DATA SOCIETY:

IntroToNeuralNetworks - BuildingNeuralNetworks - 2

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

Module completion checklist

Objective	Complete
Implement and evaluate a simple neural network using MLPClassifier	
Explain the concept of backpropagation	

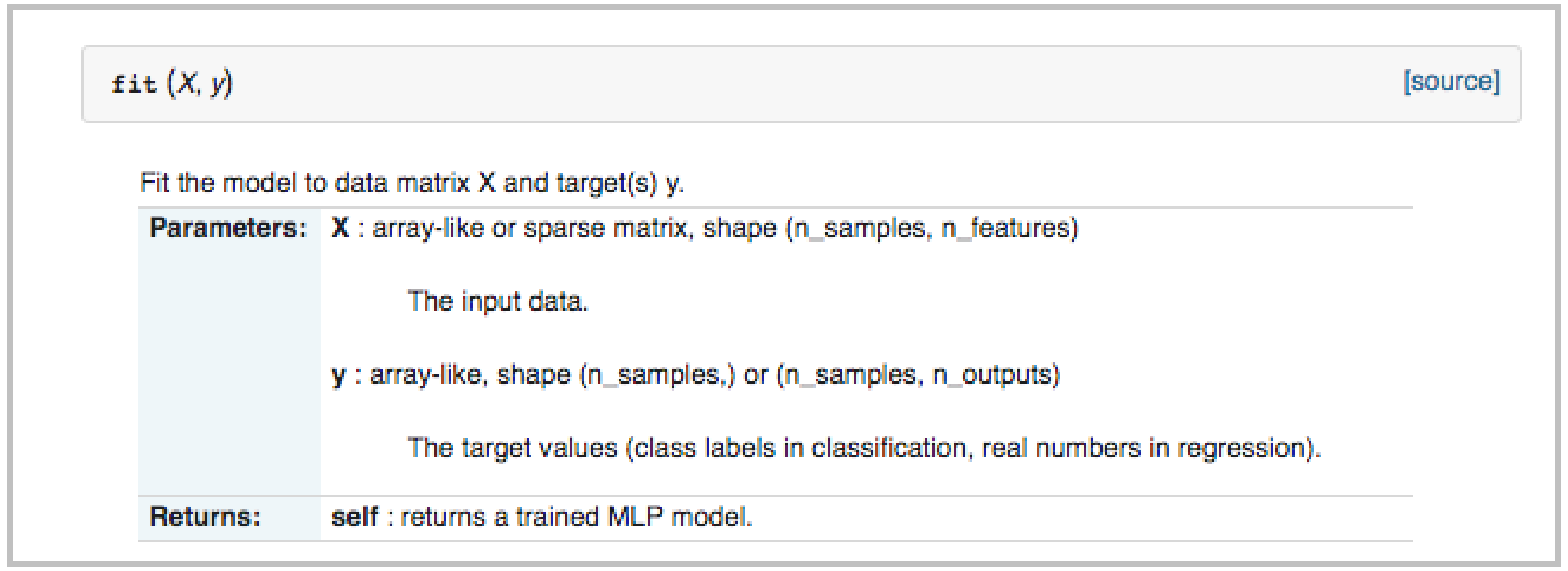
Build the model architecture

- To define a model's architecture, we need to define a few parameters, but at the very least, we must specify how many neurons our hidden layer will contain
- Let's start with **64 hidden neurons**
- **Note:** Remember to set the random state, to make sure our results are reproducible

```
# Build neural network model
nn = MLPClassifier(hidden_layer_sizes = (64), #<- 64 neurons for hidden layer
                   random_state = 1)           #<- set seed to 1
```

Fit the model to training data

- To fit the model to your training data you need to provide 2 objects:
  - the predictors (`X`)
  - the target (`y`)



fit (*X*, *y*) [source]

