Introduction to Machine Learning and Deep Learning

90-day Skill Building Study Guide

Deep Learning Nigeria 2018

for Bootcamp Qualification

DSN
Data Science Nigeria

About Data Science Nigeria

Data Science Nigeria is a non-profit organization run and managed by the Data Scientists Network Foundation, with a vision of accelerating Nigeria's development through the solution-oriented application of machine learning to solve social/business problems and to galvanize a data science knowledge revolution in Nigeria and beyond. **Our Approach**

We adopt a practitioner-led model in which experienced and hands-on data scientists in Nigeria and in the diaspora train and mentor Nigerians through face-to-face virtual coaching classes, offer project-based support, and holiday boot camps funded by individuals and corporate organizations. Data Science Nigeria is also leading the PhD4Innovation Hub project that is proactively driving the application of Big Data in solving problems in areas such as financial iInclusion, agriculture, health and social well-being.

Our Successes

- Over 10,000 participants in online and offline initiatives
- Four Nigerian-centric data science products are being supported in the ecosystem
- The largest convergence of 100+ PhD-level academia and industry practitioners across the world as mentors to support the real-world application of learning
- High impact learning bootcamps, academic engagement and direct job placements (full time, freelance and internships) for young Nigerian data scientists
- Strategic partnerships with leading firms including Microsoft, KPMG, the Nigerian Bureau of Statistics, Proshare, Diamond Bank, Interswitch etc.

Contacts

Phone/WhatsApp: +2348140000853

Website: www.datasciencenigeria.org

Email: info@datasciencenigeria.org

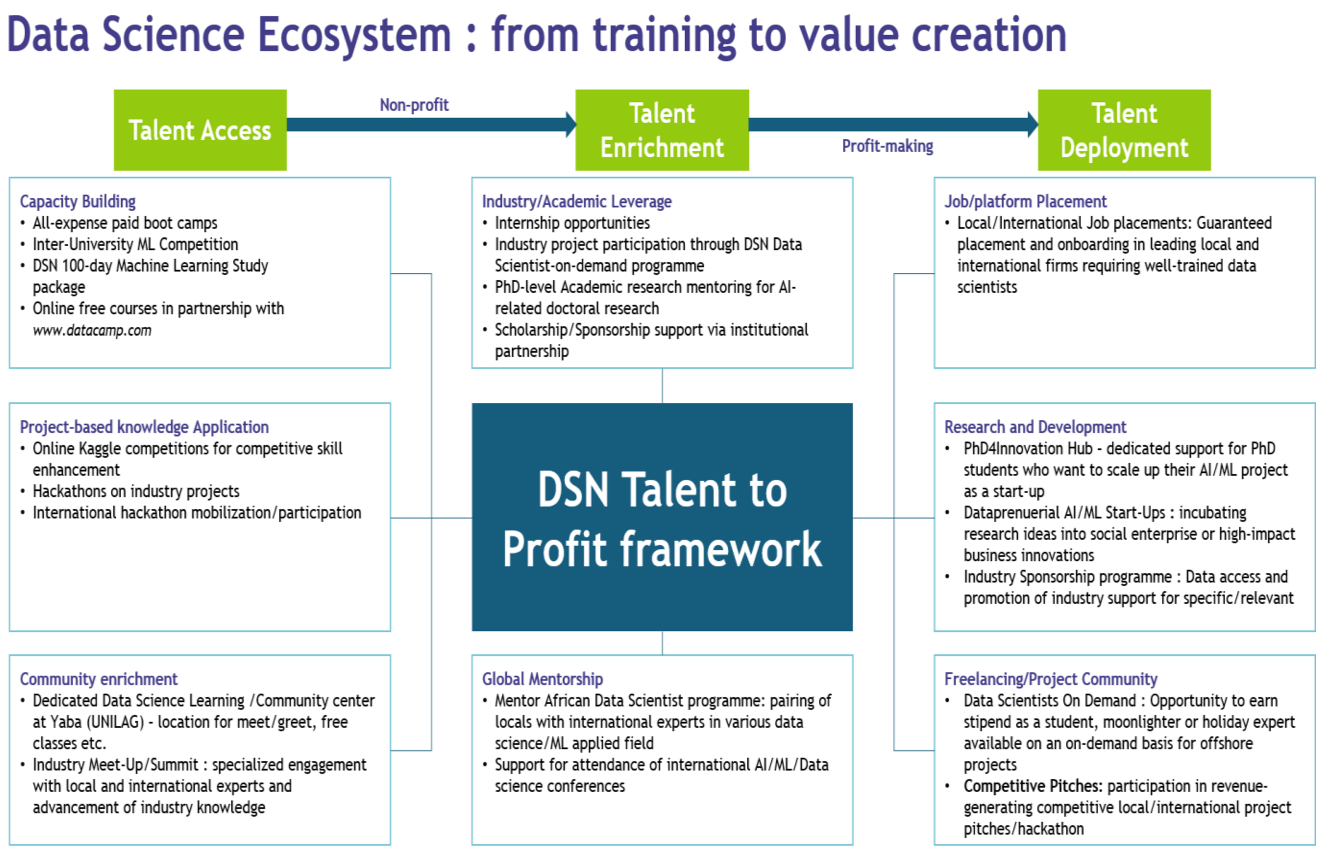
Twitter: Datasciencenig

Instagram: Datasciencenigeria

Facebook: facebook.com/datasciencenig

YouTube: https://goo.gl/Vcjjyp

LinkedIn: https://www.linkedin.com/company/datasciencenigeria/



Data Science Ecosystem : from training to value creation

Capacity Building
- All-expense paid boot camps
- Inter-University ML Competition
- DSN 100-day Machine Learning Study package
- Online free courses in partnership with *www.datacamp.com*

Project-based knowledge Application
- Online Kaggle competitions for competitive skill enhancement
- Hackathons on industry projects
- International hackathon mobilization/participation

Community enrichment
- Dedicated Data Science Learning /Community center at Yaba (UNILAG) - location for meet/greet, free classes etc.
- Industry Meet-Up/Summit : specialized engagement with local and international experts and advancement of industry knowledge

Industry/Academic Leverage
- Internship opportunities
- Industry project participation through DSN Data Scientist-on-demand programme
- PhD-level Academic research mentoring for AI-related doctoral research
- Scholarship/Sponsorship support via institutional partnership

Global Mentorship
- Mentor African Data Scientist programme: pairing of locals with international experts in various data science/ML applied field
- Support for attendance of international AI/ML/Data science conferences

Job/platform Placement
- Local/International Job placements: Guaranteed placement and onboarding in leading local and international firms requiring well-trained data scientists

Research and Development
- PhD4Innovation Hub - dedicated support for PhD students who want to scale up their AI/ML project as a start-up
- Dataprenuerial AI/ML Start-Ups : incubating research ideas into social enterprise or high-impact business innovations
- Industry Sponsorship programme : Data access and promotion of industry support for specific/relevant

Freelancing/Project Community
- Data Scientists On Demand : Opportunity to earn stipend as a student, moonlighter or holiday expert available on an on-demand basis for offshore projects
- **Competitive Pitches:** participation in revenue-generating competitive local/international project pitches/hackathon

Course Outline

Week 1 - Introduction to Machine Learning, Part 1

- What is Machine Learning?
- What Does It Actually Do, or Why Do We Need ML and AI?
- Types of Machine Learning Algorithms

Week 2 - Introduction to Machine Learning, Part 2

- Jupyter Notebooks
- Machine Learning and Concepts of Mathematics and Programming
- Datasets for Machine Learning
- Some More Machine Learning Resources

Week 3 - Introduction to Regression Analysis

- Regression Analysis
- The Concept of Linear Regression
- A Little Bit About the Math
- Activity 1: Predicting Boston Housing Prices

Week 4 - Assumptions of Regression and Tree-based Regression Models

- Assumptions of Linear Regression
- Non- Linear Regression
- Tree-based Regression
- Activity 2: Predicting Boston Housing Prices with Advanced Regression Techniques

Week 5 - Regression and Classification Comparison

- Comparison of Regression and Classification
- Binary Classification
- Multi-class Classification
- Algorithms for Classification
- Activity 3: Titanic Passenger Survival Prediction

Week 6 - Unsupervised Learning

- What is Clustering?
- Use of Clustering Algorithms
- K-means Clustering
- Hierarchical Clustering
- More Clustering Algorithms
- Activity 4: Clustering Breast Cancer Data and World Happiness Data

Week 7 - Anomaly Detection

- Algorithms for Anomaly Detection
- Applications
- Types of Anomalies
- Challenges in Anomaly Detection
- Algorithms for Anomaly Detection
- Activity 5: Numenta Anomaly Benchmark (NAB)

Week 8 - Sequential Rule Mining and Association Rule Mining

- Discovering Sequential Patterns in Sequences
- Association Analysis
- Support
- Confidence
- Lift
- A priori Algorithms
- Activity 6: Market Basket Analysis (Kaggle Competition)

Week 9 - Recommendation System

- Introduction to the Recommendation System
- Collaborative Filtering (CF)
- How the User Based Collaborative Filtering (UBCF) algorithm works
- Strengths & Weaknesses of Neighbourhood Methods
- Content-based Filtering (CBF)
- More Algorithms to Learn
- Activity 7: Build a Recommendation System on The Movies Dataset and Santander Product Recommendation

Week 10 - Introduction to Deep Learning (DL)

- TensorFlow
- Deep Learning
- MNIST Dataset
- Using TensorFlow
- Activity 8: Simple Artificial Neural Network (ANN) model with TensorFlow on MNIST

Week 11 – Convolutional Neural Networks (CNN) using TensorFlow

- Working of CNN
- Understanding the Architecture of CNN
- Activity 8 – The CNN model with TensorFlow on MNIST

Week 12 - Recurrent Neural Networks (RNN)

- Introduction to RNN
- Different Architectures of RNN
- Long Short-term Memory (LSTM)
- Building a LSTM with TensorFlow

- LSTM for Classification

Week 13 - Restricted Boltzmann Machine (RBM)

- Introduction to Restricted Boltzmann Machine (RBM)
- How Does It Work? Why are RBMs Important?
- What are the Applications of RBM?
- RBM Layers
- What can RBM Do After Being Trained?
- How Do We Train an RBM?

Week 14 - Autoencoders (AEs)

- Autoencoder
- Feature Extraction and Dimensionality Reduction with AE
- Autoencoder Structure
- Performance
- Training: Loss Function

Week 15 - More Resources for Deep Learning

- End of Course

Machine Learning Curriculum - Week 1

What is Machine Learning?

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that helps to make machines capable of learning from observational data without being explicitly programmed. ML and AI are not the same. There are a lot of differences as well a lot of similarities. We can also say that machine learning is a component of artificial intelligence.

Tom Mitchell gave a clearer definition of Machine Learning

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

But what it does actually or why do we need ML or AI?

Machine learning and artificial intelligence (AI) are based on automated and self-training algorithms and learn from prior data in order to find the pattern that exists within and then helping machines to make decisions in situations they have never encountered before.

Spam detection: Machine learning is expanding its reach to various domains, such as spam detection. When Google Mail detects spam email automatically, it is the result of machine learning techniques. Some other uses of machine learning are listed below.

Link for further reading

Credit card fraud: Identifying 'unusual' activities on a credit card is often a machine learning problem involving anomaly detection.

Link for further reading

Product recommendations: if Netflix suggests you watch House of Cards or Amazon.com insists that you finally buy those Bose headphones, it's not magic. It's the result of machine learning.

Link for further reading

Medical diagnosis: using a database of symptoms and treatments, a popular machine learning problem is to predict whether a patient has a particular illness.

Link for further reading

Facial Recognition: when Facebook automatically recognizes the faces of your friends in a photo, a machine learning process is running in the background.

Link for further reading

Customer segmentation: using the customer's data gathered during a trial period of a product helps to identify the customers who are most likely to subscribe to the paid version of the product. This a learning problem.

Link for further reading

Types of Machine Learning Algorithms

All of the machine learning applications mentioned above differ from each other. For example, the customer segmentation problem *clusters* the customers into two segments, who will pay for a subscription and the other group who won't. Another example e, the facial recognition problem, aims to *classify* a face. So, we can say that there are basically two types of machine learning algorithms under which a number of algorithms run.

- **Supervised Learning**: this consists of classification and regression algorithms. The target variable is known; therefore it is called 'supervised learning.' For example, when you rate a movie after you have viewed it on Netflix, the suggestion of further movies you might like is predicted using a database of your ratings, known as the training data. When the problem is based on continuous variables, (such as predicting a stock price) it falls under *regression*. With class labels (such as in the Netflix problem), the learning problem is called a *classification* problem.
- **Unsupervised Learning**: this consists of clustering algorithms. The target variable is unknown so it is called 'un-supervised learning.' In the case of unsupervised learning, there is no labeled data set that can be used for making further

predictions. The learning algorithm tries to find patterns or associations in the given data set. Identifying clusters in the data (as in the customer segmentation problem) and reducing the dimensions of the data fall into the unsupervised category.

So, that was a quick introduction to machine learning and different types of machine learning algorithms, and a few applications as well. In this course, we will be exploring machine learning, deep learning and artificial intelligence. Python and its open source technologies will be used as the primary programming language within this course.

Python is the world fastest growing programming language. Before 2007, there was no built-in machine learning package in Python. David Cournapeau (who also developed NumPy and SciPy) developed **Scikit-learn** as a part of a Google Summer of Code project. The project now has 30 contributors and many sponsors including Google and the Python Software Foundation. Scikit-learn provides a range of supervised and unsupervised learning algorithms via a Python interface. Similarly, TensorFlow, also built by Google, provides an open source machine learning framework with the power to handle deep learning.. We will be exploring all these packages and libraries one by one in this course from the grassroots level. First we will learn how to represent data in scikit-learn, and then we will build highly optimized and efficient machine learning models.

Links and Resources for Week 1

All the links are provided for better and enhanced learning. These all are free resources. Learning should never end, and we assume you will go through a few of these links specifically to understand the basic concepts explained in this week. A few of them are courses that are of longer duration. You can continue them along with the course.

1. Artificial Intelligence, Revealed is a quick introduction by Yann LeCun, mostly about machine learning ideas, deep learning, and convolutional neural network
2. Intro to Machine Learning on Udacity is a hands-on scikit-learn Python programming course teaching core ML concepts
3. Machine Learning: Supervised, Unsupervised & Reinforcement on Udacity
4. Machine Learning Mastery is a very carefully laid out step-by-step guide to some particular algorithms.

5. Andrew Ng's Course on Coursera, one of the best courses for machine learning and deep learning in the world, explores all the math and theories behind these fields.
6. Machine Learning is Fun, Part 1, is a simple, non-mathematical approach to machine learning
7. Machine Learning with Python Video Playlist

Machine Learning Curriculum - Week 2

This week we will be exploring the various open source libraries for data loading, data manipulation, and exploratory data visualization.

The best all-in-one IDE for learning ML and Data Science with Python is Jupyter Notebook from Anaconda Distribution. Jupyter Notebook provides a highly interactive notebook type interface where inputs as well as the outputs are displayed in the same cell.

Jupyter Notebook

Here is a video link explaining the use and importance of Jupyter Notebook. To install Jupyter Notebook, go to this link and download the version of Anaconda that matches your operating system (OS) requirements (Windows/Mac/Linux). We will be using the Python 3.6 version. Jupyter Notebooks has a lot of advantages:

1) **High-performance distribution:** easily install 1,000+ data science packages
2) **Package management:** manage packages, dependencies and environments with Conda
3) **Portal to data science:** uncover insights in your data and create interactive visualizations
4) **Interactive:** coding and visualization

Machine learning and Maths and programming concepts: to master machine learning, one has to be good at both Mathematics and programming.

Mathematics: to understand the machine learning algorithms and to **choose the right model**, one needs to understand the math behind them. You do not need to understand all the math, only some sub-branches:

- linear algebra
- probability theory
- optimization
- calculus
- information theory and decision theory

Programming: programming skills are needed for the following tasks:

1. Using ML models
2. Building new models

3. Getting data from various sources
4. Cleaning the data
5. Choosing the right features and validating the data

Some programming languages are more preferred than others for ML because there are a larger number of libraries with most of the ML models already implemented in those languages. These languages include:

- Python
- R (good but it has a slow run time)
- MATLAB (good but costly and slow)
- JULIA (Future best! Very fast, good, but limited libraries as it is new)

As mentioned, we will be using Python throughout this course. Some good books about ML and deep learning are:

- Elements of Statistical Learning
- Pattern Recognition and Machine Learning
- Deep Learning

Coursera Courses:

- Machine Learning
- Deep Learning

Practice:

- Kaggle
- Analytics Vidhya
- Driven Data

Papers:

- arXiv.org e-Print archive
- CVPR, NAACL
- NIPS, ICLR, ICML etc

If you read and implement a lot of good papers (say 100) you will become an expert in ML/DL. After that point you will be able to create your own algorithms and start publishing your work.

