INTRODUCTION TO DEEP COMPUTER VISION

John Olafenwa & Moses Olafenwa

INTRODUCTION TO DEEP COMPUTER VISION

1st Edition (BETA DRAFT)

John Olafenwa and Moses Olafenwa

PREFACE

The field of artificial intelligence has been evolving at an unprecedented rate in the past few years. So many important contributions have been due to the incredible work of a number of post-graduate students from the University of Montreal, University of Toronto, Stanford University, UC Berkeley, MIT and a number of research focused institutions. However, in recent years, a lot of undergraduate students and professionals from other fields of engineering and computer science have been contributing incredibly to the advancement of artificial intelligence. We believe, AI will advance more rapidly if more people are able to quickly get up to speed with recent advances. This can only happen when there are sufficient entry level books designed to address the needs of aspiring AI scientists who may not have a very solid background in the fields of applied mathematics and statistics. The motivation for this treatise is our own experience while transitioning from Application developers to deep learning professionals. We both were very excited to get into the field of Artificial Intelligence but we faced a number of problems, first, many of the books available treated only old-time machine learning algorithms, which were not applicable to the modern tasks facing machine learning practitioners. The other books that addressed modern deep learning approaches, included practical that used low-level deep learning libraries. In this book, we present modern techniques in computer vision and we used a very high level python library for all practical. We hope the reader will find this an easier path towards becoming AI researchers and engineers.

Notes About the Beta Draft

Based on feedback from readers of the first draft, we have corrected a number of errors and restructured some aspects of this book.

This draft includes additional topics including Convolution Arithmetic, Rectifiers and Efficiency Techniques. We have also expanded analysis on Residual Networks.

Appreciation

We sincerely thank all the readers of the first draft of this book for their invaluable comments and feedback that has helped us to produce the beta edition of this book. We specially thank the BlackInAI community for their great support. Special appreciation to Rediet Abebe, the co-founder of “Mechanism of Design for Social Good”, for helping

with the review of the first draft. Without all the love and support you have all shown, we would not be in full energy to revise the first draft.

AI Commons

This book is an effort by AI Commons to advance and make Artificial Intelligence accessible to every individual and corporate entity on the planet.

Visit https://commons.specpal.science to learn more about our projects.

CODES IN THIS BOOK CAN BE OBTAINED FROM https://github.com/johnolafenwa/DeepVision

ERRATA

This is an early access version of this book, we encourage you to report any errors to us via email to johnolafenwa@gmail.com or guymodscientist@gmail.com, in doing so, please include the specific page where the error occurred. General comments and suggestions for the further development of this book are welcome. We look forward to and appreciate your feedback.

For the latest releases of this Book, visit john.specpal.science/deepvision

CONTENTS

INTRODUCTION TO COMPUTER VISION AND MACHINE LEARNING

The eye is perhaps the most important sensory organ in the human body, not just because it can capture the view of our environment but because of the visual cortex which interpret what the eyes captures. The visual cortex is made up of four units, v1, v2, v3 and v4. They collectively enable pattern recognition with stunning accuracy and unprecedented efficiency. The eyes act as a camera and the visual cortex acts as the intelligence unit.

Computer vision is a field of study devoted to teaching computers how to recognize patterns and understand what these patterns represent. Capturing images or videos with cameras is not what we deal with here, computer vision and pattern recognition is not about the camera (the eye), it is about the intelligence layer that interpret what the camera captures. Hence, the focus of this book is not to teach the reader to create better cameras but to improve the intelligence layer.

Computer vision has a lot of practical applications and is a central component of Artificial General Intelligence. Autonomous cars, called driverless cars are powered by state-of-the-art vision systems that allows them to interpret the current state of the road, this is combined with data from radar and lidar systems to enable cars to drive themselves. Robots can also navigate better due to advances in deep computer vision. Skin Cancer detection is now possible using advanced mobile applications thanks to computer vision. The current applications are endless and the potentials are just being explored, we have only scratched the surface and one day, you can help take the field to the next level. Computers have existed for so long, but only recently did autonomous cars become possible, the reason is that, at inception, computers operated with absolute certainty, they were very good at tasks in which the conditions and outputs are certain. Complex mathematical functions could be accurately expressed using algorithms written by human programmers. This evolved to enable more abstract functionalities such as emails, the internet, office management tools etc. However, the task of interpreting ambiguous data such as pictures, speech, conversations and navigation in very dynamic environments proved to be an impossible task for computers, this is because, it is impossible to write a general algorithm that would express the solution to these problems.

If we want to build a system to recognize pictures of cars separate from lorries, then you have to understand the key features unique to them and write a long list of *if else* statements to classify new pictures, the problem is that all cars are not the same, likewise lorries differ, some features present in almost all cars maybe absent from others, their shapes are also inconsistent. Any attempt to write an algorithm that would account for all of these differences would fail miserably. One major reason for this, is variation in the presentation of the image, the same car can be presented from many perspectives, sometimes, only a part of the car would be shown.

Consider a cat and dog classifier, cats and dogs have different breeds and can appear in different poses and postures. They also share similar features.

The ambiguities of patterns extend to speech recognition and natural language processing (NLP), the focus of this book is on pattern recognition but the same approach used to solve pattern recognition extends to the other fields.

The solution to interpreting ambiguous patterns is rooted in probability theory. Probability theory presents an excellent framework for working in conditions where inputs and outputs are uncertain, conditions which are not defined by one to one function mappings.

These relaxes the need for highly explicit algorithms and allows us to make decisions even in the absence of certain features.

Based on probability theory, computer scientists developed machine learning. Various machine learning algorithms developed in the last decades of the past millennium made computer vision possible.

In classical computer programming, we program computers with instructions on how to solve a problem, but in machine learning we program computers with instructions on how to learn to solve a problem.

This is analogous to how the human brain works, we were never created with the ability to solve all tasks that we might ever encounter, rather, God created us with the ability to learn how to solve a problem based on experiences we have had.

In the context of machine learning, instead of attempting to write algorithms that would tell a car apart from a lorry, we would simply write a few instructions

telling the computer how to learn from a well labelled dataset of cars and lorries, the system would infer all by itself, the features that distinguish the two classes of vehicles, such learned features would span a wide range of car and lorry types. The learned features would be stored as a model and can then be used to predict the class of new images.

Machine Learning involves two main components; The Learning Algorithm and The Data.

The Learning Algorithm

This is a generic algorithm that states how a machine should learn from data.

A classic example of a learning algorithm is Linear Regression. This is used to predict the value of a function based on a number of input variables.

Linear Regression takes the form:

$$y = \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3 + \theta_4 X_4 + \cdots + \theta_n X_n + \beta$$

Here, the set of parameters θ_1 to θ_n and the bias β, represents the relationship between variable y and the input variables X_1 to X_n. Rather than attempting to manually find what the values of these parameters should be as done in classical programming, we simply feed in a dataset comprising of values of y and its equivalent inputs X_1 to X_n, the learning algorithm then figures out the right values of θ_1 to θ_n using an optimization technique known as Gradient Descent (which we shall explain later).

Once these values are obtained, they can then be used to accurately predict new values of y based on input values X_1 to X_n.

Machine learning algorithms involve inferring from training data, a set of parameters θ that models the relationship between a target variable and a set of input variables.

This is represented mathematically as

$$\boldsymbol{Pr}(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$$

This function entails the probability of an output y given a vector of variables x , parameterized by θ.

Using this approach, we do not have to write conditional statements like *if else*, all we need to do is feed the training data to a learning algorithm which would then infer the correct parameters.

This very simple framework has evolved over decades of vigorous research by AI scientists, notably pioneers like Rosenblatt, Geoffrey Hinton, Yoshua Bengio, Yann LeCun and many more greats.

Many powerful machine learning algorithms have been developed to solve tasks with better accuracy and efficiency.

These algorithms can be broadly classified into three distinct categories.

Supervised Learning

This class of machine learning algorithms are used to model relationships between targets and their attributes. They are "Supervised" in the sense that all data we feed in has to be fully labelled.

For instance, when building an image recognition system, we have to feed in thousands or millions of images and every single image has to be labelled according to the class they belong to. The ImageNet Large Scale Visual Recognition Challenge (ILVRC) is more like an annual Olympic of computer vision, where the best scientists in the computer vision community compete to build the best image recognition system based on a dataset comprising of 1.2 million images divided into 1000 categories.

Another example includes the Boston House Pricing dataset comprising of house attributes and their price.

So many such datasets exist and with some dedication, you can compile your own. Kaggle is the home for Data scientists where you would find so many datasets useful for a wide variety of tasks.

Supervised Learning algorithms are by far the most successful and widely used machine learning algorithms till date. Most of computer vision falls into this category. Data used in supervised learning is often structured. Often as tables where features are represented as columns and items are represented as rows.

The major downside of Supervised Learning algorithms is that they require so much data and labelling large amounts of data is a very tedious task that is usually done via crowd sourcing.

Supervised learning is further divided into two main subcategories.

Regression

Regression algorithms are used to predict real valued numbers where the output can be any of the set of real numbers. In simple terms, the set of outputs is an infinite set. Examples include predicting house prices based on the attributes of the house. The price could be any number ranging from thousands to millions, while practical limits exist, there is no theoretical limit to how large the prices can be. In theory, they can be trillions or more.

Even such practical limits depend on the unit of measurement. However, the limits are more defined in some cases such as predicting the age of a person.

Examples of such algorithms include Linear Regression, Random Forest Regression, Gradient Boosting Regression and many more.

Classification

This is also called Logistic Regression; it is primarily concerned with predicting the class of a target based on its attributes. Examples include classifying an image based on the pixels. The output of the algorithm can only be one of N classes. Hence, the output set is finite and very definite.

So many machine learning tasks falls into this category. Predicting whether a patient has cancer or not is a classification task, the output can only be one of two classes "Cancer" and "Not Cancer"

When there are only two possible outputs, the task is called "Binary Classification" or "Binary Logistic Regression." When the classes are more than two, it is called "Multinomial Logistic Regression"

Computer vision falls under this subcategory.

Unsupervised Learning

This class of algorithms are used for analyzing unlabeled and often unstructured data. They are useful for finding patterns, relationships in data and estimating the

probability density of a distribution. Their uses include segmentation and clustering.

This is very useful in data analysis. For example, market segmentation enabled by unsupervised learning can help businesses to identify groups to which their customers belong, this can help to make better business decisions and marketing strategies. The literature is less developed than supervised learning. However, the great availability of unstructured data on the world wide web makes unsupervised learning, an essential tool.

Reinforcement Learning

This class of algorithms are primarily used in the field of robotics. They define the behaviour of an agent in an environment, where the agent has full or only partial information about its environment. The agents can take a number of actions A in an environment made up of different states S, the agent learns to take the right actions in different states based on trial and error. The agent is guided towards its goals by rewards assigned for making decisions. The reinforcement learning algorithms define how the agent learns to choose the right actions.

Different algorithms exist, however they are largely based on the Framework of Markov Decision Processes.

Markov Decision Processes (MDPs) are guided by the concept of maximizing rewards and the Markov property which states that "*Given the present, the future is independent of the past*" Some popular Reinforcement learning algorithms include Dynamic Programming, Temporal Difference Learning, Monte Carlo Methods, Advantage Asynchronous Actor Critic, Double Q-Learning, Trust Region Policy Optimization, Proximal Policy Optimization Algorithms, Meta Learning Shared Hierarchies etc. The details of these algorithms are beyond the scope of this book, for a full introduction to reinforcement learning, read Sutton and Barto, 2018.

Sometimes these classes of machine learning algorithms overlap. Semi-supervised learning and Deep Reinforcement learning are classic examples of such overlap.

Components of Machine Learning

Machine Learning is composed primarily of the learning algorithm as explained above, the data, a loss function that measures our performance and an optimization algorithm. The process of machine learning goes thus. The learning algorithm is initialized with a set of default parameters θ_1 to θ_n. We then iterate over the dataset and at each row, we feed in the attributes X_1 to X_n into the learning algorithm, these outputs a prediction of the target variable based on our current set of parameters, our loss function is used to compute how close our prediction is to the actual value of the target as contained in our dataset. The loss is usually aggregated across all examples.

The optimization algorithm called gradient descent, would then update the parameters of the learning algorithm in a direction that would reduce the aggregated loss. These process is repeated until the loss can no longer be reduced, ideally we want the loss to become zero but this is not often possible in practice. More details on how this is done would be expounded under the headings Structure of Data, Loss Functions and Optimization.

Structure of Data

Data can be structured in many ways, however, most machine learning datasets used in supervised learning follows the table structure. We shall emphasis on this, but there are a couple of structures shared by most data structures.

Each individual piece of data such as information about a single house is made up of variable-value pairs. The variables refer to the features. Variables of housing information include the land area, color, height, location, owner etc. The values are the information stored by these variables.

For example, "Land Area" is the variable while $200m^2$ is the value.

Variables can be continuous or categorical. Continuous variables are real valued numbers, they include variables like price, age, length, area, temperature etc.

Categorical variables are discrete variables, they are grouped into categories and cannot be expressed as real valued numbers. Examples include gender, race, color, state, profession etc.

In the tabular structure. Every feature is represented as a column, each column is named according to the feature it represents, the order of arrangement of the columns does not make a difference.

Each item containing the values of these variables are represented as rows. Hence, if we have 20 features in our dataset, there would be 20 columns in the table, if we have 1000 items in our dataset, there would be 1000 rows in the table.

See the table below for an example

Gender	Age	Annual Salary	No of Children	Profession	Race
Male	32	200 000	3	Software Engineer	White
Female	25	80 000	1	Writer	Black
Female	30	80 000	2	Swimmer	White
Male	20	50 000	0	Fireman	White
Female	40	150 00	4	Pilot	Black

In machine learning, datasets are often divided into two types: Training Data and Test Data, sometimes a third set called the Validation data is used.

We train on the training data and the loss on it guides our optimizer, however, it happens often that a model that works well on the training data may not generalize properly to new data. Hence, we often judge the effectiveness of our model based on its performance on the test data. Note that the test data is never used to train the model, hence, if the performance on the test data is high, our model should be able to work fairly well in the real world. The validation data is useful for cross-validation, that is, for evaluating the performance of our model on previously unseen data during the training phase, this is useful for properly setting a number of hyper parameters which you would become familiar with in the next chapters.

Loss Functions

Given a set of parameters, a loss function helps us to evaluate how well our learning algorithm is performing on the training data using our current parameters.

Using *y* to denote the prediction of our algorithm and given parameters θ_1, θ_2 and θ_3

$$y = \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3 + \beta$$

If y is the actual value provided in the training data, a simple loss function is

$$L = f(x) - y$$

This gives us the difference between the prediction and the actual value.

Aggregating these losses over N items in our dataset, the loss would be the average of all loses

$$\frac{1}{n}\sum_{i=1}^{n} f(x^i) - y^i$$

These would give us the average loss over all examples. The lower the loss, the better our performance. In practice, the Mean Squared Error (MSE) is used.

$$\frac{1}{n}\sum_{i=1}^{n} (f(x^i) - y^i)^2$$

This is the average of the squared error over each example. For the purpose of computing gradients for optimization, the MSE is represented as

$$\frac{1}{2n}\sum_{i=1}^{n}(f(x^i) - y^i)^2$$

As you might have noticed, MSE is used for Regression where we are predicting real valued numbers.

For classification, we use a loss function called *Softmax Cross Entropy Loss*

Before explaining the exact form of the loss function, you should understand what softmax is.

Classification always returns scores for all the classes available in our dataset. For example, when predicting the gender of a person given variables such as profession, annual salary, favourite food and favourite sport. Based on these inputs, our classifier would output scores for male and female.

An example output is **[9,6]** representing **9** for male and **6** for female. Obviously, this prediction tells us that the subject is male, however, it is unclear how to interpret the confidence level of this prediction, especially when the classes are many.

Softmax would take in these scores and return probabilities between 0 and 1. This gives a clear picture of our confidence level.

Given a set of Scores (S)

$$P = \frac{e^{s}}{\Sigma\, e^{s}}$$

P = Probability Vector

e = Base to the Natural Logarithm = 2.71828

s = S_i = Score of each class

In plain English, softmax does the following in order.

1. Take the exponent of each score

2. Sum the exponents

3. Divide each exponent by the sum

This would give us the probability vector for any set of scores. It is an affine vector to vector transformation that preserves the dimension of the input vector. This entails that the number of elements in the set of scores is the same as in the set of probabilities.

Consider our gender example

S = [9,6]

Step1: Find the exponents

E = [e^9, e^6]

E = [8103.084, 403.429]

Step2. Sum the exponents

Σe^s = 8103.084 + 403.429 = 8506.513

3. Divide each exponent by the sum

P = [8103.084/8506.513, 403.429/8506.513]

P = [0.953, 0.047]

This gives us 95% probability for male and 0.04% for female, this is a very high confidence score and it makes it easier to interpret the results of the classifier.

In accordance with the rules of probabilities, the sum of the elements of the output of softmax is 1.

0.953 + 0.047 = 1.

Due to approximation errors, the sum may not be exactly one, but it would always be very close to 1.

The softmax cross entropy loss is simply the sum of the negative of the log of the softmax score of the correct class. This may sound confusing, but we shall make it clear.

It takes the form

$$L = \frac{1}{n}\sum -\log(S_j)$$

Where j is the index of the correct class. Hence S_j is the score of the correct class.

In the above the softmax scores are 0.953 and 0.047, assuming the subject is actually female, which means our classifier is doing a really bad job by giving a very low probability to our correct class.

The softmax cross entropy loss would be

$$L = -\log(0.047) = 1.328$$

On the other hand, if our subject is male, which entails that our correct class score is 0.953, our loss would be

$$L = -\log(0.953) = 0.020$$

As you can clearly see, the loss is very low when we are making the right prediction and very high when we are making the wrong prediction. Notice that, though we choose the correct class, there is still some small loss, this is because the wrong class is assigned some probability. If the probability of our correct class is exactly 1, then our loss would be exactly zero.

$$L = -\log(1) = 0$$

As $s_j \rightarrow 1$, $L \rightarrow 0$

When the number of subjects is more than 1, we have to sum the negative of the log of the scores for the correct class predictions and find the average of the sum.

Softmax cross entropy loss is also called negative log likelihood loss.

This loss function is the most preferred choice in classification.

Regularization

A final aspect of the loss function is *regularization*; this technique is used to prevent a weakness known as overfitting.

Overfitting refers to a situation in which our learning algorithm accurately models our training data but fails to perform well on the test data. This entails that our model does not generalize properly to new examples. The training loss in such is very low but the test loss is very high. There are various techniques for preventing overfitting, Regularization is the most common technique. This technique is based on the fact that models usually overfit when the values of the parameters is too large. Different parameter sets can yield low loss, however, parameter sets that have large values tend to result in low loss on the training set but fails to yield

correspondingly high score on the test set. Lower parameter values on the other hand, tend to yield better accuracy on the test set. To prevent overfitting due to large parameter values, regularization is added to the losses, penalizing large weights.

It is represented as:

$$\lambda \sum_{i=1}^{k} \theta^2$$

The above is the L2 regularizer

λ is the weight decay, it controls the strength of the regularizer. It is a hyper parameter, whose true value we usually determine during cross validation.

The above can be broken down into the following steps

1. Square each parameter value
2. Sum up the squared parameters
3. Multiply the sum by the weight decay λ

Adding the regularizer to the loss function, the loss function looks as below

$$L = \frac{1}{n}\sum -\log(s_j) + \lambda \sum_{i=0}^{k} \theta^2$$

The ultimate effect of this new equation is that, due to the L2 regularizer, higher parameter values would incur higher errors, hence, are less likely to be selected as the final parameter set.

The L1 Regularizer is seldom used, its form is

$$\lambda \sum_{i=0}^{k} |\theta|$$

This method has proven to be very effective in practice, however, complementary methods are used to mitigate against overfitting, we shall consider these methods in later chapters.

Underfitting can also occur, but this is not due to the form of our loss function or the size of our parameter values. This is a case whereby the model performs poorly on the training set often due to the parameters being insufficient to model the relationship between the target and the attributes. This is due to having insufficient number of features. For example, just providing the land area of a house and its location might not be enough to predict the price with high accuracy, this is underfitting, however, adding more features like the height of the house, color and type would introduce extra parameters into our regressor and better model the relationship. Another reason is having a model that is too linear in nature, when the true relationship is polynomial, we would need polynomial features to properly model the relationship.

