

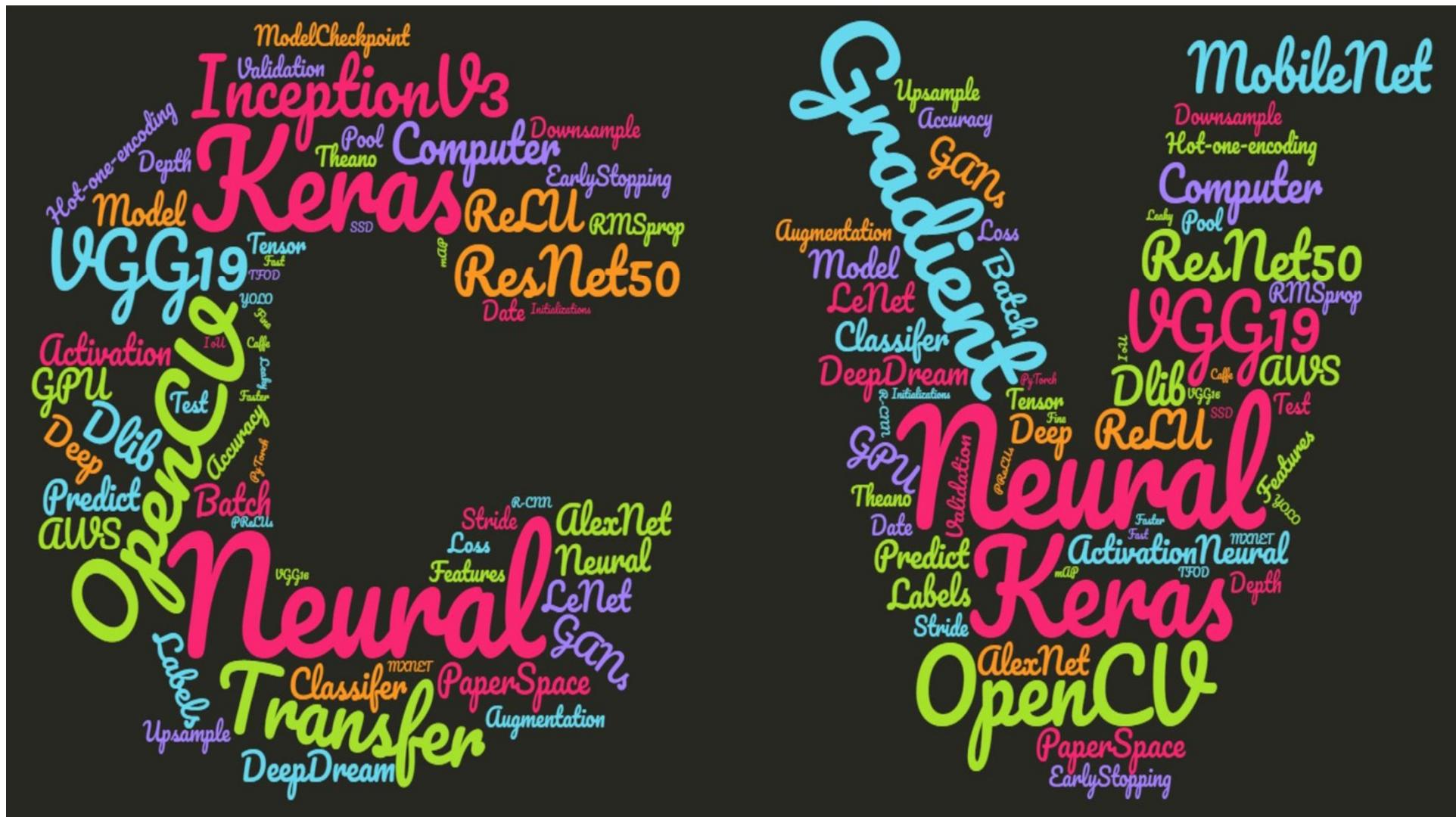


Deep Learning Computer Vision

1.0

Course Introduction

Why you'll love this course!



A word cloud visualization centered around deep learning concepts. The most prominent word is "Neural", rendered in large, bold, pink font. Surrounding it are various other terms in different colors and sizes, representing related concepts:

- Large green words include "Convolutional", "Gradient", "Batch", "Test", "Labels", "Dlib", "VGG16", "VGG19", "Keras", "Predict", "Neural", "Accuracy", "Backpropagation", "Model", "mAP", and "Faster".
- Medium-sized pink words include "Transfer", "SSD", "R-CNN", "ReLU", "Deep", "Computer", "Activation", "Classifier", "AlexNet", "LeNet", "Date", "Theano", and "mAP".
- Smaller blue words include "aws", "gan", "Pool", "Fine Loss", "Predict", "Neural", "Accuracy", "Backpropagation", "Model", and "mAP".

The image is a word cloud centered around deep learning and computer vision. The most prominent word is "Neural" in a large, bold, pink font. Surrounding it are other words such as "Convolutional" (large, green), "ReLU" (medium, light blue), "Caffe" (small, orange), "Tensor" (medium, purple), "Dlib" (medium, yellow), "Keras" (medium, green), "PyTorch" (medium, blue), "Fast" (small, red), "Labels" (small, blue), "Activation" (small, red), "Theano" (small, blue), "Leaky" (small, red), "Loss" (small, blue), "Batch" (small, red), "Test" (small, blue), "Date" (small, blue), "Gradient" (large, green), "Deep" (small, red), "Computer" (small, blue), "Depth" (small, red), "Pool" (small, blue), "Predict" (small, red), "Gan" (small, green), "Faster" (small, red), "LeNet" (small, blue), "All" (small, blue), "TFOD" (small, blue), "Model" (small, blue), "IoU" (small, blue), "Neural" (small, blue), "Space" (small, blue), "DeepDream" (small, blue), "Paper" (small, blue), and "Space" (small, blue). The words are in various colors including pink, green, blue, red, and orange, and are arranged in a circular, radiating pattern from the center.

Deep Learning is Revolutionizing the World!

- It is enabling machines to learning complicated tasks, tasks that were **thought impossible a mere 5 years ago!**
- The combination of increasing computing speed, wealth of research and the rapid growth of technology, Deep Learning and AI is experiencing **MASSIVE** growth world wide and will perhaps be the one of the world's biggest industries in the near future.
- The 21st century is the birth of AI Revolution, and **data** becoming the new 'oil'
- The Computer Vision industry alone will be worth **17.38 Billion USD** (12B at 2018) by 2023 <https://www.marketsandmarkets.com/PressReleases/computer-vision.asp>
- Good Computer Vision Scientists are paid between **\$400 to \$1000 USD** a day as demand far out strips supply

But What Exactly Is Deep Learning? And Computer Vision?

- Deep Learning is a machine learning technique that enables machines to **learn complicated patterns or representations in data**.
- In the past, obtaining good results using machine learning algorithms required a lot of **tedious manual fine tuning of data features** by data scientists, and results on complicated data were typically poor.
- So what is Deep Learning? Let's pretend we have an **untrained 'brain'**, capable of learning, but with no prior knowledge.
- Imagine, we show this brain thousands of pictures of cats and then thousands more of dogs. After '**teaching' or 'training**' this brain, it's now able to tell the difference between images of cats or dogs. That is effectively what Deep Learning is, an algorithm that facilitates learning.





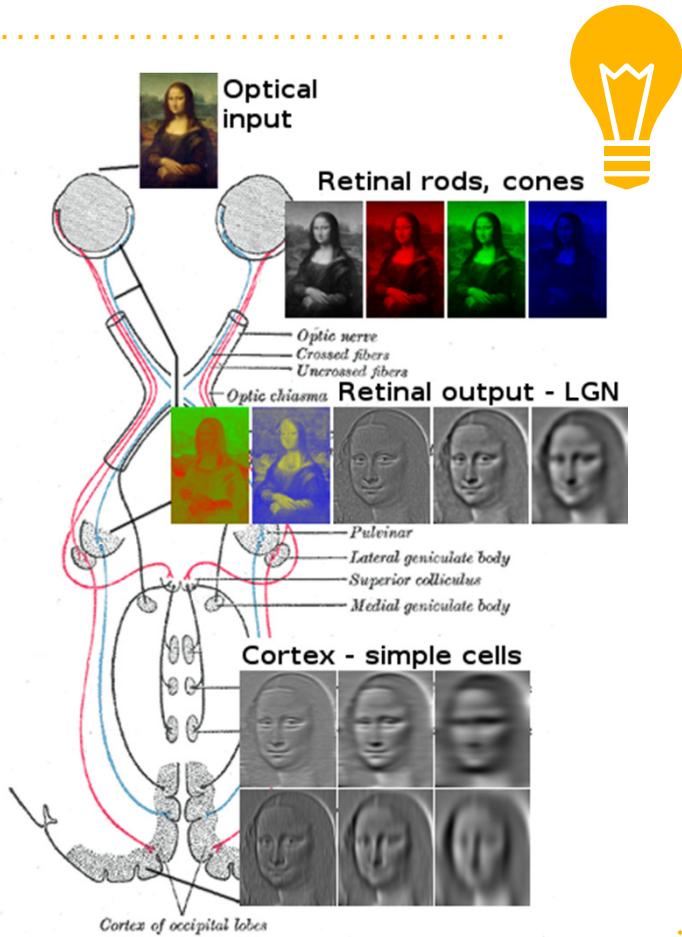
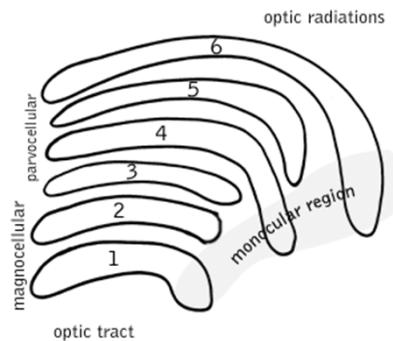
Computer Vision is perhaps the most important part of the AI Dream



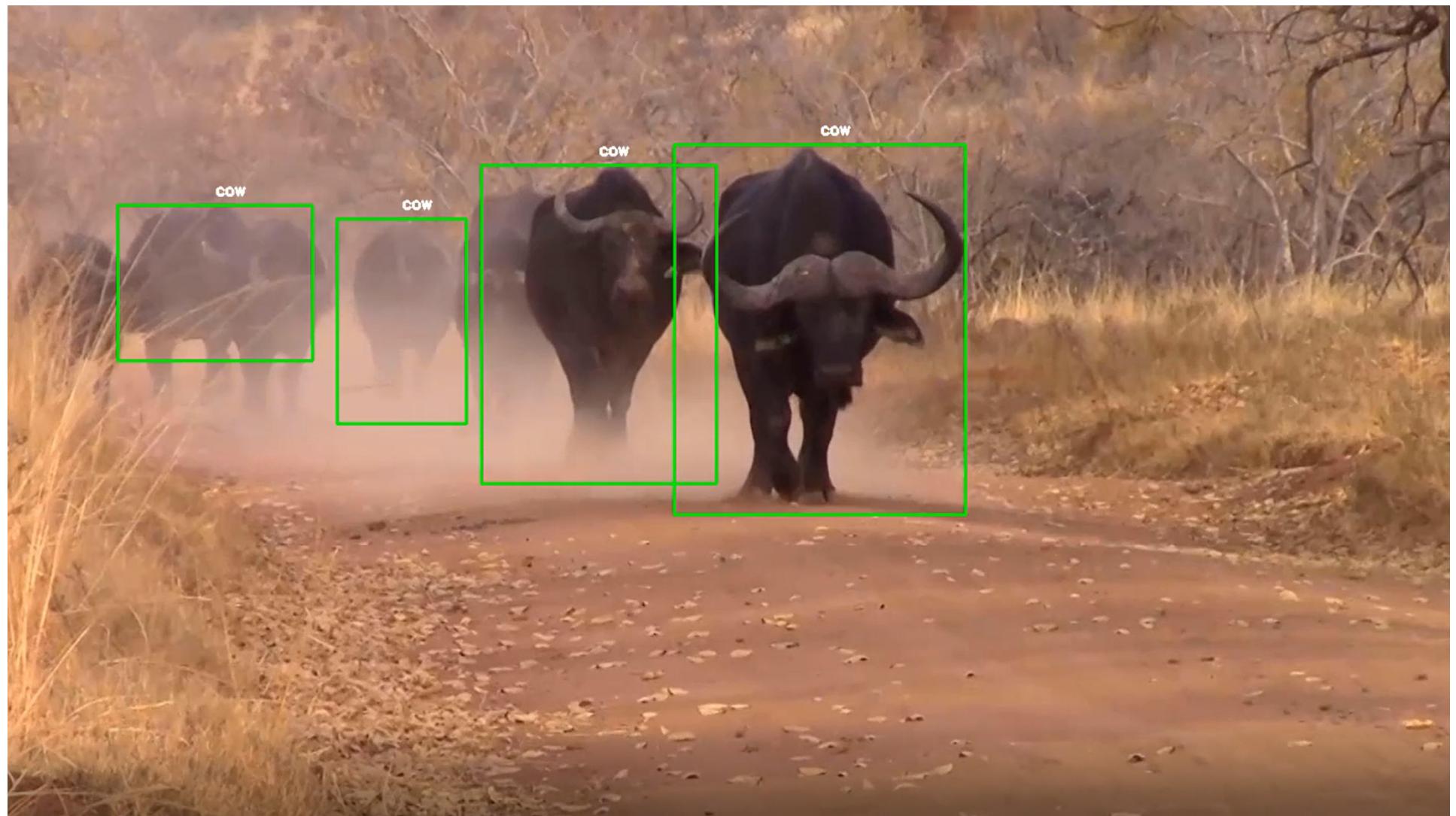
Source: *Terminator 2: Judgement Day*

Machines that can understand what they see WILL change the world!

We take for Granted How Amazing Our **Brain** is at Visual Processing

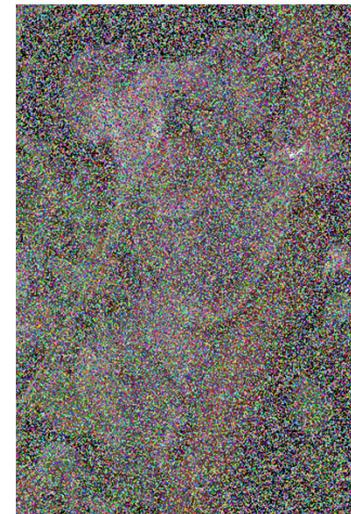


https://en.wikipedia.org/wiki/Visual_system



Robots or Machines that can see will be capable of...

- Autonomous Vehicles (self driving cars)
- Robotic Surgery & Medical Diagnostics
- Robots that can navigate our clutter world
 - Robotic chefs, farmers, assistants etc.
- Intelligent Surveillance and Drones
- Creation of Art
- Improved Image & Video Searching
- Social & Fun Applications
- Improve Photography





Who Should Take This Course?

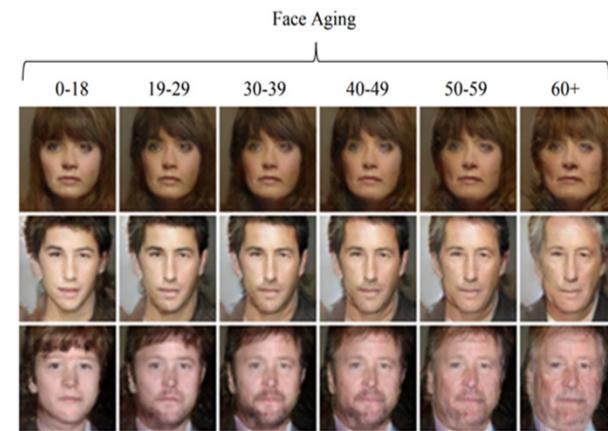
**This Course is
for you!**

What is this course about?



Applying **Deep Learning** to **Computer Vision Tasks** such as:

- Image Classification (10 Projects)
- Segmentation (1 Project)
- Object Detection (3 Projects)
- Neural Style Transfers (2 Projects)
- Generative Networks (2 Projects)



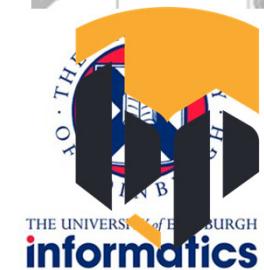
Hi

I'm Rajeev Ratan



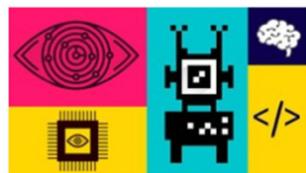
About me

- **Radio Frequency Engineer** in Trinidad & Tobago for 8 years (2006-2014). I was the Manager of Wireless Engineering and lead Radio Engineer on a number of Projects.
- **University of Edinburgh – MSc in Artificial Intelligence** (2014-2015) where I specialized in Robotics and Machine Learning
- **Entrepreneur First (EF6) London Startup Incubator** (2016)
 - CTO at Edtech Startup, Dozenal (2016)
 - VP of Engineering at KeyPla (CV Real Estate Startup) (2016-2017)
 - Head of CV at BlinkPool (eGames Gambling Startup) (2017-present)
- **Created the Udemy Course** – Mastering OpenCV in Python (12,000+ students)





My OpenCV Python Course



Master Computer Vision™ OpenCV3 in Python & Machine Learning

65 lectures • 6.5 hours • All Levels

Learn **Computer Vision** Concepts by making 12 Projects like Handwriting Recognition, Face Filters, Car & People Detection! | By Rajeev Ratan

€10.99
€199.99

★★★★★ 4.1
(2,166 ratings)



2 months ago
Sarah Shelley



I loved this course. Very detailed explanations and in depth with the latest technology and ideas. Very thoughtful help with ideas of implementation of course materials for real life projects. Thanks for a great insightful course. I'm looking forward to using the learnt material. Thank you.



4 months ago
Zdenko Nevrala



It is really straightforward. Nice basics explaining with links for extension of topic knowledge. The exercising is effective and cool. I am really appreciate there are NOT boring parts.



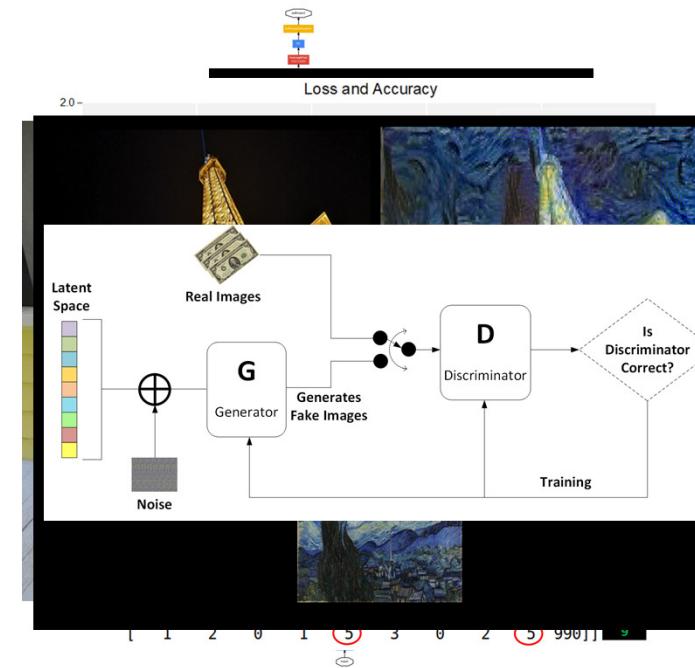
7 months ago
Kevin Kinser



I'm amazed at the possibilities. Very educational, learning more than what I ever thought was possible. Now, being able to actually use it in a practical purpose is intriguing... much more to learn & apply.

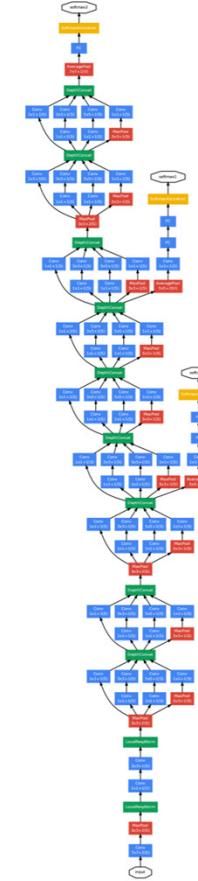
What you'll be able to do after

- Create and train your very own **Image Classifiers** using a range of techniques such as:
 - Using a variety of CNNs (ResNet50, VGG, InceptionV3 etc, and best of all, designing your own CNNs and getting amazing results quickly and efficiently)
 - Transfer Learning and Fine Tuning
 - Analyze and Improve Performance
 - Advanced Concepts (Callbacks, Making Datasets, Annotating Images, Visualizations, ImageNet, Understanding Features, Advanced Activations, & Initializations, Batch Normalization, Dropout, Data Augmentation etc.)
- Understand how to use CNNs in **Image Segmentation**
- **Object Detection** using R-CNNs, Faster R-CNNs, SSD and YOLO and Make your own!
- **Neural Style Transfers** and **Deep Dreaming**
- **Generative Adversarial Networks** (GANs)



I made this course because...

- Computer Vision is **continuously evolving** and this fast moving field is **EXTRE** hard for a beginner to know even where to begin.
 - I use Python and Keras with the TensorFlow backend, all of which are k the most popular methods to apply to Deep Learning Computer Vision are well suited to beginners.
- While there are many good tutorials out there, many use outdated code and provide broken setup and installation instructions
- Too many tutorials get too theoretical too fast, or are too simplistic and teach without explaining any of the key concepts.





Requirements

- **Little to no programming knowledge**
 - Familiarity with any language helps, especially Python but it is **NOT** a prerequisite for this course
- **High School Level Math**
 - College level math in Linear Algebra and Calculus will help you develop further, but it's not at all necessary for learning Deep Learning.
- **A moderately fast laptop or PC (no GPU required!)**
 - At least 20GB of HD space (for your virtual machine and datasets)



What you're getting

- 600+ Slides
- 15 hours of video including an optional 3 hour OpenCV 3 Tutorial
- Comprehensive Deep Learning Computer Vision Learning Path using Python with Keras (TensorFlow backend).
- Exposure to the latest Deep Learning Techniques as of late 2018
- A **FREE** VirtualBox development machine with all required libraries installed (this will save you hours of installation and troubleshooting!)
- 50+ ipython notebooks
- 18+ Amazing Deep Learning Projects



Course Outline

- Computer Vision and Deep Learning Introductions
- Install and Setup
- Start with experimenting with some Python code using Keras and OpenCV
- Learn all about Neural Networks and Convolution Neural Networks
- Start building CNN Classifiers with learning progressively more advanced techniques
- Image Segmentation using U-Net
- Object Detection with TFOD (R-CNNs and SSDs) and YOLO
- Deep Dream & Neural Style Transfer
- Generative Adversarial Networks (GANs)
- Computer Vision Overview

2.0

Introduction to Computer Vision & Deep Learning

Overview of Computer Vision and Deep Learning



Learning Summary for Intro to Computer Vision and Deep Learning

- **2.1 – What is Computer Vision and What Makes it Hard**
- **2.2 – What are Images?**
- **2.3 – Intro to OpenCV, OpenVINO™ & their Limitations**

2.1

What is Computer Vision & What Makes It Hard

“

*“Computer Vision is a cutting edge field of Computer Science that aims to enable computers to understand **what is being seen in an image**”*



Computer Vision is an incredibly interesting field that is undergoing remarkable growth and success in the last 5 years. But is still **VERY hard!**

- Computers don't '**see**' like we do.
- Our brains are incredibly good at understanding the visual information received by our eyes.
- Computers however, 'see' like this.....

A stream of raw numbers from a camera sensor, typically a grid/array of color intensities of BGR (Blue, Green, Red)

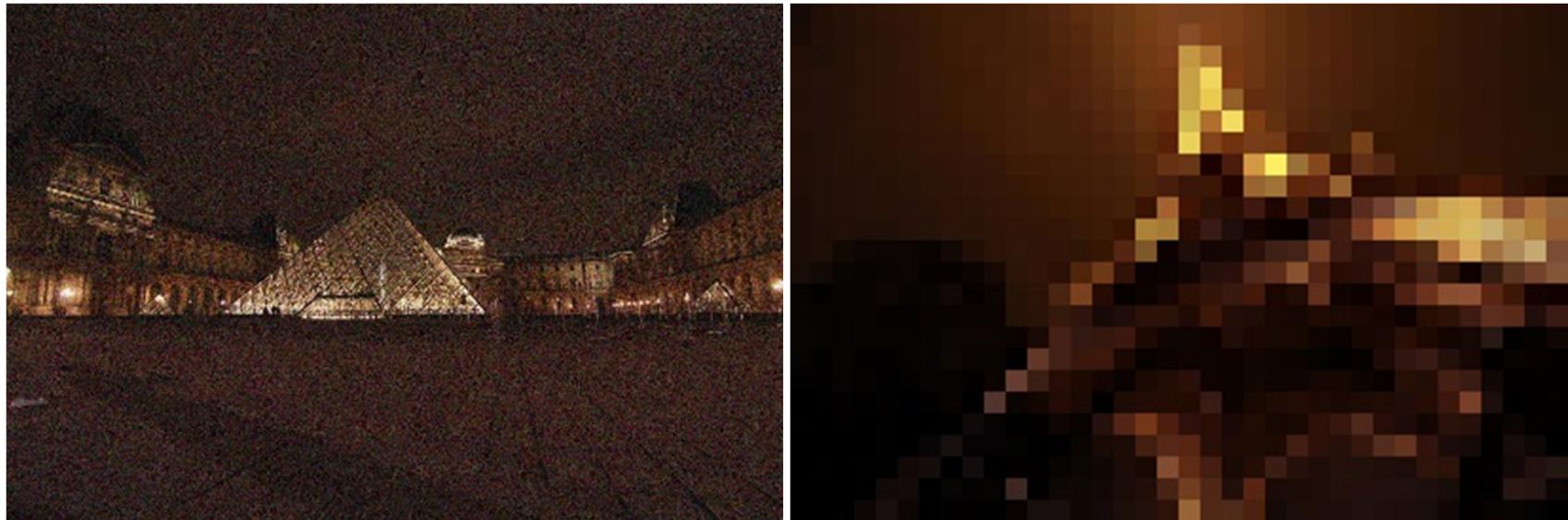


**And there are
other reasons
it's so hard**



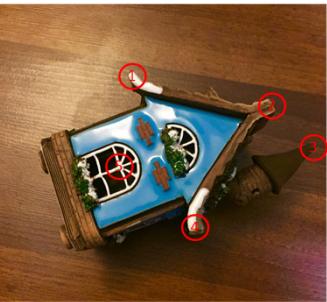


Camera sensors and lens limitations



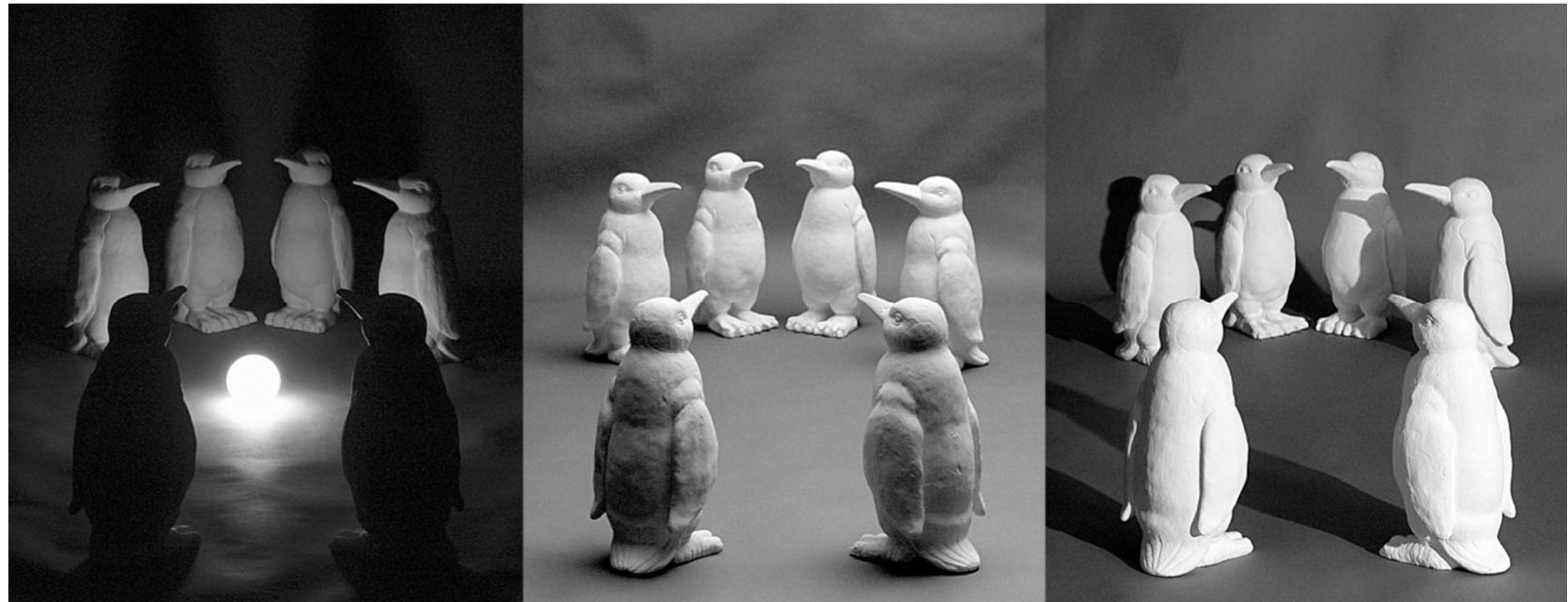


Viewpoint variations





Changing Lighting Conditions

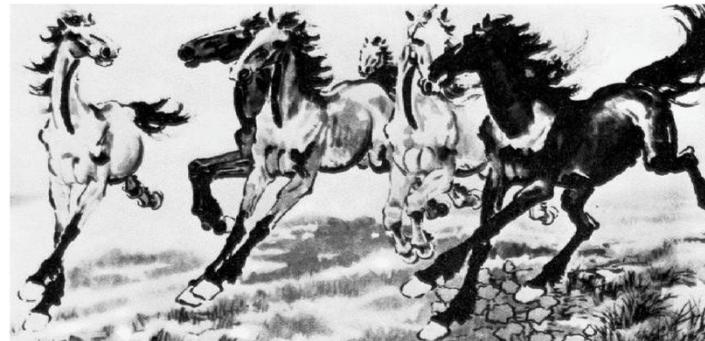


Issues of Scale



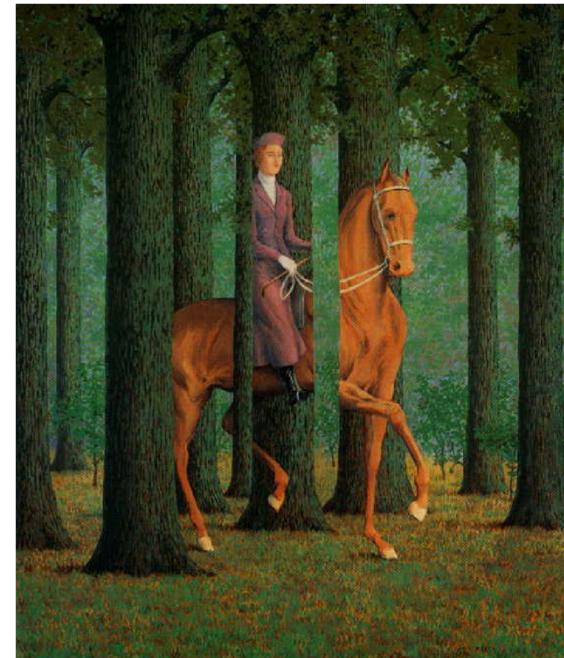


Non-rigid Deformations





Occlusion – Partially Hidden

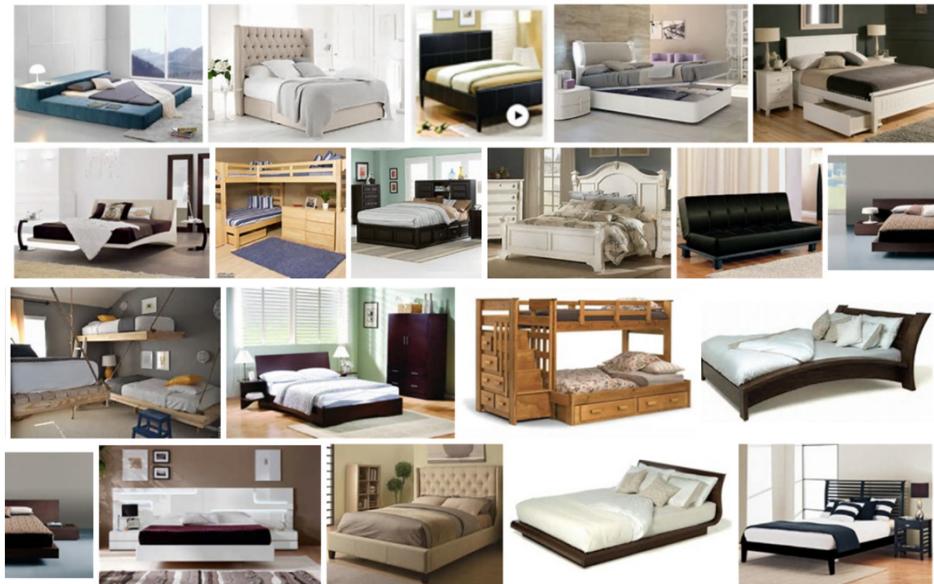


Clutter



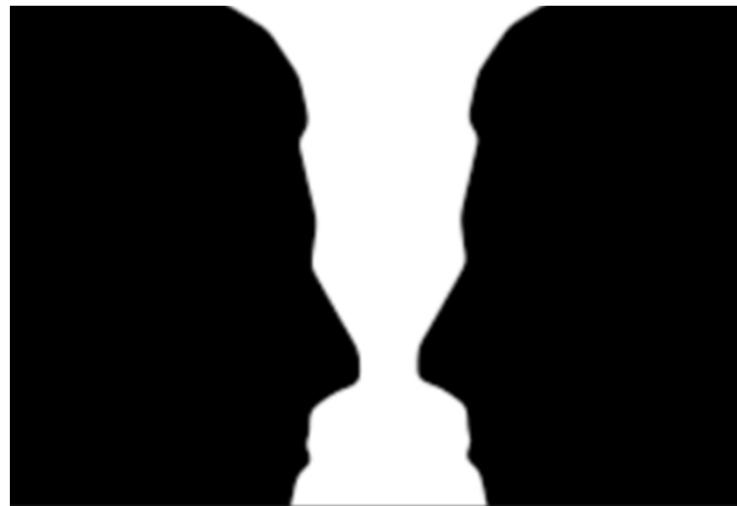


Object Class Variations





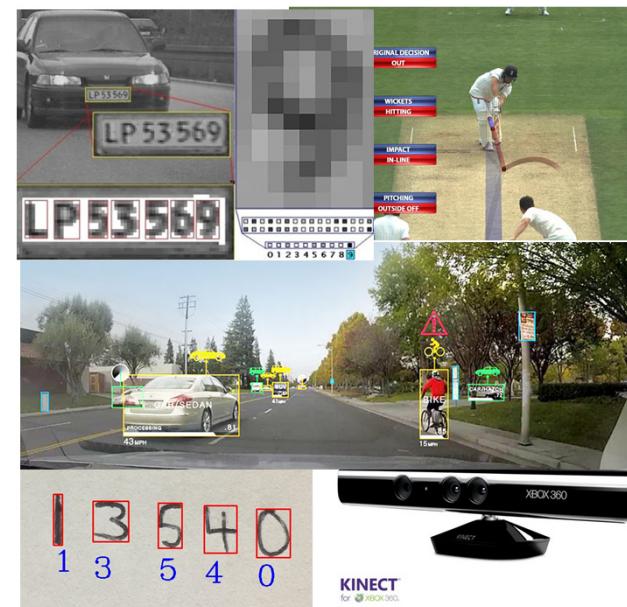
Ambiguous Optical Illusions





And yet, despite this, Computer Vision has had a lot of success stories

- Handwriting Recognition
- Image Recognition of over 1000 classes
- License Plate Readers
- Facial Recognition
- Self Driving Cars
- Hawkeye and CV Referrals in Sports
- Robotic Navigation
- SnapChat Filters



