Deep Learning Fundamentals

William Darko

Summer 2021

Contents

1 About this course

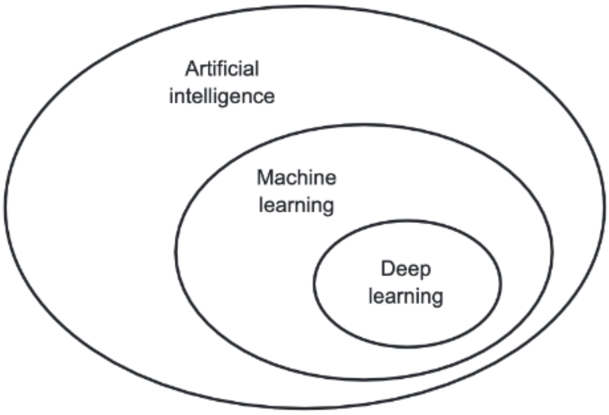
Explore deep learning from scratch or broaden understanding of deep learning, via practical hands-on code examples to solve concrete problems. We'll utilise the Python language, and deep learning framework Keras with Tensor flow as a backend engine

2 Resources

- **Deep Learning with Python** (1st Edition) by François Chollet. Manning publisher

3 What is deep learning?

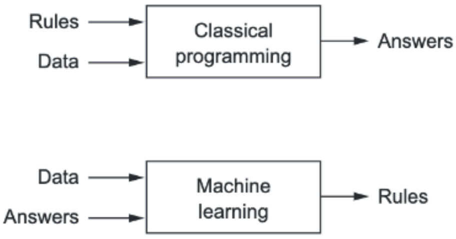
3.1 Artificial Intelligence



the umbrella of Artificial intelligence

We'll define Artificial intelligence to be the *effort to automate intellectual tasks normally performed by humans.* AI is the general field that encompasses machine learning, which deep learning is a subfield of.

- Early AI took the approach of programmers manually implementing a large set of rules for manupulating knowledge; this was known as **symbolic AI**.
- Symbolic AI turned out intractable, when applied to more complex problems like image classification, speech recogition, language translation, etc.
- Machine Learning was the new AI paradigm that rose to replace symbolic AI. It arises from the question: can computers go beyond what we tell it to do? How do computers learn on their own how to perform a specific task.



rules instead of answers in the ML paradigm

- Recent AI trend driven by increase in computing power (faster hardware), and larger datasets.

3.2 Learning representations from data

To do basic machine learning, three things are needed:

1. **Input data** such as sound files, images, text documents, etc.

2. **Examples of expected output** for training purposes

3. **A way to measure the success of the algorithm** to determine the distance between algorithm's current output, and exptected output.

The central problem in machine learning, and deep learning is to meaningfully transform data. Meaning, to learnin useful representations of the input data, which get us closer to the exptected output. The learning aspect involves finding data transformations within an already defined set of operations called the **hypothesis space**.

3.3 The "deep" in deep learning

Deep learning as a subfield of machine learning, is a new take on learning representations from data with emphasis on **learning successive leayers of increasingly meaningful representations**.

1. How many layers contribute to a deep learning model of the data is called the **depth** of the model.

2. Layered representations of data are almost always learned via models called **neural networks**.

3. Think of deep neural networks as a multistage information-distillation operation, where information goes through successive filters (layers) and comes out increasingly purified.

4. deep learning is essentially a multistage way to learn data representations.

3.4 Understanding how deep learning works

1. Deep neural networks conduct a mapping of inputs, to targets via deep sequence of data transformations.

2. The specification of what a layer in a deep neural network does to its input data is stored in the layer's **weights**

3. **Weights** of a layer are in essence, a "bunch of numbers"

4. The transformations implemented by a layer on its input data are **parameterised by its weights**. Thus weights are also called the **parameters of a layer**

5. Initially, the weights of a network are randomised values.

6. In a more detailed explanation, deep learning involves **finding a set of values for the weights of all layers in a network such that the network will correctly map example inputs to their associated targets**

7. To control the output of a neural network, we have to be able to measure how far its output is from the output we desired. This is the job of the **loss/objective function**

8. The **loss function** takes the predictions of a neural network, and the desired prediction, and computes a distance score that captures **how well/accurate the network was on that specific example**

9. The fundamental trick in deep learning is to use this distance score, the **loss score**, to adjust the values of the weights in a direction that will lower the loss score, ultimately improving the accuracy of the network predictions.

10. This adjustment of weights is the job of the **optimiser**, which implements a **Backpropagation algorithm**.

11. A network with a minimal loss score is one which's outputs/predictions are as close as possible to the desired/expected output; **a trained network**

12. The **training loop** of a deep neural network:

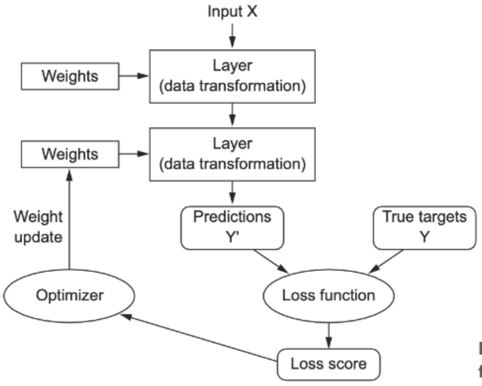


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

Network layers parameterised by weights, receive input data and **perform a sequence of meaningful data transformations picked from the hypothesis space**. The result of these transformations is an output prediction by the final layer, which is then passed to the loss function along with the desired output. The **loss function computes the loss score which is a distance of how far the networks prediction was from the "correct" output**. This loss score is then utilised by the **optimiser which implements Backpropagation to adjust the weights of the layers in a direction that minimises the loss score**.

