

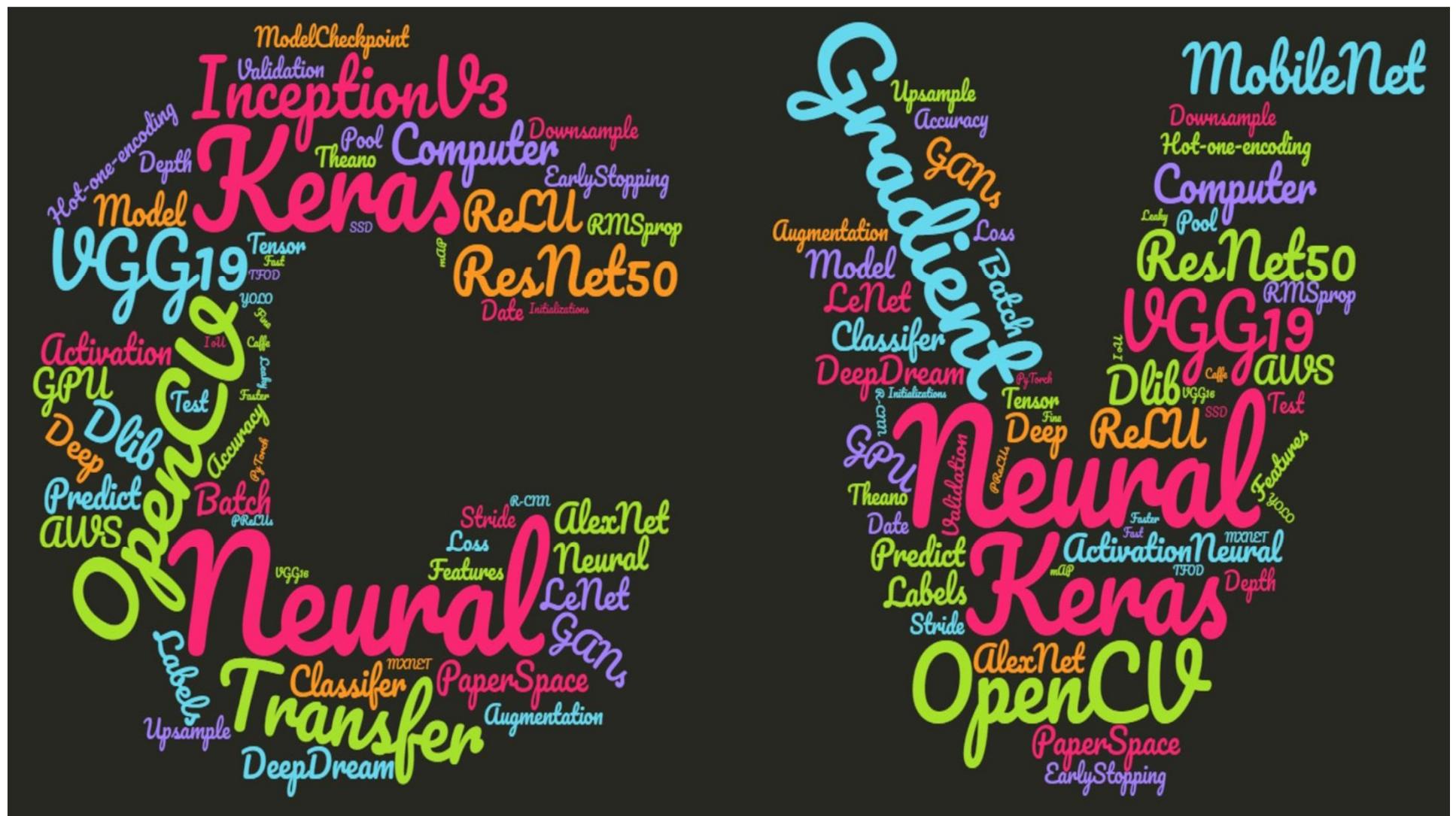


Deep Learning Computer Vision

1.0

Course Introduction

Why you'll love this course!





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.





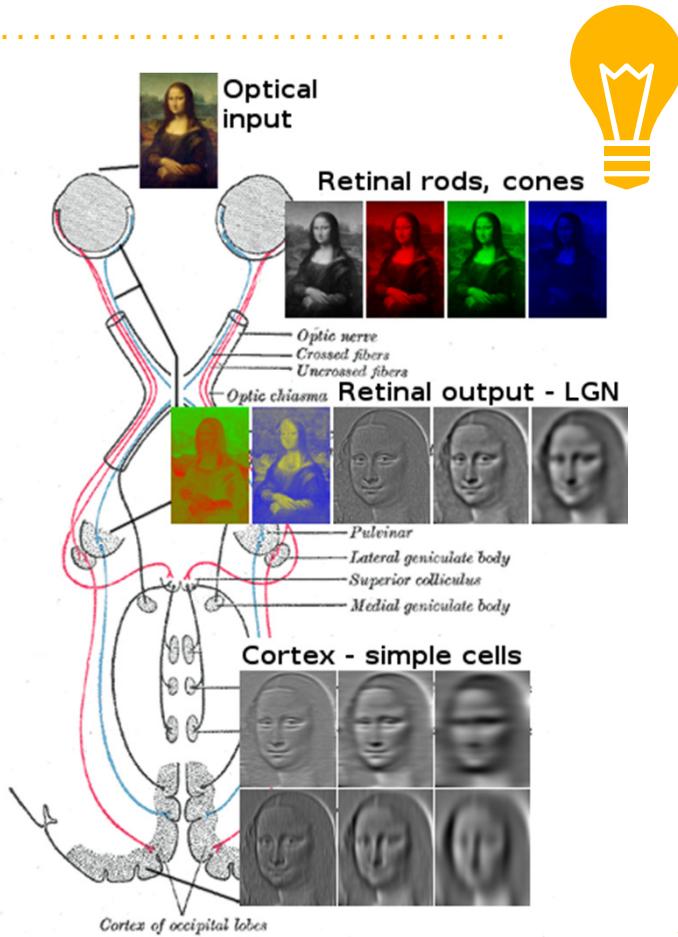
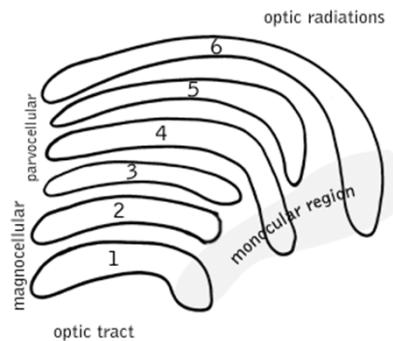
Computer Vision is perhaps the most important part of the AI Dream



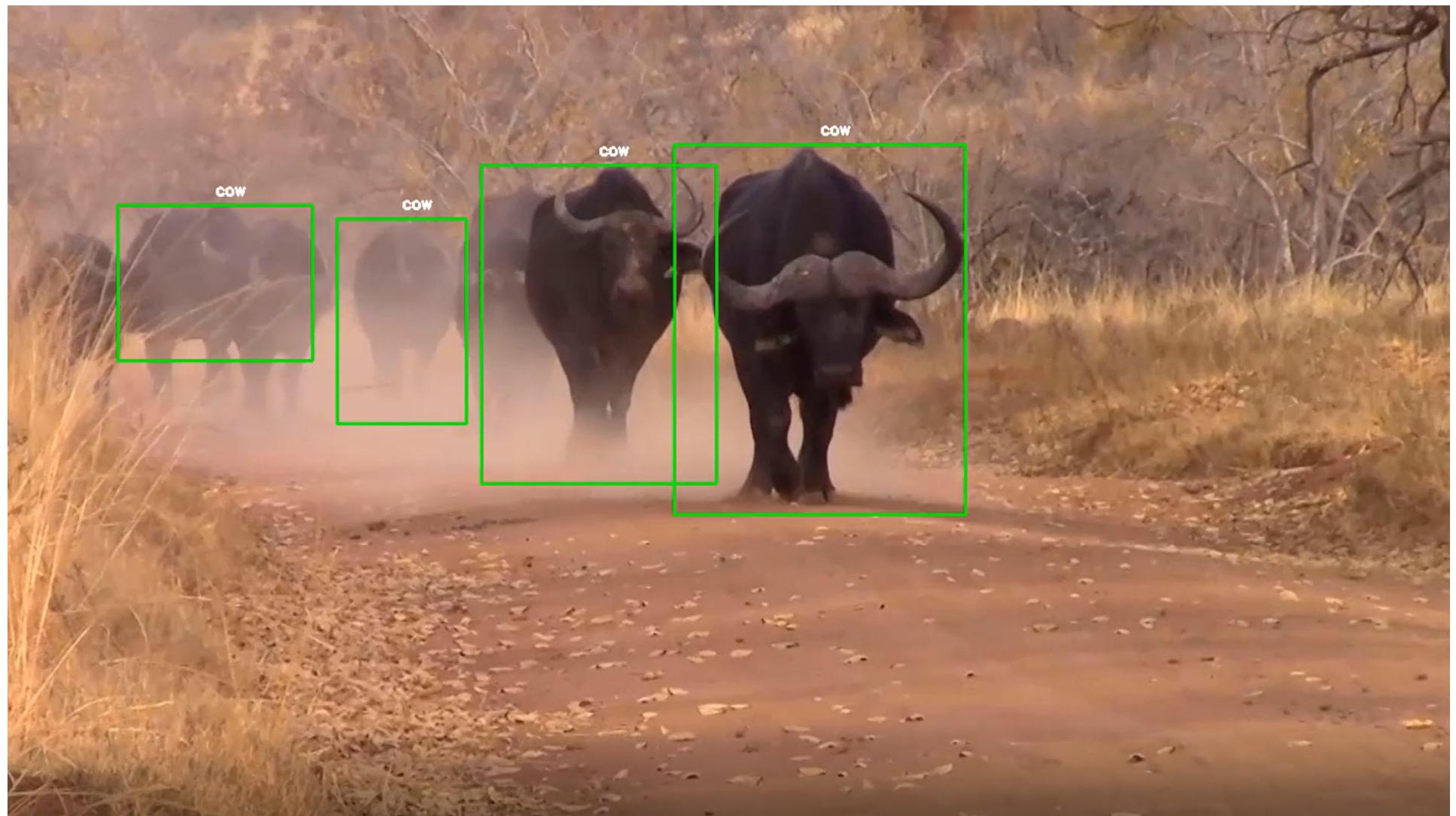
Source: *Terminator 2: Judgement Day*

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

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

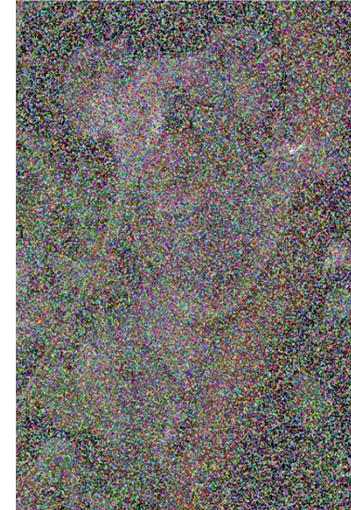


https://en.wikipedia.org/wiki/Visual_system



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?

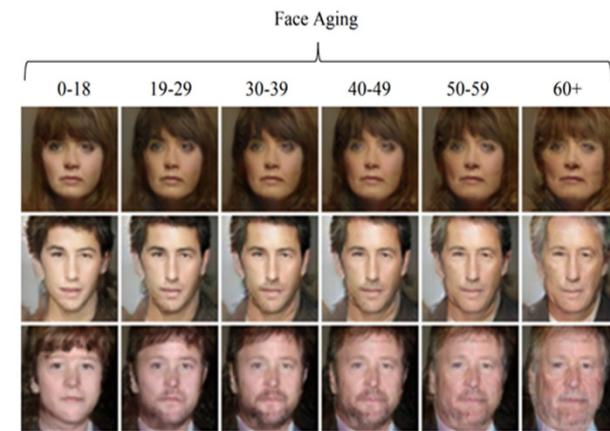
**This Course is
for you!**

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)



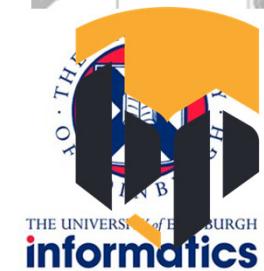
Hi

I'm Rajeev Ratan



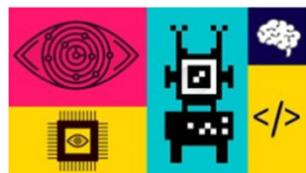
About me

- **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.



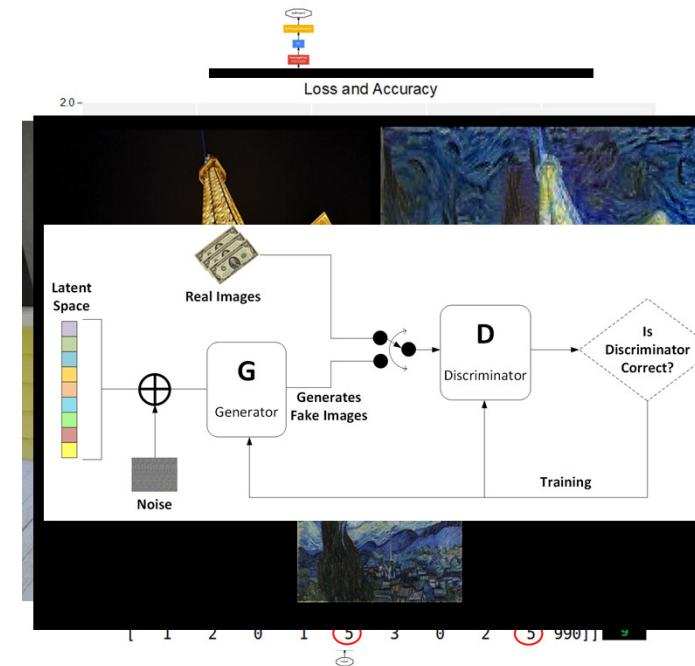
7 months ago
Kevin Kinser



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.

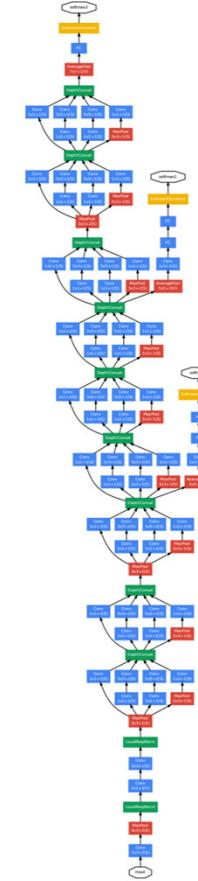
What you'll be able to do after

- 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)

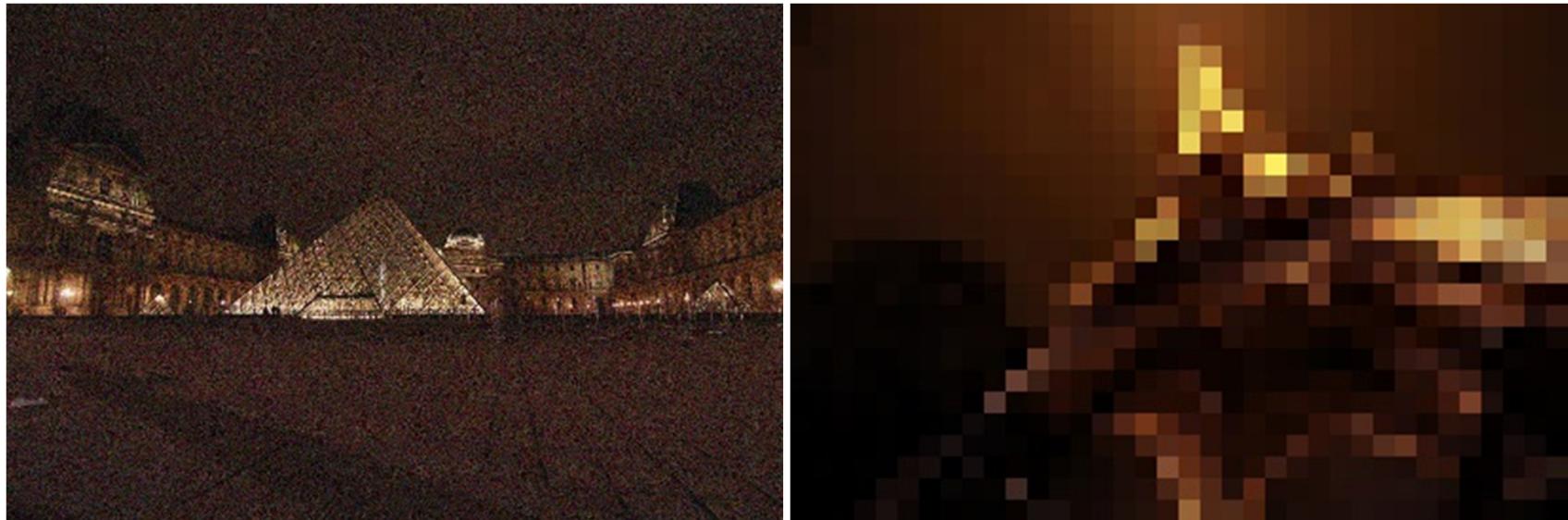


**And there are
other reasons
it's so hard**



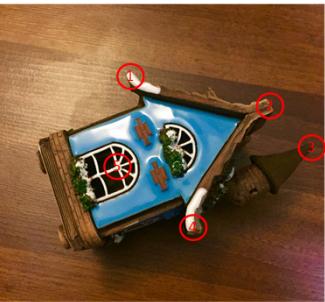


Camera sensors and lens limitations



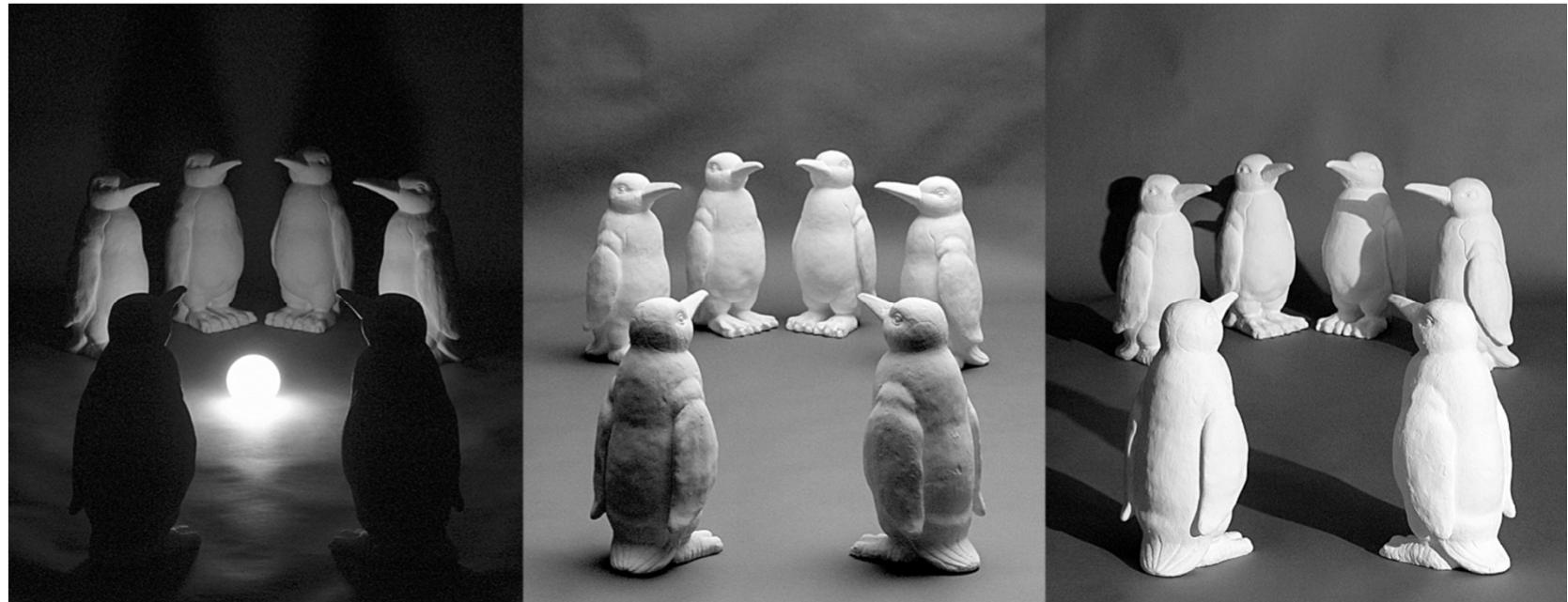


Viewpoint variations





Changing Lighting Conditions

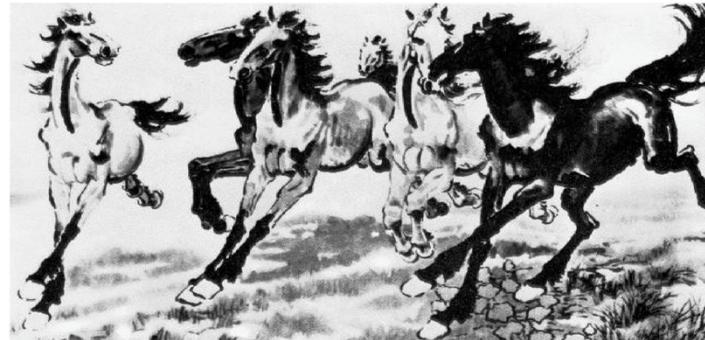


Issues of Scale



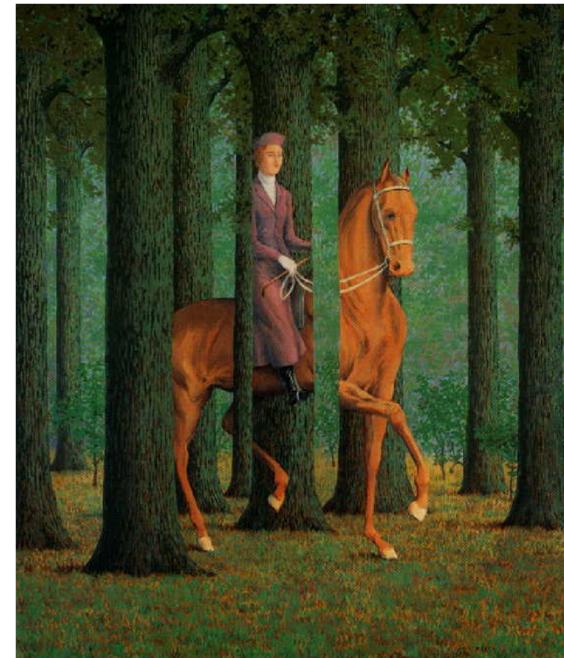


Non-rigid Deformations





Occlusion – Partially Hidden

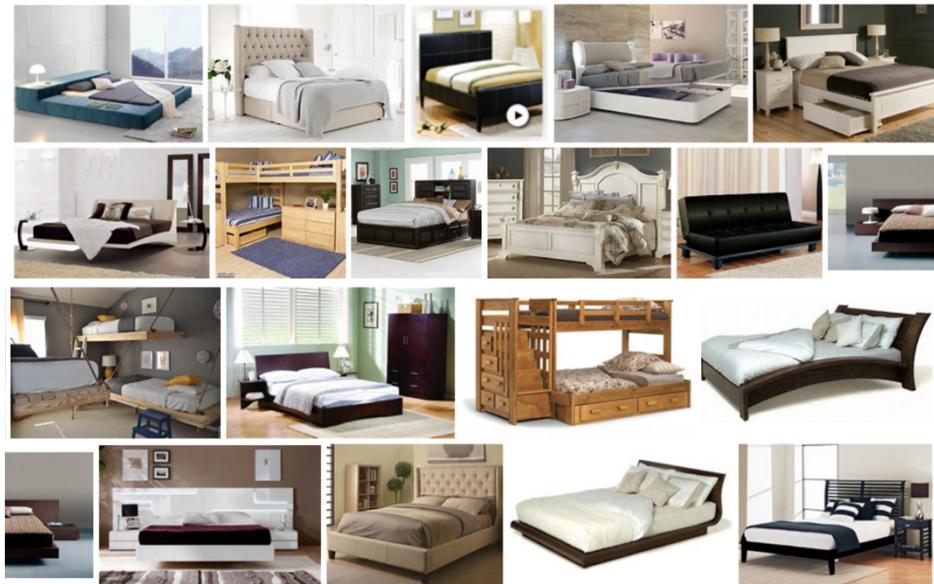


Clutter



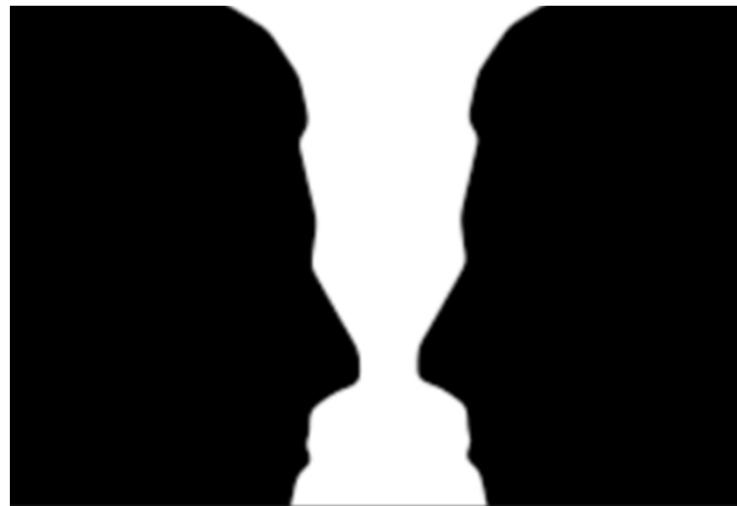


Object Class Variations





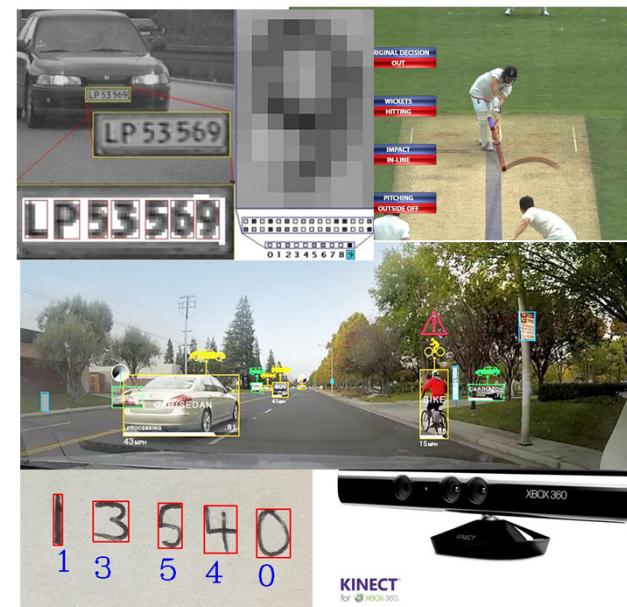
Ambiguous Optical Illusions





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



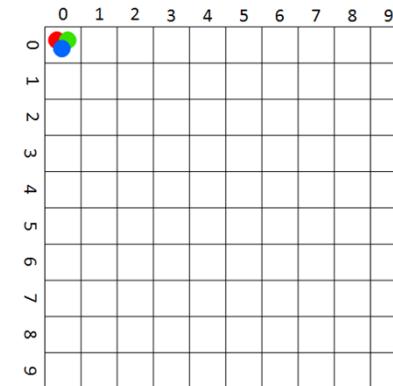
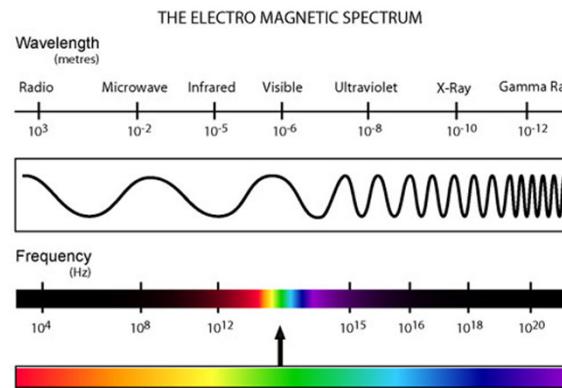
2.2

What are Images?



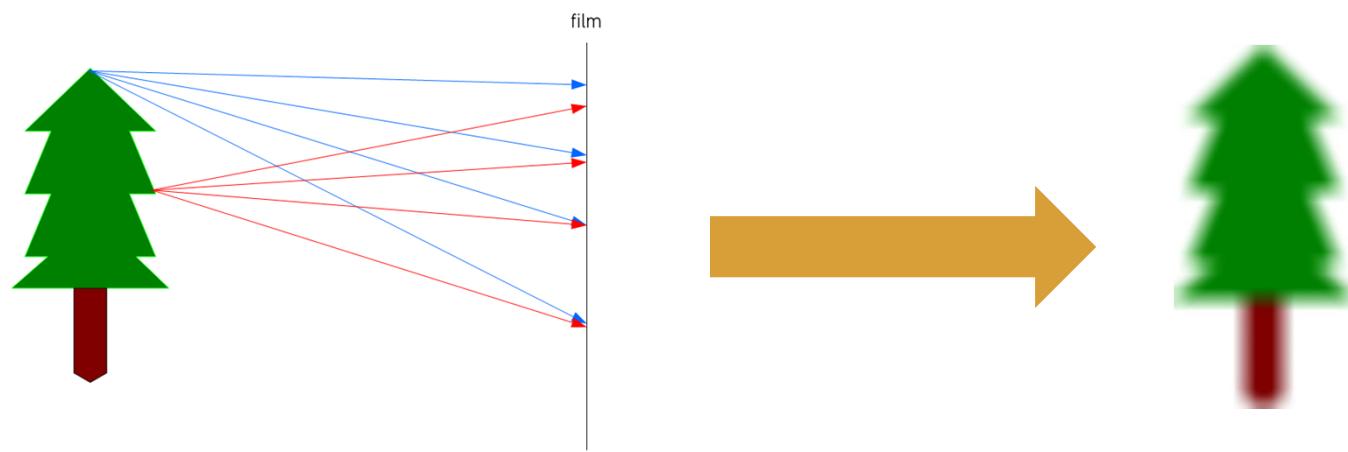
What is exactly are images?

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



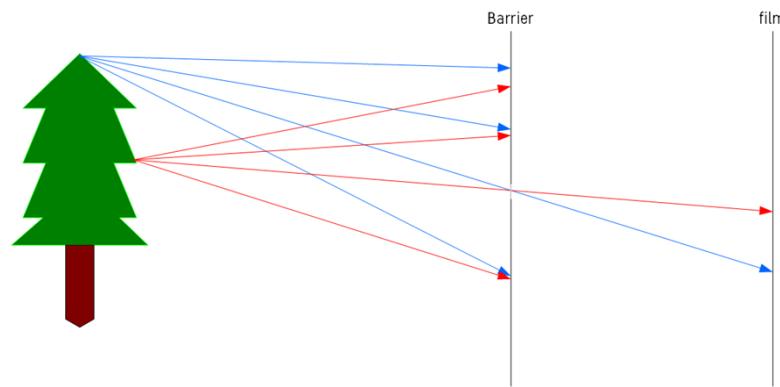


How are Images Formed?



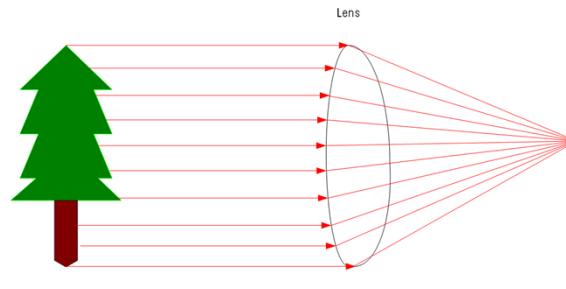
- When light reflects off the surface of an object and onto film, light sensors or our retinas.
- In this example, the image will appear blurred

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

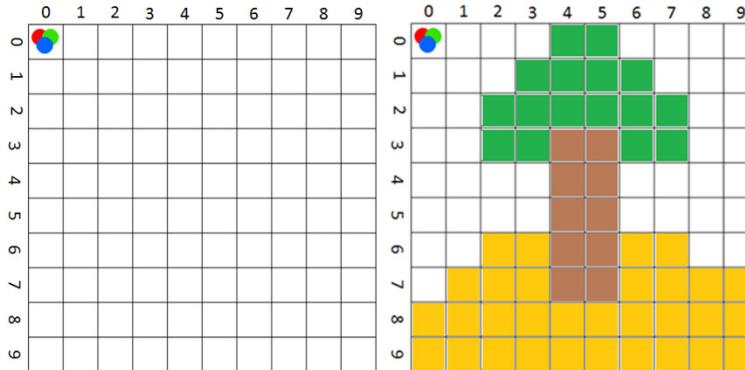
Image Formation



- 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

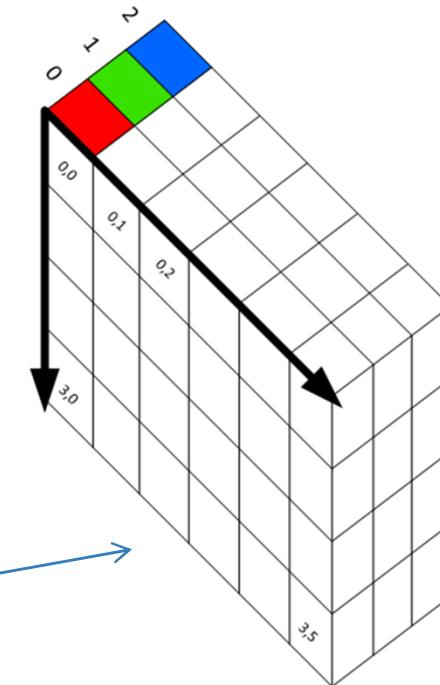
- Think of an array as a table. A 1-Dim array looks like this:

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. A main text block describes the library's purpose and usage. To the right is a "Quick Links" sidebar with links to documentation, tutorials, forums, bug reports, build farms, developer sites, and a wiki, along with a "Donate" button. Below the sidebar is a "Latest news" section with four news items. At the bottom, there are links to various social media and community platforms, and a copyright notice.

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.

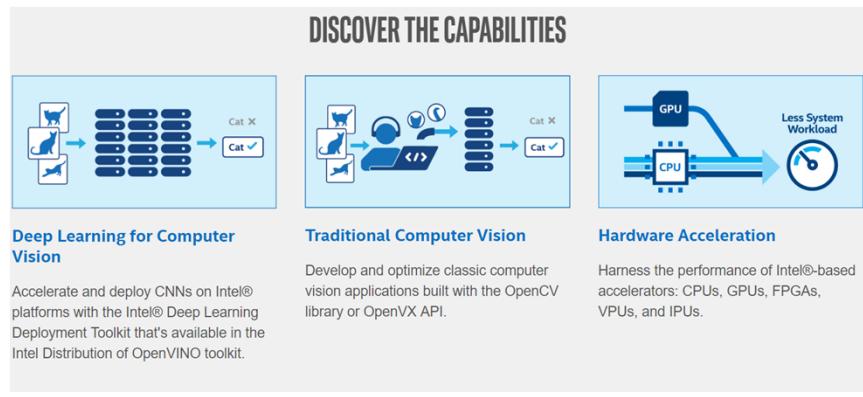
E-Mail Slack GitHub Facebook Google+ Wikipedia
Forum IRC #opencv SourceForge Twitter YouTube

© 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**

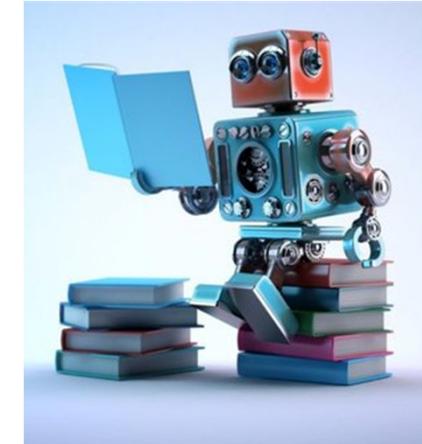
6.1

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.





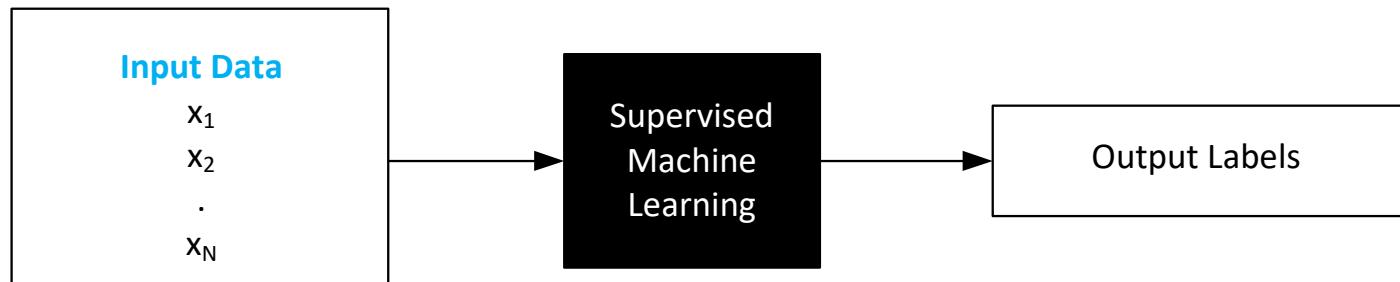
Types of Machine Learning

- 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

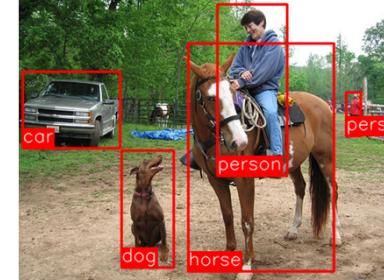
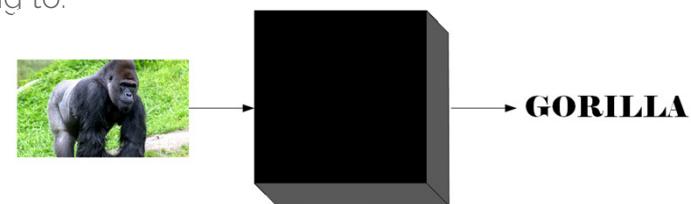
- Supervised learning is by far the most popular form of AI and ML used today.
- We take a set of labeled data (called a **dataset**) and we feed it to some ML learning algorithm that develops a model to fit this data to some outputs
- E.g. let's say we give our ML algorithm a set of 10k spam emails 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

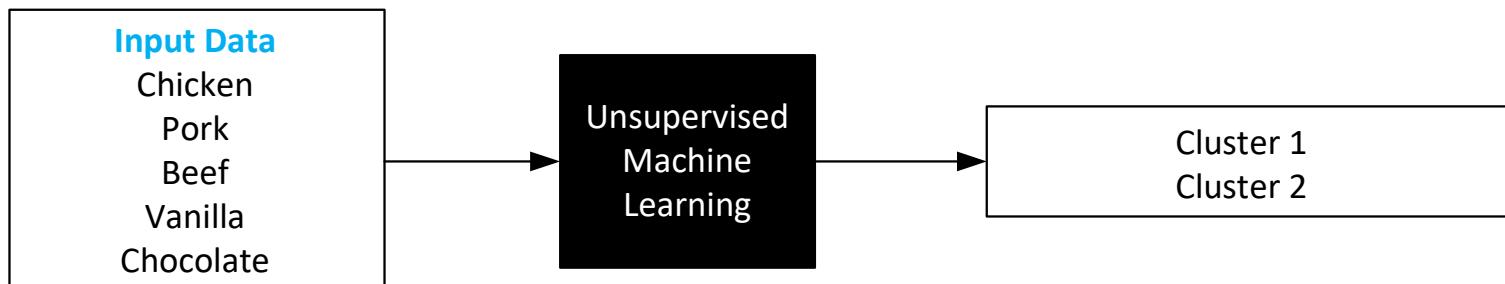
- In Computer Vision we can use Supervised Learning to:
 - Classify types of images
 - Object Detection
 - Image Segmentation





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

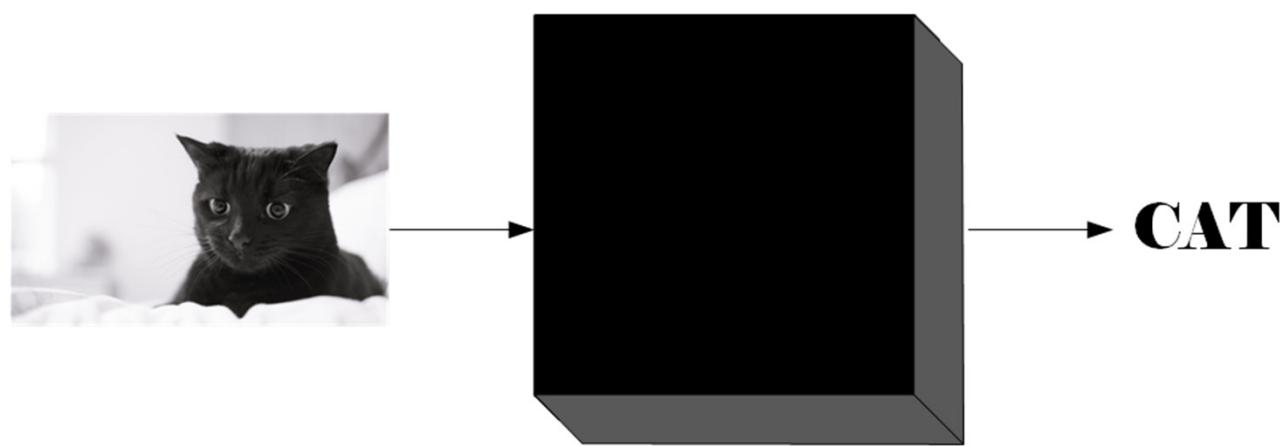


What are Neural Networks

- 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.

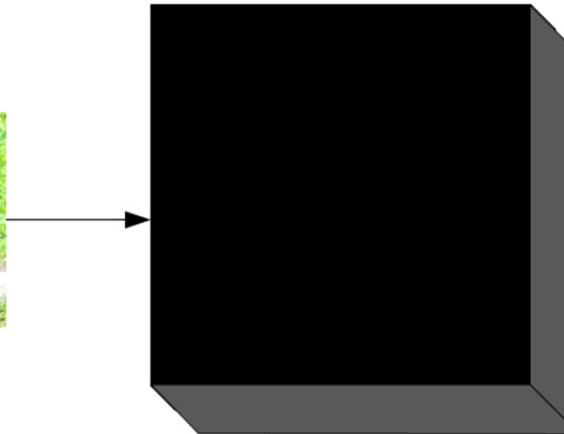


The Mysterious 'Black Box' Brain





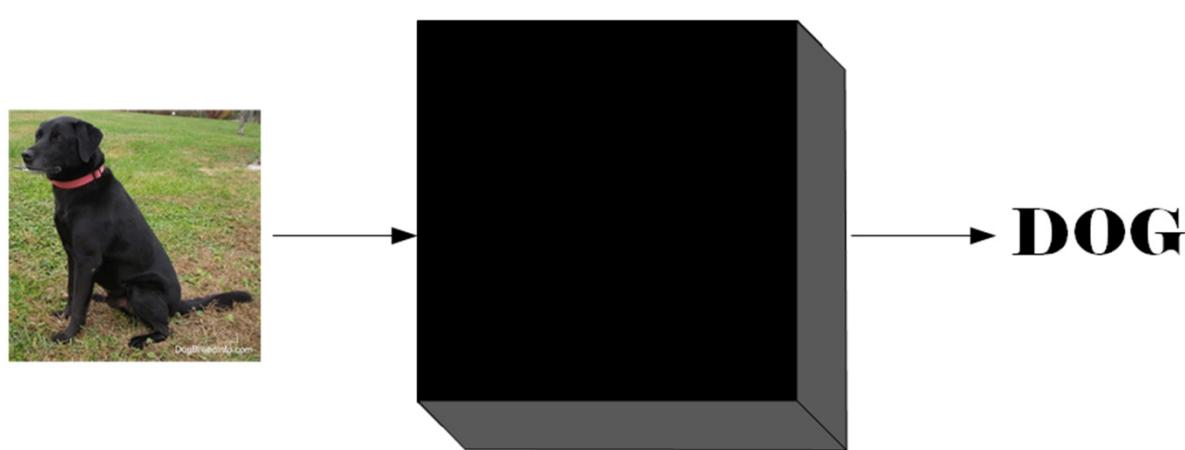
The Mysterious 'Black Box' Brain



GORILLA



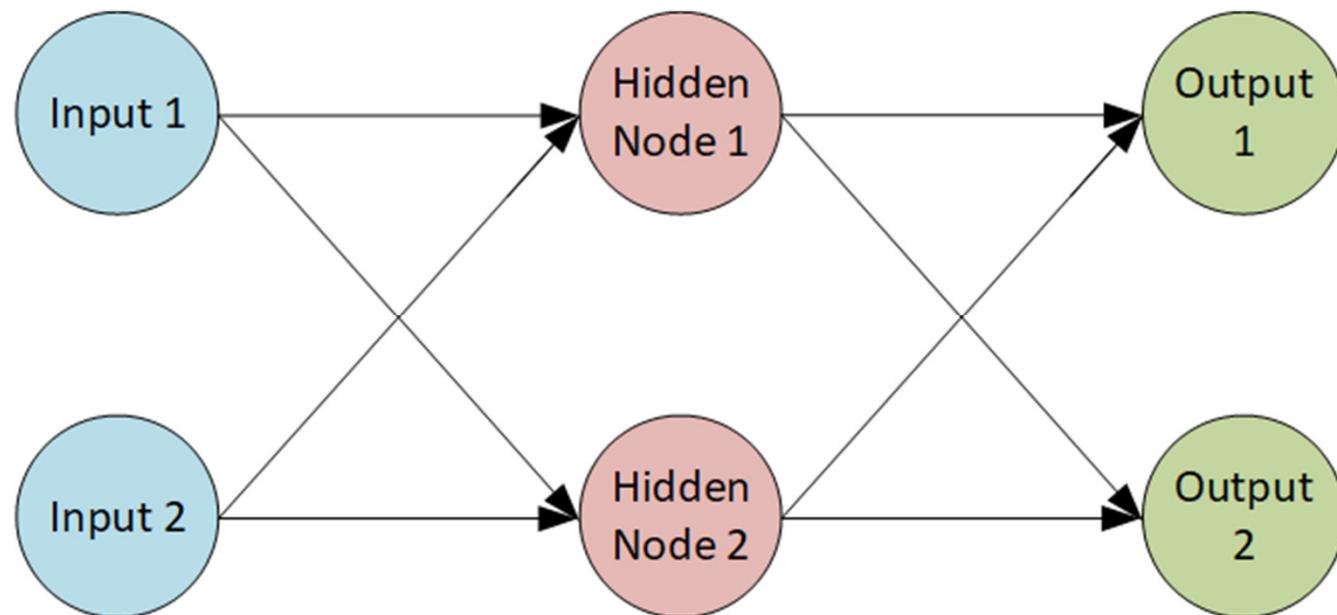
The Mysterious 'Black Box' Brain





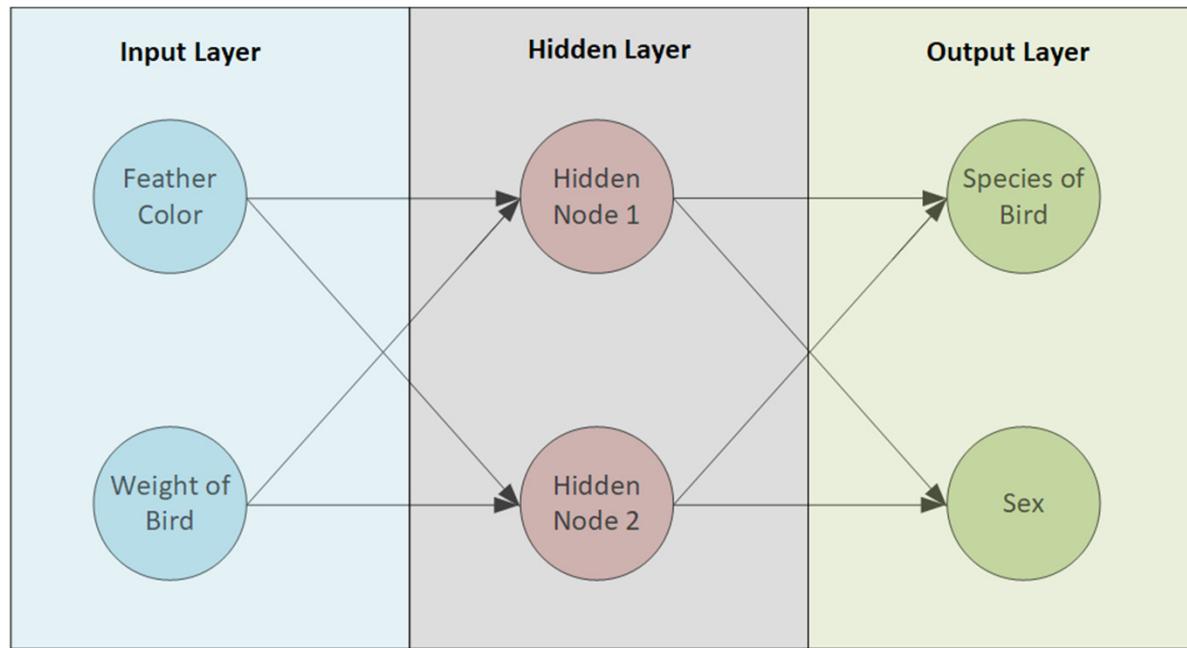
How NNs 'look'

A Simple Neural Network with 1 hidden layer





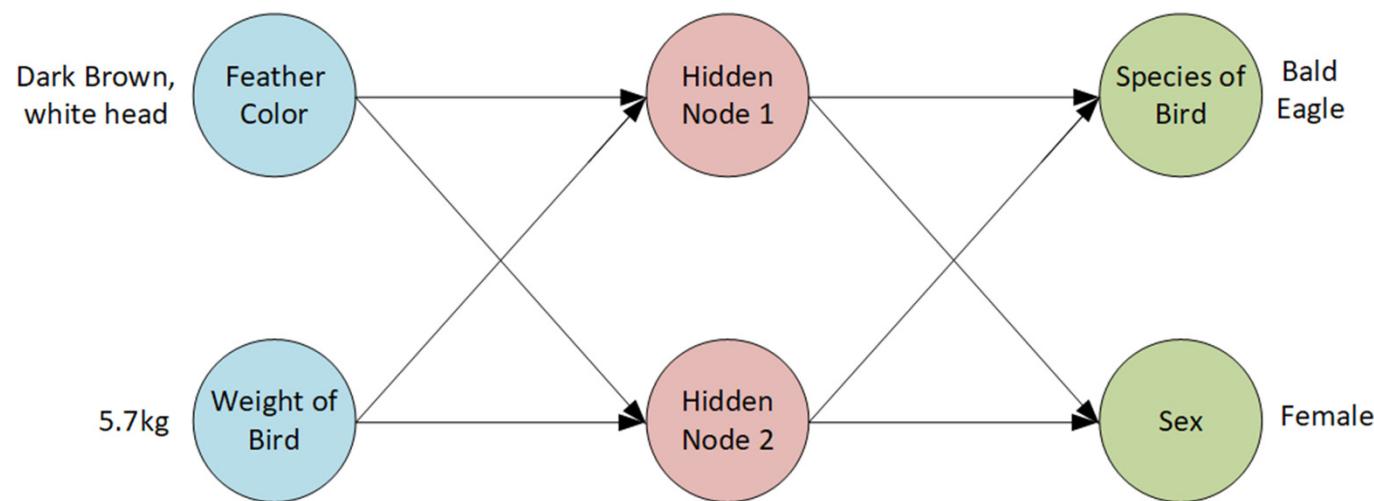
Example Neural Network





How do we get a prediction?