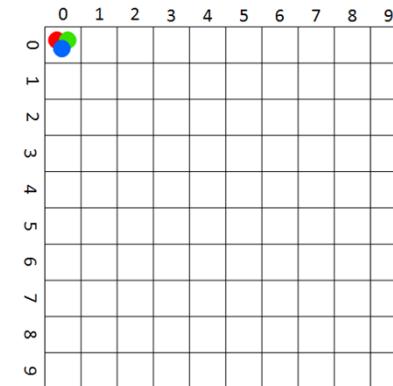
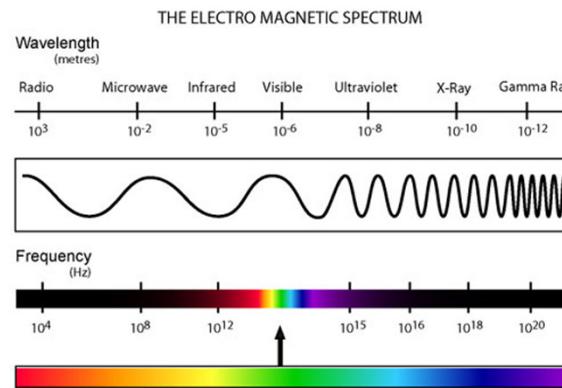
2.2

What are Images?



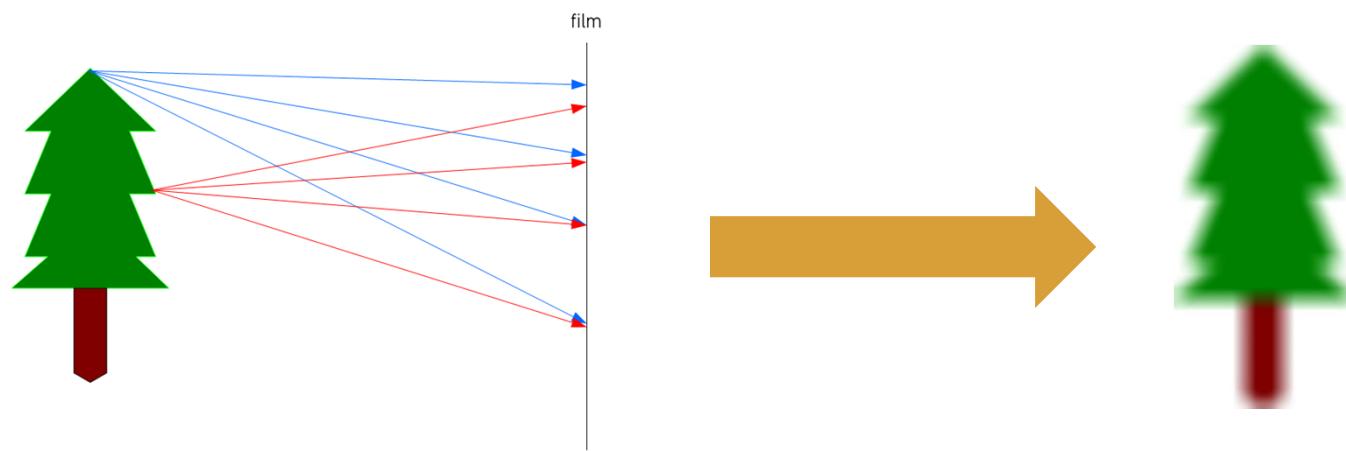
What is exactly are images?

- **2-Dimensional** representation of the **visible light spectrum**



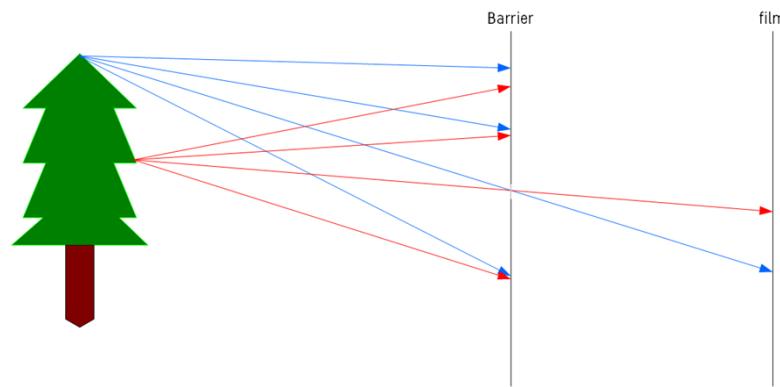


How are Images Formed?



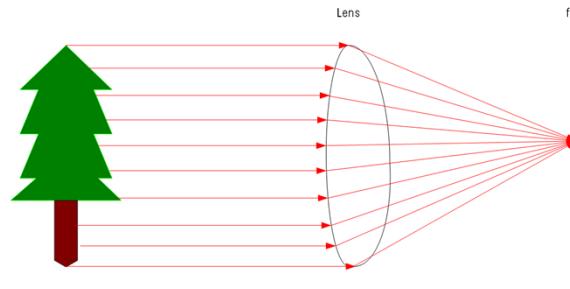
- When light reflects off the surface of an object and onto film, light sensors or our retinas.
- In this example, the image will appear blurred

Image Formation



- Using a **small opening** in the barrier (called **aperture**), we block off most of the rays of light reducing blurring on the film or sensor
- This is the **pinhole camera** model

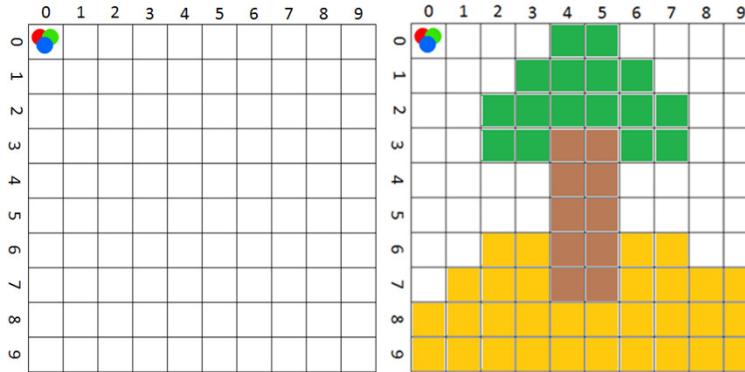
Image Formation



- Both our eyes and cameras use an adaptive lens to control many aspects of image formation such as:
 - **Aperture Size**
 - Controls the amount of light allowed through (f-stops in cameras)
 - Depth of Field (Bokeh)
 - **Lens width** - Adjusts focus distance (near or far)

How are images seen Digitally?

- Computers store images in a variety of formats, however they all involve mixing some of the following such as: colors, hues, brightness or saturation.
- The one most commonly used in Computer Vision is BGR (Blue, Green, Red)



- Each pixel is a combination of the brightness of blue, green and red, ranging from 0 to 255 (brightest)
- Yellow is represented as:
 - Red – 255
 - Green – 255
 - Blue – 0

This is essentially how an image is stored on a computer

Images are stored in Multi-Dimensional Arrays

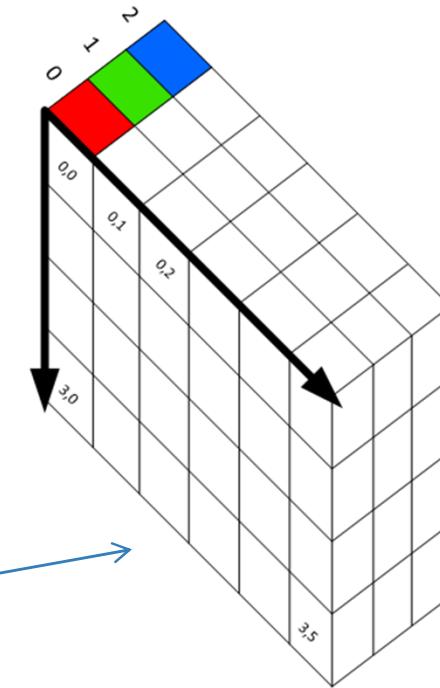
- Think of an array as a table. A 1-Dim array looks like this:

0	1	2	3	4	5	6	7

- A 2-Dim array looks like this:

0	1	2	3	4	5	6	7

- A 3-Dim array looks like this



A Grayscale Image only needs 2-Dims to be Represented

- Black and White images are stored in 2-Dimensional arrays.

2.3

Intro to OpenCV, OpenVINO™ & their Limitations



About OpenCV

- Officially launched in 1999, OpenCV (Open Source Computer Vision) from an Intel initiative.
- OpenCV's core is written in C++. In python we are simply using a wrapper that executes C++ code inside of python.
- First major release 1.0 was in 2006, second in 2009, third in 2015 and 4th in 2018. with OpenCV 4.0 Beta.
- It is an Open source library containing over 2500 optimized algorithms.
- It is EXTREMELY useful for almost all computer vision applications and is supported on Windows, Linux, MacOS, Android, iOS with bindings to Python, Java and Matlab.

<https://www.opencv.org>

The screenshot shows the official OpenCV website at <https://www.opencv.org>. The page features a header with the OpenCV logo and navigation links for About, News, Events, Releases, Platforms, Books, Links, and License. Below the header is a main content area with text about the library's history and usage. To the right is a sidebar titled "Quick Links" containing links to online documentation, tutorials, user forums, bug reports, build farms, developer sites, and a wiki. A "Donate" button is also present. At the bottom, there's a "Latest news" section with links to recent releases and events, and a footer with social media icons for E-Mail, Slack, GitHub, Facebook, Google+, Wikipedia, Forum, IRC, SourceForge, Twitter, and YouTube. The copyright notice at the bottom reads "© Copyright 2018, OpenCV team".

OpenCV (Open Source Computer Vision Library) is released under a BSD license and hence it's free for both academic and commercial use. It has C++, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. Written in optimized C/C++, the library can take advantage of multi-core processing. Enabled with OpenCL, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform.

Adopted all around the world, OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 14 million. Usage ranges from interactive art, to mines inspection, stitching maps on the web or through advanced robotics.

Latest news

New OpenCV 4.0 Beta & OpenVINO™ toolkit Oct 16, 2018	OpenCV 4.0-alpha Sep 20, 2018	AI DevCon 2018 May 11, 2018	OpenCV 3.4.1 Feb 27, 2018
OpenCV 4.0-beta and OpenVINO toolkit components have been released	OpenCV 4.0 alpha is out!	On May 23-24th Intel runs AI DevCon 2018 – IA- and CV-centric conference with lots of interesting material and some real demos.	We are glad to present the first 2018 release of OpenCV, v3.4.1, with further improved DNN module and many other improvements and bug fixes.

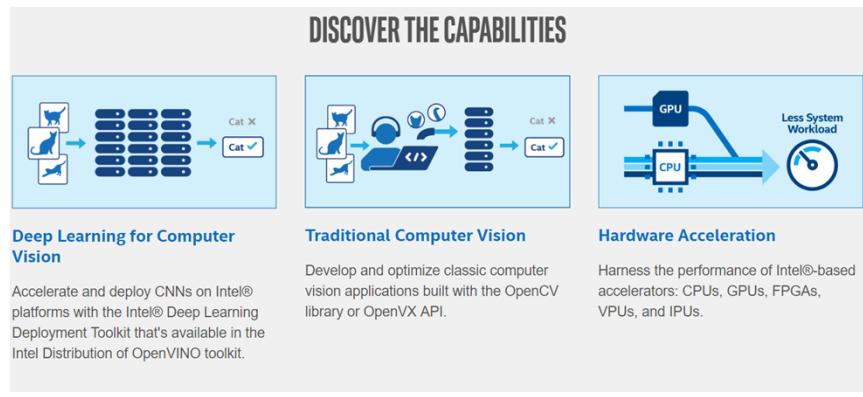
 [E-Mail](#)  [Slack](#)  [GitHub](#)  [Facebook](#)  [Google+](#)  [Wikipedia](#)
 [Forum](#)  [IRC #opencv](#)  [SourceForge](#)  [Twitter](#)  [YouTube](#)

© Copyright 2018, OpenCV team



OpenVINO™

- In 2018, Intel® Distribution released the OpenVINO™ toolkit.
- It aims to speed up deep learning using CPU and Intel's IGPU and utilize the already established OpenCV framework
- It's an excellent initiative and will be very successful.





Disadvantages of OpenVINO™

- It's a little too late to market. TensorFlow has been around since 2015 and has become the dominant Deep Learning framework in Python.
- This means tutorials, and thousands of example code in TensorFlow will have to be ported to OpenVINO
- OpenVINO will most likely not be adopted by Professionals and Researchers because any intensive training requires powerful GPUs.
- Adoption has been slow so far
- Still in Beta

3.0

Setup Your Deep Learning Development Machine

Install and download my VMI and setup your pre-configured development machine using Virtual Box

3.0

Setup Your Deep Learning Development Machine

Install and download my VMI and setup your pre-configured development machine using Virtual Box

4.0

**Get Started! Handwriting Recognition, Simple
Object Classification OpenCV Demo**



Get Started! Handwriting Recognition, Simple Object Classification & Face Detection

- **4.1 – Experiment with a Handwriting Classifier**
- **4.2 – Experiment with a Image Classifier**
- **4.3 – OpenCV Demo – Live Sketch with Webcam**

4.1

Experiment with a Handwriting Classifier

Run a CNN to Classify Hand Written Digits (MNIST)

4.2

Experiment with a Image Classifier

Classify 10 Types of Images using the CIFAR10 Dataset

4.3

OpenCV Demo – Live Sketch with Webcam

Run a simple but fun OpenCV Demo that turns your webcam feed into a live sketch!

6.0

Neural Networks Explained



Neural Networks Explained

- **6.1 Machine Learning Overview**
- **6.2 Neural Networks Explained**
- **6.3 Forward Propagation**
- **6.4 Activation Functions**
- **6.5 Training Part 1 – Loss Functions**
- **6.6 Training Part 2 – Backpropagation and Gradient Descent**
- **6.7 Backpropagation & Learning Rates – A Worked Example**
- **6.8 Regularization, Overfitting, Generalization and Test Datasets**
- **6.9 Epochs, Iterations and Batch Sizes**
- **6.10 Measuring Performance and the Confusion Matrix**
- **6.11 Review and Best Practices**

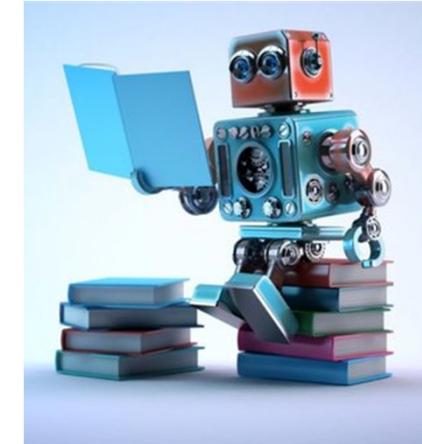
6.1

Machine Learning Overview



Machine Learning Overview

- Machine Learning is almost synonymous to Artificial Intelligence because it entails the study of how software can learn.
- It is a field of artificial intelligence that uses statistical techniques to give computer systems the ability to "learn" from data, without being explicitly programmed.
- It has seen a rapid explosion in growth in the last 5-10 years due to the combination of incredible breakthroughs with Deep Learning and the increase in training speed brought on by GPUs.





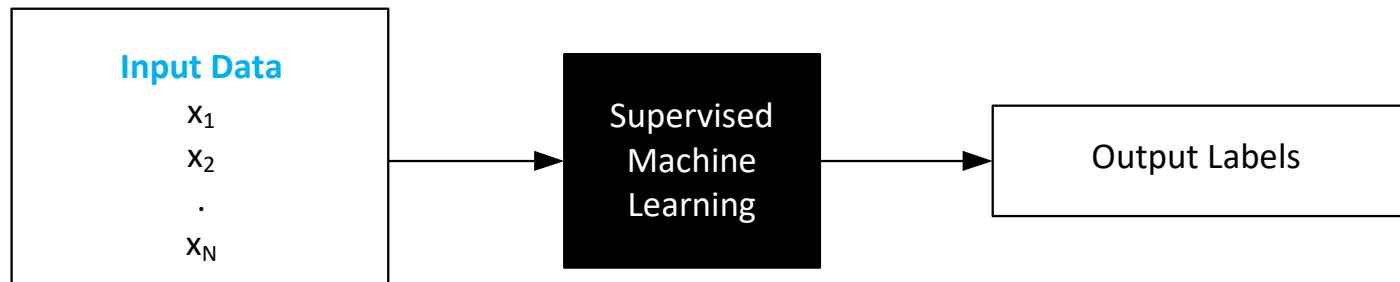
Types of Machine Learning

- There are **4 types** of Machine Learning with Neural Networks are arguably the most powerful ML algorithm. These types are:
 1. Supervised Learning
 2. Unsupervised Learning
 3. Self-supervised Learning
 4. Reinforcement Learning



Supervised Learning

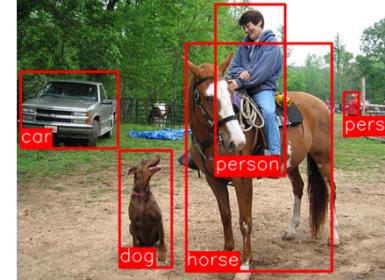
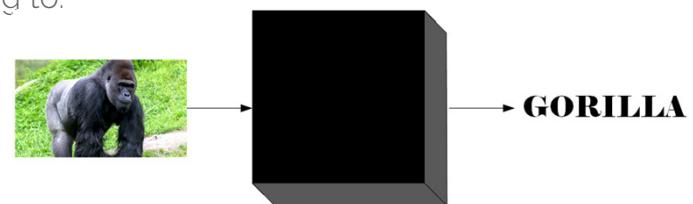
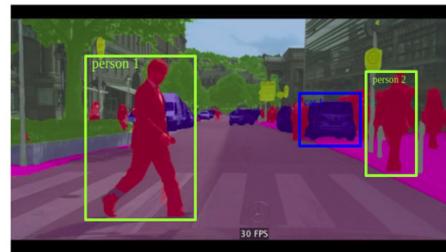
- Supervised learning is by far the most popular form of AI and ML used today.
- We take a set of labeled data (called a **dataset**) and we feed it to some ML learning algorithm that develops a model to fit this data to some outputs
- E.g. let's say we give our ML algorithm a set of 10k spam emails and 10k non spam. Our model figures out what texts or sequences of text indicate spam and thus can now be used as a spam filter in the real world.





Supervised Learning In Computer Vision

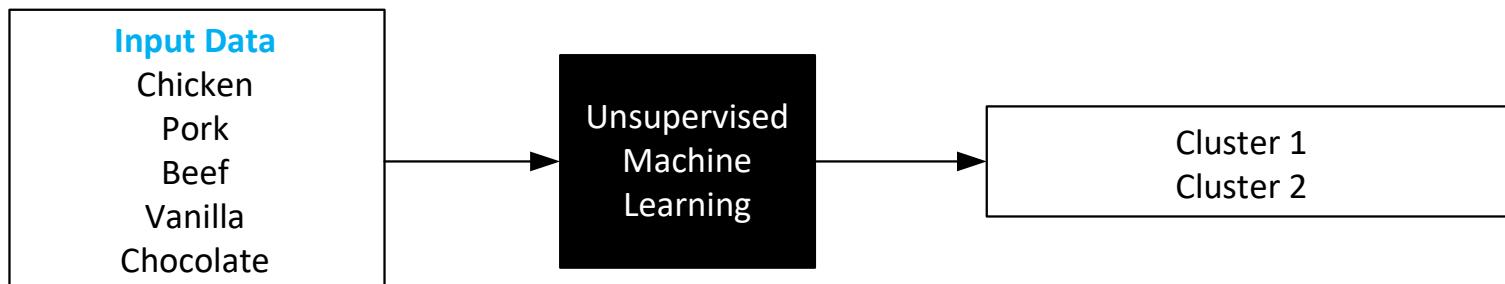
- In Computer Vision we can use Supervised Learning to:
 - Classify types of images
 - Object Detection
 - Image Segmentation





Unsupervised Learning

- Unsupervised learning is concerned with finding interesting clusters of input data. It does so without any help of data labeling.
- It does this by creating interesting transformations of the input data
- It is very important in data analytics when trying to understand data





Self-Supervised Learning

- Self-supervised learning is the same concept as supervised learning, however the data is not labeled by humans.
- How are the labels generated? Typically, it's done using a heuristic algorithm e.g. autoencoders.
- An example of this is predicting the next frame in a video given the previous frames.



Reinforcement Learning

- Reinforcement learning is potentially very interesting but hasn't become mainstream in the AI world just yet.
- The concept is simple, we teach a machine algorithm by showing them bad examples of something. It gets far more complicated with many subtypes.



ML Supervised Learning Process

- Step 1 – Obtain a labeled data set
- Step 2 – Split dataset into a training portion and validation or test portion.
- Step 3 – Fit model to training dataset
- Step 4 – Evaluate models performance on the test dataset



ML Terminology

- **Target** – Ground truth labels
- **Prediction** – The output of your trained model given some input data
- **Classes** – The categories that exist in your dataset e.g. a model that outputs Gender has two classes
- **Regression** – Refers to continuous values, e.g. a model that outputs predictions of someone's height and weight is a regression model
- **Validation/Test** – They can be different, but generally refers to unseen data that we test our trained model on.

6.2

Neural Networks Explained

The best explanation you'll ever see on Neural Networks



Why it's the Best Explanation?

- High Level Explanation of Neural Networks (NN)
- Math is made simple and introduced gradually
- Understand what makes NNs so important
- Breakdown the key elements of NNs
- Work through back propagation so you understand it well
- Walk through of Gradient Decent, Loss Functions and Activation Functions
- Understanding what goes in Training a model

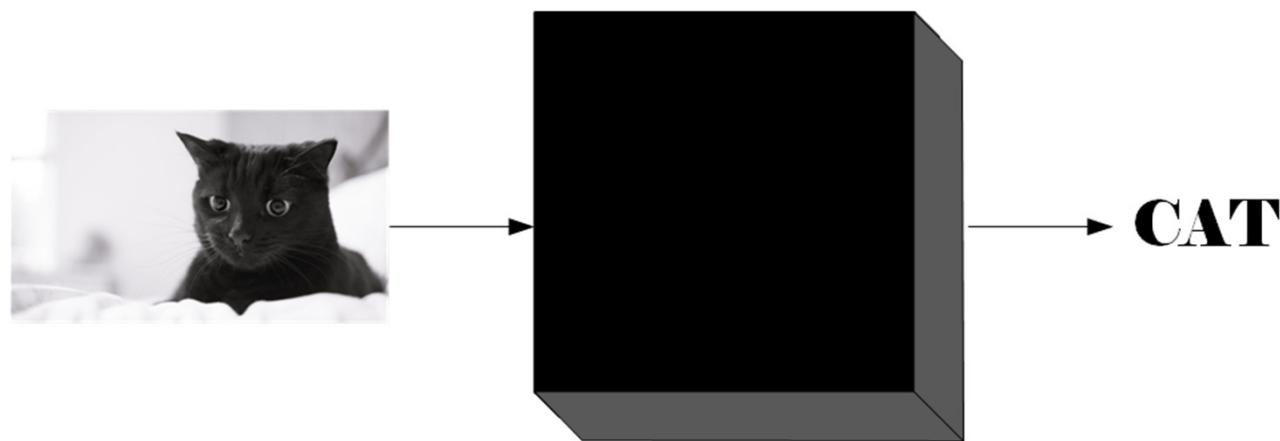


What are Neural Networks

- Neural Networks act as a 'black box' brain that takes inputs and predicts an output.
- It's different and 'better' than most traditional Machine Learning algorithms because it learns complex non-linear mappings to produce far more accurate output classification results.

