IntroToNeuralNetworks - BuildingNeuralNetworks - 3

One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

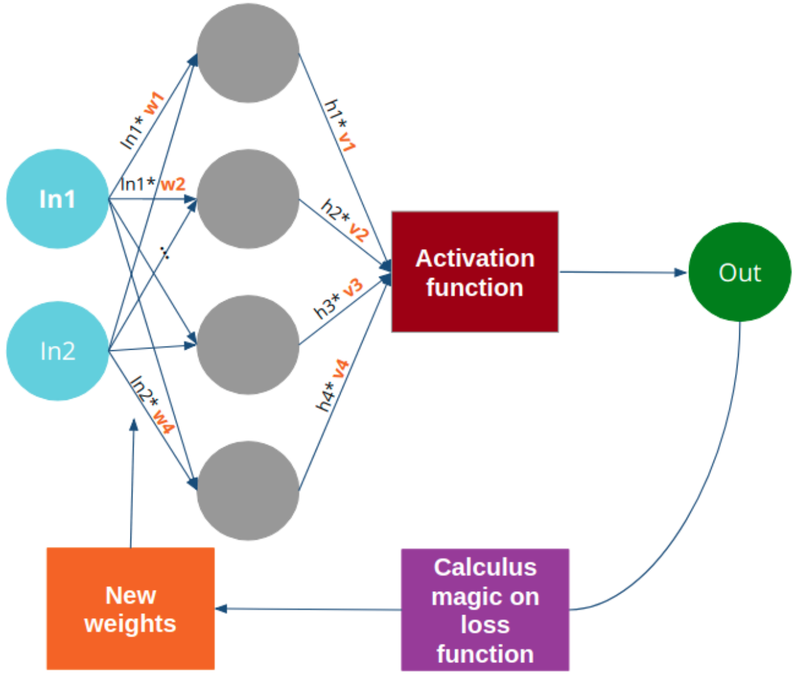
Objective	Complete
Introduce activation functions and their types	
Visualize training history accuracy and loss	

Warm up: Neural network playground

- Explore and play with a neural network by visiting the website ***Playground.tensorflow.org***
- Create different data scenarios and outputs with the program
- Chat question: What happens to the **output** as the features, layers, and neurons are manipulated?

Activation function: which one to choose?

- **Activation functions are essentially filters**
- They take the input from neurons and filter some out so that only the ones that matter fire
- Which activation function may be needed depends on the problem and your data
- One of the things we want to know about an activation function is the **range of values it generates** because it determines the values you can expect at the next stage of the process

