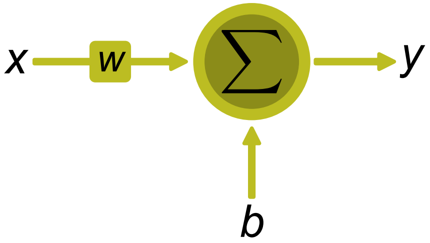
Want to fit a model of the type

$$y = wx + b$$

to data.

(We know since the lecture on linear regression that we solve this particular problem analytically to get a truly optimal solution.)

Model this with a **single neuron**.
The neuron has **one input**, a **bias** and a **one output**.
Output is linear so **no activation function** is needed.



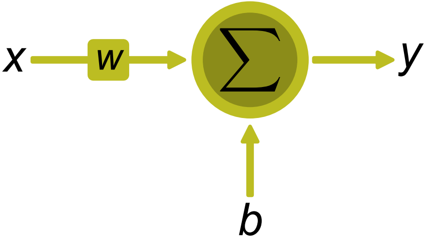
Linear regression: Python

```python
import torch
from torch import nn

class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(in_features=1,
                               out_features=1,
                               bias=True)
        return

    def forward(
        self,
        X: torch.Tensor,
    ) -> torch.Tensor:
        """ The nn.Module super class we inherit
        from expects this function to be defined.
        It will compute the output from the
        neural network given the input
        data matrix X.
        """
        X = self.layer(X)
        return X
```

Neural networks using Pytorch should inherit from the `nn.Module` base class.

Should contain all layers as member variables and a `forward(self, X)` function.

Can use `torch` to train network.



https://pytorch.org/docs/stable/nn.html
https://pytorch.org/docs/stable/generated/torch.nn.Module.html

Linear regression: Loss function

What is a **good fit** of a line to our data?

What is a **bad fit**?

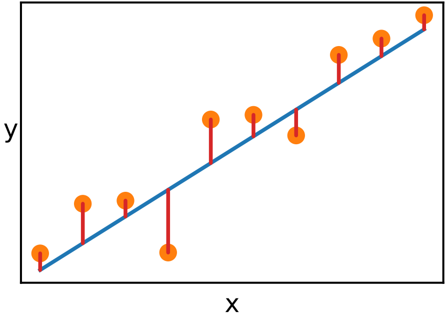
We define a **loss** function that compares the model to the data.

Mean of squared vertical distances, d, between the line and the N data points. This is almost exactly the same as we did to solve our previous regression problem!

```
# In Pytorch we instantiate a class which will
# compute the specified loss function
from torch import nn
loss_func = nn.MSELoss()
```

https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html
https://pytorch.org/docs/stable/nn.html#loss-functions

$$MSE = \frac{1}{N}\sum_i^N {d_i}^2 = \frac{1}{N}\sum_i^N (wx_i + b - y_i)^2$$



Linear regression: Optimizing parameters

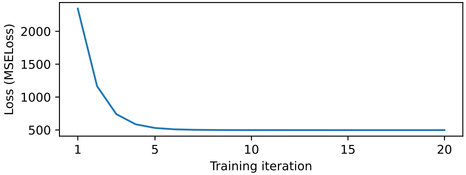
```
## Model instantiating and settings for training
from torch import optim
epochs = 20
loss_arr = np.zeros(epochs)
model = LinearRegressionModel()
loss_func = nn.MSELoss()

## The optimizer is instatiated with the parameters
## of the model. Learning rate determines how large
## steps to take in the direction of gradient
## descent. Stochastic Gradient Descent (SGD)
optimizer = optim.SGD(model.parameters(), lr=0.2)

## Train the linear regression network
for i in range(epochs):
    ## Reset optimizer
    optimizer.zero_grad()
    ## Compute output from model and loss
    y_pred = model(X_train)
    loss = loss_func(y_pred, y_train)
    ## Loss is propagated backwards in network
    loss.backward()
    ## Step in the direction of gradient descent
    optimizer.step()
    ## Store computed loss for plotting
    loss_arr[i] = loss.data.numpy()
```

https://pytorch.org/docs/stable/generated/torch.optim.SGD.html
https://pytorch.org/docs/stable/optim.html

In `sklearn` our supervised models were trained by calling the `fit(X, y)` function.

For our Pytorch models we have to specify explicitly how the training should be done!

Training iteration often called an epoch. The loss should decrease during training.



Linear regression: Python (torch and sklearn)

```
import torch
from torch import nn
from torch import optim
import numpy as np
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt

class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(in_features=1,
                               out_features=1,
                               bias=True)
        return

    def forward(
        self,
        X: torch.Tensor,
    ) -> torch.Tensor:
        X = self.layer(X)
        return X
## Create test data, cast to torch tensors
X, y = make_regression(n_samples=100, n_features=1,
                       noise=20.0, random_state=0)
X = torch.tensor(X).float()
y = torch.tensor(y[:, np.newaxis]).float()
X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=0.10)
## Model instantiating and settings for training
epochs = 20
loss_arr = np.zeros(epochs)
model = LinearRegressionModel()
loss_func = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.2)
## Train the linear regression network
for i in range(epochs):
    optimizer.zero_grad()
    y_pred = model(X_train)
    loss = loss_func(y_pred, y_train)
    loss.backward()
    optimizer.step()
    loss_arr[i] = loss.data.numpy()
## Training done. Evaluate model, 'inference_mode()'
with torch.inference_mode():
    y_pred_train = model(X_train).numpy()
    y_pred_test = model(X_test).numpy()
    r2_train = r2_score(y_train.numpy(), y_pred_train)
    r2_test  = r2_score(y_test.numpy(), y_pred_test)
    print(f"R2 train: {r2_train:.2f}")
    print(f"R2 test:  {r2_test:.2f}")
```

