

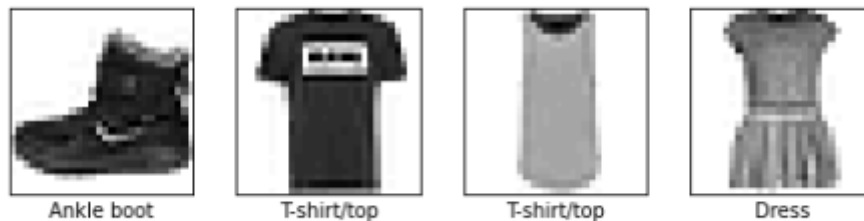
# Lecture5 Classification

## 1. Exercise Broken Down

### Task

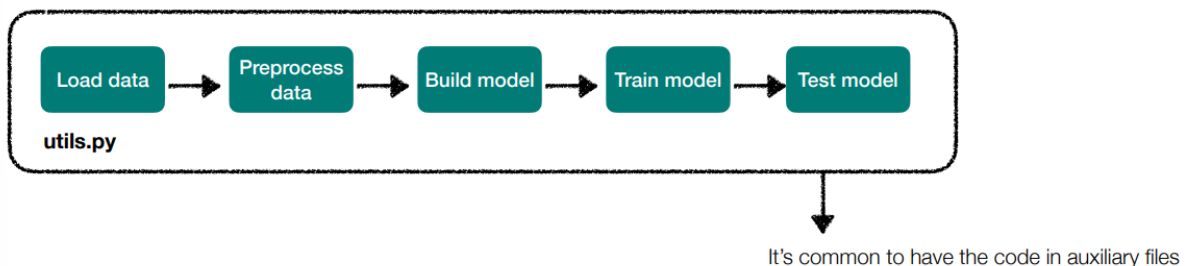
Fashion-MNIST classification with dense neural network

In this exercise, you will build a deep neural network to classify images from the Fashion-MNIST dataset using only dense layers. The Fashion-MNIST dataset is a collection of images of clothing items, where each image is a 28x28 grayscale image of one of 10 classes.



Single-label, multi-class classification problem.

### Load Data



### Pre-process Data

A lot of time typically goes into organizing data

```
def preprocess_data(x_train, y_train, x_test, y_test, val_size=10000):  
    # Normalize the data to be within the range [0,1]  
    x_train = x_train.astype('float32') / 255.  
    x_test = x_test.astype('float32') / 255.  
  
    # Slice the training data into train and validation sets  
    train_size = x_train.shape[0]  
    x_val = x_train[-val_size:]  
    y_val = y_train[-val_size:]  
    x_train = x_train[:-val_size]  
    y_train = y_train[:-val_size]  
  
    # Reshape data  
    x_train = x_train.reshape((train_size-val_size, 28 * 28))  
    x_val = x_val.reshape((val_size, 28 * 28))  
    x_test = x_test.reshape((val_size, 28 * 28))  
  
    return x_train, y_train, x_val, y_val, x_test, y_test
```

Common normalization for images

Should be parametric so you can change this values later

Common reshape to input images into Dense layers (rank3->rank2)



# Build Model

```
def build_model():  
    # Create a Keras sequential model following the instructions in the repo.  
    model = keras.Sequential([  
        keras.layers.Dense(128, activation='relu'),  
        keras.layers.Dense(10, activation='softmax')])  
  
    # Compile the model with a sparse_categorical_crossentropy loss,  
    # and assign an optimizer. Monitor the accuracy during training.  
    model.compile(optimizer='adam',  
                  loss='sparse_categorical_crossentropy',  
                  metrics=['accuracy'])  
  
    return model
```

