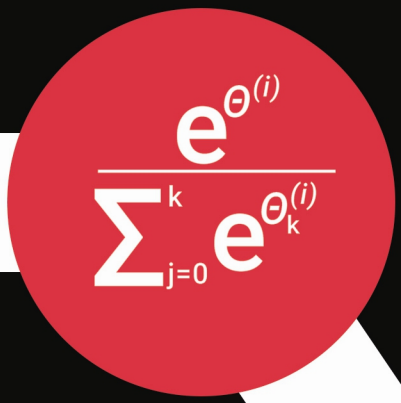
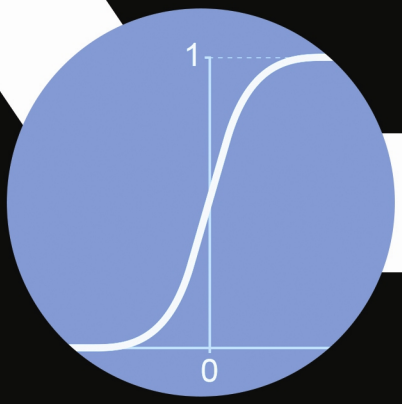
Multi-Model For
Driver Distraction
Detection and Elimination

A Deep Learning Approach





Driver Distraction System

Multi-Model For
Driver Distraction
Detection and Elimination
A Deep Learning Approach

Supervised by:

Dr. John Fayez Zaki

Prepared by:

Abdulrahman Issam AbouOuf

Mohamed Naser Elfrash

Omar Raefat Al-Ezaby

Omar Tarek Alsaqa

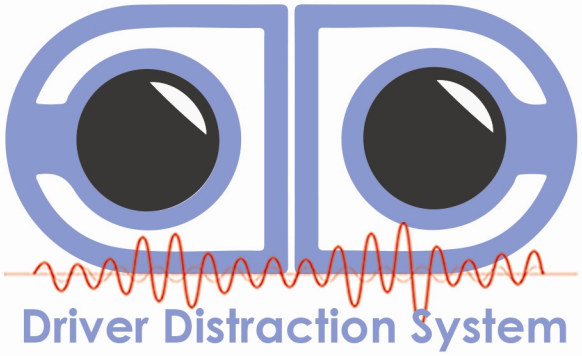


Table Of Contents

Chapter ONE INTRODUCTION 5
- What is Driver Distraction? 5
- Problem Statement 6
- Statistics – Driver Distraction in Numbers 6
- Current Solutions 8
- Our Solution – What do we introduce?! 9
- Technical Description 10

Chapter TWO SYSTEM DESIGN 13
- System Requirements 14
- Software Requirements 14
- System Constraints 14
- Technical Feasibility 14
- System Use case 15
- Sequence Diagram 16
- Software Architecture Diagram 17
- Hardware Architecture Diagram 17
- Product Development Cycle 18
- Solution deployment logistics 19
- Quality Assurance 19

Chapter THREE DRIVER ACTION CLASSIFICATION 21
- Dataset 22
- Evaluation Metrics 23
- Vanilla Model 24
- Data leakage 25
- Improving / Overfitting 26
- Results 29

Chapter FOUR Head Pose 30
- Pipeline 31
- Dataset 32

Model Details 34
Model Architecture 36
Results 40
Live testing 41
Chapter FIVE TRIGGER WORD 43
Pipeline 44
Dataset 45
Model Details 46
Model Architecture 51
Results 55
Live testing 56
Chapter SIX AUTOMATIC SPEECH RECOGNITION 59
Pipeline 60
What is ASR? 60
Types of ASR 60
The Representation of Voice 62
Parameters of an audio signal 62
Why speech recognition is difficult 66
ASR Models 68
Our Used Solution 69
Chapter SEVEN COMMANDS CLASSIFICATION 85
Pipeline 86
How the model works? 86
Dataset 86
Data Augmentation 88
How EDA Works? 89
Text Classification Model 92
Classifier Pipeline and Architecture 92
Results 95
Named Entity Recognition 97
Chapter EIGHT COMPRESSION 99
& DEPLOYMENT 99

Compression 100
Deployment 112
Appendix A PROJECT MANAGEMENT METHODOLOGY 117
Waterfall for Big Picture 118
Agile for developing the first model 118
Rest of the models 119
Final two months 119
Appendix B PLANNING 120
Gantt Chart 121
Funding 121
Mentorship 121
Competitions 121
Deployment and Training 121
Appendix C ISSUES 122
Datasets 123
Hardware Shipment and Funding 123
Models' Training 123
Contingency Plan 123
Appendix D Definitions 124
References 12487

ABSTRACT

This project aims to reduce number of car crashes by using AI systems that help the driver not to get distracted.

Our solution is to make everything inside the cabin accessible by voice (distraction elimination). Also we believe that our project would make it possible to ascertain quickly drivers' actions and their sights and to continually build use cases so we can understand when something may interfere with the driving process and create new ways to alert the driver (distraction detection).

Our project is composed of 5 AI/ML components:

1- Driver actions classification: which is done by a camera takes images of the driver and classify his action whether he's distracted or not.
2- Head pose estimation: which is done by another camera takes images of the driver's head and provide its angles.
3- Speech to text Engine: to generate text transcriptions from the driver voice to prepare it for the text classification model.
4- Text classification: which takes the text transcription and classify it to which command the driver wants to do.
5- Trigger word detection: to run the whole system by voice without getting distracted.

After a complete compression study of how we compress all these models together on a chip board we combined all these tasks together and we came up with a complete new multi-model system for driver distraction elimination and detection.

What's interesting about this project is that while developing, testing and enhancing we weren't only concentrating on the technicalities but we kept in mind the end goal of the project which is to create a business product that works offline to provide safety to the driver.

Thanks to Valeo Group Support Program we created such a product as they are one of the biggest companies regarding automotive suppliers that knows the target audience.

PART 01

INTRODUCTION

CHAPTER ONE

INTRODUCTION

This document will provide the problem of Driver Distraction in the road, what are the current solutions and systems to solve the problem? and how our project/solution will solve it (using Deep Learning Computer Vision and NLP Voice Commander) for greater impact.
The task of driving requires continuous attention to road and vehicle control. Drivers may pay insufficient attention to driving because: they are occupied with other activities such as making a phone call or texting, tuning the radio or the air conditioning, talking with a passenger. etc. and this requires a Full System that will prevent the driver from losing attention and alert him/her if they are distracted.

What is Driver Distraction?

Driver Distraction is a form of driver inattention that involves the diversion of attention away from safety critical activities within the driving task.

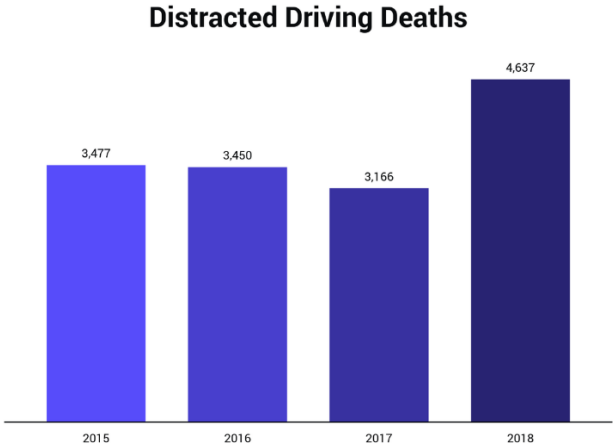
Problem Statement

According to the World Health Organization, the number of road traffic deaths continues to increase, reaching 1.35 million in 2016. A research conducted around traffic safety indicates that about 25% of car crashes have been caused by driver distraction.

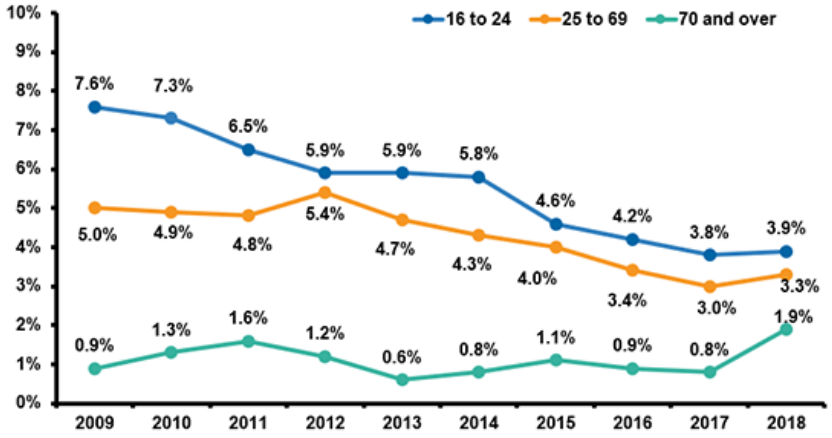
Every year around 12,000 Egyptians lose their lives as a result of road traffic crashes, 48% of those killed in motor vehicle crashes are occupants (passengers and drivers). The Egyptian Central Agency for Public Mobilization and Statistics (CAPMAS) reported that more than 20% of road accidents occur due to driver inattention/distraction (Visual – Manual – Cognitive).

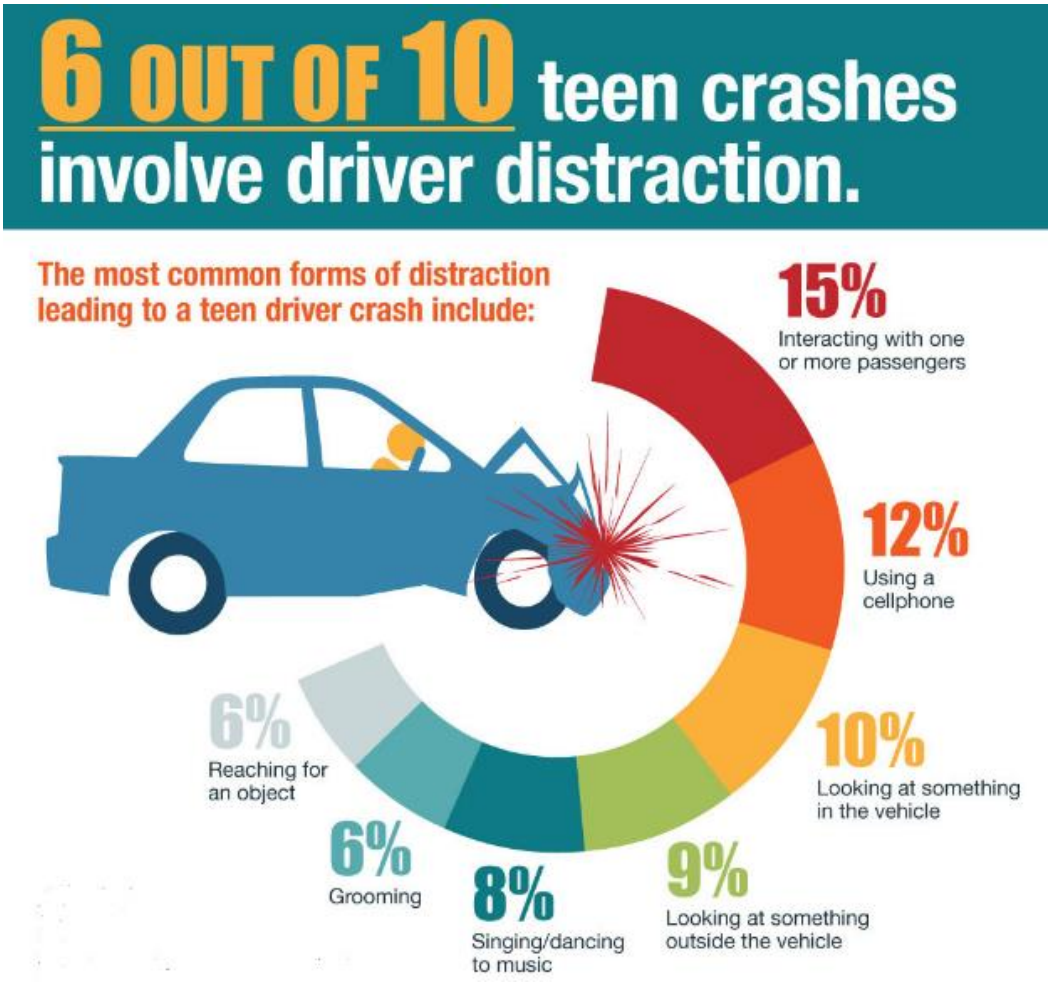
Statistics – Driver Distraction in Numbers

For many, driving is a daily activity, not requiring much thought or consideration. However, the sad reality is that there are 3,287 deaths each day due to fatal car crashes. On average, 9 of these daily fatalities are related to distracted driving.



Teens have been the largest age group that reported being distracted while driving. Driver distraction is reported to be responsible for more than 58% of teen crashes. And that is due to the using of Mobile Phones while driving (Texting – Phone calls – Listening to Music. Etc..).



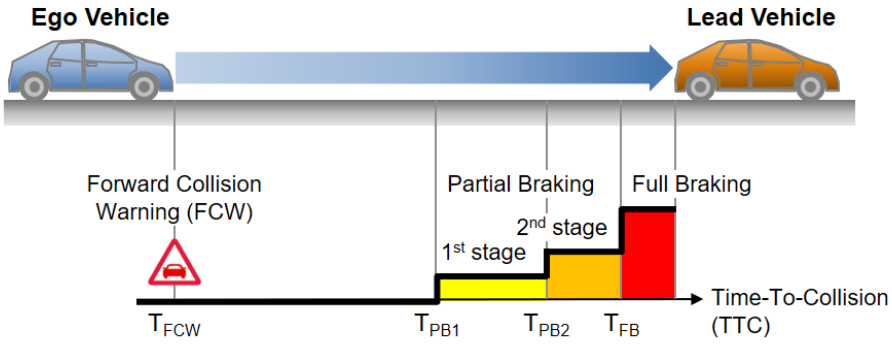


Current Solutions

Many solutions were developed to solve this rising problem using Embedded systems and sensors but they were not as accurate and not very cost efficient.

- Forward-Collision Warning (FCW)

It provides a visual, audible, and/or tactile alert to warn drivers of an impending collision with a car. Using sensors in front of the car to only alert if the car reaches a specific distance before collision.



- Automatic Emergency Braking (AEB)

If the system senses a potential collision and the driver doesn't react in time, it engages the brakes. It uses the same tech as FCW.

- Apple IOS 11 and AT&T DriveMode

This latest operating system includes a Do Not Disturb While Driving mode (DND) that can block notification of incoming calls and texts when your iPhone senses driving motion or is connected to a car via Bluetooth. (It doesn't block functions that work through Apple's CarPlay system, such as music and navigation.) The DND feature can automatically send a text reply that says you're driving and will reply later. Phone calls are allowed if the iPhone is connected via Bluetooth.Drivesafe.ly and DriveMode apps

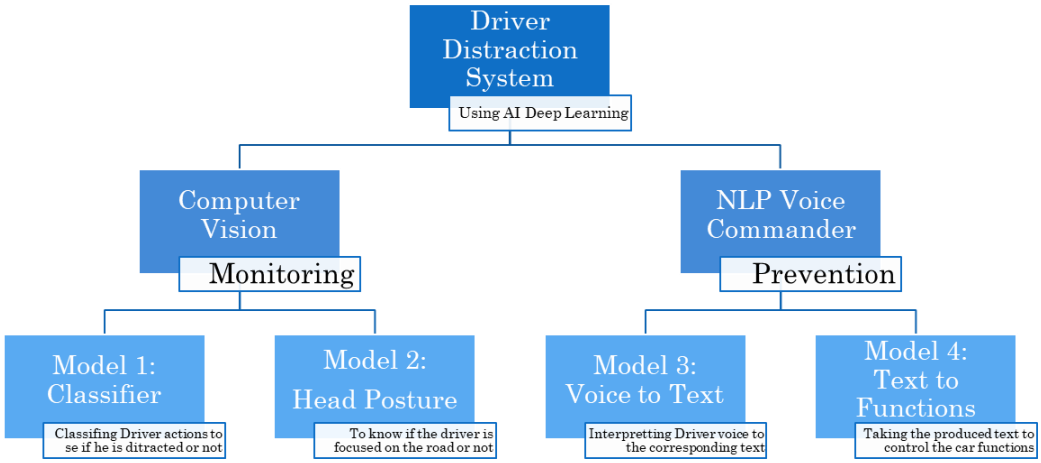
- Drivesafe.ly and DriveMode apps

This app is designed to read incoming text messages and email aloud for drivers so they keep their eyes on the road.

Our Solution – What do we introduce?!

Our Project will introduce deep learning into the equation to monitor the driver and prevent from being distracted as a packed solution.

The solution will run as an End Device on the car that doesn't need an internet connection.



I. Monitoring:

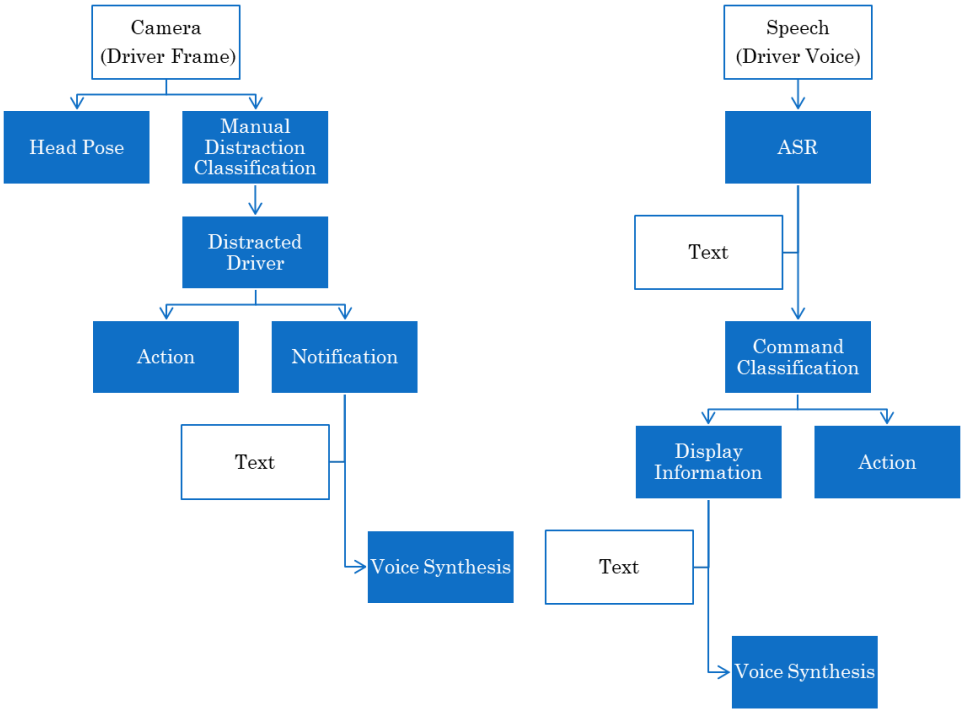
We need to monitor the driver and see if he is distracted or not (using a camera) by classifying the camera result to know if he is distracted or not, and as an added bonus we will classify what he is actually doing using classes of actions (Safe Driving – Texting – Talking on the phone – Operating the radio – Drinking – Reaching Behind – Talking to a passenger, ETC). Using a Deep Learning Neural Network. We also need to know the driver's eyes are on the road or not. This will be obtained by using a Head Posture Detection model

II. Prevention

Introducing a Car Voice Commander to help take commands from the driver and take actions upon these commands without the driver using hands and losing the grip of the steering wheel.

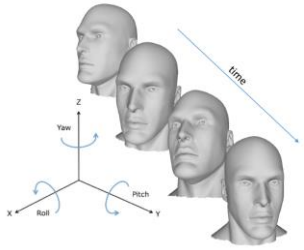
Technical Description

Technically, the system will work as follow



1) Distraction Detection Subsystem

Composed of two AI/ML models (Head pose detection, Driver actions classification).



We need to monitor the driver and see if he is distracted or not (using a camera) by classifying the camera result to know if he is distracted or not, and as an added bonus we will classify what he is actually doing using classes of actions (Safe Driving – Texting – Talking on the phone – Operating the radio – Drinking – Reaching Behind – Talking to a passenger, ETC). Using a Deep Learning Neural Network. We also need to know the driver's eyes are on the

road or not. This will be obtained by using a Head Posture Detection model

2) Distraction Elimination Subsystem

Composed of two AI/ML models (Speech recognition, Text classification).

Introducing a Car Voice Commander to help take commands from the driver and take actions upon these commands without the driver using hands and losing the grip of the steering wheel.

OPEN THE AIR CONDITIONER
OPEN RADIO
OPEN THE WINDOW
WHAT'S THE CAR STATE
WHAT'S ENGINT TEMPRETURE

CHAPTER TWO

SYSTEM DESIGN

"HOW WE DESIGNED THE SYSTEM! AND WHAT IS THE SEQUENCE OF THE SYSTEM FROM THE BEGINNING"

System Requirements

- Hardware Requirements
- 2 cameras and a mic and a flat screen to show results.
- GPUs to train our models.
- Nvidia Jetson Nano board to run our model combined.

Software Requirements

- Our models trained and compressed.
- Pre-installed packages (Keras, Tensorflow, Pytorch, etc.)

System Constraints

The system must be installed on modern cars who has electronic control unit (Computer Box) and also the system requires two cameras, microphone and a flat screen.

	Driver Actions	Head Pose	Speech Recognition	Text Classification	Trigger word
GPU RAM usage	860 MB	1.4 GB	-	1.5 MB	153 MB

The model runs on CPU real time but to get even faster response use GPU with minimum memory ram.

Technical Feasibility

User Familiarity: Low – Medium risk.

- The staff (Installing Engineers – Development Engineers) will require an easy training to interact with the system and technologies introduced.
- The driver will require an easy guide to use the system.

Hardware: Medium risk.

- GPU 4GB RAM Nvidia chip board.
- Two cameras, a mic and a flat screen.

Software: Low – Medium risk.

- The staff (Development Engineers) will require a strong background in the computer vision and NLP fields.
- The driver will smoothly use the system without any background knowledge.

Analyst and Management: Medium risk.

- Managing the developing team.

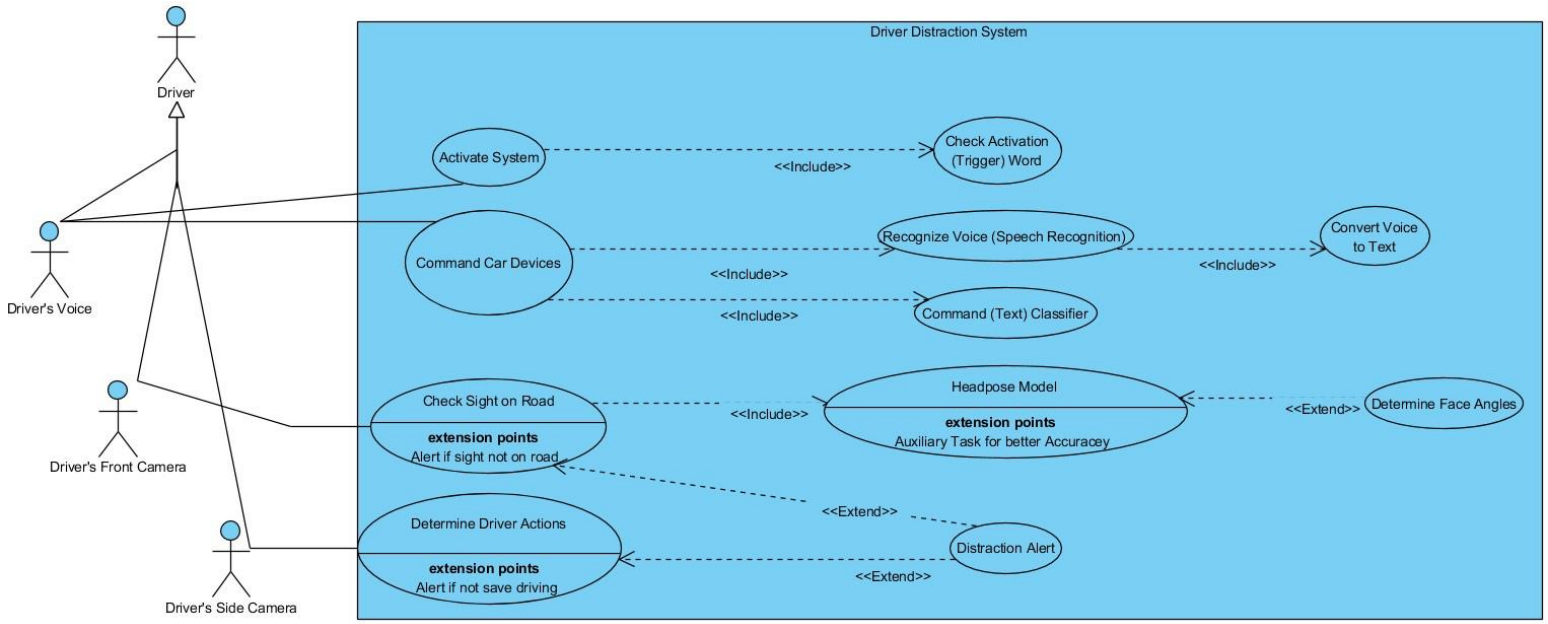
- Contact the suppliers and car manufacturers.

Cost: Low - Medium risk.

- Jetson Nano developer kit: 200$
- Cameras: 80$
- Microphone: 35$
- Connectors: 30$
- The System will totally cost 350$

System Use case

To identify the user interaction with the system we used the use case diagram as follows

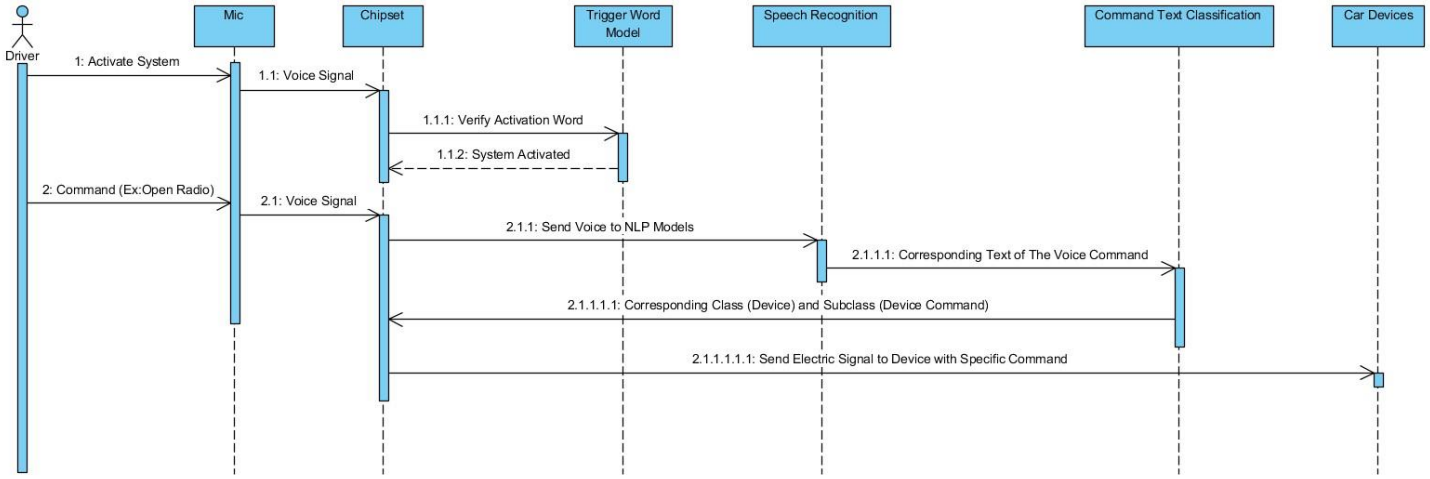


Sequence Diagram

To identify how the system components are arranged in time sequence we used the sequence diagrams as follows

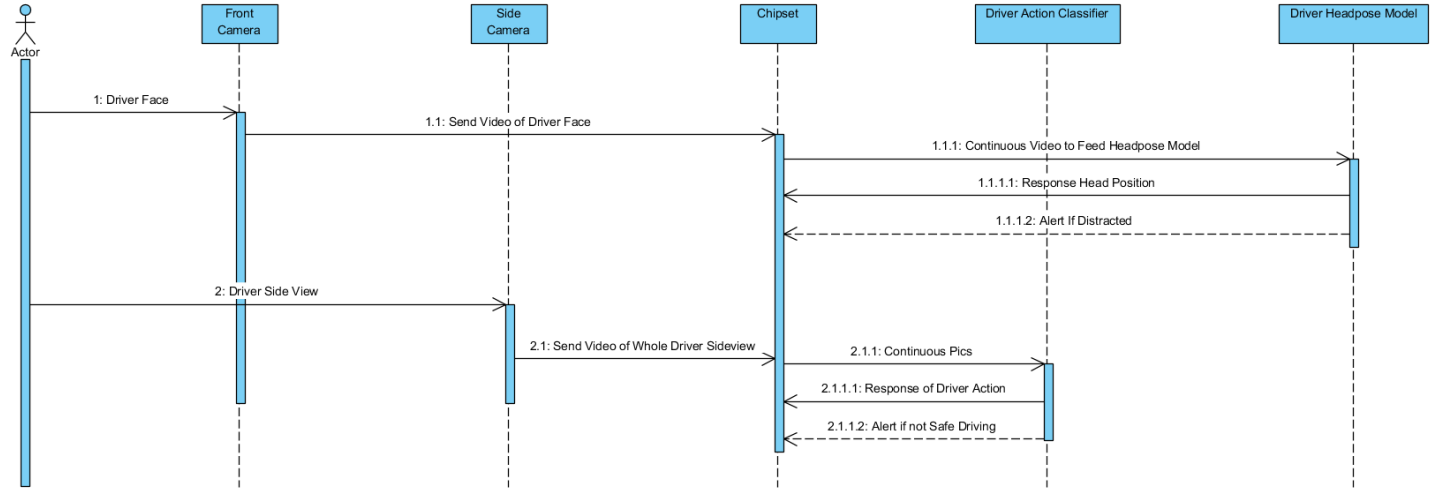
NLP Models

Shows how NLP models are used from system activation through speech recognition till they give the final actions needed.

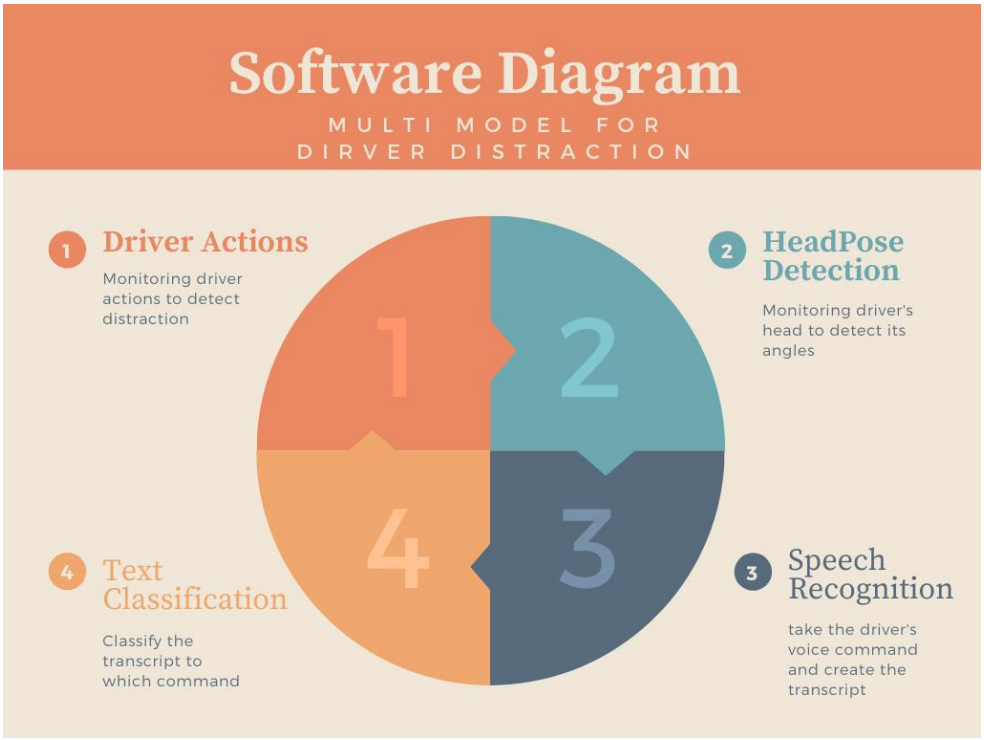


Computer Vision Models

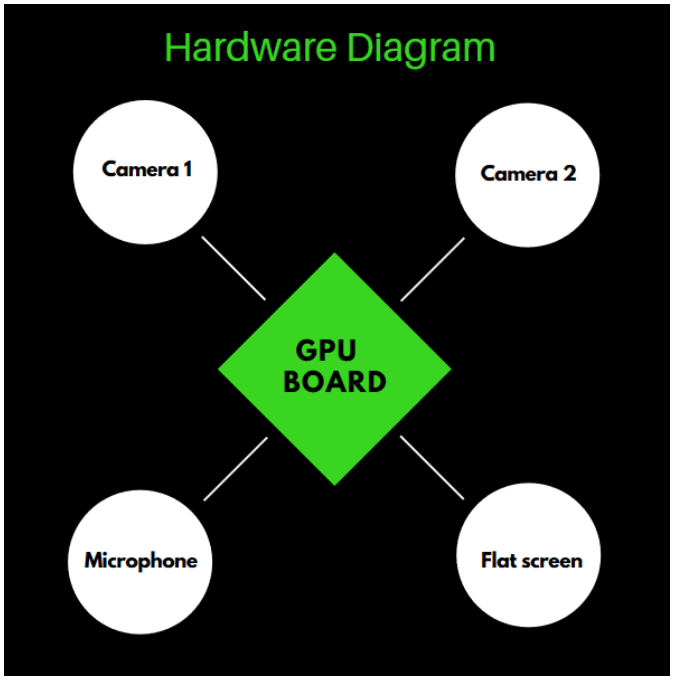
Shows how to utilize cameras to determine whether the driver is distracted or not.



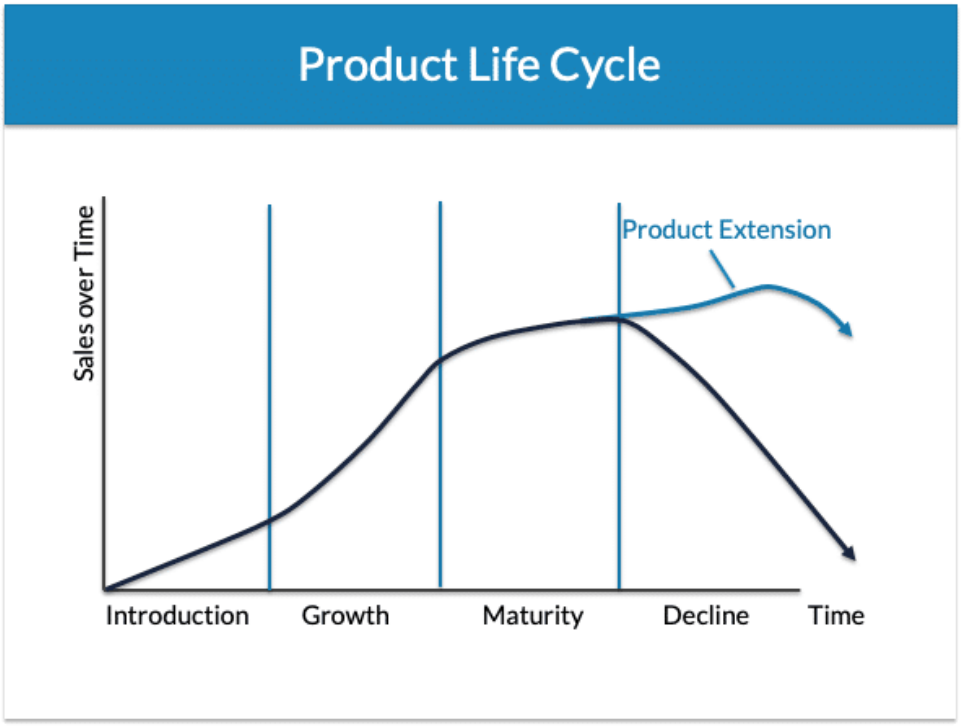
Software Architecture Diagram



Hardware Architecture Diagram



Product Development Cycle



Initiation – Introduction

After searching how common driving accidents are. And how driver distraction is one if not the main reason for causing terrible accidents.

And after searching and experiencing most common driver distraction solutions, we came to conclusion that deep learning model as discussed would be a best fit nowadays, by deploying the aspects of prevention using voice commands and detection using image recognition and action classification.

Problem Child – Growth

It is the question of how our solution will fit the market share to grow?

Most solutions in the market are focused on only just one aspect just as discussed before. And they only properly focus on distraction detection. But our solution adds prevention as a bonus.

Since our solution is a standalone solution that can be easily integrated in any car as it doesn't require internet (just a camera – mic and the chip) and it will produce signals to control the car functions.

Maturity

Our plan is to propose this model to the high branded cars. And with our Supervisor(Mercedes Benz – BMW, Egypt sectors) and since our project is supported by Valeo. We are trying to propose an MVP that is updated on the go (Producing fthe minimum Features – testing – adding more features) cycles.

Decline

To extend our project cycle we will add more features to tackle cognitive distraction. And trying to find and acquire cognitive datasets. And this will impressively extend the life span of our solution.

However, with the revolution of self-driving cars that will take our project hopefully as an initial point to go from there, our project will be declined as a standalone solution and it will be converted to a self-driving cars' feature.

Solution deployment logistics

Target audience and business process

Quality Assurance

Testing on various datasets. And actual people putting them on a distracted driving simulation.

PART 02

COMPUTER VISION

CHAPTER THREE

DRIVER ACTION CLASSIFICATION

"THE CAR CAN NOW TELL WHAT YOU ARE DOING IN THE ACT AND ACT ON IT FOR YOUR OWN SAFETY..."

This model has data about the driver's actions. And it detects the action of the driver if he is safe driving or doing an action that distracts him (talking to a passenger – holding the phone – texting, etc.…).

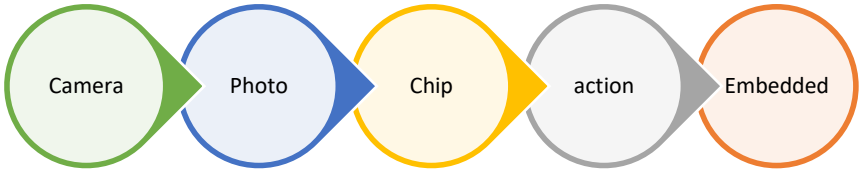
We will have a camera that will be focused on the driver's body, and every interval of time and we will pass the image to a CNN arch that will classify the actions of the driver based on a predefined 10 action that will be used if the driver is keeping attention to the driving or distracted and if the driver is classified as distracted more than a specific number consecutive frames we will notify the driver to focus on the driving or slowing the car speed and stop the car if the driver didn't take action.

We tried different CNN Models to classify the driver action and our best arch achieved 86 % accuracy, but we will try to enhance the

accuracy by focusing the regions of interest in the classification like the driver's hands and head.

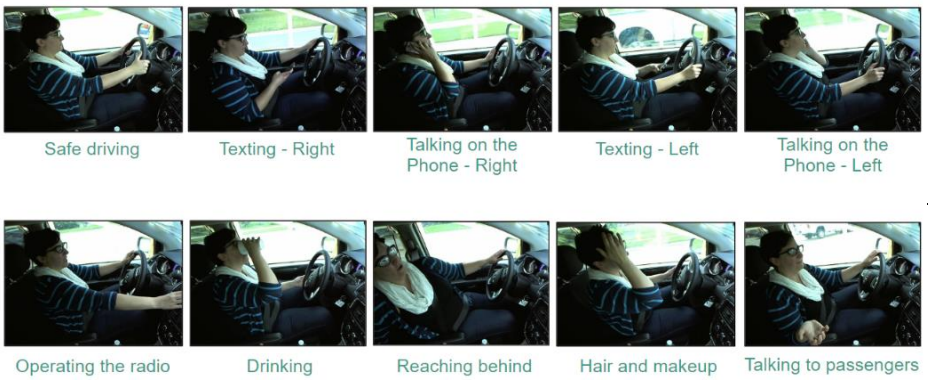
Pipeline

There is a camera positioned on the right side of driver. It takes a video and send it to a chip i.e. Jetson Nano and the chip process the video frame by a frame and tell what is the state of the driver between 10 different classes. Then send signal to the car or the embedded pasrts in it to take appropriate action to remove or reduce the distraction of the driver. The car could slow down, notify the driver, open windows or play some music.