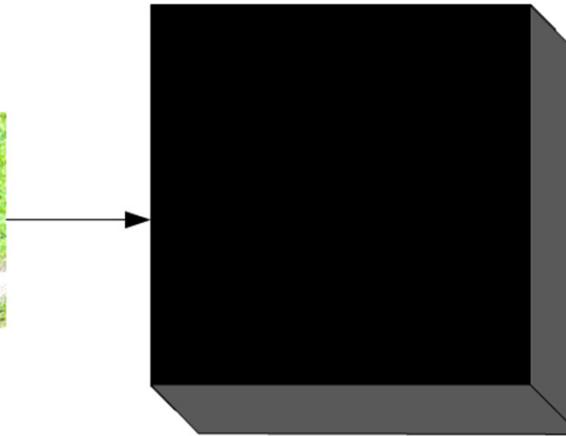


The Mysterious 'Black Box' Brain





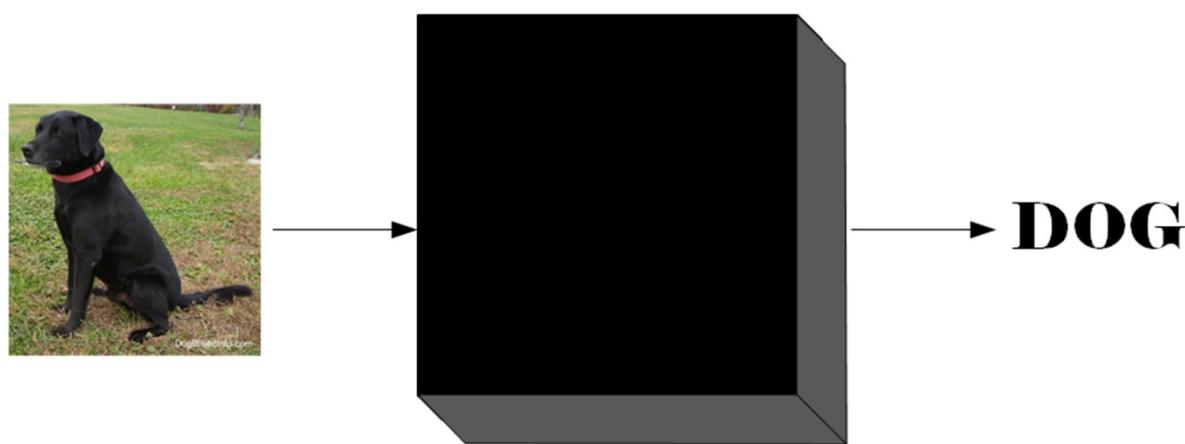
The Mysterious 'Black Box' Brain



GORILLA



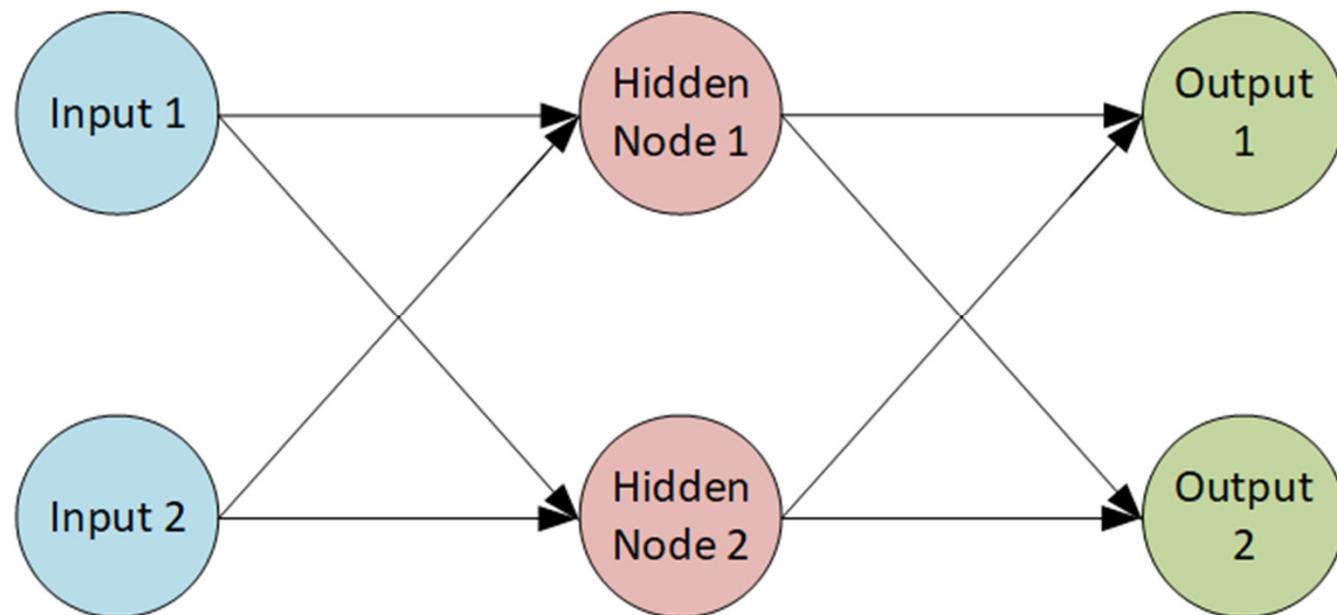
The Mysterious 'Black Box' Brain





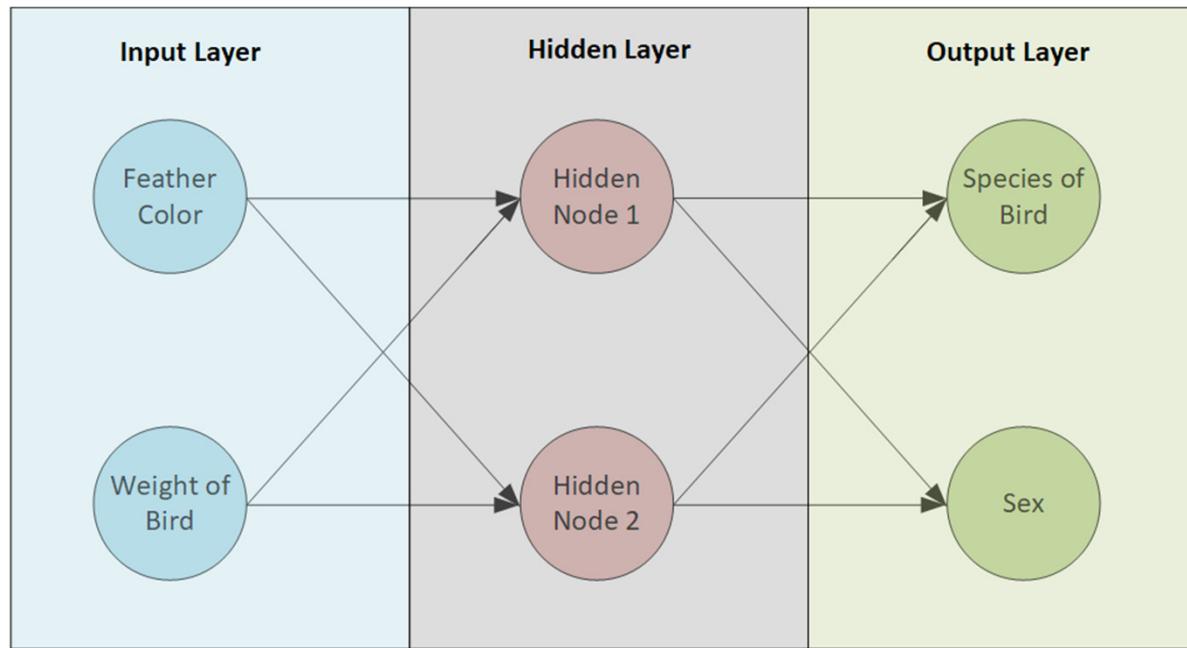
How NNs 'look'

A Simple Neural Network with 1 hidden layer





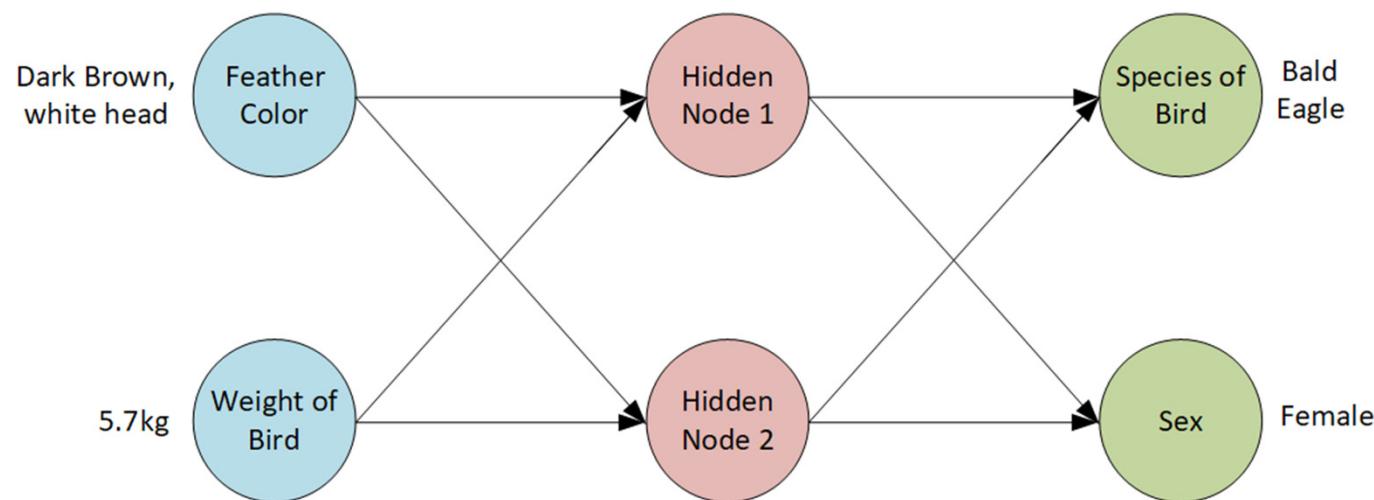
Example Neural Network





How do we get a prediction?

Pass the inputs into our NN to receive an output





How we predict example 2

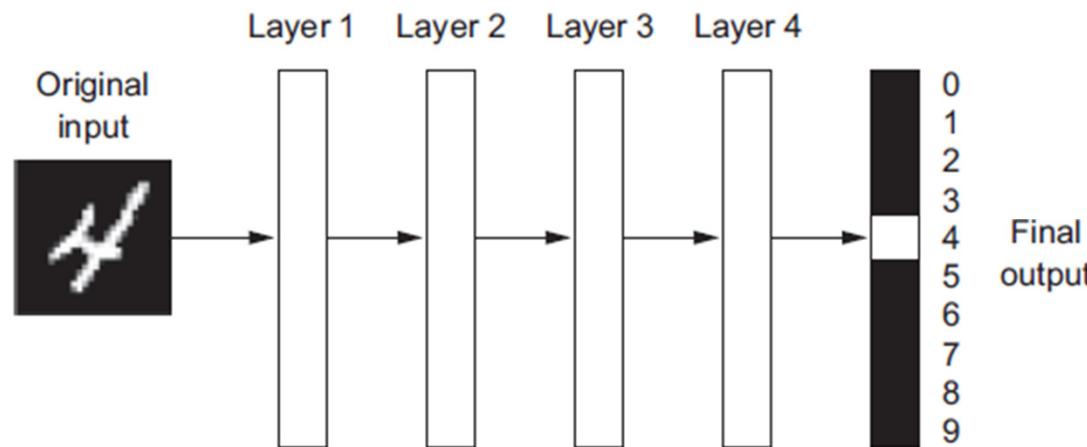


Figure 1.5 A deep neural network for digit classification

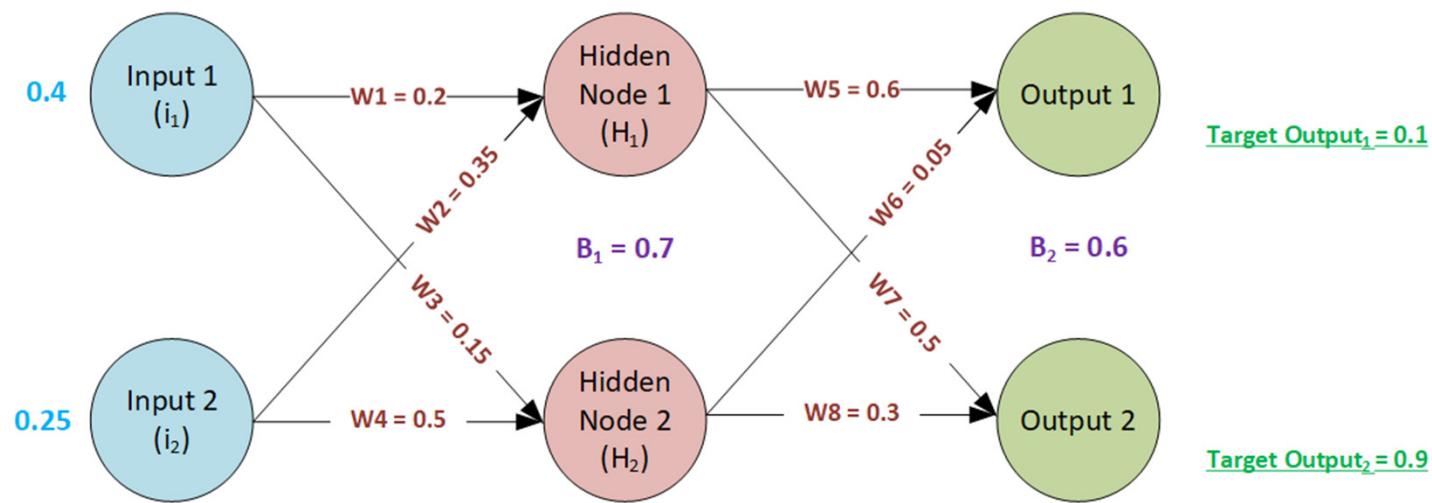
6.3

Forward Propagation

How Neural Networks process their inputs to produce an output

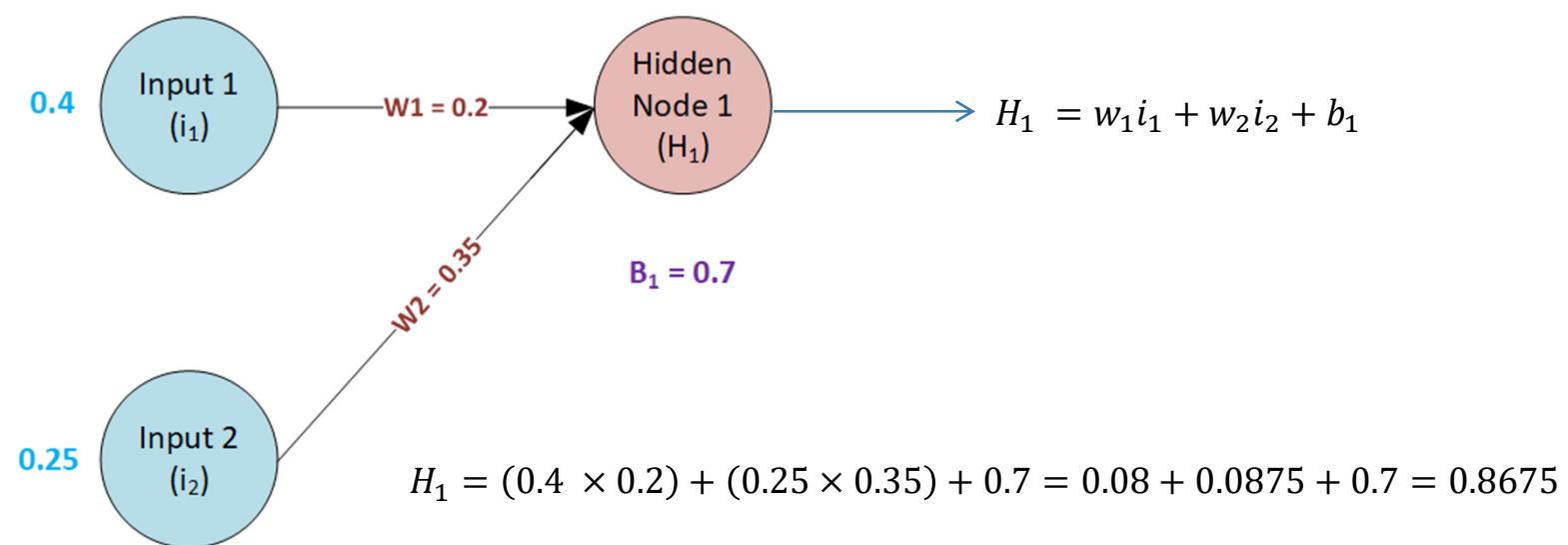


Using actual values (random)



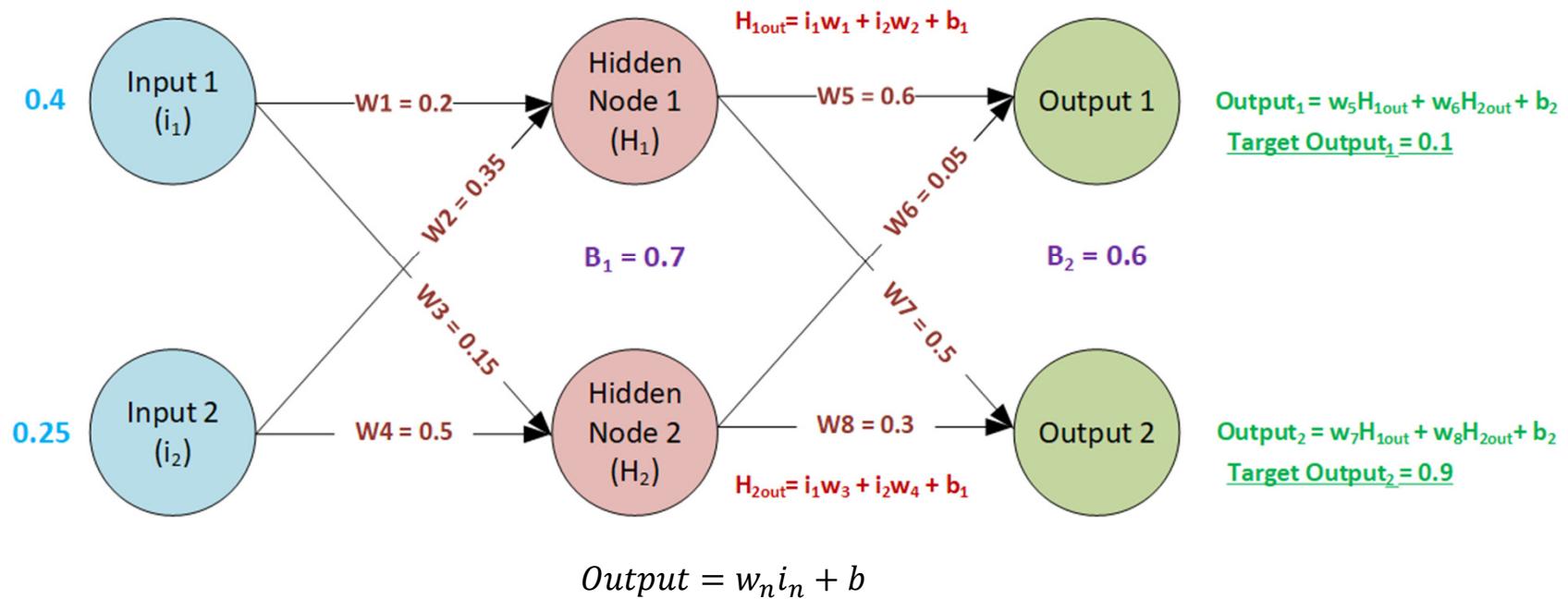


Looking at a single Node/Neuron



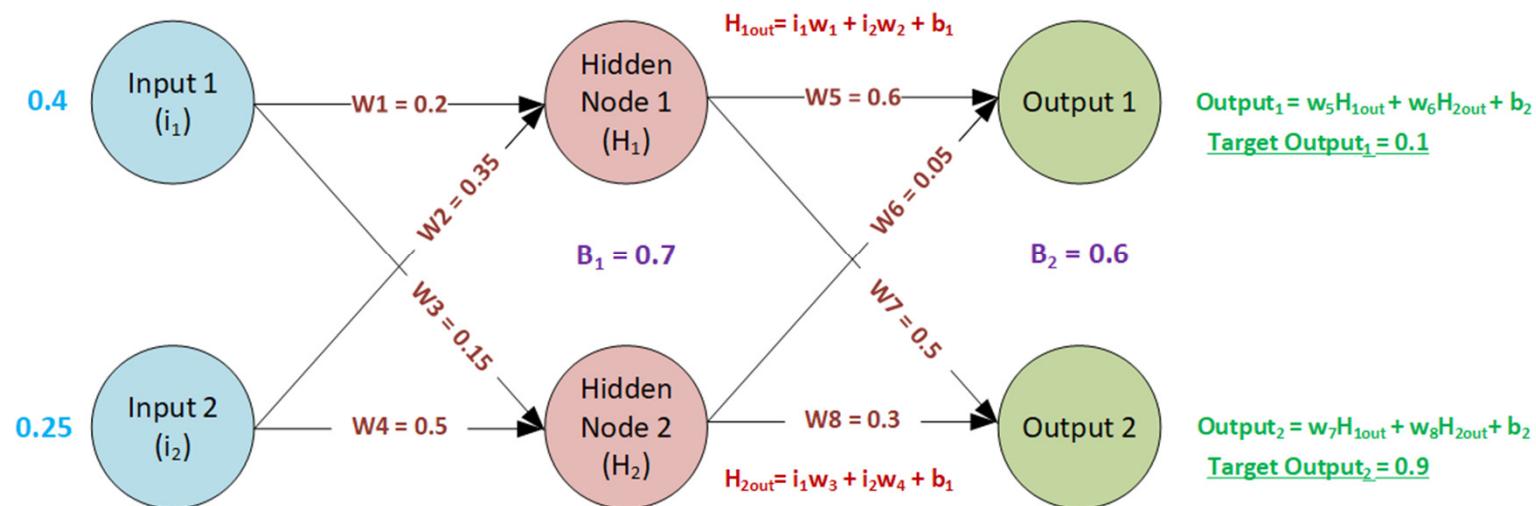


Steps for all output values





In reality these connections are simply formulas that pass values from one layer to the next



$$\text{Output of Nodes} = w_n i_n + b$$

$$H_1 = i_1 w_1 + i_2 w_2 + b_1$$

$$\text{Output}_1 = w_5 H_1 + H_2 w_6 + b_2$$

Calculating H₂

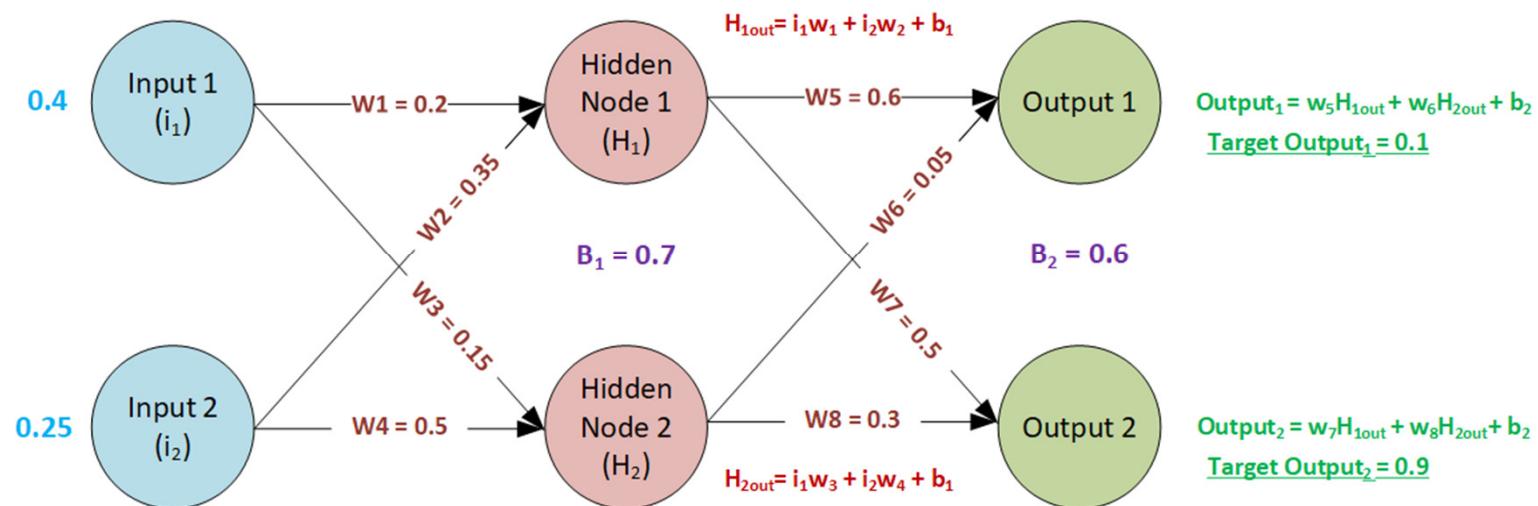


$$H_2 = i_1w_3 + i_2w_4 + b_1$$

$$H_2 = (0.4 \times 0.15) + (0.25 \times 0.5) + 0.7 = 0.06 + 0.125 + 0.7 = 0.885$$



Getting our final outputs



$$\text{Output}_1 = w_5 H_1 + H_2 w_6 + b_2$$

$$\text{Output}_1 = (0.6 \times 0.8675) + (0.05 \times 0.885) + 0.6 = 0.5205 + 0.04425 + 0.6 = 1.16475$$

$$\text{Output}_2 = 0.43375 + 0.2655 + 0.6 = 1.29925$$



What does this tell us?

- Our initial default random weights (w and b) produced very incorrect results
- Feeding numbers through a neural network is a matter of simple matrix multiplication
- Our neural network is still just a series of linear equations



The Bias Trick

- Making our weights and biases as one

- Now is a good time as any to show you the bias trick that is used to simplify our calculations.

$$\begin{array}{c} \begin{matrix} 0.2 & 0.65 & 0.54 & -3.1 \\ 1.2 & -0.23 & 0.23 & 0.5 \\ 0.35 & 1.5 & -0.7 & 0.02 \end{matrix} \\ W \end{array} \cdot \begin{array}{c} \begin{matrix} 44 \\ 73 \\ 32 \\ 64 \end{matrix} \\ x_i \end{array} + \begin{array}{c} \begin{matrix} -3.1 \\ 0.5 \\ 0.02 \end{matrix} \\ b \end{array} \longleftrightarrow \begin{array}{c} \begin{matrix} 0.2 & 0.65 & -3.1 & -3.1 \\ 1.2 & -0.23 & 0.5 & 0.5 \\ 0.35 & 1.5 & 0.02 & 0.02 \end{matrix} \\ W \end{array} \cdot \begin{array}{c} \begin{matrix} 44 \\ 73 \\ 32 \\ 64 \\ 1 \end{matrix} \\ x_i \end{array} + b$$

- x_i is our input values, instead of doing a multiplication then adding of our biases, we can simply add the biases to our weight matrix and add an addition element to our input data as 1.
- This simplifies our calculation operations as we treat the biases and weights as one.
- NOTE:** This makes our input vector size bigger by one i.e if x_i had 32 values, it will now have 33.

6.4

Activation Functions & What ‘Deep’ really means

An introduction to activation functions and their usefulness



Introducing Activation Functions

- In reality each Hidden Node also passes through an activation function.
- In the simplest terms an activation function changes the output of that function. For example, let's look at a simple activation function where values below 0 are clamped to 0. Meaning negative values are changed to 0.

$$\text{Output} = \text{Activation}(w_i x_i + b)$$

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

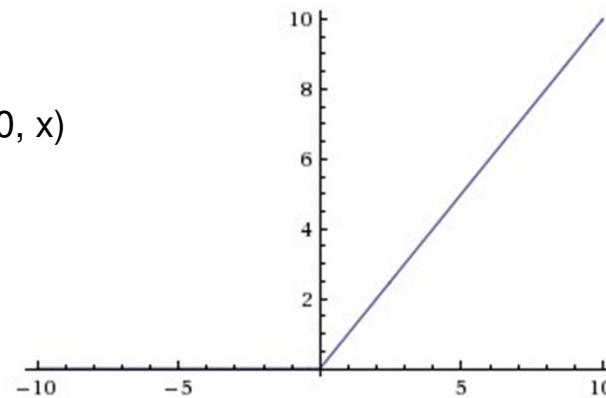
- Therefore, if $w_i x_i + b = 0.75$
- $f(x) = 0.75$
- However, if $w_i x_i + b = -0.5$ then $f(x) = 0$



The ReLU Activation Function

- This activation function is called the ReLU (Rectified Linear Unit) function and is one of the most commonly used activation functions in training Neural Networks.
- It takes the following appearance. The clamping values at 0 accounts for its non linear behavior.

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





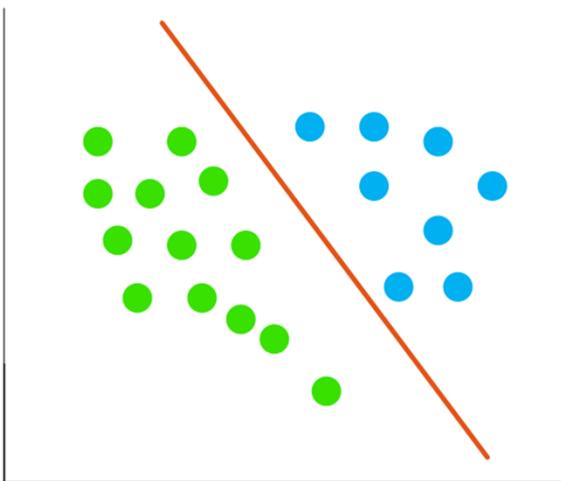
Why use Activation Functions? For Non Linearity

- Most ML algorithms find non linear data extremely hard to model
- The huge advantage of deep learning is the ability to understand nonlinear models

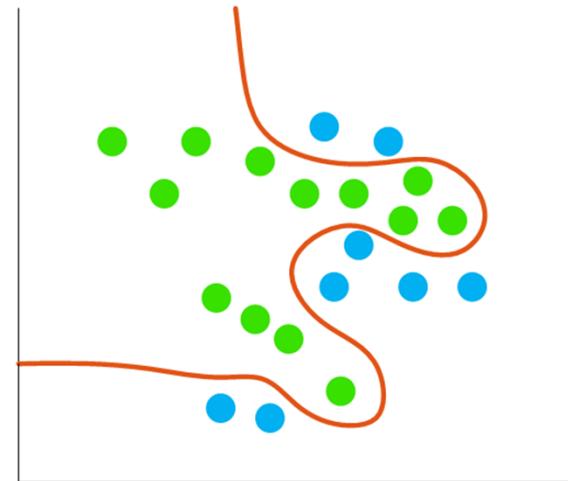


Example of Non Linear Data

Linearly Separable



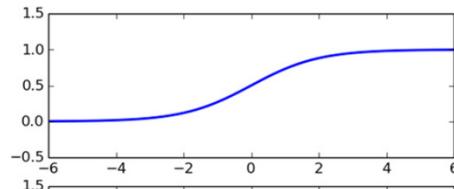
Non-Linearly Separable



NOTE: This shows just 2 Dimensions, imagine separating data with multiple dimensions

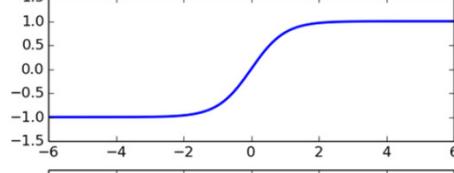


Types of Activation Functions



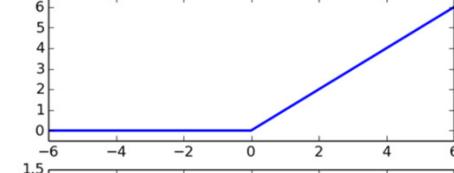
Sigmoid

$$\phi(z) = \frac{1}{1 + e^{-z}}$$



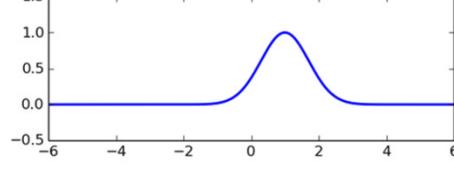
Hyperbolic Tangent

$$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Rectified Linear

$$\phi(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$



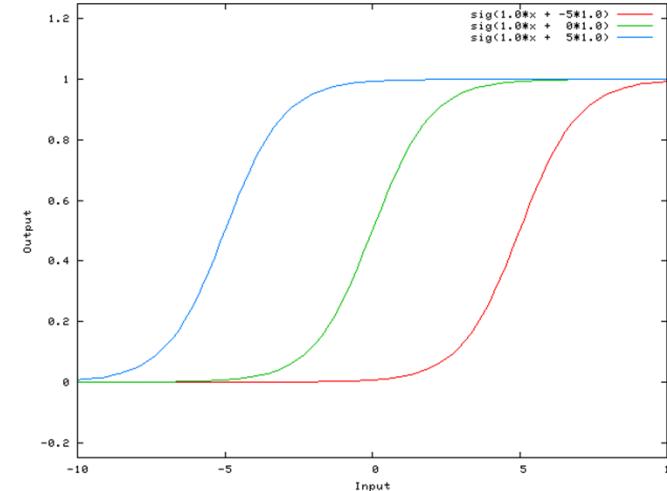
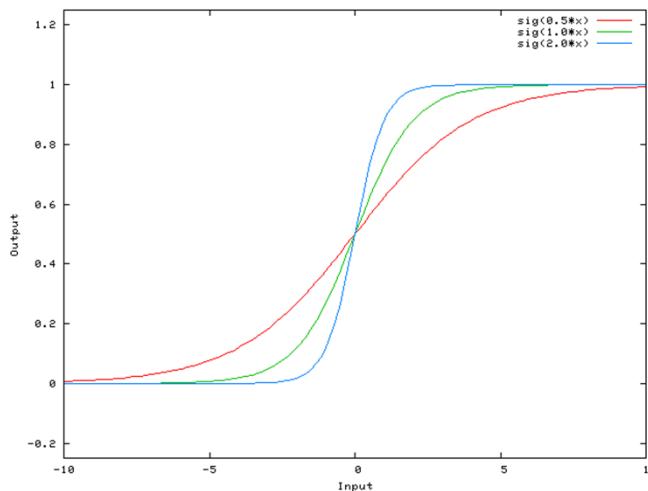
Radial Basis Function

$$\phi(z, c) = e^{-(\epsilon \|z - c\|)^2}$$



Why do we have biases in the first place?

- Biases provide every node/neuron with a trainable constant value.
- This allows us to shift the activation function output left or right.