How “free” the model is to learn representations

Number of classes of your problem

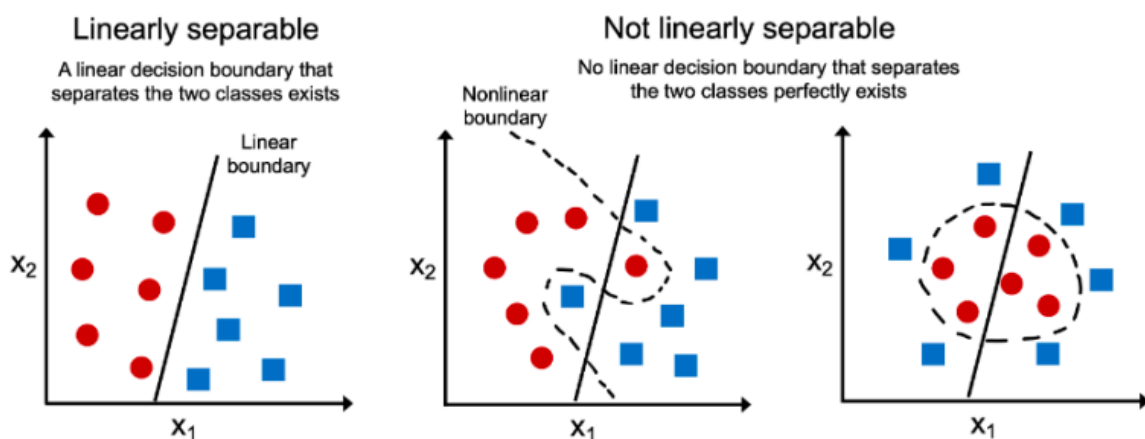
The optimizer: with Adam or rmsprop you're good to go

The loss

Activation functions

## Activation function

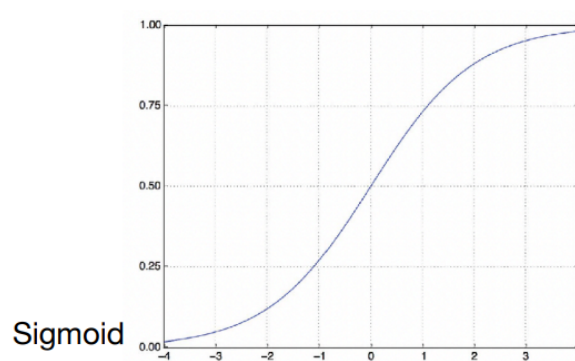
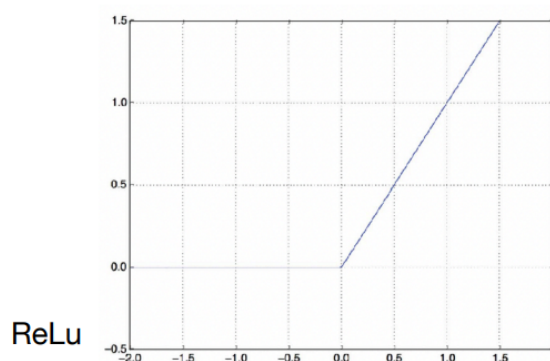
Dense layer without activation



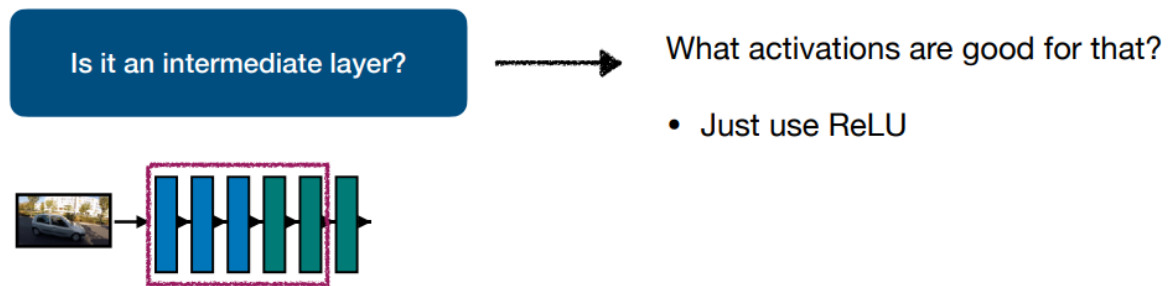
$$\text{output} = \text{dot}(\text{input}, W) + b$$

- If you stack many of this, the resulting transformation is still linear. Your model is very restricted on what it can learn.