Pass the inputs into our NN to receive an output





How we predict example 2

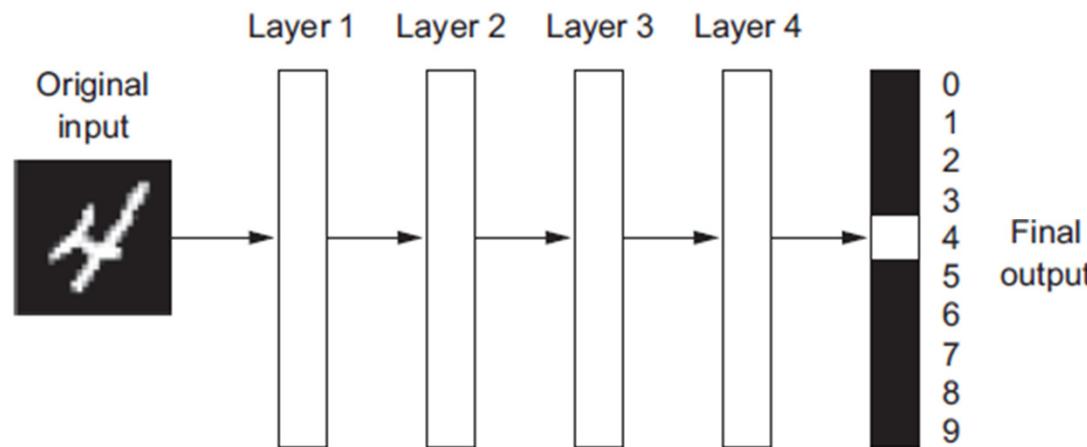


Figure 1.5 A deep neural network for digit classification

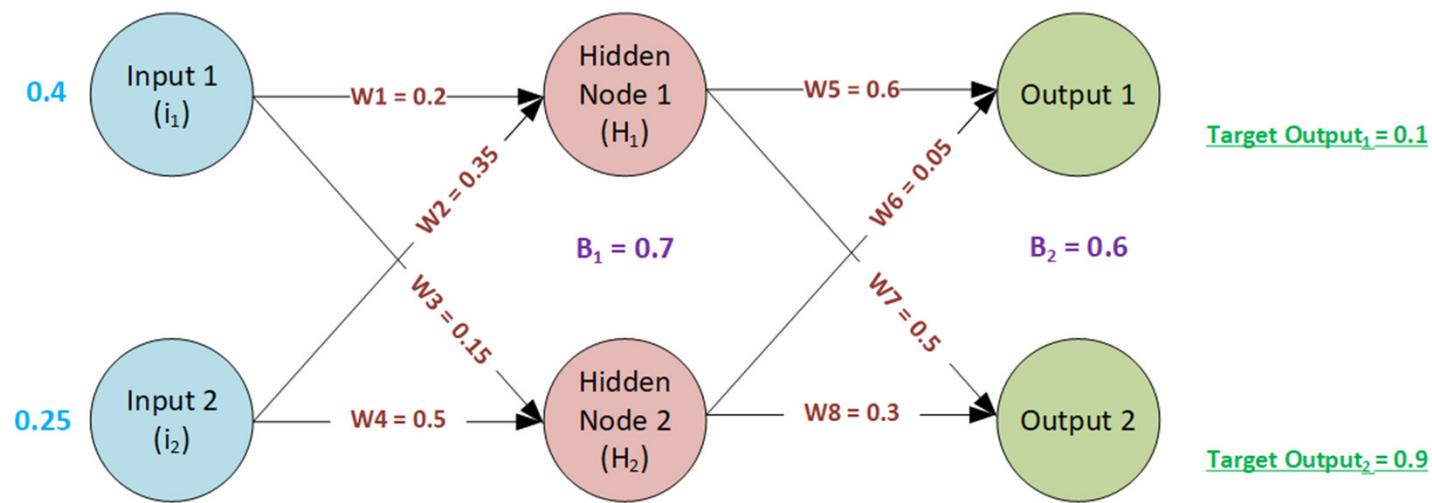
6.3

Forward Propagation

How Neural Networks process their inputs to produce an output

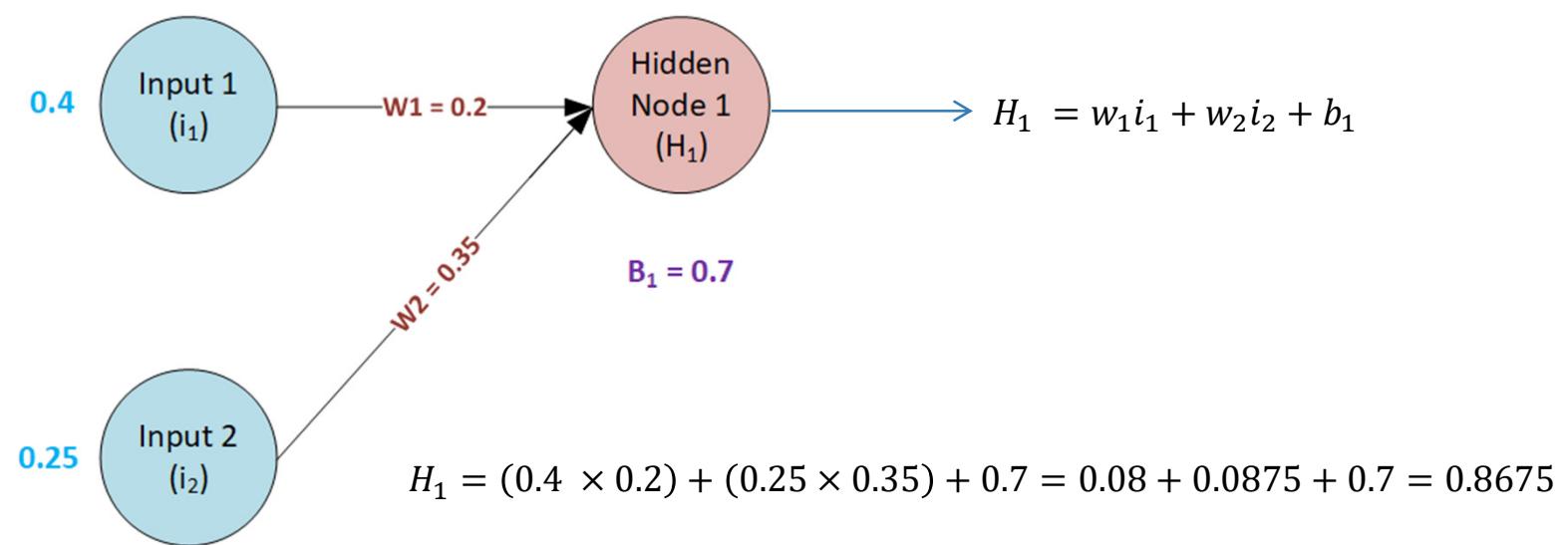


Using actual values (random)



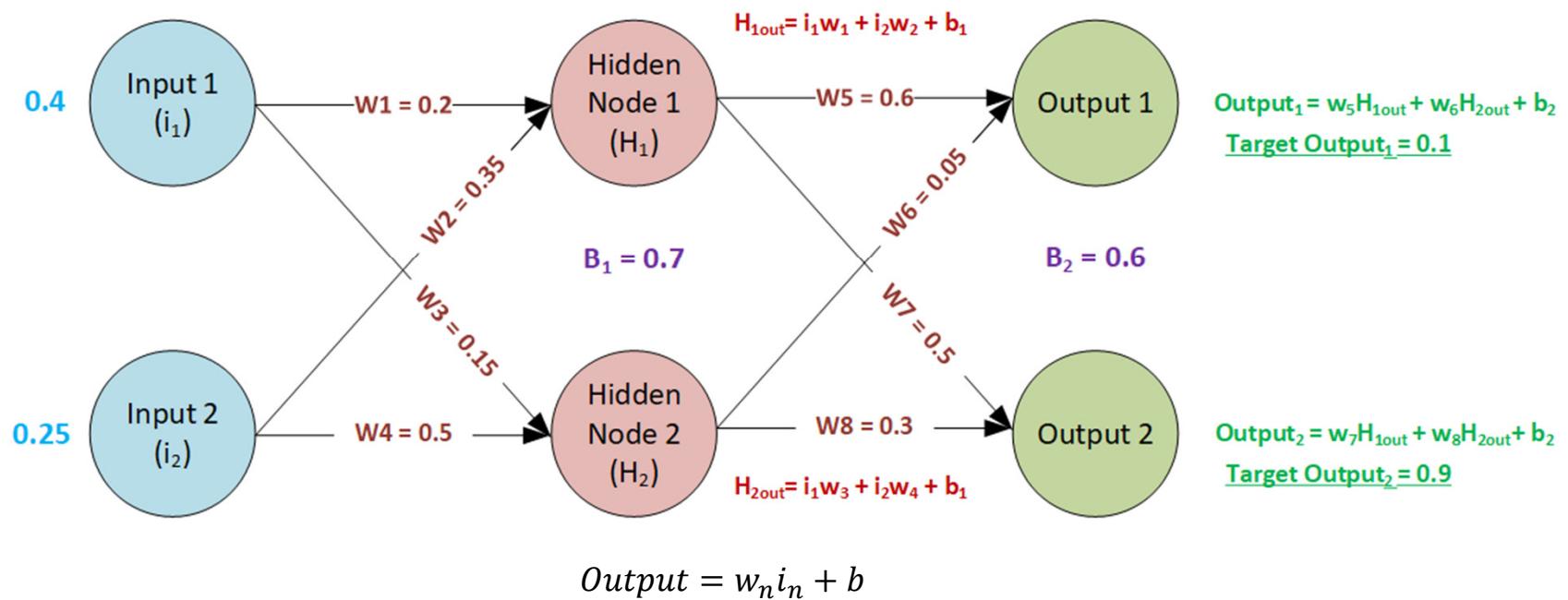


Looking at a single Node/Neuron



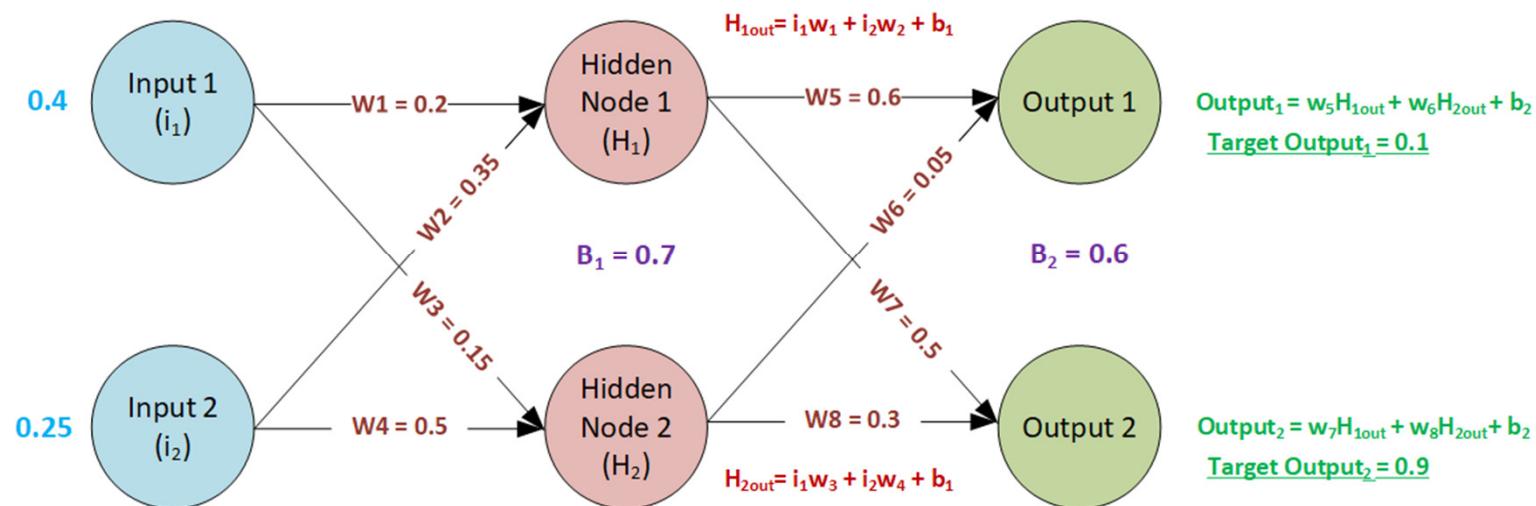


Steps for all output values





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

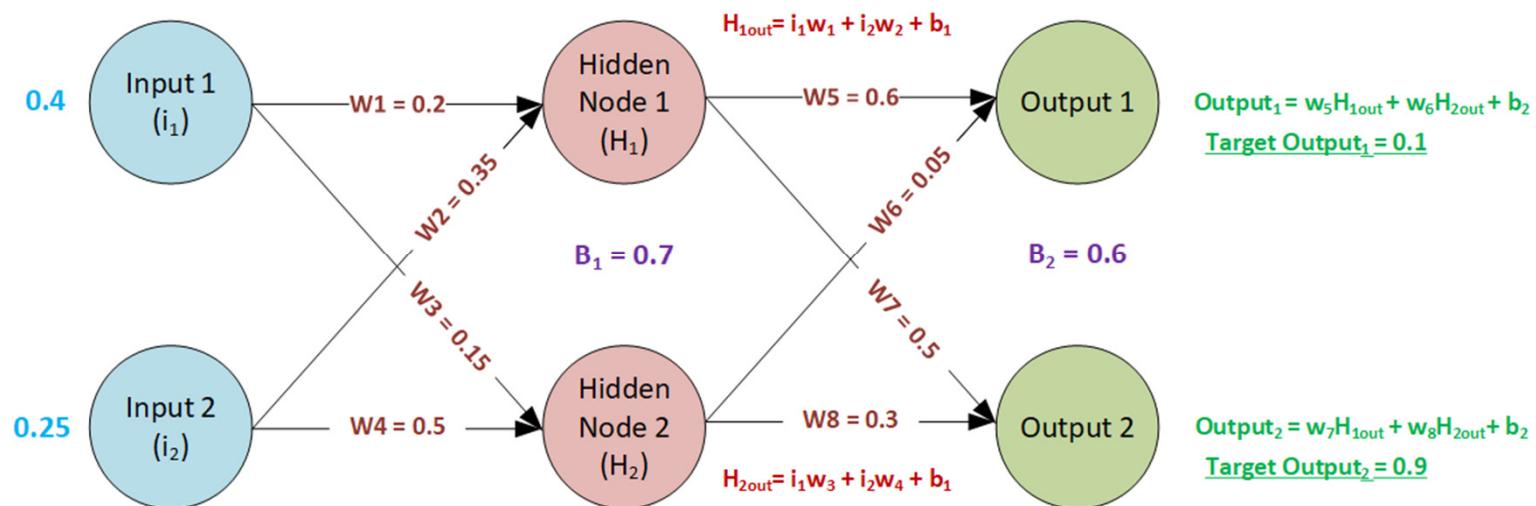


$$H_2 = i_1w_3 + i_2w_4 + b_1$$

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



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.

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

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

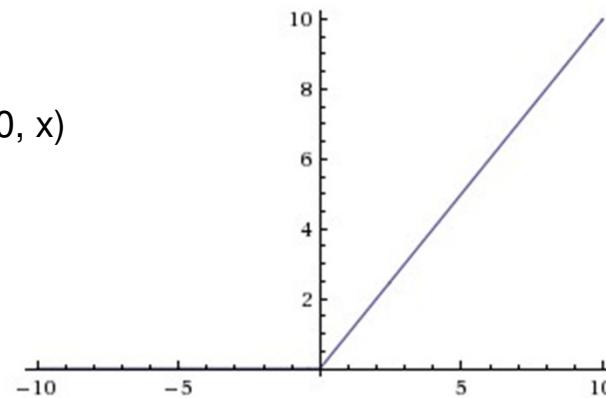
- Therefore, if $w_i x_i + b = 0.75$
- $f(x) = 0.75$
- However, if $w_i x_i + b = -0.5$ then $f(x) = 0$



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.

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





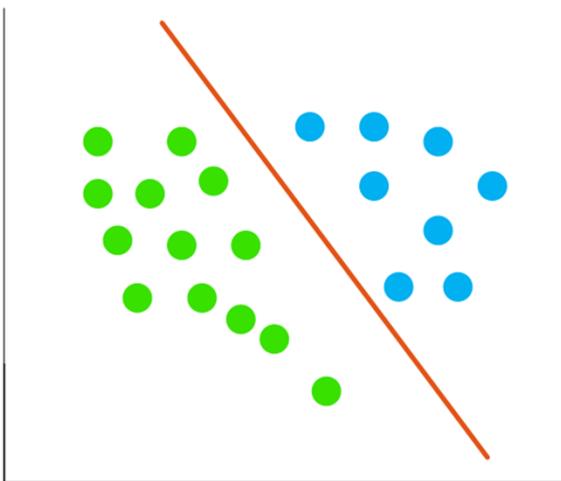
Why use Activation Functions? For Non Linearity

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

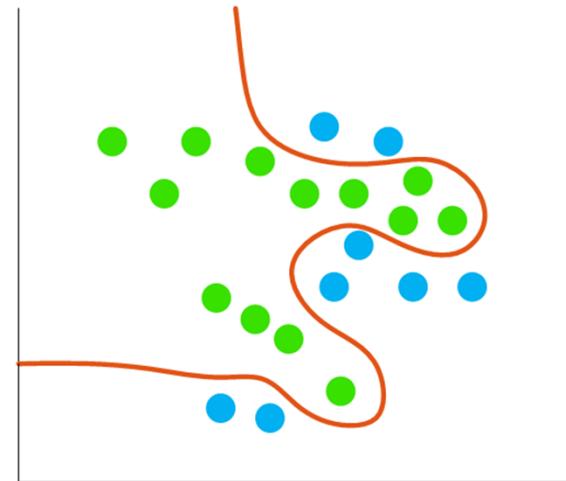


Example of Non Linear Data

Linearly Separable



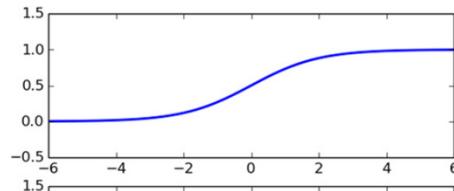
Non-Linearly Separable



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

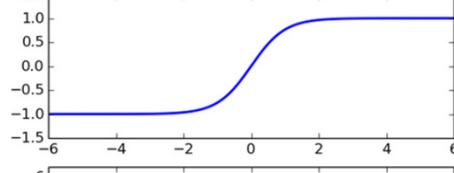


Types of Activation Functions



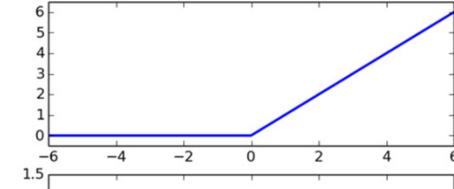
Sigmoid

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



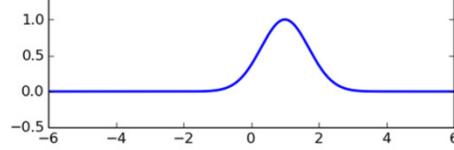
Hyperbolic Tangent

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



Rectified Linear

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



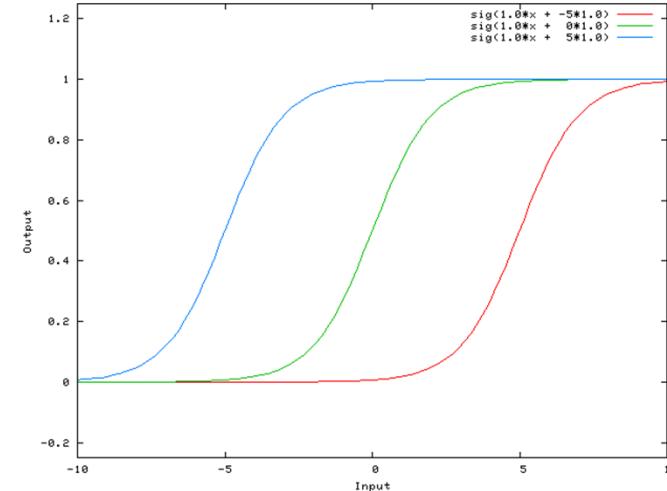
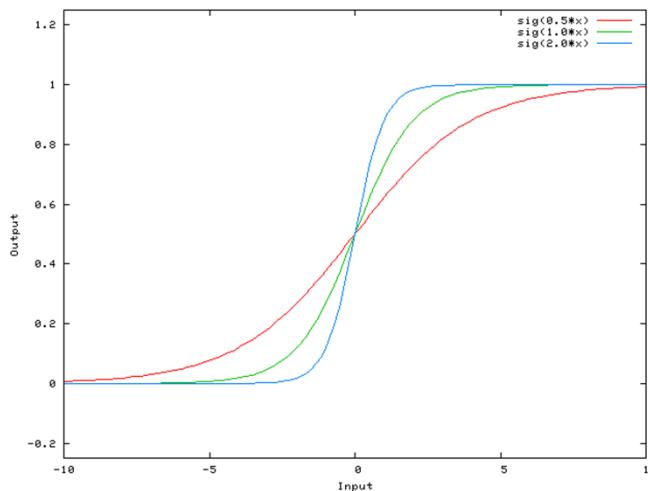
Radial Basis Function

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



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.

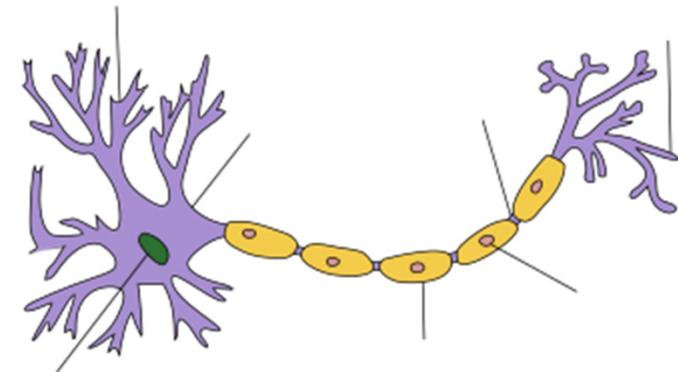


- Changes in weights simply change the gradient or steepness of the output, if we needed shift our function left or right , we need a bias.



Neuron Inspiration

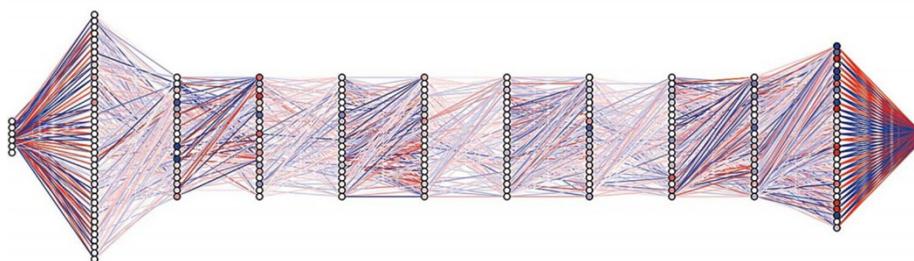
- Neuron only fires when an input threshold is reached
- Neural Networks follow that same principle





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

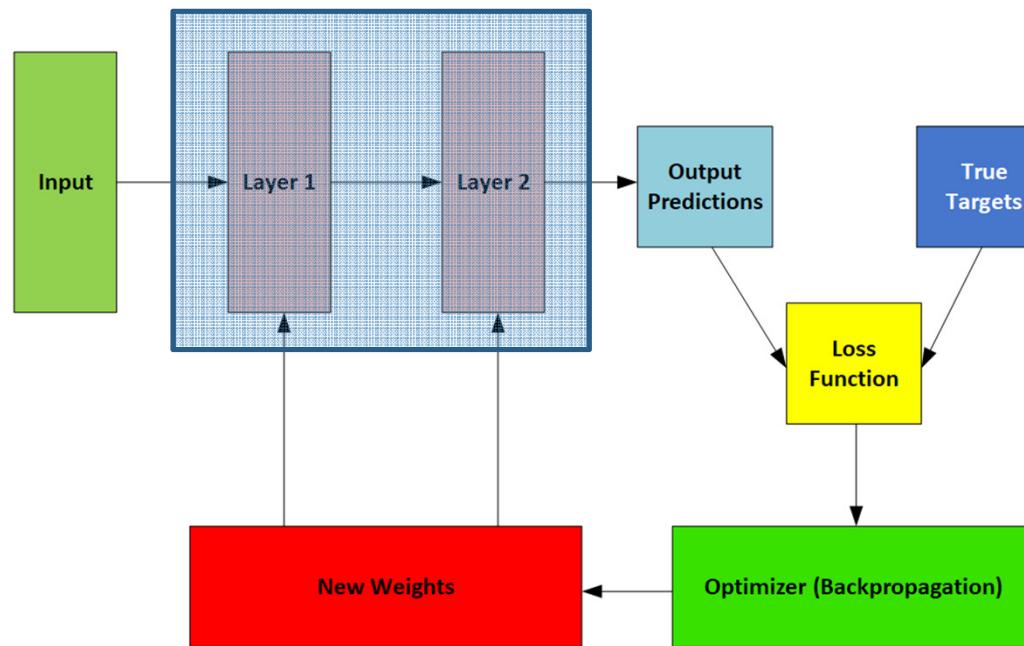


Training step by step

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)

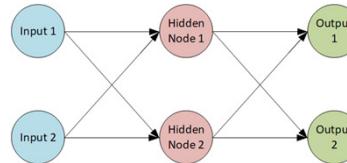
In practice, MSE is always a good safe choice. We'll discuss using different loss functions later on.

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

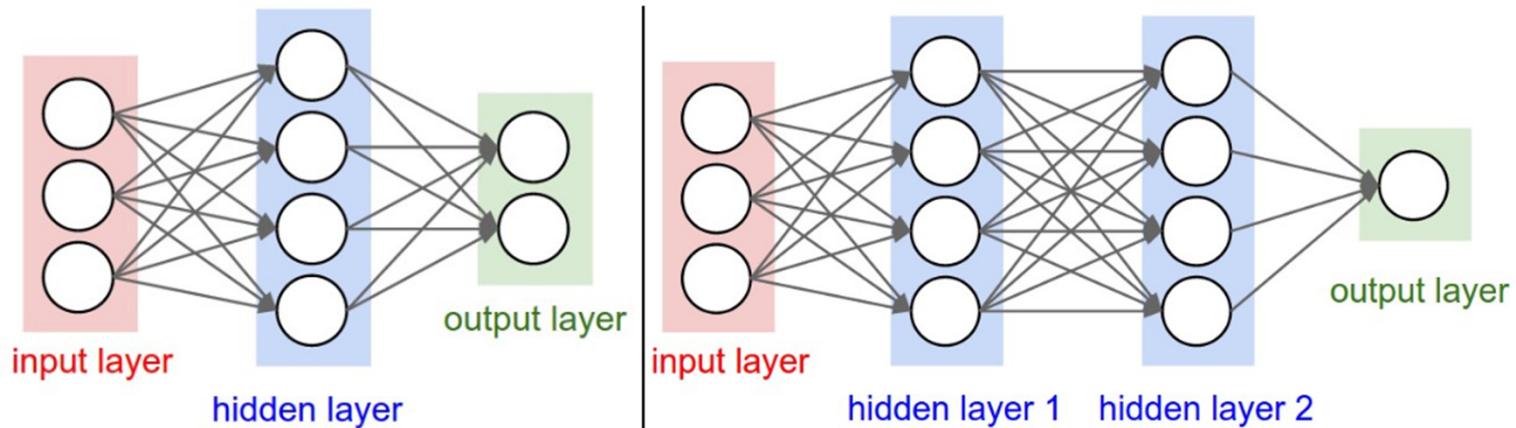
- Getting the best weights for our classifying our data is not a trivial task, especially with large image data which can contain thousands of inputs from a single image.



- 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



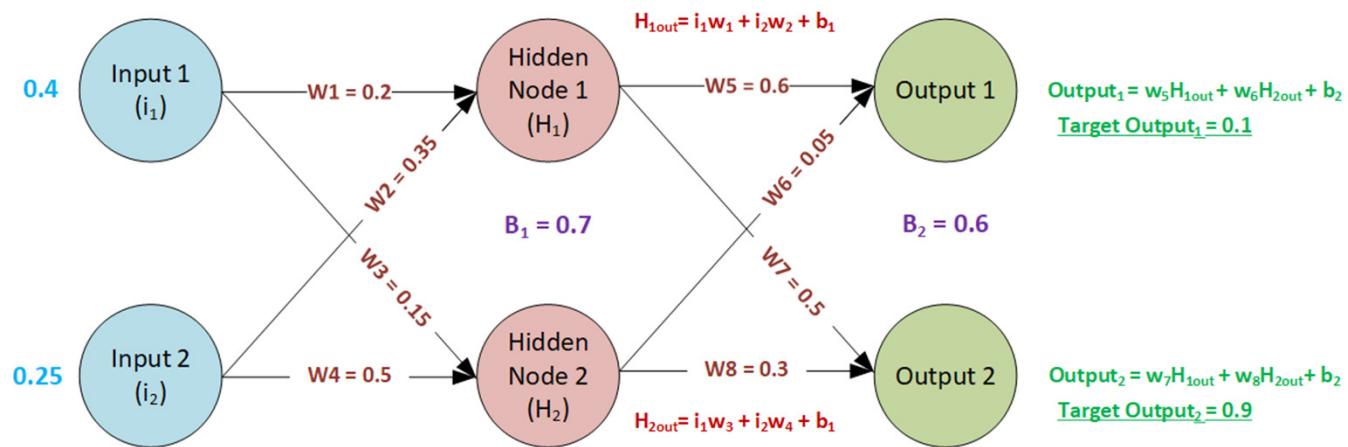
Introducing Backpropagation

- 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.



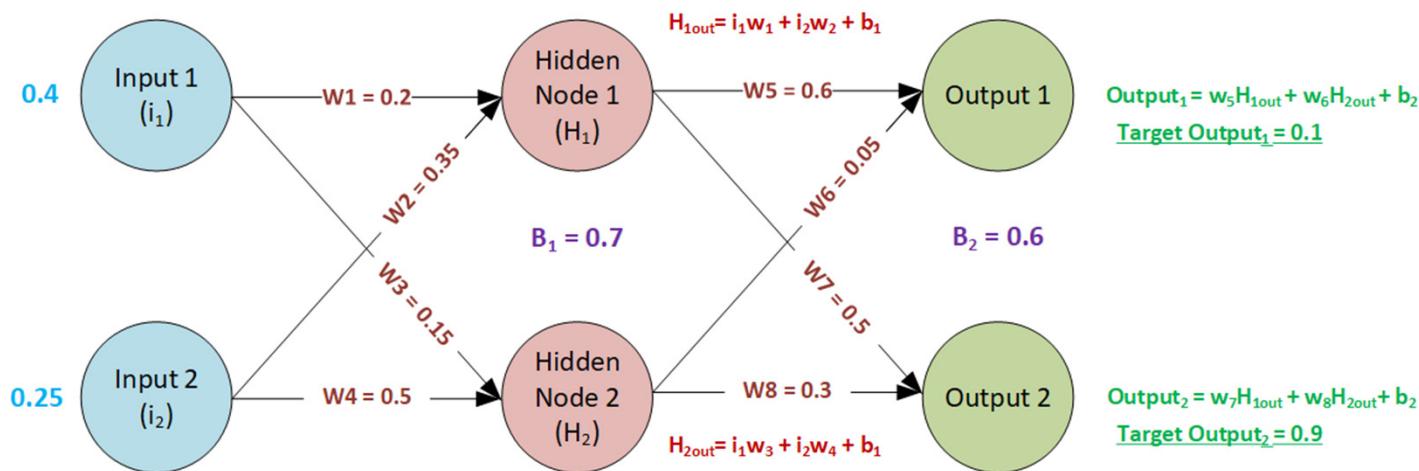
Backpropagation: Revisiting our Neural Net



- 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



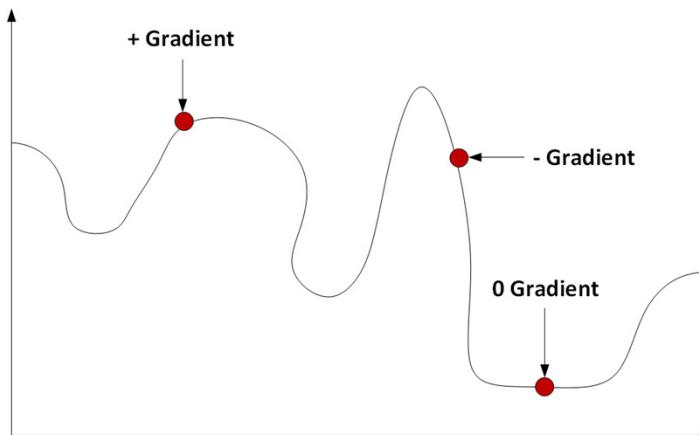
Backpropagation – The full cycle

- 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

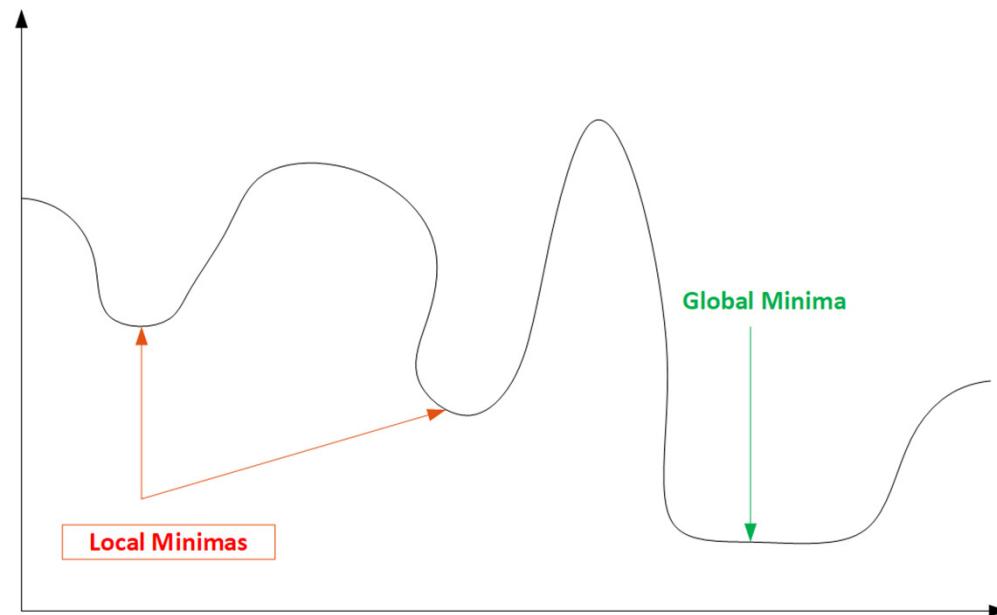
- By adjusting the weights to lower the loss, we are performing gradient descent. This is an 'optimization' problem.
- Backpropagation is simply the method by which we execute gradient descent
- Gradients (also called slope) are the direction of a function at a point, it's magnitude signifies how much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

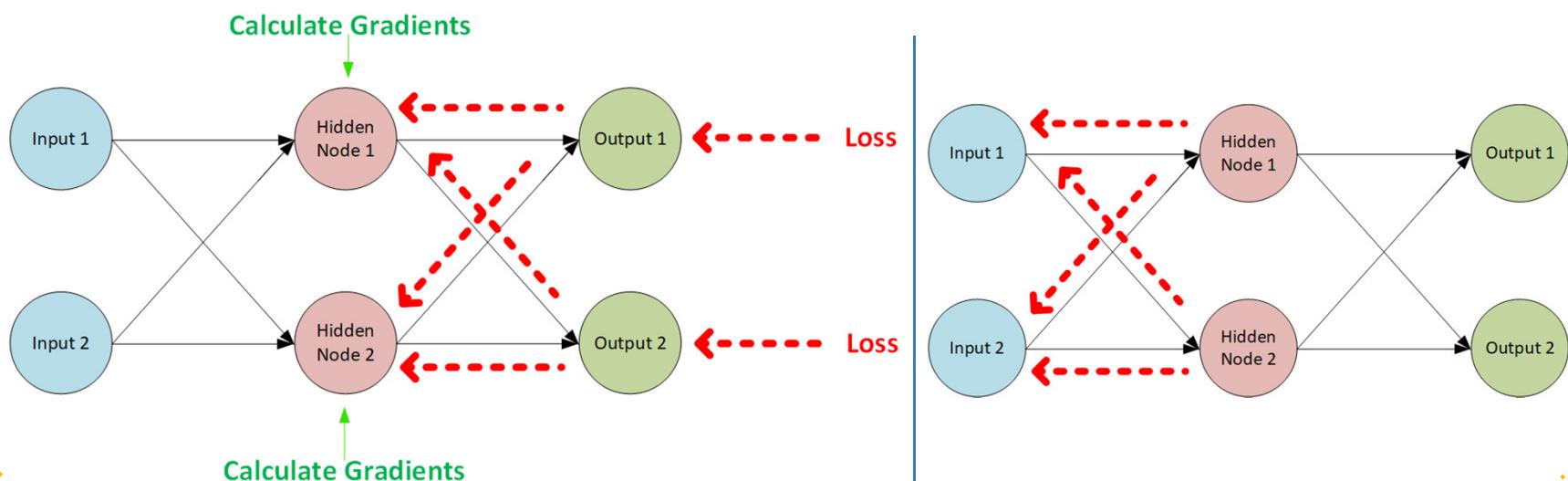
- From the previous section you have an idea of what we achieve with Backpropagation.
- It's a method of executing gradient descent or weight optimization so that we have an efficient method of getting the lowest possible loss.

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.





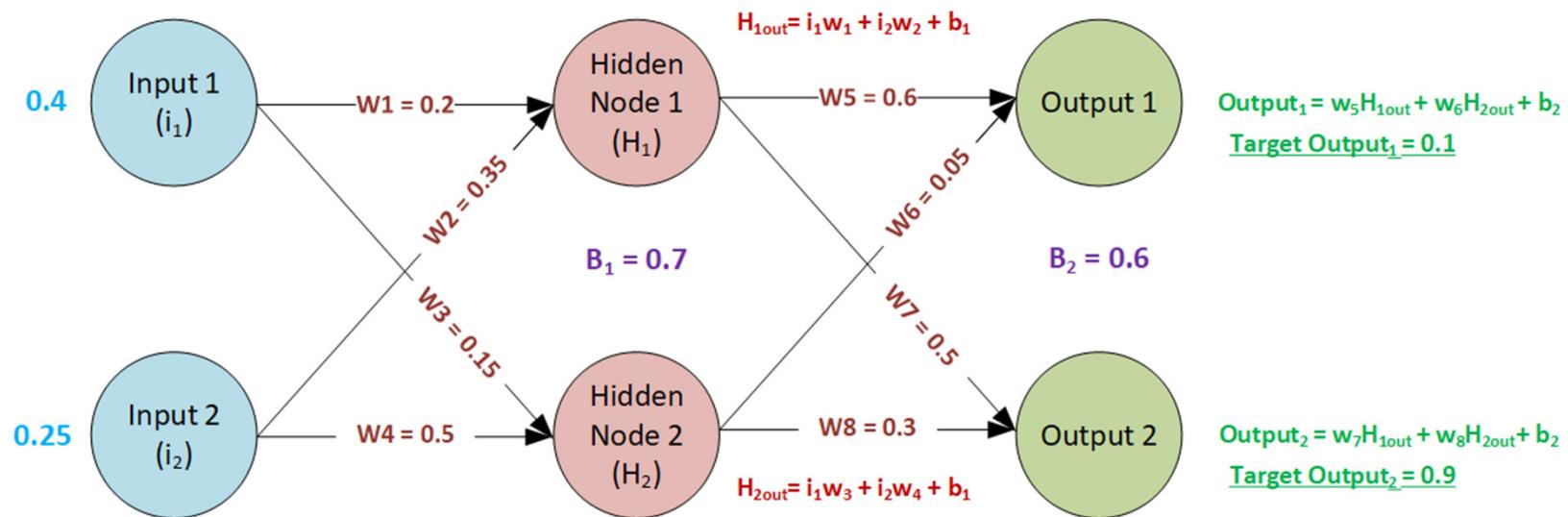
The Chain Rule

- 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:

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

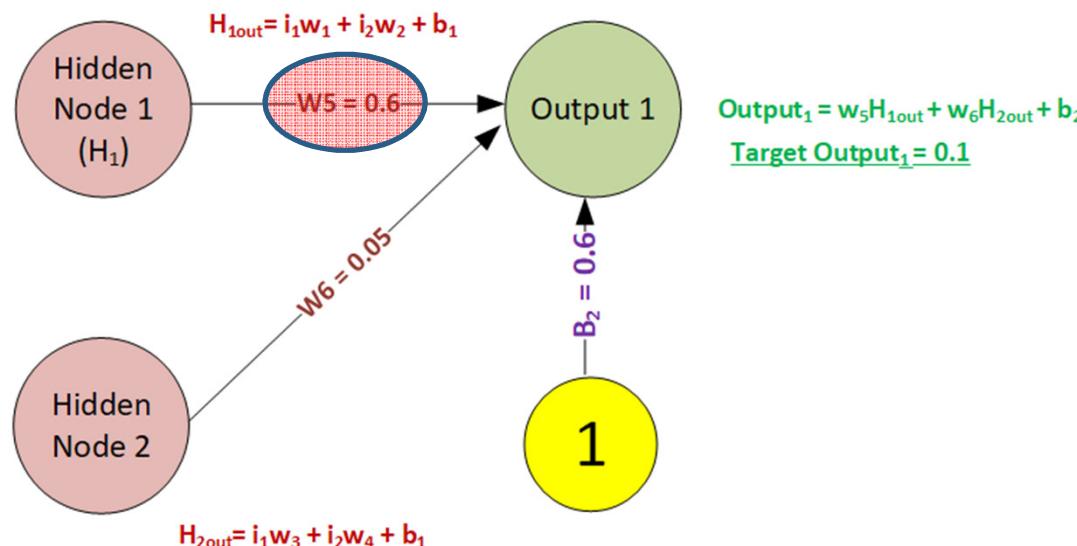


Let's take a look at our previous basic NN





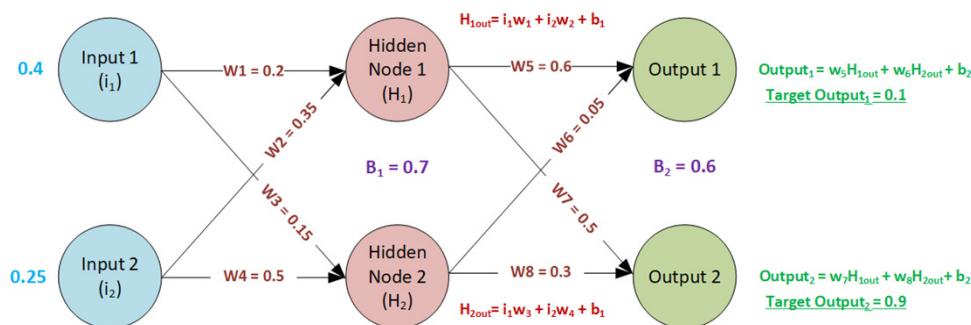
We use the Chain Rule to Determine the Direction the Weights Should Take



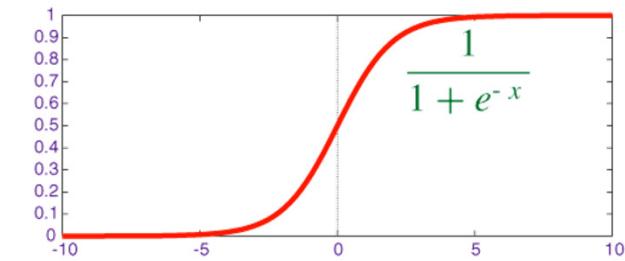
- Let's take a look at W_5 , how does a change in W_5 affect the Error at Output_1 ?
- Should W_5 be increased or decreased?



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:

- $H_1 = \mathbf{0.704225}$

- $H_2 = \mathbf{0.707857}$

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

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

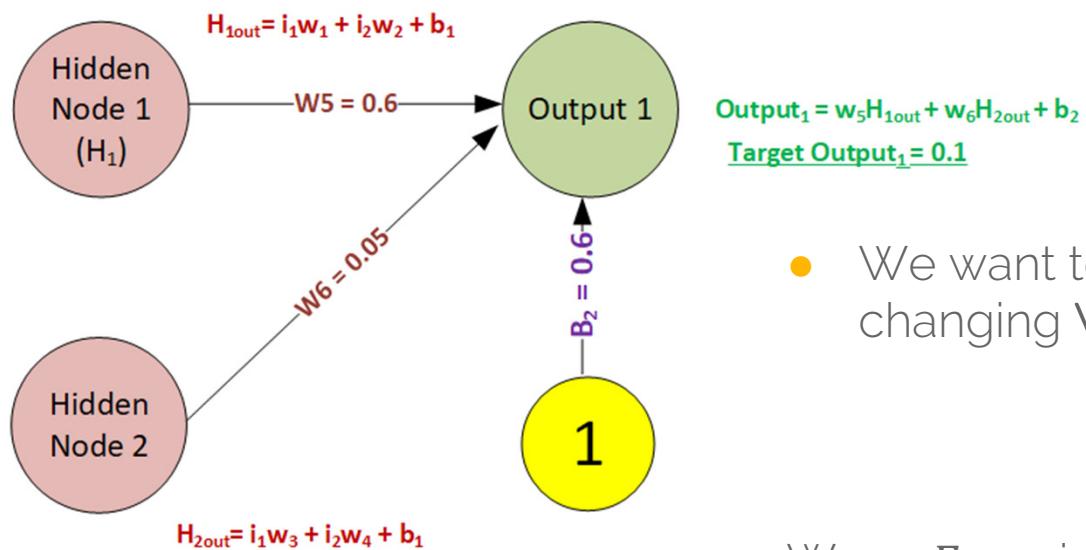


Slight Change in Our MSE Loss Function

- 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

- $\frac{dE_{total}}{dw_5} = \frac{dE_{total}}{dOut_1} \times \frac{dOut_1}{dNetOutput_1} \times \frac{dNetOutput_1}{dw_5}$
1 **2** **3**
- $E_{total} = \frac{1}{2}(target_{o1} - out_1)^2 + \frac{1}{2}(target_{o2} - out_2)^2$

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?



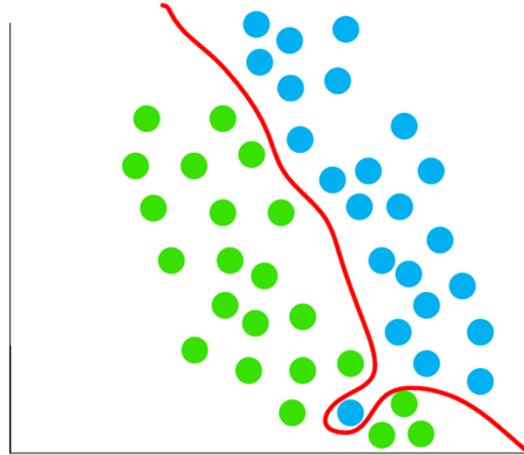
What Makes a Good Model?

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

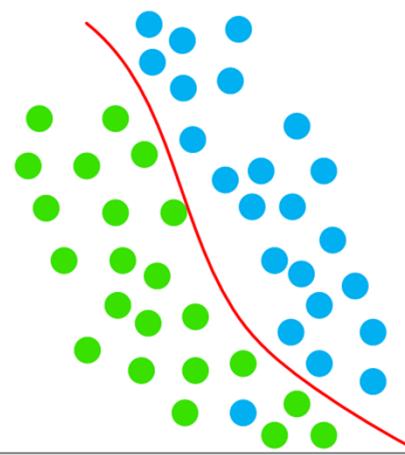


What Makes a Good Model?

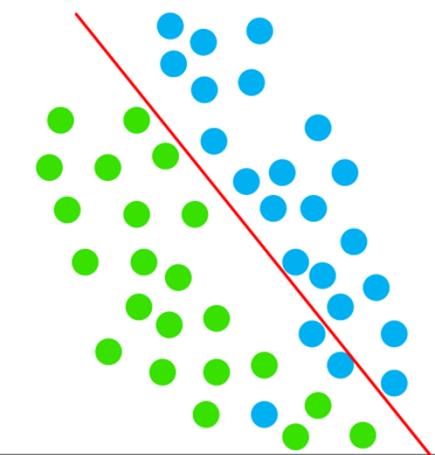
Model A



Model B



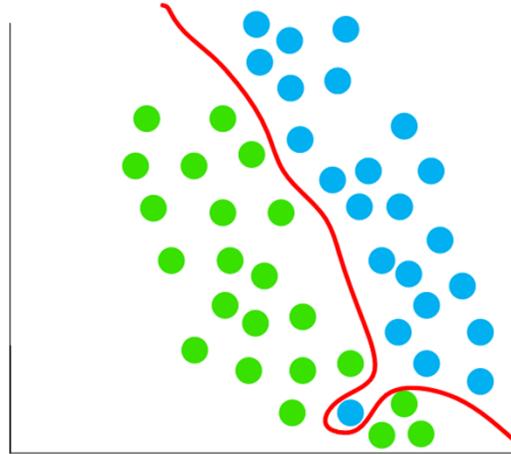
Model C



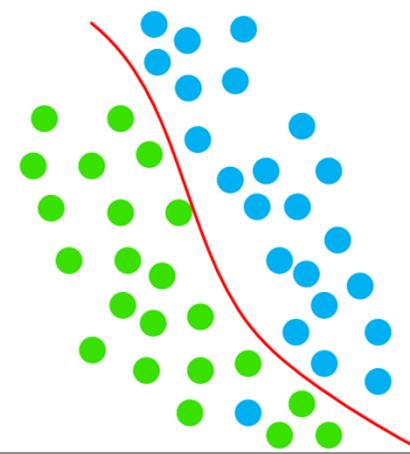


What Makes a Good Model?

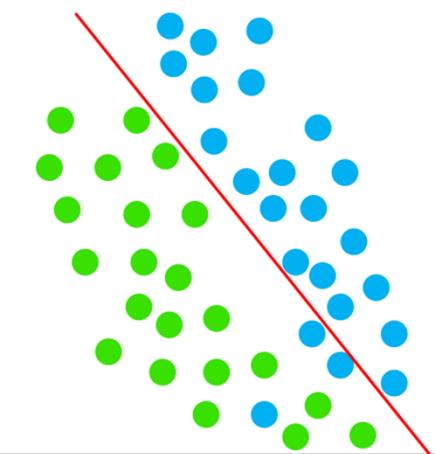
- Overfitting



- Ideal or Balanced



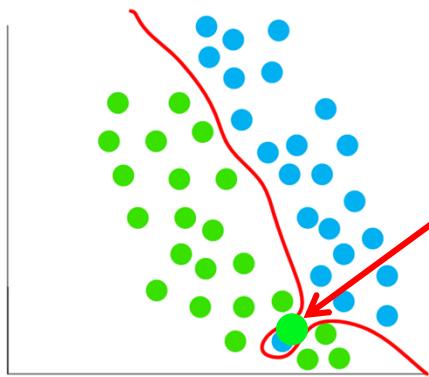
- Underfitting





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? (whose 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



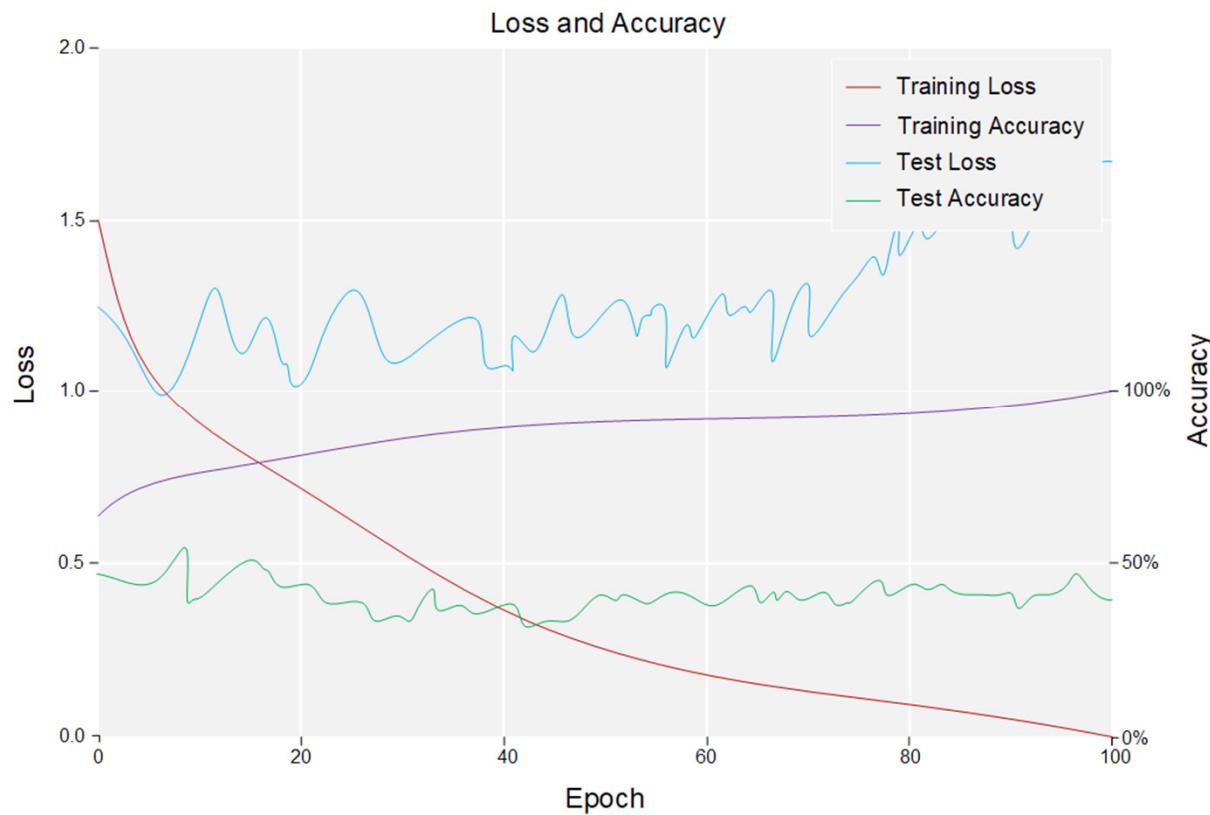
How do we know if we've Overfit?

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

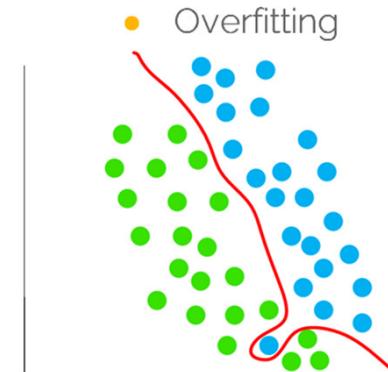


- Examine our Training Loss and Accuracy. They're both heading the right directions!
- But, what's happening to the loss and accuracy on our Test data?
- This is classic example of Overfitting to our training data



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!

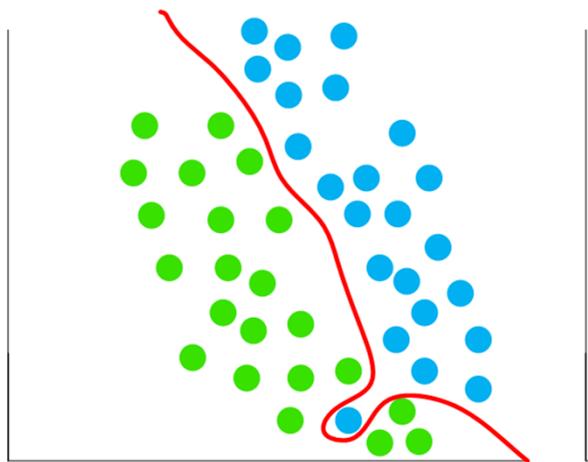
- It is better practice to regularize than reduce our model complexity.



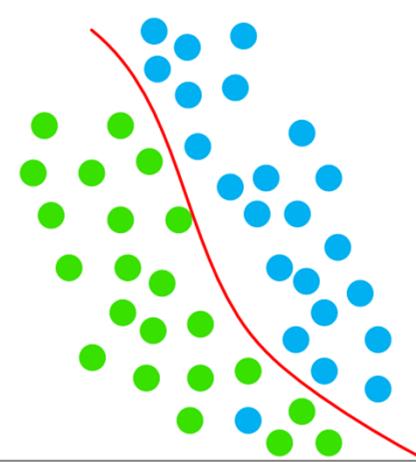
What is Regularization?