For example, if we are trying to predict a variable y based on a single feature X

The standard linear regression is

$$y = \theta X + \beta$$

This assumes the relationship is linear. In real life, most relationships are not linear, hence, applying this model to many problems would lead to high inaccuracy.

If the relationship is quadratic, we can make our model quadratic by creating new features from the existing feature.

Our model should now look like this

$$y = \theta_1 X + \theta_2 X^2 + \beta$$

This introduces a new feature X^2 derived from X and an extra parameter θ_2. This model is guaranteed to accurately model any quadratic function. We could go on and on by introducing new polynomials, for example we could write the model as

$$y = \theta_1 X + \theta_2 X^2 + \theta_3 X^3 + \theta_4 X^4 + \beta$$

However, this must be done with caution, if we arbitrarily introduce polynomial features, we run the risk of overfitting our dataset.

Note that the test data is often smaller in size than the training data while the validation data is often smaller than the test data. There are no rules about this, however, it is conventional to do so.

Optimization

Finding the right set of parameters is the single goal of training neural networks. Once we have an optimal set of parameters, we can easily make new accurate predictions with our model. Finding the right set of parameters entails finding parameters that yield the lowest error on the training dataset. This process is called optimization, and it is a common mathematics problem that extends beyond the borders of machine learning. Optimization seems very straightforward, however it gets very complicated as the number of parameters increase. One way to optimize our parameters is to randomly guess their values and see which guess results in the lowest error. This can work for a few parameters, however, with every single additional parameter, the number of possible guesses increases exponentially. To make matters a lot more complicated, the parameters depend highly on each other. The value of a single parameter in a layer would be connected to every layer after it, hence , any inaccuracy in a single parameter in layer 1 would be propagated to layer 2, 3 till the output layer. Modern machine learning problems typically involve millions of parameters, guessing the combination of a million parameters is practically impossible. This is related to the concept of "The Curse of Dimensionality" As the dimensions increase, the possible combinations of the parameters increase exponentially.

To resolve this, an optimization algorithm known as Gradient Descent was developed.

The foundation of gradient descent is calculus, however, the concept is abstract enough to be explained in a simple way. It can be illustrated with a classic example of a man on top of a mountain, he needs to descend to the bottom of the mountain but it is dark and he has only a lamp with him. Consider the mountain as our loss, we want the loss to be at its minimum. In this example, the mountaineer has no idea in what direction the bottom of the mountain is. The obvious thing he would do is to look around him to see which point steeps downward. He can only see as far as his lamp can illuminate, but as he continues to move in directions that would take you down the mountain. As you continue this careful journey down the mountain, you would end up at the bottom. In Gradient Descent we are trying to optimize many parameters, these parameters form your spatial dimension, you want to move in such a way that would respect all the dimensions. In a two parameter setting, you move only in X and Y dimensions, but in machine learning, you typically have millions of dimensions.

To determine the dimensions to move, we calculate the gradient of the loss with respect to each parameter, this gives us a perfect direction in which to adjust our parameters (move) in such a way that we move down the mountain of losses. The rate at which we move in this direction is determined by the learning rate denoted α, when α is too high, we are moving too fast, and such reckless move might would cause us to overshoot the minimum position

Below is an illustration. Here we use just a single parameter to simplify the visualization.

The above is a convex function, as you can see, when the learning rate is high, Gradient Descent overshoots the minimum and goes higher in the direction of the loss. However, when the learning rate is lower, it converges to the minimum.

The general formula for Gradient Descent is

$$\theta \rightarrow \theta - \alpha \frac{\partial L}{\partial \theta}$$

This might look a bit complicated, but it's actually simpler than it looks

It can be broken down into three steps:

1. Find the gradient of the loss with respect to the parameters.
2. Multiply the gradient by a hyper-parameter known as the learning rate
3. Subtract the result of [2] from the parameters.

This very straightforward approach often yields optimal parameters after many epochs.

The rate of update to the parameters is determined by the α hyper parameter, called the learning rate. Typical values from these range between 0.1 and 0.001, however, in practice, the value is lowered after a number of steps.

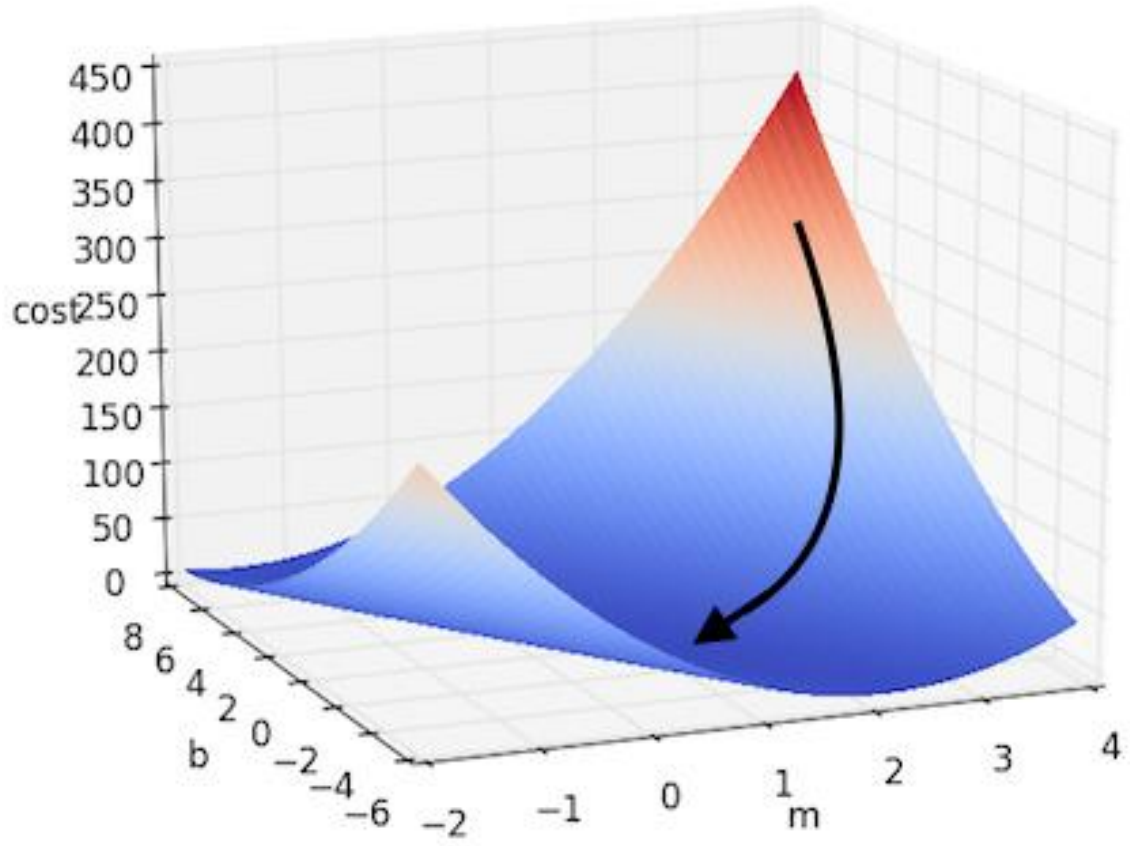
The gradient is usually calculated through a technique called *Back Propagation*,

This is an advanced topic, which you do not need to fully understand as a beginner, however, if you want to get deeper into this, read Michael Nielsen's tutorial.

Gradient Descent works excellently well, however, performing gradient descent updates over an entire training set is computationally intractable when the data is very large. This becomes a serious problem when dealing with high dimensional data like images, where the dataset could range from hundreds of megabytes to gigabytes of data. Applying gradient descent on such large amounts of data is infeasible, hence, modifications were developed, namely ***Stochastic Gradient Descent.***

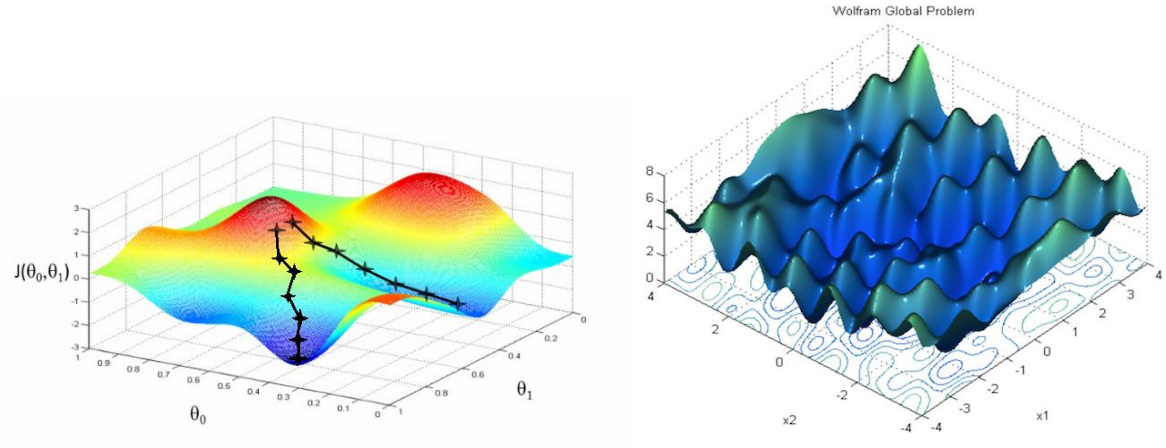
Stochastic Gradient Descent also called MiniBatch Gradient Descent operates over a batch of the dataset at a time. Instead of computing gradients based on loss aggregated across all examples in our dataset, the gradient is based on only the data contained in a single batch. Hence, we update the gradient as we compute the loss batch by batch until we exhaust the number of batches. This makes training on large datasets much easier and computationally efficient. The batch size controls the computational cost of the training. Typically, when training image recognition systems, common batch sizes are 32, 64 and 128.

Stochastic Gradient Descent helps us to take small steps in the direction of optimality, these steps are often stochastic, i.e. random, however, they are guaranteed to reach a local minimum and if training is done properly, they would reach a global minimum. Now, we are getting ahead of ourselves here with the concepts of local and global minimum. Generally, there are many parameters that need to be optimized and the steps are taken in such a way that minimizes all of them, more importantly, in linear regression, the loss function relative to the parameters is convex. For a two parameter setting, the shape is as below



Source: Machine Learning for Artists (ml4a.github.io)

A convex function has only one minimum point, which is also the Global Minimum. However, due to the application of non-linear activation functions such as Sigmoid and RELU, the shape of the loss function plotted against the parameters is non-convex, there are often many Local Minimum, points at which gradient descent would converge without being optimal. See example images below.



Source: Machine Learning for Artists (ml4a.github.io)

As you can see, so many local minima exist, ordinary gradient descent would most likely converge to a non-optimal local minimum. However, in Stochastic Gradient Descent, most local-minima are very close to the global minimum, hence, it often converges to a more optimal parameter set.

Other modifications to the Ordinary Gradient Descent exists, these are usually better. However, their form is more complex and is beyond the target of this book. Some of them include Gradient Descent with Momentum, Adagrad, AdaDelta, RMSProp and Adam Optimizer.

THE STRUCTURE OF IMAGES

The next chapter would be more practical, however, a comprehensive understanding of the way images are structured and represented in computer memory is absolutely essential in computer vision. Without it, you would be lost in a wilderness of confusion.

To the eye of man, every image is a set of well-defined patterns, but in computer memory, every image is just a Tensor of pixel values.

In case you are not very familiar with advanced linear algebra. A tensor is an N dimensional vector. A matrix is a 2 Dimensional Vector, its dimensions are its columns and rows which can also be described as its width and height. A 3 dimensional Tensor would have depth in addition to width and height, these can be used to represent any 3D shape such as cubes. Higher dimensions exist, there is no theoretical limit to the number of dimensions a Tensor can have.

Back to image structures, pixels are intensities of light. Their values in range between 0 – 255.

Images can either be grayscale (black and white) or RGB (colored).

Grayscale images are 2 Dimensional Structures, they are made up of different shades of gray in between 0 (white) and 255 (black).

156	125	0	8
5	3	90	3
0	255	0	1
200	120	213	190

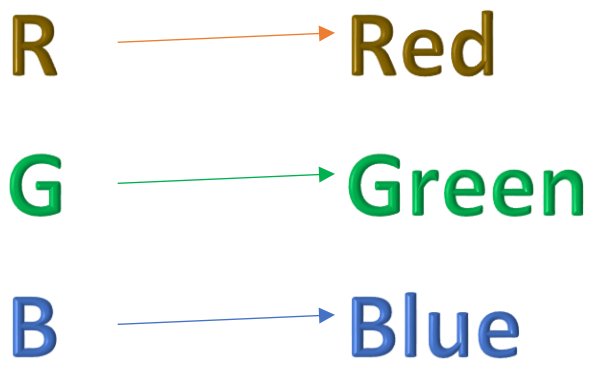
The diagram above is a 4 x 4 matrix of pixel values contained in a grayscale image.

If you look carefully at the image, this pattern represents the figure 2. If this isn't clear to you, below is a sparse matrix excluding all the lower pixels.

156	125		
		90	
	255		
200	120	213	190

This is clearly a 2. Pixels form patterns. These color intensities are what we see every time we look at a picture on our digital devices.

RGB Images are similar to grayscale images with the major difference being the presence of three channels.



RGB images are 3 Dimensional Tensors, like grayscale images, the pixels are arranged in rows and columns, but unlike grayscale images, three different channels of rows and columns are stacked upon each other. The RED channel represents intensities of red light, the GREEN channel represents intensities of green light and the BLUE channel represents intensities of blue light.

Note that these are the Primary Colors and they can be combined to form any other color. Hence, all images you see are made up of three channels, RED,

GREEN and BLUE stacked upon each other, the intensity of RED, GREEN and BLUE light at each point along the Y (rows) and X axis(columns) would determine the color at that specific point.

An RGB image has the shape (width x height x 3) where 3 is the number of channels. In contrast, a grayscale image has only one channel.

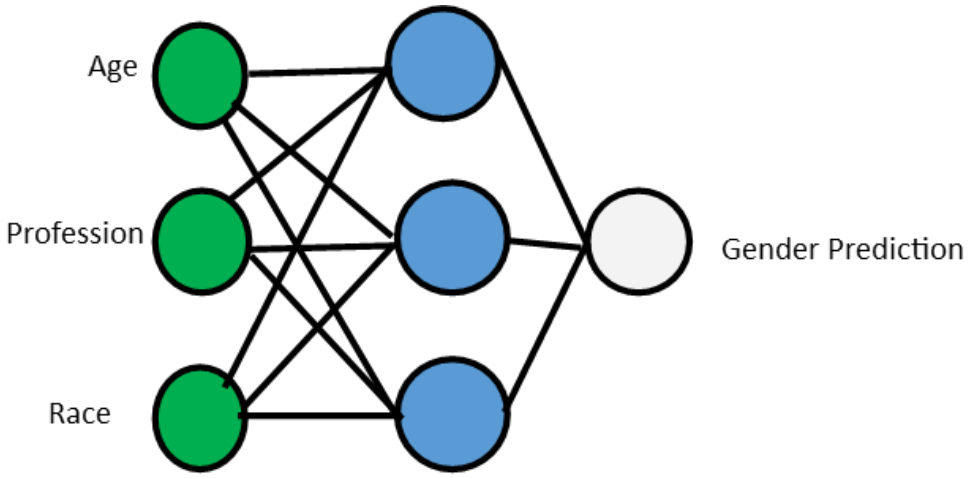
When performing image recognition, the RGB or grayscale pixel values are the only information available to our classifier, however, as seen above, the structures represents well defined patterns and is sufficient information to tell what is contained in an image. Videos are made up of many frames of images. A video is just like a series of images moving very fast. Hence, the pixel values are sufficient to perform prediction on videos as well.

ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING

So many machine learning algorithms have been developed. Chief of all of these is neural networks, a class of machine learning algorithms that has proven to be unreasonably effective at almost any task we throw at it. It has been used to surpass human performance on very complex tasks including image recognition and speech recognition. Their uses have expanded to the field of robotics and data analysis. For any Artificial Intelligence task, neural networks are the go-to method. In this chapter, we shall explain how they work, why they are so effective and how to use them yourself using a deep learning library named Keras.

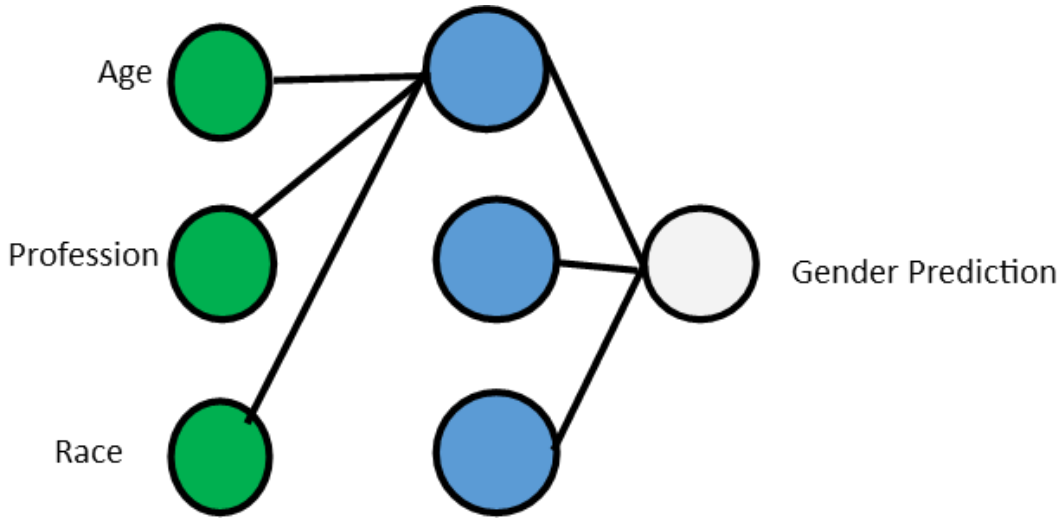
What are Artificial Neural Networks

As the name suggests, neural networks are inspired by neuroscience. They are essentially a network of interconnected neurons. Each neuron is a mathematical function. Decisions are made in neural networks based on the activations of these neurons. To understand this concept, look at the image below.



Above is a simple 3 layers neural network, the layer of green dots is the input layer, each single dot represents a feature. The features here are Age, Profession and Race. The next layer is the hidden layer of neurons. Every single blue circle(neuron) in this layer would take in all the previous neurons. If you look at the picture clearly enough, you would see connections from Age, Profession and Race going into each of the blue neurons.

To understand this better, look at the image below



Here, we stripped out the connections to the other neurons and instead show the connections each neuron receives. As you can see, all the previous inputs go into the next neuron directly.

The outputs from the middle layer are then forwarded to the final layer which decides based on the input, whether the subject is male or female.

This example is over-simplified so we can focus on the most important components you need to know. In practice, neural networks are made up of so many layers and so many neurons in each layer. The high number of layers in neural networks is the reason why they the field is called **Deep Learning**.

Also note that the number of neurons in the middle layer above could be 5, 10 or more irrespective of the fact that the first layer had just three features. Keep this in mind.

So far, we haven't explained what actually goes on within the neurons. As we earlier said, they are mathematical functions which takes in inputs and returns outputs.

The most fundamental operation that occurs within neurons is a linear function.

Let ***Q*** represent each of the blue neurons. Remember that ***Age***, ***Profession*** and ***Race*** are the values we feed into ***Q***.

The following happens in ***Q***

$$\boldsymbol{Q = \theta_1 Age + \theta_2 Profession + \theta_3 Race + \beta}$$

The equation above is the simple linear regression algorithm. We have here a set of Parameters $\boldsymbol{\theta_1, \theta_2, \theta_3}$ **and** $\boldsymbol{\beta}$. However, you might be wondering how we can multiply categorical variables like Profession and Race with real valued parameter values. In machine learning, there are tricks for dealing with categorical variables. We shall come back to this soon, lest we lose focus on the topic in this context.