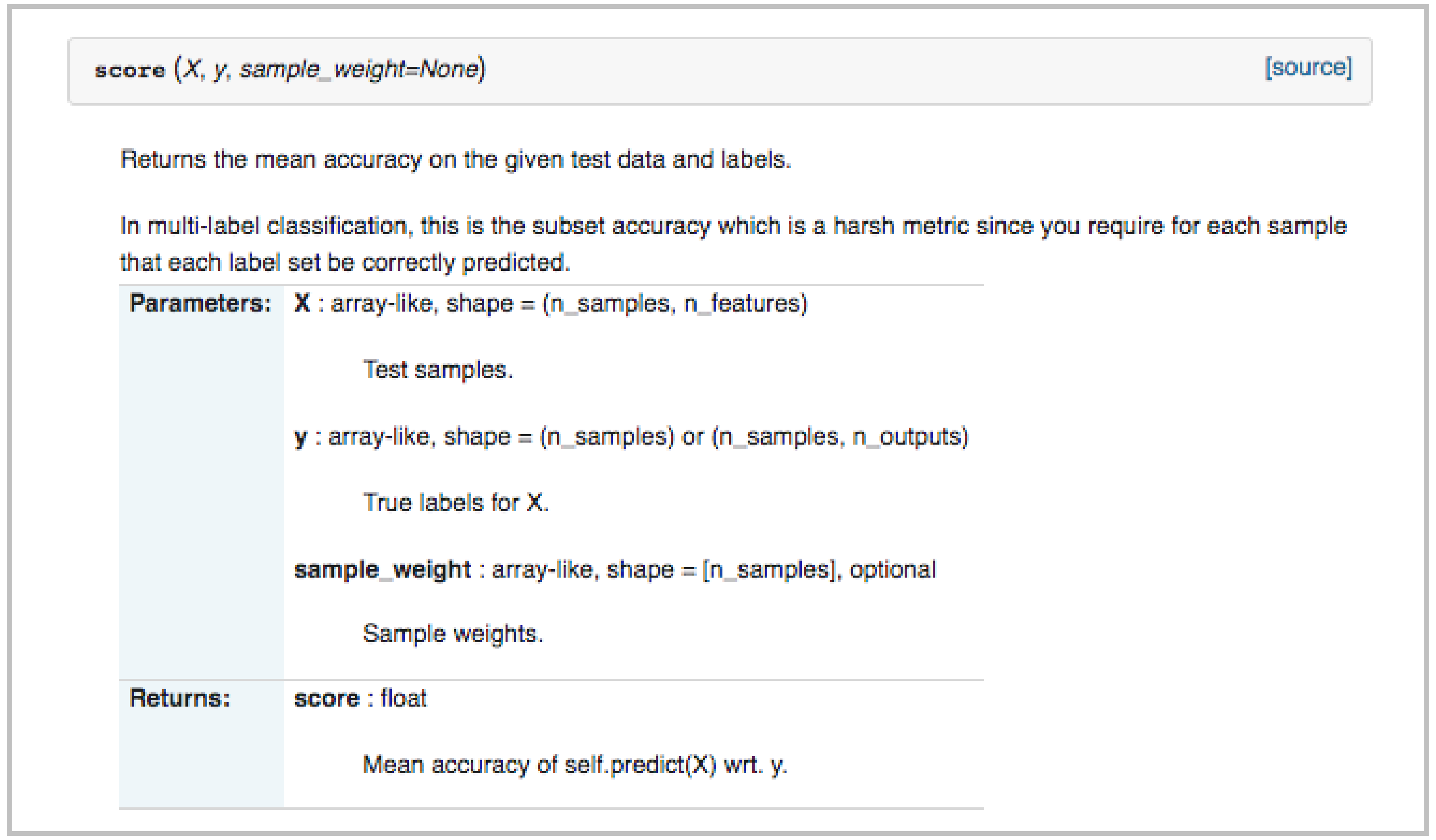
Fit the model to data matrix X and target(s) y.

Parameters:	**X** : array-like or sparse matrix, shape (n_samples, n_features) The input data. **y** : array-like, shape (n_samples,) or (n_samples, n_outputs) The target values (class labels in classification, real numbers in regression).
Returns:	**self** : returns a trained MLP model.

```
# Fit the saved model to your training data.
fit_nn = nn.fit(X_train_scaled, y_train)
```

Inspect accuracy of training model

- Let's look at the accuracy of the model we just built on the training data



```
score (X, y, sample_weight=None)                                    [source]
```

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters:	**X** : array-like, shape = (n_samples, n_features) Test samples. **y** : array-like, shape = (n_samples) or (n_samples, n_outputs) True labels for X. **sample_weight** : array-like, shape = [n_samples], optional Sample weights.
Returns:	**score** : float Mean accuracy of self.predict(X) wrt. y.

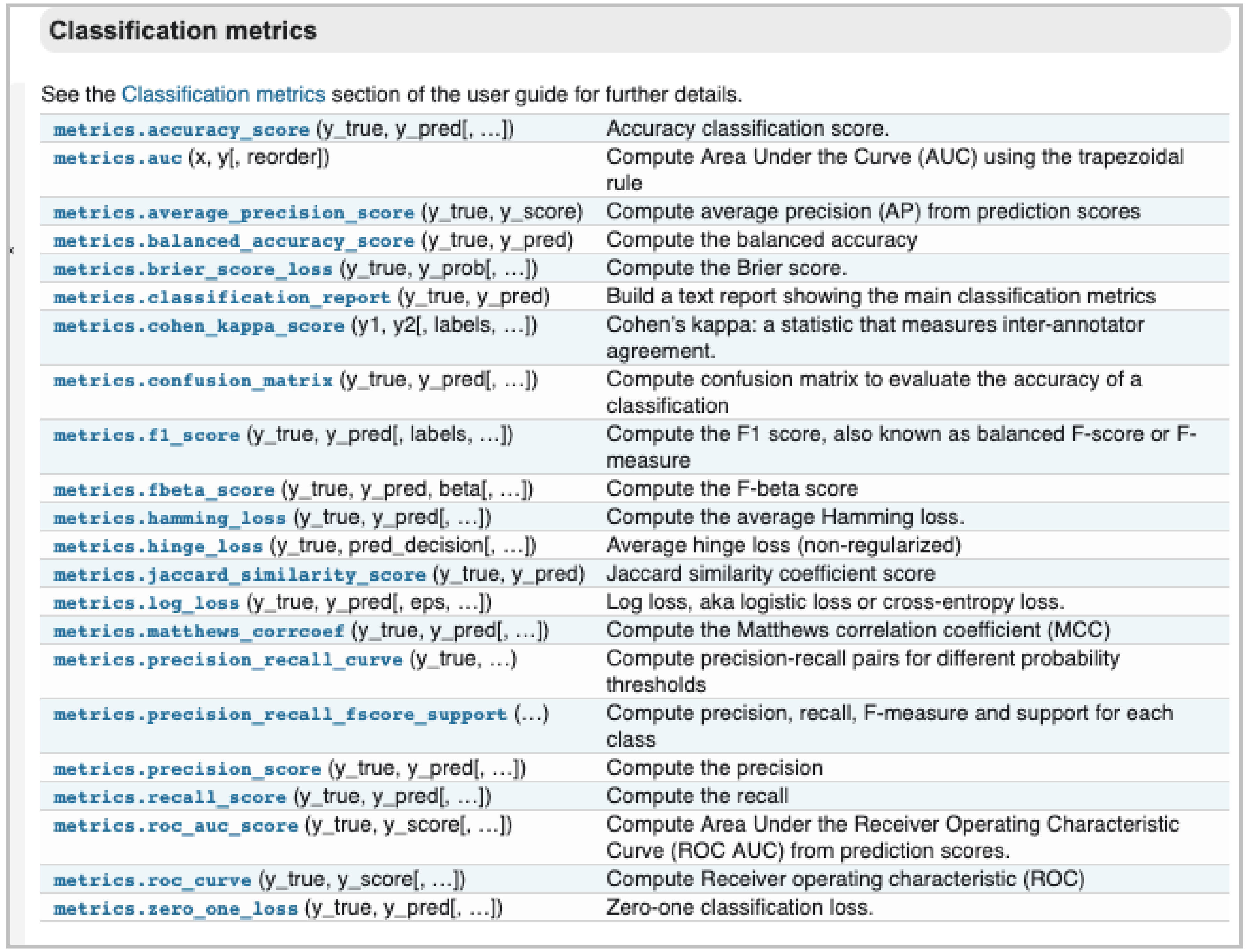
```
# Compute accuracy using training data.
acc_train_nn = fit_nn.score(X_train_scaled,
                            y_train)
print ("Train Accuracy:", acc_train_nn)
```

```
Train Accuracy: 0.8284285714285714
```

