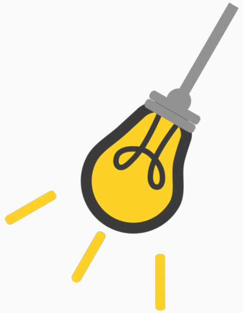
浙江大学爱丁堡大学联合学院
ZJU-UoE Institute

Lecture 16 - Improving CNN performance

Nicola Romanò - nicola.romano@ed.ac.uk

Learning objectives

- Describe common problems when training deep networks.
- Discuss good practices for generating good training data for deep networks.
- Describe and implement solutions to under/overfitting networks.



Introduction

Improving CNN performance

In the past lectures we have seen how to train a CNN to classify two *standard datasets*, MNIST and CIFAR-10.

These are good *toy* examples, but they do not reflect nowadays real-world applications.

In this and next lectures we will see how to improve the performance of CNN when using them on real-world datasets.

The problems

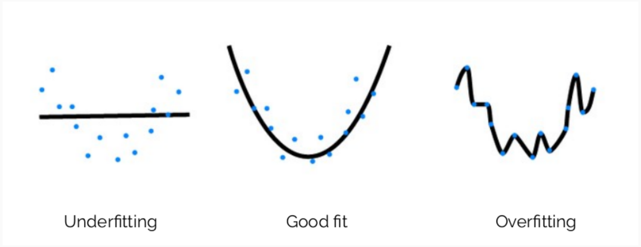
Getting the right data

- Providing good quality data is a key step in any ML project.
- Your CNN (or any other ML algorithm) will be trained on the data you provide, so any bias introduced there will be learned by the model.
- Obtaining a lot of good training data is expensive and not always possible.

How do we ensure data quality?

The bias-variance tradeoff

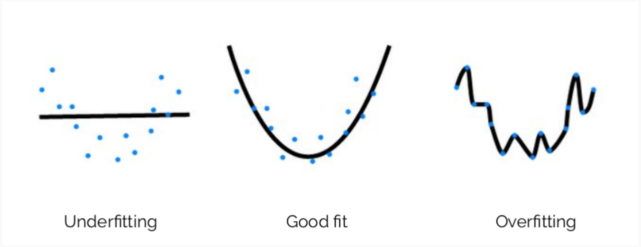
We want to train our model to perform some task. However, just like any statistical model, we don't want to **overfit**.



In ML, we often describe this in terms of **bias** and **variance** errors.

The bias-variance tradeoff

We want to train our model to perform some task. However, just like any statistical model, we don't want to **overfit**.



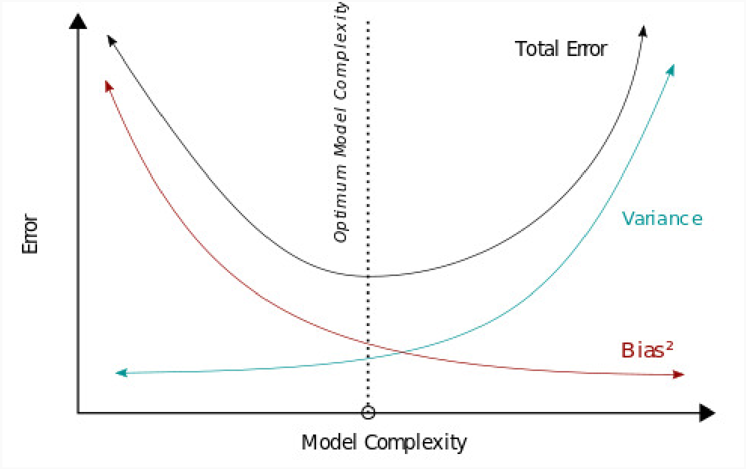
In ML, we often describe this in terms of **bias** and **variance** errors.

- **Bias** derives from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).
- **Variance** derives from sensitivity to small fluctuations in the training set. High variance may result from an algorithm modeling the random noise in the training data (overfitting).

(Adapted from Wikipedia)

The bias-variance tradeoff

We want to train our model to perform some task. However, just like any statistical model, we don't want to **overfit**.



How do we detect and fix under- and over-fitting?

Selecting hyperparameters

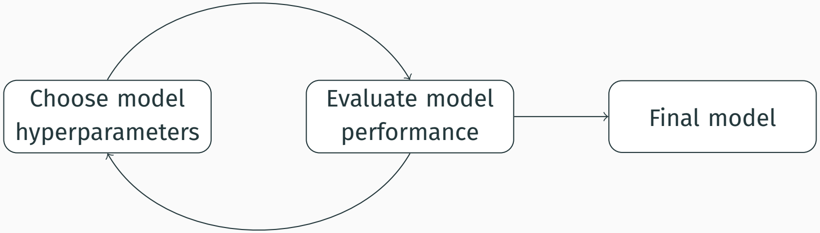
- Selecting the right hyperparameters is a crucial step in any ML project.
- The hyperparameters you choose will determine the performance of your model.
- There are a lot of hyperparameters to choose for a CNN, for example, number of hidden layers, number of nodes in each layer, number of filters, number of epochs, learning rate, etc.