4 Building blocks of neural networks

4.1 Data representation of neural networks

1. **Scalars (0-Dimensional tensors)**: A tensor that contains one number, a scalar, of some numeric type. The number of dimensions/axes of tensor, also known in maths as the **rank**, can be found using the **ndim** attribtute provided by numpy. So for a scalar x, x.ndim == 0.

2. **Vectors (1-Dimensional tensors)**: A tensor with one dimension, also known as a **vector**, is simply an array of numbers. Keep in mind a vector, as a mathematical structure can be of an arbitrary number of dimensions. For instance, we can have a vector of rank 5; a 5-dimensional vector. Thus, a tensor rank is not the same as the rank of the vector. A vector is a 1-Dimensional tensor, but can be any number of dimensions as a vector.

3. **Matrices (2D tensors)**: A matrix is a 2-dimensional tensor, but can be any arbitrary number of dimensions on row, and column axes of

the matrix. Matrices are often implemented as an array of arrays; a 2-dimensional array, where the number of inner arrays are the number of dimensions on the rows axis, and the uniform length of the inner arrays, are the number of dimensions on the columns axis.

4.1.1 Key atrributes of tensors

1. **Number of axes (rank)**: For instance a matrix is a 2d-tensor, thus a rank of 2. A 3d tensor has a rank of 3. An n-dimensional tensor has a rank of n.

2. **Shape**: Tuple of integers describing how many dimensions the tensor has on each axis. A scalar tensor has a shape of (); an empty shape tuple. A 1-dimensional tensor, a vector, of an arbitrary k dimensions, will have a shape of (k,). A 2-dimensional tensor, a matrix, is of shape: (row, column) where row, and column are the number of dimensions on the row, and columns axes, respectively.

3. **Data type**: The typeof data contained in the tensors.

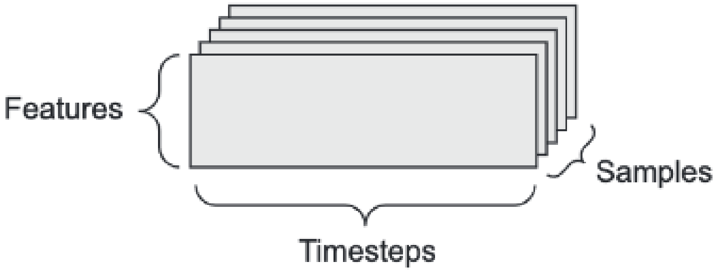
4. **Data batches**: the 0th axis of a tensor, is usually called the **samples axis, or samples dimension**. The number of samples are usually split into n equal parts, called batches. Neural nets usually don't process an entire dataset at once; training of a neural net is done in these data batches; partitions of the total number of samples.

4.1.2 Real-world examples of data tensors

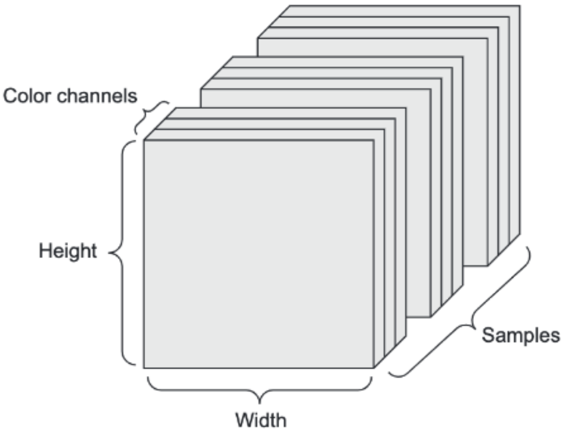
Data in deep learning will almost always fall into one of these categories:

1. **Vector data**: 2d tensors of shape **(samples, features)**

2. **Timeseries data or sequence data**: 3d tensors of shape (samples, timesteps, features)



3. **Images**: 4d tensor of shape **(samples, height, width, channels)** or **(samples, channels, height, width)**. The shape for image data can follow the two most common conventions: **the colour channel first, or colour channel last convention**.



4. **Video**: 5d tensors of shape **(samples, frames, height, width, channels)** or the colour channel first convention **(samples, channels, frames, height, width)**

5 Classification problems example

5.1 Binary classification

1. In binary classification problem (two output classes), network should end with dense layer on one hidden unit, and sigmoid activation

2. Sequences of words can be encoded as binary vectors. Other encoding options can be used to fulfill the bit of preprocessing that needs to be done on the data before feeding it as tensors into the neural network.

3. Stacks of *Dense* layers with with *relu* activations can solve a wide range of sentiment analysis, and binary classification problems.

4. When the output in binary classification is a scalar output like 1 or 0, the loss function to use is *binary_crossentropy*. *rmsprop* is generally a good enough optimiser for any classification problem.

5.2 Single-label multi-class classification

1. When trying to classify data points among N classes, network should end with a *Dense* layerof size N.
2. Network should end with a layer using *softmax* activation so that it outputs a probability distribution across all N output classes.
3. *categorical_crossentropy* is almost always the loss function used in single-label multi-class classification
4. Ways to handle labels: either categorical/one-hot encoding, and using categorical crossentropy, or leave data as integers and use *sparse_categorical_crossentropy*
5. avoid information bottlenecks due to intermediate layers that are too small

5.3 Regression

1. Loss function used in regression are different from that of other classification problems. **MSE (mean square error)** is the most common.
2. Evaluation metrics are also different in regerssoin compared to other classification problem. **MAE (mean absolute error)** is the most common regression metric observed.
3. Some preprocessing must me done on input data with values in varying ranges. Each feature should be scaled independently as preprocessing step.
4. When little data is available, using **K-fold** validation is a great way to reliably evaluate a model. K-fold validation involves dividing the data into K-partitions, and training on K-1 partitions while validation on the remaining batch.
5. Using smaller networks with fewer hidden layers is better when trying to avoid overfitting in the case that little training data is available.