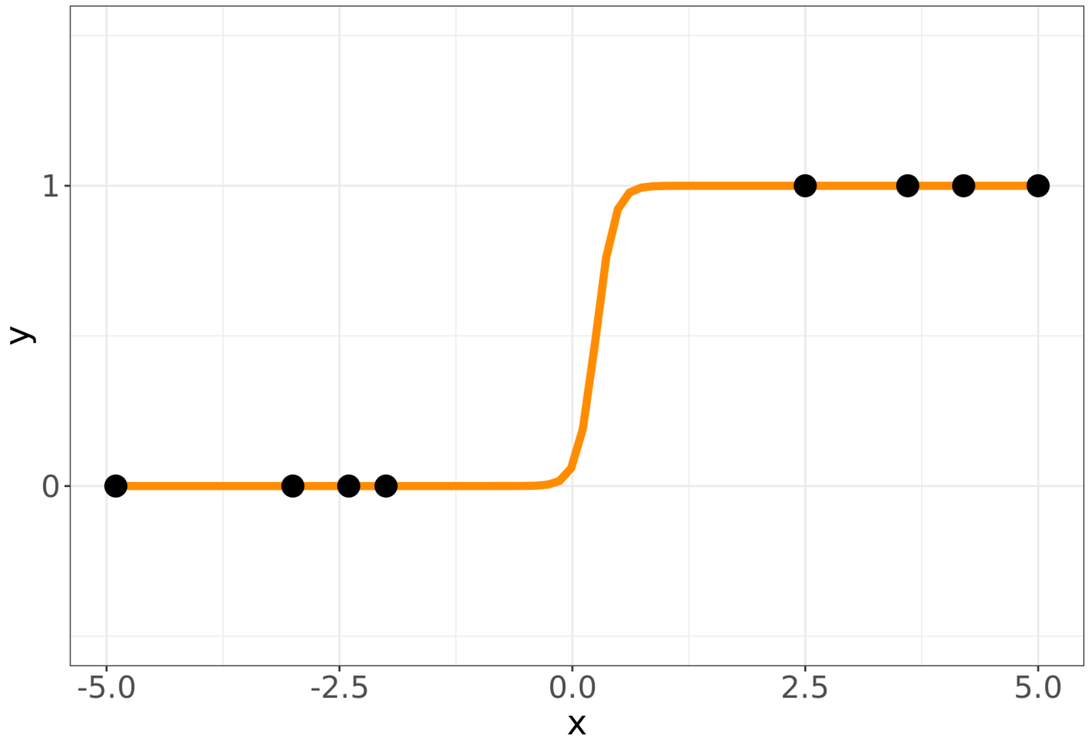


Common activation functions: sigmoid

- **Sigmoid** is a function commonly used to get activation function outputs between `0` and `1`
- It is also the activation function used for all **binary classification problems at the output layer**
- Binary classification problems involve classifying an observation as one of 2 possible categories, for example:
  - `Yes` or `No`
  - `True` or `False`
  - `0` or `1`
  - `High` or `Low`

$$p(y = 1) = \frac{exp(ax + b)}{1 + exp(ax + b)}$$

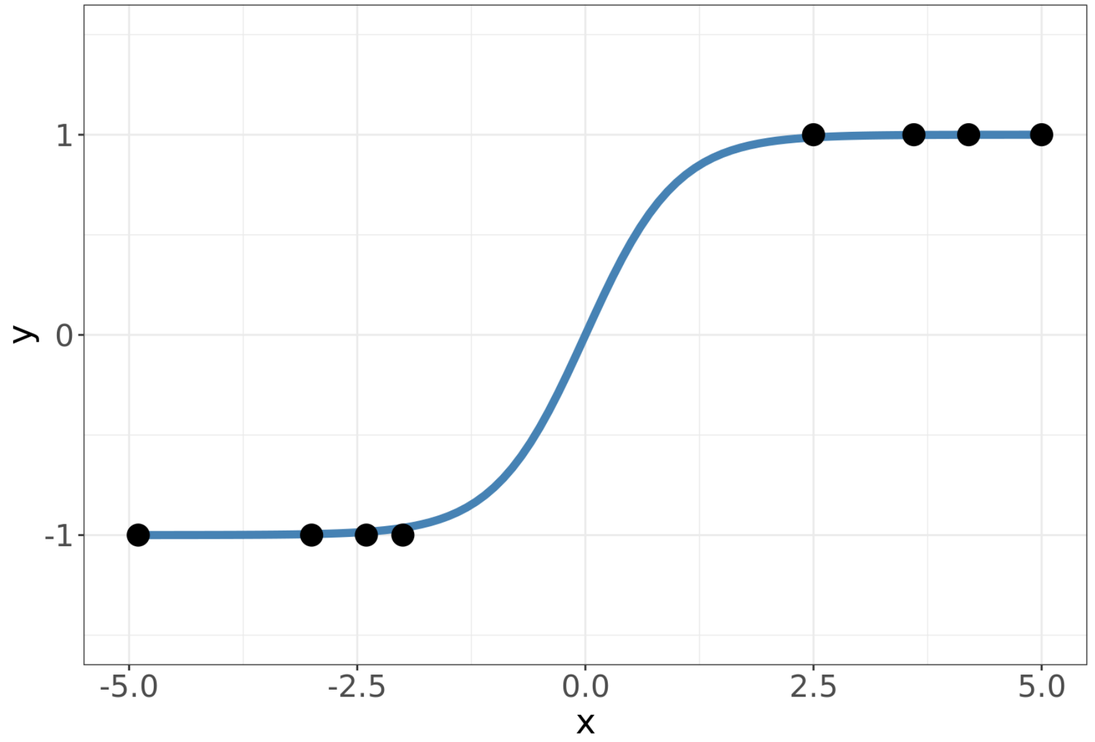
```
Warning: glm.fit: fitted probabilities
numerically 0 or 1 occurred
```



Common activation functions: tanh

- **Hyperbolic tangent** (tan) is an activation function commonly used to get outputs between -1 and 1
- It is a ratio of hyperbolic sine to hyperbolic cosine