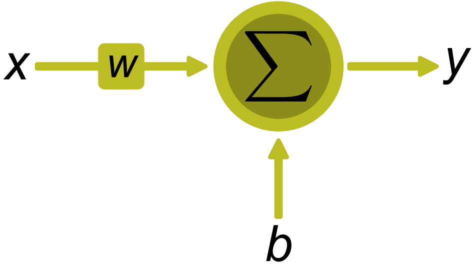
```
>> R2 train: 0.81
>> R2 test:  0.78
```

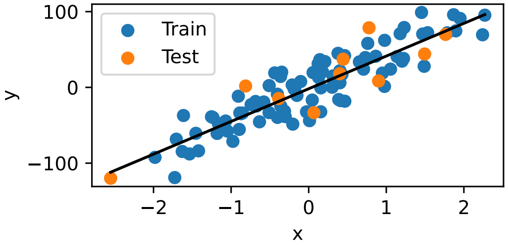
Linear regression: Neural network regression

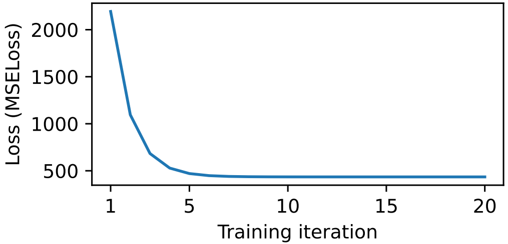
The two parameters, w and b, in the model are optimized such that it fits the data.

The model itself represents a line.

$$y = wx + b$$







Logistic regression

Logistic regression

Logistic regression: Single neuron

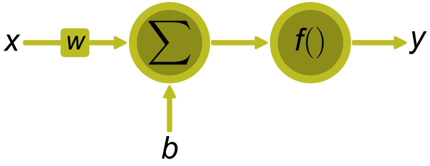
Same problem as in lecture on regression problems.

Want to fit a model of the type

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

to data.

Model this with a **single neuron**.
The neuron has **one input**, a **bias** and **one output**.
Output should be a probability, **sigmoid activation function** used.



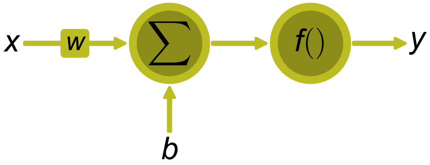
Logistic regression: Python

```python
import torch
from torch import nn
class LogisticRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(in_features=1,
                               out_features=1,
                               bias=True)
        return

    def forward(
        self,
        X: torch.Tensor,
    ) -> torch.Tensor:
        X = self.layer(X)
        X = nn.functional.sigmoid(X)
        return X

    def predict(
        self,
        X: torch.Tensor,
    ) -> torch.Tensor:
        with torch.inference_mode():
            y: torch.Tensor = self(X)
            y[y > 0.5] = 1
            y[y < 1.0] = 0
        y = y.to(torch.int32)
        return y
```

Inherit from `nn.Module` base class.

The `forward(self, X)` function is the same as for linear regression, except for the `sigmoid` activation function.

Convenience function `predict(self, X)`.



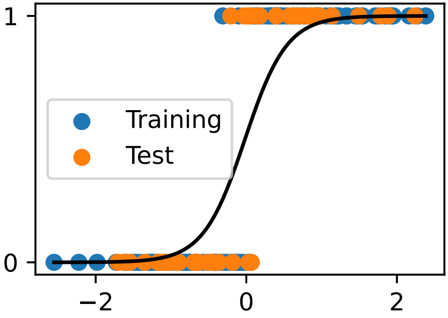
Logistic regression: Loss function and optimizer

You might remember from the lecture on regression models that we should not use the sum of squared distances as an error for logistic regression.

For single class problems should use **binary cross entropy (BCE)**.

Can use same stochastic gradient descent (SGD) optimizer as before.

```
# In Pytorch we instantiate a class which will
# compute the specified loss function
from torch import nn
loss_func = nn.BCELoss()
```



https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html
https://pytorch.org/docs/stable/nn.html#loss-functions

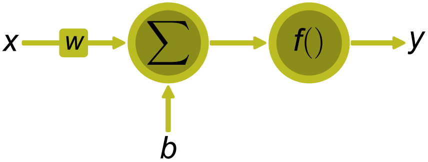
Logistic regression: Python

```
import torch
from torch import nn
from torch import optim
import numpy as np
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
class LogisticRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(in_features=1,
                               out_features=1,
                               bias=True)
        return

    def forward(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        X = self.layer(X)
        return nn.functional.sigmoid(X)

    def predict(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        with torch.inference_mode():
            y: torch.Tensor = self(X)
            y[y > 0.5] = 1
            y[y < 1.0] = 0
        return y.to(torch.int32)
```

```
X, y = make_regression(n_samples=200, n_features=1,
                       noise=20.0, random_state=0)
X = torch.tensor(X).float()
y = torch.tensor(y[:, np.newaxis]).float()
y = torch.round(nn.functional.sigmoid(y))
X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=0.25)
## Create and fit model
model = LogisticRegressionModel()
optimizer = optim.SGD(model.parameters(),
                      lr=0.02, momentum=0.5)
loss_func = nn.BCELoss()
epochs = 1000
loss_arr = np.zeros(epochs)
for i in range(epochs):
    optimizer.zero_grad()
    y_pred = model(X_train)
    loss = loss_func(y_pred, y_train)
    loss.backward()
    optimizer.step()
    loss_arr[i] = loss.data.numpy()
y_tr = y_train.to(torch.int32).numpy()
y_ts  = y_test.to(torch.int32).numpy()
y_pr_tr = model.predict(X_train).numpy()
y_pr_ts  = model.predict(X_test).numpy()
report_train = classification_report(y_tr, y_pr_tr)
report_test  = classification_report(y_ts, y_pr_ts)
print(report_train)
print(report_test)
```

Logistic regression: Neural network logistic regression

By adding a sigmoid activation function and using the binary cross entropy loss we can perform logistic regression with a neural network.




```
>>               precision    recall  f1-score   support
>>
>>            0       0.97      0.94      0.96        69
>>            1       0.95      0.98      0.96        81
>>
>>     accuracy                           0.96       150
>>    macro avg       0.96      0.96      0.96       150
>> weighted avg       0.96      0.96      0.96       150
>>
>>               precision    recall  f1-score   support
>>
>>            0       0.96      0.92      0.94        26
>>            1       0.92      0.96      0.94        24
>>
>>     accuracy                           0.94        50
>>    macro avg       0.94      0.94      0.94        50
>> weighted avg       0.94      0.94      0.94        50
>>
```

Multilayer

Multilayer

Multilayer: Multiple nodes

Power of neural networks come from connecting multiple layers.