Datasets for Machine Learning

Another important consideration while getting into machine learning and deep learning is the dataset that you are going to use. Below is a list of free datasets for data science and machine learning, organized by their use cases.

Datasets for exploratory analysis:

- Game of Thrones
- World University Rankings
- IMDB 5000 Movie Dataset
- Kaggle Datasets

Datasets for general machine learning:

- Wine Quality (Regression)
- Credit Card Default (Classification)
- US Census Data (Clustering)
- UCI Machine Learning Repository

Datasets for deep learning:

- MNIST
- CIFAR
- ImageNet
- YouTube 8M
- Deeplearning.net
- DeepLearning4J.org
- Analytics Vidhya

Datasets for natural language processing:

- Enron Dataset
- Amazon Reviews
- Newsgroup Classification
- NLP-datasets (Git-Hub)
- Quora Answer

Datasets for cloud machine learning:

- AWS Public Datasets
- Google Cloud Public Datasets
- Microsoft Azure Public Datasets

Datasets for time series analysis:

- EOD Stock Prices
- Zillow Real Estate Research
- Global Education Statistics

Datasets for recommender systems:

- MovieLens
- Jester
- Million Song Dataset

Some more Machine Learning Resources

Beginning next week, we will be studying algorithms specifically, such as those used for regression, classification, clustering, association rule mining, recommendation engines, etc. In this week we will provide you with resources to explore all the common machine learning algorithms and links to a few highly rated and renowned courses from top professionals in machine learning and deep learning. All of the resources are available for free online.

Machine learning theory:

- Machine Learning, Stanford University
- Machine Learning, Carnegie Mellon University
- Machine Learning, MIT
- Machine Learning, California Institute of Technology
- Machine Learning, Oxford University
- Machine Learning, Data School

General machine learning with Python and scikit-learn, *Data School*

- Machine Learning with scikit-learn
- Comparing Supervised Learning Algorithms
- Machine Learning with Text

Machine Learning with scikit-learn by Jake Vanderplas:

- Introduction to scikit-learn
- Basic Principles
- Linear Regression
- Support Vector Machines
- Regression Forests
- Dimensionality Reduction
- k-means Clustering
- Density Estimation
- Validation and Model Selection

Decision Trees, The Grimm Scientist

Machine Learning with scikit-learn, Andreas Mueller

- Introduction to Machine Learning with Python
- Scikit-learn tutorial
- Advanced scikit-learn

References

- Datasets for Data Science and Machine Learning

Top 15 Python Libraries for Data Science in 2017

Machine Learning Curriculum - Week 3

Regression Analysis

This is the first lesson in which we will be specifically focusing on machine learning algorithms and their practical implementation in Python using real world datasets. We will start with Regression Analysis and Predictive Modelling.

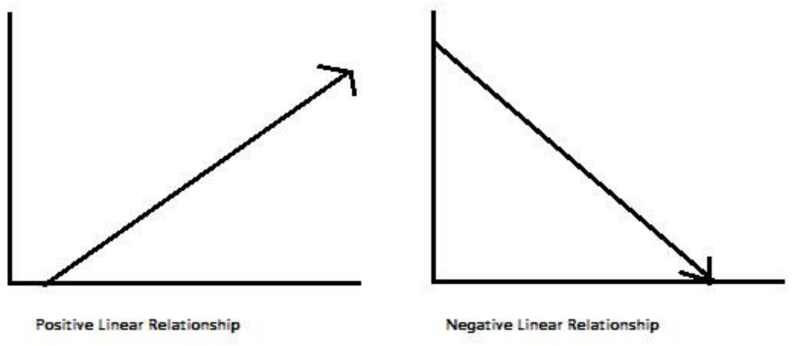
We assume that you have basic knowledge of Python programming and have used the basic libraries such as NumPy, SciPy, Pandas, Matplotlib and scikit-learn,as mentioned in the last week's notes. If you haven't, here are some links for you.

1. **NumPy**
2. **SciPy**
3. **Pandas**
4. **Matplotlib**
5. **scikit-learn**

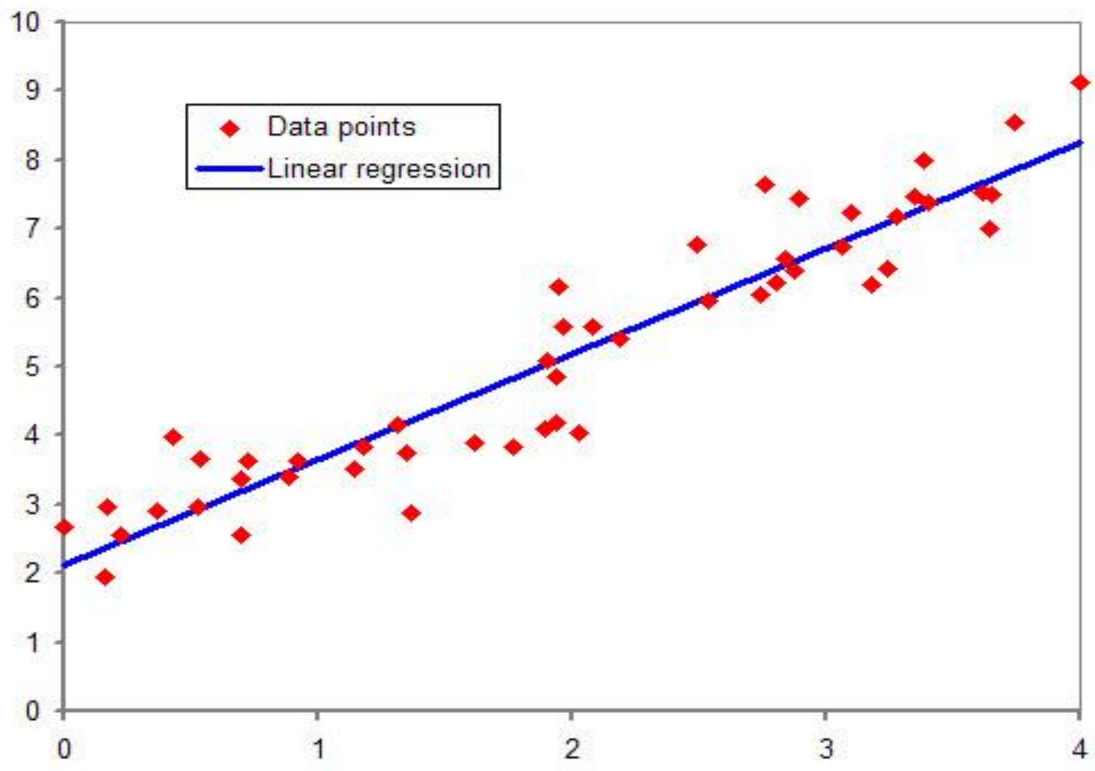
First, we will have a quick introduction to building models in Python, and what better way to start than one of the very basic models, linear regression? Linear regression will be the first algorithm used and you will also learn more complex regression models.

The Concept of Linear Regression

Linear regression is a statistical model that examines the linear relationship between two (simple linear regression, or SLR) or more (multiple linear regression, or MLR) variables, a dependent variable and independent variable(s). A linear relationship basically means that when one (or more) independent variables increase (or decrease), the dependent variable increases (or decreases) too.



As you can see, a linear relationship can be positive (the independent variable goes up, and dependent variable goes up) or negative (the independent variable goes up, but the dependent variable goes down).



A Little Bit About the Math

A relationship between variables Y and X is represented by this equation:

$$Y = mX + b$$

In this equation,

Y is the dependent variable, or the variable we are trying to predict or estimate;

X is the independent variable, the variable we are using to make predictions;

m is the slope of the regression line, and it represents the effect *X* has on *Y*.

Simple Linear Regression (SLR) models also include the errors in the data (also known as residuals). We won't go into it too much now, but residuals are basically the differences between the true value of Y and the predicted/estimated value of Y. It is important to note that in a linear regression, we are trying to predict a continuous variable. In a regression model, we are trying to minimize these errors by finding the "line of best fit"—the regression line from the errors would be minimal. We are trying **to minimize the distance of the red dots from the blue line** as close to zero as possible. It is related to (or equivalent to) minimizing the mean squared error (MSE) or the sum of squares of error (SSE), also called the 'residual sum of squares.' (RSS).

In most cases, we will have more than one independent variable, or multiple variables; they can be as few as two independent variables and up to hundreds (or theoretically even thousands) of variables. In those cases we will use a Multiple Linear Regression model (MLR). The regression equation is pretty much the same as the simple regression equation but with more variables:

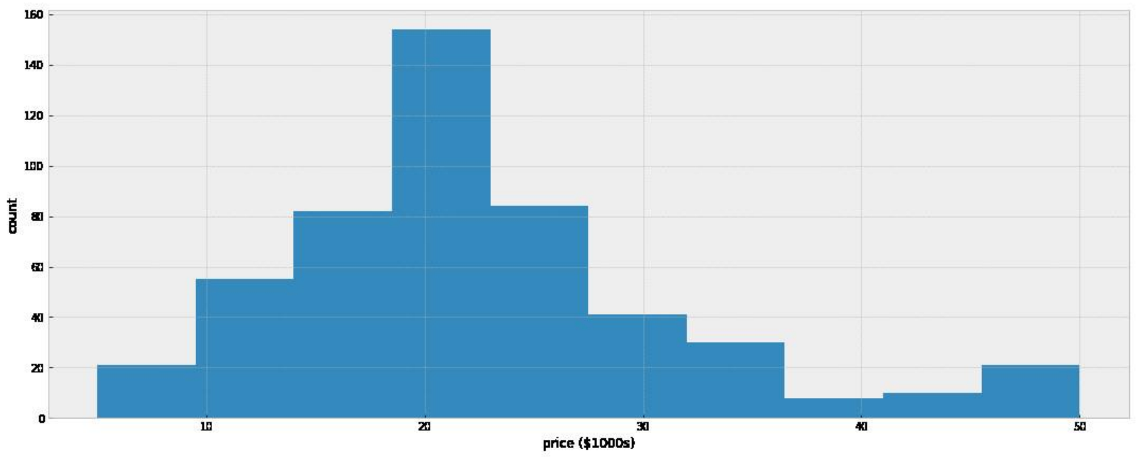
$$Y = b0 + b1X1 + b2X2$$

Activity 1: Predicting Boston Housing Prices

Now we will perform a simple regression analysis on the Boston housing data by exploring simple types of linear regression models. We will use the Boston Housing dataset; this dataset contains information about the housing values in the suburbs of Boston. This dataset was originally taken from the StatLib library maintained at Carnegie Mellon University; it is now available on the UCI Machine Learning Repository. The UCI machine learning repository contains many interesting datasets, and we encourage you to explore it. We will use scikit-learn to import the Boston data as it contains a bunch of useful datasets to practise with and we will also import a linear regression model from scikit-learn. You may also code your own linear regression model as a function or class in Python. It's easy!

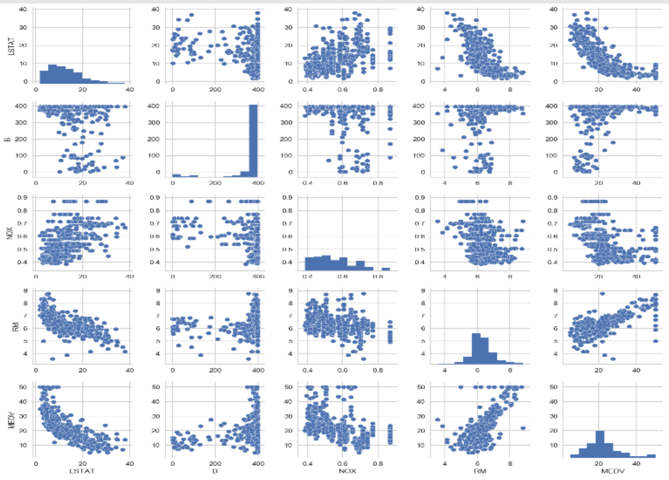
Begin by importing the dataset:

```
from sklearn.datasets import load_boston
data = load_boston()
Print a histogram of the quantity to predict: price
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
plt.figure(figsize=(15, 6))
plt.hist(data.target)
plt.xlabel('price ($1000s)')
plt.ylabel('count')
```



Print the join histogram for each feature:

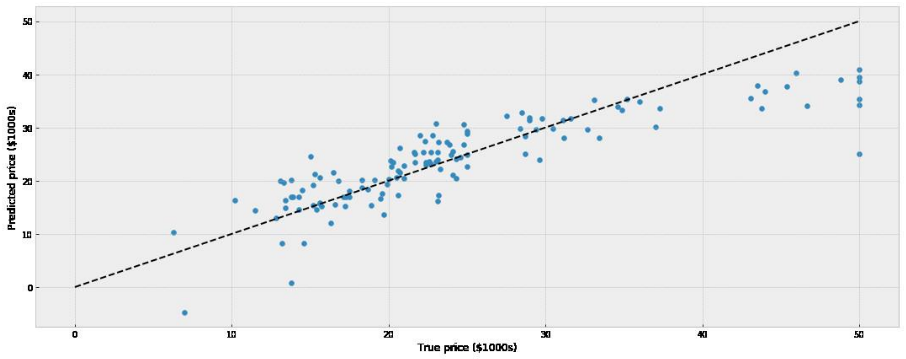
```
for index, feature_name in enumerate(data.feature_names):
   plt.figure(figsize=(4, 3))
   plt.scatter(data.data[:, index], data.target)
   plt.ylabel('Price', size=15)
   plt.xlabel(feature_name, size=15)
   plt.tight_layout()
```



Prediction:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target)
from sklearn.linear_model import LinearRegression
clf = LinearRegression()
clf.fit(X_train, y_train)
predicted = clf.predict(X_test)
expected = y_test
```