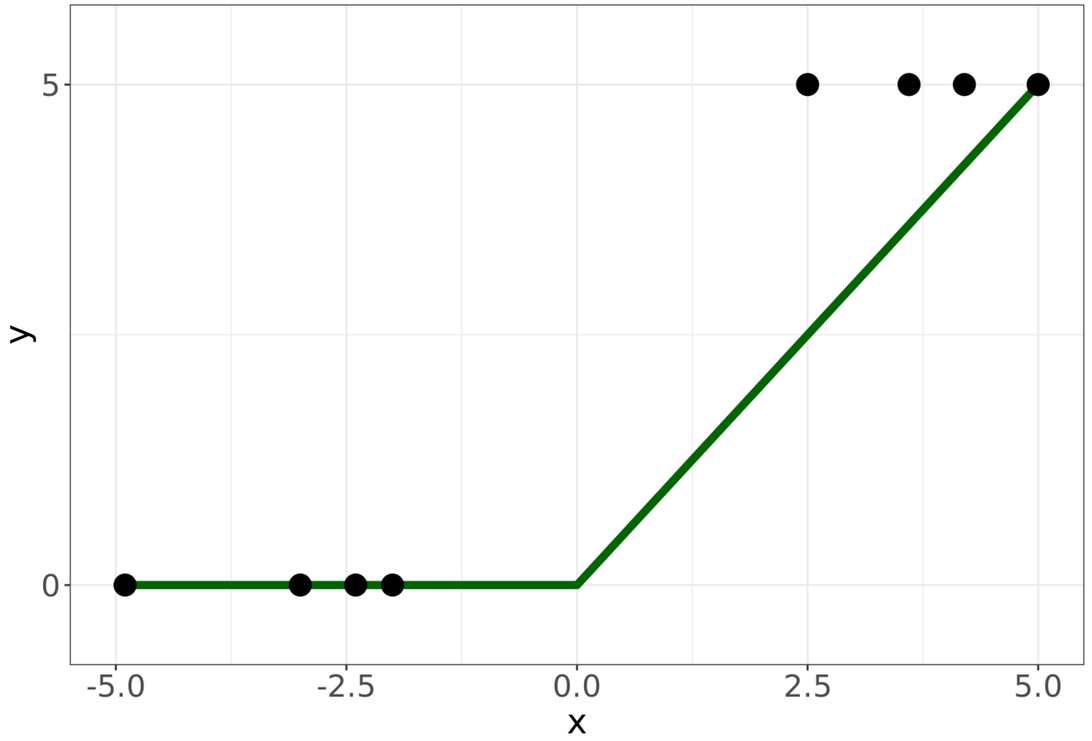
$$tanh(x) = \frac{sinh(x)}{cosh(x)}$$



Common activation functions: ReLU

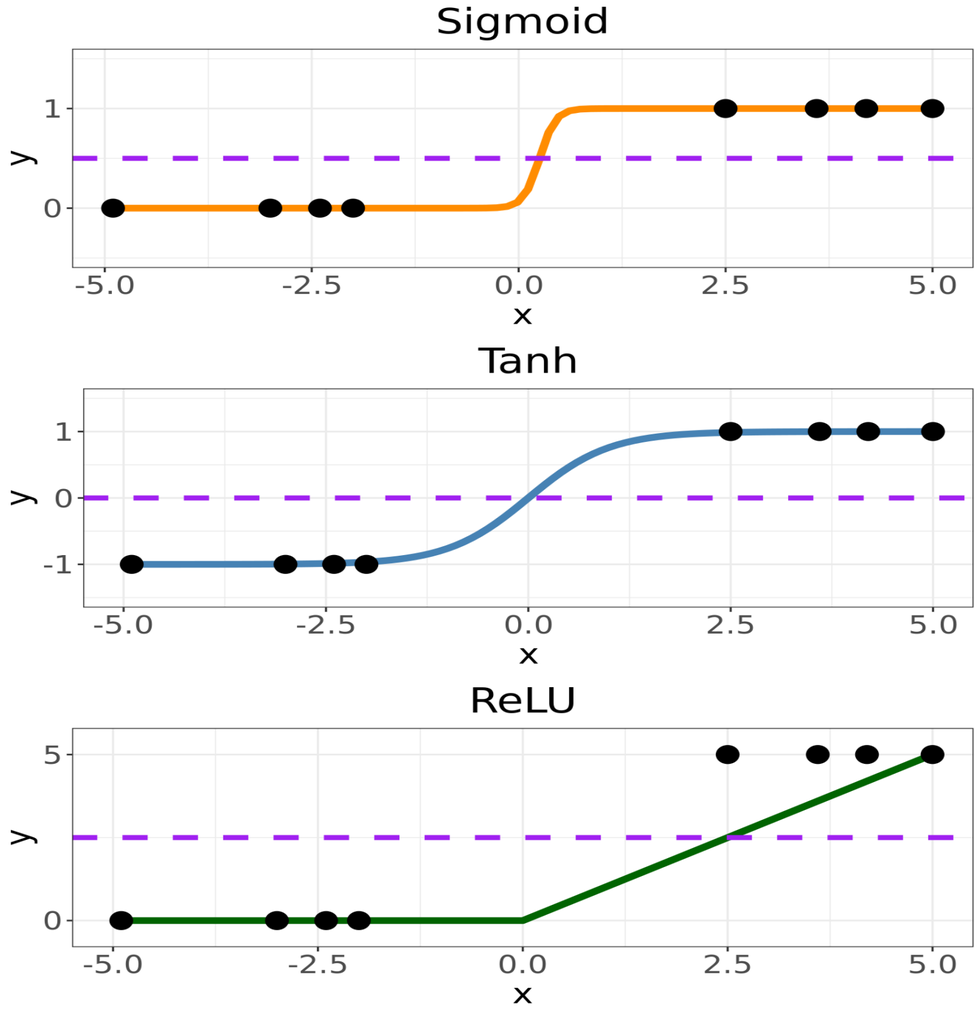
- **Re**ctified **L**inear **U**nit (**ReLU**) is an activation function, commonly used to get outputs between `0` and any positive value
- It takes a maximum between 0 and a given number x, which means that it drives all negative inputs to 0 or returns the value x otherwise
- This is one of the most popular hidden layer activation functions to date
- Using `ReLU` in lower layers (those close to the input layer) helps mitigate the problem of vanishing gradients

$$R(x) = max(0, x)$$