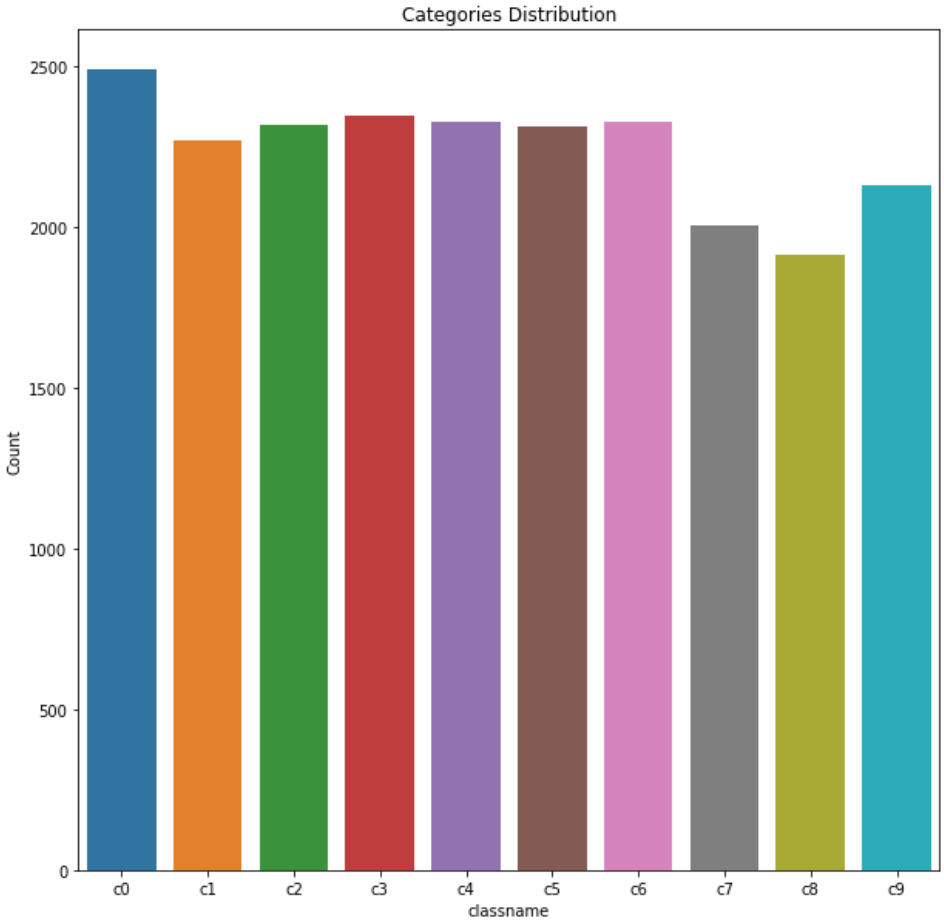


Dataset

- StateFarm Dataset contains snapshots from a video captured by a camera mounted in the car. The training set has 22.4K labeled samples with almost equal distribution among the classes for the 25 different driver in training set. In the Test set there is 79.7K unlabeled test samples. There are 10 classes of images:



'c0': 'Safe driving',
'c1': 'Texting - right',
'c2': 'Talking on the phone - right',
'c3': 'Texting - left',
'c4': 'Talking on the phone - left',
'c5': 'Operating the radio',
'c6': 'Drinking',
'c7': 'Reaching behind',
'c8': 'Hair and makeup',
'c9': 'Talking to passenger'



- AUC Distracted Driver Dataset contains 17.3K snapshots also from a camera. It has the same classes as the StateFarm dataset with 31 different drivers.

Evaluation Metrics

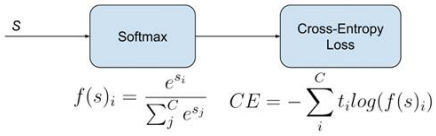
Before proceeding to build models, it's important to choose the right metric to gauge its performance.

- Accuracy: Classification Accuracy is what we usually mean, when we use the term accuracy. It is the ratio of number of correct predictions to the total number of input samples.

$$Accuracy = \frac{Number\ of\ Correct\ predictions}{Total\ number\ of\ predictions\ made}$$

  It works well only if there are almost equal number of samples belonging to each class.
- Log Loss: Accuracy only takes into account the correctness of the prediction i.e. whether the predicted label is the same as the true label. But, the confidence with which we classify a driver's action as distracted is very important in evaluating the performance of the model.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$



- Confusion Matrix: as the name suggests gives us a matrix as output and describes the complete performance of the model. It also gives you indicator to which classes the model struggle to fit to it. Also, help you to know which classes the model confuses between.

Vanilla Model

With the understanding of what needs to be achieved, we proceeded to build the CNN models from scratch. We added the usual suspects — convolution batch normalization, max pooling, and dense layers. The accuracy of this vanilla model was 99.6% on the validation set in 5 epochs.

and the results was:

In 5 epochs	*Accuracy*
Train	99.8%
Validation	99.6%

```
model = Sequential()

## CNN 1
model.add(Conv2D(32,(3,3),activation='relu',input_shape=(64, 64, 3)))
model.add(BatchNormalization())
model.add(Conv2D(32,(3,3),activation='relu',padding='same'))
model.add(BatchNormalization(axis = 3))
model.add(MaxPooling2D(pool_size=(2,2),padding='same'))

## CNN 2
model.add(Conv2D(64,(3,3),activation='relu',padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64,(3,3),activation='relu',padding='same'))
model.add(BatchNormalization(axis = 3))
model.add(MaxPooling2D(pool_size=(2,2),padding='same'))

## CNN 3
model.add(Conv2D(128,(3,3),activation='relu',padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128,(3,3),activation='relu',padding='same'))
model.add(BatchNormalization(axis = 3))
model.add(MaxPooling2D(pool_size=(2,2),padding='same'))

## Output
model.add(Flatten())
model.add(Dense(512,activation='relu'))
model.add(Dropout(0.5))
```

This is a quiet accuracy, *we should stop!*

Actually, this is rubbish as we will see in the next section...

Data leakage

This is happing due to the StateFarm Dataset has **22.4K** photo of 10 Drivers so there must be some photos for the same driver.

For example, driver with **Driver ID - p021** has **1237** photo if we divided them into 10 equal classes then every class has 123 photos at least for the same drive and same action. If we split data randomly there may be some of the 123 in the training and some in the testing or evaluation.

To counter this issue, we split the images based on the person IDs instead of using a random 80–20 split.

	driver_id	Counts
0	p021	1237
1	p022	1233
2	p024	1226
3	p026	1196
4	p016	1078
5	p066	1034
6	p049	1011
7	p051	920
8	p014	876
9	p015	875
10	p035	848

But as expected the model will overfit quickly on the training set as you have a lot of the same photos in training. The accuracy for validation was almost 30%!

Improving / Overfitting

To improve the results further, we trained and tested deep neural networks SOTA architectures.

I. Transfer Learning

The first choice for us was of course ResNet50. We used version 2.0 as it is more recent and better.

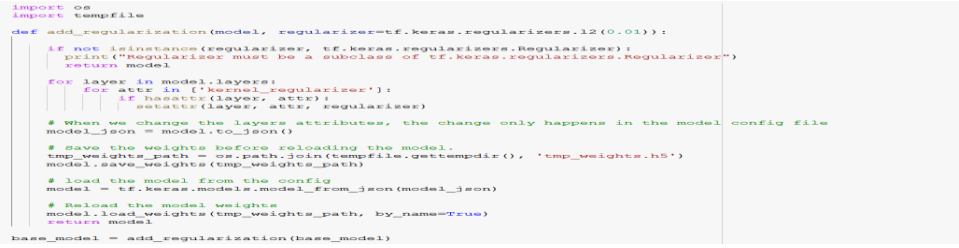
- ResNet50:

```
# create the base pre-trained model
base_model = ResNet50V2(weights='imagenet', include_top=False)
base_model.trainable = True

for layer in model.layers[:86]:
   layer.trainable = False
for layer in model.layers[86:]:
   layer.trainable = True
```

We also used transfer learning from the ImageNet and freeze the first 50% of the layers and let the other 50% to be trained to our data.

II. Regularization

As we are using a pre-trained model we can add regularization to some specific layers without missing the architecture and the weights. But we luckily found out a method to do so without any problem.



```
import os
import tempfile

def add_regularization(model, regularizer=tf.keras.regularizers.l2(0.01)):

    if not isinstance(regularizer, tf.keras.regularizers.Regularizer):
      print("Regularizer must be a subclass of tf.keras.regularizers.Regularizer")
      return model

    for layer in model.layers:
      for attr in ['kernel_regularizer']:
        if hasattr(layer, attr):
          setattr(layer, attr, regularizer)

    # When we change the layers attributes, the change only happens in the model config file
    model_json = model.to_json()

    # Save the weights before reloading the model.
    tmp_weights_path = os.path.join(tempfile.gettempdir(), 'tmp_weights.h5')
    model.save_weights(tmp_weights_path)

    # load the model from the config
    model = tf.keras.models.model_from_json(model_json)

    # Reload the model weights
    model.load_weights(tmp_weights_path, by_name=True)
    return model
base_model = add_regularization(base_model)
```

III. Other techniques

- We used a small learning rate and small batch size to gain more regularization effect for the model. *-As mentioned in (Wilson and Martinez, 2003) page 276*. Also, added Learning Rate scheduler for every 10 epochs we reduce the learning rate.

- Spatial dropout.

```
def lr_schedule(epoch):
        lr = 1e-4
        if epoch > 40:
                lr *= 0.5e-3
        elif epoch > 30:
                lr *= 1e-3
        elif epoch > 20:
                lr *= 1e-2
        elif epoch > 10:
                lr *= 1e-1
        print('Learning rate: ', lr)
        return lr
```

IV. Augmentation

Using ImageDataGenerator we found it is suitable to use:

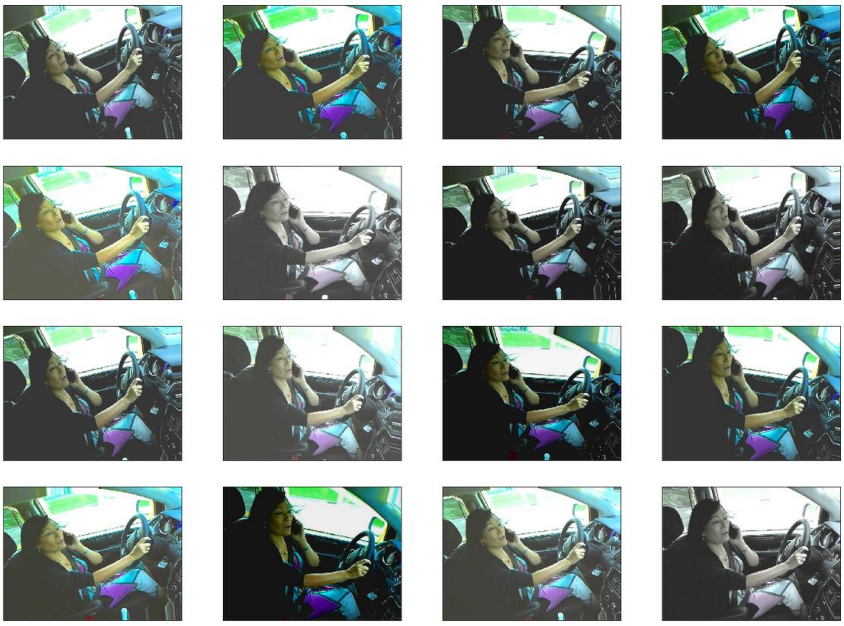
- Rotation Range: We decided to go with 15, we found we can do from 10 to 30.

- Shear Range:

In addition to Keras ImageDataGenerator

```
[471]:
def special_function(img):
    '''
    The default:
    - adjust_brightness: contrast range is 0.05 to 0.25
    -
    '''
    x = not x
    brightness_factor = random.uniform(0.05, 0.25)
    im = tf.image.adjust_brightness(img, brightness_factor)

    #contrast_factor = random.uniform(0.05, 0.95)
    #im = tf.image.adjust_contrast(im, contrast_factor)

    if x :
        saturation_factor = random.uniform(0.25, 3)
        im = tf.image.adjust_saturation(im, saturation_factor, name=None)

    #delta = random.uniform(0.1, 0.19)
    #im = tf.image.adjust_hue(im, delta, name=None)
    return im
```

```
#channel_shift_range=0.14, rotation_range=15, shear_range=5, zoom_range=0.1,
train_gen = ImageDataGenerator(
    rotation_range=15, # Good with 20, could be more
    #width_shift_range=width_shift_range, # Not recommended
    #height_shift_range=height_shift_range, # Not recommended
    #brightness_range=brightness_range, # Dont's use it not brightness!
    shear_range=5, # Small effect on this dataset, but good to use with 10
    zoom_range=0.1, # Tricky, but i'd use it, Good with 0.14
    channel_shift_range=0.14, # Good with 0.14
    fill_mode="nearest", # Don't Change
    rescale=1.0/255.0,
    preprocessing_function=special_function #you colud add other Augmentation Function
    )
```



- More Data

As we mentioned previously we only worked so far on the StateFarm dataset. We decided to use more data when we found the model is robust and his performance as it was expected but we used the AUC dataset to increase the total accuracy.

We have taken 650 from each class from AUC dataset to preserve the balance in our data set so the final distribution for training and

validation data is: 24.5K for training set. 3.9K for validation set. Also, we added from the AUC dataset 1123 photo for test set.

Results

In 5 epochs	*Accuracy*
Train	99.1%
Validation	95.4%
Test	93.5%

For a video 1:23 Min:

- on CPU:

total: 312.742 sec (5.2 Min)

avg All (read, resize, etc..): 0.1375 sec

avg predict(only): 0.11 sec

- on GPU:

total: 217.280 sec (3.6 Min)

avg All (read, resize, etc..): 0.0883 sec

avg predict(only): 0.0618 sec

CHAPTER FOUR

Head Pose

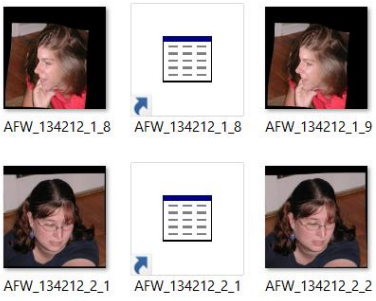
This Model will detect the angel of the driver's head to know if he is focused on the road or not. It uses three angles as follow:
The car will have a camera in the front of the driver, and every interval of time, we will take a photo of the driver, then we will use a YOLO Face detection arch to detect the location of the driver face in the image, then we will pass the image of the driver face to a head pose model that first will extract the features from the driver's face Using CNN Arch and then pass a classifier that will detect the angels of the driver's face in the 3D space.

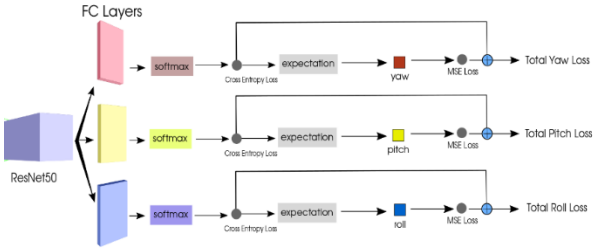
Pipeline

Estimating the head pose of a person is a crucial problem that has a large amount of applications such as aiding in gaze estimation, modeling attention, fitting 3D models to video and performing face alignment. Traditionally head pose is computed by estimating some key-points from the target face and solving the 2D to 3D correspondence problem with a mean human head model. We argue that this is a fragile method because it relies entirely on landmark detection performance, the extraneous head model and an ad-hoc fitting step. We present an elegant and robust way to determine pose by training a multi-loss convolutional neural network on 300W-LP, a large synthetically expanded dataset, to predict intrinsic Euler angles (yaw, pitch and roll) directly from image intensities through joint binned pose classification and regression. We present empirical tests on common in-the-wild pose benchmark datasets which show state-of-the-art results. Additionally, we test our method on a dataset usually used for pose estimation using depth and start to close the gap with state-of-the-art depth pose methods.

In the sake of simplicity our HopeNet is an accurate and easy to use head pose estimation network. Models have been trained on the 300W-LP dataset and have been tested on real data with good qualitative performance.

The overall pipeline of the model goes like that:





First things first we started by studying the data, and play with it a little bit by doing some augmentation like flipping and images blur (Fig. 1), then the image goes through face detection to detect the exact location of the face (Fig. 2), now when the preprocessing is done on the images everything is ready for the image to go through the model (Fig. 3), and we do our testing and evaluation on a real environment dataset, the we start to make prediction of the three different angles of the face (Fig. 4).

Dataset

Training Set Information,

300W-LP dataset, a synthetic expansion of the 300W dataset. Augmentation of 300W was performed in order to obtain face appearances in larger poses. This dataset provides annotations for both 2D landmarks and the 2D projections of 3D landmarks.

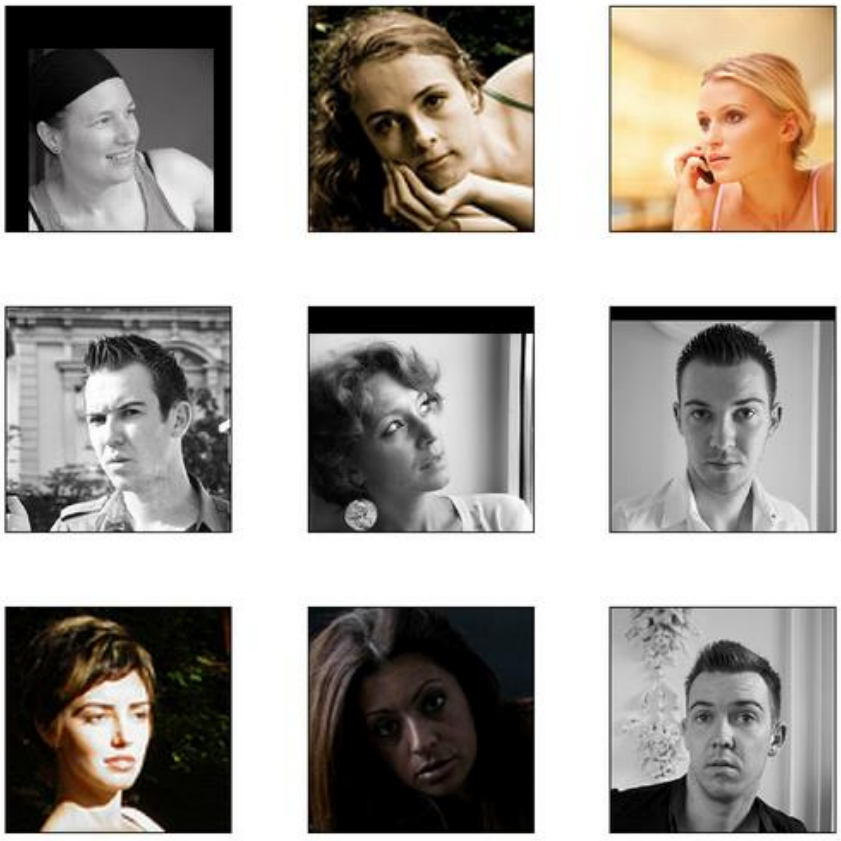
300W-LP Dataset is expanded from 300W, which standardizes multiple alignment databases with 68 landmarks, including AFW, LFPW, HELEN, IBUG and XM2VTS. With 300W, 300W-LP adopt the proposed face profiling to generate 61,225 samples across large poses (1,786 from IBUG, 5,207 from AFW, 16,556 from LFPW and 37,676 from HELEN, XM2VTS is not used).

The dataset can be employed as the training set for the following computer vision tasks: face attribute recognition and landmark (or facial part) localization.

Our training validation split is done by 90/10 split which means we got almost 55K images to train on.

Test set information,

AFLW2000-3D dataset, a dataset of 2000 images that have been annotated with image-level 68-point 3D facial landmarks. This dataset is typically used for evaluation of 3D facial landmark detection models. The head poses are very diverse and often hard to be detected by a CNN-based face detector. The 2D



landmarks are skipped in this dataset, since some of the data are not consistent to 21 points, as the original paper mentioned

In order to truly make progress in the problem of predicting pose from image intensities we have to find real dataset which contain precise pose annotations, numerous identities, and different lighting conditions, all of this across large poses. We identify two very different datasets which fill these requirements.

Consequently, this dataset contains accurate fine-grained pose annotations and is a prime candidate to be used as a test set for our task.

Model Details

As mentioned in the overall pipeline we started the model by studying the dataset and preparing it for our model, first thing to do was drawing axis and face segmentation which was already done for us by the dataset as it provided us with the rectangle around the face!

```
def draw_axis(img, yaw, pitch, roll, tdx=None, tdy=None, size = 100):

    pitch = pitch * np.pi / 180
    yaw = -(yaw * np.pi / 180)
    roll = roll * np.pi / 180

    if tdx != None and tdy != None:
        tdx = tdx
        tdy = tdy
    else:
        height, width = img.shape[:2]
        tdx = width / 2
        tdy = height / 2

    # X-Axis pointing to right. drawn in red
    x1 = size * (cos(yaw) * cos(roll)) + tdx
    y1 = size * (cos(pitch) * sin(roll) + cos(roll) * sin(pitch) * sin(yaw)) + tdy

    # Y-Axis | drawn in green
    #        v
    x2 = size * (-cos(yaw) * sin(roll)) + tdx
    y2 = size * (cos(pitch) * cos(roll) - sin(pitch) * sin(yaw) * sin(roll)) + tdy

    # Z-Axis (out of the screen) drawn in blue
    x3 = size * (sin(yaw)) + tdx
    y3 = size * (-cos(yaw) * sin(pitch)) + tdy

    cv2.line(img, (int(tdx), int(tdy)), (int(x1),int(y1)),(0,0,255),3)
    cv2.line(img, (int(tdx), int(tdy)), (int(x2),int(y2)),(0,255,0),3)
    cv2.line(img, (int(tdx), int(tdy)), (int(x3),int(y3)),(255,0,0),2)

    return img
```

```python
def plot_pose_cube(img, yaw, pitch, roll, tdx=None, tdy=None, size=150.):
    # Input is a cv2 image
    # pose_params: (pitch, yaw, roll, tdx, tdy)
    # Where (tdx, tdy) is the translation of the face.
    # For pose we have [pitch yaw roll tdx tdy tdz scale_factor]

    p = pitch * np.pi / 180
    y = -(yaw * np.pi / 180)
    r = roll * np.pi / 180
    if tdx != None and tdy != None:
        face_x = tdx - 0.50 * size
        face_y = tdy - 0.50 * size
    else:
        height, width = img.shape[:2]
        face_x = width / 2 - 0.5 * size
        face_y = height / 2 - 0.5 * size

    x1 = size * (cos(y) * cos(r)) + face_x
    y1 = size * (cos(p) * sin(r) + cos(r) * sin(p) * sin(y)) + face_y
    x2 = size * (-cos(y) * sin(r)) + face_x
    y2 = size * (cos(p) * cos(r) - sin(p) * sin(y) * sin(r)) + face_y
    x3 = size * (sin(y)) + face_x
    y3 = size * (-cos(y) * sin(p)) + face_y

    # Draw base in red
    cv2.line(img, (int(face_x), int(face_y)), (int(x1),int(y1)),(0,0,255),3)
    cv2.line(img, (int(face_x), int(face_y)), (int(x2),int(y2)),(0,0,255),3)
    cv2.line(img, (int(x2), int(y2)), (int(x2+x1-face_x),int(y2+y1-face_y)),(0,0,255),3)
    cv2.line(img, (int(x1), int(y1)), (int(x1+x2-face_x),int(y1+y2-face_y)),(0,0,255),3)
    # Draw pillars in blue
    cv2.line(img, (int(face_x), int(face_y)), (int(x3),int(y3)),(255,0,0),2)
    cv2.line(img, (int(x1), int(y1)), (int(x1+x3-face_x),int(y1+y3-face_y)),(255,0,0),2)
    cv2.line(img, (int(x2), int(y2)), (int(x2+x3-face_x),int(y2+y3-face_y)),(255,0,0),2)
    cv2.line(img, (int(x2+x1-face_x),int(y2+y1-face_y)), (int(x3+x1+x2-2*face_x),int(y3+y2+y1-2*face_y)),(255,0,0),2)
    # Draw top in green
    cv2.line(img, (int(x3+x1-face_x),int(y3+y1-face_y)), (int(x3+x1+x2-2*face_x),int(y3+y2+y1-2*face_y)),(0,255,0),2)
    cv2.line(img, (int(x2+x3-face_x),int(y2+y3-face_y)), (int(x3+x1+x2-2*face_x),int(y3+y2+y1-2*face_y)),(0,255,0),2)
    cv2.line(img, (int(x3), int(y3)), (int(x3+x1-face_x),int(y3+y1-face_y)),(0,255,0),2)
    cv2.line(img, (int(x3), int(y3)), (int(x3+x2-face_x),int(y3+y2-face_y)),(0,255,0),2)

    return img
```

After detecting the face and drawing axis, now let's make some augmentation which is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. In our task we didn't want to ruin the images for the model so we were careful choosing the defect methods, we chose only flipping and images blur.

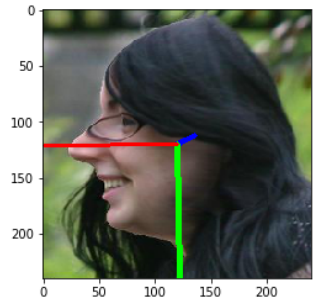
```python
        # Flip Augmentaion
        rnd = np.random.random_sample()
        if rnd < 0.5 and self.augment:
            yaw = -yaw
            roll = -roll
            img = img.transpose(Image.FLIP_LEFT_RIGHT)

        # Blur Augmentaion
        rnd = np.random.random_sample()
        if rnd < 0.05 and self.augment:
            img = img.filter(ImageFilter.BLUR)
```

After preparing the images for the model let's check the images and the effect of the augmentation!

```
# test dataset
dataset = Pose_300W_LP(data_dir = 'dataset/300W_LP', filename_path = 'dataset/300W_LP/filename_list.txt', transform = None)
dl = DataLoader(dataset, batch_size=1, shuffle=True)

for img, dig_labels, exact_labels, img_path in dl:
    img, dig_labels, exact_labels, img_path = img[0], dig_labels[0], exact_labels[0], img_path[0]
    # draw the head pose estimation axes
    image = img.permute((1,2,0)).numpy()
    image = cv2.resize(image, dsize=(240, 240), interpolation=cv2.INTER_CUBIC)
    #image = cv2.imread(os.path.join('dataset/300W_LP', img_path + '.jpg'))
    print(image.shape, type(image))
    draw_axis(image, exact_labels[0], exact_labels[1], exact_labels[2],
              tdx = image.shape[1] / 2, tdy= (image.shape[0]) / 2, size = image.shape[0]/2)
    plt.imshow(image)
    break
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

```
(240, 240, 3) <class 'numpy.ndarray'>
```

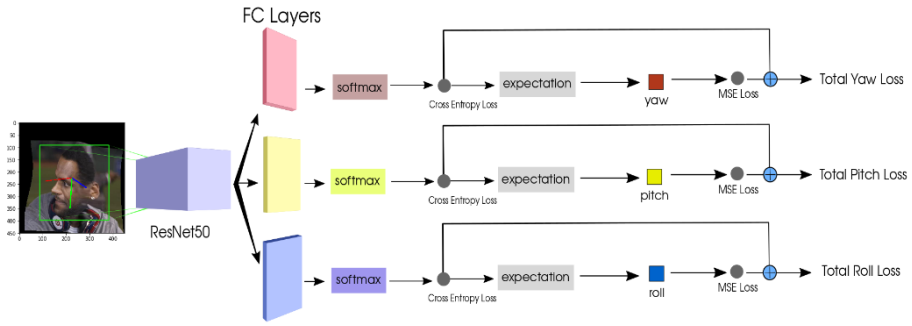


Model Architecture

Now that we've set everything let's train our Head Pose model! Let's load the packages that will allow us to use all different kinds of functions in Pytorch.

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torchvision
import torch.utils.model_zoo as model_zoo

import matplotlib.pyplot as plt
```

But before implementing the model let's first describe what's going on exactly to the images in the model and how the model predict the 3 different angles of the face (Yaw,Pitch,Roll).



Let's walk through the journey of the image from the beginning until the angles detections, first thing is face detection which is done before entre the very first layer of our model architecture. Then the face goes through ResNet50 (Fig. 7). Then the architecture divided to 3 branches every branch is responsible for an angle of the three angles. Each branch goes through a classification phase in which we have 66 classes! But why 66 classes? Because every class represent a 3 degree of the 198 degrees that an angle can be set! The minimum degree that an angle can be set to is -99 and the maximum degree is +99 so we have 198 degrees of each angle! First it goes through a softmax which classify the angle to which 3 degrees it could be then goes through mean square error (MSE) as it detect which degree specifically the angle is. Of course this happens in each branch of the model.

And here is the model implementation:

```
class Hopenet(nn.Module):
    # Hopenet with 3 output layers for yaw, pitch and roll
    # Predicts Euler angles by binning and regression with the expected value
    def __init__(self, block, layers, num_bins):
        self.inplanes = 64
        super(Hopenet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(7)
        self.fc_yaw = nn.Linear(512 * block.expansion, num_bins)
        self.fc_pitch = nn.Linear(512 * block.expansion, num_bins)
        self.fc_roll = nn.Linear(512 * block.expansion, num_bins)

        # Vestigial layer from previous experiments
        self.fc_finetune = nn.Linear(512 * block.expansion + 3, 3)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        pre_yaw = self.fc_yaw(x)
        pre_pitch = self.fc_pitch(x)
        pre_roll = self.fc_roll(x)

        return pre_yaw, pre_pitch, pre_roll
```

```
# intialize model
# ResNet50 structure
model = Hopenet(torchvision.models.resnet.Bottleneck, [3, 4, 6, 3], 66)
model.to(device)
softmax = nn.Softmax(dim=1).to(device)
```

Now let's get our hands dirty and start our training we set some variables like number of epochs and learning rate and weight loss (alpha) and batch size of the training.

```
num_epochs = 25
batch_size = 16
device = 'cuda' if torch.cuda.is_available() else 'cpu'
checkpoints_file = 'checkpoints/headpose'
last_checkpoint = f'{checkpoints_file}/headpose_last.pth'
best_checkpoint = f'{checkpoints_file}/headpose_best.pth'
best_checkpoint_acc = f'{checkpoints_file}/headpose_best_acc.pth'
# create checkpoints file
os.makedirs(checkpoints_file, exist_ok=True)
# dataset variables
data_dir = 'dataset/300W_LP'
filename_list_train = f'{data_dir}/filename_list_train.txt'
filename_list_val = f'{data_dir}/filename_list_val.txt'

lr = 0.00001

best_loss = 1e6
best_acc = 0

alpha = 1
```

And setting the image scaling and preparing the data loader.

```
# dataset transformer
transformations = {}
transformations['train'] = transforms.Compose([transforms.Resize(240),
                                               transforms.RandomCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

transformations['val'] = transforms.Compose([transforms.Resize(240),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])
# create training dataloader
pose_train = Pose_300W_LP(data_dir, filename_list_train, transform=transformations['train'], augment=True)
train_loader = DataLoader(pose_train, batch_size=batch_size, shuffle=True)

# create val dataloader
pose_val = Pose_300W_LP(data_dir, filename_list_val, transform=transformations['val'], augment=False)
val_loader = DataLoader(pose_val, batch_size=batch_size, shuffle=False)
```

And setting the optimizer to 'Adam' and changing the learning rate over the training.

```
# optimizer, set three diffrent lr through the network
optimizer = torch.optim.Adam([{'params': get_ignored_params(model), 'lr': 0},
                              {'params': get_non_ignored_params(model), 'lr': lr},
                              {'params': get_fc_params(model), 'lr': lr * 5}],
                               lr = lr)
```

And preparing the output class (from 66 classes) to go through one more regression phase to detect exactly what degree the angle is by checking the probability distribution of the class (3 degrees).

```
def get_reg_form_cls(pred, prop_func, device):
    # compute predected prop
    pred_prop = softmax(pred)
    # get regression value for proplities
    idx_tensor = torch.FloatTensor([idx for idx in range(66)]).to(device)
    reg_pred = torch.sum(pred_prop * idx_tensor, dim=1) * 3 - 99 # [-99, 99]
    return reg_pred
```

And here is the training code.

```python
def train_epoch(model, prop_func, optimizer, train_loader, logger, epoch_num,
                cls_criterion, reg_criterion, disp_every, device):

    # values to compute
    total_loss = 0
    total_loss_perAng = [0, 0, 0] # yaw, pitch, roll
    total_acc_perAng = [0, 0, 0] # yaw, pitch, roll
    dataset_size = len(train_loader.dataset)

    for i, (img, bins_angles, angles, img_name) in enumerate(train_loader):

        # load to device
        img = img.to(device)
        angles = angles.to(device)
        bins_angles = bins_angles.to(device)

        # pass image to the model
        pred_yaw, pred_pitch, pred_roll = model(img)

        # compute classification losses
        loss_cls_yaw = cls_criterion(pred_yaw, bins_angles[:, 0])
        loss_cls_pitch = cls_criterion(pred_pitch, bins_angles[:, 1])
        loss_cls_roll = cls_criterion(pred_roll, bins_angles[:, 2])

        # get the regression values for the output of the model
        reg_yaw = get_reg_form_cls(pred_yaw, prop_func, device)
        reg_pitch = get_reg_form_cls(pred_pitch, prop_func, device)
        reg_roll = get_reg_form_cls(pred_roll, prop_func, device)

        # computer regression losses
        loss_reg_yaw = reg_criterion(reg_yaw, angles[:, 0])
        loss_reg_pitch = reg_criterion(reg_pitch, angles[:, 1])
        loss_reg_roll = reg_criterion(reg_roll, angles[:, 2])

        # total losses
        loss_yaw = loss_cls_yaw + alpha * loss_reg_yaw
        loss_pitch = loss_cls_pitch + alpha * loss_reg_pitch
        loss_roll = loss_cls_roll + alpha * loss_reg_roll

        # backprop and optimize
        loss = loss_yaw + loss_pitch + loss_roll
        optimizer.zero_grad()
        loss.backward()
        # grad clipping, (max_norm=5.0)
        #torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)
        optimizer.step()

        # compute total loss
        total_loss += loss.cpu().detach().numpy()
        total_loss_perAng[0] += loss_yaw.cpu().detach().numpy() * batch_size
        total_loss_perAng[1] += loss_pitch.cpu().detach().numpy() * batch_size
        total_loss_perAng[2] += loss_roll.cpu().detach().numpy() * batch_size

        # compute Accuracy
        total_acc_perAng[0] += torch.sum(torch.abs(reg_yaw - angles[:, 0])).cpu().detach().numpy()
        total_acc_perAng[1] += torch.sum(torch.abs(reg_pitch - angles[:, 1])).cpu().detach().numpy()
        total_acc_perAng[2] += torch.sum(torch.abs(reg_roll - angles[:, 2])).cpu().detach().numpy()

        # print results and save to logger
        if i % disp_every == 0:
            print(f'Epoch[{epoch_num}], Itr[{i}/{len(train_loader)}], \
            Losses: Yaw {loss_yaw:.2f}, Pitch {loss_pitch:.2f}, Roll {loss_roll:.2f}')
```

Results

First on the training dataset:

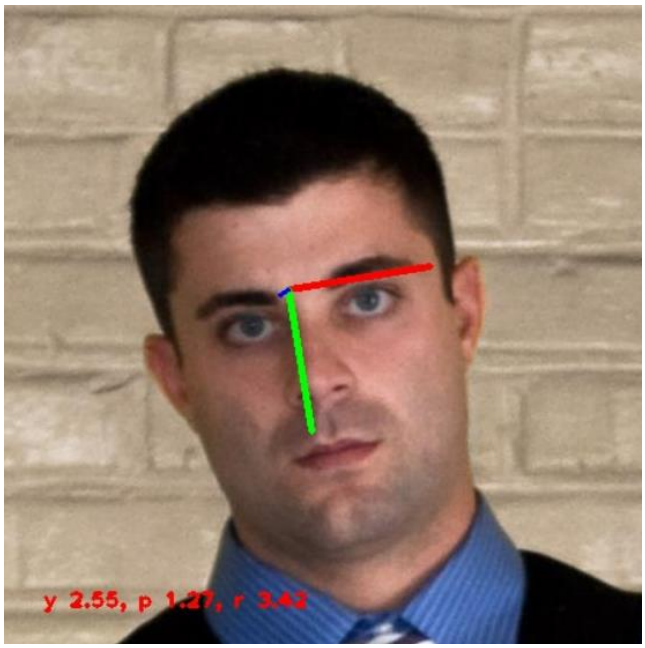
Mean average error of Euler angles on the 300W-LP validation dataset

Losses per angle: Yaw 2.43, Pitch 3.10, Roll 2.73

Second on the testing dataset:

Mean average error of Euler angles on the AFLW2000 dataset (alpha = 1)

Losses per angle: Yaw 7.30, Pitch 6.28, Roll 5.04, MAE 6.20



Live testing

In this work we showed that a multi-loss deep network can directly, accurately and robustly predict head rotation from image intensities. We show that such a network outperforms landmark-to-pose methods using state-of-the-art landmark detection methods. Landmark-to-pose methods are studied in this work to show their dependence on extraneous factors such as head model and landmark detection accuracy.

There is some future work can be added to this model to make it work even better, we actually working on some ideas that will do so, like using some addition task like face segmentation end to end with predicting the angles as we believe that the two different tasks could actually have a better feedback to each other and we could predict the angles with more accuracy without needing some face recognition to be done for us before the training.

And also Synthetic data generation for extreme poses seems to be a way to improve performance for the proposed method as are studies into more intricate network architectures that might take into account full body pose for example.

PART 03

NATURAL LANGUAGE PROCESSING

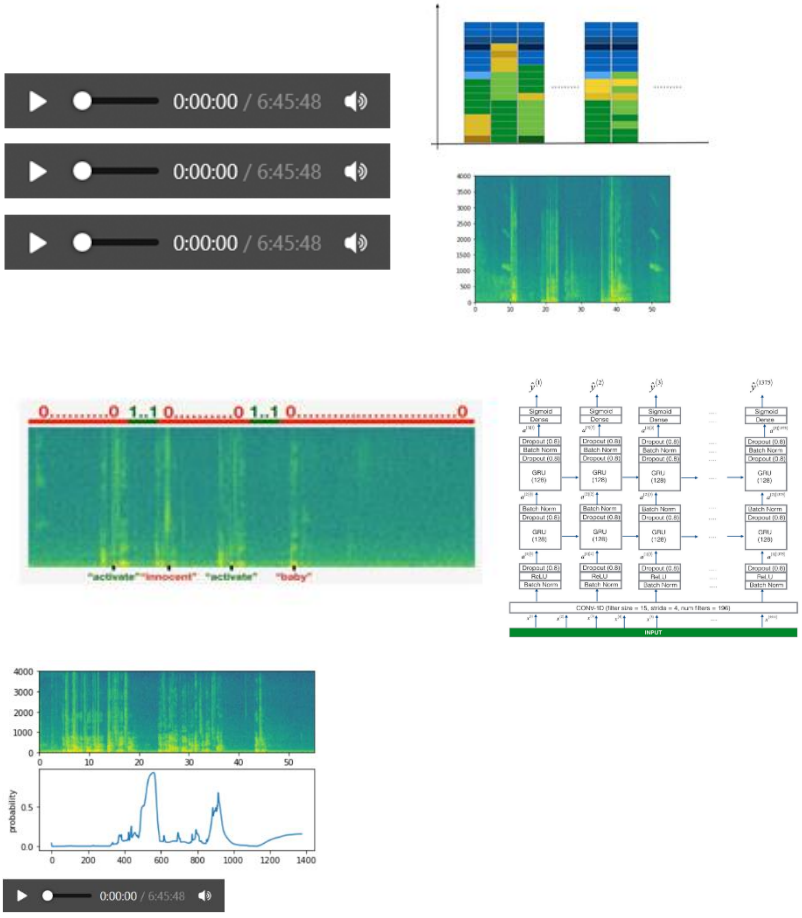
CHAPTER FIVE

TRIGGER WORD

ESSENTIALLY, ANY WORD THAT GETS SOMEONE TO DO SOMETHING — ANYTHING — CAN BE DEFINED AS A TRIGGER WORD.

Pipeline

This is the trigger word detection, aka. Wake / hot word detection. Like when you yell at Amazon Alexa or Google Home to wake them up. Trigger word detection is the technology that allows devices like Amazon Alexa, Google Home, Apple Siri, and Baidu DuerOS to wake up upon hearing a certain word. This is the UI of our project, it's how you get the project to start working.



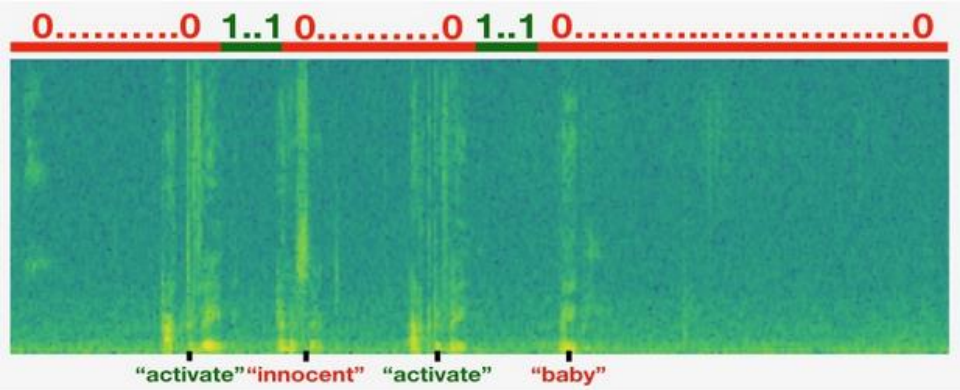
To make things easier let us first thread the overall pipeline or all the steps was done in this model. We started by creating the speech dataset to work on, For the sake of simplicity, let's take the word "Activate" as our trigger word. The training dataset needs to be as similar to the real test environment as possible. For example, the model needs to be exposed to non-trigger words and background noise in the speech during training so it will not generate the trigger signal when we say other words or there is only background noise. We generated different kinds of audio recordings with different kinds of backgrounds (Fig. 1), after

generating (creating) the speech dataset we had to transform the audio recordings to spectrogram using Fourier transformation (Fig. 2) to construct a proper format to be the input of the model which we will discuss later with details. Now all sets to generate our first single training example (Fig. 3), after that we generated our full training set and development set. Now we designed our model (Fig. 4) and we started the training on a proper GPU for a proper amount of time. After finishing the training, we started to test our model and make predictions, then we formed our results (Fig. 5) and our model evaluation. And at last when we had a good model evaluation we started the live testing with our own voice and environment. And when everything is set we linked the whole model with the other models in the project. So now when this model triggered by the 'Activate' word the whole project is working.

Dataset

The training dataset needs to be as similar to the real test environment as possible. For example, the model needs to be exposed to non-trigger words and background noise in the speech during training so it will not generate the trigger signal when we say other words or there is only background noise.
As you may expect training a good speech model requires a lot of labeled training samples. Do we just have to record each audio and label where the trigger words were spoken? Here is a simple trick to solve this problem. We generate them!
First, we have 3 types of audio recordings:

- Recordings of different backgrounds audios. They might just as simple as two clips of background noise, 10 seconds each, coffee shop, and living room.
- Recordings of the trigger word "activate". They might be just you speaking the word 10 times in different tones, 1 second each.
- Recordings of the negative words. They might be you speaking other words like "baby", "coffee", 1 second for each recording.