This first thing we do with the inputs to each neuron is the equation above, but we do not output the result of the equation, instead, the output is passed into an activation function, the result of the activation function would become the output of our neuron.

The equation for each neuron becomes

$$\boldsymbol{Z = \sigma(\theta X + \beta)}$$

Where **Z** is the output from each neuron, $\boldsymbol{\sigma}$ is the activation function, $\boldsymbol{\theta}$ is the vector of parameters contained in each neuron, $\boldsymbol{X}$ is the vector of features and $\boldsymbol{\beta}$ is a scalar denoting the bias.

What we have done here is simply put linear function into an activation function $\boldsymbol{\sigma}$.

Different activation functions exist, and we shall explain the most important ones.

The Perceptron

This was the very first activation function, before others came. It is no longer much in use, but it is very important to start with it.

The perceptron takes the form

$$\sigma = f(Z) = \begin{cases} 1, & Z > 0 \\ 0, & Z \leq 0 \end{cases}$$

This entails that the activation function σ is a function f that takes in input Z (the output of our linear equation) and returns 1 if Z is greater than zero and returns 0 if Z is zero or less.

This is a very simple binary function; the output of perceptron neurons is always 0 or 1. This is analogous to logic gates. It is always true or false.

This output is passed forward to the next neurons in the layers after or to the output neuron if the next layer is the output layer. The output of the final neuron, white in the gender example would also be a zero or 1. In this case, depending on how we encode the target, an output of 1 can indicate male and an output of 0 can indicate female.

The perceptron is very simple and effective for simple tasks, but it is not a really great algorithm and it just can't solve some tasks properly. As you might have observed, a limitation of the perceptron algorithm is that, it can only tell whether a condition is true or not, this does not really conform to the core principles of machine learning which depends on probabilities of events rather than absolute certainties. While it does involve probabilities, we are limited to a yes or no answer, which is hardly satisfactory. A condition might be true, but the best decisions are made when we know how true it is. For example, when taking temperature readings, we might say a place is cold because the temperature is below 25 Celsius, but that doesn't tell us how cold it is, 0 Celsius and 15 Celsius are very different weather conditions, and often our decisions under these two different circumstances would be very different. Due to this limitations, better activation functions have been developed, one of which is the Sigmoid function.

Sigmoid

These takes an input and smashes it into values between 0 and 1. The form of the function is:

$$\sigma = f(Z) = \frac{1}{1 + e^{-Z}}$$

This function could really be scary at first sight, but it's actually very simple. The only unusual member of this function is e. This is the base to the natural logarithm of numbers. Its value is **2.71828**

If the result of the linear equation is 2, the output of the sigmoid would be calculated as follows.

$$\sigma = \frac{1}{1 + e^{-2}} = \frac{1}{1 + 0.1353} = \frac{1}{1.1353} = 0.739$$

As you can see from the flow above, the sigmoid function takes in any number and transforms it into a value between 0 and 1.

This is way better than our perceptron based neurons. First, it can take as input, any number between 0 and 1 and can output any value between 0 and 1. So now we can determine the degree of coldness and not just "cold" or "not cold".

RELU

Rectified Linear Units (RELU) and its variants are at present the best and most commonly used activation functions especially in the domain of computer vision. Fortunately, they are also much simpler compared to the sigmoid function. It takes the form;

$$f(Z) = \begin{cases} Z, & Z > 0 \\ 0, & Z \leq 0 \end{cases}$$

The above formula nearly matches the perceptron with one single difference, in the perceptron, we return 1 whenever Z is greater than zero. In RELU, we return Z itself when the value is greater than zero and we return 0 when the number is zero or less.

Most often, you would see the formula of RELU as

$$f(Z) = \max(0, Z)$$

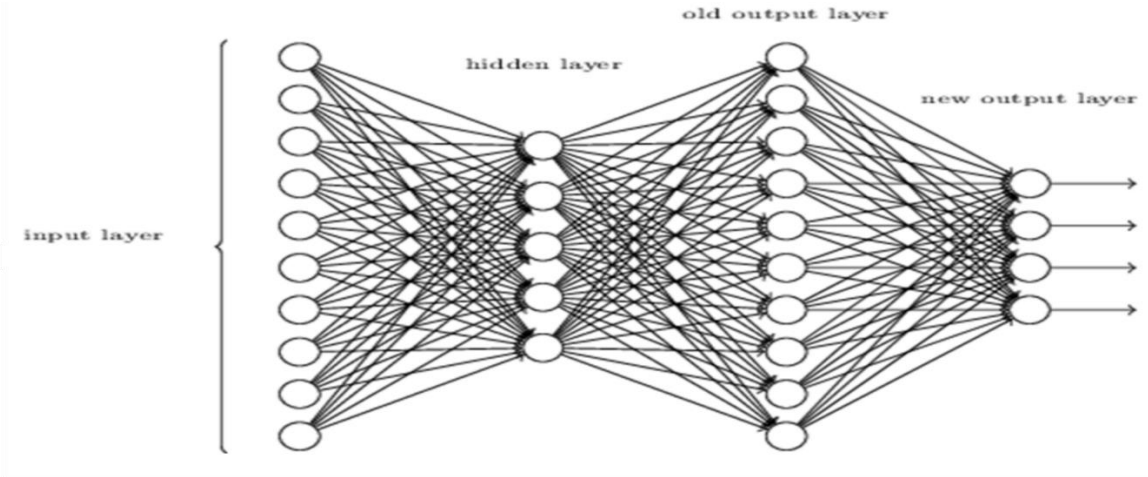
This is absolutely equivalent to the first formula we gave, just in a more compact form. It entails that the function would return the maximum between 0 and Z, hence if Z is negative, which means less than 0, the function would return 0, but if it is greater than 0, it would return Z, when Z is equal to zero, it is obvious that the return value would be 0.

This is the most convenient formula for RELU, we gave the initial one so you can see the similarity to the perceptron.

Final Layers

The final layer of an artificial neural network can have more than one output, especially in the classification setting.

Consider the image below



Source: Michael Nielsen (neuralnetworksanddeeplearning.com)

The above is a little deeper than our previous examples, here we have two hidden layers (middle layers) and an output layer with four neurons.

When performing classification, the number of neurons in our output layer would be equal to the number of classes in our dataset. In a profession classification setting, if the set of professions include only Doctors, Nurses and Lawyers, we would have three neurons in the final layer, each neuron corresponding to our score for each class, the class with the highest score becomes our prediction.

Why Hidden Neurons

The concept of hidden neurons can seem confusing a little bit. We are used to feeding data into a single functional layer that then gives us outputs, in fact our previous diagrams show a very similar pattern, but the last diagram above contains additional two hidden units, and in practice we could have so many more hidden layers. This multiple layers are actually the secret of the success of neural networks.

Neural networks are based on function decomposition. Every layer is a function of the layer before it.

$$L_n = f(L_{n-1})$$

As the layers go deeper, the representation becomes more powerful. By decomposition functions into sub-functions, we are able to represent very complex relationships and highly implicit correlations.

Instead of having just a single layer function with neurons to model the relationship between a target and the attributes, we instead have a stack of functions. The benefits of this hierarchical setting is well observed in an image classification setting. A man is a function of the head, legs, arms and other parts, while each part like the head is itself a function of the nose, eyes, ears, mouth. The nose also differs from person to person, and we can as well have a function modelling the structure of the nose. A single layer would only model the man as a function of the head, legs and arms. But it would fail to model the composition of the head and other parts.

Function decomposition also makes neural networks highly invariant to the nature of different datasets. Observing decomposition from a mathematical point, if you have a number 125, we can choose to recognize it as an ordered combination of 1, 2 and 5. That looks great and consistent because we always work with base 10. However, if we decide to use base 2, the same number 125 can no longer be interpreted as an ordered combination of 1, 2 and 5. Here it becomes confusing how to represent this number in a way that would be consistent across all bases. To resolve this, we can simply decompose 125 into its prime factors. Hence 125 becomes a 5 x 5 x 5. Irrespective of the base, 125 = 5 x 5

x 5 holds. The actual representation of each 5 component would differ but the representation of 125 remains exactly the same across all bases.

The same concept applies to neural networks.

Artificial Neural Networks are universal function approximators. Given sufficient number of neurons and layers, they can model any relationship, irrespective of the complexity.

NEURAL NETWORKS IN ACTION

So far we have discussed the theories of machine learning and neural networks.

In this chapter, you would learn how to apply neural networks to the task of image classification. Everything you have learned; Datasets, Loss functions, Optimization and the neural network structures would all come into play here.

The Data – MNIST

The exact problem we are to solve here is to classify 10 000 images of handwritten digits. This Dataset was compiled by Yann LeCun, from the main NIST Database of handwritten digits. Note that these are handwritings of real people.

MNIST is made up of 70 000 images. This is divided into two sets: The training set with 60 000 images and the test set with 10 000 images. We did state earlier that we have to classify 10 000 images, this refers to the test set, which in this case is acting as our benchmark for how good our neural network is.

The images are 28 x 28 grayscale pixels. The entire dataset is approximately 15 mb in size, you can download it yourself from www.yann.lecun.com/exdb/mnist

However, the Deep Learning library we are going to use has functions to automate the downloading and loading of the MNIST dataset, so you can focus on the code aspect.

Environment Setup

All practical in this book use Python 3.5 and the Deep Learning libraries; Tensorflow and Keras. We assume you are familiar with the python programming language.

First you need to install python 3.5 if you haven't done so already.

Then you need to install two python packages

Install Tensorflow

Tensorflow is the most popular Deep Learning library, righty so because it is incredibly robust and contains implementations for almost all the oldest and

newest algorithms used in machine learning. For more information about tensorflow, visit tensorflow.org.

If you have a system with an NVIDIA GPU. Then you need to install the CUDA Framework and CudNN in order to take advantage of the GPU. GPUs are about 100 – 300 times faster than the best CPUs when training neural networks. The reason for this is that they can do parallel computation much better than CPUs with limited cores.

Once you have installed CUDA and CudNN, you should install the GPU version of tensorflow via pip

```
pip3 install tensorflow-gpu
```

If you do not have a NVIDIA GPU. You should install the CPU version

```
pip3 install tensorflow
```

Note that no other GPU except those from Nvidia would work for most deep learning libraries. If your GPU is not from Nvidia then, you have to bear with the speed of the CPU. GPU as a service offerings are available from major cloud providers, we shall emphasize on this when we get to the chapter on CONVOLUTIONAL NEURAL NETWORKS

Install Keras

Keras is a single unified API that runs across major deep learning libraries including tensorflow, theano, Microsoft CNTK, MXNET and Deeplearning4J. It is also very simple to use. It is not technically a deep learning library itself, instead, it abstracts the backends listed above. Tensorflow already includes the keras library however, we would rather use the general keras distribution, as this would allow you to change to other backends if you so desire. MXNET and CNTK for example are known for being much faster.

Install keras via pip

```
pip3 install keras
```

Last, we would need the h5py package for saving models and matplotlib for visualization purposes. Install them via pip

```
pip3 install h5py
pip3 install matplotlib
```

If you do not already have a python IDE, I suggest installing PyCharm, it is a very fantastic IDE with all the features you would love as a python developer, plus it allows you to easily click on functions and classes and dig throught their source code. PyCharm is more memory consuming though, if you prefer a more light IDE, install SublimeText.

Now we are setup!

Our First Neural Network

Fire Up your IDE and create a new python file.

Step1:

Add some important import statements

```
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
```

Some of the imports above are self-explanatory, first we need to import keras itself, then we import the mnist class which handles the downloading and loading of the MNIST dataset, The Dense class represents each layer in a neural network, Sequential holds a stack of Layers and finally we import the optimizer we need, in this case SGD (Stochastic Gradient Descent). All of these is pretty basic.

Step2. Load the data

```
(train_x, train_y), (test_x, test_y) = mnist.load_data()
```

Here we call the load function of the mnist class and it returns the training images and their labels, (train_x for images, train_y for labels) as well as the test images and their labels.
Notice that the first time you call this function, the MNIST dataset would be automatically downloaded and cached for future use. So ensure your system is connected to the internet the first time you run this code. The entire MNIST is just 15 mb, so with a good internet connection, in a minute, it will be done.

Step3: Normalize the data

```
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255
```

In machine learning, it is often preferable to work with smaller values, computation with larger numbers is often numerically unstable. What we did here is to scale the pixel values from 0 to 1 by dividing every pixel with 255 which as we earlier explained is the maximum value a pixel can be.
This process is called normalization

It is also common to normalize the pixel values between -1 to +1.
Here is the code to do this

```
train_x = train_x.astype('float32') - 127.5 / 127.5
test_x = test_x.astype('float32') - 127.5 / 127.5
```

The major difference is that, in this one, we first subtract 127.5 (half of 255) from each pixel, we then divided the result by 127.5.
The choice of which to use is yours, but the first is just cool and we shall use it throughout this book.

To verify that everything is setup up properly, we add code to print the shape of our data arrays, run the full code below.

```
#import needed classes
import keras
```

```
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)
```

The output of this code should be:

```
Train Images:  (60000, 28, 28)
Train Labels:  (60000,)
Test Images:  (10000, 28, 28)
Test Labels:  (10000,)
```

The output is consistent with our earlier description of the MNIST dataset. As you can see there are 60000, 28 x 28 pixel images in the training set and 60000 labels for the 60000 images, the same goes for the test set except that the images and labels are 10000.

If you get the same output, then everything is setup correctly.

Step 4: Flatten the image pixels

```
train_x = train_x.reshape(60000,784)
test_x = test_x.reshape(10000,784)
```

Here, we reshape each of our 28 x 28 pixels into a 1 Dimensional array of 784 pixels (28 * 28)

This is an undesirable but necessary step. Undesirable because by flattening our image into a single dimension, the entire 2 D structure of the image is lost. It is necessary because the fully connected neural network we are about to develop cannot handle such high dimensional data. However, in the next chapter, we shall explain Convolutional Neural Networks, a special class of neural networks that takes full advantage of the structure of any pattern.

Step 5: Convert Labels to Vectors

```
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)
```

Machine learning algorithms cannot work with categorical labels such as the string values representing the label of each image in the MNIST dataset. To deal with this problem, a technique called one-hot encoding is used to convert the labels to vectors:

This process is very simple, imagine if we had the following labels,
Dog, Cat and Lion.
To convert these to vectors, we would assign to each of them a vector with zeros in all positions except the index of the vector. So
Dog = [1,0,0]
Cat = [0,1,0]
Lion = [0,0,1]

However, keras already provides a handy function to do this for you. As seen above, the function converts each label into a vector of 10 elements. Just like the Dog is a vector of 3 elements. The number of classes would determine the number of elements in the vector.

Step 6: Define your model/network

```
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
```

```
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))
```

This network is a four layer fully connected neural network. The Dense function represents each single layer, the number of units is the number of neurons in the layer, the activation part has been explained earlier, the input shape passed to the first layer is necessary for keras to determine the shape of the data. In this case, the shape of each 28 x 28 image has become 784 after we flatten it.

The other layers would automatically infer their input shapes.

The last layer uses the softmax function we explained under Loss Functions. Since our classes are ten, 0 – 9, this final output layer needs exactly ten neurons to output ten softmax computed probabilities for each class.

Our output would be an array of ten probabilities.

Step 7: Compile the function

```
model.compile(optimizer=SGD(0.01),loss="categorical_crossentropy",metrics=["accuracy"])
```

This part is very vital to our training procedure, first we specify the optimizer, which in this case is SGD (Stochastic Gradient Descent) with a learning rate of 0.01. This is a very good learning rate value, if it is too small, training would become unnecessarily slow, while if it's too high, SGD is likely to overshoot, failing to converge to neither a global minima nor a local minima.

Next we specify our loss function, in this case, categorical_crossentropy , just another name for softmax crossentropy which we discussed under Loss Functions.

Finally, we state the metrics we want to obtain, in this case we are most interested in the accuracy, i.e the ratio of images classified correctly. In a classification setting, we are most interested in the accuracy metric, however, in regression setting, we are primarily concerned with the Mean Squared Error.

Step 8: Fit the function

```
model.fit(train_x,train_y,batch_size=32,epochs=10,shuffle=True,verbose=1)
```

Here we pass the training images and their corresponding labels into our model, we also specify the batch size to be 32. This entails we want the model to process only 32 images at a time, finally, we specify that training should run for 10 iterations. Note that at each iteration, the training would be done on all the batches. In this case, each of the 10 iterations is divided into 60000/32 iterations.

Verbose = 1 simply tells keras to log to the console at every iteration.

Calling the fit function would invoke the training process right away.

Final Step: Evaluate Accuracy

```
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

After training is complete, we need to know the performance of our model on the test dataset, hence we pass in the test images and their corresponding labels to the evaluate function of the model and it would return the accuracy as the ratio of images classified correctly.

Here is the full code

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
```

```
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)

#Flatten the images
train_x = train_x.reshape(60000,784)
test_x = test_x.reshape(10000,784)

#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))

#Specify the training components
model.compile(optimizer=SGD(0.01),loss="categorical_crossentropy",metr
ics=["accuracy"])

#Fit the model
model.fit(train_x,train_y,batch_size=32,epochs=10,shuffle=True,verbose
=1)

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

This code should complete the 10 epochs within 3 – 5 miniutes depending on your CPU. On GPU, in less than a minute, it is done.

Your test accuracy should be about 97%.

That's very exciting but we can do better. Try to achieve a better accuracy yourself by tweaking the parameters.

Here are some tips

1. Increase the number of epochs to 20

2. Increase the number of neurons in each layer
3. Try different learning rates
4. Try replacing SGD with other optimizers you would find in the keras.optimizers package.
5. Increase the number of layers

Try this experiments on your own and see if you can get a higher score than 97%.

KERAS BASICS

Now that you have successfully created your first image classification system, before you proceed to learning convolutional neural networks, it is important that you have a basic understanding of how things are done in keras.

Saving and Loading Models

We train our models once and then use them for prediction every time we need to. This is not apparent from the last chapter as we ran evaluation only after training. The key is to save our models permanently. We can then load the model file anytime and use them for evaluation/prediction without training them again.

Keras provides a very simple function to save and load models.

To save a model, simply add this after the fit function

```
model.save("mnistmodel.h5")
```

This would save your model as a file named mnistmodel.h5, note that the extension is not fixed, you can decide to save it as mymodelname.model or anything else that you wish.

Once your model is saved, to perform prediction, you don't need the fit function anymore, simply remove or comment it out and replace it with the code below

```
model.load_weights("mnistmodel.h5")
```

You can then proceed to perform evaluation on your test set and your accuracy would still remain intact.

Your new code for evaluation would then be

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense,Dropout
from keras.models import Sequential
from keras.optimizers import SGD

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
```

```
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)

#Flatten the images
train_x = train_x.reshape(60000,784)
test_x = test_x.reshape(10000,784)

#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))


#Specify the training components
model.compile(optimizer=SGD(0.01),loss="categorical_crossentropy",metr
ics=["accuracy"])

#Uncomment both lines during training
#model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,verbos
e=1)

#model.save("mnistmodel.h5")

#Comment out this line during training
model.load_weights("mnistmodel.h5")

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

Predicting Specific Images

Often we are not interested in predicting the entire evaluation set, we instead want to predict the class of a specific single image. Here we shall pick a specific image from the test set and predict the class.

Here is full code to do this

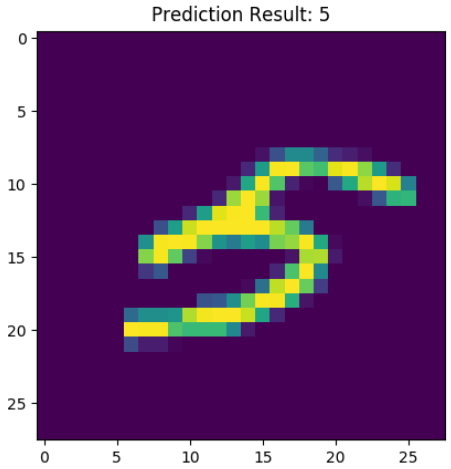
```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense,Dropout
from keras.models import Sequential
from keras.optimizers import SGD
import matplotlib.pyplot as plt

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#Define the model
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))


#Specify the training components
model.compile(optimizer=SGD(0.01),loss="categorical_crossentropy",metr
ics=["accuracy"])

#Load the pretrained model
model.load_weights("mnistmodel.h5")

#Normalize the test dataset
test_x = test_x.astype('float32') / 255

#Extract a specific image
img = test_x[167]

#Create a flattened copy of the image
test_img = img.reshape((1,784))

#Predict the class
img_class = model.predict_classes(test_img)
classname = img_class[0]
print("Class: ",classname)

#Display the original non-flattened copy of the image
plt.title("Prediction Result: %s"%(classname))
plt.imshow(img)
plt.show()
```

Notice some different things here, first we have excluded the fit function since we are not training. We also avoided normalizing the train dataset since we are not making use of it anyway. Then we extract an image at a specific index, your index could be anything between 1 to the 10000 images contained in the test set.

We create a flattened copy of the image while keeping the original intact for visualization purpose.

The flattened image is passed into the `predict_classes` function and it returns our class prediction, we print this to the console. We then create a plot with matplotlib, setting the title to reflect our prediction and we call `imshow` to display the original image.

The output for this example is this.



As you can see, the prediction is very accurate, if you used a different index, then the digit might be different.

We can also load in image from an external source and feed it into our classifier.

Download this image from our github repository.

And run the code below

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
import matplotlib.pyplot as plt
from keras.preprocessing import image

#Define the model
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))


#Specify the training components
model.compile(optimizer=SGD(0.01),loss="categorical_crossentropy",metrics=["a
ccuracy"])

#Load the pretrained model
model.load_weights("mnistmodel.h5")


#Load an image from your system
img = image.load_img(path="testimage.png",grayscale=True,target_size=(28,28))
img = image.img_to_array(img)
img = img.reshape((28,28))

#Create a flattened copy of the image
test_img = img.reshape((1,784))

#Predict the class
img_class = model.predict_classes(test_img)
classname = img_class[0]
print("Class: ",classname)

#Display the original non-flattened copy of the image
plt.title("Prediction Result: %s"%(classname))
plt.imshow(img)
plt.show()
```

The only difference here is how we load the image from the system. Study the code and run it. Check the results yourself.