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

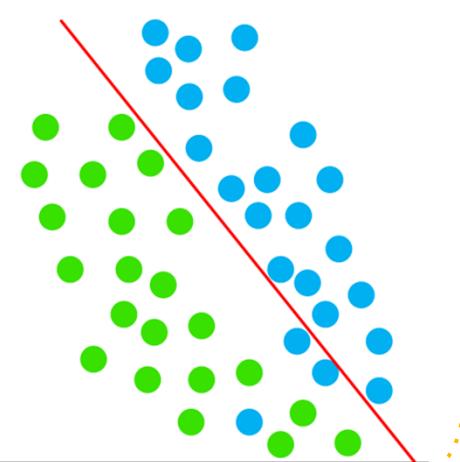
- Overfitting



- Ideal or Balanced



- Underfitting





Types of Regularization

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



L1 And L2 Regularization

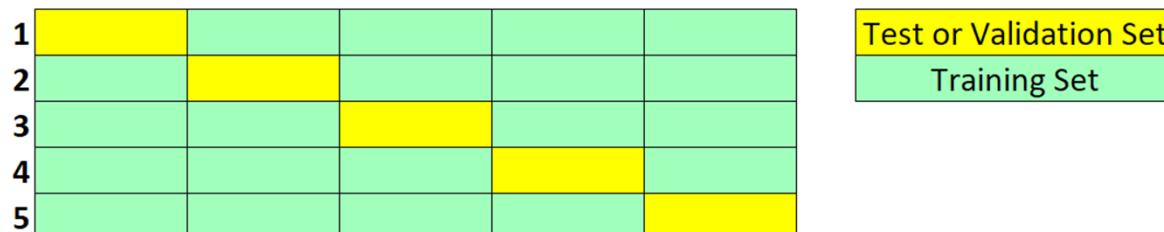
- L1 & L2 regularization are techniques we use to penalize large weights. Large weights or gradients manifest as abrupt changes in our model's decision boundary. By penalizing, we're really making them smaller.
- L2 also known as Ridge Regression
 - $Error = \frac{1}{2}(target_{01} - out_1)^2 + \frac{\lambda}{2} \sum w_i^2$
- 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.)



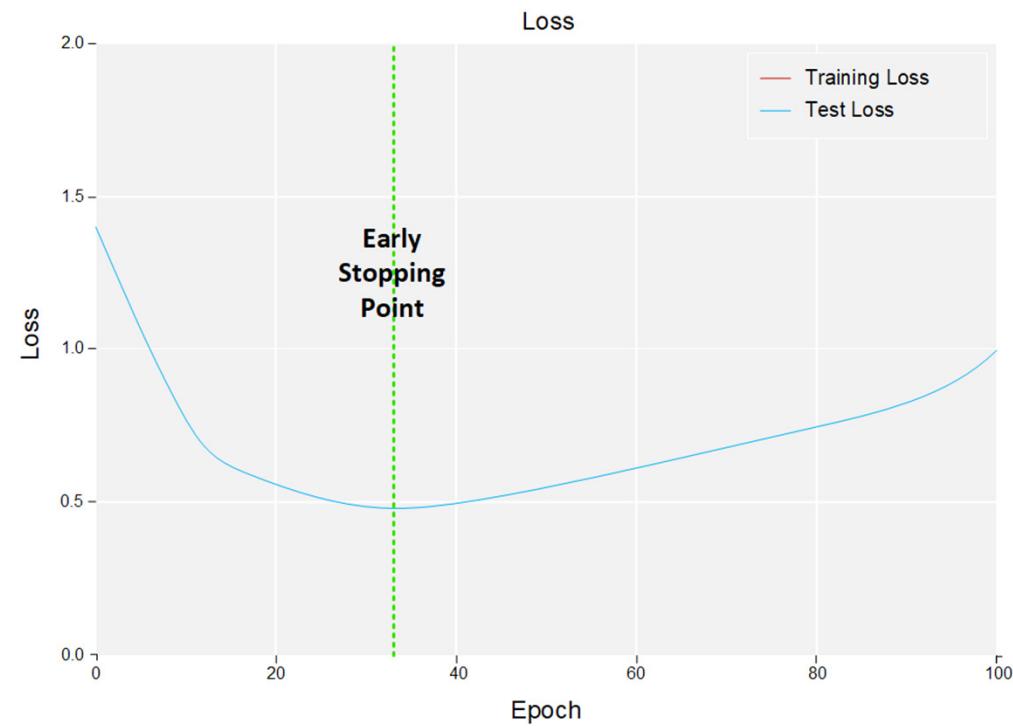
Cross Validation

- 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



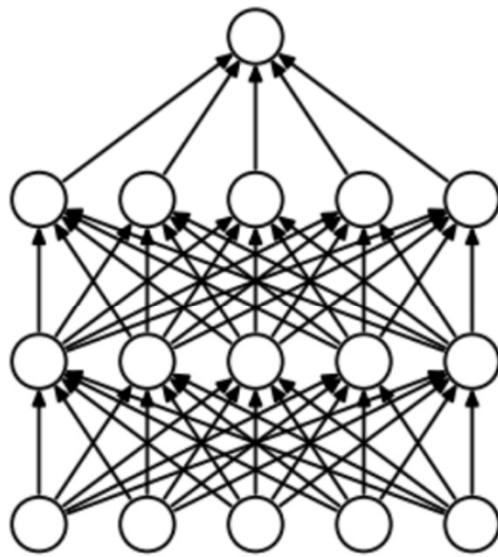


Dropout

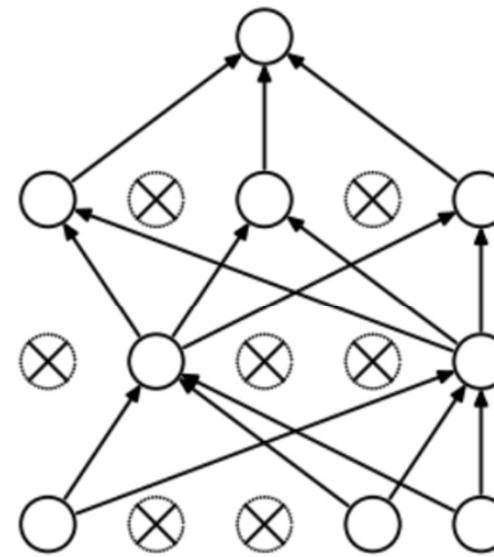
- Dropout refers to dropping nodes (both hidden and visible) in a neural network with the aim of reducing overfitting.
- In training certain parts of the neural network are ignored during some forward and backward propagations.
- Dropout is an approach to regularization in neural networks which helps 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

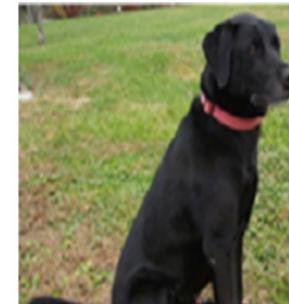
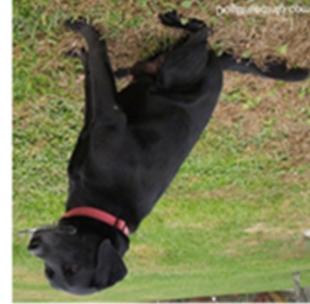


(b) After applying dropout.



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

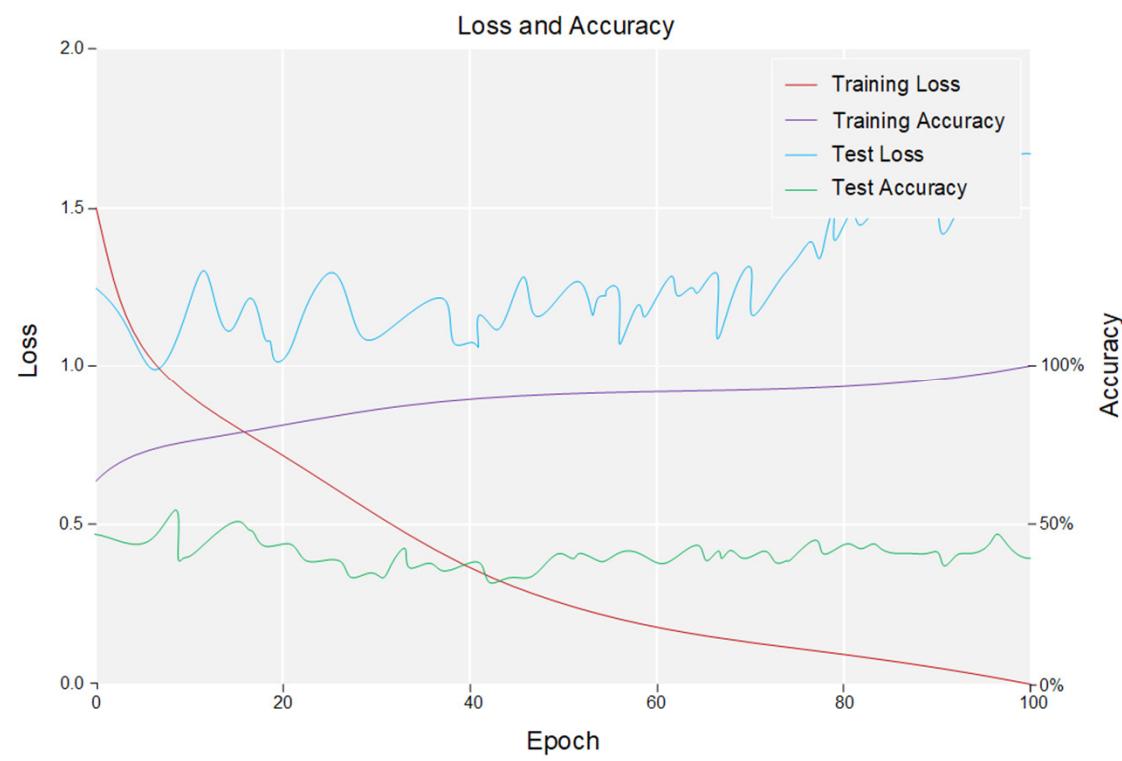
- Many confuse iterations and Epochs (I was one of them)
- However, the difference is quite simple, Iterations are the number of batches we need to complete one Epoch.
- In our previous example, we had 1000 items in our dataset, and set a batch size of 100. Therefore, we'll need 10 iterations (100×10)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

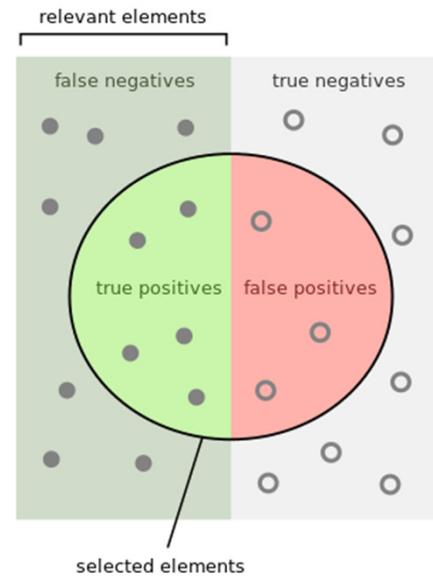
- Recall – How much of the positive classes did we get correct
 - $$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$
- Precision – Out of all the samples how much did the classifier get right
 - $$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$
- F-Score – 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

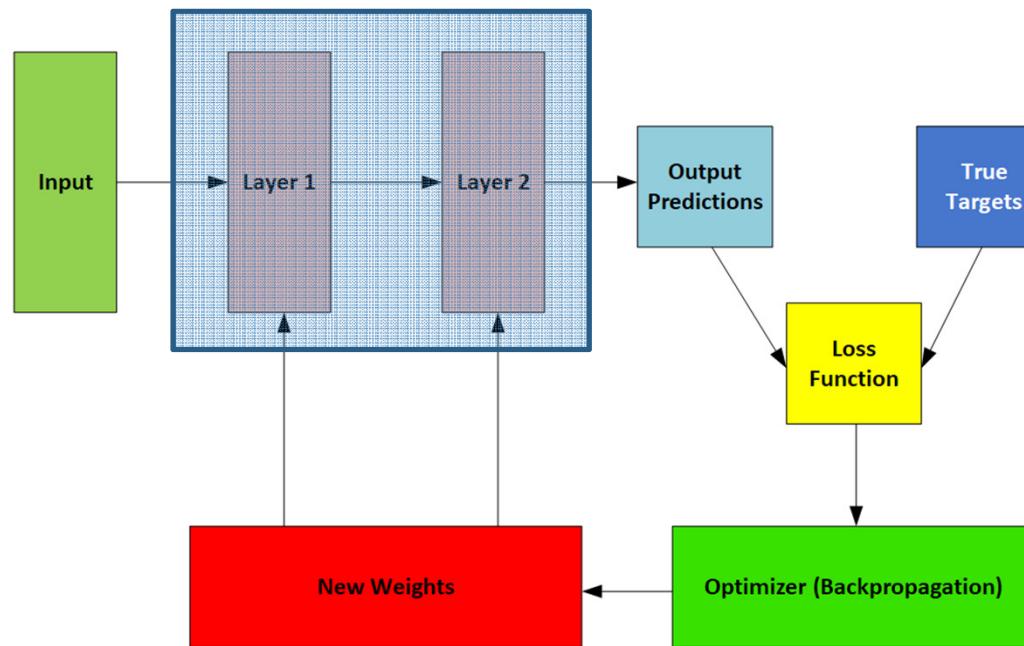


Training step by step

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.

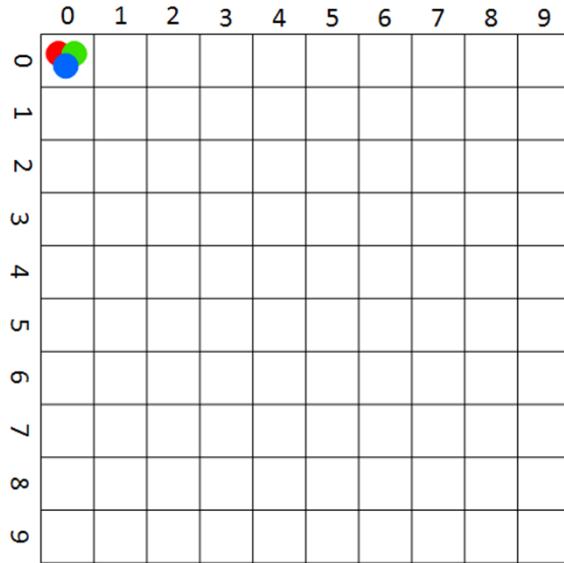


Why CNNs?

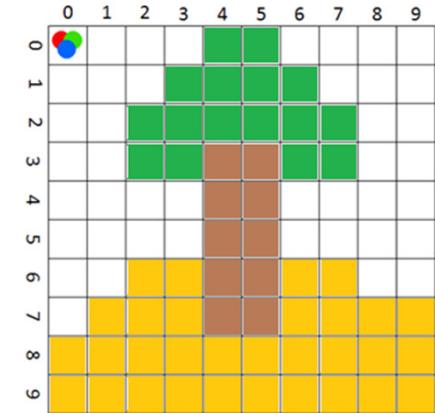
- Because Neural Networks don't scale well to images data
- Remember in our intro slides we discussed how images are stored.



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

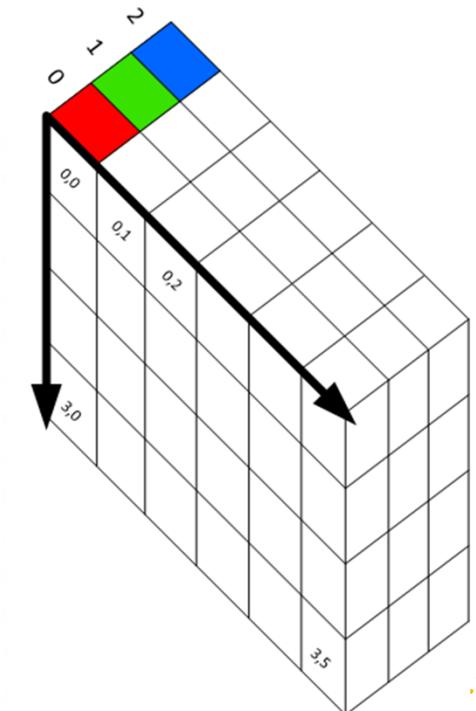


How do Computers Store Images?



- 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



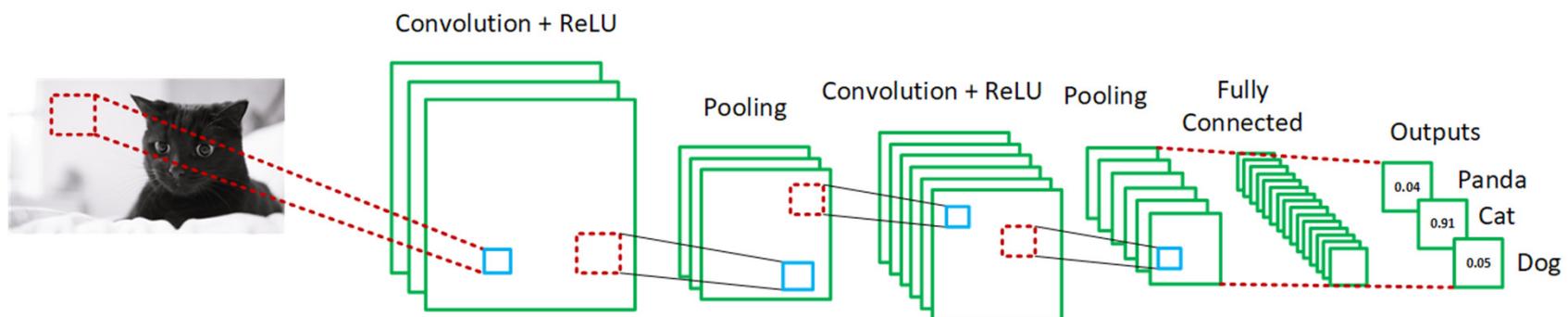


Why NNs Don't Scale Well to Image Data

- 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





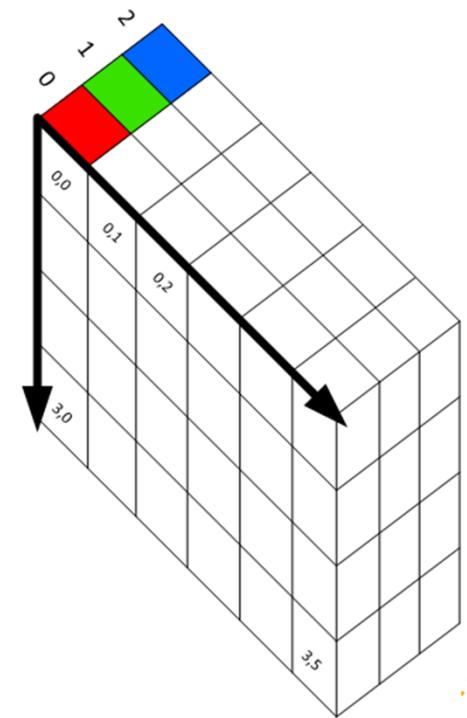
Why of 3D Layers?

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



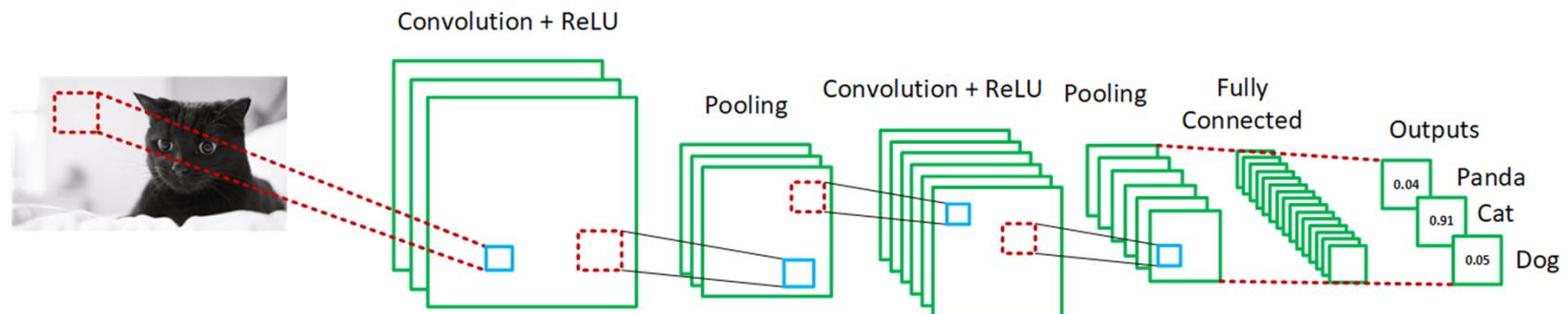
CNN's use a 3D Volume Arrangement for it's Neurons

- 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

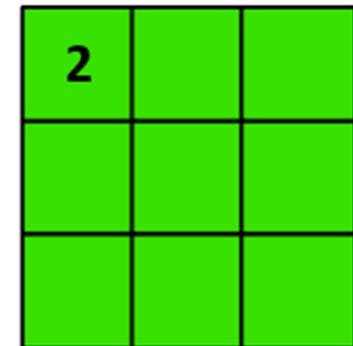


Applying out Kernel / Filter

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

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

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

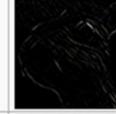
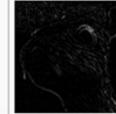
2	1	



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/

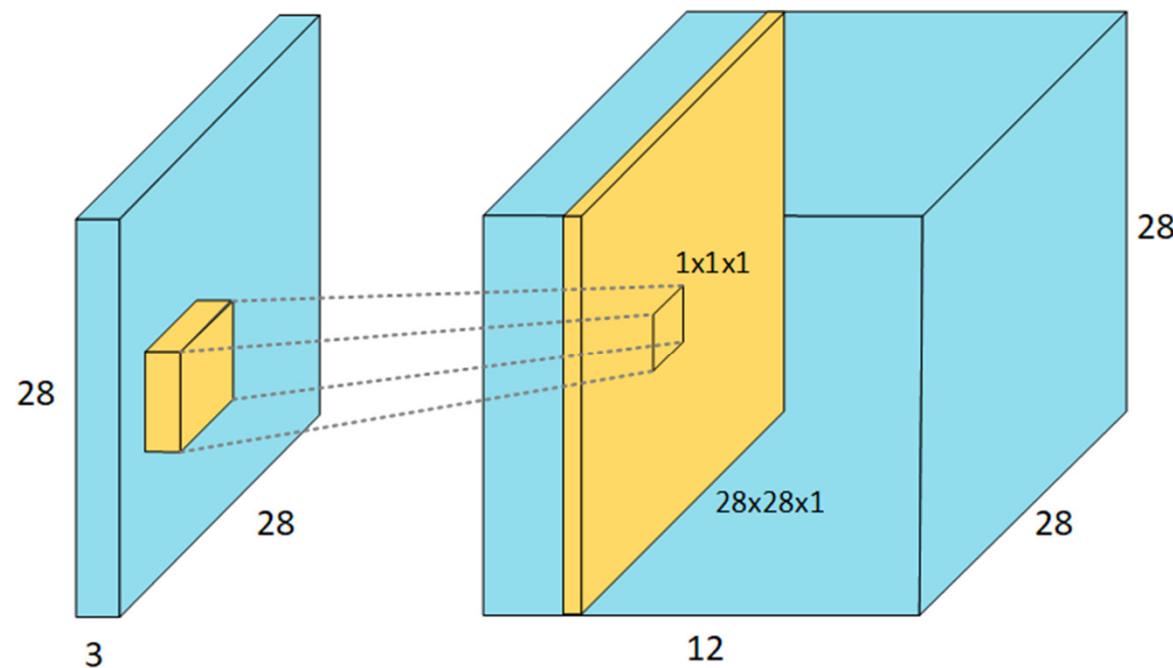


We Need Many Filters, Not Just One

- 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?



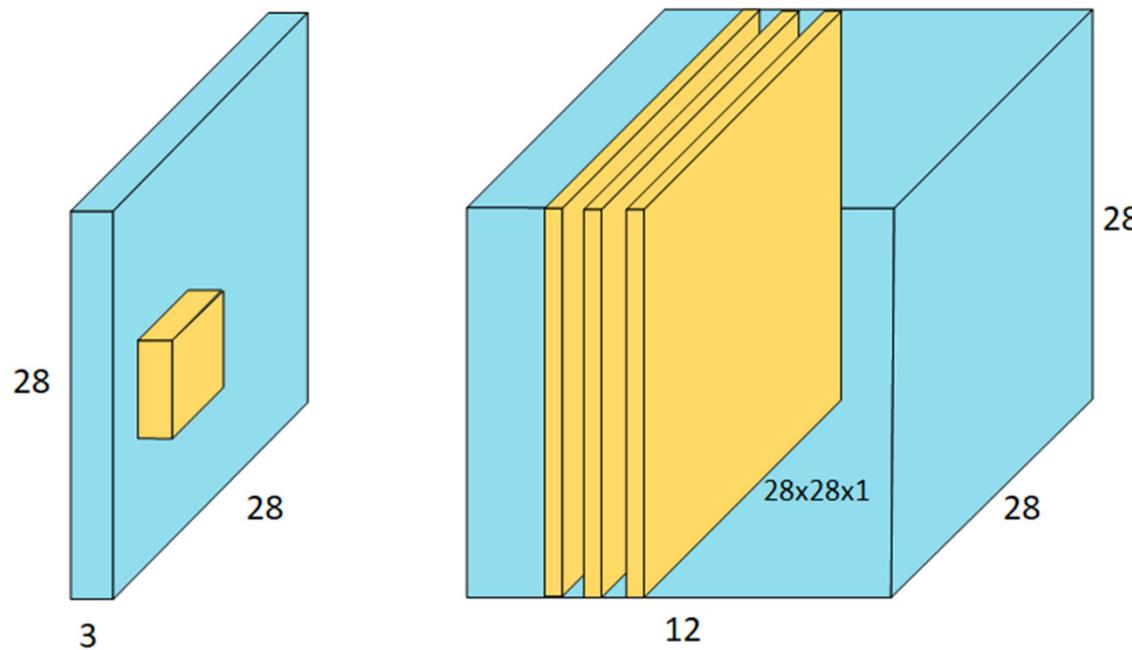
Illustration of our Multiple 3D Kernels/Filters



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



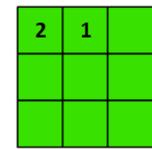
Designing Feature Maps

- 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

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

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





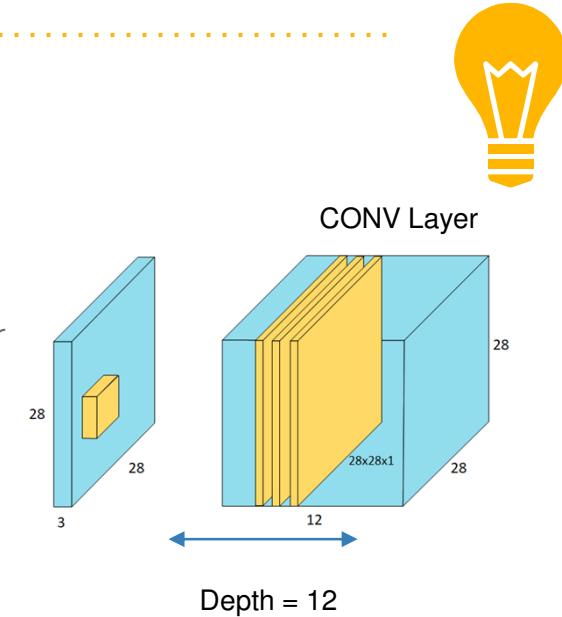
Designing Feature Maps

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

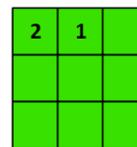


- Zero-padding is a very simple concept that refers to a border we apply to the input volume. Why is this volume needed?
- Remember in our first example, with stride set to 1, and our output of feature map was a 3x3 matrix. We haven't discussed deep networks much yet, but imagine we had multiple Convolution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

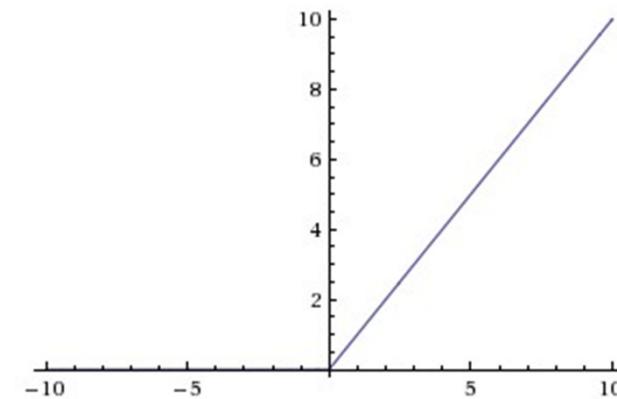
- Similarly as we saw in NN, we need to introduce non-linearity into our model (the convolution process is linear), we need to apply an activation function to the output of our CONV Layer



ReLU is the Activation Function of Choice

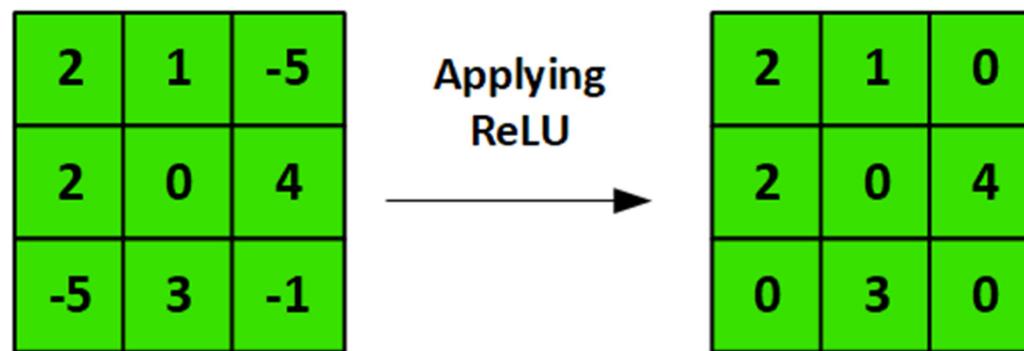
- ReLU or Rectified Linear Unit simply changes all the negative values to 0 while leaving the positive values unchanged.

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



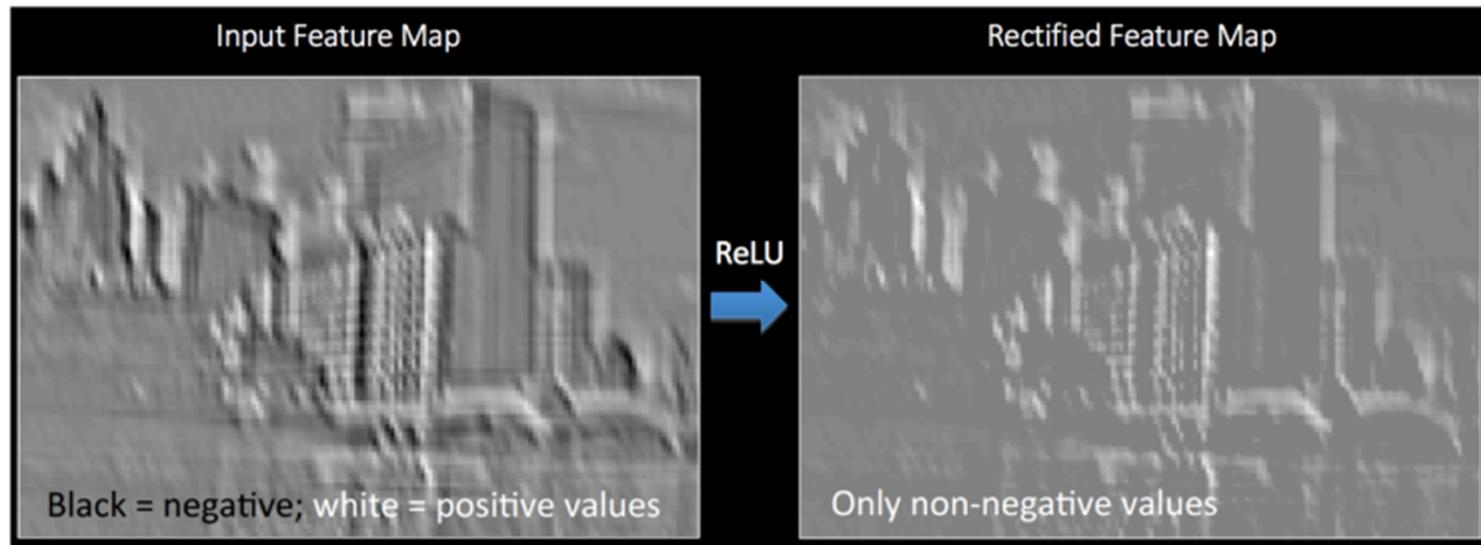


Applying ReLU





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



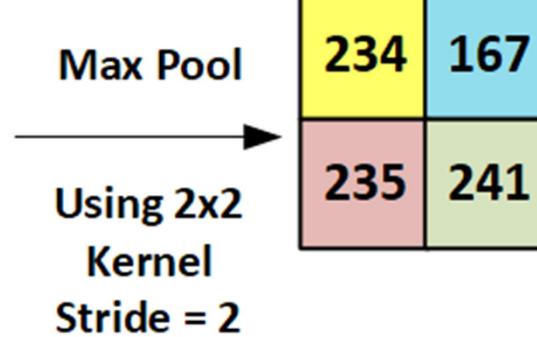
Pooling

- 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.



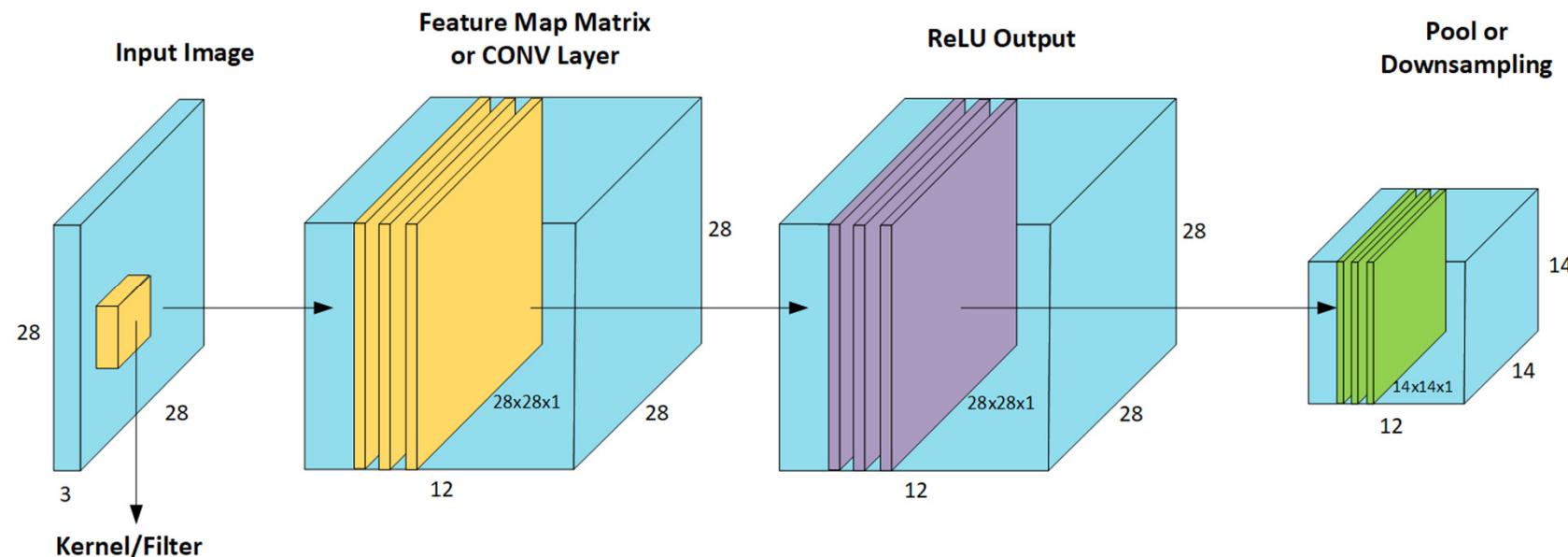
Example of Max Pooling

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





Our Layers so far



228



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



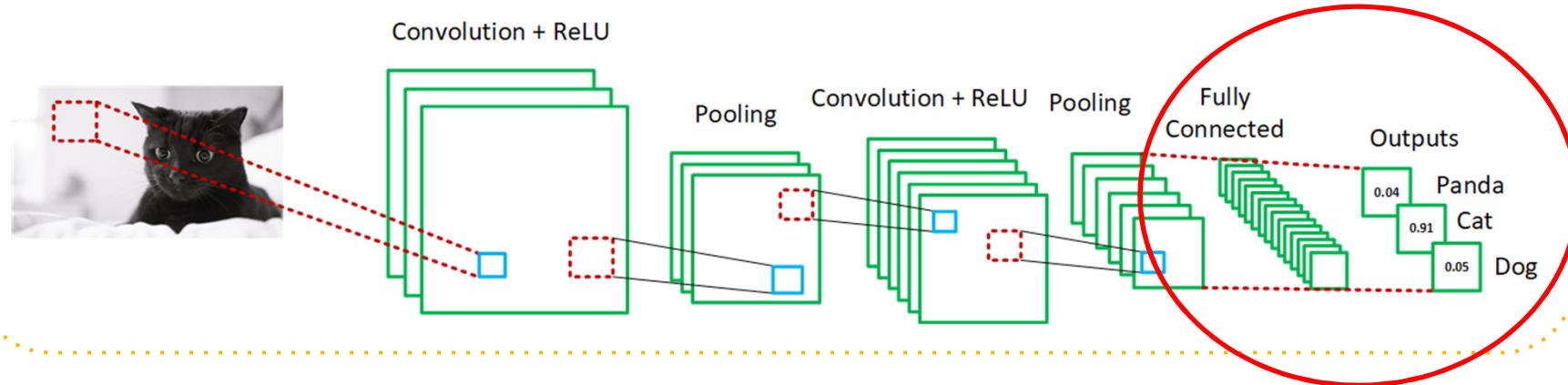
The Fully Connected (FC) Layer

- 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

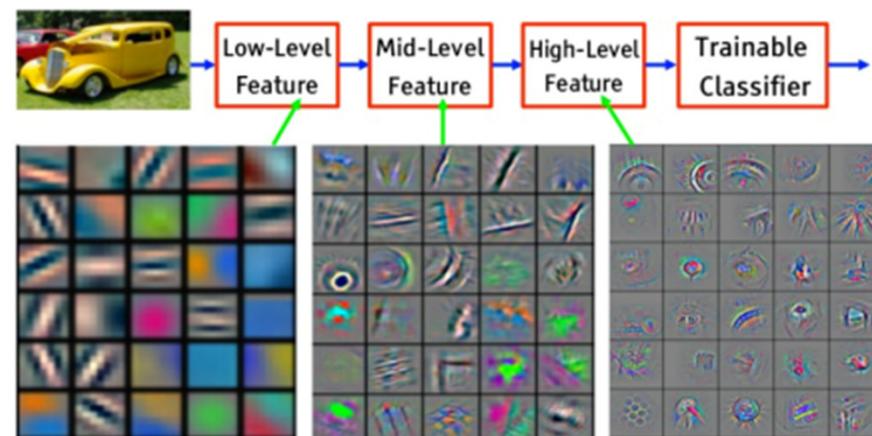
Training CNNs

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

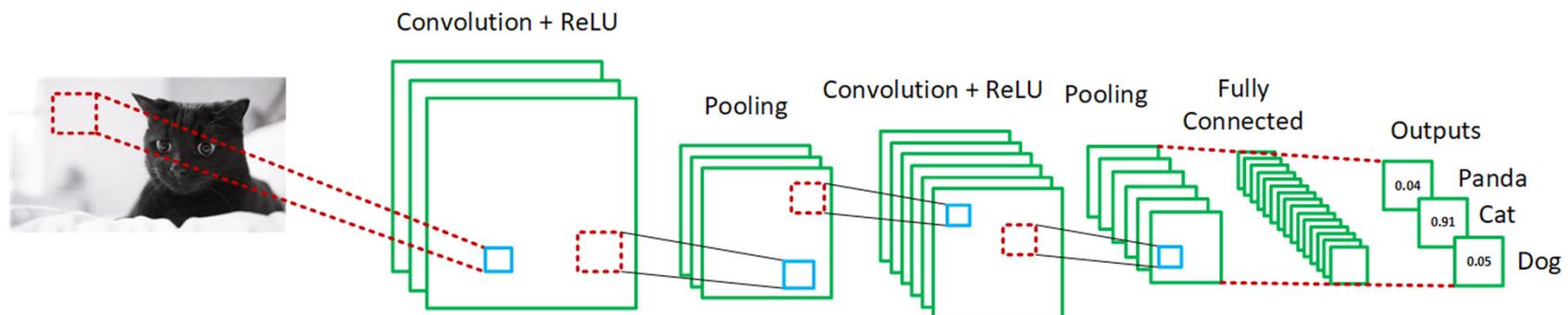


Source -
<http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf>



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.





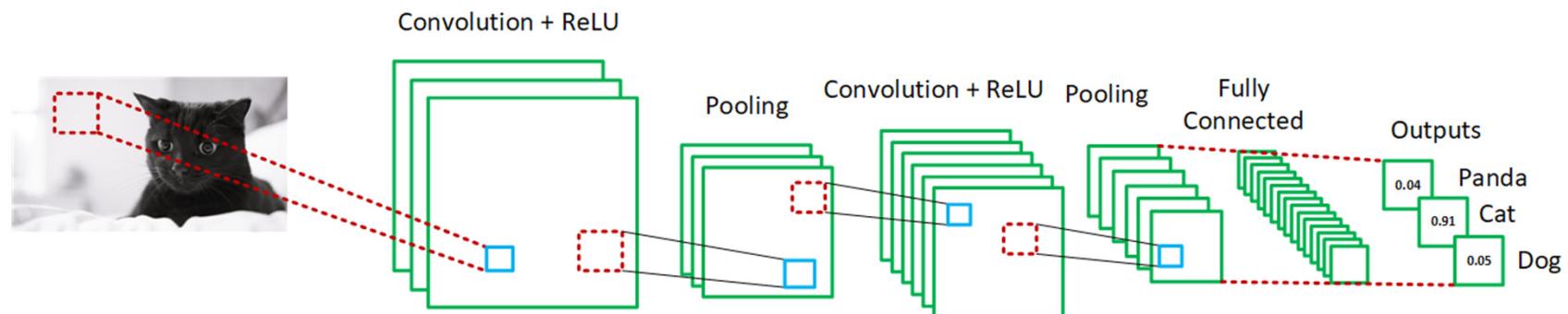
Basic CNN Design Rules 1

- 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

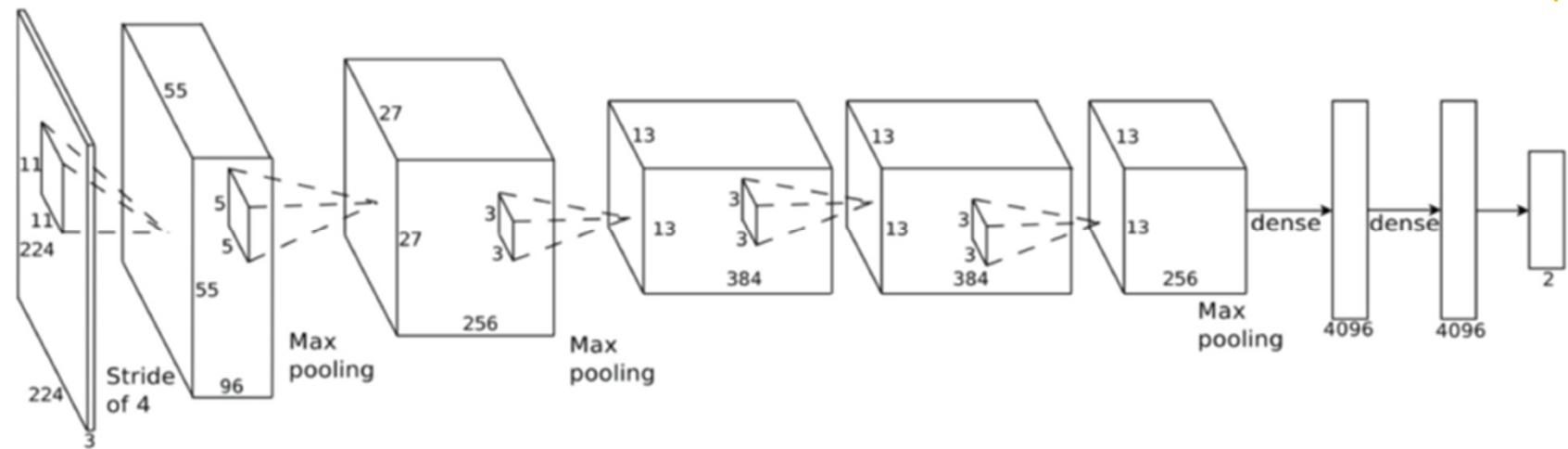
- The more hidden layers the more features, particularly high level features a CNN can learn. I like to use a minimum of 2, which is shown in the diagram below.
- Its flow is: Input -> Conv -> ReLU -> Pool -> Conv -> ReLU -> Pool -> FC - Output