6 Four branches of machine learning

6.1 Supervised learning

1. consists of learning to map input ata to known targets called **annotations**

2. some examples of supervised learning: classification, regression, sequence generation, syntax tree prediction, object detection, image segmentation.

6.2 Unsupervised

1. Consists of finding interesting transformation of input data without help of any targets.

2. often a necessary step in better understanding dataset before attempting to solve a supervised learning problem.

3. **Dimensionality reduction** and **clustering** are well-known categories of unsupervised learning.

6.3 Self-supervised learning

1. Supervised learning without the human annotated labels.

2. Labels are still involved, but generated from input data typically using a heuristic algorithm.

3. **autoencoders** are well-known instance of self-supervised learning where generated targets are the input, but unmodified.

4. Supervision in self-supervised learnign comes from future input data.

5. For instance, predicting the next frame in a video clip giving previous frames, where the next frame in the input data is the target to map to, hence the supervision.

6.4 Reinforcement learning

1. involves an autonomous *agent* receiving information about its environment, and learning to choose actions that maximise or minimise some metric or "reward".

6.5 Classification and regression glossary

- **Sample, or input**: Single data point that goes into your model

- **Prediction or output**: What is outputted by your model

- **Target**: the ground truth. What we desire our model to output

- **Prediction error or loss value**: Measure of the distance between what model predicts, and the actual ground truth (the target)

- **Classes**: Set of possible labels to choose from in a classification problem

- **Label**: a specific instance of class annotation in classification problem. For instance if picture #1056 is annotated as "dog", then "dog" is the label of picture #1056.

- **Ground-truth annotations**: all targets for a dataset, collected, and labeled by humans

- **Binary classification**: Classification task where each input is mapped to one of two exclusive categories

- **Multi-class classification**: Classifiation task where inputs can be mapped to more than two categories. Outputs are usually a probability distribution over all categories.

- **Multi-label classification**: classification sample where each input sample can be mapped to multiple labels.

- **Scalar regression**: Task of mapping inputs to a target that is a continuous scalar value.

- **Vector regression**: task of mapping inputs to targets that are a set of continuous values.

- **Mini-batch or batch**: small set of smaples processed simultaneously by the model

7 Evaluating machine learning models

In machine learning, the central goal is to achieve **generalisation**; a model that performs well on data its never seen before. However, **overfitting**, is the central obstacle to this goal. Overfitting is when the perfomance of a model degrades, and worsens on data its never seen before, compared to their perfomance on training data.

There are several strategies for mitigating overfitting and maximising generalisation. To measure generalisation, we look at ways to evaluate our machine learning models.

- A part of machine learning involves tuning the **hyperparameters** of the model; hyperparameters such as the number of layers, or size of the layers. These hyperparameters are configured according to the networks performance on validation data.
- **Information leaks, and overfitting to validation data**: after doing this step of tuning the network's hyperparameters according to perfomance on validation data, and then evaluating the network with that validation data, what begins to happen is that the **network starts to overfitt** to the validation data, and **some information about the validation data leaks into the model**.
- Since we care about generalisation, and not optimisation on the validation set, we should have a test data set completely separate from the validation set. The model should have no access to this set, not even indirectly.

7.1 Splitting data into training, validation, and test sets

Methods such as ***simple hold-out validation***, ***k-folds validation***, ***iterated k-folds validation with shuffling***.

7.1.1 Simple hold-out validation

1. Set apart fraction of the data as test set, and use the remaining fraction as a training set.
2. Because we want to prevent infromation leaks, we reserve a validation set as well, which we use as the validation set we tune our hyperparameters to.

3. It’s common to train the final model once again on both the training set, and validation set concatenated.
4. It’s also common to shuffle the entire available data before partitioning into training, and validation, and test set

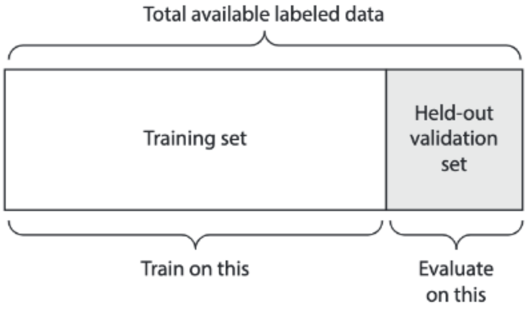
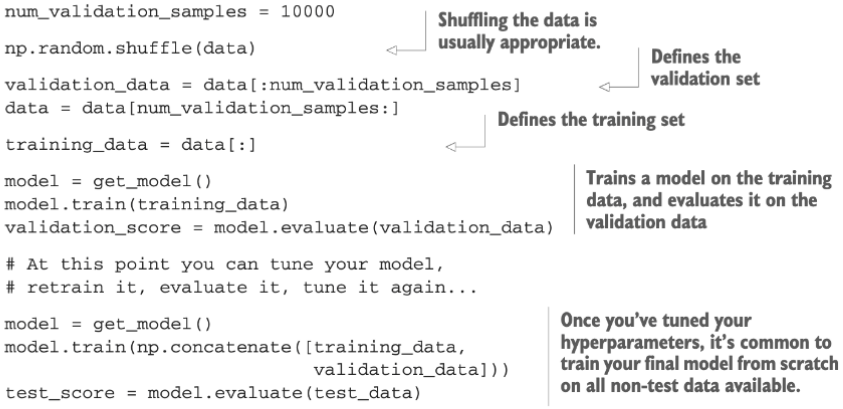


Figure 4.1 Simple hold-out validation split



```
num_validation_samples = 10000

np.random.shuffle(data)

validation_data = data[:num_validation_samples]
data = data[num_validation_samples:]

training_data = data[:]

model = get_model()
model.train(training_data)
validation_score = model.evaluate(validation_data)

# At this point you can tune your model,
# retrain it, evaluate it, tune it again...

model = get_model()
model.train(np.concatenate([training_data,
                            validation_data]))
test_score = model.evaluate(test_data)
```

Shuffling the data is usually appropriate.