Here is the step to generate the training input audio clips (fig. 6)

- Pick a random 10-second background audio clip
- Randomly overlay 0-4 audio clips of "activate" into this 10sec clip
- Randomly overlay 0-2 audio clips of negative words into this 10sec clip

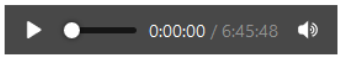
We choose overlay since we want to mix the spoken words with the background noise to sounds more realistic. For the output labels, we want it to represent whether or not someone has just finished saying "activate".

Model Details

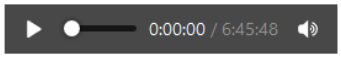
As mentioned in the pipeline we started the model with the data (audio recordings) preprocessing and setting it for the model.
First things first, we start by listening to some samples and transform them to spectrogram.
We used audio sampled at 44100 Hz (or 44100 Hertz). This means the microphone gives us 44100 numbers per second. Thus, a 10 second audio clip is represented by 441000 numbers.
And we will be working with 10 second audio clips as the "standard length" for our training examples. The number of time-steps of the spectrogram will be 5511.
So the time steps in the audio recording before spectrogram is 441000, and the time steps in input of the model (after the spectrogram) is (101, 5511)
Note that: '10 seconds of time can be discretized to different numbers of value. You've seen 441000 (raw audio) and 5511 (spectrogram). In the former case, each step represents

10/441000≈0.00002310 seconds. In the second case, each step represents 10/5511≈0.001810 seconds.'.

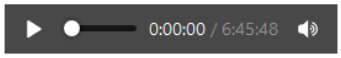
```
# We will use audio sampled at 44100 Hz (or 44100 Hertz).
IPython.display.Audio("./raw_data/activates/1.wav")
```



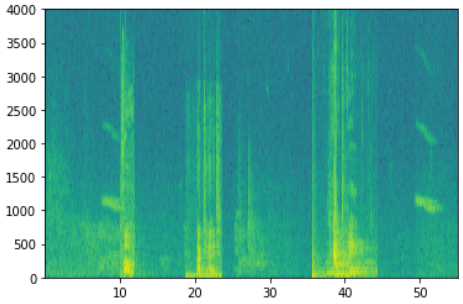

```
IPython.display.Audio("./raw_data/negatives/4.wav")
```




```
#From audio recordings to spectrograms
IPython.display.Audio("audio_examples/example_train.wav")
```




```
x = graph_spectrogram("audio_examples/example_train.wav")
```



So the key values in this model:

- 441000 (raw audio)
- 5511=Tx (spectrogram output, and dimension of input to the neural network).
- 10000 (used by the pydub module to synthesize audio)
- 1375=Ty (the number of steps in the output of the GRU you'll build).

```
# we will be working with 10 second audio clips as the "standard length" for our training examples. The number of timesteps o

_, data = wavfile.read("audio_examples/example_train.wav")
print("Time steps in audio recording before spectrogram", data[:,0].shape)
print("Time steps in input after spectrogram", x.shape)
```

```
Time steps in audio recording before spectrogram (441000,)
Time steps in input after spectrogram (101, 5511)
```

Now after generating the spectrogram our mission now is to create our first training sample, and we will do that with the help of some helper functions that we wrote:

- get_random_time_segment
    - Gets a random time segment of duration 'segment_ms' in a 10,000ms audio clip.

```
def get_random_time_segment(segment_ms):
    """
    Gets a random time segment of duration 'segment_ms' in a 10,000 ms audio clip.

    Arguments:
    segment_ms -- the duration of the audio clip in ms ("ms" stands for "milliseconds")

    Returns:
    segment_time -- a tuple of (segment_start, segment_end) in ms
    """

    segment_start = np.random.randint(low=0, high=10000-segment_ms)   # Make sure segment doesn't run past the 10sec backgrou
    segment_end = segment_start + segment_ms - 1

    return (segment_start, segment_end)
```

- is_overlapping
    - Checks if the time of a segment overlaps with the times of existing segments.

```
def is_overlapping(segment_time, previous_segments):
    """
    Checks if the time of a segment overlaps with the times of existing segments.

    Arguments:
    segment_time -- a tuple of (segment_start, segment_end) for the new segment
    previous_segments -- a list of tuples of (segment_start, segment_end) for the existing segments

    Returns:
    True if the time segment overlaps with any of the existing segments, False otherwise
    """

    segment_start, segment_end = segment_time
    overlap = False

    # Compare start/end times and set the flag to True if there is an overlap (≈ 3 lines)
    for previous_start, previous_end in previous_segments:
        if segment_start <= previous_end and segment_end >= previous_start:
            overlap = True

    return overlap
```

- insert_audio_clipds
    - Insert a new audio segment over the background noise at a random time step, ensuring that the audio segment does not overlap with existing segments.

```
def insert_audio_clip(background, audio_clip, previous_segments):
    """
    Insert a new audio segment over the background noise at a random time step, ensuring that the
    audio segment does not overlap with existing segments.

    Arguments:
    background -- a 10 second background audio recording.
    audio_clip -- the audio clip to be inserted/overlaid.
    previous_segments -- times where audio segments have already been placed

    Returns:
    new_background -- the updated background audio
    """

    # Get the duration of the audio clip in ms
    segment_ms = len(audio_clip)

    # the new audio clip.
    segment_time = get_random_time_segment(segment_ms)

    # picking new segment_time at random until it doesn't overlap.
    while is_overlapping(segment_time, previous_segments):
        segment_time = get_random_time_segment(segment_ms)

    # Adding the new segment_time to the list of previous_segments.
    previous_segments.append(segment_time)

    # Superpose audio segment and background
    new_background = background.overlay(audio_clip, position = segment_time[0])

    return new_background, segment_time
```

- insert_ones
    - Update the label vector y. The labels of the 50 output steps strictly after the end of the segment should be set to 1. By strictly we mean that the label of segment_end_y should be 0 while, the 50 following labels should be ones.

```
def insert_ones(y, segment_end_ms):
    """
    Update the label vector y. The labels of the 50 output steps strictly after the end of the segment
    should be set to 1. By strictly we mean that the label of segment_end_y should be 0 while, the
    50 followinf labels should be ones.


    Arguments:
    y -- numpy array of shape (1, Ty), the labels of the training example
    segment_end_ms -- the end time of the segment in ms

    Returns:
    y -- updated labels
    """

    # duration of the background (in terms of spectrogram time-steps)
    segment_end_y = int(segment_end_ms * Ty / 10000.0)

    # Add 1 to the correct index in the background label (y)
    for i in range(segment_end_y+1, segment_end_y+51):
        if i < Ty:
            y[0, i] = 1.0
    return y
```

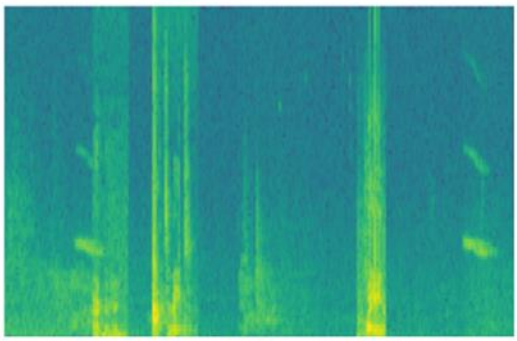
Now everything is set for us to create the training example.

```
def create_training_example(background, activates, negatives):
    """
    Creates a training example with a given background, activates, and negatives.
    Arguments:
    background -- a 10 second background audio recording
    activates -- a list of audio segments of the word "activate"
    negatives -- a list of audio segments of random words that are not "activate"
    Returns:
    x -- the spectrogram of the training example
    y -- the label at each time step of the spectrogram
    """
    # Make background quieter
    background = background - 20

    # Initialize y (label vector) of zeros
    y = np.zeros((1, Ty))

    # Initialize segment times as empty list
    previous_segments = []

    # Select 0-4 random "activate" audio clips from the entire list of "activates" recordings
    number_of_activates = np.random.randint(0, 5)
    random_indices = np.random.randint(len(activates), size=number_of_activates)
    random_activates = [activates[i] for i in random_indices]

    # Loop over randomly selected "activate" clips and insert in background
    for random_activate in random_activates:
        # Insert the audio clip on the background
        background, segment_time = insert_audio_clip(background, random_activate, previous_segments)
        # Retrieve segment_start and segment_end from segment_time
        segment_start, segment_end = segment_time
        # Insert labels in "y"
        y = insert_ones(y, segment_end)

    # Select 0-2 random negatives audio recordings from the entire list of "negatives" recordings
    number_of_negatives = np.random.randint(0, 3)
    random_indices = np.random.randint(len(negatives), size=number_of_negatives)
    random_negatives = [negatives[i] for i in random_indices]

    # Loop over randomly selected negative clips and insert in background
    for random_negative in random_negatives:
        # Insert the audio clip on the background
        background, _ = insert_audio_clip(background, random_negative, previous_segments)

    # Standardize the volume of the audio clip
    background = match_target_amplitude(background, -20.0)

    # Export new training example
    file_handle = background.export("train" + ".wav", format="wav")
    print("File (train.wav) was saved in your directory.")

    # Get and plot spectrogram of the new recording (background with superposition of positive and negatives)
    x = graph_spectrogram("train.wav")

    return x, y
```

```
x, y = create_training_example(backgrounds[0], activates, negatives)
```

```
File (train.wav) was saved in your directory.
```

```
/Users/poudel/Library/Enthought/Canopy/edm/envs/deeplr/lib/python3.6/site-packages/matplotlib/axes/_axes.py:7674: RuntimeWa
rning: divide by zero encountered in log10
  Z = 10. * np.log10(spec)
```



We've now implemented the code needed to generate a single training example. We used this process to generate a large training set. To save time, we've already generated a set of training examples in XY_train and another small dataset in XY_dev.

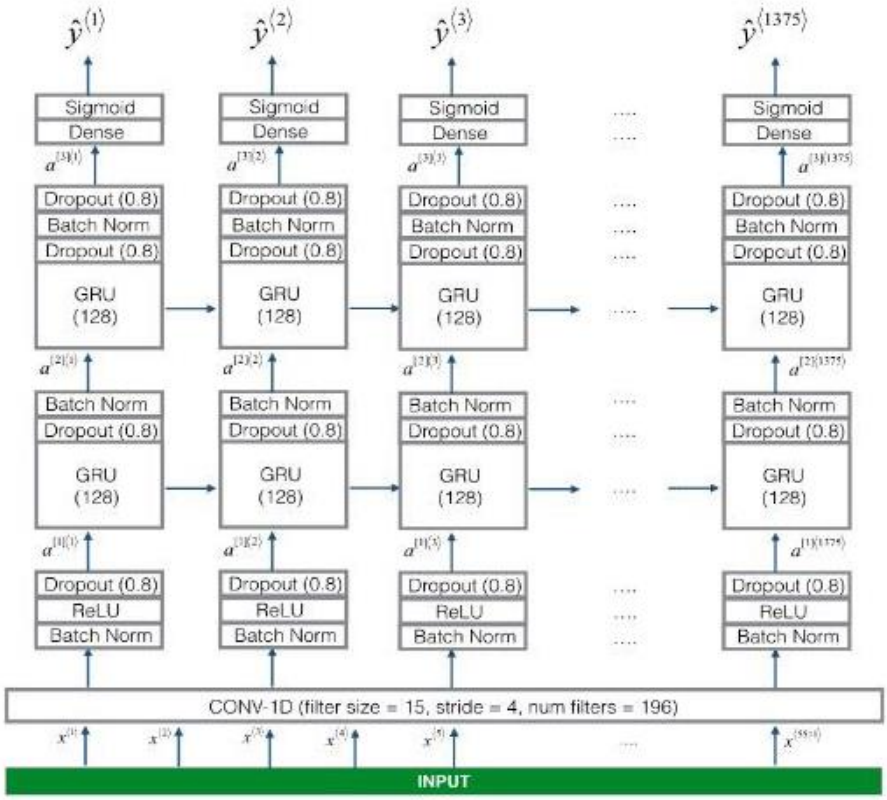
Model Architecture

Now that you've built a dataset, let's write and train a trigger word detection model!

The model will use 1-D convolutional layers, GRU layers, and dense layers. Let's load the packages that will allow you to use these layers in Keras.

```
from keras.callbacks import ModelCheckpoint
from keras.models import Model, load_model, Sequential
from keras.layers import Dense, Activation, Dropout, Input, Masking, TimeDistributed, LSTM, Conv1D
from keras.layers import GRU, Bidirectional, BatchNormalization, Reshape
from keras.optimizers import Adam
```

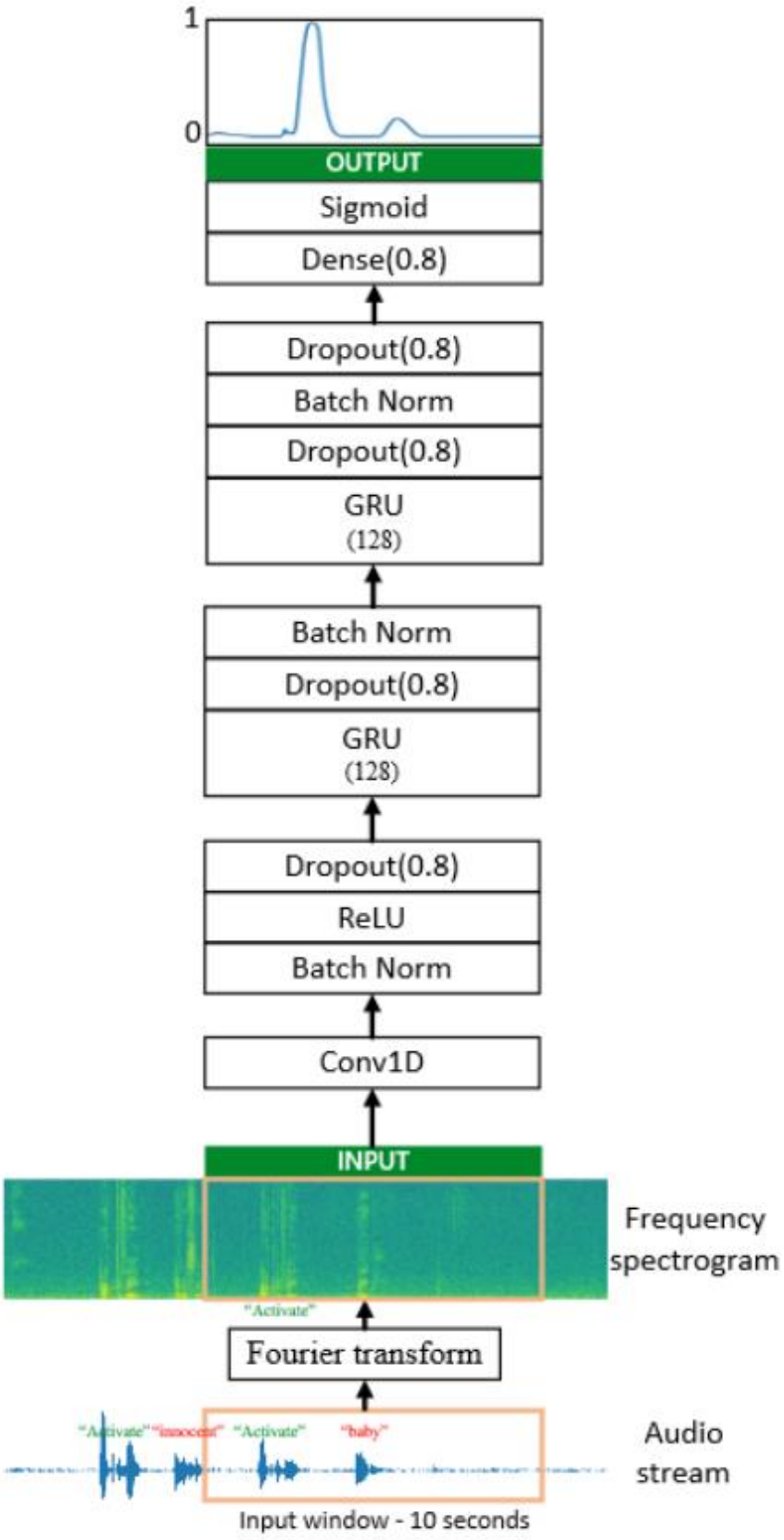
```
Using TensorFlow backend.
```



One key step of this model is the 1D convolutional step (near the bottom of Fig. 7). It inputs the 5511 step spectrogram, and outputs

a 1375 step output, which is then further processed by multiple layers to get the final Ty=1375 step output. This layer plays a role similar to the 2D convolutions, Extracting low-level features and then possibly generating an output of a smaller dimension.
Computationally, the 1-D conv layer also helps speed up the model because now the GRU has to process only 1375 time-steps rather than 5511 time-steps. The two GRU layers read the sequence of inputs from left to right, then ultimately uses a dense + sigmoid layer to make a prediction for $y^{\langle t \rangle}$. Because y is binary valued (0 or 1), we use a sigmoid output at the last layer to estimate the chance of the output being 1, corresponding to the user having just said "activate."
Note that we use a uni-directional RNN rather than a bi-directional RNN. This is really important for trigger word detection, since we want to be able to detect the trigger word almost immediately after it is said. If we used a bi-directional RNN, we would have to wait for the whole 10sec of audio to be recorded before we could tell if "activate" was said in the first second of the audio clip.

Looks a bit intimidating but let's try to break it down and understand the whole process (Fig. 8) from the first thing at the beginning, audio stream goes through a Fourier transformer then form the spectrogram and goes through the layers till we form the output which is the spike in the figure if the model detected the wake up word which is 'Activate'.
Now let's see the implementation part.


1
0
OUTPUT
Sigmoid
Dense(0.8)
Dropout(0.8)
Batch Norm
Dropout(0.8)
GRU
(128)
Batch Norm
Dropout(0.8)
GRU
(128)
Dropout(0.8)
ReLU
Batch Norm
Conv1D
INPUT
Frequency
spectrogram
"Activate"
Fourier transform
"Activate" "innocent" "Activate" "baby"
Audio
stream
Input window - 10 seconds

```
def model(input_shape):
    """
    Function creating the model's graph in Keras.

    Argument:
    input_shape -- shape of the model's input data (using Keras conventions)

    Returns:
    model -- Keras model instance
    """

    X_input = Input(shape = input_shape)


    X = Conv1D(196, 15, strides=4)(X_input)                    # CONV1D
    X = BatchNormalization()(X)                                # Batch normalization
    X = Activation('relu')(X)                                  # ReLu activation
    X = Dropout(0.8)(X)                                        # dropout (use 0.8)

    X = GRU(units = 128, return_sequences=True)(X)             # GRU (use 128 units and return the sequences)
    X = Dropout(0.8)(X)                                        # dropout (use 0.8)
    X = BatchNormalization()(X)                                # Batch normalization

    X = GRU(units = 128, return_sequences=True)(X)             # GRU (use 128 units and return the sequences)
    X = Dropout(0.8)(X)                                        # dropout (use 0.8)
    X = BatchNormalization()(X)                                # Batch normalization
    X = Dropout(0.8)(X)                                        # dropout (use 0.8)

    X = TimeDistributed(Dense(1, activation = "sigmoid"))(X) # time distributed  (sigmoid)


    model = Model(inputs = X_input, outputs = X)

    return model
```

Keras allows us to see the summary of our model. It shows the layers their shapes and the number of parameters it has.
And as you can see that our model has about 500K trainable parameters. There are a few steps required to train the model.

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 5511, 101)         0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 1375, 196)         297136
_________________________________________________________________
batch_normalization_1 (Batch (None, 1375, 196)         784
_________________________________________________________________
activation_1 (Activation)    (None, 1375, 196)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1375, 196)         0
_________________________________________________________________
gru_1 (GRU)                  (None, 1375, 128)         124800
_________________________________________________________________
dropout_2 (Dropout)          (None, 1375, 128)         0
_________________________________________________________________
batch_normalization_2 (Batch (None, 1375, 128)         512
_________________________________________________________________
gru_2 (GRU)                  (None, 1375, 128)         98688
_________________________________________________________________
dropout_3 (Dropout)          (None, 1375, 128)         0
_________________________________________________________________
batch_normalization_3 (Batch (None, 1375, 128)         512
_________________________________________________________________
dropout_4 (Dropout)          (None, 1375, 128)         0
_________________________________________________________________
time_distributed_1 (TimeDist (None, 1375, 1)           129
=================================================================
Total params: 522,561
Trainable params: 521,657
Non-trainable params: 904
```

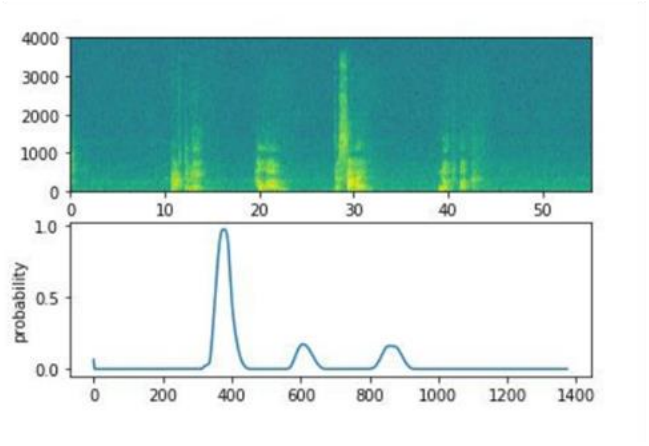
Results

Now we have a trained model to detect our trigger word from an audio file. Let's see how to do the prediction.

```
def detect_triggerword(filename):
    plt.subplot(2, 1, 1)

    x = graph_spectrogram(filename)
    # the spectogram outputs (freqs, Tx) and we want (Tx, freqs) to input into the model
    x  = x.swapaxes(0,1)
    x = np.expand_dims(x, axis=0)
    predictions = model.predict(x)

    plt.subplot(2, 1, 2)
    plt.plot(predictions[0,:,0])
    plt.ylabel('probability')
    plt.show()
    return predictions
```

The above function returns the predictions in a form of a vector of values between 0 and 1 depicting the probabilities of presence of a trigger word at a time step. If we plot the spectrogram and the output prediction values, we will see a peak depicting that the model has detected a trigger word.

```
filename  = "./raw_data/dev/2.wav"
prediction = detect_triggerword(filename)
chime_on_activate(filename, prediction, 0.5)
IPython.display.Audio("./chime_output.wav")
```



You can insert an audio cue, blink a light, print something on your screen, or maybe connect it to your lamp to light it just after the trigger word is detected. On the other hand, in our project we connected it to the other models as a UI to the whole project.

Live testing

So far our model can only take a static 10 seconds audio clip and make the prediction of the trigger word location.

Here is the fun part, let's replace with the live audio stream instead!

The model we have built expect 10 seconds audio clips as input. While training another model that takes shorter audio clips is possible but needs us retraining the model on a GPU for several hours.

We also don't want to wait for 10-second for the model tells us the trigger word is detected. So one solution is to have a moving 10 seconds audio stream window with a step size of 0.5 second. Which means we ask the model to predict every 0.5 seconds, that reduce the delay and make it responsive.

We also add the silence detection mechanism to skip prediction if the loudness is below a threshold, this can save some computing power. Let's see how we built it!

The input 10 seconds audio is updated every 0.5 second. Meaning for every 0.5 second, the oldest 0.5 second chunk of audio will be discarded and the fresh 0.5 second audio will be shifted in. The job of the model is to tell if there is a new trigger word detected in the fresh 0.5 second audio chunk.

And here is the code to make it happen.

```
def has_new_triggerword(predictions, chunk_duration, feed_duration, threshold=0.5):
    """
    Function to detect new trigger word in the latest chunk of input audio.
    It is looking for the rising edge of the predictions data belongs to the
    last/latest chunk.

    Argument:
    predictions -- predicted labels from model
    chunk_duration -- time in second of a chunk
    feed_duration -- time in second of the input to model
    threshold -- threshold for probability above a certain to be considered positive

    Returns:
    True if new trigger word detected in the latest chunk
    """
    predictions = predictions > threshold
    chunk_predictions_samples = int(len(predictions) * chunk_duration / feed_duration)
    chunk_predictions = predictions[-chunk_predictions_samples:]
    level = chunk_predictions[0]
    for pred in chunk_predictions:
        if pred > level:
            return True
        else:
            level = pred
    return False
```

To get the audio stream, we use the pyaudio library. Which has an option to read the audio stream asynchronously. That means the audio stream recording happens in another thread and when a new fixed length of audio data is available, it notifies our model to process it in the main thread.

You may ask why not just read a fixed length of audio and just process it in one function?

Since for the model to generate the prediction, it takes quite some time, sometimes measured in tens of milliseconds. By doing so, we are risking creating gaps in the audio stream while we are doing the computation.

Here is the code for the pyaudio library's callback, in the callback function we send a queue to notify the model to process the data in the main thread.

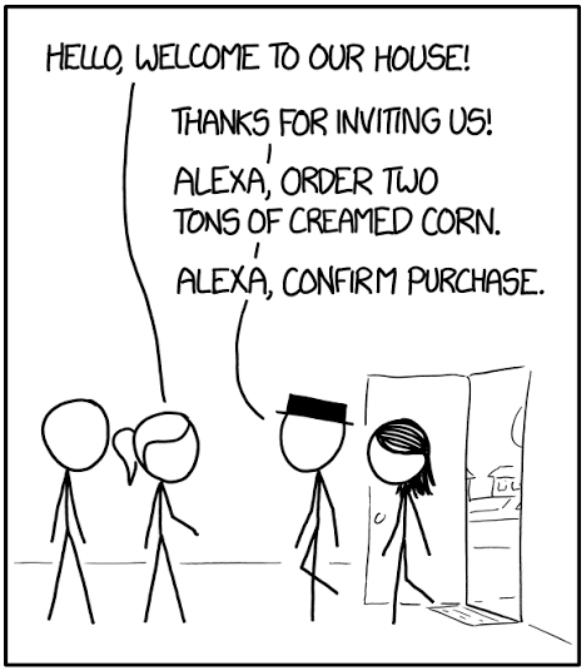
```
import pyaudio
from queue import Queue
from threading import Thread
import sys
import time
from matplotlib import mlab
```

```
# Queue to communiate between the audio callback and main thread
q = Queue()
run = True
silence_threshold = 100
trigger_word = False
# Run the demo for a timeout seconds
timeout = time.time() + 0.5*60  # 0.5 minutes from now

# Data buffer for the input wavform
data = np.zeros(feed_samples, dtype='int16')

def callback(in_data, frame_count, time_info, status):
    global run, timeout, data, silence_threshold
    if time.time() > timeout:
        run = False
    data0 = np.frombuffer(in_data, dtype='int16')
    if np.abs(data0).mean() < silence_threshold:
        sys.stdout.write('-')
        return (in_data, pyaudio.paContinue)
    else:
        sys.stdout.write('.')
    data = np.append(data,data0)
    if len(data) > feed_samples:
        data = data[-feed_samples:]
        # Process data async by sending a queue.
        q.put(data)
    return (in_data, pyaudio.paContinue)

stream = get_audio_input_stream(callback)
stream.start_stream()


try:
    while run:
        data = q.get()
        spectrum = get_spectrogram(data)
        preds = detect_triggerword_spectrum(spectrum)
        new_trigger = has_new_triggerword(preds, chunk_duration, feed_duration)
        if new_trigger:
            sys.stdout.write('1')
            trigger_word = True
except (KeyboardInterrupt, SystemExit):
    stream.stop_stream()
    stream.close()
    timeout = time.time()
    run = False

stream.stop_stream()
stream.close()
```

When you run it, it outputs one of the 3 characters every 0.5 second. "-" means silence, "." means not silence and no trigger word, "1" means a new trigger word is detected.