Activation functions allow the network to learn **non-linear transformations**

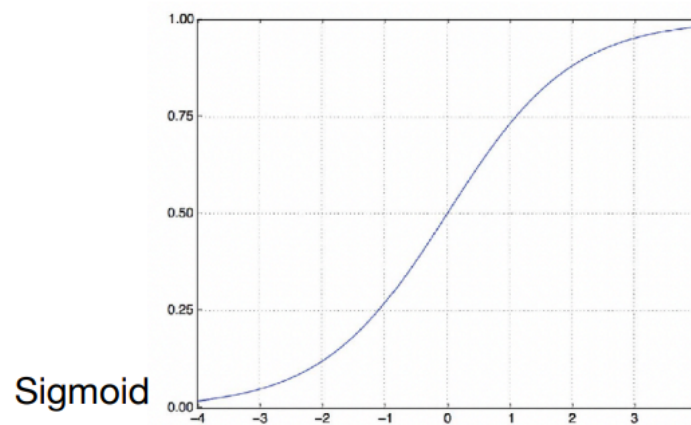


## Activation Function in the Intermediate Layer



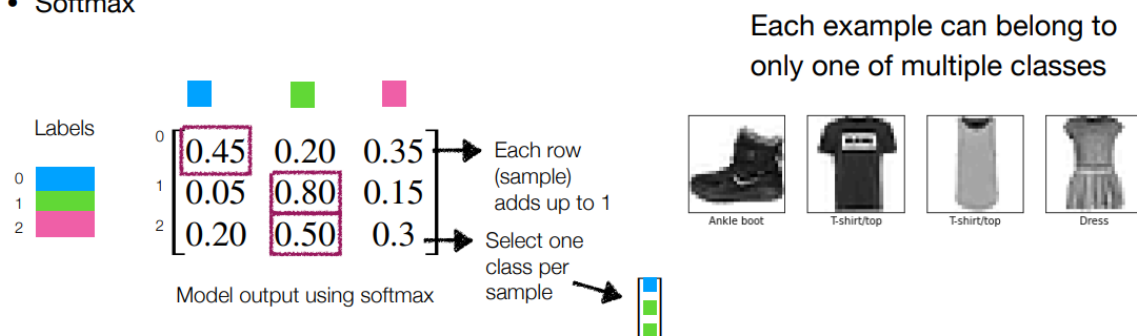
## Activation Function in the Last Layer

Binary-class Classification Problem: **Sigmoid**



Multi-class Classification Problem: **Softmax**

- Softmax



## Loss function

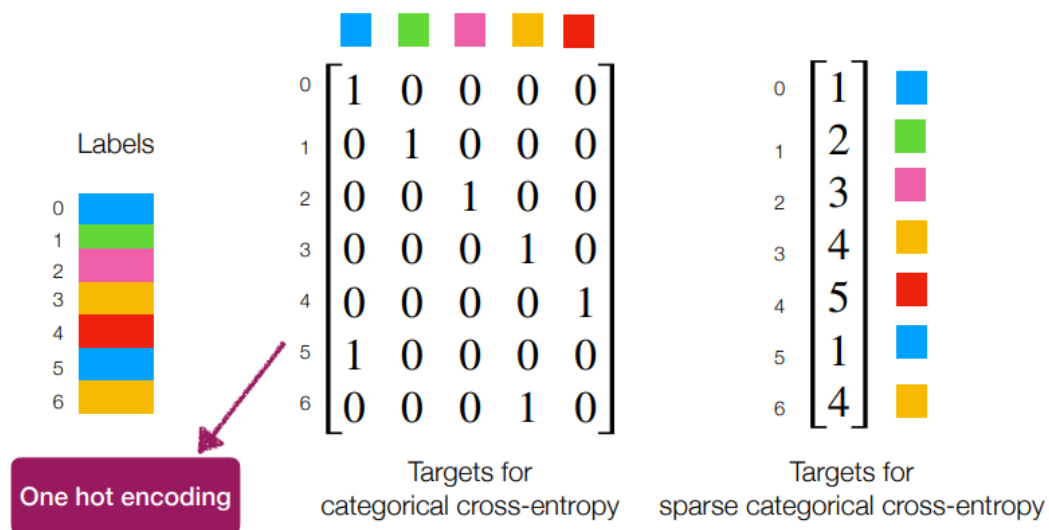
A measure of **how good** your model predictions are. Signal used in training.

The choice of the loss function depends on the problem.

- For our: Single-label, multi-class classification problem
  - Categorical cross-entropy
  - Sparse categorical cross-entropy

## Cross-entropy loss

A quantity that measures distances between probability distributions, or ground truth distributions and predictions.



- The cross-entropy loss and sparse categorical cross-entropy loss are mathematically equivalent, is just a different interface