Examples of some CNNs will be creating in this course

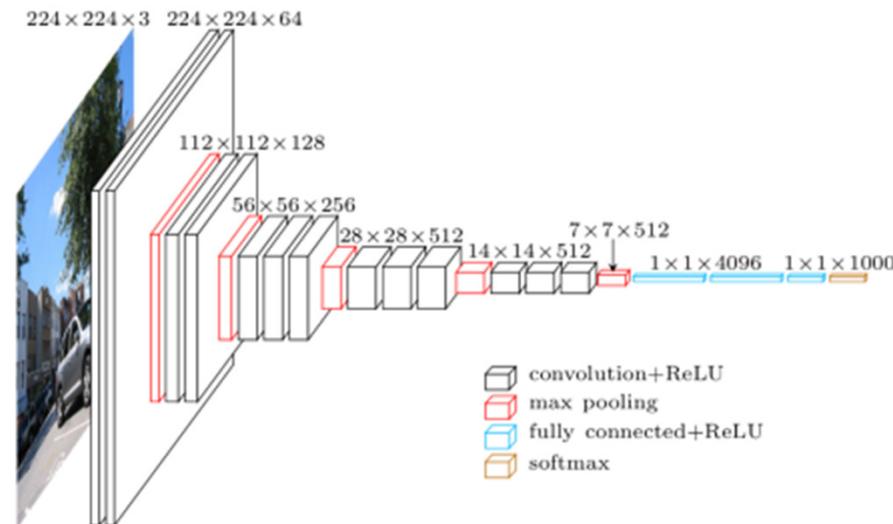
- AlexNet





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()
```



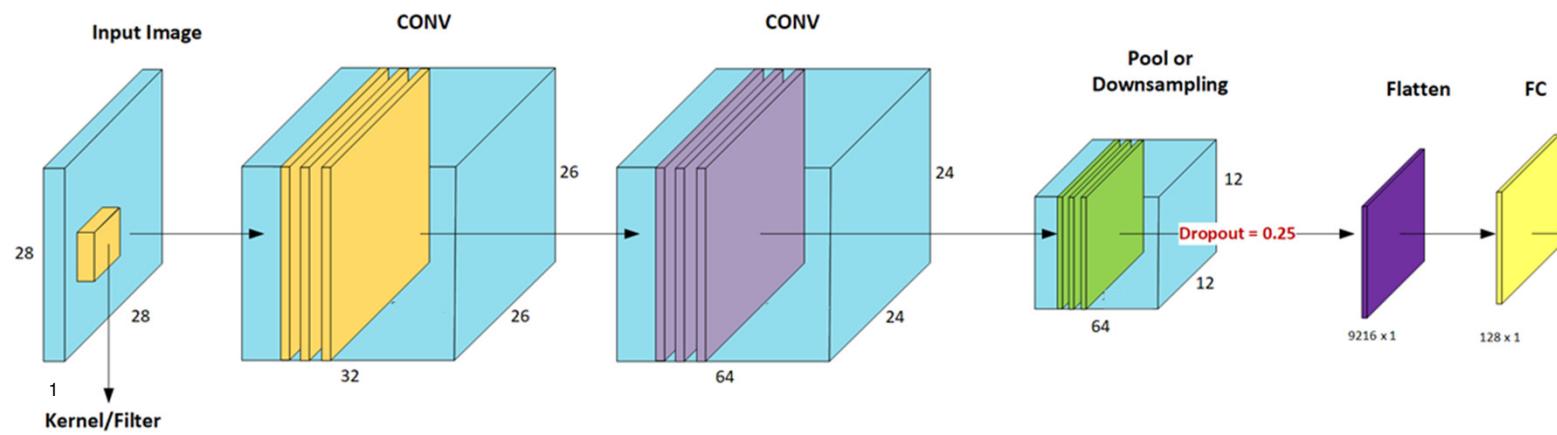
Creating our Convolution Layer

- 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

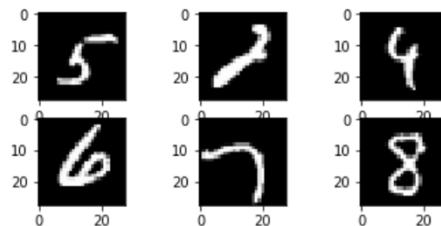
Building a Handwriting Recognition CNN

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.

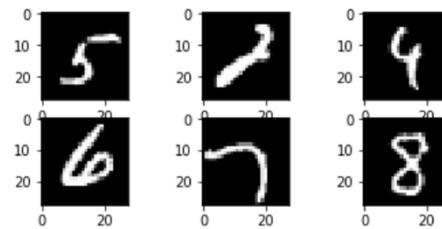


- Read more about MNIST here:
 - <http://yann.lecun.com/exdb/mnist/>
 - https://en.wikipedia.org/wiki/MNIST_database



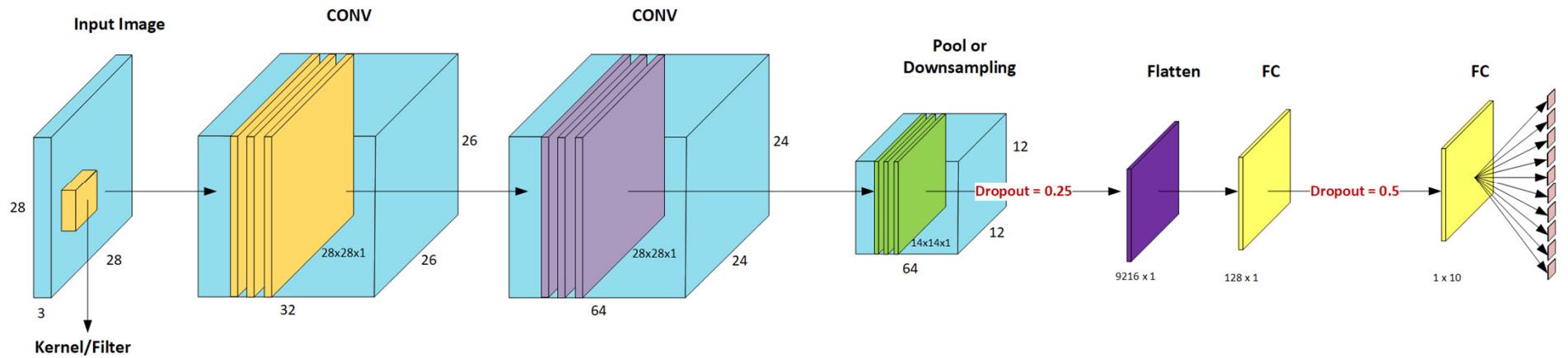
Problem and Aim

- 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

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

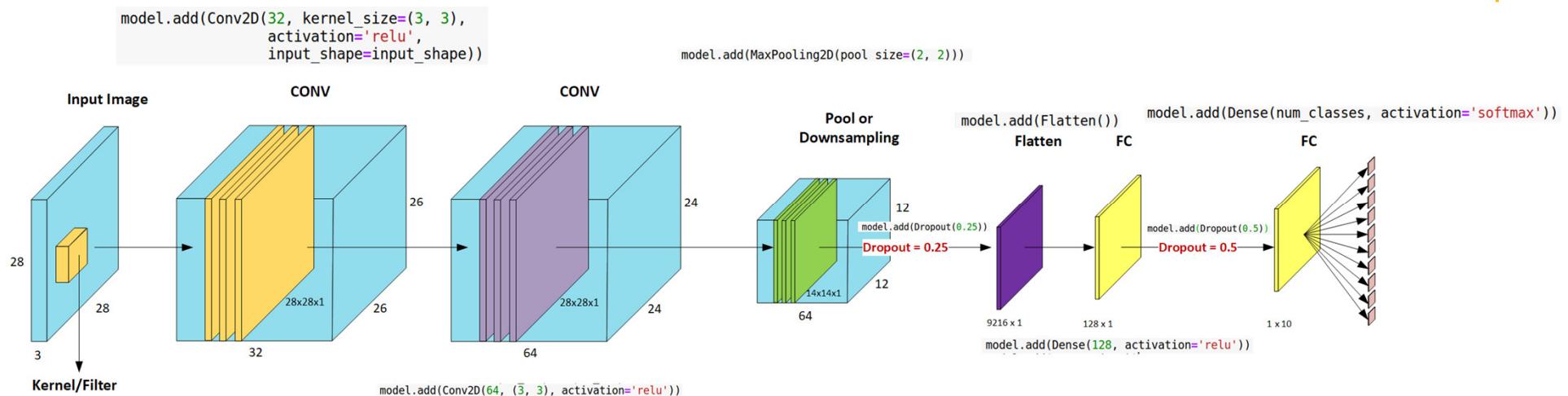
Label	0	1	2	3	4	5	6	7	8	9
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0		1	0	0	0

8.6

Building & Compiling Our Model



Our CNN Illustrated with Keras code



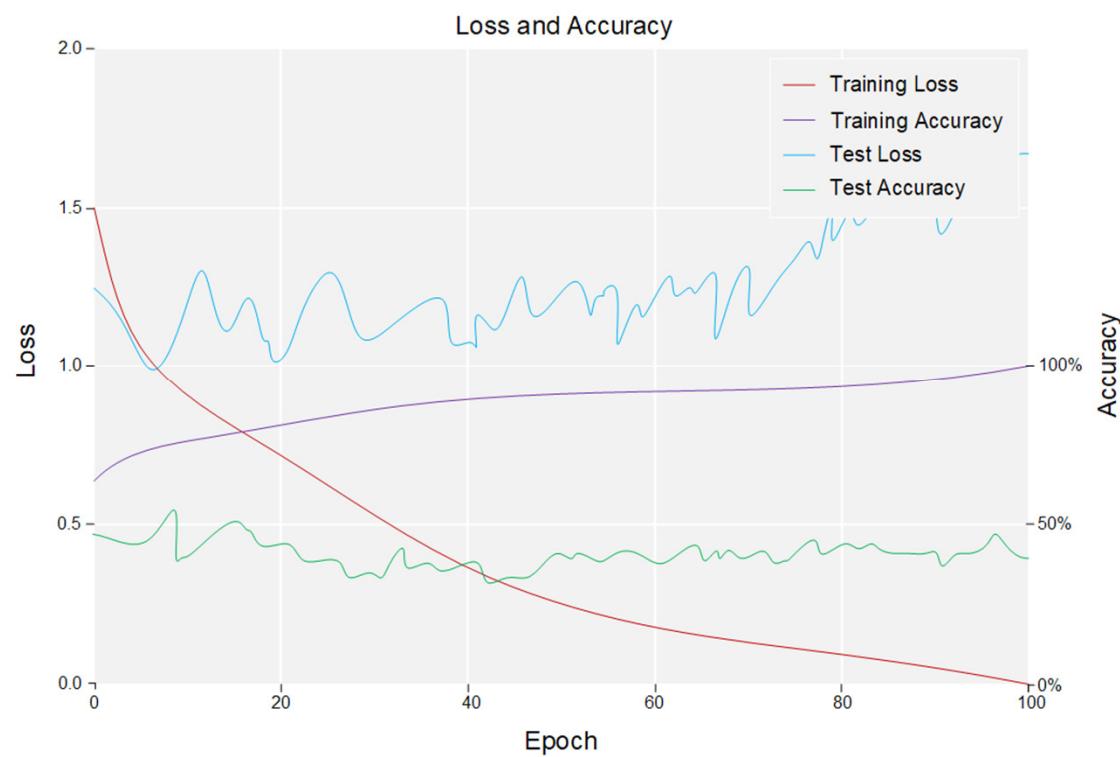
8.7

Training Our Classifier

8.8

Plotting Loss and Accuracy Charts

Loss and Accuracy



8.9

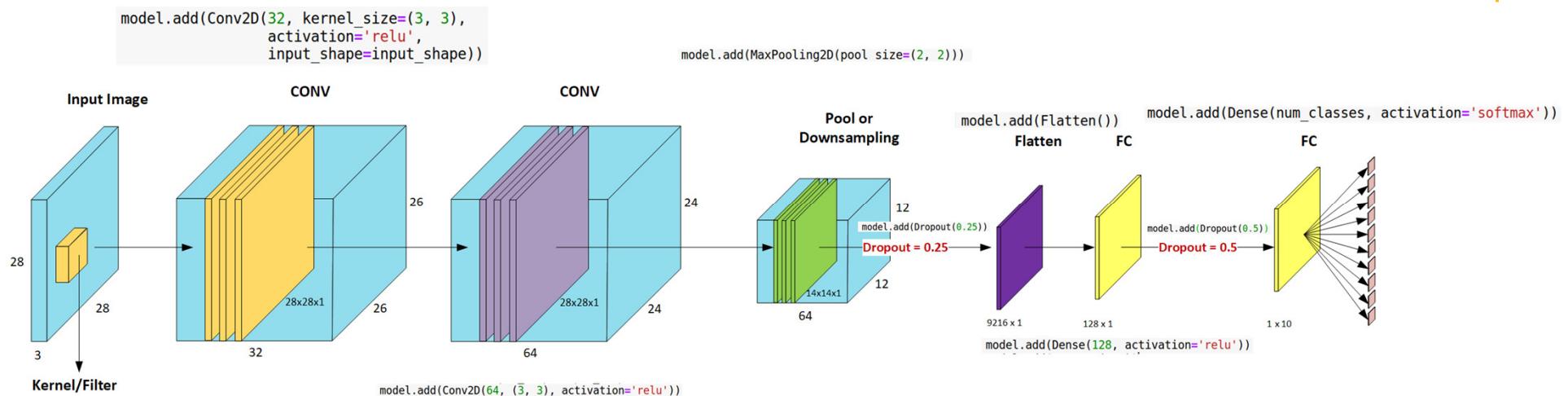
Saving and Loading Your Model

8.10

Displaying Your Model Visually



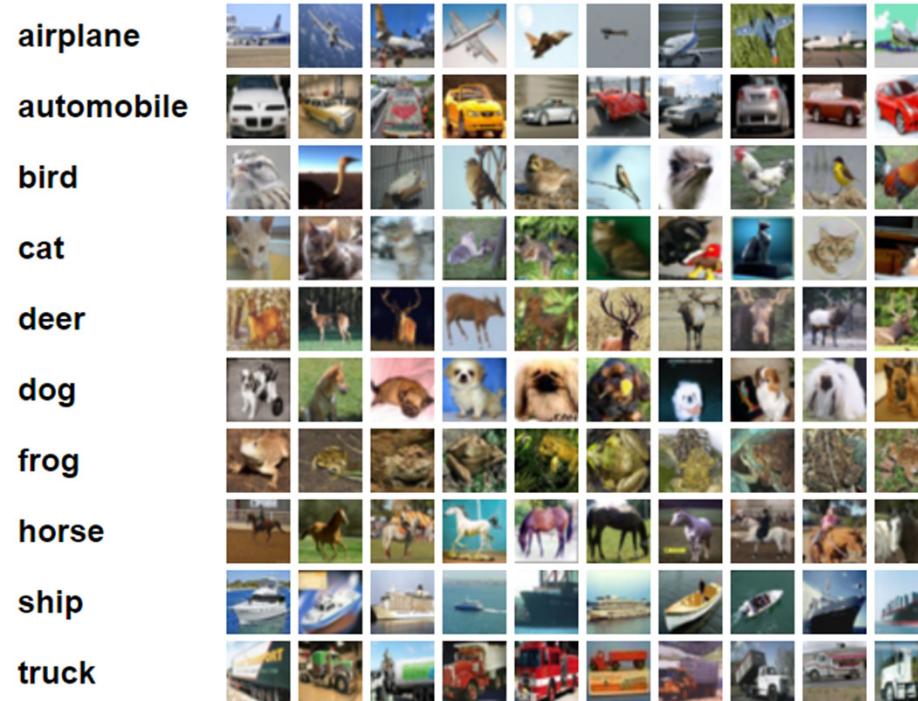
Our CNN Illustrated with Keras code



8.11

Building a Simple Image Classifier using CIFAR10

CIFAR-10



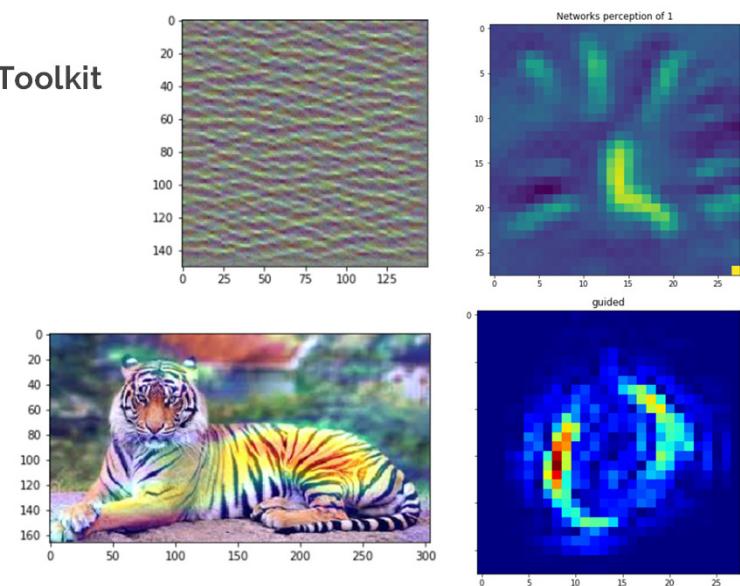
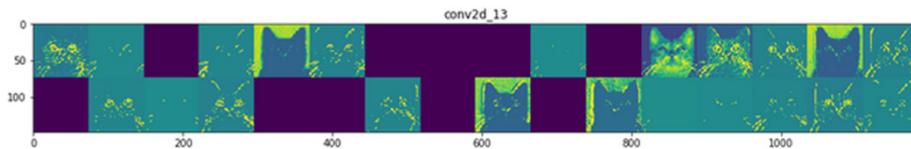
9.0

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

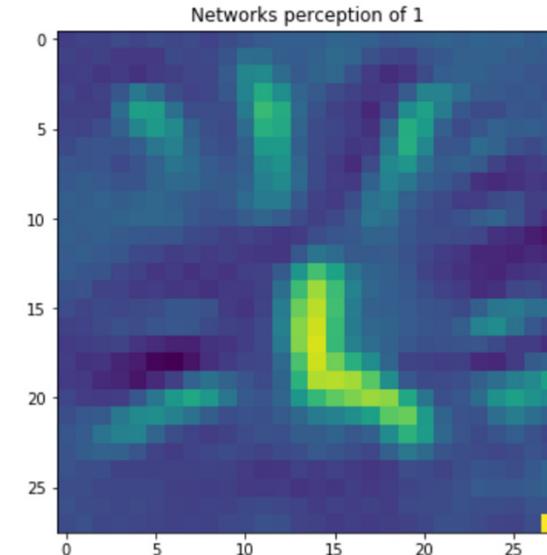
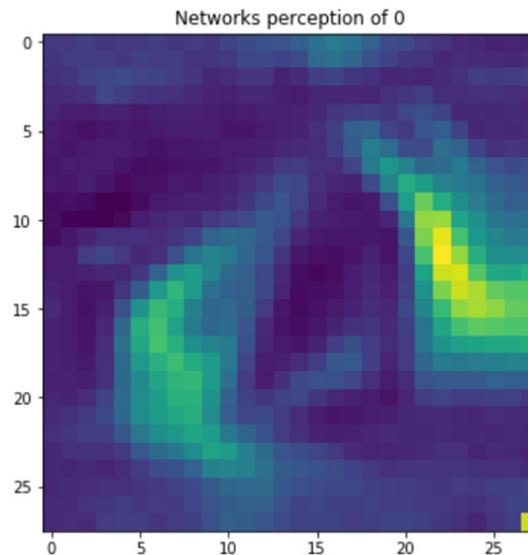


We can generate visualizations that show this, these visuals are called Activation Visualization

- 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?

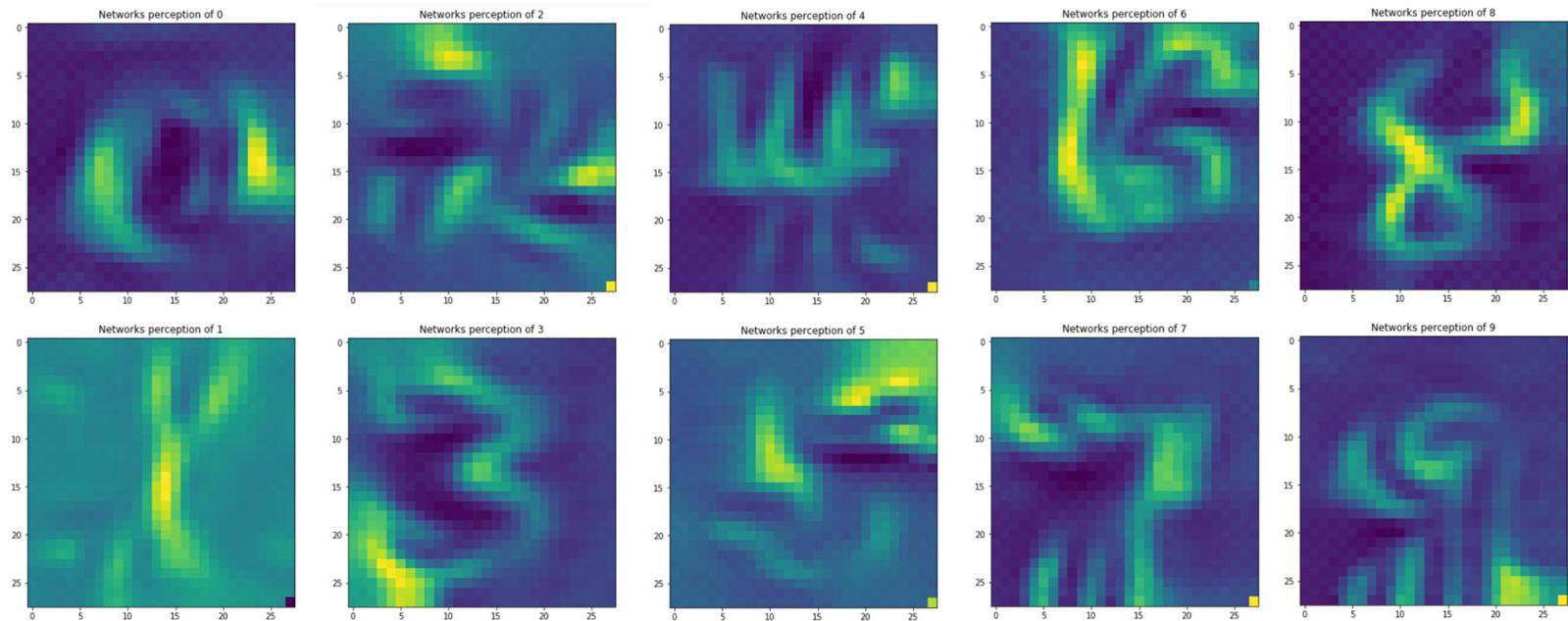
- We are generating an input image that maximizes the filter output activations. Thus we are computing,

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

- 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

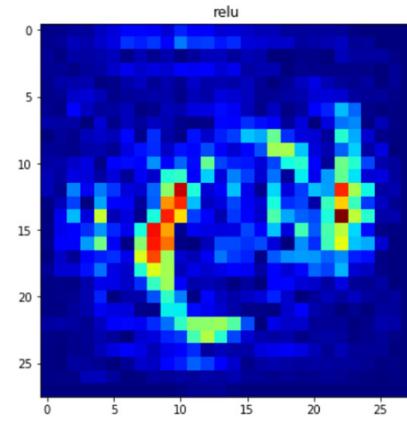
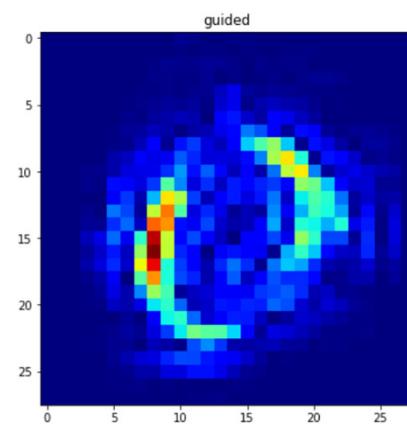


Saliency

- 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

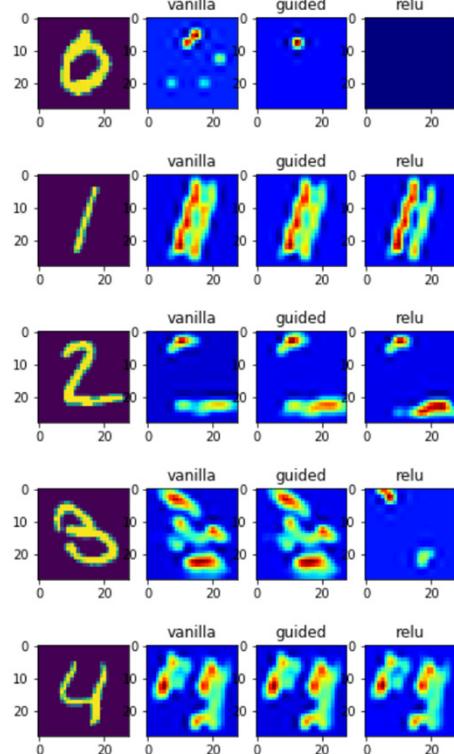




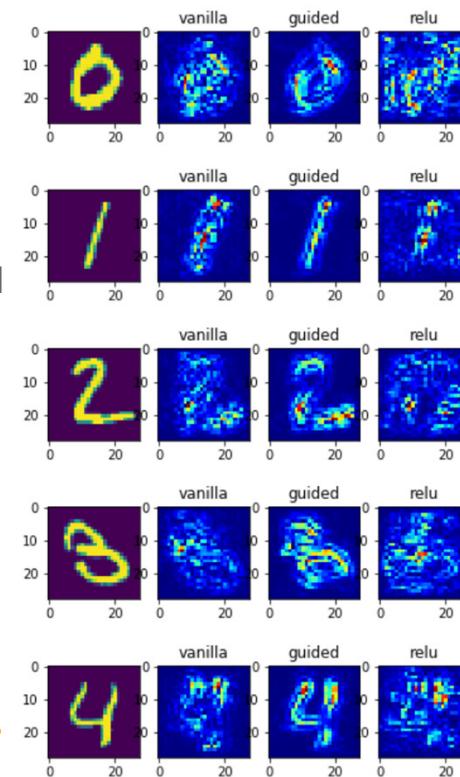
Class Activation Maps

- 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.



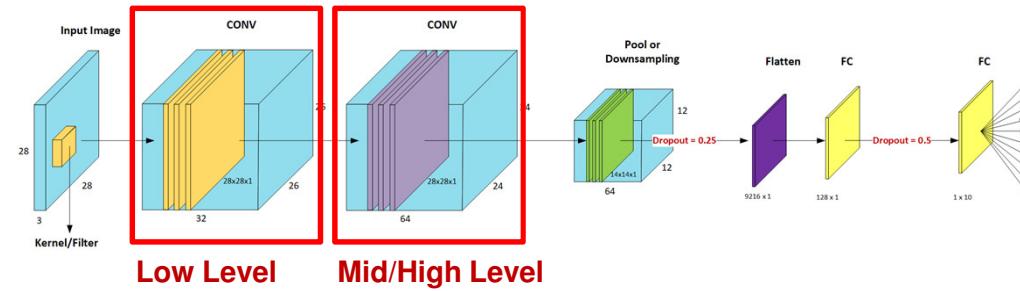
9.3

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

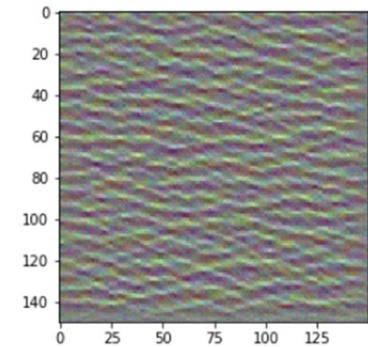
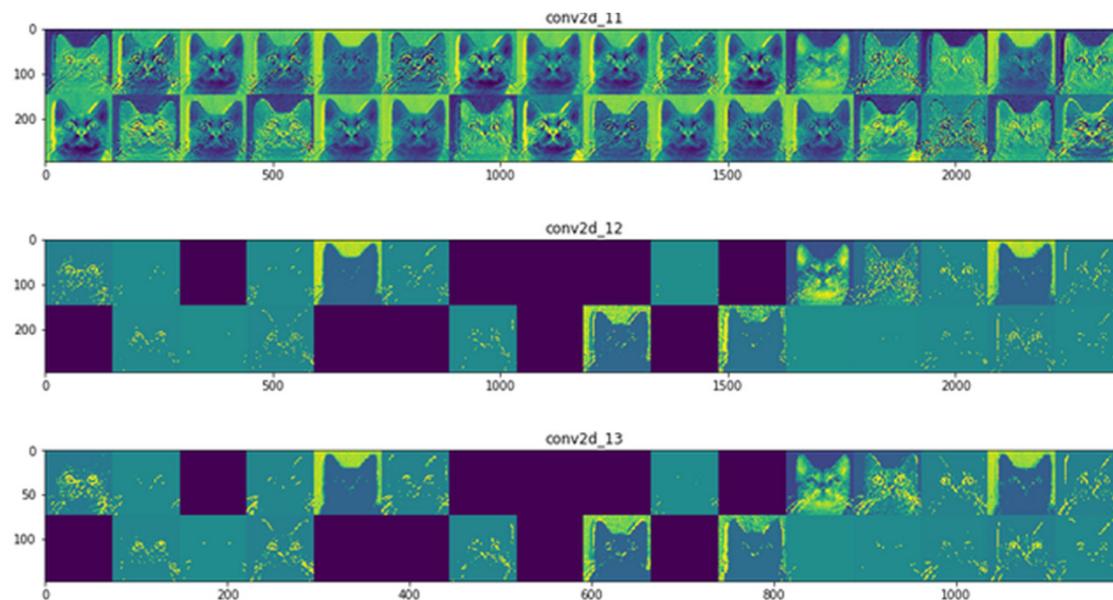
By Visualizing Filters we can **conceptualize what our CNN has 'learnt'**, this is because CNNs are essentially representations of a visual concepts.

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



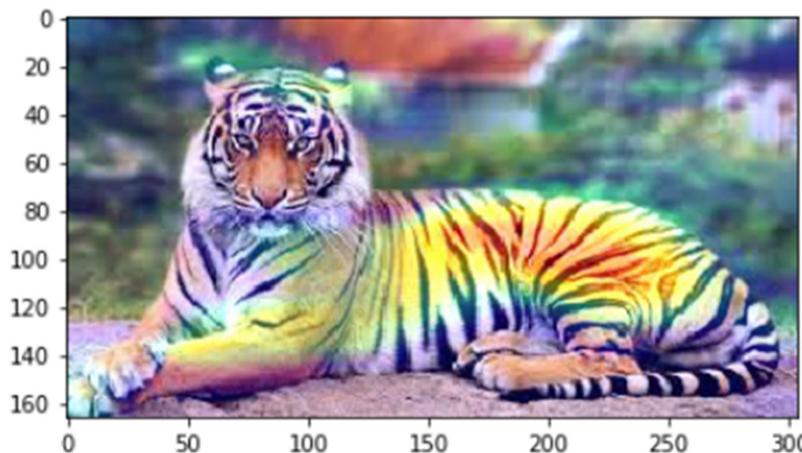
9.4

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

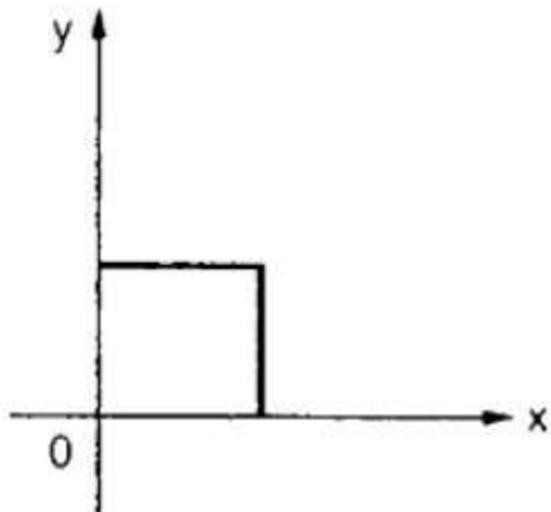


Types of Data Augmentation

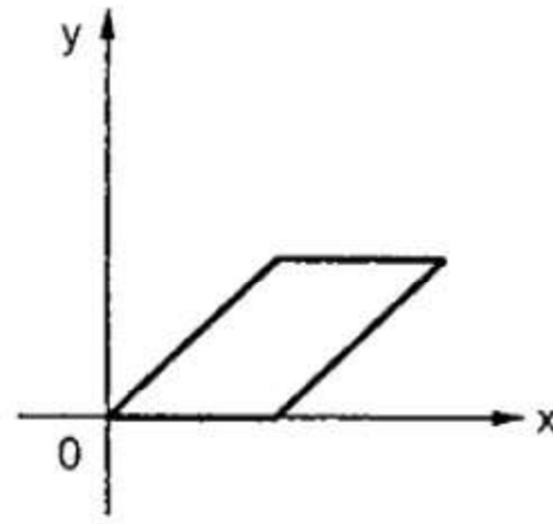
- At <https://keras.io/preprocessing/image/> we can see there are many types of augmentation available.
- In practice I've found the following add the most benefit.
 - Rotations
 - Horizontal and vertical shifts
 - Shearing
 - Zooming



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

11.0

The Confusion Matrix & Viewing Misclassifications



The Confusion Matrix & Viewing Misclassifications

- **11.1 Understanding the Confusion Matrix**
- **11.2 Finding and Viewing Misclassified Data**

11.1

Understanding the Confusion Matrix, Precision and Recall



Using the Confusion Matrix

- We use scikit-learn to generate our confusion matrix. Let's analyze our results on our MNIST dataset

```
[[ 977   0   0   0   0   0   1   1   1   0]
 [  0 1130   3   1   0   0   1   0   0   0]
 [  1   0 1029   0   1   0   0   1   0   0]
 [  0   0   4 1003   0   1   0   1   1   0]
 [  0   0   0   0  976   0   0   0   2   4]
 [  3   0   0   5   0  881   3   0   0   0]
 [  6   2   0   0   1   1  948   0   0   0]
 [  1   2  11   2   0   0   0 1010   1   1]
 [  4   0   3   0   0   0   0   2  963   2]
 [  1   2   0   1   5   3   0   2   5 990]]
```



Confusion Matrix Analysis – Multi Class

Predicted	0	1	2	3	4	5	6	7	8	9	True Values
[[977	0	0	0	0	0	1	1	1	0	0
0	1130	3	1	0	0	0	1	0	0	0	1
[1	0	1029	0	1	0	0	1	0	0	2
1	0	0	4	1003	0	1	0	1	1	0	3
[0	0	0	0	976	0	0	0	2	4	4
0	0	0	5	0	881	3	0	0	0	0	5
[3	0	0	0	1	1	948	0	0	0	6
6	2	0	0	1	0	0	0	0	0	0	7
[1	2	11	2	0	0	0	1010	1	1	7
1	4	0	3	0	0	0	0	2	963	2	8
[1	2	0	1	5	3	0	2	5	990	9
]											



Confusion Matrix Analysis

- Classifying 7s as 2s, 6s as 0s, 9s as 4s and 9s as 8s.

0	1	2	3	4	5	6	7	8	9	
977	0	0	0	0	0	1	1	1	0	0
0	1130	3	1	0	0	1	0	0	0	1
1	0	1029	0	1	0	0	1	0	0	2
0	0	4	1003	0	1	0	1	1	0	3
0	0	0	0	976	0	0	0	2	4	4
3	0	0	5	0	881	3	0	0	0	5
6	2	0	0	1	1	948	0	0	0	6
1	2	11	2	0	0	0	1010	1	1	7
4	0	3	0	0	0	0	2	963	2	8
1	2	0	1	5	3	0	2	5	990	9



Recall

- **Recall** – Actual true positives over how many times the classifier predicted that class
- Let's look at the number 7:
 - TP = 1010 and FN = 18
 - $1010 / 1028 = 98.24\%$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

0	1	2	3	4	5	6	7	8	9	
977	0	0	0	0	0	1	1	1	0	0
0	1130	3	1	0	0	1	0	0	0	1
1	0	1029	0	1	0	0	1	0	0	2
0	0	4	1003	0	1	0	1	1	0	3
0	0	0	0	976	0	0	0	2	4	4
3	0	0	5	0	881	3	0	0	0	5
6	2	0	0	1	1	948	0	0	0	6
1	2	11	2	0	0	0	1010	1	1	7
4	0	3	0	0	0	0	2	963	2	8
1	2	0	1	5	3	0	2	5	990	9



Precision

- **Precision** – Number of correct predictions over how many occurrences of that class were in the test dataset.
- Let's look at the number 7:
 - TP = 1010 and FP = 7
 - $1010 / 1017 = 99.31\%$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

0	1	2	3	4	5	6	7	8	9	
977	0	0	0	0	0	1	1	1	0	0
0	1130	3	1	0	0	1	0	0	0	1
1	0	1029	0	1	0	0	1	0	0	2
0	0	4	1003	0	1	0	1	1	0	3
0	0	0	0	976	0	0	0	2	4	4
3	0	0	5	0	881	3	0	0	0	5
6	2	0	0	1	1	948	0	0	0	6
1	2	11	2	0	0	0	1010	1	1	7
4	0	3	0	0	0	0	2	963	2	8
1	2	0	1	5	3	0	2	5	990	9



Classification Report

Using `scikit-learn` we can automatically generate a Classification Report that gives us Recall, Precision, F1 and Support.

	precision	recall	f1-score	support
0	0.98	1.00	0.99	980
1	0.99	1.00	1.00	1135
2	0.98	1.00	0.99	1032
3	0.99	0.99	0.99	1010
4	0.99	0.99	0.99	982
5	0.99	0.99	0.99	892
6	0.99	0.99	0.99	958
7	0.99	0.98	0.99	1028
8	0.99	0.99	0.99	974
9	0.99	0.98	0.99	1009



Classification Report Analysis

- **High recall with low precision.**
 - This tells us that most of the positive examples are correctly recognized (low False Negatives) but there are a lot of false positives i.e. other classes being predicted as our class in question.
- **Low recall with high precision.**
 - Our classifier is missing a lot of positive examples (high FN) but those we predict as positive are indeed positive (low False Positives)

11.2

Finding and Viewing our Misclassified Data



Finding and Viewing our Misclassifications

- This is often an underused technique in understanding your classifier's weaknesses.
- Viewing the misclassified test data can tell us a lot sometimes:
 - Is it confusing similar looking classes? Add more layers
 - Is it because of a complex pattern? Add more deep layers
 - Maybe our training data is mislabeled (it happens, humans aren't perfect)



Some Misclassified Data Inputs on the MNIST Dataset

Date Input	Predictions	True Value
6	0	6
3	2	8
9	9	8
1	9	4
6	5	6

12.0

Types of Optimizers, Learning Rates & Callbacks



Types of Optimizers, Learning Rates & Callbacks

- **12.1 Types Optimizers and Adaptive Learning Rate Methods**
- **12.2 Keras Callbacks and Checkpoint, Early Stopping and Adjust Learning Rates that Plateau**
- **12.3 Build a Fruit Classifier**

12.1

Types Optimizers and Adaptive Learning Rate Methods



Optimizers

- You may remember from our Neural Network explanation, optimizers were the algorithm we used to **minimize our Loss**. Examples were:
 - Gradient Descent
 - Stochastic Gradient Descent
 - Mini Batch Gradient Descent



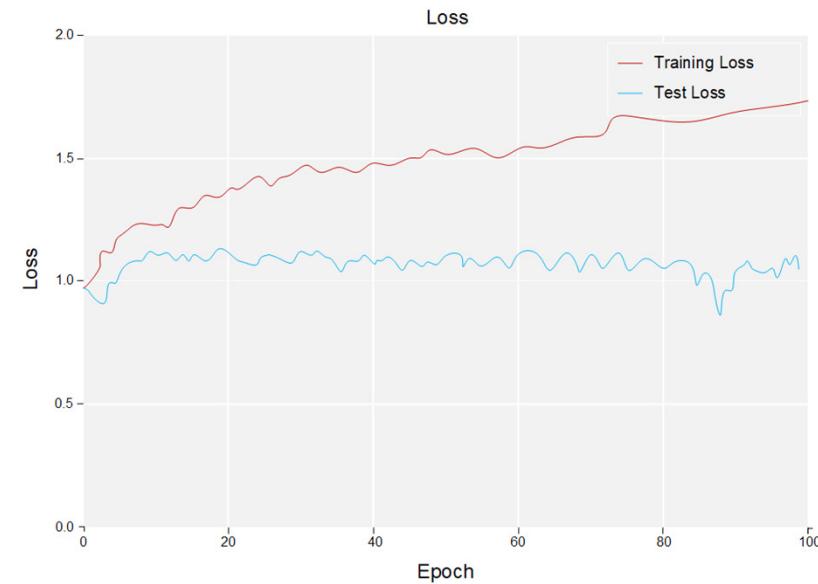
Keras's Built In Optimizers

- Fortunately, Keras has a few useful optimizers available for us:
 - Stochastic Gradient Descent
 - RMSprop
 - AdaGrad
 - AdaDelta
 - Adam
 - Adamax
 - Nadam



An aside about Learning Rates

- Constant learning rates are bad!
 - Imagine after 50 long epochs we're close to convergence, but there's a problem.
 - We set our learning rate too high!





So Many Choices! What's the difference?

- The main difference in these algorithms is how they **manipulate** the **learning rate** to allow for faster convergence and better validation accuracy.
 - Some require manual setting of parameters to adjust our **learning rate schedule**
 - Some use a heuristic approach to provide adaptive learning rates.

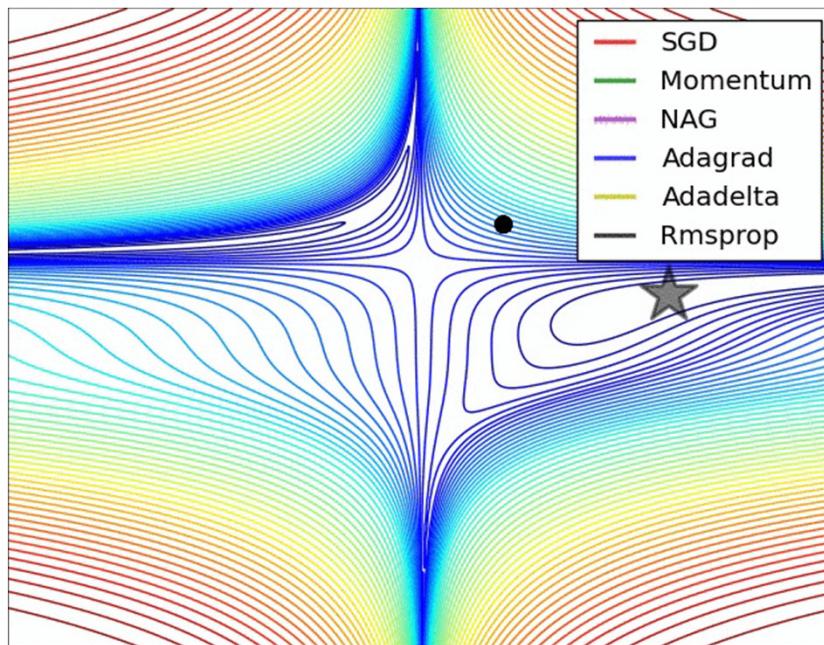


Stochastic Gradient Descent

- By default Keras uses a constant learning rate in the SGD optimizers. However, we can set **momentum**, **decay** as well as enabling **Nesterov Momentum**.
- **Momentum** – is a technique that accelerates **SGD** by pushing the **gradient steps** along the **relevant direction** but reducing the jump in oscillations away from the relevant directions.
- **Decay** – is setting that **decays the learning rate every batch update** (not epoch, so be aware of how you set your batch size). A good rule of thumb for setting decay is = $(\text{learning rate} / \text{epochs})$
- **Nesterov** – solves the problem of oscillating around our minima when momentum is high and unable to slow down. It first makes a big jump then a small correction after the gradient is calculated.



Nesterov Momentum Illustration



Source:

<http://cs231n.github.io/neural-networks-3/>

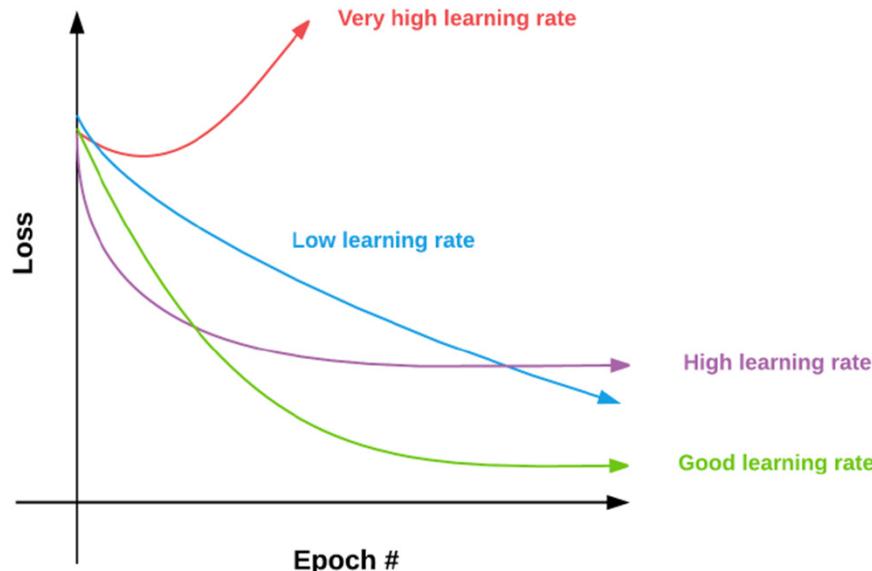


Adaptive Learning Rate Methods

- As we just saw we have to set our hyperparameters to control our learning rate schedule. This often ends up being a process of trial and error leading to time wastage. As such, adaptive learning rate methods have been developed to solve this problem. These are:
 - AdaGrad – performs large updates for more sparse parameters and smaller updates for less sparse parameters. It is thus well suited for sparse data. However, because learning rate is always decreasing monotonically, after many epochs learning slows down to a crawl.
 - AdaDelta – Solves monotonically decreasing gradient in AdaGrad
 - RMSprop – Similar to AdaDelta
 - Adam – Similar to AdaDelta, but also stores momentum for earning rates for each of the parameters separately.



What Good Learning Rates Look Like



Source - <http://cs231n.github.io/neural-networks-3/>

- As we just saw we have to set our hyperparameters to control our learning rate schedule. This often ends up being a process of trial and error leading to time wastage. As such, adaptive learning rate methods have been developed to solve

12.2

**Keras CallBacks and Checkpoint, Early
Stopping and Adjust Learnings Rates that
Plateau**



Checkpoint Models

- This is a simple but very useful way saving your best model before it starts overfitting (i.e. Loss on our test/validation data starts increases as our Epochs increase)
- Checkpointing allows us keep saving our weights/models after each epoch.
- Keras then allows us to keep saving the 'best' model by monitoring validation loss (we can monitor accuracy if you desire)



Creating Checkpoint Models

```
from keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint("/home/deeplearningcv/DeepLearningCV/Trained Models/",
                            monitor="val_loss",
                            mode="min",
                            save_best_only = True,
                            verbose=1)

callbacks = [checkpoint]

history = model.fit(x_train, y_train,
                     batch_size=64,
                     epochs=3,
                     verbose=2,
                     callbacks = callbacks,
                     validation_data=(x_test, y_test))
```

- We create a `callback` that monitors validation loss. Here we look at the lowest value and save only the best model.



How to we stop Training once our Model stops getting better?

- This is another useful feature that can save you valuable computing resources.
- What if after epochs 20 you stopped seeing any improvements (i.e. Validation Loss or Accuracy no longer was increasing)? Would you wait to run an extra 30 epochs and risk overfitting?
- Even after executing all 50, Check Pointing ensures we save the best model, but running a pointless 30 epochs more is wasteful use of time and resources!



Use Early Stopping

- Early stopping is another Keras Callback that allows us to stop training once the value being monitored (e.g. val_loss) has stopped getting better (decreasing).
- We can even use a “patience” parameter to wait X amount of epochs before stopping.

```
earlystop = EarlyStopping(monitor = 'val_loss', # value being monitored for improvement
                           min_delta = 0, #Abs value and is the min change required before we stop
                           patience = 3, #Number of epochs we wait before stopping
                           verbose = 1,
                           restore_best_weights = True) #keeps the best weights once stopped

# we put our call backs into a callback list
callbacks = [earlystop, checkpoint]
```



Reducing Learning Rate on Plateau

- Keras also comes with a Learning Rate adjustment callback.
- We can avoid having our loss oscillate around the global minimum by attempting to reduce the Learn Rate by a certain fact. If no improvement is seen in our monitored metric (val_loss typically), we wait a certain number of epochs (patience) then this callback reduces the learning rate by a factor

```
from keras.callbacks import ReduceLROnPlateau  
  
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.2, patience = 3, verbose = 1, min_delta = 0.0001)
```

12.3

Building a Fruit Classifier



Fruits-360

- Our Fruit Classifier will be made using the Fruits 360 dataset found on Kaggle
 - <https://www.kaggle.com/moltean/fruits>
- It consists of 81 types of fruits with approximately 45 images of fruits per class
- Images are all 100 x 100 and in color



Fruits-360 Sample Images



13.0

**Build LeNet, AlexNet in Keras & Understand
Batch Normalization**



Build LeNet, AlexNet in Keras & Understand Batch Normalization

- **13.1 Build LeNet and test on MNIST**
- **13.2 Build AlexNet and test on CIFAR10**
- **13.3 Batch Normalization**
- **13.4 Build a Clothing & Apparel Classifier (Fashion MNIST)**

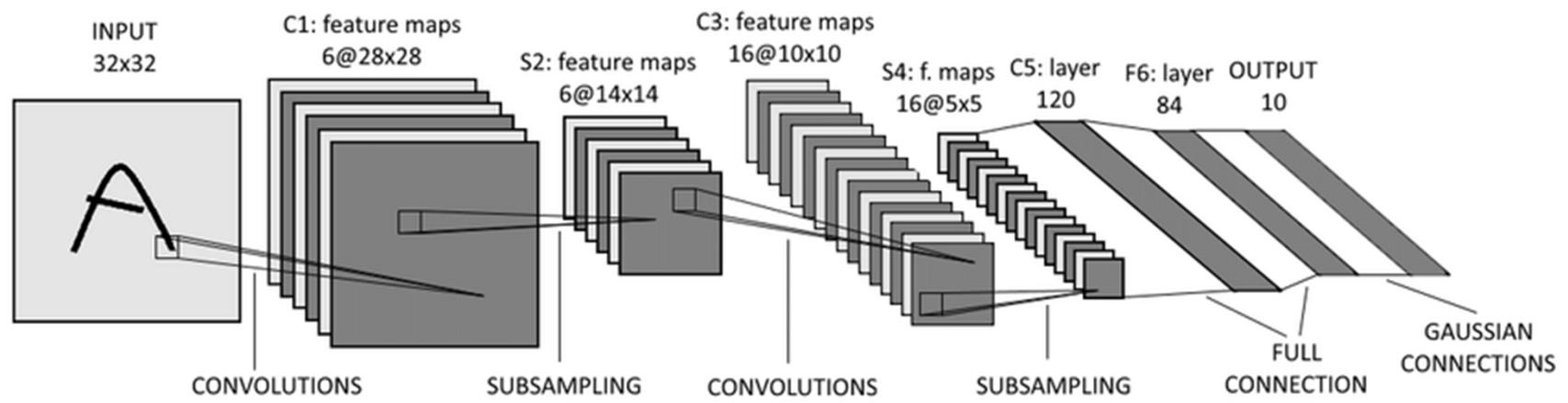
13.1

Building LeNet in Keras

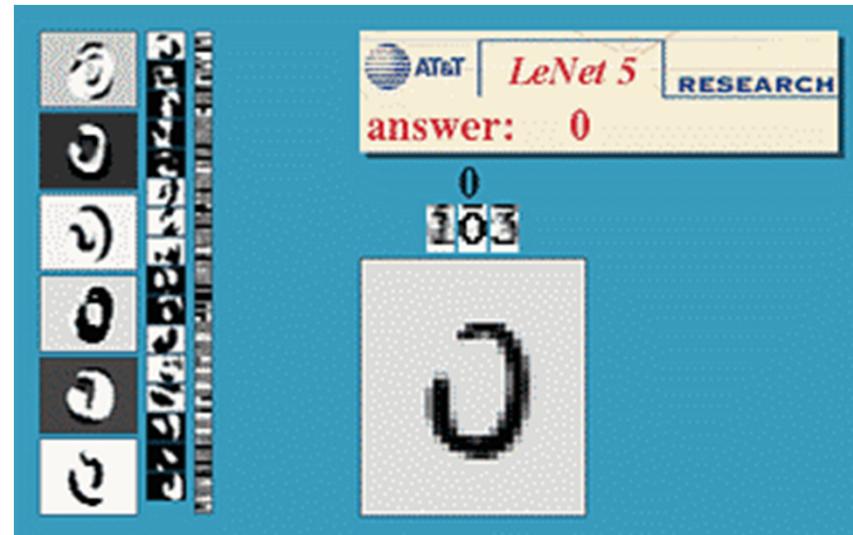
LeNet

- Introduced way back in 1998 by LeCun, LeNet was a very effective CNN developed for handwriting recognition.
- <http://yann.lecun.com/exdb/lenet/>

LeNet Illustrated



LeNet Animation



LeNet Performance on MNIST

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 310s 5ms/step - loss: 0.1855 - acc: 0.9435 - val_loss: 0.0853 - val_acc: 0.9720
Epoch 2/10
60000/60000 [=====] - 306s 5ms/step - loss: 0.0449 - acc: 0.9862 - val_loss: 0.0371 - val_acc: 0.9877
Epoch 3/10
60000/60000 [=====] - 384s 6ms/step - loss: 0.0298 - acc: 0.9908 - val_loss: 0.0268 - val_acc: 0.9905
Epoch 4/10
60000/60000 [=====] - 290s 5ms/step - loss: 0.0207 - acc: 0.9936 - val_loss: 0.0307 - val_acc: 0.9896
Epoch 5/10
60000/60000 [=====] - 324s 5ms/step - loss: 0.0153 - acc: 0.9952 - val_loss: 0.0246 - val_acc: 0.9920
Epoch 6/10
60000/60000 [=====] - 317s 5ms/step - loss: 0.0110 - acc: 0.9967 - val_loss: 0.0236 - val_acc: 0.9920
Epoch 7/10
60000/60000 [=====] - 330s 5ms/step - loss: 0.0088 - acc: 0.9972 - val_loss: 0.0264 - val_acc: 0.9916
Epoch 8/10
60000/60000 [=====] - 316s 5ms/step - loss: 0.0069 - acc: 0.9978 - val_loss: 0.0223 - val_acc: 0.9935
Epoch 9/10
60000/60000 [=====] - 282s 5ms/step - loss: 0.0052 - acc: 0.9983 - val_loss: 0.0267 - val_acc: 0.9920
Epoch 10/10
60000/60000 [=====] - 278s 5ms/step - loss: 0.0037 - acc: 0.9989 - val_loss: 0.0237 - val_acc: 0.9931
10000/10000 [=====] - 28s 3ms/step
Test loss: 0.02373578137872546
Test accuracy: 0.9931
```

Our Previous CNN's Performance on MNIST

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 367s 6ms/step - loss: 0.2656 - acc: 0.9180 - val_loss: 0.0651 - val_acc: 0.9781
Epoch 2/10
60000/60000 [=====] - 450s 8ms/step - loss: 0.0918 - acc: 0.9727 - val_loss: 0.0403 - val_acc: 0.9863
Epoch 3/10
60000/60000 [=====] - 475s 8ms/step - loss: 0.0683 - acc: 0.9799 - val_loss: 0.0328 - val_acc: 0.9876
Epoch 4/10
60000/60000 [=====] - 441s 7ms/step - loss: 0.0560 - acc: 0.9838 - val_loss: 0.0310 - val_acc: 0.9888
Epoch 5/10
60000/60000 [=====] - 448s 7ms/step - loss: 0.0471 - acc: 0.9855 - val_loss: 0.0301 - val_acc: 0.9899
Epoch 6/10
60000/60000 [=====] - 458s 8ms/step - loss: 0.0413 - acc: 0.9872 - val_loss: 0.0302 - val_acc: 0.9905
Epoch 7/10
60000/60000 [=====] - 390s 6ms/step - loss: 0.0373 - acc: 0.9887 - val_loss: 0.0274 - val_acc: 0.9911
Epoch 8/10
60000/60000 [=====] - 329s 5ms/step - loss: 0.0343 - acc: 0.9895 - val_loss: 0.0286 - val_acc: 0.9903
Epoch 9/10
60000/60000 [=====] - 199s 3ms/step - loss: 0.0313 - acc: 0.9904 - val_loss: 0.0274 - val_acc: 0.9904
Epoch 10/10
60000/60000 [=====] - 200s 3ms/step - loss: 0.0299 - acc: 0.9909 - val_loss: 0.0272 - val_acc: 0.9915
Test loss: 0.027194885472155875
Test accuracy: 0.9915
```

13.2

Building AlexNet in Keras

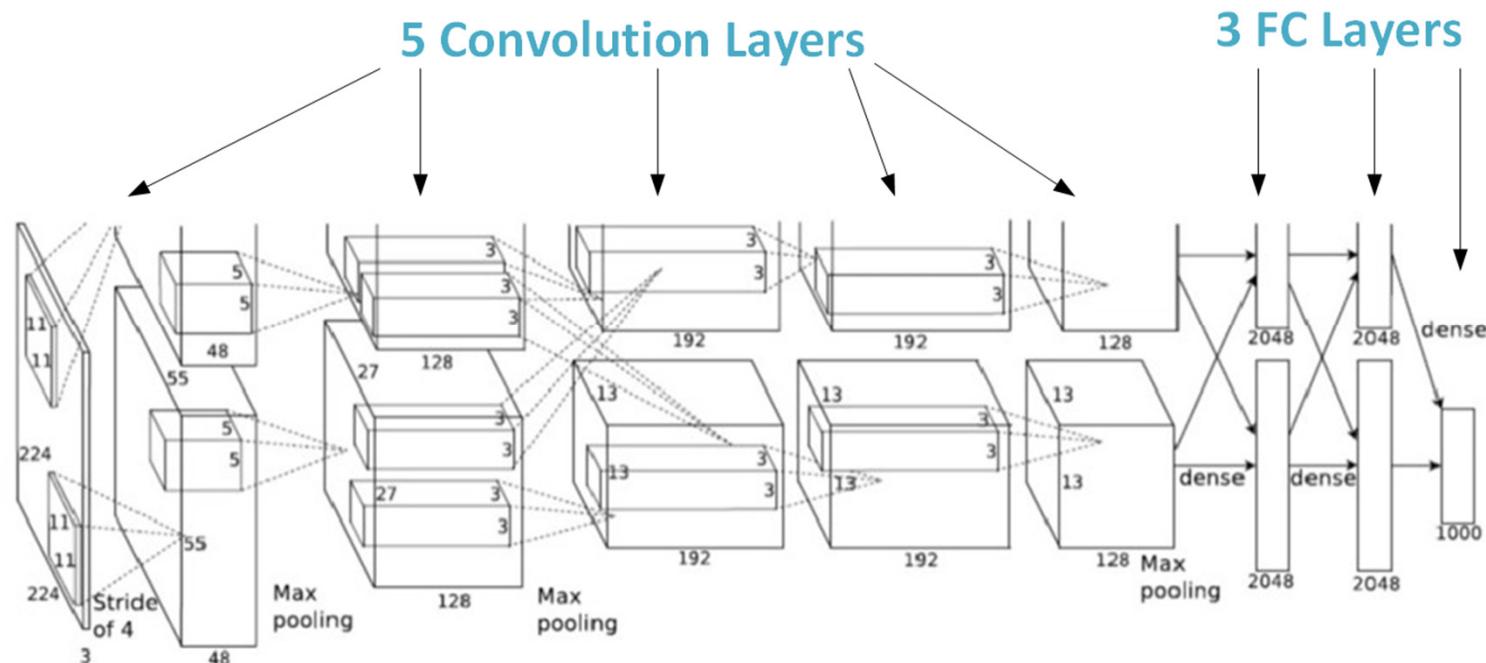


AlexNet

- AlexNet was developed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton from the University of Toronto in 2012 and was the ILSVRC winner in 2012.
- AlexNet contains 8 layers with the first five being Convolutional Layers and the last 3 being FC layers.
- It has over 60 Million parameters and was trained on two GPUs for over a week!

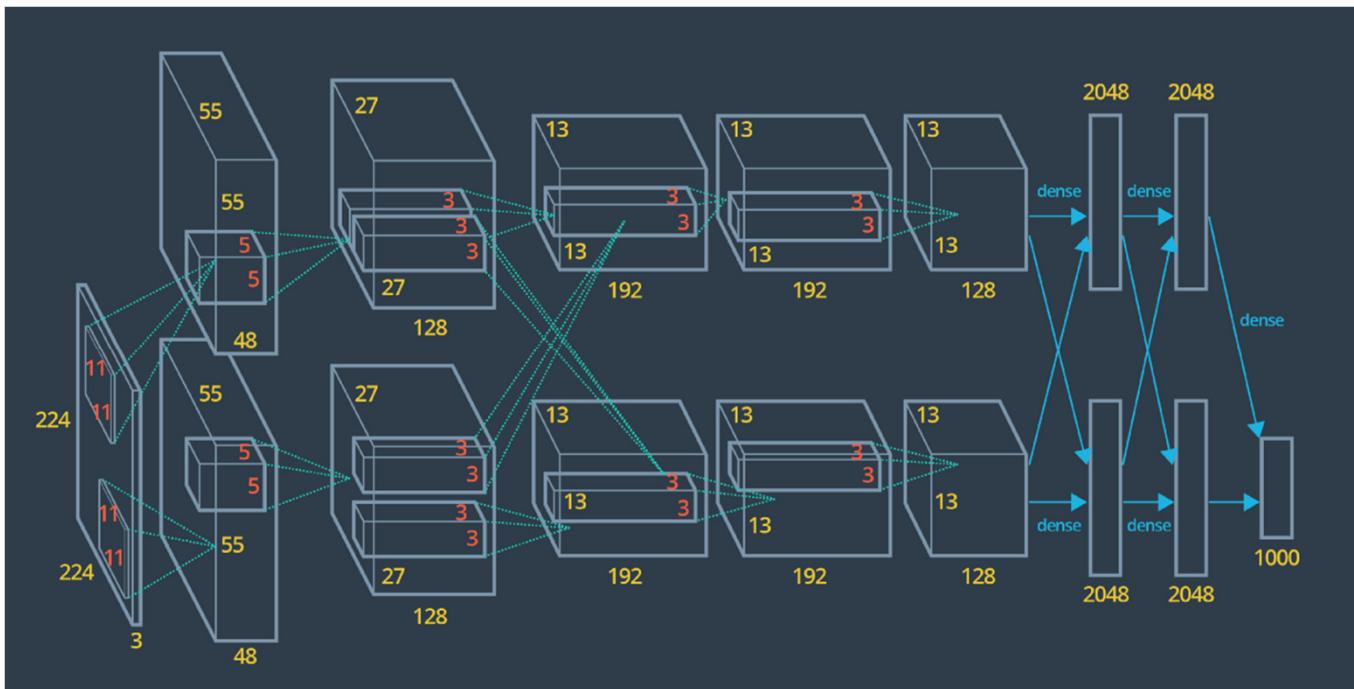


AlexNet Illustrated





AlexNet Illustrated



360

13.3

Batch Normalization



Batch Normalization

- Batch Norm was proposed by Ioffe and Szegedy in 2015 in their paper titled "*Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*"
- Previously, we applied a generic form of Normalization scaling our image data from values between 0-255 to values between 0 to 1 (this was done by dividing it by 255).
- We do this to reduce the influence of larger data points

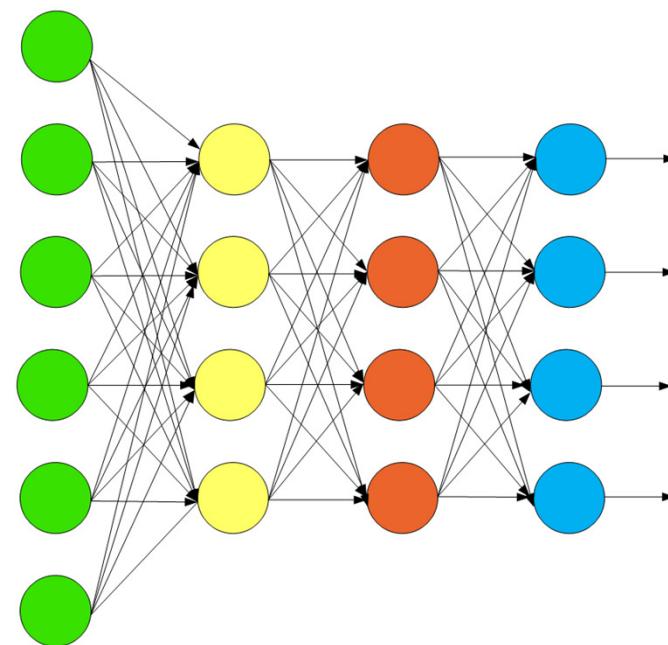


Batch Normalization

- Batch Norm however is used to normalize the activations of a input tensor before passing it into the next layer in the network.

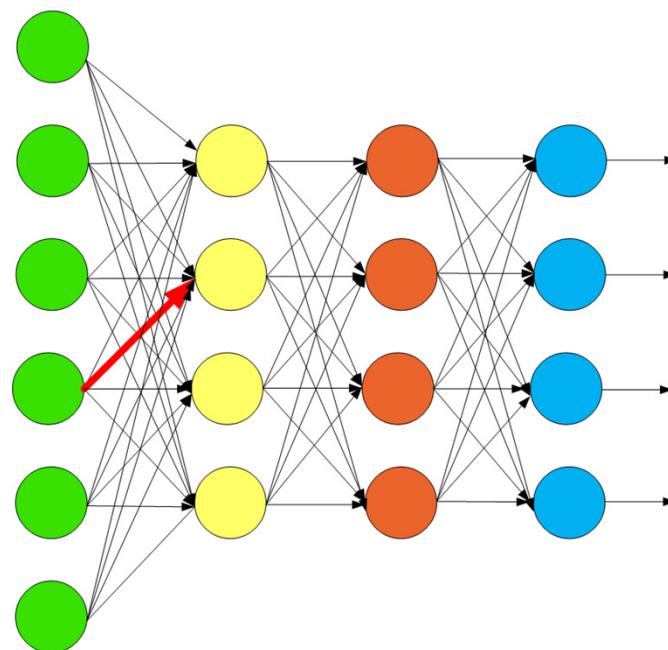


Before we begin training our NN, we randomize our weights



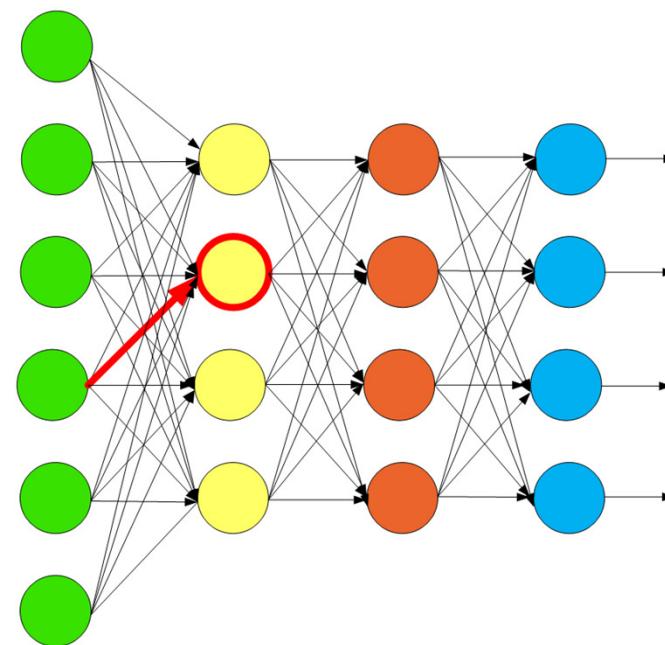


But what if one our weights becomes extremely large!



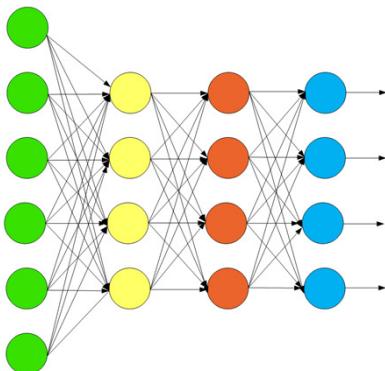


It's corresponding Neuron's output will be very large! Cascading throughout our NN causing instability.





Batch Normalization is applied to a layer we select in our NN



- Batch Norm normalizes the output from the activation functions of a selected layer.
- It then normalizes the output by multiplying it by a parameter and then adding another parameter to this result.
- The result is that all the activations leaving a batch normalization layer will have approximately zero mean.
- The weights now don't become imbalanced with extreme values since normalization is now included in the gradient process



Batch Normalization Benefits