Dynamic Learning Rates

So far we have only used a fixed learning rate, but in practice, when working with larger datasets where you need to run about 200 to 300 epochs. Using a single

learning rate would stop your accuracy from improving beyond certain points, the key is to gradually reduce your accuracy after certain number of epochs. If you started with a learning rate of 0.1, you might divide the learning rate by 10 after 30 epochs, 60 epochs and 90 epochs, hence at epoch 90, you would have much lower learning rate.

Keras provides a handy Learning Rate Scheduler to do this.

Below is our first example, modified to use a dynamic learning rate.

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.callbacks import LearningRateScheduler

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)

#Flatten the images
train_x = train_x.reshape(60000,784)
test_x = test_x.reshape(10000,784)

#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))
```

```
#Define the Learning rate schedule function

def lr_schedule(epoch):

    lr = 0.1

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
        lr = lr / 10
    elif epoch > 5:
        lr = lr / 5

    print("Learning Rate: ",lr)

    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
lr_scheduler = LearningRateScheduler(lr_schedule)

#Specify the training components
model.compile(optimizer=SGD(lr_schedule(0)),loss="categorical_crossentropy",metrics=["accuracy"])

#Fit the model
model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,verbose=1,callbacks=[lr_scheduler])

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

Here, we define a function that accepts the current epoch as a parameter and returns a new learning rate as needed. This function is passed to the learning rate scheduler that keras provides, finally we specify the Learning Rate Scheduler as a callback to keras.

Model Checkpoints

In our examples so far, we only save the model after complete training. However, in practice, you would want to save your model after very *N* epochs. The reason is that sometimes, our final epoch maybe less accurate than some epochs, usually

we want the best, so by saving many, we can go back to a previously saved model that is better than our final model.

Keras provides the ModelCheckpoint utility to handle this.

Here is the full code with model checkpoint

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import SGD
from keras.callbacks import LearningRateScheduler
from keras.callbacks import ModelCheckpoint
import os

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)

#Flatten the images
train_x = train_x.reshape(60000,784)
test_x = test_x.reshape(10000,784)

#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model
model = Sequential()
model.add(Dense(units=128,activation="relu",input_shape=(784,)))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=128,activation="relu"))
model.add(Dense(units=10,activation="softmax"))


#Print a Summary of the model
model.summary()

#Define the Learning rate schedule function

def lr_schedule(epoch):
```

```
    lr = 0.1

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
        lr = lr / 10
    elif epoch > 5:
        lr = lr / 5

    print("Learning Rate: ",lr)

    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
lr_scheduler = LearningRateScheduler(lr_schedule)

#Directory in which to create models
save_direc = os.path.join(os.getcwd(), 'mnistsavedmodels')

#Name of model files
model_name = 'mnistmodel.{epoch:03d}.h5'

#Create Directory if it doesn't exist
if not os.path.isdir(save_direc):
    os.makedirs(save_direc)

#Join the directory with the model file
modelpath = os.path.join(save_direc, model_name)

checkpoint = ModelCheckpoint(filepath=modelpath,
                             monitor='val_acc',
                             verbose=1,
                             period=1)

#Specify the training components
model.compile(optimizer=SGD(lr_schedule(0)),loss="categorical_crossentropy",m
etrics=["accuracy"])

#Fit the model
model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,verbose=1,call
backs=[checkpoint,lr_scheduler])

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

Here we specify that the model should be saved after every epoch, we could specify higher values with the period parameter.

Also notice the part `model.summary()`, this has nothing to do with saving models, its function is to print a full summary of all the layers and parameters of your model.

When you run this code, you should see an initial output like this:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
```

Total params: 134,794

Trainable params: 134,794

Non-trainable params: 0

As you can see, our network consists of 134,794 parameters.

Validation Data

You might have noticed that in the above examples, we were blind to the performance of the model on unseen data until the very last epoch. In practice, when we are running up to about 200 epochs, we wish to know the performance of our model at every epoch, besides, we can't truly identify the best epoch unless we provided a validation data. We can do these by either validating on the

test set at every epoch or specifying a validation split, in the latter case, the training data would be split according to the ratio we specify.

Try running the code above with the fit function changed as follows

```
model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,validat
ion_split=0.1,verbose=1,callbacks=[checkpoint,lr_scheduler])
```

Notice the `validation_split` above, by setting it to 0.1, we are using 10% of the training data for validation and 90% for training.
When you run the code, you would see a message like the following after each epoch
54000/54000 [==============================] - 18s - loss: 0.2953 - acc: 0.9080 - val_loss: 0.1223 - val_acc: 0.9622
The first accuracy report is training accuracy, the later accuracy report **0.9622** is the validation accuracy.

We could also use the test set for our validation set, in that case, our fit function would as this:

```
model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,validat
ion_data=[test_x,test_y],verbose=1,callbacks=[checkpoint,lr_scheduler]
)
```

Functional API

Keras has two APIs for constructing models, the first is the Sequential API which we have used so far for simplicity sake, however, going forward we shall be using the more advanced functional API. The advantages might not seem obvious at this stage, but it is absolutely essential when designing more complex networks, as we shall do in later chapters.

Here is our previous validation example written using the functional API.

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Model,Input
from keras.optimizers import SGD
from keras.callbacks import LearningRateScheduler
```

```
from keras.callbacks import ModelCheckpoint
import os

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)

#Flatten the images
train_x = train_x.reshape(60000,784)
test_x = test_x.reshape(10000,784)

#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model


def MiniModel(input_shape):
    images = Input(input_shape)

    net = Dense(units=128,activation="relu")(images)
    net = Dense(units=128, activation="relu")(net)
    net = Dense(units=128, activation="relu")(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model

#Print a Summary of the model

model = MiniModel((784,))

model.summary()

#Define the Learning rate schedule function

def lr_schedule(epoch):

    lr = 0.1

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
        lr = lr / 10
    elif epoch > 5:
```

```
        lr = lr / 5

    print("Learning Rate: ",lr)

    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
lr_scheduler = LearningRateScheduler(lr_schedule)

#Directory in which to create models
save_direc = os.path.join(os.getcwd(), 'mnistsavedmodels')

#Name of model files
model_name = 'mnistmodel.{epoch:03d}.h5'

#Create Directory if it doesn't exist
if not os.path.isdir(save_direc):
    os.makedirs(save_direc)

#Join the directory with the model file
modelpath = os.path.join(save_direc, model_name)

checkpoint = ModelCheckpoint(filepath=modelpath,
                             monitor='val_acc',
                             verbose=1,
                             period=1)

#Specify the training components
model.compile(optimizer=SGD(lr_schedule(0)),loss="categorical_crossentropy",m
etrics=["accuracy"])

#Fit the model
model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,validation_spl
it=0.1,verbose=1,callbacks=[checkpoint,lr_scheduler])

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

There are a couple of different things here

First import in two new classes, Model and Input

Most importantly is the MiniModel function, it could take any name

```
def MiniModel(input_shape):
    images = Input(input_shape)

    net = Dense(units=128,activation="relu")(images)
    net = Dense(units=128, activation="relu")(net)
    net = Dense(units=128, activation="relu")(net)
```