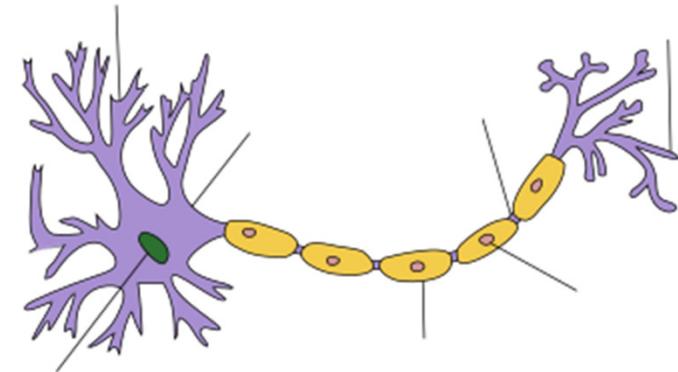


- Changes in weights simply change the gradient or steepness of the output, if we needed shift our function left or right , we need a bias.



Neuron Inspiration

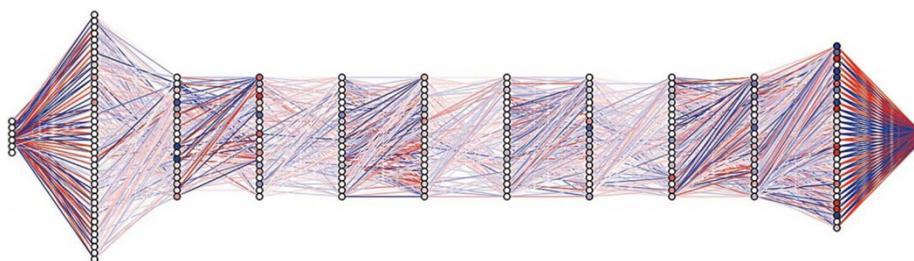
- Neuron only fires when an input threshold is reached
- Neural Networks follow that same principle





The 'Deep' in Deep Learning: Hidden Layers

- Depth refers to the number of hidden layers
- The deeper the network the better it learns non-linear mappings
- Deeper is always better, however there becomes a point of diminishing returns and overly long training time.
- Deeper Networks can lead to over fitting



Source: A visualization of Meade's neural network for predicting earthquakes
<https://harvardmagazine.com/2017/11/earthquakes-around-the-world>



The Real Magic Happens in Training

- We've seen Neural Networks are simple to execute, just matrix multiplications
- How do we determine those weights and biases?

6.5

Training Part 1: Loss Functions

The first step in determining the best weights for our Neural Network



Learning the Weights

- As you saw previously, our random default weights produced some very bad results.
- What we need is a way to figure out how to change the weights so that our results are more accurate.
- This is where the brilliance of Loss Functions , Gradient Descent and Backpropagation show their worth.



How do we beginin training a NN? What exactly do we need?

- Some (*more than some*) accurately labeled data, that we'll call a dataset
- A Neural Network Library/Framework (*Keras*)
- Patience and a decently fast computer

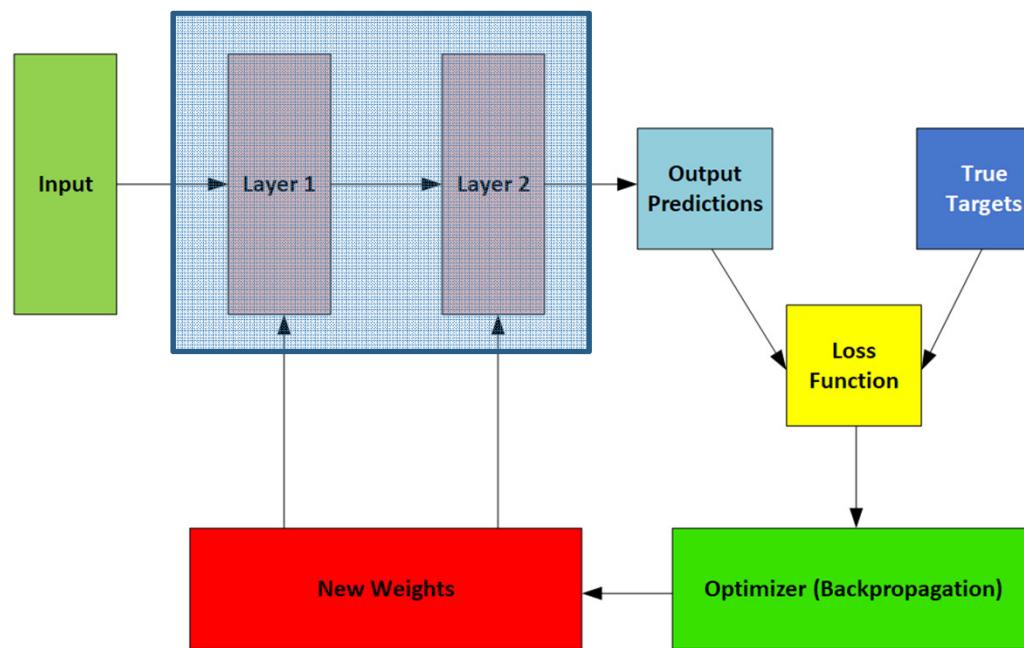


Training step by step

1. Initialize some random values for our weights and bias
2. Input a single sample of our data
3. Compare our output with the actual value it was supposed to be, we'll be calling this our Target values.
4. Quantify how 'bad' these random weights were, we'll call this our Loss.
5. Adjust weights so that the Loss lower
6. Keep doing this for each sample in our dataset
7. Then send the entire dataset through this weight 'optimization' program to see if we get an even lower loss
8. Stop training when the loss stops decreasing.



Training Process Visualized





Quantifying Loss with Loss Functions

- In our previous example our Neural Network produced some very 'bad' results, but how do we measure how bad they are?

Outputs	Predicted Results	Target Values	Difference (P-T)
1	1.16475	0.1	1.06475
2	1.29925	0.9	0.39925



Loss Functions

- Loss functions are integral in training Neural Nets as they measure the inconsistency or difference between the predicted results & the actual target results.
- They are always positive and penalize big errors well
- The lower the loss the 'better' the model
- There are many loss functions, Mean Squared Error (MSE) is popular
- $\text{MSE} = (\text{Target} - \text{Predicted})^2$

Outputs	Predicted Results	Target Values	Error (T-P)	MSE
1	1.16475	0.1	-1.06475	1.1336925625
2	1.29925	0.9	-0.39925	0.1594005625



Types of Loss Functions

- There are many types of loss functions such as:
 - L1
 - L2
 - Cross Entropy – Used in binary classifications
 - Hinge Loss
 - Mean Absolute Error (MAE)

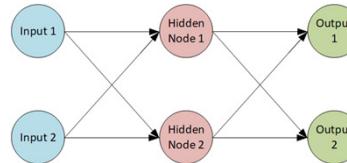
In practice, MSE is always a good safe choice. We'll discuss using different loss functions later on.

NOTE: A low loss goes hand in hand with accuracy. However, there is more to a good model than good training accuracy and low loss. We'll learn more about this soon.



Using the Loss to Correct the Weights

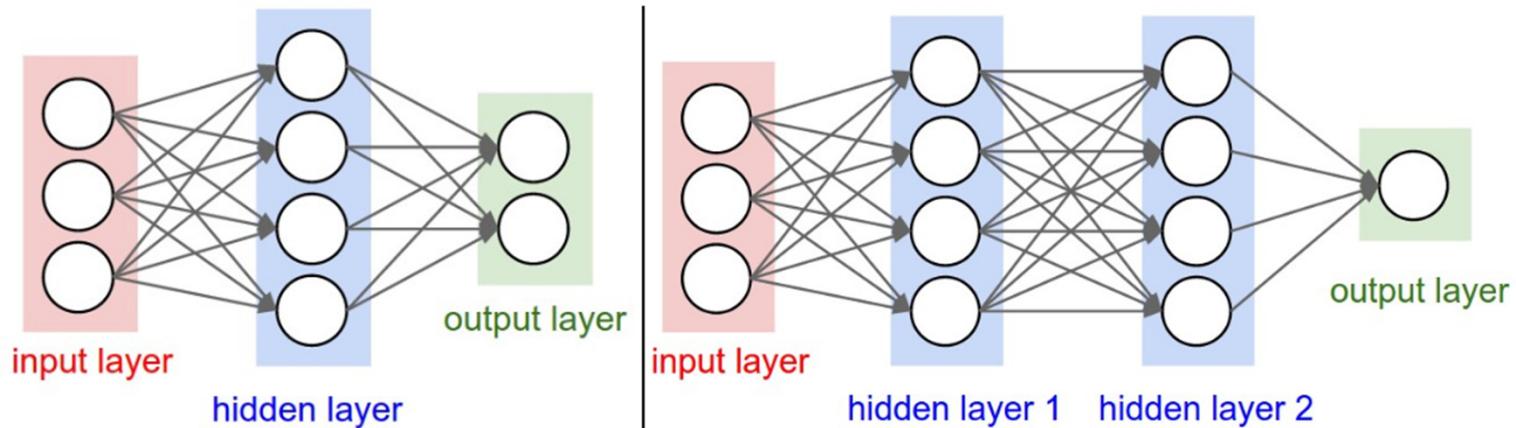
- Getting the best weights for our classifying our data is not a trivial task, especially with large image data which can contain thousands of inputs from a single image.



- Our simple 2 input, 2 output and 1 hidden layer network has X parameters.
 - Input Nodes X Hidden Layers + Hidden Layers x Output + Biases
 - In our case this is: $(2 \times 2) + (2 \times 2) + 4 = 12$ Learnable Parameters



Calculating the Number of Parameters



6.6

Training Part 2: Backpropagation & Gradient Descent

Determining the best weights efficiently



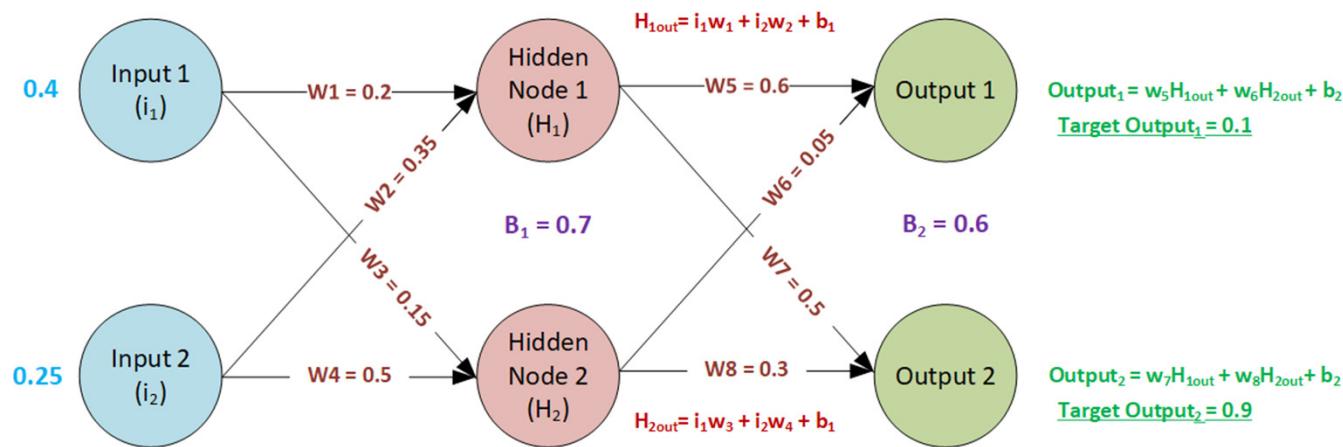
Introducing Backpropagation

- What if there was a way we could use the loss to determine how to adjust the weights.
- That is the brilliance of Backpropagation

Backpropagation tells us how much would a change in each weight affect the overall loss.



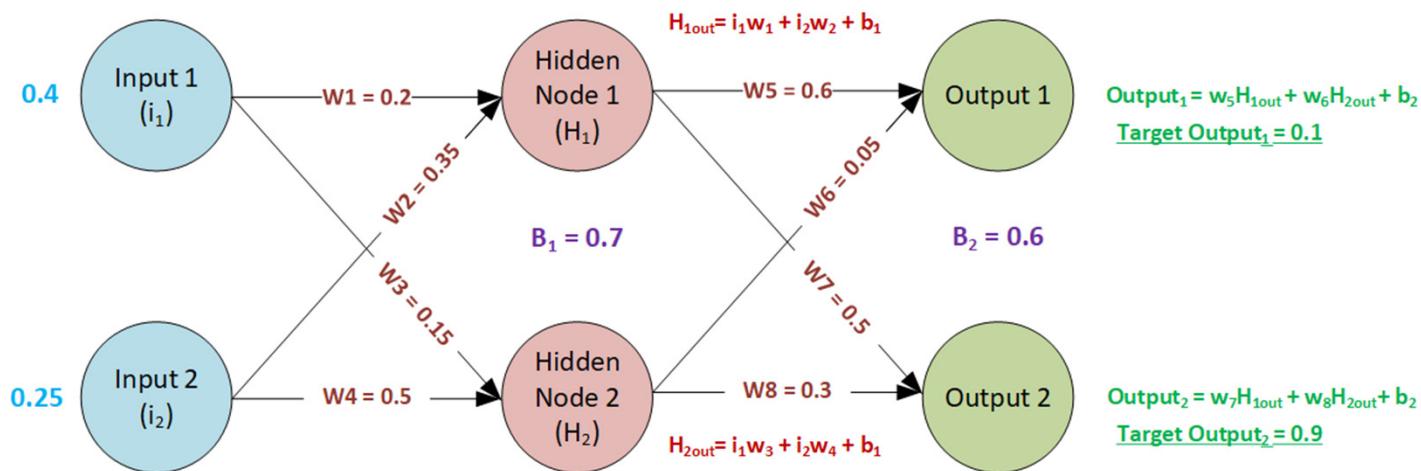
Backpropagation: Revisiting our Neural Net



- Using the MSE obtained from Output 1, Backpropagation allows us know:
 - If changing w_5 from 0.6 by a small amount, say to 0.6001 or 0.5999,
 - Whether our overall Error or Loss has increased or decreased.



Backpropagation: Revisiting our Neural Net



- We then backpropagate this loss to each node (Right to Left) to determine which direction the weight should move (negative or positive)
- We do this for all nodes in the Neural Network



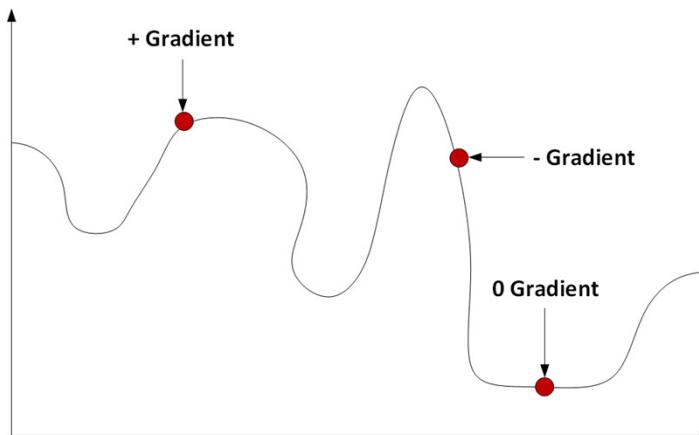
Backpropagation – The full cycle

- Therefore, by simply passing one set of inputs of a single piece of our training data, we can adjust all weights to reduce the loss or error.
- However, this tunes the weights for that specific input data. How do make our Neural Network generalize?
- We do this for each training samples in our training data (called an Epoch or Iteration).



Introducing Gradient Descent

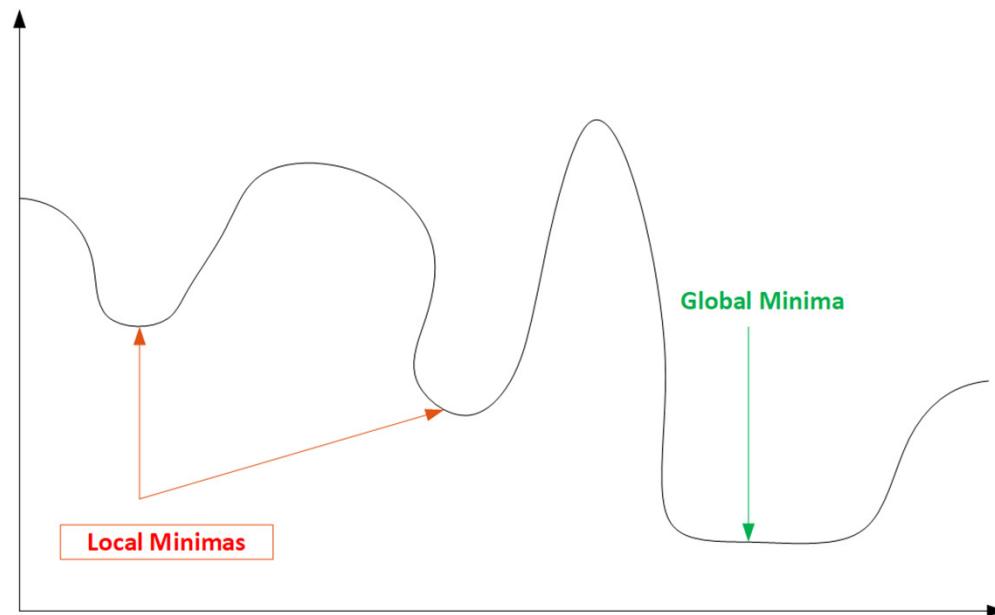
- By adjusting the weights to lower the loss, we are performing gradient descent. This is an 'optimization' problem.
- Backpropagation is simply the method by which we execute gradient descent.
- Gradients (also called slope) are the direction of a function at a point, it's magnitude signifies how much the function is changing at that point.



- By adjusting the weights to lower the loss, we are performing gradient descent.
- Gradients are the direction of a function at a point



Gradient Descent



Imagine our global minima is the bottom of this rough bowl. We need to traverse through many peaks and valleys before we find it



Stochastic Gradient Descent

- Naïve Gradient Decent is very computationally expensive/slow as it requires exposure to the entire dataset, then updates the gradient.
- Stochastic Gradient Descent (SGD) does the updates after every input sample. This produces noisy or fluctuating loss outputs. However, again this method can be slow.
- Mini Batch Gradient Descent is a combination of the two. It takes a batch of input samples and updates the gradient after that batch is processed (batches are typical 20-500, though no clear rule exists). It leads to much faster training (i.e. faster convergence to the global minima)



Overview

We learned:

- That Loss Functions (such as MSE) quantify how much error our current weights produce.
- That Backpropagation can be used to determine how to change the weights so that our loss is lower.
- This process of optimizing or lowering the weights is called Gradient Descent, and an efficient method of doing this is the Mini Batch Gradient Descent algorithm.

6.7

Back Propagation & Learning Rates: A Worked Example

A worked example of Back Propagation.



Backpropagation

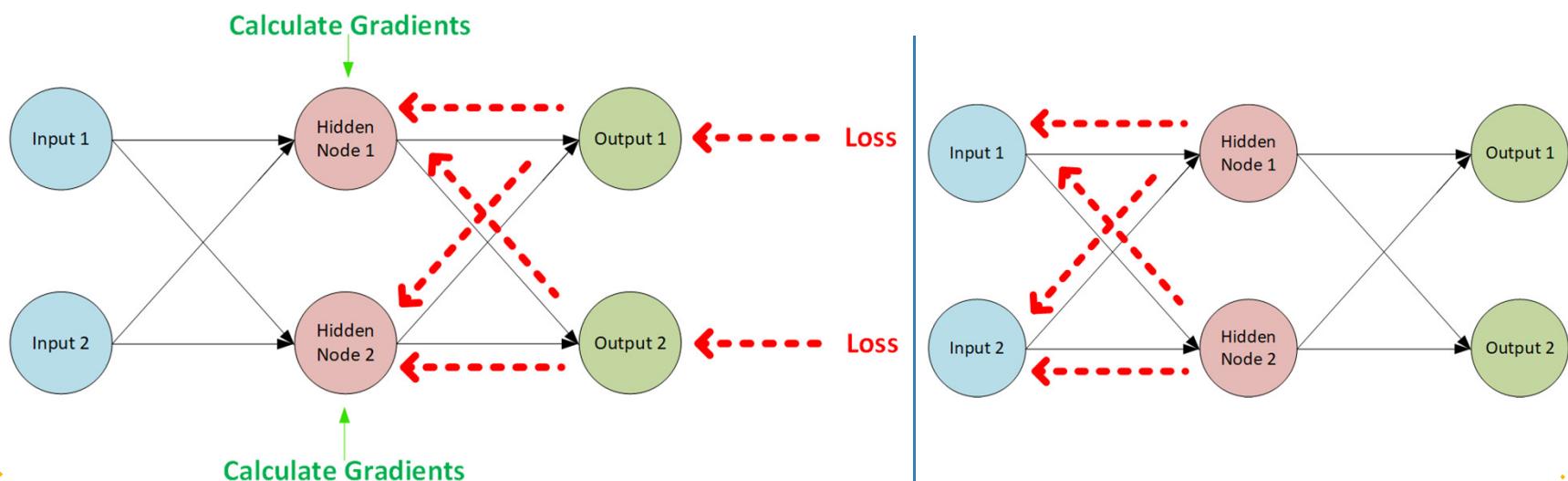
- From the previous section you have an idea of what we achieve with Backpropagation.
- It's a method of executing gradient descent or weight optimization so that we have an efficient method of getting the lowest possible loss.

How does this 'black magic' actually work?

Backpropagation Simplified



- We obtain the total error at the output nodes and then propagate these errors back through the network using Backpropagation to obtain the new and better gradients or weights.





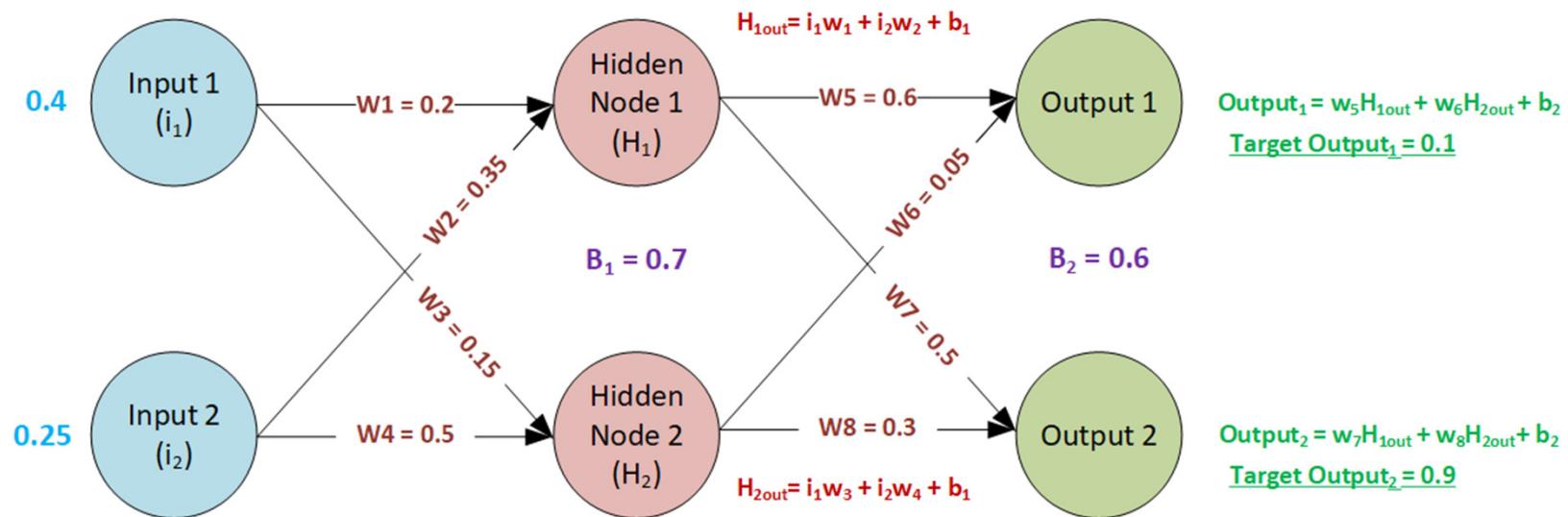
The Chain Rule

- Backpropagation is made possible by the *Chain Rule*.
- What is the Chain Rule? Without over complicating things, it's defined as:
 - If we have two functions: $y = f(u)$ and $u = g(x)$ then the derivative of y is:

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

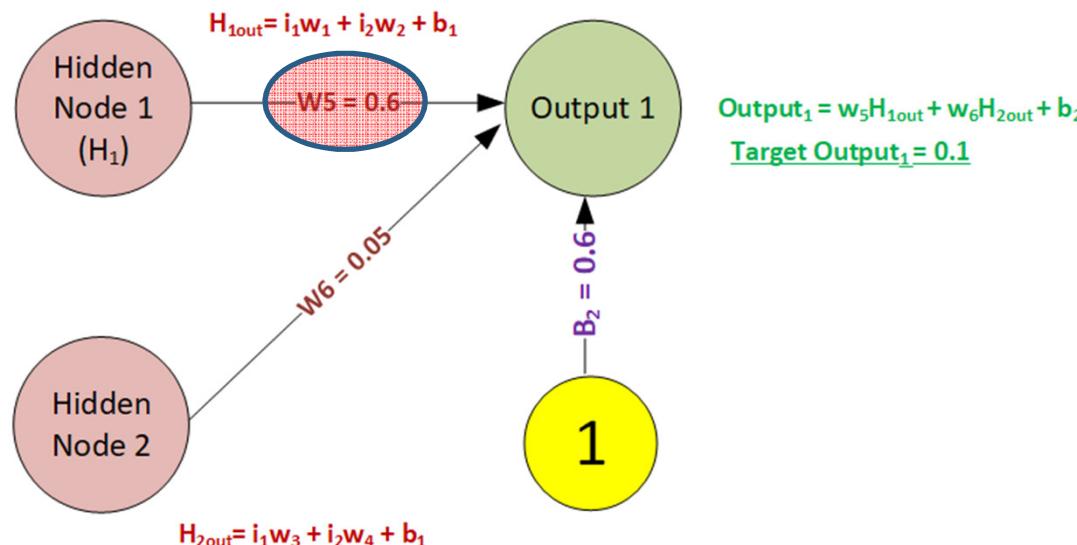


Let's take a look at our previous basic NN





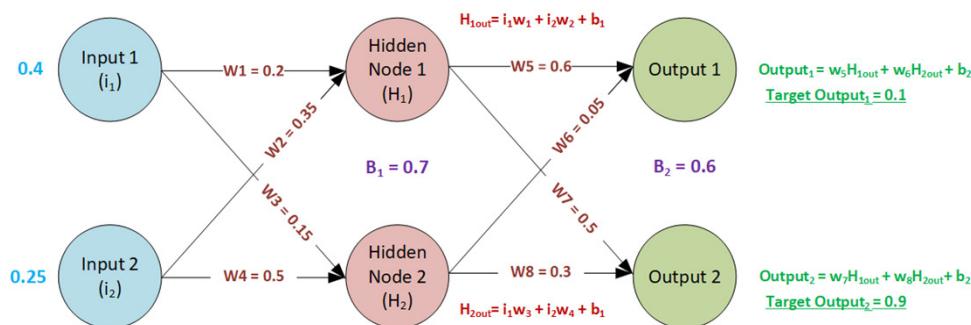
We use the Chain Rule to Determine the Direction the Weights Should Take



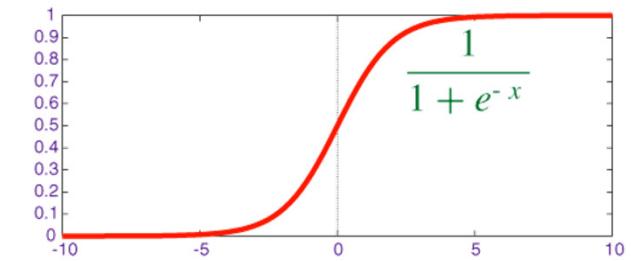
- Let's take a look at W_5 , how does a change in W_5 affect the Error at Output_1 ?
- Should W_5 be increased or decreased?



Our Calculated Forward Propagation and Loss Values



Logistic Activation Function
Squashes the output between 0 and 1



- Using a Logistic Activation Function at each node, our Forward Propagation values become:

- $H_1 = \mathbf{0.704225}$

- $H_2 = \mathbf{0.707857}$

- $\text{Output } 1 = \mathbf{0.742294}$

- $\text{Output } 2 = \mathbf{0.762144}$

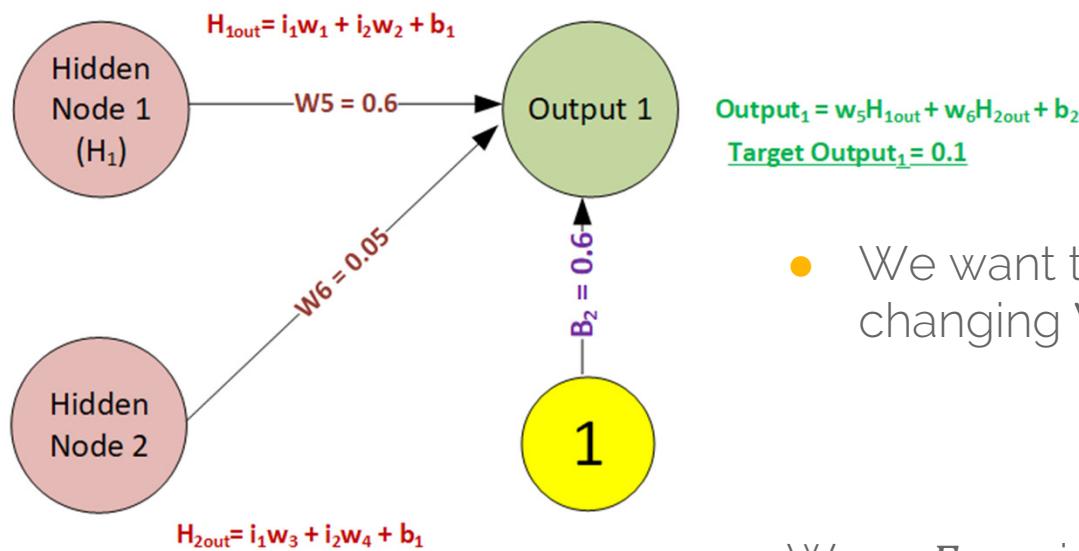


Slight Change in Our MSE Loss Function

- Let's define the MSE as
 - = Error = $\frac{1}{2}(\text{target} - \text{output})^2$
- The $\frac{1}{2}$ is included to cancel the exponent when differentiated (looks better)
- Output 1 Loss = **0.206271039**
- Output 2 Loss = **0.009502145**



Exploring W_5



- We want to know how much changing W_5 changes to total Error.

$$\frac{dE_{total}}{dw_5}$$

Where E_{total} is the sum of the Error from Output 1 and Output 2



Using the Chain Rule to Calculate W_5

- $\frac{dE_{total}}{dw_5} = \frac{1}{\frac{dE_{total}}{dout_1}} \times \frac{2}{\frac{dout_1}{dNetOutput_1}} \times \frac{3}{\frac{dNetOutput_1}{dw_5}}$
- $E_{total} = \frac{1}{2}(target_{o1} - out_1)^2 + \frac{1}{2}(target_{o2} - out_2)^2$

Therefore differentiating E_{total} with respect to out_1 gives us

$$\frac{dE_{total}}{dout_1} = 2 \times \frac{1}{2}(target_{o1} - out_1)^{2-1} \times (-1) + 0$$

$$\frac{dE_{total}}{dout_1} = out_1 - target_{o1}$$

$$1 \quad \frac{dE_{total}}{dout_1} = (0.742294385 - 0.1) = 0.642294385$$



Let's get $\frac{dOut_1}{dNetOutput_1}$

- $dOut_1 = \frac{1}{1+e^{-netO_1}}$
- Fortunately, the partial derivative of the logistic function is the output multiplied by 1 minus the output:
$$\frac{dOut_1}{dnetO_1} = out_{o1}(1 - out_{o1}) = 0.742294385(1 - 0.742294385)$$
- 2 •
$$\frac{dOut_1}{dnetO_1} = -0.191293$$



Let's get $\frac{dnetO_1}{dw_5}$

- $netO_1 = (w_5 \times outH_1) + (w_6 \times outH_2) + (b_2 \times 1)$
- $\frac{dnetO_1}{dw_5} = 1 \times outH_1 \times w_5^{(1-1)} + 0 + 0$
- 3 • $\frac{dnetO_1}{dw_5} = outH_1 = 0.704225234$



We now have all the pieces to get $\frac{dE_{total}}{dw_5}$

- $\frac{dE_{total}}{dw_5} = \frac{dE_{total}}{dOut_1} \times \frac{dOut_1}{dNetOutput_1} \times \frac{dNetOutput_1}{dw_5}$
- $\frac{dE_{total}}{dw_5} = 0.642294385 \times 0.191293431* \times 0.704225234$
- $\frac{dE_{total}}{dw_5} = 0.086526$



So what's the new weight for W_5 ?