```
model.add(Conv2D(96, (11, 11), input_shape=x_train.shape[1:],  
    padding='same', kernel_regularizer=l2(l2_reg)))  
model.add(BatchNormalization())  
model.add(Activation('relu'))  
model.add(MaxPooling2D(pool_size=(2, 2)))
```

- Batch Normalization reduces the number of Epochs it takes our NN to converge
- It aids in regularization (reducing overfitting)
- It allows us to improve stability of our training, thus allowing us to use large learning rates
- Batch Normalization slows down training

13.4

**Clothing/Apparel Recognition using
Fashion MNIST**



Fashion MNIST

- Fashion MNIST is a dataset found on Kaggle that serves as a direct replacement for MNIST. However, instead of handwritten digits, the classes are 10 items of clothing/apparel.

T-shirt/top	Sandal
Trouser	Shirt
Pullover	Sneaker
Dress	Bag
Coat	Ankle boot

- Just like MNIST, there are 60,000 examples and each image is 28 x 28 grayscale.



Fashion MNIST Samples





Fashion MNIST Approach

- We're going to use the famous LeNet (made popular by its performance on MNIST) on our Fashion MNIST.
- This isn't the world leader in MNIST, nor Fashion MNIST, but it is able to reach very high accuracy while being trained on CPU (like most of us doing this course).

14.0

**ImageNet & using pre-trained Models in
Keras (VG16, VG19, InceptionV3, ResNet50 &
MobileNet)**



ImageNet & using pre-trained Models in Keras (VG16, VG19, InceptionV3, ResNet50 & MobileNet)

- 14.1 ImageNet - Experimenting with pre-trained Models in Keras (VGG16, ResNet50, MobileNet, InceptionV3)
- 14.2 Understanding VGG16 and VGG19
- 14.3 Understanding ResNet50
- 14.4 Understanding InceptionV3

14.1

**Experimenting with pre-trained Models in
Keras (VG16, VG19, InceptionV3, ResNet50 &
MobileNet)**



Famous CNNs

- As a computer vision enthusiast, you've surely heard of the famous CNN's such as:
 - LeNet
 - AlexNet
 - VGGNet (VGG16 & VGG19)
 - ResNet50
 - InceptionV3
 - Xception



What made them famous was ImageNet

- Or rather their performance on the **ImageNet** dataset
- **ImageNet** dataset refers to the ImageNet Large Scale Visual Recognition Challenge (**ILSVRC**). –
 - <http://www.image-net.org/challenges/LSVRC/>
- It is dataset comprising of over **1.2 million training images** separated into **1000** object classes.
- It is **THE** benchmark/standard used to test modern Deep Learning CNN's.



ImageNet

IMAGENET

14,197,122 images, 21841 synsets indexed

[SEARCH](#)

[Home](#) [About](#) [Explore](#) [Download](#)

Not logged in. [Login](#) | [Signup](#)

Bird

Warm-blooded egg-laying vertebrates characterized by feathers and forelimbs modified as wings

Numbers in brackets: (the number of synsets in the subtree).

- ImageNet 2011 Fall Release (32326)
 - plant, flora, plant life (4486)
 - geological formation, formation (17)
 - natural object (1112)
 - sport, athletics (176)
 - artifact, artefact (10504)
 - fungus (308)
 - person, individual, someone, some animal, animate being, beast, brute
 - invertebrate (766)
 - homeotherm, homoiotherm, hor
 - work animal (4)
 - darter (0)
 - survivor (0)
 - range animal (0)
 - creepy-crawly (0)
 - domestic animal, domesticated
 - molter, moulter (0)
 - varmint, varment (0)
 - mutant (0)
 - critter (0)
 - game (47)
 - young, offspring (45)
 - poikilotherm, ectotherm (0)
 - herbivore (0)
 - peeper (0)
 - pest (1)
 - female (4)
 - insectivore (0)
 - pet (0)

[Treemap Visualization](#) [Images of the Synset](#) [Downloads](#)

ImageNet 2011 Fall Release > Vertebrate, craniate > Bird

Aquatic	Bird	Gallinaceous
Archaeornithine	Nonpasserine	Night
Twitterer	Trogon	Carinate
Passerine	Dickeybird	Caprimulgiformes
Archaeopteryx	Apodiform	Coraciiformes
Hen	Cuculiformes	Piciform
Nester	Cock	Ratite
		Parrot

378



ImageNet Winner Summary

Year	CNN	Top-5 Error Rate	# of Parameters
1998	LeNet	*did not compete	60,000
2012	AlexNet	15.3%	60 M
2014	GoogLeNet or Inception V1	6.67%	4 M
2014	VGGNet	7.3%	138 M
2015	ResNet	3.6%	25.6 M



We can load and experiment with these models in Keras

To view all the pre-trained models shipped with Keras

- Visit <https://keras.io/applications/>
- Loading and executing them is straight forward
- We can even tweak these models for your usage (next chapter!)

14.2

Understanding VGG16 and VGG19



VGGNet

- The VGGNet model was developed by Simonyan and Zisserman in 2014 paper titled “*Very Deep Convolutional Networks for Large Scale Image Recognition.*”
- *VGGNET* was famous for only 3x3 Convolutional Layers throughout
- It was pretty deep by 2014 standards of 16 and 19 layers (hence the names VGG16 and VGG19).
- Due to its depth, VGGNet’s first few layers were pre-trained before moving onto its deeper layers. As you may imagine, this makes training VGGNet extremely slow.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Source –

<https://arxiv.org/abs/1409.1556>

14.3

Understanding ResNet50



ResNet50

- Introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, ResNet50 was the CNN that won the ILSRVC competition in 2015.
- ResNet is short for Residual Network, which uses the concept of Residual Learning.

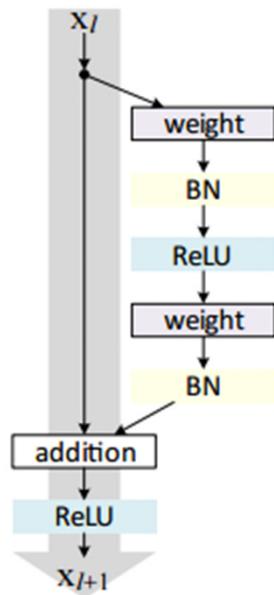


The Beauty of ResNet50

- In 2015, the trend in Deep Learning CNN's was to keep going deeper. However, this presented a problem as training became more difficult due to accuracy becoming saturated and then degrading rapidly.
- ResNet proposed a shallower architecture with a *deep residual learning* framework
- ResNet instead of learning low/mid/high level features learns Residuals by using shortcut connections (directly connecting input of nth layer to an (n+x) layer).
- This results in far easier training and resolves the degrading accuracy problem



The Residual Module



Source –
<https://arxiv.org/abs/1512.03385>

14.4

Understanding InceptionV3

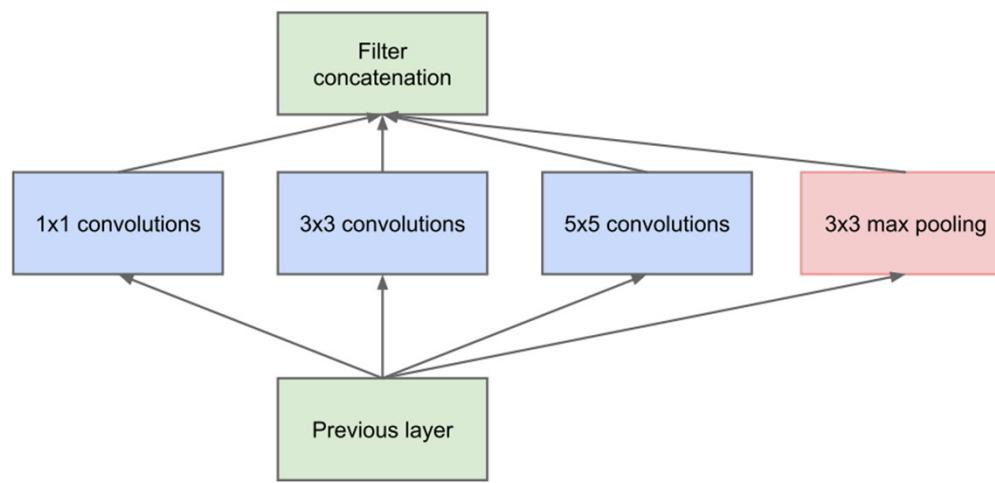


The History of Inception

- The Inception architecture was introduced in Szegedy et al. (2014) paper titled “*Going deeper with convolutions*”.
- InceptionV1 was the winner of ILSVRC in 2014 that was implemented by GoogLeNet.
- Further improvements have been made over the years, each iteration named Inception V2, V3 etc.



The Beauty of Inception



- The inception module is a concatenation of multiple filter sizes, instead of using a single sized filter for each convolution layer
- Total number of parameters is much less than VGG16/VGG19

Fully Architecture of Inception V1



Source –
<https://arxiv.org/abs/1409.4842>





Inception V3 & V4

- The evolution of the Inception model led to several improvements and tweaks such as more efficient representation of convolution filters (e.g. replacing a large 7×7 filter with a pair of 1×7 and 7×1 Conv layers)
- Other tweaks include factorization of convolutions and improved normalization.
- Adopting these tweaks led to Inception V3
- Inception V4 was introduced in 2016 which was a better performing more streamlined version of Inception V3

15.0

Transfer Learning and Fine Tuning



Transfer Learning and Fine Tuning

- **15.1 What is Transfer Learning and Fine Tuning**
- **15.2 Build a Flower Classifier with VGG16**
- **15.3 Build a Monkey Breed Identified with MobileNet using Transfer Learning**

15.1

What is Transfer Learning and Fine Tuning



Training Complicated and Deep CNNs is Slow!

- AlexNet and VGG (in particular) are deep, parameter laden networks. **VGG has over 138M** parameters! ResNet50 has 50 hidden layers
- While these networks, attain relatively excellent performance on ImageNet, training them on a **CPU is an exercise in futility**.
- These CNNs are often trained for a **couple weeks or more** using arrays of GPUs

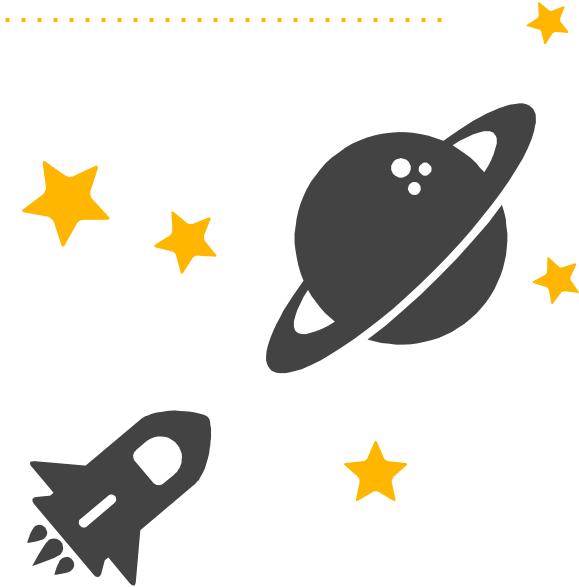


What if there was a way to re-use these trained models for our own classifiers?

- We've seen Keras ships with the ImageNet trained weights for ResNet50, VGG16, InceptionV3 and others,
- These weights are already tuned to detect thousands of low/mid/high level features.
- What if we could use these pre-trained networks to train our very own classifiers!
- Well we can!

Transfer Learning & Fine Tuning

Introducing the concept of Transfer Learning
and Fine Tuning!



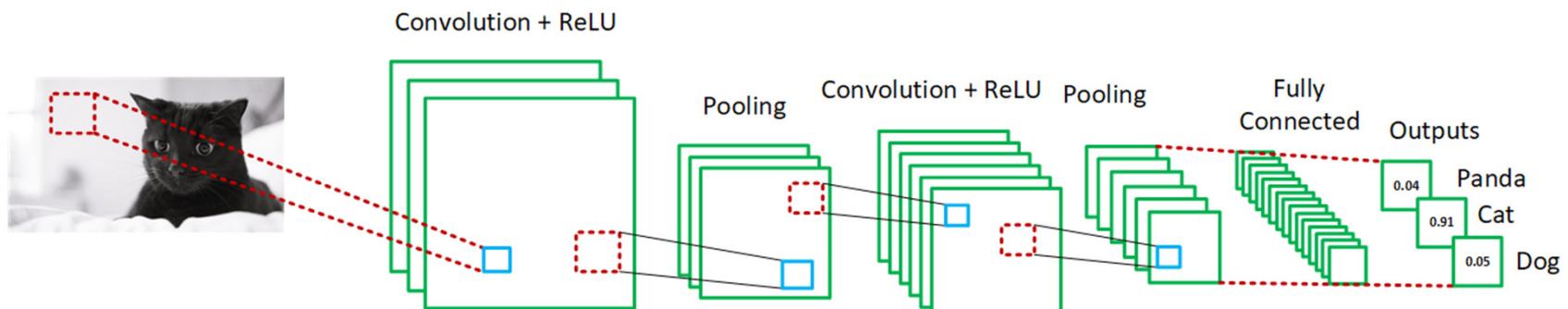


Fine Tuning

- The concept of Fine Tuning is often and justifiably confused with Transfer Learning. However it is merely a type of Transfer Learning.
- **Fine Tuning** is where we take a pre-trained Deep CNN (ResNet50, Inception or VGG) and use the already trained (typically on ImageNet) Convolutional Layers to aid our new image classification task. Typically in fine tuning we are taking an already trained CNN and training it on our new dataset
- We then either Freeze these lower layers, and train the top or FC layers only. Sometimes we do go back and train the lower layers to improve our performance.



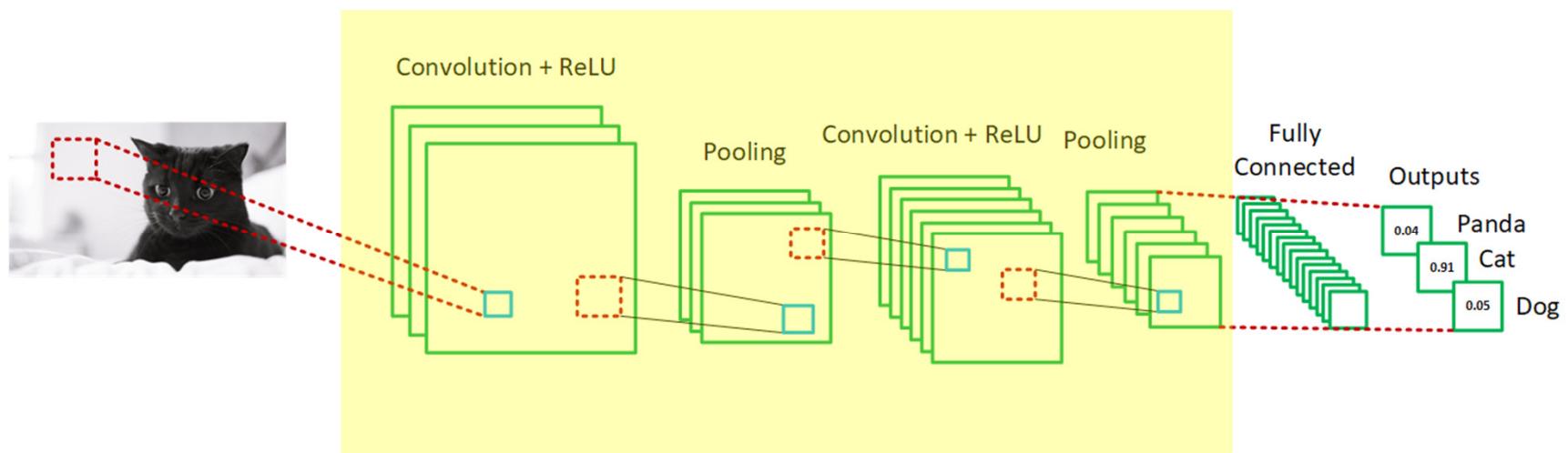
Imagine this is a Pre-trained Deep CNN



400

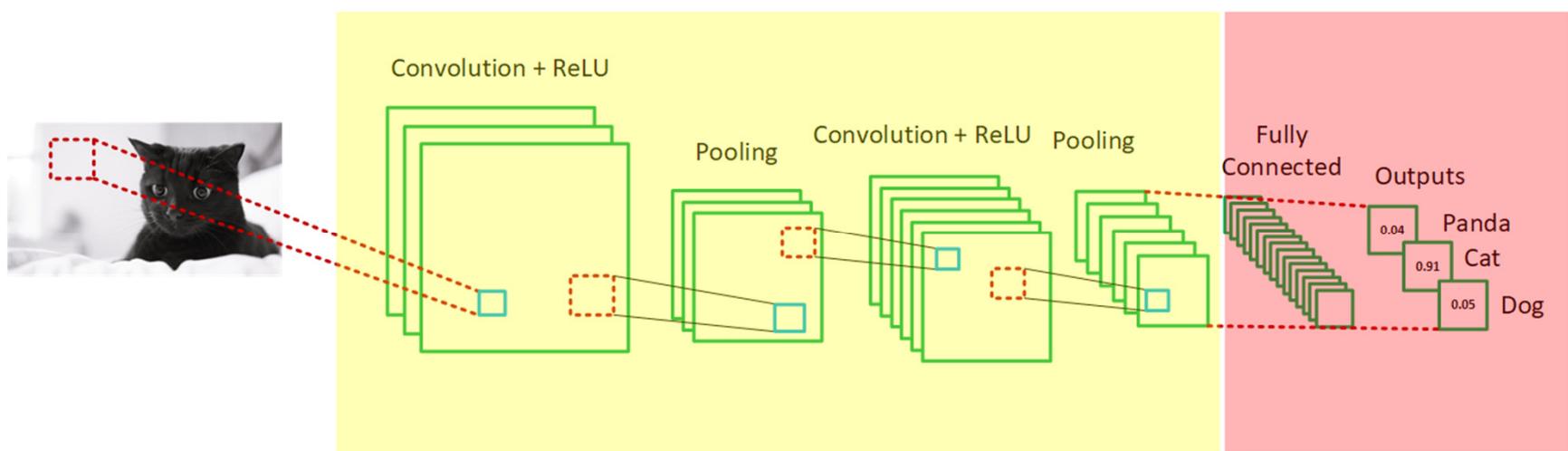


Let's Freeze all the CONV weights





We only train the Layers in Red





Fine Tuning

- In most deep CNN's the first few CONV layers learn low level features (example color blobs) and as we progress through the network, it learns more mid/high level features or patterns.
- In Fine Tuning we keep or **Freeze** these trained low level features, and only train our high level features needed for our new image classification problem.



Fine Tuning Steps



- We **freeze** the already pre-trained lower CONV layers as they well trained to capture universal type features (curves, edges, blobs)
- We replace the top FC layer with our own, mainly because we maybe training to fit a different number of classes e.g. going from 1000 to 2
- Using a small learning rate helps us to avoid distorting the newly initialized weights to quickly



Transfer Learning

- As we've done in Fine Tuning, we have taken a pre-trained network and trained it (or segments/layers of it) on some new data for a new image classification task. (it doesn't have to be new data or a new goal, Fine Tuning simply implies we further train an already trained network).
- Transfer Learning**, is almost the same thing and is often used interchangeably with Fine Tuning.
- Transfer Learning though implies we are taking the 'knowledge' learned from a pre-trained network and applying to a similar task and therefore not re-training the network..



Transfer Learning A Definition

"Transfer learning and domain adaptation refer to the situation where what has been learned in one setting ... is exploited to improve generalization in another setting"

- Page 526, [Deep Learning](#), 2016.



Our Practical Examples

- Use MobileNet to create a Monkey Breed Classifier
- Use VGG16 as stated above to make a Flower Classifier

15.2

Building a Monkey Breed Classifier



10 Monkey Species Dataset

- Our dataset originates on Kaggle and consists of 10 species of monkeys.
 - <https://www.kaggle.com/slothkong/10-monkey-species/home>

mantled_howler	white_headed_capuchin
patas_monkey	silvery_marmoset
bald_uakari	common_squirrel_monkey
japanese_macaque	black_headed_night_monkey
pygmy_marmoset	nilgiri_langur

Sample Images



- Images were of varying sizes and quality
- Each class has between 131-152 images





Our Approach

- Use a pre-trained MobileNet with all it's weights except the top layer frozen.
- We only train the Fully Connected (FC) or Dense Layers with a final output of 10
- We then try the more complex and highly parametrized InceptionV3 network, with the same methodology (i.e. freezing all weights except the top layer)

15.3

Building a Flower Classifier



Flowers-17

- Flowers-17 was created by the University of Oxford's Visual Geometry Group at the Department of Engineering.
- It consists of 17 categories of flowers commonly found in the UK.
- There were 80 images of each flower in this dataset

17 Category Flower Dataset

[Maria-Elena Nilsback](#) and [Andrew Zisserman](#)



Overview

We have created a 17 category flower dataset with 80 images for each class. The flowers chosen are some common flowers in the UK. The images have large scale, pose and light variations and there are also classes with large variations of images within the class and close similarity to other classes. The categories can be seen in the figure below. We randomly split the dataset into 3 different training, validation and test sets. A subset of the images have been groundtruth labelled for segmentation.

Downloads

The data needed for evaluation are:

1. [Dataset images](#)
2. [The data splits](#)
3. [Segmentation groundtruth data](#)
4. [\$\&Chi^2\$ distances CVPR 2006](#) - distance matrices for features and segmentation used in CVPR 2006 publication.
5. [\$\&Chi^2\$ distances ICVGIP 2008](#) - distance matrices for features and segmentation used in ICVGIP 2008 publication.

The [README](#) file explains everything.

Class Examples



<http://www.robots.ox.ac.uk/~vgg/data/flowers/17/>



Our Approach

- Use a pre-trained VGG16 with all it's weights except the top layer frozen
- We only train the Fully Connected (FC) or Dense layers, with a final output of 17



Adjusting for Image Size

- Previously, we noticed VGG16, MobileNet and InceptionV3 all used an input image size of 224×224 .
- Can you use another image size though?
- What if we wanted to improve CIFAR10 model, but its images are 32×32 .
- It would be a waste to upscale and resize these images, producing a network with many unnecessary parameters and redundant inputs.



We can use different input shapes as long as the Stride fits

"However, some changes are straight-forward: Due to parameter sharing, you can easily run a pretrained network on images of different spatial size. This is clearly evident in the case of Conv/Pool layers because their forward function is independent of the input volume spatial size (as long as the strides "fit")."

Source – Transfer Learning at Stanford's CS231n
<http://cs231n.github.io/transfer-learning/>

- Remember the stride, controls how big a jump our filters take when moving across our image.
- Once our strides 'fits' as in can move as expected without any issues (which can occur if the image is too small) fine tuning will be fine

16.0

Design Your Own CNN - LittleVGG



Design Your Own CNN - LittleVGG

- **16.1 Introducing LittleVGG**
- **16.2 Simpsons Character Recognition using LittleVGG**

16.1

Introducing LittleVGG



LittleVGG

- LittleVGG is my own downsized version of VGG16
- VGG inspiring networks all use:
 - A series of 3x3 Convolution Layers
 - Convolution to ReLU Blocks where the number of these filters increases the deeper we go.
- Let's take a look at a visualization of our LittleVGG

LittleVGG Configuration Compared to VGG Family

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

LittleVGG or VGG9

9 weight layers

Input (224 x 224 RGB image)

- 1. Conv3-64
- 2. Conv3-64

Maxpool

- 3. Conv3-128
- 4. Conv3-128

Maxpool

- 5. Conv3-256
- 6. Conv3-256

Maxpool

- 7. FC-256

Maxpool

- 8. FC-256

Maxpool

- 9. FC-20

Soft-max

2,270,804 Parameters



16.2

Simpsons Character Recognition using LittleVGG



The Simpsons Dataset

- The awesome Simpsons dataset was uploaded to Kaggle
 - <https://www.kaggle.com/alexattia/the-simpsons-characters-dataset/home>
- It consists of 20 classes or characters with 200-400 RGB pictures each
- Images were of various sizes, but all had one character being the only character in the image or at least he/she was the main focus



Sample Images of Simpson Dataset



Our Approach



- Our first test run with LittleVGG and we compare it's results with our previous CNN.

17.0

Advanced Activation Functions and Initializations



Advanced Activation Functions and Initializations

- 17.1 Dying ReLU Problem and Introduction to Leaky ReLU, ELU and PReLU
- 17.2 Advanced Initializations

17.1

Advanced Activations – Leaky ReLU, PReLU and ELU



Advanced Activations

- Activation Functions introduce the non-linearity that is needed for Neural Networks achieving their incredible performance.
- Previously we learnt about **ReLU** (Rectified Linear Units) and stated that they are the default activation function used in CNNs.
- ReLU isn't always perfect...



The Dying ReLU Problem

- ReLU units can often "die" during training. This can happen when a large gradient flows through that neuron, which causes the weights to update in such a way that the **unit never activates** with any input data ever again.
- When this happens, the output of this ReLU unit is always **zero**, effectively wasting that unit and not playing any role in our model.
- Sometimes as much as 40% of network can be dead i.e. have ReLU units that output zero for every training data input.



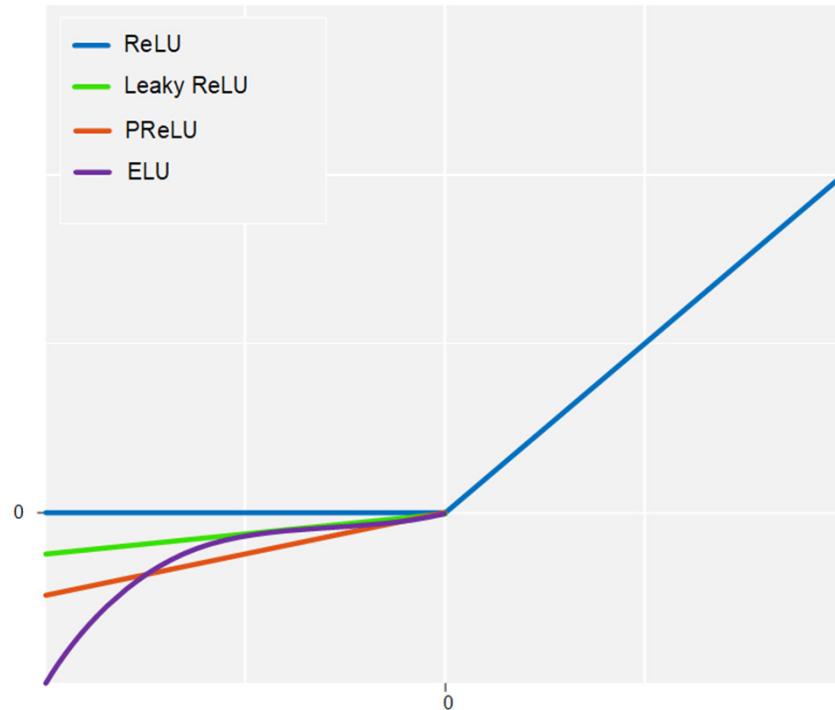
Fixing the Dying ReLU problem

- Leaky ReLU (small negative slope)
- Parametric or PReLU
- Exponential or ELU (sometimes called Scaled ELU or SELU)

$$y = a(e^x - 1)$$

All of the above activation functions fix the dying ReLU problem by having no zero slope parts.

ELU tends to be a good mix of the good parts of ReLU and Leaky ReLU, however it can saturate on large negative values.





Other Exotic Activation Functions

- Concatenated ReLU or CReLU - <https://arxiv.org/abs/1603.05201>
 - CReLU combines or concatenates the outputs of two ReLU functions, one positive ($x, 0$) and one negative ($0, x$), thus doubling the output value
- ReLU-6 - <http://www.cs.utoronto.ca/~kriz/conv-cifar10-aug2010.pdf>
This is simply a ReLU function that is capped at 6. There was no special reason for selecting 6, other than it worked best on the CIFAR10 dataset.
- There are many others as well too (Maxout, Offset ReLU etc.)



When to use something other than ReLU?

- So we've just seen many ReLU variations, which do you choose and when?
- There's no hard and fast rule in Deep Learning. Generally my advice is to stick with ReLU first. Adjust learning rates to get the best accuracy you can with your CNN at first. Once that's done, then you experiment with different ReLU functions.
- You can go from Leaky ReLU to ELU, however in most cases you can skip Leaky ReLU and go to ELU where you should see some marginal improvements in your model.

17.2

Advanced Initialization



Initializations

- Something we've glossed over for most of this course is how are the initial weights randomized when we start learning.
- We know they're random, but are they truly random (i.e. pulled from a uniform distribution)?
- Keras has quite a few random initializers with the default one indeed being pulled from a random uniform distribution.

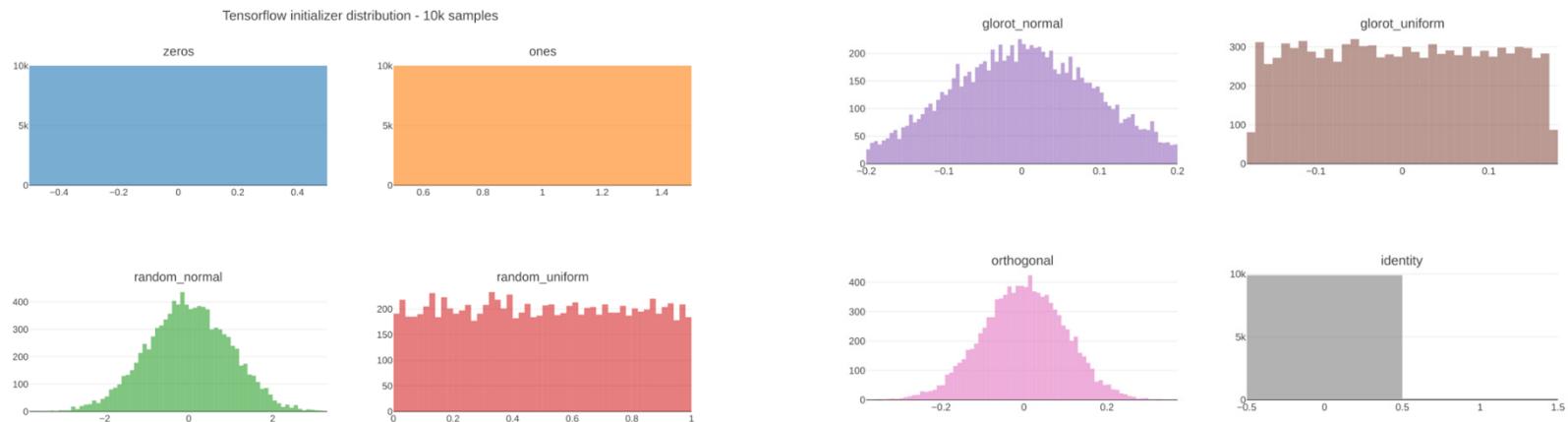


Available Initializations

- Zeros
- Ones
- Constant
- Random Normal
- Random Uniform
- Truncated Normal (only up to 2 standard deviations)
- Variance Scaling
- Orthogonal
- Identity
- Lecun Uniform
- Glorot Normal
- Glorot Uniform
- He Normal
- Lecun Normal
- He Uniform



Illustrations of Some Initializer Distributions





Lots of Available Initializers! Which do we use?

- A zero centered initialization within a small range, e.g. -1 to 1 is typically best or with small random numbers (CS231 recommendation)
- Good choices for Initializers
 - He Normal initialization works well with ReLU activations
 - Glorot Normal
 - Glorot Uniform – This is the Keras default
- Most times, initializer choice doesn't impact our accuracy in the end, it can affect the number of Epochs we take to get there though.

18.0

**Deep Surveillance - Build a Face Detector
with Emotion, Age and Gender Recognition**



Deep Surveillance - Build a Face Detector with Emotion, Age and Gender Recognition

- **18.1 Build an Emotion, Facial Expression Detector**
- **18.2 Build an Age and Gender Detector**
- **18.3 Build Emotion/Age/Gender Recognition in our Deep Surveillance Monitor**

18.1

Build an Emotion, Facial Expression Recognition



Our Facial Expression Dataset

- Our dataset is hosted on Kaggle and has been around since 2013.
- It consists of 28,709 examples of faces showing the 7 following expressions

Angry	Happy	Neutral
Disgust	Sad	
Fear	Surprise	

- Our images are grayscale and 48 x 48 pixels



Examples of our Dataset



444

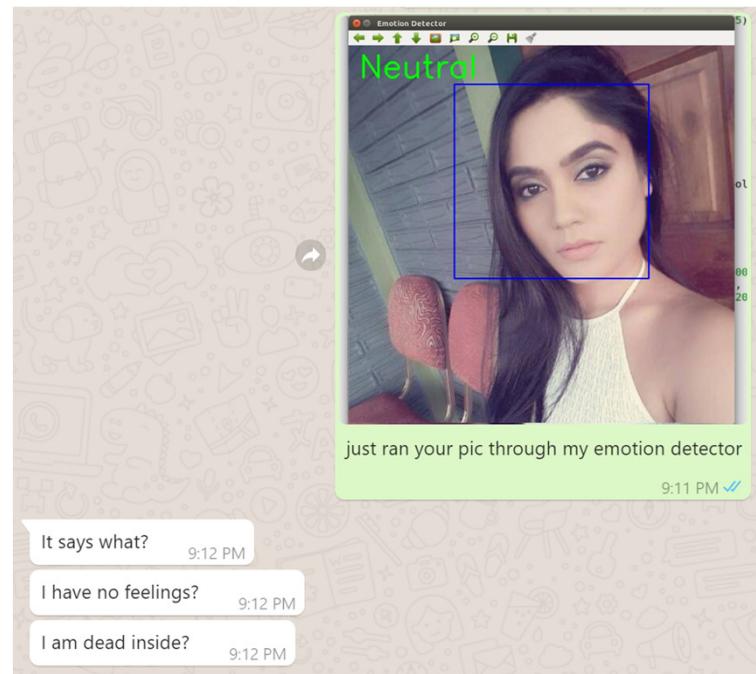


Our Approach

- Use our LittleVGG model with the following changes
 - Swapping ReLU activations for ELU
 - Changing our initializations from Kera's default Glorot Normal to He Normal (tends to work better for VGG type Networks)
- We then use OpenCV with a HAAR Cascade Classifier trained on faces, to extract our face from our webcams and classifier our emotions in real time!



Be Careful how you interpret the results



18.2

Build an Age and Gender Detector

18.3

**Deep Surveillance Combining
Emotion/Age/Gender Recognition**

19.0

Image Segmentation & Medical Imaging in U-Net



Image Segmentation & Medical Imaging in U-Net

- **19.1 What is Segmentation? And Applications in Medical Imaging**
- **19.2 U-Net: Image Segmentation with CNNs**
- **19.3 The Intersection over Union Metric**
- **19.4 Finding the Nuclei in Divergent Images**

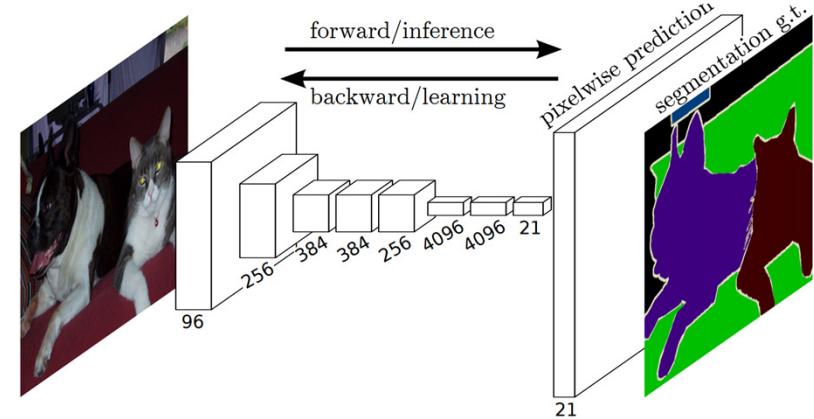
19.1

What is Segmentation? And Applications in Medical Imaging

What is Segmentation?



- The goal of segmentation is to separate different parts of image, into **sensible coherent parts**. Where we are basically predicting the class for each pixel in an image.
- There are two types of Segmentation.
 1. Semantic Segmentation
 2. Instance Segmentation





Type 1 – Semantic Segmentation

- Pixel classifications based on defined classes e.g. roads, persons, cars, trees etc.



Source - <https://medium.com/@keremturgutlu/semantic-segmentation-u-net-part-1-d8d6f6005066>



Type 2 – Instance Segmentation

- Pixel classifications combined with classifying object entities e.g. separate persons, cars etc.



Source - <https://medium.com/@keremturgutlu/semantic-segmentation-u-net-part-1-d8d6f6005066>

Applications in Medical Imaging

- Many medical applications necessitates finding and accurately labeling things found in medial scans.
- This is often done using advanced software to assist medical technicians and doctors. However, this task still requires human intervention and as such, can be tedious, slow, expensive and prone to human error.
- There's a huge initiative for use Computer Vision and Deep Learning to automate many of these tasks

There are hundreds of tasks in Medicine that can be improved with Computer Vision

- Robotic Surgery
- Analyzing medical scans such as:
 - CAT Scans
 - X-Rays
 - Ultrasound
 - PET
 - NMR
- The use cases are endless, from cancer detection, to disease monitoring, Alzheimer's and many other ailments. Computer Vision can revolutionize the medical industry allowing doctors and researchers to find cures faster, and provide better patient care.

19.2

U-Net – Image Segmentation with CNNs

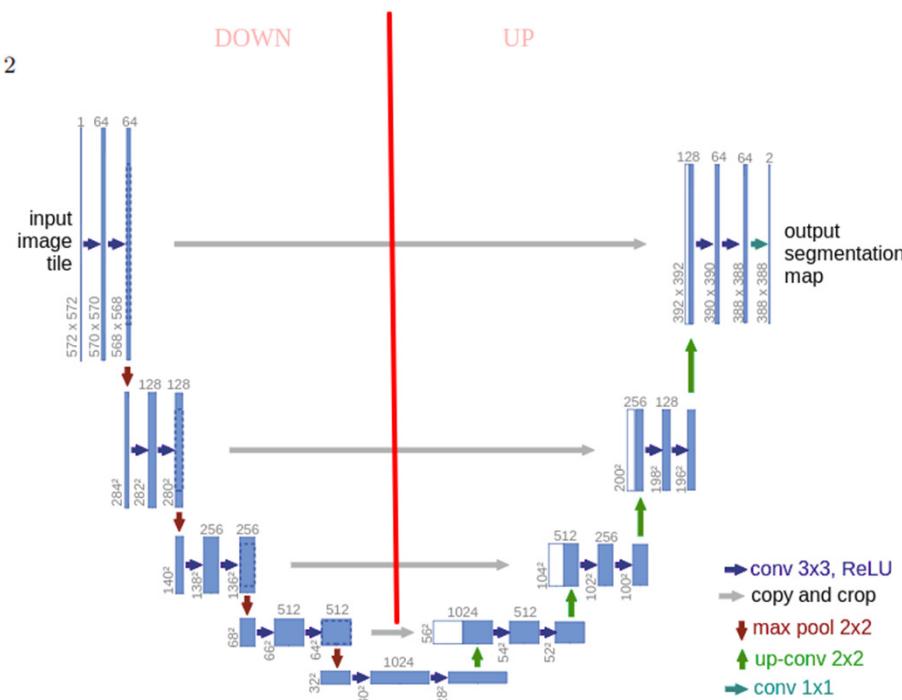


U-Net

- Created in 2015, U-Net was a unique CNN developed for Biomedical Image Segmentation.
- U-Net has now become a very popular end-to-end encoder-decoder network for semantic segmentation
- It has a unique Up-Down architecture which has a Contracting path and an Expansive path.



U-Net's Architecture





U-Net's Structure

Down Sample



Bottleneck



Up Sample

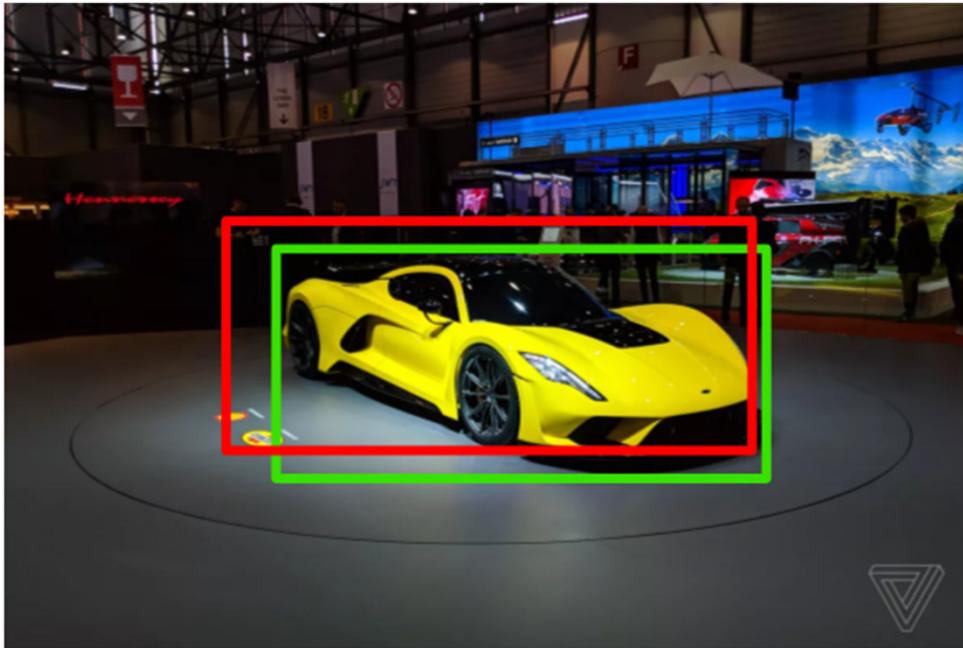
1. The downsampling path in U-Net consists of 4 blocks with the following layers:
 - 3x3 CONV (ReLU + Batch Normalization and Dropout used)
 - 3x3 CONV (ReLU + Batch Normalization and Dropout used)
 - 2x2 Max Pooling
 - Feature maps double as we go down the blocks, starting at 64, then 128, 256 and 512.
2. Bottleneck consists of 2 CONV layers with Batch Normalization & Dropout
3. The up sampling path consists of 4 blocks with the following layers:
 - Deconvolution layer
 - Concatenation with the feature map from the corresponding contracting path
 - 3x3 CONV (ReLU + Batch Normalization and Dropout used)
 - 3x3 CONV (ReLU + Batch Normalization and Dropout used)

19.3

The Intersection over Union Metric



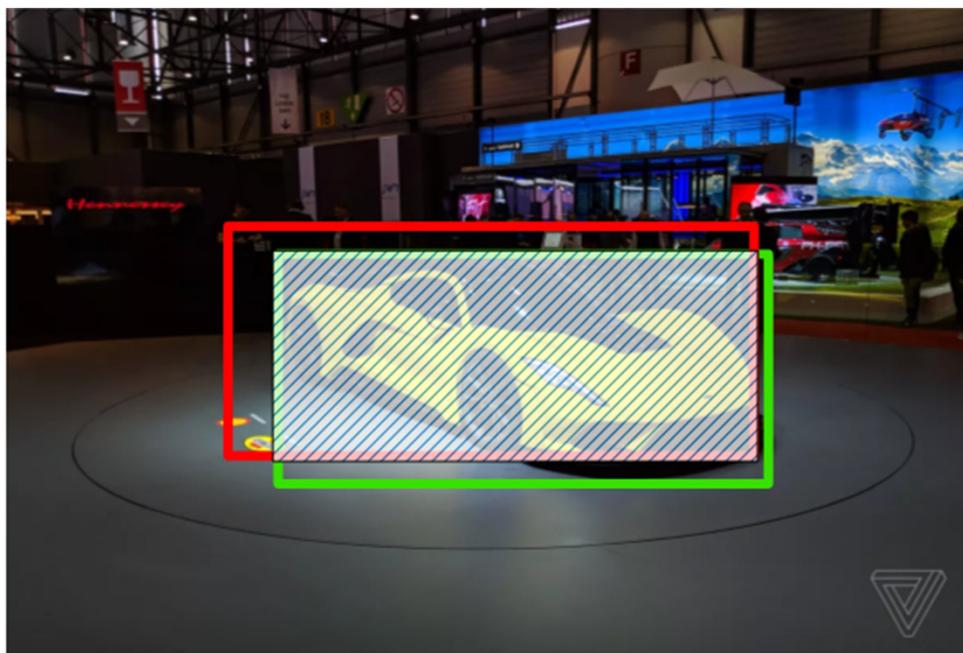
Segmentation Metrics



- Green is our true bounding box
- Red is our predicted bounding box



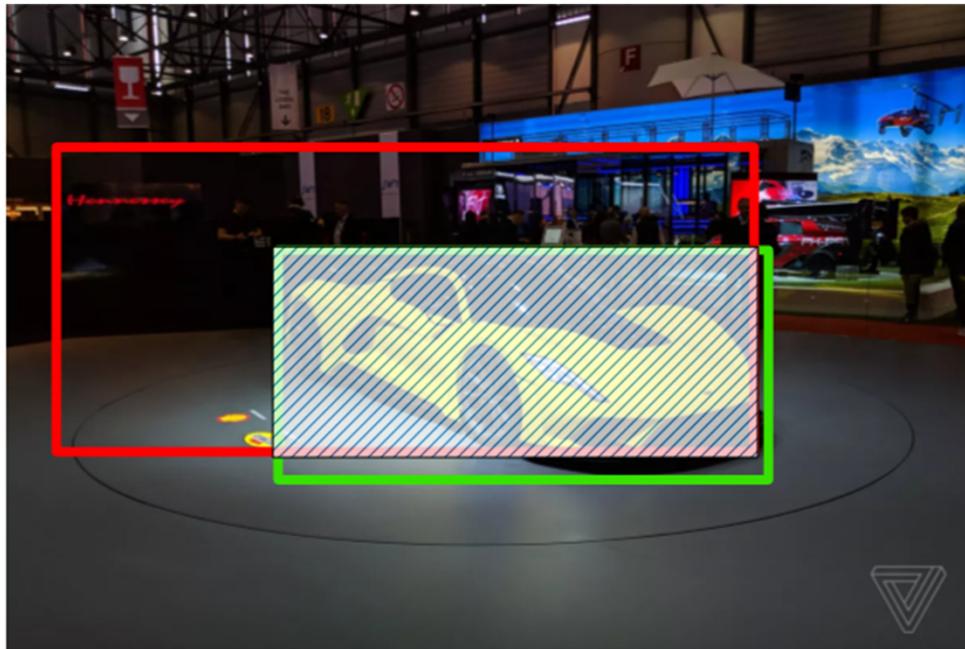
How much of the correct area is covered by our predicted bounding box?



- Honestly it seems close to 90%. However, is this really a good metric?



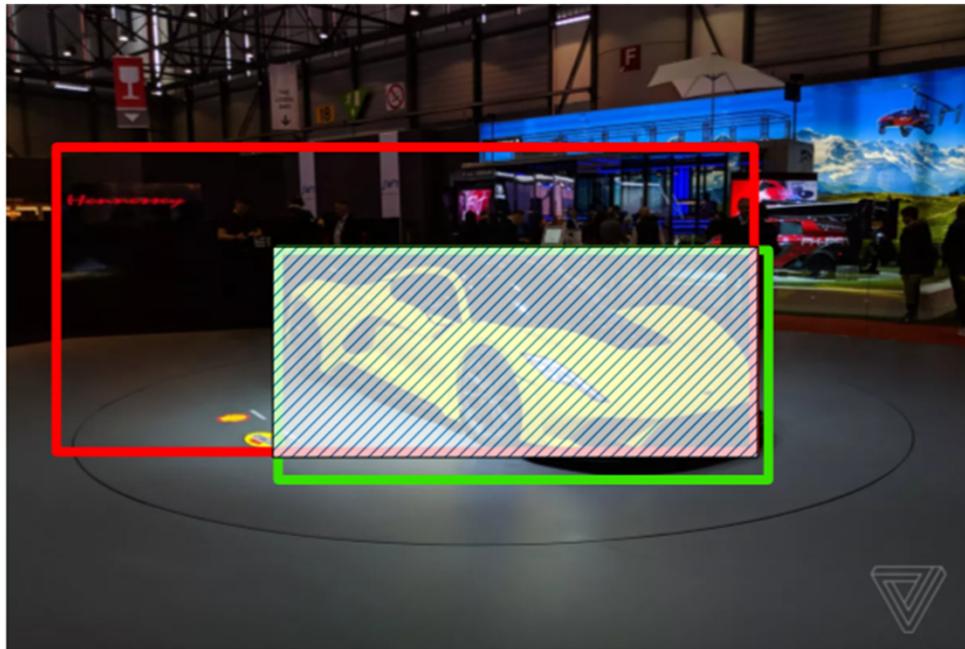
What if this was our predicted bounding box



- Still covers 90% of our original ground truth.
- This is where Intersection over Union comes in.



Intersection over Union (IoU)



- $IoU = \frac{\text{Size of Union}}{\text{Size of Prediction Box}}$
- Typically an IoU over 0.5 is considered acceptable
- The higher the IoU the better the prediction.
- IoU is essentially a measure of overlap



In Keras we can define Custom Metrics

- We can define a function that takes our ground truth masks and compares it with our predicted masks.