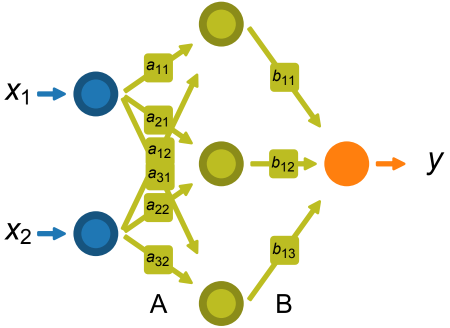
This allows us to build estimators beyond simple linear regression.

This type of network is sometimes called multilayered perceptron.

The inputs to the upper most node in the green, hidden, layer is

$$\hat{y}_1 = a_{11}x_1 + a_{12}x_2$$

(We disregard bias inputs and activation functions for the moment.)



Multilayer: Vector matrix multiplication

The complete output from one layer to the next can be calculated as a vector matrix product.

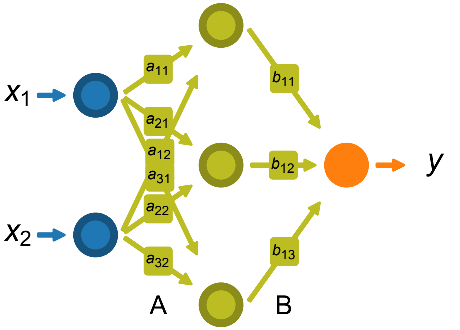
E.g. output from first hidden layer is:

$$\widehat{\mathbf{y}} = A\mathbf{x} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Output from output layer is:

$$y = B\widehat{\mathbf{y}} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \end{pmatrix} \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_2 \end{pmatrix}$$

These matrices represents **connections** between nodes and a **linear transform** of vectors



Multilayer: Matrix multiplication

In a network without activation functions we can pre-multiply the matrices that connects the nodes between each layer.

$$
\begin{aligned}
y &= BA\mathbf{x} \\
&= \begin{pmatrix} b_{11} & b_{12} & b_{13} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\
&= (BA)\mathbf{x} = C\mathbf{x} \\
&= \begin{pmatrix} c_{11} & c_{12} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}
\end{aligned}
$$

```python
import numpy as np
rng = np.random.default_rng(seed=0)
x = rng.integers(1, 10, (2, 1))
A = rng.integers(1, 10, (3, 2))
B = rng.integers(1, 10, (1, 3))
C = B @ A
print(f"Shape of A: {A.shape}")
print(f"Shape of B: {B.shape}")
print(f"Shape of C: {C.shape}")
y = B @ A @ x
z = C @ x
print(y.ravel())
print(z.ravel())
```

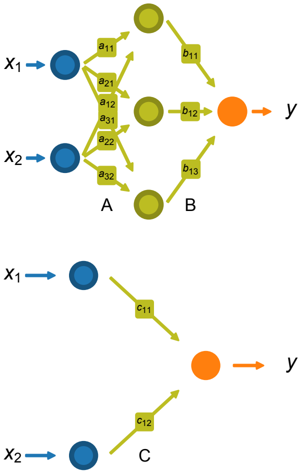
```
>> Shape of A: (3, 2)
>> Shape of B: (1, 3)
>> Shape of C: (1, 2)
>> [440]
>> [440]
```

Multilayer: Layer collapse

Without activation functions the multiple layers will collapse into one, single layer!
Get no benefit from our additional layers!

$$
\begin{aligned}
y &= BA\mathbf{x} \\
&= \begin{pmatrix} b_{11} & b_{12} & b_{13} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\
&= (BA)\mathbf{x} = C\mathbf{x} \\
&= \begin{pmatrix} c_{11} & c_{12} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = c_{11}x_1 + c_{12}x_2
\end{aligned}
$$

The remedy is to add nonlinear activation functions after each hidden layer.



Multilayer: Nonlinear activation functions

If we push the result from each hidden layer through a nonlinear function the complete transform can not be expressed as a single linear operation.

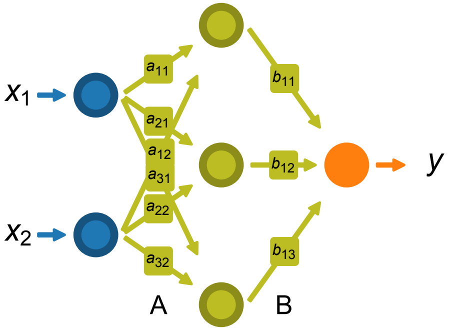
Rather it becomes the composite function

$$\begin{cases} \widehat{\mathbf{y}} = f_A(A\mathbf{x})) \\ y = f_B(B\widehat{\mathbf{y}}) \end{cases}$$

or as a single expression

$$y = f_B(Bf_A(A\mathbf{x}))$$

f_A and f_B can be two different **activation functions**.



Multilayer: Activation functions in pytorch

There are several activation functions with different properties available in pytorch.

All are available as functions in `nn.functional` and as classes in `nn`. They are equivalent and the choice of which one to use is up to the user.