- *New* $w_5 = w_5 - \eta \times \frac{dE_{total}}{dw_5}$
- *New* $w_5 = 0.6 - (0.5 \times 0.086526) = 0.556737$

Learning Rate

- Notice we introduced a new parameter ' η ' and gave it a value of **0.5**
- Look carefully at the first formula. The learning rate simply controls how a big a magnitude jump we take in the direction of $\frac{dE_{total}}{dw_5}$
- Learning rates are always positive and range from $\gamma > 0 \leq 1$
- A large learning rate will allow faster training, but can overshoot the global minimum (getting trapped in a local minima instead). A small learning will take longer to train but will more reliably find the global minimum.

Check your answers



- $\text{new } w_6 = 0.400981$
- $\text{new } w_7 = 0.508845$
- $\text{new } w_8 = 0.558799$



You've just used Backpropagation to Calculate the new W_5

- You can now calculate the new updates for W_6 , W_7 and W_8 similarly.
- W_1 , W_2 , W_3 and W_4 are similar:

$$\frac{dE_{total}}{dw_1} = \frac{dE_{total}}{dOut_{H1}} \times \frac{dOut_{H1}}{dNetOutput_{H1}} \times \frac{dNetOutput_{H1}}{dw_1}$$

- I'll leave this as an exercise for you.

6.8

Regularization, Overfitting, Generalization and Test Datasets

How do we know our model is good?



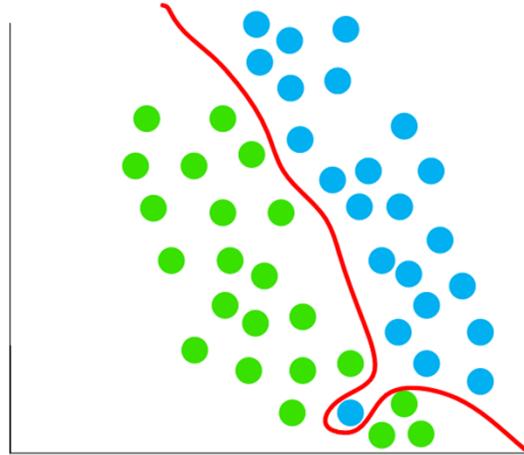
What Makes a Good Model?

- A good model is accurate
- Generalizes well
- Does not overfit

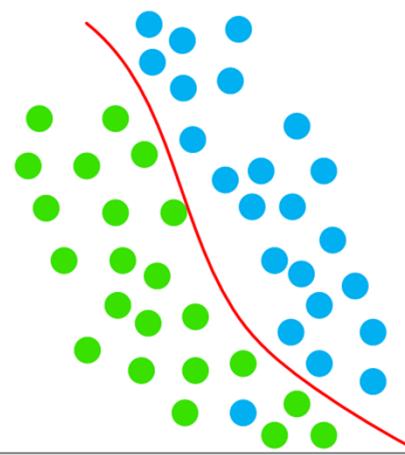


What Makes a Good Model?

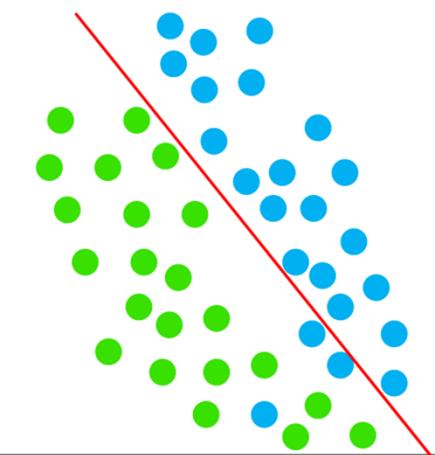
Model A



Model B



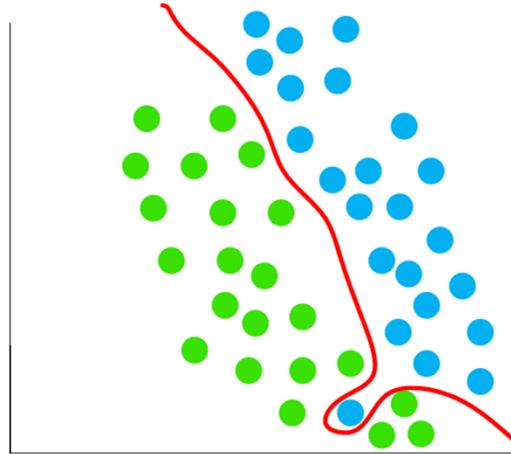
Model C



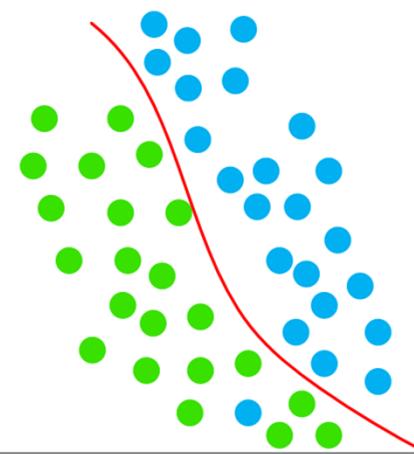


What Makes a Good Model?

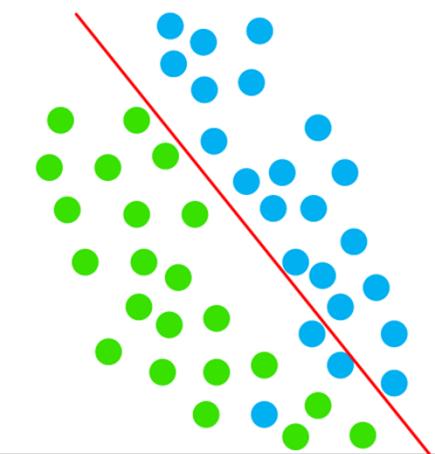
- Overfitting



- Ideal or Balanced



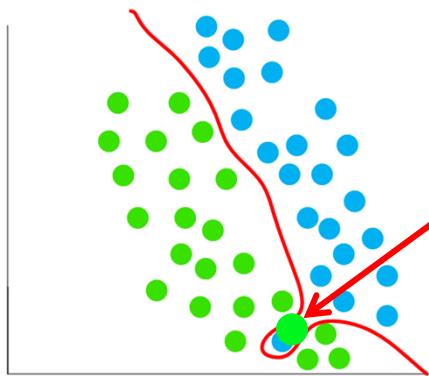
- Underfitting





Overfitting

- Overfitting leads to *poor models* and is one of the most common problems faced developing in AI/Machine Learning/Neural Nets.
- Overfitting occurs when our Model fits near perfectly to our training data, as we saw in the previous slide with Model A. However, fitting to closely to training data isn't always a good thing.



- What happens if we try to classify a brand new point that occurs at the position shown on the left? (who's true color is green)
- It will be misclassified because our model has overfit the test data
- Models don't need to be complex to be good

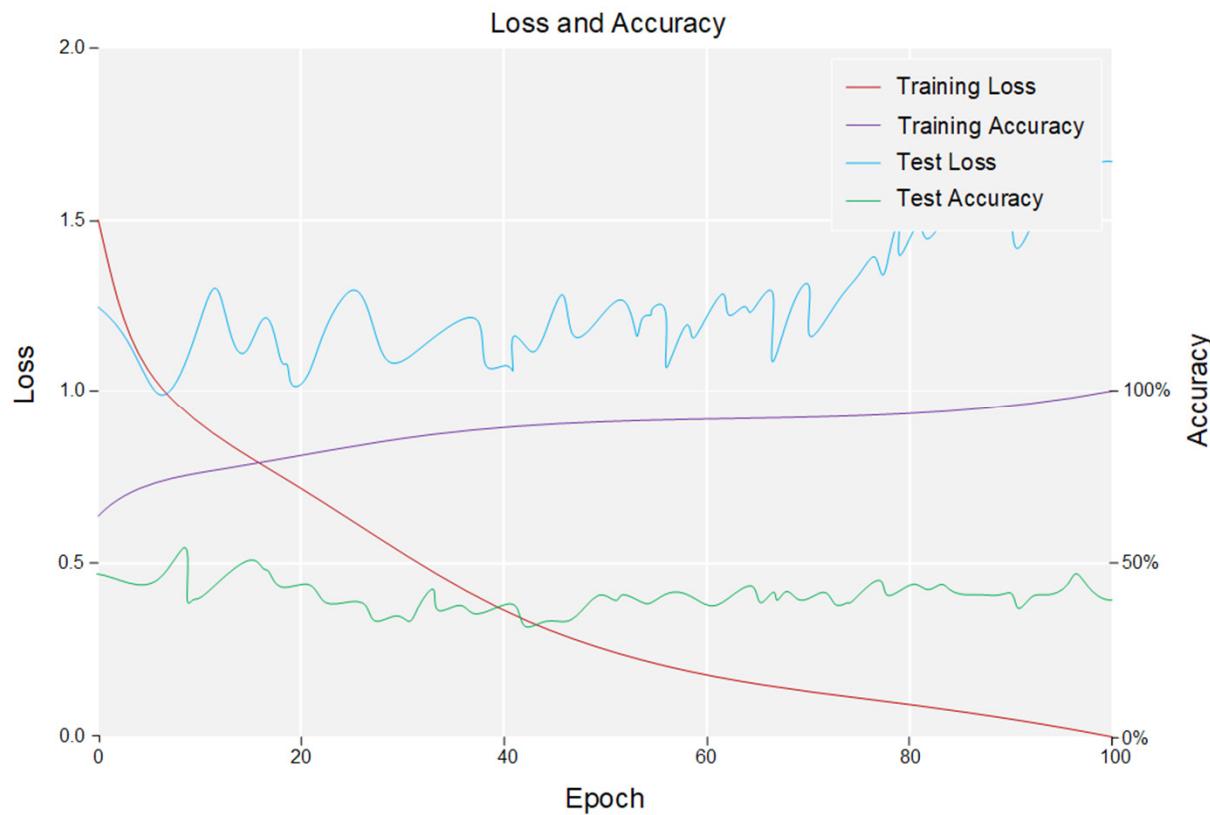


How do we know if we've Overfit? Test on your model on.....Test Data!

- In all Machine Learning it is extremely important we hold back a portion of our data (10-30%) as pure untouched test data.

Training Data	Test Data
---------------	-----------
- Untouched meaning that this data is NEVER seen by the training algorithm. It is used purely to test the performance of our model to assess its accuracy in classifying new never before seen data.
- Many times when Overfitting we can achieve high accuracy 95%+ on our test data, but then get abysmal (~70%) results on the test data. That is a perfect example of Overfitting.

Overfitting Illustrated Graphically

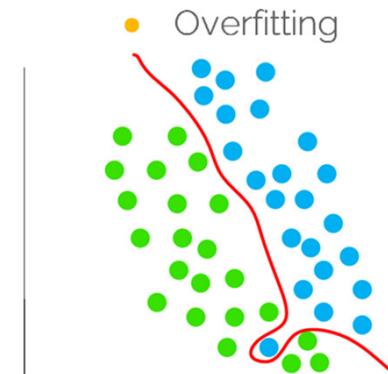


- Examine our Training Loss and Accuracy. They're both heading the right directions!
- But, what's happening to the loss and accuracy on our Test data?
- This is classic example of Overfitting to our training data



How do we avoid overfitting?

- Overfitting is a consequence of our weights. Our weights have been tuned to fit our Training Data well but due to this 'over tuning' it performs poorly on unseen Test Data.
- We know our weights are a decent model, just too sensitively tuned. If only there were a way to fix this?





How do we avoid overfitting?

- We can use less weights to get smaller/less deep Neural Networks
 - Deeper models can sometimes find features or interpret noise to be important in data, due to their abilities to memorize more features (called memorization capacity)



How do we avoid overfitting?

- We can use less weights to get smaller/less deep Neural Networks
 - Deeper models can sometimes find features or interpret noise to be important in data, due to their abilities to memorize more features (called memorization capacity)

Or Regularization!

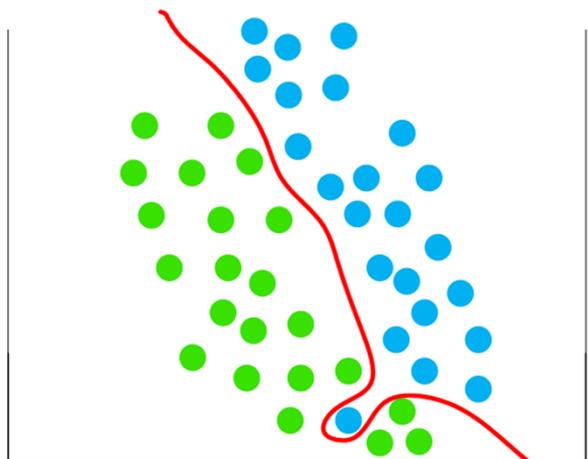
- It is better practice to regularize than reduce our model complexity.



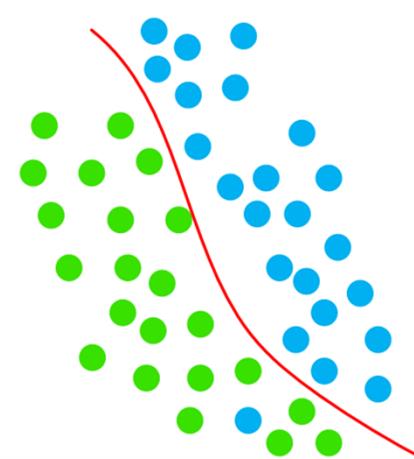
What is Regularization?

- It is a method of making our model more general to our dataset.

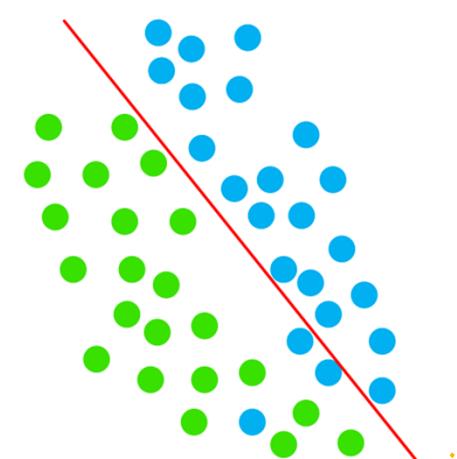
- Overfitting



- Ideal or Balanced



- Underfitting





Types of Regularization

- L1 & L2 Regularization
- Cross Validation
- Early Stopping
- Drop Out
- Dataset Augmentation



L1 And L2 Regularization

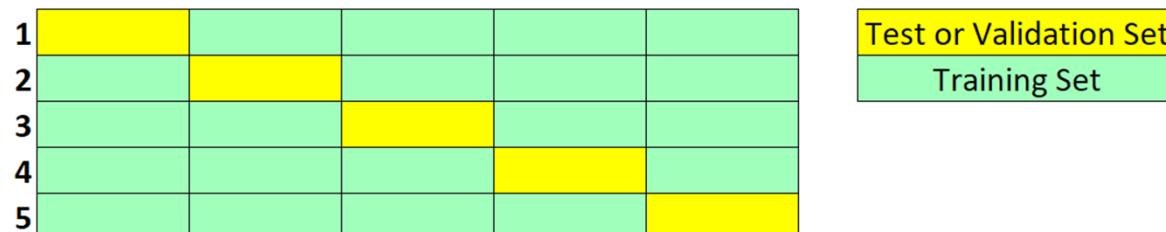
- L1 & L2 regularization are techniques we use to penalize large weights. Large weights or gradients manifest as abrupt changes in our model's decision boundary. By penalizing, we're really making them smaller.
- L2 also known as Ridge Regression
 - $Error = \frac{1}{2}(target_{01} - out_1)^2 + \frac{\lambda}{2} \sum w_i^2$
- L1 also known as Lasso Regression
 - $Error = \frac{1}{2}(target_{01} - out_1)^2 + \frac{\lambda}{2} \sum |w_i|$
- λ controls the degree of penalty we apply.
- Via Backpropagation, the penalty on the weights is applied to the weight updates
- The difference between them is that L1 brings the weights of the unimportant features to 0, thus acting as feature selection algorithm (known as sparse models or models with reduced parameters.)



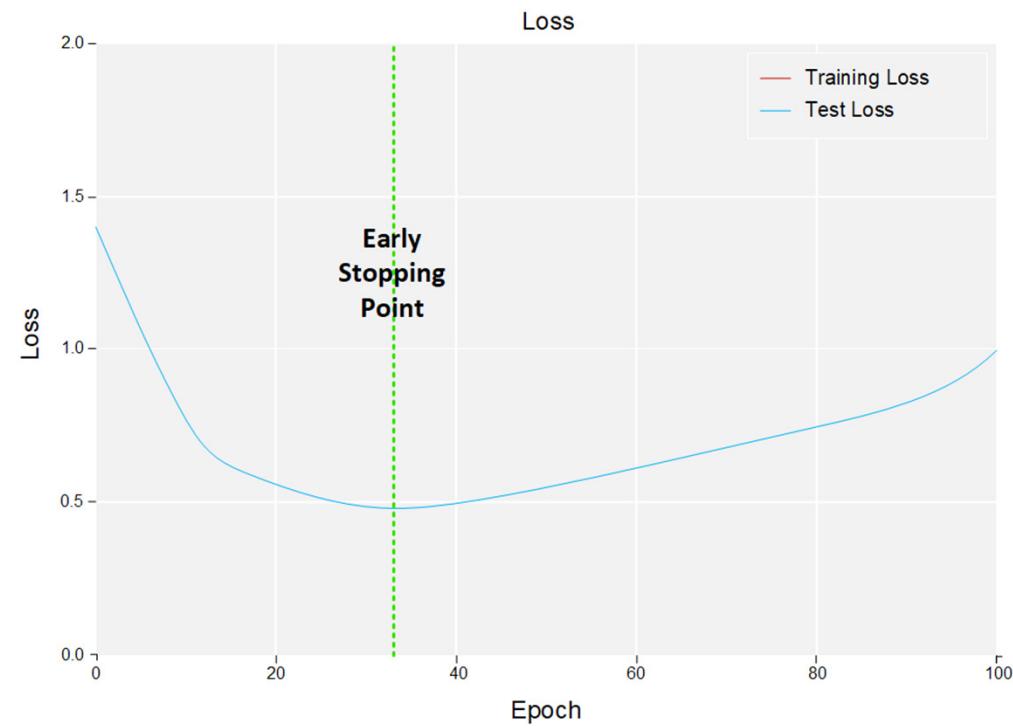
Cross Validation

- Cross validation or also known as k-fold cross validation is a method of training where we split our dataset into k folds instead of a typical training and test split.
- For example, let's say we're using 5 folds. We train on 4, and use the 5th final fold as our test. We then train on the other 4 folds, and test on another.
- We then use the average weights across coming out of each cycle.
- Cross Validation reduces overfitting but slows the training process

k-folds (5 shown)



Early Stopping



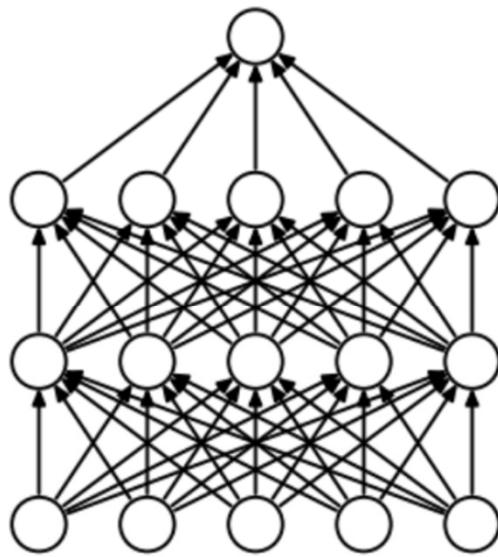


Dropout

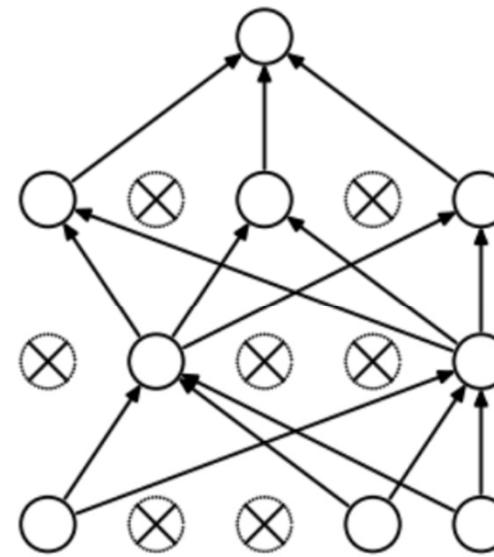
- Dropout refers to dropping nodes (both hidden and visible) in a neural network with the aim of reducing overfitting.
- In training certain parts of the neural network are ignored during some forward and backward propagations.
- Dropout is an approach to regularization in neural networks which helps reducing interdependent learning amongst the neurons. Thus the NN learns more robust or meaningful features.
- In Dropout we set a parameter ' P ' that sets the probability of which nodes are kept or $(1-p)$ for those that are dropped.
- Dropout almost doubles the time to converge in training



Dropout Illustrated



(a) Standard Neural Net

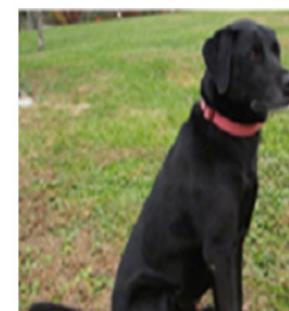
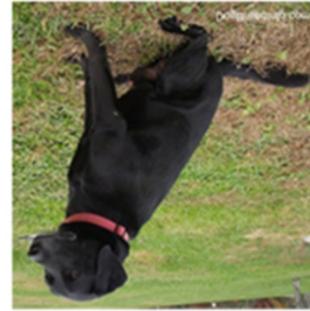
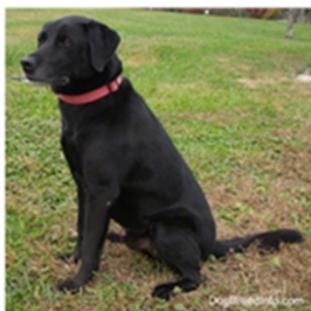


(b) After applying dropout.



Data Augmentation

- Data Augmentation is one of the easiest ways to improve our models.
- It's simply taking our input dataset and making slight variations to it in order to improve the amount of data we have for training. Examples below.
- This allows us to build more robust models that don't overfit.



6.9

Epochs, Iterations and Batch Sizes

Understanding some Neural Network Training Terminology



Epochs

- You may have seen or heard me mention Epochs in the training process, so what exactly is an Epoch?
 - An Epoch occurs when the full set of our training data is passed/forward propagated and then backpropagated through our neural network.
 - After the first Epoch, we will have a decent set of weights, however, by feeding our training data again and again into our Neural Network, we can further improve the weights. This is why we train for several iterations/epochs (50+ usually)



Batches

- Unless we had huge volumes of RAM, we can't simply pass all our training data to our Neural Network in training. We need to split the data up into segments or Batches.
- Batch Size is the number of training samples we use in a single batch.
- Example, say we had 1000 samples of data, and specified a batch size of 100. In training, we'd take 100 samples of that data and use it in the forward/backward pass then update our weights. If the batch size is 1, we're simply doing Stochastic Gradient Descent.



Iterations

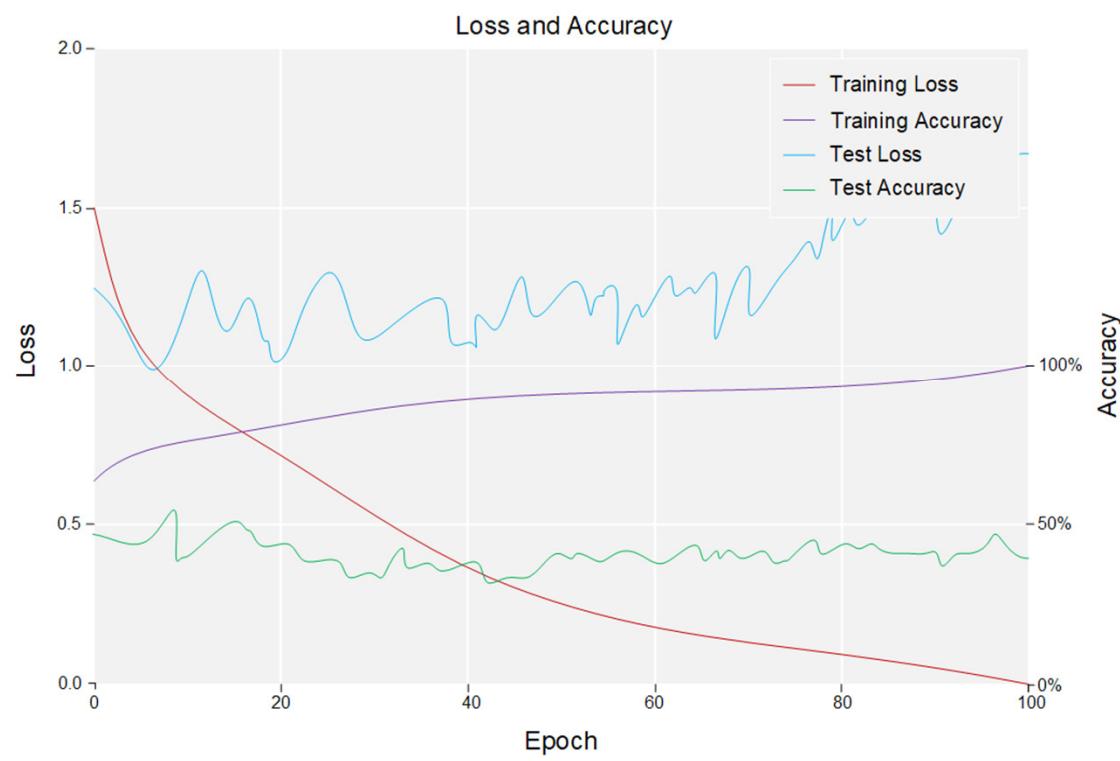
- Many confuse iterations and Epochs (I was one of them)
- However, the difference is quite simple, Iterations are the number of batches we need to complete one Epoch.
- In our previous example, we had 1000 items in our dataset, and set a batch size of 100. Therefore, we'll need 10 iterations (100×10) to complete one Epoch.

6.10

Measuring Performance

How we measure the performance of our Neural Network

Loss and Accuracy





Loss and Accuracy

- It is important to realize while Loss and Accuracy represent different things, they are essentially measuring the same thing. The performance of our NN on our training Data.
- Accuracy is simply a measure of how much of our training data did our model classify correctly
 - $$\text{Accuracy} = \frac{\text{Correct Classifications}}{\text{Total Number of Classifications}}$$
- Loss values go up when classifications are incorrect i.e. different to the expected Target values. As such, Loss and Accuracy on the Training dataset WILL correlate.



Is Accuracy the only way to assess a model's performance?

- While very important, accuracy alone doesn't tell us the whole story.
- Imagine we're using a NN to predict whether a person has a life threatening disease based on a blood test.
- There are now 4 possible scenarios.
 1. TRUE POSITIVE
 - Test Predicts **Positive** for the disease and the person **has the disease**
 2. TRUE NEGATIVE
 - Test Predicts **Negative** for the disease and the person **does NOT have the disease**
 3. FALSE POSITIVE
 - Test Predicts **Positive** for the disease but the person **does NOT have the disease**
 4. FALSE NEGATIVE
 - Test Predicts **Negative** for the disease but the person actually **has the disease**



For a 2 or Binary Class Classification Problem

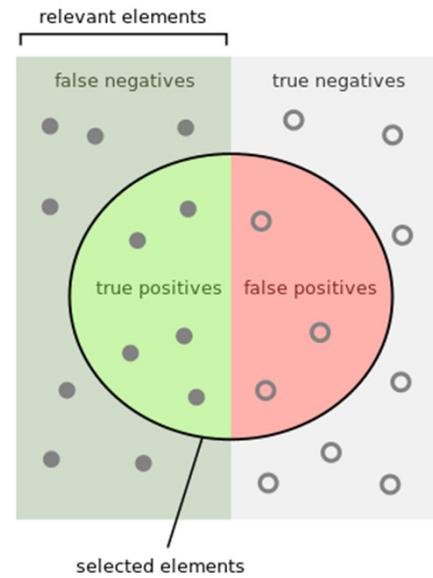
- Recall – How much of the positive classes did we get correct
 - $$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$
- Precision – Out of all the samples how much did the classifier get right
 - $$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$
- F-Score – Is a metric that attempts to measure both Recall & Precision
 - $$F - Score = \frac{2 \times Recall \times Precision}{Precision + Recall}$$



An Example

Let's say we've built a classifier to identify gender (male vs female) in an image where there are:

- 10 Male Faces & 5 Female Faces
- Our classifier identifies 6 male faces
- Out of the 6 male faces - 4 were male and 2 female.
- Our Precision is $4 / 6$ or 0.66 or 66%
- Our Recall is $4 / (4 + 6)$ (**6 male faces were missed**) or 0.4 or 40%
- Our F-Score is $= \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Precision} + \text{Recall}} = \frac{2 \times 0.4 \times 0.66}{0.4 + 0.66} = \frac{0.528}{1.06} = 0.498$



How many selected items are relevant?

$$\text{Precision} = \frac{\text{green}}{\text{green} + \text{red}}$$

How many relevant items are selected?

$$\text{Recall} = \frac{\text{green}}{\text{green} + \text{red}}$$

https://en.wikipedia.org/wiki/Precision_and_recall

6.11

Review and Best Practices

Review on the entire training process and some general guidelines to designing your own Neural Nets

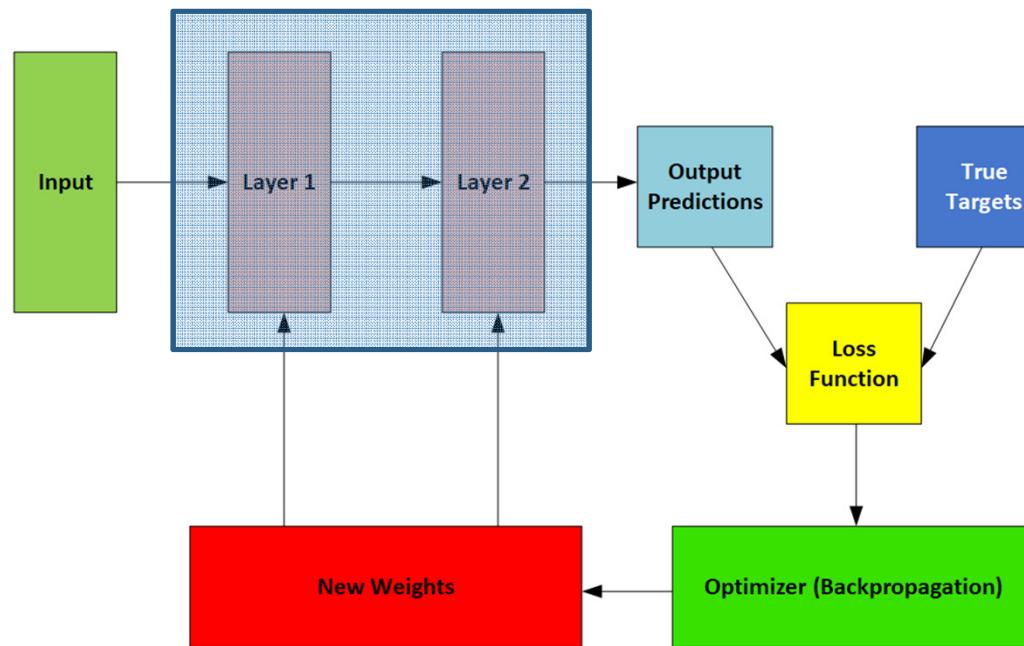


Training step by step

1. Initialize some random values for our weights and bias
2. Input a single sample of our data or batch of samples
3. Compare our output with the actual value target values.
4. Use our loss function to put a value to our loss.
5. Use Backpropagation to perform mini batch gradient descent to update our weights
6. Keep doing this for each batch of data in our dataset (iteration) until we complete an Epoch
7. Complete several Epochs and stop training when the Accuracy on our test dataset



Training Process Visualized





Best Practices

- Activation Functions - Use ReLU or Leaky ReLU
- Loss Function - MSE
- Regularization
 - Use L2 and start small and increase accordingly (0.01, 0.02, 0.05, 0.1.....)
 - Use Dropout and set P between 0.2 to 0.5
- Learning Rate – 0.001
- Number of Hidden Layers – As deep as your machine's performance will allow
- Number of Epochs – Try 10-100 (ideally at least 50)

7.0

Convolutional Neural Networks (CNNs) Explained

The best explanation you'll ever see on Convolutional Neural Networks



Convolutional Neural Networks Explained

- **7.1 Introduction to Convolutional Neural Networks (CNNs)**
- **7.2 Convolutions & Image Features**
- **7.3 Depth, Stride and Padding**
- **7.4 ReLU**
- **7.5 Pooling**
- **7.6. The Fully Connected Layer**
- **7.7 Training CNNs**
- **7.8 Designing your own CNNs**

7.1

Introduction to Convolutional Neural Networks (CNNs)



Why are CNNs needed?