```
def iou_metric(y_true_in, y_pred_in):
    # Some IoU calculations
    return IoU score
```

- We use our custom metric function

```
model = Model(inputs=[inputs], outputs=[outputs])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[my_iou_metric])
model.summary()
```

- We can then see the output of our new custom function being used in training:

```
Epoch 1/10
603/603 [=====] - 83s 138ms/step - loss: 0.3371 - my_iou_metric: 0.0746 - val_loss: 0.3056 - val_my_iou_metric: 0.1810
```

19.4

Finding the Nuclei in Divergent Images



The Kaggle Data Science Bowl 2018 Challenge

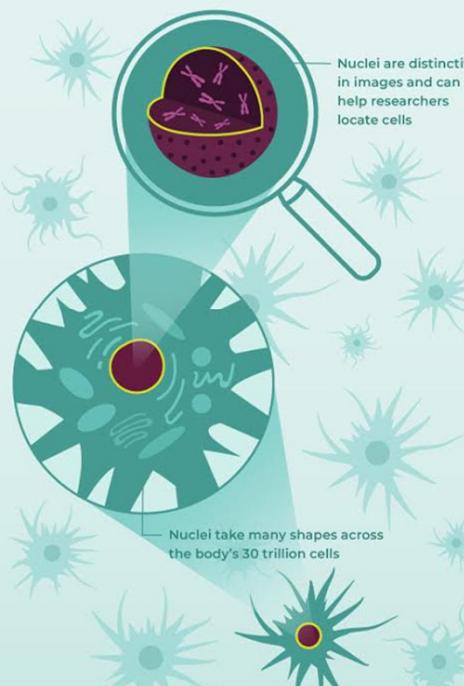
Spot Nuclei. Speed Cures.

- By automating nucleus detection, you could help unlock cures faster.
- “Identifying the cells' nuclei is the starting point for most analyses because most of the human body's 30 trillion cells contain a nucleus full of DNA, the genetic code that programs each cell. Identifying nuclei allows researchers to identify each individual cell in a sample, and by measuring how cells react to various treatments, the researcher can understand the underlying biological processes at work.”

Spot Nuclei. Speed Cures.

The challenge: Create an algorithm to automate nucleus detection

40% of all deaths are caused by illnesses like heart disease and cancer ¹	75% of rare diseases affect children ²	30% of affected children with rare diseases die before age 5 ³
--	---	---



Nuclei are distinctive in images and can help researchers locate cells

Nuclei take many shapes across the body's 30 trillion cells

Finding the nucleus helps to...

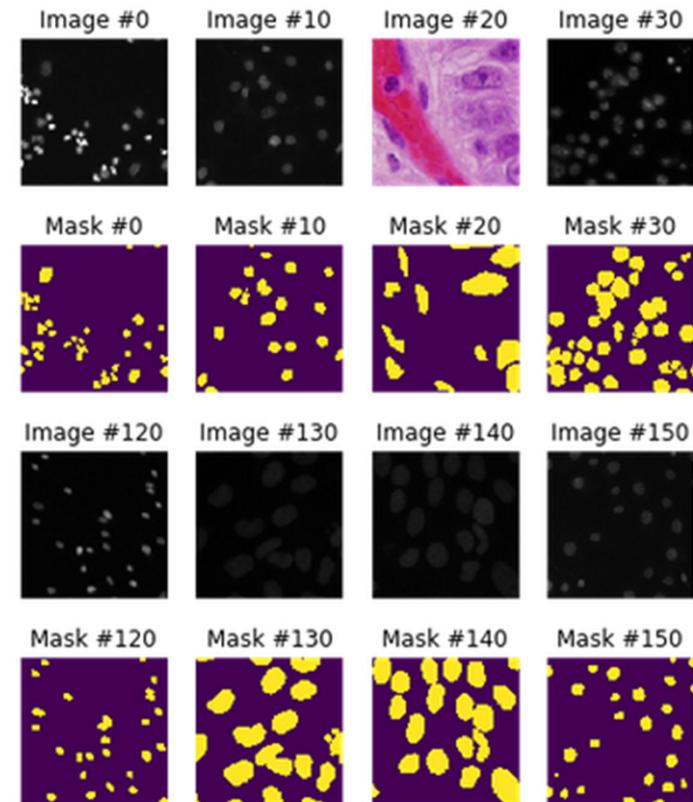
-  locate cells in varied conditions to enable faster cures
-  free biologists to focus on solutions
-  improve throughput for research and insight
-  reduce time-to-market for new drugs—currently 10 years
-  increase # of compounds for experiments
-  improve health and increase quality of life

SOURCES: ¹Heart disease and cancer causing 40% of all deaths – World Health Organization (WHO). 2015 – 56.4 million deaths. <http://www.who.int/mediacentre/factsheets/fs310/en/>
 2015 – 1.77 deaths from cardiovascular diseases (CVDs). <http://www.who.int/mediacentre/factsheets/fs317/en/>
 2015 – 8.8 million deaths from cancer. <http://www.who.int/mediacentre/factsheets/fs297/en/>

²Childhood diseases - 75% of rare diseases affect children; 30% die before age 5 – European Society of Paediatric Oncology (SIOP). <http://www.siop-eu.org/EU/English/SIOP-EU/Advocacy-Activities/Rare-Diseases.page.aspx/249>

Sample Images, Objective and our approach

- Objective - Automate the generation of the image masks
- Approach – Use U-Net, a special CNN designed for segmentation tasks to generate these masks.



20.0

Principles of Object Detection



Principles of Object Detection

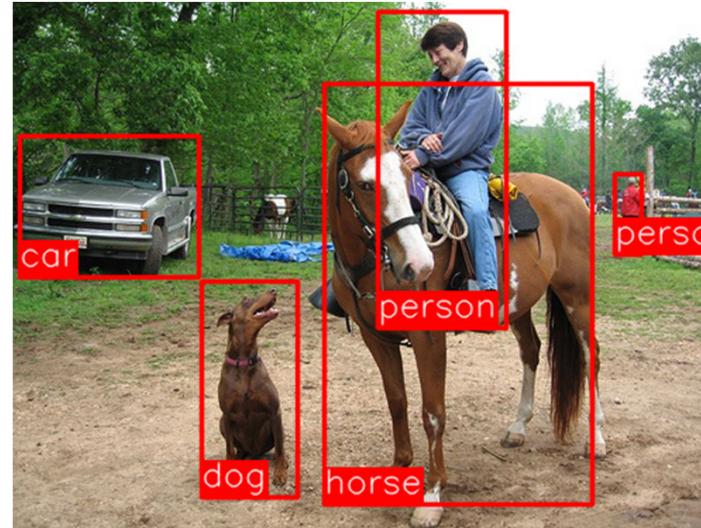
- **20.1 Object Detection Introduction - Sliding Windows with HOGs**
- **20.2 R-CNN, Fast R-CNN, Faster R-CNN and Mask R-CNN**
- **20.3 Single Shot Detectors (SSDs)**
- **20.4 YOLO to YOLOv3**

20.1

Object Detection Introduction - Sliding Windows with HOGs

Object Detection – The Holy Grail of Computer Vision

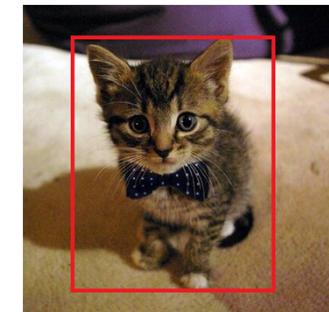
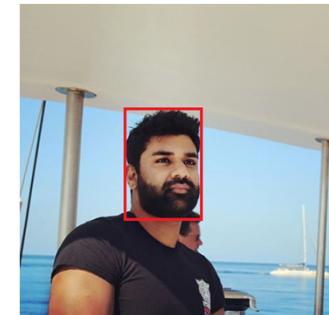
- Previously, we have simply been doing various image classification tasks with pretty good success even on our CPU systems.
- But can we do this?





Object Detection

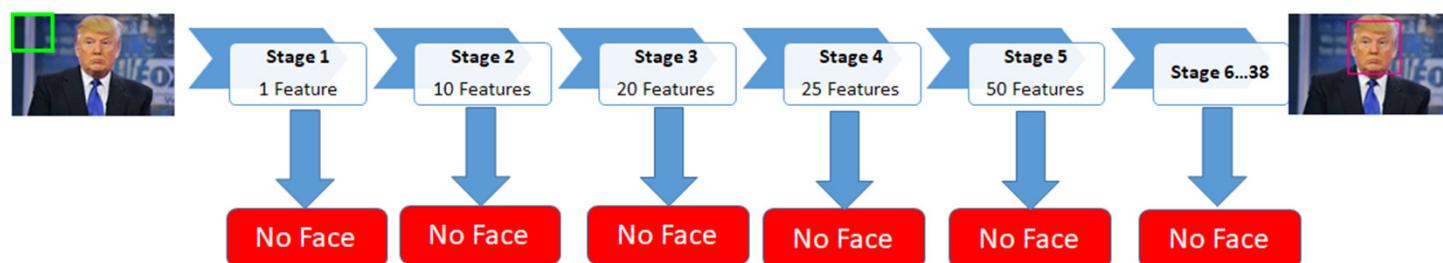
- It is a mix of Object Classification and Localization.
- Object Detection is the **identification of a bounding box**, outlining the object we wish to detect.
- Face Detection as seen in almost all camera apps, is a perfect example of object detection.
- Instead of stating whether an image is a cat or not, Object Detection tells answers the question, "**Where is the cat?**"





History of Object Detection – HAAR Cascade Classifiers

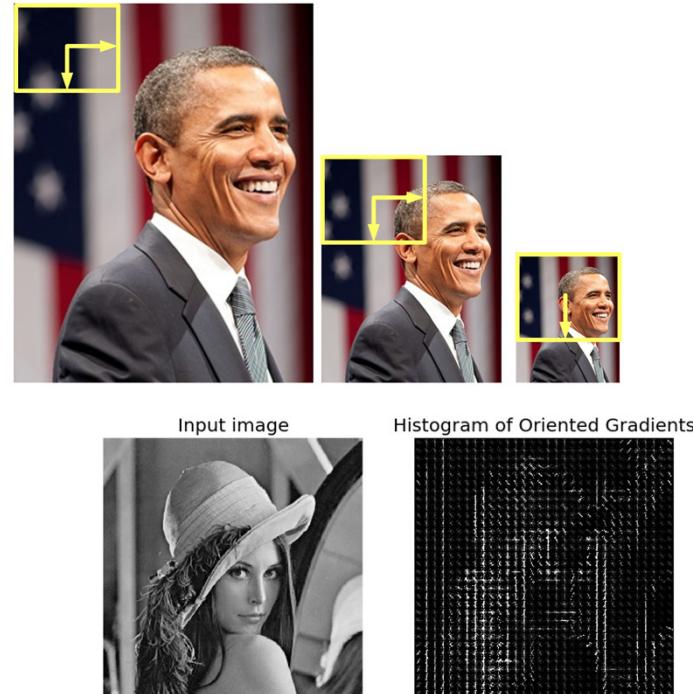
- Object Detection algorithms have been around since the early days of Computer Vision (first good attempts were in the 1990s).
- However, the first really successful implementation of Object Detection was in 2001 with the introduction of Haar Cascade Classifiers used by Viola-Jones in their Face Detection algorithm. A very fast and easy to use detector, with good enough but not great accuracy.
- It takes Haar features into a series of cascaded classifiers and uses a series of steps to determine if a face has been detected.
- Haar cascades are very effective but difficult to develop, train and optimize.





Histogram of Gradients and SVM Sliding Window Detectors

- Sliding Windows is a method where we extract segments of our image, piece by piece in the form of a rectangle extractor window.
- In each window we computer the Histogram of Gradients (HoGs) and compute how closely it matches the object we are trying to detect.
- SVMs are typically used to classify our HoG features.





Previous Object Detection Methods are Simply Manual Feature Extraction combined with Sliding Windows

- As you may have noticed, previous methods such as Haar Cascade Classifiers and HoG + SVM are essentially manual Feature Extraction Methods combined with the use of a Sliding window..
- We can eliminate some complexity by using a sliding window method with a CNN. However, the following **problems** still occur.
 - Using a sliding window of across an image results in hundreds even thousands of **classifications** done for a single image.
 - **Scaling issues**, what size window do we use? Is pyramiding robust enough?
 - What if the window isn't defined as we set it out to be. Do we use windows of different ratios? You can easily see how this can blow up into **millions of classifications** needed for a large image.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2																			
3																			
4																			
5																			
6																			
7																			
8																			
9																			
10																			
11																			
12																			
13																			
14																			
15																			
16																			
17																			
18																			
19																			
20																			



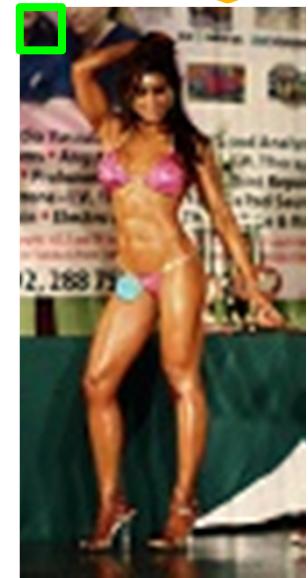
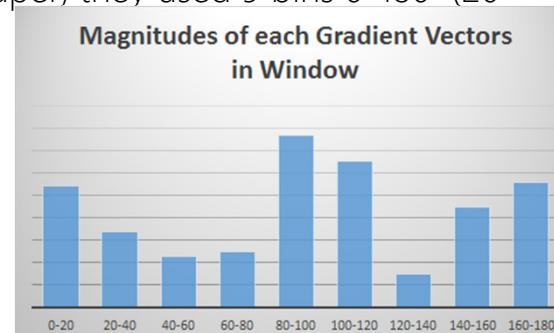
Histogram of Gradients (HoGs)

- HOGs are a feature descriptor that has been widely and successfully used for object detection.
- It represents objects as a single feature vector as opposed to a set of feature vectors where each represents a segment of the image.
- It's computed by sliding window detector over an image, where a HOG descriptor is a computed for each position. Like SIFT the scale of the image is adjusted (pyramiding).
- HOGs are often used with SVM (support vector machine) classifiers. Each HOG descriptor that is computed is fed to a SVM classifier to determine if the object was found or not).

Paper by Dalal & Triggs on using HOGs for Human Detection: - <https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf>

Histogram of Gradients (HoGs) Step by Step

1. Using an 8 x 8 pixel detection window or cell (in green), we compute the gradient vector or edge orientations at each pixel.
2. This generates 64 (8 x 8) gradient vectors which are then represented as a histogram.
3. Each cell is then split into angular bins, where each bin corresponds to a gradient direction (e.g. x, y). In the Dalal and Triggs paper, they used 9 bins $0-180^\circ$ (20° each bin).
4. This effectively reduces 64 vectors to just 9 values.
5. As it stores gradients magnitudes, it's relatively immune to deformations.



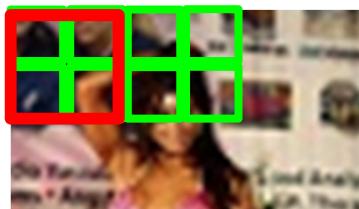
64 x 128



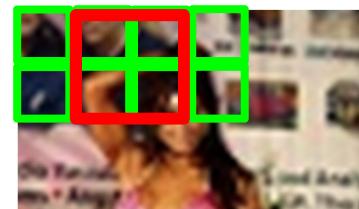
Histogram of Gradients (HoGs) Step by Step

6. We then **Normalize** the gradients to ensure invariance to illumination changes i.e. Brightness and Contrast. E.g. in the images on the right, if **we divide the vectors by the gradient magnitudes** we get 0.707 for all, this is normalization.

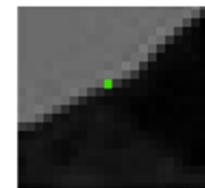
7. Instead of individual window cell normalization, a method called Block Normalization is used. This takes into account neighboring blocks so we normalize taking into consideration larger segments of the image.



Block 1

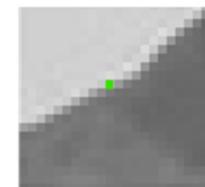


Block 2



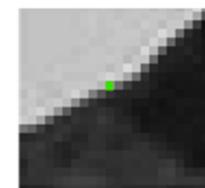
101		
91		51
	61	

$$\begin{aligned}\Delta_H &= 50 \\ \Delta_V &= 50 \\ |\Delta| &= \sqrt{50^2 + 50^2} = 70.72\end{aligned}$$



171		
161		121
	121	

$$\begin{aligned}\Delta_H &= 50 \\ \Delta_V &= 50 \\ |\Delta| &= \sqrt{50^2 + 50^2} = 70.72\end{aligned}$$



181		
171		71
	81	

$$\begin{aligned}\Delta_H &= 100 \\ \Delta_V &= 100 \\ |\Delta| &= \sqrt{100^2 + 100^2} = 141.42\end{aligned}$$

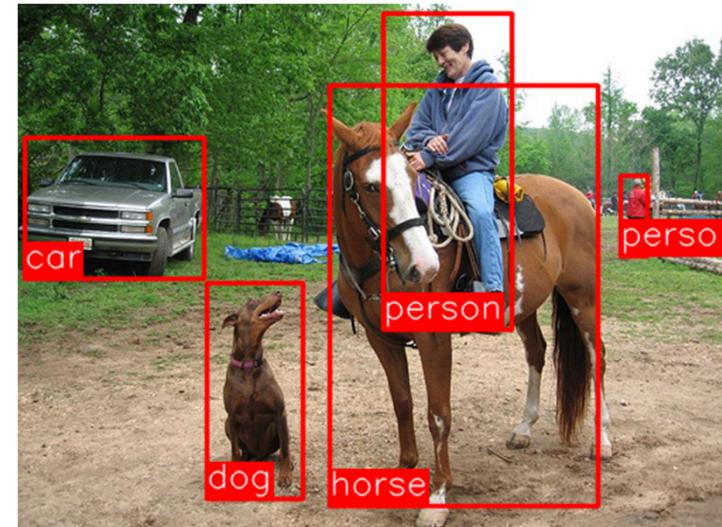
20.2

**R-CNN, Fast R-CNN, Faster R-CNN and Mask
R-CNN**



Regions with CNNs or R-CNNs

- Introduced in 2014 by researchers at University College of Berkeley, R-CNNs obtained dramatically higher performance in the PASCAL VOC Challenge (ImageNet for Object Detection testing).



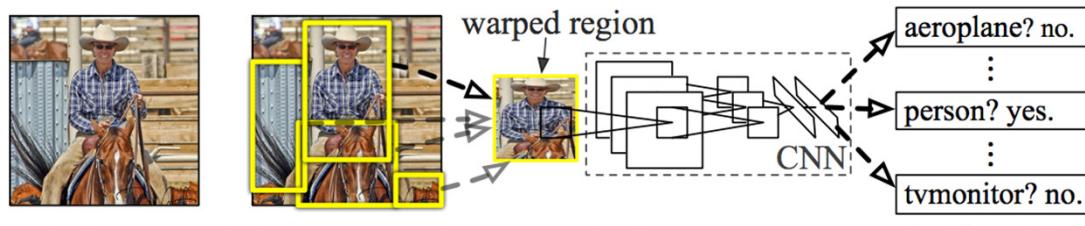
<https://arxiv.org/abs/1311.2524>



R-CNNs Principles

- R-CNNs attempted to solve the exhaustive search previously performed by sliding windows, by proposing bounding boxes and passing these extracted boxes to an image classifier.
- How do we do these bounding box proposals? By using the Selective Search algorithm.

R-CNN: *Regions with CNN features*



1. Input
image

2. Extract region
proposals (~2k)

3. Compute
CNN features

4. Classify
regions



Selective Search

- Selective Search attempts to segment the image into groups by combining similar areas such as colors/textures and propose these regions as “interesting” bounding boxes.

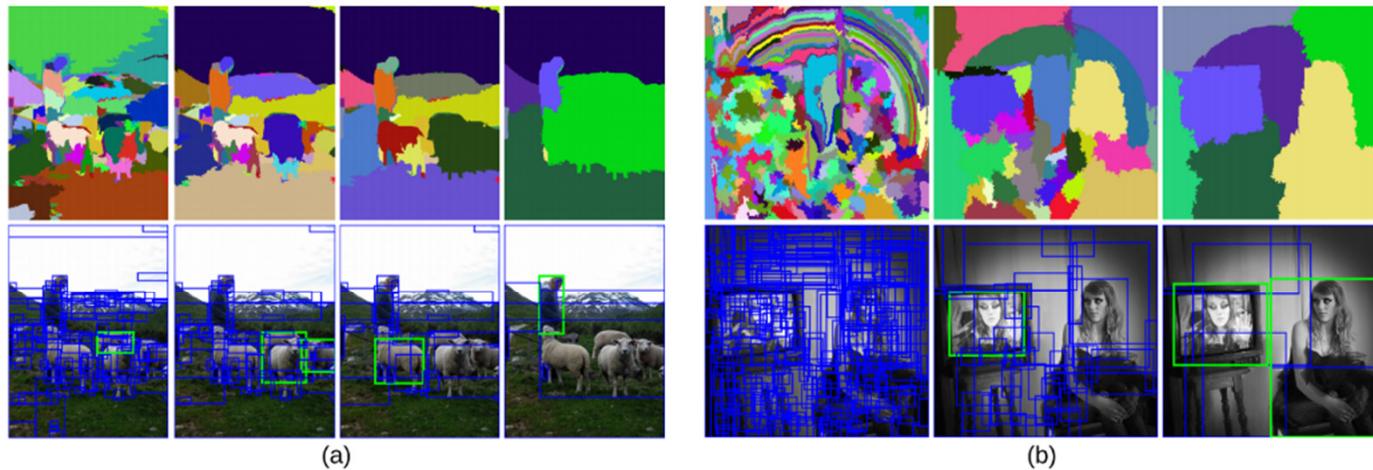
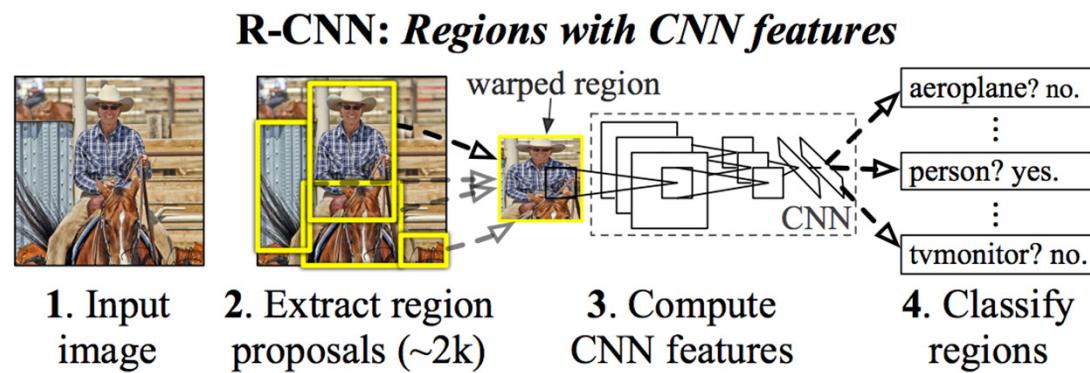


Figure 2: Two examples of our selective search showing the necessity of different scales. On the left we find many objects at different scales. On the right we necessarily find the objects at different scales as the girl is contained by the tv.



R-CNNs Principles

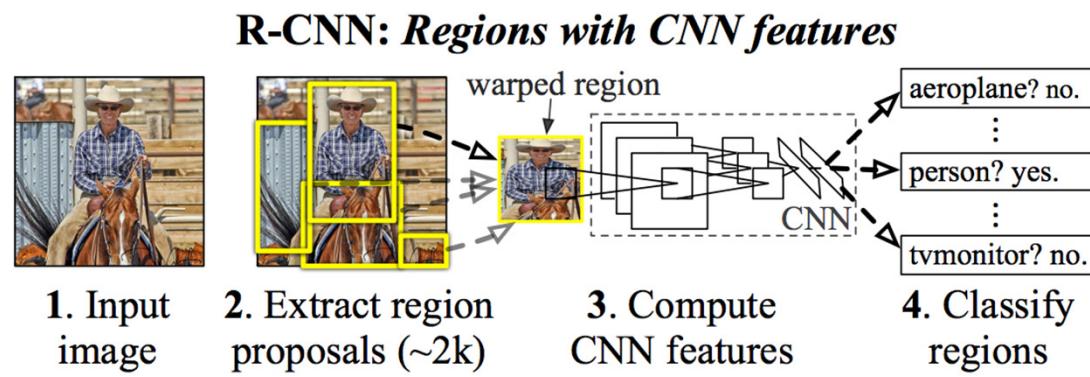
- When selective search has identified these regions/boxes it passes this extracted image to our CNN (e.g. one trained on ImageNet) for classification. We don't use the CNN directly for classification (although we can), we use an SVM to classify the CNN extracted features.





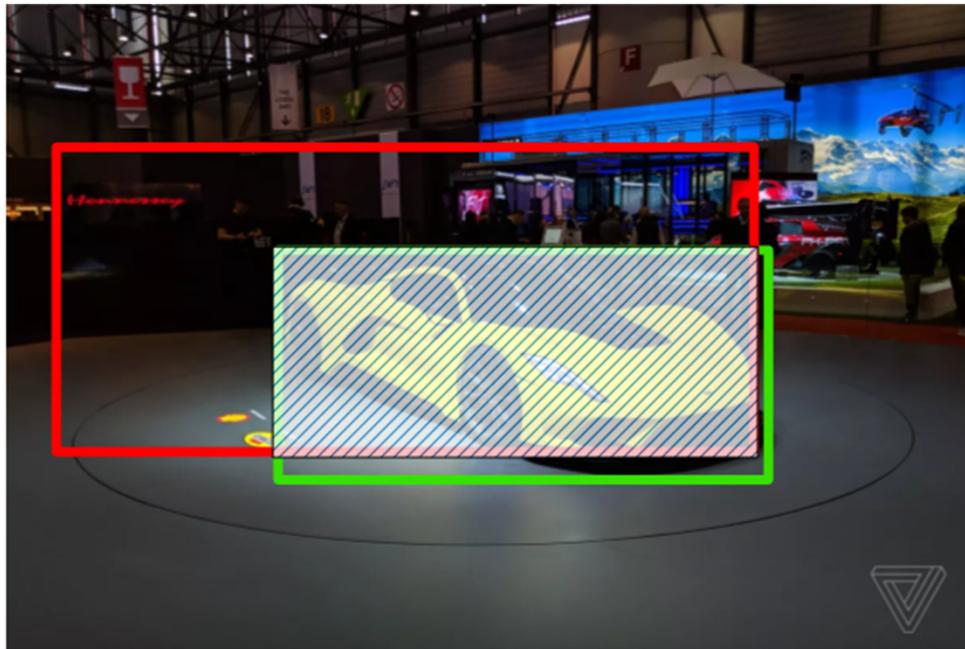
R-CNN – Tightening of Boxes

- After the first region proposal has been classified, we then use a simple linear regression to generate a tighter bounding box.
- But what is a good box?





Remember our Intersection over Union (IoU) Metric?

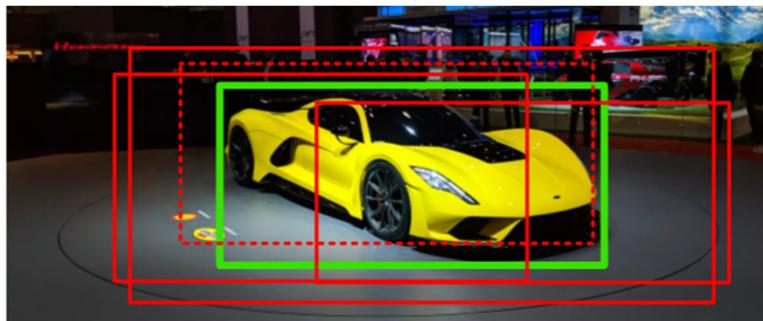


- $IoU = \frac{\text{Size of Union}}{\text{Size of Prediction Box}}$
- Typically an IoU over 0.5 is considered acceptable
- The higher the IoU the better the prediction.
- IoU is essentially a measure of overlap



Mean Average Precision (mAP)

- Remember and IoU of 0.5 was considered a positive result. What if we have multiple boxes predicted for the same object (see below). This is a common problem with Object Detectors are overlapping boxes on the same object..
- In the figure below we have 4 predicted boxes (in red) with one being the best (dotted red line). As such, we have one True-Positive and 3 False-Positives.



mAP is a confusing metric, an excellent explanation is found here:
<https://sanchom.wordpress.com/tag/average-precision/>

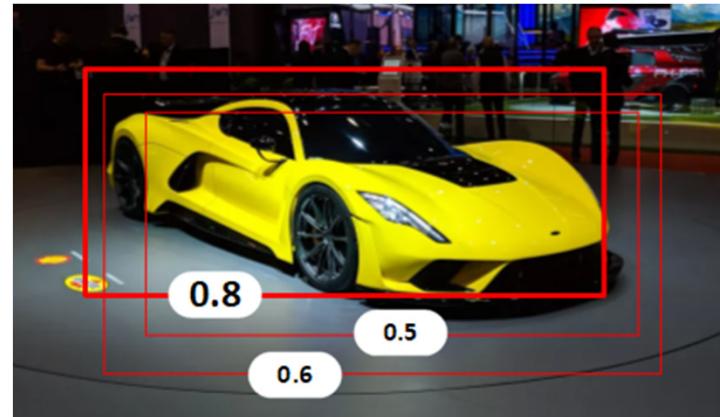
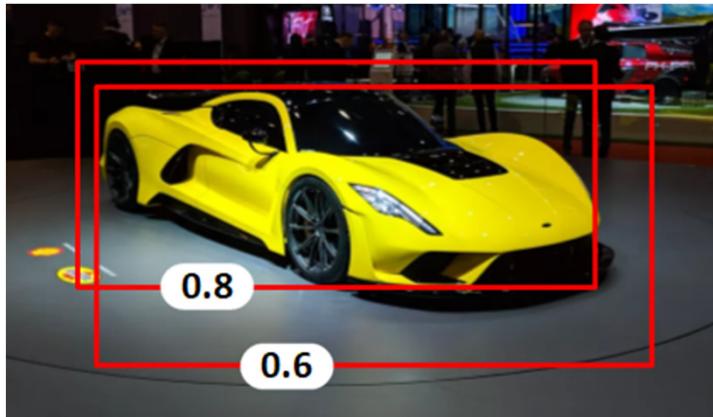
$$\text{Ave Precision}_{\text{Class}} = \frac{\sum \text{True Positives}_{\text{Class}}}{\sum \text{True Positives}_{\text{Class}} + \sum \text{False Positives}_{\text{Class}}}$$

$$mAP = \frac{1}{\text{Number of classes}} \sum_{\text{All Classes}} \frac{\sum \text{True Positives}_{\text{Class}}}{\sum \text{True Positives}_{\text{Class}} + \sum \text{False Positives}_{\text{Class}}}$$



Non-Maximum Suppression

- One technique used to improve mAP scores from providing overlapping boundary boxes.
- STEP 1 – It looks at the maximum probabilities associated with each box (i.e. the probability of the object belonging to its class e.g. a car)
- STEP 2 – Suppress boundary boxes with a high overlap or IoU with the high probability boxes





Fast R-CNN - 2015

- R-CNN was effective but extremely slow as each proposed bounding box has to be classified by our CNN, as such doing real-time Object Detection was far from possible.
- It required 3 models be trained separately.
 1. Feature extraction CNN
 2. SVM to predict the class
 3. Linear Regression model to tighten the bounding box
- Faster R-CNNs solve this problem!



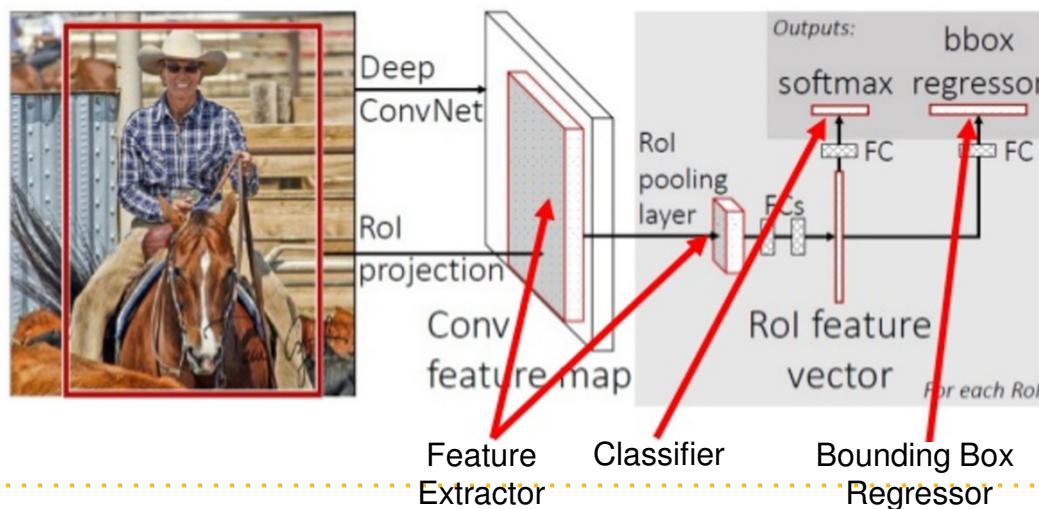
Reducing overlap in Bounding Boxes

- Faster R-CNNs firstly reduced the number of proposed bounding boxes by removing the overlap generated. How?
- We run the CNN across the image just once use a technique called Region of Interest Pooling (RoIPool).
- RoIPool allows us to share the forward pass of a CNN for the image across its sub regions. This works because previously regions are simply extracted from the CNN feature map and then pooled. Therefore, there is only need to run our CNN once on the image!



Combining the training of the CNN, Classifier and Bounding Box Regressor into a single Model

- SVM Feature Classifier -> Softmax layer at the top of the CNN
- Linear Regression -> Bounding Box output layer (parallel to our Softmax)





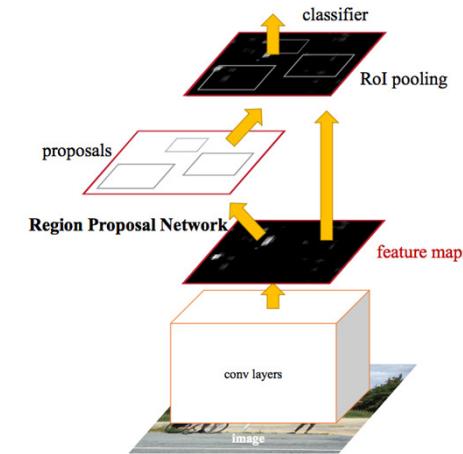
Faster R-CNN - 2016

- Fast R-CNNs made significant speed increases however, region proposal still remained relatively slow as it still relied of Selective Search.
- Fortunately, a Microsoft Research team figured out a how to eliminate this bottleneck



Speeding up Region Proposal

- Selective Search relies on features extracted from the image. What if we just reused those features to do Region Proposal instead?
- That was the insight that made Faster R-CNNs extremely efficient.



<https://arxiv.org/abs/1506.01497>



Region Proposals with Faster R-CNNs

- Faster R-CNNs add a fully convolutional network on top of the features of the CNN to create a Region Proposal Network.
- The authors of the paper state "*The Region Proposal Network slides a window over the features of the CNN. At each window location, the network outputs a score and a bounding box per anchor (hence $4k$ box coordinates where k is the number of anchors).*" <https://arxiv.org/abs/1506.01497>
- After each pass of this sliding window, it outputs k potential bounding boxes and a score or confidence of how good this box is expected to be.



Producing Bounding Boxes

- Previously we mentioned we produced 'k' potential bounding boxes. These bounding box proposals are proposals of common or expected boxes of certain shapes/aspect ratios and sizes. These aspect ratios are called **anchor boxes**.
- The Region Proposal Network outputs a bounding box per anchor and a score of how likely the image in the box will be an object.



Mask R-CNNs – Pixel Level Segmentation

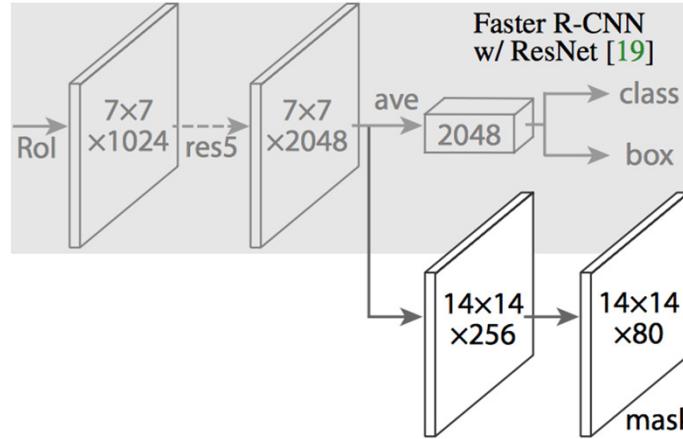


- Mask R-CNNs aim to combine object detection and classification with segmentation.
- As we previously saw, segmentation is the labeling of objects per pixel level as seen the above image.



Mask R-CNNs

- Mask R-CNNs are an extension of Faster R-CNNs where a binary mask is created for the object in the detected bounding box.



<https://arxiv.org/abs/1703.06870>

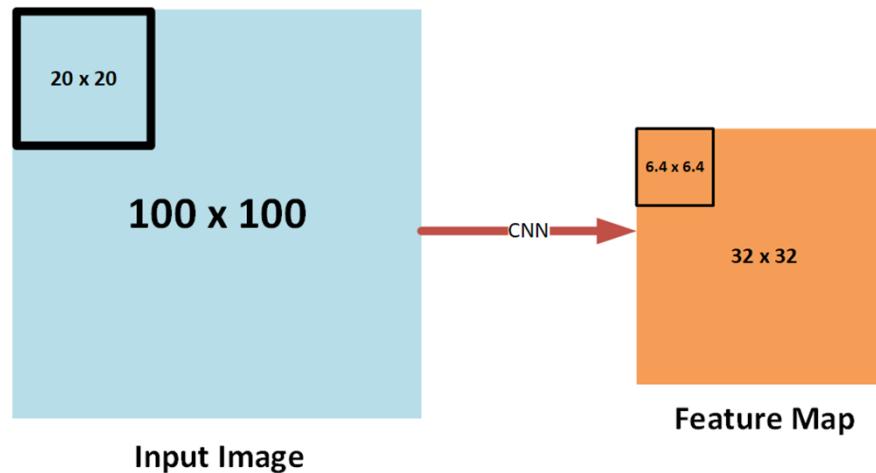


Mask R-CNNs – RoIAlign

- The Mask output uses the CNN extracted features to create it's binary mask.
- How did the authors of this paper achieve this?
- By using RoIAlign instead of RoIPool as RoIPool's feature map were misaligned from regions of the original image.



Mapping a Region of Interest onto a Feature Map



- Our 100×100 image is now mapped to a 32×32 feature map. Therefore, our window of 20×20 on the original image is mapped to 6.4×6.4 on our Feature Map.
- RoIPool however, rounds down our pixel map to 6×6 , causing the misalignment.
- RoIAlign however, uses bilinear interpolation to know what would be pixel 6.4 .



Examples of Mask R-CNN segmenting and classifying



<https://arxiv.org/abs/1703.06870>

20.3

Single Shot Detectors (SSDs)



Single Shot Detectors (SSDs)

- We've just discussed the R-CNN family and we've seen how successful they can be. However, their performance on video is still not optimal, running typically at 7 fps.
- SSDs aim to improve this speed by eliminating the need for the Region Proposal Network.

System	VOC2007 test <i>mAP</i>	FPS (Titan X)	Number of Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	~6000	~1000 x 600
YOLO (customized)	63.4	45	98	448 x 448
SSD300* (VGG16)	77.2	46	8732	300 x 300
SSD512* (VGG16)	79.8	19	24564	512 x 512

<https://arxiv.org/pdf/1512.02325.pdf>



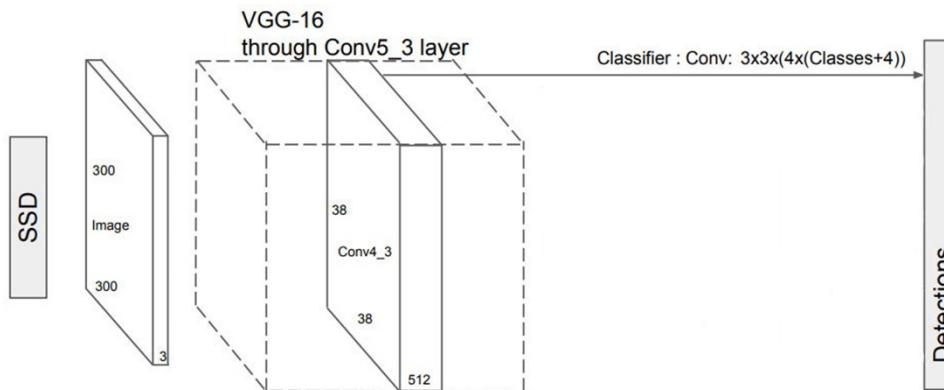
How do SSD's Improve Speed?

- SSD's use multi-scale features and default boxes as well as dropping the resolution images to improve speed.
- This allows SSD's to achieve real-time speed with almost no drop (sometimes even improved) accuracy.



SSD Structure

- SSD's are composed of two main parts
 - Feature Map Extractor (VGG16 was used in the published paper but ResNet or DenseNet may provide better results))
 - Convolution Filter for Object Detection

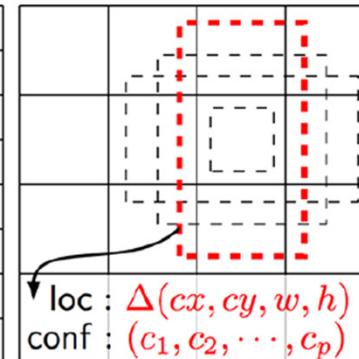
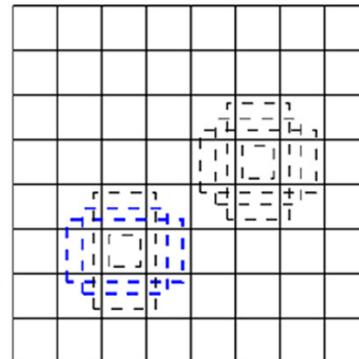
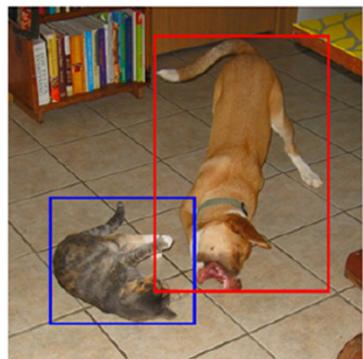


<https://arxiv.org/pdf/1512.02325.pdf>



Discretizing Feature Maps

- Using the VGG16's CONV4_3 Layer it makes 4 to 6 (user set) object predictions (shown below) for each cell.
- It predicts the class scores and adds one extra for no object being found.
- Fewer cells allow larger objects to be detected (e.g. the dog and right rightmost diagram with the red box) and large number of cells allow more granular detection of smaller objects (e.g. the cat).



loc : $\Delta(cx, cy, w, h)$
conf : (c_1, c_2, \dots, c_p)



VGG16 to Extract Feature Maps

- For each bounding box we obtain the probabilities of all classes within the region.
- This allows us to produce overlapping boxes where multiple objects occur.



MultiBox Algorithm & Loss Functions

Making multiple predictions for boundary boxes and confidence scores is called MultiBox.

Loss Functions uses in Training:

- **Class predictions** - Categorical cross-entropy
- **Location or localization loss** - Smooth L1-loss. SSD only penalizes predictions of positive matches and ignores negative matches as we want to get our positives as close to the ground truth (getting it exact is impossible, but close enough works for us here)



SSD Summary

- SSDs are faster than Faster R-CNN but less accurate in detecting small objects.
- Accuracy increases if we increase the number of default boxes as well as better designed boxes
- Multi-scale feature maps improve detection at varying scales.

20.4

YOLO to YOLOv3



YOLO or You Only Look Once

- The idea behind YOLO is that a single neural network is applied to full image. This allows YOLO to reason globally about the image when generating predictions
- It is a direct development of MultiBox, but it turns MultiBox from region proposal in to an objection recognition method by adding a softmax layer in parallel with a box regressor and box classifier layer.
- It divides the image into regions and predicts bounding boxes and probabilities for each region.
- YOLO uses a Fully Convolution Neural Network allowing for input of various image sizes.





YOLO or You Only Look Once

- The input image is divided into an $S \times S$ grid, if the center of an object falls into this grid cell, that cell is responsible for detecting that object.
- Each grid predicts a number of bounding boxes and confidence scores for those boxes.
- Confidence here is defined as Probability of an Object multiplied by the thresholded IoU score, where IoU scores of < 0.5 are given a confidence of zero.
- The bounding box is defined by x, y, w, h where x, y are the center of the box and $w & h$ are the height and width,
- By multiplying the conditional class probability and the individual box confidence predictions, we get the class-specific confidence score for each box.

$$Prob(\text{Class}|\text{Object}) \times Prob(\text{Object}) \times IoU = Prob(\text{Class}) \times IoU$$

YOLO Model

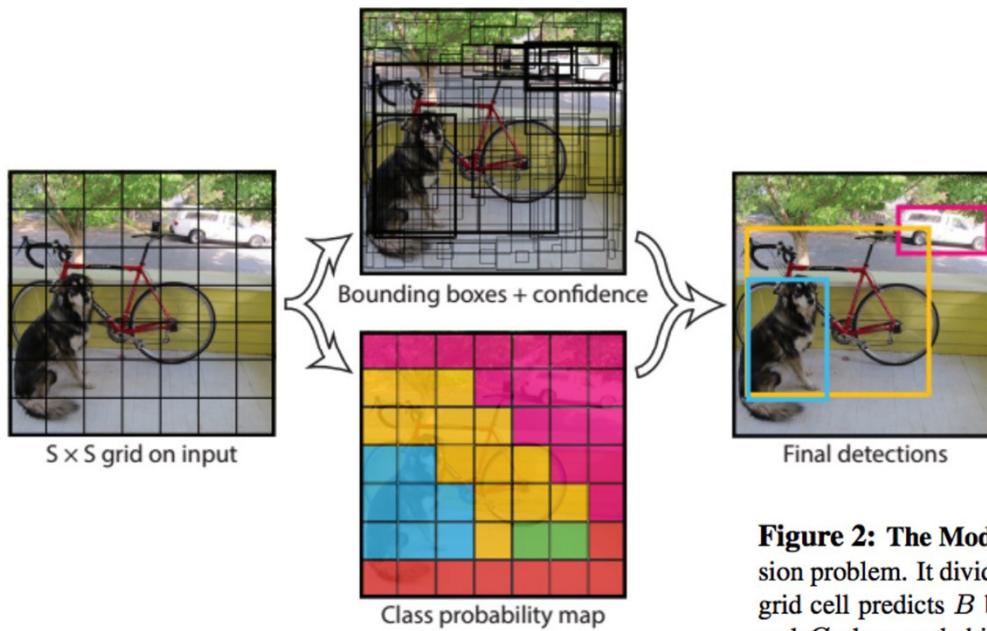


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.



Loss Function Adjustments

- During training, YOLO uses differential weight for confidence predictions from boxes that contain object and boxes that do not contain objects.
- It penalizes errors in small and large objects differently by predicting the square root of the bounding box width and height.



YOLO Architecture

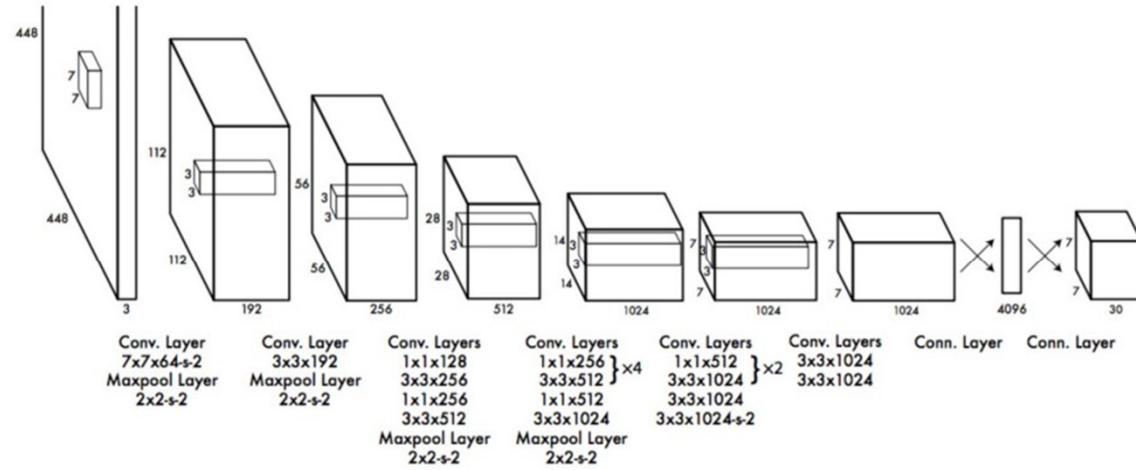


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.



YOLO Evolution

- YOLO first appeared in 2016 and was voted the OpenCV's People Choice Award at CVPR.
- YOLOv2 was later released where Batch Normalization was added which resulted in mAP improvements of 2%. It was also fine tuned to work at higher resolution (448 x 448) giving a 4% increase in mAP.
- YOLO3 was fine tuned even further and introduced multi-scale training to better help detect smaller objects.

21.0

TensorFlow Object Detection API



TensorFlow Object Detection API

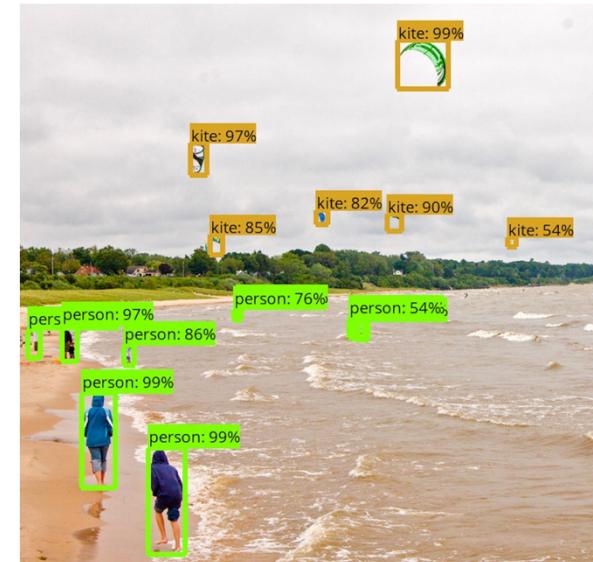
- **21.1 TFOD API Install and Setup**
- **21.2 Experiment with a ResNet SSD on images, webcam and videos**
- **21.3 How to Train a TFOD Model**

21.1

TFODI Install and Setup

TensorFlow Object Detection

- The TFOD API is one of the more mature and relatively easy to use Object Detection frameworks.
- Typically most other existing Object Detection frameworks are finicky, difficult to use and have way too many moving parts that break easily.
- TFOD attempts to solve that by creating a framework API that uses TensorFlow (no surprise there) to create Object Detection models using both the R-CNN family as well as SSD.
- While the TFOD API makes it far easier than alternative methods, it's still has a bit a of a learning curve.





TFOD Install and Setup

```
# Clone your existing CV environment to avoid re-installing tones of packages and libraries
source activate cv
conda create --name <tfod> --clone base
sudo apt-get install protobuf-compiler python-pil python-lxml python-tk
pip install --user Cython
pip install --user contextlib2
pip install --user jupyter
pip install --user matplotlib
cd
# Create this directory in your home folder
mkdir models
git clone https://github.com/tensorflow/models
cd
git clone https://github.com/cocodataset/cocoapi.git
cd cocoapi/PythonAPI
make
cp -r pycocotools /home/deeplearningcv/models/models/research
# From tensorflow/models/research/
wget -O protobuf.zip https://github.com/google/protobuf/releases/download/v3.0.0/protoc-3.0.0-linux-x86_64.zip
unzip protobuf.zip
# From tensorflow/models/research/ lets run the test to make sure everything works
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
python object_detection/builders/model_builder_test.py
```

If the above does not work updates may have been made to TFOD so check this link to verify:

https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/installation.md

Running the TFOD Demo

- Download the ipython Notebook in the resource section of this chapter and place it in this folder:
 - + models
 - + models
 - + research
 - + **object_detection**
 - **object_detection_tutorial.ipynb**
- Or simply navigate to that directory from your existing ipython notebook (make sure you're in the TFOD environment we just created)



21.2

**Experiment with a ResNet SSD on images,
webcam and videos**

21.3

How to train a custom TensorFlow Object Detection Model



Training the TFOD

- The training process of the TFOD is a bit messy and could be better explained in the online tutorial. However, let's take a look at the steps involved:





TF_Records

- TFOD expects our training and test data to be in the `tf_record` format.
- Luckily we can convert existing datasets e.g. Pascal VOC (which is stored in a XML structure) directly to `tf_records` using one of the provided scripts.
 - `create_pascal_tf_record.py`
 - Located in:
`models/research/object_detection/dataset_tools`

```
example_tf_record = {  
    'image/height': height,  
    'image/width': width,  
    'image/filename': filename,  
    'image/source_id': filename,  
    'image/encoded': encoded_image_data,  
    'image/format': image_format,  
    'image/object/bbox/xmin': featurexmin,  
    'image/object/bbox/xmax': featurexmax,  
    'image/object/bbox/ymin': featureymin,  
    'image/object/bbox/ymax': featureymax,  
    'image/object/class/text': classes_text,  
    'image/object/class/label': classes,  
}
```

Class Label Files (.pbtxt)



- The third file required by TFOD is the class label files.
- These files simply are a mapping to the id and name and also provide the number of classes in our training data.