Defines the validation set

Defines the training set

Trains a model on the training data, and evaluates it on the validation data

Once you’ve tuned your hyperparameters, it’s common to train your final model from scratch on all non-test data available.

7.1.2 K-fold validation

Simple hold-out validaion is only effective when there's ample amount of data to be partitioned into training, validation, and testing datasets. Thus, when little data is available, K-folds validation is an appropriate approach.

1. Data is split into ***K partitions*** of equal size.
2. For ***each partition i, train model on the remaining K-1 partitions*** and ***evaluate model on partition i***
3. Final validation score is the **averages of the K scores obtained**

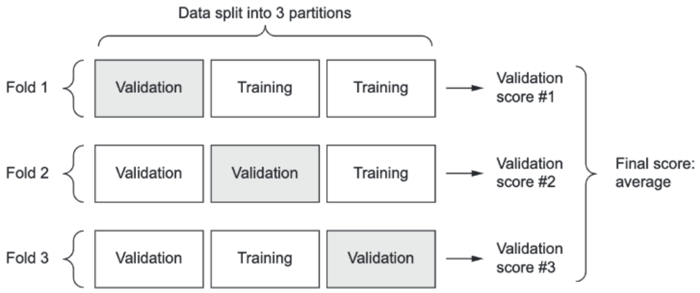
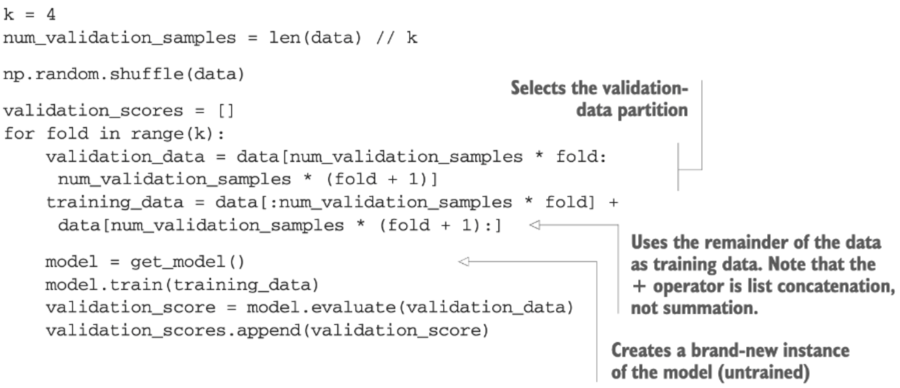


Figure 4.2 Three-fold validation

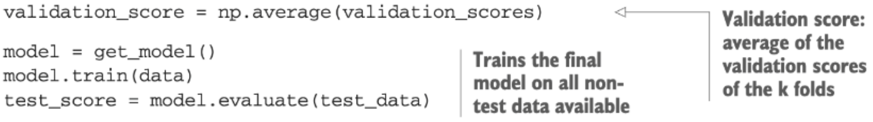


```
k = 4
num_validation_samples = len(data) // k

np.random.shuffle(data)

validation_scores = []
for fold in range(k):
    validation_data = data[num_validation_samples * fold:
     num_validation_samples * (fold + 1)]
    training_data = data[:num_validation_samples * fold] +
     data[num_validation_samples * (fold + 1):]

    model = get_model()
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)
```

Selects the validation-data partition

Uses the remainder of the data as training data. Note that the + operator is list concatenation, not summation.

Creates a brand-new instance of the model (untrained)



```
validation_score = np.average(validation_scores)

model = get_model()
model.train(data)
test_score = model.evaluate(test_data)
```

Validation score: average of the validation scores of the k folds

Trains the final model on all non-test data available

7.1.3 Iterated K-folds validation with shuffling

1. Used when little data available, and need model evaluation as precise as possible

2. Involves applying K-folds validation multiple times (hence iterated k-folds), but shuffling the data each time before partitioning K-ways.

3. Final score, similar to K-folds validation, is the average of scores from each iteration of K-fold validation.

4. Thus the model is trained and evaluated ***P* * *K*** times, where ***P*** is the number of iterations of K-folds validation.

7.2 Things to consider while choosing evaluation method

1. **Data representativeness**: To make sure the training and test sets, are representative of the data at hand, we want to avoid cases where we only train on a subset of the classes of data available. For instance, when trying to classify images of digits, and the dataset is soted by the classes of digits, simply taking a subset of the dataset as it was, would mean only training the model on a set of only small class of numbers, effectively ignoring other digits. Hence, in this case, **randomly shuffling the data before creating any partitions** would allow there to be a diverse training data set.

2. **The arrow of time**: If the task involves predicting the future, given past data such as predicting the next frame of a video, next word in a sentence, tomorrows weather given past weather data, randomly shuffling the data before splitting into training, and testing sets, would create a ***temporal leak***. Meaning, model would effectively be trained on data from the future.

   In such a situation, we should always ensure that test data is ***posterior*** to data in the training set.

