

Learning curves

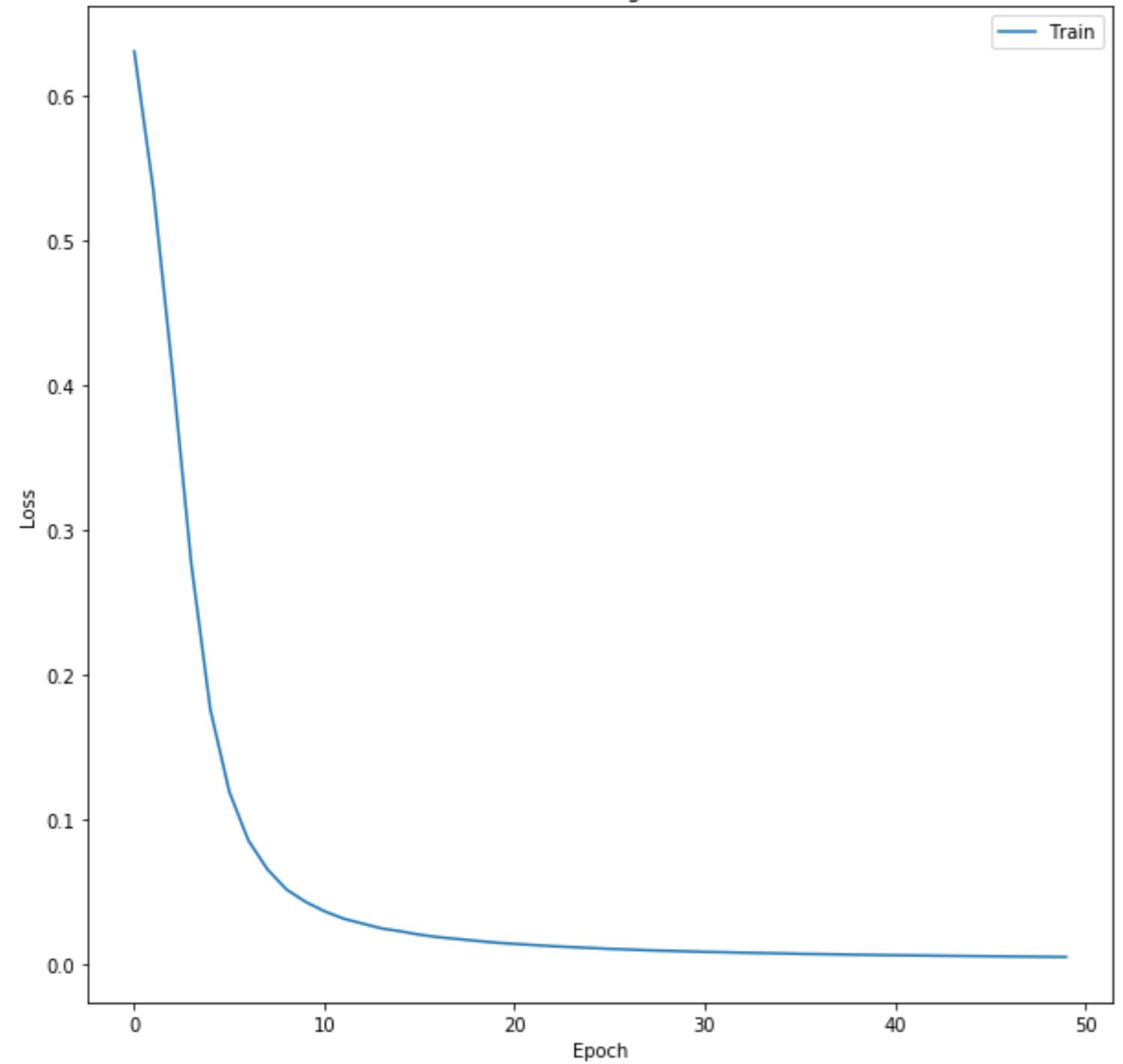
INTRODUCTION TO DEEP LEARNING WITH KERAS