```
plt.figure(figsize=(15, 6))
plt.scatter(expected, predicted)
plt.plot([0, 50], [0, 50], '--k')
plt.axis('tight')
plt.xlabel('True price ($1000s)')
plt.ylabel('Predicted price ($1000s)')
plt.tight_layout()
```



As we can see from the results above, the linear regression model is able to predict the values in a good manner, fitting the linear line in best manner.

Next, we will also understand the **various assumptions in regression models**. Moreover, we will also discuss if these assumptions get violated, how do you build your linear regression model then?

Regression is not just fitting a line to the predicted values or defining it as an equation (y = m*x + b) like we just did. There is much more to it. Regression is considered to be the simplest algorithm in machine learning. When we start playing with ML, most of us start with regressions, but it is not always understood well by new learners. It is important, and your understanding of regressions will describe how well you understand the math behind ML. Please note that machine learning is not just loading classes from scikit-learn and fitting data and predicting targets. It is more than that.

Regression is a parametric approach. This means it is going to make assumptions about your data for the purpose of analysis. And, for this reason, it has some limited uses; other regression techniques such as tree-based regression and deep nets are used practically. **Linear regression surely fails to deliver good results with data sets that**

don't fulfill its assumptions. Therefore, for a successful regression analysis, it's essential to validate these assumptions.

So, how would you check whether your dataset fulfills all the regression assumptions?

To understand all the regression assumptions, here is a link. All the assumptions are well explained. More Links for Learning:

Simple and Multiple Linear Regression in Python

Regression analysis using Python

7 Types of Regression Techniques you should know!

Step-by-step guide to execute Linear Regression in Python

Linear Regression (Python Implementation)

Linear Regression in Python: A Tutorial

How to run Linear regression in Python scikit-learn

How to Implement Simple Linear Regression from Scratch with Python

8 ways to perform simple linear regression and measure their speed using Python

Exercises

This week, you have to test yourself in the below- mentioned Kaggle Competition. Have fun!

Sberbank Russian Housing Market

Machine Learning Curriculum - Week 4

Last week we discussed regression analysis, and we learnt about the assumptions or so-called limitations of linear regressions. Linear regression is assumed to be the simplest machine learning algorithm the world has ever seen, and yes! it is. We also discussed how your model can give you poor predictions in real time if you don't obey the assumptions of linear regression. Whatever you are going to predict, whether it is a stock value, sales or some revenue, linear regression must be handled with care if you want to get best predictions from it. Linear regression says that if the data is linear in nature, there must be a linear relationship. But wait! Real-world data is always non-linear. So, what we should we do? Should we try to bring non-linearity into the regression model, or check out the residuals and fitted values, keep applying transformations, and work harder and harder to get the best predictive model using linear regression? This would be a pain and it would take too much time.

Now, the question is, should it be considered as the solution? Or is there any other way to deal with this, so that we can get a better predictive model without getting into these assumptions of linear regression?

Yes! There is a solution; in fact there are several solutions.

There are many different analytical procedures for fitting regressive models of a nonlinear nature (e.g., Generalized Linear/Nonlinear Models (GLZ), Generalized Additive Models (GAM), etc.), or some other better models called tree-based regressive models.

Some of us know about random forest and decision trees. They are very common. In case of classification or regression, they often perform far better than other models. This week we will learn tree-based models such as decision trees and assemble tree-based models including the Random forest (RF), Gradient Boosted Tree (GBT), AdaBoost Tree, and extreme boosted tree for regression analysis. Tree-based models have proven themselves to be both reliable and effective and are now part of any modern predictive modeler's toolkit.

But, there are some cases when a linear regression is based on more accurate assumptions than tree-based models, such as in the following cases:

1. when the underlying function is truly linear, and/or
2. When there are a very large number of features, especially with a very low signal to noise ratio. Tree-based models have a little trouble modelling linear combinations of a large number of features.

The point is, there are probably only a few cases in which linear models like SLR are better than tree-based models or other non-linear models as these fit the data better from the beginning without needing to use transformations.

Tree-based models are more forgiving in almost every way. We don't need to scale the data, and we don't need to do any monotonic transformations (log, square root, etc). We often don't even need to remove outliers. We can throw in features, and it'll automatically partition the data if it aids the fit. We don't have to spend any time generating interaction terms as in case of linear models. And perhaps most importantly, in most cases, tree-based models will probably be notably more accurate.

The bottom line is, we can spend hours playing with the data, generating features and interaction variables and get a 77% R-squared; or, we can use "from sklearn.ensemble import RandomForestRegressor" and in a few minutes get an 82% R-squared.

Check out this link. Let me explain it to you using some examples for clear intuition with an example. Linear regression is a linear model, which means it works really nicely when the data has a linear shape. But, when the data has a non-linear shape a linear model cannot capture the non-linear features. So in this case, we can use the decision trees, which do a better job of capturing the non-linearity in the data by dividing the space into smaller sub-spaces depending on the questions asked.

Now, the question is, when do you use linear regression vs. decision trees? I suspect that the Quora answer would do a better job than me of explaining the difference between them and their applications. Let me quote that for you. Let's suppose you are trying to predict income. The predictor variables that are available are education, age, and city. In a linear regression model, we have an equation with these three attributes. Fine. You'd expect higher degrees of education, higher age and larger cities to be associated with higher income. But what about a PhD who is 40 years old and living in Scranton, Pennsylvania? Is that person likely to earn more than a person with a Bachelor of Science degree who is 35 and living in the Upper West Side of New York City? Maybe not. Maybe education totally loses its predictive power in a city like Scranton? Maybe age is a very ineffective, weak variable in a city like New York? This is a case where decision trees are handy. The tree can split by city and we get to use a different set of variables for each city. Maybe age will be a strong second-level split variable in Scranton, but it might not feature at all in the New York branch of the tree. Education may be a stronger variable in New York. Decision trees, whether Random Forest (RF) or Gradient Boosting Model (GBM), handle messier data and messier relationships better than regression models; and there is seldom a dataset in the real world where relationships are not messy. No wonder we will seldom see a linear regression model outperforming an RF or GBM. So, this is the main idea behind tree (decision tree regression) and ensemble-based models (forest regression/gradient boosting regression/extreme boosting regression).

In the GitHub link, you might have seen a number of other models and their comparisons as well. These are all the available regressive models present in scikit-learn. As you can see, GBM/RF perform the best.

Below are the links to almost all the regression techniques in scikit-learn:

1. ordinary least squares linear regression
2. linear least squares with l2 regularization
3. linear Model trained with L1 prior as regularised (aka the Lasso)
4. support vector regression
5. decision tree regression
6. random forest regression
7. linear model fitted by minimizing a regularized empirical loss with SGD, and
8. gradient boosting for regression

There is a lot more we need to explore in regression analysis. Here are some links:

1. bias and variance tradeoff in regression models

2. underfitting and overfitting in regression models
3. how we optimize our model to avoid underfitting and overfitting
4. regularization techniques
5. L1 and L2 – Ridge and Lasso Regression

Exercises

This week, test yourself on the below- mentioned Kaggle Competition.

Sberbank Russian Housing Market

House Prices: Advanced Regression Techniques

Machine Learning Curriculum - Week 5

Last week we discussed regression modelling and the different types of regression models. This week, we will compare regression analysis with classification analysis and learn a few classification algorithms and when to implement them.

Regression and classification are both related to prediction. Regression predicts a value from a continuous set, whereas classification predicts the 'belonging' to the class.

Let me explain this with the help of an example. In the last few lessons, we saw how we can use regression to predict the price of a house depending on the 'size' (square feet or whatever unit) and, for example, the 'location' of the house, which can be some numerical value that gives us the house’s price - This relates to regression.

On the other hand, classification works in the following manner.

Classification can be built on top of regression:

```
if score > 0.5
   class = 'a'
else
   class = 'b'
end
```

Usually the difference is in what's called the loss function. In regression and classification, the goal of the optimization algorithm (linear, support vector, decision tree, etc.) is to optimize the output of the loss function.

To understand it in a much better way, here are a few links we must dive deeply into:

Difference Between Classification and Regression in Machine Learning

Classification and Regression

What is the main difference between classification and regression problems?

Similarly, if instead of the prediction of price we can try to predict the classes. In this case the price can be classified using labels such as, 'very costly', 'costly', 'affordable', 'cheap', and 'very cheap'. This relates to classification. Each class may correspond to some range of values as explained here.

Classification and ***regression*** both come under the same umbrella of supervised machine learning and share the common concept of using past data to make future predictions or make decisions. That's where their similarity ends. Let me explain using an example.

Have you ever thought about how your email service is able to separate out something as spam, or ham (not spam)?

The process behind it is to teach a model to identify the incoming email by training it with millions of emails that have already been determined to be spam. To classify email as spam, the following aspects are taken into consideration:

1. Whether the email contains spam related words like 'lottery', 'free' etc.
2. Whether a similar email has been classified as spam by the user.
3. How often email with these words is received in this email account.

Classification: The model is trained to identify new emails.

So in this case, after the system has been trained to identify emails that contain spam or don't contain spam, when new emails arrive in your inbox, each email will automatically be classified as spam or not spam. *Classification problems require items to be divided into different categories based on past data.* In a way, we're solving a yes/no problem. Classification can be multi-label as well, as discussed above in case of example of house prices.

Regression: With regression problems, the system attempts to predict a value for an input based on past data. Unlike classification, we are predicting *a value* based on past data, rather than classifying the data into different categories.

Let's say that we wanted to predict whether it would rain, and if it does rain how much rain we would get, maybe in centimeters, based on atmospheric variables such as humidity, temperature, pressure, wind speed and wind direction.

Let's see some more common examples of using classification algorithms, which we will be working with in our Python implementation.

Given a set of input features, predict whether a breast cancer is benign or malignant.

Given an image, correctly classify it as containing cats or dogs.

From a given email, predict whether it's spam email or not.

Let's also discuss the types of classification:

1. Binary classification: this is used when there are only two classes to predict, usually 1 or 0 values.

2. Multi-Class Classification: this is used when there are more than two class labels to predict. We call this a multi-classification task. For example, if we are predicting three types of iris flower species using image classification with thousands of classes (cat, dog, fish, car).

Algorithms for classification:

- *decision trees*
- *logistic regression*
- *naive Bayes*
- *K-nearest neighbors*
- *support vector machines*
- *random forests*
- *many more*

Further Reading

This section provides more resources on the topic if you wish to go deeper.

- How to Build a Machine Learning Classifier in Python with scikit-learn
- Solving Multi-Label Classification Problems (case studies included)
- Machine Learning Classification Strategy In Python
- Python Machine Learning: Scikit-Learn Tutorial
- Spot-Check Classification Machine Learning Algorithms in Python with scikit-learn

Exercises

For your practice for this week, test yourself on the below- mentioned Kaggle Competition.
Titanic: Machine Learning from Disaster

Machine Learning Curriculum - Week 6

Last week we explored classification algorithms; that was a part of supervised machine learning, which also consists of regression analysis and predictive modelling. There are also unsupervised machine learning algorithms. This week we will explore unsupervised machine learning algorithms, such as clustering.

Supervised Learning

Machine learning categories are divided into supervised and un-supervised machine learning. Some of the well-known supervised machine learning algorithms are Support Vector Machine (SVM), Linear Regression, Neural Network (NN) and Naive Bayes (NB). In supervised learning, the training data is labelled, meaning that we already know the target variable we are going to predict when we test the model.

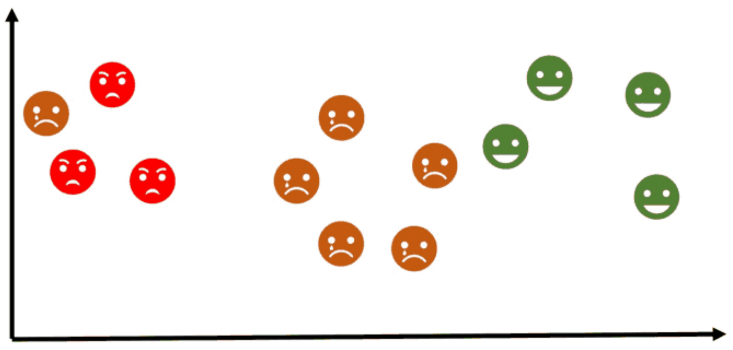
Unsupervised Classification

In unsupervised learning the training data is not labelled and the system tries to learn without a trainer. Some of the most important unsupervised algorithms are clustering, k-means, and association rule learning.

What Is Clustering?

Cluster analysis or **clustering** is the task of grouping a set of objects in such a way that objects in the same group (called a **cluster**) are more similar in some way to each other than to those in other groups (clusters). Clustering is a main task of exploratory data mining, and a common technique used for statistical data analysis. Clustering is used in many fields including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics.

Clustering is widely used in marketing to find naturally occurring groups of customers with similar characteristics, resulting in customer segmentation that more accurately depicts and predicts customer behaviour, and leading to more personalized sales and customer service efforts.



There are a lot of clustering algorithms and each serves a specific purpose and has its own use cases. To learn more about clustering and it definition, here are a few links:

[What is Clustering in Data Mining?]

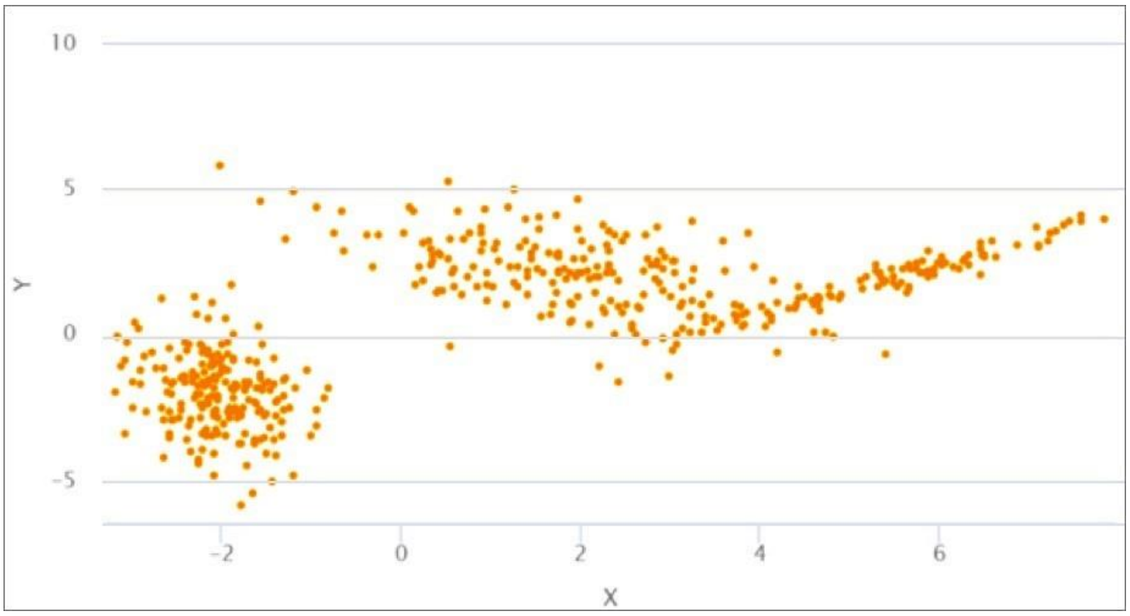
[Data Mining - Cluster Analysis]

[Clustering in Data Mining]

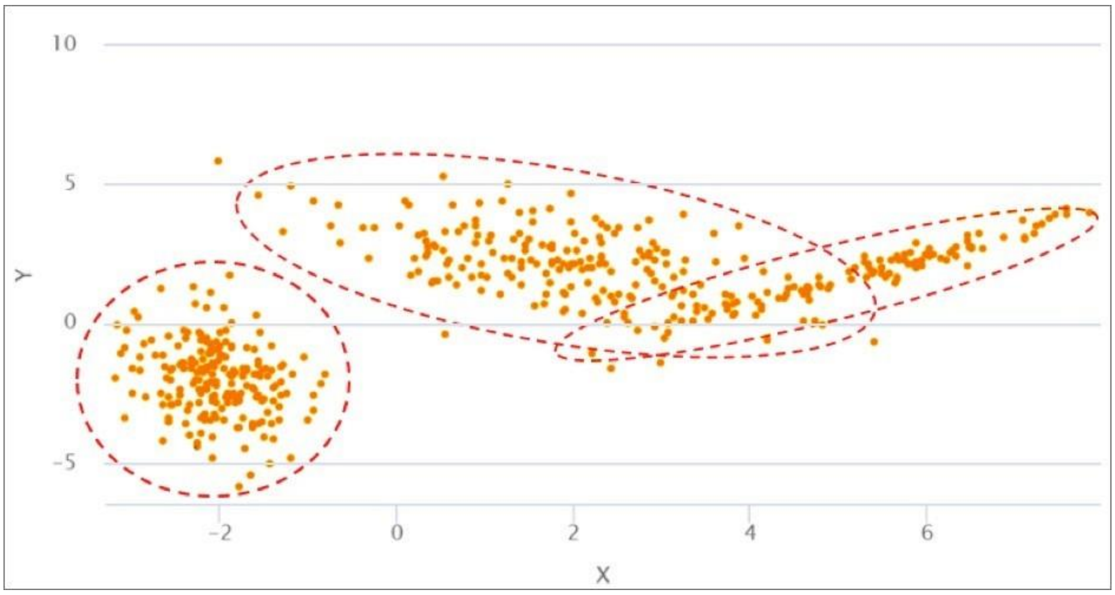
[Data Mining Concepts]

[How Businesses Can Use Clustering in Data Mining]

Various clustering techniques work best for different types of data. Let’s assume that the data is numeric and continuous two-dimensional data as shown in figure below in form of a scatter plot.

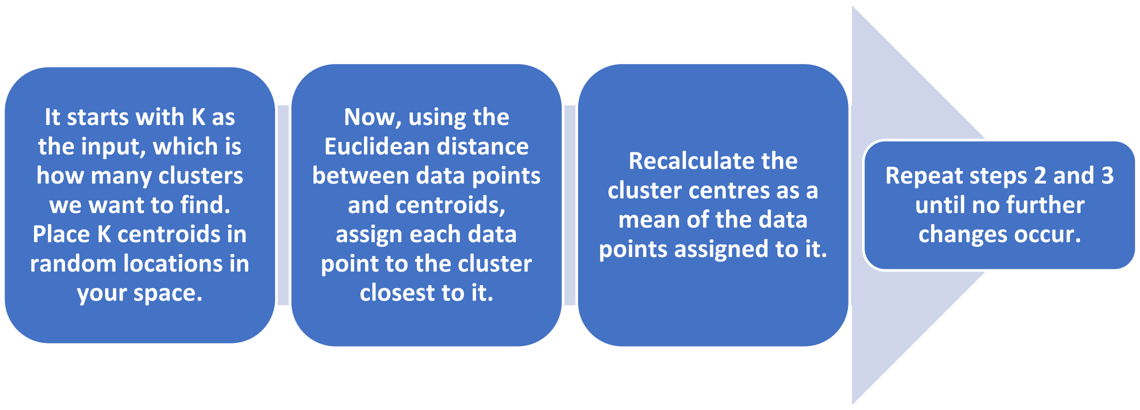


This another scatter plot is created from several "blobs" of different sizes and shapes showing the clusters that exists in the data



We will discuss the K-means and hierarchical clustering algorithms.

K-means clustering



It starts with K as the input, which is how many clusters we want to find. Place K centroids in random locations in your space.

Now, using the Euclidean distance between data points and centroids, assign each data point to the cluster closest to it.

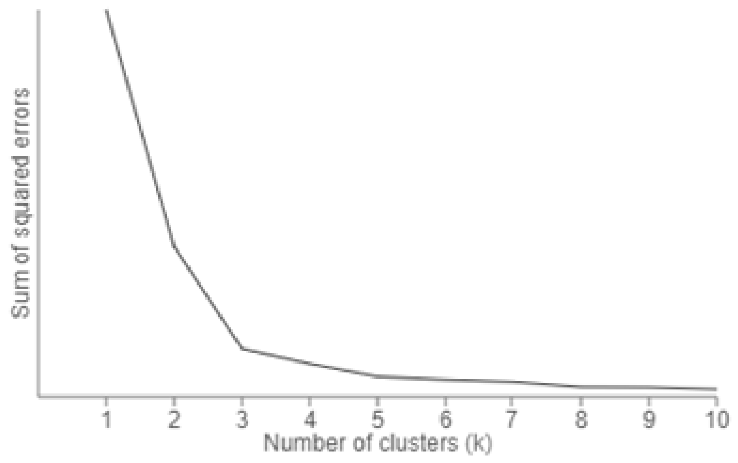
Recalculate the cluster centres as a mean of the data points assigned to it.

Repeat steps 2 and 3 until no further changes occur.

You might be wondering, ‘how do I decide the value of K in the first step’?

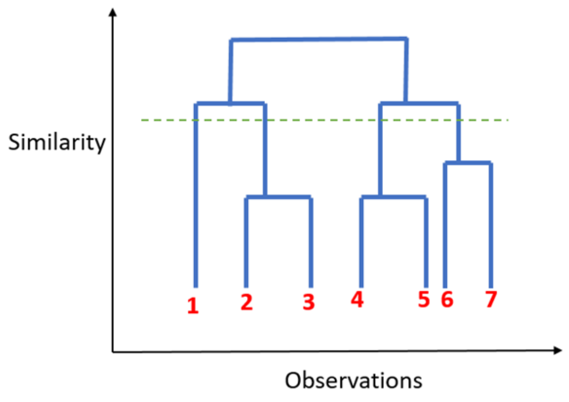
One of the methods is called the Elbow method and can be used to decide an optimal number of clusters. To use it we would run K-mean clustering on a range of K values and plot the percentage of variance explained on the Y-axis and “K” on X-axis as shown in the figure below. If we add more than 3 clusters it doesn't affect the variance explained.

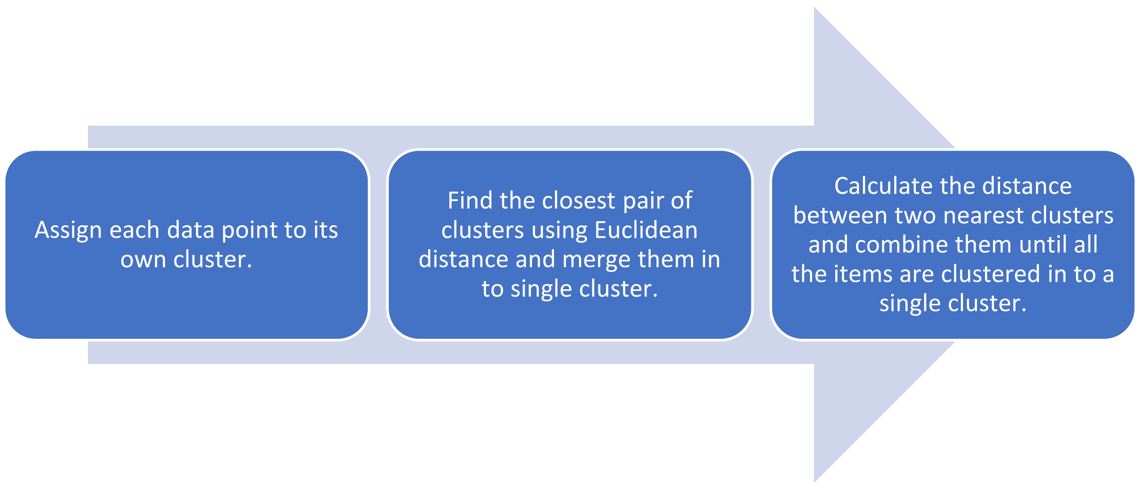
Here is another link for you to explore.



Hierarchical Clustering

Unlike K-mean clustering, hierarchical clustering starts by assigning all data points as their own cluster and thus building the hierarchy; and, it combines the two nearest data point and merges them together into one cluster as shown in the Dendrogram below.




Assign each data point to its own cluster.
Find the closest pair of clusters using Euclidean distance and merge them in to single cluster.
Calculate the distance between two nearest clusters and combine them until all the items are clustered in to a single cluster.

More Algorithms to Learn:

- Mean-Shift Clustering
- Expectation–Maximization (EM) Clustering using Gaussian Mixture Models (GMM)
- Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

More resources for this week:

- The 5 Clustering Algorithms Data Scientists Need to Know
- As for the practice for this week, you have to implement all the clustering algorithms available in Sklearn on these two Kaggle datasets.
- Breast Cancer Wisconsin (Diagnostic) Data Set
- World Happiness Report

Machine Learning Curriculum - Week 7

What is Anomaly Detection?

Anomaly detection refers to the problem of finding instances or patterns in data that deviate from normal behaviour. Depending on the context and domain these deviations can be referred to as anomalies, outliers, or novelties. This week we will be discussing the concept of detection of anomalies and also identifying the anomalies using Python.

Anomaly detection algorithms detect observations that are significantly different from most of what we've seen before. One classic example is detecting credit card fraud. How can we automatically detect purchases that a legitimate credit card owner is very unlikely to have made? Another example is in computer systems security. How can we detect activity on a network that's unlikely to be caused be a legitimate user?

Anomaly detection is often done by building a probabilistic model of the data. This means that you can see what the probability of observing every possible event is according to your model. When you observe an event that has a sufficiently low probability, the model will label it as anomalous.

Anomaly detection is utilized in a wide array of fields, such as the following use cases:

- fraud detection for financial transactions
- Fault Detection in industrial systems
- intrusion detection
- artificial bot listeners

Here are some more links to help you understand the concept of anomaly detection in depth:

- Introduction to Anomaly Detection
- Network traffic anomaly detection using machine learning approaches
- A Novel Algorithm for Network Anomaly Detection Using Adaptive Machine Learning
- Machine learning for anomaly detection
- Machine Learning Techniques for Anomaly Detection: An Overview

Anomaly detection is important because anomalies often indicate useful, critical, and actionable information that can benefit businesses and organizations.

Types of Anomalies

Anomalies can be classified into four categories:

1. **Point Anomalies:** A data point is considered a point anomaly if it is considerably different from rest of the data points. Extreme values in a dataset lie in this category.

2. **Collective Anomalies:** If there is a set of related points that are normal individually but anomalous if taken together, then the set is a collective anomaly. Time series sequences that deviate from the usual pattern come under collective anomalies.

3. **Contextual Anomalies:** If a data point is abnormal when viewed in a particular context but normal otherwise it is regarded as a contextual anomaly. Context is often present in the form of an additional variable, e.g. a temporal or spatial attribute. A point or collective anomaly can be a contextual anomaly if some contextual attribute is present. Most common examples of this kind are present in time series data when a point is within the normal range but does not conform to the expected temporal pattern.

4. **Change Points:** This is unique to time series data and refers to points in time where the typical pattern changes or evolves. Change points are not always considered to be anomalies.

Challenges in Anomaly Detection

These categories have a significant influence on the type of anomaly detection algorithm employed. Anomaly detection is considered to be a hard problem. Anomaly is also defined as a deviation from normal pattern.

However, it is not easy to come up with a definition of normality that accounts for every variation of normal patterns. Defining anomalies is harder still. Anomalies are rare events, and it is not possible to have a prior knowledge of every type of anomaly. Moreover, the definition of anomalies varies across application,. although it is commonly assumed that anomalies and normal points are generated from different processes.

Another major obstacle in building and evaluating anomaly detection systems is the lack of labeled datasets. Though anomaly detection has been a widely studied problem there is still a lack of commonly agreed upon benchmark datasets. In many real-world applications anomalies represent critical failures that are too costly and difficult to obtain. In some domains it is sufficient to have tolerance levels, and any value outside the tolerance intervals can be marked as an anomaly. In many cases labelling anomalies is a time-consuming process and human experts with knowledge of the underlying physical process are required to annotate anomalies. Anomaly detection for time series presents its own unique challenges. This is mainly due to the issues inherent in time series analysis, which is considered to be one of the ten most challenging problems in data mining research.

Algorithms for Anomaly Detection:

- Cluster based anomaly detection (K-mean)
- Elliptic Envelope
- Markov Chain
- Isolation Forest
- One class SVM
- Local Outlier Factor

Exercises

For this week's practice, build a recommendation system using the Kaggle dataset below.

Numenta Anomaly Benchmark (NAB)

Machine Learning Curriculum - Week 8

Sequential rule mining is a data mining technique that consists of discovering rules in sequences. Sequential rule mining has many applications; for example, it is used for analysing the behaviour of customers in supermarkets, users on a website, or passengers at an airport.

Discovering sequential patterns in sequences

An important data mining problem is to design algorithm. To understand the sequences of the patterns in an activities pattern dataset used for discovering hidden patterns in sequences, we have to implement a bunch of Sequence rule mining algorithms and pattern mining techniques. There has been a lot of research on this topic in the field of data mining and various algorithms have been proposed.

A sequential pattern is a sub-sequence that appear in several sequences of a dataset. For example, the sequential pattern <{a}{c}{e}> appears in the two first sequences of our dataset. This pattern is quite interesting. It indicates that customers who bought {a} often bought {c} afterwards, followed by buying {e}.
Such a pattern is said to have the **support** of two sequences because it appears in two sequences from the dataset. Several algorithms have been proposed for finding all **sequential patterns** in a dataset, such as Apriori, SPADE, Prefix Span and GSP. These algorithms use as input a *sequence dataset* and a *minimum support threshold (min-sup).* Then, they will output all sequential patterns having a support no less than *min-sup.* Those patterns are said to be the *frequent sequential patterns.*

Association Analysis

There are a couple of terms used in association analysis that are important to understand. **Association rules** are normally written like this: {Diapers} -> {Beer} means that there is a strong relationship between customers that purchased diapers and also purchased beer in the same transaction. In the above example, {Diaper} is the **antecedent** and {Beer} is the **consequent**. Both antecedents and consequents can have multiple items. In other words, {Diaper, Gum} -> {Beer, Chips} is a valid rule.

Support is the relative frequency with which the rules show up. In many instances, we may want to look for high support in order to make sure it is a useful relationship. However, there may be instances where a low support is useful if you are trying to find 'hidden' relationships.

Confidence is a measure of the reliability of the rule. A confidence of 0.5 in the above example would mean that in 50% of the cases where Diaper and Gum were purchased, the purchase also included Beer and Chips. For product recommendations, a 50% confidence may be perfectly acceptable; but in a medical situation, this level may not be high enough.

Lift is the ratio of the observed support to the level expected if the two rules were independent. The basic rule of thumb is that a lift value close to 1 means the rules were completely independent. Lift values > 1 are generally more "interesting" and could be indicative of a useful rule pattern.

$$\begin{tikzcd}
 & Supprt = \dfrac{Frequency(X,Y)}{N} \\
Rule: X \Rightarrow Y \arrow[ur] \arrow[r] \arrow[dr] & Confidence = \dfrac{Frequency(X,Y)}{Frequency(X)} \\
 & Lift = \dfrac{Support}{Support(X) \times Support(Y)}
\end{tikzcd}$$

The Apriori algorithm is based on conditional probabilities and helps us determine the likelihood of items being bought together based on a- priori data. There are three important parameters: support, confidence and lift. Suppose there is a set of transactions with item1 --> item 2. So, support for item 1 will be defined by n (item1)/n (total transactions). Confidence, on the other hand, is defined as, n (item1 & item2)/n(item1). So, confidence tells us the strength of the association and support tells us the relevance of the rule because we don't want to include rules about items that are seldom bought or, in other words, have low support. Lift is confidence/support. The higher the lift, more the significance there is of applying the Apriori algorithm to determine the rule.

More Resources to explore:

- Introduction to Market Basket Analysis in Python
- Machine learning and Data Mining - Association Analysis with Python
- Apriori Algorithm (Python 3.0)
- Association rules and frequent itemset
- Association Rules and the Apriori Algorithm: A Tutorial
- How to Create Data Visualization for Association Rules in Data Mining

Exercise:

For this week's practice, enjoy doing the Association Mining Algorithms in the Kaggle Competition below.

- Association Rules Mining/Market Basket Analysis

Machine Learning Curriculum - Week 9

Why do we care about the recommendation systems? The answers to this question may be different based on different perspectives. For example, for companies like Amazon, Spotify and Netflix to generate more and more revenues and drive a significant amount of engagement to their websites, recommendation systems have helped to drive exponential growth in their marketplaces. But, for people using Amazon, Spotify and Netflix, recommendation systems save their time and helping them to find the things they are interested in, and incorporating those that are highly liked into their suggestions, so that people don't have to search for them. This is the essence of recommendation systems or recommendation engines.

Conceptually, recommendation systems or recommendation engines use two types of recommendation approaches:

1. collaborative filtering (CF)and
2. content-based filtering (CBF).

Collaborative Filtering

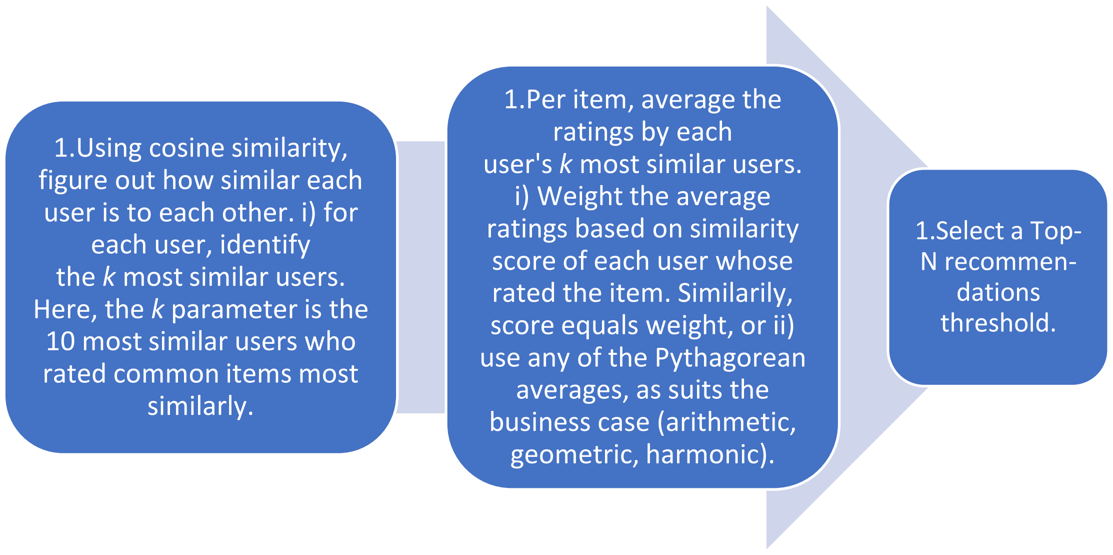
Collaborative filtering is one of the earliest forms of recommendation systems. The earliest developed forms of these algorithms are also known as neighborhood based or memory based algorithms. If using machine learning or statistical model methods, they're referred to as model based algorithms. The basic idea of collaborative filtering is that given a large database of ratings profiles for individual users on what they rated/purchased, we can impute or predict ratings to items not rated/purchased by those users, forming the basis of recommendation scores or top-N recommended items.

Under user-based collaborative filtering (*UBCF*), this memory-based method works with the assumption that users with similar item tastes will also rate items similarly. Therefore, the missing ratings from a user can be predicted by finding other similar users (a neighbourhood). Within the neighbourhood we can aggregate the neighbours' ratings of items unknown to the user, and use that as basis for a prediction.

An inverted approach to nearest neighbours-based recommendations is item-based collaborative filtering. Instead of finding the most similar users to each individual, an algorithm assesses the similarities between the items that are correlated in their ratings or the purchase profile amongst all users.

Some additional starter articles to learning more about collaborative filtering can be found here and here (http://recommender-systems.org/collaborative-filtering/).

How the UBCF algorithm works


1.Using cosine similarity, figure out how similar each user is to each other. i) for each user, identify the *k* most similar users. Here, the *k* parameter is the 10 most similar users who rated common items most similarly.
1.Per item, average the ratings by each user's *k* most similar users. i) Weight the average ratings based on similarity score of each user whose rated the item. Similarily, score equals weight, or ii) use any of the Pythagorean averages, as suits the business case (arithmetic, geometric, harmonic).
1.Select a Top-N recommen-dations threshold.

Strengths & Weaknesses of Neighbourhood Methods

- Strengths: this is simple to implement, and recommendations are easy to explain to users. Transparency about the recommendation to a user can be a great boost to the user's confidence in trusting a rating.
- Weaknesses: these algorithms do not too work well with very sparse ratings matrices. Additionally, they are computationally expensive as the entire user database needs to be processed as the basis of forming recommendations. These algorithms will not work from a cold start since a new user has no historic data profile or ratings for the algorithm to start from.

- Data Requirements: this include a user ratings profile containing items the users has already rated/clicked/purchased. A "rating" can be defined however it fits the business use case.

Content-based filtering (CBF)

Content-based filtering recommenders are broken into three components:

- a model class, TFIDFModel.
- a model provider, TFIDFModelProvider, that computes TF-IDF vectors for items.
- a scorer/recommender class that uses the precomputed model to score items thus computing the user-personalized scores for items.

TF-IDF Recommender with Unweighted Profiles

This is used to compute the unit-normalized TF-IDF vector for each item in the data set. The model contains a mapping of item IDs to TF-IDF vectors, normalized to unit vectors, for each item. The heart of the recommendation process is the scoring method of the item scorer, which is TFIDF Item Scorer, scoring each item by using cosine similarity. The score for an item is the cosine between that item's tag vector and the user's profile vector.

Weighted User Profile

In this variant, rather than just summing the vectors for all positively-rated items a weighted sum of the item vectors is computed for all rated items, with weights being based on the user's rating.

More Algorithms to Learn:

- Recommendation systems: Principles, methods and evaluation
- Amazon.com Recommendations Item-to-Item Collaborative Filtering
- Quick Guide to Build a Recommendation Engine in Python
- Understanding basics of Recommendation Engines (with case study)
- Beginners Guide to learn about Content Based Recommender Engines
- Recommender Systems in Python: Beginner Tutorial
- An Introductory Recommender Systems Tutorial

- Implementing your own recommender systems in Python
- Implementing your own recommender systems in Python

Exercises

For your practice for this week, build a recommendation system using the following Kaggle datasets:

- The Movies Dataset
- Santander Product Recommendation

Machine Learning Curriculum - Week 10

This week we will be getting into deep learning and we will start working with TensorFlow, learning the basic workflow of using TensorFlow with a simple linear model. After loading the so-called MNIST dataset with images of hand-written digits, we will define and optimize a simple mathematical model in TensorFlow. The results will be plotted and discussed as well. We expect that you are familiar with basic linear algebra; otherwise here are some links for you:

- Linear Algebra for Machine Learning
- Basics of Linear Algebra for Machine Learning
- Linear Algebra - Machine Learning
- A comprehensive beginners guide to Linear Algebra for Data Scientists
- Basic Linear Algebra for Deep Learning

Firstly, we will load the MNIST dataset, which is about 12 MB and will be downloaded automatically using the following command:

```
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets("data/MNIST/", one_hot=True)
```

Now the MNIST dataset has now been loaded and consists of about 70.000 images and associated labels (i.e. classifications of the images). The dataset is split into 3 mutually exclusive sub-sets. We will only use the training and testsets in this tutorial.

```
Size of:
- Training-set:         55000
- Test-set:             10000
- Validation-set:       5000
```

The labels are one hot encoded, which means the labels have been converted from a single number to a vector whose length equals the number of possible classes. All elements of the vector are zero except for the ith element. Read about one hot encoding in this link.

Data dimensions

```
MNIST images are 28 pixels in each dimension.
img_size = 28
Images are stored in one-dimensional arrays of this length.
```

```
img_size_flat = img_size * img_size
```

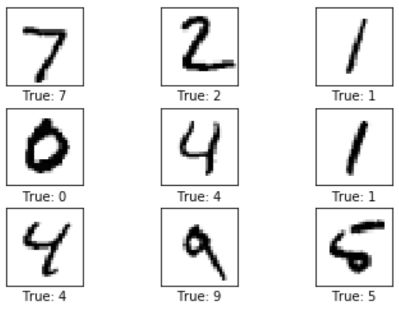
Tuple with height and width of images used to reshape arrays.

```
img_shape = (img_size, img_size)
```

Number of classes, one class for each of 10 digits.

```
num_classes = 10
```

Plot a few images to see if data is correct



The entire purpose of TensorFlow is to have a so-called computational graph that can be executed much more efficiently than if the same calculations were to be performed directly in Python. TensorFlow can be more efficient than NumPy because TensorFlow knows the entire computation graph that must be executed, while NumPy only knows the computation of a single mathematical operation at a time. A TensorFlow graph consists of the following parts as listed below:

- Placeholder variables used to change the input to the graph.
- Model variables that are going to be optimized so as to make the model perform better.
- The model which is essentially just a mathematical function that calculates some output given the input in the placeholder variables and the model variables.
- A cost measure that can be used to guide the optimization of the variables.
- An optimization method which updates the variables of the model.

Placeholder variables

Placeholder variables serve as the inputs to the graph, which we may change each time we execute the graph. First, we define the placeholder variable for the input images. This allows us to change the images that are input to the TensorFlow graph. This is a so-called tensor, which just means that it is a multi-dimensional vector or matrix. The data-type is set to float32 and the shape is set to [None, img_size_flat], where None means that the tensor may hold an arbitrary number of images with each image being a vector of length img_size_flat.

```
x = tf.placeholder(tf.float32, [None, img_size_flat])
```

Next, we have the placeholder variable for the true labels associated with the images that were input in the placeholder variable x. The shape of this placeholder variable is [None, num_classes]; this means it may hold an arbitrary number of labels and each label is a vector of length num_classes, which is 10 in this case.

```
y_true = tf.placeholder(tf.float32, [None, num_classes])
```

Finally, we have the placeholder variable for the true class of each image in the placeholder variable x. These are integers and the dimensionality of this placeholder variable is set to [None], which means the placeholder variable is a one-dimensional vector of arbitrary length.

```
y_true_cls = tf.placeholder(tf.int64, [None])
```

Variables to be optimized

Apart from the placeholder variables defined above and which serve as feeding input data into the model, there are also some model variables that must be changed by TensorFlow so as to make the model perform better using the training data.

The first variable that must be optimized is called weights, and is defined here as a TensorFlow variable that must be initialized with zeros and whose shape is [img_size_flat, num_classes], so it is a 2-dimensional tensor (or matrix) with img_size_flat rows and num_classes columns.

```
weights = tf.Variable(tf.zeros([img_size_flat, num_classes]))
```

The second variable that must be optimized is called biases, and is defined as a 1-dimensional tensor (or vector) of length num_classes.

```
biases = tf.Variable(tf.zeros([num_classes]))
```

Model

This simple mathematical model multiplies the images in the placeholder variable x with the weights and then adds the biases. The result is a matrix of shape [num_images, num_classes] because x has shape [num_images, img_size_flat] and weights has shape [img_size_flat, num_classes], so the multiplication of those two matrices is a matrix with shape [num_images, num_classes] and then the biases vector is added to each row of that matrix. Note that the name logits is typical TensorFlow terminology, but other people may call the variable something else.

```
logits = tf.matmul(x, weights) + biases
```

Now logits is a matrix with num_images rows and num_classes columns, where the element of the i'th row and j'th column is an estimate of how likely the i'th input image is to be of the j'th class. However, these estimates are a bit rough and difficult to interpret because the numbers may be very small or large, so we want to normalize them so that each row of the logits matrix sums to one, and each element is limited between zero and one. This is calculated using the so-called softmax function and the result is stored in y_pred.

```
y_pred = tf.nn.softmax(logits)
```

The predicted class can be calculated from the y_pred matrix by taking the index of the largest element in each row.

```
y_pred_cls = tf.argmax(y_pred, axis=1)
```

Cost-function to be optimized

To make the model better at classifying the input images, we must somehow change the variables for weights and biases. The cross-entropy is a performance measure used in classification. The goal of optimization is therefore to minimize the cross-entropy so it gets as close to zero as possible by changing the weights and biases of the model. TensorFlow has a built-in function for calculating the cross-entropy. Note that it uses the values of the logits because it also calculates the softmax internally.

```
cross_entropy= tf.nn.softmax_cross_entropy_with_logits(logits=logits,labels=y_true)
```

We have now calculated the cross-entropy for each of the image classifications so we have a measure of how well the model performs on each image individually. But in order to use the cross-entropy to guide the optimization of the model's variables we need a single scalar value, so we simply take the average of the cross-entropy for all the image classifications.

```
cost = tf.reduce_mean(cross_entropy)
```

Optimization method

Now that we have a cost measure that must be minimized, we can then create an optimizer. In this case it is the basic form of gradient descent where the step size is set to 0.5. Note that optimization is not performed at this point. In fact, nothing is calculated at all; we just add the optimizer-object to the TensorFlow graph for later execution.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(cost)
```

Performance measures

We need a few more performance measures to display the progress to the user. This is a vector of booleans whether the predicted class equals the true class of each image.

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

This calculates the classification accuracy by first type-casting the vector of booleans to floats, so that False becomes 0 and True becomes 1, and then calculating the average of these numbers.

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Conclusion

The model, after being trained for 1000 optimization iterations and with each iteration using 100 images from the training-set, gave an accuracy of 91%. Refer the notebook for getting the whole code to replicate the same results in your work as well.

Optimization Number	*Accuracy*
Performance before any optimization	The accuracy on the test-set is 9.8%. This is because the model has only been initialized and not optimized at all, so it always predicts that the image shows a zero digit, as demonstrated in the plot below, and it turns out that 9.8% of the images in the test-set happens to be zero digits.
Performance after 1 optimization iteration	After a single optimization iteration the model has increased its accuracy on the test-set to 40.7% up from 9.8%. This means that it mis-classifies the images about 6 out of 10 times, as demonstrated on a few examples below.
Performance after 10 optimization iterations	Accuracy on test-set: 79.3%
Performance after 1000 optimization iterations	After 1,000 optimization iterations, the model only mis-classifies about one in ten images. As demonstrated below, some of the mis-classifications are justified because the images are very hard to determine with certainty even for humans, while others are quite obvious and should have been classified correctly

by a good model. But this simple model cannot reach much better performance and therefore more complex models are needed.

You have had your first look at how we can use TensorFlow, and maybe you are a bit confused and could not understand the terms etc. very precisely,

Here is a link to all the keywords and the glossary used in deep learning. Read it and then come back to the tutorial and go through it again to achieve a deeper understanding of the material.

The notebook is available at the link; you can download it, run it and understand it. Here are a few suggestions for exercises that will help you to improve your skills with TensorFlow. It is important to get hands-on experience with TensorFlow in order to learn how to use it properly.

You may want to back-up this Notebook before you make any changes to it.

- Change the learning-rate for the optimizer.
- Change the optimizer to e.g. AdagradOptimizer or AdamOptimizer.
- Change the batch size, e.e. to 1 or 1000.
- How do these changes affect TensorFlow's performance?
- Do you think these changes will have the same effect (if any) on other classification problems and mathematical models?
- Do you get the exact same results if you run the Notebook multiple times without changing any parameters? Why or why not?
- Change the function plot_example_errors() so it also prints the logits and y_pred values for the mis-classified examples.
- Use sparse_softmax_cross_entropy_with_logits instead of softmax_cross_entropy_with_logits. This may require several changes to multiple places in the source-code. Discuss the advantages and disadvantages of using the two methods.
- Remake the program yourself without looking too much at the source code.

Explain to a friend how the program works.

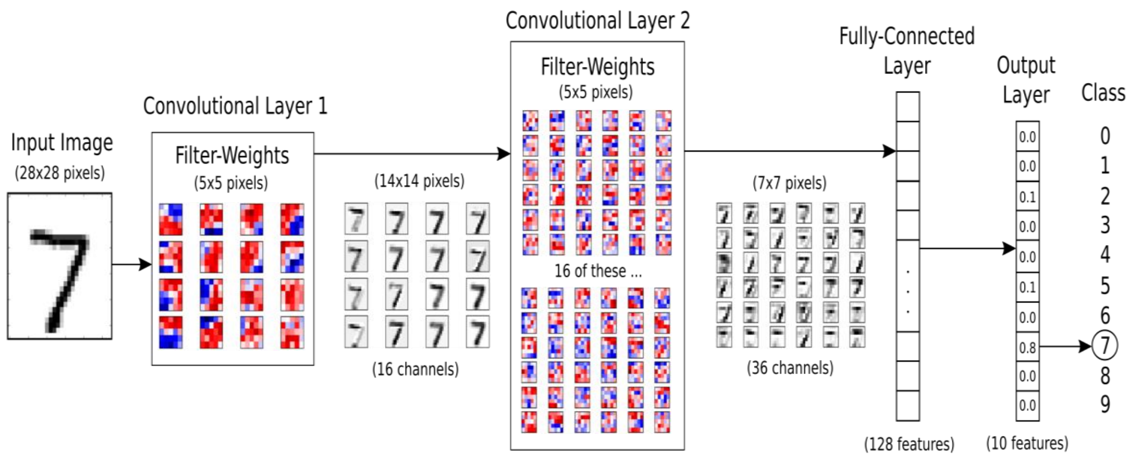
Machine Learning Curriculum - Week 11

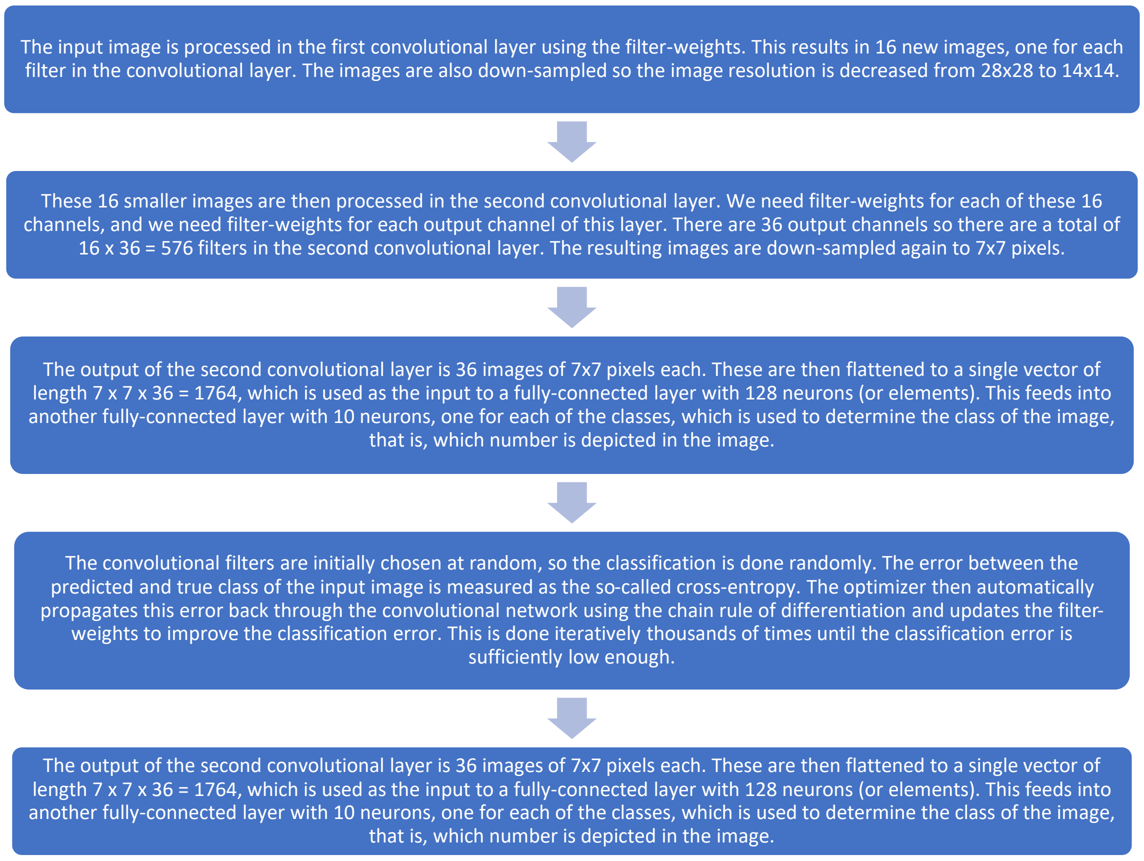
In the previous week, we showed that a simple linear model had about 91% classification accuracy for recognizing hand-written digits in the MNIST data-set. This week we will implement a simple convolutional neural network in TensorFlow that has a classification accuracy of about 99%, or more if you complete some of the suggested exercises. Convolutional networks work by moving small filters across the input image. This means the filters are re-used for recognizing patterns throughout the entire input image. This makes convolutional networks much more powerful than fully-connected networks with the same number of variables. This in turn makes it faster to train convolutional networks.