Activation function: a threshold

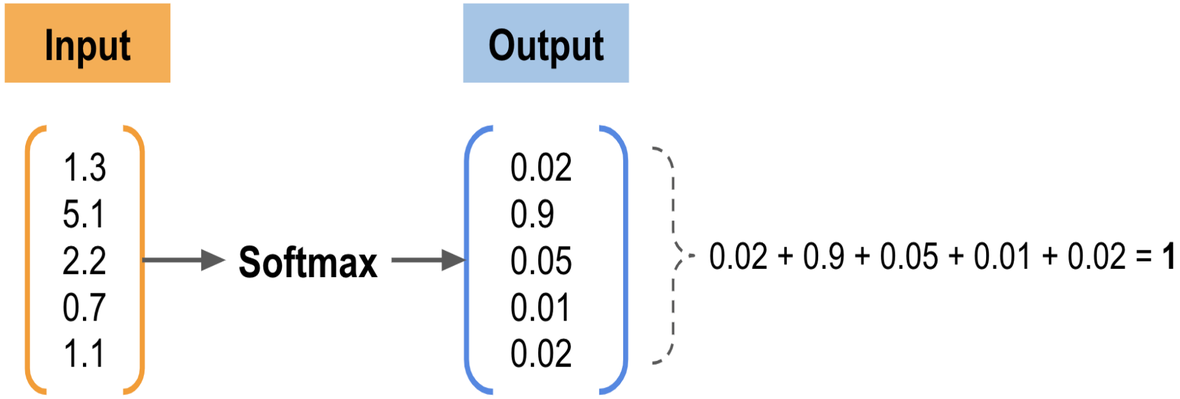
- Threshold is the cut off value of the function
- What makes the functions similar to classification problems is that they all take a **threshold** value:
  - If the value fired by a neuron and "filtered" through our activation function is **below** that threshold, it is assigned to one class (e.g., 0 for sigmoid, -1 for tanh, or 0 for ReLU)
  - If the value fired by a neuron and "filtered" through our activation function is **above** that threshold, it is assigned to another class (e.g., 1 for sigmoid, 1 for tanh or *any positive number* for ReLU)