- Since this is accuracy on the **training data**, it will generally be higher than the accuracy on the **test data**

Classification: sklearn.metrics

- `sklearn.metrics` has many packages used to calculate metrics for various models
- We will be using metrics found within the *Classification metrics* section
- Refer to the following table to get an idea of what we can calculate using this library



Classification metrics

See the Classification metrics section of the user guide for further details.

`metrics.accuracy_score` (y_true, y_pred[, …])	Accuracy classification score.
`metrics.auc` (x, y[, reorder])	Compute Area Under the Curve (AUC) using the trapezoidal rule
`metrics.average_precision_score` (y_true, y_score)	Compute average precision (AP) from prediction scores
`metrics.balanced_accuracy_score` (y_true, y_pred)	Compute the balanced accuracy
`metrics.brier_score_loss` (y_true, y_prob[, …])	Compute the Brier score.
`metrics.classification_report` (y_true, y_pred)	Build a text report showing the main classification metrics
`metrics.cohen_kappa_score` (y1, y2[, labels, …])	Cohen's kappa: a statistic that measures inter-annotator agreement.
`metrics.confusion_matrix` (y_true, y_pred[, …])	Compute confusion matrix to evaluate the accuracy of a classification
`metrics.f1_score` (y_true, y_pred[, labels, …])	Compute the F1 score, also known as balanced F-score or F-measure
`metrics.fbeta_score` (y_true, y_pred, beta[, …])	Compute the F-beta score
`metrics.hamming_loss` (y_true, y_pred[, …])	Compute the average Hamming loss.
`metrics.hinge_loss` (y_true, pred_decision[, …])	Average hinge loss (non-regularized)
`metrics.jaccard_similarity_score` (y_true, y_pred)	Jaccard similarity coefficient score
`metrics.log_loss` (y_true, y_pred[, eps, …])	Log loss, aka logistic loss or cross-entropy loss.
`metrics.matthews_corrcoef` (y_true, y_pred[, …])	Compute the Matthews correlation coefficient (MCC)
`metrics.precision_recall_curve` (y_true, …)	Compute precision-recall pairs for different probability thresholds
`metrics.precision_recall_fscore_support` (…)	Compute precision, recall, F-measure and support for each class
`metrics.precision_score` (y_true, y_pred[, …])	Compute the precision
`metrics.recall_score` (y_true, y_pred[, …])	Compute the recall
`metrics.roc_auc_score` (y_true, y_score[, …])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
`metrics.roc_curve` (y_true, y_score[, …])	Compute Receiver operating characteristic (ROC)
`metrics.zero_one_loss` (y_true, y_pred[, …])	Zero-one classification loss.

Confusion matrix

- A **confusion matrix** consists of counts of true vs. predicted values in our model outputs
- We don't necessarily use a confusion matrix directly in our performance assessment, but we derive the metrics we need from it, such as:
  - Accuracy
  - Precision
  - True positive rate or sensitivity or recall
  - Specificity

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

Confusion matrix: summary

- Here is a table with all the metrics:

Metric name	Formula
Accuracy	True positive + True Negative / Overall total
Misclassification rate	False positive + False Negative / Overall total
True positive rate	True positive / Actual yes (True positive + False negative)
False positive rate	False positive / Actual no (False positive + True negative)
Specificity	True negative / Actual no (False positive + True negative)

Confusion matrix: accuracy

- **Accuracy**: overall, how often is the classifier correct?