How to find the best hyperparameters? (next lecture)

Data quality

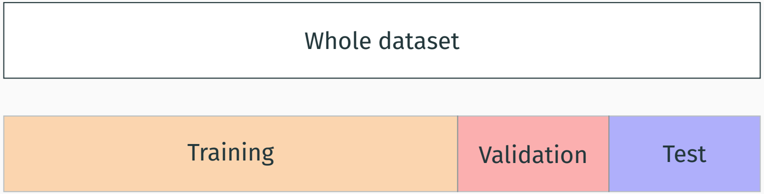
Deep learning is an iterative process

Very often when you are building a deep learning model you will need to iterate through several version of your model to find the good one to use.



Data splitting

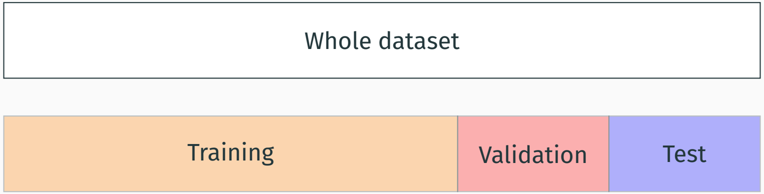
We need to ensure our data is correctly split into training, validation and test sets.



- Optimize the model by training it on the training set and evaluating it on the validation set.
- The validation set is not seen during training, so we can compare multiple models by evaluating them on it.
- Once we have a final model we can evaluate it on the test set.

Data splitting

We need to ensure our data is correctly split into training, validation and test sets.



- Optimize the model by training it on the training set and evaluating it on the validation set.
- The validation set is not seen during training, so we can compare multiple models by evaluating them on it.
- Once we have a final model we can evaluate it on the test set.
- In Python you can use the `test_train_split` function in `sklearn.model_selection` module to split your data into training and test set (call it twice to further split the training set into training and validation).

How to split the data?

- Traditionally, 60/20/20 (or 70/20/10) train/validation/test split has been used.
- If you are dealing with extremely large datasets (millions or hundreds of millions of observations) you will probably use much smaller validation and test sets.
- In those cases maybe 98% or even more can be used as training set, since you will still have enough data in the validation and test set.

Match training and test set

It is important to train the model with data that is similar to the test data (and to the new data you want to use the model on).

Sometimes training data is hard to come by and training is performed using available sources that might not match your own data.

This might make your model inefficient.

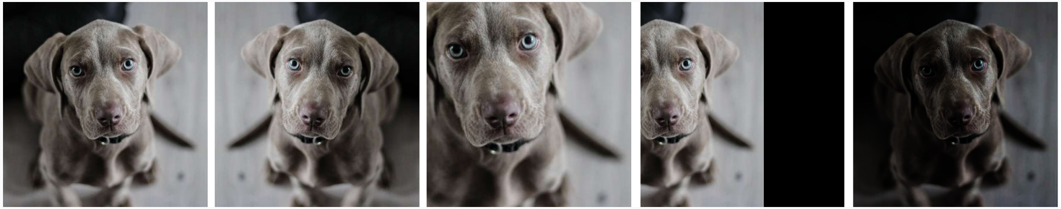
Data augmentation

What if you only have a small amount of training data?

You can use **data augmentation** to increase the amount of training data you have, by generating synthetic data!

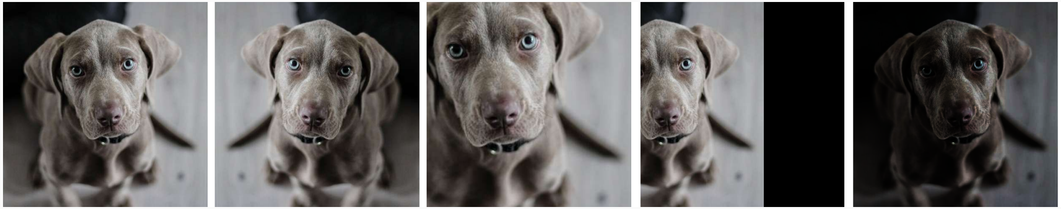
For images, data augmentation is done by rotating, flipping, and cropping or otherwise distorting the images.

Data augmentation - an example



The original image (left) can be flipped, rotated,cropped etc.