```
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model
```

The input_shape parameter specifies the shape of each image, which in this case was supposed to be (28,28) but due to flattening is now (784,)

Next we construct the images by passing in the shape to the Input function,

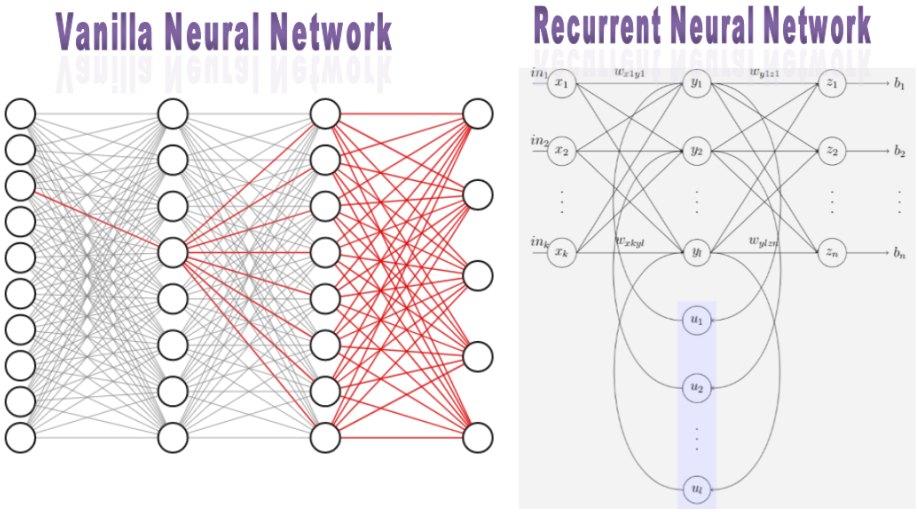
The rest are familiar, except that we explicitly pass in the images and outputs from previous layers into the new layers. This is where the real power of the functional API comes in. By explicitly passing in the input to each layer, we can effectively control the flow of information into each layer, we can also write conditional statements while constructing a network with the functional API.

Finally, we create an instance of the Model class and we pass in the images and the network. We return this model and everything else is just like we did in the previous examples.

There are many more different keras operations. However, the ones we have explained here will suffice for most work you would be doing in this book.

INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS

The vanilla neural networks that we used in the previous chapter are a perfect fit for tasks where neither the order of arrangement of the data nor the context is important. Tasks like Natural Language Processing, Speech Recognition and Time Series forecasting are context based problems. This set of problems are often solved with Recurrent Neural Networks, a special class of artificial neural networks where the neurons are far more complex, capable of learning what to remember and what to forget, and most importantly, are connected in a recurrent manner unlike vanilla networks that are connected in a feed-forward fashion.



Left Source: Machine Learning for Artists (ml4a.github.io), Right Source: Wikipedia

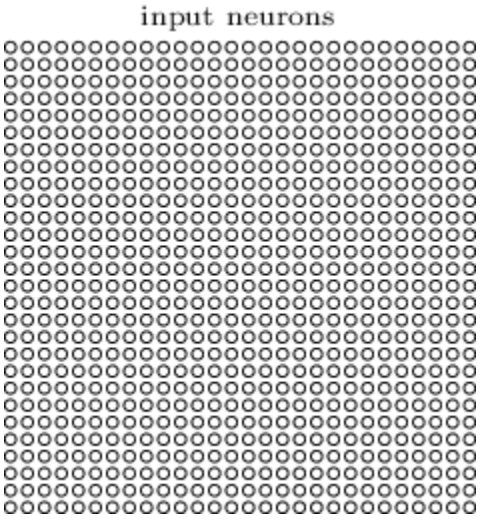
The second class of Neural Networks, which is the ultimate focus of this book is Convolutional Neural Networks. They are the best and most effective for all tasks where the order or arrangement of the data is of absolute importance. Computer Vision and Pattern Recognition falls into this category. Convolutional Neural Networks, henceforth to be called CNNs, were pioneered by Yann LeCun et al in 1998.

In this chapter, we shall explain the core concepts of Convolutional Neural Networks and how to apply them to the task of image recognition.

CNNs are feed-forward networks, just like the vanilla neural networks, however, they are locally connected while the vanilla neural networks are fully connected.

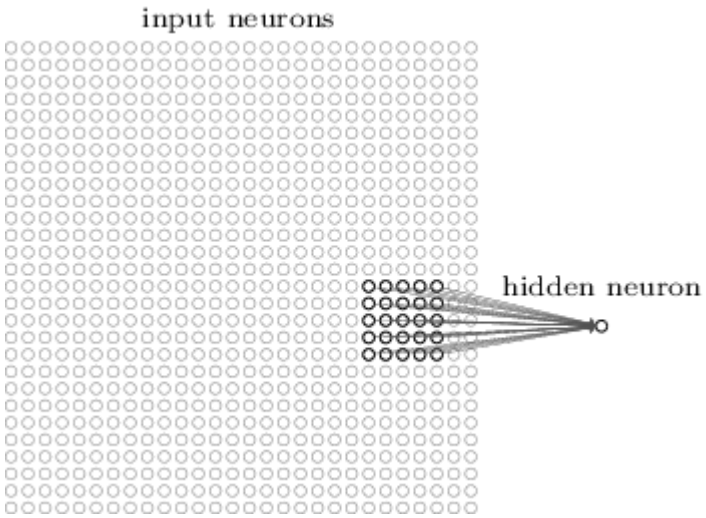
To gain a good understanding of CNNs, we have to first consider the problem of classifying images. Here we shall not flatten the images, instead we would keep the structure exactly as it is.

An image is composed of various patterns, in a human face, patterns include the nose, eyes, mouth, cheeks and other implicitly expressed patterns. These patterns are often a combination of smaller patterns which in turn are combinations of some patterns as well, forming a hierarchy of patterns.



Typical Structure of An Image. *Source: Michael Nielsen (neuralnetworksanddeeplearning.com)*

CNNs work by detecting specific patterns or features across the entire image.



Detecting A Single Pattern at a local Region of the Image, *Source: Michael Nielsen (neuralnetworksanddeeplearning.com)*

The more the number of feature detectors present in a CNN, the better it can classify images. A feature detector in a CNN is called a kernel or filter. The two terms are used interchangeably.

A filter is a fixed sized tensor. Each filter detects a specific pattern; multiple filters can detect multiple features. Each filter is a tensor of numbers representing the filter. Typical filter sizes are 5 x 5, 3 x 3, 2 x 2 and 1 x 1. However, the only limit to the size of a filter is the image size. Since, the maximum feature you can detect is the maximum size of the image. As you can see from the image above, we are detecting a very small pattern, this can reduce in size or increase in size up to the width and height of the image. Remember that each filter is a N Dimensional array of numbers with a dimension corresponding to the filter size.

The operation of a CNN can be broken down into the following steps

Step1. Compute the dot product of a filter with a small local region of the image

Step2: Repeat the first step by moving across the entire image, computing the dot product of the filter with a small N X N region of the image

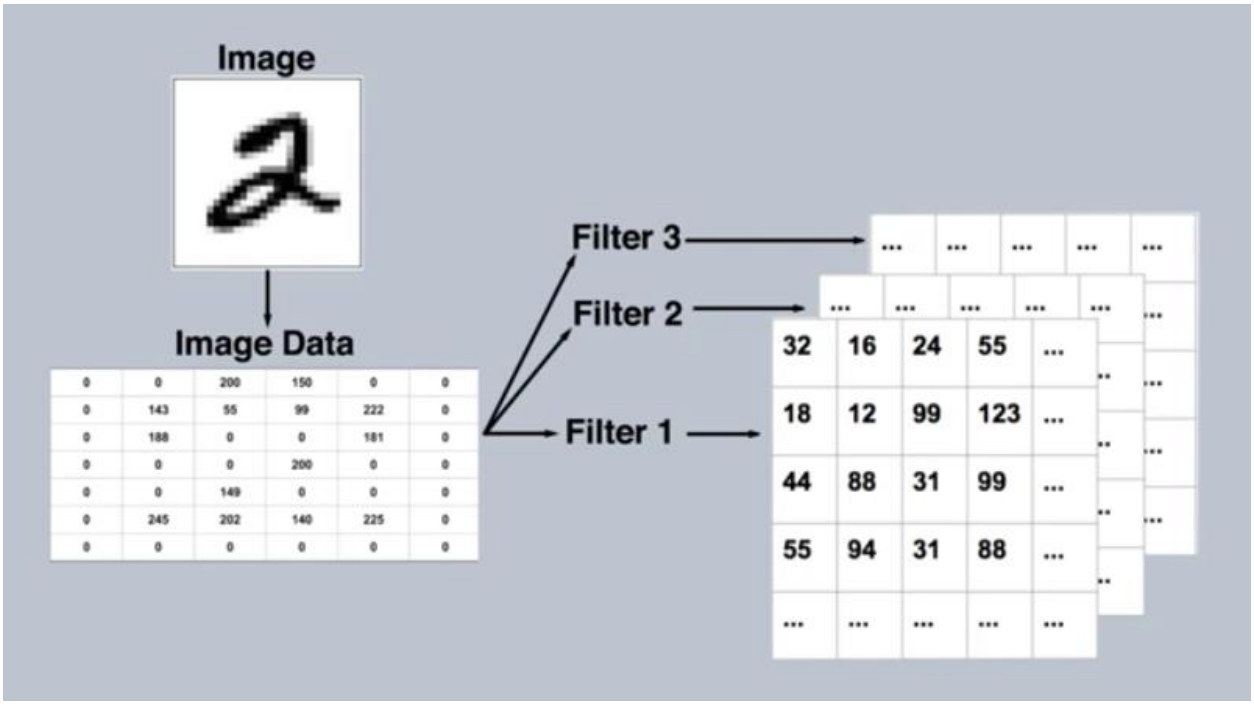
Step 3: Store the result of each region by region dot product in a 2D feature map. The 2D feature map represents the locations of the filter in the image. Points

where the values are high denote the presence of the feature defined by the filter, while points lower in value denote the absence of the feature in the image.

Step 4: Repeat the processes 1, 2 and 3 with as many filters as you wish to use, storing the output feature map of each

Step 5. Stack all the 2D feature maps from each filter on top of each other to form A 3 Dimensional feature map.

This is the basic operation of convolutional neural networks



Each filter produces a separate feature map, the feature maps are then stacked together, *Source: Kaggle Deep Learning Series (kaggle.com/learn)*

CNNs are called locally connected because at each dot product, they are only connected to the local region of the image, unlike in vanilla neural networks where each neuron is connected to every neuron in the layer before it.

The term **DOT PRODUCT** might confuse you if you don't have a strong background in advanced linear algebra. It turns out to be a very simple operation.

In simple terms, the dot product of two vectors is the sum of the element wise multiplication of the vectors. Given two matrices A and B, the dot product is represented as:

$$\sum Aij \times Bij$$

The operation is broken down into the following steps

Step 1: Pick each scalar element Aij from matrix A and Bij from matrix B, multiply them together.

Step 2: Sum the results from Step 1 to form a single scalar.

Example:

$$A = \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 1 \\ 6 & 0 \end{bmatrix}$$

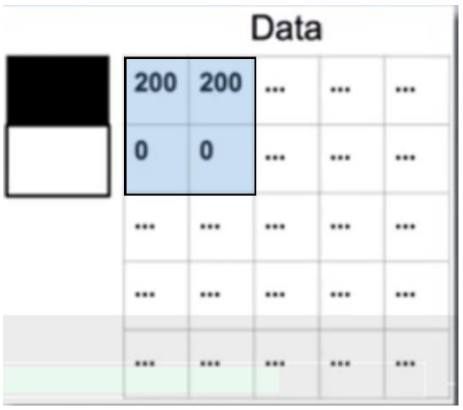
$$\boldsymbol{A.B \ or \ A^T B = 2 * 5 + 3 * 1 + 1 * 6 + 4 * 0 = 22}$$

As you can see, the dot product is a vector to scalar transformation, it's quite central to the operation of CNNs. However, we often add a bias term to the result of the dot product, just as we do in vanilla neural networks.

To illustrate how filters act as feature detectors, consider the example below.

1.5	1.5
-1.5	-1.5

This is a 2 x 2 filter, it is in fact a line detector, when you compute the dot product of this filter with a 2 x 2 region of an image, it would result in high values if a straight line is present.



Source: Kaggle Deep Learning Series (kaggle.com/learn)

In the image above, you can clearly see the marked 2 x 2 region in the image above contains a black horizontal straight line in the first row. Our eyes can perceive this pattern, but can our line detector initially described, detect this line?

Let's put it to test.

$$1.5 * 200 + 1.5 * 200 - 1.5 * 0 - 1.5 * 0 = 600$$

As you can see, the result is very high, which shows we have a distinct horizontal line present in this location of the image.

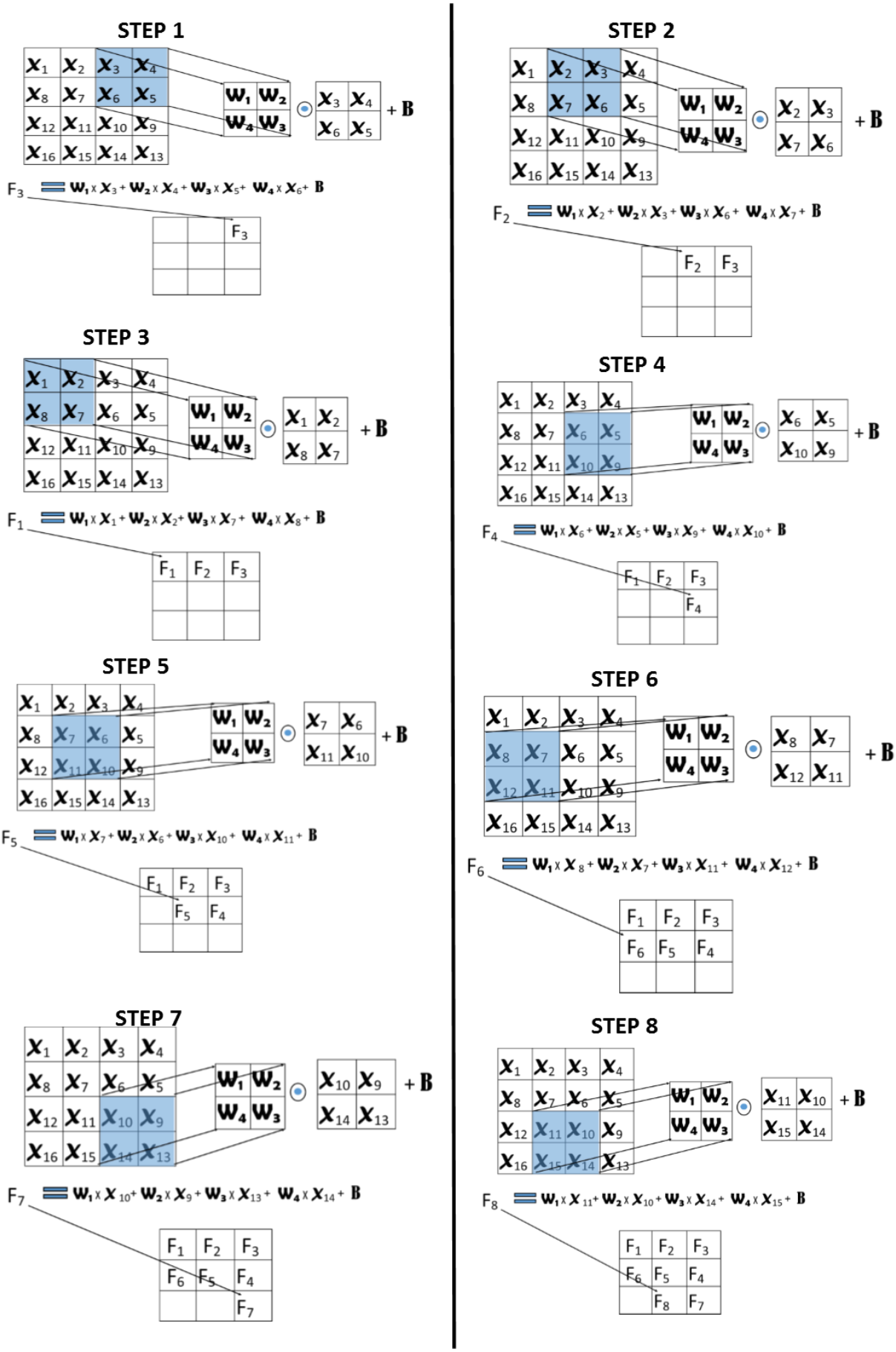
Here is another example, where we have a square pattern. Applying the line detector should result in a low value, Since a square is a different pattern from a line.

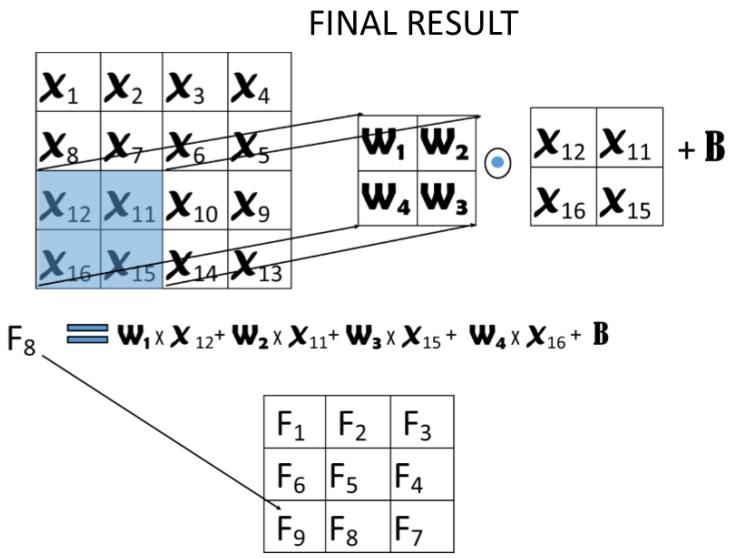
$$1.5 * 200 + 1.5 * 200 - 1.5 * 200 - 1.5 * 200 = 0$$

As you can see from the result of our dot product, the output is zero indicating that the pattern above which is a square is not a horizontal line.

CNNs are incredibly effective at detecting patterns, hence, Deep Computer Vision rests heavily on their shoulders.

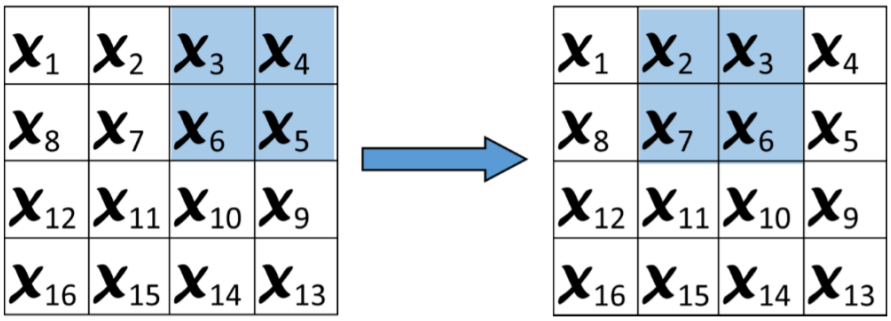
See the next page for an illustration of how CNNs work.


STEP 1
STEP 2
STEP 3
STEP 4
STEP 5
STEP 6
STEP 7
STEP 8

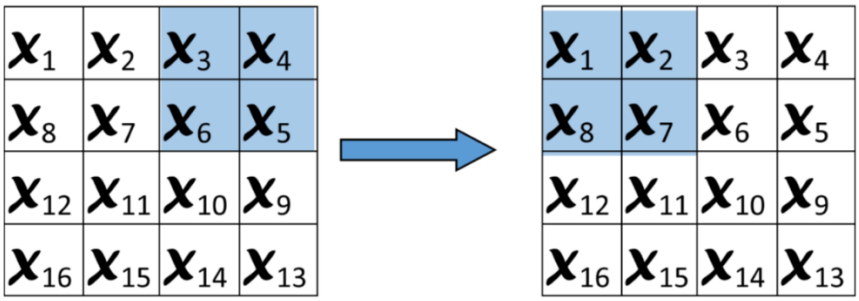


In the example above, the convolutions move by one pixel, however, this is not always the case. The number of pixels by which the convolutions move is called the stride and can be more than one.

Stride 1

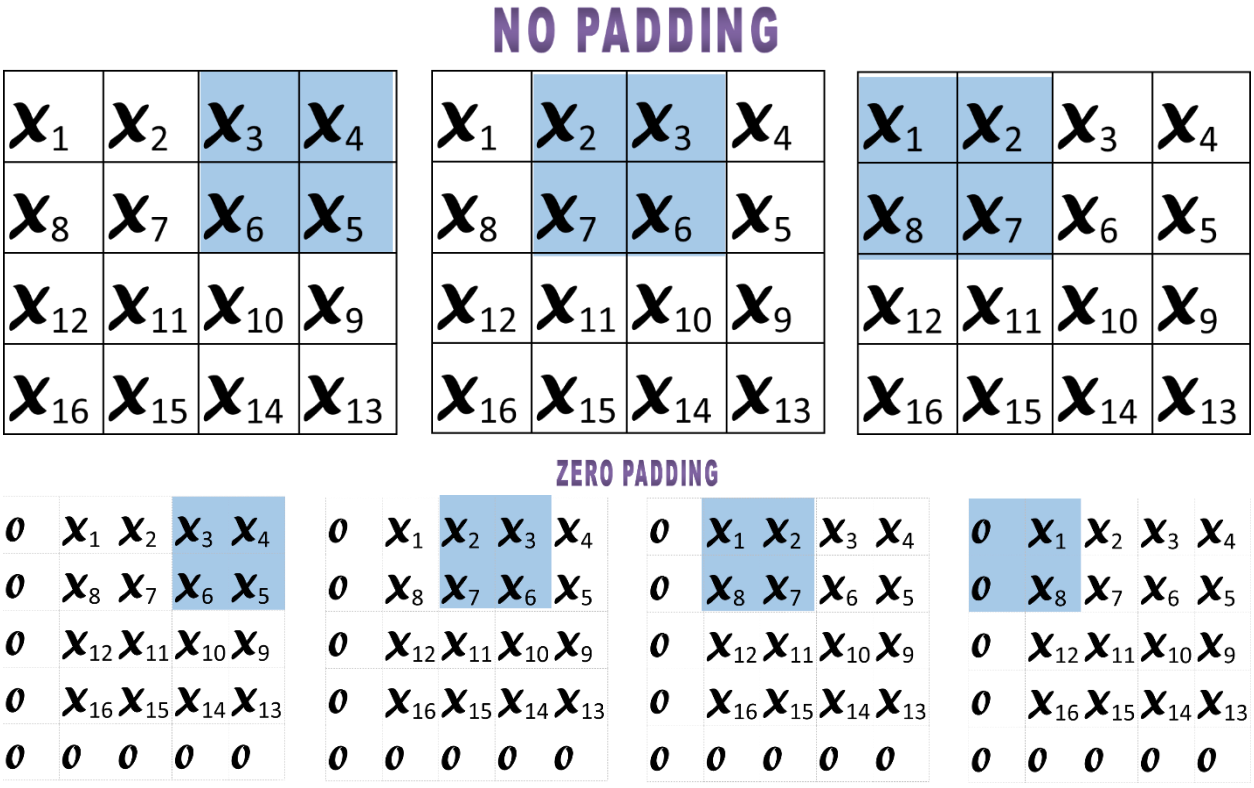


Stride 2



Usually, only strides of 1 are used, except on special occasions when we wish to significantly reduce the dimensions of our feature map.

Another factor you need to consider is the padding. Notice that in the above convolution, our 4 x 4 image was reduced to 3 x 3. This is not always desirable, hence, we can use padding to ensure the dimensions remain unaltered.



As you can see, by padding our image with zeros at both the width and height dimensions, we are able to perform the convolution at 4 different regions, hence our output would be a 4 x 4 feature map, maintaining the dimension of our image.

Generally, given a filter K, stride S, Image Width(same as height) W and padding P.

The formula for calculating the size output feature map is

$$size = \frac{W + 2P - K}{S} + 1$$

We can easily verify this from the above examples,

In our first example, we used a stride of 1 and No Padding, hence our output size should be

$$size = \frac{4 + 2 * 0 - 2}{1} + 1 = 3$$

The value 3 agrees with the shape of the output we got earlier, you can check again to verify.

Same example but using stride of 2 we get

$$size = \frac{4 + 2 * 0 - 2}{2} + 1 = 2$$

Strided convolutions down samples our image, in this case, a stride of 2 reduces the width and height by half.

Convolving an Image of width 4, using a stride of 1, kernel size of 3 and single Zero Padding, our output is

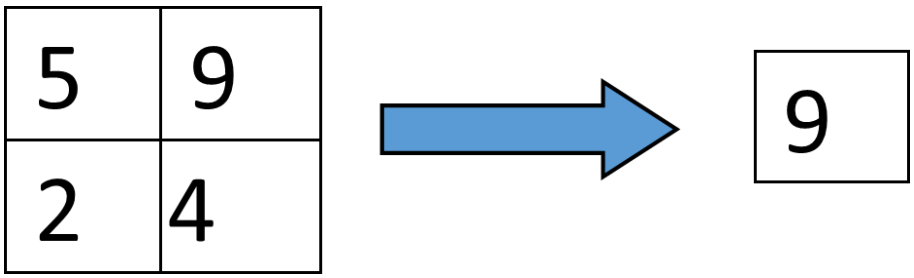
$$size = \frac{4 + 2 * 1 - 3}{1} + 1 = 4$$

In this case, the image dimensions are preserved.

Another common operation used alongside convolution is **Pooling**.

Pooling is a simple strategy that reduces the dimensions of our image.

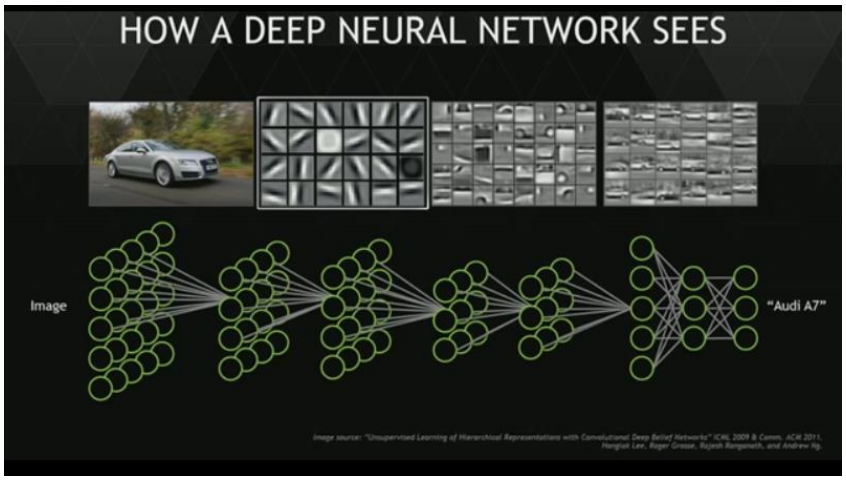
Pooling operates similarly to convolution except that instead of having filters and performing dot products, it simply takes a N x N region of an image, and computes the average value. A new feature map is produced, resulting in each N x N region becoming a single scalar value reducing the size by a factor of N. In practice, Max Pooling is mostly used. It differs from the average pooling because it takes the maximum pixel and discards the rest. This might seem unproductive but surprisingly, it often works better than average pooling.



Max Pooling simply picks the Maximum pixel and Discards the rest

Pooling helps to make CNNs invariant to the final presentation of the image, by picking the most important feature in a given pooling region.

As in vanilla neural networks, we often stack up many Convolution layers with a few Pooling layers in between. The early layers learn simple patterns like lines and edges, while later layers learn more abstract patterns like the eyes, nose and even the entire face.



First layers detect simple patterns while later layers detect more abstract patterns, Source: Kaggle Deep Learning Series (kaggle.com/learn)

A Practical Introduction

We have discussed at length, some of the fundamental principles concerning convolutional neural networks, now we shall create a CNN model using keras.

Before you proceed to run any of the codes in this section, please, ensure you have a system with a Nvidia GPU and that you installed the GPU version of tensorflow. If you don't have, don't stop. In fact, most machine learning entrants don't have, and even when you have, Desktop GPUs are not powerful enough to run some CNN architectures in a feasible period of time. Convolutions are very expensive operations that are way too slow on even the best CPUs. However, the parallel nature of GPUs makes them an excellent fit for CNNs.

Thanks to cloud computing, you can get powerful GPUs from Microsoft Azure, Amazon Web Services, Google Cloud Platform, Leader GPU and a few others.

The choice is up to you.

Compare their offerings and determine which is best for you. There are plenty of official documentation and online tutorials available online to set up GPU accelerated machine learning labs on each of these platforms. Once you are done with that, you can continue the rest of this tutorials.

If you can't afford to pay for Cloud GPUs, visit https://colab.research.google.com

Google provides free GPU accelerated VMs for research purposes.

Using convolutional neural networks in keras is very similar to what you already learned I the previous chapters. The only difference when using CNNs is the model definition.

Our model function would look like this

```
def MiniModel(input_shape):
    images = Input(input_shape)

    net =
Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(images)
    net =
Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(net)
    net = MaxPooling2D(pool_size=(2,2))(net)
    net =
Conv2D(filters=128,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(net)
    net = Conv2D(filters=128, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model
```

The code above is very similar to the previous model, here, instead of using a Dense Layer, we use the Conv2D layer. We specify 64 filters, indicating 64 feature detectors in this single layer. Next we specify the filter size to be 3 x 3, this is the most commonly used filter size. We used a stride of 1 and specify padding to be same, this ensures our image is padded with zeros to ensure the image dimensions stays the same. Finally, we specify activation to be RELU, this functions the same way we did in vanilla neural networks. In this case, it would threshold all negative activations to zero.

We add another layer with this exact configuration and then we add a maxpooling layer, with a pool size of 2 x 2, this reduces our image dimension by a factor of 2. In the case of MNIST, our 28 x 28 pixels becomes 14 x 14 after maxpooling.

It is conventional to double the number of filters after every pooling operation, so we add two new convolution layers but with 128 filters. Finally, the output layer has to be a fully connected softmax layer, and remember from our previous chapter that 2 Dimensional structures has to be flattened before they are passed into a dense layer, hence, we add the Flatten layer to do this and we pass the output to the softmax layer.

Our complete code is as below

```
#import needed classes
import keras
from keras.datasets import mnist
from keras.layers import Dense,Conv2D,MaxPooling2D,Flatten
from keras.models import Model,Input
from keras.optimizers import SGD
from keras.callbacks import LearningRateScheduler
from keras.callbacks import ModelCheckpoint
import os

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = mnist.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)

#Reshape from (28,28) to (28,28,1)
train_x = train_x.reshape(train_x.shape[0], 28, 28, 1)
test_x = test_x.reshape(test_x.shape[0], 28, 28, 1)