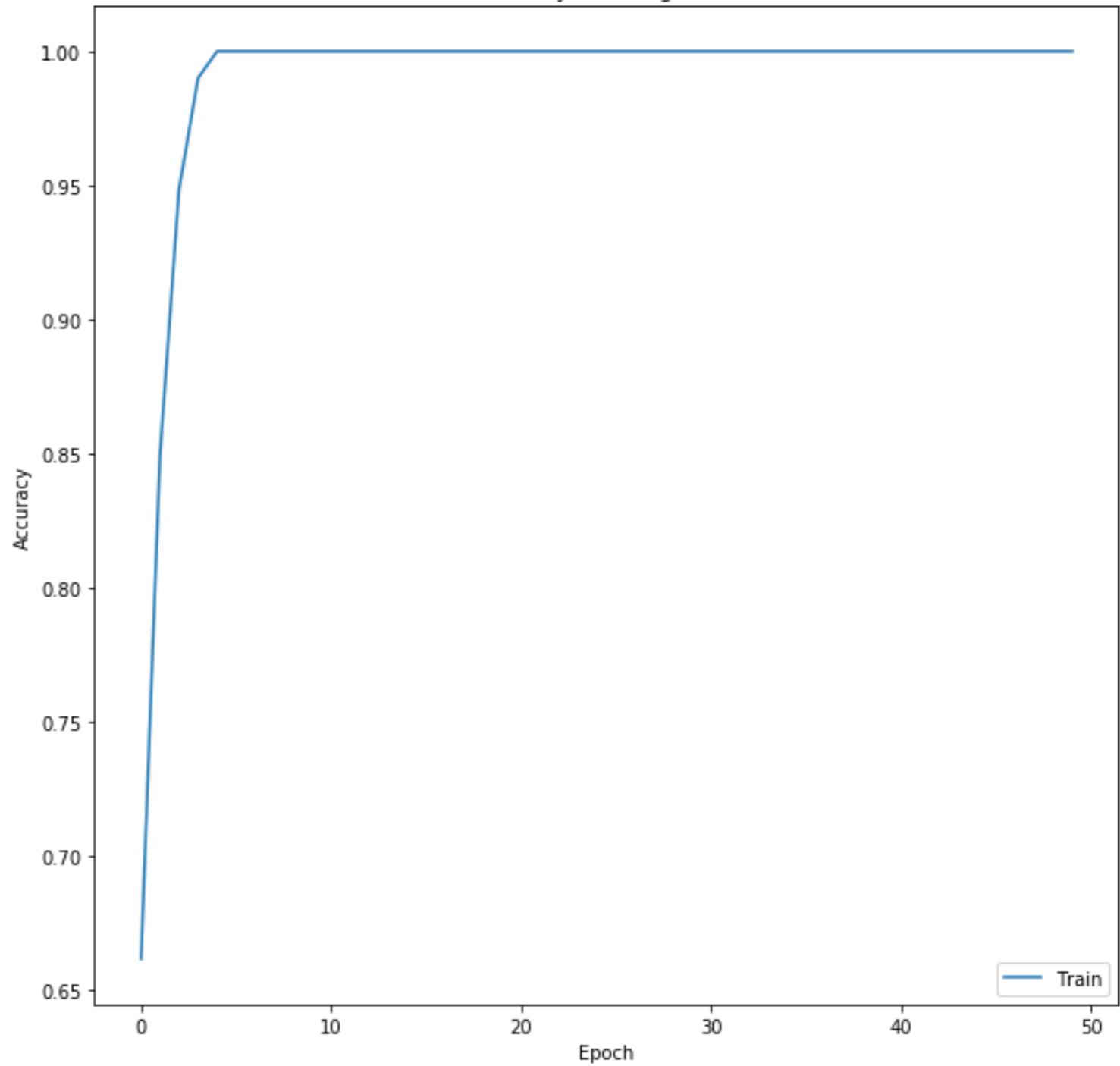
Miguel Esteban

Data Scientist & Founder

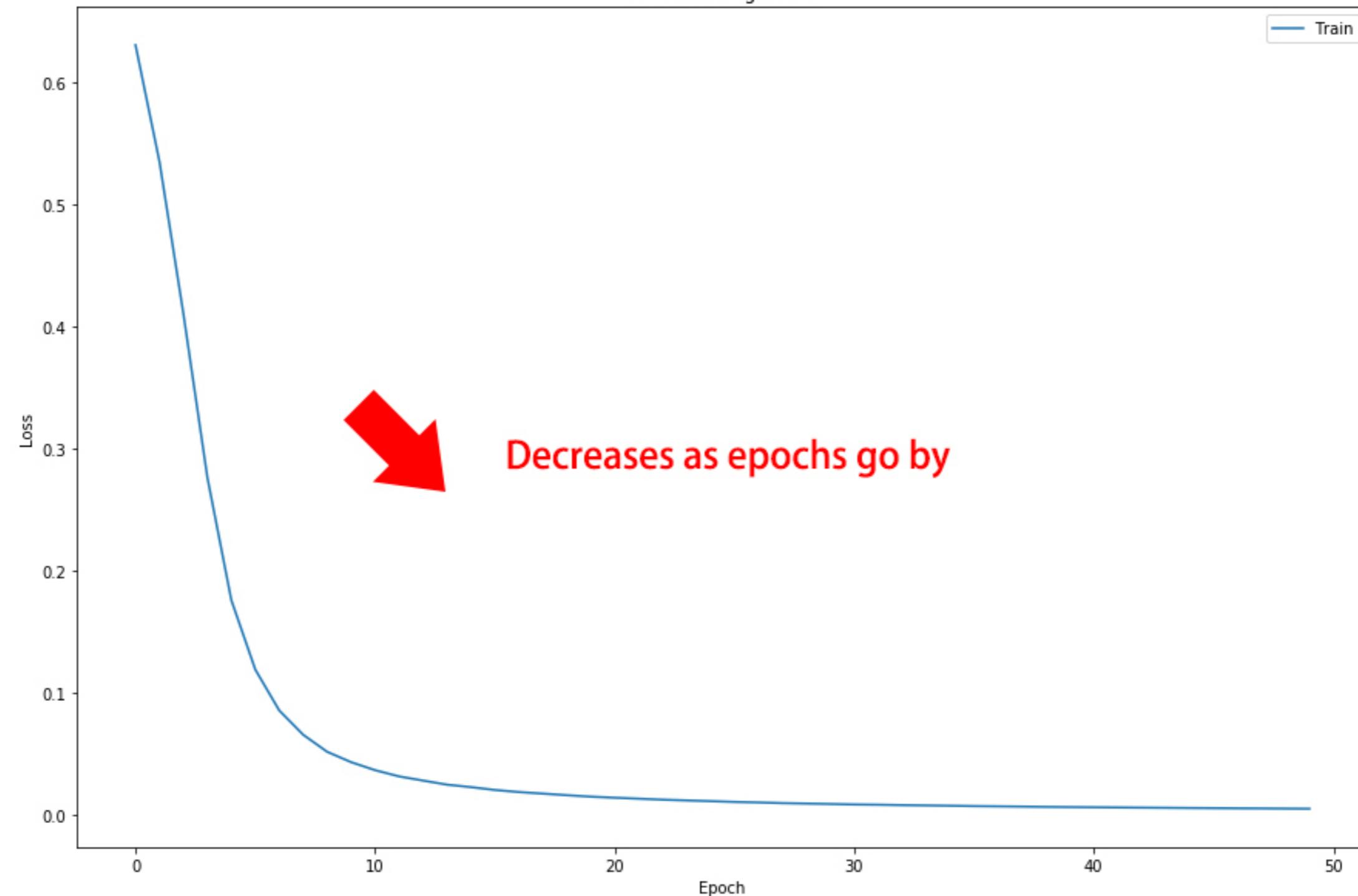
Loss Learning Curve

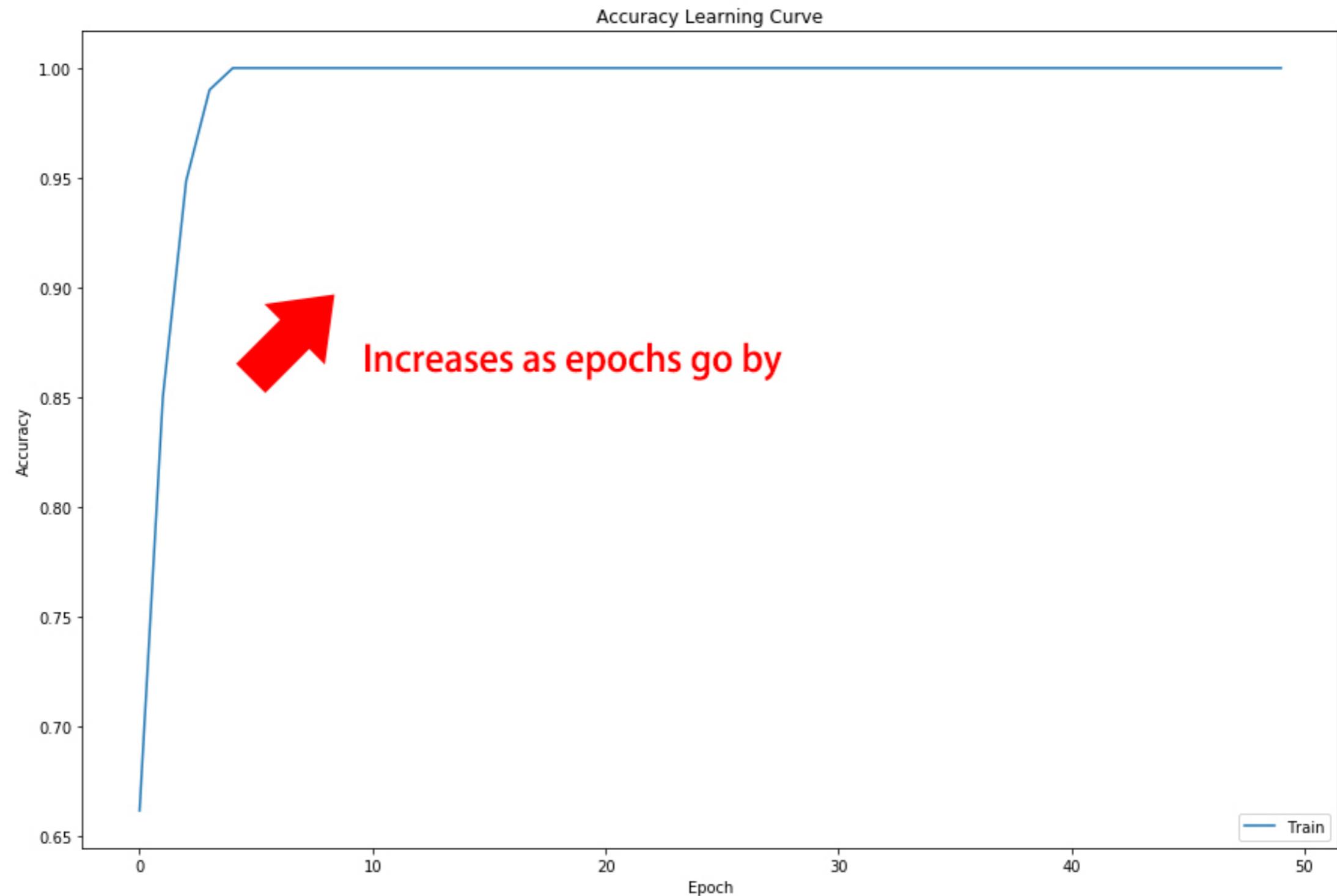


Accuracy Learning Curve

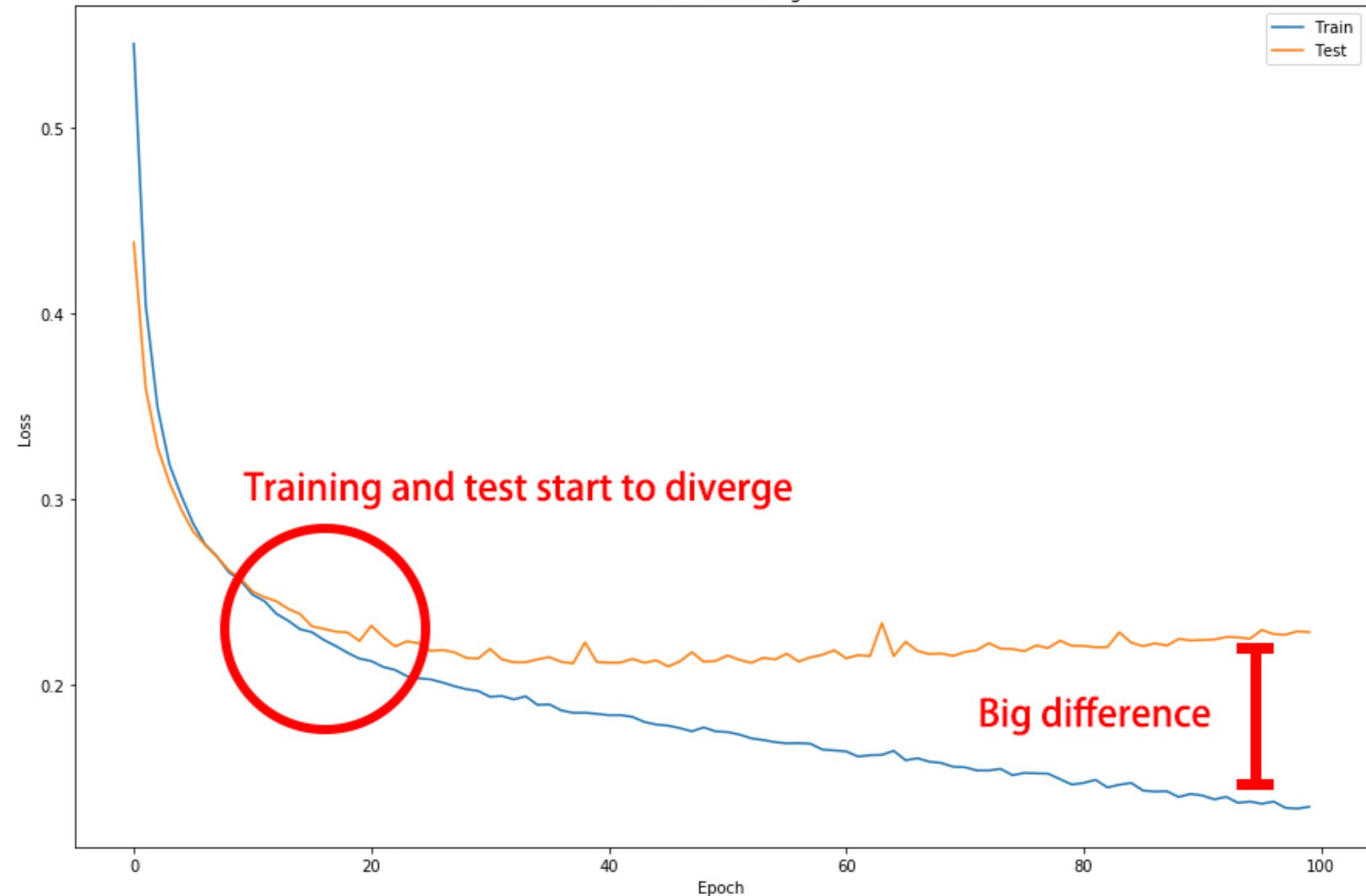


Loss Learning Curve

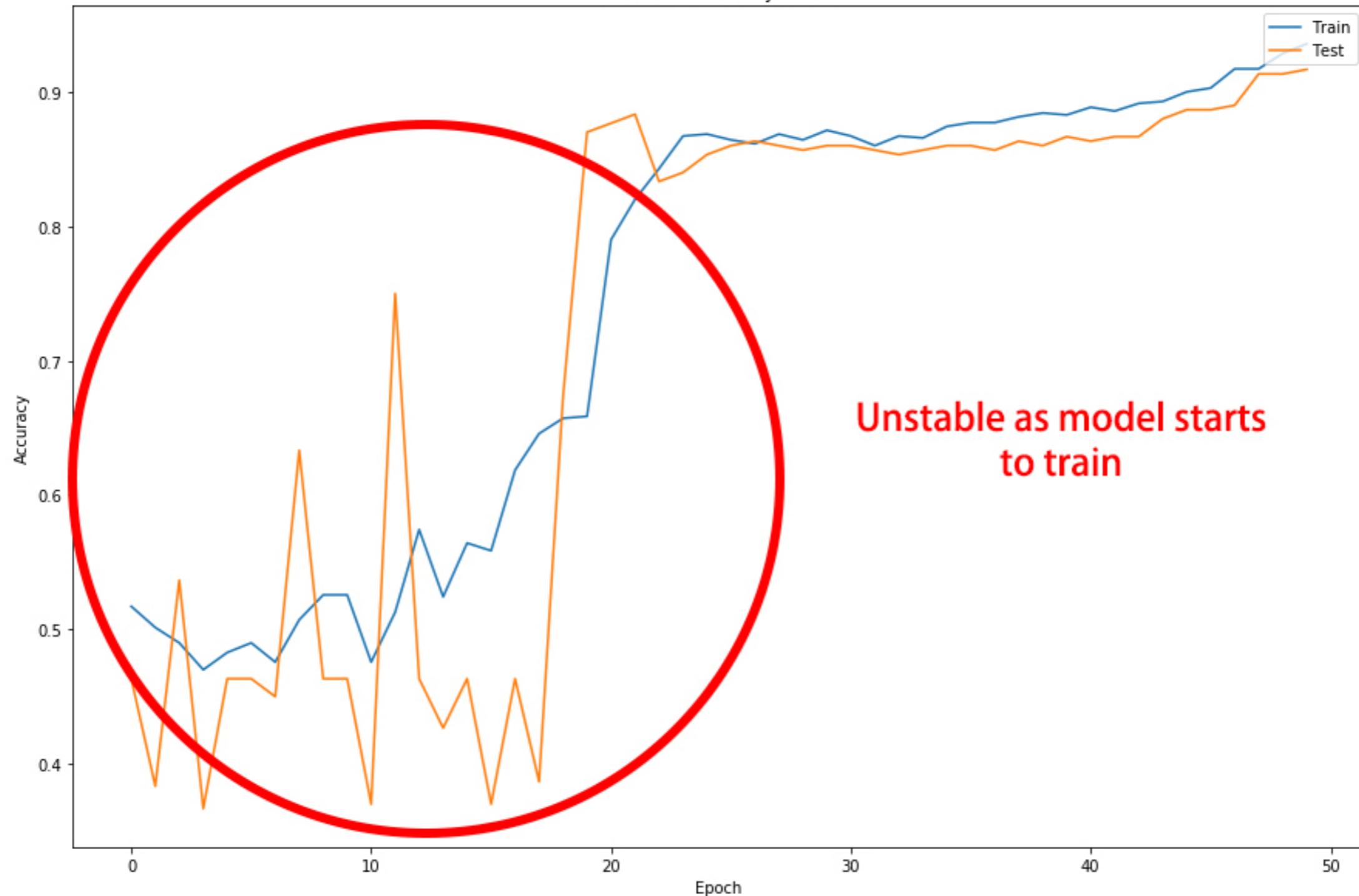




Model Overfitting



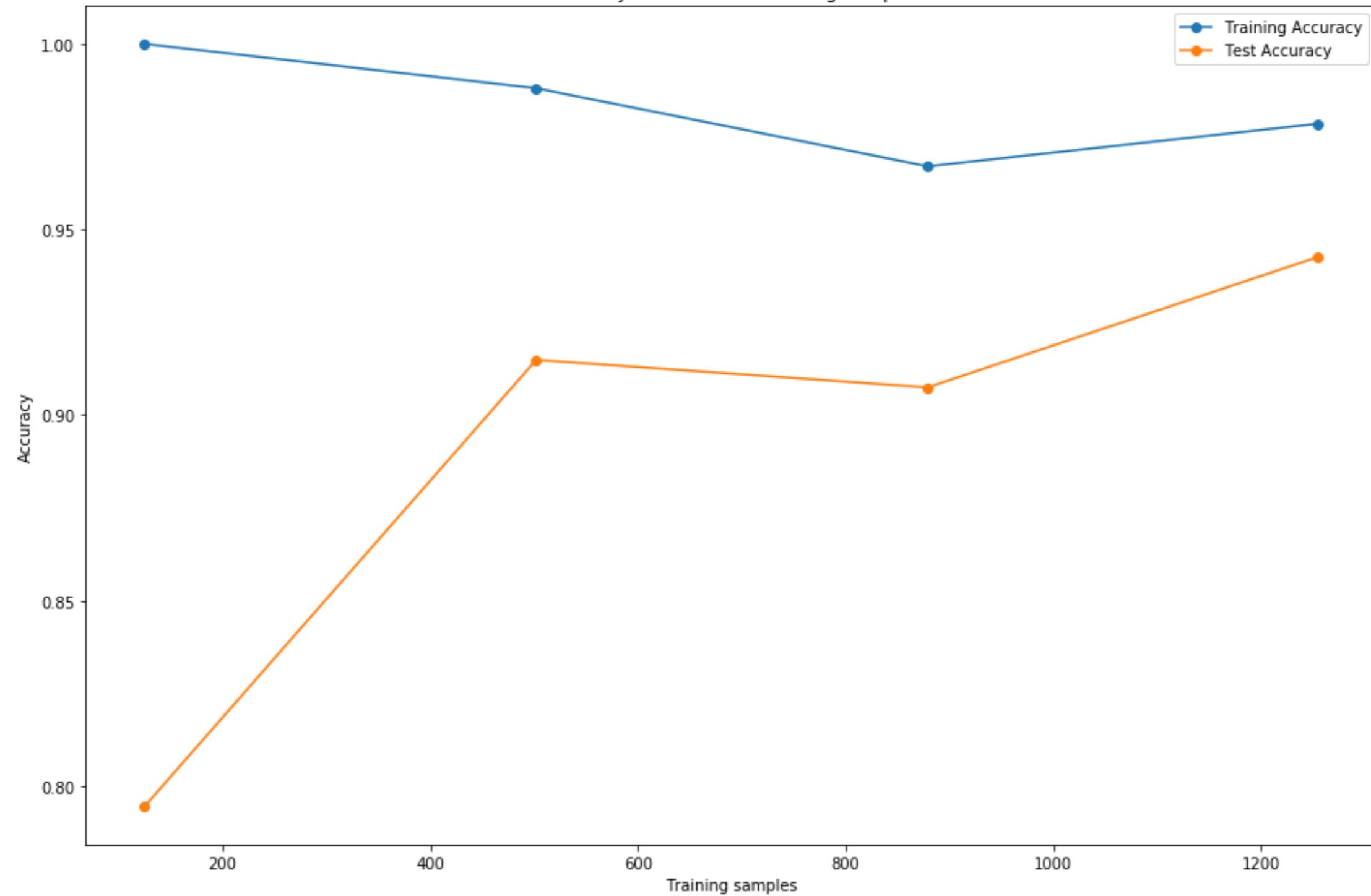
Unstable Accuracy Curve

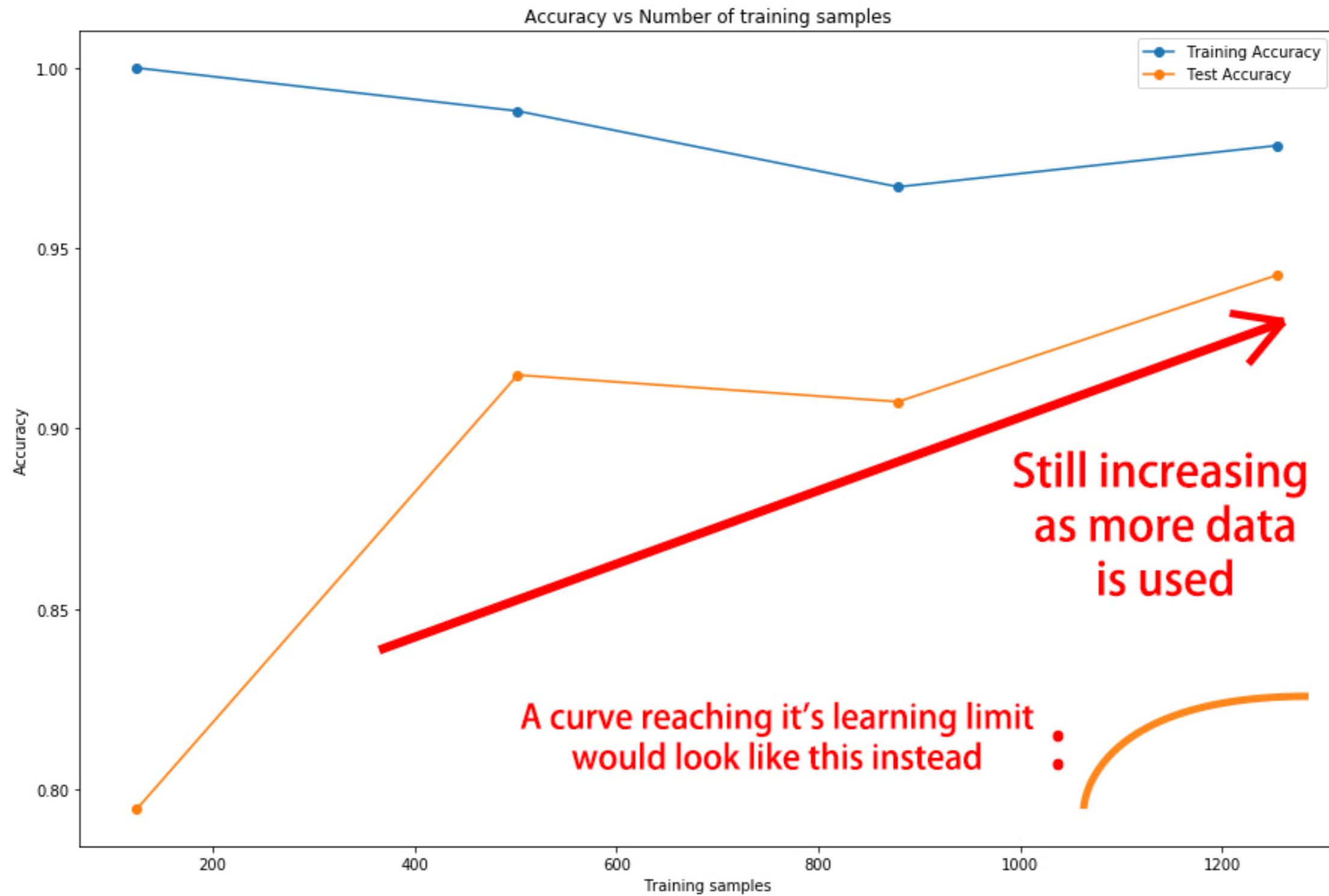




Neural networks like to be fed
with a **BIG** and **VARIED**
amount of data.

Accuracy vs Number of training samples





```
# Store initial model weights  
init_weights = model.get_weights()  
  
# Lists for storing accuracies  
train_accs = []  
tests_accs = []
```

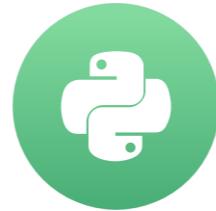
```
for train_size in train_sizes:  
    # Split a fraction according to train_size  