- TP + TN / **total**

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

When is accuracy not so accurate

- *Why not just use accuracy?*
  - Because it takes both true positives and true negatives into account, but doesn't tell us how many of either we had
  - It doesn't work for **class-imbalanced datasets**

- For example, let's imagine that we need to identify customers who will default on their credit card payments:
  - We know that about `1%` of all households overall will default
  - If the algorithm says No 100% of the time (i.e., none of the households will default), it will be accurate about 99% of the time
  - However, this is not a good model because we won't be able to detect customers that will actually default

Predict on test data

```
# Predict on test data.
predicted_values_nn = fit_nn.predict(X_test_scaled)
print(predicted_values_nn)
```

```
[0 0 0 ... 0 0 0]
```

```
# Compute test model accuracy score.
test_accuracy_score = metrics.accuracy_score(y_test, predicted_values_nn)
print("Accuracy on test data: ", test_accuracy_score)
```

```
Accuracy on test data:  0.8136666666666666
```

Confusion matrix

- Similar to `accuracy_score`, the `confusion_matrix` method takes actual values and compares them to those we predicted

```
# Take a look at test data confusion matrix.
conf_matrix_test = metrics.confusion_matrix(y_test, predicted_values_nn)
print(conf_matrix_test)
```

```
[[6624  376]
 [1301  699]]
```

Precision

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

- $PR = \frac{(TP)}{(TP+FP)}$
- It is the proportion of values that is truly positive out of all predicted positive values (A.K.A PPV)

Precision (cont’d)

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

- In our example, if $PR = 0$, it makes our seemingly accurate algorithm a total miss
- It’s a great metric to use when we want to be very sure of our prediction like convicting a person who is genuinely guilty
- The flip side in being very precise is letting some criminals walk free or catching too many false negatives

Recall

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

- $RE = \frac{(TP)}{(TP+FN)}$
- It is the proportion of actual positives that are classified correctly
- A.K.A. sensitivity, hit rate, or true positive rate (TPR)

Recall (cont'd)

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

- In the credit card payment default prediction case, if our algorithm failed to mark a single positive value, the RE would be 0
- It's a great metric **to capture as many positives as possible**
- Recall is 1 if we say that every cardholder *will default*
- In this case, we are running into a problem of recording too many false positives

Specificity

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

- $Specificity = \frac{(TN)}{(TN+FP)}$
- It is the proportion of actual negatives, that were predicted as negative (A.K.A. True negative rate)

Specificity (cont’d)

	Actual positive	Actual negative	
Predicted positive	TP	FP	Total predicted positives
Predicted negative	FN	TN	Total predicted negatives
	Total actual positives	Total actual negatives	

- It’s a great metric **to capture as many negatives as possible**
- Specificity is `1` if we say that every single credit card holder *will NOT default on a credit card payment*
- In this case, we are running into a problem of recording too many false negatives

F1: precision vs. recall

- The $F1$ score gives a **numeric value to the precision vs. recall tradeoff**
- The score can be between 0 and 1; the higher, the better
- $F1 = 2 \times \frac{(PR*RE)}{(PR+RE)}$

Classification report

- The classification report allows to look at all of the metrics at once

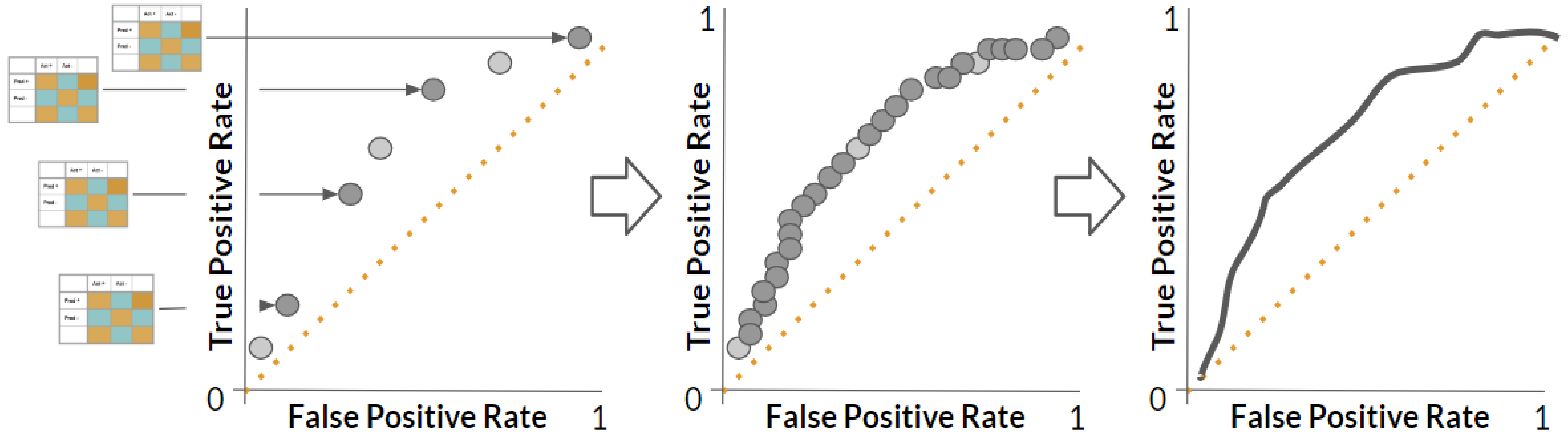
```
# Create a list of target names to interpret class assignments.
target_names = ['default_payment_0', 'default_payment_1']
```

```
# Print an entire classification report.
class_report = metrics.classification_report(y_test,
                                              predicted_values_nn,
                                              target_names = target_names)
print(class_report)
```