Common activation functions: Softmax

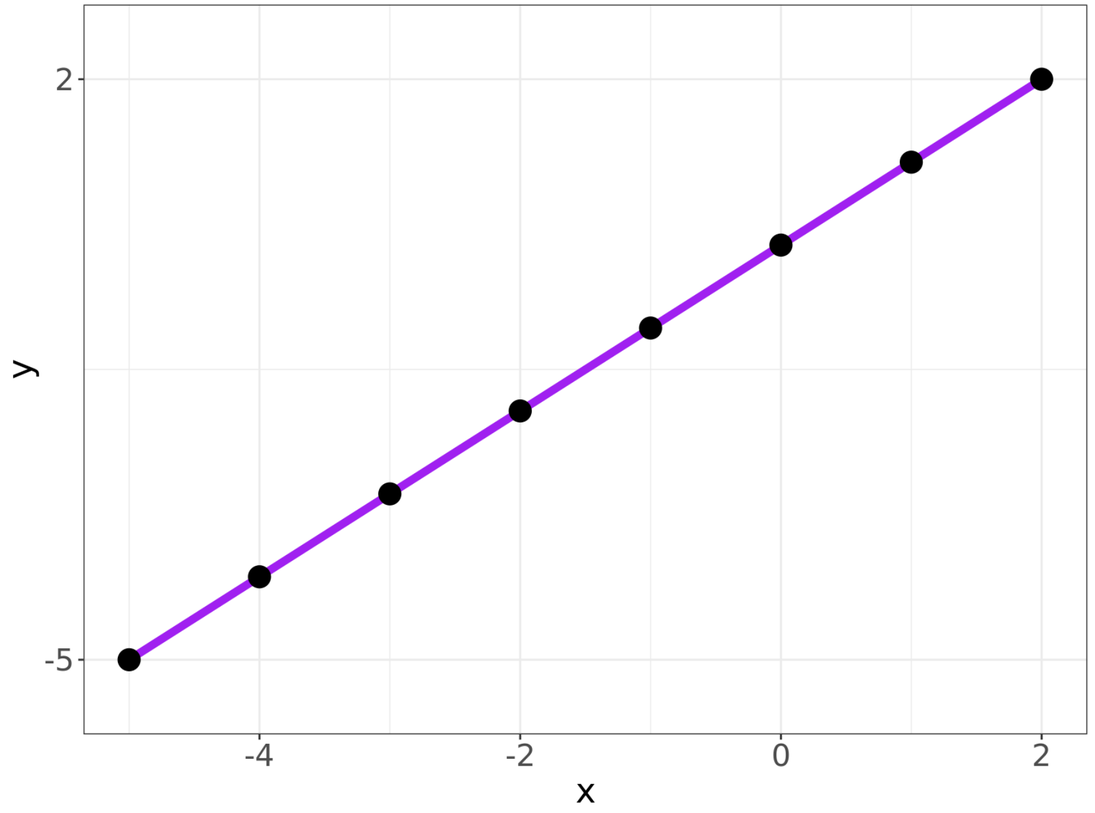
- **Softmax** is commonly used for **multi-class classification problems** at the *output* layer
- It takes an input vector, normalizes it and converts it to a vector of categorical probabilities
- The sum of all the probabilities in the output vector would be equal to `1`

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$



Common activation functions: linear

- The **linear** activation function outputs the same value as the input (i.e., $f(x) = x$)
- It is also know as the “pass-through” activation, because it lets every input pass through to the output without modification
- It is used in **regression** problems and sometimes in time series and sequence modeling problems



Module completion checklist

Objective	Complete
Introduce activation functions and their types	✔
Visualize training history accuracy and loss	

Accuracy and loss of the model

- Now that we know more about how a neural network works let's inspect the accuracy and loss of our model

Fitting & visualizing training history: accuracy

- Initialize the classifier and set up variables and constants

```
N_EPOCHS = 25                          #<- number of epochs
N_CLASSES = np.unique(y_train)     #<- number of classes in the target variable

# Build neural network model by creating a classifier:
# add the number of hidden neurons in the 1st hidden layer and set random state.
mlp = MLPClassifier(hidden_layer_sizes = (64), random_state = 1)

scores_train = [] #<- we will store scores for training history here
scores_test = []  #<- we will store scores for testing history here

epoch = 0 #<- set epoch count
```