- We spent a while discussing Neural Networks, which probably makes you wonder, why are we now discussing Convolution Neural Networks or CNNs?
- Understanding Neural Networks is critical in understanding CNNs as they form the foundation of Deep Learning Image Classification.

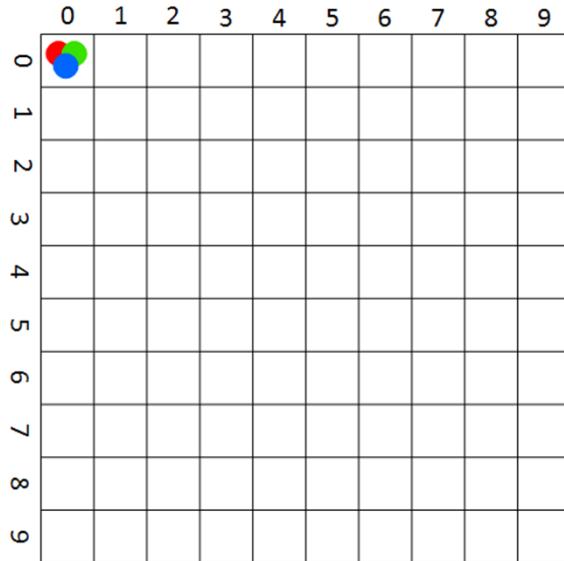


Why CNNs?

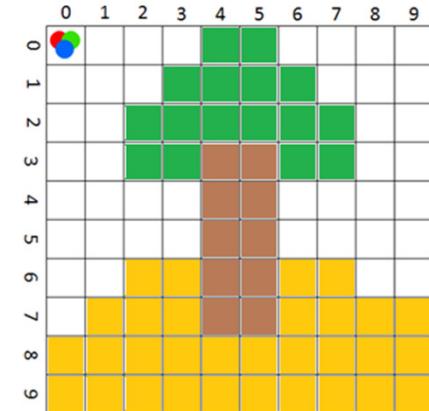
- Because Neural Networks don't scale well to images data
- Remember in our intro slides we discussed how images are stored.



How do Computers Store Images?



- Each pixel coordinate (x, y) contains 3 values ranging for intensities of 0 to 255 (8-bit).
 - Red
 - Green
 - Blue

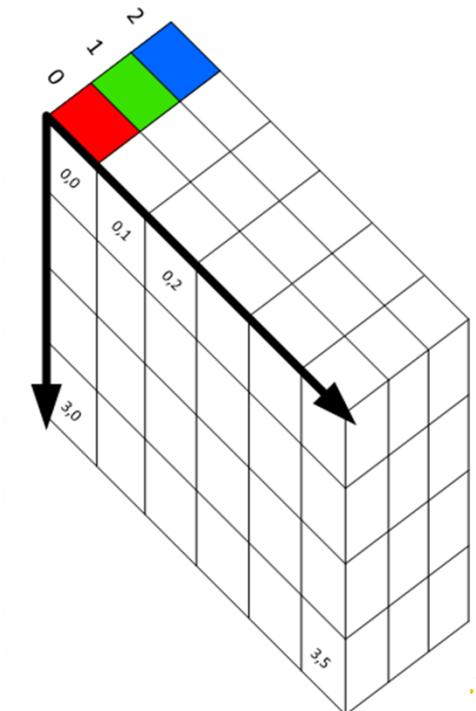


How do Computers Store Images?



- A Color image would consist of 3 channels (RGB) - Right
- And a Grayscale image would consist of one channel - Below

255	255	255	255	255	255	255	220	146	237	255	255	255	255	255
255	255	255	255	255	255	255	91	68	170	255	255	255	255	255
255	255	255	255	255	253	238	30	62	220	255	255	255	255	255
255	255	255	255	255	218	145	64	241	255	255	255	255	255	255
255	255	255	255	255	81	72	73	146	244	255	255	255	255	255
255	255	255	255	255	72	64	71	184	255	255	255	255	255	255
255	255	255	255	255	63	68	68	188	255	255	255	255	255	255
255	255	255	255	255	71	71	71	187	255	255	255	255	255	255
255	255	255	255	255	68	68	68	190	255	255	255	255	255	255
255	255	255	255	255	83	83	83	160	255	255	255	255	255	255
255	255	255	255	255	181	181	181	184	255	255	255	255	255	255



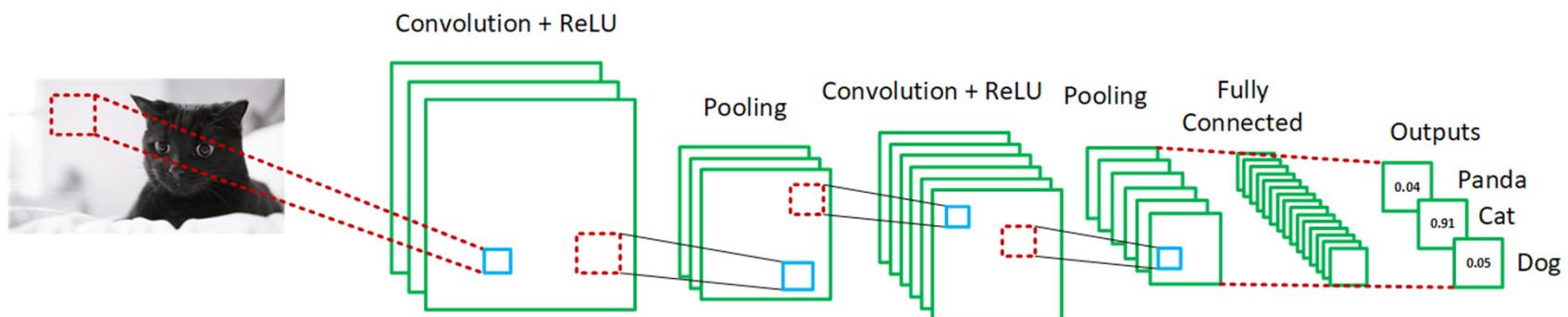


Why NNs Don't Scale Well to Image Data

- Imagine a simple image classifier that takes color images of size 64×64 (height, width).
- The input size to our NN would be $64 \times 64 \times 3 = 12,288$
- Therefore, our input layer will thus have 12,288 weights. While not an insanely large amount, imagine using input images of 640×480 ? You'd have 921,600 weights! Add some more hidden layers and you'll see how fast this can grow out of control. Leading to very long training times
- However, our input is image data, data that consist of patterns and correlated inputs. There must be a way to take advantage of this.



Introducing CNNs





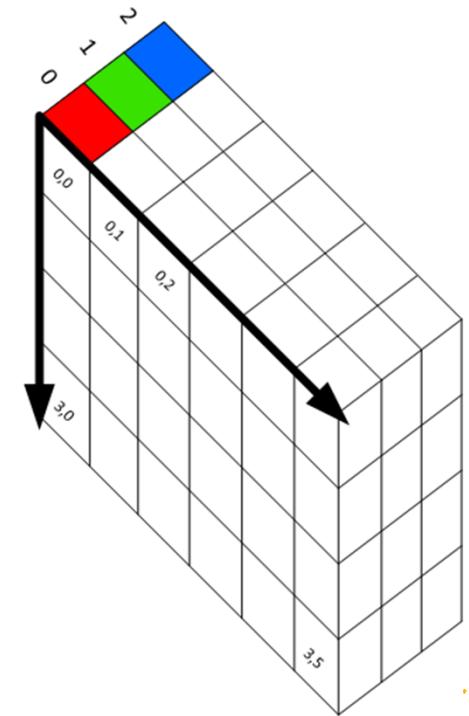
Why of 3D Layers?

- Allow us to use convolutions to learn image features.
- Use far less weights in our deep network, allowing for significantly faster training.



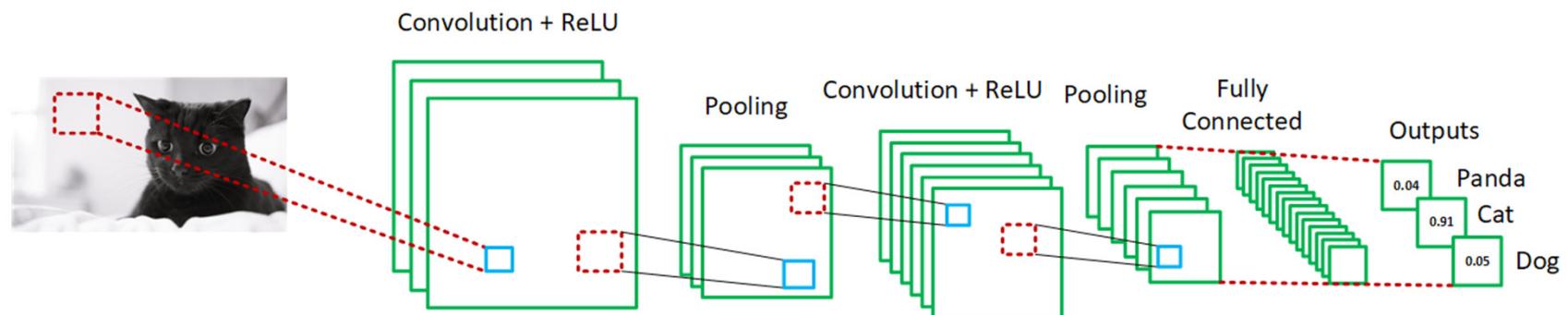
CNN's use a 3D Volume Arrangement for it's Neurons

- Because our input data is an image, we can constrain or design our Neural Network to better suit this type of data
- Thus, we arrange our layers in 3 Dimensions. Why 3?
- Because of image data consists of:
 - Height
 - Width
 - Depth (RGB) our colors components





CNN Layers



- Input
- Convolution Layer
- ReLU Layer
- Pool Layer
- Fully Connected Layer



It's all in the name

- The Convolution Layer is the most significant part of a CNN as it is this layer that learns image features which aid our classifier.
- But what exactly is a Convolution?

7.2

Convolutions & Image Features

How convolutions works and how they learn image features



Image Features

- Before dive into Convolutions, let's discuss Image Features

Image Features

- Image Features are simply interesting areas of an image. Examples:
 - Edges
 - Colors
 - Patterns/Shapes





Before CNN's Feature Engineering was done Manually

- In the old days, we computer scientists had to understand our images and select appropriate features manually such as:
 - Histogram of Gradients
 - Color Histograms
 - Binarization
 - And many more!
- This process was tedious, and highly dependent on the original dataset (meaning the feature engineering done for one set of images wouldn't always be appropriate to another image dataset).



Examples of Image Features



- Example filters learned by Krizhevsky et al.



What are Convolutions?

- Convolution is a mathematical term to describe the process of combining two functions to produce a **third function**.
- This third function or the output is called a Feature Map
- A convolution is the action of using a **filter** or **kernel** that is applied to the input. In our case, the input being our input image.



The Convolution Process

- Convolutions are executed by sliding the filter or kernel over the input image.
- This sliding process is a simple matrix multiplication or dot product.



The Convolution Process

0	123	127	167	124
54	45	124	136	163
64	76	65	241	188
36	235	222	215	171
23	221	124	199	34

● Input

2	-1	2
-1	3	-1
2	0	2

● Convolution

● Output or Feature Map



The Convolution Process

1	0	1	0	1
1	0	0	1	1
0	1	1	0	0
1	0	0	1	0
0	0	1	1	0

● Input

0	1	0
1	0	-1
0	1	0

● Convolution

● Output or Feature Map

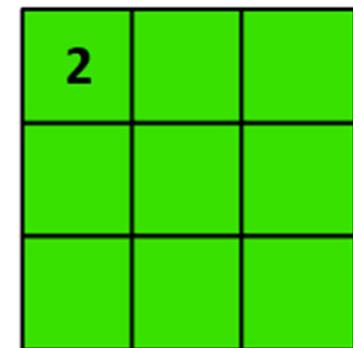


Applying out Kernel / Filter

1x0	0x1	1x0	0	1
1x1	0x0	0x-1	1	1
0x0	1x1	1x0	0	0
1	0	0	1	0
0	0	1	1	0

$$(1 \times 0) + (0 \times 1) + (1 \times 0) + \\ (1 \times 1) + (0 \times 0) + (0 \times -1) + \\ (0 \times 0) + (1 \times 1) + (1 \times 0)$$

$$0+0+0+ \\ 1+0+0+ \\ 0+1+0 = 2$$



- Output or Feature Map



Applying out Kernel / Filter - Sliding

1	0x0	1x1	0x0	1
1	0x1	0x0	1x-1	1
0	1x0	1x1	0x1	0
1	0	0	1	0
0	0	1	1	0

$(0x0)+(1x1)+(0x0)+$
 $(0x1)+(0x0)+(1x-1)+$
 $(1x0)+(1x1)+(0x1)$

$0+1+0+$
 $0+0-1+$
 $0+1+0 = 1$

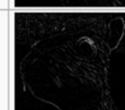
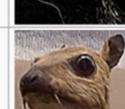
2	1	



What Are the Effects of the Kernel?

- Depending on the values on the kernel matrix, we produce different feature maps. Applying our kernel produces scalar outputs as we just saw.
- Convolving with different kernels produces interesting feature maps that can be used to detect different features.
- Convolution keeps the spatial relationship between pixels by learning image features over the small segments we pass over on the input image.

Examples of Kernel Feature Maps

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	





A Convolution Operation in Action



Source – Deep Learning Methods for Vision
https://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/

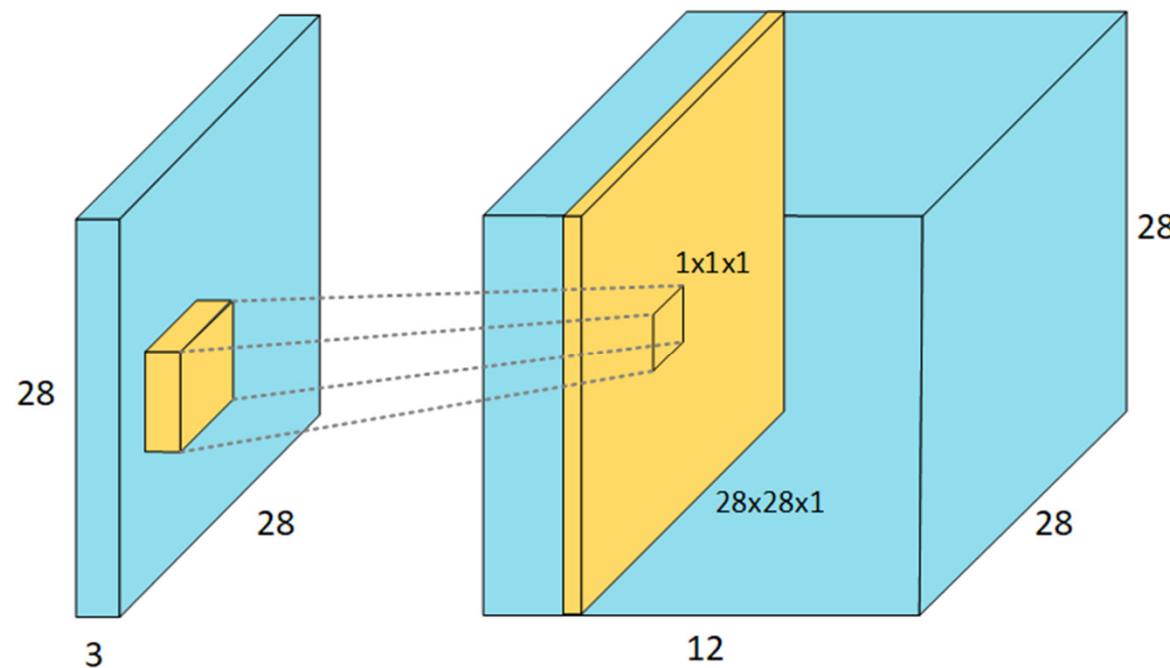


We Need Many Filters, Not Just One

- We can design our CNNs to use as many filters as we wish (within reason).
- Assume we use 12 filters. How can we visualize that?



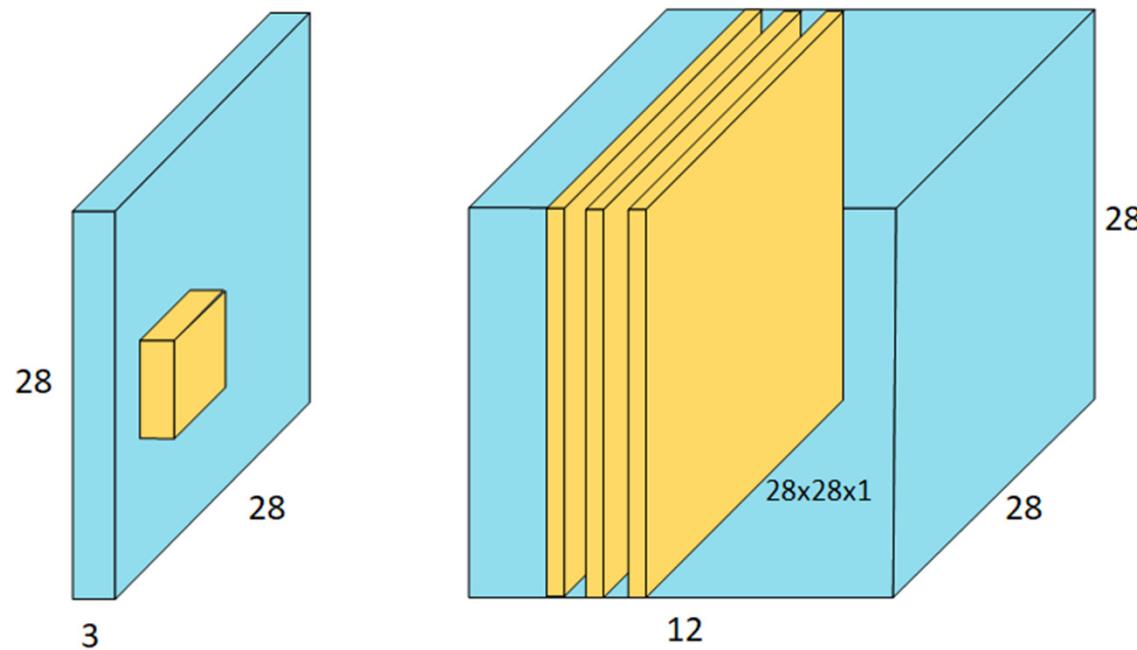
Illustration of our Multiple 3D Kernels/Filters



196



Illustration of our Multiple 3D Kernels/Filters





The Outputs of Our Convolutions

- From our last slide we saw by applying **12** filters (of size $3 \times 3 \times 3$) to our input image (of size $28 \times 28 \times 3$) we produced **12 Feature Maps** (also called Activation Maps).
- These outputs are stacked together and treated as another 3D Matrix (in our example, our output was of size $28 \times 28 \times 12$).
- This 3D output is the **input** to our next layer in our CNN



Feature/Activation Maps and Image Features

- Each cell in our Activation Map Matrix can be considered a feature extractor or neuron that looks at a specific region of the image.
- In the first few layers, our neurons activation when edges or other low level features are detected. In Deeper Layers, our neurons will be able to detect high-level or 'big picture' features or patterns such as a bicycle, face, cat etc.

In the next section we examine the hyper parameters we use to create our Activation Matrix



- You've seen us use an arbitrary filter size of 3x3
- How do we choose this and what else can we tweak?

7.3

Depth, Stride and Padding



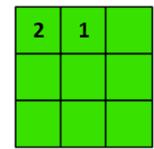
Designing Feature Maps

- Remember Feature or Activation Maps were the output of applying our convolution operator.
- We applied a **3x3 filter or kernel** in our example.
- But, does the filter have to be 3x3? How many filters do we need? Do we need to pass over pixel by pixel?

1	0x0	1x1	0x0	1
1	0x1	0x0	1x-1	1
0	1x0	1x1	0	0
1	0	0	1	0
0	0	1	1	0

$$(0x0)+(1x1)+(0x0)+\\(0x1)+(0x0)+(1x-1)+\\(1x0)+(1x1)+(0x0)$$

$$0+1+0+\\0+0-1+\\0+1+0 = 1$$





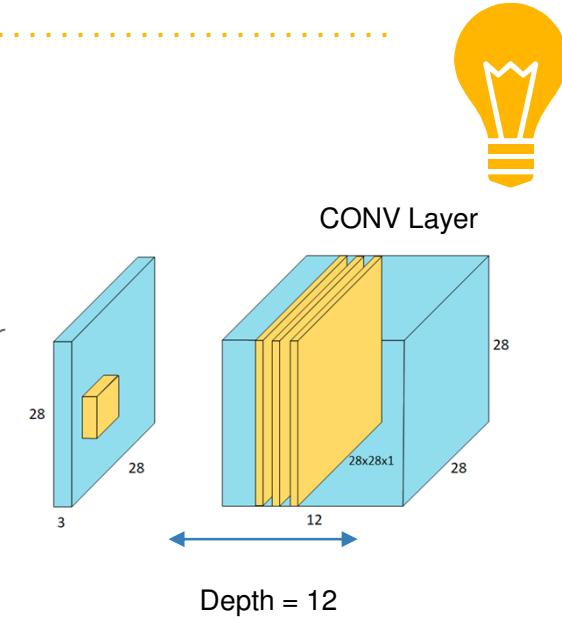
Designing Feature Maps

By tweaking the following **parameters** to control the size of our **feature maps**.

- Kernel Size ($K \times K$)
- Depth
- Stride
- Padding

Depth

- Depth describes the **number of filters used**. It does not relate the image depth (3 channels) nor does it describe the number of hidden layers in our CNN.
- Each filter learns different feature maps that are activated in the presence of different image features (edges, patterns, colors, layouts)



Stride



- Stride simply refers to the step size we take when we slide our kernel the input image.
- In the last example we used a stride of 1



Using a Stride of 1

1x0	0x1	1x0	0	1
1x1	0x0	0x-1	1	1
0x0	1x1	1x0	0	0
1	0	0	1	0
0	0	1	1	0



Using a Stride of 1

1	0x0	1x1	0x0	1
1	0x1	0x0	1x-1	1
0	1x0	1x1	0	0
1	0	0	1	0
0	0	1	1	0



Using a Stride of 1

1	0	1x0	0x1	1x0
1	0	0x1	1x0	1x-1
0	1	1x0	0x1	0x1
1	0	0	1	0
0	0	1	1	0



Using a Stride of 1

- Start on the second line
- As we can see we make 9 passes moving one step at a time over the input matrix.
- A stride of 1 gives us a 3×3 output

1	0	1	0	1
1x0	0x1	0x0	1	1
0x1	1x0	1x-1	0	0
1x0	0x1	0x0	1	0
0	0	1	1	0



Let's try a Stride of 2

- Using a stride of 2 we start off the same

1x0	0x1	1x0	0	1
1x1	0x0	0x-1	1	1
0x0	1x1	1x0	0	0
1	0	0	1	0
0	0	1	1	0



Let's try a Stride of 2

- But then we jump 2 places instead of one, as we slide horizontally.

1	0	1x0	0x1	1x0
1	0	0x1	1x0	1x-1
0	1	1x0	0x1	0x1
1	0	0	1	0
0	0	1	1	0



Let's try a Stride of 2

- As we move down, we also skip two spots.



1	0	1	0	1
1	0	0	1	1
0x0	1x1	1x0	0	0
1x1	0x0	0x-1	1	0
0x0	0x1	1x0	1	0



Let's try a Stride of 2

- Therefore, with a stride of 2 we only make 4 passes over the input. Thus producing a smaller output of 2x2

1	0	1	0	1
1	0	0	1	1
0	1	0x0	0x1	0x0
1	0	1x1	1x0	0x-1
0	0	0x0	1x1	0x0

What do we use Stride for?



- Stride controls the size of the Convolution Layer output.
- Using a larger Stride produce less overlap in kernels.
- Stride is one of the methods we can control the spatial input size i.e. the volume of the inputs into the other layers of our CNN,

Zero-Padding

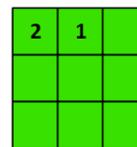


- Zero-padding is a very simple concept that refers to a border we apply to the input volume. Why is this volume needed?
- Remember in our first example, with stride set to 1, and our output of feature map was a 3x3 matrix. We haven't discussed deep networks much yet, but imagine we had multiple Convolution layers. You can quickly see that even with a stride of 1, we end up with a tiny output matrix quickly.

1	0x0	1x1	0x0	1
1	0x1	0x0	1x-1	1
0	1x0	1x1	0	0
1	0	0	1	0
0	0	1	1	0

$(0x0)+(1x1)+(0x0)+$
 $(0x1)+(0x0)+(1x-1)+$
 $(1x0)+(1x1)+(0x0)$

$0+1+0+$
 $0+0-1+$
 $0+1+0 = 1$



Zero-Padding Illustrated.



- We add a border of 0s around our input. Basically this is equivalent of adding a black border around an image
- We can set our Padding P to 2 if needed.

0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	1	0	0	1	1	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	0	0	1	1	0	0
0	0	0	0	0	0	0

Zero-Padding makes our output larger - Illustrated



0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	1	0	0	1	1	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	0	0	1	1	0	0
0	0	0	0	0	0	0

1				

Zero-Padding makes our output larger - Illustrated



0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	1	0	0	1	1	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	0	0	1	1	0	0
0	0	0	0	0	0	0

1	2			



Calculating our Convolution Output

- Our CONV Layer Output size
 - Kernel/Filter Size – K
 - Depth – D
 - Stride – S
 - Zero Padding – P
 - Input Image Size - I
- To ensure our filters cover the full input image symmetrically, we used the following equation to do this sanity check. Once the result of this equation is an integer, our settings are valid.
- $\frac{((I-K+2P))}{S} + 1$ for example using our previous design with a stride of 1.
- $\frac{((5-3+(2 \times 1))}{1} + 1 = \frac{(2+2)}{1} + 1 = 5 \text{ (Integer)}$

7.4

ReLU

The Activation Layer of Choice for CNNs



Activation Layers

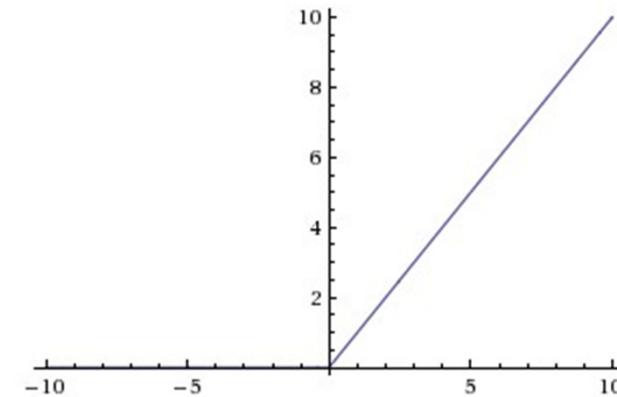
- Similarly as we saw in NN, we need to introduce non-linearity into our model (the convolution process is linear), we need to apply an activation function to the output of our CONV Layer



ReLU is the Activation Function of Choice

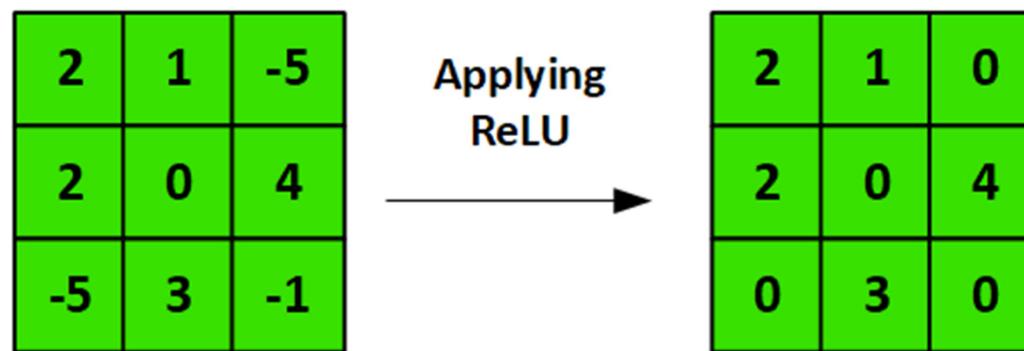
- ReLU or Rectified Linear Unit simply changes all the negative values to 0 while leaving the positive values unchanged.