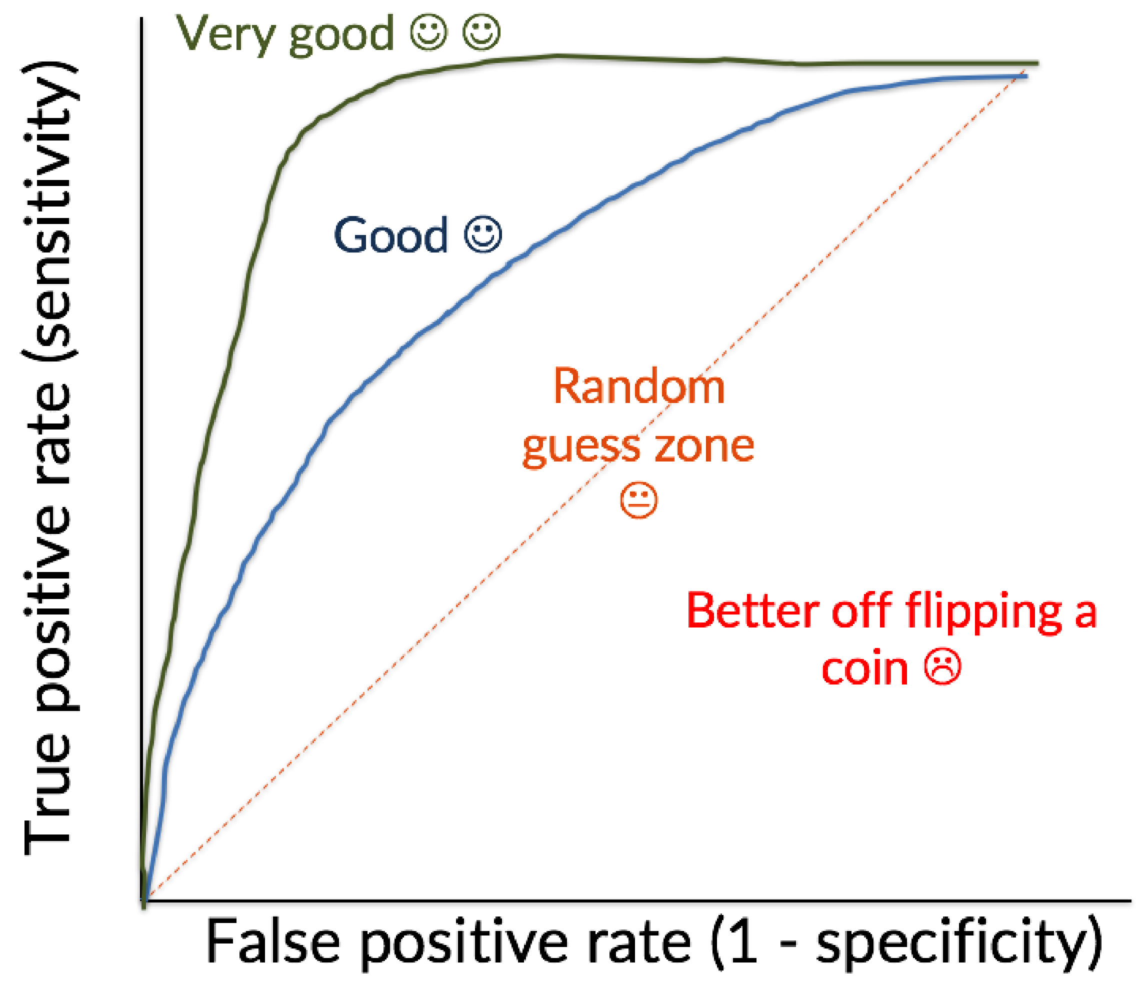
```
                   precision    recall  f1-score   support

default_payment_0       0.84      0.95      0.89      7000
default_payment_1       0.65      0.35      0.45      2000

         accuracy                           0.81      9000
        macro avg       0.74      0.65      0.67      9000
     weighted avg       0.79      0.81      0.79      9000
```

Receiver Operator Characteristic (ROC) curve

- The **Receiver Operator Characteristic** (ROC) curve displays the **tradeoff** between TPR (a.k.a. Recall) and FPR (a.k.a. Specificity)
- This name was derived because it was invented for operators of military radar receivers
- Every point on the curve reflects **TPR vs. FPR** for a different value of decision threshold



Area Under the ROC Curve (AUC)



- The AUC metric is commonly used to indicate how well the algorithm performed
  - The closer to `1` the better
  - Anything below `0.5` is considered worse than a random guess

Getting probabilities instead of class labels

- If we would like to construct a ROC curve and measure the AUC, we need to compute the probabilities of an observation being classified as one class or the other instead of the actual class labels

```
# Get probabilities instead of predicted values.
test_probabilities = fit_nn.predict_proba(X_test_scaled)

# Get probabilities of test predictions only.
test_predictions = test_probabilities[:, 1]
```

Computing FPR, TPR, and threshold

- The next step is to compute the ROC curve, which is constructed using the false positive and true positive rates
- It may also be useful to look at the threshold values used to compute the probabilities

```
# Get FPR, TPR, and threshold values.
fpr, tpr, threshold = metrics.roc_curve(y_test,             #<- test data labels
                                        test_predictions)   #<- predicted probabilities
print("False positive: ", fpr)
```

```
False positive:  [0.         0.         0.         ... 0.99628571 0.99628571 1.        ]
```

```
print("True positive: ", tpr)
```

```
True positive:  [0.000e+00 5.000e-04 2.500e-03 ... 9.995e-01 1.000e+00 1.000e+00]
```

```
print("Threshold: ", threshold)
```

```
Threshold:  [1.96926196e+00 9.69261960e-01 9.26069291e-01 ... 5.87868313e-03
 5.84374563e-03 1.59399562e-06]
```

Computing AUC

- To get the area under the ROC curve, we use the `auc` metric from the `metrics` module

```
# Get AUC by providing the FPR and TPR.
auc = metrics.auc(fpr, tpr)
print("Area under the ROC curve: ", auc)
```

```
Area under the ROC curve:  0.7730955714285714
```