#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model


def MiniModel(input_shape):
    images = Input(input_shape)

    net =
Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(images)
    net =
Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(net)
    net = MaxPooling2D(pool_size=(2,2))(net)
    net =
Conv2D(filters=128,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(net)
    net = Conv2D(filters=128, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model

input_shape = (28,28,1)
model = MiniModel(input_shape)

#Print a Summary of the model

model.summary()

#Define the Learning rate schedule function

def lr_schedule(epoch):

    lr = 0.1

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
```

```
        lr = lr / 10
    elif epoch > 5:
        lr = lr / 5

    print("Learning Rate: ",lr)

    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
lr_scheduler = LearningRateScheduler(lr_schedule)

#Directory in which to create models
save_direc = os.path.join(os.getcwd(), 'mnistsavedmodels')

#Name of model files
model_name = 'mnistmodel.{epoch:03d}.h5'

#Create Directory if it doesn't exist
if not os.path.isdir(save_direc):
    os.makedirs(save_direc)

#Join the directory with the model file
modelpath = os.path.join(save_direc, model_name)

checkpoint = ModelCheckpoint(filepath=modelpath,
                             monitor='val_acc',
                             verbose=1,
                             save_best_only=True,
                             period=1)

#Specify the training components
model.compile(optimizer=SGD(lr_schedule(0)),loss="categorical_crossentropy",metrics=["accuracy"])

#Fit the model
model.fit(train_x,train_y,batch_size=32,epochs=20,shuffle=True,validation_split=0.1,verbose=1,cal
lbacks=[checkpoint,lr_scheduler])

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=32)

print("Accuracy: ",accuracy[1])
```

When you run this code, you would get an accuracy of over 99%. That's really impressive. Notice that we did not flatten the train and test data here. Convolutions takes advantage of the true structure of the images. The advantages of Convolutional Neural Networks are not obvious enough when classifying MNIST images, this is because the dataset is less challenging and linear classifiers could easily reach high accuracy. However, the big difference between CNNs and other methods becomes more obvious when working with more challenging real world datasets like CIFAR 10, CIFAR 100 and ImageNet.

Hence, this is the point where we say goodbye to MNIST. From here on, we would use CIFAR10 for all our experiments. We need not reiterate that the CIFAR 10 is even more computationally expensive than MNIST and you should ensure you have gotten a good GPU at this point.

There is a lot more to CNNs than just a stack of Convolutions and Pooling operations. As we shall see in the next chapter. A lot of additional components greatly improve the efficiency of CNNS.

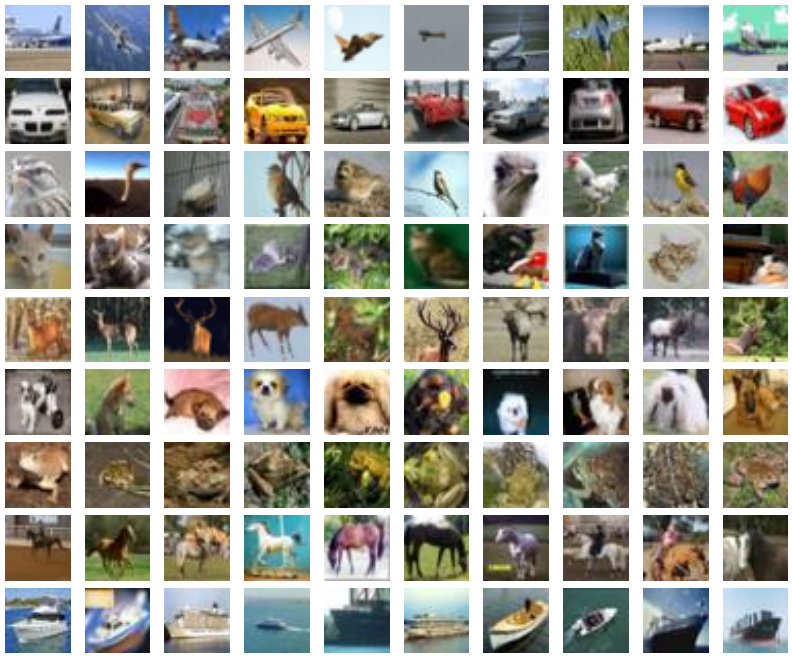
COMPONENTS OF CONVOLUTIONAL NEURAL NETWORKS

CNNs have evolved over the past few years, while the convolution operation itself has hardly changed, there have been a number of helper layers that has become crucial to every state-of-the-art architecture. The best architectures are usually characterized by high depth (number of layers), high width (number of filters), careful use of pooling, dropout layers for preventing overfitting and batch normalization. We shall explain each of these components in this chapter.

The CIFAR 10 Dataset

All experiments from now on would make use of the CIFAR 10 dataset. This dataset contains 60 000, 32 X 32 RGB images. The dataset is split into a train set of 50 000 images and a test set of 10 000 images. There are ten classes in total, namely, airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Below are some of the images from the dataset.

Source: https://www.cs.toronto.edu/~kriz/cifar.html



They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. For more information, visit https://www.cs.toronto.edu/~kriz/cifar.html

It is a standard benchmark for Image Recognition algorithms. Keras provides a handy function for downloading and loading the dataset. Note that the size is about 170mb so it may take a while to download, the first time you use it. This depends on your connection speed.

Dropout

This is a technique for reducing overfitting. It prevents a situation described as "Co-adaptation of Features". In a network with millions of parameters, sometimes some learned features would be too dependent on other features. This would make the model capture unnecessary relationships in the dataset. Background noise is often common, for example, a cat maybe present in a scene showing a car, we don't want the network to associate cars with cats. But this is exactly what happens when the network is too dependent on some specific features. It would perform well on the train set, but when these features are absent in a test image, it would fail badly. Dropout corrects this by randomly switching off some activations by setting them to zero. Hence, the network learns to explore other means to model the relationships. This technique has proven to be very effective in practice.

Below is our first example on cifar 10

```
#import needed classes
import keras
from keras.datasets import cifar10
from keras.layers import Dense,Conv2D,MaxPooling2D,Flatten,AveragePooling2D,Dropout
from keras.models import Model,Input
from keras.optimizers import Adam
from keras.callbacks import LearningRateScheduler
from keras.callbacks import ModelCheckpoint
import os

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = cifar10.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
```

```
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)



#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model


def MiniModel(input_shape):
    images = Input(input_shape)

    net =
Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(images)
    net =
Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(net)
    net = Conv2D(filters=64, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = MaxPooling2D(pool_size=(2,2))(net)
    net =
Conv2D(filters=128,kernel_size=[3,3],strides=[1,1],padding="same",activation="relu")(net)
    net = Conv2D(filters=128, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = Conv2D(filters=128, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = MaxPooling2D(pool_size=(2, 2))(net)
    net = Conv2D(filters=256, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = Conv2D(filters=256, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = Conv2D(filters=256, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = Dropout(0.25)(net)
    net = AveragePooling2D(pool_size=(8,8))(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model

input_shape = (32,32,3)
model = MiniModel(input_shape)

#Print a Summary of the model

model.summary()

#Define the Learning rate schedule function

def lr_schedule(epoch):

    lr = 0.001

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
        lr = lr / 10
    elif epoch > 5:
        lr = lr / 5

    print("Learning Rate: ",lr)

    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
lr_scheduler = LearningRateScheduler(lr_schedule)
```

```
#Directory in which to create models
save_direc = os.path.join(os.getcwd(), 'cifar10savedmodels')

#Name of model files
model_name = 'cifar10model.{epoch:03d}.h5'

#Create Directory if it doesn't exist
if not os.path.isdir(save_direc):
    os.makedirs(save_direc)

#Join the directory with the model file
modelpath = os.path.join(save_direc, model_name)

checkpoint = ModelCheckpoint(filepath=modelpath,
                             monitor='val_acc',
                             verbose=1,
                             save_best_only=True,
                             period=1)

#Specify the training components
model.compile(optimizer=Adam(lr_schedule(0)),loss="categorical_crossentropy",metrics=["accuracy"]
)

#Fit the model
model.fit(train_x,train_y,batch_size=128,epochs=20,shuffle=True,validation_split=0.1,verbose=1,ca
llbacks=[checkpoint,lr_scheduler])

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=128)

print("Accuracy: ",accuracy[1])
```

The above is only slightly different from our earlier example, the first major difference here is the input shape which is now (32,32,3) reflecting the three channels of our RGB image unlike the single channel of grayscale MNIST images.

Second important difference is the dropout layer we earlier explained, here we set the dropout ratio to be 0.25, this indicates that 25% of all activations should be disabled. Where to place dropouts is not a straightforward decision, and overuse of it can be rather detrimental.

Finally, we added an AveragePooling layer of size (8,8). This might confuse you. Note that at this stage, we have already had two maxpooling layers, hence our feature map at this point becomes 8 x 8. What we did here is called GlobalAveragePooling, at this point we have 256 feature maps, each being 8 x 8 in size, hence, we turned every single activation map into a single scalar by using a pool size equal to the size of the image at this point. This technique has proven very useful in nearly every single CNN architecture.

When you run this code, your accuracy would be about 79%. The state-of-the-art on CIFAR10 is a little over 96%, but that is achieved using data augmentation techniques we shall still discuss, much higher depth and training for about 200 –

300 epochs, we used just 20 epochs. If we allowed this simple setup to run for 300 epochs, we are likely to have much higher scores.

Notes about using Dropouts

Dropouts are only used during training, hence most deep learning libraries would switch of dropout layers during inference or evaluation. In cases where this is not automatically handled by the deep learning library, just remove the dropout layers from your code during inference.

Batch Normalization

After Convolutions, this is the single most important component of every CNN architecture. It is common to see modern CNN architectures not to have dropout layers, however, Batch Normalization is never absent. As you would see, we can use it to push up our accuracy using exactly the same setup that achieved just 79% accuracy.

Introduced by Ioffe and Szegedy in 2015. They help correct a phenomenon which the authors described as "Internal Covariate Shift". This refers to the problem of vanishing gradients as information flows through a network. Remember that CNNs train with stochastic gradient descent or any of its variants, as the layers get deeper, the gradients could get smaller and this terribly affects the performance of our networks. Batch Normalization corrects this by normalizing each batch of feature maps to have zero mean and unit variance with respect to statistics obtained per batch during training.

The formula for batch normalization is:

$$Y \leftarrow \gamma \frac{x_i - \mu}{\sqrt{\sigma^2 - \varepsilon}} + \beta$$

There is nothing difficult about the above, as usual we shall break the formula down into simple steps:
Step 1: Find the mean μ of the images in the batch
Step 2: Find the variance σ^2 using the mean
Step 3: Subtract the mean μ from each image

Step 4: Subtract a small constant ε from the variance σ^2 and find the square root of the result
Step 5: Divide the result of step 3 by the result of step 4
Step 6: Multiply each output of step 5 with a value γ and add β to the result

Keras already abstracts this for you but it is important to understand how the system works. Note that γ and β are parameters learned during training. Hence Batch Normalization adds a few extra parameters to our network.

Below is our earlier example but with two modifications, first we used RELU before the Convolution instead of after it, but more importantly, we added BatchNormalization before the RELU activation.

```
#import needed classes
import keras
from keras.datasets import cifar10
from keras.layers import
Dense,Conv2D,MaxPooling2D,Flatten,AveragePooling2D,Dropout,BatchNormalization,Activation
from keras.models import Model,Input
from keras.optimizers import Adam
from keras.callbacks import LearningRateScheduler
from keras.callbacks import ModelCheckpoint
import os

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = cifar10.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)



#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#Define the model


def MiniModel(input_shape):
    images = Input(input_shape)

    net = BatchNormalization()(images)
    net = Activation("relu")(net)
    net = Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same")(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=64,kernel_size=[3,3],strides=[1,1],padding="same")(net)
```

```
    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=64, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = MaxPooling2D(pool_size=(2,2))(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=128,kernel_size=[3,3],strides=[1,1],padding="same")(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=128, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=128, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)
    net = MaxPooling2D(pool_size=(2, 2))(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=256, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=256, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Conv2D(filters=256, kernel_size=[3, 3], strides=[1, 1], padding="same",
activation="relu")(net)

    net = Dropout(0.25)(net)
    net = AveragePooling2D(pool_size=(8,8))(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model

input_shape = (32,32,3)
model = MiniModel(input_shape)

#Print a Summary of the model

model.summary()

#Define the Learning rate schedule function

def lr_schedule(epoch):

    lr = 0.001

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
        lr = lr / 10
    elif epoch > 5:
        lr = lr / 5

    print("Learning Rate: ",lr)

    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
```

```
lr_scheduler = LearningRateScheduler(lr_schedule)

#Directory in which to create models
save_direc = os.path.join(os.getcwd(), 'cifar10savedmodels')

#Name of model files
model_name = 'cifar10model.{epoch:03d}.h5'

#Create Directory if it doesn't exist
if not os.path.isdir(save_direc):
    os.makedirs(save_direc)

#Join the directory with the model file
modelpath = os.path.join(save_direc, model_name)

checkpoint = ModelCheckpoint(filepath=modelpath,
                             monitor='val_acc',
                             verbose=1,
                             save_best_only=True,
                             period=1)

#Specify the training components
model.compile(optimizer=Adam(lr_schedule(0)),loss="categorical_crossentropy",metrics=["accuracy"]
)

#Fit the model
model.fit(train_x,train_y,batch_size=128,epochs=20,shuffle=True,validation_split=0.1,verbose=1,ca
llbacks=[checkpoint,lr_scheduler])

#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=128)

print("Accuracy: ",accuracy[1])
```

As you can see, first we added Batch Normalization, next instead of putting the activation inside the convolution operation, we added it right before it. This is called Pre-Activation. Other books might use Conv-BatchNorm-RELU setup, but this approach conforms with the latest discoveries.

This setup achieves 85.77% accuracy using just 20 epochs, that's quite better and if we had used about 200 epochs, quite likely we would be over 90%.

Batch Normalization makes the big difference here.

The code above is way too long and not neat, we could make it modular and cleaner by creating a function that returns each unit.

Create a function like this.

```
#define a common unit
def Unit(x,filters):
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1],
padding="same")(out)

    return out
```

This function takes in the input feature map and the number of channels we desire and it then it passes the input through Batch Normalization, RELU and Convolution. It then returns the final output.

Now our model would look like this:

```
def MiniModel(input_shape):
    images = Input(input_shape)

    net = Unit(images,64)
    net = Unit(net,64)
    net = Unit(net,64)
    net = MaxPooling2D(pool_size=(2,2))(net)

    net = Unit(net,128)
    net = Unit(net,128)
    net = Unit(net,128)
    net = MaxPooling2D(pool_size=(2, 2))(net)

    net = Unit(net,256)
    net = Unit(net,256)
    net = Unit(net,256)

    net = Dropout(0.25)(net)
    net = AveragePooling2D(pool_size=(8,8))(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model
```

This is way so beautiful. This is exactly equivalent of our previous example and it would yield the same accuracy.

Data Augmentation

Take an image, flip it horizontally or vertically or shift the pixels a little, your eyes can still perceive what it is. But CNNs could fail at this. To resolve the problem, we

use data augmentation, which slightly improves the performance of our network. The key is to randomly apply flipping and shifting to our images, so our network can learn different representations of our image.

Different augmentation techniques exist, some commonly used ones are horizontal flipping, shifting, rotation, scaling and whitening.

A key data augmentation technique to is to subtract the mean image from every image and sometimes divide by the standard deviation

Here is our previous example, re-written to use data augmentation

```
#import needed classes
import keras
from keras.datasets import cifar10
from keras.layers import
Dense,Conv2D,MaxPooling2D,Flatten,AveragePooling2D,Dropout,BatchNormalization,Activati
on
from keras.models import Model,Input
from keras.optimizers import Adam
from keras.callbacks import LearningRateScheduler
from keras.callbacks import ModelCheckpoint
from math import ceil
import os
from keras.preprocessing.image import ImageDataGenerator

#load the mnist dataset
(train_x, train_y) , (test_x, test_y) = cifar10.load_data()

#normalize the data
train_x = train_x.astype('float32') / 255
test_x = test_x.astype('float32') / 255

#Subtract the mean image from both train and test set
train_x = train_x - train_x.mean()
test_x = test_x - test_x.mean()

#Divide by the standard deviation
train_x = train_x / train_x.std(axis=0)
test_x = test_x / test_x.std(axis=0)

#print the shapes of the data arrays
print("Train Images: ",train_x.shape)
print("Train Labels: ",train_y.shape)
print("Test Images: ",test_x.shape)
print("Test Labels: ",test_y.shape)
```

```
#Encode the labels to vectors
train_y = keras.utils.to_categorical(train_y,10)
test_y = keras.utils.to_categorical(test_y,10)

#define a common unit
def Unit(x,filters):
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1],
padding="same")(out)

    return out

#Define the model


def MiniModel(input_shape):
    images = Input(input_shape)

    net = Unit(images,64)
    net = Unit(net,64)
    net = Unit(net,64)
    net = MaxPooling2D(pool_size=(2,2))(net)

    net = Unit(net,128)
    net = Unit(net,128)
    net = Unit(net,128)
    net = MaxPooling2D(pool_size=(2, 2))(net)

    net = Unit(net,256)
    net = Unit(net,256)
    net = Unit(net,256)

    net = Dropout(0.25)(net)
    net = AveragePooling2D(pool_size=(8,8))(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model

input_shape = (32,32,3)
model = MiniModel(input_shape)

#Print a Summary of the model

model.summary()

#Define the Learning rate schedule function

def lr_schedule(epoch):

    lr = 0.001

    if epoch > 15:
        lr = lr / 100
    elif epoch > 10:
        lr = lr / 10
    elif epoch > 5:
        lr = lr / 5

    print("Learning Rate: ",lr)
```

```
    return lr

#Pass the scheduler function to the Learning Rate Scheduler class
lr_scheduler = LearningRateScheduler(lr_schedule)

#Directory in which to create models
save_direc = os.path.join(os.getcwd(), 'cifar10savedmodels')

#Name of model files
model_name = 'cifar10model.{epoch:03d}.h5'

#Create Directory if it doesn't exist
if not os.path.isdir(save_direc):
    os.makedirs(save_direc)

#Join the directory with the model file
modelpath = os.path.join(save_direc, model_name)

checkpoint = ModelCheckpoint(filepath=modelpath,
                             monitor='val_acc',
                             verbose=1,
                             save_best_only=True,
                             period=1)

#Specify the training components
model.compile(optimizer=Adam(lr_schedule(0)),loss="categorical_crossentropy",metrics=[
"accuracy"])


datagen = ImageDataGenerator(rotation_range=10,
                             width_shift_range=5. / 32,
                             height_shift_range=5. / 32,
                             horizontal_flip=True)

# Compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(train_x)

epochs = 20
steps_per_epoch = ceil(50000/128)

# Fit the model on the batches generated by datagen.flow().
model.fit_generator(datagen.flow(train_x, train_y, batch_size=128),
                    validation_data=[test_x,test_y],
                    epochs=epochs,steps_per_epoch=steps_per_epoch, verbose=1,
workers=4,
                    callbacks=[ checkpoint,lr_scheduler])


#Evaluate the accuracy of the test dataset
accuracy = model.evaluate(x=test_x,y=test_y,batch_size=128)
```

Core to the data augmentation is the Image Generator class. The major differences here are the fit function, the generator and our mean and std preprocessing.

```
#Subtract the mean image from both train and test set
train_x = train_x - train_x.mean()
test_x = test_x - test_x.mean()

#Divide by the standard deviation
train_x = train_x / train_x.std(axis=0)
test_x = test_x / test_x.std(axis=0)
```

This part is very simple, we first subtract the mean image from the training and test set, and we divided them by the standard deviation of the images.

The generator performs all the other data augmentation, we just need to tell it the transformations we desire.

```
datagen = ImageDataGenerator(rotation_range=10,
                             width_shift_range=5. / 32,
                             height_shift_range=5. / 32,
                             horizontal_flip=True)
```

We randomly rotate images by 10 degrees, shift the height and width by a factor of 5/32 and last, horizontal flipping. This is all the transformations we desire here. More options are available and you can try them yourself.

Unlike the mean and std preprocessing, we do not apply the other transformations to the test set, only to the training set.

Next we need to fit the generator our training images.