Here are a few links to get started with CNNs and explain the difference between artificial neural networks (ANNs) and CNNs:

A Beginner's Guide to Understanding Convolutional Neural Networks
Convolutional Neural Networks (CNNs): An Illustrated Explanation
Convolutional Neural Network

The following figure shows how the data flows in a convolutional neural network.




The input image is processed in the first convolutional layer using the filter-weights. This results in 16 new images, one for each filter in the convolutional layer. The images are also down-sampled so the image resolution is decreased from 28x28 to 14x14.

These 16 smaller images are then processed in the second convolutional layer. We need filter-weights for each of these 16 channels, and we need filter-weights for each output channel of this layer. There are 36 output channels so there are a total of 16 x 36 = 576 filters in the second convolutional layer. The resulting images are down-sampled again to 7x7 pixels.

The output of the second convolutional layer is 36 images of 7x7 pixels each. These are then flattened to a single vector of length 7 x 7 x 36 = 1764, which is used as the input to a fully-connected layer with 128 neurons (or elements). This feeds into another fully-connected layer with 10 neurons, one for each of the classes, which is used to determine the class of the image, that is, which number is depicted in the image.

The convolutional filters are initially chosen at random, so the classification is done randomly. The error between the predicted and true class of the input image is measured as the so-called cross-entropy. The optimizer then automatically propagates this error back through the convolutional network using the chain rule of differentiation and updates the filter-weights to improve the classification error. This is done iteratively thousands of times until the classification error is sufficiently low enough.

The output of the second convolutional layer is 36 images of 7x7 pixels each. These are then flattened to a single vector of length 7 x 7 x 36 = 1764, which is used as the input to a fully-connected layer with 128 neurons (or elements). This feeds into another fully-connected layer with 10 neurons, one for each of the classes, which is used to determine the class of the image, that is, which number is depicted in the image.

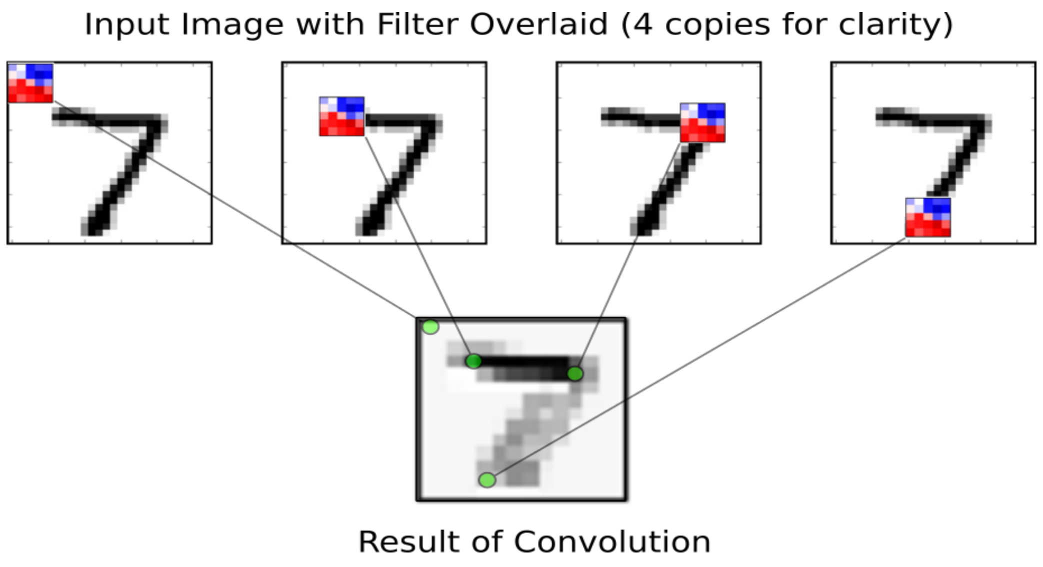
Here are a few more links to understand how a CNN works. If you want to deep dive into the methodology of a CNN model and its functioning, do give them a read.

The best explanation of Convolutional Neural Networks
How do Convolutional Neural Networks work?

Convolutional Layer

The above chart shows the basic idea of processing an image in the first convolutional layer. The input image depicts the number 7, and four copies of the image are shown here so we can see more clearly how the filter is being moved to different parts of the image. For each position of the filter, the dot-product is being calculated between the filter and the image pixels under the filter, which results in a single pixel in the output image. So, moving the filter across the entire input image results in a new image being generated.

The red filter-weights means that the filter has a positive reaction to black pixels in the input image, while blue pixels means the filter has a negative reaction to black pixels. In this case it appears that the filter recognizes the horizontal line of the 7-digit, as can be seen from its stronger reaction to that line in the output image.



The step size for moving the filter across the input is called the stride. There is a stride for moving the filter horizontally (x-axis) and another stride for moving vertically (y-axis). In the source code below the stride is set to 1 in both directions. This means the filter starts in the upper left corner of the input image and is moved 1 pixel to the right in each step. When the filter reaches the end of the image on the right side, the filter is moved back to the left side and 1 pixel down the image. This continues until the filter has reached the lower right corner of the input image and the entire output image has been generated.

When the filter reaches the end of the right side as well as the bottom of the input image, it can be padded with zeroes (white pixels). This causes the output image to be of the exact same dimension as the input image.

Furthermore, the output of the convolution may be passed through a rectified linear unit (ReLU), which merely ensures that the output is positive because negative values are set to zero. The output may also be down-sampled by so-called max pooling, which considers small windows of 2x2 pixels and only keeps the largest of those pixels. This halves the resolution of the input image, e.g. from 28x28 to 14x14 pixels.

Note that the second convolutional layer is more complicated because it uses 16 input channels. We want a separate filter for each input channel, so we need 16 filters instead of just one. Furthermore, we want 36 output channels from the second convolutional layer, so in total we need 16 x 36 = 576 filters for the second convolutional layer. It can be a bit challenging to understand how this works.

Configuration of Convolutional Neural Network

The configuration of the convolutional neural network is defined here for convenience, so you can easily find and change these numbers and re-run the Notebook.

```
# Convolutional Layer 1.
filter_size1 = 5          # Convolution filters are 5 x 5 pixels.
num_filters1 = 16         # There are 16 of these filters.
# Convolutional Layer 2.
filter_size2 = 5          # Convolution filters are 5 x 5 pixels.
num_filters2 = 36         # There are 36 of these filters.
# Fully-connected layer.
fc_size = 128             # Number of neurons in fully-connected layer.
```

Steps for building and using CNN model on MNIST dataset

The following steps are explained in the notebook as well. Make sure that you download the notebook for this week and replicate the same steps yourself. Here is a link to the notebook. Most of the steps are same as the previous week.

- **Load the MNIST Data**
- **Process the features and target variable**
- **Create a helper-functions for creating new variables (weights and biases)**
- **Create a helper function for creating a new Convolutional Layer** (add two or three convolutional layers)
- **Create a helper function for flattening a layer** (A convolutional layer produces an output tensor with 4 dimensions. We will add fully-connected layers after the convolution layers, so we need to reduce the 4-dim tensor to 2-dim which can be used as input to the fully-connected layer.)
- **Create a helper function for creating a new Fully-Connected Layer**
- **Use the model to Predict Class**
- **Cost-function needs to be optimized** (The cross-entropy is a performance measure used in classification. The cross-entropy is a continuous function that is always positive and if the predicted output of the model exactly matches the desired output then the cross-entropy equals zero. The goal of optimization is therefore to minimize the cross-entropy so it gets as close to zero as possible by changing the variables of the network layers.)
- **Optimization Method** (Now that we have a cost measure that must be minimized, we can then create an optimizer. In this case it is the Adam Optimizer which is an advanced form of Gradient Descent.)

No. of Optimizations	***Analysis***	*Accuracy*
Performance before any optimization	The accuracy on the test-set is very low because the model	Accuracy on Test-Set: 10.4% (1036 / 10000)

	variables have only been initialized and not optimized at all, so it just classifies the images randomly.	
Performance after 1 optimization iteration	The classification accuracy does not improve much from just 1 optimization iteration, because the learning-rate for the optimizer is set very low.	Accuracy on Test-Set: 10.9% (1090 / 10000)
Performance after 100 optimization iterations	After 100 optimization iterations, the model has significantly improved its classification accuracy.	Accuracy on Test-Set: 66.3% (6634 / 10000)
Performance after 1000 optimization iterations	After 1000 optimization iterations, the model has greatly increased its accuracy on the test-set to more than 90%.	Accuracy on Test-Set: 93.3% (9329 / 10000)
Performance after 10,000 optimization iterations	After 10,000 optimization iterations, the model has a classification accuracy on the test-set of about 99%.	Accuracy on Test-Set: 98.5% (9852 / 10000)

Conclusion

We have seen that a convolutional neural network works much better at recognizing hand-written digits than the simple linear model we used last week. The convolutional network has a classification accuracy of about 99%, or even more if you make some adjustments, compared to only 91% for the simple linear model. However, the convolutional network is also much more complicated to implement. Therefore, we would like an easier way to program convolutional neural networks and we would also like a better way of visualizing their inner workings. **There are a lot of application programming interfaces (APIs) for deep learning that make the use of these complex models and coding them super easy. One such API is Keras API.**

To code ANN/CNN/RNN models using Keras API is super easy. Go check it out yourself using this link.

Exercises for this week

You may want to backup this Notebook before making any changes.

- Do you get the exact same results if you run the Notebook multiple times without changing any parameters? What are the sources of randomness?
- Run another 10,000 optimization iterations. Are the results better?
- Change the learning rate for the optimizer.
- Change the configuration of the layers, such as the number of convolutional filters, the size of those filters, the number of neurons in the fully-connected layer, etc.
- Add a so-called drop-out layer after the fully-connected layer. Note that the drop-out probability should be zero when calculating the classification accuracy, so you will need a placeholder variable for this probability.
- Change the order of ReLU and max-pooling in the convolutional layer. Does it calculate the same thing? What is the fastest way of computing it? How many calculations are saved? Does it also work for Sigmoid functions and average-pooling?
- Add one or more convolutional and fully-connected layers. Does it help performance?
- What is the smallest possible configuration that still gives good results?
- Try using ReLU in the last fully-connected layer. Does the performance change? Why?
- Try not using pooling in the convolutional layers. Does it change the classification accuracy and training time?
- Try using a 2x2 stride in the convolution instead of max-pooling? What is the difference?
- Remake the program yourself without looking too much at this source-code.
- Explain to a friend how the program works.

There is a Kaggle competition that uses this dataset (check the scripts and forum sections for sample code). You have to participate in this competition and see your standing against the best data scientists in the world. Good luck!

Machine Learning Curriculum - Week 12

Recurrent neural networks are deep learning models with simple structures and a built-in feedback mechanism, or in different words, the output of a layer is added to the next input and fed back to the same layer.

The Recurrent neural network is a specialized type of neural network that solves the issue of **maintaining context for sequential data** -- such as weather data, stock prices, genes, time series etc. At each step, the processing unit takes in an input and the current state of the network and produces an output and a new state that is **re-fed into the network**.

However, **this model has some problems**. It's very computationally expensive to maintain the state for a large number of units, even more so over a long amount of time. Additionally, recurrent networks are very sensitive to changes in their parameters. As such, they are prone to different problems with their gradient descent optimizer -- they either grow exponentially (an exploding gradient) or they drop down to near zero and stabilize (a vanishing gradient), both problems that greatly harm a model's learning capability.

To solve these problems, Hochreiter and Schmidhuber published a paper in 1997 describing a way to keep information over long periods of time and also to solve the network's oversensitivity to parameter changes, i.e., to make backpropagating through recurrent networks more viable.

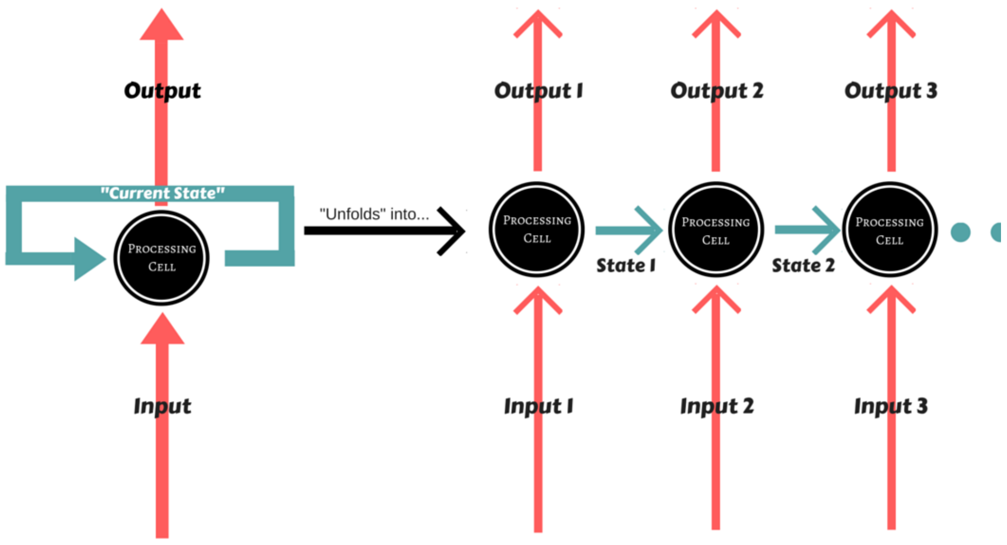
This week we will cover only long short-term memory (LSTM) and its implementation using TensorFlow. The implementation will be provided in Jupyter notebook using the same MNIST dataset that we have been using from last three weeks on deep earning.

Different Architectures of RNN:

- The Long Short-Term Memory Model (LSTM)
- Fully Recurrent Network
- Recursive Neural Networks
- Hopfield Networks

- Elman Networks and Jordan Networks
- Echo State Networks
- Neural history compressor

LSTM



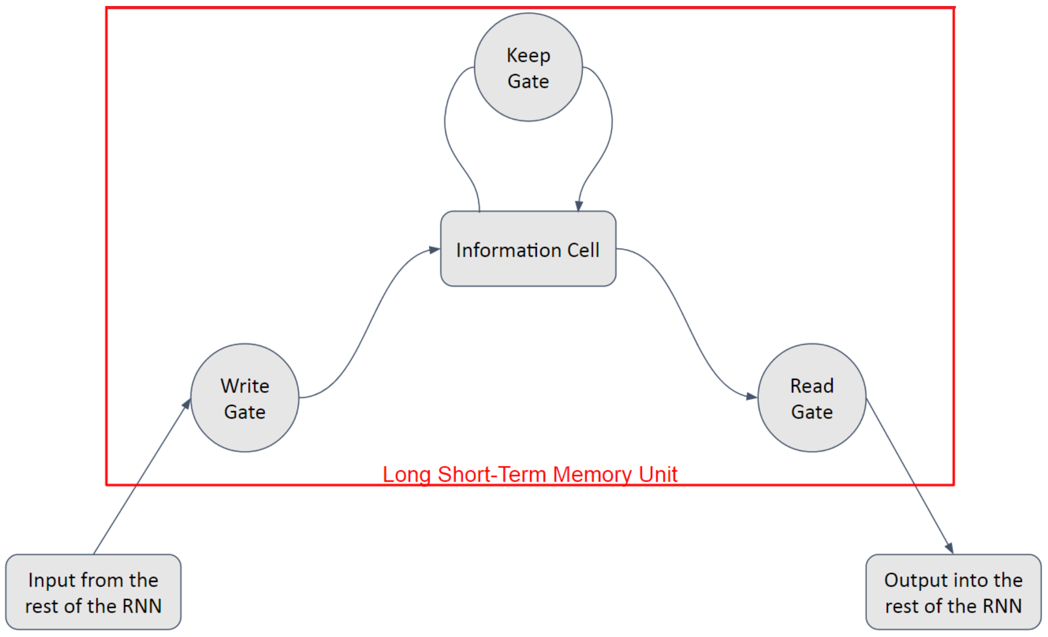
LSTM is one of the proposed solutions or upgrades to the recurrent neural network model. It is an abstraction of how computer memory works. It is "bundled" with whatever processing unit is implemented in the recurrent network, although outside of its flow, and is responsible for keeping, reading, and outputting information for the model. The way it works is simple: there is a linear unit, which is the information cell itself, surrounded by three logistic gates responsible for maintaining the data. One gate is for inputting data into the information cell, one is for outputting data from the input cell, and the last one is to keep or forget data depending on the needs of the network.