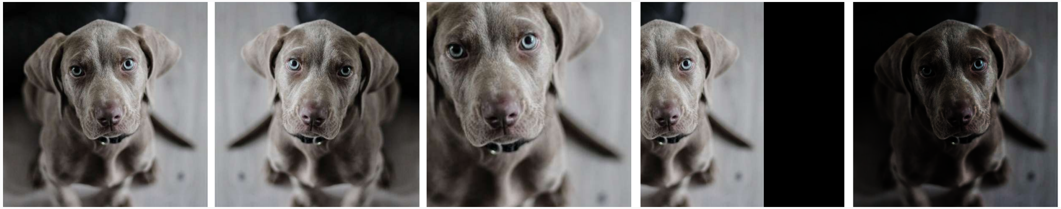
Data augmentation - an example



The original image (left) can be flipped, rotated,cropped etc.

Make sure your augmentation makes sense!

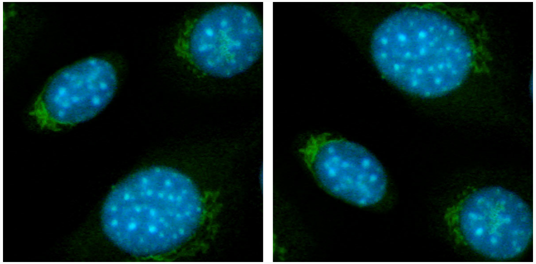
Data augmentation - an example



The original image (left) can be flipped, rotated,cropped etc.

Make sure your augmentation makes sense!

This is fine, though!



Augmentation in Keras

The tf.keras.preprocessing.image.ImageDataGenerator class can be used to augment image data.

```
# Load data
x, y = ...

# Create a generator allowing translation, zooming and rotation
generator = ImageDataGenerator(height_shift_range=[-50,50],
       width_shift_range=[-50,50],
       zoom_range=[0.75,1.25],
       rotation_range=10)
```

Augmentation in Keras

The tf.keras.preprocessing.image.ImageDataGenerator class can be used to augment image data.

```
# Load data
x, y = ...

# Create a generator allowing translation, zooming and rotation
generator = ImageDataGenerator(height_shift_range=[-50,50],
       width_shift_range=[-50,50],
       zoom_range=[0.75,1.25],
       rotation_range=10)

# Create an iterator, that will generate 64 augmented images at a time
# The iterator can be passed directly to the model
iterator = generator.flow(x, y, batch_size=64)
model.fit(iterator, epochs=100, ...)
```

Dealing with unbalanced datasets

In classification problems a common problem is that the classes are not equally represented.

For example, you might build a CNN to classify different types of cells. However, you might only have a small amount of cells for a specific type.

If you randomly sample the data for your training set you might not have enough information to classify that cell type.

Solutions to this problem are:

- Balance the dataset by sampling the data in a skewed way. Problem: you might not have enough data to train the model.
- Create augmented data only for the rare classes.
- Collect more data!

Data preprocessing

In most cases you will want to scale your data.

Networks using unscaled data can be very slow to train.

The most used transformations used in deep learning are:

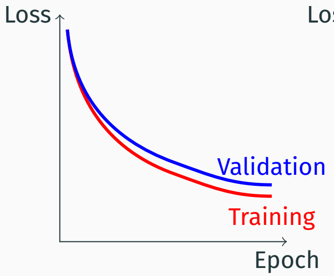
- `min-max normalization` - scale between 0 and 1
- `standardization` - scale to mean 0 and standard deviation 1

Underfitting and overfitting

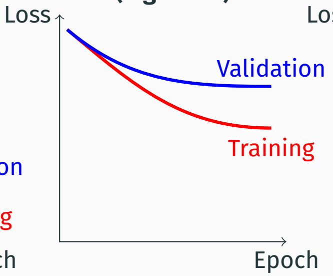
Detecting underfitting/overfitting in a CNN

Plotting the loss during training can help you detect underfitting and overfitting.

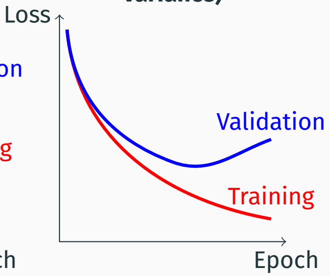
Good fit



Underfit (high bias)

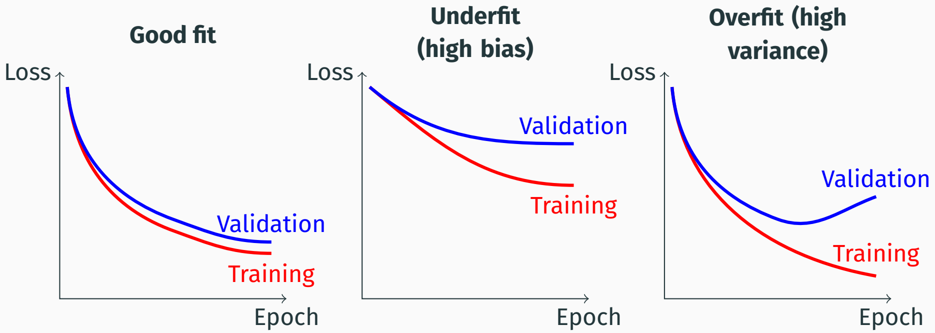


Overfit (high variance)



Detecting underfitting/overfitting in a CNN

Plotting the loss during training can help you detect underfitting and overfitting.