Non-linear activation functions

`threshold`	Apply a threshold to each element of the input Tensor.
`threshold_`	In-place version of `threshold()`.
`relu`	Applies the rectified linear unit function element-wise.
`relu_`	In-place version of `relu()`.
`hardtanh`	Applies the HardTanh function element-wise.
`hardtanh_`	In-place version of `hardtanh()`.
`hardswish`	Apply hardswish function, element-wise.
`relu6`	Applies the element-wise function $\mathrm{ReLU6}(x) = \min(\max(0, x), 6)$.
`elu`	Apply the Exponential Linear Unit (ELU) function element-wise.

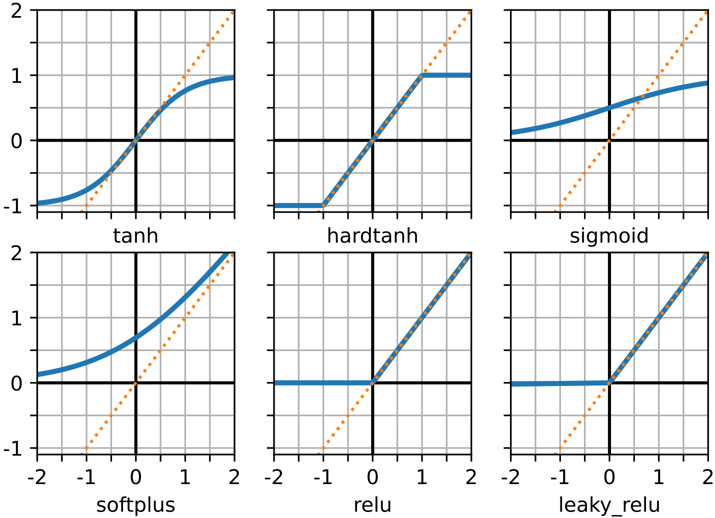
https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions
https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Multilayer: Activation functions

The limited active region of the sigmoid like functions (`tanh`, `hardtanh` and `sigmoid`) make them less popular for hidden layers.
They are useful in the output layer for classification.

Rectifying linear unit, `relu`, and its variants are popular in hidden layers.

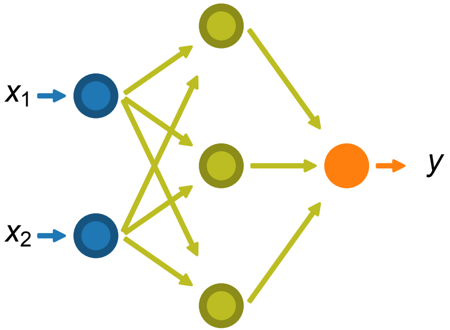
(All have derivatives! Important!)



https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions

Multilayer: Python

```python
import torch
from torch import nn

class MultiLayerModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=2,
                                 out_features=3,
                                 bias=True)
        self.layer_2 = nn.Linear(in_features=3,
                                 out_features=1,
                                 bias=True)
        return

    def forward(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        """ Add an activation function for the
        first, hidden layer.
        No activation function for the output
        layer implies a regression type model.
        """
        X = self.layer_1(X)
        X = nn.functional.relu(X)
        X = self.layer_2(X)
        return X
```



https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions
https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Backpropagation

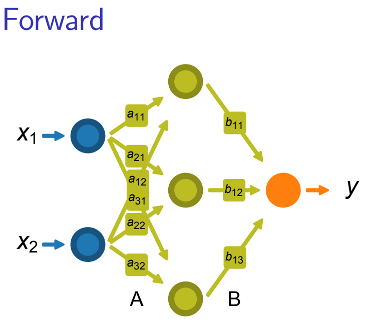
Backpropagation

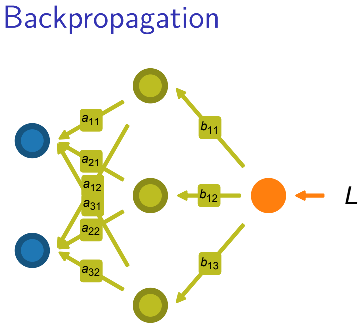
Backpropagation: Optimizing the parameters

The output from the network is calculated by feeding values forward in the network.

The output is compared to the expected output using a loss function.

The loss, L, is then backpropagated through the network and the weights are updated.





Backpropagation: Gradient chain rule

The weights are optimized using **gradient descent**. Look for region with zero gradient!

The **gradient of the loss** with respect to each weight has to be calculated. Use **chain rule**.

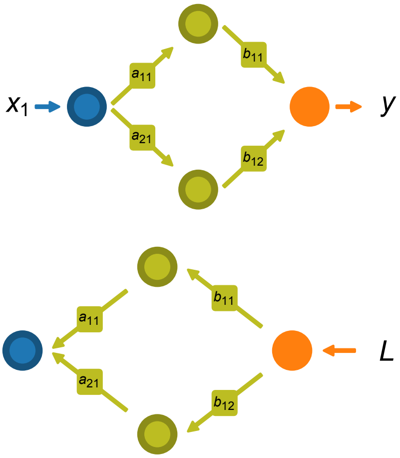
$$\frac{\partial L}{\partial b_{11}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b_{11}}$$

$$\frac{\partial L}{\partial b_{12}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b_{12}}$$

$$\frac{\partial L}{\partial a_{11}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \widehat{\mathbf{y}}_1}\frac{\partial \widehat{\mathbf{y}}_1}{\partial a_{11}}$$

$$\frac{\partial L}{\partial a_{21}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \widehat{\mathbf{y}}_2}\frac{\partial \widehat{\mathbf{y}}_1}{\partial a_{21}}$$

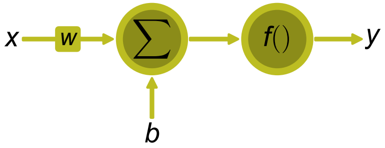
This is why the loss function and all activation functions need to be differentiable!