Thanks to that, it not only solves the problem of keeping states, because the network can choose to forget data whenever information is not needed; it also solves the gradient problems, since the logistic gates have a very nice derivative.

Long Short-Term Memory Architecture

As seen before, long short-term memory is composed of a linear unit surrounded by three logistic gates. The name for these gates varies from place to place, but the most usual names for them are:

- the "input" or "write" gate, which handles the writing of data into the information cell,
- the "output" or "read" gate, which handles the sending of data back into the recurrent network, and
- the "keep" or "forget" gate, which handles the maintaining and modification of the data stored in the information cell.


Keep Gate
Information Cell
Write Gate
Read Gate
Long Short-Term Memory Unit
Input from the rest of the RNN
Output into the rest of the RNN

The three gates are the centerpiece of the LSTM unit. The gates, when activated by the network, perform their respective functions. For example, the input gate will write whatever data it is passed onto the information cell, the output gate will return whatever data is in the information cell, and the keep gate will maintain the data in the information cell. These gates are analogous and multiplicative, and as such they can modify the data based on the signals they are sent.

Building an LSTM with TensorFlow

LSTM for Classification

Although RNN is mostly used to model sequences and predict sequential data, we can still classify images using a LSTM network. If we consider every image row as a sequence of pixels, we can feed a LSTM network for classification. Because MNIST image shape is 28*28px, we will then handle 28 sequences of 28 steps for every sample.

MNIST Dataset

The function **input_data.read_data_sets(...)** loads the entire dataset and returns an object **tensorflow.contrib.learn.python.learn.datasets.mnist.DataSets**

Train Images: (55000, 784)

Train Labels (55000, 10)

Test Images: (10000, 784)

Test Labels: (10000, 10)

Let's understand the parameters, inputs and outputs

We will treat the MNIST image 28×28 sequences of a vector.

Our simple RNN consists of:

1. one input layer which converts a 28*28-dimensional input to a 128-dimensional hidden layer,
2. one intermediate recurrent neural network (LSTM), and
3. one output layer which converts a 128-dimensional output of the LSTM to 10-dimensional output indicating a class label.

```
n_input = 28 # MNIST data input (img shape: 28*28)
n_steps = 28 # timesteps
n_hidden = 128 # hidden layer num of features
n_classes = 10 # MNIST total classes (0-9 digits)
learning_rate = 0.001
training_iters = 100000
batch_size = 100
display_step = 10

x = tf.placeholder(dtype="float", shape=[None, n_steps, n_input], name="x")
y = tf.placeholder(dtype="float", shape=[None, n_classes], name="y")
The input should be a Tensor of shape: [batch_size, max_time, ...], in our case it would be (?, 28, 28)
Here is a function for the RNN LSTM model.
```

```
def RNN(x, weights, biases):

   # Prepare data shape to match `rnn` function requirements
   # Current data input shape: (batch_size, n_steps, n_input) [100x28x28]

   # Define a lstm cell with tensorflow
   lstm_cell = tf.contrib.rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)

   # Get lstm cell output
   outputs, states = tf.nn.dynamic_rnn(lstm_cell, inputs=x, dtype=tf.float32)

   # Get lstm cell output
   #outputs, states = lstm_cell(x , initial_state)

   # The output of the rnn would be a [100x28x128] matrix. we use the linear activation      to
map it to a [?x10 matrix]
   # Linear activation, using rnn inner loop last output
   # output [100x128] x  weight [128, 10] + []
   output = tf.reshape(tf.split(outputs, 28, axis=1, num=None, name='split')[-1],[-1,128])
   return tf.matmul(output, weights['out']) + biases['out']
```

The whole implementation is available in the notebook here. You will replicate the results to obtain the similar testing accuracy of 96%.

More Algorithms to Learn:

Fundamentals of Deep Learning – Introduction to Recurrent Neural Networks

A Beginner's Guide to Recurrent Networks and LSTMs

Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs

Recurrent Neural Network Model – Andre Ng (Videos)

Recurrent Neural Networks for Beginners

Recurrent Neural Networks and LSTM

How Recurrent Neural Networks work

Recurrent Neural Networks

Exercises

The practice for this week is to build a recommendation system using these Kaggle datasets:

- New York Stock Exchange
- Appliances Energy Prediction

Machine Learning Curriculum - Week 13

Restricted Boltzmann Machine (RBM): RBMs are shallow neural nets that learn to reconstruct data by themselves in an unsupervised fashion.

How does it work?

Simply, an RBM takes the inputs and translates them into a set of numbers that represents them. Then these numbers can be translated back to reconstruct the inputs. Through several forward and backward passes, the RBM will be trained, and a trained RBM can reveal which features are the most important ones when detecting patterns.

Why are RBMs important?

They can automatically extract **meaningful** features from a given input.

What are the applications of RBM?

RBM is useful for collaborative filtering, dimensionality reduction, classification, regression, feature learning, topic modelling and even deep belief Networks.

Is RBM a generative model?

Yes, RBM is a generative model. What is a generative model?

First, let's see what is different between discriminative and generative models.

Discriminative model: consider a classification problem in which we want to learn to distinguish between sedan cars (y = 1) and SUV cars (y = 0) based on some features of cars. Given a training set, an algorithm such as a logistic regression tries to find a straight line—that is, a decision boundary—that separates the SUV from a sedan.

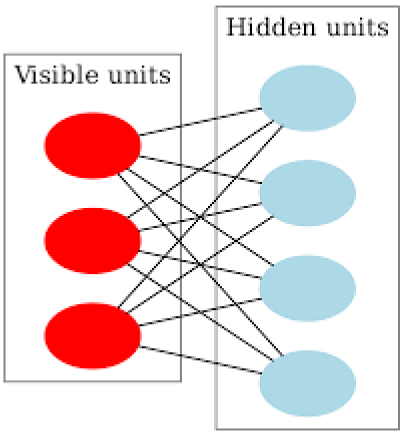
Generative: using the example of cars, we can build a model of what sedan cars look like. Then, looking at SUVs, we can build a separate model of what SUV cars look like. Finally, to classify a new car, we can compare the new car to the sedan model, and compare it to the SUV model to see whether the new car looks more like an SUV or a sedan.

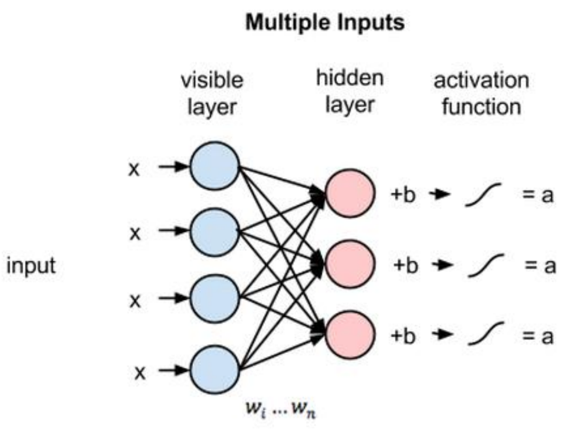
Generative models specify a probability distribution over a dataset of input vectors. We can do both supervised and unsupervised tasks with generative models.

- In an unsupervised task, we try to form a model for P(x), where x is an input vector.
- In the supervised task, we first form a model for P(x|y), where y is the label for x. For example, if y indicates whether an example is an SUV (0) or a sedan (1), then p(x|y = 0) models the distribution of SUVs' features, and p(x|y = 1) models the distribution of sedans' features. If we manage to find P(x|y) and P(y), then we can use Bayes rule to estimate P(y|x), because: p(y|x) = p(x|y)p(y)/p(x)

RBM layers

An RBM has two layers. The first layer of the RBM is called the visible or input layer. MNIST images have 784 pixels, so the visible layer must have 784 input nodes. The second layer is the hidden layer, which possesses i neurons in this case. Each hidden unit has a binary state, which we'll call it si, and turns either on or off (i.e., si = 1 or si = 0) with a probability that is a logistic function of the inputs it receives from the other j visible units, called for example, p (si = 0) which is also shown in the figure below.





Each node in the first layer also has a bias shared among all visible units. We also define the bias of the second layer as well as the bias shared among all visible units.

What RBM can do after training?

Think of an RBM as a model that has been trained, and now it can calculate the probability of observing a case (e.g. a wet road) given some hidden/latent values (e.g. rain). That is, the RBM can be viewed as a generative model that assigns a probability to each possible binary state vectors over its visible units (v).

So, for example, if we have 784 units in the visible layer, it will generate a probability distribution over all the 784 possible visible vectors, i.e, p(v).

It would be really cool if a model, after training, can calculate the probability of a visible layer, given the hidden layer values.

How do we train an RBM?

1) There are two phases of training an RBM: the forward pass, and
2) The backward pass or reconstruction.

Phase 1 forward pass: Processing happens in each node in the hidden layer. That is, input data from all the visible nodes are being passed to all the hidden nodes. This computation begins by making stochastic decisions about whether to transmit that input or not (i.e. to determine the state of each hidden layer). At the hidden layer's nodes, X is multiplied by a W and added to h_bias. The result of those two operations is fed into the sigmoid function, which produces the node’s output/state. As a result, one output is produced for each hidden node. So, for each row in the training set, a tensor of probabilities is generated, which in our case it is of size [1x500], and totally 55000 vectors (*h0*=[55000x500]).

Then, we take the tensor of probabilities (from a sigmoidal activation) and make samples from all the distributions, h0. That is, we sample the activation vector from the probability distribution of hidden layer values. Samples are used to estimate the negative phase gradient; this will be explained later.

The whole implementation of RBM can be found in this notebook.

More Algorithms to learn:

A Beginner’s Tutorial for Restricted Boltzmann Machines

Introduction to Restricted Boltzmann Machines

Oversimplified introduction to Boltzmann Machines

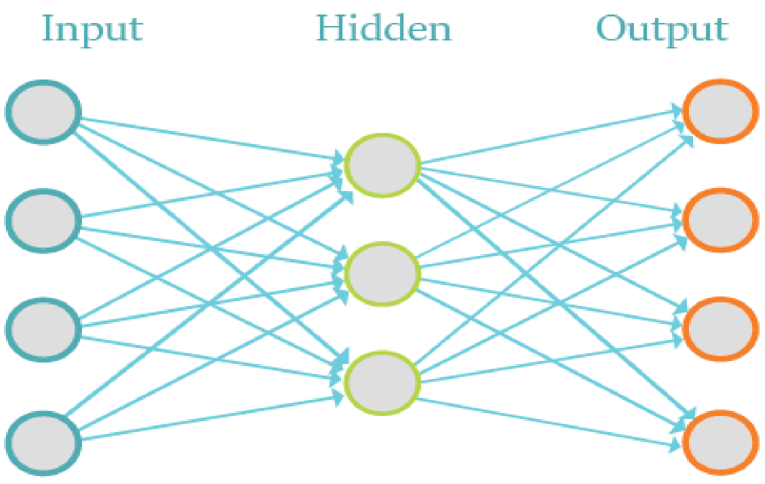
Applying deep learning and a RBM to MNIST using Python

Exercises

This week, you will build an RBM on MNIST datasets using the code given in the notebook, and compare it with the simple model, CNN model and LSTM model we built earlier.

An autoencoder, also known as an auto associator or Diabolo network, is an artificial neural network employed to recreate the given input. It takes a set of **unlabelled** inputs, encodes them, and then tries to extract the most valuable information from them. They are used for feature extraction, learning generative models of data, dimensionality reduction, and can be used for compression as well.

A 2006 paper named “Reducing the Dimensionality of Data with Neural Networks,” done by G. E. Hinton and R. R. Salakhutdinov, showed better results than years of refining other types of network; it was a breakthrough in the field of neural networks, a field that had been stagnant for 10 years. Now, autoencoders based on restricted Boltzmann machines are employed in some of the largest deep learning applications. They are the building blocks of deep belief networks (DBN). The figure below shows the basic architecture of an auto-encoder.



Feature Extraction and Dimensionality Reduction

An example given by Nikhil Buduma in KdNuggets can excellently explains the utility of this type of neural network.. Let’s say that we want to extract what emotion the person in a photograph is feeling using as an example the following 256 x 256 grayscale picture:

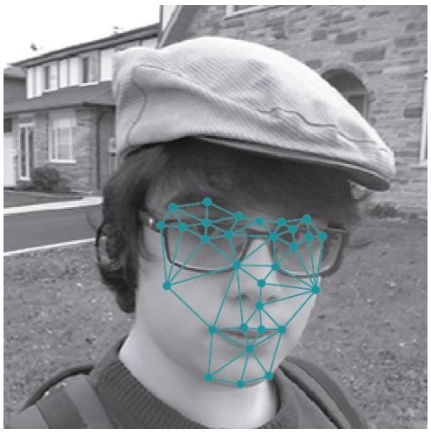
But then we start facing a bottleneck. This image, being 256 x 256 corresponds with an input vector of 65536 dimensions! If we used an image produced by a conventional cell phone camera that generates images of 4000 x 3000 pixels, we would have 12 million dimensions to analyse.

This bottleneck becomes even more difficult to overcome as the difficulty of a machine learning problem is increased as more dimensions are involved. According to a 1982 study by C.J. Stone, the time to fit a model, at best, is:

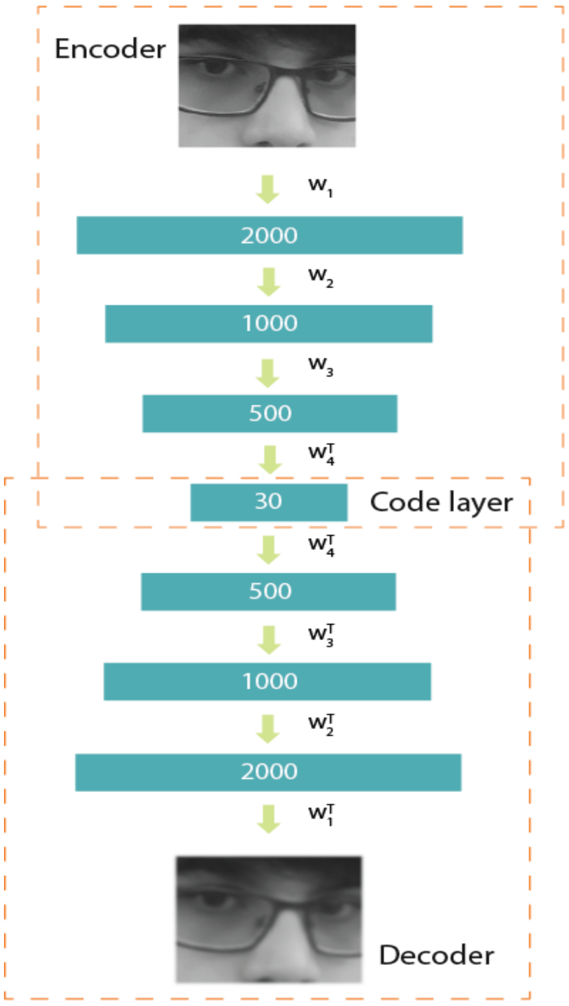
$$m^{-p/(2p+d)}$$

m: Number of data points
d: Dimensionality of the data
p: Parameter that depends on the model

As you can see, it increases exponentially! Returning to our example, we don't need to use all of the 65,536 dimensions to classify an emotion. A human identifies emotions according to some specific facial expressions, and some **key features**, like the shapes of the mouth and eyebrows.