How do we solve these problems?

Underfitting

Underfitting generally derives from the model being too simple and not being able to learn how to solve the problem.

- The model is too shallow.
  Solution: Increase the number of layers.

Underfitting

Underfitting generally derives from the model being too simple and not being able to learn how to solve the problem.

- The model is too shallow.
  Solution: Increase the number of layers.
- Not enough features.
  Solution: Increase the number of filters.

Underfitting

Underfitting generally derives from the model being too simple and not being able to learn how to solve the problem.

- The model is too shallow.
  Solution: Increase the number of layers.
- Not enough features.
  Solution: Increase the number of filters.
- Too short training.
  Solution: train for more epochs.

Underfitting

Underfitting generally derives from the model being too simple and not being able to learn how to solve the problem.

- The model is too shallow.
  Solution: Increase the number of layers.
- Not enough features.
  Solution: Increase the number of filters.
- Too short training.
  Solution: train for more epochs.
- Learning rate too low.
  Solution: Increase the learning rate.

Overfitting

Overfitting is often the result of a model that is too complex, learns from the training data too well, and is not able to generalise.

- Not enough training data.
  Solution: Increase the size of the training set, gather more data, use image augmentation.
- Model is too complex.
  Solution:Use dropout or regularization.
- Model trained for too long.
  Solution: Use early stopping.

Weight regularization

To simplify the model, you can use weight regularization. This is a technique that pushes the weights of the model to be small.

This is done by adding a *cost* to the loss function that penalizes large weights (weighted by a constant λ).

$$J = \text{error}(y, \hat{y}) + \lambda \text{ cost}$$

Two types of regularization are commonly used:

- L1 regularization: the cost added to the loss is proportional to the sum of the absolute values of the weights (i.e. to the *L1 norm* of the weight).
- L2 regularization: the cost added to the loss is proportional to the sum of the squares of the weights (i.e. to the *L2 norm* of the weight).

Weight regularization in Keras

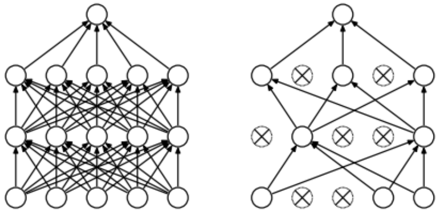
To add L1 or L2 regularization to a layer in Keras, you can set the `kernel_regularizer` parameter when creating the layer.

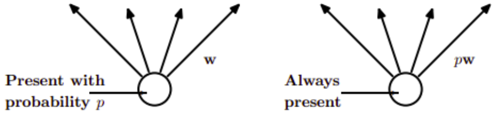
For example

```
layer = keras.layers.Dense(
   units=64,
   activation='relu',
   kernel_regularizer=keras.regularizers.l1(0.01))
```

Dropout

- A type of "regularization" technique, used to prevent overfitting
- A random subset of the weights is set to zero at each training step.
- Originally introduced in "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Srivastava et al. 2014


(a) Standard Neural Net (b) After applying dropout.

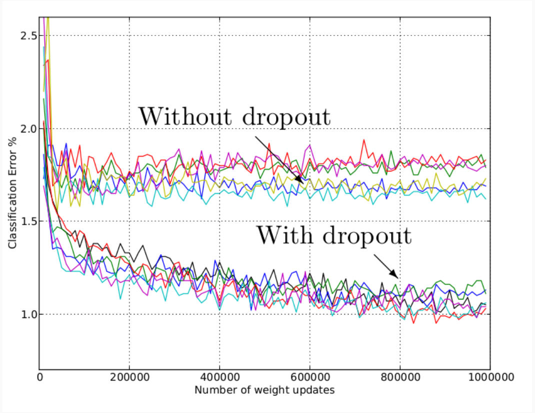


(a) At training time (b) At test time

Srivastava et al. 2014

Dropout

- A type of "regularization" technique, used to prevent overfitting
- A random subset of the weights is set to zero at each training step.
- Originally introduced in "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Srivastava et al. 2014



Srivastava et al. 2014

Early stopping

Early stopping is a technique that stops the training of a model if the validation loss does not improve for a certain number of epochs.

It's easily implemented through the `EarlyStopping` Keras callback.

```
# Create a callback that stops training when there is no improvement in the
validation loss in 3 epochs
early_stop = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
# Train the model
model.fit(x_train, y_train,
   epochs=100,
   validation_data=(x_val, y_val),
   callbacks=[early_stop])
```