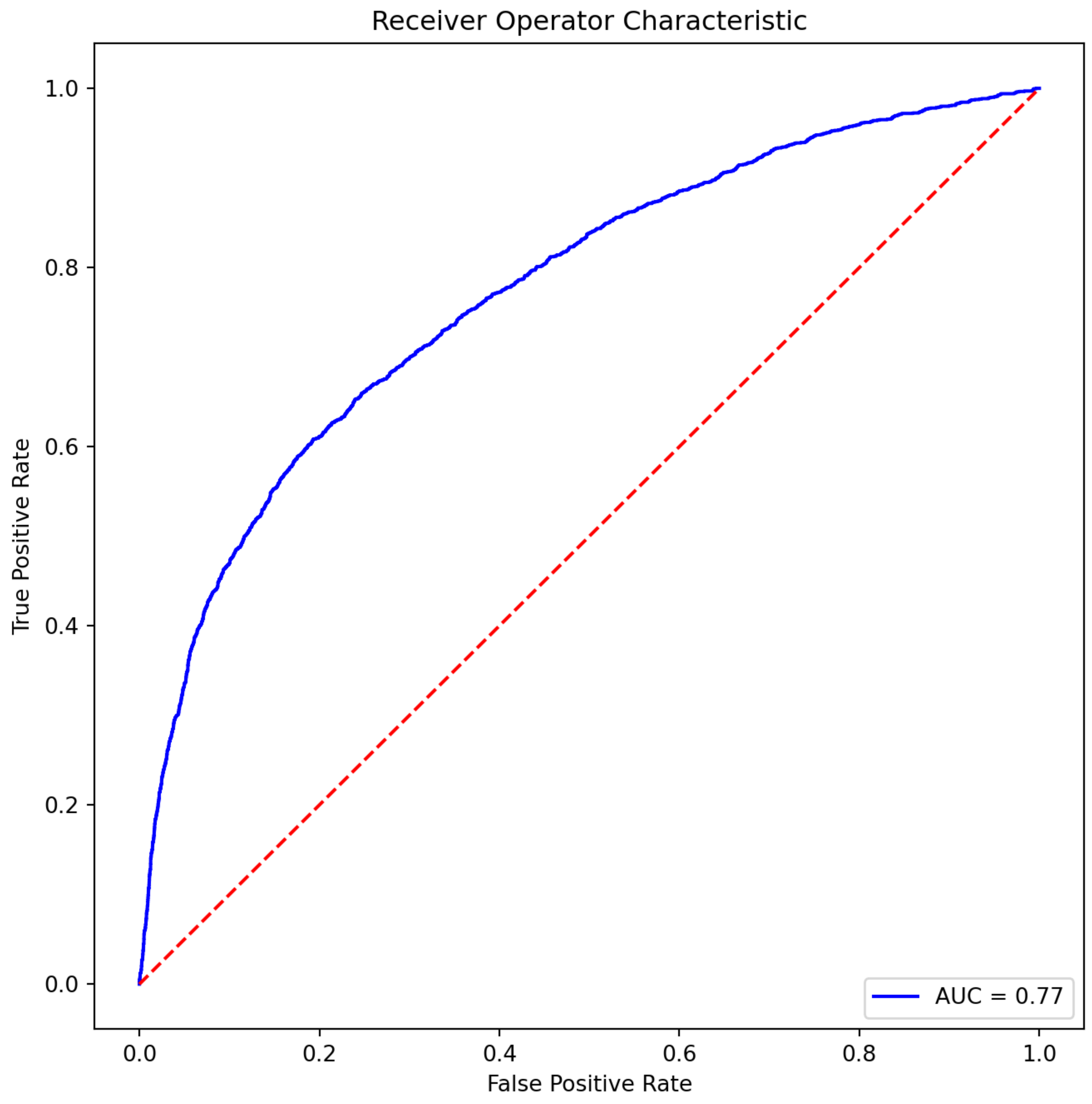
- Our model achieved an accuracy of about 0.81
- We have an AUC of about 0.77
- Given that we have not done any model tuning, we will take these results and see if we can do better