For binary classification

$$L = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

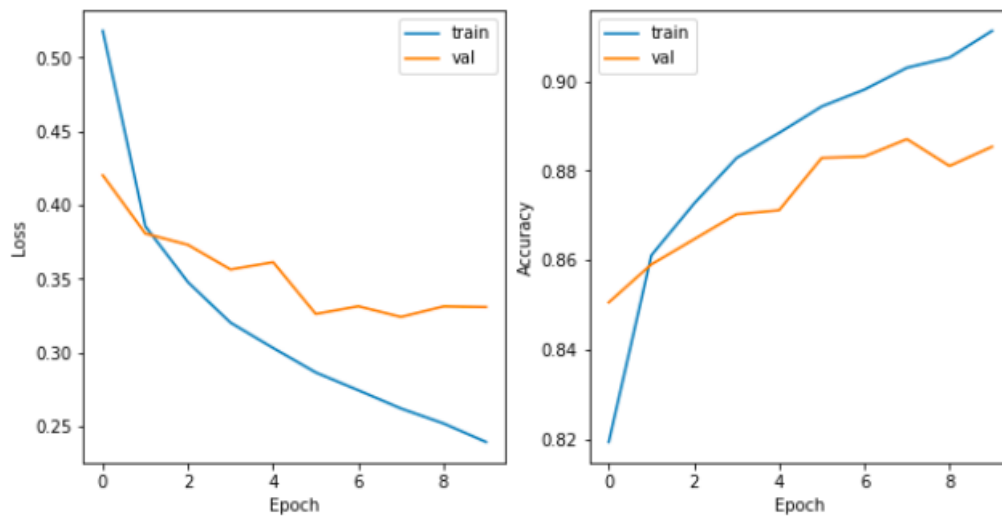
- $L$  is the loss for the entire dataset
- $N$  is the number of samples
- $y_i$  is the actual label (0 or 1)
- $p_i$  is the predicted probability of the class being 1 for the  $i_{th}$  sample

For multi-class classification

$$L = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^M y_{ic} \log(p_{ic})$$

- $L$  is the loss for the entire dataset
- $N$  is the number of samples
- $M$  is the number of classes
- $y_{ic}$ : a binary indicator (0 or 1) if class label  $c$  is the correct classification for observation  $i$
- $p_{ic}$ : is the predicted probability that observation  $i$  is of class  $c$

## Metrics



We use them to know how training is progressing.

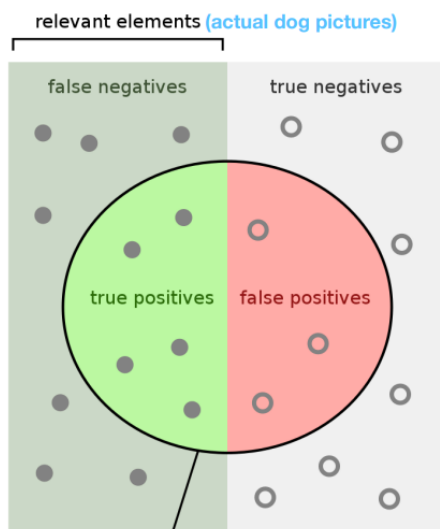
This signal is NOT used by the model during training.

## Accuracy



$$acc = \frac{\text{correct predictions}}{\text{total number of predictions}}$$

$$acc = \frac{4}{6} = 0.67$$



How many retrieved items are relevant?  
(Among our predictions, how many are actually dog pictures?)

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

How many relevant items are retrieved?  
(Among real dog pictures, how many predicted correctly?)

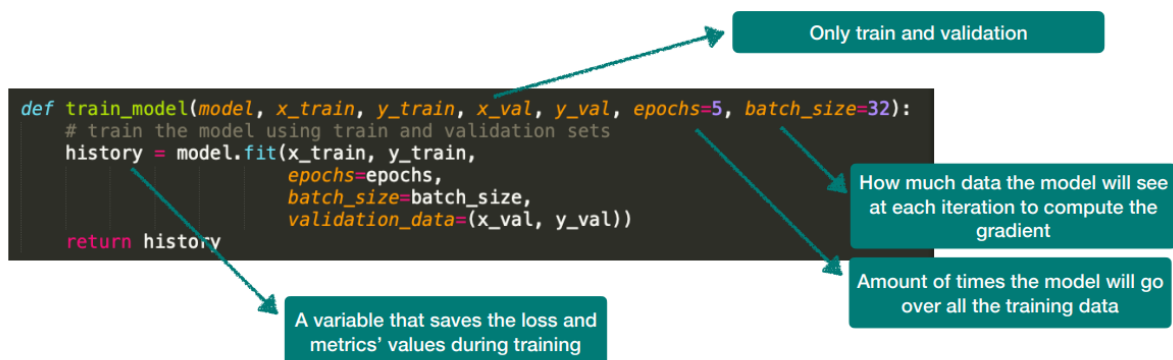
$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Best to compute Acc/P/R per class.

Ideally we want both P and R be high, but in practice, they are often in conflict.