    X_train_frac, _, y_train_frac, _ =  
        train_test_split(X_train, y_train, train_size=train_size)  
    # Set model initial weights  
    model.set_weights(initial_weights)  
    # Fit model on the training set fraction  
    model.fit(X_train_frac, y_train_frac, epochs=100,  
              verbose=0,  
              callbacks=[EarlyStopping(monitor='loss', patience=1)])  
    # Get the accuracy for this training set fraction  
    train_acc = model.evaluate(X_train_frac, y_train_frac, verbose=0)[1]  
    train_accs.append(train_acc)  
    # Get the accuracy on the whole test set  
    test_acc = model.evaluate(X_test, y_test, verbose=0)[1]  
    test_accs.append(test_acc)  
    print("Done with size: ", train_size)
```

**Time to dominate all
curves!**

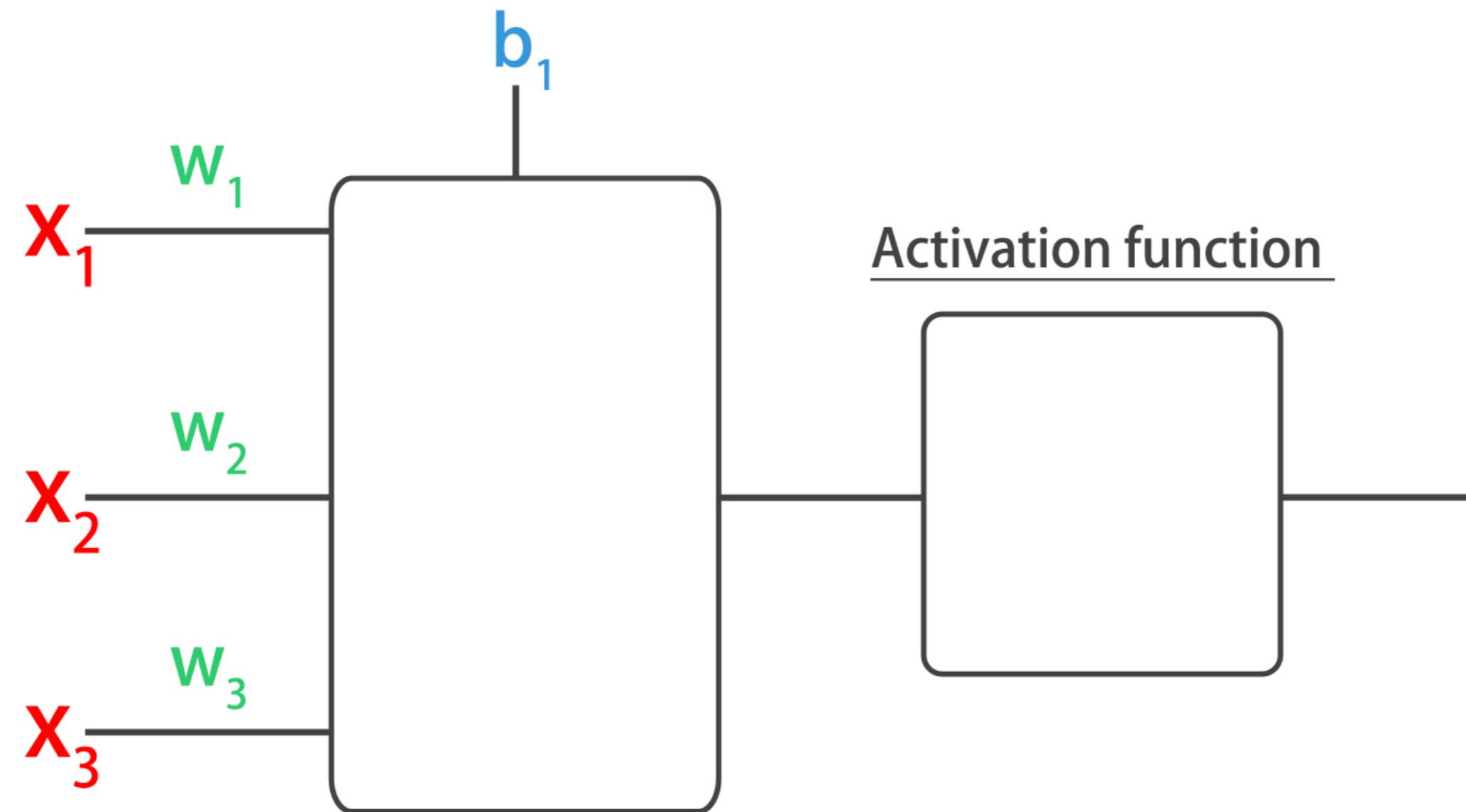
INTRODUCTION TO DEEP LEARNING WITH KERAS