Putting it all together: ROC plot

```
# Make an ROC curve plot.
plt.title('Receiver Operator Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

- Let's look at the plot:
  - *What can you tell about it?*
  - *What factors do you think influenced our model performance?*

Putting it all together: ROC plot (cont'd)



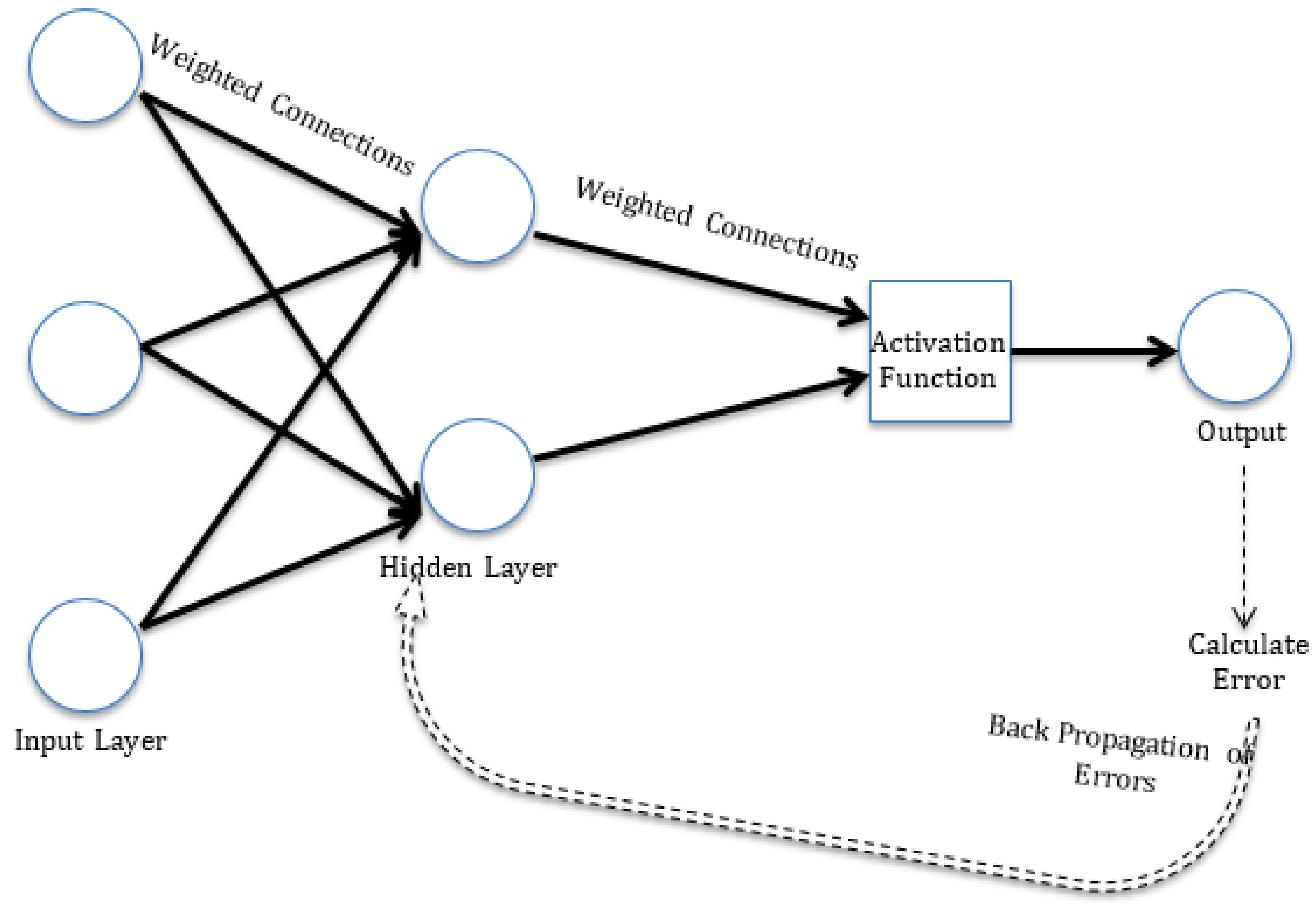
Module completion checklist

Objective	Complete
Implement and evaluate a simple neural network using MLPClassifier	✔
Explain the concept of backpropagation	

The single-layer feed-forward perceptron

- We practiced building the most straightforward type of neural network, i.e., a **single-layer feed-forward perceptron**

- Since *feed-forward* refers to a neural network that only travels one way, you may ask how it manages to learn from its mistakes



Backpropagation

- Information does **flow forward**, but pairs of input and output values are fed into the network for many cycles called **epochs**
- This way, the network “learns” the relationship between the input and output
- This process is called **backpropagation**

- **Note:** Although backpropagation feeds adjusted information back into the network, it is NOT recurrent and is still considered a network that flows “forward”! Backpropagation is only adjusting the weights of the neurons, not changing the topology of the network!

Ultimate goal: get to the desired output

- Let’s say that our neural network is about to learn a function:

$$y = 2x$$

- This is what the table of inputs and desired outputs would look like:

Input	Desired output
0	0
1	2
2	4
3	6
4	8

Random initialization

- While our network is being set up, we will generate a list of outputs that assign a random weight w to each input
- For example, if we make $w = 3$, our function becomes $y' = 3x$
- This step of the process in neural networks is called **forward propagation**, which means that we will feed the network information, and it will make the computations along the way to get the output values

Input	Current output	Desired output
0	0	0
1	3	2
2	6	4
3	9	6
4	12	8

Calculating error

- To know how well the model performed, we compute the **error**, which is simply the difference between the desired and current output

Input	Current output	Desired output	Error
0	0	0	0
1	3	2	1
2	6	4	2
3	9	6	3
4	12	8	4

Calculating error: loss function

- In machine learning, we often hear about *absolute error*, *squared error*, or *root mean squared error* because we are only interested in the magnitude of the error, not whether it is positive or negative
- When we square the errors, we penalize significant errors more severely
- The formula that computes the total error *(RMSE in our case)* is called the **loss function**