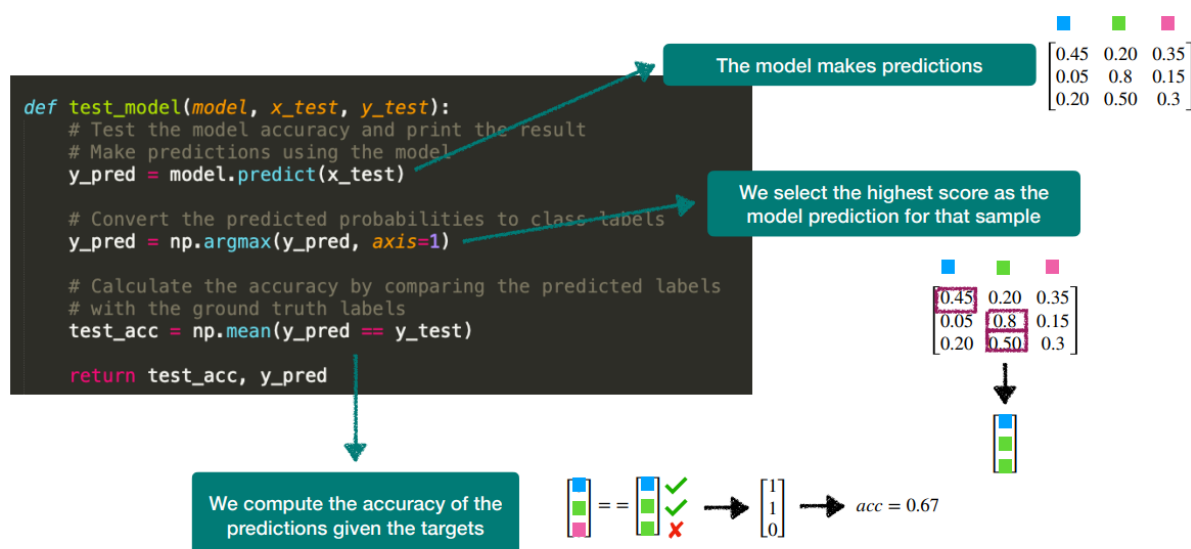
retrieved elements (model predicted these are dog pictures)

## Train Model



- The `fit()` function has other input parameters you can use to tune the training of your model.
- Take a look at the documentation!

## Test Model



## Classification Problem

## Single-label, Binary (=2-class)

## Are these mushroom poisonous?



Yes / No

- Binary classification is a special case of classification where `num_class == 2`.
- Treat it as such — use sparse categorical cross entropy, softmax, etc.