- Loop through each epoch evaluating the model and appending results to the score lists

```
while epoch < N_EPOCHS:
    mlp_fit = mlp.partial_fit(X_train_scaled, y_train, classes=N_CLASSES)

    # Compute score for train data.
    scores_train.append(mlp.score(X_train_scaled, y_train))

    # Compute score for test data.
    scores_test.append(mlp.score(X_test_scaled, y_test))
    epoch += 1 #<- increment the epoch
```

Inspect model accuracy and loss

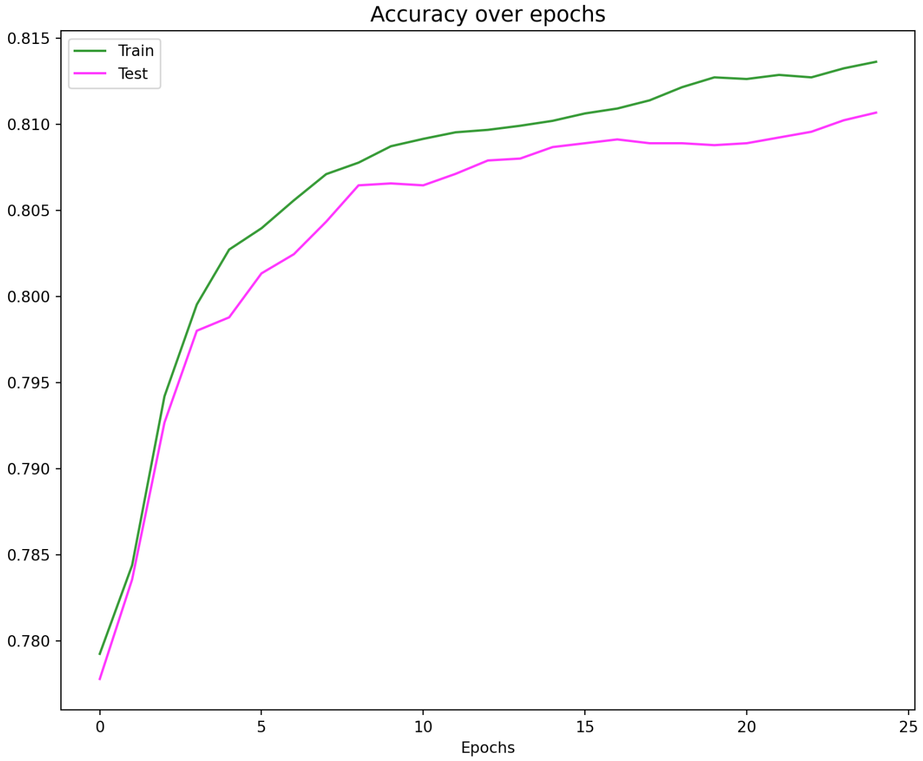
- Plot model accuracy

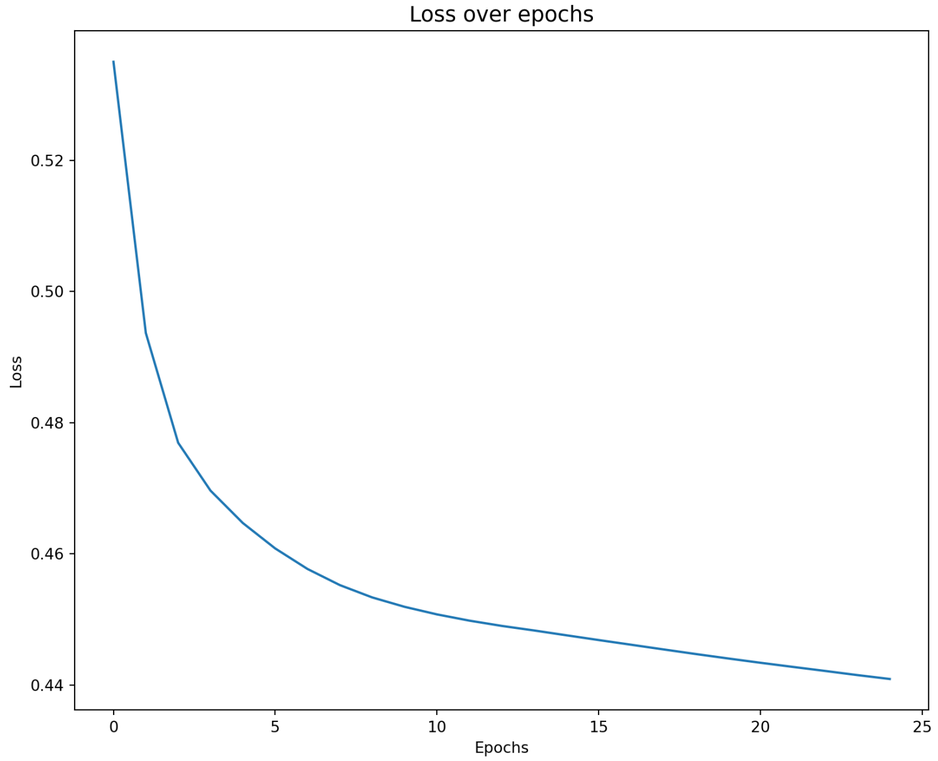
```
plt.plot(scores_train, color='green', alpha=0.8,
label='Train')
plt.plot(scores_test, color='magenta', alpha=0.8,
label='Test')
plt.title("Accuracy over epochs", fontsize=14)
plt.xlabel('Epochs')
plt.legend(loc='upper left')
plt.show()
```