$$E_{total} = \sqrt{\frac{\sum_{i=1}^{n}(d_i - c_i)^2}{N}} =$$

$$\sqrt{\frac{(0^2 + 1^2 + 2^2 + 3^2 + 4^2)}{5}} = 2.44949$$

Why loss?

- The **loss function** is an indicator of how much precision we lose if we replace the desired output by the actual output generated by our trained neural network model

- If we adjust the weight in our example from $w = 3 \Rightarrow w = 3.1$ our error will increase and we will lose more information

- If we adjust the weight from $w = 3 \Rightarrow w = 2.5$, our error will decrease and we won't lose as much information

What do we compare loss to?

- We don't really know if the total error of 2.44949 is big or not; we need to compare it to something
- When we have $w = 3.1$, the increase in the weight was 0.1, the new $RMSE = 2.531403$
- The difference in errors isn't big with respect to each other $2.531403 - 2.44949 = 0.081913$
- But with respect to the 0.1 increase in weight, that difference is $10\times$ bigger $0.081913/0.1 = 0.81913$
- This ratio of error difference toward the difference in weight is the **derivative**, which tells us the **rate of change of the error with respect to weight adjustment**

How do we update the weights then?

- We can derive the new weight from knowing the old weight and the error

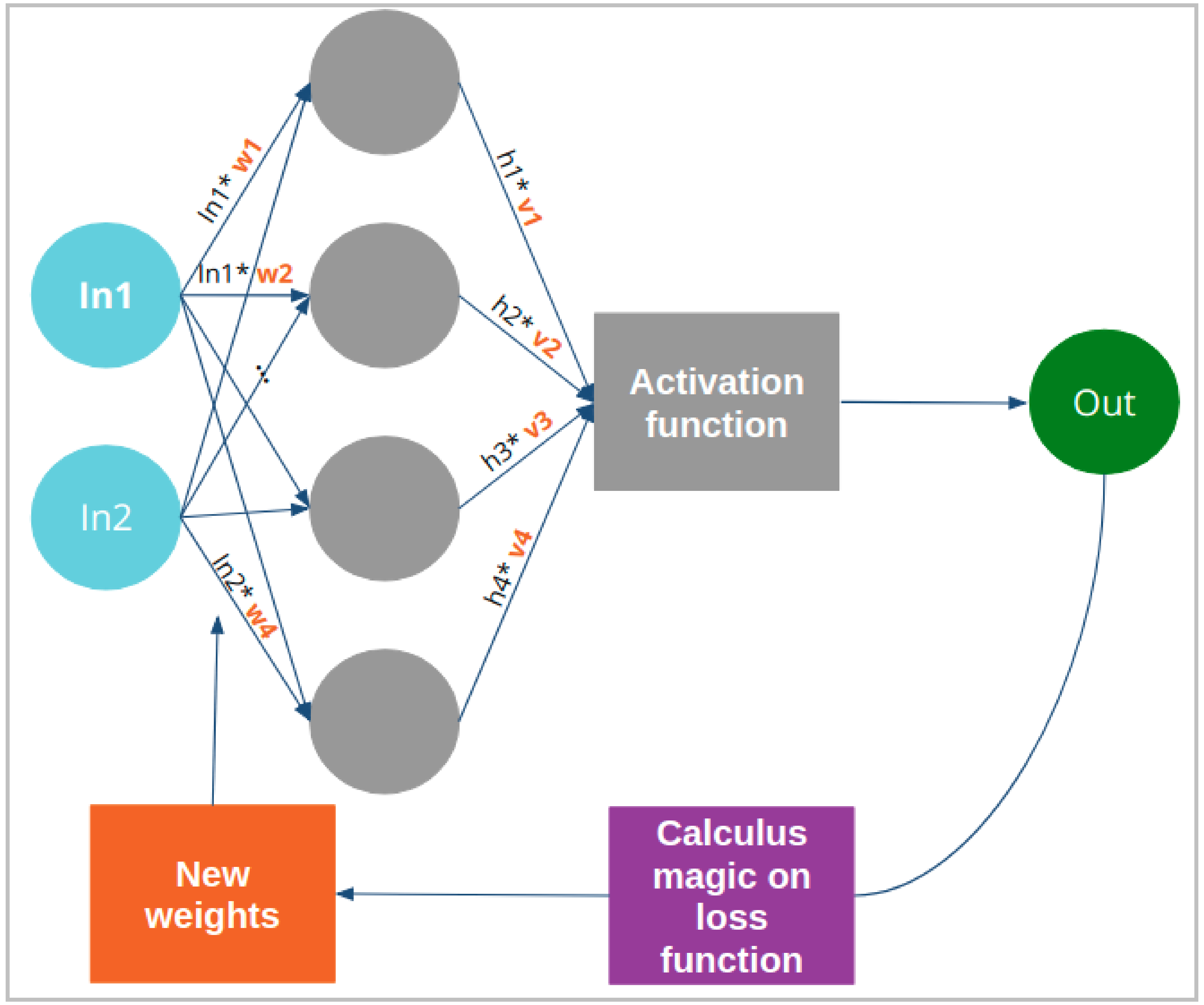
$$w_{new} = w_{old} - N\frac{dErr}{dw}$$

- Where $\frac{dErr}{dw}$ is that derivative (or ratio) of the **difference in error vs. the difference in weight** between the two *epochs* of the neural network's cycle

Determining the direction

- The primary question in any machine learning problem is: *Should we increase or decrease the weight?*
- Knowing and going in the right direction to minimize the error is critical to a **fast convergence** to the correct result
- The answer to that question is called **gradient**
- We won't go into the calculus, but all you need to know is that by optimizing the function $w_{new} = w_{old} - N\frac{dErr}{dw}$, we perform a so-called **gradient descent** to find the optimum solution that will minimize the error and make our approximated result as close to the desired result as possible
- This **iterative approach of adjusting the weights and feeding them back into the network** makes up the entire process of **backpropagation**

To sum it up


In1
In2
In1* w1
In1* w2
In2* w4
h1* v1
h2* v2
h3* v3
h4* v4
Activation function
Out
Calculus magic on loss function
New weights

Knowledge check

Module completion checklist

Objective	Complete
Implement and evaluate a simple neural network using MLPClassifier	✔
Explain the concept of backpropagation	✔

Congratulations on completing this module!

You are now ready to try tasks 8-13 in the Exercise for this topic