### Single-label, Multi-class

## What animal is this?

Input

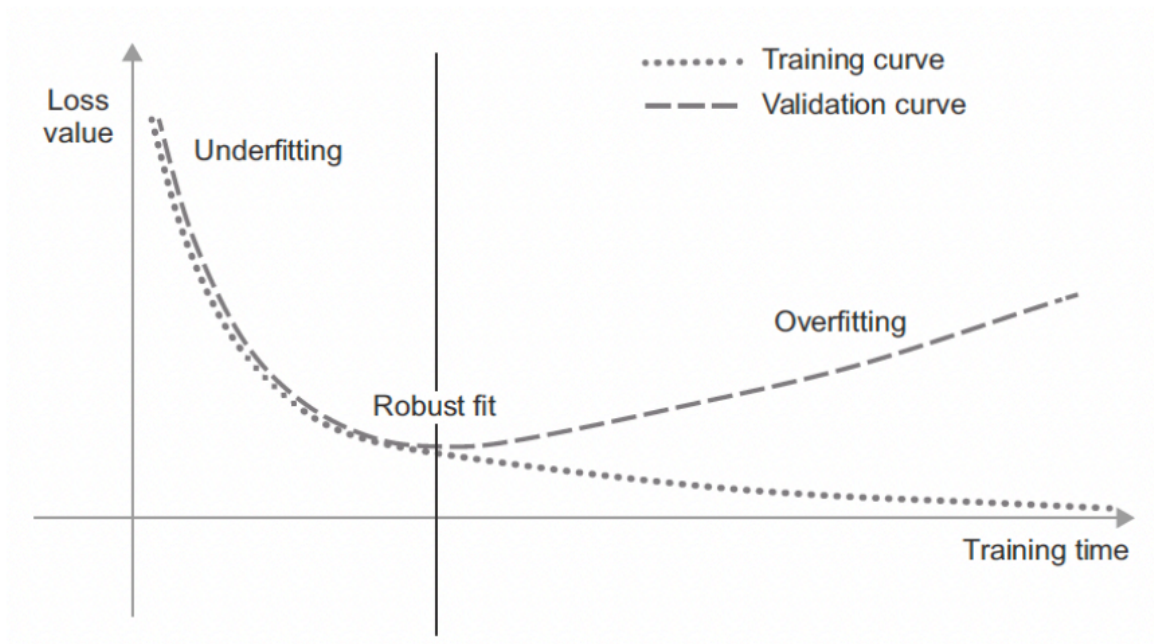


Predicted likelihoods

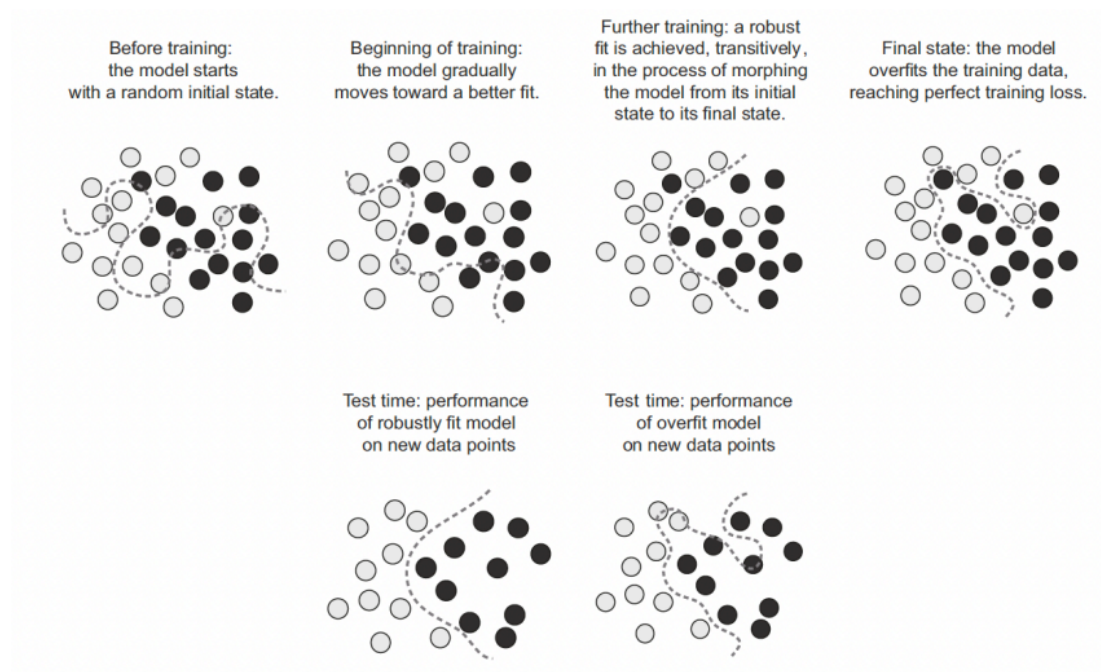
Cat	Dog	Rabbit	Duck
0.2	0.1	<b>0.7</b>	0.0