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



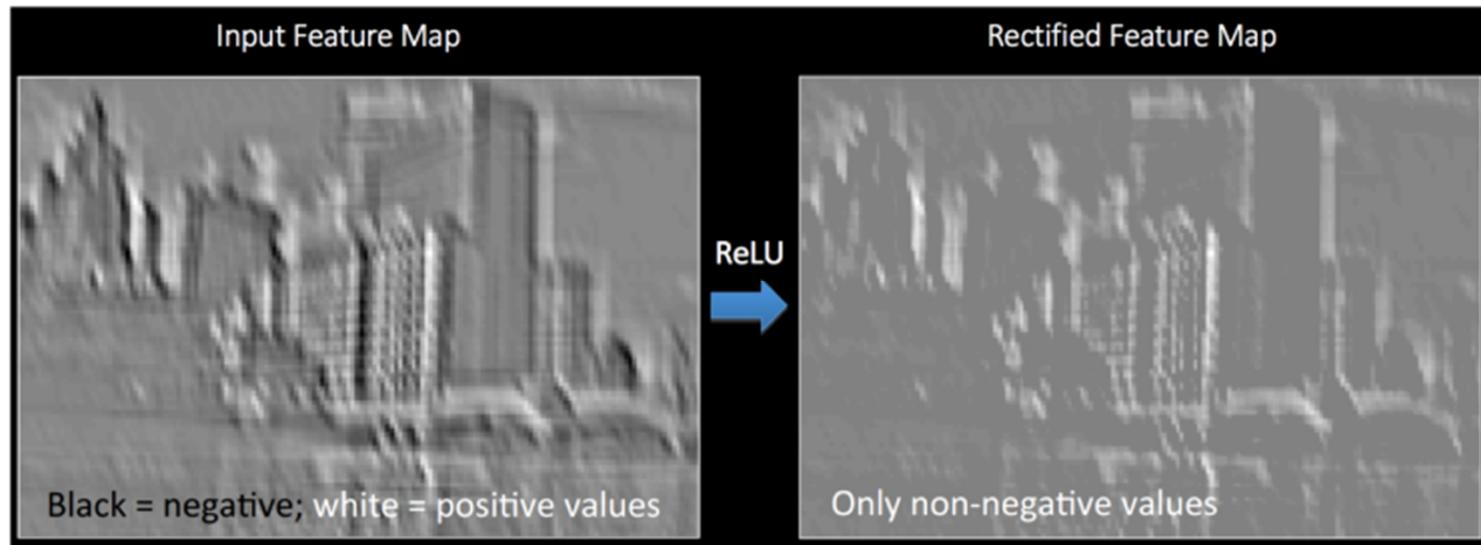


Applying ReLU





ReLU Visualized



Source - http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf

7.5

Pooling

All about next layer in CNN's Pooling or Subsampling



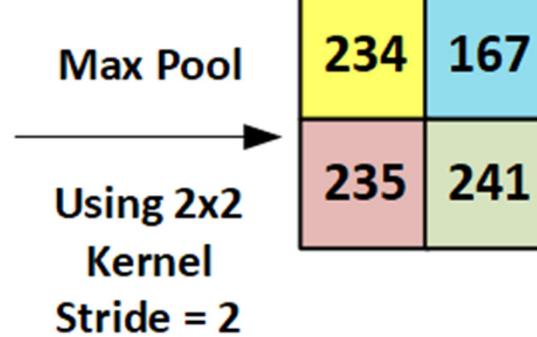
Pooling

- Pooling, also known as subsampling or downsampling, is a simple process where we **reduce the size** or dimensionality of the Feature Map.
- The purpose of this reduction is to **reduce the number of parameters** needed to train, whilst retaining the most important features.
- There are 3 types of Pooling we can apply, Max, Average and Sum.



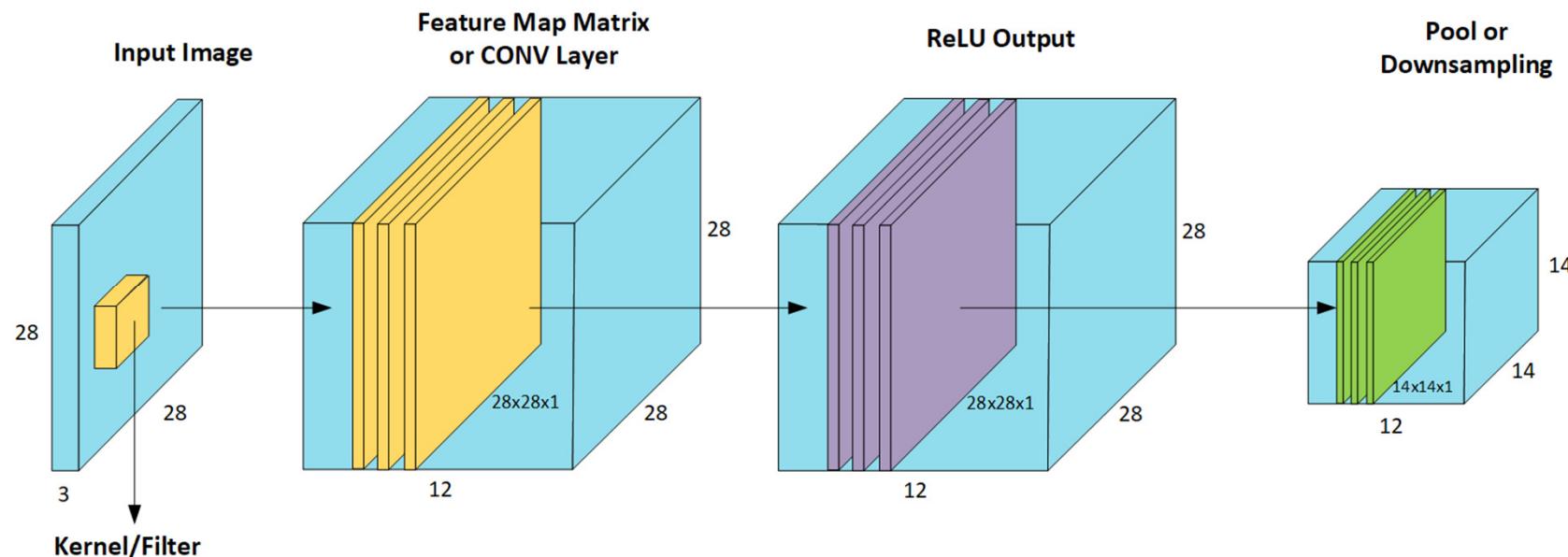
Example of Max Pooling

0	123	127	167
54	234	124	0
64	76	0	241
0	235	222	0





Our Layers so far





More on Pooling

- Typically Pooling is done using **2x2** windows with a **Stride of 2** and **with no padding applied**
- For smaller input images, for larger images we use larger pool windows of **3x3**.
- Using the above settings, pooling has the effect of reducing the dimensionality (width and height) of the previous layer by half and thus removing **¾** or **75%** of the activations seen in the previous layer.



More on Pooling – 2

- Makes our model more invariant to small to minor transformations or distortions in our input image since we're now averaging or taking the max output from small area of our image.

7.6

The Fully Connected Layer

The importance of the FC Layer



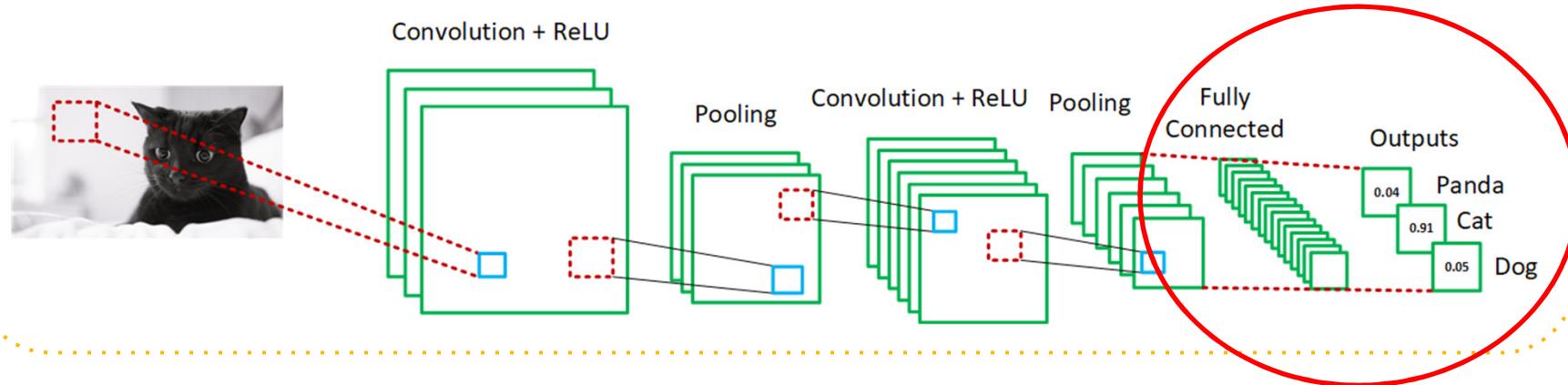
The Fully Connected (FC) Layer

- Previously we saw in our NN that all layers were Fully Connected.
- Fully Connected simply means all nodes in one layer are connected to the outputs of the next layer.



The Final Activation Function

- The FC Layer outputs the class probabilities, where each class is assigned a probability.
- All probabilities must sum to 1, e.g (0.2, 0.5, 0.3)





Soft Max

- The activation function used to produce these probabilities is the Soft Max Function as it turns the outputs of the FC layer (last layer) into probabilities.
- Example:
 - Lets say the output of the last FC layer was [2, 1, 1]
 - Applying the softmax 'squashes' these real value numbers into probabilities that sum to one: the output would therefore be: [0.7, 0.2, 0.1]

7.7

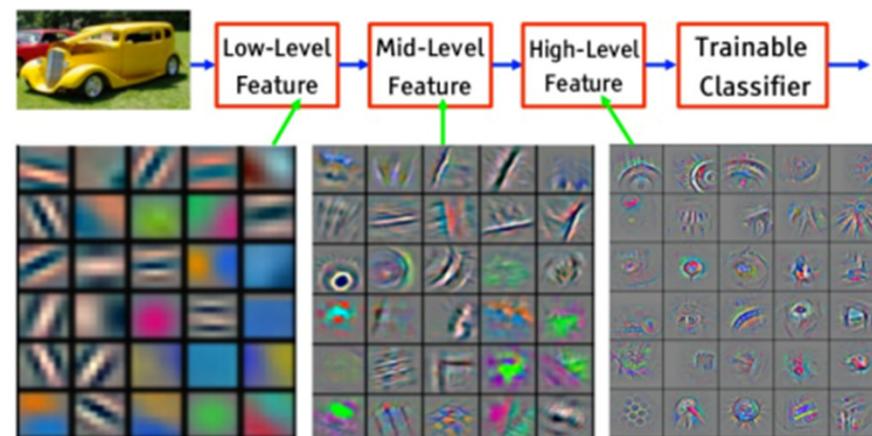
Training CNNs

Let's review the entire process of training a CNN



This is a quick review of CNNs

- CNNs are the NN of choice for Image Classification.
- It learns spatial image features, but because of the Max Pooling and multiple filters, is somewhat scale and distortion invariant

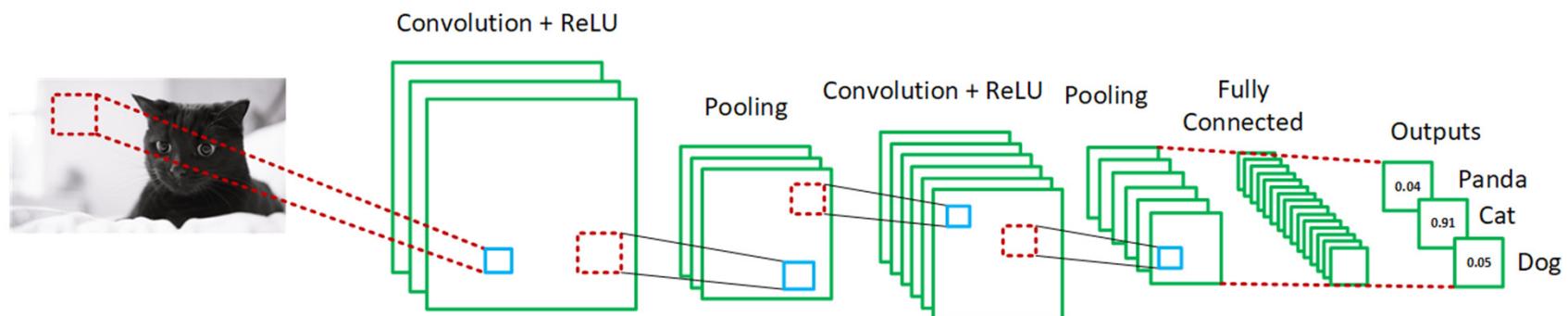


Source -
<http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf>



This is a quick review of CNNs

- It involves 4 Distinct Layers
 - Convolution
 - ReLU
 - Pooling
 - Fully Connected





Training CNNs

- Just like NN, training CNNs is essentially the same once we setup our Network Layers. The flow follows the following steps
 1. Random Weight initialization in the Convolution Kernels
 2. Forward Propagates an image through our network (Input -> CONV -> ReLU -> Pooling -> FC)
 3. Calculate the Total Error e.g. say we got an output of [0.2, 0.4, 0.4] while the true probabilities were [0, 1, 0]
 4. Use Back Propagation to update our gradients (i.e. the weights in the kernel filters) via gradient descent.
 5. Keep propagating all images through our network till an Epoch is complete
 6. Keep completing Epochs till our loss and accuracy are satisfactory

7.8

Designing your own CNNs

Layer Patterns



Designing CNNs

- Basic CNN's all follow a simple set of rules and layer sequences.





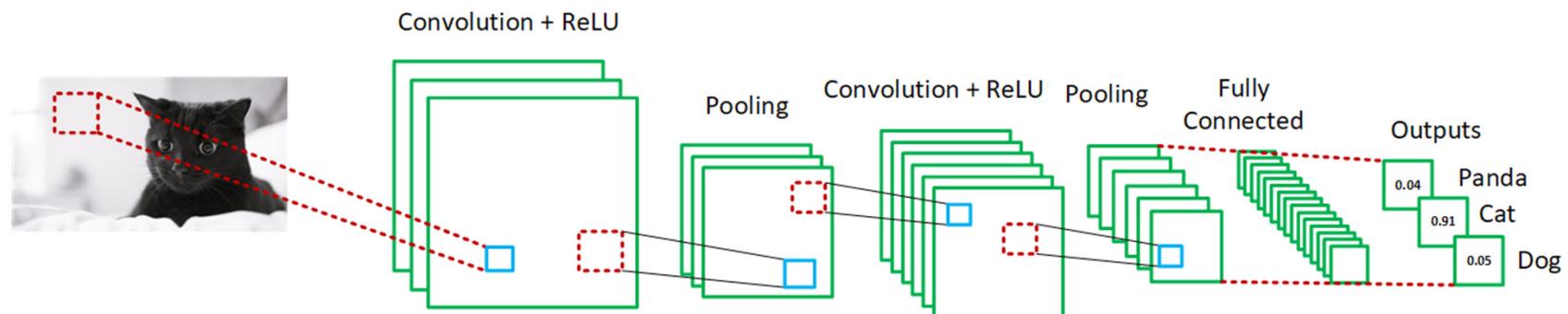
Basic CNN Design Rules 1

- Input Layer typically Square e.g. $28 \times 28 \times 3$, or $64 \times 64 \times 3$ (this isn't necessary but simplifies our design and speeds up our matrix calculations)
- Input should be divisible by at least 4 which allows for downsampling
- Filters are typically small either 3×3 or 5×5
- Stride is typically 1 or 2 (if inputs are large)
- Zero padding is used typically to allow the output Conv layer to be the same size as the input.
- Pool kernel size is typically 2×2
- Dropout is a very useful technique to avoid overfitting in CNNs



Basic CNN Design Rules 2

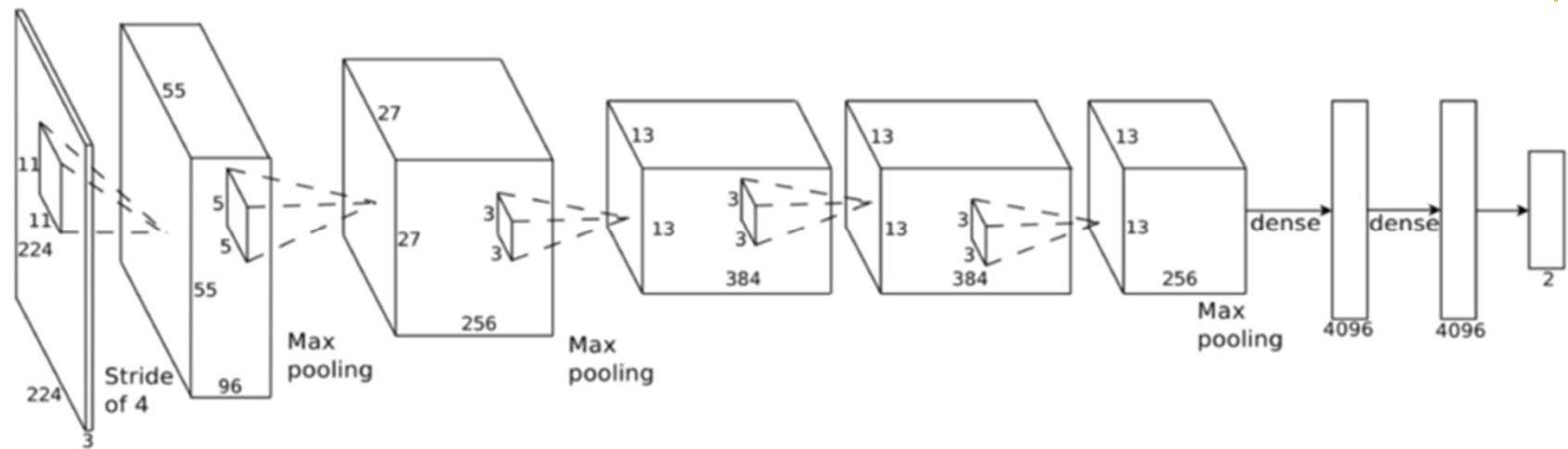
- The more hidden layers the more features, particularly high level features a CNN can learn. I like to use a minimum of 2, which is shown in the diagram below.
- Its flow is: Input -> Conv -> ReLU -> Pool -> Conv -> ReLU -> Pool -> FC - Output





Examples of some CNNs will be creating in this course

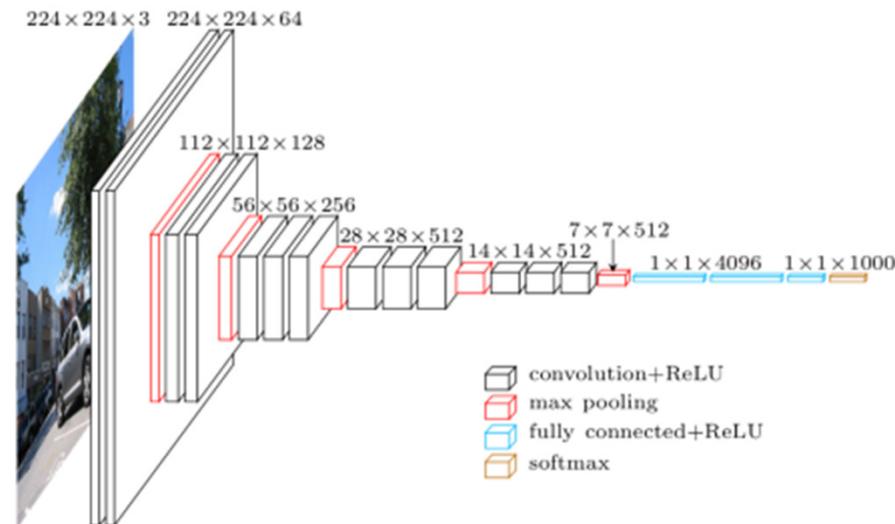
- AlexNet





Examples of some CNNs will be creating in this course

- VGNet (VG16 & VG19)





Creating Convolutional Neural Networks in Keras

8.0

Building a CNN in Keras



Building a CNN in Keras

- **8.1 Introduction to Keras & Tensorflow**
- **8.2 Building a Handwriting Recognition CNN**
- **8.3 Loading Our Data**
- **8.4 Getting our data in 'Shape'**
- **8.5 Hot One Encoding**
- **8.6. Building & Compiling Our Model**
- **8.7 Training Our Classifier**
- **8.8 Plotting Loss and Accuracy Charts**
- **8.9 Saving and Loading Your Model**
- **8.10 Displaying Your Model Visually**
- **8.11 Building a Simple Image Classifier using CIFAR10**

8.1

Introduction to Keras & Tensorflow

What is Keras?



- Keras is a high-level neural network API for Python
- It makes constructing Neural Networks (all types but especially CNNs and recurrent NNs) extremely easy and modular.
- It has the ability to use TensorFlow, CNTK or Theano back ends
- It was developed by **François Chollet** and has been a tremendous success in making deep learning more accessible

What is TensorFlow?



- We just mentioned Keras is an API that uses TensorFlow (among others) as its backend.
- TensorFlow is an open source library created by the Google Brain team in 2015.
- It is an extremely powerful **Machine Learning** framework that is used for high performance numerical computation across a variety of platforms such as CPUs, GPUs and TPUs.
- It too has a python API that makes it very accessible and easy to use



Why use Keras instead of pure TensorFlow?

- Keras is **extremely easy to use** and follows a pythonic style of coding
- **Modularity** – meaning, we can easily add and remove building blocks of Neural Network code, while easily changing cost functions, optimizers, initialization schemes, activation functions, regularization schemes
- Allows us to build **powerful Neural Networks quickly and efficiently**
- Works in **Python**
- TensorFlow is not as user friendly and modular as Keras
- Unless you're doing academic research or constructing ground breaking Neural Networks, there is no need to use pure TensorFlow.



Composing a Neural Network in Keras

- First we import our Sequential model type
 - Sequential model is a linear stack of layers

```
from keras.models import Sequential  
  
model = Sequential()
```



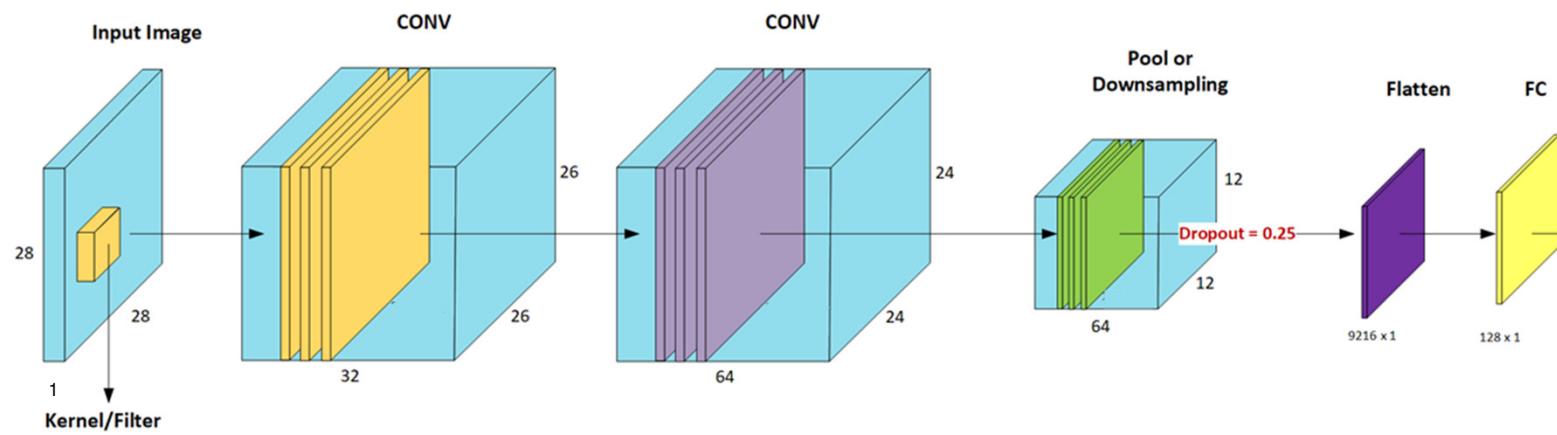
Creating our Convolution Layer

- To add our Conv layer, we use `model.add(Conv2D)`
 - Specifying firstly the number of kernels or filters
 - The kernel size
 - The input shape

```
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Our Model So Far



That's it, we can now compile our model



- Compiling simply creates an object that stores our model we've just created.
- We can specify our loss algorithm, optimizer and define our performance metrics. Additionally, we can specify parameters for our optimizer such as learning rates and momentum)

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



Fit or Train our Model

- Following the language from the most established Python ML library (Sklearn), we can fit or train our model with the follow code.

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```



Evaluate our Model and Generate Predictions

- We can now simply evaluate our model's performance with this line of code.

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

- Or generate predictions by feeding our Test data to the *model.predict* function

```
classes = model.predict(x_test, batch_size=128)
```

8.2

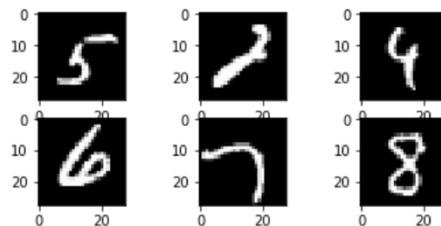
Building a Handwriting Recognition CNN

Let's build a power handwritten digit reconigition Convolutional Neural Network in Keras



Our First CNN – Handwriting Recognition on the MNIST Dataset

- The MNIST dataset is ubiquitous with Computer Vision as it is one of the most famous computer vision challenges.
- It is a fairly large dataset consisting of 60,000 training images and 10,000 test images.

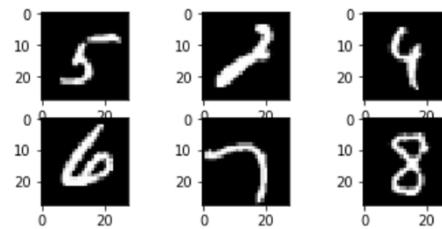


- Read more about MNIST here:
 - <http://yann.lecun.com/exdb/mnist/>
 - https://en.wikipedia.org/wiki/MNIST_database



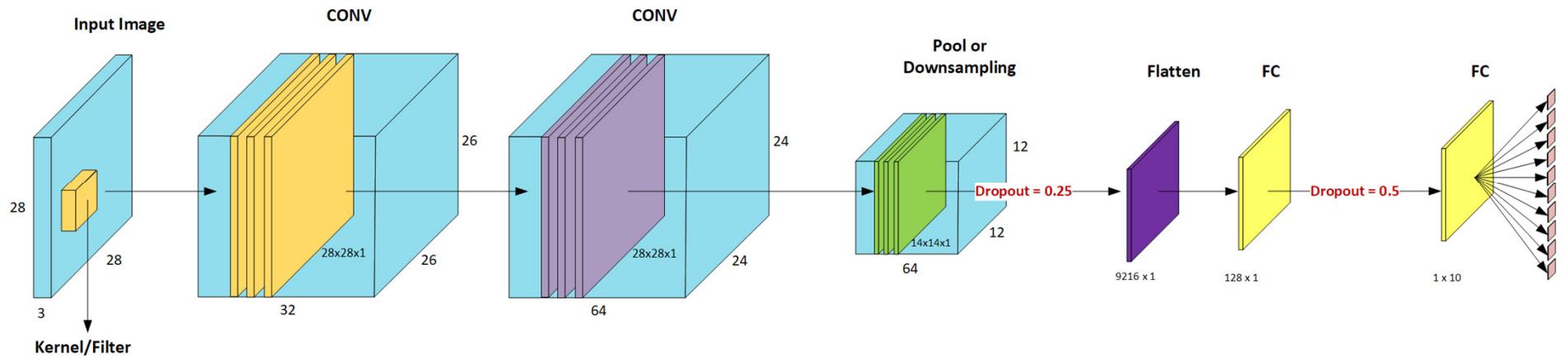
Problem and Aim

- The MNIST dataset was developed as the US Postal Service needed a way to automatically read written postcodes on mail.
- The aim of our classifier is simple, take the digits in the format provided and correctly identify the digit that was written





We are going to build this CNN in Keras!



- And achieve ~99.25% Accuracy

8.3

Loading Our Data



Keras has build in data loaders

- Loading the data is a simple but obviously integral first step in creating a deep learning model.
- Fortunately, Keras has some build-in data loaders that are simple to execute.
- Data is stored in an array

```
from keras.datasets import mnist  
  
# loads the MNIST dataset  
(x_train, y_train), (x_test, y_test) = mnist.load_data()  
  
print (x_train.shape)  
(60000, 28, 28)
```

8.4

Getting our data in 'Shape'



Input Image Shape for Keras

- One of the confusing things new comers face when using Keras is getting their dataset in the **correct shape (dimensionality) required for Keras**.
- When first load our dataset in Keras it comes in the form of 60,000 images, 28 x 28. Inspecting this in python gives us:

```
from keras.datasets import mnist

# loads the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

print (x_train.shape)
(60000, 28, 28)
```

- However, the input required by Keras requires us to define it in the following format:
 - Number of Samples, Rows, Cols, Depth**
 - Therefore, since our MNIST dataset is **grayscale**, we need it in the form of:
 - 60000, 28, 28, 1** (*if it were in color, it would be 60000, 28, 28, 3*)
- Using Numpy's **reshape** function, we can easily add that 4th dimension to our data