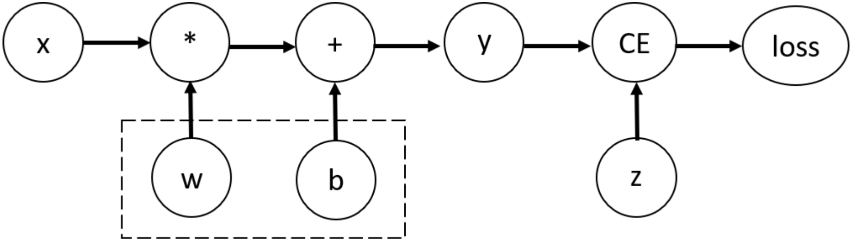


Backpropagation: Automatic differentiation

Keeping track of all these derivatives requires a lot of effort!

Pytorch does this for us by **storing** all **operations** in a **graph**.
Gradients are then **computed** using **automatic differentiation**.

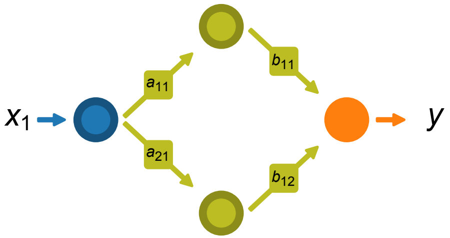


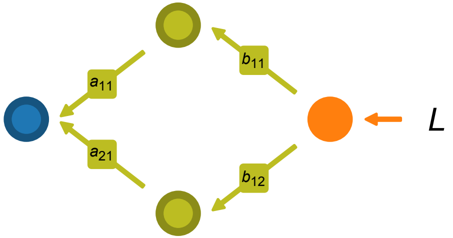


https://pytorch.org/tutorials/beginner/basics/autograd_tutorial.html

Backpropagation: In pytorch

```
## Model instantiating and settings for training
from torch import optim
epochs = 20
loss_arr = np.zeros(epochs)
model = LinearRegressionModel()
loss_func = nn.MSELoss()

## The optimizer is instatiated with the parameters
## of the model. Learning rate determines how large
## steps to take in the direction of gradient
## descent. Stochastic Gradient Descent (SGD)
optimizer = optim.SGD(model.parameters(), lr=0.2)

## Train the linear regression network
for i in range(epochs):
    ## Reset optimizer
    optimizer.zero_grad()
    ## Compute output from model and loss
    y_pred = model(X_train)
    loss = loss_func(y_pred, y_train)
    ## Loss is propagated backwards in network
    loss.backward()
    ## Step in the direction of gradient descent
    optimizer.step()
    ## Store computed loss for plotting
    loss_arr[i] = loss.data.numpy()
```





Pytorch

Pytorch

Pytorch: Tensors

Torch tensors are similar to numpy arrays.
Often numerical type, **int** or **float**.

Can have as many dimensions as required but commonly 1D, 2D or 3D.

1D Vector

2D Matrix

ND "Tensor"

To use **torch** it is standard to **import** it:
`import torch.`

```python
import torch
a = torch.tensor([1, 6, 8])
b = torch.tensor([[1.0, 6.0],
                  [5.0, 3.0]])
print(type(a))
print(a)
print(b)
```

```
>> <class 'torch.Tensor'>
>> tensor([1, 6, 8])
>> tensor([[1., 6.],
>>         [5., 3.]])
```

Pytorch: Numpy interoperability

Torch tensors can be created from numpy arrays.
They can also be converted back to numpy arrays.

Torch expects tensors! Other packages might expect numpy arrays!
Make sure you have the correct type!

```python
import numpy as np
import torch
import torch.nn
a_np = np.array([1.0, 6.0, 8.0])
a = torch.tensor(a_np)
b = torch.tensor([[1.0, 6.0],
                  [5.0, 3.0]])
b_np = b.numpy()
print(type(a_np))
print(type(a))
print(b)
print(b_np)

loss_func = torch.nn.MSELoss()
print(f"Loss: {loss_func(a, a)}")
try:
    print(f"Loss: {loss_func(a_np, a_np)}")
except Exception as e:
    print(e)
```

```
>> <class 'numpy.ndarray'>
>> <class 'torch.Tensor'>
>> tensor([[1., 6.],
>>         [5., 3.]])
>> [[1. 6.]
>>  [5. 3.]]
>> Loss: 0.0
>> 'int' object is not callable
```

Pytorch: Operations

Torch contains many of the same functions for operating on tensors as numpy does for operating on arrays.

Creation	a = `torch.ones((2,3))`
Operations	*E.g.* `C = B @ A`
Aggregation	*E.g.* `torch.amax(C)`

Torch also contains many special functions, *e.g.* the activation and the loss functions.

```
import numpy as np
import torch
rng = np.random.default_rng(seed=0)
A = torch.tensor(rng.integers(1, 10, (2,3))).float()
x = torch.tensor(rng.integers(1, 10, (3,1))).float()
b = A @ x
print(A)
print(x)
print(b)
print(f"Max in A: {torch.amax(A)}")
```

```
>> tensor([[8., 6., 5.],
>>         [3., 3., 1.]])
>> tensor([[1.],
>>         [1.],
>>         [2.]])
>> tensor([[24.],
>>         [ 8.]])
>> Max in A: 8.0
```

Pytorch: Computational graph of gradients

A difference between torch tensors and numpy array is that the tensors can contain a computational graph.

This is used when optimizing the neural networks with backpropagation and gradient descent.

```
import numpy as np
import torch
x = torch.tensor([3.0, 4.0],
                 requires_grad=True)
w = torch.tensor([5.0], requires_grad=True)
a = w * x
print("The variable will keep track of how to
     compute its own gradient!")
print(a)

## Turn of gradient tracking and convert to
     numpy
a = w.detach().numpy() * x.detach().numpy()
print("Numpy does not do this.")
print(a)
```

```
>> The variable will keep track of how to
     compute its own gradient!
>> tensor([15., 20.], grad_fn=<MulBackward0>)
>> Numpy does not do this.
>> [15. 20.]
```

Pytorch: Particularities

Many torch functions used for deep learning expects the tensors to be of either float32 or integer32 type.