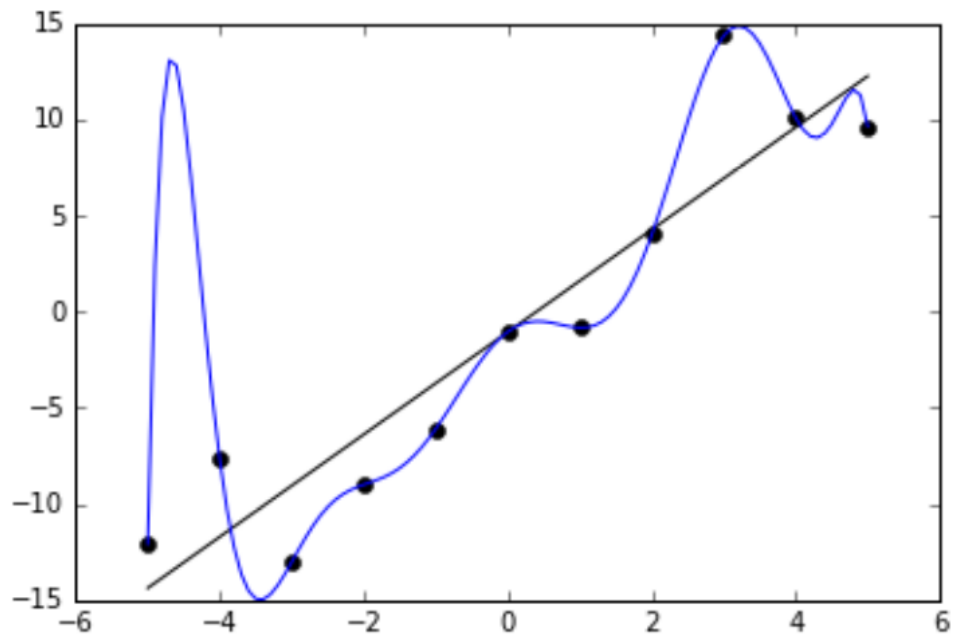
## 2. The fitting problem



### Overfitting



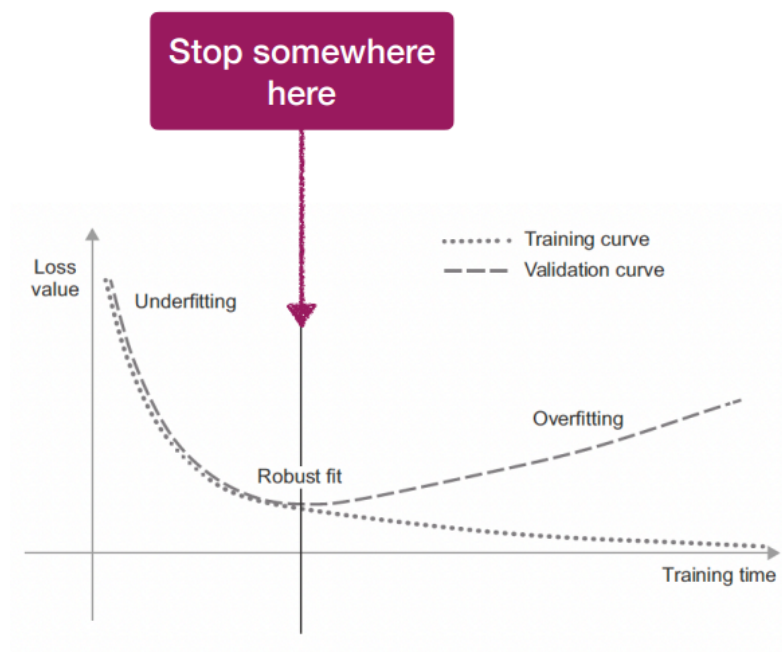




- The model focuses on what it has seen (training examples)
- Too much that it only memorizes the seen examples by focusing on irrelevant details
- Fail at unseen examples

## How can you mitigate overfitting?

Early stopping



- Looking at the train and validation losses, you can estimate the number of epochs needed for a robust fit (or use the early stopping callback in `fit()`)

All the things you can find in other resources (web, books, ..) — such as

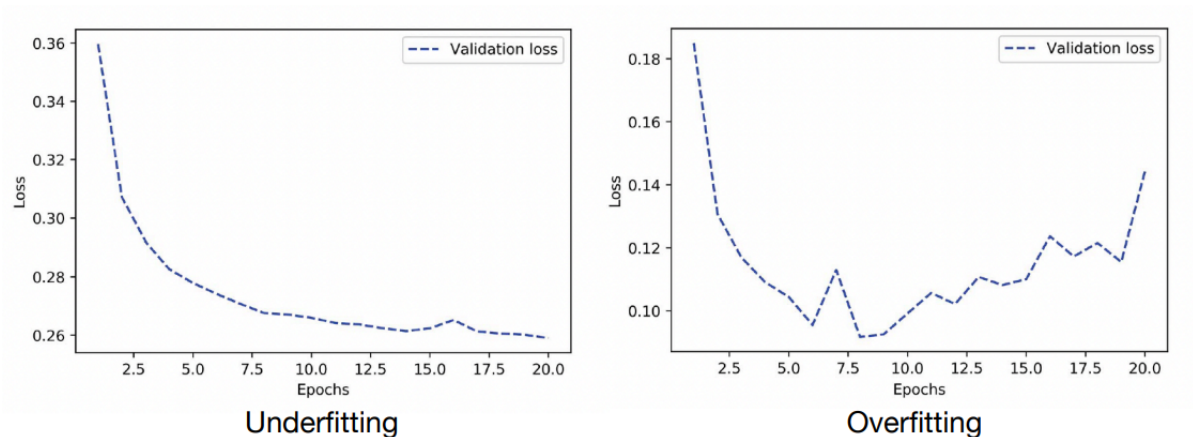
- regularizers
- adding noises
- dropout
- batch norm

- getting more data

are about preventing the model to memorize training examples by their irrelevant aspects

## Underfitting

It could also happen that your model doesn't overfit ever. Then it's too small!