```
item {  
  id: 1  
  name: 'Person'  
}
```

```
item {  
  id: 2  
  name: 'Dog'  
}
```

Using Pretrained Models

- Training Object Detection on a CPU is almost impossible, GPUs are an absolute necessity here and even then, they can take weeks if using a single powerful GPU.
- TensorFlow hosts several pretrained models on COCO and can be found here:
 - https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md
- Untar this file the models/research directory by:

```
wget http://storage.googleapis.com/download.tensorflow.org/models/object_detection/faster_rcnn_resnet101_coco_11_06_2017.tar.gz  
tar -xvf faster_rcnn_resnet101_coco_11_06_2017.tar.gz
```

What is COCO?



COCO is a large-scale object detection, segmentation, and captioning dataset. COCO has several features:

- ✓ Object segmentation
- ✓ Recognition in context
- ✓ Superpixel stuff segmentation
- ✓ 330K images (>200K labeled)
- ✓ 1.5 million object instances
- ✓ 80 object categories
- ✓ 91 stuff categories
- ✓ 5 captions per image
- ✓ 250,000 people with keypoints

COCO-trained models

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_0.75_depth_coco ⋆	26	18	Boxes
ssd_mobilenet_v1_quantized_coco ⋆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ⋆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ⋆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ⋆	56	32	Boxes
ssd_resnet_50_fpn_coco ⋆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes
mask_rcnn_inception_resnet_v2_atrous_coco	771	36	Masks
mask_rcnn_inception_v2_coco	79	25	Masks
mask_rcnn_resnet101_atrous_coco	470	33	Masks
mask_rcnn_resnet50_atrous_coco	343	29	Masks





Configuring The Object Detection Pipeline

- The object detection pipeline configuration file is composed of 5 sections:
 - model
 - train_config
 - train_input_reader
 - eval_config
 - eval_input_reader

```
model {  
(... Add model config here...)  
}  
  
train_config : {  
(... Add train_config here...)  
}  
  
train_input_reader: {  
(... Add train_input configuration here...)  
}  
  
eval_config: {  
}  
  
eval_input_reader: {  
(... Add eval_input configuration here...)  
}
```

Model

- Example used from the "Faster R-CNN with Resnet-101 (v1), configured for Pascal VOC Dataset." file.
- We do not need to re-write this file, simply edit the one belonging to the pretrained model that you will be using.
- The mode defines all the necessary R-CNN or SSD parameters.
- When using pre-trained model it is best to leave this config file unchanged (only update num_classes if training a different number of objects)

```
model {
  faster_rcnn {
    num_classes: 20
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1024
      }
    }
    feature_extractor {
      type: 'faster_rcnn_resnet101'
      first_stage_features_stride: 16
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        scales: [0.25, 0.5, 1.0, 2.0]
        aspect_ratios: [0.5, 1.0, 2.0]
        height_stride: 16
        width_stride: 16
      }
    }
    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.01
        }
      }
    }
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.7
    first_stage_max_proposals: 300
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
    initial_crop_size: 14
    maxpool_kernel_size: 2
    maxpool_stride: 2
    second_stage_box_predictor {
      mask_rcnn_box_predictor {
        use_dropout: false
        dropout_keep_probability: 1.0
        fc_hyperparams {
          op: FC
          regularizer {
            l2_regularizer {
              weight: 0.0
            }
          }
          initializer {
            variance_scaling_initializer {
              factor: 1.0
              uniform: true
              mode: FAN_AVG
            }
          }
        }
      }
      second_stage_post_processing {
        batch_non_max_suppression {
          score_threshold: 0.0
          iou_threshold: 0.6
          max_detections_per_class: 100
          max_total_detections: 300
        }
        score_converter: SOFTMAX
      }
      second_stage_localization_loss_weight: 2.0
      second_stage_classification_loss_weight: 1.0
    }
  }
}
```





Train_Input_Reader, Eval_Config and Eval_Input_Reader

```
train_input_reader: {  
    tf_record_input_reader {  
        input_path: "/home/deeplearningcv/models/models/research/data/data/pascal_train.record"  
    }  
    label_map_path: "/home/deeplearningcv/models/models/research/data/data/pascal_label_map.pbtxt"  
}  
  
eval_config: {  
    num_examples: 4952  
}  
  
eval_input_reader: {  
    tf_record_input_reader {  
        input_path: "/home/deeplearningcv/models/models/research/data/data/pascal_val.record"  
    }  
    label_map_path: "/home/deeplearningcv/models/models/research/data/data/pascal_label_map.pbtxt"  
    shuffle: false  
    num_readers: 1  
}
```

- The train_input_reader and eval_input_reader are simply directory mappings (you will need to update these) that point to our training and validation tf_record files. As well as, our label mapping .pbtxt file
- The eval_config simply sets the number of examples, this does not need to be changed.



Directory Structure

- TFOD recommends the following structure for local training.

```
+data
  -label_map file #(*.pbtxt)
  -train TFRecord file #(*.record)
  -eval TFRecord file #(*.record)
+models
  + model
    -pipeline config file #faster_rcnn_resnet101_pets.config
    +train
    +eval
```



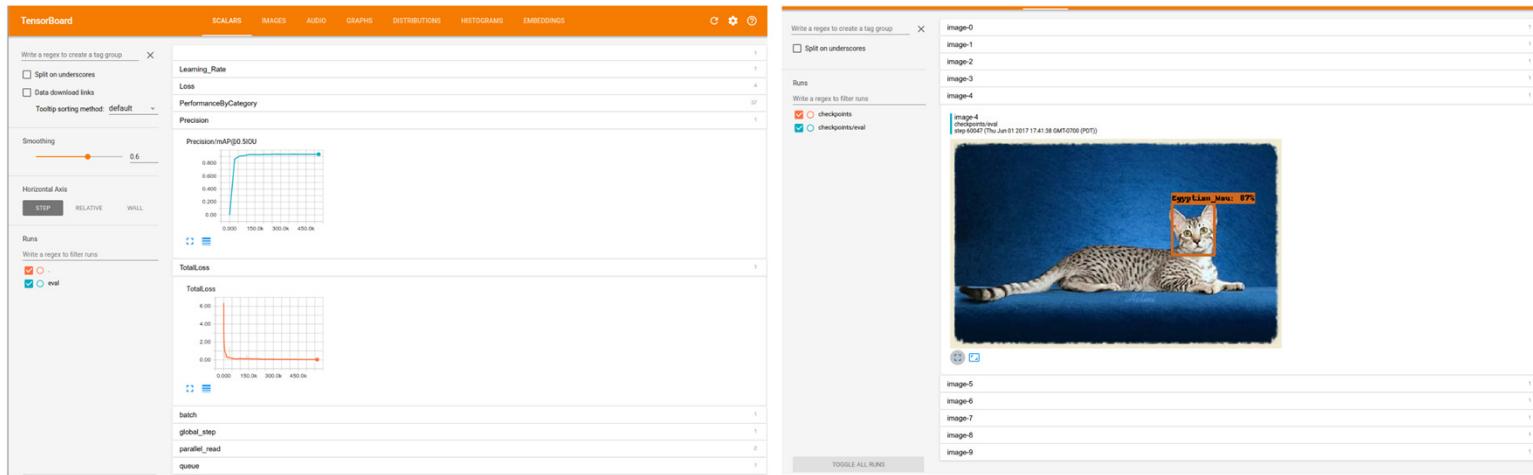
Starting the Training Process

```
PIPELINE_CONFIG_PATH=/home/deeplearningcv/models/models/research/data/models/model/faster_rcnn_resnet101_voc07.config  
MODEL_DIR=/home/deeplearningcv/models/models/research/data/data/  
NUM_TRAIN_STEPS=50000  
SAMPLE_1_OF_N_EVAL_EXAMPLES=1  
python object_detection/model_main.py \  
--pipeline_config_path=${PIPELINE_CONFIG_PATH} \  
--model_dir=${MODEL_DIR} \  
--num_train_steps=${NUM_TRAIN_STEPS} \  
--sample_1_of_n_eval_examples=${SAMPLE_1_OF_N_EVAL_EXAMPLES} \  
--alsologtostderr
```

- Launch the training process by executing the above instructions via terminal.
- Note the custom paths used in red (you will need to change these to suit your directory structure).

Monitoring on TensorBoard

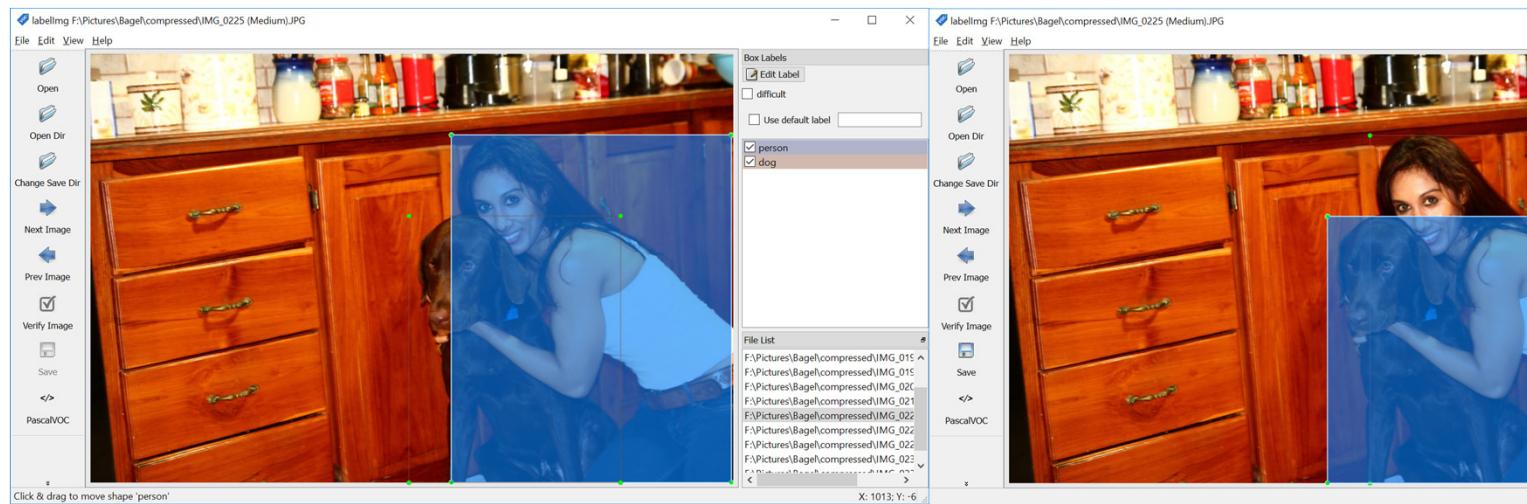
```
tensorboard --logdir=${/home/deeplearningcv/models/models/research/data/data/}
```





Creating your own Annotations using LabelImg

- Download LabelImg – It is super easy to use and can save files directly in the PascalVOC XML format which can be easily converted to TF_Records
 - <https://github.com/tzutalin/labelImg>



The PascalVOC XML Format

```
<annotation>
  <folder>compressed</folder>
  <filename>IMG_0225 (Medium).JPG</filename>
  <path>F:\Pictures\Bagel\compressed\IMG_0225 (Medium).JPG</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>1152</width>
    <height>768</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>person</name>
    <pose>Unspecified</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>646</xmin>
      <ymin>145</ymin>
      <xmax>1148</xmax>
      <ymax>768</ymax>
    </bndbox>
  </object>
  <object>
    <name>dog</name>
    <pose>Unspecified</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>570</xmin>
      <ymin>290</ymin>
      <xmax>950</xmax>
      <ymax>768</ymax>
    </bndbox>
  </object>
</annotation>
```





TensorFlow Object Detection API Summary

- We didn't do a full project here for the following reasons:
 - Training even a simple SSD or Faster R-CNN on a CPU is horrendously slow and impractical
 - You will need to use a GPU or Cloud GPU for effective training
 - TFOD datasets, models require a few gigs of storage.
 - Setting up GPUs is often a tricky task
- However, I have outlined the general steps required to train a TFOD model. For some very good tutorials on this please check out these blogs:
 - <https://towardsdatascience.com/how-to-train-your-own-object-detector-with-tensorflows-object-detector-api-bec72ecfe1d9>
 - <https://medium.com/@WuStangDan/step-by-step-tensorflow-object-detection-api-tutorial-part-1-selecting-a-model-a02b6aabe39e>

22.0

Object Detection with YOLO v3 and DarkFlow



Object Detection with YOLO v3 and DarkFlow

- **22.1 Setting up and install Yolo DarkNet and DarkFlow**
- **22.2 Experiment with YOLO on still images, webcam and videos**
- **22.3 Build your own YOLO Object Detector - Detecting London Underground Signs**

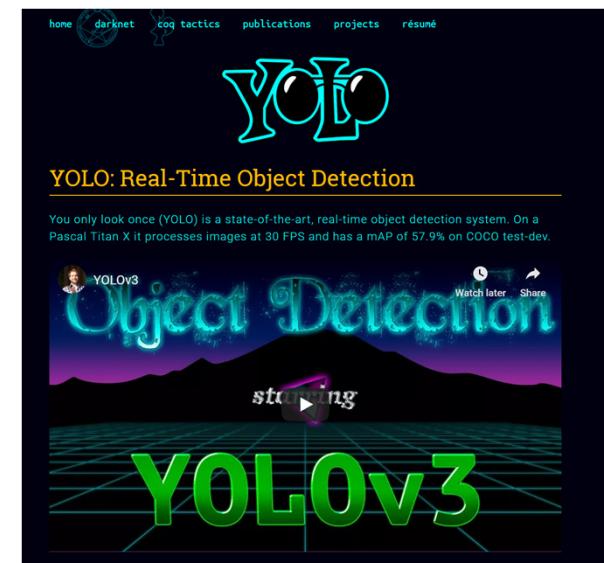
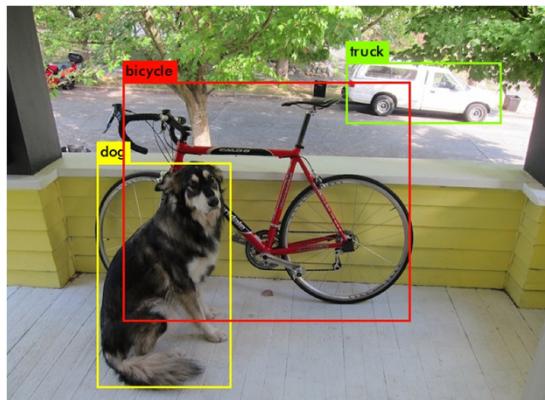
22.1

**Setting up and install YOLO DarkNet and
DarkFlow**



YOLO – You Only Look Once

- Using YOLO Is fairly simple to use and setup.
- Let's follow the instructions online at the official YOLO site to get our first exposure to YOLO.
 - <https://pjreddie.com/darknet/yolo/>

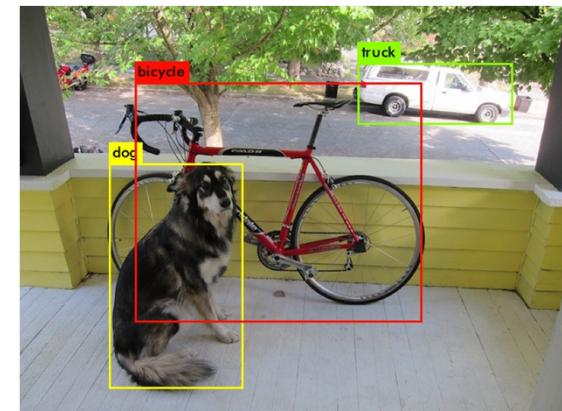




Installing DarkNet

- What is DarkNet?
- It's the official name for the YOLO framework (these guys are kind of awesome, read their papers, it's extremely entertaining)
 - <https://pjreddie.com/publications/>

```
cd  
mkdir darknet  
git clone https://github.com/pjreddie/darknet  
cd darknet  
make  
  
wget https://pjreddie.com/media/files/yolov3.weights  
./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```



- In your darknet directory, you'll see a new file created called "predictions.jpg"

```

deeplearningcv@deeplearningcv-VirtualBox:~/darknet$ ./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
layer         type        output
  0 conv      32 3 x 3 / 1   608 x 608 x 3    -> 608 x 608 x 32  0.639 BFLOPs
  1 conv      64 3 x 3 / 2   608 x 608 x 32    -> 304 x 304 x 64  3.407 BFLOPs
  2 conv      32 1 x 1 / 1   304 x 304 x 64    -> 304 x 304 x 32  0.379 BFLOPs
  3 conv      64 3 x 3 / 1   304 x 304 x 32    -> 304 x 304 x 64  3.407 BFLOPs
  4 res      1             304 x 304 x 64    -> 304 x 304 x 64
  5 conv      128 3 x 3 / 2  304 x 304 x 64    -> 152 x 152 x 128  3.407 BFLOPs
  6 conv      64 1 x 1 / 1   152 x 152 x 128   -> 152 x 152 x 64  0.379 BFLOPs
  7 conv      128 3 x 3 / 1  152 x 152 x 64    -> 152 x 152 x 128  3.407 BFLOPs
  8 res      5             152 x 152 x 128   -> 152 x 152 x 64  0.379 BFLOPs
  9 conv      64 1 x 1 / 1   152 x 152 x 64    -> 152 x 152 x 128  3.407 BFLOPs
 10 conv     128 3 x 3 / 1  152 x 152 x 64    -> 152 x 152 x 128  3.407 BFLOPs
 11 res      8             152 x 152 x 128   -> 152 x 152 x 128
 12 conv     256 3 x 3 / 2  152 x 152 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 13 conv     128 1 x 1 / 1   76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 14 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 15 res      12            76 x 76 x 256   -> 76 x 76 x 256
 16 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 17 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 18 res      15            76 x 76 x 256   -> 76 x 76 x 256
 19 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 20 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 21 res      16            76 x 76 x 256   -> 76 x 76 x 256
 22 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 23 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 24 res      21            76 x 76 x 256   -> 76 x 76 x 256
 25 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 26 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 27 res      24            76 x 76 x 256   -> 76 x 76 x 256
 28 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 29 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 30 res      27            76 x 76 x 256   -> 76 x 76 x 256
 31 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 32 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 33 res      30            76 x 76 x 256   -> 76 x 76 x 256
 34 conv     128 1 x 1 / 1  76 x 76 x 256   -> 76 x 76 x 128  0.379 BFLOPs
 35 conv     256 3 x 3 / 1  76 x 76 x 128   -> 76 x 76 x 256  3.407 BFLOPs
 36 res      33            76 x 76 x 256   -> 76 x 76 x 256
 37 conv     512 3 x 3 / 2  76 x 76 x 256   -> 38 x 38 x 512  3.407 BFLOPs
 38 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 39 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 40 res      37            38 x 38 x 512  -> 38 x 38 x 512
 41 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 42 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 43 res      40            38 x 38 x 512  -> 38 x 38 x 512
 44 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 45 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 46 res      43            38 x 38 x 512  -> 38 x 38 x 512
 47 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 48 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 49 res      46            38 x 38 x 512  -> 38 x 38 x 512
 50 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 51 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 52 res      49            38 x 38 x 512  -> 38 x 38 x 512
 53 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 54 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 55 res      52            38 x 38 x 512  -> 38 x 38 x 512
 56 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 57 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 58 res      55            38 x 38 x 512  -> 38 x 38 x 512
 59 conv     256 1 x 1 / 1  38 x 38 x 512  -> 38 x 38 x 256  0.379 BFLOPs
 60 conv     512 3 x 3 / 1  38 x 38 x 256  -> 38 x 38 x 512  3.407 BFLOPs
 61 res      58            38 x 38 x 512  -> 38 x 38 x 512
 62 conv     1024 3 x 3 / 2  38 x 38 x 512  -> 19 x 19 x 1024  3.407 BFLOPs
 63 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 64 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 65 res      62            19 x 19 x 1024  -> 19 x 19 x 1024
 66 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 67 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 68 res      65            19 x 19 x 1024  -> 19 x 19 x 1024
 69 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 70 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 71 res      68            19 x 19 x 1024  -> 19 x 19 x 1024
 72 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 73 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 74 res      71            19 x 19 x 1024  -> 19 x 19 x 1024
 75 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 76 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 77 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 78 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 79 conv     512 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 512  0.379 BFLOPs
 80 conv     1024 3 x 3 / 1  19 x 19 x 512   -> 19 x 19 x 1024  3.407 BFLOPs
 81 conv     255 1 x 1 / 1   19 x 19 x 1024  -> 19 x 19 x 255  0.189 BFLOPs
 82 yolo
 83 route    79
 84 conv     256 1 x 1 / 1  19 x 19 x 512   -> 19 x 19 x 256  0.095 BFLOPs
 85 upsample 2x           19 x 19 x 256   -> 38 x 38 x 256

```

What it looks like after executing the
*“./darknet detect cfg/yolov3.cfg
yolov3.weights data/dog.jpg”*
commands



*It takes ~ 10 seconds to load the
model and then runs it through its
object detection model.*

```

Loading weights from yolov3.weights...Done!
data/dog.jpg: Predicted in 40.946581 seconds.
dog: 100%
truck: 92%
bicycle: 99%

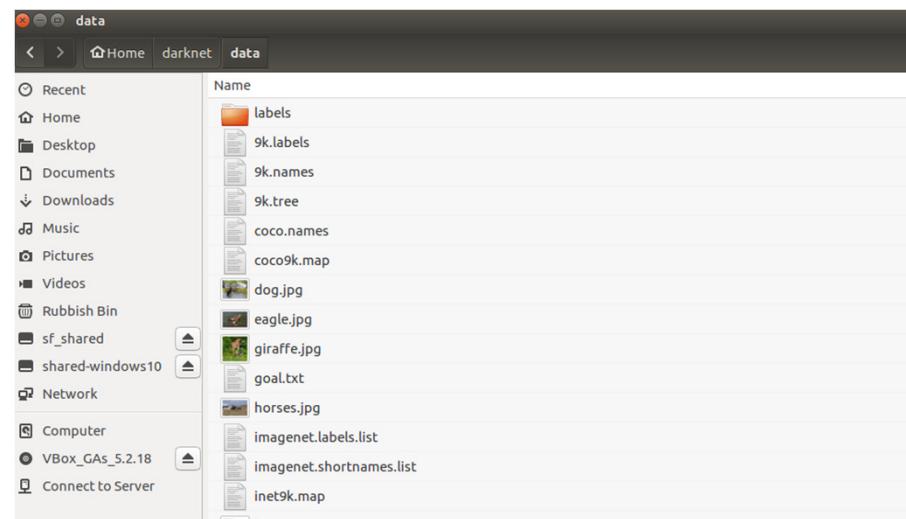
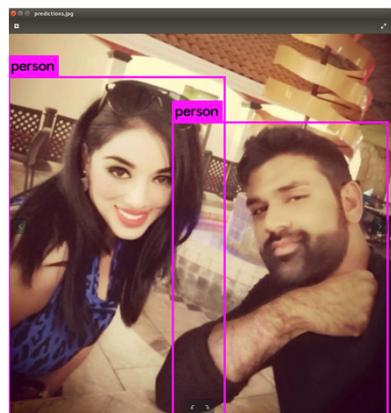
```

*If completed successfully, it displays
that it predicted successfully in X
seconds and then lists the
probabilities of image belong to that
class for each bounding box found.*



You can use any image for testing YOLO

- Keeping the directory structure as is, just place your test images into this directory:
 - darknet/data/





That was fun, we just used YOLO from the command line 😊

- You can do a lot from the command line in YOLO, even initiate training.
- But what if we wanted to use it inside of Python?
- Well we can with DarkFlow!

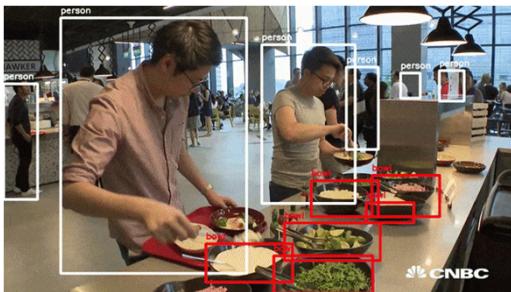
22.2

**DarkFlow – Using YOLO in Python for
images, videos & webcam**



DarkFlow Install – Using the TFODAPI Environment or creating a new one

- DarkFlow is a Python implementation of the YOLO2 Object Detection system using TensorFlow as the backend.
- You can re-use the “*tfodapi*” environment we created earlier. In case you haven’t, you can clone the existing “cv” environment or create a brand new environment (using python



anaconda-navigator

```
## - create environment with 3.6 and name it YOLO (I used my tfod_api env_
pip install --upgrade pip
pip install tensorflow
pip install keras
conda install scikit-learn
pip install PIL
conda install -c conda-forge opencv
python -m pip install -U matplotlib
pip install pandas
pip install tqdm
pip install pydot
pip install scikit-image
ip install graphviz
https://packages.ubuntu.com/search?keywords=graphviz&searchon=names
go to anaconda navigator and change environment to deeplearning and
install jupyter
```



DarkFlow Install

- Run the following instructions in terminal to install DarkFlow

```
cd  
mkdir darkflow  
cd darkflow  
pip install Cython  
git clone https://github.com/thtrieu/darkflow.git  
cd darkflow  
pip install .  
wget https://pjreddie.com/media/files/yolov2.weights  
cd  
cd darkflow/cfg  
ls
```



Now let's mess around with YOLO in Python!

```
from darkflow.net.build import TFNet
```

22.3

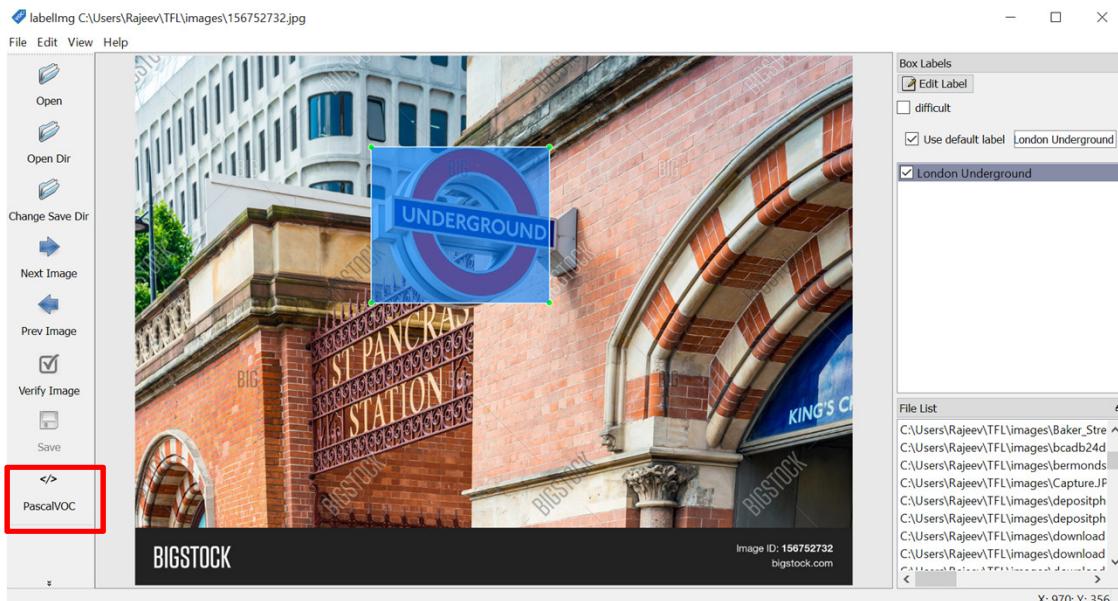
**Build your own YOLO Object Detector - Detecting
London Underground Signs**



Making a Custom Object Detector in YOLO

- Again, doing this without a GPU is going to be difficult. However, we can actually make a very crappy (overfits!) custom object detector.
- Because I'm teaching the course from my VirtualBox machine that cannot access GPUs, I'll be doing this exercise using a CPU (just like you).
- NOTE: To train this using a GPU doesn't require any different setup except a few more commands, but you will need to do this from a non-virtual machine install

Step 1 – Making a Dataset using LabelImg



- Save files using PascalVOC format

- Install LabelImg
<https://github.com/tzutalin/labelImg>
- Open a folder containing images. I've supplied you with just over 100 images where there's a London Underground Sign.
- We're making a London Underground Sign Detector 😊



Step 1 – Things to Note

Things I learnt the hard way.

1. Ensure your images are labeled numerically e.g. 0001.jpg....0010.jpg....0100.jpg BEFORE doing any labeling with LabelImg.
2. When using LabelImg, save files as Pasal VOC format. Older YOLO used the "YOLO" format, but unfortunately (or fortunately) YOLO now ingests the ".XML" files used in VOC.

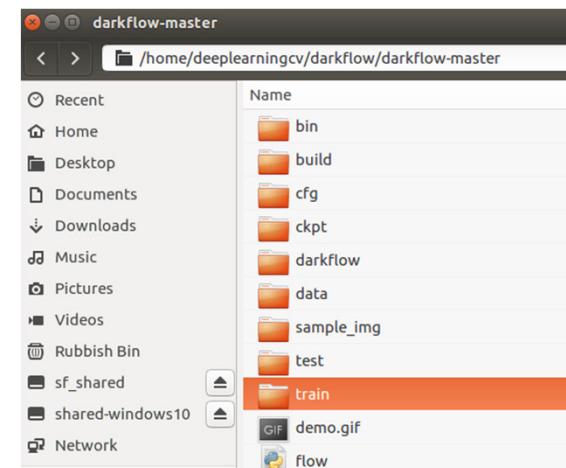
Step 1 – Things to Note



Things I learnt the hard way continued...

3. Setup your file structure directories before labeling images. That ensures LabelImg puts the correct information in the XML file. (It isn't the end of the world if you do it in a different directory, but you will need to write some scripts to update your XML files)

+ darkflow
+ darkflow-master
+ train
+ images
+ annotations





Step 1 – Things to Note, continued...

- 3. XML files labels need to be "correct"

```
<annotation>
  <folder>annotations</folder>
  <filename>0002.jpg</filename>
  <path>/home/deeplearningcv/darkflow/darkflow-master/train/images/0002.jpg</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>259</width>
    <height>194</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>London Underground</name>
    <pose>Unspecified</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>24</xmin>
      <ymin>14</ymin>
      <xmax>229</xmax>
      <ymax>194</ymax>
    </bndbox>
  </object>
</annotation>
```



Step 2 – Create a Configuration File

- Go to the “darkflow-master” and in the “cfg” directory, find the file “yolo.cfg”. Copy this file and rename the copy “yolo_1_class.cfg” assuming you’re using one class.





Step 3 – Edit your Configuration File

```
[net]
# Testing
batch=1
subdivisions=1
# Training
batch=64
subdivisions=64
width=288
height=288
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

You will need to make a few changes to your config file.

- Open your config file in a text editor (*I used gedit*)
- Firstly at the top of file, you'll need to change the **batch size** to **64** and subdivisions to **64** (they will be commented out)
- Change **height** and **width** to **288** and **288** for faster training



Step 3 – Edit your Configuration File

```
[convolutional]
size=1
stride=1
pad=1
filters=30
activation=linear

[region]
anchors = 0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47
bias_match=1
classes=1
coords=4
num=5
softmax=1
jitter=.3
rescore=1

object_scale=5
noobject_scale=1
class_scale=1
coord_scale=1

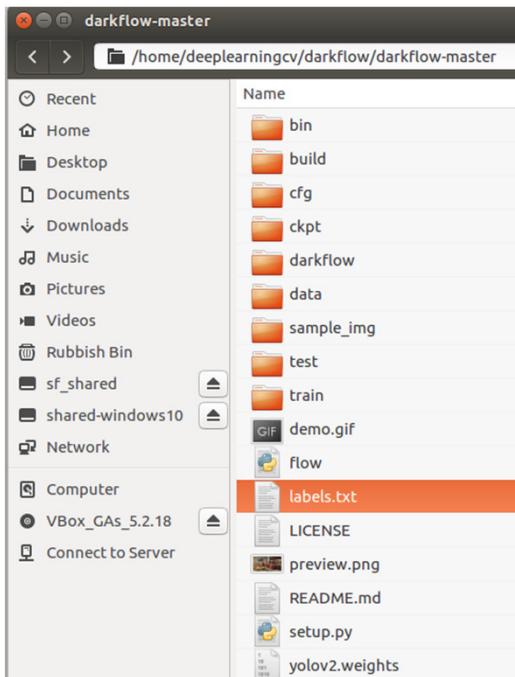
absolute=1
thresh = .1
random=1
```

At the bottom of this file change the following:

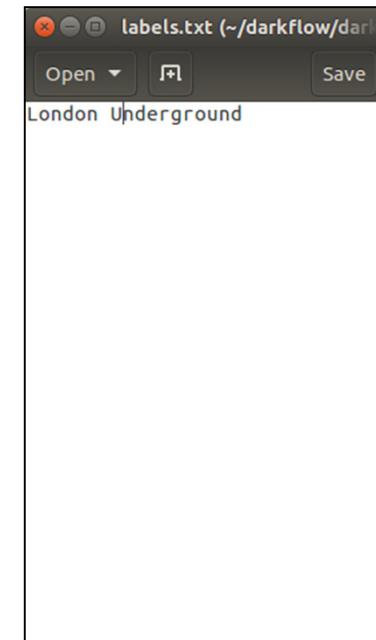
- The last convolution layer has a certain number of filters that is dependent on the number of classes in our custom dataset.
 - *Filters = 5 (number of classes + 5)*
 - With 1 class filters = $5 \times (1 + 6) = 30$
 - With 3 classes filters = $5 \times (3 + 6) = 45$
- Number of Classes needs to be updated to number of classes in your dataset



Step 4 – Edit or Create a labels.txt



- Located in the main folder **darkflow-master** folder, open the file (create one if it isn't there for some reason) called **labels.txt**.
- This file contains a list of our label names e.g.
 - Cat
 - Dog
 - Donald Trump
- Or in our case, it's called "London Underground"





Step 5 – Training Time

- To initial training go to your Terminal
- In the darkflow/darkflow-master directory, execute the following line:



```
[root@deeplearningcv deeplearningcv-VirtualBox:~/darkflow/darkflow-master$ python flow --model cfg/yolo_1_class.cfg --load yolov2.weights --train --annotation train/annotations --dataset train/images --epoch 500
Parsing .cfg/yolo2.cfg
Parsing cfg/yolo_1_class.cfg
Loading yolov2.weights ...
Successfully identified 203934260 bytes
Finished in 0.016780138015747075

Building net ...
Source | Train? | Layer description | Output size
-----+-----+-----+-----+
Load | Yep! | input | (7, 288, 288, 3)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 288, 288, 32)
Load | Yep! | maxp 2x2p0_2 | (7, 144, 144, 32)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 144, 144, 64)
Load | Yep! | maxp 2x2p0_2 | (7, 72, 72, 64)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 72, 72, 64)
Load | Yep! | conv 1x1p0_1 +bnorm leaky | (7, 72, 72, 64)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 72, 72, 128)
Load | Yep! | conv 2x2p0_2 | (7, 36, 36, 128)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 36, 36, 256)
Load | Yep! | conv 1x1p0_1 +bnorm leaky | (7, 36, 36, 128)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 36, 36, 256)
Load | Yep! | maxp 2x2p0_2 | (7, 18, 18, 256)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 18, 18, 512)
Load | Yep! | conv 2x2p0_1 +bnorm leaky | (7, 9, 9, 512)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 18, 18, 512)
Load | Yep! | conv 1x1p0_1 +bnorm leaky | (7, 18, 18, 256)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 18, 18, 512)
Load | Yep! | maxp 2x2p0_2 | (7, 9, 9, 512)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 9, 9, 1024)
Load | Yep! | conv 1x1p0_1 +bnorm leaky | (7, 9, 9, 512)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 9, 9, 1024)
Load | Yep! | conv 1x1p0_1 +bnorm leaky | (7, 9, 9, 512)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 9, 9, 1024)
Load | Yep! | concat [16] | (7, 18, 18, 512)
Load | Yep! | conv 1x1p0_1 +bnorm leaky | (7, 18, 18, 64)
Load | Yep! | local flatten 2x2 | (7, 9, 9, 256)
Load | Yep! | concat [27, 24] | (7, 9, 9, 1280)
Load | Yep! | conv 3x3p1_1 +bnorm leaky | (7, 9, 9, 1024)
Load | Yep! | conv 1x1p0_1 llinear | (7, 9, 9, 30)

Running entirely on CPU
cfg/yolo_1_class.cfg loss hyper-parameters:
H      = 9
W      = 9
box    = 5
classes = 5
scales = [1.0, 2.0, 1.0, 1.0]
Building cfg/yolo_1_class.cfg loss
building cfg/yolo_1-class.cfg train op
2018-11-03 01:19:42,539717: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
Finished in 14.941243648529053s

Enter training ...
cfg/yolo_1_class.cfg parsing train/annotations
Parsign for ['London Underground']
=====>[100% 0001.xml
Statistics:
London Underground: 6
Dataset size: 4
Dataset of 4 instance(s)
Training statistics:
    Learning rate : 1e-05
    Batch size   : 4
    Epoch number : 500
    Batch every   : 2000
step 1 - loss 49.62251281738281 - moving ave loss 49.62251281738281
Finish 1 epoch(es)
step 2 - loss 49.24868392944336 - moving ave loss 49.58512992858887
Finish 2 epoch(es)
step 3 - loss 48.95505142211914 - moving ave loss 49.5221220779419
Finish 3 epoch(es)
step 4 - loss 48.65061569213867 - moving ave loss 49.434971439361576
Finish 4 epoch(es)
step 5 - loss 47.97720184326172 - moving ave loss 49.34919447975159
Finish 5 epoch(es)
step 6 - loss 48.27556228637695 - moving ave loss 49.24183126041413
Finish 6 epoch(es)
step 7 - loss 48.05352028263672 - moving ave loss 49.12300615463639
Finish 7 epoch(es)
step 8 - loss 47.97898483276367 - moving ave loss 49.00859862244912
Finish 8 epoch(es)
...
```



Step 5 – Training Time

- To initial training go to your Terminal
- In the darkflow/darkflow-master directory, execute the following line:

```
python flow --model cfg/yolo_1_class.cfg --load yolov2.weights --train --annotation train/annotations --dataset train/images --epoch 500
```

- If using a GPU enter the following additional arguments:

```
python flow --model cfg/yolo_1_class.cfg --load yolov2.weights --train --annotation train/annotations --dataset train/images --gpu 0.7 --epoch 500
```



Training Completed!

```
Finish 496 epoch(es)
step 497 - loss 1.8804212808609009 - moving ave loss 2.0546289510115403
Finish 497 epoch(es)
step 498 - loss 2.0536890029907227 - moving ave loss 2.0545349562094586
Finish 498 epoch(es)
step 499 - loss 1.778516173362732 - moving ave loss 2.0269330779247863
Finish 499 epoch(es)
step 500 - loss 1.7606858015060425 - moving ave loss 2.000308350282912
Checkpoint at step 500
Finish 500 epoch(es)
Training finished, exit.
(tfodapi) deeplearningcv@deeplearningcv-VirtualBox:~/darkflow/darkflow-master$
```

Test Your Model!



- Firstly, where is your model?
- It's in the `ckpt` directory

The screenshot shows a file explorer window titled "ckpt" with the path "/home/deeplearningcv/darkflow/darkflow-master/ckpt" in the address bar. The left sidebar lists recent locations and network drives. The main pane displays a table of files in the "ckpt" directory, including meta, index, and data files for YOLO models across three versions (500, 100, and 5).

	Name	Size	Type	Modified
Recent	yolo_1_class-500.meta	203.6 MB	Binary	01:56
Home	yolo_1_class-500.index	9.5 kB	Binary	01:56
Desktop	yolo_1_class-500.data-00000-of-00001	606.8 MB	Binary	01:56
Documents	checkpoint	89 bytes	Text	01:56
Downloads	yolo_1_class-500.profile	31.4 kB	Binary	01:55
Music	yolo_1c-100.meta	203.6 MB	Binary	00:39
Pictures	yolo_1c-100.index	9.5 kB	Binary	00:39
Videos	yolo_1c-100.data-00000-of-00001	606.8 MB	Binary	00:39
Rubbish Bin	yolo_1c-100.profile	5.8 kB	Binary	00:39
sf_shared	yolo_1c-5.meta	203.6 MB	Binary	00:23
shared-windows10	yolo_1c-5.index	9.5 kB	Binary	00:23
Network	yolo_1c-5.data-00000-of-00001	606.8 MB	Binary	00:23
Computer	yolo_1c-5.profile	358 bytes	Binary	00:23
VBox_GAs_5.2.18				
Connect to Server				



Loading Our Model in Python

- Firstly, where is your model? It's in the `ckpt` directory, but just need to point our load option to the number of steps we used in training our model (it will point to the last trained checkpoint)

Loading our Trained Custom Dataset Model

```
from darkflow.net.build import TFNet
import cv2
import tensorflow as tf

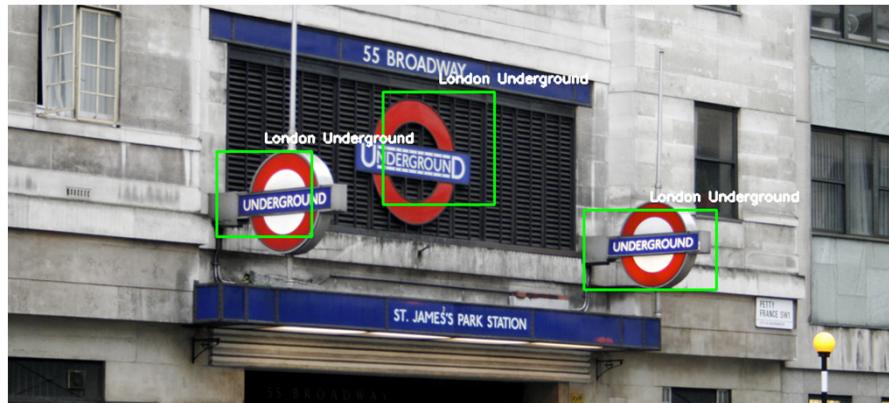
# Config TF, set True if using GPU
config = tf.ConfigProto(log_device_placement = False)
config.gpu_options.allow_growth = False