```
--.--.....-1----.-..--...-1---..-------..1---------..-1---------.----.--------.----.---.--
```

Now that you know how the trigger word detection works. With this new knowledge you can now ask the question, is this product listens to all your conversations?



Further work:

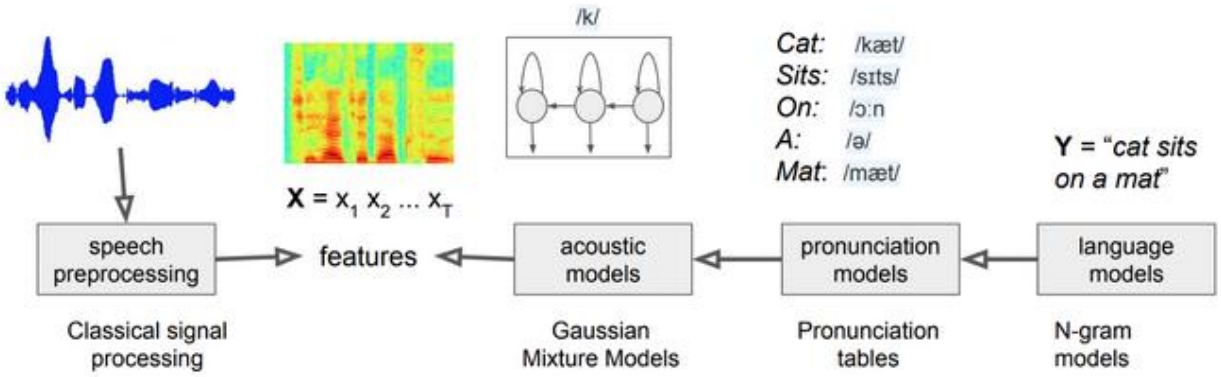
After finishing the trigger word detection, you can retrain it with another wake up word other than 'Activate' and make your own model and link it to anything thing that you want! Mabey you can open your computer with your voice or give permission to an application to start running.

CHAPTER SIX

AUTOMATIC SPEECH RECOGNITION

THE COMPUTER TAKES IN THE WAVEFORM OF YOUR SPEECH. THEN IT BREAKS THAT UP INTO WORDS, WHICH IT DOES BY LOOKING AT THE MICRO PAUSES YOU TAKE IN BETWEEN WORDS AS YOU TALK.

Pipeline



What is ASR?

Automatic Speech Recognition (ASR) is the process of deriving the transcription (word sequence) of an utterance, given the speech waveform. Speech understanding goes one step further, and gleans the meaning of the utterance in order to carry out the speaker's command. ASR systems facilitate a physically handicapped person to command and control a machine. Even ordinary persons would prefer a voice interface over a keyboard or mouse. The advantage is more obvious in case of small handheld devices. Dictation machine is a well-known application of ASR. Thanks to the ubiquitous telecommunication systems, speech interface is very convenient for data entry, access of information from remote databases, interactive services such as ticket reservation. ASR systems are expedient in cases where hands and eyes are busy such as driving or surgery. They are useful for teaching phonetic and programmed teaching as well.

Types of ASR

Speech Recognition Systems can be categorized into different groups depending on the constraints imposed on the nature of the input speech.

- Number of speakers: A system is said to be speaker independent if it can recognize speech of any and every speaker; such a system has learnt the characteristics of a large number of speakers. A large amount of a user's speech data is necessary for training a speaker dependent system. Such a system does not recognize other's speech well. Speaker adaptive systems, on the other hand, are

speaker independent systems to start with, but have the capability to adapt to the voice of a new speaker provided sufficient amount of his/her speech is provided for training the system. Popular dictation machine is a speaker adapted system.

- Nature of the utterance: A user is required to utter words with clear pause between words in an Isolated Word Recognition system. Connected Word Recognition system can recognize words, drawn from a small set, spoken without need for a pause between words. On the other hand, Continuous Speech Recognition systems recognize sentences spoken continuously. Spontaneous speech recognition system can handle speech disfluencies such as ah, am or false starts, grammatical errors present in a conversational speech. A Keyword Spotting System keeps looking for a pre-specified set of words and detects the presence of any one of them in the input speech.
- Vocabulary size: An ASR system that can recognize a small number of words (say, 10 digits) is called a small vocabulary system. Medium vocabulary systems can recognize a few hundreds of words. Large and Very Large ASR systems are trained with several thousands and several tens of thousands of words, respectively. Examples application domains of small, medium and very large vocabulary systems are telephone/credit card number recognition, command and control, dictation systems, respectively.
- Spectral bandwidth: The bandwidth of telephone/mobile channel is limited to 300-3400Hz and therefore attenuates frequency components outside this passband. Such a speech is called narrow-band speech. In contrast, normal speech that does not go through such a channel is call wide band speech; it contains a wider spectrum limited only by the sampling frequency. As a result, recognition accuracy of ASR systems trained with wide band speech is better. Moreover, an ASR system trained with narrow band speech performs poorly with wideband speech and vice versa.

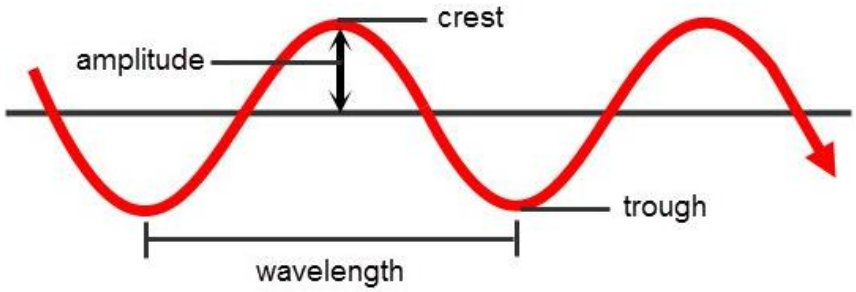
The Representation of Voice

This is pretty intuitive — any object that vibrates produces sound waves. Have you ever thought of how we are able to hear someone's voice? It is due to the audio waves. Let's quickly understand the process behind it.

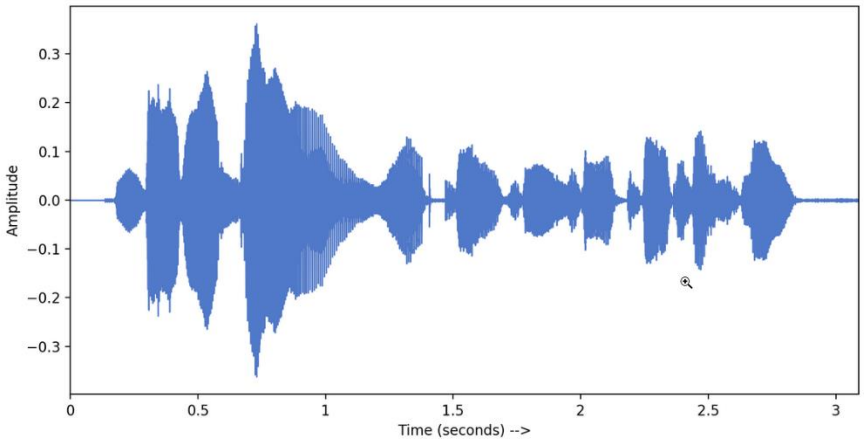
When an object vibrates, the air molecules oscillate to and from their rest position and transmits its energy to neighboring molecules. This results in the transmission of energy from one molecule to another which in turn produces a sound wave.

Parameters of an audio signal

- ***Amplitude:*** Amplitude refers to the maximum displacement of the air molecules from the rest position
- ***Crest and Trough:*** The crest is the highest point in the wave whereas trough is the lowest point.
- ***Wavelength:*** The distance between 2 successive crests or troughs is known as a wavelength
- ***Cycle:*** Every audio signal traverses in the form of cycles. One complete upward movement and downward movement of the signal form a cycle
- ***Frequency:*** Frequency refers to how fast a signal is changing over a period of time

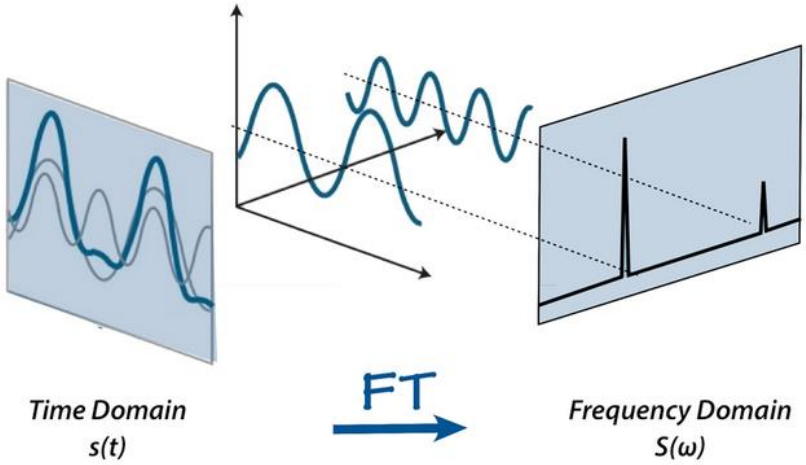


This visualization is called the time-domain representation of a given signal. This shows us the loudness (amplitude) of sound wave changing with time. Here amplitude = 0 represents silence. (From the definition of sound waves — This amplitude is actually the amplitude of air particles which are oscillating because of the pressure change in the atmosphere due to sound).



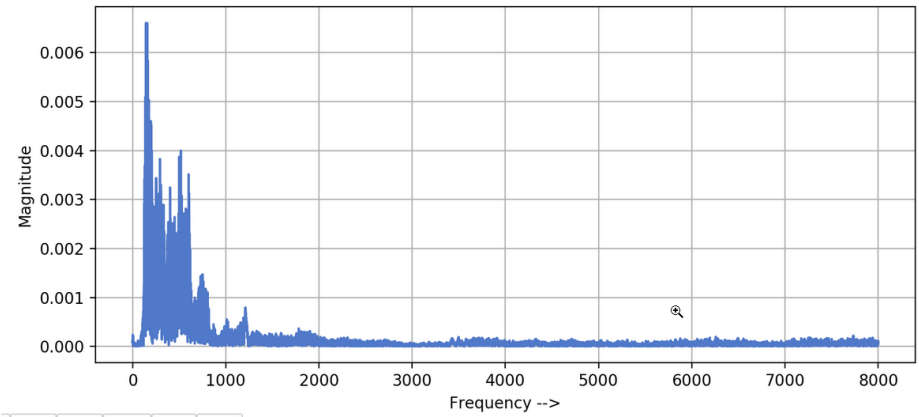
- Fourier Transform (FT)

These amplitudes are not very informative, as they only talk about the loudness of audio recording. To better understand the audio signal, it is necessary to transform it into the frequency-domain. The frequency-domain representation of a signal tells us what different frequencies are present in the signal. Fourier Transform is a mathematical concept that can convert a continuous signal from time-domain to frequency-domain. Let's learn more about Fourier Transform. An audio signal is a complex signal composed of multiple 'single-frequency sound waves' which travel together as a disturbance(pressure-change) in the medium.



When sound is recorded, we only capture the resultant amplitudes of those multiple waves. Fourier Transform is a mathematical concept that can decompose a signal into its constituent frequencies. Fourier transform does not just give the frequencies present in the signal, it also gives the magnitude of each frequency present in the signal.

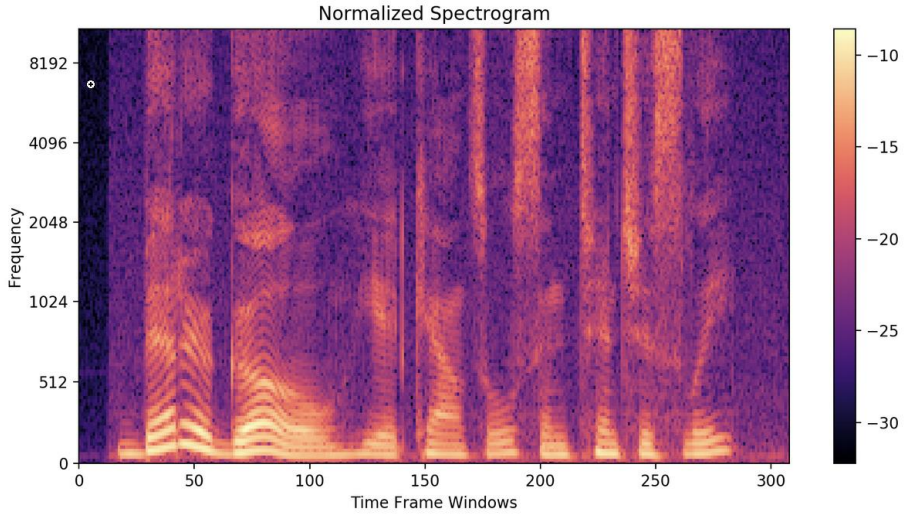
Fast Fourier Transformation (FFT) is a mathematical algorithm that calculates Discrete Fourier Transform (DFT) of a given sequence. The only difference between FT (Fourier Transform) and FFT is that FT considers a continuous signal while FFT takes a discrete signal as input. DFT converts a sequence (discrete signal) into its frequency constituents just like FT does for a continuous signal. In our case, we have a sequence of amplitudes that were sampled from a continuous audio signal. DFT or FFT algorithm can convert this time-domain discrete signal into a frequency-domain.
After applying FFT to the wave form image displayed before, Strong frequencies are ranging from 0 to 1kHz only because this audio clip was human speech. We know that in a typical human speech this range of frequencies dominates.



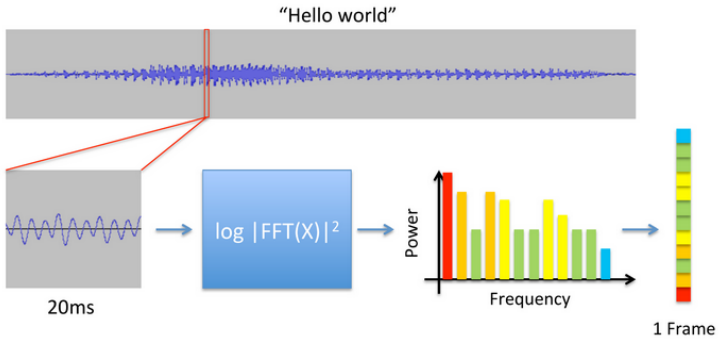
- Spectrogram:

Suppose you are working on a Speech Recognition task. You have an audio file in which someone is speaking a phrase (for example: How are you). Your recognition system should be able to predict these three words in the same order (1. 'how', 2. 'are', 3. 'you'). in the previous figure we broke our signal into its frequency values which will serve as features for our recognition system. But when we applied FFT to our signal, it gave us only frequency values and we lost the track of time information. Now our system won't be able to tell what was spoken first if we use these frequencies as features. We need to find a different way to calculate features for our system such that it has frequency values along with the time at which they were observed. Here Spectrograms come into the picture.
Visual representation of frequencies of a given signal with time is called Spectrogram. In a spectrogram representation plot — one

axis represents the time, the second axis represents frequencies and the colors represent magnitude (amplitude) of the observed frequency at a particular time. The following screenshot represents the spectrogram; Bright colors represent strong frequencies. Similar to earlier FFT plot, smaller frequencies ranging from (0–1kHz) are strong(bright).



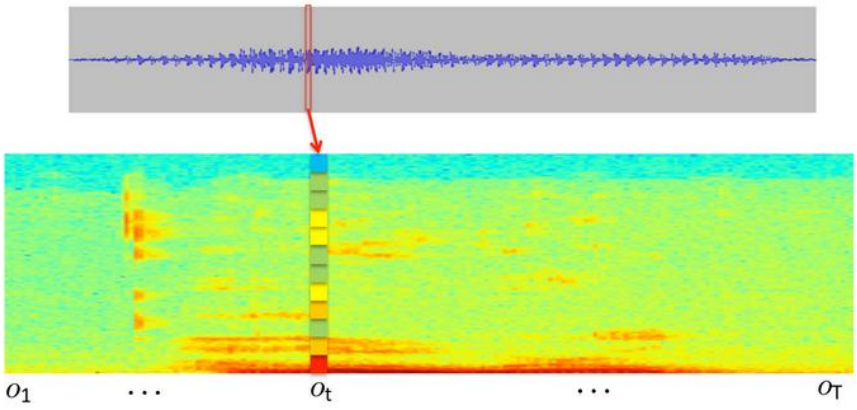
The of spectrogram Idea is to break the audio signal into smaller frames(windows) and calculate DFT (or FFT) for each window. This way we will be getting frequencies for each window and window number will represent the time. As window 1 comes first, window 2 next…and so on. It's a good practice to keep these windows overlapping otherwise we might lose a few frequencies. Window size depends upon the problem you are solving.



Short-time Fourier transform

For a typical speech recognition task, a window of 20 to 30ms long is recommended. A human can't possibly speak more than one phoneme in this time window. So keeping the window this much smaller we won't lose any phoneme while classifying. The frame

(window) overlap can vary from 25% to 75% as per your need, generally, it is kept 50% for speech recognition.



Spectrogram of a speech signal

- Mel-frequency cepstral coefficients (MFCCs):

the sounds generated by a human are filtered by the shape of the vocal tract including tongue, teeth etc. This shape determines what sound comes out. If we can determine the shape accurately, this should give us an accurate representation of the phoneme being produced. The shape of the vocal tract manifests itself in the envelope of the short time power spectrum, and the job of MFCCs is to accurately represent this envelope ASR Models. MFCC is a representation of the short-term power spectrum of a sound, which in simple terms represents the shape of the vocal tract.

Why speech recognition is difficult

Following are the some of the difficulties with ASR:

- Human comprehension of speech: Human use the knowledge about the speaker and the subject while listening more than the ears. Words are not sequences together arbitrarily but there is a grammatical structure that human use to predict words not yet uttered. In ASR, we have only speech signal. We can construct a model for grammatical structure and use some statistical model to improve prediction but there are still the problem how to model world knowledge and the knowledge of speaker. Of course, we can not model world knowledge

but the question is how much we actually require to measure up to human comprehension in the ASR.

- Spoken language is not equal to written language: Spoken language is less critical than written language and human make much more performance error while speaking. Speech is two-way communication as compared to written communication which is one-way. The most important issue is disfluencies in speech e.g. normal speech contains repetitions, tongue slips, change of subject in the middle of an utterance etc. In the last few years, it has become clear that spoken language is different from written language. In ASR, we have to identify and address these differences.
- Noise: Speech is uttered in an environment of sound such as ticking a clock, another speaker in the background, TV playing in another room etc. This unwanted information in the speech signal is known as noise. In ASR, we have to identify these noise and filter out it from the speech signal. Echo effect is another kind of noise in which speech signal is bounced on some surrounding object and it appears in the microphone a few milliseconds later.
- Body Language: A human speaker not only communicates with speech but also with body gestures such as waving hands, moving eyes, postures etc. In ASR, such information is not available. This problem is addressed in Multimodality research area where studies are conducted to incorporate body language to improve human-machine communication
- Channel Variability: Variability is the context in which acoustic wave is uttered. Different types of microphones, noise that changes over time and anything that affects the content of acoustic wave from the speaker to the discrete representation in a computer is known as channel variability.
- Speaker Variability: Human speak differently. The voice is not only different between speakers but also wide variation within one specific speaker. Some of the variations are given below.
    - Speaking Style: All speakers speak differently due to their unique personality. They have a distinctive

way to pronounce words. Speaking style varies in different situations. We do not speak in the same way in the public area, with our teachers or friends. Humans also express emotions while speaking i.e. happy, sad, fear, surprise, anger.

- Realization: if same word were pronounced again and again, the resulting speech signal would never be same. There will be some small differences in the acoustic wave produced. Speech realization changes over time.
- Speaker Sex: Male and Female have different voices. Female have shorter vocal tract than male. That is why the pitch of female voice is roughly twice than male.
- Dialects: Dialects are group related variations with in a language. Regional dialect, It involves features of vocabulary, pronunciation and grammar according to geographical area the speaker belongs.

Now, let us see how such a wide variety of variabilities are handled by an ASR system.

ASR Models

According to the speech structure, two main models are used in speech recognition:

- An ***acoustic model*** is used in automatic speech recognition to represent the relationship between an audio signal and the phonemes or other linguistic units that make up speech. The model is learned from a set of audio recordings and their corresponding transcripts. It is created by taking audio recordings of speech, and their text transcriptions, and using software to create statistical representations of the sounds that make up each word. There is a lot of different types of acoustic model throughout the years.
- A ***language model*** is used to restrict word search. It defines which word could follow previously recognized words and helps to significantly restrict the matching process by stripping words that are not probable.

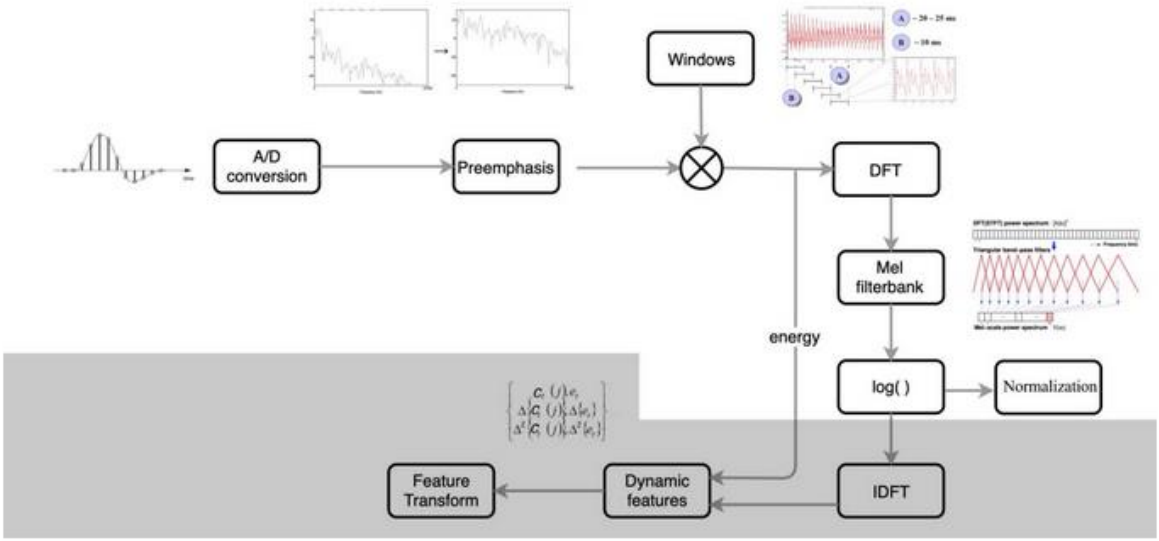
Our Used Solution

We Used Deep Speech, a Deep Neural Network speech recognition system, the advances in the field of Speech Recognition using Deep Learning made the accuracy of the models prediction a lot better than before and more robust to different voice data which made it an easy choice to chase doing our speech recognition system using deep learning. We choose Deep Speech as it's an open source project that is active and maintainable and adapt the state of the art technologies in speech recognition to enhance the accuracy in every release, and also they provide an easy to use modular API, and an open source model that was trained in almost 100 thousand hours of voice for a month in a lot of GPU, as training speech recognition system require a lot of resource which we don't have, so it will be impossible to us to train our model form scratch and achieve a usable accuracy.

We will introduce in this section the deep speech pipeline, model architecture, and API code for speech recognition. (figure pipeline)

1. Feature Extraction

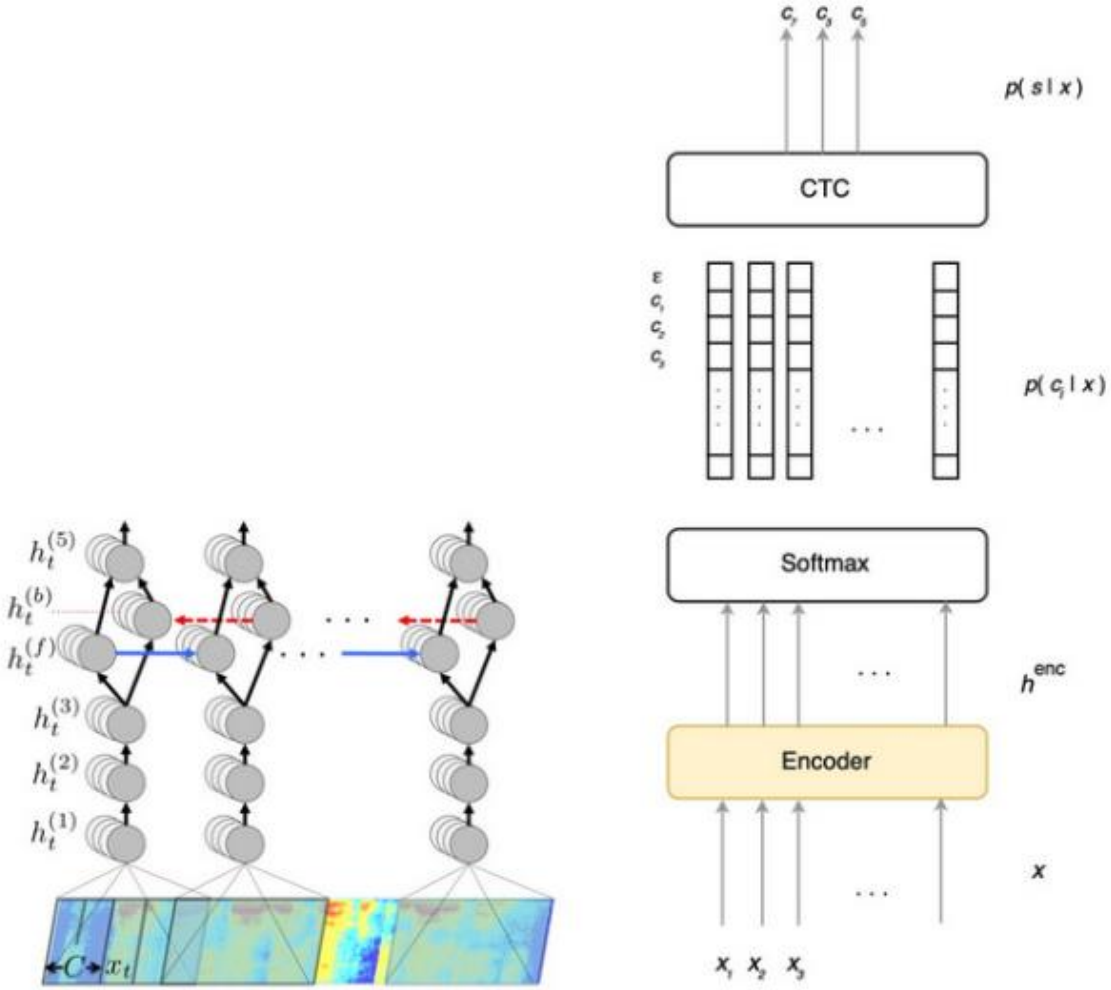
In ML speech recognition, we extract Mel-frequency cepstral coefficients (MFCC) or Perceptual Linear Prediction (PLP) features from the audio frames. It includes steps like an inverse discrete Fourier transform (IDFT) to make features less correlated with each other. Also, we only take the lower 12 coefficients. In general, ML is stingier in feature selection. DL is good at untangling data and finding correlations. Therefore, it requires no or less pre-processing. It can handle a much larger number of input features and there is no particular need to un-correlate them first. We just need to design a more complex model and lets it learned from the data. In ASR with DL, we skip the last few steps including the IDFT. We simply take the Mel filter bank, apply the log and normalize it with the speaker or corpus' mean and variance.


Windows
A/D conversion
Preemphasis
DFT
Mel filterbank
energy
log( )
Normalization
Feature Transform
Dynamic features
IDFT

2. Deep Speech

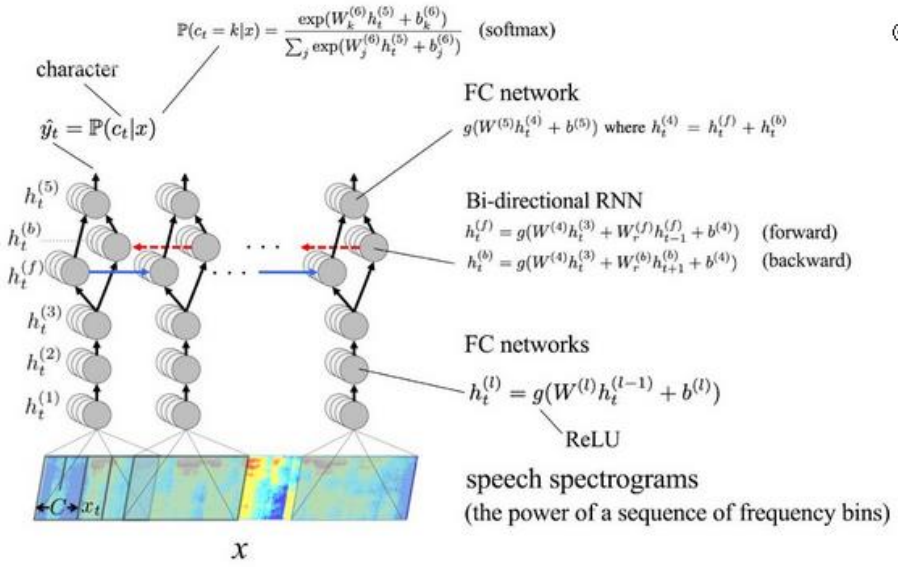
```
def create_model(batch_x, seq_length, dropout, reuse=False, batch_size=None, previous_state=None, overlap=True, rnn_impl=rnn_impl_lstmblockfusedcell):
    layers = {}

    # Input shape: [batch_size, n_steps, n_input + 2*n_input*n_context]
    if not batch_size:
        batch_size = tf.shape(input=batch_x)[0]

    # Create overlapping feature windows if needed
    if overlap:
        batch_x = create_overlapping_windows(batch_x)

    # Reshaping `batch_x` to a tensor with shape `[n_steps*batch_size, n_input + 2*n_input*n_context]`.
    # This is done to prepare the batch for input into the first layer which expects a tensor of rank `2`.

    # Permute n_steps and batch_size
    batch_x = tf.transpose(a=batch_x, perm=[1, 0, 2, 3])
    # Reshape to prepare input for first layer
    batch_x = tf.reshape(batch_x, [-1, Config.n_input + 2*Config.n_input*Config.n_context]) # (n_steps*batch_size, n_input + 2*n_input*n_context)
    layers['input_reshaped'] = batch_x

    # The next three blocks will pass `batch_x` through three hidden layers with
    # clipped RELU activation and dropout.
    layers['layer_1'] = layer_1 = dense('layer_1', batch_x, Config.n_hidden_1, dropout_rate=dropout[0])
    layers['layer_2'] = layer_2 = dense('layer_2', layer_1, Config.n_hidden_2, dropout_rate=dropout[1])
    layers['layer_3'] = layer_3 = dense('layer_3', layer_2, Config.n_hidden_3, dropout_rate=dropout[2])

    # `layer_3` is now reshaped into `[n_steps, batch_size, 2*n_cell_dim]`,
    # as the LSTM RNN expects its input to be of shape `[max_time, batch_size, input_size]`.
    layer_3 = tf.reshape(layer_3, [-1, batch_size, Config.n_hidden_3])

    # Run through parametrized RNN implementation, as we use different RNNs
    # for training and inference
    output, output_state = rnn_impl(layer_3, seq_length, previous_state, reuse)

    # Reshape output from a tensor of shape [n_steps, batch_size, n_cell_dim]
    # to a tensor of shape [n_steps*batch_size, n_cell_dim]
    output = tf.reshape(output, [-1, Config.n_cell_dim])
    layers['rnn_output'] = output
    layers['rnn_output_state'] = output_state

    # Now we feed `output` to the fifth hidden layer with clipped RELU activation
    layers['layer_5'] = layer_5 = dense('layer_5', output, Config.n_hidden_5, dropout_rate=dropout[5])

    # Now we apply a final linear layer creating `n_classes` dimensional vectors, the logits.
    layers['layer_6'] = layer_6 = dense('layer_6', layer_5, Config.n_hidden_6, relu=False)

    # Finally we reshape layer_6 from a tensor of shape [n_steps*batch_size, n_hidden_6]
    # to the slightly more useful shape [n_steps, batch_size, n_hidden_6].
    # Note, that this differs from the input in that it is time-major.
    layer_6 = tf.reshape(layer_6, [-1, batch_size, Config.n_hidden_6], name='raw_logits')
    layers['raw_logits'] = layer_6

    # Output shape: [n_steps, batch_size, n_hidden_6]
    return layer_6, layers
```

Next, let's look at building a deep network specifically from what has been learned. Below is a deep network called Deep Speech.



At each time step, the design above applies a sliding window to extract the power at different frequencies.

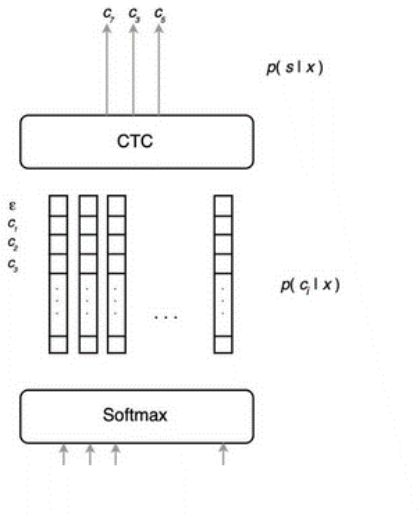
At each time step, Deep Speech applies three FC layers to extract features. Then, it is feed into a bi-directional RNN to explore the context of the speech. Deep Speech adds the results from the forward and backward direction together for each time step. Then, it applies another FC layer to transform the result. Finally, the probability for each character at each time step is computed with a Softmax.



Here are the characters that are included for the probability distribution:

Output: character probabilities (a-z, <apostrophe>, <space>, <blank>)

Deep speech uses the objective function below to find the optimal sequence of characters. This objective function is based on the distribution output of the deep network, an N-gram language model and the word count of the decoded sentence. Both α and β are hyperparameters to be tuned. The language model allows us to produce grammatically sound sentences. But the deep network may be biased to create un-necessary more or fewer words than it should be. So we add a word count objective to tune the selection process.
Then, it applies CTC and beam-search to find the most likely word sequence for the utterances.



Deep speech uses the objective function below to find the optimal sequence of characters. This objective function is based on the distribution output of the deep network, an N-gram language model and the word count of the decoded sentence. Both α and β are hyperparameters to be tunned. The language model allows us to produce grammatically sound sentences. But the deep network may be biased to create un-necessary more or fewer words than it should be. So we add a word count objective to tune the selection process.

hyperparameters

$$Q(c) = \log(\mathbb{P}(c|x)) + \alpha \log(\mathbb{P}_{\text{lm}}(c)) + \beta \, \text{word\_count}(c)$$

a sequence of characters

language model

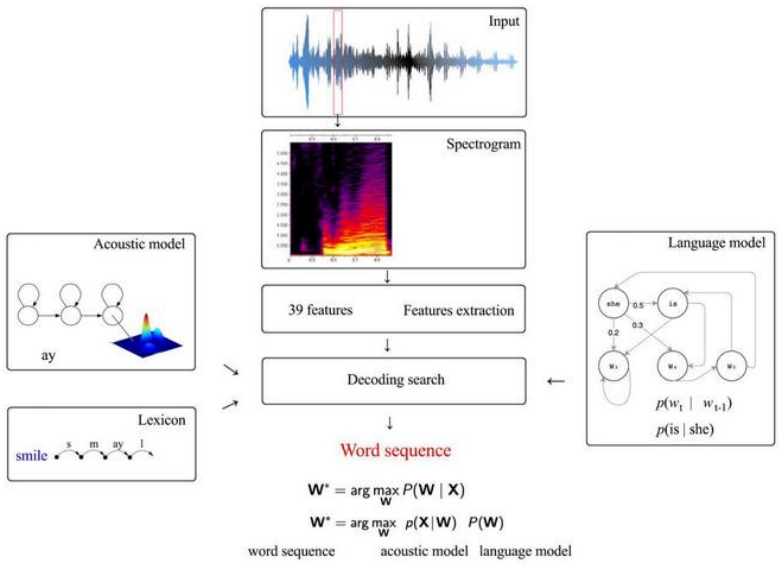
probability of c according to a N-gram model

Modified from source

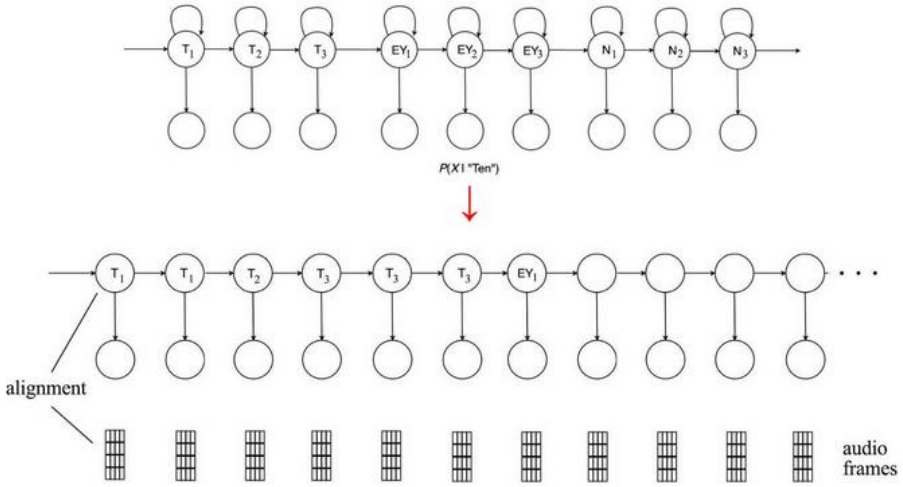
Connectionist temporal classification (CTC)

```
def calculate_mean_edit_distance_and_loss(iterator, dropout, reuse):
    r'''
    This routine beam search decodes a mini-batch and calculates the loss and mean edit distance.
    Next to total and average loss it returns the mean edit distance,
    the decoded result and the batch's original Y.
    '''
    # Obtain the next batch of data
    batch_filenames, (batch_x, batch_seq_len), batch_y = iterator.get_next()

    if FLAGS.train_cudnn:
        rnn_impl = rnn_impl_cudnn_rnn
    else:
        rnn_impl = rnn_impl_lstmblockfusedcell

    # Calculate the logits of the batch
    logits, _ = create_model(batch_x, batch_seq_len, dropout, reuse=reuse, rnn_impl=rnn_impl)

    # Compute the CTC loss using TensorFlow's `ctc_loss`
    total_loss = tfv1.nn.ctc_loss(labels=batch_y, inputs=logits, sequence_length=batch_seq_len)

    # Check if any files lead to non finite loss
    non_finite_files = tf.gather(batch_filenames, tfv1.where(~tf.math.is_finite(total_loss)))

    # Calculate the average loss across the batch
    avg_loss = tf.reduce_mean(input_tensor=total_loss)

    # Finally we return the average loss
    return avg_loss, non_finite_files
```

ASR decoding is mainly composed of two major steps: the mapping and the searching.

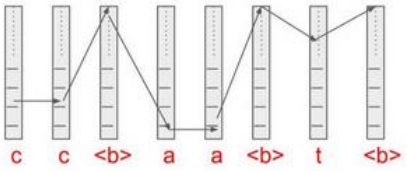


In the mapping, we map the acoustic information of an audio frame to a triphone state. This is the alignment process. This is a many-to-one mapping. It maps multiple audio frames to the same triphone state.



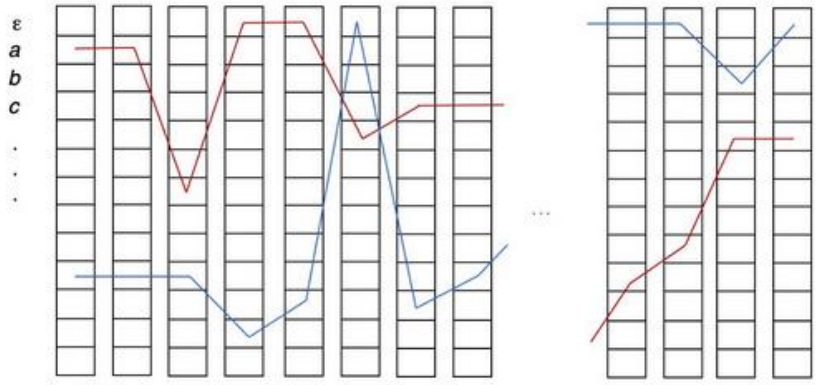
The alignment maps acoustic information to a phone. Then, we search for the phone sequence for the optimal word sequence. However, we need to account for multiple paths that produce the same results. This makes things complicated and requires us to use complex algorithms, like forward-backward (FB) or Viterbi algorithm.

DL learning is good at mapping. It opens the door for an alignment-free one-to-one mapping which maps an audio frame to a relatively high-level component. Then, we can search for it. This gives us a head start and bypasses the complicated alignment process. That is the core concept of CTC. While in early research, the deep network works with phonemes, now the deep network works with character sequences. This frees us further from the pronunciation lexicon and the phonetic decision tree.

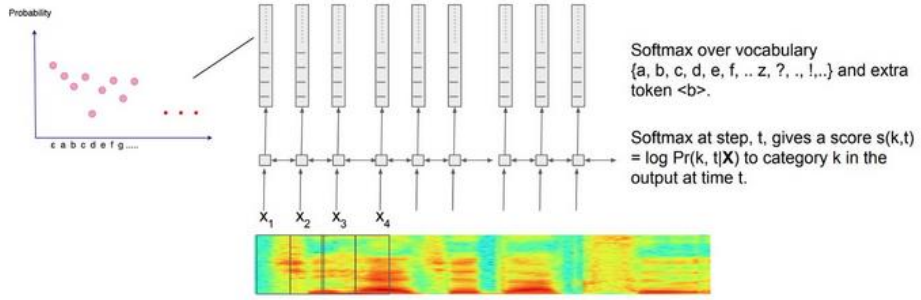


cc<b>aa<b>t<b> maps to cat
cc<b><b>a<b>t<b> maps to cat
ccccc<b>aaaaa<b>ttttt<b> maps to cat
ccccc<b>aaaaa<b>ttttt<b> maps to cat

The deep network generates a probability distribution for all characters. We don't greedily pick the most likely character in each time step. There are multiple paths that represent the same word (details later). We need to sum over them to find the optimal paths.

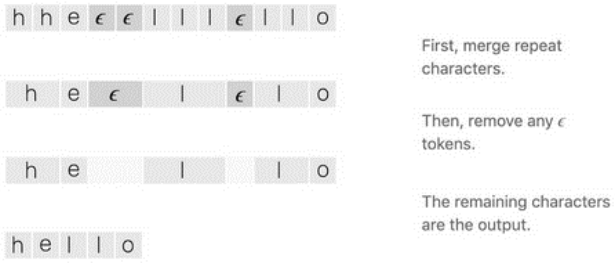


Since people pause in speech, within or between words, we introduce empty tokens <b> (or written as **ε**) to model such behavior. Other characters like ? and ! are also introduced. The following diagram indicates how we use bi-direction RNN to score characters for each observed audio frame. Then we apply softmax to create the probability distribution for characters in each time step.

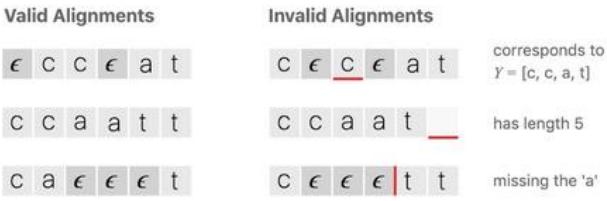


CTC Compression Rule:

Previously, we said multiple paths may produce the same word. Given the path "aaapεpplleε", what is the word that it represents? For that, we apply the CTC compression rule. Any repeated characters will be merged into one. Then, we remove **ε** for the final word.



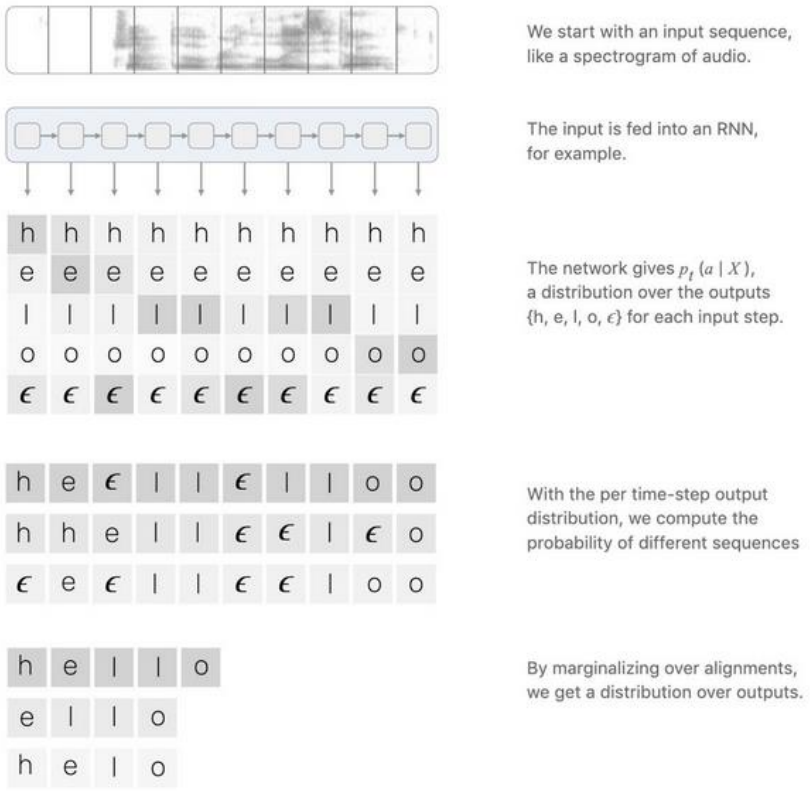
As noted in this example, to output the word apple, the deep network needs to predict a **ε** between the two pp in apple. Therefore, **ε** also serves the purpose of separating repeated characters in a word. Here are the valid and invalid predictions for the word "cat".



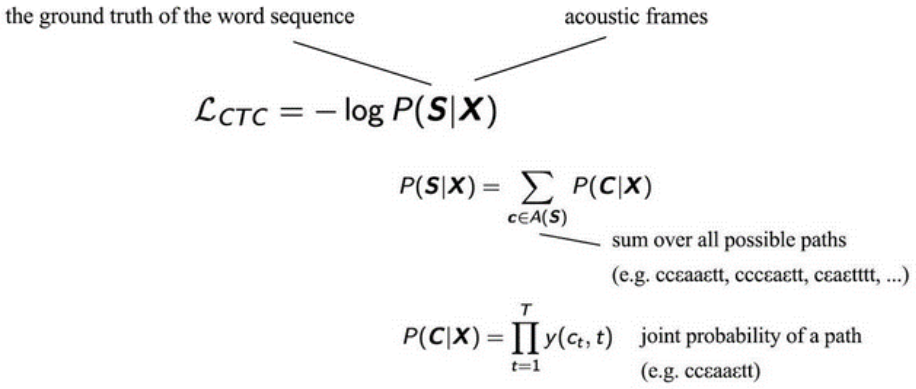
The alignment-free concept in CTC actually pushes the complexity from mapping to searching. But by outputting character distributions and get rid of lexicon and decision trees, the search may not be that bad.

CTC Loss:

To find the most likely word sequence, we search for different path combinations. We need to sum over all paths that generate the same word sequence. For example, to find the probability for the word "hello", we sum over all the corresponding paths like "heεllεlloo", "hhellεεlεo", "εeεllεεloo" etc.



Here is the corresponding CTC loss function:



3. Language Model:

When trained from large quantities of labeled speech data, the RNN model can learn to produce readable character-level transcriptions. Indeed for many of the transcriptions, the most likely character sequence predicted by the RNN is exactly correct without external language constraints. The errors made by the

RNN in this case tend to be phonetically plausible renderings of English words Table shows some examples.

RNN output	Decoded Transcription
what is the weather like in bostin right now	what is the weather like in boston right now
prime miniter nerenr modi	prime minister narendra modi
arther n tickets for the game	are there any tickets for the game

Many of the errors occur on words that rarely or never appear in our training set. In practice, this is hard to avoid: training from enough speech data to hear all of the words or language constructions we might need to know is impractical. Therefore, they integrate their system with an N-gram language model since these models are easily trained from huge unlabeled text corpora. For comparison, while speech datasets typically include up to 3 million utterances, the N-gram language model used for the experiments is trained from a corpus of 220 million phrases, supporting a vocabulary of 495,000 words.
Given the output P(c|x) of RNN they perform a search to find the sequence of characters c1, c2, ... that is most probable according to both the RNN output and the language model (where the language model interprets the string of characters as words). Specifically, the model aim to find a sequence c that maximizes the combined objective:

$$Q(c) = \log(\mathbb{P}(c|x)) + \alpha \log(\mathbb{P}_{\text{lm}}(c)) + \beta \, \text{word\_count}(c)$$

where α and β are tunable parameters (set by cross-validation) that control the trade-off between the RNN, the language model constraint and the length of the sentence. The term Plm denotes the probability of the sequence c according to the N-gram model. The model maximize this objective using a highly optimized beam search algorithm, with a typical beam size in the range 1000-8000.

4. Deep Speech API:

- This code uses the Deep Speech API to initialize the deep speech model and load the model weights.
- We connect to the computer mice and starts listening to the driver, we don't bass to the voice signals to the model unless we detect a voice from the driver.
- We then bass the model the voice of the driver and get the transcript, that take to analysis after to figure out the driver specific command and take the right action.

Audio Class

- In this class we initialize the connection with the device microphone to listen to the user voice using pyaudio library.
- We also initialize the connection variable like the scampling rate, the device number and so on.

```
class Audio(object):
    """Streams raw audio from microphone. Data is received in a separate thread, and stored in a buffer, to be read from."""

    FORMAT = pyaudio.paInt16
    # Network/VAD rate-space
    RATE_PROCESS = 16000
    CHANNELS = 1
    BLOCKS_PER_SECOND = 50

    def __init__(self, callback=None, device=None, input_rate=RATE_PROCESS, file=None):
        def proxy_callback(in_data, frame_count, time_info, status):
            #pylint: disable=unused-argument
            if self.chunk is not None:
                in_data = self.wf.readframes(self.chunk)
            callback(in_data)
            return (None, pyaudio.paContinue)
        if callback is None: callback = lambda in_data: self.buffer_queue.put(in_data)
        self.buffer_queue = queue.Queue()
        self.device = device
        self.input_rate = input_rate
        self.sample_rate = self.RATE_PROCESS
        self.block_size = int(self.RATE_PROCESS / float(self.BLOCKS_PER_SECOND))
        self.block_size_input = int(self.input_rate / float(self.BLOCKS_PER_SECOND))
        self.pa = pyaudio.PyAudio()

        kwargs = {
            'format': self.FORMAT,
            'channels': self.CHANNELS,
            'rate': self.input_rate,
            'input': True,
            'frames_per_buffer': self.block_size_input,
            'stream_callback': proxy_callback,
        }

        self.chunk = None
        # if not default device
        if self.device:
            kwargs['input_device_index'] = self.device
        elif file is not None:
            self.chunk = 320
            self.wf = wave.open(file, 'rb')

        self.stream = self.pa.open(**kwargs)
        self.stream.start_stream()

    # change signal to 16mHZ for the models
    def resample(self, data, input_rate):
        """
        Microphone may not support our native processing sampling rate, so
        resample from input_rate to RATE_PROCESS here for webrtcvad and
        deepspeech

        Args:
            data (binary): Input audio stream
            input_rate (int): Input audio rate to resample from
        """
        data16 = np.fromstring(string=data, dtype=np.int16)
        resample_size = int(len(data16) / self.input_rate * self.RATE_PROCESS)
        resample = signal.resample(data16, resample_size)
        resample16 = np.array(resample, dtype=np.int16)
        return resample16.tostring()

    def read_resampled(self):
        """Return a block of audio data resampled to 16000hz, blocking if necessary."""
        return self.resample(data=self.buffer_queue.get(),
                             input_rate=self.input_rate)

```

```
    def read(self):
        """Return a block of audio data, blocking if necessary."""
        return self.buffer_queue.get()

    # stop the Recording
    def destroy(self):
        self.stream.stop_stream()
        self.stream.close()
        self.pa.terminate()

    frame_duration_ms = property(lambda self: 1000 * self.block_size // self.sample_rate)

    def write_wav(self, filename, data):
        logging.info("write wav %s", filename)
        wf = wave.open(filename, 'wb')
        wf.setnchannels(self.CHANNELS)
        # wf.setsampwidth(self.pa.get_sample_size(FORMAT))
        assert self.FORMAT == pyaudio.paInt16
        wf.setsampwidth(2)
        wf.setframerate(self.sample_rate)
        wf.writeframes(data)
        wf.close()
```

VAD Audio

- This class is responsible to collect the driver voice and process it.
- This class take a small widow (around 20ms voice window), and try to detect if there is someone taking in this frame by finding the frequency in the frame (if there is only small frequency that means no speaker).
- If there is a number of consecutive frames that contains a voice then we know that the driver is taking and start to collect these frames to give it to the model.
- Then we collect frames and give it to the model until the class detect no one specking then we give a flag to the model to start processing and return the prediction.

```
class VADAudio(Audio):
    """Filter & segment audio with voice activity detection."""

    def __init__(self, aggressiveness=3, device=None, input_rate=None, file=None):
        super().__init__(device=device, input_rate=input_rate, file=file)
        self.vad = webrtcvad.Vad(aggressiveness)

    def frame_generator(self):
        """Generator that yields all audio frames from microphone."""
        if self.input_rate == self.RATE_PROCESS:
            while True:
                yield self.read()
        else:
            while True:
                yield self.read_resampled()

    def vad_collector(self, padding_ms=300, ratio=0.75, frames=None):
        """Generator that yields series of consecutive audio frames comprising each utterence, sep
arated by yielding a single None.
            Determines voice activity by ratio of frames in padding_ms. Uses a buffer to include p
adding_ms prior to being triggered.
            Example: (frame, ..., frame, None, frame, ..., frame, None, ...)
                      |---utterence---|        |---utterence---|
        """
        if frames is None: frames = self.frame_generator()
        num_padding_frames = padding_ms // self.frame_duration_ms
        ring_buffer = collections.deque(maxlen=num_padding_frames) # used to calculate the preiod
of silence
```

```
        triggered = False

        for frame in frames:
            if len(frame) < 640:
                return

            is_speech = self.vad.is_speech(frame, self.sample_rate)
            #print(is_speech)

            if not triggered:
                ring_buffer.append((frame, is_speech))
                num_voiced = len([f for f, speech in ring_buffer if speech])
                if num_voiced > ratio * ring_buffer.maxlen:  # there is enough active frames to in
dicate the user is active
                    triggered = True
                    for f, s in ring_buffer:
                        yield f
                    ring_buffer.clear()

            else:
                yield frame
                ring_buffer.append((frame, is_speech))
                num_unvoiced = len([f for f, speech in ring_buffer if not speech])
                if num_unvoiced > ratio * ring_buffer.maxlen:  # there is enough unactive frames t
o indicate the user is not active any more
                    triggered = False
                    yield None
                    ring_buffer.clear()
```

Main Calss

- in the main class we configure the deepspeech model by calling the deepspeech API and pass to it the configuration varilbes like (n_features: the number of features after decoding, n_contect, represent the number of windows overlapping, and so on) and also the bath to the deep speech model weights
- We also enable the language model by using (model.enableDecoderWithLM) an passing the model parameters (the language model path, the characters, the words, and so on)
- We then call the VADAudio Class that initialize the connection with the Audio device and starts to pass the voice frames to the model (model.feedAudioContent) when we detect a speaker
- We then pass finish the model stream (model.finishStream) after we receive no further frames, when the speaker stop talking
- Then the model return transcript text that we will take to the next model to process it.