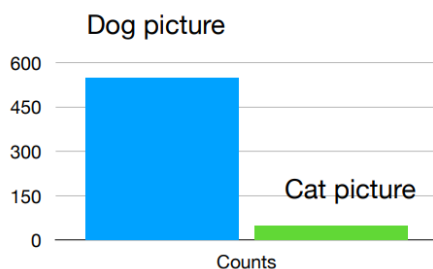
### How can you mitigate underfitting?

- We need a larger model capacity.
- Increase the model size or stack more layers until the model is able to overfit.

## 3. The generalization problem

### Distribution of training samples

Given a picture - dog or cat?



#### Eval Dataset.

Dog picture count: 550.

Cat picture count: 50.

Total count: 600.

$$acc = \frac{550}{600} = 92\%$$

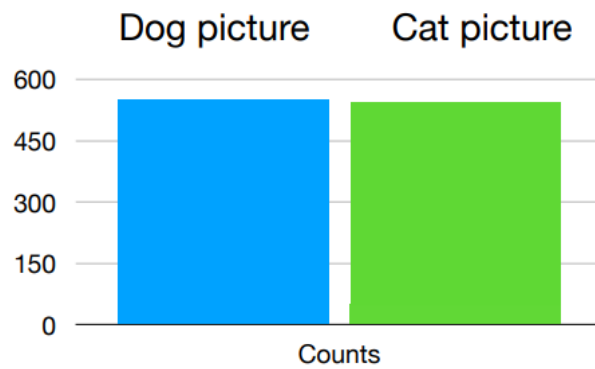


### The bias problem

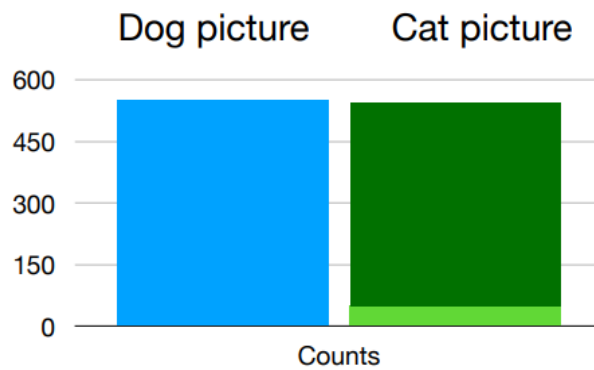
- Our models are not impartial.
- There are always biases we have to take care of.

### How can you mitigate biases?

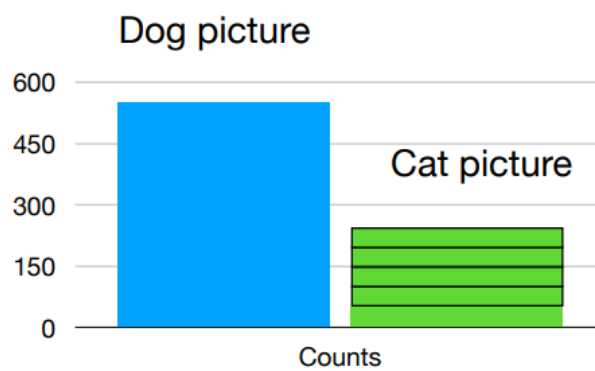
Data: Collect more data to balance the dataset.



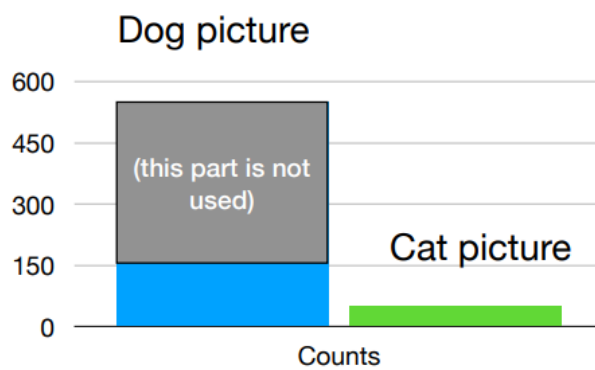
Data: Balance the distribution with synthetic data.



Sampling: Oversample the less represented class.



Sampling: Undersample the majority class.



Loss: Weight the loss to compensate for imbalance.

```
# Train model
class_weights = {0: 1, 1: 10}
```

```
history = model.fit(X_train,
                    y_train,
                    validation_data=(X_val, y_val),
                    epochs=epochs,
                    batch_size=batch_size,
                    class_weight=class_weights)
```