Can be source of exception!

Can easily convert to these types with `a = a.float()` or `a = a.integer()`.

```python
import numpy as np
import torch
import torch.nn
x_np = np.array([1.0, 6.0, 8.0], dtype=np.
    float64)
x = torch.tensor(x_np)

print("Before converting to float:")
layer = torch.nn.Linear(3, 1)
try:
    y = layer(x)
    print(f"y: {y}")
except Exception as e:
    print(e)
print("\nAfter converting to float:")
x = x.float()
y = layer(x)
print(f"y: {y}")
```

```
>> Before converting to float:
>> mat1 and mat2 must have the same dtype, but
     got Double and Float
>>
>> After converting to float:
>> y: tensor([-3.8239], grad_fn=<ViewBackward0
    >)
```

Pytorch: GPU

Pytorch has support for computing on GPU (graphics processing unit).

More on that tomorrow.

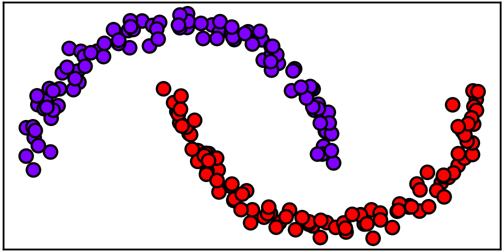
Single class

Single class

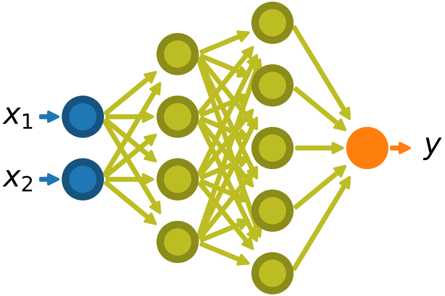
Single class: Nonlinear classification

We have previously explored how linear and nonlinear classification and clustering methods behave on nonlinear test data.

We have observed how linear methods fail since they place a line or plane that attempts to split the data.



A multilayered, nonlinear, neural network should be able to classify nonlinear data.

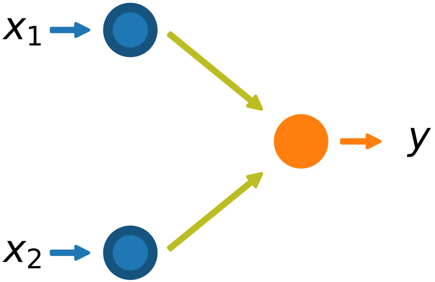


Single class: Python linear classifier

```python
class LinearClassificationModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.layers = nn.Linear(in_features=2,
                                out_features=1,
                                bias=True)
        return

    def forward(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        X = self.layers(X)
        return nn.functional.sigmoid(X)

    def predict(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        with torch.inference_mode():
            y: torch.Tensor = self(X)
            y = torch.round(y)
        return y.to(torch.int32)
```

The linear classifier contains a single layer that connects the two inputs to the single output.

A `sigmoid` activation function is used for the output layer.



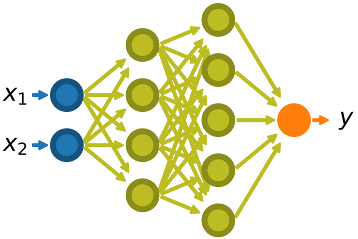
Single class: Python nonlinear classifier

```python
class NonLinearClassificationModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(in_features=2,
                                out_features=4,
                                bias=True)
        self.layer2 = nn.Linear(in_features=4,
                                out_features=5,
                                bias=True)
        self.layer3 = nn.Linear(in_features=5,
                                out_features=1,
                                bias=True)
        return

    def forward(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        X = nn.functional.leaky_relu(self.layer1(X))
        X = nn.functional.leaky_relu(self.layer2(X))
        X = nn.functional.leaky_relu(self.layer3(X))
        return nn.functional.sigmoid(X)

    def predict(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        with torch.inference_mode():
            y: torch.Tensor = torch.round(self(X))
        return y.to(torch.int32)
```

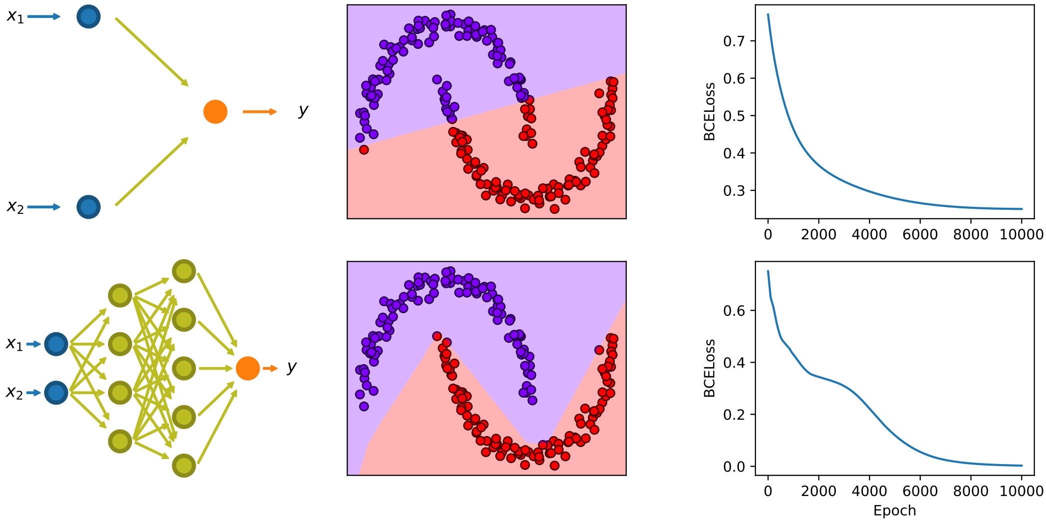
The nonlinear classifier contains several layers between the two inputs to the single output.