```
def main(ARGS):
    # Load DeepSpeech model
    if os.path.isdir(ARGS.model):
        model_dir = ARGS.model
        ARGS.model = os.path.join(model_dir, 'output_graph.pb')
        ARGS.alphabet = os.path.join(model_dir, ARGS.alphabet if ARGS.alphabet else 'alphabet.txt'
)
        ARGS.lm = os.path.join(model_dir, ARGS.lm)
        ARGS.trie = os.path.join(model_dir, ARGS.trie)
```

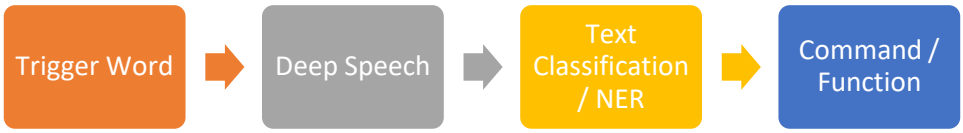
```
9.
10.      print('Initializing model...')
11.      logging.info("ARGS.model: %s", ARGS.model)
12.      logging.info("ARGS.alphabet: %s", ARGS.alphabet)
13.      model = deepspeech.Model(ARGS.model, ARGS.n_features, ARGS.n_context, ARGS.alphabet, ARGS.beam
     _width)
14.      if ARGS.lm and ARGS.trie:
15.          logging.info("ARGS.lm: %s", ARGS.lm)
16.          logging.info("ARGS.trie: %s", ARGS.trie)
17.          model.enableDecoderWithLM(ARGS.alphabet, ARGS.lm, ARGS.trie, ARGS.lm_alpha, ARGS.lm_beta)
18.
19.      # Start audio with VAD
20.      vad_audio = VADAudio(aggressiveness=ARGS.vad_aggressiveness,
21.                           device=ARGS.device,
22.                           input_rate=ARGS.rate,
23.                           file=ARGS.file)
24.      # this create a generator from vad_collector function.
25.      # if new block is saved it will be saved in the buffer queue
26.      # when a new frame is required to process the function vad_collector.
27.      # provids the next uttrance by using yeild, so it will provide frame and
28.      # when next one is asked it will return to the function from the last yeild postion.
29.      print("Listening (ctrl-C to exit)...")
30.      frames = vad_audio.vad_collector()
31.
32.      # Stream from microphone to DeepSpeech using VAD
33.      spinner = None
34.      if not ARGS.nospinner:
35.          spinner = Halo(spinner='line')
36.      stream_context = model.setupStream()
37.      wav_data = bytearray()
38.      for frame in frames:
39.
40.          if frame is not None:
41.              if spinner: spinner.start()
42.              logging.debug("streaming frame")
43.              model.feedAudioContent(stream_context, np.frombuffer(frame, np.int16))
44.              if ARGS.savewav: wav_data.extend(frame)
45.          else:
46.              if spinner: spinner.stop()
47.              logging.debug("end utterence")
48.              if ARGS.savewav:
49.                  vad_audio.write_wav(os.path.join(ARGS.savewav, datetime.now().strftime("savewav_%Y
     -%m-%d_%H-%M-%S_%f.wav")), wav_data)
50.                  wav_data = bytearray()
51.              text = model.finishStream(stream_context)
52.              print("Recognized: %s" % text)
53.              stream_context = model.setupStream()
```

CHAPTER SEVEN

COMMANDS CLASSIFICATION

FOR TEXT CLASSIFICATION, YOU OFTEN BEGIN WITH SOME TEXT YOU WANT TO CLASSIFY. FOR INSTANCE, YOU HAVE QUOTES AND WANTS TO FIND THE QUOTES ABOUT LOVE. OR YOU HAVE EMAILS AND YOU WANT TO SEPARATE SPAM FROM LEGITIMATE EMAILS...

Pipeline



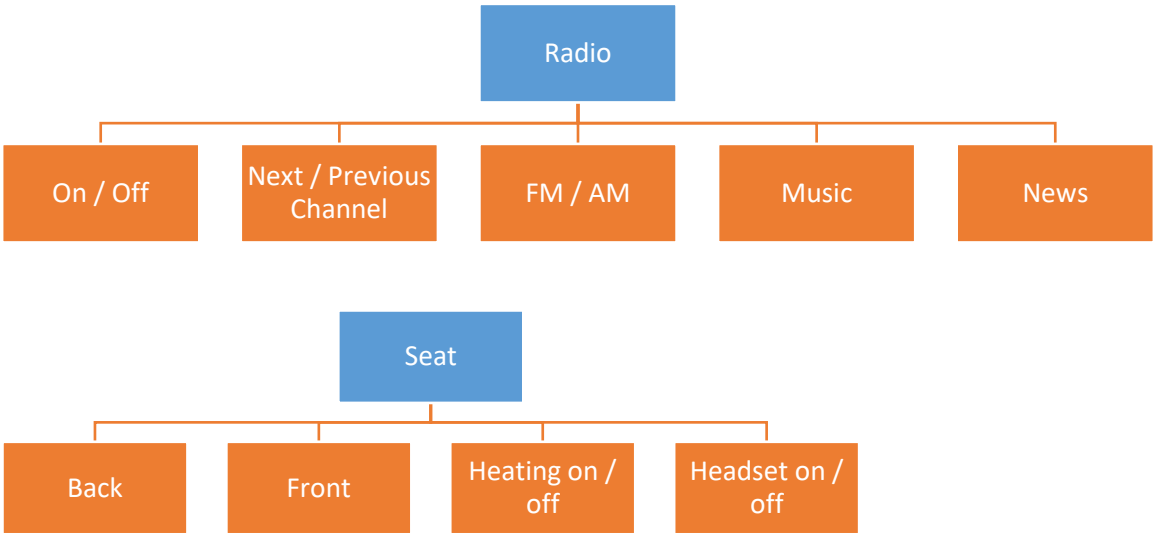
How the model works?

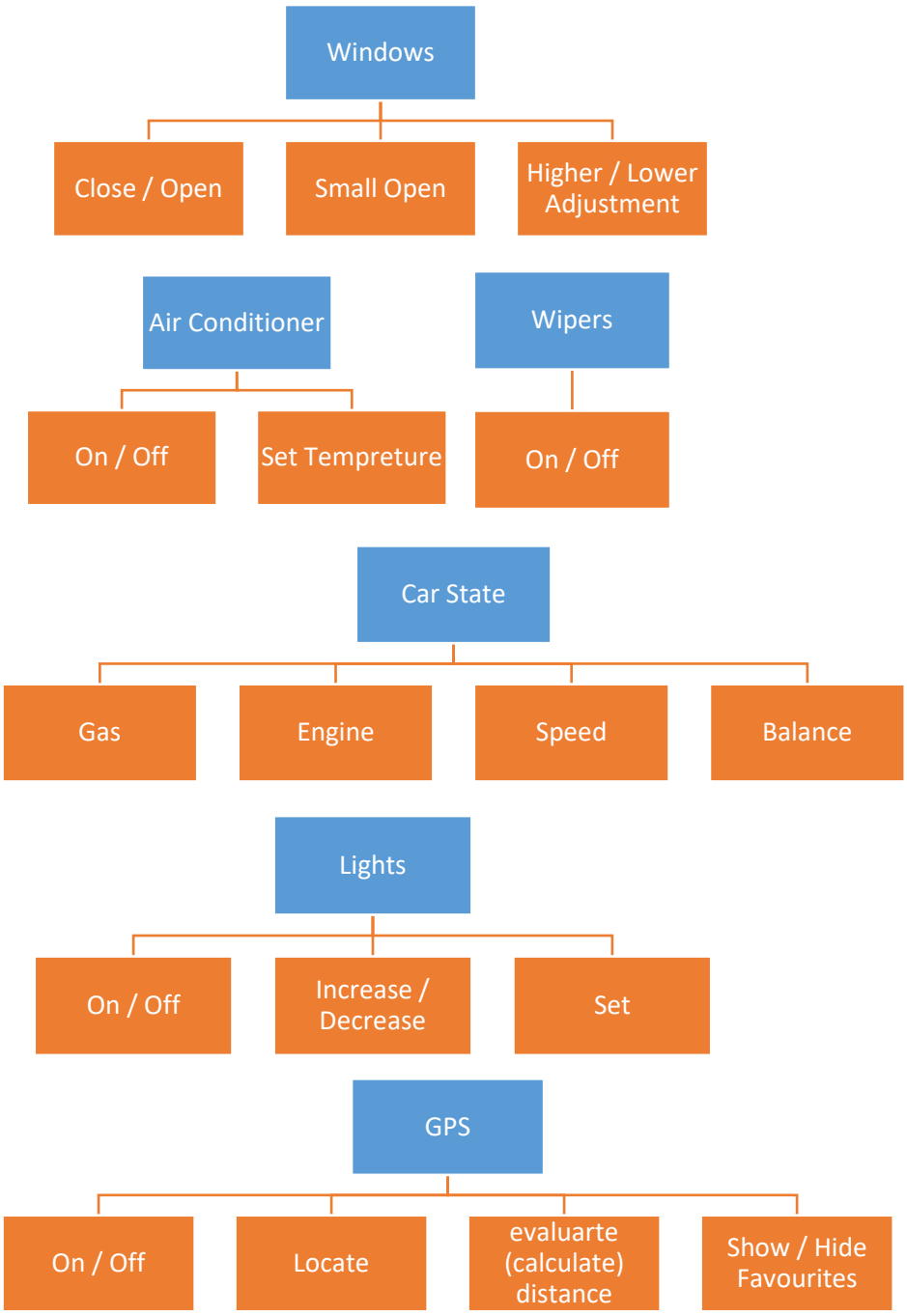
This model will take the text produced by The Deep Speech (Model 3). After the ASR convert the voice command to text, we will then classify the text to specify the desired command from a predefined set of commands and we will also extract the information from text that will be relevant to the command.

Dataset

The predefined commands are the most common commands used in the car, for example (turn on the radio, Set the air conditioning to 55 degrees, etc). and we couldn't find a well-suited dataset for our classifier model so we had to create our own simple dataset.

The dataset consists of (Command – Subcommand – Sentence) as the command represent the object and the subcommand is the action desired from that object, i.e (Set the air conditioning to 55 degrees, Command: Air Conditioner , Subcommand: Set)




Windows
Close / Open
Small Open
Higher / Lower Adjustment
Air Conditioner
Wipers
On / Off
Set Tempreture
On / Off
Car State
Gas
Engine
Speed
Balance
Lights
On / Off
Increase / Decrease
Set
GPS
On / Off
Locate
evaluarte (calculate) distance
Show / Hide Favourites

Sentence	Command	Subcommand / Entity
turn on the radio	Radio	radio on
radio on	Radio	radio on
turn the radio on	Radio	radio on
play the radio	Radio	radio on
play radio	Radio	radio on
listen to the radio	Radio	radio on
resume radio	Radio	radio on
resume	Radio	radio on
turn off the radio	Radio	radio off
radio off	Radio	radio off
turn the radio off	Radio	radio off
pause	Radio	radio off
pause the radio	Radio	radio off
stop	Radio	radio off
stop the radio	Radio	radio off
radio next channel	Radio	next cahnnel
change the radio channel	Radio	next cahnnel
change the radio	Radio	next cahnnel
next radio channel	Radio	next cahnnel
next channel	Radio	next cahnnel
change channel	Radio	next cahnnel
channel up	Radio	next cahnnel
radio channel up	Radio	next cahnnel
radio previous channel	Radio	previous channel
radio last channel	Radio	previous channel
previous radio channel	Radio	previous channel
previous channel	Radio	previous channel

Data Augmentation

Since the created data is pretty small (only 419 sentences) and it may lead to overfitting, we used augmentation to add more data.

- Easy Data Augmentation (EDA)

We used EDA (Easy Data Augmentation) as it deals better with smaller Text Classification datasets to add more sentences to each command, hence avoiding overfitting.

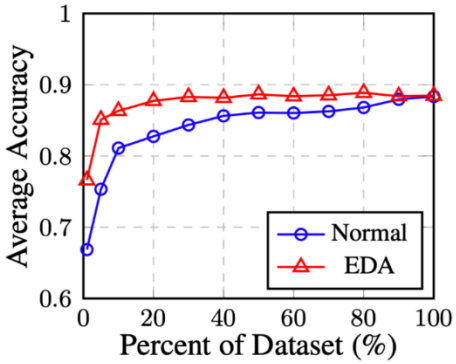
How EDA Works?

Given a sentence in the training set, EDA applies the following:

- Synonym Replacement (SR): Randomly choose n words from the sentence that are not stop words. Replace each of these words with one of its synonyms chosen at random.
- Random Insertion (RI): Find a random synonym of a random word in the sentence that is not a stop word. Insert that synonym into a random position in the sentence. Do this n times.
- Random Swap (RS): Randomly choose two words in the sentence and swap their positions. Do this n times.
- Random Deletion (RD): For each word in the sentence, randomly remove it with probability p.

```
pip install - U nltk
import nltk;
nltk.download('wordnet')

# Cloning the Repo
!git clone https: //github.com/jasonwei20/eda_nlp.git

#The input files in the format label\tsentence(note the\t (tab))
!python augment.py--input = 'input.txt'--output = output_augmented_16sens_0.05 alpaha.txt--num_aug = 16--alpha = 0.05
```

You can specify the number of generated augmented sentences per original sentence using --num_aug Furthermore, you can specify the alpha parameter, which approximately means the percent of words in the sentence that will be changed.



EDA Architecture – Code

The EDA uses simple text modifications and tasks, not as many other techniques that use Deep Learning which are waste of time and memory in our case, here are the modifications in detail.

Synonym Replacement

Using WordNet, the function takes a sentence and replaces the words (that are not stopwords) to other words with the same meaning

```
def synonym_replacement(words, n):
    new_words = words.copy()
    random_word_list = list(set([word for word in words if word not in stop_words]))
    random.shuffle(random_word_list)
    num_replaced = 0
    for random_word in random_word_list:
        #get_synonyms is called to get another word that has a close meaning to the word using wordnet
        synonyms = get_synonyms(random_word)
        if len(synonyms) >= 1:
            synonym = random.choice(list(synonyms))
            new_words = [synonym if word == random_word else word for word in new_words]
            num_replaced += 1
        if num_replaced >= n: #only replace up to n words
            break

    sentence = ' '.join(new_words)
    new_words = sentence.split(' ')

    return new_words
```

```
def get_synonyms(word):
    synonyms = set()
    for syn in wordnet.synsets(word):
        for l in syn.lemmas():
            synonym = l.name().replace("_", " ").replace("-", " ").lower()
            synonym = "".join([char for char in synonym if char in ' qwertyuiopasdfghjklzxcvbnm'])
            synonyms.add(synonym)
    if word in synonyms:
        synonyms.remove(word)
    return list(synonyms)
```

Random Deletion

This function takes the sentence and deletes randomly a word from it using a probability, it can be changed using the hyperparameter alpha or manually

```python
def random_deletion(words, p): #P=0.1 by default and can be changed

    #obviously, if there's only one word, don't delete it
    if len(words) == 1:
        return words

    #randomly delete words with probability p
    #Higher P means less deletion
    new_words = []
    for word in words:
        r = random.uniform(0, 1)
        if r > p:
            new_words.append(word)

    #if you end up deleting all words, just return a random word
    if len(new_words) == 0:
        rand_int = random.randint(0, len(words)-1)
        return [words[rand_int]]

    return new_words
```

Random Swap

Swaps words in a sentence randomly (even if the meaning if the sentence is lost (we don't use it a lot specially after we filtered the augmented data and removed the sentences that don't make sense))

```python
def random_swap(words, n):
    new_words = words.copy()
    for _ in range(n):
        #swap_word gets the sentence and randomly swaps words posotions
        new_words = swap_word(new_words)
    return new_words

def swap_word(new_words):
    random_idx_1 = random.randint(0, len(new_words)-1) #iterator to a random pos
    random_idx_2 = random_idx_1
    counter = 0
    while random_idx_2 == random_idx_1:
        random_idx_2 = random.randint(0, len(new_words)-1)
        counter += 1
        if counter > 3:
            return new_words
    new_words[random_idx_1], new_words[random_idx_2] = new_words[random_idx_2], new_words[random_idx_1]
    return new_words
```

Random Insertion

Instead of replacing the words in Synonym Replacement, we use the same functions to add the synonym words to the augmented sentence.

```python
def random_insertion(words, n):
    new_words = words.copy()
    for _ in range(n):
        add_word(new_words)
    return new_words

def add_word(new_words):
    synonyms = []
    counter = 0
    while len(synonyms) < 1:
        random_word = new_words[random.randint(0, len(new_words)-1)]
        synonyms = get_synonyms(random_word)
        counter += 1
        if counter >= 10:
            return
    random_synonym = synonyms[0]
    random_idx = random.randint(0, len(new_words)-1)
    new_words.insert(random_idx, random_synonym)
```

Alpha Hyperparameter

Since the Input File has classes and sentences to each class and we can change the output using 2 parameters (Number of Augmented Sentences) and (Alpha) which controls how much augmentation is done to a sentence. Here is what alpha simple does

Alpha ranges from 0 to 1, more alpha means more augmentation is passed to the above functions as follow

```
num_new_per_technique = int(num_aug/4)+1
n_sr = max(1, int(alpha_sr*num_words)) #Synonym Replacment Parameter
n_ri = max(1, int(alpha_ri*num_words)) #Random Insertion Parameter
n_rs = max(1, int(alpha_rs*num_words)) #Random Swaping Parameter
```

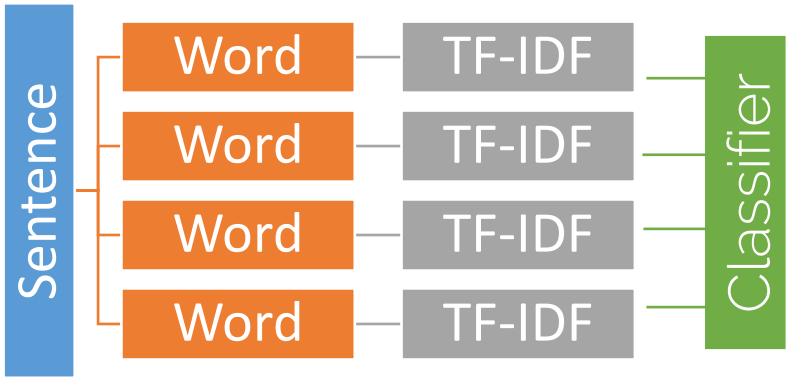
EDA Experiments:

We augment the data sets with four sentences and the hyper parameter Alpha is 0.1 but it wasn't enough for the classification we needed so we decide to go with 16 sentences and we achieved the mentioned accuracies.

Text Classification Model

Now it is time to pass the augmented data to the classifier for training and testing.

Classifier Pipeline and Architecture



Using sklearn as it's very powerful and fast in simple text classification problems with no need to use Neural Networks (RNN, LSTM, etc) as it uses simple machine learning methods in classification (SVM, SGD, Naïve Bayes, etc) with more in detail.

Vectorization:

Text Vectorization is the process of converting text into numerical representation so that the machine can understand, so each word of our sentence will have its vector and each vector represent how important this word is. Techniques vary (Bag Of Words, etc) but BoW doesn't do good with noise and doesn't give importance to certain words that distinguish each class.

TF-IDF "Term Frequency times Inverse Document Frequency"

Convert a collection of raw documents to a matrix of TF-IDF features.

```
sentence = "The oldest human fossil is the skull discovered in the Cave of Aroeira in Almonda."

TfidfVec = TfidfVectorizer()
tfidf = TfidfVec.fit_transform([sentence])

cols = TfidfVec.get_feature_names()
matrix = tfidf.todense()
pd.DataFrame(matrix,columns = cols,index=["Tf-Idf"])
```

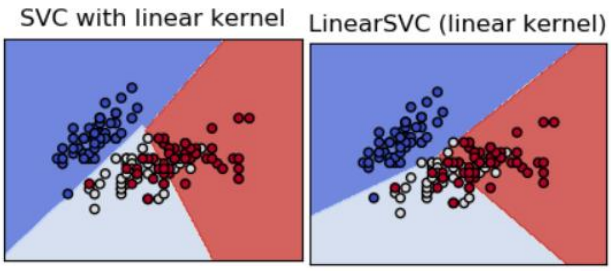
	almonda	aroeira	cave	discovered	fossil	human	in	is	of	oldest	skull	the
Tf-Idf	0.208514	0.208514	0.208514	0.208514	0.208514	0.208514	0.417029	0.208514	0.208514	0.208514	0.208514	0.625543

TF-IDF downscales weights for words (Vectors) that occur in many sentences (less informative) and distinguishes the important words of each class.

Classification:

Classification is the main part of our pipeline, after each sentence is represented in vectors of importance. We used many classifiers that sklearn offer, and will be discussed in detail.

- *(Linear) Support Vector Classification (SVC and LinearSVC):*



SVM or Support Vector Machine is a linear model for classification and regression problems. The idea of SVM is simple: The algorithm

creates a line or a hyperplane which separates the data into classes. If there are more than one class, the model uses n classes and n seperators and uses One-Vs-All to classify a class from all other classes and then does that n times to create and train n classes.

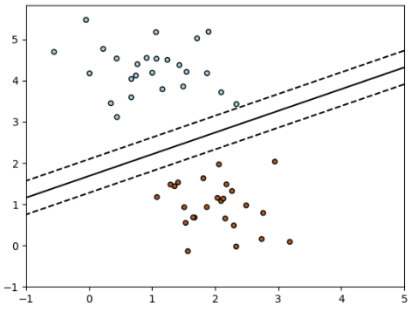
We used SVC and LinearSVC offered in sklearn as follow.

They are both with parameter kernel='linear', but LinearSVC is implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

```
from sklearn.svm import SVC
ext_clf_SVC = Pipeline([('vect', TfidfVectorizer(ngram_range=(1, 2), stop_words="english", sublinear_tf=True)),
                        ('chi',  SelectKBest(chi2, k='all')),
                        ('clf', SVC(kernel="linear"))])

pipeline = Pipeline([('vect', TfidfVectorizer(ngram_range=(1, 2), stop_words="english", sublinear_tf=True)),
                        ('chi',  SelectKBest(chi2, k='all')),
                        ('clf', LinearSVC(C=1.0, penalty='l1', max_iter=10000, dual=False))])
```

- *Stochastic Gradient Descent (SGDClassifier):*

This is also a linear model of classification but with stochastic gradient descent learning so that the loss is estimated each sample at a time and the model is updated along the way.




```
from sklearn.linear_model import SGDClassifier
ext_clf_SGD = Pipeline([('vect', TfidfVectorizer(ngram_range=(1, 2), stop_words="english", sublinear_tf=True)),
                        ('chi',  SelectKBest(chi2, k='all')),
                        ('clf', SGDClassifier())])
```

- *XBGBoost:*

The XGBoost stands for Extreme Gradient Boosting and it is a boosting algorithm based on Gradient Boosting Machines. XGboost applies regularization technique to reduce overfitting,

and it is one of the differences from the gradient boosting. Another advantage of XGBoost over classical gradient boosting is that it is fast in execution speed.


```
from xgboost import XGBClassifier
ext_clf_XGB = Pipeline([('vect', TfidfVectorizer(ngram_range=(1, 2), stop_words="english", sublinear_tf=True)),
                        ('chi',  SelectKBest(chi2, k='all')),
                        ('clf', XGBClassifier(max_depth=3, n_estimators=300, learning_rate=0.1))])
```

Results

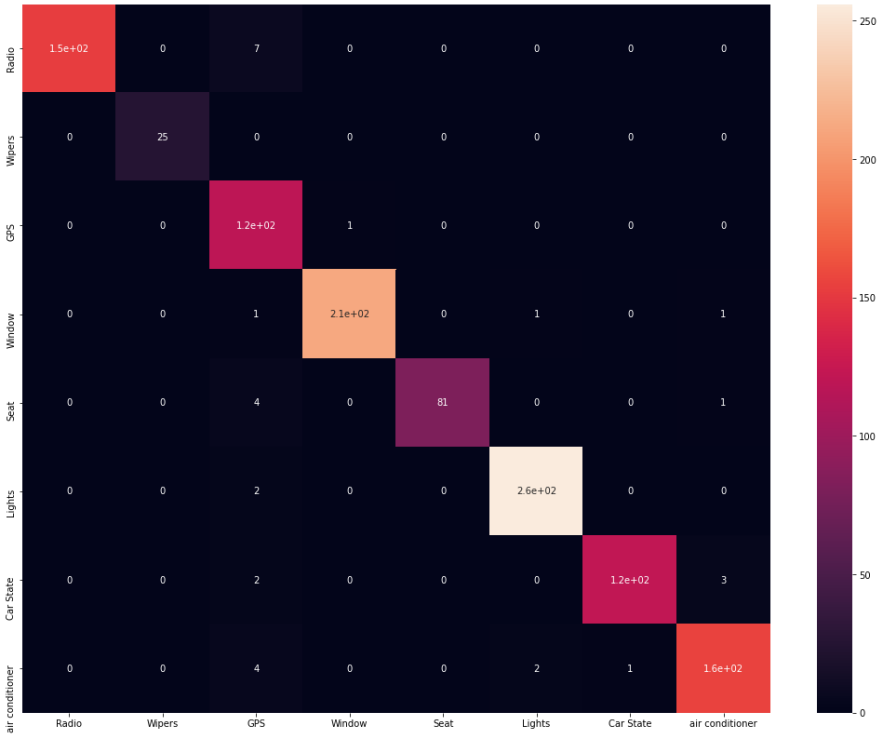
After trying all the above classifiers and augmentation techniques. We found that the best Augmentation is (16 Sentences augmentation with alpha = 0.1) as it was the most helpful to avoid overfitting the data and was recommended by EDA developers.

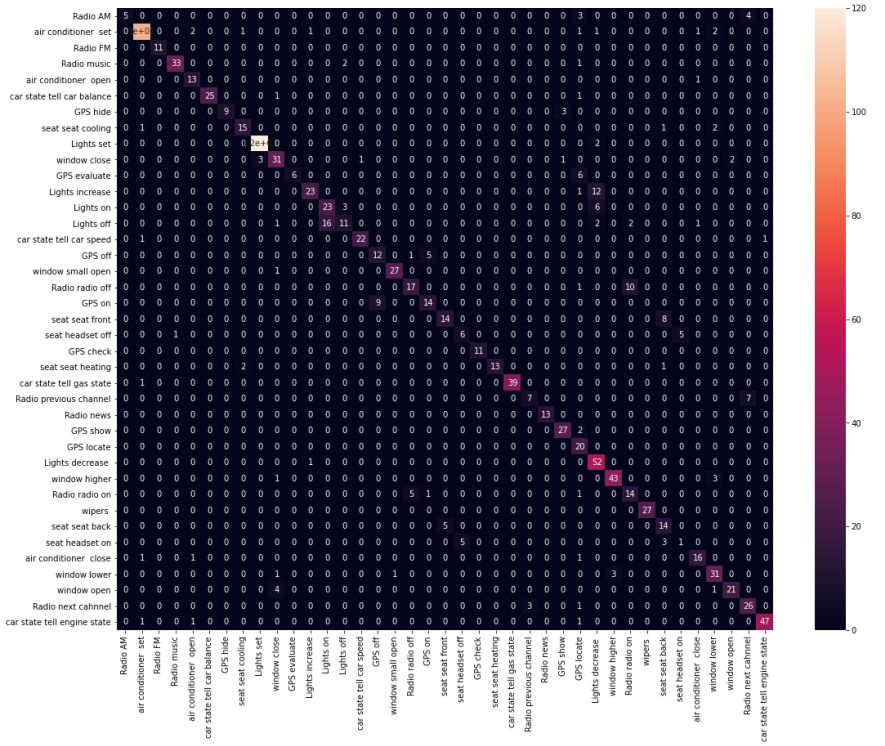
The best classifier found is SGDClassifier. Here are table of all experiments.

Classifier (CLF)	Accuracey (Main Commands) (Radio, Air Conditioner, etc)	Accuracy (Commands and Subcommands altogether) (open the radio, set the air conditioner)
SGDClassifier	All reaching 96%	86.2%
SVC		82.2608%
XGBClassifier		83.1304%
LinearSVC		83.5652%

- Confusion Matrix

Since accuracy of classifiers may mislead us, as one class may behave badly and confuse with other classes and the accuracy is still high, we tested our classifiers using confusion matrix.

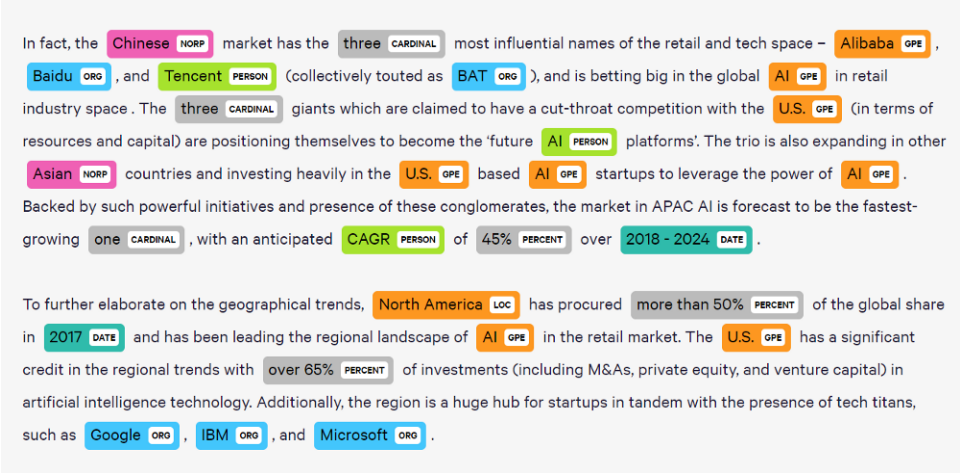




And the Confusion Matrix shows that our Classifier behaves good in the test data.

Named Entity Recognition

Since some commands maybe more specific, like (set the air conditioner to 23 degrees) or (open the window by half) we needed a technique to catch those added commands and output them. Here comer NER offered by Spacy.



NER is a subtask of information extraction that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc.

```
import spacy
from spacy import displacy
from collections import Counter
import en_core_web_sm
nermodel = en_core_web_sm.load()

#this function returns the class with the entity (precentage, vaulue, etc)
def textclass_ner(sen):
  sen = [sen]
  textclass = model.predict(sen)
  nerpre = nermodel(str(sen))
  ner = ([(X.text, X.label_) for X in nerpre.ents])
  return textclass, ner
```

PART 04

SYSTEM DEPLOYMENT

CHAPTER EIGHT

COMPRESSION & DEPLOYMENT

THE COMPUTER TAKES IN THE WAVEFORM OF YOUR SPEECH. THEN IT BREAKS THAT UP INTO WORDS, WHICH IT DOES BY LOOKING AT THE MICRO PAUSES YOU TAKE IN BETWEEN WORDS AS YOU TALK.

Compression

One of the topics R&D most concerned about is the compression of deep learning models.

Why think about compression?

Deep Convolutional Neural Networks (CNN) have set the state of the art for a variety of applications, especially in the field of computer vision. One of the main obstacles for this technology to become more widespread is the huge data storage requirement. The original AlexNet architecture requires about 240MB of memory to store the weight parameters needed for classifying a single image, while its deeper successor VGG requires considerably larger memory (528MB). One of the reasons for this, especially in those early years of deep learning, may have been given by the fact that progress was mostly driven by achieving the best results on benchmarks such as ImgeNet. Besides, it was really more about making deep learning work at all.

In a cloud-based environment with abundant computational capabilities, enabled by multiple graphical processing units (GPUs), such massive memory requirements may not be considered a restriction. However, in case of mobile or edge-based embedded devices with limited computational capabilities, such resource intensive deep neural networks cannot be readily applied. Recently, the proliferation of deep learning applications on mobile IoT devices, including smartphones, has unveiled this as a major hurdle for a wide spread use.

Thus, the design of deep neural networks that require less storage and computation power has established itself as a new research direction. Particularly, the modification of large cumbersome models that reduces the memory requirements while retaining as much of its performance as possible is referred to as compression of neural networks. Another direction is the design of more memory efficient network architectures from scratch. In the following Iwill discuss the different approaches in more detail.

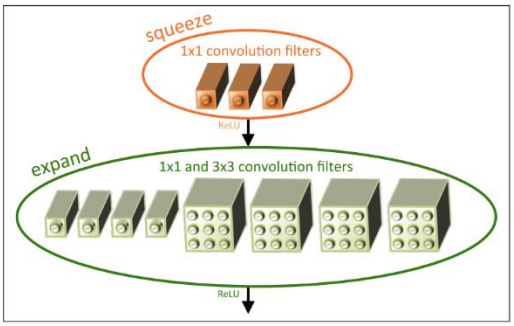
Memory efficient architectures

Rethinking the design of the CNN architecture is the most straightforward idea to arrive at more memory friendly models. First, let's recap the memory requirements for the standard building blocks of CNNs: For a fully connected layer with i input nodes and j output nodes the amount of necessary weights is given by i x j. For a convolutional layer with M input channels, N output channels and Kernel size K by K the number of parameters for this layer is given by M x N x K x K.

For instance, in VGG16 the amount of weights in one layer of the last convolutional block is given by 512 x 512 x 3 x 3 = 2.4M while feeding the final 7-by-7-feature maps into the first 4096-node-fc-layer accounts for 512 x 7 x 7 x 4096 = 102.8M parameters alone. Thus, abandoning the final fc-layers has become common practice in more recent architectures such as ResNet or Inception, since it leads to a significant decrease in memory usage while these models are by an order of magnitude deeper [3][4].

In recent time there have been a variety of papers that address the individual factors that contribute to the complexity of convolutional layers.

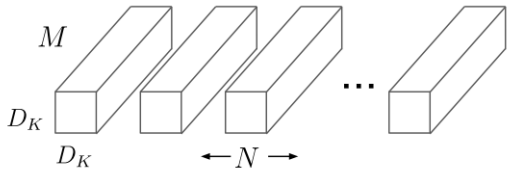
SqueezeNet in early 2016 was the first paper that was concerned with building a memory efficient architecture [5]. Here 1x1-convolutional kernels are applied to "squeeze" the input, i.e. reduce the number of channels before applying the more expensive 3x3-kernels. More precisely, the architecture consists of consecutive "Fire-Modules" that consist of 2 convolutional layers. The first layer consists entirely of 1x1-convolutions through which the number of channels is being reduced (i.e. N < M), before applying a combination of 1x1- and 3x3-kernels in the next layer, arriving at 4.8MB of parameters at AlexNet level accuracy.



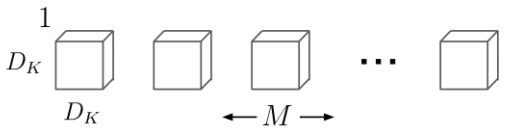
Google's MobileNets goes one step further by modifying the convolutional operation as such [6].

The key feature of this architecture is the so called depthwise separable convolutions. In standard convolutions, if we interpret a kernel as a 3-dimensional object, there are N x (K x K x M) kernels where each kernel takes the same M feature maps and modifies them according to its weights to arrive at N feature maps for the next layer. In depthwise separable convolutions the standard convolutions are replaced by 2 steps. First, a single Kernel of size K x K x M is applied to the input (Depthwise convolution). Then, after batch normalization and ReLU activation, N 1x1-Filters are applied (Pointwise convolution) to arrive at N separate feature maps in the output layer. By effectively reducing the amount of expensive K x K-Kernels by a factor of N, a significant amount of memory and computation can be saved.

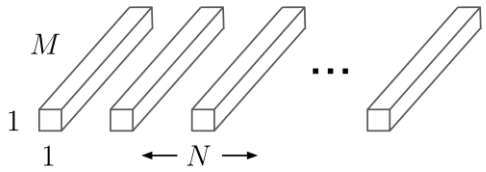
By applying this technique MobileNets offers 70.6% top-1 accuracy on ImageNet (compared to ~57% for AlexNet) with a memory requirement of 16MB.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

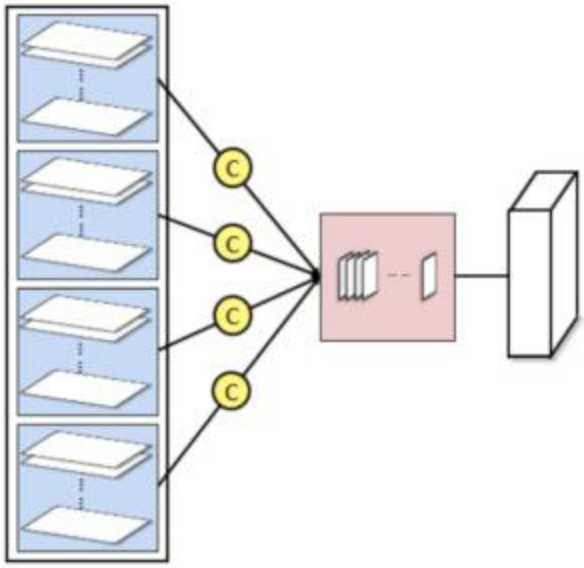


(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.

The idea of depthwise convolutions in MobileNets can be generalized to group-wise convolutions as in SEP-Nets [7] or ShuffleNets [8]. In MobileNets' Depthwise convolutions a single K x K x M-Kernel is applied to the input. Group-wise convolutions generalized this idea by partitioning the feature maps of the input into k groups and applying the same Kernel to all feature maps of the group, reducing the number of parameters by a factor of k/M compared to regular convolutions. (i.e. Group-wise convolution = Depthwise convolution iff. k=1 and Groupwise convolution = regular convolution iff. k=M).

The number of desired groups is now implemented as an argument for a convolutional layer in Pytorch.



From Float to low bitwidth — Compression through Quantization

Another straightforward way to exploit the redundancy of a Deep Neural Network is to reduce the bitwidth of its parameters. It appears that reducing the bitwidth of a large convolutional neural network from 32bit floating point down to 8bit floating point numbers as a post processing step does not significantly impact performance. This holds even for network architectures that were already designed to save memory, such as MobileNets or SqueezeNet. In fact, this is already supported by Tensorflow [9]. However, if going below 8 bits, it might be unavoidable to train the network with respect to the low bitwidth weights. Since all the optimization algorithms, such as stochastic gradient descent or Adam, employed for neural networks rely on higher precision weights and gradients, the question of how to train such networks has let to an emerging research direction.

Another strongly related idea is the quantization of not only weights but also activations (i.e. inputs into the convolution operation). While the quantization of weights is sufficient to reduce the memory requirements for the models, additionally quantizing the activations allows for significant speed up on dedicated hardware (such as FPGAs) where in the case of 1 or 2 bits for both weights and activations the matrix multiplications could be entirely replaced by logical operations. Partly for this reason, hardware acceleration of deep learning has established itself as a separate

research field that could easily fill another blog post. I will stick to the software aspect and summarize relevant papers regarding the quantization of neural networks.

The general idea is to first train a quantized model on GPU before deploying it on an edge device for inference.

The most extreme case of quantization is binarization, i.e. restricting the weights to be either -1 or +1 during interference which has been extensively studied by Courbariaux et al. [10][11][12]. In their original BinaryConnect paper, they propose to train those networks by retaining the full precision weights for the weight update but backpropagate with respect to the binary weights. In other words, the loss is calculated with respect to the binary weights, but the parameter update is performed in full precision. The binarization operation that is performed after every weight update, is simply given by the sign function (or a stochastic version of it that sets a weight w to +1 with probability max(0, min(1, (w+1)/2)) to regularize the training).

They extend this idea to binary activations in their BNN paper [11]. In this case not only the weight update but also backpropagation has to be performed with respect to the full precision weight since the gradient would be zero almost everywhere with respect to the discrete values. In their paper called XNOR [13] Rastegari et al. extend this approach with their main contribution being the introduction of a scaling factor for the binary convolution which is given by the average value of the full precision weights for every Kernel.

In their QNN paper [12] Courbariaux et al. further extend the strictly binary convolutions to higher bitcounts by generalizing the use of the signum function for binarization to linear quantization.

Another approach is given by Ternary weights where the model parameters can take the values -1, 0 and +1. Li et al. propose an appropriate scaling factor and threshold delta to extend the approach of BinaryConnect to the ternary case such that a full precision weight w is set to -1 if w<-delta, 0 if w<|delta| and 1 if w>delta[14].

A nonlinear quantization approach is proposed by Zhou et al. where weights are quantized to either powers of two or zero [15]. This idea allows efficient bit shift operations to replace floating-point multiplications even for higher bitwidths. Furthermore, the authors introduce a quantization procedure different from aforementioned publications: Instead of quantizing all weights at the same time, quantization is performed incrementally. In each iteration, only some weights are quantized and the remaining floating-point weights are retrained to compensate for the loss in accuracy. The fraction of quantized weights is stepwise increased until reaching 100%. Experiments with different architectures on ImageNet show improved accuracy compared to the full-precision model when quantizing to 5bits. Using a bitwidth of 2, hence ternary weights, ResNet-18 could be quantized with a loss in accuracy of 2.3%.

Optimal Brain Damage — Removing Redundancy through Pruning

Despite the utilization of powerful regularization techniques like dropout or weight decay some weights of a network will always contribute more to the prediction than others. The process of removing the less contributing weights to compress (and or further regularize) the network is called pruning. After the some weights have been pruned, the network typically has to be fine tuned again to have it adapt to the change.

This idea was first proposed by Yann Le Cunn et.al back in 1990 in their famous paper called Optimal Brain Damage (OBD) and was later applied to modern deep networks [16].

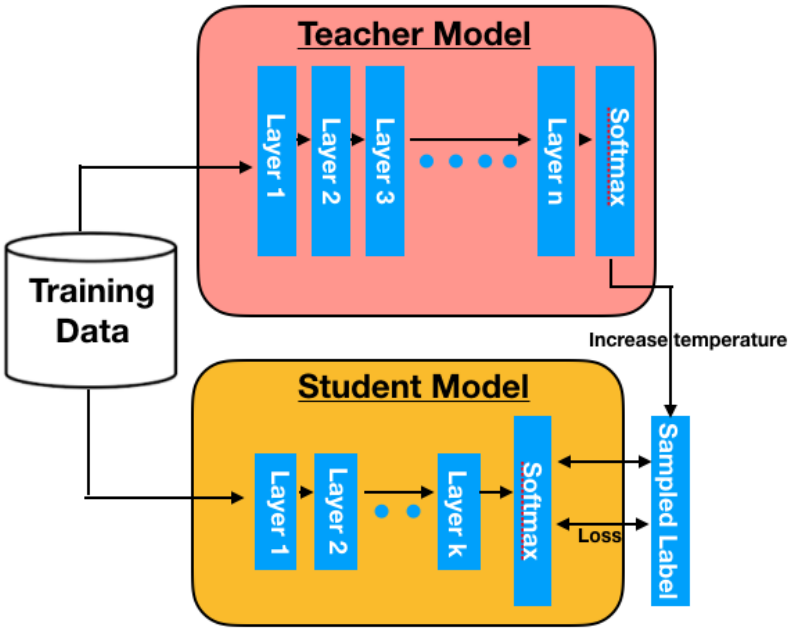
Research on pruning is mostly concerned with the question of how the contribution of the weights should be measured. In OBD the contribution is measured by the effect on the training error in the case of setting this particular parameter to zero. Obviously, this method becomes computationally infeasible for deep networks. In Deep Compression, Han et. al simply prune the weights with lowest absolute value, reducing the amount of weights to 10% of its original size for fully connected, and around 60% for convolutional layers at no loss of prediction accuracy [17].

Tu et al. proposed a method to accurately measure the Fisher information associated with a weight and use it as a measure for its contribution [18]. More recently, more advanced layer-wise methods have also been proposed [19][20].

From a large model to a small one — Knowledge Transfer through Distillation

A more generic way to compress a given model, is to force a smaller model to mimic its behaviour. In the context of Deep Learning, this idea is known as the teacher-student approach or knowledge distillation [22]. Let's say we have trained a large state-of-the-art model like ResNet or Inception (i.e. the teacher) and want to make use of its predictive power to train a smaller model (i.e. the student). To achieve this, when training the student, we perform forward passes on both, the teacher and the student, and calculate the cross entropy between the prediction of the teacher model and the student model which is added to the loss of the student. Trained in this way, the student does not only learn based on the ground truth labels but additionally learns from the teacher what is referred to as the "dark knowledge" of the model, i.e. which categories can be considered close to one another.

For example, let's say our task is to classify images into four categories: Cats, dogs, horses and zebras. A strong model will learn that cats are more similar to dogs than to horses and zebras and can be expected to give a softmax output like (0.7, 0.25, 0.035, 0.015). In practice the softmax outputs of the teacher model are smoothed by dividing the pre-softmax outputs by a factor — which becomes a hyperparameter referred to as temperature — to drive them further away from the hard targets (i.e. the ground truth labels). Thus, if the student model is additionally trained with this information rather than the ground truth labels alone it can be expected to perform better than if trained from scratch. In a more abstract sense we can say that we have compressed the teacher model by transferring its knowledge to a smaller model that gives (ideally) the same output.



This idea was first proposed by Ba and Caruana [21] and later by Hinton [22] and since then some modifications to the algorithm have been proposed that can boost the performance in specific cases [23]. Distillation has also been successfully applied to Object detection, showcasing that it scales to large benchmark datasets, even for more complex tasks [24][25].

Putting it all Together

Until today the only research papers in which several of the above techniques have been successfully combined are the work by Song Han et al. [5][17]. In their work called Deep Compression, first, a trained network is pruned by setting connections to zero if the absolute value of the weight is below a certain threshold. Second, quantization and weight sharing is applied. The weights are clustered into 256 groups for convolutional, and 32 groups for fc-layers, respectively, using k-means. Hence a weight can be represented using 8 bit (in convolutional layers) and 5 bit (in fc-layers) indices representing the centroids of the corresponding cluster. Weights are not shared across layers. The centroids are then fine tuned in an additional retraining phase, where the loss with respect to the centroid of a cluster is simply given by the additive loss of the weights that belong to it. Finally Huffman-coding is applied to the indices and centroids to further compress

the representation of the parameters. Applying this method to SqueezeNet resulted in 10x compression using 64 clusters (i.e. 6 bit representations for weights) without loss of accuracy on a network architecture which is already optimized for compression.

Deep Compression was first published in 2015 and is already quite old by the standards of literature on deep learning. In fact, the vast majority of research papers that we introduced above are much more recent, however, they are usually only concerned with exploring a single aspect of compression in isolation, e.g. pruning full precision weights or quantizing large cumbersome networks like ResNet.

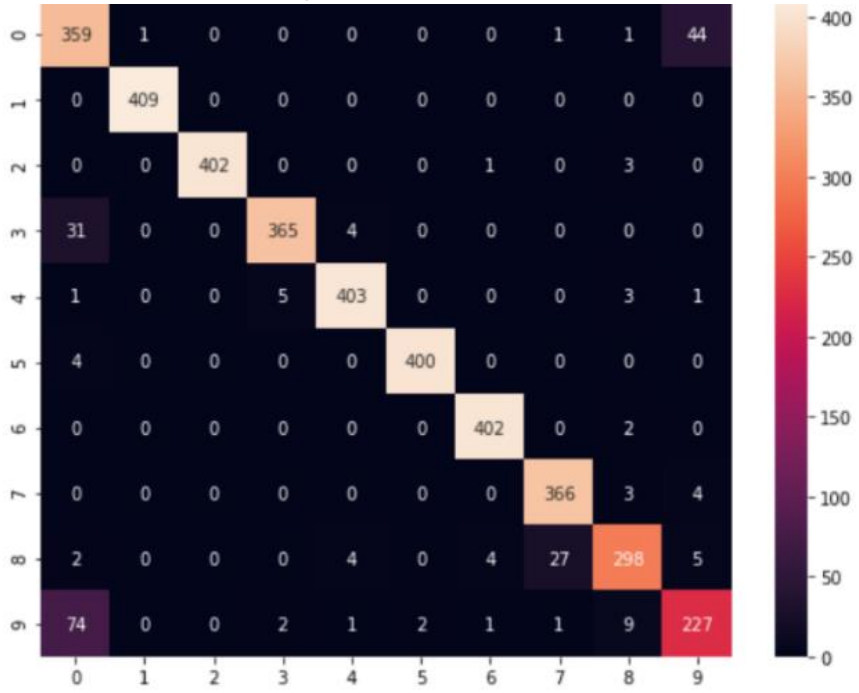
Unfortunately, it is yet not very well understood how those methods behave in combination, and there is an intuitive concern that all those approaches just exploit the same redundancy contained in deep learning models in a different way. If you don't believe me, take a MobileNets, prune it, and then binarize the weights.

Coming up with new ways for compression and combining existing approaches in an innovative fashion will be a key factor to make deep learning empowered mobile devices a widespread reality.

1- Compressing Driver Action Classifier:

We used Quantization and Pruning to compress the model while maintaining the same accuracy

The following diagram shows the confusion matrix after compression, which is exactly the same as before. While reducing the size to ¼ of its original size.



RAM usage 860 MB after Pruning compression GPU RAM usage 265 MB.

	Accuracy	Precision	Recall	F1
Driver Actions	95.4%	94%	94%	94%

2- HeadPose Model:

Since headpose model concern is accuracy to detect face angles.

The system maintains the same accuracy after pruning as follow

	Yaw	Pitch	Roll	MAE
Face's Angles	7.31	6.31	5.04	6.21

CPU RAM usage 1.4 GB after a complete compression study the best way to compress while saving the mean absolute error from exploding was Quantization, we got GPU RAM usage of 466 MB

Compression Study