```
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
```

8.5

Hot One Encoding



Transformations on Training and Test Image Data, But What about the Label Data, `y_train` and `y_test`?

- For `x_train` and `x_test` we:
 - Added a 4th Dimension going from (60000, 28, 28) to (60000, 28, 28, 1)
 - Changed it to Float32 data type
 - Normalized it between 0 and 1 (by dividing by 255)



Hot Encode Outputs

- $y_{train} = [0, 6, 4, 7, 2, 4, 1, 0, 5, 8, \dots]$

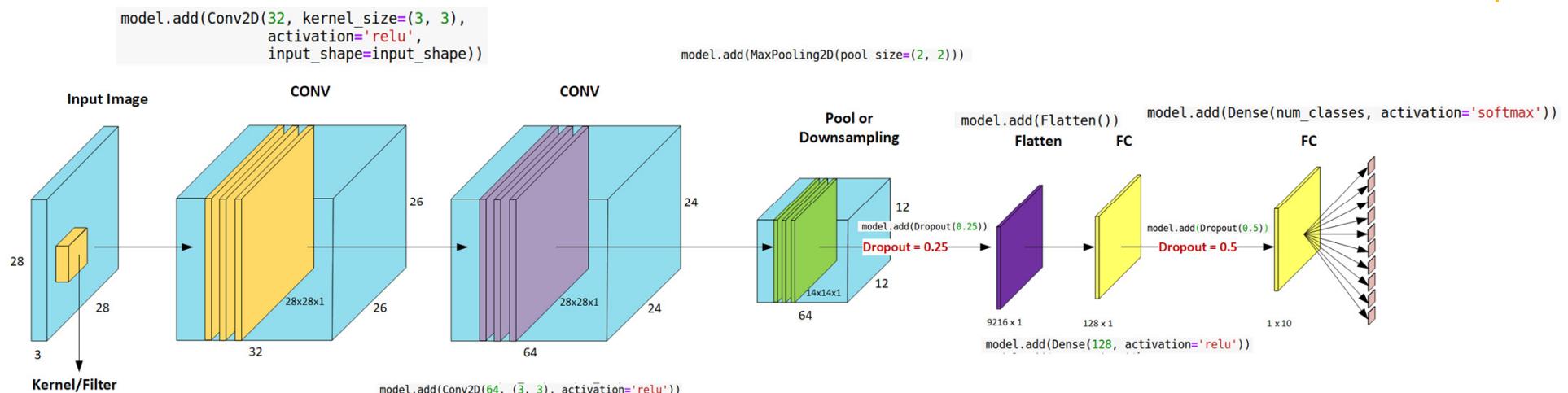
Label	0	1	2	3	4	5	6	7	8	9
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0		1	0	0	0

8.6

Building & Compiling Our Model



Our CNN Illustrated with Keras code



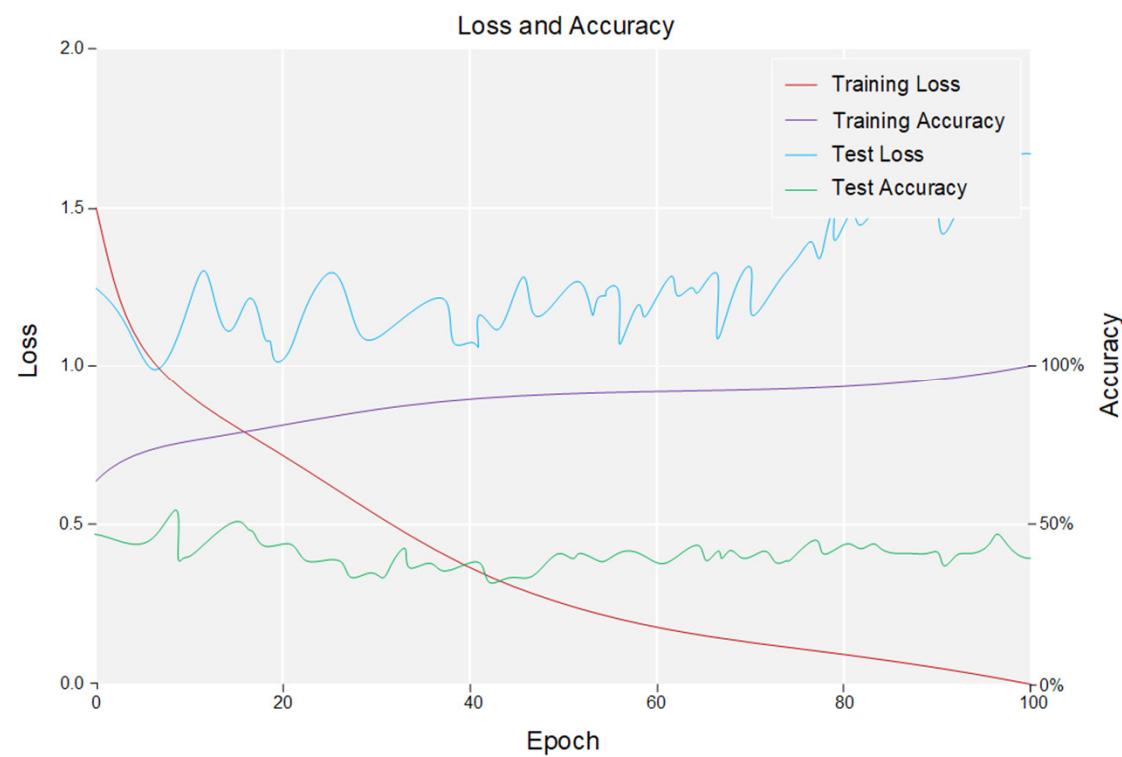
8.7

Training Our Classifier

8.8

Plotting Loss and Accuracy Charts

Loss and Accuracy



8.9

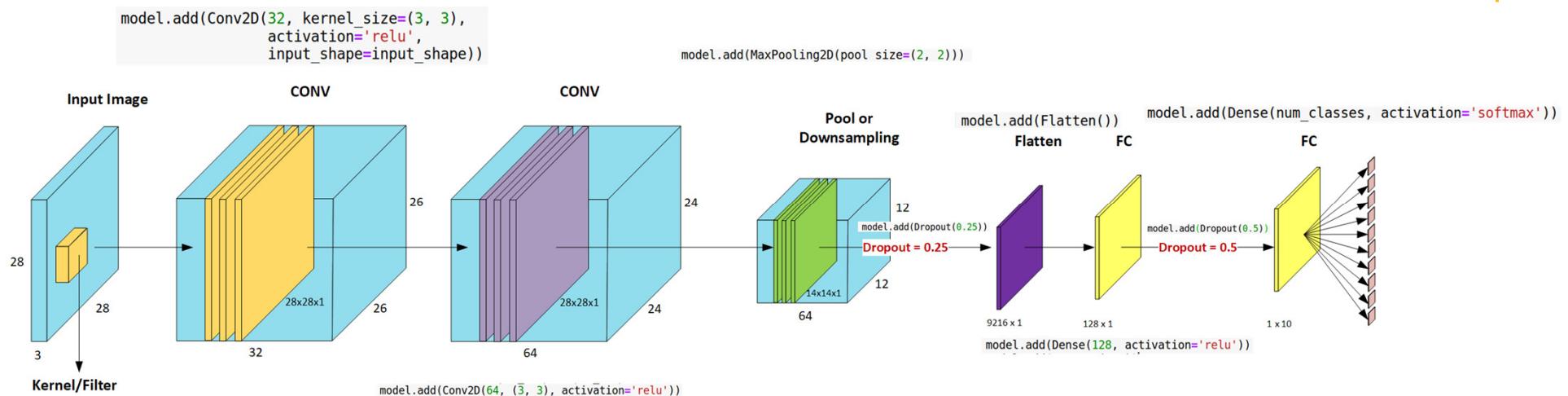
Saving and Loading Your Model

8.10

Displaying Your Model Visually



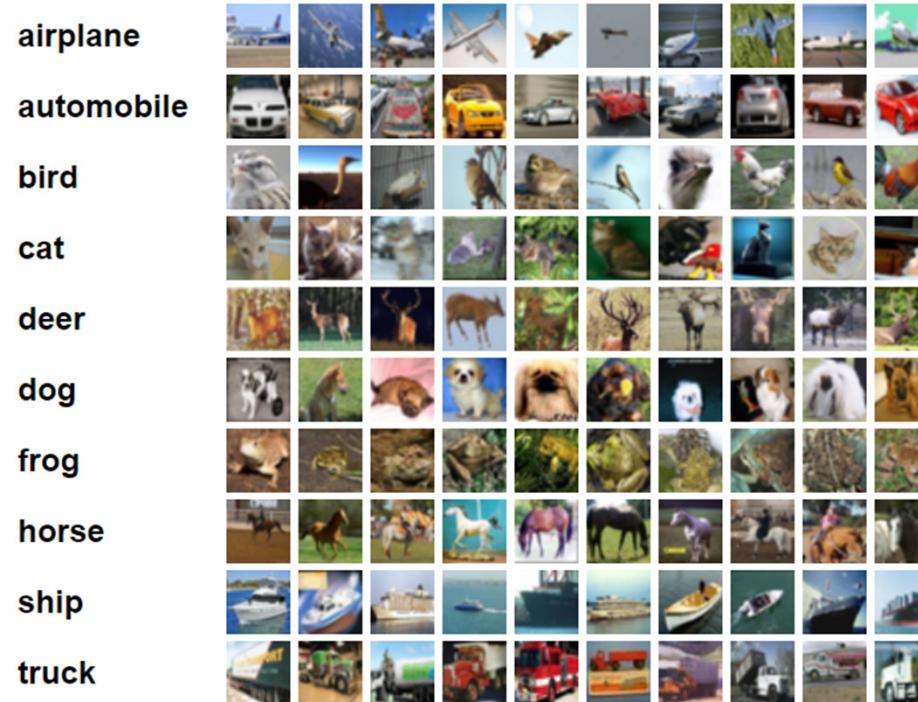
Our CNN Illustrated with Keras code



8.11

Building a Simple Image Classifier using CIFAR10

CIFAR-10



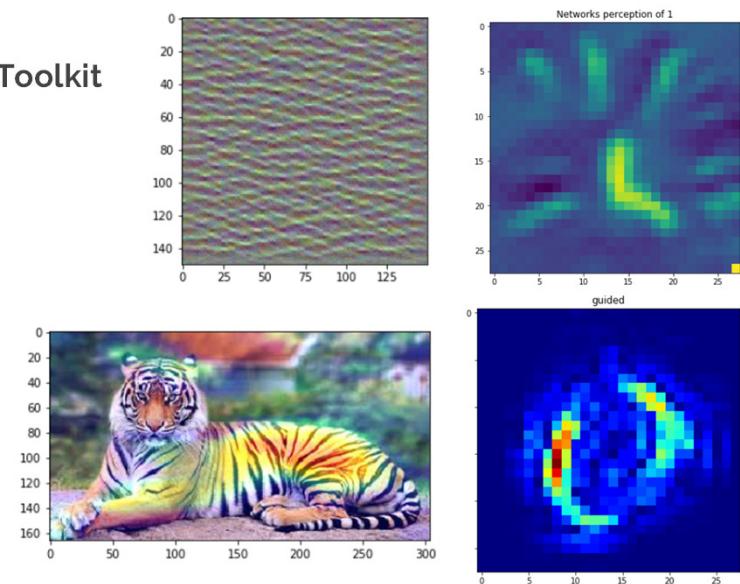
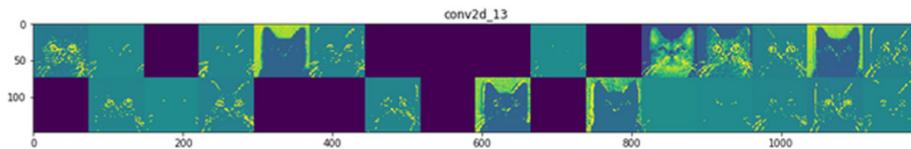
9.0

Visualizing What CNNs 'see' & Filter Visualizations



Visualizing What CNNs 'see' & Filter Visualizations

- 9.1 Activation Maximization using Keras Visualization Toolkit
- 9.2 Saliency Maps & Class Activation Maps
- 9.3 Filter Visualizations
- 9.4 Heat Map Visualizations of Class Activations



9.1

Activation Maximization using Keras Visualization Toolkit



What does your CNN think Object Classes look like?

Let's consider the MNIST Handwritten Digit Dataset.

- After training on thousands of images, what does your CNN actually think each digit looks like?

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

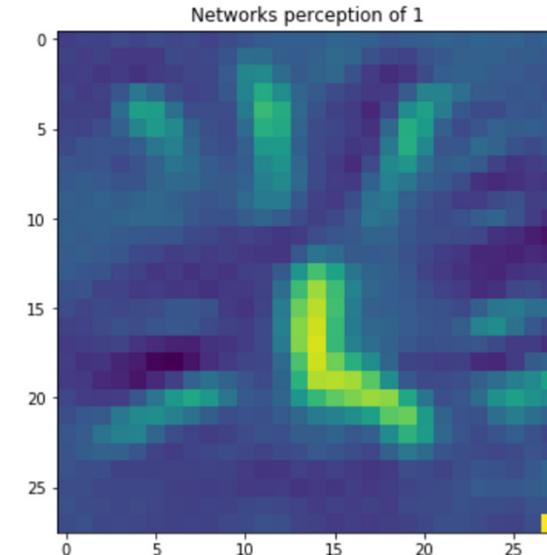
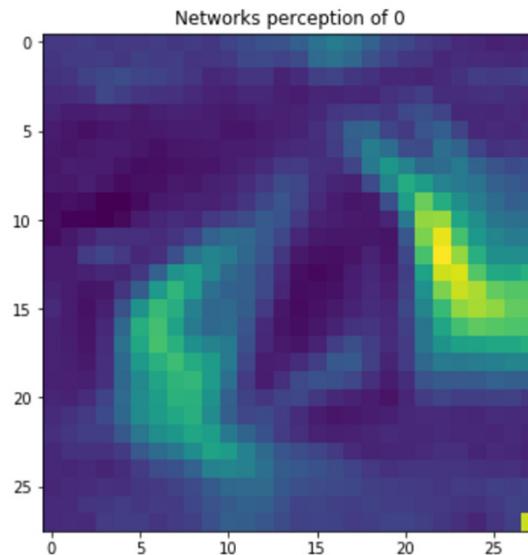


We can generate visualizations that show this, these visuals are called Activation Visualization

- Activation Visualizations show how successive Convolutional layers transform their input.
- We do this by visualizing intermediate activations, which display the feature maps that are output by various convolution and pooling layers in a network, given a certain input. The output of these layer is called an **Activation**.
- This shows us how an input is decomposed into the different filters learned by the network.



We are going to generate these plots
visualize our network perception





What are we looking at?

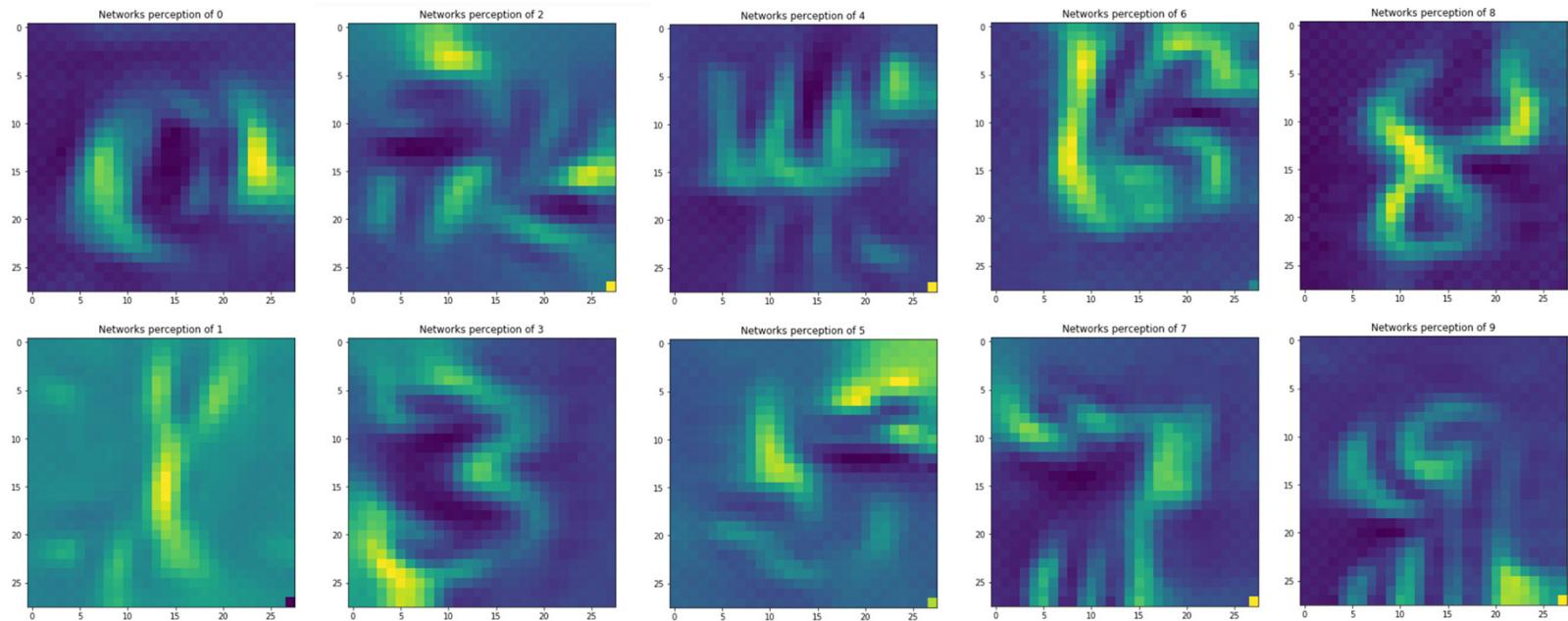
- We are generating an input image that maximizes the filter output activations. Thus we are computing,

$$\frac{\partial \text{ActivationMaximizationLoss}}{\partial \text{input}}$$

- and using that estimate to update the input.
- Activation Maximization loss simply outputs small values for large filter activations (we are minimizing losses during gradient descent iterations). This allows us to understand **what sort of input patterns activate a particular filter**. For example, there could be an eye filter that activates for the presence of eye within the input image.
- The best way to conceptualize what your CNN perceives is to visualize the **Dense Layer Visualizations**.



Activation Maximization Visualizations



9.2

Saliency Maps & Class Activation Maps

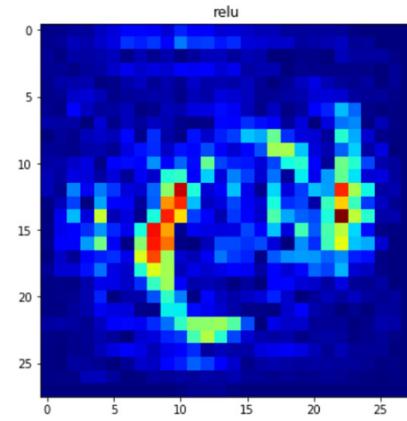
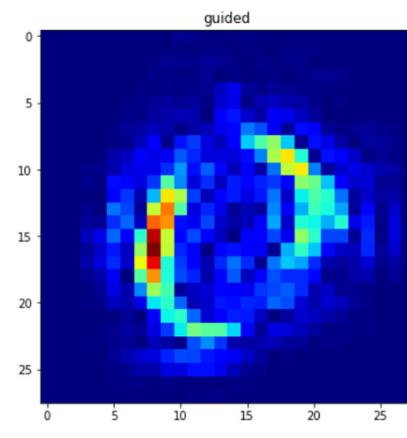


Saliency

- Suppose that all the training images of **bird** class contains a tree with leaves. How do we know whether the CNN is using bird-related pixels, as opposed to some other features such as the **tree** or **leaves** in the image? This actually happens more often than you think and you should be especially suspicious if you have a small training set.
- Saliency maps was first introduced in the paper: *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps* (<https://arxiv.org/pdf/1312.6034v2.pdf>)
- The idea is simple. We compute the gradient of output category with respect to input image. This should tell us how output category value changes with respect to a small change in input image pixels. All the positive values in the gradients tell us that a small change to that pixel will increase the output value. Hence, visualizing these gradients, which are the same shape as the image should provide some intuition of attention.

<https://raghakot.github.io/keras-vis/visualizations/saliency/>

Saliency Visualizations

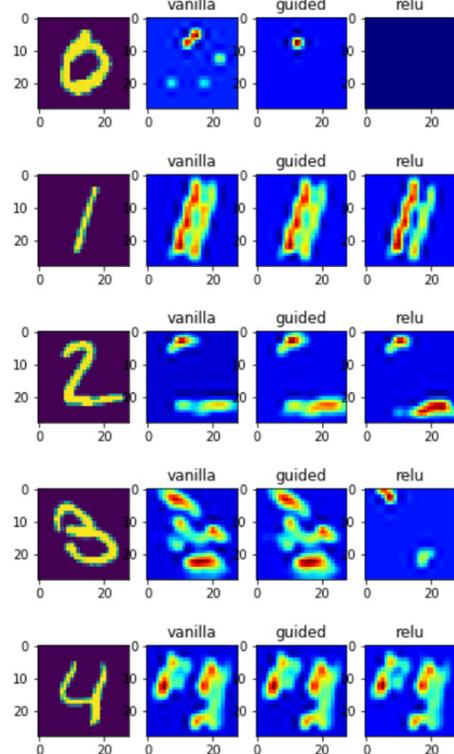




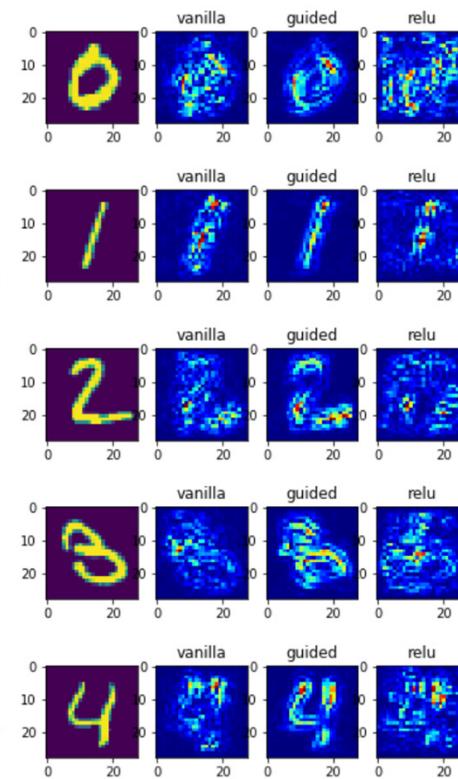
Class Activation Maps

- Class activation maps or **grad-CAM** is another way of visualizing attention over input.
- Instead of using gradients with respect to output (re: **saliency**), grad-CAM uses penultimate (pre Denselayer) Conv layer output.
- The intuition is to use the nearest Conv layer to utilize spatial information that gets completely lost in Dense layers.
- In keras-vis, we use grad-CAM as its considered more general than Class Activation maps.

Class Activation or Grad-CAM Visualizations



- Grad-CAM on left versus Saliency on right.
- In this case it appears that saliency is better than grad-CAM as the penultimate MaxPooling2D layer has (12, 12) spatial resolution which is relatively large as compared to input of (28, 28). It is likely that the CONV layer hasn't captured enough high level information and most of that is likely within dense_4 layer.



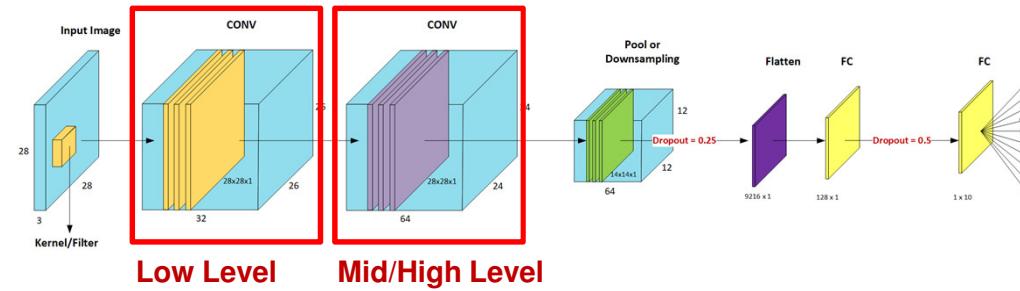
9.3

Filter Visualizations



Visualizing Filters

- We just spent sometime visualizing how CNN's perceive classes via Activation Maximization. Saliency Maps and Class Activations (grad-CAM).
- However, what if we wanted to inspect individual filters?
- Low level filters (i.e. CONV layers at the beginning of a Network, learn low level features while high level filters learn larger spatial patterns.





Visualizing Filters

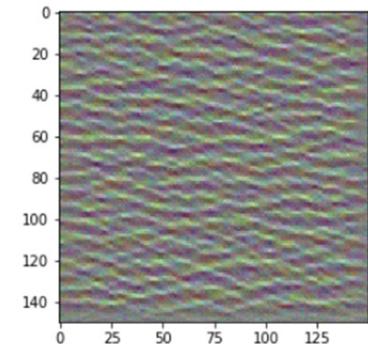
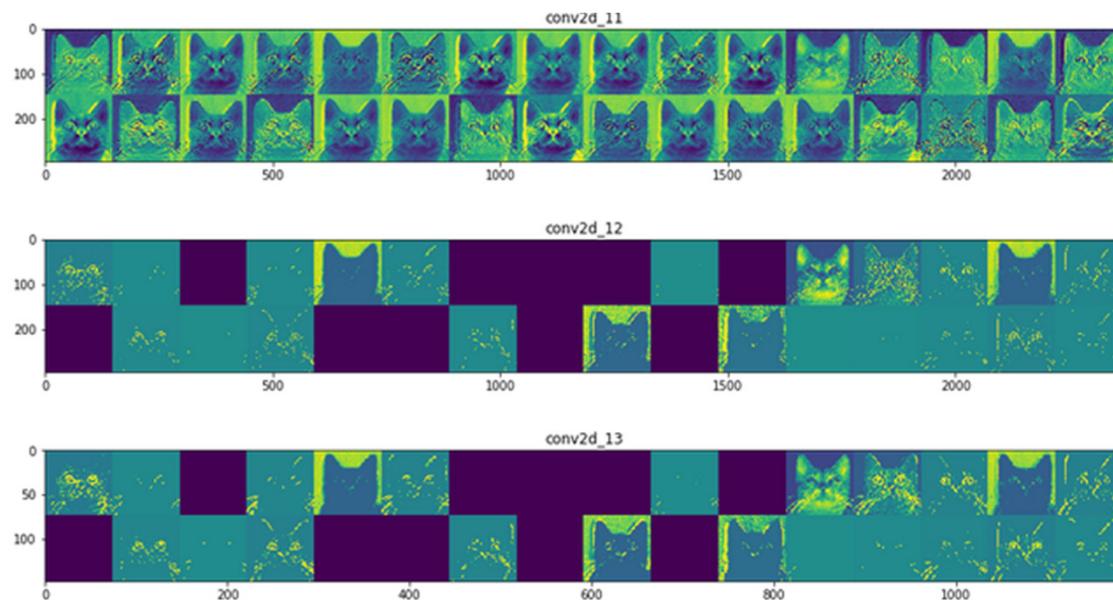
By Visualizing Filters we can **conceptualize what our CNN has 'learnt'**, this is because CNNs are essentially representations of a visual concepts.

We are going to perform the following **Filter or CONV layer visualizations**

1. Visualizing the intermediate CONV layer outputs – this is used to see how successive CONV layers transform their inputs
2. Visualizing Individual CONV layer Filters - to see what pattern/concept the filter is responding too
3. Heat Maps of class activation in an image – very useful in understanding which areas in an image corresponded to a certain class (this forms a building block to object detection)



Looking at some Filters



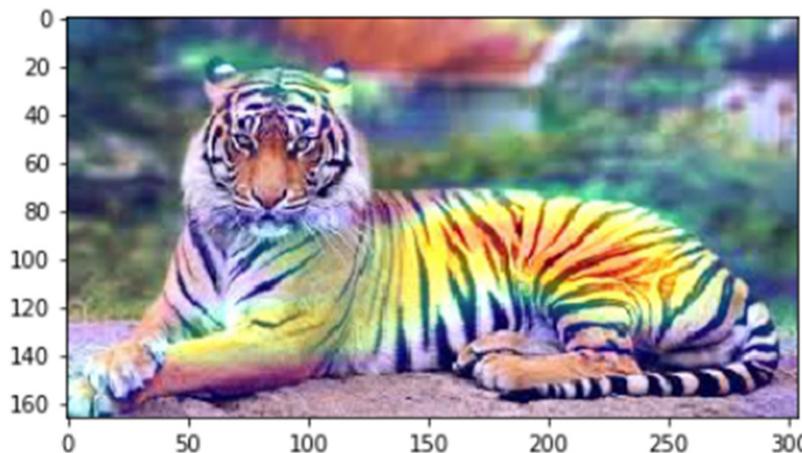
9.4

Heat Map Visualizations of Class Activation



Heat Maps of Class Activations

- Heat maps of class activations are very useful in identifying which parts of an image led the CNN to the final classification. It becomes very important when analyzing misclassified data.
- We use a pretrained VGG16 mode to create these maps



10.0

Data Augmentation



Data Augmentation

- **10.1 Splitting Data into Test and Training Datasets**
- **10.2 Build a Cats vs. Dogs Classifier**
- **10.3 Boosting Accuracy with Data Augmentation**
- **10.4 Types of Data Augmentation**

10.1

Splitting Data into Test and Training Datasets

Let's make a simple Cats vs Dogs Classifier

10.2

Build a Cats vs. Dogs Classifier



Cats vs. Dogs

- In the previous two CNNs we made simple classifiers using tiny images (28×28 and 32×32) to create some fairly decent image classifiers
- However, in both datasets we had thousands of image samples per category.
- In deep learning, the more training data/examples we have the better our model will be on unseen data (test data)
- However, what if we had less than 1000 examples per image class?
- Let's see what happens

10.3

Boosting Accuracy with Data Augmentation

Let's use Kera's Built-in Data Augmentation Tool



What is Data Augmentation?

- In the previous two CNNs we made simple classifiers using tiny images (28×28 and 32×32) to create some fairly decent image classifiers
- However, in both datasets we had thousands of image samples per category.
- In deep learning, the more training data/examples we have the better our model will be on unseen data (test data)
- However, what if we had less than 1000 examples per image class?
- Let's see what happens...

Data Augmentation

- We created 36 versions of our original image





Benefits of Data Augmentation

- Take a small dataset and make it much larger!
- Require much less effort in creating a dataset
- Adding variations such as rotations, shifts, zooming etc make our classifier much more invariant to changes in our images. Thus making it far more robust.
- Reduces overfitting due to the increased variety in the training dataset



Using Kera's Data Augmentation API

- Kera's built-in Data Augmentation API performs a just-in-time augmented image dataset. This means images aren't created and dumped to a directory (which will be wasteful storage). Instead it generates this dataset during the training process



Creating Our Image Generator

```
train_datagen = ImageDataGenerator(  
    rescale = 1. / 255,  
    shear_range = 0.2,  
    zoom_range = 0.2,  
    horizontal_flip = True)  
  
test_datagen = ImageDataGenerator(rescale=1. / 255)
```

- We use the above code to create our generator with types of augmentation to perform specified in the input arguments



Configuring Batch Sizes

```
test_generator = train_datagen.flow_from_directory(  
    train_data_dir,  
    target_size = (img_width, img_height),  
    batch_size = batch_size,  
    class_mode = 'binary')  
  
validation_generator = test_datagen.flow_from_directory(  
    validation_data_dir,  
    target_size = (img_width, img_height),  
    batch_size = batch_size,  
    class_mode = 'binary')
```

- The `flow_from_directory()` function takes our image data and creates an iterator (a memory efficient method of returning a sequence of data). (note we can use `flow()` as well when needed)
- We can specify batch sizes, class mode and image size etc.



Fitting Our Generator

```
model.fit_generator(  
    train_generator,  
    steps_per_epoch = nb_train_samples // batch_size,  
    epochs = epochs,  
    validation_data = test_generator,  
    validation_steps = nb_validation_samples // batch_size)
```

- Lastly, we simply use our `model.fit()` but with our data augmentation generator – so we're using `model.fit_generator` instead.
- This starts our training process, but now we're using our augmented dataset!



Cats vs. Dogs Performance with Data Augmentation

- With Data Augmentation after 25 Epochs we got 76.4% Accuracy
- After 25 Epochs without Data Augmentation we got 74.6% Accuracy

10.4

Types of Data Augmentation

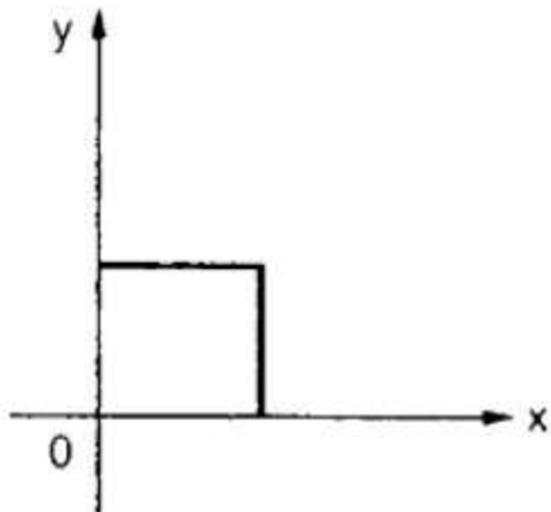


Types of Data Augmentation

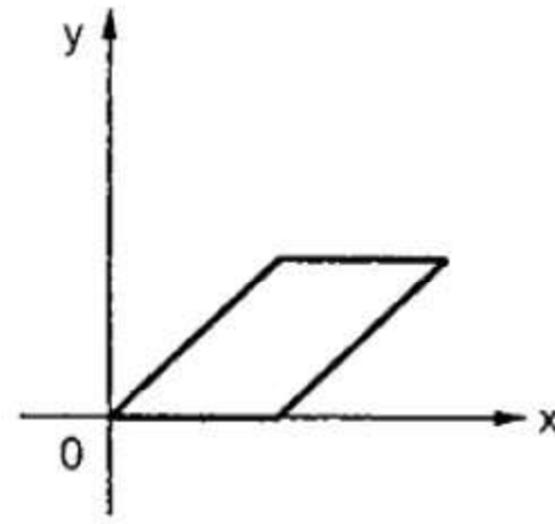
- At <https://keras.io/preprocessing/image/> we can see there are many types of augmentation available.
- In practice I've found the following add the most benefit.
 - Rotations
 - Horizontal and vertical shifts
 - Shearing
 - Zooming



Shearing



(a) Original object



(b) Object after x shear



Fill Mode

- **fill_mode** : One of {"constant", "nearest", "reflect" or "wrap"}. Points outside the boundaries of the input are filled according to the given mode.
 - Constant – either black or white
 - Nearest – takes the color of the last pixel before the cut off
 - Reflect – mirror reflection of the image from that point
 - Wrap – starts replicating the image again
- Using Nearest as that doesn't tend to create new features that may be add some confusion to what our classifier is supposed to learning