Activation functions

INTRODUCTION TO DEEP LEARNING WITH KERAS

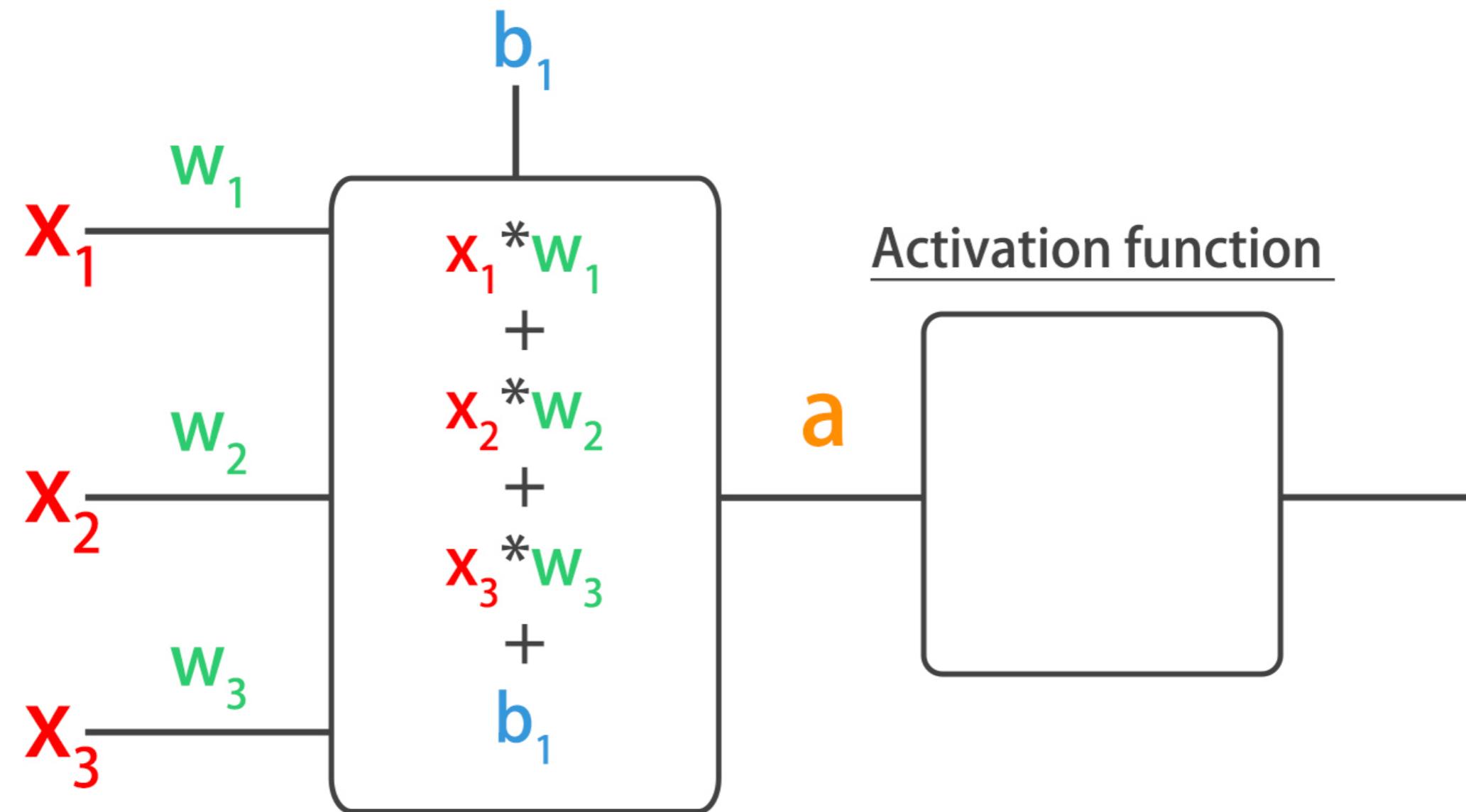


Miguel Esteban
Data Scientist & Founder



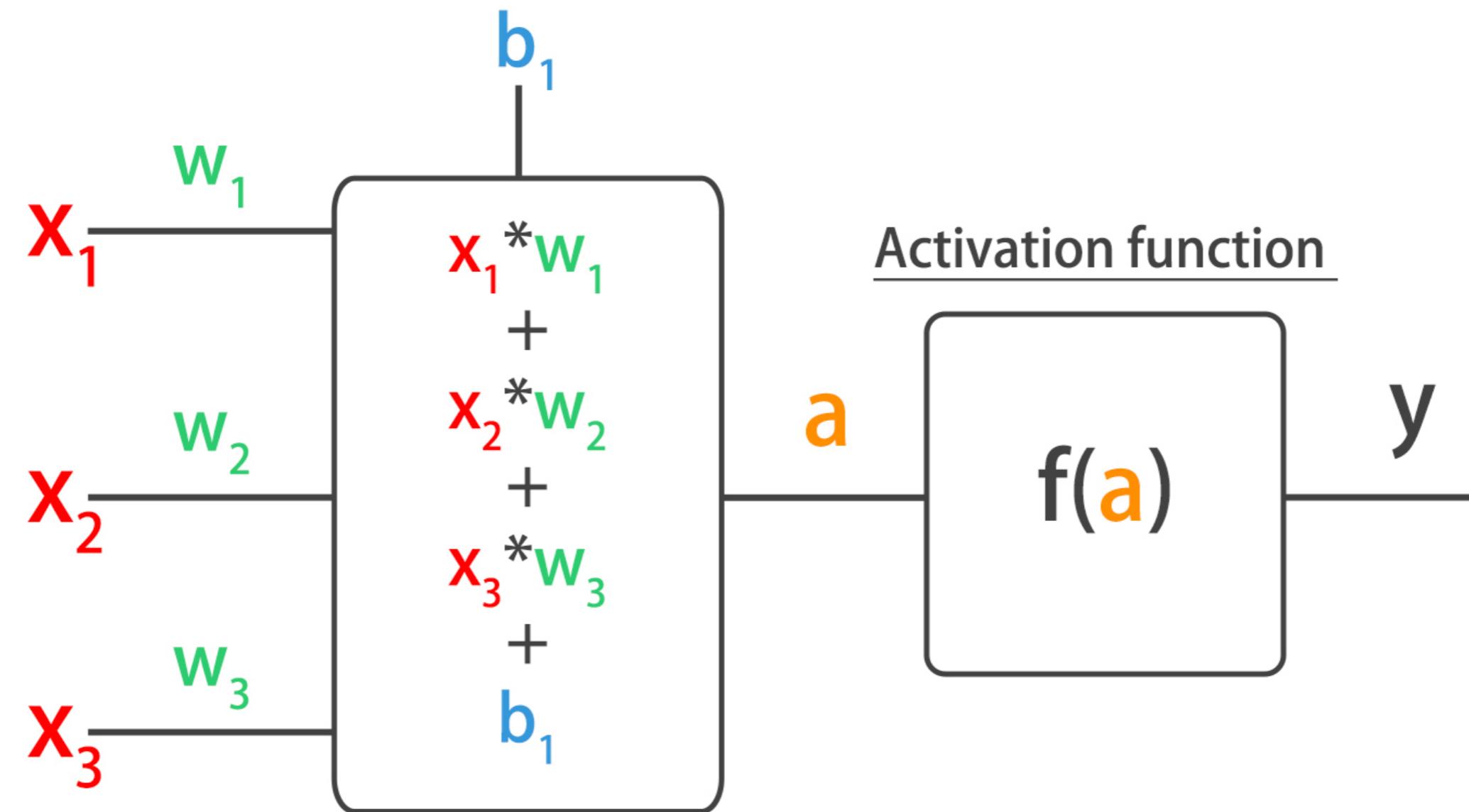
Neuron

$a = \text{sum of inputs} * \text{weights} + \text{bias}$



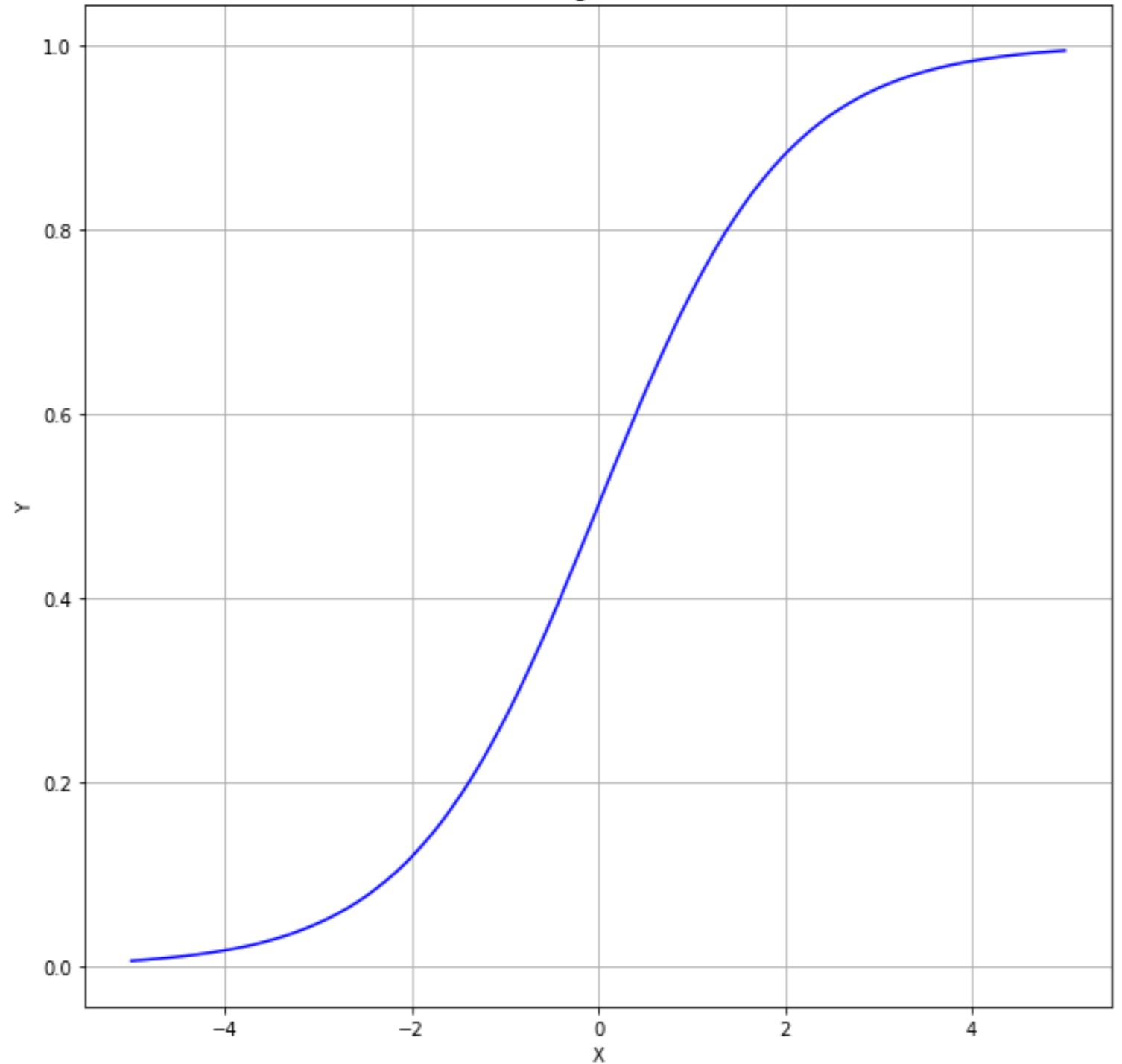
Neuron

$$a = \text{sum of inputs} * \text{weights} + \text{bias}$$

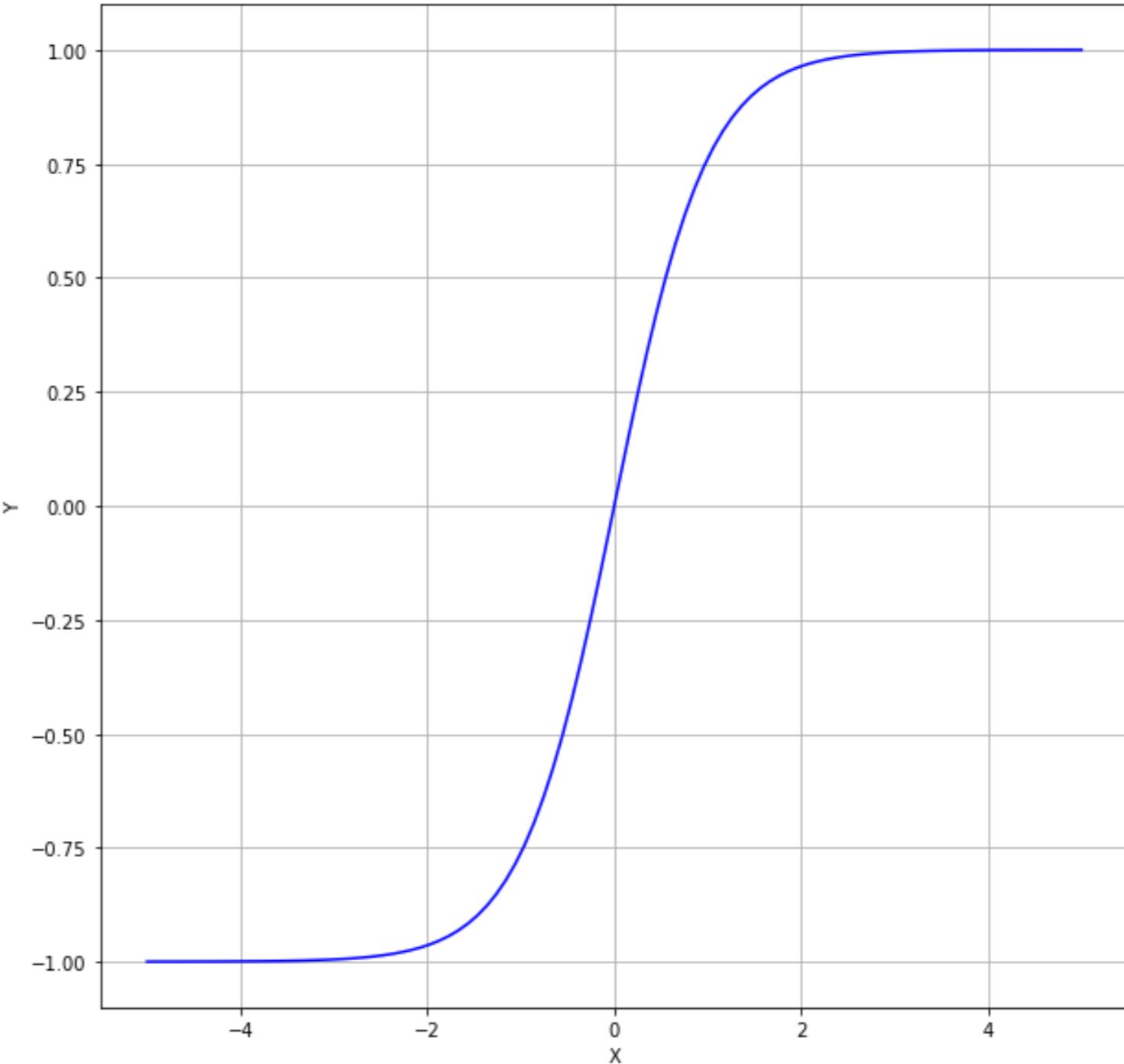


$$a = \text{sum of } x_i * w_i + b_1$$

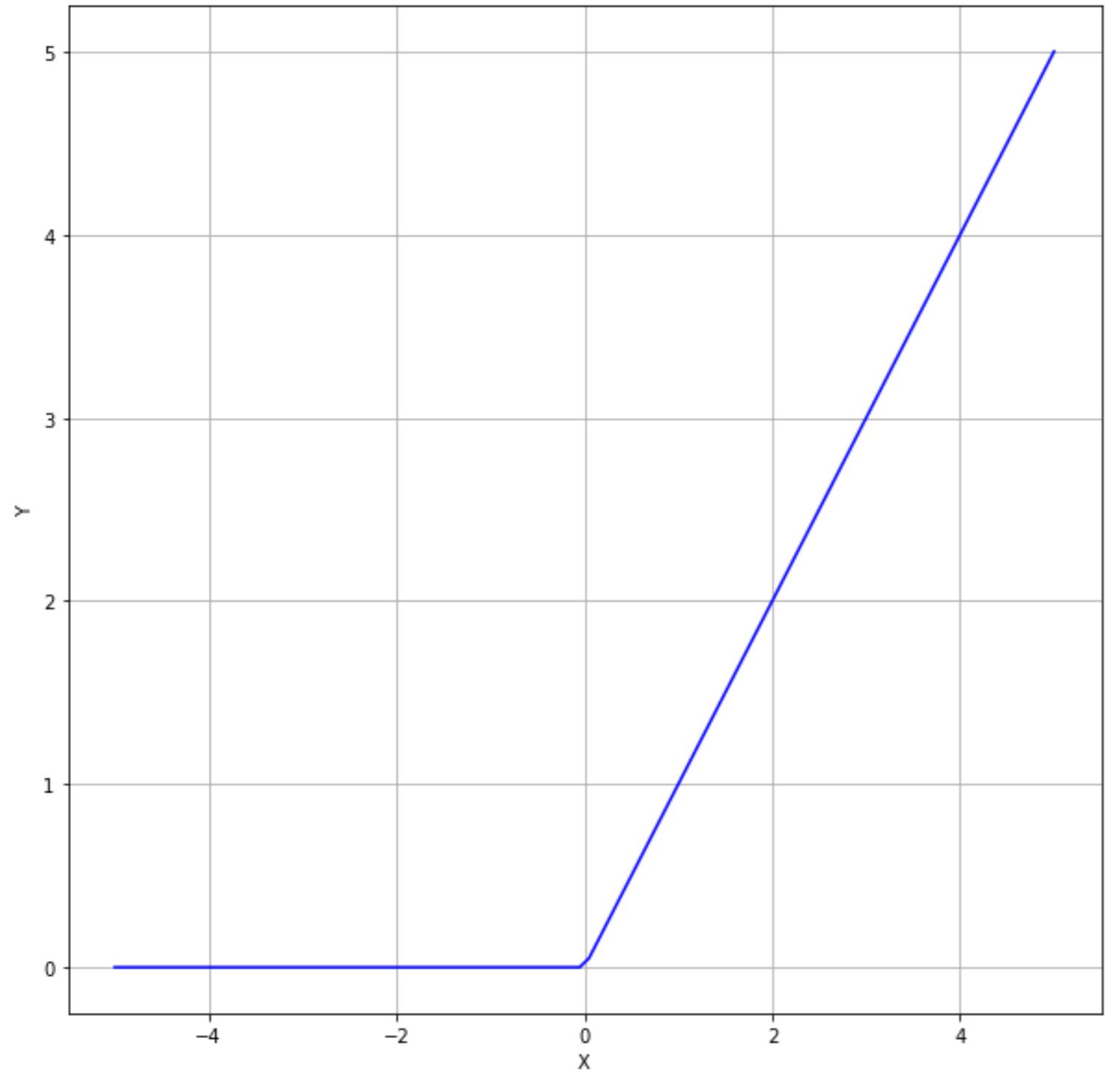
Sigmoid



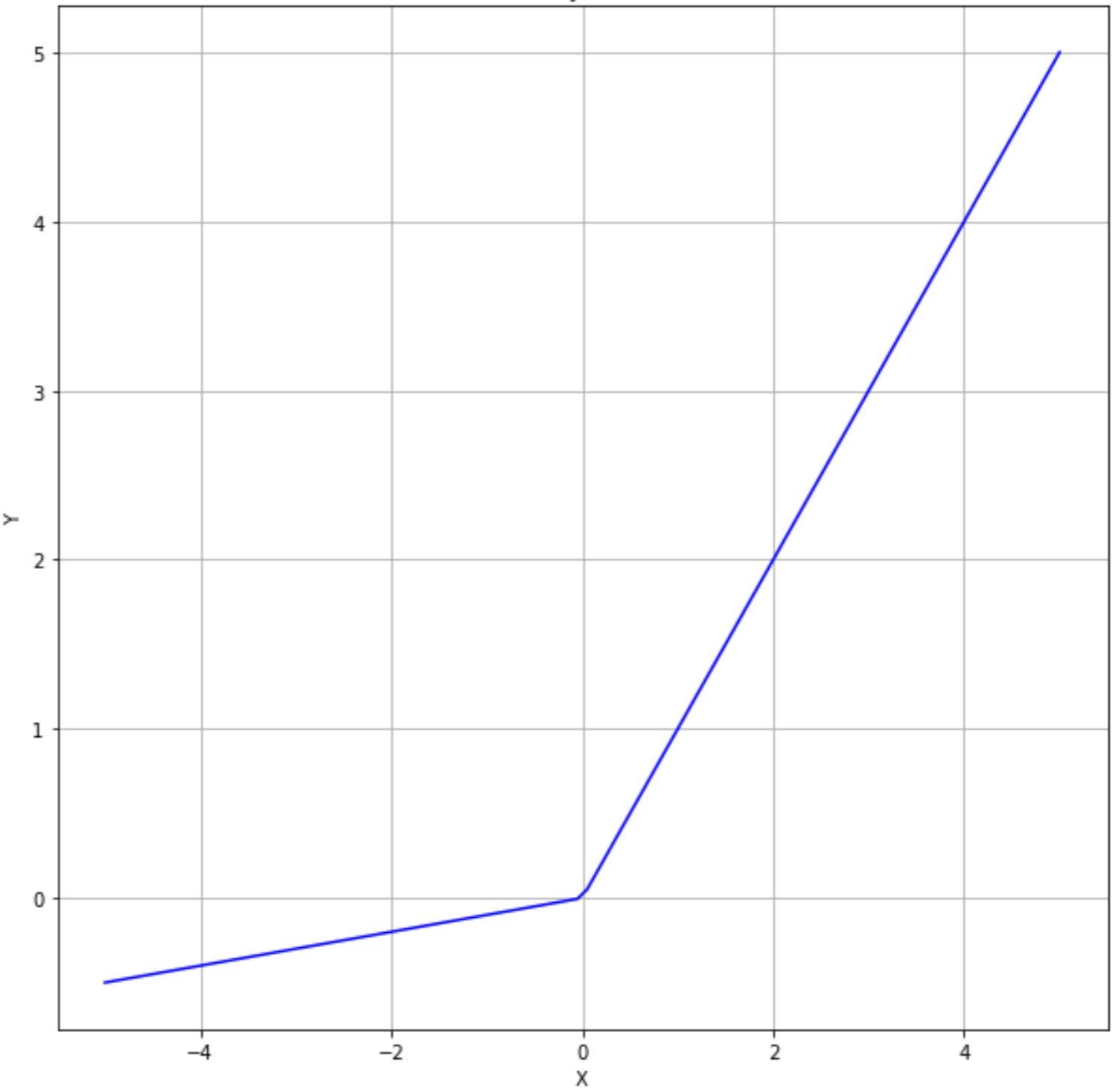
Tanh



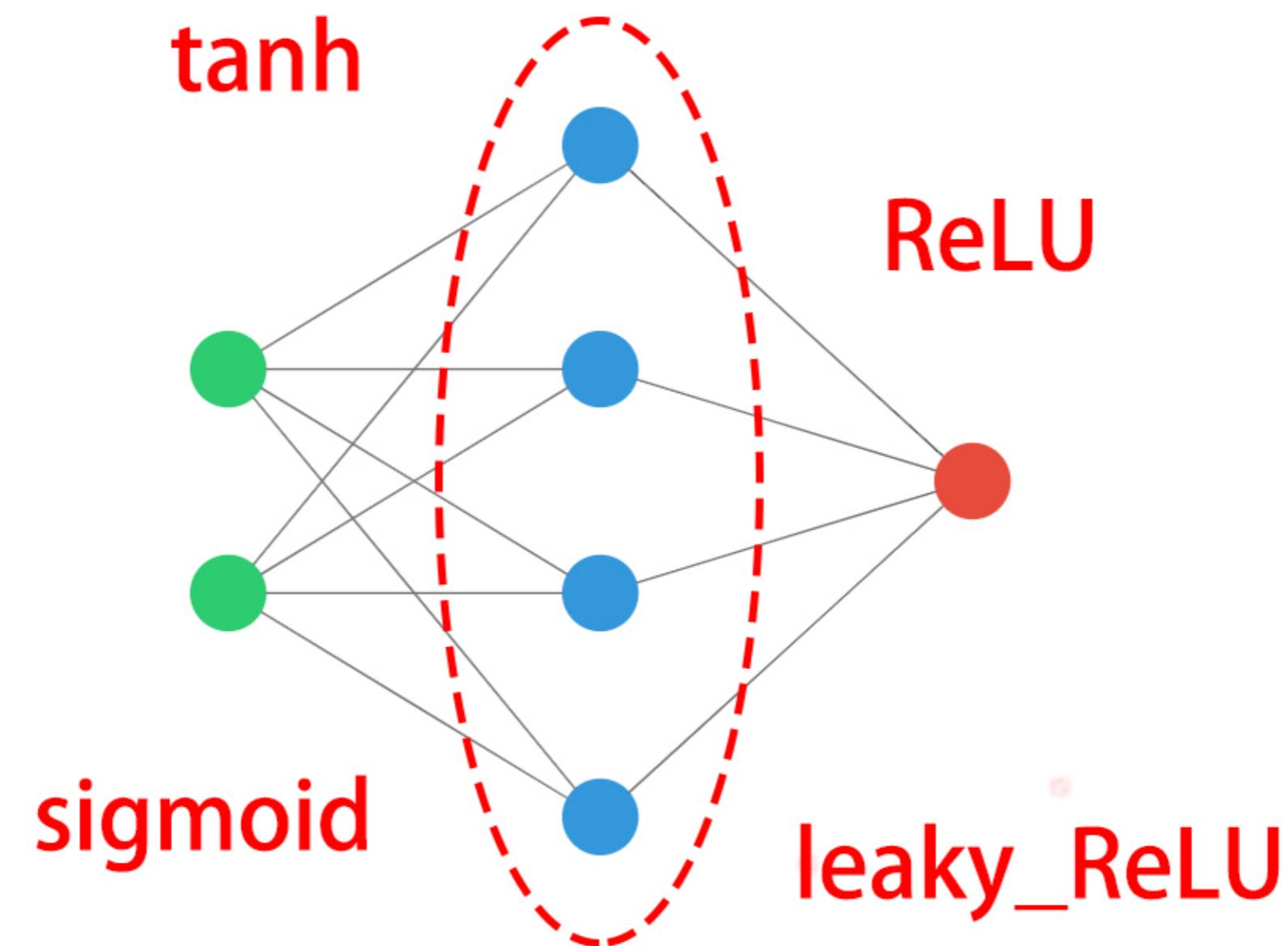
ReLU



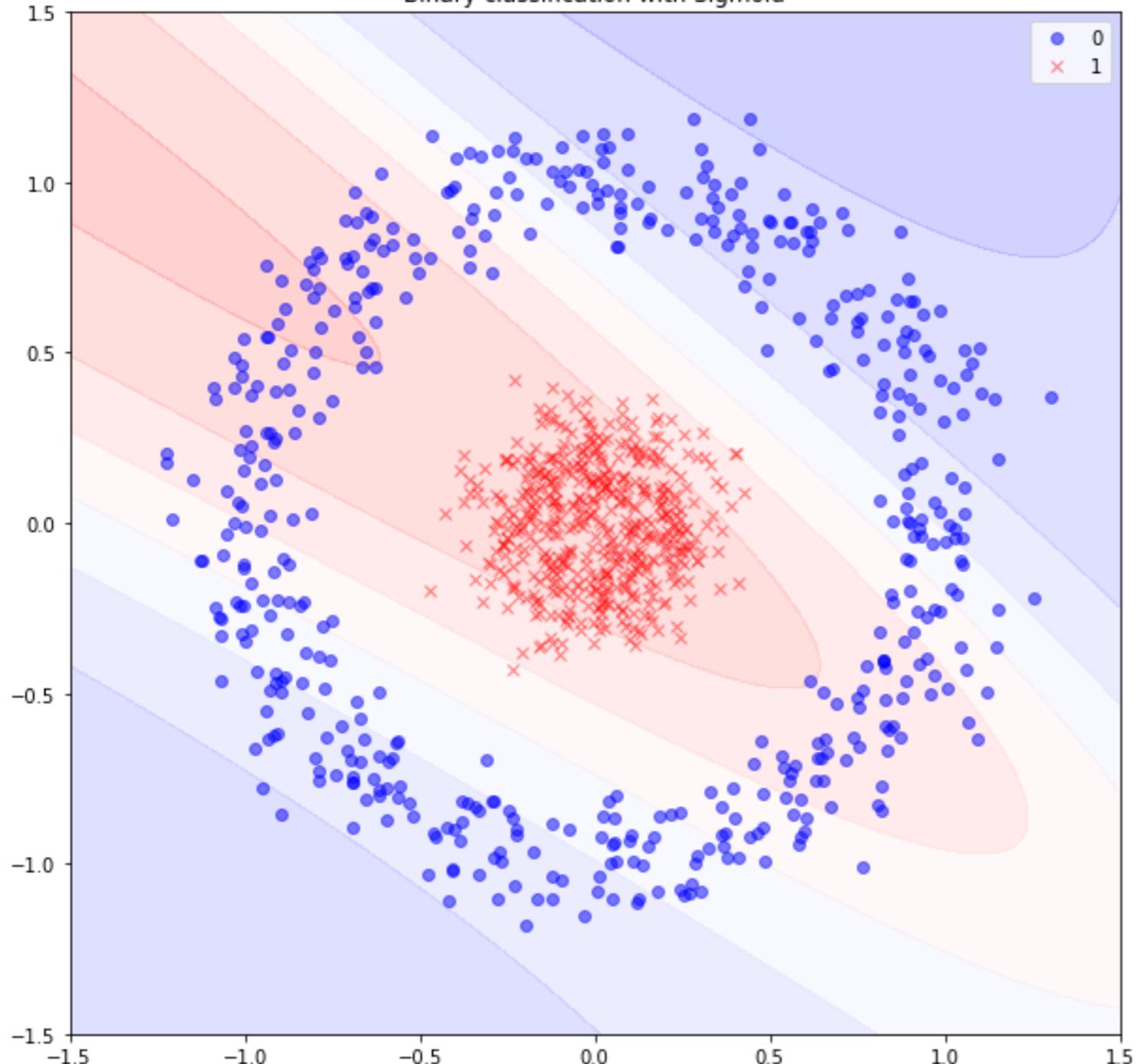
Leaky ReLU



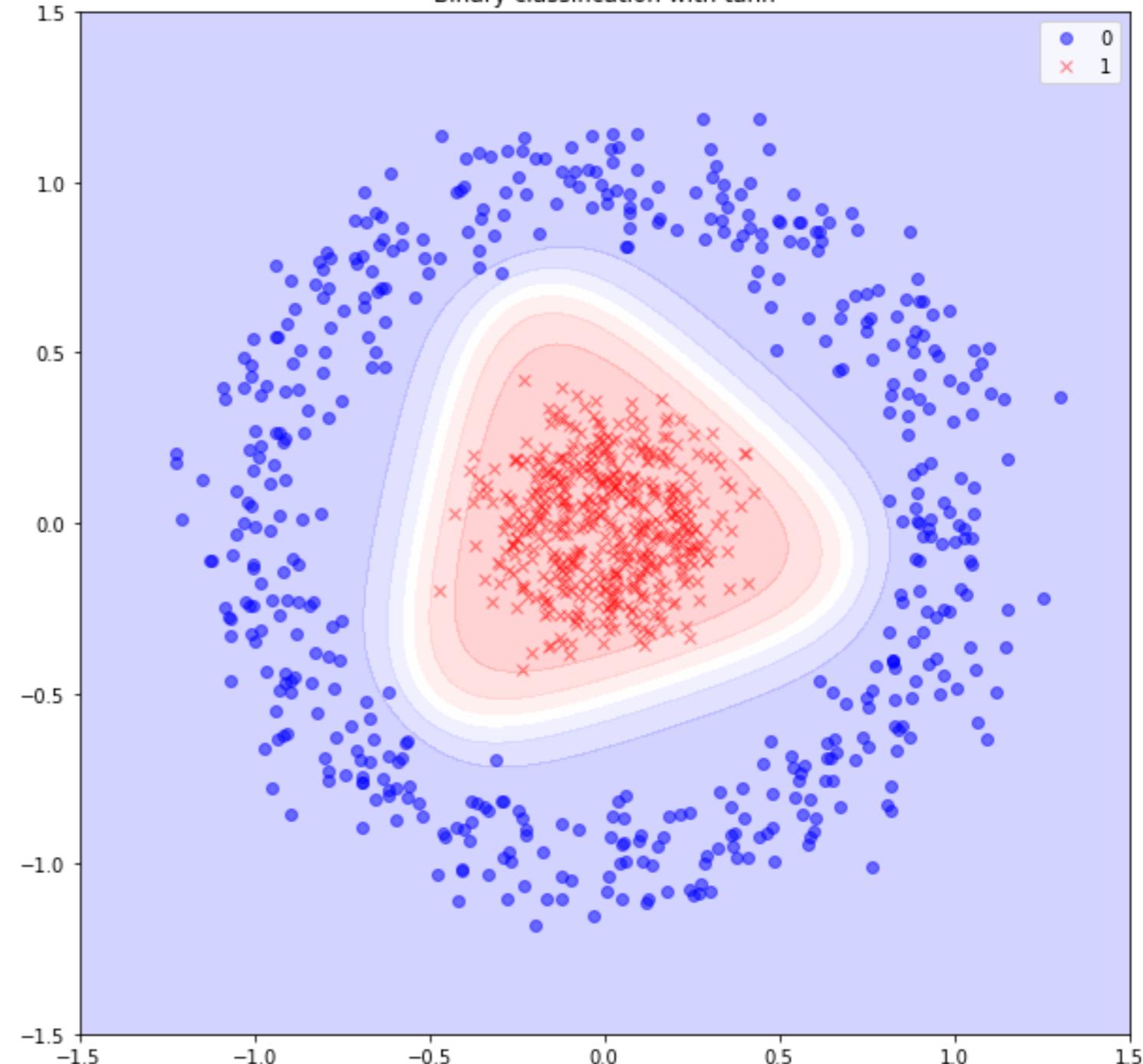
Effects of activation functions



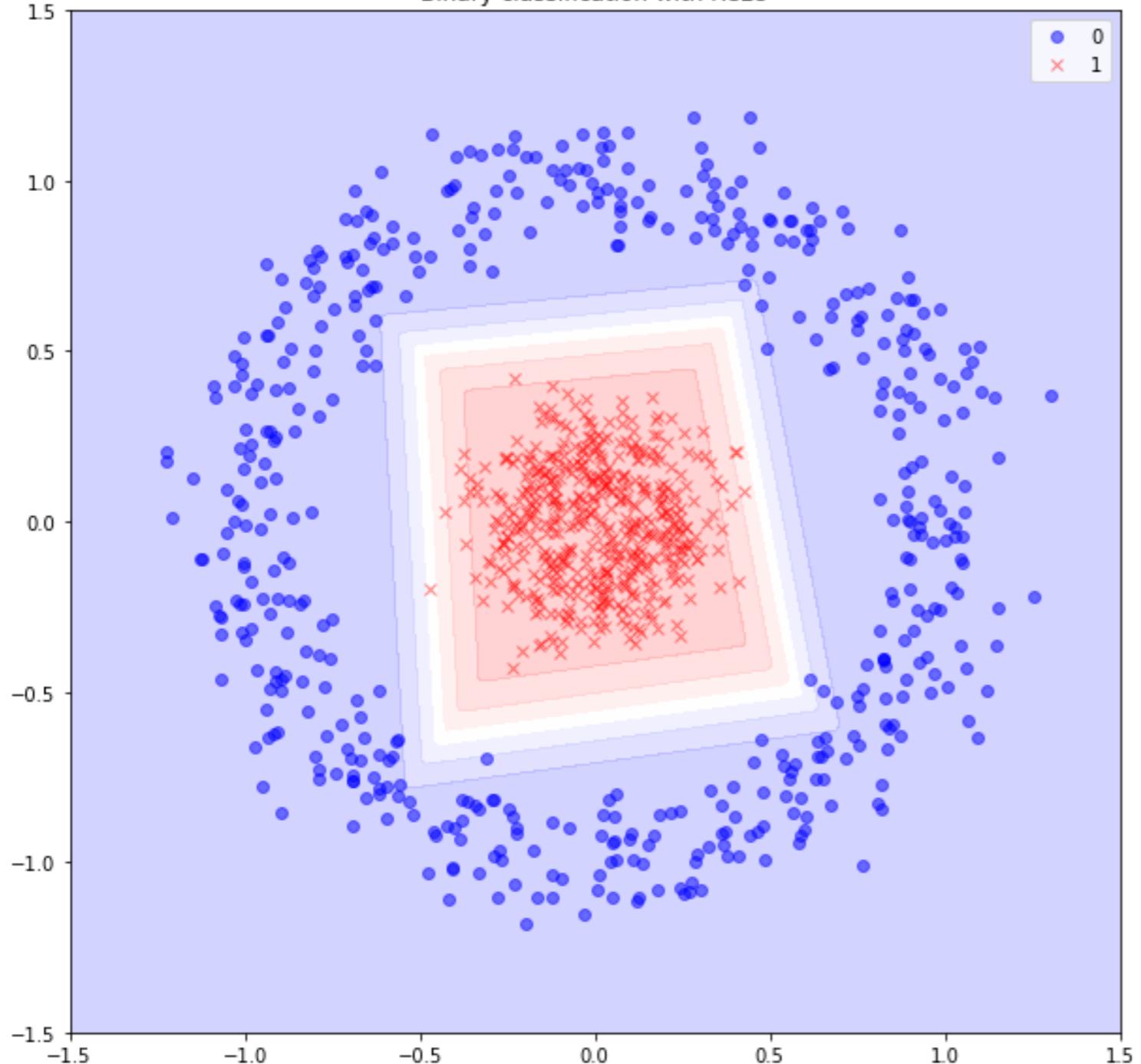
Binary classification with Sigmoid