Autoencoder Structure



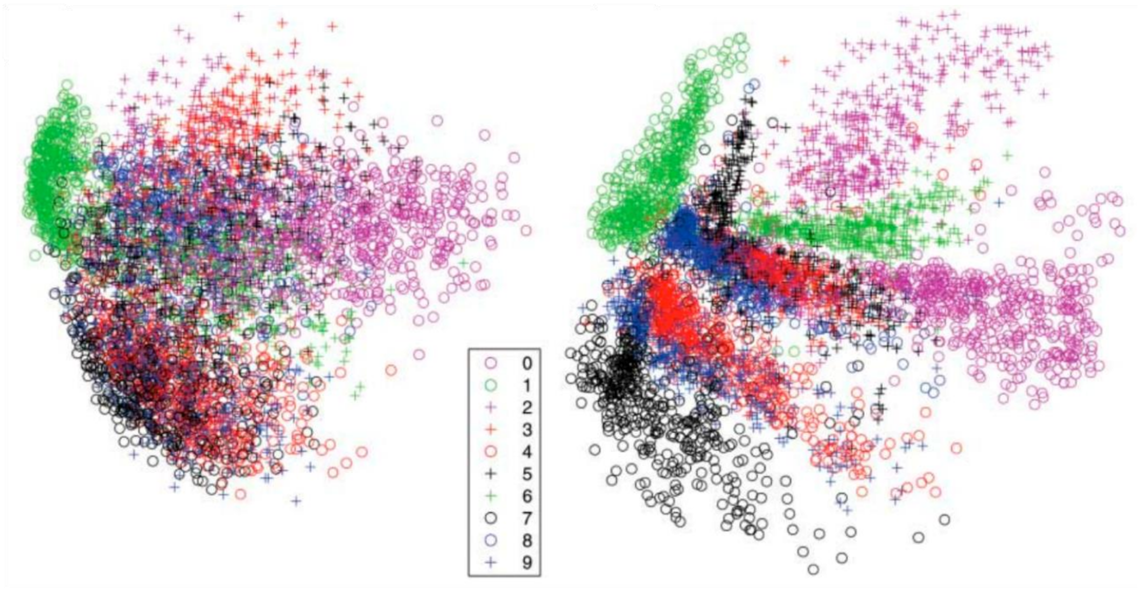
An autoencoder can be divided in two parts, the encoder and the decoder.

The encoder compresses the representation of an input. In this case we are going to compress the face of our actor, consisting of 2000-dimensional data to only 30 dimensions, taking some steps between the compressions.

The decoder is a reflection of the encoder network. It works to recreate the input as closely as possible. It has an important role during training, to force the autoencoder to select the most important features in the compressed representation.

After the training has been done, you can use the encoded data as reliable dimensionally-reduced data, applying it to any problems that a dimensionality reduction problem seems to fit.

This image was extracted from the Hinton paper comparing the two-dimensional reduction for 500 digits of the MNIST, with PCA on the left and autoencoder on the right. We can see that the autoencoder provided us with a better separation of data.



Training: loss function

An autoencoder uses the loss function to properly train the network. The loss function will calculate the differences between the output and the expected results. After that, we can minimize this error by doing gradient descent. There is more than one type of loss function; it depends on the type of data used.

Binary Values:

$$l(f(x)) = -\sum_k (x_k log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k)\ )$$

For binary values, we can use an equation based on the sum of Bernoulli's cross-entropy.

x_k is one of our inputs and $\hat{x}_k$ is the respective output.

We use this function so that if x_k equals to one, we want to push $\hat{x}_k$ as close as possible to one. The same if x_k equals to zero.

If the value is one, we just need to calculate the first part of the formula, that is, $-x_k log(\hat{x}_k)$. Which, turns out to just calculate $-log(\hat{x}_k)$.

And if the value is zero, we need to calculate just the second part, $(1 - x_k) \log(1 - \hat{x}_k)\ )$ - which turns out to be $log(1 - \hat{x}_k)$.

Real values:

$$l(f(x)) = -1/2 \sum_k (\hat{x}_k - x_k\ )^2$$

As the above function would behave badly with inputs that are not 0 or 1, we can use the sum of squared differences for our Loss function. If you use this loss function, it's necessary that you use a linear activation function for the output layer.

As it was with the above example, x_k is one of our inputs and $\hat{x}_k$ is the respective output, and we want to make our output as similar as possible to our input.

Loss Gradient:

$$\nabla_{\hat{a}(x^{(t)})}\ l(\ f(x^{(t)})) = \hat{x}^{(t)} - x^{(t)}$$

We use the gradient descent to reach the local minumum of our function $l(\ f(x^{(t)})$, taking steps towards the negative of the gradient of the function in the current point.

Our function talks about the preactivation of the output layer $(\nabla_{\hat{a}(x^{(t)})})$ of the loss $l(\ f(x^{(t)})$.

It's actually a simple formula, it just calculates the difference between our output $\hat{x}^{(t)}$ and our input $x^{(t)}$.

Then our network just backpropagates our gradient $\nabla_{\hat{a}(x^{(t)})}\ l(\ f(x^{(t)}))$ through the network using **backpropagation**.

Find the notebook for the implementation of AE at this link and replicate the code on yourself.

More algorithms to learn:

- Autoencoders
- Deep Learning: Autoencoders Fundamentals and types
- Autoencoders — Bits and Bytes of Deep Learning
- Autoencoders – Deep Learning Book

- A Beginner’s guide to Deep Autoencoders
- Building Autoencoders in Keras

Exercises

For your practice week, build an AE in this Kaggle Competition:

Credit Card Fraud Detection.

Here is a link for help.

Extra Learning References

In this last week, we will provide you with the best machine learning and deep learning resources to explore.

We hope you have found this course exciting and you have learnt and explored a lot through the contents and resources we have shared with you. Here are the awesome ML/DL resources that you can look into to enhance your knowledge.

Free Online Books

1. Deep Learning by Yoshua Bengio, Ian Goodfellow and Aaron Courville (May 2015)
2. Neural Networks and Deep Learning by Michael Nielsen (Dec 2014)
3. Deep Learning by Microsoft Research (2013)
4. Deep Learning Tutorial by LISA lab, University of Montreal (Jan 6 2015)
5. neuraltalk by Andrej Karpathy: NumOy-based RNN/LSTM implementation
6. An introduction to genetic algorithms
7. Artificial Intelligence: A Modern Approach
8. Deep Learning in Neural Networks: An Overview

Courses

1. Machine Learning - Stanford by Andrew Ng in Coursera (2010-2014)
2. Machine Learning - Caltech by Yaser Abu-Mostafa (2012-2014)
3. Machine Learning - Carnegie Mellon by Tom Mitchell (Spring 2011)
4. Neural Networks for Machine Learning by Geoffrey Hinton in Coursera (2012)
5. Neural networks class by Hugo Larochelle from Université de Sherbrooke (2013)
6. Deep Learning Course by CILVR lab @ NYU (2014)
7. A.I - Berkeley by Dan Klein and Pieter Abbeel (2013)
8. A.I - MIT by Patrick Henry Winston (2010)
9. Vision and learning - computers and brains by Shimon Ullman, Tomaso Poggio, Ethan Meyers @ MIT (2013)
10. Convolutional Neural Networks for Visual Recognition - Stanford by Fei-Fei Li, Andrej Karpathy (2017)
11. Deep Learning for Natural Language Processing - Stanford
12. Neural Networks - usherbrooke
13. Machine Learning - Oxford (2014-2015)

14. Deep Learning - Nvidia (2015)
15. Graduate Summer School: Deep Learning, Feature Learning by Geoffrey Hinton, Yoshua Bengio, Yann LeCun, Andrew Ng, Nando de Freitas and several others @ IPAM, UCLA (2012)
16. Deep Learning - Udacity/Google by Vincent Vanhoucke and Arpan Chakraborty (2016)
17. Deep Learning - UWaterloo by Prof. Ali Ghodsi at University of Waterloo (2015)
18. Statistical Machine Learning - CMU by Prof. Larry Wasserman
19. Deep Learning Course by Yann LeCun (2016)
20. Designing, Visualizing and Understanding Deep Neural Networks-UC Berkeley
21. UVA Deep Learning Course, MSc in Artificial Intelligence for the University of Amsterdam.
22. MIT 6.S094: Deep Learning for Self-Driving Cars
23. MIT 6.S191: Introduction to Deep Learning
24. Berkeley CS 294: Deep Reinforcement Learning
25. Keras in Motion video course
26. Practical Deep Learning For Coders by Jeremy Howard - Fast.ai
27. Introduction to Deep Learning by Prof. Bhiksha Raj (2017)

Videos and Lectures

1. Deep Learning, Self-Taught Learning and Unsupervised Feature Learning by Andrew Ng
2. Recent Developments in Deep Learning by Geoff Hinton
3. The Unreasonable Effectiveness of Deep Learning by Yann LeCun
4. Deep Learning of Representations by Yoshua bengio
5. Principles of Hierarchical Temporal Memory by Jeff Hawkins
6. Machine Learning Discussion Group - Deep Learning w/ Stanford AI Lab by Adam Coates
7. Making Sense of the World with Deep Learning by Adam Coates
8. Demystifying Unsupervised Feature Learning by Adam Coates
9. Visual Perception with Deep Learning by Yann LeCun
10. The Next Generation of Neural Networks by Geoffrey Hinton at GoogleTechTalks
11. The wonderful and terrifying implications of computers that can learn by Jeremy Howard at TEDxBrussels
12. Unsupervised Deep Learning - Stanford by Andrew Ng in Stanford (2011)
13. Natural Language Processing by Chris Manning at Stanford

14. A beginners Guide to Deep Neural Networks by Natalie Hammel and Lorraine Yurshansky
15. Deep Learning: Intelligence from Big Data by Steve Jurvetson (and panel) at VLAB in Stanford
16. Introduction to Artificial Neural Networks and Deep Learning by Leo Isikdogan at Motorola Mobility HQ
17. NIPS 2016 lecture and workshop videos - NIPS 2016
18. Deep Learning Crash Course: a series of mini-lectures by Leo Isikdogan on YouTube (2018)

Papers

You can also find the most cited deep learning papers here

1. ImageNet Classification with Deep Convolutional Neural Networks
2. Using Very Deep Autoencoders for Content Based Image Retrieval
3. Learning Deep Architectures for AI
4. CMU's list of papers
5. Neural Networks for Named Entity Recognition zip
6. Training tricks by YB
7. Geoff Hinton's reading list (all papers)
8. Supervised Sequence Labelling with Recurrent Neural Networks
9. Statistical Language Models based on Neural Networks
10. Training Recurrent Neural Networks
11. Recursive Deep Learning for Natural Language Processing and Computer Vision
12. Bi-directional RNN
13. LSTM
14. GRU - Gated Recurrent Unit
15. GFRNN .
16. LSTM: A Search Space Odyssey
17. A Critical Review of Recurrent Neural Networks for Sequence Learning
18. Visualizing and Understanding Recurrent Networks
19. Wojciech Zaremba, Ilya Sutskever, An Empirical Exploration of Recurrent Network Architectures
20. Recurrent Neural Network based Language Model
21. Extensions of Recurrent Neural Network Language Model
22. Recurrent Neural Network based Language Modeling in Meeting Recognition

23. Deep Neural Networks for Acoustic Modeling in Speech Recognition
24. Speech Recognition with Deep Recurrent Neural Networks
25. Reinforcement Learning Neural Turing Machines
26. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
27. Google - Sequence to Sequence Learning with Neural Networks
28. Memory Networks
29. Policy Learning with Continuous Memory States for Partially Observed Robotic Control
30. Microsoft - Jointly Modeling Embedding and Translation to Bridge Video and Language
31. Neural Turing Machines
32. Ask Me Anything: Dynamic Memory Networks for Natural Language Processing
33. Mastering the Game of Go with Deep Neural Networks and Tree Search
34. Batch Normalization
35. Residual Learning
36. Image-to-Image Translation with Conditional Adversarial Networks
37. Berkeley AI Research (BAIR) Laboratory
38. MobileNets by Google
39. Cross Audio-Visual Recognition in the Wild Using Deep Learning
40. Dynamic Routing Between Capsules
41. Matrix Capsules With Em Routing
42. Efficient BackProp

Tutorials

1. Deep Learning for NLP (without Magic)
2. A Deep Learning Tutorial: From Perceptrons to Deep Networks
3. Deep Learning from the Bottom up
4. Theano Tutorial
5. Neural Networks for Matlab
6. Using convolutional neural nets to detect facial keypoints tutorial
7. Torch7 Tutorials
8. The Best Machine Learning Tutorials On The Web
9. VGG Convolutional Neural Networks Practical
10. TensorFlow tutorials

11. More TensorFlow tutorials
12. TensorFlow Python Notebooks
13. Keras and Lasagne Deep Learning Tutorials
14. Classification on raw time series in TensorFlow with a LSTM RNN
15. Using convolutional neural nets to detect facial keypoints tutorial
16. TensorFlow-World
17. Deep Learning with Python
18. Grokking Deep Learning
19. Deep Learning for Search
20. Keras Tutorial: Content Based Image Retrieval Using a Convolutional Denoising Autoencoder
21. Pytorch Tutorial by Yunjey Choi

Websites

1. deeplearning.net
2. deeplearning.stanford.edu
3. nlp.stanford.edu
4. ai-junkie.com
5. cs.brown.edu/research/ai
6. eecs.umich.edu/ai
7. cs.utexas.edu/users/ai-lab
8. cs.washington.edu/research/ai
9. aiai.ed.ac.uk
10. www-aig.jpl.nasa.gov
11. csail.mit.edu
12. cgi.cse.unsw.edu.au/~aishare
13. cs.rochester.edu/research/ai
14. ai.sri.com
15. isi.edu/AI/isd.htm
16. nrl.navy.mil/itd/aic
17. hips.seas.harvard.edu
18. AI Weekly
19. stat.ucla.edu

20. deeplearning.cs.toronto.edu
21. jeffdonahue.com/lrcn/
22. visualqa.org
23. www.mpi-inf.mpg.de/departments/computer-vision...
24. Deep Learning News
25. Machine Learning is Fun! Adam Geitgey's Blog
26. Guide to Machine Learning
27. Deep Learning for Beginners

Datasets

1. MNIST Handwritten digits
2. Google House Numbers from street view
3. CIFAR-10 and CIFAR-100
4. IMAGENET
5. Tiny Images 80 Million tiny images6.
6. Flickr Data 100 Million Yahoo dataset
7. Berkeley Segmentation Dataset 500
8. UC Irvine Machine Learning Repository
9. Flickr 8k
10. Flickr 30k
11. Microsoft COCO
12. VQA
13. Image QA
14. AT&T Laboratories Cambridge face database
15. AVHRR Pathfinder

Conferences

1. CVPR - IEEE Conference on Computer Vision and Pattern Recognition
2. AAMAS - International Joint Conference on Autonomous Agents and Multiagent Systems
3. IJCAI - International Joint Conference on Artificial Intelligence
4. ICML - International Conference on Machine Learning
5. ECML - European Conference on Machine Learning
6. KDD - Knowledge Discovery and Data Mining

7. NIPS - Neural Information Processing Systems
8. O'Reilly AI Conference - O'Reilly Artificial Intelligence Conference
9. ICDM - International Conference on Data Mining
10. ICCV - International Conference on Computer Vision
11. AAAI - Association for the Advancement of Artificial Intelligence

Frameworks

1. Caffe
2. Torch7
3. Theano
4. cuda-convnet
5. convetjs
6. Ccv
7. NuPIC
8. DeepLearning4J
9. Brain
10. DeepLearnToolbox
11. Deepnet
12. Deeppy
13. JavaNN
14. hebel
15. Mocha.jl
16. OpenDL
17. cuDNN
18. MGL
19. Knet.jl
20. Nvidia DIGITS - a web app based on Caffe
21. Neon - Python based Deep Learning Framework
22. Keras - Theano based Deep Learning Library
23. Chainer - A flexible framework of neural networks for deep learning
24. RNNLM Toolkit
25. RNNLIB - A recurrent neural network library
26. char-rnn

27. MatConvNet: CNNs for MATLAB
28. Minerva - a fast and flexible tool for deep learning on multi-GPU
29. Brainstorm - Fast, flexible and fun neural networks.
30. Tensorflow - Open source software library for numerical computation using data flow graphs
31. DMTK - Microsoft Distributed Machine Learning Tookit
32. Scikit Flow - Simplified interface for TensorFlow (mimicking Scikit Learn)
33. MXnet - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning framework
34. Veles - Samsung Distributed machine learning platform
35. Marvin - A Minimalist GPU-only N-Dimensional ConvNets Framework
36. Apache SINGA - A General Distributed Deep Learning Platform
37. DSSTNE - Amazon's library for building Deep Learning models
38. SyntaxNet - Google's syntactic parser - A TensorFlow dependency library
39. mlpack - A scalable Machine Learning library
40. Torchnet - Torch based Deep Learning Library
41. Paddle - PArallel Distributed Deep LEarning by Baidu
42. NeuPy - Theano based Python library for ANN and Deep Learning
43. Lasagne - a lightweight library to build and train neural networks in Theano
44. nolearn - wrappers and abstractions around existing neural network libraries, most notably Lasagne
45. Sonnet - a library for constructing neural networks by Google's DeepMind
46. PyTorch - Tensors and Dynamic neural networks in Python with strong GPU acceleration
47. CNTK - Microsoft Cognitive Toolkit
48. Serpent.AI - Game agent framework: Use any video game as a deep learning sandbox
49. Caffe2 - A New Lightweight, Modular, and Scalable Deep Learning Framework
50. deeplearn.js - Hardware-accelerated deep learning and linear algebra (NumPy) library for the web