- Plot model loss

```
plt.plot(mlp.loss_curve_)
plt.title("Loss over epochs", fontsize=14)
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.show()
```

Inspect model accuracy and loss (cont'd)





Inspect model accuracy and loss (cont'd)

- These plots show some important things:
  - the accuracy scores for the model for each epoch **train** and **test** data (*Note: there is a small gap between them!*)
  - the test accuracy is slightly lower than train accuracy, as expected
  - the trend of accuracy is generally increasing for both **train** data and **test** data
  - the loss is decreasing with every epoch
- Given the simplicity of the model and its implementation, these results are fairly decent

Working with imbalanced data

- Our classifier did a decent job without target-balancing our data
- However, if you ever need to target balance the dataset, you can use the following slides as your guide

Class-imbalanced dataset challenges

- Check the value frequency of the categorical target variable

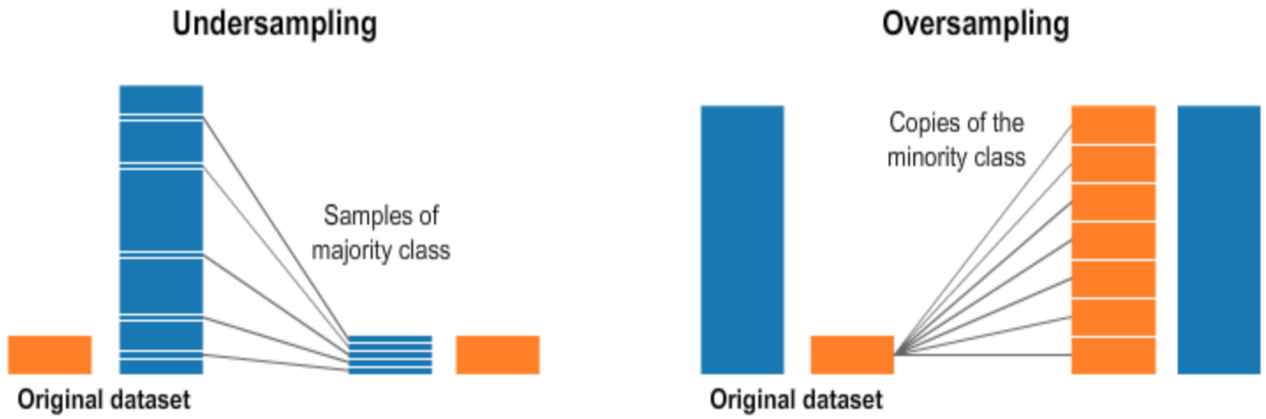
```
print(credit_card['default_payment_next_month'].value_counts())
```

```
0    23364
1     6636
Name: default_payment_next_month, dtype: int64
```

- It has **two levels**, `0` and `1`, where `0` is cardholders who **did not** default on a payment
- We can see that the target is somewhat **imbalanced** as it has a disproportionate ratio of observations in each class
- Imbalanced datasets lead to weak predictive power in a model
- Because ML models aim to maximize accuracy and reduce error, they will always predict the “majority class”