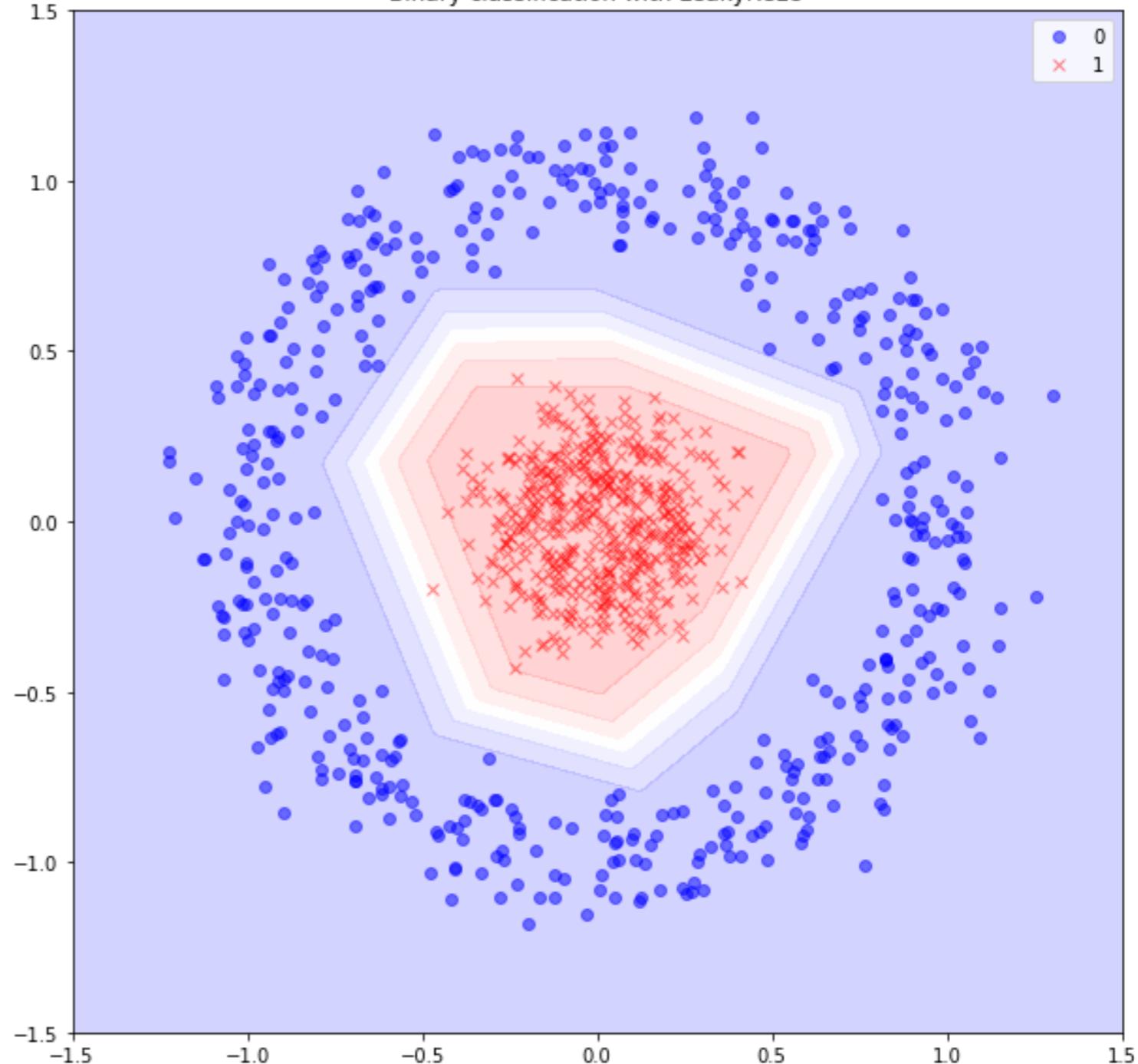
Binary classification with tanh



Binary classification with ReLU

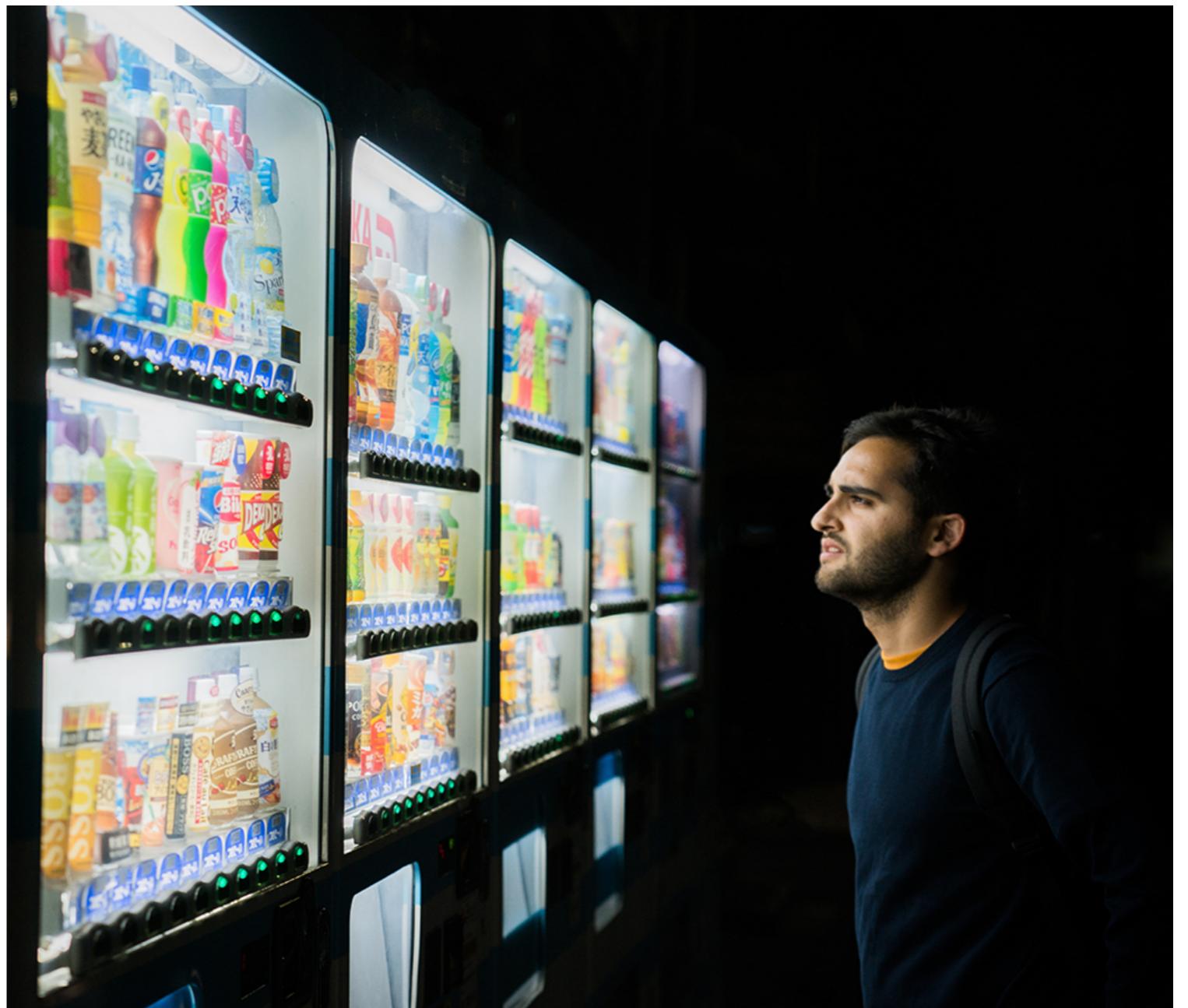


Binary classification with LeakyReLU



Which activation function to use?

- No magic formula
- Different properties
- Depends on our problem
- Goal to achieve in a given layer
- ReLU are a good first choice
- Sigmoids not recommended for deep models
- Tune with experimentation



Comparing activation functions

```
# Set a random seed
np.random.seed(1)

# Return a new model with the given activation
def get_model(act_function):

    model = Sequential()
    model.add(Dense(4, input_shape=(2,), activation=act_function))
    model.add(Dense(1, activation='sigmoid'))
    return model
```

Comparing activation functions

```
# Activation functions to try out
activations = ['relu', 'sigmoid', 'tanh']

# Dictionary to store results
activation_results = {}

for funct in activations:
    model = get_model(act_function=funct)
    history = model.fit(X_train, y_train,
                         validation_data=(X_test, y_test),
                         epochs=100, verbose=0)
    activation_results[funct] = history
```

Comparing activation functions

```
import pandas as pd

# Extract val_loss history of each activation function
val_loss_per_funct = {k:v.history['val_loss'] for k,v in activation_results.items()}

# Turn the dictionary into a pandas dataframe
val_loss_curves = pd.DataFrame(val_loss_per_funct)

# Plot the curves
val_loss_curves.plot(title='Loss per Activation function')
```

Let's practice!