```
datagen.fit(train_x)
```

The big part here is the fit_generator function

```
# Fit the model on the batches generated by datagen.flow().
model.fit_generator(datagen.flow(train_x, train_y, batch_size=128),
                    validation_data=[test_x,test_y],
                    epochs=epochs,steps_per_epoch=steps_per_epoch, verbose=1,
workers=4,
                    callbacks=[lr_scheduler])
```

We replace the standard fit function with this specialized version.

Then we pass in the images and labels using the data generator. All other things remain the same except for the `steps_per_epoch` which explicitly tells it how many iterations is required to go over all the batches.

Finally, we specify the test dataset as validation data since this function doesn't accept a validation split ratio.

The above setup achieves an accuracy of 87.45% which is considerably better than our earlier 85.77% accuracy using batch normalization without any data augmentation. At just 20 epochs, this is a very impressive setup, you should try running it for about 200 epochs and see the final result. However, you should adjust your learning rate schedules given the higher number of epochs. The schedule function below might be a good candidate.

```
def lr_schedule(epoch):

    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:
        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1

    print('Learning rate: ', lr)

    return lr
```

You are welcomed to modify other parameters as appropriate.

RECTIFIERS

Rectifiers are variants of the Recitified Linear Unit (RELU). Recall that relu takes the form

$$\sigma = f(Z) = \begin{cases} Z, & Z > 0 \\ 0, & Z \leq 0 \end{cases}$$

RELU in this form is the most widely used activation function at present, however, a number of variants have found great use in more specialized applications such as Generative Adversarial Networks, occasionally, some of these variants have been demonstrated to yield greater accuracy than the standard relu on image classification tasks. The major variants of RELU include LeakyRelu, Parametric Relu, Exponential Linear Units and Thresholded Relu. The ultimate motivation for this is preventing vanishing gradients due to zero activations.

Leaky Relu

Leaky Relu improves the stability of relu by introducing an alpha hyper parameter that is used to multiply any negative gradient. The formula is:

$$f(Z) = \begin{cases} Z, & Z > 0 \\ \alpha Z, & Z \leq 0 \end{cases}$$

This can also be simply expressed as

$$f(Z) = Z \; if \; Z > 0 \; else \; \alpha Z$$

This allows for a small gradient when the input is less than zero. Typical setting of α is 0.3.

Parametric Relu

Simply shortened to PRELU, it is the same as Leaky Relu with the only difference being that α is not set explicitly, it is learned as a parameter during training. This was first introduced by Kaiming et al 2016. It results in a slight increase in parameters but with sufficient training, but reduces the number of hyper parameters we need to set.

Exponential Linear Units

Shortened as ELU, it is regarded by many as the effective replacement for RELU. It takes the form.

$$f(Z) = \begin{cases} Z, & Z > 0 \\ \alpha(e^z - 1), & Z \leq 0 \end{cases}$$

ELU was demonstrated by it's authors to surpass the performance of the other variants of relu. It involves the α hyper parameter just like leaky relu, the major difference is that we multiply α with the result of subtracting 1 from the exponent of the input z.

Thresholded Relu

This has a hyper parameter θ that allows us to set the minimum we want our activation to be, any value below θ is set to zero. It takes the form.

$$f(Z) = \begin{cases} Z, & Z > \theta \\ 0, & Z \leq \theta \end{cases}$$

Using Other Activations

To use activations besides the non-parametric ones like RELU, Sigmoid and Tanh

You should import them from the **keras.layers** package.

Example:

```
from keras.layers import LeakyReLU
```

Next, you should place it just after batch normalization.

```
out = BatchNormalization()(x)
out = LeakyReLU(alpha=0.25)(out)
out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)
```

ARCHITECTURE CASE STUDIES

We have explained to some details the theories and practical approaches to Deep Computer Vision using Convolutional Neural Networks. However, you must be aware that new CNN architectures and methods appear every year. In fact, it is hard to tell what the state-of-the-art architecture is at any time. New ground breaking papers are published so often in the past two years. The only way to truly stay thoroughly informed is to read the latest papers published in the field of computer vision. You should regularly visit the computer vision and pattern recognition archive on arxiv.org, the home for all scientific papers.

Note that the use of CNNs goes far beyond classification, other uses in the computer vision category include object detection, image segmentation and generative models. Also their use has been extended to tasks usually tackled by Recurrent Neural Networks. Notably, a recent paper from Facebook AI Research proved that convolutional neural networks could beat recurrent neural networks in the task of language translation. They used a fully convolutional architecture to beat the best RNN methods both in terms of speed and accuracy. They are also used in Deep Reinforcement learning, in the famous AlphaGO Zero, they were used to capture the current state of the board game. Their uses are so many and the list is gradually expanding to include sentiment analysis and sequence to sequence learning.

You can look through the references section of this book for other materials that could help you study further.

Here we shall explain key aspects of some of the most popular and effective CNN architectures. We shall begin our review from 2012, the miracle year of deep learning. The first CNN architecture was the LeNet by Yann LeCun et al. The entire artificial intelligence community owe him and the other authors, great thanks. Everything we have done has been an evolution of this great work.

AlexNet

AlexNet was published by Alex Krizhevsky, Illya Suskever and Geofrrey Hinton, in the year 2012. This single publication is highly similar in effect to Albert Einstein's five breakthrough papers in 1905. It set off the whole Deep Learning hype around today.

It was the first time a Convolutional Neural Network would significantly outperform other methods on a large dataset(ImageNet 2012) by a large margin.

AlexNet was composed of five convolutional layers followed by three fully connected (Dense) layers. By today's standards, this was too simple, but back then it was the best. Their main contributions was not actually the simple architecture but the training process and the components of the network.
First they used RELU activation instead of sigmoid and tanh. You must know that the latter two were the commonly used activation functions in those days.
They also used dropouts, which significantly improved their performance.

Their most important contribution was the training process. First they used data augmentation to artificially increase the training dataset. Last but the greatest contribution was the CUDA-Covnet code which was an incredibly efficient implementation of the convolution operation. It effectively parallelized the training process across two GPUs. In those days, there were no Deep Learning libraries, hence, you really have to respect the incredible pioneering work they did.

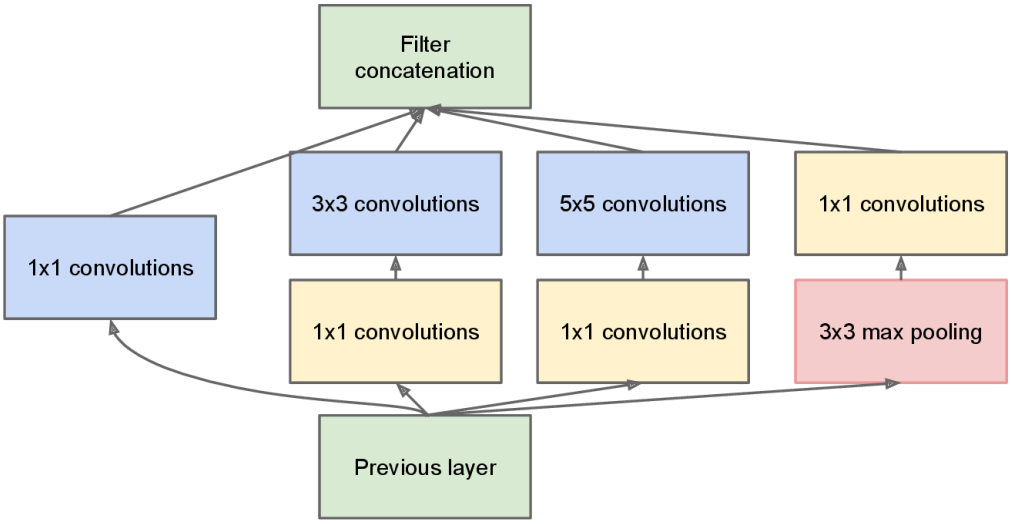
VGG

Karen Simonyan and Andrew Zisserman published VGG net in 2014. It won second place in the classification task on ImageNet 2014. This work really proved the importance of depth in CNNs. The two most popular variants of VGG are 16 layers VGG and 19 layers VGG. It used 3 × 3 filters for all convolutions. MaxPooling was used at different points. This proved to be highly effective. The network had three fully connected layers at the end.
VGG is a very large network with about 144 million parameters , but the simplicity of the architecture made it very popular and it is still greatly used till today.

Inception

Inception was an excellent work from Christian Szegedy et al. It won the first place on the ImageNet 2014 classification task. First it was a 22-layers network, the deepest architecture as of then, more than this, the architecture was composed of Inception modules. Each inception module was a mini-network itself, made up of convolutions with different filters, this enabled multi-scale

processing of the image. It was an incredibly efficient architecture. And it became by far, the most popular CNN architecture after AlexNet.



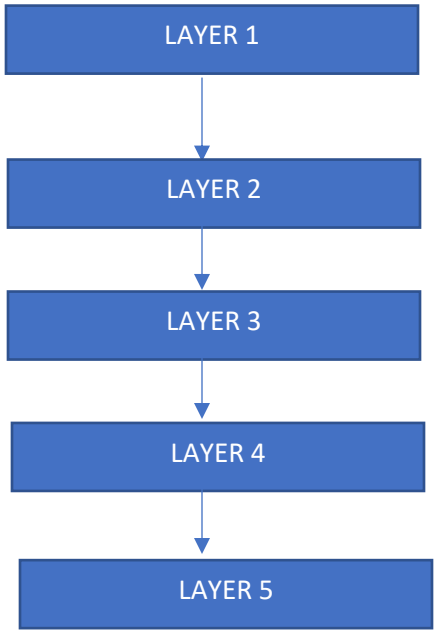
An Inception Module

Residual Networks

Convolutional Neural Networks works so well when many layers are stacked up together. This is because, with many layers, feature detectors are more fine grained. A particular pattern detected at layer 1 can be subdivided into newer patterns at layer 2. The ultimate secret to the success of neural networks for artificial intelligence is the depth of the layers, hence the term “Deep Learning”.

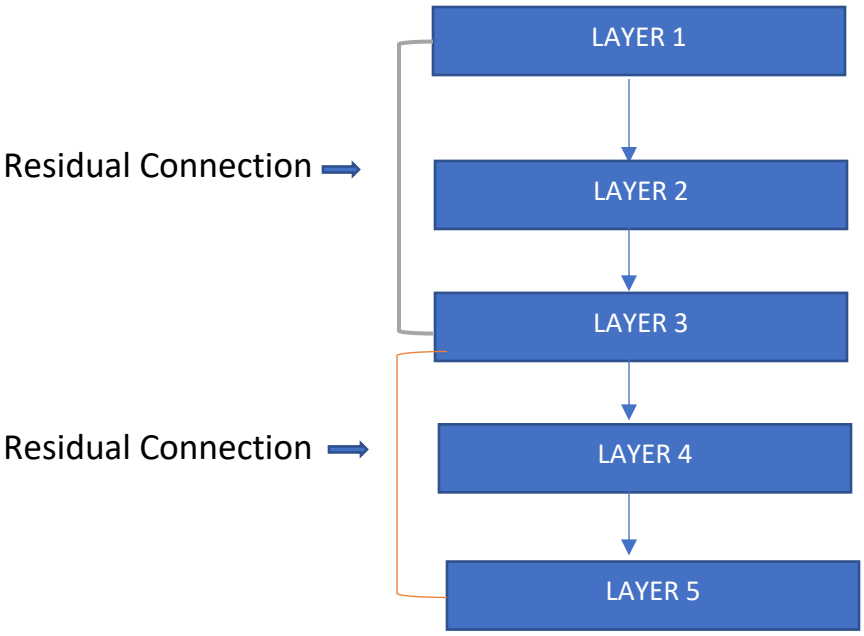
However, it turns out that depth comes with its own challenges. As layers gets deeper, information about the original inputs diminish. Gradients also tend to vanish as layers gets too deep. Due to these factors, beyond certain number of layers, accuracy would rather reduce. This makes it hard to optimize deep neural networks. While it is not a problem unique to image recognition alone, it was first studied in the image classification setting by Kaiming He et al in 2015. Their paper titled “A Residual Framework for Image Classification” was a ground-breaking work in the field of artificial intelligence, as it not only led to significant breakthrough in image recognition, but the ideas has since been successfully applied to virtually all other domains of deep learning, including speech recognition, natural language processing and reinforcement learning. The network they developed was called ResNet.

Through series of experiments, the authors of resnet realized that simply stacking more and more layers resulted in lower accuracy, they also established that this was not due to over-fitting, hence regularization methods like dropout could not alleviate the problem. To solve this problem, they formulated the idea of residual connections, which is basically, a connection between previous layers and later layers. To simplify this, imagine you have a network with 5 layers, the connections would be as below.

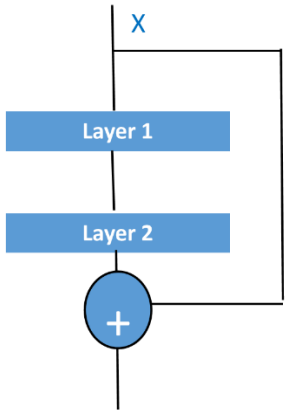


This is a regular network with connections between every layer L_t and L_{t+1}.

A residual network is connected as below.



As you can see above, residual networks include connections between layers L_t and L_{t+2}. The residual connection is an add operation. This entails, given an input x, the standard connection is $y = f(x)$, however, in resnet, $y = f(x) + x$



Structure of a Resnet Module

For a perfect understanding of this concept, consider the resnet module below.

```
def ResnetModule(x,filters):
    res = x
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1],
padding="same")(out)

    out = BatchNormalization()(out)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1],
padding="same")(out)

    out = keras.layers.add([res,out])

    return out
```

As you can see above, we store a copy of the input for reference, next we pass the input into a batch normalization layer, followed by relu activation and finally convolution. In a standard module, we would return the output of the convolution, however, in this resnet module, we add the output of the convolution with the residual, which is a copy of the input to this module.

This is done on the line

```
out = keras.layers.add([res,out])
```

A residual network is formed by stacking many of this layer together.

Due to the use of the residual connections, deep learning scientists have successfully trained convolutional neural networks as deep as 2000 layers with increasing accuracy.

More recent research has also beyond making ultra-deep networks highly accurate, residual demonstrated that networks can improve the accuracy of any convolutional neural network whether shallow or deep. Hence, even when your network is made up of only few layers, it is still highly beneficial to use residual connections.

Note that when adding two outputs together, their dimensions must be exactly the same, the width, height and number of channels must be same. This becomes an issue when we perform pooling, since the width and height are reduced and it

is a general practice to double the number of layers anytime we pool, hence, the number of channels would also differ. To account for this, all we need to do is to pass the residual through a 1 x 1 convolution whose channels is equal to the number of channels of the output we need to add it to, we shall also set strides greater than 1 to ensure the width and height are same. For example, if you perform pooling of 2 x 2 and increase channels to 64, then you should pass the residual through a 1 x 1 convolution with 64 channels and a stride of 2. This simple technique would guarantee equality of the dimensions.

Below is a modified version of the resnet module that takes pooling into consideration.

```
def ResnetModule(x,filters,pool=False):
    res = x
    if pool:
        x = MaxPooling2D(pool_size=(2, 2))(x)
        res = Conv2D(filters=filters,kernel_size=[1,1],strides=(2,2),padding="same")(res)
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

    out = BatchNormalization()(out)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

    out = keras.layers.add([res,out])

    return out
```

We shall now build a simple residual network with this.

```
def MiniModel(input_shape):
    images = Input(input_shape)
    net = Conv2D(filters=32, kernel_size=[3, 3], strides=[1, 1],
padding="same")(images)
    net = ResnetModule(net,32)
    net = ResnetModule(net,32)
    net = ResnetModule(net,32)

    net = ResnetModule(net,64,pool=True)
    net = ResnetModule(net,64)
    net = ResnetModule(net,64)

    net = ResnetModule(net,128,pool=True)
    net = ResnetModule(net,128)
    net = ResnetModule(net,128)

    net = ResnetModule(net, 256,pool=True)
    net = ResnetModule(net, 256)
    net = ResnetModule(net, 256)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
```

```
    net = Dropout(0.25)(net)

    net = AveragePooling2D(pool_size=(4,4))(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model
```

The network above first consist of a standard convolution, we did not include relu or batch normalization after this since the next layer is a resnet module which begins with batch normalization and activation. We stack up 12 resnet modules, with each module consisting of two layers of bn-relu-conv, finally, since the output of the final resnet module is the addition of two convolution outputs, we apply batch normalization and relu.

Using our previous examples as guide, train and evaluate a network with this model or a modification as you wish.

Evolutions of Resnet

Densenet by researchers from Facebook AI Research, Cornell University and Tsinghua University is slightly different from the original Resnet, it uses filter concatenation instead of addition as used in Resnet. In densenet, every layer is connected to all layers after it. You would appreciate what these entails when you

look at the diagram below.

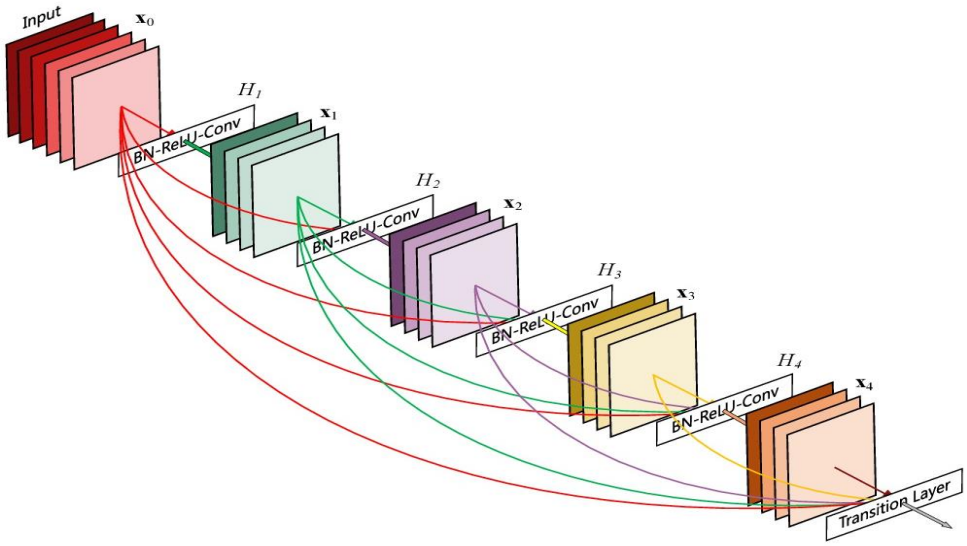


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

The latest member of the Resnet family of architectures is ResNext by Saining Xie, et al. The major difference here is, instead of higher depth and greater width, the authors added a new dimension which was based on the idea of grouped convolutions. In ResNext, rather than have a single convolution layer with 128 filters, we instead construct 32 separate convolutions, each with 4 filters, we then concatenate the outputs from all the 4-filter convolutions, to form a layer with 128 filters.

SE NET

Squeeze and Excitation networks won the first place in ImageNet 2017. That's pretty big deal! Interestingly, it also brought back into CNNs, the long abandoned sigmoid function. Although sigmoid has always been used in Recurrent Neural Networks and sometimes in Generative CNN models, but in standard object recognition, detection and image segmentation, they have been abandoned. But not back to replace relu, just a complement. The major highlight of SE Net is a SE module that can be placed between any successive layers in a network. In Resnet for example, the SE module can be placed after every residual block.

Below is the keras code for the SE module