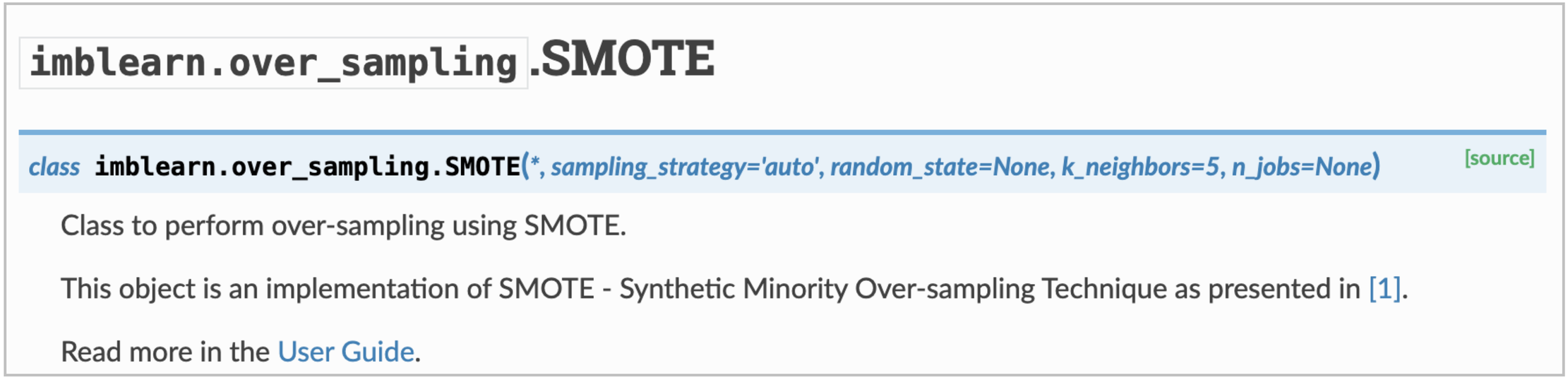
Using oversampling to balance target

- There are multiple ways to balance the target, each of which has pros and cons
- We will use the **Synthetic Minority Oversampling Technique (SMOTE)**
  - It consists of synthesizing elements for the minority class based on existing data points
  - It randomly picks a point from the minority class and computes the k-nearest neighbors for this point
  - The synthetic points are then added between the chosen point and its neighbors



Balancing target with SMOTE

- Let's use `SMOTE` to balance the target and save the objects as `X_train_sm` and `y_train_sm`
- We will use a module `over_sampling` from the package `imblearn`, which contains the `SMOTE` method



imblearn.over_sampling.SMOTE

class **imblearn.over_sampling.SMOTE**(**, sampling_strategy='auto', random_state=None, k_neighbors=5, n_jobs=None*) [source]

Class to perform over-sampling using SMOTE.

This object is an implementation of SMOTE - Synthetic Minority Over-sampling Technique as presented in [1].

Read more in the User Guide.

- For more information, refer to package ***documentation***

Balancing target with SMOTE (cont'd)

```
from imblearn.over_sampling import SMOTE

# Let's initialize SMOTE object.
smote = SMOTE()

# We can now fit the sampling method to our train data and labels.
X_train_sm, y_train_sm = smote.fit_resample(X_train_scaled, y_train)
print(y_train_sm.value_counts())
```

```
0    16364
1    16364
Name: default_payment_next_month, dtype: int64
```

- We can now see that the targets are balanced in our training data and ready for data modeling!

- **Note:** We **do not** target balance the test data, because our test dataset should mimic the new unlabeled data we would get in a real life use case

Additional reading reference

- If you want to dig deeper into some of the concepts we've learned about, here are links to some additional material:
  - ***On the Impact of the Activation Function on Deep Neural Networks Training***: This paper goes into detail about the importance of activation functions and hyperparameter tuning–not just to improve model performance but also to accelerate training
  - ***Overview of Gradient Descent Optimization Algorithms***: This paper goes into detail about gradient descent optimization algorithms and where they would be most appropriate to use
  - ***Disney uses Deep Neural Networks to render clouds***: This article shows how Disney used deep learning to synthesize images of realistic-looking clouds for their movies

Knowledge check

Exercise

You are now ready to try tasks 14-16 in the Exercise for this topic

Module completion checklist

Objective	Complete
Introduce activation functions and their types	✔
Visualize training history accuracy and loss	✔

Building Neural Networks: Topic summary

In this part of the course, we have covered:

- Create a basic neural network model
- Evaluate models using various performance metrics
- Visualize accuracy and loss

Congratulations on completing this module!