3. **Redundancy in data**: If the entire dataset contains duplicate data points, which is quite common in real-world data, then its likely that both training, and test datasets would contain common data points, hence we'll be testing, and evaluating our model on part of our training dataset, which is the worst thing that can happen!

   If the testing set includes training data, the model fails to generalise, and instead, optimises for the training data. Thus, its expedient to ensure that training, and testing data are **disjoint**!

7.3 Preprocessing, feature engineering, and feature learning

The aim with preoprocessing is to make data more suitable for neural networks; whether its via vectorisation, normalisation, or handling missing values, feature extraction, raw data shouldn't be passed directly into neural networks.

7.3.1 Vectorisation

1. All inputs, and targets are made to be tensors of floating points, or in special cases, tensors of integers.
2. the preprocessing step of making all data into tensors is called ***data vectorisation***. One example is using **one-hot-encoding**

7.3.2 Value Normalisation

1. Normaliing features independently so they have standard deviations of 1, and mean of 0
2. Generally considered because its unsafe to feed neural networks data that take relatively large values or heterogeneous data where data points take values in a different ranges.
3. heterogeneous data should be normalised to achieve standard deviation of 1, and mean of 0 before fed into neural networks.
4. normalisation is necessary to prevent large gradient updates that disallow network from converging.
5. Take small values; most values should be in range 0-1
6. Data should be homogeneous; all features should take values in roughly the same range.

7.3.3 Handling missing values

1. When features aren't available for all samples, thus some samples are missing certain featuers, its often **safe to input missing values as 0**, with the condition that 0 isn't a meaningful value in the dataset.
2. Network will learn, and make the connection that **0 stands for missing value**, and would start to ignore it altogether.

3. Note that missing data has to be consistent across both the test dataset, and training dataset. If the network is trained on data with no missing data points, but features are missing in the test dataset, then the validation performace would be poor. One way to mitigate this is to artificially generate training samples with missing features consistent with that of the test samples.

7.3.4 Feature engineering

1. Involves using your knowledge about the tast, the data at hand, and your knowledge of the machine learning algorithm, to **apply hard-coded transformations to the data before it goes into the model**.

2. Data needs to present to the model in a way that makes learning easier.

8 Mitigating overfitting and underfitting

Mitigating overfitting is essential to machine learning. Overfitting happens when our model degrades in performance when exposed to never before seen data. This is because the model started to memorise the training data, and learn irrelevant patterns, instead of trying to generalise.

The fundamental issue in machine learning is the tension between **optimisation** and **generalisation**. ***Optimisation*** refers to tuning our model's hyperparameters and architecture to attain the best perfomance possible. While ***generalisation*** refers to the model performing well on never before seen data. The problem is, this optimisation step, may lead to the data overfitting to our validation set, or even training set after several epochs of training, and tuning.

Our model initially is ***underfit***; meaning there's still progress to be made. The network is yet to learn all the relevant parts of our training data. However as several iterations of training, and optimisation pass, the network stops generalising, validation metrics begin to degrade and ultimately the model starts to overfit. It starts to **learn patterns specific to the training data, but misleading, and irrelevant to never before seend data**.

The best, and simplest way to mitigate overfitting is to simply get more training data to train the network. A network will naturally generalise the more training data its exposed to. However, if more training data isn't an option, the next best thing is ***regularisation***.

Regularisation involves **modulating the quantity of information the model is allowed to store, or constraining what information the**

model should store. The idea is that if the a network can only afford to memorise a small number of patterns, the **optimisation process froces the model to focus on the most prominent, hence, most important data patterns**. This offers a better chance of generalisation.

8.1 Reducing size of the network

1. One of the simplest way to prevent overfitting is to reduce the size of the model; **the number of learnable parameters in the model, which are determined by the number of layers, and the number of units per layer**

2. The number of learnable parameters in a deep learning model, is often referred to as the model's ***capacity***. A model with more parameters has more ***memorisation capacity***

3. A model with limited learning capacity would have to learn compressed learning representations that have predictive power regarding the targets, which is exactly the type of representations we're interested in.

4. A compromise has to be found between ***too much capacity*** and ***too little capacity***

8.2 Adding weight regularisation

1. ***Simple model*** is one where the distribution of parameter values has less entropy; thus one with fewer parameters.

2. Common way to mitigate overfitting is to put contraints on the complexity of the network, by forcing the weights of the work to only take small values, making the weight distribution more ***regular***.

3. This technique of adding constraints to the complexity of the network is called ***weight regularisation***. It involves **adding to the loss function, a cost associated with having large weights**.

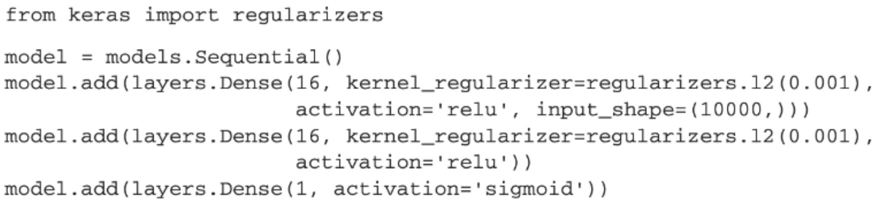
4. weight regularisation *costs* come in two flavours:

   L1 regularisation: The cost added to the loss function is proportional to the ***absolute value of the weight coefficients (L1 norm of the weights)***

   L2 regularisation: The cost added to the loss function is proportional to the ***square of the value of the weight coefficients (L2***

norm of the weights). L2 regularisation is also known as ***weight decay*** in the context of neural networks.