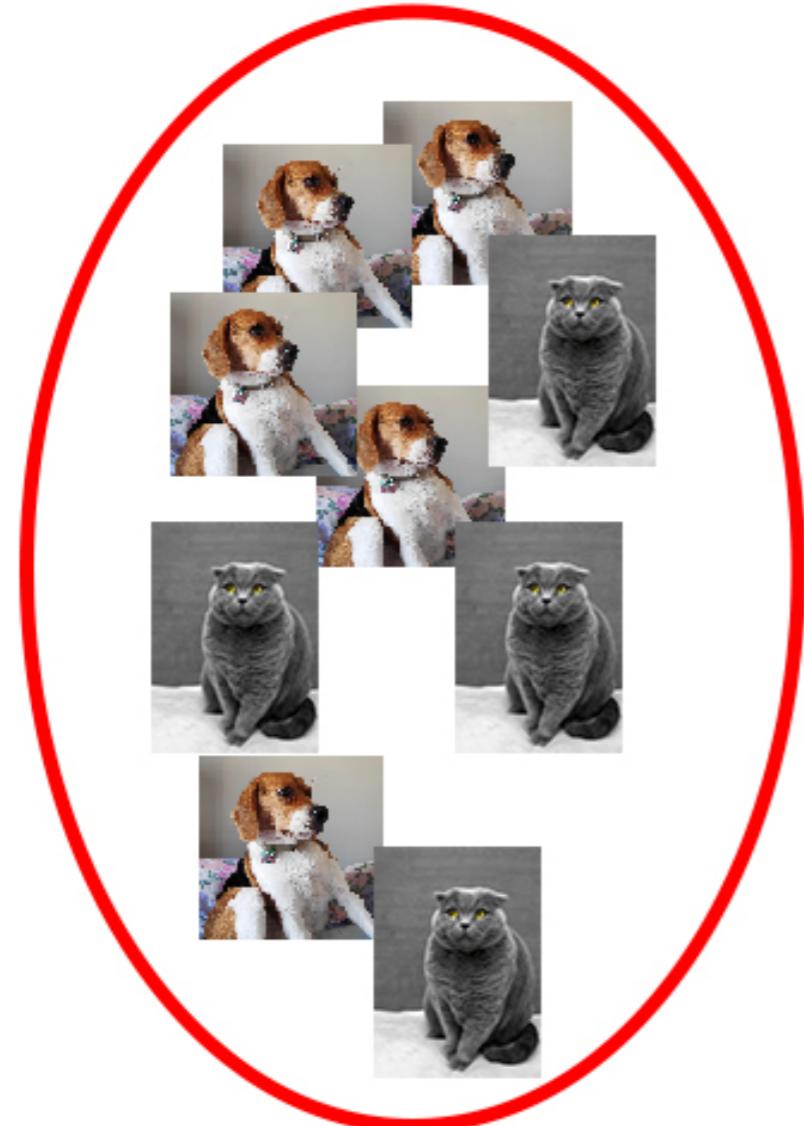
INTRODUCTION TO DEEP LEARNING WITH KERAS

Batch size and batch normalization

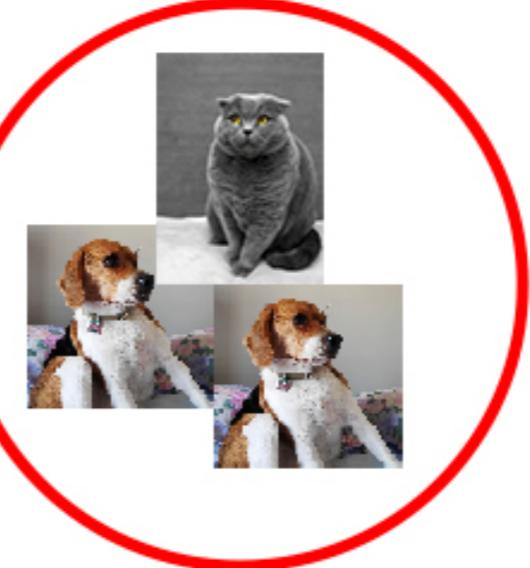
INTRODUCTION TO DEEP LEARNING WITH KERAS



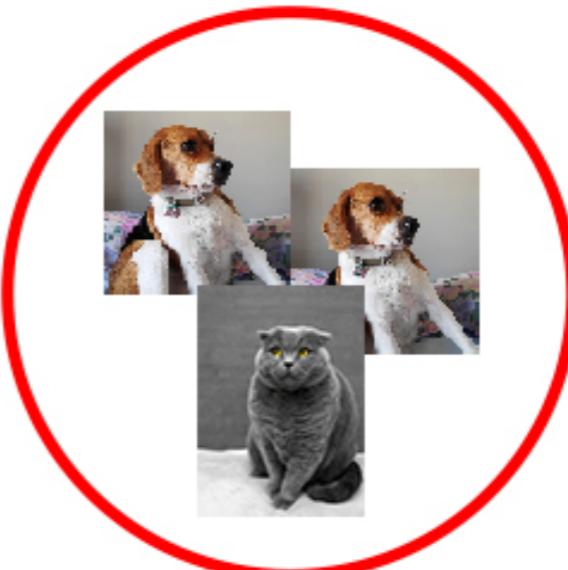
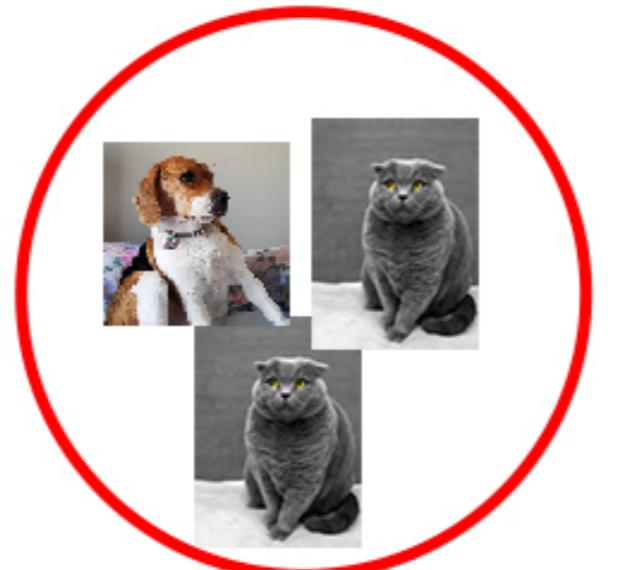
Miguel Esteban
Data Scientist & Founder



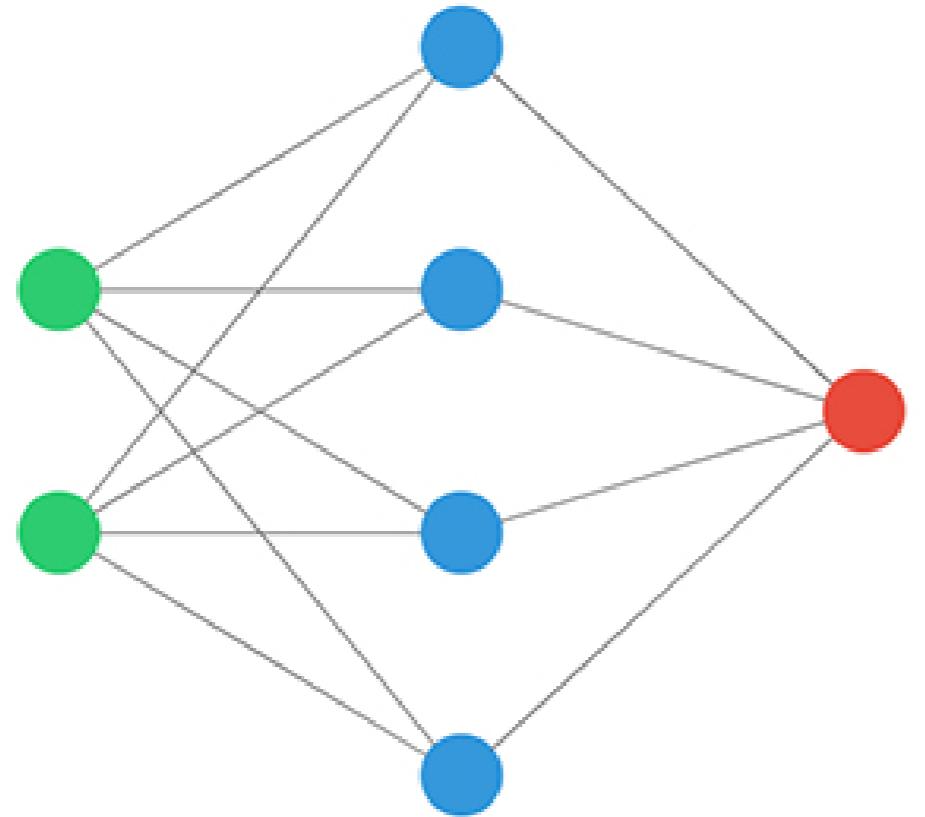
Batch



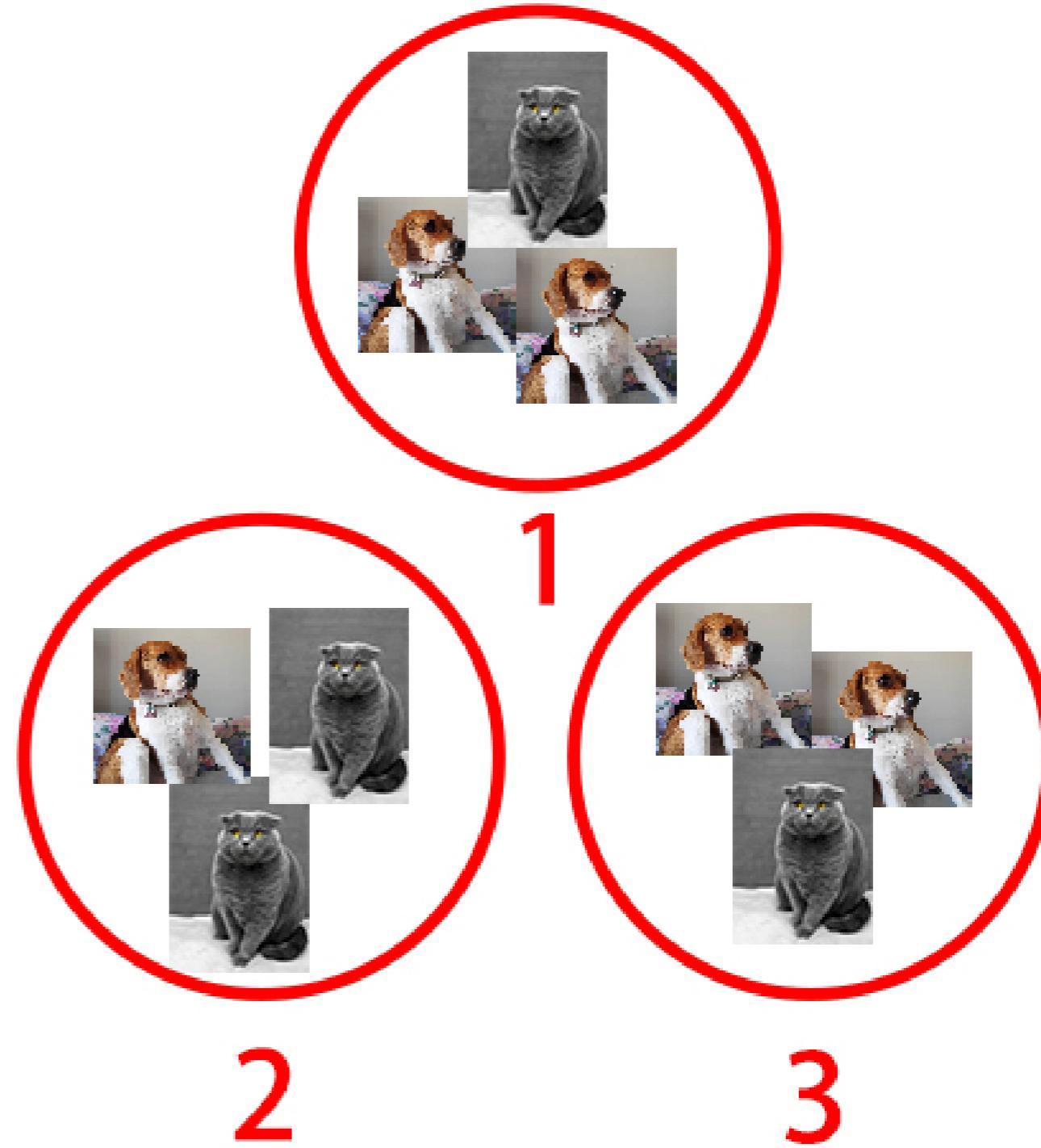
Mini-batches



The network is fed with 3 mini-batches



1 Epoch = 3 weight updates



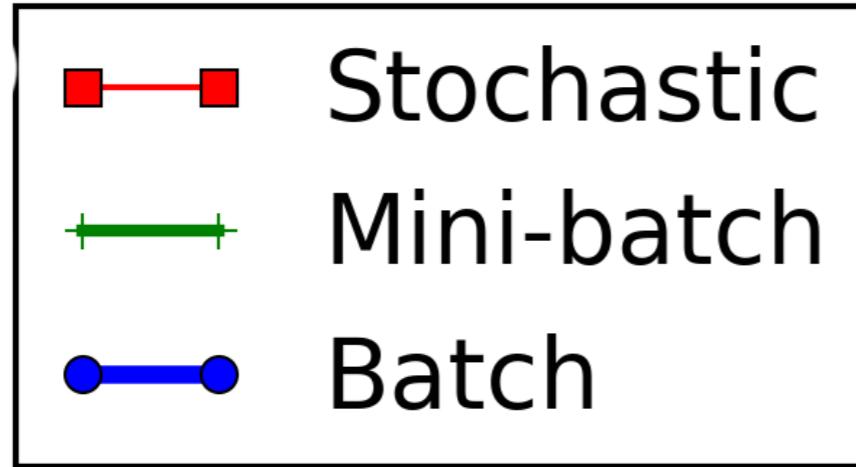
Mini-batches

Advantages

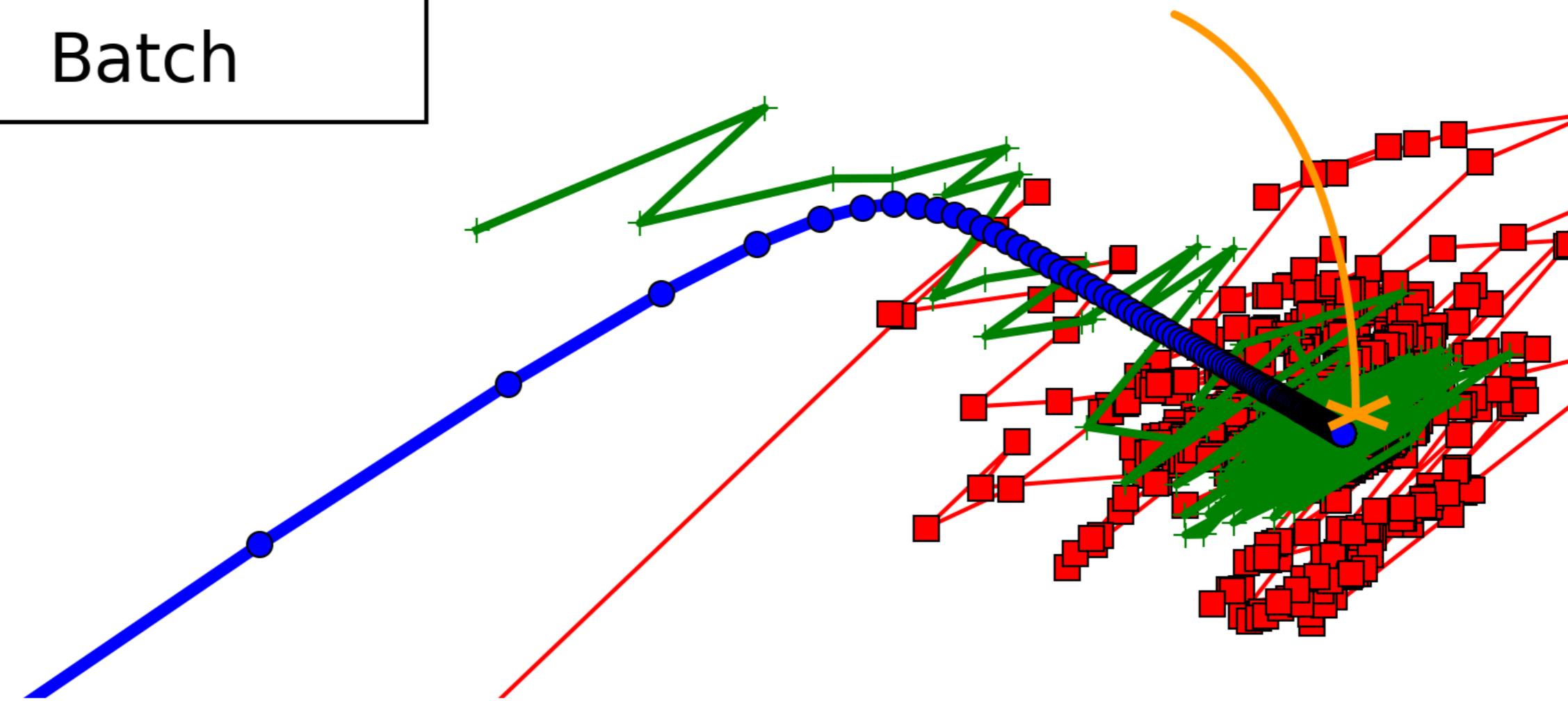
- Networks train faster (more weight updates in same amount of time)
- Less RAM memory required, can train on huge datasets
- Noise can help networks reach a lower error, escaping local minima