with tf.Session(config=config) as sess:
    options = {
        'model': './cfg/yolo_1_class.cfg',
        'load': 500, #This is # of steps/epochs used in training, it tells it load the last saved model
        'threshold': 0.5,
        #'gpu': 1.0 # uncomment these if using GPU
    }
    tfnet = TFNet(options)
```



Does it Work?

Sort of, but not bad for using just 5 training images and under an hour of training time on CPU!



23.0

DeepDream and Neural Style Transfers



Deep Dream and Neural Style Transfers

- **23.1 DeepDream – How AI Generated Art All Started**
- **23.2 Neural Style Transfer**

23.1

DeepDream – How AI Generated Art All Started



DeepDream or DeepDreaming

- What if we ran a pre-trained CNN in reverse?



<https://deeplab4j.org/>

573

AI Generated Art and Content

- Before Google's DeepDream was announced, very few, even in the AI world, thought AI (in the near future) would be able to produce art and content indistinguishable from human creations.
- While, we aren't quite there yet, Google's DeepDream was the first to really show, AI can generate exciting and enjoyable content.

DeepDream

- Is an **artistic image-modification** technique that uses representations learned by CNNs to produce hallucinogenic or 'trippy' images. It was introduced by C. Olah A. Mordvintsev and M. Tyka of Google Research in 2015 in publication titled "*DeepDream: A Code Example for Visualizing Neural Networks*"
- It became extremely popular since being announced and has produced many iconic DeepDream art.





DeepDream Algorithm

- Remember how we created our filter visualizations?
- We ran our **CNN** in **reverse**, doing **gradient ascent** on the **input** to **maximize the activation** of specific **filters** in the **top** layers of a **pre-trained CNN**
- Well DeepDream follows a similar methodology with some variations.
- We first attempt to **maximize the activation of entire layers** instead of one specific filter. This allows us to mix several visualizations into one.
- We start with an **existing image** (instead of a blank or noisy input). This allows the effects of the CONV filters to work their magic by distorting elements in the image that produce pleasing visualizations.
- The input images are processed at **different scales** (also called **octaves**) to improve the quality of the visualization

23.2

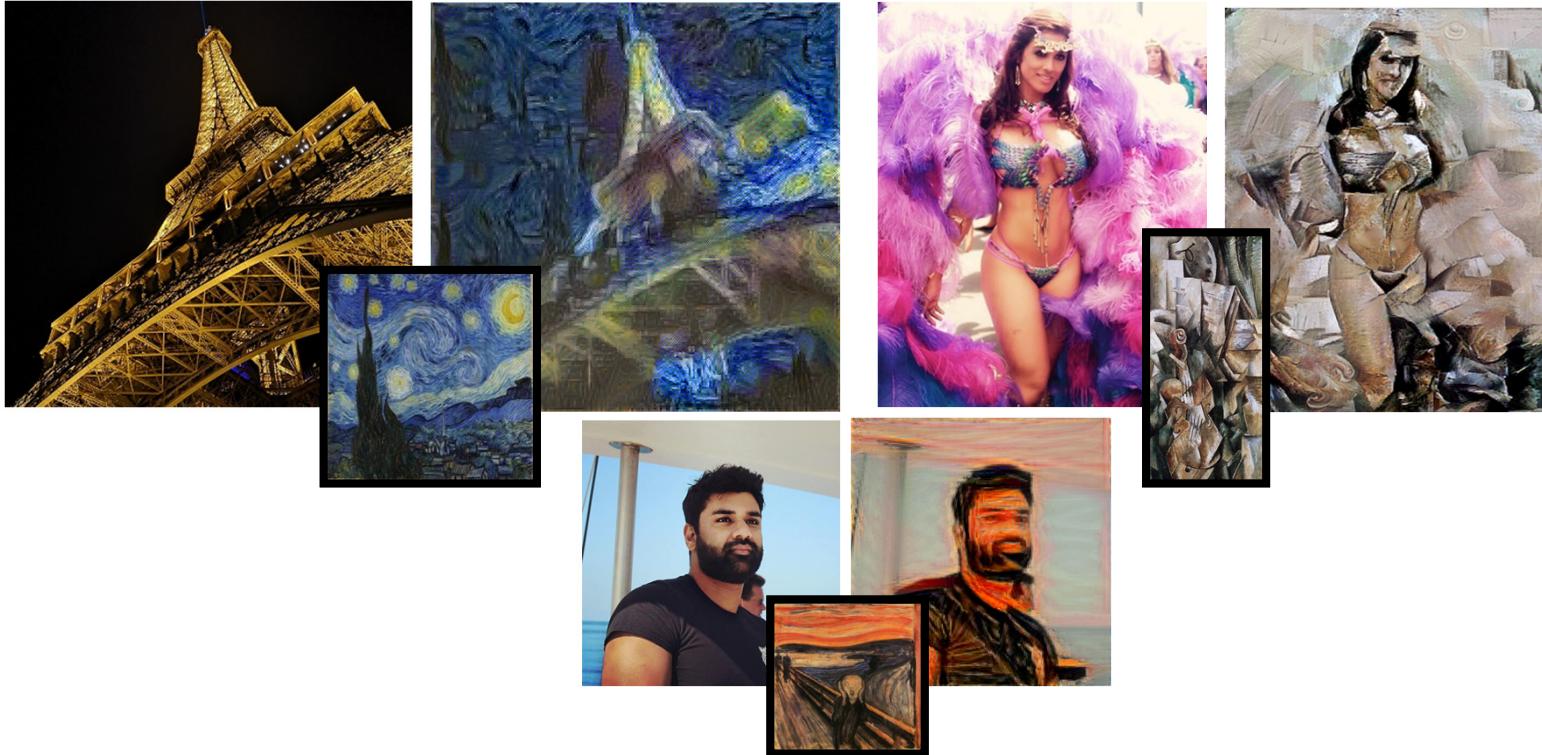
Neural Style Transfer



Neural Style Transfer

- Introduced by Leon Gatys et al. in 2015, in their paper titled "[A Neural Algorithm for Artistic Style](#)", the Neural Style Transfer algorithm went viral resulting in an explosion of further work and mobile apps.
- Neural Style Transfer enables the **artistic style** of an image to be applied to another image! It copies the color patterns, combinations and brush strokes of the original source image and applies it to your input image.
- And is one the most impressive implementations of Neural Networks in my opinion.

Neural Style Transfer Examples



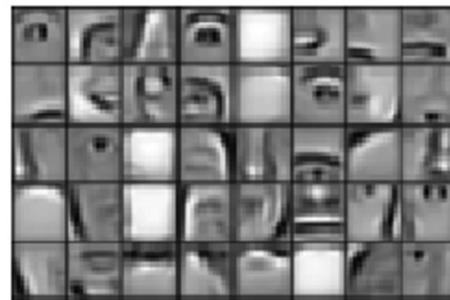
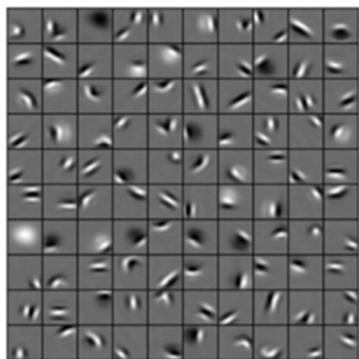


What enables Neural Style Transfer to Perform this Magic?

- Neural Style Transfers are enabled not by having a fancy exotic Architecture
- Or by a lengthy training process.
- It's all about special Loss Functions.



Remember When We Visualized Our Filters?

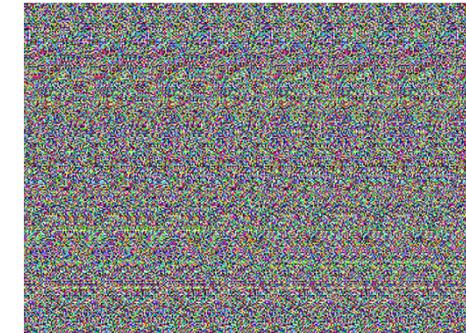


- We saw lower layers (on the left) look for simple patterns, while higher layers (right) look for complex patterns.
- So it's established that CONV layers stores a hierarchical representation of image features.
- This forms a key component of Neural Style Transfer.



The Process

1. First we start with a blank image of random pixel values.
2. We keep changing pixel values as to optimize a **Cost Function** (keeping our pre-trained CNN weights constant).
 - Remember we want to preserve the content of the original image while adopting the style of the reference image.





The Cost Function

The Cost Function we just described consists of two parts:

1. A Content Loss
2. A Style Loss



Content Loss

- Remember how our filters learned special decompositions of an image (low and high level features).
- We can assume that the high level features from the upper CONV layers store a global representation of the image.
- Therefore, our Content Loss can be defined as the L2 Norm between the activations in the upper layers of a pretrained CNN computed versus the target image and the activations of same layer computed over the generated image.
- This ensures our generated image retains a similar look to the original images.



Style Loss

- The Content Loss uses a single upper layer however, the style layer is composed of multiple layers of the CNN. Why?
- Because now we're trying to capture the styles from the reference image at all spatial scales extracted by the CNN.
- For the style loss, Gatys et al. use the *Gram matrix* of a layer's activations: the inner product of the feature maps of a given layer.
- The inner product can be interpreted as representing a map of the correlations between the layer's features.
- The style loss aims to preserve similar internal correlations within the activations of different layers, across the style reference image and the generated image. This ensures our textures between images look similar



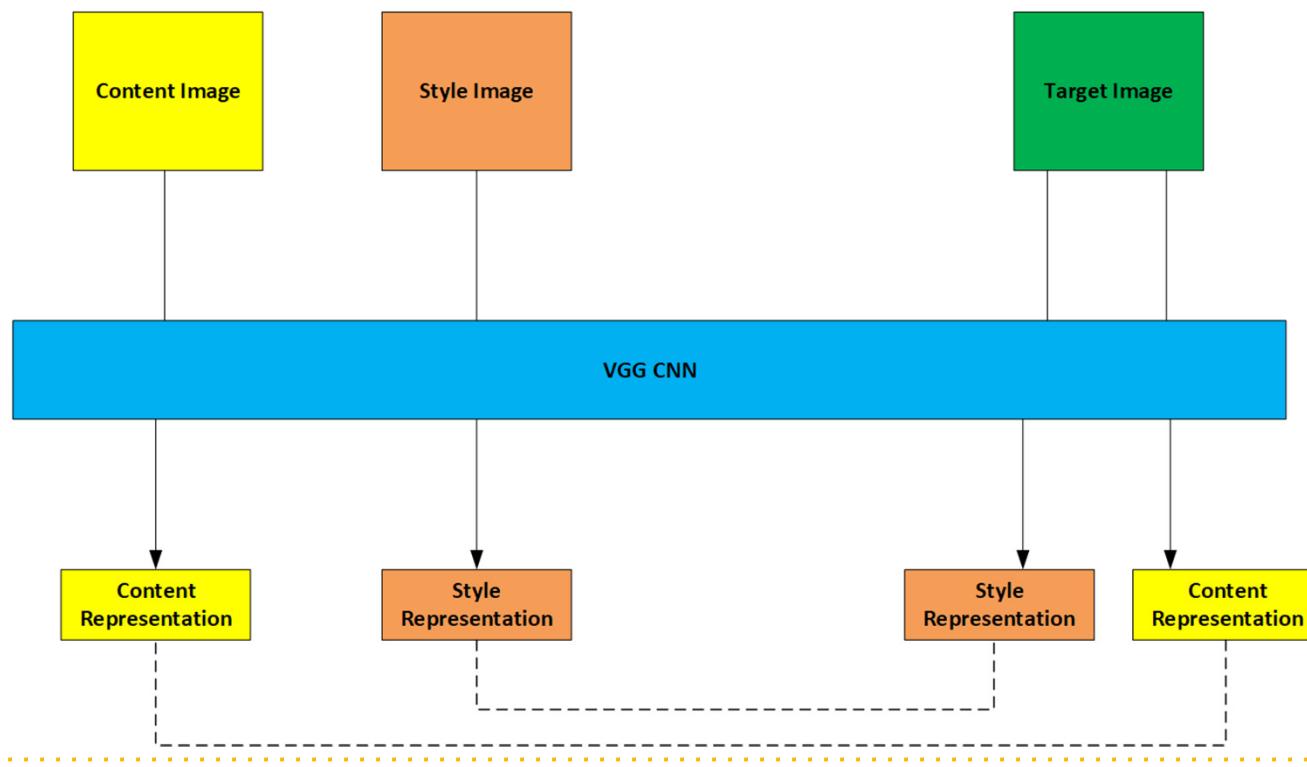
Minimizing our Total Loss

- So now we have our Content Loss and Style Loss, we can now use any optimizer to perform gradient descent to generate the new image.

Total Loss

$$\begin{aligned} &= \text{Dist}(\text{style}(\text{ref_img}) - \text{style}(\text{gen_img})) \\ &\quad + \text{Dist}(\text{content}(\text{orig_img}) - \text{content}(\text{gen_img})) \end{aligned}$$

$$\text{Total Loss} = \alpha Loss_{Content} + \beta Loss_{Style}$$



24.0

Introduction to Generative Adversarial Networks (GANs)



Learning Summary for Generative Adversarial Networks

- **24.1 – Introduction To GANs**
 - Lead up to GANs and examples
 - High Level Intuitive Explanation of GANs
 - How GANs are Trained
- **24.2 – Mathematics of GANs**
 - The Math of GANs made simple & the Algorithm
- **24.3 – Implementing GANs in Keras**
 - Generating Digits from MNIST Dataset
- **24.4 – Face Aging GAN**
 - Take your pretty young face and make you look 60+

**THIS IS GOING TO BE
GANTASTIC!**

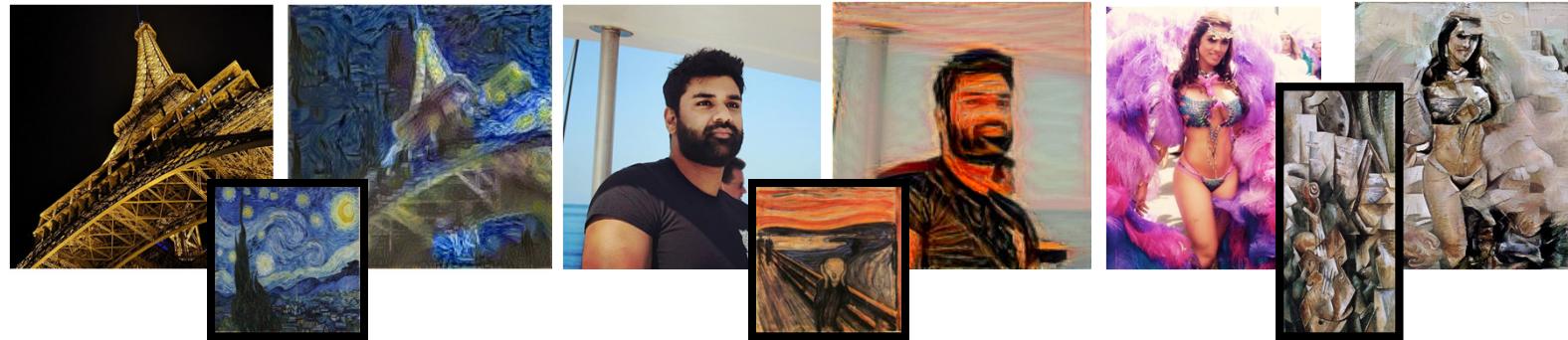
24.1

Introduction to Generative Adversarial Networks (GANs)



Recap on Image Creation with Neural Networks

- So far we've used pre-trained CNN's (VGG, InceptionV3 and ResNet50) to generate **DeepDream** Images and **Neural Style Transfers**.





What Else Is Possible?

- Remember when applying **DeepDream** or **Neural Style Transfer**, we needed an input image to manipulate.
- What if, our Network could '***think up***' and **create** its own image.





The Brilliance of GANs

- **GANs** allow us generate images created by our Neural Networks, completely removing a human (yes you) out of the loop.
- Allowing it to generate beautiful images like this!

Like this!





Examples of GANs

- Before we dive into the theory, I like showing you the abilities of GANs to build your excitement. **Turn Horses into Zebras** (vice versa)



<https://gananath.github.io/drugai-gan.html>



Generate Anime Characters



<https://arxiv.org/pdf/1708.05509.pdf>



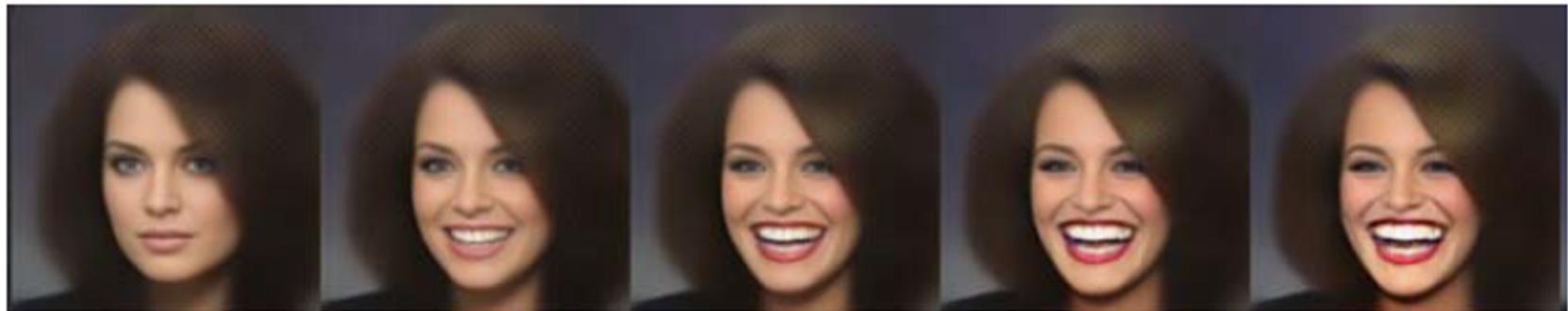
Generate Fake Celebrity Photos



https://research.nvidia.com/publication/2017-10_Progressive-Growing-of



Creating a Smiling Face



<https://arxiv.org/abs/1705.09368>



Super Resolution (SRGAN)

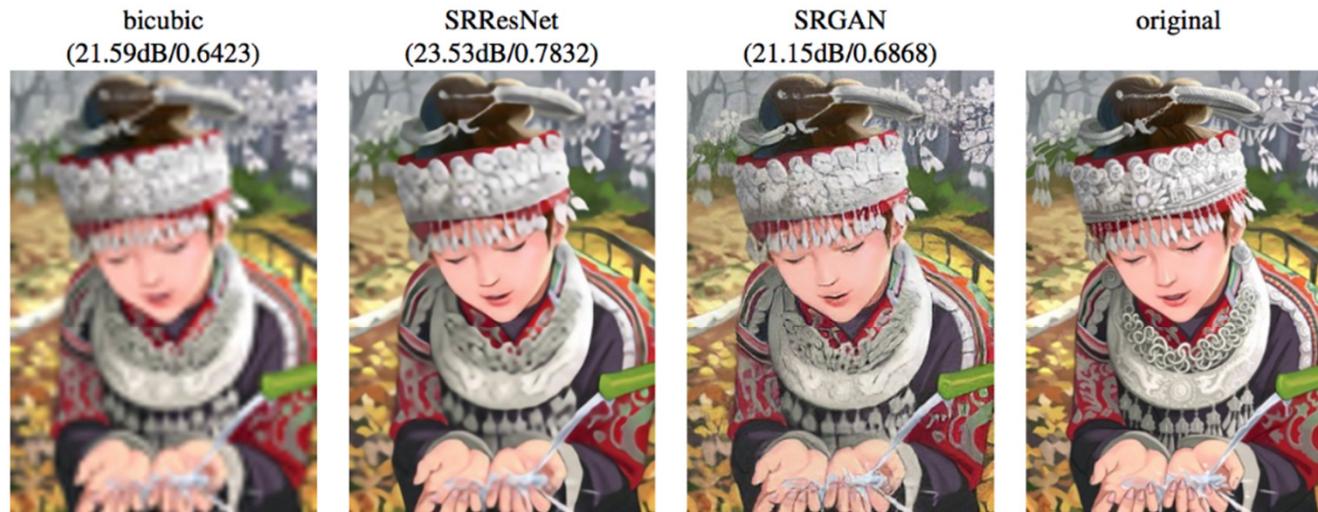
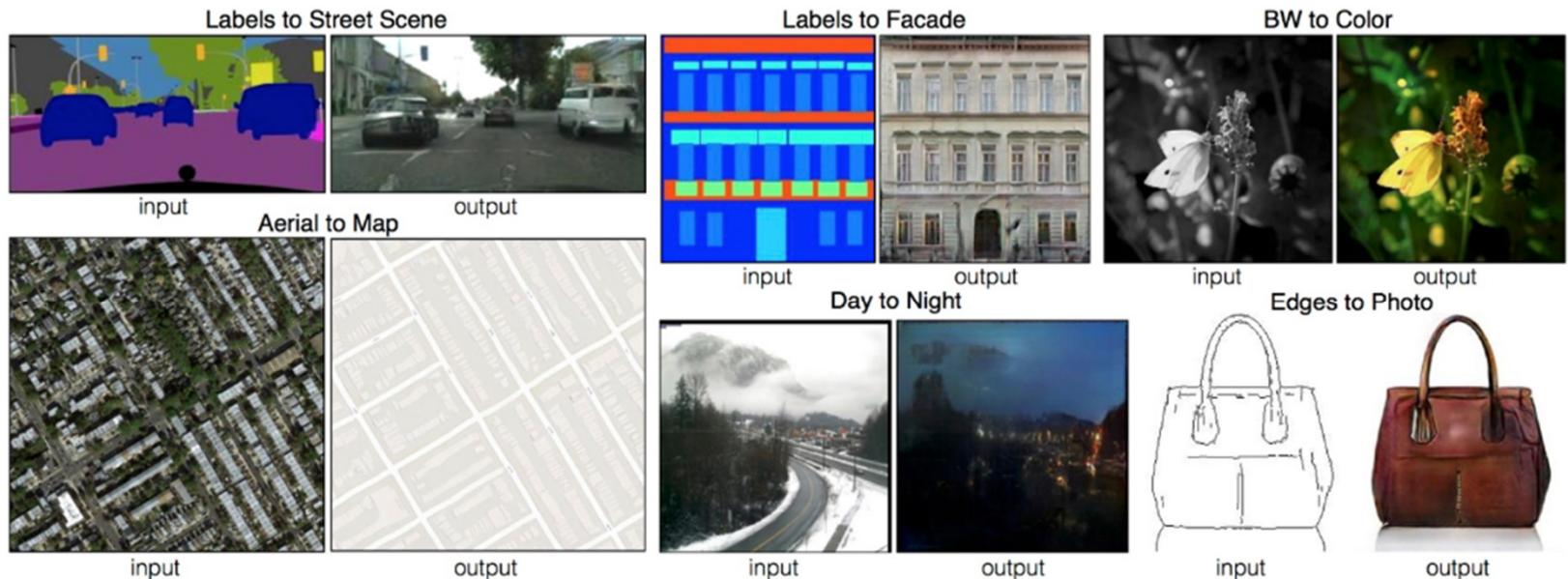


Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

<https://arxiv.org/pdf/1609.04802.pdf>



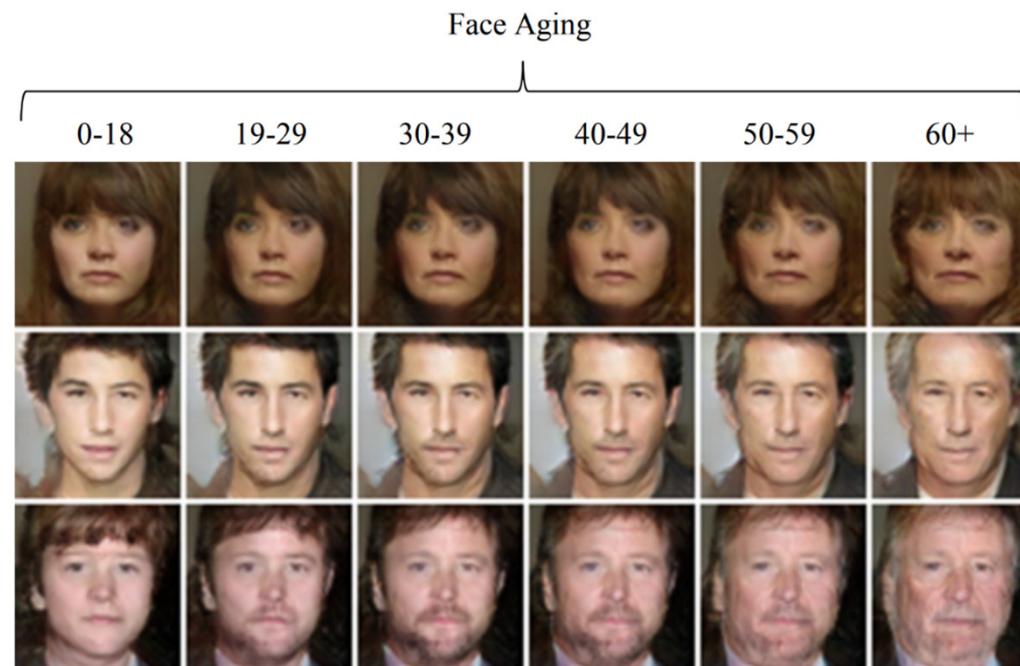
Pix2Pix



<https://arxiv.org/pdf/1611.07004.pdf>



Face aging (Age-cGAN)



<https://arxiv.org/pdf/1702.01983.pdf>

600



History of GANs

- Generative adversarial networks (GANs) was introduced by Ian Goodfellow (the **GANFather** of GANs) et al. in 2014, in his paper appropriately titled "*Generative Adversarial Networks*"
- It was proposed as an alternative to Variational Auto Encoders (VAEs) which learn the latent spaces of images, to generate synthetic images.
- It's aimed to create realistic artificial images that could be almost indistinguishable from real ones.



Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie*, Mehdi Mirza, Bing Xu, David Warde-Farley,
Sherjil Ozair†, Aaron Courville, Yoshua Bengio‡

Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

<https://arxiv.org/abs/1406.2661>

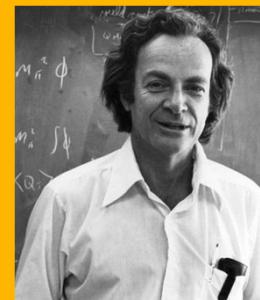
Abstract

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G . The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D , a unique solution exists, with G recovering the training data distribution and D equal to $\frac{1}{2}$ everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation. There is no need for any Markov chains or unrolled approximate inference networks during either training or generation of samples. Experiments demonstrate the potential of the framework through qualitative and quantitative evaluation of the generated samples.

“

*What I cannot create,
I do not understand*

- Richard Feynman



The GAN Analogy or GANology

A high level intuitive explanation of GANs



The GAN High Level Explanation

- Imagine there's an ambitious young criminal who wants to counterfeit money and sell to a mobster who specializes in handling counterfeit money.



High Level GANs Explanation

- At first, the young counterfeiter is **not good** and our expert mobster tells him, he's money is way off from looking real.



I'll try
harder
next time!



Take this
back and
start over!

High Level GANs Explanation

- Slowly he gets better and makes a good 'copy' every so often. The mobster tells him when it's good.



High Level GANs Explanation

- After some time, both the forger (our counterfeiter) and expert mobster get better at their jobs and now they have created almost real looking but fake money.





The Generator & Discriminator Networks

- The purpose of the **Generator** Network is to take a random image initialization and decode it into a synthetic image.
- The purpose of the **Discriminator** Network is take this input and predict whether this image came from a real dataset or is synthetic.



GANs Components

- As we just saw, this is effectively what GANs are, **two** antagonistic networks that are contesting against each other. The two components are called:
 - 1. Generator Network** – in our example this was the young criminal creating counterfeit money
 - 2. Discriminator Network** – the mobster in our example.



Training GANs

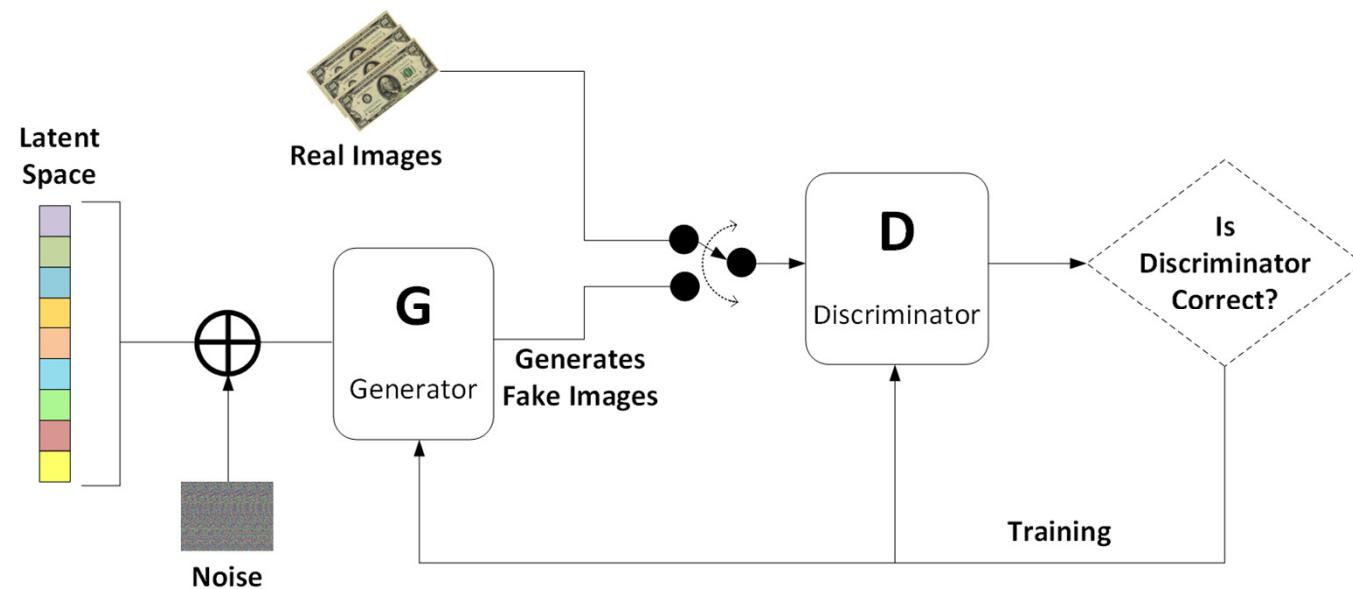
- Training GANs is notoriously difficult. Previously we used gradient descent to change our weights to reduce our loss.
- However, in a GANs, every weight change, changes the entire balance of our dynamic system.
- In GANs we are not seeking to minimize loss, but finding an equilibrium between our two opposing Networks.



The GAN Training Process

1. Input randomly generated noisy images into our Generator Network to generate a sample images.
2. We take some sample images from our real data and mix it with some of our generated images
3. Input these mixed images to our Discriminator who will then be trained on this mixed set and will update it's weights accordingly.
4. We then make some more fake images and input it into the Discriminator but we label all as real. This is done to train the Generator. We've frozen the weights of the discriminator at this stage (Discriminator learning stops), and we use the feedback from the discriminator to now update the weights of the generator. This is how we teach both our Generator (to make better synthetic images) and Discriminator to get better at spotting fakes.

GAN Block Diagram



Summary

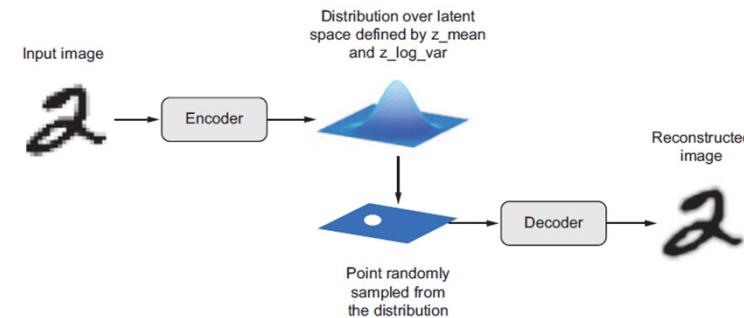
- We understand what are GANs and their capabilities
- How GANs work intuitively
- How GANs are train the Generator and Discriminator

UP NEXT

- Learn a bit of the Mathematics involved in training GANs
- How to implement GANs in Keras

GANs vs. Variational Autoencoders (VAEs)

- Both VAEs and GANs use a module or Network to map low dimension image representations to realistic looking images. In GANs, this is the generator and in VAEs this is called the decoder.
- However, VAEs learn this latent space representation differently to GANs. It works best when their latent space is well structured, where specific directions encode a meaningful axis of variation in the data.



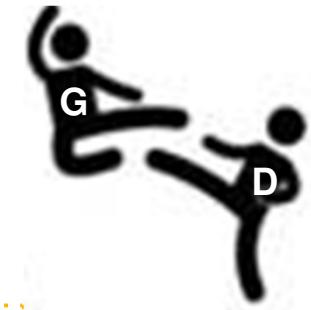
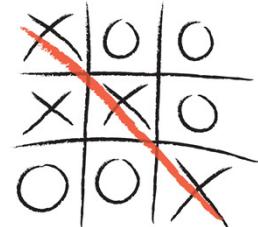
24.2

Math Behind GANs



Adversarial Games

1. Adversarial games is a subfield of Artificial Intelligence where two or more **agents** play games **opposing** each other in **zero-sum games** (their loss is my gain). Examples of zero-sum games are Tic-Tac-Toe.
2. In GANs we have two adversarial networks, the Generator and the Discriminator, where each network is being trained to win over the other. The Generator tries to make better fakes to beat the Discriminator while the Discriminator learns to get better at spotting fakes.





The Mathematics

- Let X be our true dataset and Z be the normal distributed noise.
- Let $p(z)$ be data from latent space Z .
- G and D are differentiable functions of generative network and discriminative network respectively.
- $D(x)$ represents probability that data come from real dataset X . We train D to maximize the probability $\log(D(x))$ and train G to minimize $\log(1 - D(G(z)))$.
- In short they play min max game as explained above with each other and obtain global optimality.



The Mathematics

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

- Above function serves as a loss function for our generative adversarial network. Now, it is important to note that $\log(1 - D(G(z)))$ saturates so we don't minimize it rather we maximize $\log(D(G(z)))$.
- In order to prove that sample generated by generator network is exactly the same as X we need to go to deeper in mathematics and use Kullback-Leibler divergence theorem and Jensen-Shannon divergence.

24.3

Using GANs to generate a Fake Handwritten Digits



Generating Fake Handwritten Digits

- In this experiment we're going to create a GAN that uses the MNIST dataset (as it's source of real data) and build a Generator and Discriminator Network that creates Fake Handwritten Digits!





Our Generator

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 6272)	633472
leaky_re_lu_13 (LeakyReLU)	(None, 6272)	0
reshape_4 (Reshape)	(None, 128, 7, 7)	0
up_sampling2d_7 (UpSampling2	(None, 128, 14, 14)	0
conv2d_13 (Conv2D)	(None, 64, 14, 14)	204864
leaky_re_lu_14 (LeakyReLU)	(None, 64, 14, 14)	0
up_sampling2d_8 (UpSampling2	(None, 64, 28, 28)	0
conv2d_14 (Conv2D)	(None, 1, 28, 28)	1601
<hr/>		
Total params:	839,937	
Trainable params:	839,937	
Non-trainable params:	0	

- Note our final output shape
- It's no longer a softmax that outputs class scores, but it's in the form of an MNIST Digit i.e. 28 x 28 x 1
- Also we Up Sample instead of Down sample to create an image.



Our Discriminator

None		
Layer (type)	Output Shape	Param #
conv2d_15 (Conv2D)	(None, 64, 14, 14)	1664
leaky_re_lu_15 (LeakyReLU)	(None, 64, 14, 14)	0
dropout_7 (Dropout)	(None, 64, 14, 14)	0
conv2d_16 (Conv2D)	(None, 128, 7, 7)	204928
leaky_re_lu_16 (LeakyReLU)	(None, 128, 7, 7)	0
dropout_8 (Dropout)	(None, 128, 7, 7)	0
flatten_4 (Flatten)	(None, 6272)	0
dense_8 (Dense)	(None, 1)	6273
=====		
Total params:	212,865	
Trainable params:	212,865	
Non-trainable params:	0	

- Note our final output shape of our Discriminator.
- It's a binary output of yes or no regarding if the input digit is fake.



Creating the Adversarial Network in Keras

- In Keras after creating those two models, we need to combine them in a way that creates the adversarial network. This is how it's done.

```
# Creating the Adversarial Network. We need to make the Discriminator weights
# non trainable. This only applies to the GAN model.
discriminator.trainable = False
ganInput = Input(shape=(latent_dim,))
x = generator(ganInput)
ganOutput = discriminator(x)
gan = Model(inputs=ganInput, outputs=ganOutput)
gan.compile(loss='binary_crossentropy', optimizer=adam)
```



The Results after 1 Epoch

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



The Results after 2 Epochs

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



The Results after 3 Epochs

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



The Results after 4 Epochs

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



The Results after 5 Epochs

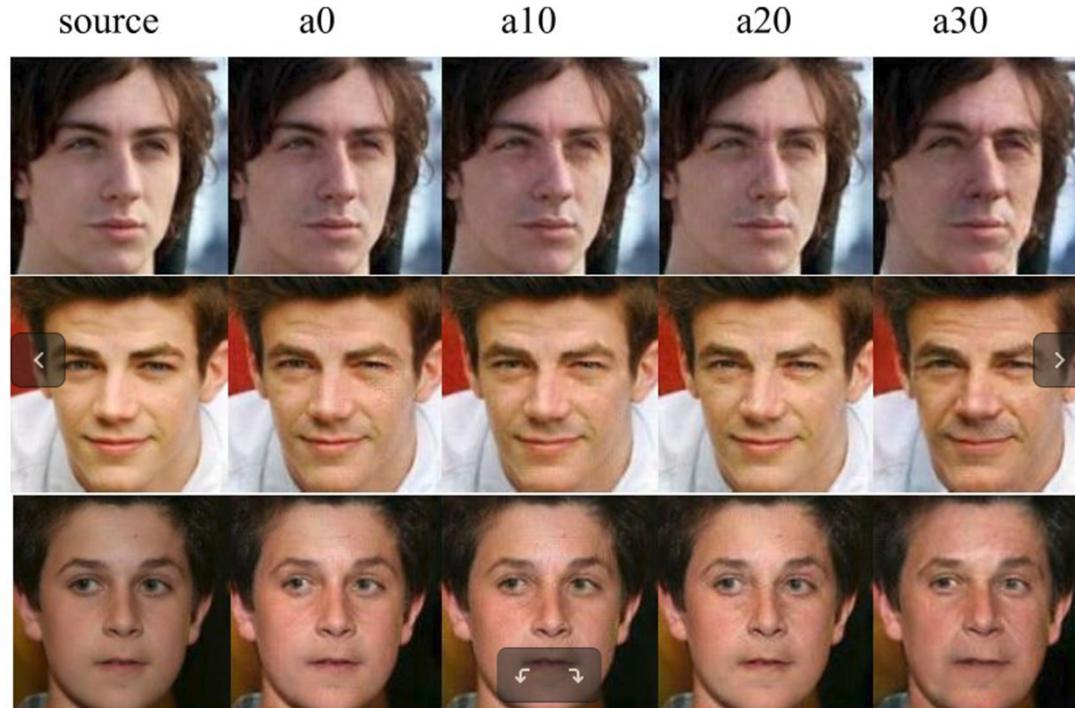
2	8	4	9	2	0	8	3	3	5
7	9	3	8	2	5	4	2	2	2
0	9	2	6	2	5	2	8	0	7
5	6	8	3	2	9	2	0	1	9

24.3

**Build a Face aging (Age-cGAN) GAN, aging faces up
to 60+ years**



Build a Face aging (Age-cGAN) GAN, aging faces up to 60+ years



631



Project Source

- This is an open source project taken from the 2018 paper published in CVPR
 - http://openaccess.thecvf.com/content_cvpr_2018/papers/Wang_Face_Aging_With_CVPR_2018_paper.pdf

Face Aging with Identity-Preserved Conditional Generative Adversarial Networks

Zongwei Wang
ShanghaiTech University
wangzw@shanghaitech.edu.cn

Xu Tang
Baidu
tangxu02@baidu.com

Weixin Luo, Shenghua Gao*
ShanghaiTech University
{luowx, gaoshh}@shanghaitech.edu.cn

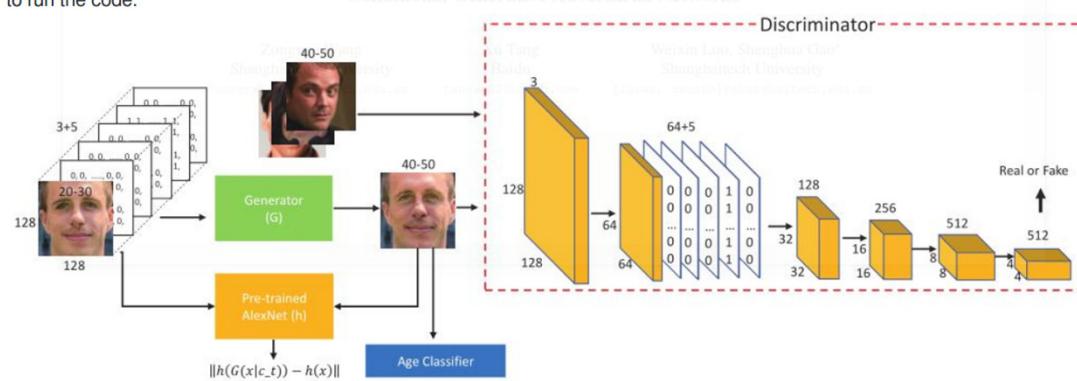


GitHub Repo

<https://github.com/dawei6875797/Face-Aging-with-Identity-Preserved-Conditional-Generative-Adversarial-Networks>

Face Aging with Identity-Preserved Conditional Generative Adversarial Networks

This repo is the official open source of Face Aging with Identity-Preserved Conditional Generative Adversarial Networks, CVPR 2018 by Zongwei Wang, Xu Tang, Weixin Luo and Shenghua Gao. It is implemented in tensorflow. Please follow the instructions to run the code.

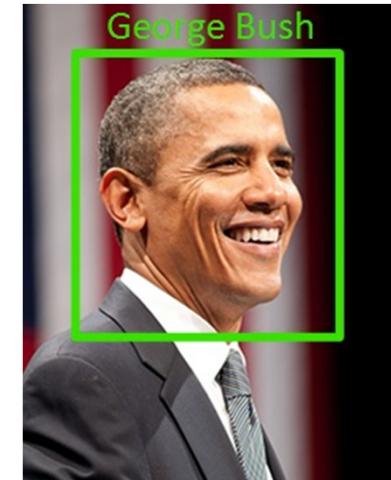


Face Recognition

What is Face Recognition



- Face Recognition is the ability to distinguish and recognize individual faces.
- Our brains are remarkably good at this task. Think of how many people you can instantly recognize – celebrities, politicians, your friends, family co-workers, complete strangers!
 - Read about our brains ability at face perception here - https://en.wikipedia.org/wiki/Face_perception
- This has always been a relatively difficult task for computer vision



Why is Face Recognition Hard?

- Typically our CNNs are trying to learn medium to high level features.
- Learning the very subtle differences between faces can be extremely challenging, even for a fairly deep CNN.
- Previously, 'shallow' methods were used which started by extracting a representation of the face image using handcrafted local image descriptors such as SIFT, LBP, HOG, then they aggregate such local descriptors into an overall face descriptor by using a pooling mechanism, for example the Fisher Vector etc.
- Even using the typical method of giving a deep CNN thousands of examples of faces from specific people do not work well.





Why Tradition CNN's Suck at Face Recognition?

- Examine the faces of the Actor for Joey Tribbiani (played by Matt LeBlanc).
- His head is facing several directions, the color in some of the images is different.
- Throwing these labeled images at a CNN, may have some success, but learning low level features or incorrect features such as a head facing down, tilts, hair styles, clothing color – may work well on training data.
- But as soon as we test a method like this on some unseen data, it inevitably screws up.



1061_1.jpg



108_0.jpg



1077_1.jpg



124_0.jpg



1261_0.jpg



1045_1.jpg



Introducing VGGFace

- VGGFace was developed by Oxford University Researchers and was compiled using over 2.6 million images of 2,600 different identities.
- <http://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf>
- http://www.robots.ox.ac.uk/~vgg/data/vgg_face/



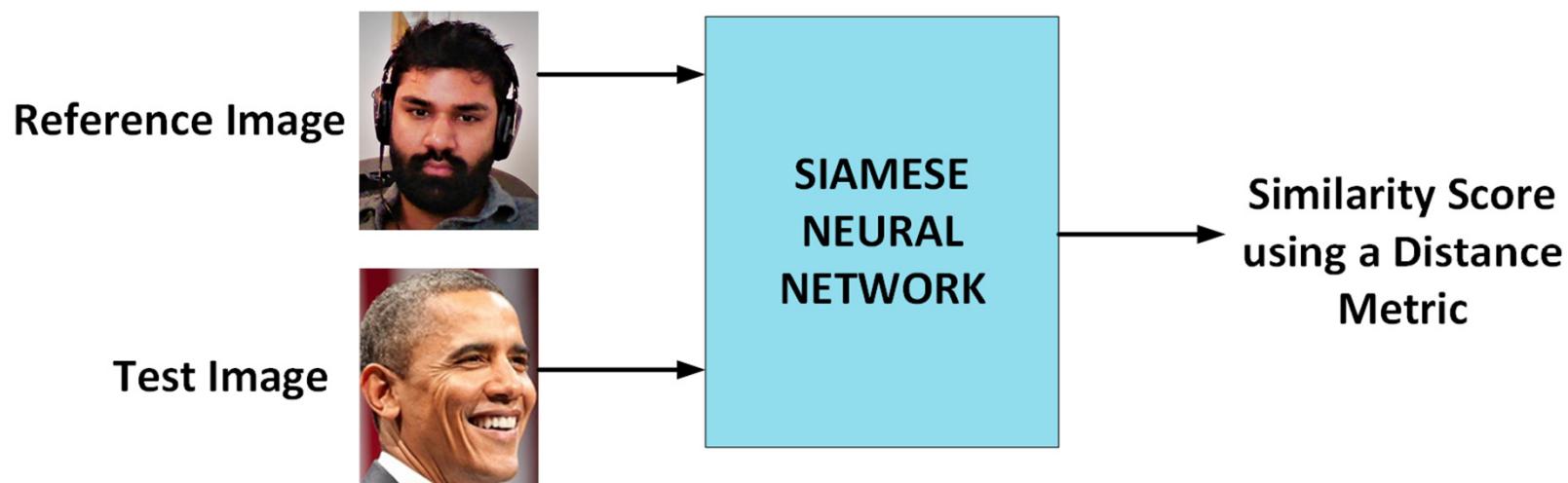


Introduction to Siamese Networks

- Now, we spoke about the issues of using traditional methods to create a Face Recognition classifier, so what did the Oxford Researchers do to overcome these? Using **Siamese Networks**
- What are Siamese Networks? This is where the same CNN is applied to pairs of faces to obtain descriptors that are then compared using the Euclidean distance.



Introduction to Siamese Networks



- Instead of taking an input image and outputting the class, Siamese networks uses the two images (shown above) and produces a similarity score. In this case, it will give a score of how likely the two face images are of the same person.



Usefulness of Siamese Networks in Facial Recognition

- Instead of using hundreds or thousands of images of a person to train the classifier (imagine we already have a dataset of thousands of people too) to train a new classifier. We are using a single image of this person to 'train'.
- This is called **One-Shot Learning** and is very useful for dynamic Facial Recognition tasks such as employee identification.
- Let's implement and test VGGFace using Keras and the pre-trained VGGFace weights.

25.0

The Computer Vision World



The Computer Vision World

- **25.1 – Alternative Frameworks: PyTorch, MXNet, Caffe, Theano & OpenVINO**
- **25.2 – Popular APIs Google, Microsoft, ClarifAI Amazon Rekognition and others**
- **25.3 – Popular Computer Vision Conferences & Finding Datasets**
- **25.4 – Building a Deep Learning Machine vs. Cloud GPUs**

25.1

Alternative Frameworks: PyTorch, MXNet, Caffe, Theano & OpenVINO



Keras and TensorFlow aren't the only players

- While Keras and TensorFlow are by far the most popular CV API/Libraries around, they aren't the ones competing in this space. We've got these major players.
 - PyTorch
 - MXNet
 - Caffe
 - Theano
 - OpenVINO

PyTorch

<https://pytorch.org>



- PyTorch was released in 2016 and was primarily developed by Facebook's AI Research group.
- It is very well supported and 2nd to Keras/TensorFlow in popularity
- It's a Python-First framework which means it's fully designed to be deeply integrated into existing Python libraries.
- More user-friendly than TensorFlow, but less flexible

MXNet

<https://mxnet.apache.org>



- Developed by the Apache Software Foundation, MXNet is another open-source deep learning framework.
- Its advantages is that it scales well to using multiple GPUs and is supported by cloud services such as AWS and Microsoft Azure.
- It was written in primarily C++ and has wrappers for Python.

Caffe

<http://caffe.berkeleyvision.org>

- Caffe was developed specifically for Deep Learning by the Berkeley Vision.
- It is bit difficult to use and thus primarily lent itself to researchers looking to build exotic models.
- Given it's base in Computer Vision a lot of the famous models were developed in Caffe first before being ported to TensorFlow and PyTorch.



Theano

theano



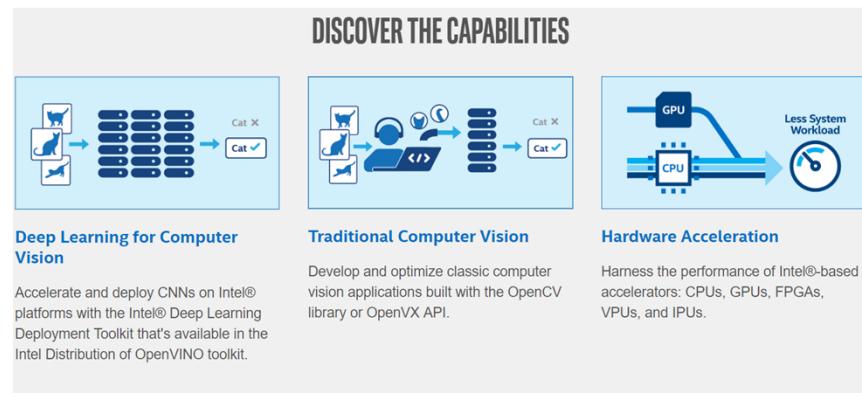
- Developed in 2007 by the Montreal Institute for Learning Algorithms (MILA), University of Montreal
- Deprecated officially on 2017 due to competing platforms.
- Keras supports Theano backend so all of our Keras code can also be run using Theano instead of Tensorflow.



OpenVINO

<https://software.intel.com/en-us/openvino-toolkit>

- 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



25.2

**Popular APIs, Google, Microsoft, Clarifai
Amazon Rekognition and others**



Computer Vision APIs

- If you didn't want to go through all the trouble of creating your own model, deploying it in production and ensuring it can scale properly with traffic. You can use some existing computer vision APIs. There are many very good ones that offer services such as:
 - Object detection
 - Face recognition
 - Image classification.
 - Image Annotation
 - Video Analysis
 - Text Recognition



Popular Computer Vision APIs

- Google's Cloud Vision - <https://cloud.google.com/vision>
- Microsoft's Vision API - <https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/>
- Amazon's Rekognition - <https://aws.amazon.com/rekognition/>
- Clarifai - <https://clarifai.com/>
- IBM's Watson Visual Recognition - <https://www.ibm.com/watson/services/visual-recognition/>
- Kairos (specialized in Face Recognition) - <https://www.kairos.com/>

25.3

Popular Computer Vision Conferences & Finding Datasets



The Top Computer Vision Conferences

- **CVPR** Conference on Computer Vision and Pattern Recognition - <http://cvpr2018.thecvf.com/>
- **NIPS** – Neural Information Processing - <https://nips.cc/>
- **ICCV** – Internation Conference on Computer Vision <http://iccv2017.thecvf.com/>
- **ECCV** – European Conference on Computer Vision - <https://eccv2018.org/>



Where to find Computer Vision Datasets

- By far the best place to find datasets is on Kaggle.
(www.kaggle.com)
- Other places to find good datasets are:
 - <https://hackernoon.com/rare-datasets-for-computer-vision-every-machine-learning-expert-must-work-with-2ddaf52ad862>
 - <https://computervisiononline.com/datasets>
 - <http://www.cvpapers.com/datasets.html>

25.4

Building a Deep Learning Machine vs. Cloud GPUs



Hardware Recommendations

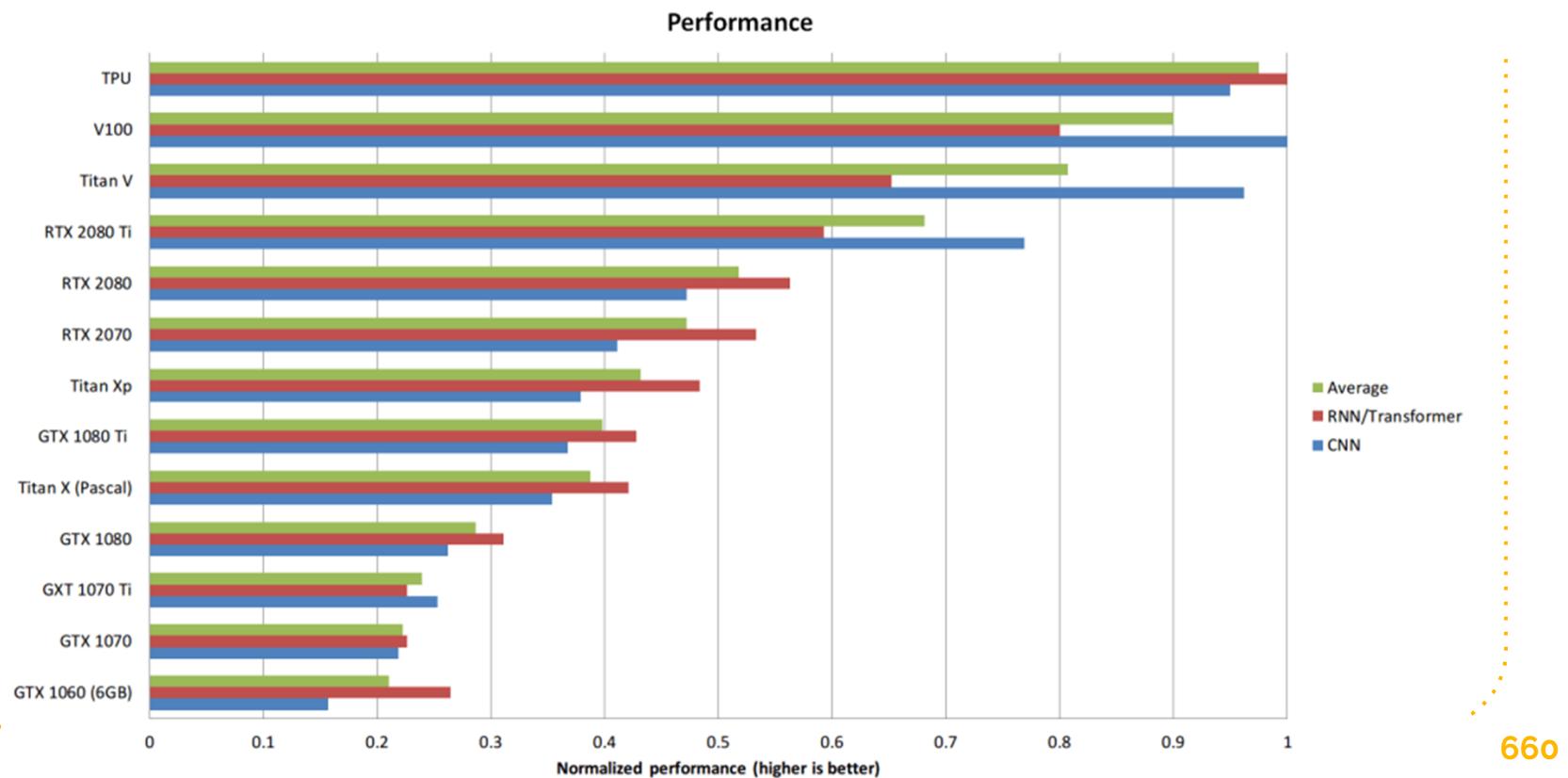
- It's all about the GPU, and you'll need an NVIDIA GPU since only NVIDIA supports CUDA which is used to speed up deep learning processing.
- As for Nov 2018, the best NVIDIA GPU you can use at home is the Titan V (\$3000 USD).
- Best consumer level (gamer) NVIDIA GPU was the TRX 2080 Ti (\$1200USD)
- Best value GPU to go with is the RTX 2070 (\$600 USD)
- Best budget NVidia GPU is the GTX 1050Ti (\$200 USD)
- Read more here - <http://timdettmers.com/2018/11/05/which-gpu-for-deep-learning/>



Other Hardware Recommendations

- At least 16gb of RAM
- SSD for faster data loading and transfer
- Large Hard drive for storing large image datasets
- Fast CPU at least quad core (AMD Ryzen line or Intel i5, i7 and i9s)

Overall GPU Performance



GPU performance per dollar

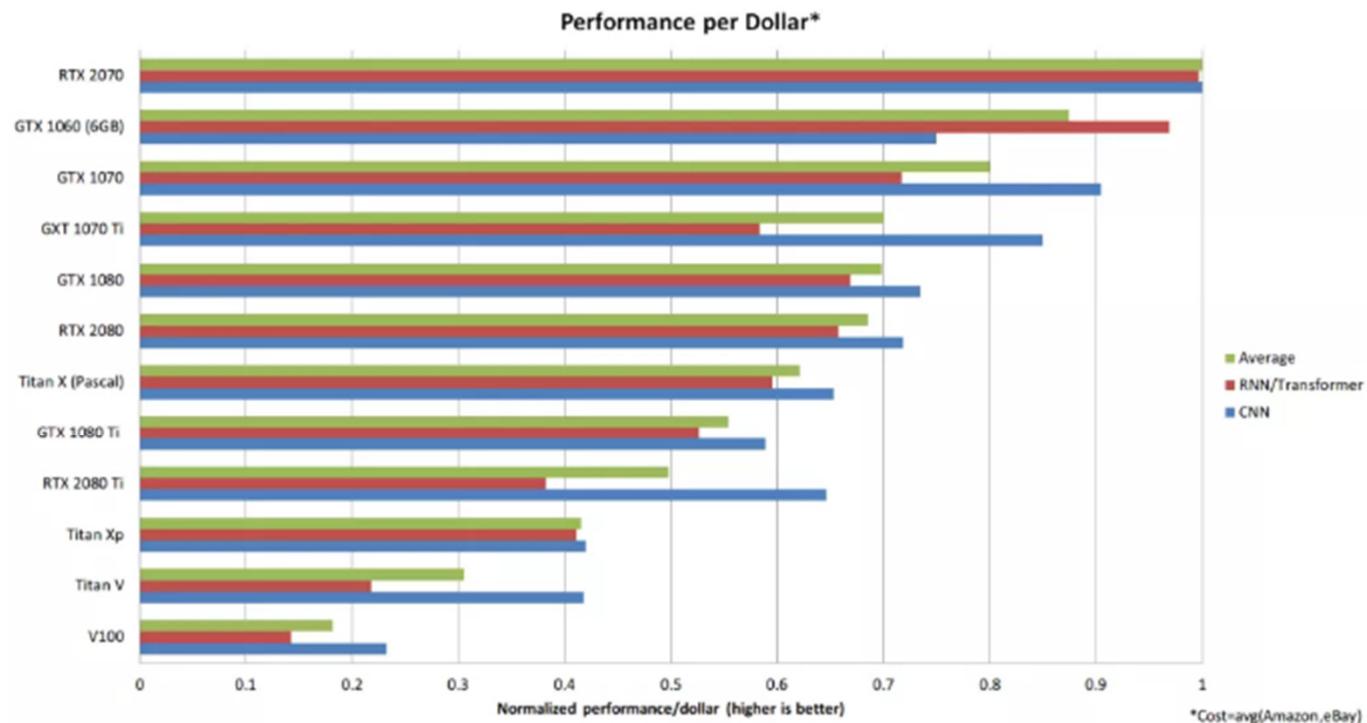


Figure 3: Normalized performance/cost numbers for convolutional networks (CNN), recurrent networks (RNN) and Transformers. Higher is better. An RTX 2070 is more than 5 times more cost-efficient than a Tesla V100.



There are number of places we can find Cloud GPUs

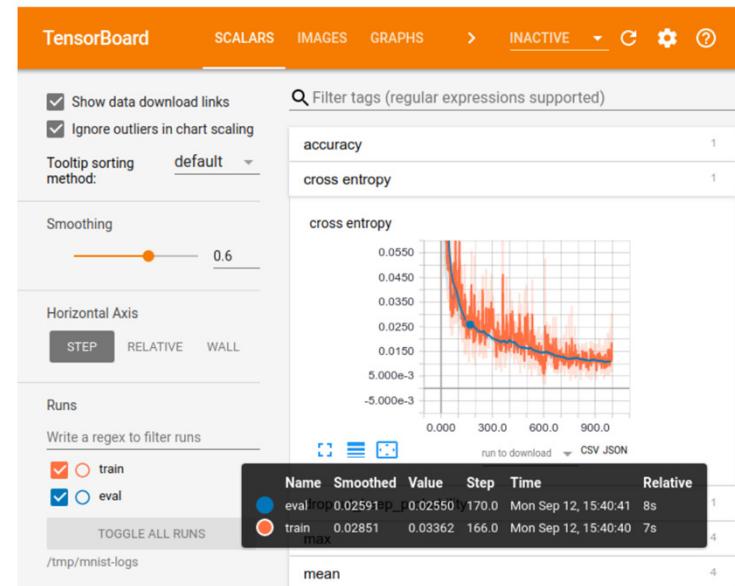
- AWS, Google Cloud, Azure and many others offer cloud GPUs at reasonable rates. That start as low as \$0.2 per hour using low performance GPUs to multiples of that for high performances GPUs.
- However, one of the best value cloud GPUs as of Nov 2018, is PaperSpace.

Using TensorBoard in Keras

What is TensorBoard



- TensorBoard is a set of visualization tools that aim to assist debugging, optimizing and visualizing aspects of training.
- Google has an excellent 30min tutorial about using TensorBoard here:
 - <https://www.youtube.com/watch?v=eBbEDRsCmv4>





What can you do with TensorBoard

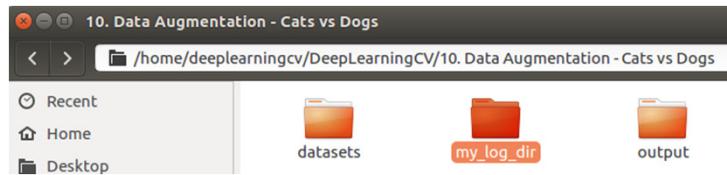
There are several views on TensorBoard

- **Scalars** - Visualize scalar values, such as classification accuracy.
- **Graph** - Visualize the computational graph of your model, such as the neural network model.
- **Distributions** - Visualize how data changes over time, such as the weights of a neural network.
- **Histograms** - A fancier view of the distribution that shows distributions in a 3-dimensional perspective
- **Projector** - Can be used to visualize word embedding's
- **Image** - Visualizing image data
- **Audio** - Visualizing audio data
- **Text** - Visualizing text (string) data



Viewing the TensorBoard in Keras

- Create a log directory in the directory



- Create a new callback where we call TensorBoard

```
callbacks = [
    keras.callbacks.TensorBoard(
        # Where our log files will be saved
        log_dir = './my_log_dir/',

        # Update frequencies for histograms and embeddings (1 - 1 epoch)
        histogram_freq = 1,
        embeddings_freq = 1,
    )
]
```

- Place the callbacks in the model.fit function
- Run "tensorboard --logdir=my_log_dir" in terminal

```
history = model.fit(x_train, y_train,
                     batch_size = batch_size,
                     epochs = epochs,
                     verbose = 1,
                     validation_data = (x_test, y_test),
                     callbacks = callbacks)
```

Best Approaches to CNN Design and Training



Best Approaches to CNN Design and Training

- As of now, we've never gone into how to you really design your own custom CNNs. All we've done is applied some models to some datasets with adequate results. However, there exists a framework with which you can work in.
- I've developed a procedure that is very useful when approaching Image Classification with CNNs,



Step 1 – Understand your data

Visualize your data/images and counts for each class to learn the following:

1. How complex are my image categories? Am I trying to get my CNN to decipher between images that are relatively challenging for humans? E.g, species of Tigers.
2. How many classes do I have?
3. Are the classes balanced or imbalanced?



Step 2 – Baseline CNN

Try using a simple baseline CNN without any fancy shenanigans such as:

- No complex activation functions and initializations, use ReLU and Kera's default activation functions for your weight matrices.
- No dropout or batch normalization
- Use a simple CNN similar to what we used for the CIFAR10 (see next slide for code)
- Train for at least 30 epochs (50 ideally but if time or GPUs not available, 20-30 is reasonable)

Step 2 – Baseline CNN Example Code

```
model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same',
                input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```





Step 2 – Assess Baseline CNN

- Is your baseline performing well (i.e. 75%+ in validation accuracy)?
- If so, then you can optimize the baseline by adding/changing the following (in this order) which should lead to better validation loss.
- First add all the following together or one at time to mark the improvement of each:
 - Add Dropout
 - Batch Normalization
 - Data Augmentation
 - Use SeparableConv2D
- Tinker with
 - Different Optimizers
 - Learning Rate Adjustment (try smaller values, e.g. 0.00001)



Step 3 – If Validation Accuracy Not Adequate for your tasks then

- Try either Fine Tuning/Transfer Learning or a Deeper CNN.
- If your dataset has similar classes or types of images as ImageNet then Fine Tuning/Transfer Learning would be a good option
- If still not sufficient, then try adding more Layers and more filters.
- Lastly, you can attempt to combine a few models (called Model Ensembling)



Improving Accuracy with Model Ensembling

```
preds_1 = model_1.predict(x_val)
preds_2 = model_2.predict(x_val)
preds_3 = model_3.predict(x_val)
preds_4 = model_4.predict(x_val)
preds_5 = model_5.predict(x_val)

final_preds = 0.2 * (preds_1 + _ + preds_3 + preds_4 + preds_5)
```