`leaky_relu` activation functions are used for the hidden layers.

A `sigmoid` activation function is used for the output layer.



Single class: Linear and nonlinear classifiers



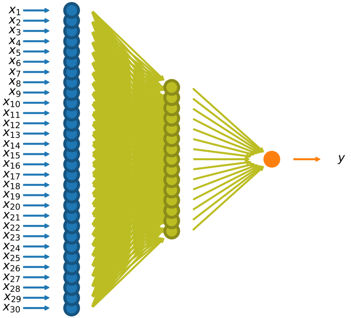
Single class: Breast cancer classifier

```python
class BreastCancerClassificationModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=30,
                                 out_features=15,
                                 bias=True)
        self.layer_2 = nn.Linear(in_features=15,
                                 out_features=1,
                                 bias=True)
        return

    def forward(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        X = self.layer_1(X)
        X = nn.functional.relu(X)
        X = self.layer_2(X)
        X = nn.functional.sigmoid(X)
        return X

    def predict(
        self, X: torch.Tensor,
    ) -> torch.Tensor:
        with torch.inference_mode():
            y: torch.Tensor = self(X)
            y = torch.round(y)
        y = y.to(torch.int32)
        return y
```

Inherit from `nn.Module` base class.

Should have 30 inputs and 1 output.

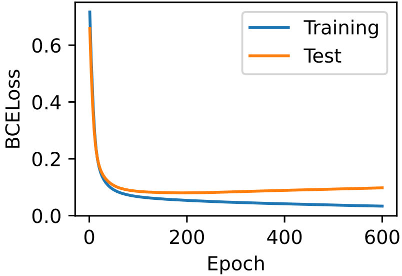
Use `sigmoid` activation function in output layer and `BCELoss()`.



Single class: Breast cancer results

We can get very good classification results with this relatively simple network.

(As we have seen before the breast cancer data should be relatively easy to classify!)




```
>> Failed to classify: 8 of 569 (1.4%)
>> ## Performance on training set:
>>               precision    recall  f1-score   support
>>
>>            0       1.00      0.98      0.99       165
>>            1       0.99      1.00      0.99       290
>>
>>     accuracy                           0.99       455
>>    macro avg       0.99      0.99      0.99       455
>> weighted avg       0.99      0.99      0.99       455
>>
>> ## Performance on test set:
>>               precision    recall  f1-score   support
>>
>>            0       0.96      0.96      0.96        47
>>            1       0.97      0.97      0.97        67
>>
>>     accuracy                           0.96       114
>>    macro avg       0.96      0.96      0.96       114
>> weighted avg       0.96      0.96      0.96       114
>>
```

Multi-class

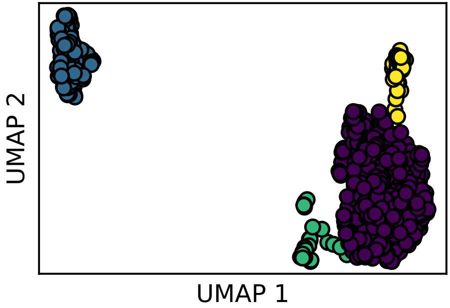
Multi-class

Multi-class: Omics data classifier

The cancer associated fibroblast data contains four classes.

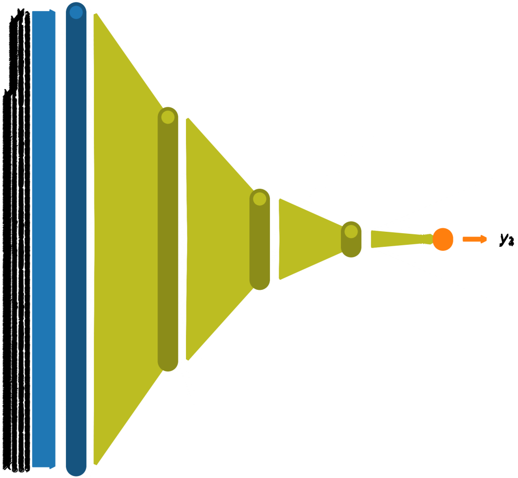
We hypothesize that we should be able to create a nonlinear classifier for this data since we can separate the classes using UMAP.

The data has 557 features so our network should have this many inputs.




```
>> Shape of data matrix X:
>> (716, 557)
```

Multi-class: Network



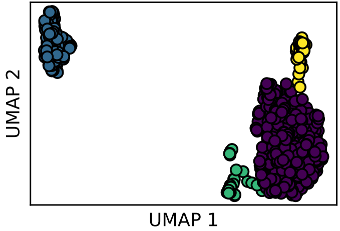
Multi-class: Python class

```python
class CAFClassificationModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features=557,
                                 out_features=300,
                                 bias=True)
        self.layer_2 = nn.Linear(in_features=300,
                                 out_features=100,
                                 bias=True)
        self.layer_3 = nn.Linear(in_features=100,
                                 out_features=20,
                                 bias=True)
        self.layer_4 = nn.Linear(in_features=20,
                                 out_features=4,
                                 bias=True)
        return

    def forward(
        self,
        X: torch.Tensor,
    ) -> torch.Tensor:
        X = nn.functional.relu(self.layer_1(X))
        X = nn.functional.relu(self.layer_2(X))
        X = nn.functional.relu(self.layer_3(X))
        X = self.layer_4(X)
        X = nn.functional.log_softmax(X, dim=1)
        return X
```

Inherit from `nn.Module` base class.

Should have 557 inputs and 4 outputs.

Output layer activation function depends on the loss function used. Use `NLLLoss()` and `log_softmax()`.



The model should be able to classify the four different types in the CAF data.