Disadvantages

- More iterations need to be run
- Need to be adjusted, we need to find a good batch size



Best value for our weights
(were loss is smaller)



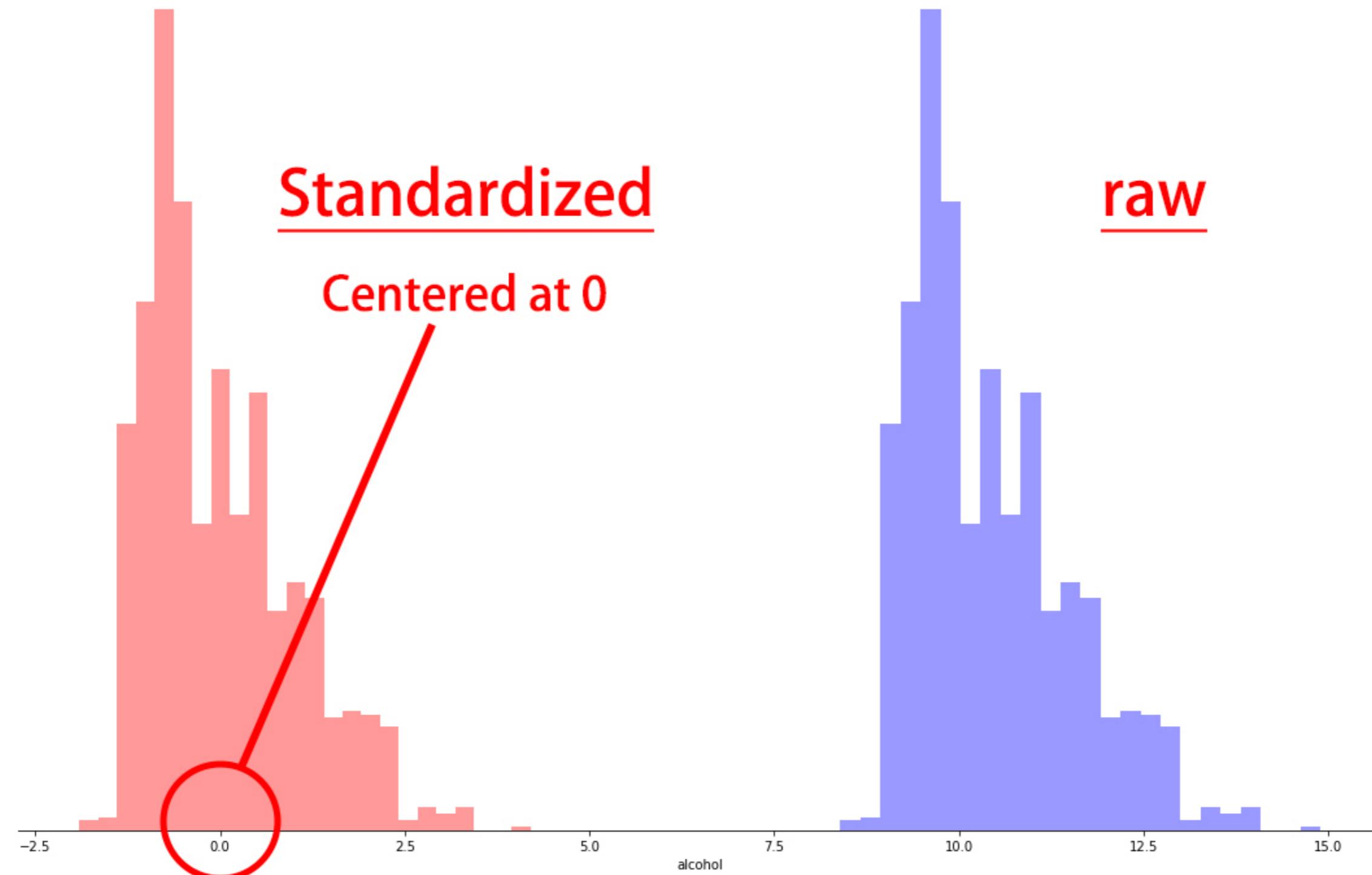
¹ Stack Exchange

Batch size in Keras

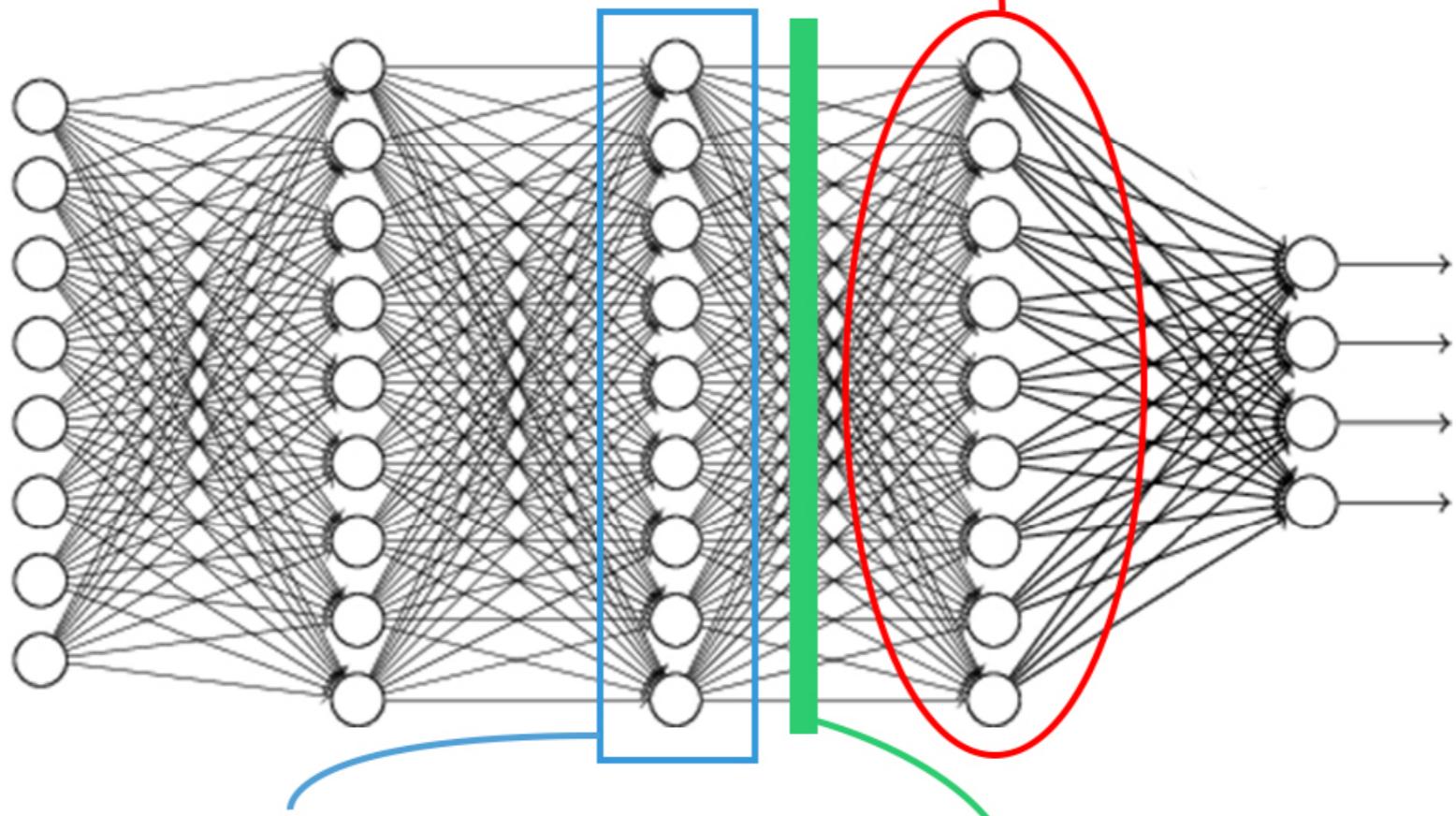
```
# Fitting an already built and compiled model  
model.fit(X_train, y_train, epochs=100, batch_size=128)  
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
```

Standardization (a normalization approach)

$\frac{\text{data} - \text{mean}}{\text{standard deviation}}$



1. This layer weights are trained based on the previous layer outputs it receives



2. But when this layer updates its weights via gradient descent, its outputs are also updated

3. Batch normalization makes sure that independently of the changes, the inputs to the **next layer** are normalized

Batch normalization advantages

- Improves gradient flow
- Allows higher learning rates
- Reduces dependence on weight initializations
- Acts as an unintended form of regularization
- Limits internal covariate shift

Batch normalization in Keras

```
# Import BatchNormalization from keras layers
from keras.layers import BatchNormalization

# Instantiate a Sequential model
model = Sequential()

# Add an input layer
model.add(Dense(3, input_shape=(2,), activation = 'relu'))

# Add batch normalization for the outputs of the layer above
model.add(BatchNormalization())

# Add an output layer
model.add(Dense(1, activation='sigmoid'))
```

Let's practice!

INTRODUCTION TO DEEP LEARNING WITH KERAS

Hyperparameter tuning

INTRODUCTION TO DEEP LEARNING WITH KERAS



Miguel Esteban
Data Scientist & Founder

Neural network hyperparameters

- Number of layers
- Number of neurons per layer
- Layer order
- Layer activations
- Batch sizes
- Learning rates
- Optimizers
- ...

Sklearn recap

```
# Import RandomizedSearchCV
from sklearn.model_selection import RandomizedSearchCV

# Instantiate your classifier
tree = DecisionTreeClassifier()

# Define a series of parameters to look over
params = {'max_depth':[3,None], "max_features":range(1,4), 'min_samples_leaf': range(1,4)}

# Perform random search with cross validation
tree_cv = RandomizedSearchCV(tree, params, cv=5)
tree_cv.fit(X,y)

# Print the best parameters
print(tree_cv.best_params_)
```

```
{'min_samples_leaf': 1, 'max_features': 3, 'max_depth': 3}
```

Turn a Keras model into a Sklearn estimator

```
# Function that creates our Keras model
def create_model(optimizer='adam', activation='relu'):

    model = Sequential()
    model.add(Dense(16, input_shape=(2, ), activation=activation))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=optimizer, loss='binary_crossentropy')
    return model

# Import sklearn wrapper from keras
from keras.wrappers.scikit_learn import KerasClassifier

# Create a model as a sklearn estimator
model = KerasClassifier(build_fn=create_model, epochs=6, batch_size=16)
```

Cross-validation

```
# Import cross_val_score
from sklearn.model_selection import cross_val_score

# Check how your keras model performs with 5 fold crossvalidation
kfold = cross_val_score(model, X, y, cv=5)

# Print the mean accuracy per fold
kfold.mean()
```

0.913333

```
# Print the standard deviation per fold
kfold.std()
```

0.110754

Tips for neural networks hyperparameter tuning

- Random search is preferred over grid search
- Don't use many epochs
- Use a smaller sample of your dataset
- Play with batch sizes, activations, optimizers and learning rates

Random search on Keras models

```
# Define a series of parameters
params = dict(optimizer=['sgd', 'adam'], epochs=3,
               batch_size=[5, 10, 20], activation=['relu', 'tanh'])

# Create a random search cv object and fit it to the data
random_search = RandomizedSearchCV(model, params_dist=params, cv=3)

random_search_results = random_search.fit(X, y)

# Print results
print("Best: %f using %s".format(random_search_results.best_score_,
random_search_results.best_params_))
```

```
Best: 0.94 using {'optimizer': 'adam', 'epochs': 3, 'batch_size': 10, 'activation': 'rel
```

Tuning other hyperparameters

```
def create_model(nl=1,nn=256):  
    model = Sequential()  
    model.add(Dense(16, input_shape=(2,), activation='relu'))  
    # Add as many hidden layers as specified in nl  
    for i in range(nl):  
        # Layers have nn neurons  
        model.add(Dense(nn, activation='relu'))  
    # End defining and compiling your model...
```

Tuning other hyperparameters

```
# Define parameters, named just like in create_model()  
params = dict(nl=[1, 2, 9], nn=[128,256,1000])  
  
# Repeat the random search...  
  
# Print results...
```

```
Best: 0.87 using {'nl': 2, 'nn': 128}
```

Let's tune some networks!

INTRODUCTION TO DEEP LEARNING WITH KERAS