5. In Keras, weight regularisation is added via pssing **weight regularizer instances** to layers as keyword arguments.



```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                       activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                       activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

8.3 Adding dropout

1. Dropout is one of the most effective, most commonly used regularisation techniques in for neural networks.

2. Consists of **randomly *dropping out* (setting to zero)** a number of output features of the layer during training.

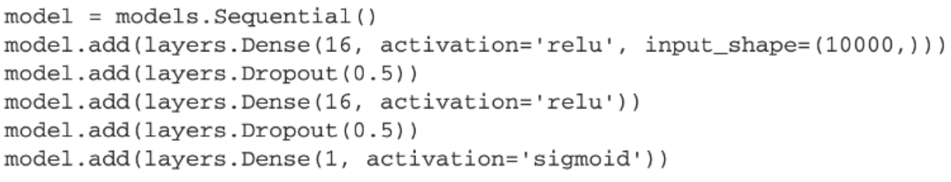
3. ***dropout rate*** is the fraction of features zeroed out; usually between 0.2 and 0.5

4. At test time, no units are droopped out; instead, layer's outputs are scaled down by factor equivalent to dropout rate.

5. Units are scaled down during test time to balance out for the fact that more units are active than at training time.

6. The idea behind the dropout technique is that **introducing noise in the output values of a layer can break up patterns that are insignificant to the network**. This prevents the network from memorising irrelevant patterns.

7. In Keras, dropout is implemented via the Dropout layer, which is applied to the output of the layer immediately preceeding it.



```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

8.4 Recap

Generally, the most common ways to prevent overfitting in neural networks:

- Getting **more training data**. Network would naturally generalise this way.
- **Reducing the capacity of the network**.
- Adding **weight regularisation**.
- Adding **dropout**

9 Universal machine learning workflow

Defining the problem, evaluating the model, feature engineering, mitigating overfitting.

9.1 Problem Definition

- We must first define the task/problem at hand
- What will our inputs be?
- What are we trying to predict?
- What type of problem are we facing (regression, binary classification, multiclass classification etc).

To move on to the next step, it's crucial to know what our inputs, and outputs are, and what data we're going to be using. We make certain hypotheses at this stage:

- Hypothesise that ou puts can be predicted given inputs
- Hypothesise that available data is sufficiently informative to learn relationships between inputs and outputs

9.2 Choosing a measure of success

To control something, we must be able to observe it. We must define what we mean by success. Is it accuracy of the model's predictions, precision and recall, customer retention, etc. Our metric of success will **guide what we choose for our loss function; what our model will optimise**. Metric of success **should align with higher level goals**, such as success of a business.

- **For balanced-classification problems**, where all classes are equally likely to be mapped to an input, a metric like **accuracy** or ***area under the receiver operating characteristic curve (ROC AUC)*** are common metrics to use.
- **For class-imbalanced problems**, its common to use **precision and recall** as metrics of success.
- **For ranking problems, or multi-label classification**, we can use **mean average precision**.
- It's also quite common to define custom metrics of success.

9.3 Choosing an evaluation protocol

Once we know what we're trying to measure, and what our data is, we must pick a way to measure our current progress; the evaluation protocol.

- ***Hold-out validation set***: the way to go when there's plenty of data
- ***K-fold cross validation***: When not enough data is present for hold-out technique
- ***Iterated K-fold validation***: When you want highly accurate model evaluation, but little data is present.

9.4 Preparing the data

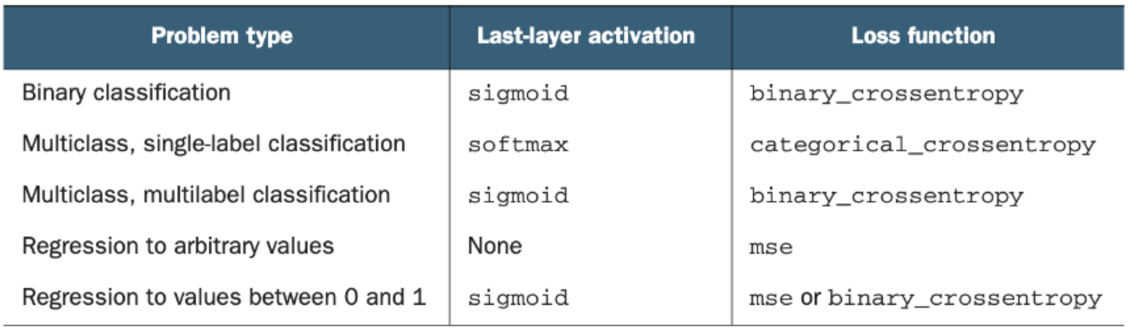
- Data should be formatted as tensors
- Values taken by tensors should be scaled to take small values within uniform ranges
- heterogeneous data should be normalised
- Do some feature engineering where possible

9.5 Developing a model

Goal here is to develop a model that achieves ***statistical power***. This means to develop a model that beats the baseline model in terms of performance. Three key choices for making first working model:

1. ***Last-layer activation***: establishes useful constraints on layer's output
2. ***Loss function***: should match the type of problem at hand.
3. ***Optimisation configuration***: What optimiser will you use, what learning rate will you choose, etc.

Its not always straightforward to optimise for some metrics of success, or turning some metric into a loss function. Loss functions mut be **differentiable** (otherwise Backpropagation can't be used to train network). Loss functions have to computable with as little as one data point. However, for the most common types of problems, the following are sufficient:



Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

9.6 Developing a model that overfits

To develop the best model for our problem, we must find that sweet spot between an underfit model, and one that overfits. Before we get to this best case model, we must first find the border where overfitting begins. We can achieve this fairly easily.

1. Add layers
2. Make layers bigger
3. Train for more epochs

Once we observe that the model's performance on validation data begins to degrade, we've achieved overfitting.