```
def SEModule(x,filters,compress_ratio=16):

    compression = int(filters/compress_ratio)

    se = GlobalAveragePooling2D()(x)
    se = Dense(compression)(se)
    se = Activation("relu")(se)
    se = Dense(channels)(se)
    se = Activation("sigmoid")(se)

    out = multiply([x, se])

    return out
```

Filters in the above represents the number of channels in our feature map. The compression ratio defines how much we want to squeeze our feature map.

Next, we apply GlobalAveragePooling and follow this with a Dense layer with number of units equal to the filters divided by the compression ratio, RELU activation is applied to the result and then a Dense layer with number of units equal to the number of channels of the input. Finally, our old friend, the sigmoid activation is applied and the input X is multiplied with the output of the sigmoid function.

When applied to any existing architecture, this simple setup has been shown to improve accuracy.

CONVOLUTION ARITHMETIC

Convolutional Neural Networks works greatly when the depth of the network is sufficiently high and the width(channels) per layer is sufficiently wide. Thanks to residual networks, we can increase width and height to obtain better accuracies. However, there are limits to the computational capacity of even the most advanced GPUs. Consequently, training CNNs on very large datasets such as ImageNet can take months, at best, using multiple high performance GPUs, training usually takes weeks. Ideally, we want training to be much faster. Of greater concern is inference speed. A model trained on high performance systems might be deployed in more resource constrained systems, including Mobile Phones, FPGAs and other embedded systems. In such cases, inference is done on CPU or lower grade GPUs. This makes the computational cost of a network, an important factor to consider when designing new architectures or choosing any of the existing architectures for a specific task.

Computational cost is best measured in terms of how long it takes to perform inference on a single image on a specific device, however, this is impossible to calculate without empirical evaluation. In practice, we can easily calculate the cost of our model in terms floating point operations called flops.

The flops defined in terms of multiply - adds gives us an approximation of how fast our network is, the lower the flops, the better. However, bear in mind that other factors such as network structure could cause a network with higher flops to run faster than a network with lower flops. For example, due to the parallel nature of GPUs, a wider network with more flops than a very deep but thin network, might run significantly faster. This was the motivation for WideResnet(cite).

Having said that, flops are still the best way to estimate the speed of neural networks. In this chapter, we shall explain exactly how to calculate the flops of any given convolutional neural network as well as how to calculate the number of parameters in a CNN.

Dense Layers

Dense layers are fully connected, this entails that for a dense layer with 10 units, if the input is of size 50, then, each of the 10 units would receive the entire 50 inputs.

In each of the 10 units, 50 weights would be assigned for each of the incoming 50 inputs, with a bias of 1 added.

This would result in 50 multiplications happening between the weights and the inputs with 49 additions to sum up the result of the 50 multiplications, plus 1 bias added, makes 50 adds.

Given 10 such units, the total flops would be (50 adds + 50 multiplies) * 10 units = 1000 flops.

When calculating flops, we usually treat the multiplication-add as 1 unit called multiply-adds. Hence, instead of referring to 50 + 50 operations, we simply refer to 50 multiply-adds, this gives us 500 MACCS.

To make things simpler, consider the example below.

Input X of size 100 is passed into a dense layer with 64 units.

The computation in each unit would go as:

$$y = \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3 + \cdots + \theta_{100} X_{100} + \beta$$

As you can see here, there are a total of 100 multiplications and 100 additions. So we count this as 100 multiply-adds.

With 64 units, 64 such operations would occur, so our total multiply-adds would be **64 * 100 = 6400.**

Generally, given M inputs into a fully connected layer with N units(neurons), the total multiply-adds is **M * N.**

Convolution Layers

Remember how convolutions work, we have a filter matrix called a kernel of dimension $K * K * C_{in}$ (note that rectangular filters of dimension $K_1 * K_2 * C_{in}$ are also used, but for simplicity, we only consider square $K * K * C_{in}$ kernels here) The C_{in} is the input channels. For a 224 x 224 x 3 image, $C_{in} = 3$.

We then compute a dot product of the $K * K * C_{in}$ kernel with a $K * K * C_{in}$ region of the Image, this single operation has a cost of $K * K * C_{in}$ multiply-adds.
Next we move the convolution over W regions along the width as well as H regions along the Height, note that W and H refer to output Width and Height respectively.
The computational cost becomes $K * K * C_{in} * W * H$ multiply-adds.
Finally, we have to do this for C_{out} channels.
The final equation becomes
compute cost = $K * K * C_{in} * W * H * C_{out}$
Note that if the kernel is a $K_1 * K_2 * C_{in}$ rectangular matrix,
compute cost = $K_1 * K_2 * C_{in} * W * H * C_{out}$ MACS

Example :
Consider a ***224 * 224 * 3*** image formatted as ***Width * Height * Channels***
Let's set up a CNN with just ***2*** convolution layers, setting the Kernel size to be ***3 * 3*** with no padding, a stride of **1** and ***64*** output channels for the first layer, and the second layer would have kernel size of ***4 * 4***,padding of ***1***,stride of ***2*** and ***64*** output channels

For simplicity, we assume ***Width*** = ***Height*** =***size***, hence, we shall simply use **size** to represent both **Width** and **Height**

For the first layer
According to the formula for the output size of a convolution

out_size = (**size** + **2*padding** – **kernel_size) /stride + 1**
out_size = ***(224 + 2*0 - 3)/1 + 1 = 222***

Cost = $K * K * C_{in} * out\_size * out\_size * C_{out}$

Cost = ***3 *3 * 3 * 222 * 222 * 6*** = **28,387,584**

That's for the first layer, since the output dimension of the first layer is ***222 * 222 * 64***,this would be the input to the next layer

Calculating the output of the second layer,

Since kernel = 4,padding = 1,strides=2 and channels = 64 for the second layer we have

out_size = (222 +2*1 - 4)/2 + 1 = 111

Cost = ***4 * 4 * 64 * 111 * 111 * 64*** = **807,469,056**

Summing up the flops for the two layers, we have a total of **835,856,640** multiply-adds, which is equivalent to **835,856,640 * 2 flops =**

1 671 713 280 flops!

This is **1.67** giga flops.

Usually, standard networks are in giga flops while the most efficient networks have millions of flops

CALCULATING PARAMETERS

When you use the keras summary() function, you get a detailed view of the parameters of the layers in your network, however, it is important that you should be able to calculate this yourself. We shall go over this by layer type.

Dense Layer

A fully connected layer has ***M*** units , each unit has ***N + 1*** parameters, where ***N*** is the number of incoming features and ***1*** represents the extra count introduced by the bias parameter. In summary, we have in each neuron, ***N*** weights and ***1*** bias, To get the total number of parameters, you have to multiply ***N + 1*** by ***M*** units.

params = M(N + 1)

Example:

For a Dense layer with ***256*** units receiving an input vector of size ***100***,

params = 256(100 + 1) = 25856

Convolution Layer

Calculating parameters for convolutions is very straight forward, remember that a kernel is of size ***K * K * C_{in}***, where ***C_{in}*** is the number of input channels.

For each output channel, all convolutions would use exactly the same kernel, this is known as parameter sharing. Basically, for each output channel, the

convolution is searching for a single feature across multiple regions of the image. Hence, the total number of parameters is equal to the number of parameters of a single kernel multiplied by the number of output channels.

Params = $\boldsymbol{K * K * C_{in} * C_{out}}$

Example:

A convolution layer that receives an input feature map of 224 * 224 * 128 formatted as Width x Height x Channels, given a kernel of size 3 x 3 and 256 output channels. The number of parameters is calculated as

3 * 3 * 128 * 256 = 294 912 params

Efficiency Techniques

In the previous chapter, you learnt how to calculate the computational cost of a convolutional neural network. Given the flops, we can estimate the time required to perform both training and inference with a network. Our ultimate desire is to achieve very high accuracy with the lowest possible inference speed. However, most of the techniques employed to increase accuracy, including increasing depth, width and kernel size, does increase the flops and consequently increase the inference time. Hence, we have to find a balance of speed and accuracy that would satisfy our needs. This tradeoff becomes very important in the context of embedded applications such as mobile applications, Internet of Things (IoT) devices and driverless car systems requiring both high accuracy and low power consumption. In recent years, researchers at NVIDIA have demonstrated the effectiveness of convolutional neural networks in end to end control of driverless cars. In this case, the speed of inference is extremely critical to safety.
Fortunately, a number of techniques have been developed in recent years to accelerate the inference speed of cnns. In this chapter, we shall examine some of them and how they have been applied to embedded scenarios.

Depthwise Separable Convolutions

This has been the secret sauce of the best mobile networks. Recall that in a standard convolution, each single filter is applied to all the incoming filters. Hence, the filter has number of channels equal to the number of input channels. To illustrate, consider a CNN layer that receives as input , a ***56 * 56 * 64*** feature map , (***64*** is the input channels), using a ***3 * 3*** filter, the filter size would be ***3 * 3 * 64***, this leads to higher parameters and higher flops. Since each filter is ***3 * 3 * 64***, applying ***128*** filters, the flops is ***3 * 3 * 64 * 56 * 56 * 128***,

Depth Wise Separable Convolutions on the other hand, applies filters channel by channel , hence for a ***3 * 3 * 64*** input , the filter size would be ***3 * 3 * 1***, this would be applied channel by channel, since the input channels is ***64***, there would be ***64*** filters, each of size ***3 * 3 * 1*** , hence the flops becomes
3 * 3 * 56 * 56 * 64.
This dramatically reduces the computational cost of cnns.
The output of Separable Convolutions is passed into a ***1 * 1*** conv which is 9 times more efficient than a ***3 * 3*** conv.

Practical Example

To demonstrate the effectiveness of DepthWise separable convolutions, we shall build a basic mobile network.

Step1: Create Main Module

```
def Unit(x,filters,pool=False):
    res = x
    if pool:
        x = MaxPooling2D(pool_size=(2, 2))(x)
        res = Conv2D(filters=filters,kernel_size=[1,1],strides=(2,2),padding="same")(res)
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = DepthwiseConv2D(kernel_size=[3, 3], strides=[1, 1], padding="same")(out)

    out = BatchNormalization()(out)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[1, 1], strides=[1, 1], padding="same")(out)

    out = keras.layers.add([res,out])

    return out
```

The Module is very identical to the resnet block we earlier constructed, the major differences here is that instead of having two blocks of 3 x 3 standard convolutions, the first layer is a 3 x 3 Depthwise Convolution which performs channels by channel convolution, the output of this is fed intro a 1 x 1 standard convolution.

To really understand what difference this makes, we shall calculate the computation cost of this and compare it to if the 3 x 3 convolution were a standard convolution. Below we assume that both the input filters and output filters are 32.

USING DEPTHWISE SEPARABLE CONVOLUTIONS

Layer	MACS
3 X 3 Depthwise Conv	3 x 3 x 32 x 112 x 112 = 3 612 672
1 x 1 Standard Conv	32 x 112 x 112 x 32 = 12 845 056
Total	**16 457 728**

USING STANDARD CONVOLUTIONS

Layer	MACS
3 X 3 Standard Conv	3 x 3 x 32 x 112 x 112 x 32 = 115 605 504
1 x 1 Standard Conv	32 x 112 x 112 x 32 = 12 845 056
Total	**128 450 560**

As you can see, the difference is so big! 16 million compared to 128 million. You might be wondering why Deptwise convolutions has not entirely replaced standard convolutions. The simple reason is that, they are not as accurate as standard convolutions, hence, when accuracy is first priority, the good-old standard convolutions are still the best. However, when efficiency is of first priority, Depthwise Convolutions are the obvious choice.

As always, we shall stack many modules together to form a model.

```
def MiniModel(input_shape):
    images = Input(input_shape)
    net = Conv2D(filters=32, kernel_size=[3, 3], strides=[1, 1], padding="same")(images)
    net = Unit(net,32)
    net = Unit(net,32)


    net = Unit(net,64,pool=True)
    net = Unit(net,64)
    net = Unit(net, 64)


    net = Unit(net,128,pool=True)
    net = Unit(net,128)
    net = Unit(net, 128)

    net = Unit(net, 256,pool=True)
    net = Unit(net, 256)
    net = Unit(net, 256)

    net = BatchNormalization()(net)
    net = Activation("relu")(net)
    net = Dropout(0.25)(net)

    net = AveragePooling2D(pool_size=(4,4))(net)
    net = Flatten()(net)
    net = Dense(units=10,activation="softmax")(net)

    model = Model(inputs=images,outputs=net)

    return model
```

Notes About Depthwise Convolutions

Depthwise Convolutions have a parameter named depth multiplier which controls the number of feature maps generated per channel. In the above examples, only one feature map is generated per channel, giving rise to a cost of $\boldsymbol{K * K * C_{in}}$ *

out_size * ***out_size*** unlike a standard convolution which has a cost of ***K * K * C_{in} * out_size * out_size * C_{out}***

This assumes a depth multiplier of 1, however, for increased accuracy, we can set the depth multiplier to a number greater than 1, in that case the number of output features would change from ***C_{in}*** to ***C_{in} * D.***

D being the depth multiplier, consequently, the computational cost becomes

K * K * C_{in} * out_size * out_size * D

The depth multiplier allows us to easily tradeoff between accuracy and efficiency**.**

BottleNeck Layers

This is a simple technique that formed the basis of the efficiency of the famous Inception Model and SqueezeNet. A bottleneck layer is a simple 1 x 1 convolution that takes in an image of Large channels and produces an output with fewer number of channels. Since a 1 x 1 convolution incurs minimal overhead, it is used to reduce the number of channels that goes into a 3 x 3 convolution, this applies to any kernel size, not just 3 x 3 convolutions, but to keep things clearer, we shall stick to referencing 3 x 3 convolutions in most of our examples. Although the extra 1 x 1 convolution incurs additional cost, but it greatly reduces computation cost so much that it's effect is negligible.

Here is a typical bottleneck module

```
def Unit(x,output_filters,input_filters,pool=False):
    res = x
    if pool:
        x = MaxPooling2D(pool_size=(2, 2))(x)
        res = Conv2D(filters=output_filters,kernel_size=[1,1],strides=(2,2),padding="same")(res)
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=int(input_filters * 0.5),kernel_size=[1, 1], strides=[1, 1],
padding="same")(out)

    out = BatchNormalization()(out)
    out = Activation("relu")(out)
    out = Conv2D(filters=output_filters, kernel_size=[1, 1], strides=[1, 1], padding="same")(out)

    out = keras.layers.add([res,out])

    return out
```

As seen above, we pass the feature map through a 1 x 1 convolution whose output filters is just half of the input filters, this effectively reduces the computational cost of this module by nearly half.

BOTTLENECK LAYERS

Layer	MACS
1 X 1 Conv (16 filters)	32 x 112 x 112 x 16 = 6 422 528
3 x 3 Conv (32 filters)	3 x 3 x 16 x 112 x 112 x 32 = 57 802 752
Total	**64 225 280**

3 X 3 CONV WITH NO BOTTLENECK

Layer	MACS
3 x 3 Conv (32 filters)	3 x 3 x 32 x 112 x 112 x 32 =
Total	**115 605 504**

As you can see, the bottleneck greatly reduces the computational cost of the network.

In standard networks where accuracy comes first, bottlenecks are the preferred choice for reducing computational cost, however, in mobile nets, depthwise convolutions are generally preferred.

FURTHER READING

In this book, we have discussed the fundamentals of Machine Learning, Artificial Neural Networks and Convolutional Neural Networks for image classification. However, for further study and research, we recommend the following books and papers.

The Deep Learning Book, Ian Goodfellow, Yoshua Bengio and Aaron Courville

Neural Networks and Deep Learning, Michael Nielsen

Stanford's CS231 Course, Andrej Karpathy

Fast.ai

Deeplearning.ai, Andrew Ng

Learning Deep Architectures for AI, Yoshua Bengio

Machine Learning for Humans, Vishal Maini and Samer Sabri

Deep Learning Tutorial, Yoshua Bengio

Deep Learning Tutorial, ICML 2013 Slides, Yann LeCun

Parallelizing Convolutional Neural Networks, Alex Krizhevsky

Reinforcement Learning: An Introduction, Richard Sutton and Andrew Barto

Artificial Intelligence: A modern Approach, Stuart Russell and Peter Norwig

ABOUT THE AUTHORS

JOHN OLAFENWA

John is a self-taught computer programmer, Deep Learning researcher and Machine Learning engineer. He is the CTO of AI Commons Global Limited. He is very passionate about building large-scale systems that will enable developers and researchers more productive in their work. He loves writing tutorials and articles on artificial intelligence, spends a lot of time reading Deep Learning papers and currently works extensively on Computer Vision, Generative Adversarial Networks and Natural Language Processing

Email: johnolafenwa@gmail.com,

Website: john.aicommons.science

Twitter: @johnolafenwa

MOSES OLAFENWA

Moses is a self-taught Computer Programmer, Software Developer, Computer Vision and Deep Learning practitioner with a dream to build technologies that will benefit all of mankind. He is the CEO of AI Commons Global Limited. Skilled in multiple programming languages and multiple frameworks, he is a knowledge-centric fellow who believes in continuous learning, adapting and self development. He is highly experienced in team management, business strategy, project organization, business and project collaboration.

Email: guymodscientist@gmail.com,

Website: moses.aicommons.science

Twitter: @OlafenwaMoses

REFRENCES(INCOMPLETE)

[1] John Olafenwa, Moses Olafenwa. On The Subject of Thinking Machines. https://vixra.org/abs/1801.0413

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition . https://arxiv.org/abs/1512.03385

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun . Identity Mappings in Deep Residual Networks. https://arxiv.org/abs/1603.05027

[4] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, Kilian Weinberger . Deep Networks with Stochastic Depth. https://arxiv.org/abs/1603.09382

[5] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks, https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[6] Sergey Zagoruyko, Nikos Komodakis. Wide Residual Networks, https://arxiv.org/abs/1605.07146

[7] Gustav Larsson, Michael Maire, Gregory Shakhnarovich. FractalNet: Ultra-Deep Neural Networks without Residuals , https://arxiv.org/abs/1605.07648

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun . Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, https://arxiv.org/abs/1502.01852

[9] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, Yoshua Bengio. Maxout Networks, https://arxiv.org/abs/1302.4389

[10] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer . SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, https://arxiv.org/abs/1602.07360

[11] Sergey Ioffe, Christian Szegedy . Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, https://arxiv.org/abs/1502.03167

[12] Min Lin, Qiang Chen, Shuicheng Yan . Network In Network, https://arxiv.org/abs/1312.4400

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions, https://arxiv.org/abs/1409.4842

[14] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna . Rethinking the Inception Architecture for Computer Vision, https://arxiv.org/abs/1512.00567

[15] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi . Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, https://arxiv.org/abs/1602.07261

[16] Karen Simonyan, Andrew Zisserman . Very Deep Convolutional Networks for Large-Scale Image Recognition, https://arxiv.org/abs/1409.1556

[17] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller . Striving for Simplicity: The All Convolutional Net, https://arxiv.org/abs/1412.6806

[18] Rupesh Kumar Srivastava, Klaus Greff, Jürgen Schmidhuber . Highway Networks, https://arxiv.org/abs/1505.00387

[19] Benjamin Graham . Fractional Max-Pooling, https://arxiv.org/abs/1412.6071

[20] Alex Krizhevsky . Learning Multiple Layers of Features from Tiny Images, https://www.researchgate.net/publication/265748773_Learning_Multiple_Layers_of_Features_from_Tiny_Images

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, Li Fei-Fei. ImageNet: A large-scale hierarchical image database, http://ieeexplore.ieee.org/document/5206848

[22] Yoshua Bengio. "Learning Deep Architectures for AI," in *Learning Deep Architectures for AI* , 1, Now Foundations and Trends, 2009, pp.136-
doi: 10.1561/2200000006

[23] Alexandre Boulch . ShaResNet: reducing residual network parameter number by sharing weights, https://arxiv.org/abs/1702.08782

[24] Jie Hu, Li Shen, Gang Sun. Squeeze and Excitation Networks. https://arxiv.org/abs/1709.01507

[25] Gao Huang, Zhuang Liu, Laurens van der Maaten. Densely Connected Convolutional Neural Networks. https://arxiv.org/abs/1608.06993

[26] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, Kaiming He. Agregated Residual Transformations for Deep Neural Networks. https://arxiv.org/abs/1611.05431

[27] Ross Girshick, Fast R-CNN. https://arxiv.org/abs/1504.08083

[28] Shaoqing Ren, Kaimin He, Ross Girshick, Jian Sun. Faster R-CNN: Towards Real Time Object Detection with Region Proposal Networks. https://arxiv.org/abs/111506.01497

[29] Tsung-Yi Lin, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan, Serge Belongie, Feature Pyramid Networks for Object Detection. https://arxiv.org/abs/1612.03144

[30] Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girshick. Mask R-CNN. https://arxiv.org/abs/1703.06870