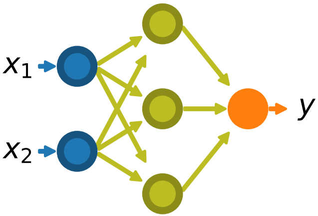
Multi-class: Loss function

Single binary output

Binary cross entropy

Note that the targets y should be numbers between 0 and 1.

```
from torch import nn
loss_func = nn.BCELoss()
```

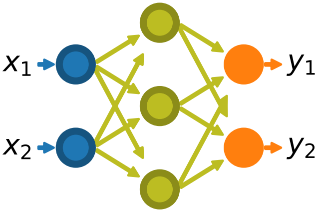


Multiple class output

Cross entropy

The input is expected to contain the unnormalized logits for each class.

```
from torch import nn
loss_func = nn.CrossEntropyLoss()
```

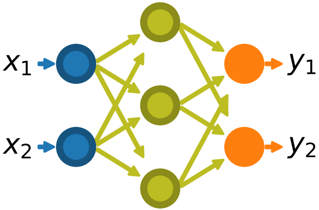


Multiple class output

Negative log likelihood

The input given through a forward call is expected to contain log-probabilities of each class.

```
from torch import nn
loss_func = nn.NLLLoss()
```



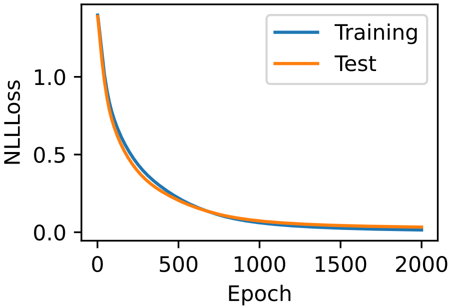
https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html
https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html
https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
https://pytorch.org/docs/stable/nn.html#loss-functions

Multi-class: Model training

```
## Load data
X, y = common.load_CAF()
X = np.log2(X + 1)
y = y - 1
y = torch.tensor(y).long()
X = torch.tensor(X).float()
X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=0.30,
                         random_state=2)
## Create model, optimizer and loss function
model = CAFClassificationModel()
optimizer = optim.SGD(model.parameters(),
                      lr=0.001, momentum=0.5)
loss_func = nn.NLLLoss()
## Train the model
epochs = 2000
loss_train_arr = np.zeros(epochs)
loss_test_arr = np.zeros(epochs)
for i in range(epochs):
    optimizer.zero_grad()
    y_pred = model(X_train)
    loss = loss_func(y_pred, y_train)
    loss.backward()
    optimizer.step()
    loss_train_arr[i] = loss.data.numpy()
    ## Calculate loss on validation set
    y_pred = model.predict_proba(X_test)
    loss = loss_func(y_pred, y_test)
    loss_test_arr[i] = loss.data.numpy()
```

Train the model using **NLLLoss** and stochastic gradient descent.

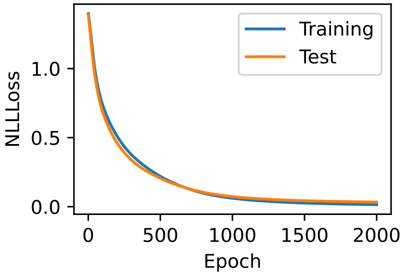
Can make use of sklearn to `train_test_split` the data.



Multi-class: Results

We can get very good classification results with this relatively simple network.

(As we suspected from the UMAP the data should be relatively easy to classify!)




```
>> Failed to classify: 3 of 716 (0.4%)
>> ## Performance on training set:
>>               precision    recall  f1-score   support
>>
>>            0       1.00      1.00      1.00       339
>>            1       1.00      1.00      1.00       101
>>            2       1.00      1.00      1.00        30
>>            3       1.00      1.00      1.00        31
>>
>>     accuracy                           1.00       501
>>    macro avg       1.00      1.00      1.00       501
>> weighted avg       1.00      1.00      1.00       501
>>
>> ## Performance on test set:
>>               precision    recall  f1-score   support
>>
>>            0       0.98      1.00      0.99       151
>>            1       1.00      1.00      1.00        42
>>            2       1.00      0.91      0.95        11
>>            3       1.00      0.82      0.90        11
>>
>>     accuracy                           0.99       215
>>    macro avg       1.00      0.93      0.96       215
>> weighted avg       0.99      0.99      0.99       215
>>
```

Conclusions

Conclusions

Conclusions: Neural networks in sklearn

Sklearn also contains a neural networks module for regression and classification.

Get the benefit of the simpler sklearn interface.
But with less control over the details.

Has only multi-linear layered type networks.
Does not support computations on GPU.

Today we only looked at multi-linear layered networks and did not use GPU!

sklearn.neural_network

Models based on neural networks.

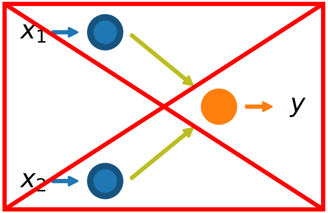
User guide. See the Neural network models (supervised) and Neural network models (unsupervised) sections for further details.

`BernoulliRBM`	Bernoulli Restricted Boltzmann Machine (RBM).
`MLPClassifier`	Multi-layer Perceptron classifier.
`MLPRegressor`	Multi-layer Perceptron regressor.

https://scikit-learn.org/stable/modules/neural_networks_supervised.html
https://scikit-learn.org/stable/api/sklearn.neural_network.html
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

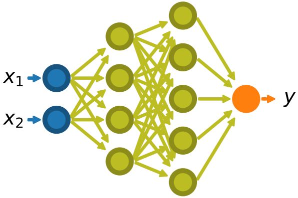
Conclusions: Neural networks

Feed-forward, multi-layered perceptron, type neural networks can be used for nonlinear regression and classification.
For linear regression/logistic regression use modules in sklearn!



Model implementation done by inheriting from `nn.Module`.

Loss function must fit the task.
Training performed in a loop using back propagation and gradient descent.



The examples shown in this lecture were all computed on a laptop without GPU.

https://pytorch.org

Conclusions: Pytorch vs sklearn

https://xkcd.com/1838/

scikit learn

PyTorch