9.7 Regularising model, tuning hyperparameters

This step will take the most time; repeatedly training the model, evaluating it, tuning hyperparameters and the network architecture, over and over again until the best model is achieved.

10 Part 2: convnets

Convolutional Neural Networks, also known as ***convnets*** are a type of deep learning model, well known in the field of computer vision.

1. Convnets take input tensors of shape (image_height, image_width, image_channels)

2. In Keras, convnets can be implemented using the **Conv2D** and **MaxPooling2D** layers.

3. Every output of the Conv2D, and MaxPooling2D layers are 3d tensors of shape (height, width, channels)

10.1 The convolution operation

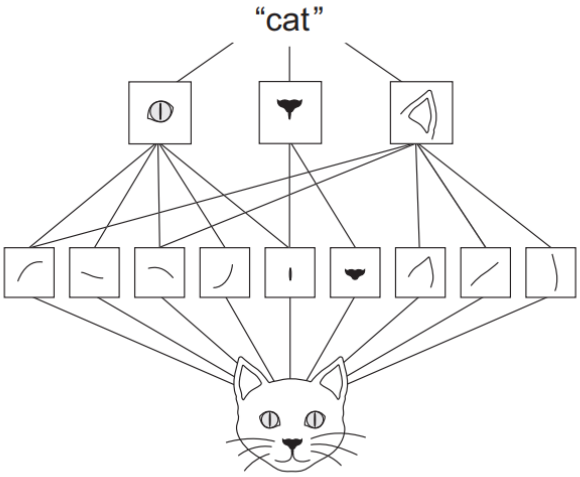
While Dense layers can only learn global patterns in the input feature space (for example patterns involving all pixels), **convolution layers are able to learn local patterns**, that found in **small windows of the input**. These windows are typically, **2x2 or 3x3** in size.

This key characteristic of learning local patterns in small windows of inputs gives convnets **two key characteristics**:

1. ***The patterns learned are invariant to translation.*** This prevents the inefficiency of re-learning patterns already learnt. For example, if a pattern is learned in the lower left corner of an image, a convnet can recognise that same pattern anywere else in the image. **This makes convnets data-efficient**, hence why they need fewer training smaples to learn representations.

2. ***Ability to learn spatial hierarchies of patterns.*** This means that successive convolution layers learn larger patterns composed of features learned in the previous layer. For example, the first convolution layer learns curves, lines, and edges, and the second layer learns features composed of those lines, curves, etc, learned in the first layer. **This allows convnets to learn increasingly complex and abstract visual concepts**

The visual world is fundamentally translation invariant, and spatially hierarchial.



1. Convolutions operate over **3D tensors** called ***feature maps***.
2. **Feature Maps** have **2 spatial axes**, and a **depth** axis.
3. The depth axis is sometimes called the **channels axis** like in the case of RGB images.
4. **spatial axis**: ***height, width***
5. The convolution operation **extracts patches** from the input feature map, and **applies the same transformation on all of them**, producing an ***output feature map***
6. Output feature map is also a 3D tensor, with height, and width spatial axis, as well as a depth axis. However the depth axis may vary, from layer to layer.
7. Depth axis can be arbitrary because the depth is a **parameter of the layer**, thus different channels in the depth axis no longer stand for RGB, but instead, they stand for ***filters***
8. **Filters** encode specific aspects of the input data. For instance, a single filter can encode the concept of **"presence of a face"** in the input.

To look at filters more closely, say a convolution layer takes a feature map of shape **(28, 28, 1)**; thus inputs of size 28 by 28, with 1 colour channel. Lets say the convolution outputs a an output feature map of size **(26, 26, 32)**. This in essense means an output feature map of **32 ouput channels**, but these 32 channles on the depth axis don't account for colour channels, but instead, the **32 filters computed over the input**. Hence, each of these 32 output channels contains a 26 x 26 grid of values, indicating the response of the filter pattern at different locations of the input. This 26 x 26 grid of values is called the ***response map*** of the filter over the input. The 2D tensor: output [:, :, n] is the spatial map of the response.

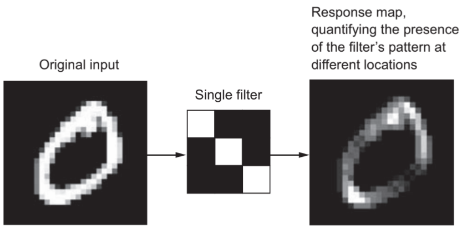

Original input
Single filter
Response map, quantifying the presence of the filter's pattern at different locations

Figure 5.3 The concept of a *response map*: a 2D map of the presence of a pattern at different locations in an input

Convolutions are defined by two key parameters:

1. ***Size of the patches extracted from the inputs***. In other words, the window sizes; these are typically, **3 x 3 or 5 x 5**

2. ***Depth of output feature map***. In other words, the number of filters computed by the convolution over the input.

In Keras, window/patch size, and output depth are passed as:

Conv2D(output_depth, (window_height, window_width))

10.1.1 How convolution works (at a high level)

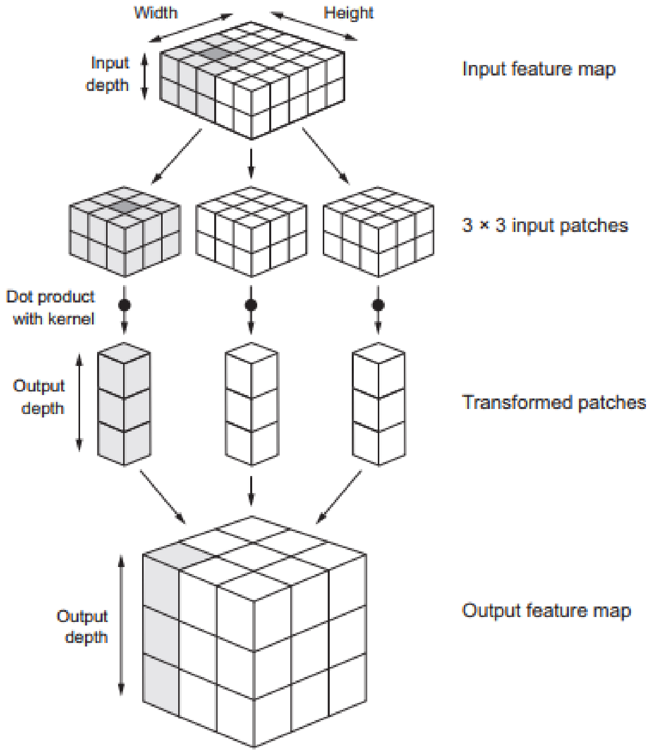
A convolution works in a certain number of steps:

1. **sliding down patch windows of 3x3 or 5x5** over 3D input feature map and **stoping at every possible location, extracting the 3D patch of surrounding features** of shape **(window_height, window_width, input_depth)**

2. Each extracted 3D patch is then **transformed into a 1D vector of shape (output_depth), via a tensor product with the same learned weight matrix called the *convolution kernel***

3. All of the 1D vectors resulting from the tensor product with the kernel, are then **spatially re-assembled** into a 3D output feature map of shape (height, width, output_depth)

4. Every spatial location in the output feature map, corresponds to the same location in the input feature map. For instance, the bottom right corner of the output, contains information about the bottom right corner of the input.



Output width, and height, may differ from the input width and height for two reasons:

1. **Border effects**

2. Use of **strides**

10.1.2 Border effects and padding

Suppose we have a 5 x 5 input featue map, thus 25 tiles in total. With a window/patch size of 3 x 3, there's onlu 9 possible tiles for which we can fit/centre our 3 x 3 window. The output feature map, is then of dimension 3 x 3. Thus our map shrinks a little, exactly by two tiles along each dimension. We can see this **border effect** in play, spatial dimensions shrink after each convolution layer.

To mitigate this border effect, thus to get **output feature maps with the same spatial dimension** as the input, we introduce a technique called ***padding***.

Padding involves adding the appropriate number of rows, and, or, columns on each side of the input feature map to allow fitting center convolution windows around every input tile. Padding will result in more patches being extracted, thus more 1D tensors after the tensor prodcut to spatially reassemble.

In Keras, Conv2D layers inplement padding via the padding argument. Padding argument takes two possible values: **" valid " which means only valid window locations will be used, thus no padding**, and **"same" which indicates to add padding, to achieve an output feature map with the same spatial dimension as the input**.

10.1.3 Convolution Strides

Strides are the other factor that may affect the spatial dimensions of the output feature map. So far, we've observed convolution windows where centre tiles are contiguous, thus the windows slide only one unit horizontally, or vertically. However, the distance between two successive windows is a **parameter of the convolution known as its *stride***. Stride **defaults to 1** in Keras, which is why our patch windows initially only shift one tile horizontally or vertically during convolution.

Its possible to have ***strided convolutions***, so in other words, strides **greater than 1**. To use a stride of k greater than 1, means to **downsample the width and height of the feature map by a factor** of k.

In practise, strides are rarely used, however its good to be familiar with the concept. To downsample feature maps, instead of strides, whats usually used is the ***max-pooling*** operation.

10.1.4 Max-pooling operation

The purpose of the *max-pooling* operation is to **aggressively downsample feature maps** much like what we achieve using strides.

Conceptually similar to convolution, max-pooling uses **2x2 windows to extract patches from the input feature map**. However, unlike convolution, max-pooling operation doesn't perfrom a kernel dot transformation, but instead **outputs the max vlaue of each channel via a hardcoded *max* tensor operation**.

Max-pooling is usually performed with 2x2 windows and a stride of 2, in order to downsample the feature maps by a factor of 2.

10.1.5 Why downsample feature maps?

Without downsampling, our model **will not be conducive to learning spatial hierarchies of features.** In other words, the high level patterns learned by convnets will be very small, and insufficient to classify anything. We need features from the last convolution layer to include information about the totality of the input.

Another reason to downsample **reduce the number of coefficients being after the last convolution layer**. Suppose we begin with an input of shape $(28, 28, 1)$, and the convolution process reduces the spatial axes by a factor of 2 with each layer. Without a downsampling operation like max-pooling, after 3 convolution layers, our final output is of shape $(22, 22, 64)$; this final feature map has $22 * 22 * 64 = 30,976$ coefficients **per sample!** Now suppose we flatten it to apply a Dense layer of size 512; this layer would have $30,976 * (512 + 1) \approx 15.8$ **million** parameters, which is **huuuuuugeeee!!!!** Far too large for such a small, trivial model.

Ultimately, the reason to downsample is to:

1. **induce learning spatial hierarchies of features, by making successive convolution layers look at increasingly large windows (in terms of fractions of the initial input)**

2. **reduce the number of feature map coefficients**

Although max-pooling isn't the only way to acheive downsampling, its often the best solution as opposed to alternatives like strides, or something like *average-pooling.*

Because features tend to encode for some high-level presence of some pattern or concept, across different windows, of the feature map, it is definitely **more informative**, in the context of deep learning, to look at the ***maximal presence*** **(max-pooling)** of those features, as opposed to the *average presence* (average-pooling).

11 Training convnet from scratch