```
# model Quantization
torch.quantization.prepare(model, inplace=True)
torch.quantization.convert(model, inplace=True)

yaw_error, pitch_error, roll_error = Test_HeadPose(model, softmax, test_loader, disp_every, save_out, device)
```

```
# model Pruning
for name, module in model.named_modules():
    # prune 20% of connections in all 2D-conv layers
    if isinstance(module, torch.nn.Conv2d):
        prune.l1_unstructured(module, name='weight', amount=0.2)
    elif isinstance(module, torch.nn.Linear):
        prune.l1_unstructured(module, name='weight', amount=0.2)

yaw_error, pitch_error, roll_error = Test_HeadPose(model, softmax, test_loader, disp_every, save_out, device)
```

Deployment

Due to the pandemic we are facing, we couldn't ship the Jetson Nano Board. So to compromise we used a simple CPU to test our final model pipeline and it worked great. For more on the deployment visit (LINK).

Still we searched and studied how to deploy the project on Jetson Nano.

The NVIDIA® Jetson Nano™ Developer Kit is a small AI computer for makers, learners, and developers. After following along with this brief guide, you'll be ready to start building practical AI applications, cool AI robots, and more.

To prepare your microSD card, you'll need a computer with Internet connection and the ability to read and write SD cards, either via a built-in SD card slot or adapter.

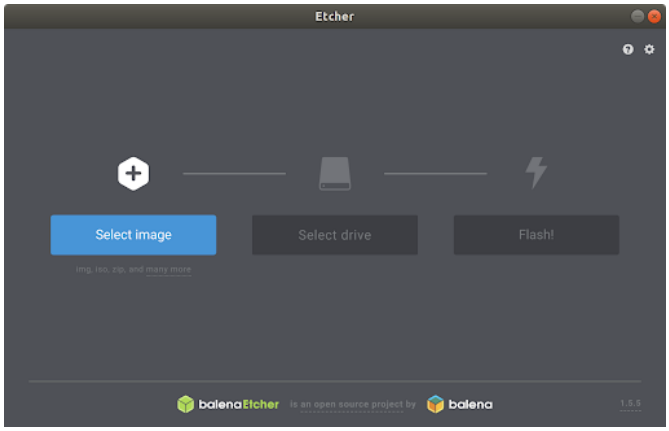
1. Download the Jetson Nano Developer Kit SD Card Image, and note where it was saved on the computer.

2. Write the image to your microSD card by following the instructions below according to the type of computer you are using: Windows, Mac, or Linux.

You can either write the SD card image using a graphical program like Etcher, or via command line.

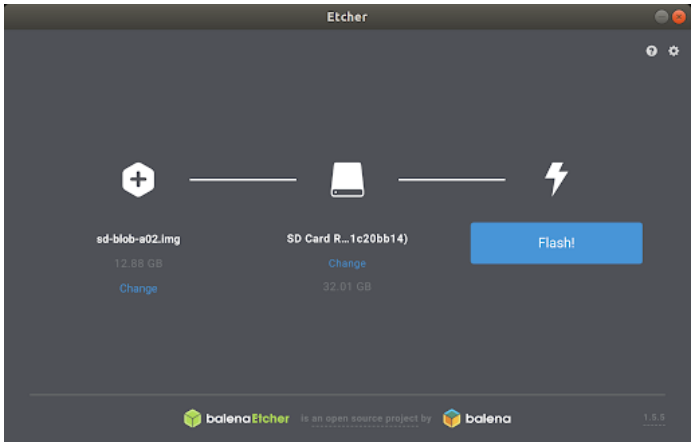
Etcher Instructions

I. Download, install, and launch Etcher.



II. Click “Select image” and choose the zipped image file downloaded earlier.

III. Insert your microSD card. If you have no other external drives attached, Etcher will automatically select the microSD card as target device. Otherwise, click “Change” and choose the correct device.



IV. Click “Flash!” Your OS may prompt for your username and password before it allows Etcher to proceed.

V. It will take Etcher 10-15 minutes to write and validate the image if your microSD card is connected via USB3.

VI. After Etcher finishes, eject the SD Card using Files application:

VII. Physically remove microSD card from the computer.

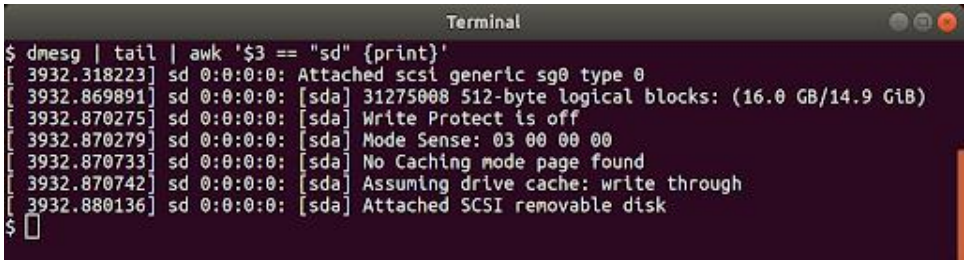
Command Line Instructions

I. Open the Terminal application by pressing Ctrl + Alt + t.

II. Insert your microSD card, then use a command like this to show which disk device was assigned to it:

```
dmesg | tail | awk '$3 == "sd" {print}'
```

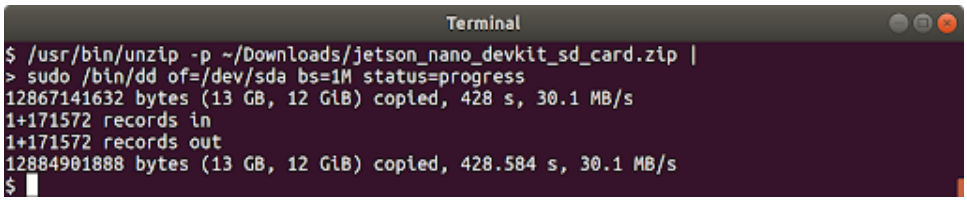
III. In this example, we can see the 16GB microSD card was assigned /dev/sda:



IV. Use this command to write the zipped SD card image to the microSD card:

```
/usr/bin/unzip -p ~/Downloads/jetson_nano_devkit_sd_card.zip |
sudo /bin/dd of=/dev/sd<x> bs=1M status=progress
```

V. For example:



VI. When the dd command finishes, eject the disk device from the command line:

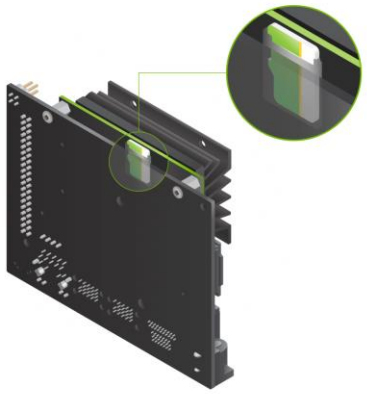
```
sudo eject /dev/sd<x>
```

VII. Physically remove microSD card from the computer.

VIII. After your microSD card is ready, proceed to Setup your developer kit.

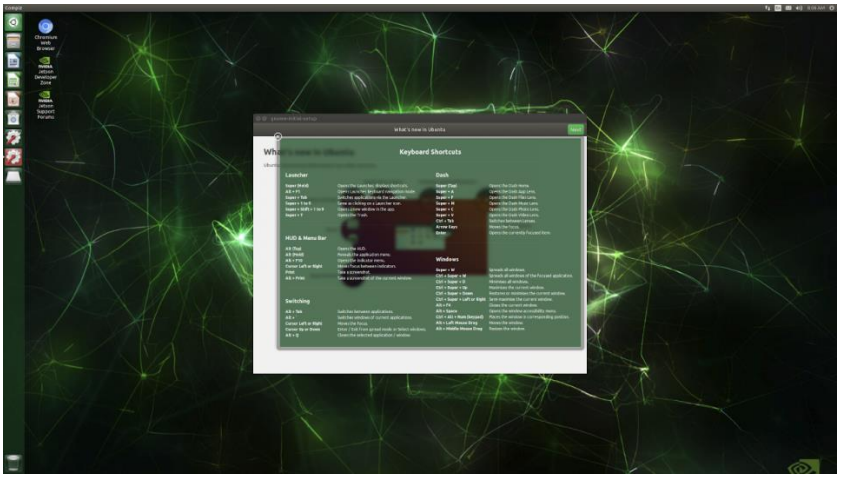
Setup Steps

1. Unfold the paper stand and place inside the developer kit box.
2. Set the developer kit on top of the paper stand.
3. Insert the microSD card (with system image already written to it) into the slot on the underside of the Jetson Nano module.



4. Power on your computer display and connect it.
5. Connect the USB keyboard and mouse.
6. Connect your Micro-USB power supply (5V⎓2A). The Jetson Nano Developer Kit will power on and boot automatically.

After Logging In You will see this screen.



PART 05

APPENDIX

APPENDIX A

PROJECT MANAGEMENT METHODOLOGY

We used multiple methodologies to fit the project requirements in specific times as follow

Waterfall for Big Picture

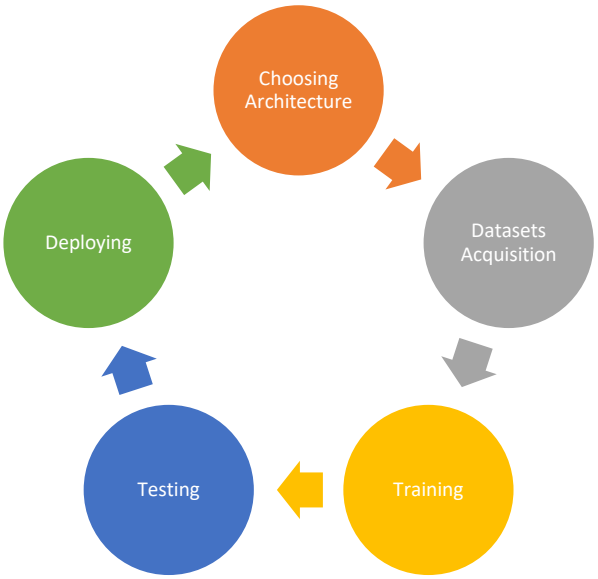
To make sure we don't fall behind. We planned the system headlines and models far ahead from the beginning (October 2019) also to make sure we have a proposed document for findings, competitions and mentorship programs. The Gantt chart plan is referred in (Appendix B).

Agile for developing the first model

The first model we developed was the Vision model (Drive Distraction System) back in November, and to make sure that each team member has a solid understanding of the system, we all worked collaboratively in this model. By using multiple architectures and training techniques.

This methodology helped us to have a better understanding of Deep Learning and Machine Learning techniques that we will use in the next models.

The first model was developed through cycles of training and testing. And once we had a good accuracy to move on to the next model, we deployed it. (Agile) for example we used Resnet Arch (18 – 50 - 151) and each team member's responsibility is to develop the model to have a better understanding of the whole picture.



Rest of the models

Each team member was assigned to a model to finish and discuss with the rest of the team and mentor. Then we would switch models to gain understanding of the whole system.

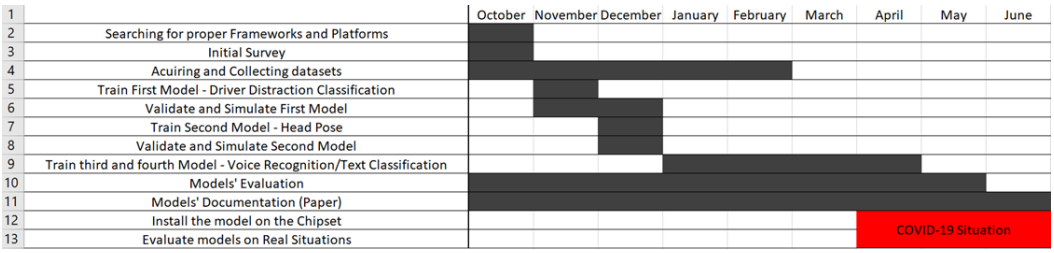
Final two months

Final two months had a huge impact because we raised the accuracy of the whole system because of what we learned throughout the year (Whole Accuracy was raised by 9% average). Then we were ready for deploying the system on the Nvidia board but due to the pandemic we couldn't ship the board. So we tested our system on mini CPU refereed in Section 8.

APPENDIX B

PLANNING

Gantt Chart



Funding

To develop the hardware system, we needed funding from organizations and funding teams. We signed to multiple organizations (names won't be added)

Mentorship

As the system software is complicated, we signed to a mentorship program and we were accepted and they were a great help to us through feedback and improving the system.

Competitions

We were ready to get in multiple competitions as it is complementary work in our University Graduation Project Judgement team. And to get recognized in multiple platforms to enhance our future careers.

We had a system proposal and a funding report as follow (Before COVID-19).

Deployment and Training

Frameworks:

- Python – Pytorch – tensorflow – Kearas – Deep Speech API.

Models Training Solutions

- Local machine with GPU – Google Cloud – Google CoLab.

APPENDIX C

ISSUES

Datasets

The main issues we encountered were dataset acquisition, we had to search for datasets to fit our system we had in mind. We found datasets from Kaggle and Local car suppliers Companies (issued by proposing the idea) in our country to use. In the text classification model, we didn't find the data that could match our criteria, so we created our simple dataset as in section 7.

Hardware Shipment and Funding

To run the model, we had to ship the board and utilities from abroad (Appendix B)

Models' Training

To train the model we had to use powerful GPUs, luckily, we had good PCs to train the models and we used Google colab

Contingency Plan

For any future risk, we had a contingency plan for any issue we thought we could face.

Scenario	Problem / Future Risk	Recovery
NLP Dataset and Language Model	We couldn't find a specific dataset for our problem of car commands	Creating our own dataset
NLP Models Training	The time for training is very huge almost a month and will require a lot of GPUs	We will use a pre-trained model that is provided by deep speech team
NVIDIA Jetson Board shipment	Not available in our country, can't arrive on time	We will acquire it from Amazon, we will try to borrow one from other people
Chipset Installment	Mic, Cameras and models will need an integrated GPU	Mics and Cameras Interfaces to the NVIDIA Jetson Board
Camera and Mics	Our system needs clean audio and clean Picture Frames	We will acquire Cameras and Mics with a nice performance from Amazon

APPENDIX D

Definitions

Artificial intelligence (AI) is wide-ranging branch of computer science concerned with building smart machines capable of performing tasks the typically require human intelligence. AI is an interdisciplinary science with multiple approaches, but advancements in machine learning and deep learning are creating a paradigm shift in virtually every sector of the tech industry.
In computer science, **artificial intelligence** (**AI**), sometimes called **machine intelligence**, is intelligence demonstrated by machines, in contrast to the **natural intelligence** displayed by humans. Leading AI textbooks define the field as the study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. Colloquially, the term "artificial intelligence" is often used to describe machines (or computers) that mimic "cognitive" functions that humans associate with the human mind, such as "learning" and "problem solving".

As machines become increasingly capable, tasks considered to require "intelligence" are often removed from the definition of AI, a phenomenon known as the AI effect. A quip in Tesler's Theorem says "AI is whatever hasn't been done yet." For instance, optical character recognition is frequently excluded from things considered to be AI, having become a routine technology.[5] Modern machine capabilities generally classified as AI include successfully understanding human speech, competing at the highest level in strategic game systems (such as chess and Go), autonomously operating cars, intelligent routing in content delivery networks, and military simulations.
Artificial intelligence was founded as an academic discipline in 1956, and in the years since has experienced several waves of optimism, followed by disappointment and the loss of funding (known as an "AI winter"), followed by new approaches, success and renewed funding. For most of its history, AI research has been divided into subfields that often fail to communicate with each other. These sub-fields are based on technical considerations, such as particular goals (e.g. "robotics" or "machine learning"), the use of particular tools ("logic" or artificial neural networks), or deep philosophical differences. Subfields have also been based on social factors (particular institutions or the work of particular researchers).
The traditional problems (or goals) of AI research include reasoning, knowledge representation, planning, learning, natural language processing, perception and the ability to move and

manipulate objects. General intelligence is among the field's long-term goals. Approaches include statistical methods, computational intelligence, and traditional symbolic AI. Many tools are used in AI, including versions of search and mathematical optimization, artificial neural networks, and methods based on statistics, probability and economics. The AI field draws upon computer science, information engineering, mathematics, psychology, linguistics, philosophy, and many other fields.

The field was founded on the assumption that human intelligence "can be so precisely described that a machine can be made to simulate it". This raises philosophical arguments about the nature of the mind and the ethics of creating artificial beings endowed with human-like intelligence. These issues have been explored by myth, fiction and philosophy since antiquity. Some people also consider AI to be a danger to humanity if it progresses unabated. Others believe that AI, unlike previous technological revolutions, will create a risk of mass unemployment.
In the twenty-first century, AI techniques have experienced a resurgence following concurrent advances in computer power, large amounts of data, and theoretical understanding; and AI techniques have become an essential part of the technology industry, helping to solve many challenging problems in computer science, software engineering and operations research
Computer science defines AI research as the study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. A more elaborate definition characterizes AI as “a system's ability to correctly interpret external data, to learn from such data, and to use those leanings to achieve specific goals and tasks through flexible adaptation”.
A typical AI analyzes its environment and takes actions that maximize its chance of success. An AI's intended utility function (or goal) can be simple ("1 if the AI wins a game of go, 0 otherwise") or complex ("Do mathematically similar actions to the ones succeeded in the past"). Goals can be explicitly defined or induced. If the AI is programmed for "reinforcement learning", goals can be implicitly induced by rewarding some types of behavior or punishing others. Alternatively, an evolutionary system can induce goals by using a "fitness function" to mutate and preferentially replicate high-scoring AI systems, like how animals evolved to innately desire certain goals such as finding food. Some AI systems, such as nearest neighbor, instead of reason by analogy, these systems are not generally given goals, except to the degree that goals are implicit in their training data. Such systems can still

be benchmarked if the non-goal system is framed as a system whose "goal" is to successfully accomplish its narrow classification task.
AI often revolves around the use of algorithms. An algorithm is a set of unambiguous instructions that a mechanical computer can execute. A complex algorithm is often built on top of other, simpler, algorithms. A simple example of an algorithm is the following (optimal for first player) recipe for play at tic-tac-toe:
1. If someone has a "threat" (that is, two in a row), take the remaining square. Otherwise,
2. If a move "forks" to create two threats at once, play that move. Otherwise,
3. Take the center square if it is free. Otherwise,
4. If your opponent has played in a corner, take the opposite corner. Otherwise,
5. Take an empty corner if one exists. Otherwise,
6. Take any empty square.

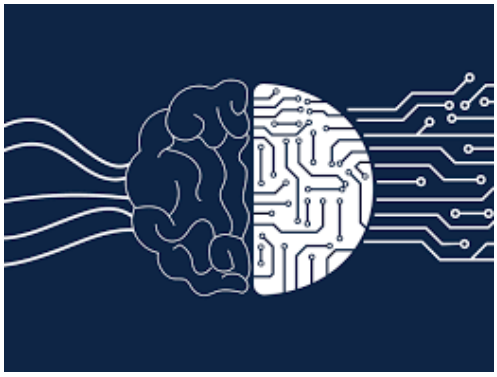
Many AI algorithms are capable of learning from data; they can enhance themselves by learning new heuristics (strategies, or "rules of thumb", that have worked well in the past), or can themselves write other algorithms. Some of the "learners" described below, including Bayesian networks, decision trees, and nearest-neighbor, could theoretically, (given infinite data, time, and memory) learn to approximate any function, including which combination of mathematical functions would best describe the world. These learners could therefore, derive all possible knowledge, by considering every possible hypothesis and matching them against the data. In practice, it is almost never possible to consider every possibility, because of the phenomenon of "combinatorial explosion", where the amount of time needed to solve a problem grows exponentially. Much of AI research involves figuring out how to identify and avoid considering broad range of possibilities that are unlikely to be beneficial.

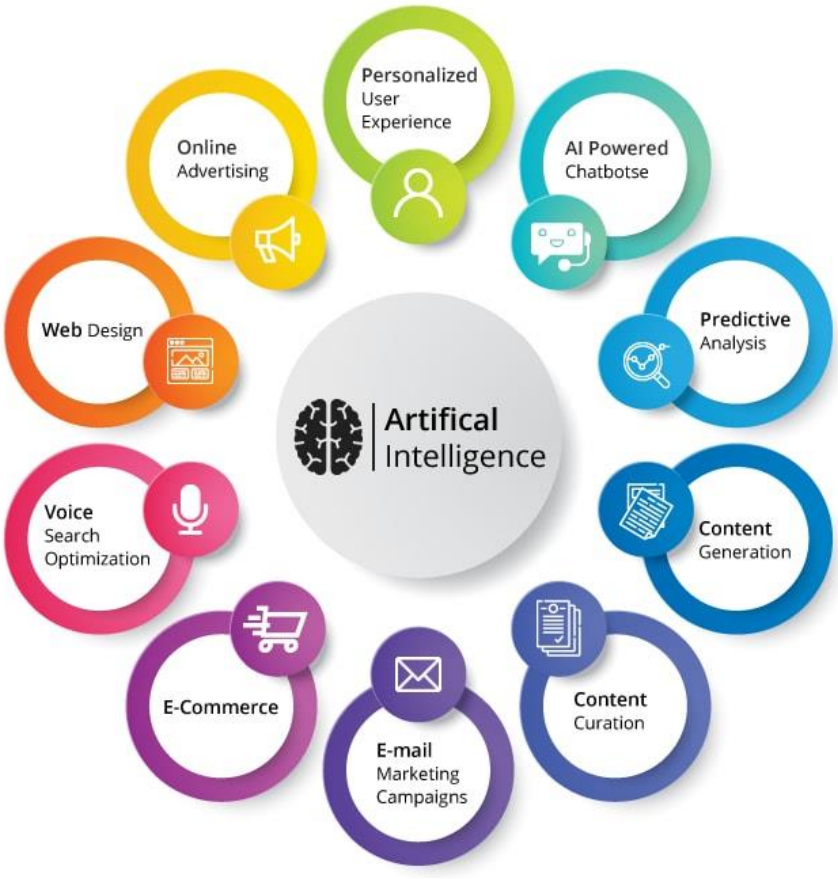
The earliest (and easiest to understand) approach to AI was symbolism (such as formal logic): "If an otherwise healthy adult has a fever, then they may have influenza". A second, more general, approach is Bayesian inference: "If the current patient has a fever, adjust the probability they have influenza in such-and-such way". The third major approach, extremely popular in routine business AI applications, are analogizes such as SVM and nearest-neighbor: "After examining the records of known past patients whose temperature, symptoms, age, and other factors mostly match the

current patient, X% of those patients turned out to have influenza". A fourth approach is harder to intuitively understand, but is inspired by how the brain's machinery works: the artificial neural network approach uses artificial "neurons" that can learn by comparing itself to the desired output and altering the strengths of the connections between its internal neurons to "reinforce" connections that seemed to be useful. These four main approaches can overlap with each other and with evolutionary systems; for example, neural nets can learn to make inferences, to generalize, and to make analogies. Some systems implicitly or explicitly use multiple of these approaches, alongside many other AI and non-AI algorithms; the best approach is often different depending on the problem.

Learning algorithms work on the basis that strategies, algorithms, and inferences that worked well in the past are likely to continue working well in the future. These inferences can be obvious, such as "since the sun rose every morning for the last 10,000 days, it will probably rise tomorrow morning as well". They can be nuanced, such as "X% of families have geographically separate species with color variants, so there is a Y% chance that undiscovered black swans exist". Learners also work on the basis of "Occam's razor": The simplest theory that explains the data is the likeliest. Therefore, according to Occam's razor principle, a learner must be designed such that it prefers simpler theories to complex theories, except in cases where the complex theory is proven substantially better.

Compared with humans, existing AI lacks several features of human "commonsense reasoning"; most notably, humans have powerful mechanisms for reasoning about "naïve physics" such as space, time, and physical interactions. This enables even young children to easily make inferences like "If I roll this pen off a table, it will fall on the floor". Humans also have a powerful mechanism of "folk psychology" that helps them to interpret natural-language sentences such as "The city councilmen refused the demonstrators a permit because they advocated violence".




Personalized User Experience
AI Powered Chatbotse
Predictive Analysis
Content Generation
Content Curation
E-mail Marketing Campaigns
E-Commerce
Voice Search Optimization
Web Design
Online Advertising
Artifical Intelligence

Machine learning & Deep learning

Machine learning is enabling computers to tackle tasks that have, until now, only been carried out by people.

From driving cars to translating speech, machine learning is driving an explosion in the capabilities of artificial intelligence, helping software make sense of the messy and unpredictable real world.

But what exactly is machine learning and what is making the current boom in machine learning possible?

Now let's talk about Machine learning! At a very high level, machine learning is the process of teaching a computer system how to make accurate predictions when fed data.

Those predictions could be answering whether a piece of fruit in a photo is a banana or an apple, spotting people crossing the road in front of a self-driving car, whether the use of the word book in a sentence relates to a paperback or a hotel reservation, whether an email is spam, or recognizing speech accurately enough to generate captions for a YouTube video.

The key difference from traditional computer software is that a human developer hasn't written code that instructs the system how to tell the difference between the banana and the apple.

Instead a machine-learning model has been taught how to reliably discriminate between the fruits by being trained on a large amount of data, in this instance likely a huge number of images labelled as containing a banana or an apple.

We talked about AI and Machine learning but what exactly is the difference between them? Machine learning may have enjoyed enormous success of late, but it is just one method for achieving artificial intelligence.

At the birth of the field of AI in the 1950s, AI was defined as any machine capable of performing a task that would typically require human intelligence.

AI systems will generally demonstrate at least some of the following traits: planning, learning, reasoning, problem solving, knowledge representation, perception, motion, and manipulation and, to a lesser extent, social intelligence and creativity.

Alongside machine learning, there are various other approaches used to build AI systems, including evolutionary computation, where algorithms undergo random mutations and combinations between generations in an attempt to "evolve" optimal solutions, and expert systems, where computers are programmed with rules that allow them to mimic the behavior of a human expert in a specific domain, for example an autopilot system flying a plane.

Now let's dig deeper into machine learning, what are machine learning types? This approach basically teaches machines by example.

During training for supervised learning, systems are exposed to large amounts of labelled data, for example images of handwritten figures annotated to indicate which number they correspond to. Given sufficient examples, a supervised-learning system would learn to recognize the clusters of pixels and shapes associated with each number and eventually be able to recognize handwritten numbers, able to reliably distinguish between the numbers 9 and 4 or 6 and 8.

However, training these systems typically requires huge amounts of labelled data, with some systems needing to be exposed to millions of examples to master a task.

As a result, the datasets used to train these systems can be vast, with Google's Open Images Dataset having about nine million images, its labeled video repository YouTube-8M linking to seven million labeled videos and ImageNet, one of the early databases of this kind, having more than 14 million categorized images. The size of training datasets continues to grow, with Facebook recently announcing it had compiled 3.5 billion images publicly available on Instagram, using hashtags attached to each image as labels. Using one billion of these photos to train an image-recognition system yielded record levels of accuracy -- of 85.4 percent -- on ImageNet's benchmark.

The laborious process of labeling the datasets used in training is often carried out using crowd working services, such as Amazon Mechanical Turk, which provides access to a large pool of low-cost labor spread across the globe. For instance, ImageNet was put together over two years by nearly 50,000 people, mainly recruited through Amazon Mechanical Turk. However, Facebook's approach of using publicly available data to train systems could

provide an alternative way of training systems using billion-strong datasets without the overhead of manual labeling.

Now what's unsupervised learning? In contrast, unsupervised learning tasks algorithms with identifying patterns in data, trying to spot similarities that split that data into categories.

An example might be Airbnb clustering together houses available to rent by neighborhood, or Google News grouping together stories on similar topics each day.

The algorithm isn't designed to single out specific types of data, it simply looks for data that can be grouped by its similarities, or for anomalies that stand out.

Now what's semi-supervised learning? The importance of huge sets of labelled data for training machine-learning systems may diminish over time, due to the rise of semi-supervised learning.

As the name suggests, the approach mixes supervised and unsupervised learning. The technique relies upon using a small amount of labelled data and a large amount of unlabeled data to train systems. The labelled data is used to partially train a machine-learning model, and then that partially trained model is used to label the unlabeled data, a process called pseudo-labelling. The model is then trained on the resulting mix of the labelled and pseudo-labelled data.

The viability of semi-supervised learning has been boosted recently by Generative Adversarial Networks (GANs), machine-learning systems that can use labelled data to generate completely new data, for example creating new images of Pokémon from existing images, which in turn can be used to help train a machine-learning model.

Were semi-supervised learning to become as effective as supervised learning, then access to huge amounts of computing power may end up being more important for successfully training machine-learning systems than access to large, labelled datasets.

Now what is reinforcement learning? A way to understand reinforcement learning is to think about how someone might learn to play an old school computer game for the first time, when they aren't familiar with the rules or how to control the game. While they may be a complete novice, eventually, by looking at the

relationship between the buttons they press, what happens on screen and their in-game score, their performance will get better.

An example of reinforcement learning is Google DeepMind's Deep Q-network, which has beaten humans in a wide range of vintage video games. The system is fed pixels from each game and determines various information about the state of the game, such as the distance between objects on screen. It then considers how the state of the game and the actions it performs in game relate to the score it achieves.

Over the process of many cycles of playing the game, eventually the system builds a model of which actions will maximize the score in which circumstance, for instance, in the case of the video game Breakout, where the paddle should be moved to in order to intercept the ball.

Let's now get back a bit and clear how the supervised machine learning work does? Everything begins with training a machine-learning model, a mathematical function capable of repeatedly modifying how it operates until it can make accurate predictions when given fresh data.

Before training begins, you first have to choose which data to gather and decide which features of the data are important.

A hugely simplified example of what data features are is given in this explainer by Google, where a machine learning model is trained to recognize the difference between beer and wine, based on two features, the drinks' color and their alcoholic volume (ABV).

Each drink is labelled as a beer or a wine, and then the relevant data is collected, using a spectrometer to measure their color and hydrometer to measure their alcohol content.

An important point to note is that the data has to be balanced, in this instance to have a roughly equal number of examples of beer and wine.

The gathered data is then split, into a larger proportion for training, say about 70 percent, and a smaller proportion for evaluation, say the remaining 30 percent. This evaluation data allows the trained model to be tested to see how well it is likely to perform on real-world data.

Before training gets underway there will generally also be a data-preparation step, during which processes such as deduplication, normalization and error correction will be carried out.

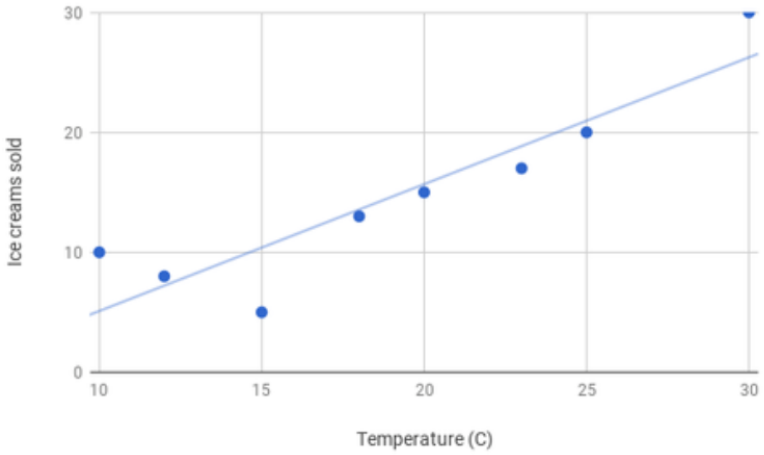
The next step will be choosing an appropriate machine-learning model from the wide variety available. Each have strengths and weaknesses depending on the type of data, for example some are suited to handling images, some to text, and some to purely numerical data.

But how does supervised machine-learning training work? Basically, the training process involves the machine-learning model automatically tweaking how it functions until it can make accurate predictions from data, in the Google example, correctly labeling a drink as beer or wine when the model is given a drink's color and ABV.

A good way to explain the training process is to consider an example using a simple machine-learning model, known as linear regression with gradient descent. In the following example, the model is used to estimate how many ice creams will be sold based on the outside temperature.

Imagine taking past data showing ice cream sales and outside temperature, and plotting that data against each other on a scatter graph -- basically creating a scattering of discrete points.

To predict how many ice creams will be sold in future based on the outdoor temperature, you can draw a line that passes through the middle of all these points, similar to the illustration below.



Once this is done, ice cream sales can be predicted at any temperature by finding the point at which the line passes through a particular temperature and reading off the corresponding sales at that point.

Bringing it back to training a machine-learning model, in this instance training a linear regression model would involve adjusting the vertical position and slope of the line until it lies in the middle of all of the points on the scatter graph.

At each step of the training process, the vertical distance of each of these points from the line is measured. If a change in slope or position of the line results in the distance to these points increasing, then the slope or position of the line is changed in the opposite direction, and a new measurement is taken.

While training for more complex machine-learning models such as neural networks differs in several respects, it is similar in that it also uses a "gradient descent" approach, where the value of "weights" that modify input data are repeatedly tweaked until the output values produced by the model are as close as possible to what is desired.

Let's check out now How to evaluate machine-learning models? Once training of the model is complete, the model is evaluated using the remaining data that wasn't used during training, helping to gauge its real-world performance.

To further improve performance, training parameters can be tuned. An example might be altering the extent to which the "weights" are altered at each step in the training process.

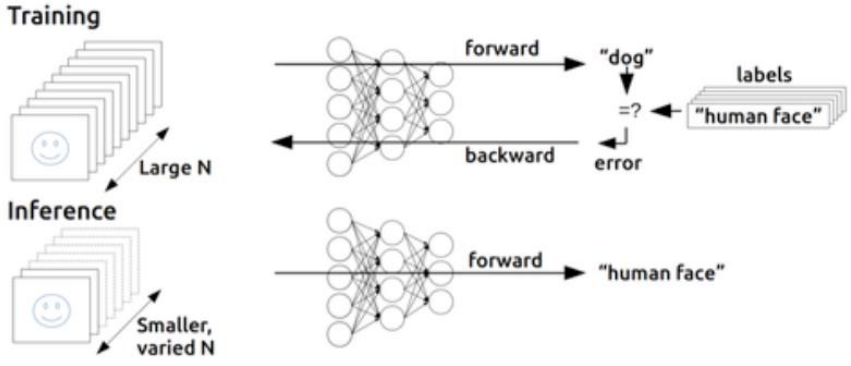
Let's talk now about what are neural networks and how are they trained? A very important group of algorithms for both supervised and unsupervised machine learning are neural networks. These underlie much of machine learning, and while simple models like linear regression used can be used to make predictions based on a small number of data features, as in the Google example with beer and wine, neural networks are useful when dealing with large sets of data with many features.

Neural networks, whose structure is loosely inspired by that of the brain, are interconnected layers of algorithms, called neurons, which feed data into each other, with the output of the preceding layer being the input of the subsequent layer.

Each layer can be thought of as recognizing different features of the overall data. For instance, consider the example of using machine learning to recognize handwritten numbers between 0 and 9. The first layer in the neural network might measure the color of the individual pixels in the image, the second layer could spot shapes, such as lines and curves, the next layer might look for larger components of the written number -- for example, the rounded loop at the base of the number 6. This carries on all the way through to the final layer, which will output the probability that a given handwritten figure is a number between 0 and 9.

The network learns how to recognize each component of the numbers during the training process, by gradually tweaking the importance of data as it flows between the layers of the network. This is possible due to each link between layers having an attached weight, whose value can be increased or decreased to alter that link's significance. At the end of each training cycle the system will examine whether the neural network's final output is getting closer or further away from what is desired -- for instance is the network getting better or worse at identifying a handwritten number 6. To close the gap between the actual output and desired output, the system will then work backwards through the neural network, altering the weights attached to all of these links between layers, as well as an associated value called bias. This process is called back-propagation.

Eventually this process will settle on values for these weights and biases that will allow the network to reliably perform a given task, such as recognizing handwritten numbers, and the network can be said to have "learned" how to carry out a specific task



An illustration of the structure of a neural network and how training works.

But what is the different between deep learning and deep neural networks? A subset of machine learning is deep learning, where neural networks are expanded into sprawling networks with a huge number of layers that are trained using massive amounts of data. It is these deep neural networks that have fueled the current leap forward in the ability of computers to carry out task like speech recognition and computer vision.

There are various types of neural networks, with different strengths and weaknesses. Recurrent neural networks are a type of neural net particularly well suited to language processing and speech recognition, while convolutional neural networks are more commonly used in image recognition. The design of neural networks is also evolving, with researchers recently devising a more efficient design for an effective type of deep neural network called long short-term memory or LSTM, allowing it to operate fast enough to be used in on-demand systems like Google Translate.

The AI technique of evolutionary algorithms is even being used to optimize neural networks, thanks to a process called neuro-evolution. The approach was recently showcased by Uber AI Labs, which released papers on using genetic algorithms to train deep neural networks for reinforcement learning problems.

And at last a man could ask why is machine learning so useful after all? While machine learning is not a new technique, interest in the field has exploded in recent years.

This resurgence comes on the back of a series of breakthroughs, with deep learning setting new records for accuracy in areas such as speech and language recognition, and computer vision.

What's made these successes possible are primarily two factors, one being the vast quantities of images, speech, video and text that is accessible to researchers looking to train machine-learning systems.

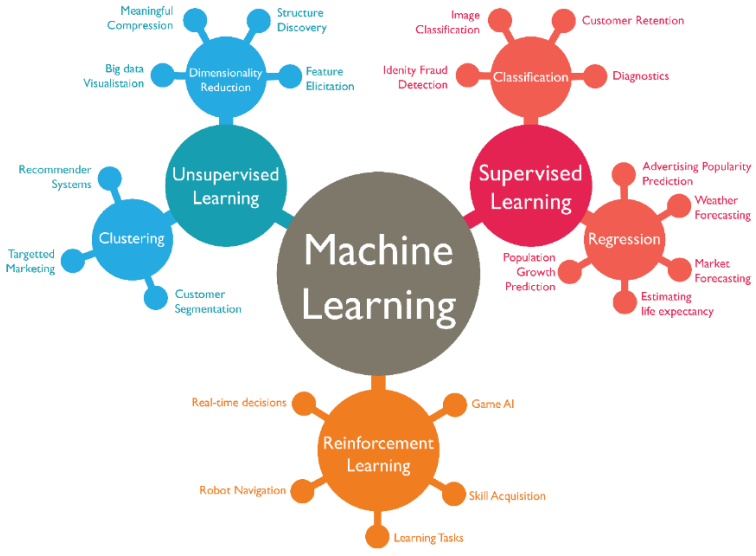
But even more important is the availability of vast amounts of parallel-processing power, courtesy of modern graphics processing units (GPUs), which can be linked together into clusters to form machine-learning powerhouses.

Today anyone with an internet connection can use these clusters to train machine-learning models, via cloud services provided by firms like Amazon, Google and Microsoft.

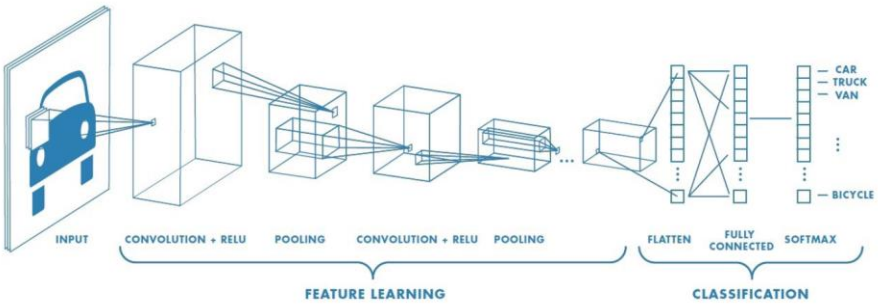
As the use of machine-learning has taken off, so companies are now creating specialized hardware tailored to running and training machine-learning models. An example of one of these custom chips is Google's Tensor Processing Unit (TPU), the latest version of which accelerates the rate at which machine-learning models built using Google's Tensorflow software library can infer information from data, as well as the rate at which they can be trained.

These chips are not just used to train models for Google DeepMind and Google Brain, but also the models that underpin Google Translate and the image recognition in Google Photo, as well as services that allow the public to build machine learning models using Google's Tensorflow Research Cloud. The second generation of these chips was unveiled at Google's I/O conference in May last year, with an array of these new TPUs able to train a Google machine-learning model used for translation in half the time it would take an array of the top-end GPUs, and the recently announced third-generation TPUs able to accelerate training and inference even further.

As hardware becomes increasingly specialized and machine-learning software frameworks are refined, it's becoming increasingly common for ML tasks to be carried out on consumer-grade phones and computers, rather than in cloud datacenters. In the summer of 2018, Google took a step towards offering the same quality of automated translation on phones that are offline as is available online, by rolling out local neural machine translation for 59 languages to the Google Translate app for iOS and Android.



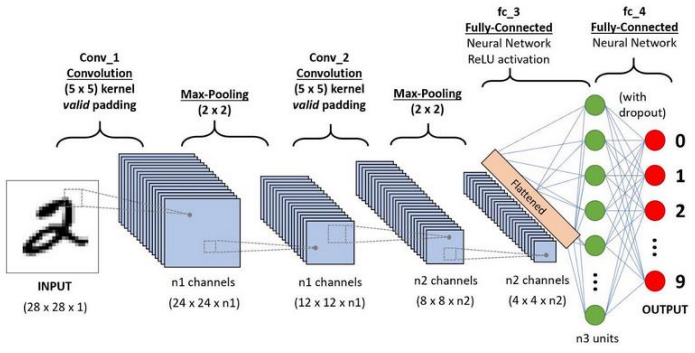
CNN



Artificial Intelligence has been witnessing a monumental growth in bridging the gap between the capabilities of humans and machines. Researchers and enthusiasts alike, work on numerous aspects of the field to make amazing things happen. One of many such areas is the domain of Computer Vision.

The agenda for this field is to enable machines to view the world as humans do, perceive it in a similar manner and even use the knowledge for a multitude of tasks such as Image & Video recognition, Image Analysis & Classification, Media Recreation, Recommendation Systems, Natural Language Processing, etc. The advancements in Computer Vision with Deep Learning has been constructed and perfected with time, primarily over one particular algorithm — a Convolutional Neural Network.

Introduction:



A CNN sequence to classify handwritten digits

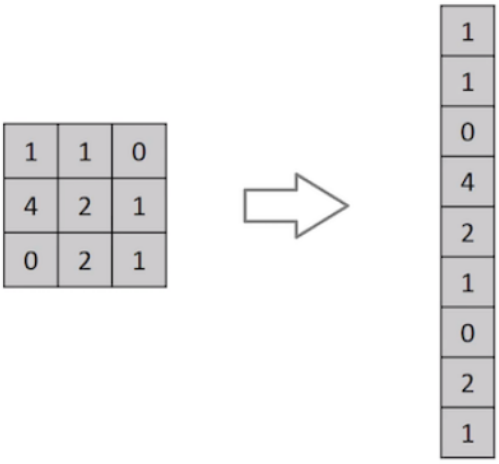
A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various

aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

Why ConvNets over Feed-Forward Neural Nets?

An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes? Uh... not really

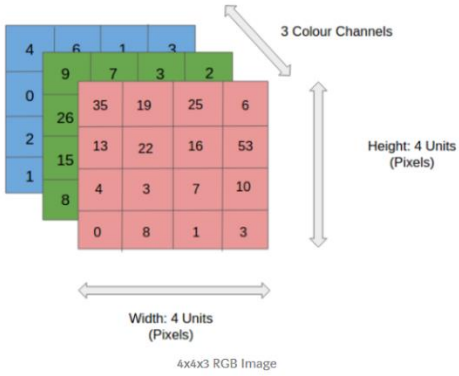


Flattening of a 3x3 image matrix into a 9x1 vector

In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

Input Image



In the figure, we have an RGB image which has been separated by its three color planes — Red, Green, and Blue. There are a number of such color spaces in which images exist — Grayscale, RGB, HSV, CMYK, etc.

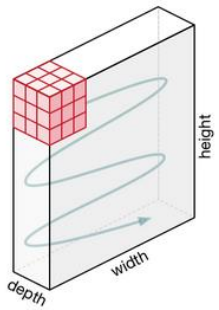
You can imagine how computationally intensive things would get once the images reach dimensions, say 8K (7680×4320). The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets

Convolution Layer — the Kernel:

Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB)

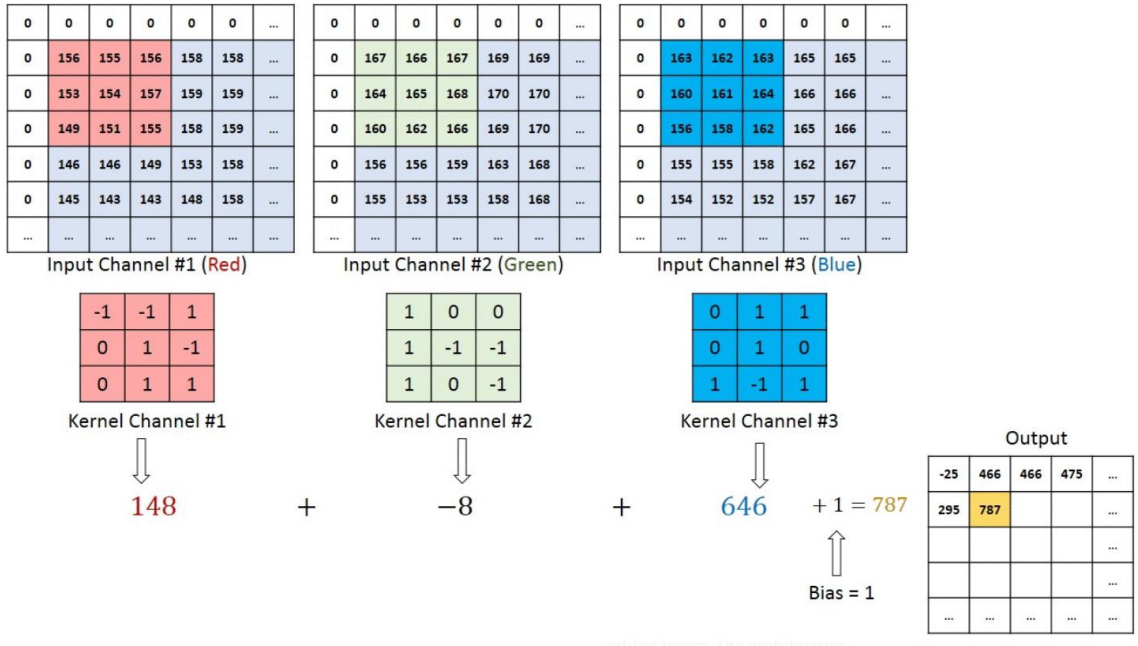
In the above demonstration, the green section resembles our 5x5x1 input image, I. The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the Kernel/Filter, K, represented in the color yellow. We have selected K as a 3x3x1 matrix.

The Kernel shifts 9 times because of Stride Length = 1 (Non-Strided), every time performing a matrix multiplication operation between K and the portion P of the image over which the kernel is hovering.



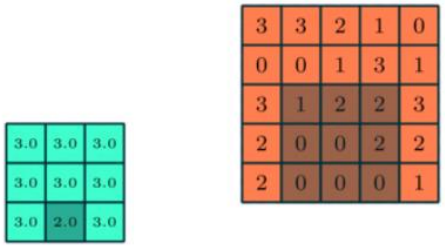
Movement of the Kernel

The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.



Pooling Layer

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.



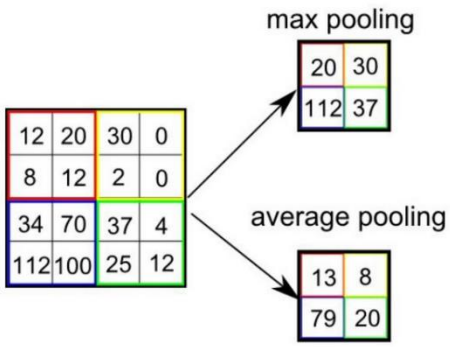
3x3 pooling over 5x5 convolved feature

There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that Max Pooling performs a lot better than Average Pooling.

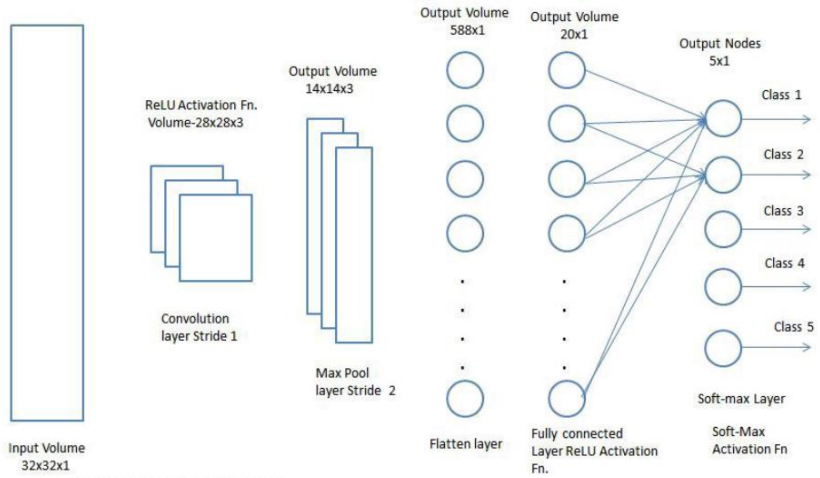
The Convolutional Layer and the Pooling Layer, together form i-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-levels details even further, but at the cost of more computational power.

After going through the above process, we have successfully enabled the model to understand the features. Moving on, we are going to flatten the final output and feed it to a regular Neural Network for classification purposes.



Types of Pooling

Classification — Fully Connected Layer (FC Layer):



Adding a Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space.

Now that we have converted our input image into a suitable form for our Multi-Level Perceptron, we shall flatten the image into a column vector. The flattened output is fed to a feed-forward neural network and backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the Softmax Classification technique.

There are various architectures of CNNs available which have been key in building algorithms which power and shall power AI as a whole in the foreseeable future. Some of them have been listed below:

- LeNet
- AlexNet
- VGGNet
- GoogLeNet
- ResNet
- ZFNet

And now we will talk about one of the most important ideas and we used it a lot in our project which is ResNet architecture!

ResNet

ResNet is one of the most powerful deep neural networks which has achieved fantabulous performance results in the ILSVRC 2015 classification challenge. ResNet has achieved excellent generalization performance on other recognition tasks and won the first place on ImageNet detection, ImageNet localization, COCO detection and COCO segmentation in ILSVRC and COCO 2015 competitions. There are many variants of ResNet architecture i.e. same concept but with a different number of layers. We have ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-110, ResNet-152, ResNet-164, ResNet-1202 etc. The name ResNet followed by a two or more digit number simply implies the ResNet architecture with a certain number of neural network layers. In this post, we are going to cover ResNet-50 in detail which is one of the most vibrant networks on its own.

Although the object classification problem is a very old problem, people are still solving it to make the model more robust. LeNet was the first Deep Neural Network that came into existence in 1998 to solve the digit recognition problem. It has 7 layers which are stacked up one over the other to recognize the digits written in the Bank Cheques. Despite the introduction of LeNet, more advanced data such as high-resolution images can't be used to train LeNet. Moreover, the computation power of computer systems during 1998 was very less.

The Deep Learning community had achieved groundbreaking results during the year 2012 when AlexNet was introduced to solve the ImageNet classification challenge. AlexNet has a total of 8 layers which are further subdivided into 5 convolution layers and 3 fully connected layers. Unlike LeNet, AlexNet has more filters to perform the convolution operation in each convolutional layer. In addition to the number of filters, the size of filters used in AlexNet was 11×11, 5×5 and 3×3. The number of parameters present in the AlexNet is around 62 million. The training of AlexNet was done in a parallel manner i.e. two Nvidia GPUs were used to train the network on the ImageNet dataset. AlexNet achieved 57% and 80.3% as its top-1 and top-5 accuracy respectively. Furthermore, the idea of Dropout was introduced to protect the model from overfitting. Consequently, a few million parameters were reduced

from 60 million parameters of AlexNet due to the introduction of Dropout.

Now let's dig deeper!

Addition of layers to make the network deep.

After the AlexNet, the next champion of ImageNet (ILSVRC-2014) classification challenge was VGG-16. There are a lot of differences between AlexNet and VGG-16. Firstly, VGG-16 has more convolution layers which imply that deep learning researchers started focusing to increase the depth of the network. Secondly, VGG-16 only uses 3×3 kernels in every convolution layer to perform the convolution operation. Unlike AlexNet, the small kernels of VGG-16 can extract fine features present in images. The architecture of VGG-16 has an overall 5 blocks. The first two blocks of the network have 2 convolution layers and 1 max-pooling layer in each block. The remaining three blocks of the network have 3 convolution layers and 1 max-pooling layer. Thirdly, three fully connected layers are added after block 5 of the network: the first two layers have 4096 neurons and the third one has 1000 neurons to do the classification task in ImageNet. Therefore, the deep learning community also refers to VGG-16 as one the widest network ever built. Moreover, the number of parameters in the first two fully-connected layers of VGG-16 has around a contribution of 100 million out of 138 million parameters of the network. The final layer is the Soft-max layer. The top-1 and top-5 accuracy of VGG-16 was 71.3% and 90.1% respectively.

GOOGLENET: NETWORK IN NETWORK:

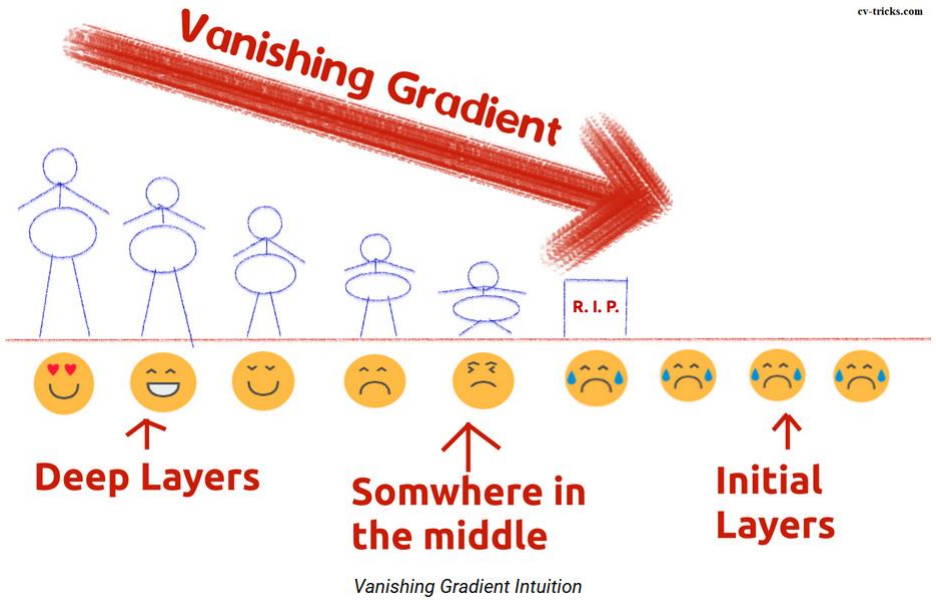
After the VGG-16 show, Google gave birth to the GoogleNet (Inception-V1): the other champion of ILSVRC-2014 with higher accuracy value than its predecessors. Unlike the prior networks, GoogleNet has a little strange architecture. Firstly, the networks such as VGG-16 have convolution layers stacked one over the other but GoogleNet arranges the convolution and pooling layers in a parallel manner to extract features using different kernel sizes. The overall intention was to increase the depth of the network and to gain a higher performance level as compared to previous winners of the ImageNet classification challenge. Secondly, the network uses 1×1 convolution operation to control the size of the volume passed for further processing in each inception module. The inception module is the collection of convolution and pooling operation performed in a parallel manner so that features can be extracted using different scales. Thirdly, the number of parameters present in the network is 24 million which makes GoogleNet a less compute-intensive model as compared to AlexNet and VGG-16. Fourthly, the network uses a Global Average Pooling layer in place of fully-connected layers. Ultimately, GoogleNet had achieved the lowest top-5 error of 6.67% in ILSVRC-2014.

STILL GRADIENTS

NOW, IT'S TIME TO TALK ABOUT THE MAIN SUBJECT OF THIS POST: THE WINNER OF ILSVRC 2015: THE DEEP RESIDUAL NETWORK

The winner of the ImageNet competition in 2015 was ResNet152 i.e. Residual Network having 152 layers variant. In this post, we will cover the concept of ResNet50 which can be generalized to any other variant of ResNet. Prior to the explanation of the deep residual network, I would like to talk about simple deep networks (networks having more number of convolution, pooling and activation layers stacked one over the other). Since 2013, the Deep Learning community started to build deeper networks because they were able to achieve high accuracy values. Furthermore, deeper networks can represent more complex features, therefore the model robustness and performance can be increased. However, stacking up more layers didn't work for the researchers.

While training deeper networks, the problem of accuracy degradation was observed. In other words, adding more layers to the network either made the accuracy value to saturate or it abruptly started to decrease. The culprit for accuracy degradation was vanishing gradient effect which can only be observed in deeper networks.



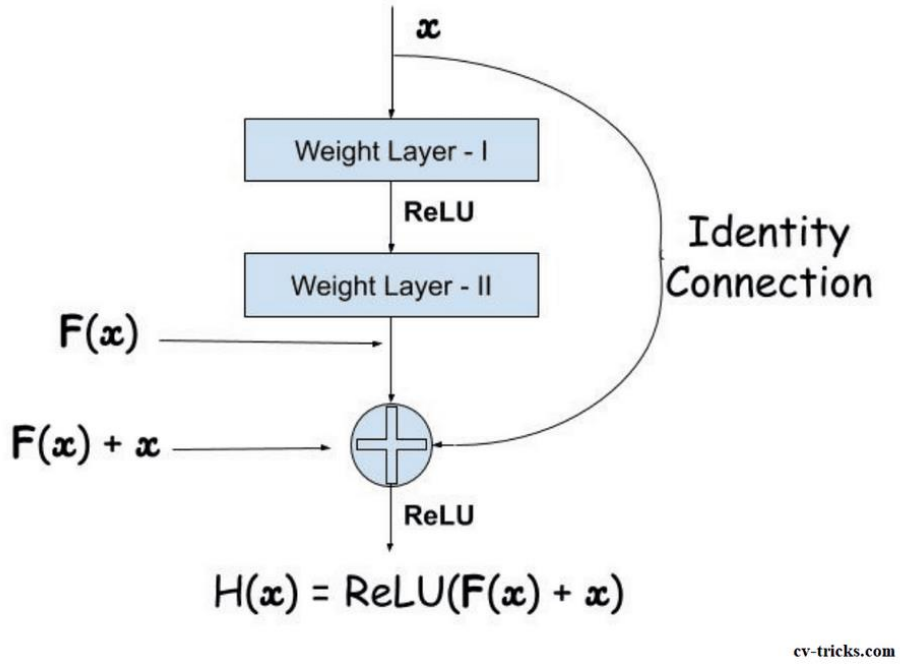
Vanishing Gradient Intuition

During the backpropagation stage, the error is calculated and gradient values are determined. The gradients are sent back to hidden layers and the weights are updated accordingly. The process of gradient determination and sending it back to the next hidden layer is continued until the input layer is reached. The gradient becomes smaller and smaller as it reaches the bottom of the network. Therefore, the weights of the initial layers will either update very slowly or remains the same. In other words, the initial layers of the network won't learn effectively. Hence, deep network training will not converge and accuracy will either starts to degrade or saturate at a particular value. Although vanishing gradient problem was addressed using the normalized initialization of weights, deeper network accuracy was still not increasing.

Now let's talk about the Residual Network

Deep Residual Network is almost similar to the networks which have convolution, pooling, activation and fully-connected layers stacked one over the other. The only construction to the simple

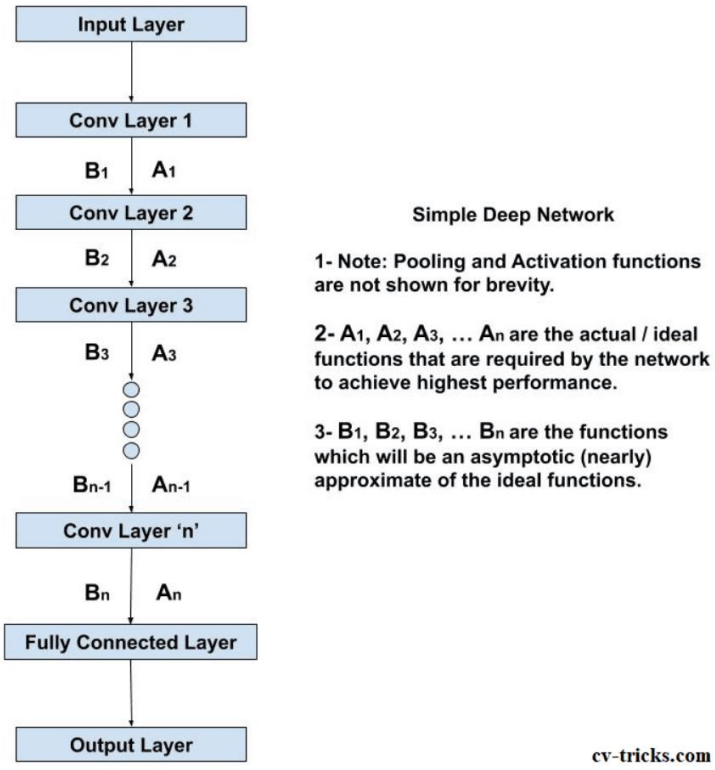
network to make it a residual network is the identity connection between the layers. The screenshot below shows the residual block used in the network. You can see the identity connection as the curved arrow originating from the input and sinking to the end of the residual block.



A Residual Block of Deep Residual Network

What is the intuition behind the residual block?

As we have learned earlier that increasing the number of layers in the network abruptly degrades the accuracy. The deep learning community wanted a deeper network architecture that can either perform well or at least the same as the shallower networks. Now, try to imagine a deep network with convolution, pooling, etc layers stacked one over the other. Let us assume that, the actual function that we are trying to learn after every layer is given by Ai(x) where A is the output function of the i-th layer for the given input x. You can refer to the next screenshot to understand the context. You can see that the output functions after every layer are A1, A2, A3, .. , An.



Assumed Deep Convolutional Neural Network

In this way of learning, the network is directly trying to learn these output functions i.e. without any extra support. Practically, it is not possible for the network to learn these ideal functions (A1, A2, A3, …. An). The network can only learn the functions say B1, B2, B3, …. Bn that are closer to A1, A2, A3, …. An. However, our assumed deep network is so much worse that even it can't learn B1, B2, B3, …. Bn which will be closer to A1, A2, A3, …. An because of the vanishing gradient effect and also due to the unsupported way of training.

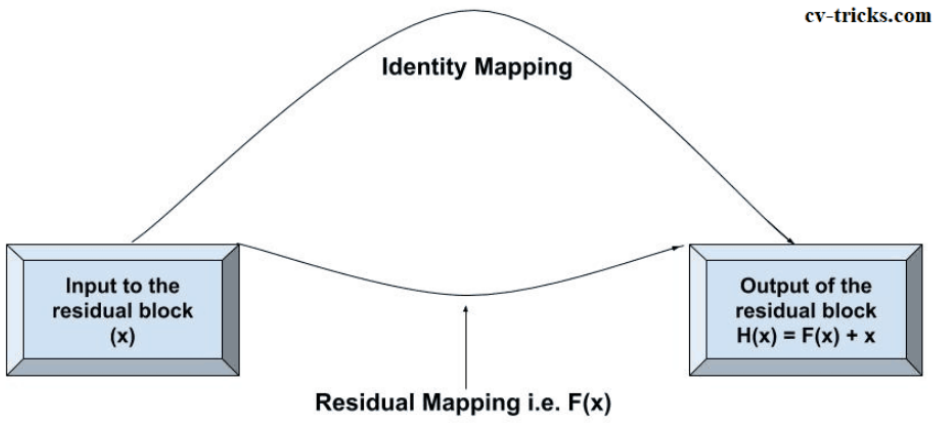
The support for the training will be given by the identity mapping addition to the residual output. Firstly, let us see what is the meaning of identity mapping? In a nutshell, applying identity mapping to the input will give you the output which is the same as the input (AI = A: where A is input matrix and I is Identity Mapping).

Traditional networks such as AlexNet, VGG-16, etc try to learn the A1, A2, A3, …. An directly as shown in the diagram of the simple deep network. In the forward pass, the input (image) is passed to the network to get the output. The error is calculated and gradients are determined and backpropagation helps the network to approximate the functions A1, A2, A3, …. An in the form of B1, B2, B3, …. Bn. The creators of ResNet thought that:

"IF THE MULTIPLE NON-LINEAR LAYERS CAN ASYMPTOTICALLY APPROXIMATE COMPLICATED FUNCTIONS, THEN IT IS EQUIVALENT TO HYPOTHESIZE THAT THEY CAN ASYMPTOTICALLY APPROXIMATE THE RESIDUAL FUNCTION".

What is a Residual Function?

The simple answer to this question is that the residual function (also known as residual mapping) is the difference between the input and output of the residual block under question. In other words, residual mapping is the value that will be added to the input to approximate the final function (A1, A2, A3, …. An) of the block. You can also assume that the residual mapping is the amount of error which can be added to input so as to reach the final destination i.e. to approximate the final function. You can visualize the Residual Mapping as shown in the next figure. You can see that the Residual Mapping is acting as a bridge between the input and the output of the block. Note that the weight layers and activation function are not shown in the diagram but they are actually present in the network.

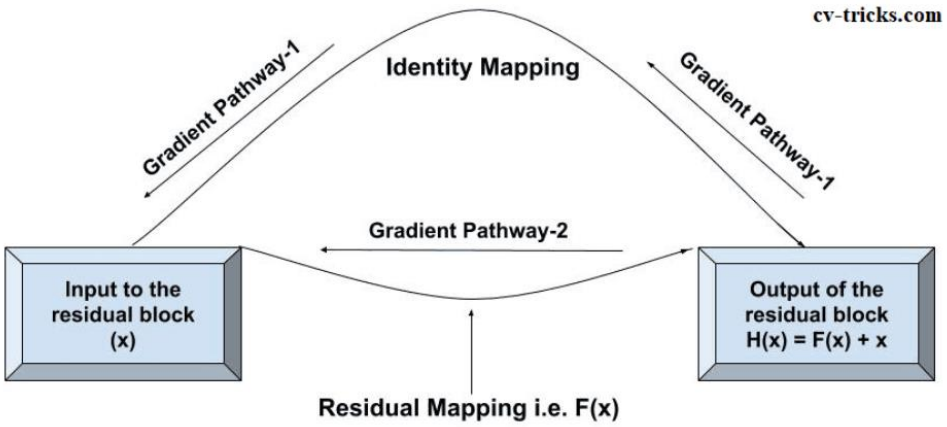


Intuitive representation of Residual Function

Let us change our naming conventions to make this post compatible with ResNet paper. Fig-1, shows the residual block. The function which should be learned as a final result of the block is represented as H(x). The input to the block is x and the residual mapping

Why identity mapping will work? How does the identity connection affect the performance of the network?

During the time of backpropagation, there are two pathways for the gradients to transit back to the input layer while traversing a residual block. In the next diagram, you can see that there are two pathways: pathway-1 is the identity mapping way and pathway-2 is the residual mapping way



Gradient Pathways in ResNet

We have already discussed the vanishing gradients problem in the simple deep network. Now we will try to explain the solution for the same keeping the identity connection in mind. Firstly, we will see how to represent the residual block F(x) mathematically?

$$y = F(x, \{W_i\}) + x$$

Where y is the output function, x is the input to the residual block and F(x, {Wi}) is the residual block. Note that the residual block contains weight layers which are represented as Wi where 1 ≤ i ≤ number of layers in a residual block. Also, the term F(x, {Wi}) for 2 weight layers in a residual block can be simplified and can be written as follows:

$$F(x, \{W_i\}) = W_2\,\sigma(W_1 x)$$

Where $\boldsymbol{\sigma}$ is the ReLU activation function and the second non-linearity is added after the addition with identity mapping i.e. H(x) = $\boldsymbol{\sigma}$(y).

When the computed gradients pass from the Gradient Pathway-2, two weight layers are encountered which are W1 and W2 in our residual function F(x). The weights or the kernels in the weight layers W1 and W2 are updated and new gradient values are calculated. In the case of initial layers, the newly computed values will either become small or eventually vanish. To save the gradient values from vanishing, the shortcut connection (identity mapping) will come into the picture. The gradients can directly pass through the Gradient Pathway-1 shown in the previous diagram. In Gradient Pathway-1, the gradients don't have to encounter any weight layer, hence, there won't be any change in the value of computed gradients. The residual block will be skipped at once and the gradients can reach the initial layers which will help them to learn the correct weights. Also, ResNet version 1 has ReLU function after the addition operation, therefore, gradient values will be changed as soon as they are getting inside the residual block.

Basic properties and assumptions regarding the identity connection:

- The addition of the identity connection does not introduce extra parameters. Therefore, the computation complexity for simple deep networks and deep residual networks is almost the same.
- The dimensions of x and F must be the same for performing the addition operation. The dimensions can be matched by using one of the following ways:
    - Extra Zero entries should be padded for increasing dimensions. This is not going to introduce any extra parameter.
    - Projection shortcut can be used to match the dimensions (1 x 1 convolutions).

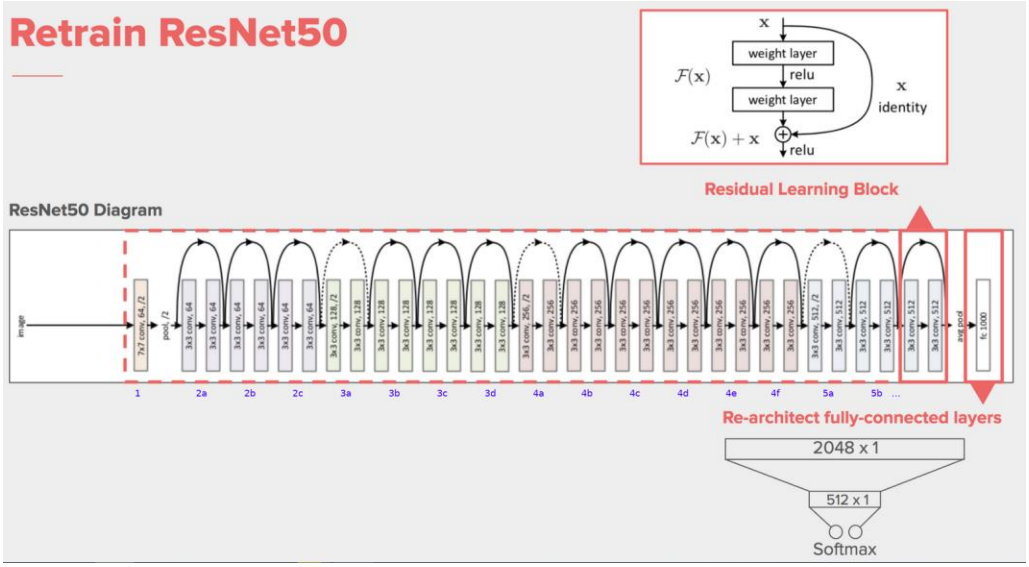
$$y = F(x, \{W_i\}) + W_s x$$

Architecture of ResNet-50:

Now we'll talk about the architecture of ResNet50. The architecture of ResNet50 has 4 stages as shown in the diagram below. The network can take the input image having height, width as multiples of 32 and 3 as channel width. For the sake of explanation, we will consider the input size as 224 x 224 x 3. Every ResNet architecture performs the initial convolution and max-pooling using 7×7 and 3×3 kernel sizes respectively. Afterward, Stage 1 of the network starts and it has 3 Residual blocks containing 3 layers each. The size of kernels used to perform the convolution operation in all 3 layers of the block of stage 1 are 64, 64 and 128 respectively. The curved arrows refer to the identity connection. The dashed connected arrow represents that the convolution operation in the Residual Block is performed with stride 2, hence, the size of input will be reduced to half in terms of height and width but the channel width will be doubled. As we progress from one stage to another, the channel width is doubled and the size of the input is reduced to half.

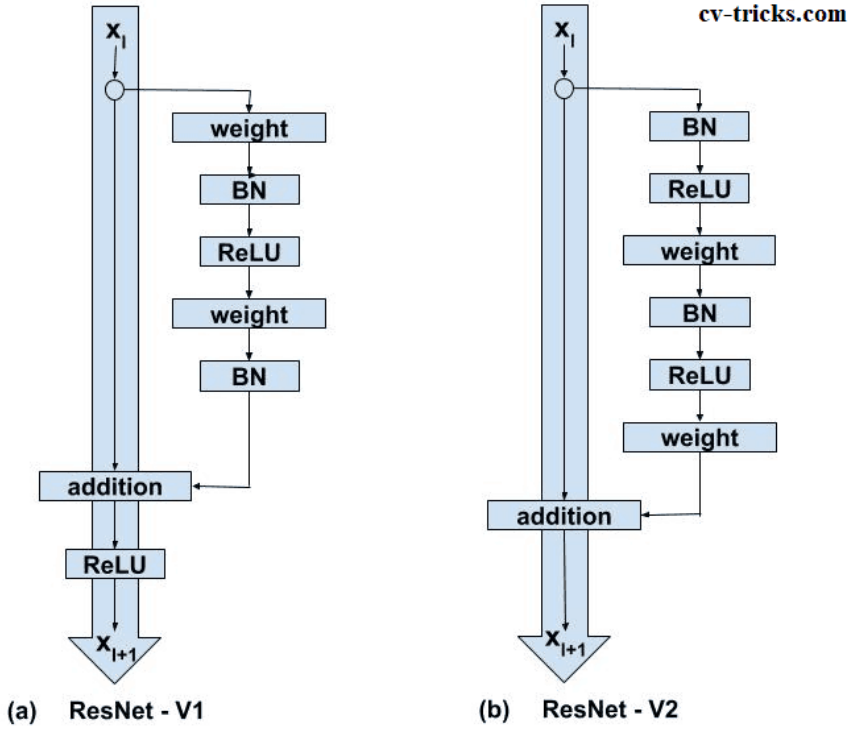
For deeper networks like ResNet50, ResNet152, etc, bottleneck design is used. For each residual function F, 3 layers are stacked one over the other. The three layers are 1×1, 3×3, 1×1 convolutions. The 1×1 convolution layers are responsible for reducing and then restoring the dimensions. The 3×3 layer is left as a bottleneck with smaller input/output dimensions.

Finally, the network has an Average Pooling layer followed by a fully connected layer having 1000 neurons (ImageNet class output).



ResNet – V2:

Till now we have discussed the ResNet50 version 1. Now, we will discuss the ResNet50 version 2 which is all about using the pre-activation of weight layers instead of post-activation. The figure below shows the basic architecture of the post-activation (original version 1) and the pre-activation (version 2) of versions of ResNet.



ResNet V1 and ResNet V2

The major differences between ResNet – V1 and ResNet – V2 are as follows:

- ResNet V1 adds the second non-linearity after the addition operation is performed in between the x and F(x). ResNet V2 has removed the last non-linearity, therefore, clearing the path of the input to output in the form of identity connection.

- ResNet V2 applies Batch Normalization and ReLU activation to the input before the multiplication with the weight matrix (convolution operation). ResNet V1 performs the convolution followed by Batch Normalization and ReLU activation.

RNN

Recurrent Neural Networks (RNNs) add an interesting twist to basic neural networks. A vanilla neural network takes in a fixed size vector as input which limits its usage in situations that involve a 'series' type input with no predetermined size.

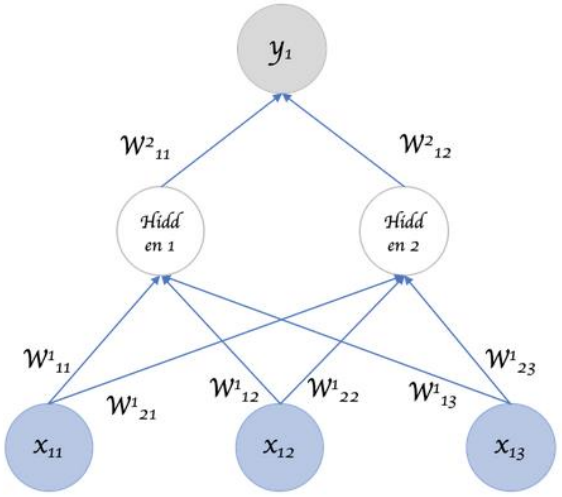


Figure 1: A vanilla network representation, with an input of size 3 and one hidden layer and one output layer of size 1.

RNNs are designed to take a series of input with no predetermined limit on size. One could ask what the big deal is, I can call a regular NN repeatedly too?

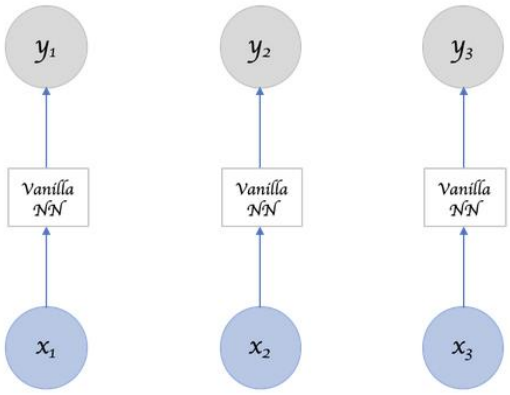


Figure 2: Can I simply not call a vanilla network repeatedly for a 'series' input?

Sure can, but the 'series' part of the input means something. A single input item from the series is related to others and likely has an influence on its neighbors. Otherwise it's just "many" inputs, not a "series" input (duh!).

So we need something that captures this relationship across inputs meaningfully

Recurrent Neural Networks

Recurrent Neural Network remembers the past and it's decisions are influenced by what it has learnt from the past. Note: Basic feed forward networks "remember" things too, but they remember things they learnt during training. For example, an image classifier learns what a "1" looks like during training and then uses that knowledge to classify things in production.

While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s). It's part of the network. RNNs can take one or more input vectors and produce one or more output vectors and the output(s) are influenced not just by weights applied on inputs like a regular NN, but also by a "hidden" state vector representing the context based on prior input(s)/output(s). So, the same input could produce a different output depending on previous inputs in the series.

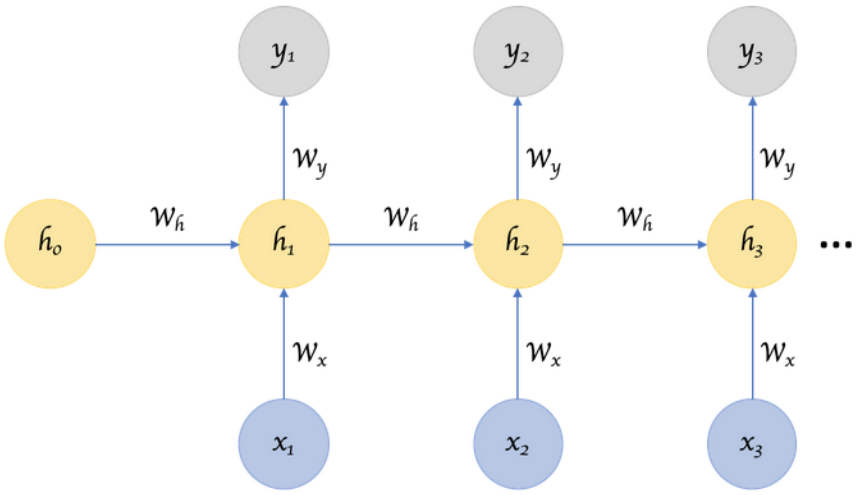


Figure 3: A Recurrent Neural Network, with a hidden state that is meant to carry pertinent information from one input item in the series to others.

Parameter Sharing:

You might have noticed another key difference between Figure 1 and Figure 3. In the earlier, multiple different weights are applied to the different parts of an input item generating a hidden layer neuron, which in turn is transformed using further weights to produce an output. There seems to be a lot of weights in play here. Whereas in Figure 3, we seem to be applying the same weights over and over again to different items in the input series.

I am sure you are quick to point out that we are kinda comparing apples and oranges here. The first figure deals with "a" single input whereas the second figure represents multiple inputs from a series. But nevertheless, intuitively speaking, as the number of inputs increase, shouldn't the number of weights in play increase as well? Are we losing some versatility and depth in Figure 3?

Perhaps we are. We are sharing parameters across inputs in Figure 3. If we don't share parameters across inputs, then it becomes like a vanilla neural network where each input node requires weights of their own. This introduces the constraint that the length of the input has to be fixed and that makes it impossible to leverage a series type input where the lengths differ and is not always known.

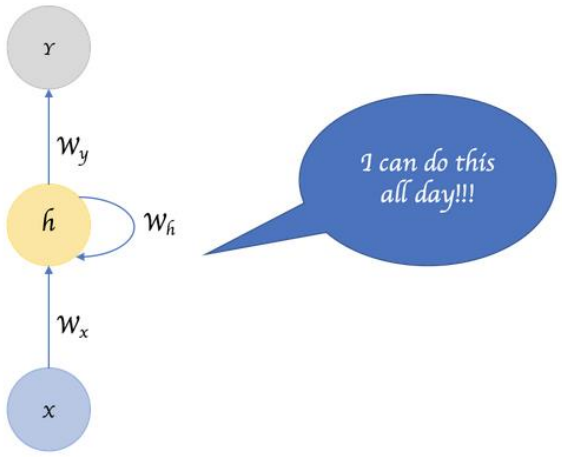


Figure 4: Parameter sharing helps get rid of size limitations

Deep RNNs

While it's good that the introduction of hidden state enabled us to effectively identify the relationship between the inputs, is there a way we can make a RNN "deep" and gain the multi-level abstractions and representations we gain through "depth" in a typical neural network?

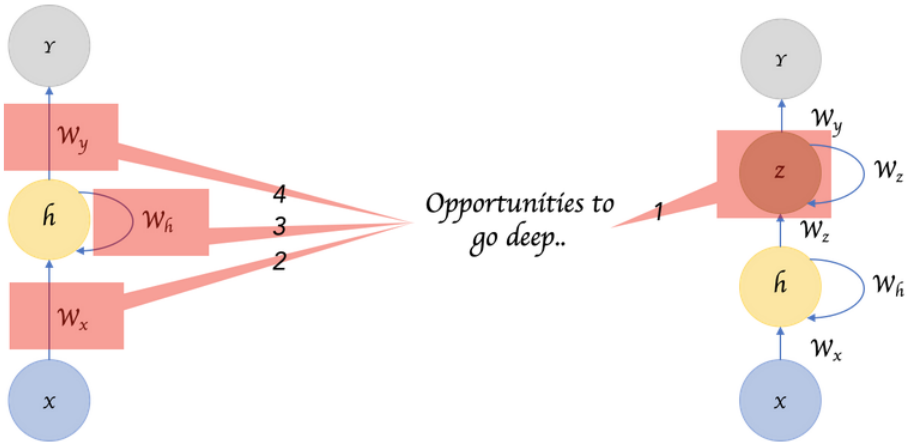


Figure 4: We can increase depth in three possible places in a typical RNN. This paper by Pascanu et al., explores this in detail.

Here are four possible ways to add depth. (1) Perhaps the most obvious of all, is to add hidden states, one on top of another, feeding the output of one to the next. (2) We can also add additional nonlinear hidden layers between input to hidden state (3) we can increase depth in the hidden to hidden transition (4) we can increase depth in the hidden to output transition. This paper by

Pascanu et al., explores this in detail and in general established that deep RNNs perform better than shallow RNNs.

Bidirectional RNNs

Sometimes it's not just about learning from the past to predict the future, but we also need to look into the future to fix the past. In speech recognition and handwriting recognition tasks, where there could be considerable ambiguity given just one part of the input, we often need to know what's coming next to better understand the context and detect the present.

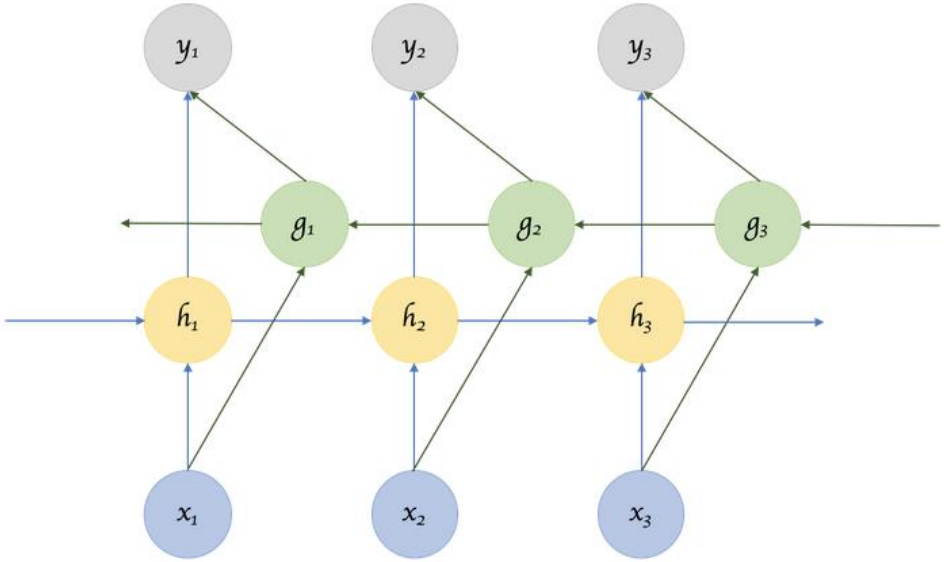


Figure 5: Bidirectional RNNs

This does introduce the obvious challenge of how much into the future we need to look into, because if we have to wait to see all inputs then the entire operation will become costly. And in cases like speech recognition, waiting till an entire sentence is spoken might make for a less compelling use case. Whereas for NLP tasks, where the inputs tend to be available, we can likely consider entire sentences all at once. Also, depending on the application, if the sensitivity to immediate and closer neighbors is higher than inputs that come further away, a variant that looks only into a limited future/past can be modeled.

Recursive Neural Networks:

A recurrent neural network parses the inputs in a sequential fashion. A recursive neural network is similar to the extent that the transitions are repeatedly applied to inputs, but not necessarily in a sequential fashion. Recursive Neural Networks are a more general form of Recurrent Neural Networks. It can operate on any hierarchical tree structure. Parsing through input nodes, combining child nodes into parent nodes and combining them with other child/parent nodes to create a tree like structure. Recurrent Neural Networks do the same, but the structure there is strictly linear. I.e. weights are applied on the first input node, then the second, third and so on.

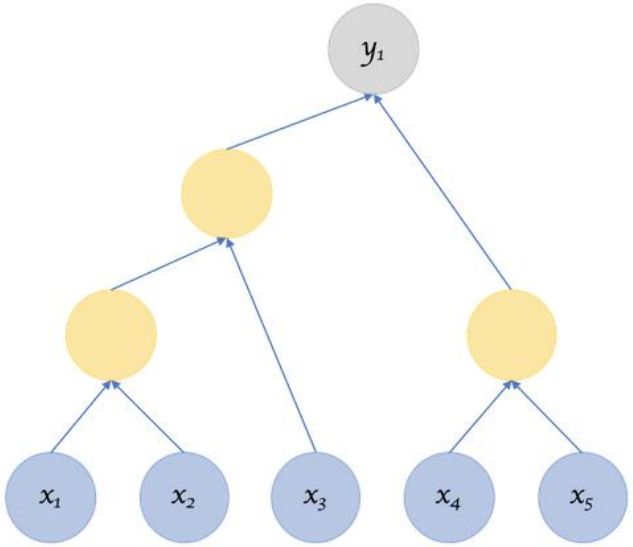


Figure 6: Recursive Neural Net

But this raises questions pertaining to the structure. How do we decide that? If the structure is fixed like in Recurrent Neural Networks then the process of training, backprop etc makes sense in that they are similar to a regular neural network. But if the structure isn't fixed, is that learnt as well? This paper and this paper by Socher et al., explores some of the ways to parse and define the structure, but given the complexity involved, both computationally and even more importantly, in getting the requisite training data, recursive neural networks seem to be lagging in popularity to their recurrent cousin.

Encoder Decoder Sequence to Sequence RNNs

Encoder Decoder or Sequence to Sequence RNNs are used a lot in translation services. The basic idea is that there are two RNNs, one an encoder that keeps updating its hidden state and produces a final single "Context" output. This is then fed to the decoder, which translates this context to a sequence of outputs. Another key difference in this arrangement is that the length of the input sequence and the length of the output sequence need not necessarily be the same.

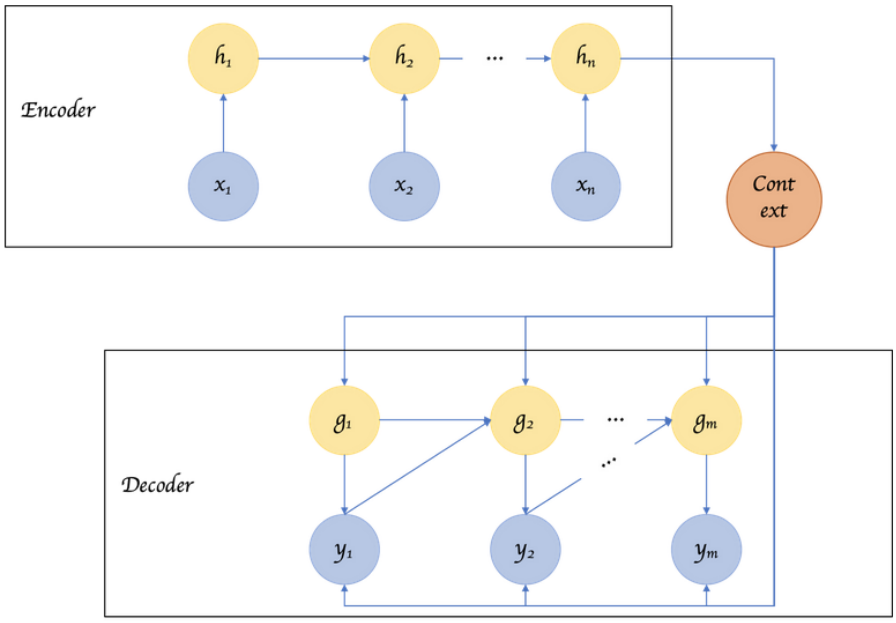


Figure 6: Encoder Decoder or Sequence to Sequence RNNs

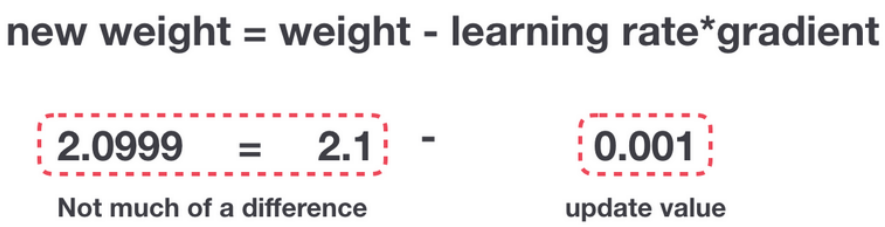
LSTM & GRU

In this article, we'll start with the intuition behind LSTM's and GRU's. Then I'll explain the internal mechanisms that allow LSTM's and GRU's to perform so well. If you want to understand what's happening under the hood for these two networks.

The Problem, Short-term Memory:

Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.

During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.

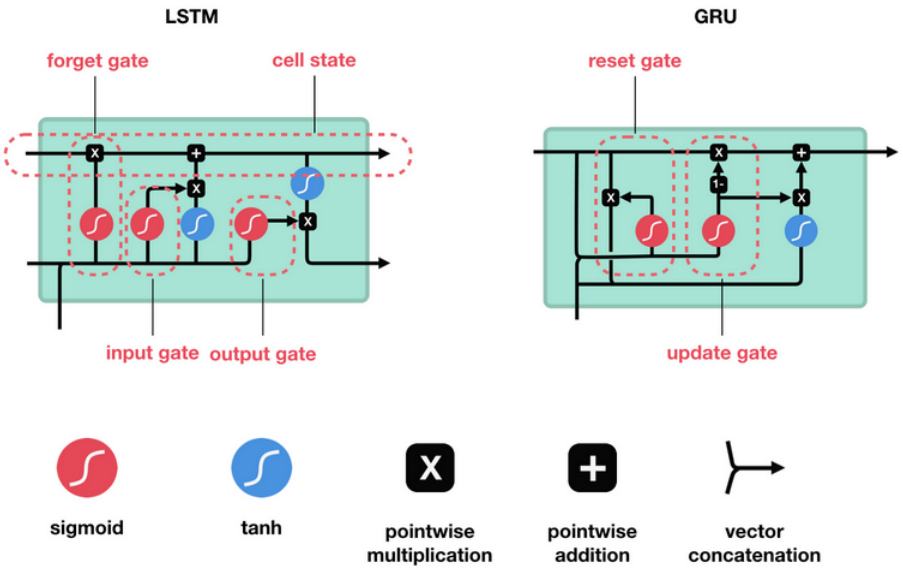


Gradient Update Rule

So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory. If you want to know more about the mechanics of recurrent neural networks in general, you can read my previous post here.

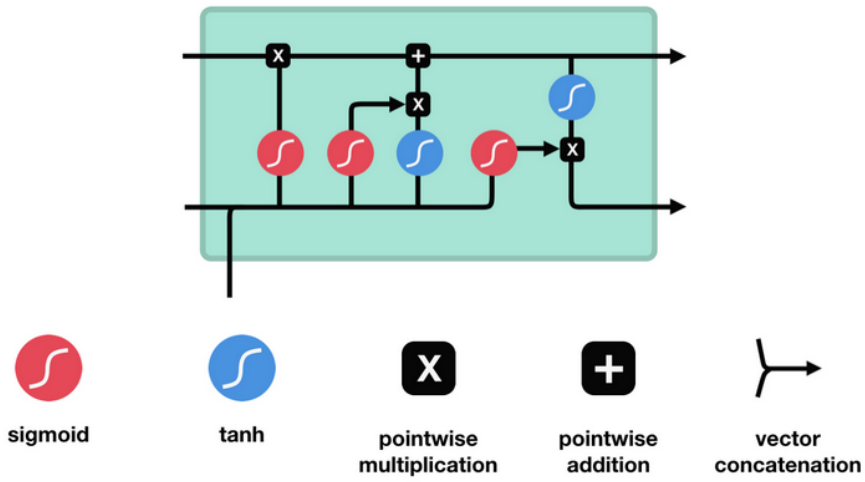
LSTM's and GRU's as a solution:

LSTM's and GRU's were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.



These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state of the art results based on recurrent neural networks are achieved with these two networks. LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

LSTM:



LSTM Cell and It's Operations

These operations are used to allow the LSTM to keep or forget information. Now looking at these operations can get a little overwhelming so we'll go over this step by step.

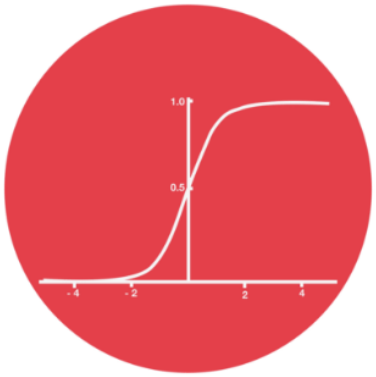
Core Concept

The core concept of LSTM's are the cell state, and it's various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make it's way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information gets added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.

Sigmoid

Gates contains sigmoid activations. A sigmoid activation is similar to the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1. That is helpful to update or forget data because any number getting multiplied by 0 is 0,

causing values to disappears or be "forgotten." Any number multiplied by 1 is the same value therefore that value stay's the same or is "kept." The network can learn which data is not important therefore can be forgotten or which data is important to keep.



Sigmoid squishes values to be between 0 and 1

Let's dig a little deeper into what the various gates are doing, shall we? So we have three different gates that regulate information flow in an LSTM cell. A forget gate, input gate, and output gate.

Forget gate

First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

Input Gate

To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the

sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

Cell State

Now we should have enough information to calculate the cell state. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.

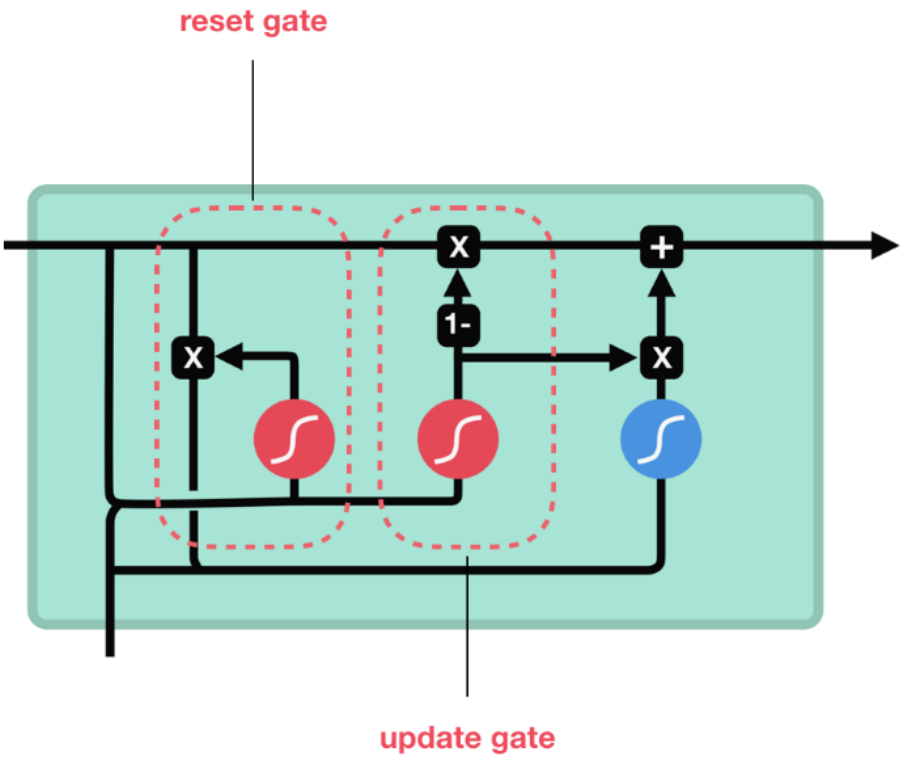
Output Gate

Last we have the output gate. The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.

To review, the Forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

GRU

So now we know how an LSTM work, let's briefly look at the GRU. The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.



GRU cell and it's gates

Update Gate

The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.

Reset Gate

The reset gate is another gate is used to decide how much past information to forget.

Fourier Transformation

Introduction

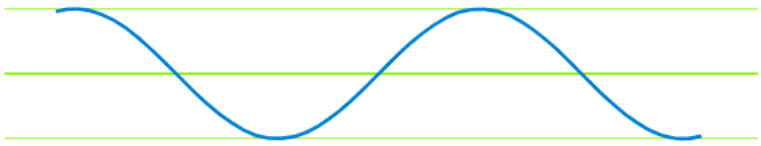
The Fourier Transform is one of the deepest insights ever made in mathematics but unfortunately, the meaning is buried deep inside some ridiculous equations.

The Fourier transform is a way of splitting something up into a bunch of sine waves. As usual, the name comes from some person who lived a long time ago called Fourier.

In mathematical terms, The Fourier Transform is a technique that transforms a signal into its constituent components and frequencies.

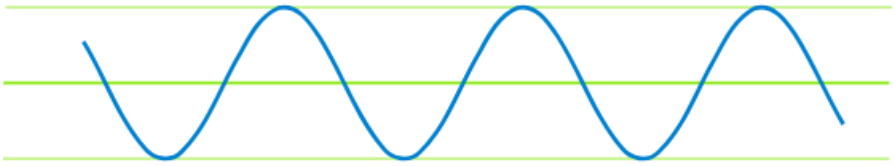
Fourier transform is widely used not only in signal (radio, acoustic, etc.) processing but also in image analysis eg. Edge detection, image filtering, image reconstruction, and image compression. One example: Fourier transform of transmission electron microscopy images helps to check the periodicity of the samples. Periodicity — means pattern. Fourier transform of your data can expand accessible information about the analyzed sample.

To understand it better consider a signal x(t):



If we do the same for another signal and select the same moment in time and we measure its amplitude.

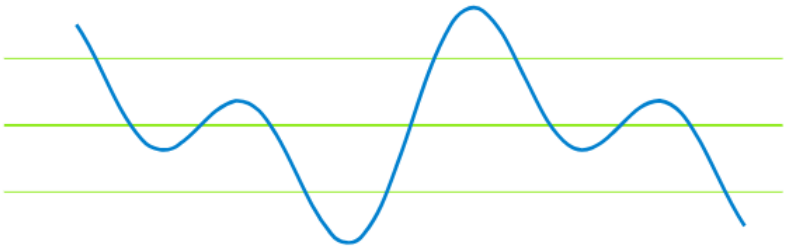
Consider another signal y(t):



What happens when we emit these two signals at the same time or if we add them together?

When we emit these two signals at the same moment of time, we get a new signal which is the *sum of the amplitude of these two signals*. This is so because these two signals are being added together.

Sum both the signals: z(t) = x(t) + y(t)

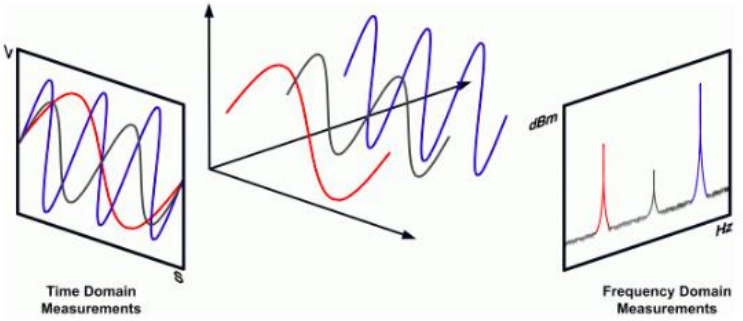


If we are given only one signal (which is the sum of signals x(t) and y(t)).Can we recover the original signals x(t) and y(t)?

Yes. That's what a Fourier transform does. It takes up a signal and decomposes it to the frequencies that made it up.

In our example, a Fourier transform would decompose the signal z(t) into its constituent frequencies like signals x(t) and y(t).

What Fourier transform does is it kind of moves us from the time domain to the frequency domain.



In case, if anyone is wondering, What if we want to go back from the frequency domain to the time domain?

We can do so by using the Inverse Fourier transform (IFT).

Maths you need to know.

"Any continuous signal in the time domain can be represented uniquely and unambiguously by an infinite series of sinusoids."

What does this mean?

It means that, If we have a signal which is generated by some function x(t) then we can come up with another function f(t) such that :

$$f(t) = \frac{a0}{2} \dot{+} \sum_{k=1}^{\infty} (a_k cos(2\pi kt) + b_k sin(2\pi kt))$$

So, It doesn't matter how strong the signal is, we can find a function like f(t) which is a sum of an infinite series of sinusoids that will actually represent the signal perfectly.

Now, the question that arises now is, How do we find the coefficients here in the above equation because these are the values that would determine the shape of the output and thus the signal.

$$a_k \ and \ b_k \ ?$$

So, to get these coefficients we use Fourier transforms and the result from Fourier transform is a group of coefficients. So, we use X(w) to denote the Fourier coefficients and it is a function of frequency which we get by solving the integral such that :

The Fourier transform is represented as an indefinite integral:

X(w) : Fourier Transform

x(t) : Inverse Fourier transform

$$X(\omega) = \int_{-\infty}^{+\infty} x(t) e^{-j\omega t} dt \qquad x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) e^{j\omega t} d\omega$$

Fourier Transform and Inverse Fourier transform

Also, when we actually solve the above integral, we get these complex numbers where a and b correspond to the coefficients that we are after.

The continuous Fourier transform converts a time-domain signal of infinite duration into a continuous spectrum composed of an infinite number of sinusoids. In practice, we deal with signals that are discretely sampled, usually at constant intervals, and of finite duration or periodic. For this purpose, the classical Fourier transform algorithm can be expressed as a Discrete Fourier transform (DFT), which converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform:

$$X_k = \sum_{n=0}^{N-1} x_n . e^{-2i\pi kn/N}$$

So, this is essentially the Discrete Fourier Transform. We can do this computation and it will produce a complex number in the form of a + ib where we have two coefficients for the Fourier series.

Now, we know how to sample signals and how to apply a Discrete Fourier Transform. The last thing we would like to do is, we would like to get rid of the complex number I because it's not supported in 'mllib' or 'systemML' by using something known as Euler's formula which states:

$$e^{i\theta} = \cos\theta + i\sin\theta$$

So, if we plug Euler's formula in the Fourier Transform equation and solve it, it will produce a real and imaginary part.

$$X_k = \sum_{n=0}^{N-1} x_n . [cos\,(\,2\pi kn/N) - isin(2\pi kn/N))]$$

As you can see X consist of a complex number of the format a+ib or a-ib. So if you solve the above equation you will get the Fourier coefficients a and b.

$$a_k = \sum_{n=0}^{N-1} x_n . \, cos\,(\,2\pi kn/N)$$

$$b_k = -\sum_{n=0}^{N-1} x_n . \, sin\,(\,2\pi kn/N)$$

Now if you just put the values of a and b in the equation of f(t) then you can define a signal in terms of its frequency.

In general practice, we use Fast Fourier Transformation (FFT) algorithm which recursively divides the DFT in smaller DFT's bringing down the needed computation time drastically. The time complexity of DFT is $2N^2$ while that of FFT is2NlogN.

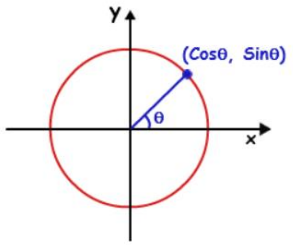
Why are cosine and sine functions used when representing a signal?

While Sine and Cosine functions were originally defined based on right-angle triangles, looking at that point of view in the current scenario isn't really the best thing. You might have been taught to recognize the Sine function as "opposite by hypotenuse", but now it's time to have a slightly different point of view.

Consider the unit circle:

$$x^2 + y^2 = 1$$

On a Cartesian plane. Suppose a line passing through the origin makes an angle θ with the x-axis in a counterclockwise direction, the point of intersection of the line and the circle is (cosθ, sinθ).



Think about it. Does this point of view correlate with the earlier one? Both of the definitions are the same.

The Sine and Cosine functions are arguably the most important periodic functions in several cases:

- The periodic functions of how displacement, velocity, and acceleration change with time in SHM oscillators are sinusoidal functions.
- Every particle has a wave nature and vice versa. This is de Broglie's Wave-Particle duality. Waves are always sinusoidal functions of some physical quantity (such as Electric Field for EM Waves, and Pressure for Sound Waves).

The sound itself is a pressure disturbance that propagates through material media capable of compressing and expanding. It's the pressure at a point along with the sound wave that varies sinusoidal with time.

Fourier Transformation in AI

Fourier Transformation is a linear function, to induce non-linearity. Convolutions are used.

''Fourier Transformation of the product of 2 signals is the convolution of the 2 signals.''

Let x(t) and y(t) be two functions with convolution X(t)*Y(t), and F represents Fourier transformation, then

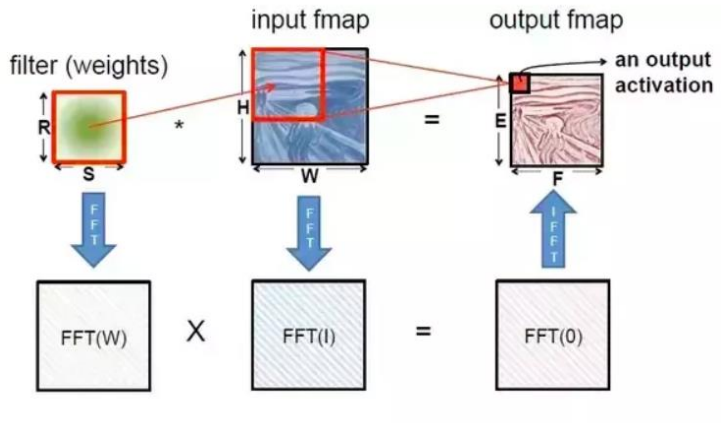
$$\mathbf{F\{x(t).y(t)\} = X(t)*Y(t)}$$

Remember the fact that a convolution in the time domain is a multiplication in the frequency domain. This is how Fourier Transform is mostly used in machine learning and more specifically deep learning algorithms.

I'll take Convolutional Neural Networks, CNNs as an example;

90% of computations in CNNs are convolutions and there have been many approaches to reduce the intensity of such computations, one of them is Fast Fourier Transform (FFT).

Instead of convolutions, the input and filter matrices are converted into the frequency domain by FFT, to do multiplications. Then, the output is converted back into the time domain by Inverse FFT (IFFT).



Another use of FFT is that it can be used for dimensionality reduction or feature extraction.

When each sample in the dataset is a signal (time series, or images, etc.), it may consist of thousands of samples. But they might actually correspond to just a few points in the Fourier domain (especially if there is some periodicity). This simplifies the problem a lot.

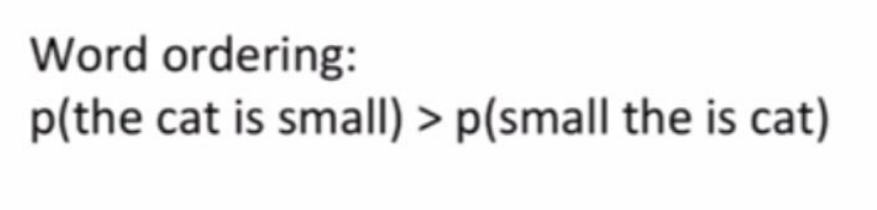
Or sometimes using the Fourier domain might provide translation-invariance. That is, even if there are lags between the signals, such variances will not affect their presentation in the Fourier domain.

Language Model

A language model learns to predict the probability of a sequence of words. But why do we need to learn the probability of words? Let's understand that with an example.

I'm sure you have used Google Translate at some point. We all use it to translate one language to another for varying reasons. This is an example of a popular NLP application called Machine Translation.

In Machine Translation, you take in a bunch of words from a language and convert these words into another language. Now, there can be many potential translations that a system might give you and you will want to compute the probability of each of these translations to understand which one is the most accurate.



In the above example, we know that the probability of the first sentence will be more than the second, right? That's how we arrive at the right translation.

This ability to model the rules of a language as a probability gives great power for NLP related tasks. Language models are used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval, and many other daily tasks.

Types of Language Models

There are primarily two types of Language Models:

- Statistical Language Models: These models use traditional statistical techniques like N-grams, Hidden Markov Models (HMM) and certain linguistic rules to learn the probability distribution of words
- Neural Language Models: These are new players in the NLP town and have surpassed the statistical language models in

their effectiveness. They use different kinds of Neural Networks to model language

What are N-grams (unigram, bigram, trigrams)?

> *An N-gram is a sequence of N tokens (or words).*

Let's understand N-gram with an example. Consider the following sentence:

"I love reading blogs about data science on Analytics Vidhya."

A 1-gram (or unigram) is a one-word sequence. For the above sentence, the unigrams would simply be: "I", "love", "reading", "blogs", "about", "data", "science", "on", "Analytics", "Vidhya".

A 2-gram (or bigram) is a two-word sequence of words, like "I love", "love reading", or "Analytics Vidhya". And a 3-gram (or trigram) is a three-word sequence of words like "I love reading", "about data science" or "on Analytics Vidhya".

Fairly straightforward stuff!

How do N-gram Language Models work?

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. If we have a good N-gram model, we can predict p(w | h) – what is the probability of seeing the word w given a history of previous words h – where the history contains n-1 words.

We must estimate this probability to construct an N-gram model.

We compute this probability in two steps:

- Apply the chain rule of probability.
- We then apply a very strong simplification assumption to allow us to compute p(w1…ws) in an easy manner

The chain rule of probability is:

```
p(w1...ws) = p(w1) . p(w2 | w1) . p(w3 | w1 w2) . p(w4 | w1 w2 w3) ..... p(wn | w1...wn-1)
```

So what is the chain rule? It tells us how to compute the joint probability of a sequence by using the conditional probability of a word given previous words.

But we do not have access to these conditional probabilities with complex conditions of up to n-1 words. So how do we proceed?

This is where we introduce a simplification assumption. We can assume for all conditions, that:

```
p(wk | w1...wk-1) = p(wk | wk-1)
```

Here, we approximate the history (the context) of the word wk by looking only at the last word of the context. This assumption is called the Markov assumption. (We used it here with a simplified context of length 1 – which corresponds to a bigram model – we could use larger fixed-sized histories in general).

Limitations of N-gram approach to Language Modeling

N-gram based language models do have a few drawbacks:

- The higher the N, the better is the model usually. But this leads to lots of computation overhead that requires large computation power in terms of RAM.
- N-grams are a sparse representation of language. This is because we build the model based on the probability of words co-occurring. It will give zero probability to all the words that are not present in the training corpus.

Building a Neural Language Model

Deep Learning has been shown to perform really well on many NLP tasks like Text Summarization, Machine Translation, etc. and since these tasks are essentially built upon Language Modeling,

there has been a tremendous research effort with great results to use Neural Networks for Language Modeling.

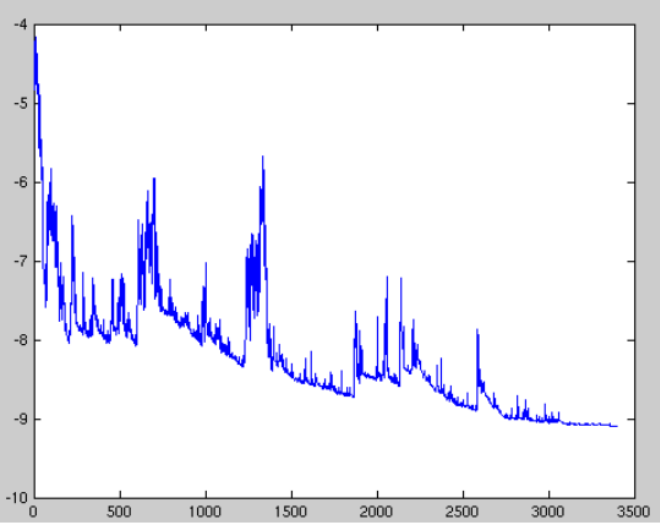
We can essentially build two kinds of language models – character level and word level. And even under each category, we can have many subcategories based on the simple fact of how we are framing the learning problem. We will be taking the most straightforward approach – building a character-level language model.

Optimizers

With the rapid development of deep learning has come a plethora of optimizers one can choose to compile their neural networks. With so many optimizers, it's difficult to choose one to use. This article will briefly explain how various neural network optimizers differ from each other

SGD (Stochastic Gradient Descent)

Stochastic Gradient Descent, in contrast to batch gradient descent or vanilla gradient descent, updates the parameters for each training example x and y. SGD performs frequent updates with a high variance, causing the objective function to fluctuate heavily.



SGD's fluctuation enables it to jump from a local minima to a potentially better local minima, but complicates convergence to an exact minimum.

Momentum is a parameter of SGD that can be added to assist SGD in ravines — areas where the surface curves more steeply in one dimension than in another, common around optima. In these scenarios, SGD oscillates around the slopes of the ravine, making hesitant progress along the bottom of the local optimum.

Image 2: SGD without momentum

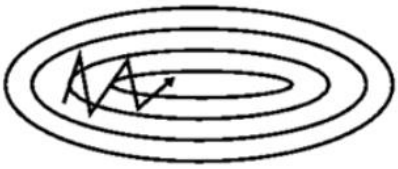


Image 3: SGD with momentum

Momentum helps accelerate SGD in the correct direction, therefore dampening the redundant oscillations

Adagrad

Adagrad adapts the learning rate to the parameters, performing smaller updates (low learning rates) for parameters associated with frequently occurring features and large updates (high learning rates) for parameters associated with infrequent features. Therefore, Adagrad is helpful in dealing with sparse data.

Adagrad eliminates the need to manually tune the learning rate — most implementations leave the default value at 0.01. However, Adagrad's algorithm causes the learning rate to shrink with every iteration and eventually become infinitesimally small, at which the algorithm cannot acquire any new knowledge.

Adadelta is an extension of Adagrad that seeks to solve the model's convergence of learning rate to 0. RMSprop is another version of Adadelta and seeks to solve the same problems Adadelta tried to.

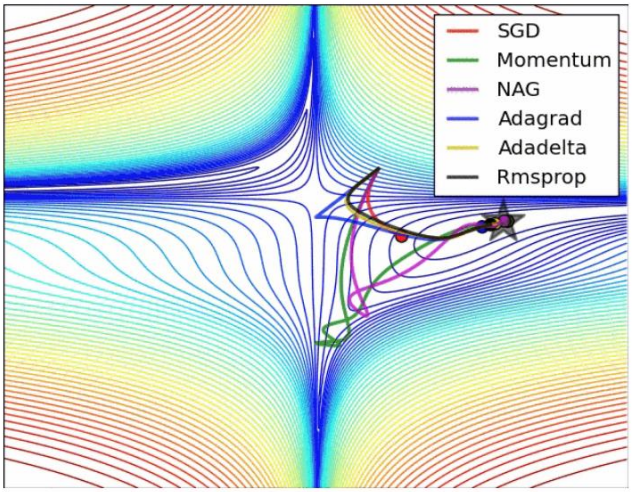
Adam

Adam is another method that computers adaptive learning rates for each parameter. In addition to storing previous gradients like Adadelta and RMSprop, Adam also implements a version of momentum. Adam behaves like a heavy ball with friction,

preferring flat minima in the error surface, and can be viewed as a combination of RMSprop and SGD with momentum.

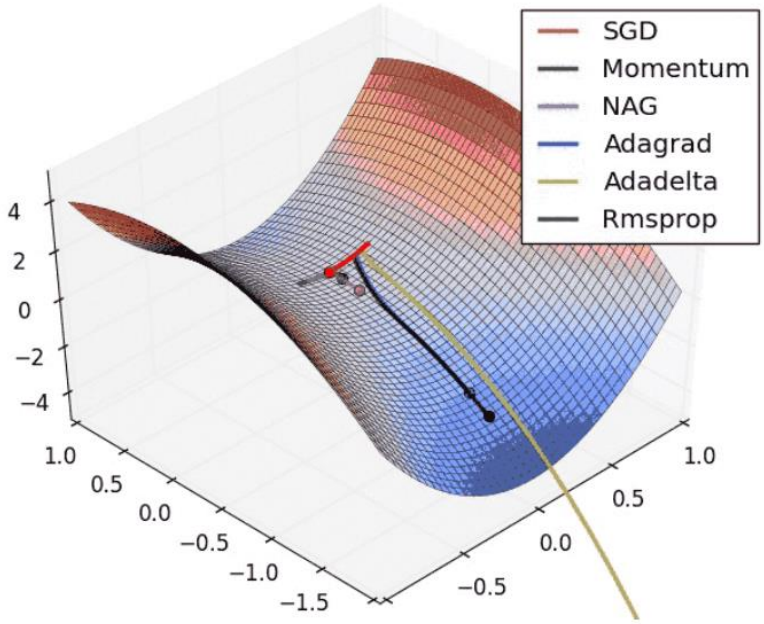
Nadam is another optimizer that is a combination of Adam and NAG.

Visualizations



Note that Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge very quickly, while SGD with momentum and NAG are led off-track, evoking the image of a ball rolling down the hill. However, NAG is quickly able to correct its course by looking ahead.

Saddlepoint



Notice that SGD (with and without momentum) and NAG find it difficult to break towards the minima and are stuck in the middle. However, SGD with momentum and NAG eventually escape the saddle point. Adagrad, RMSprop, and Adadelta quickly head down the negative slope.

Which optimizer should I use?

If the input data is sparse, the best results will come from an adaptive-learning rate method. Overall Adam may be the best overall choice for deep neural networks

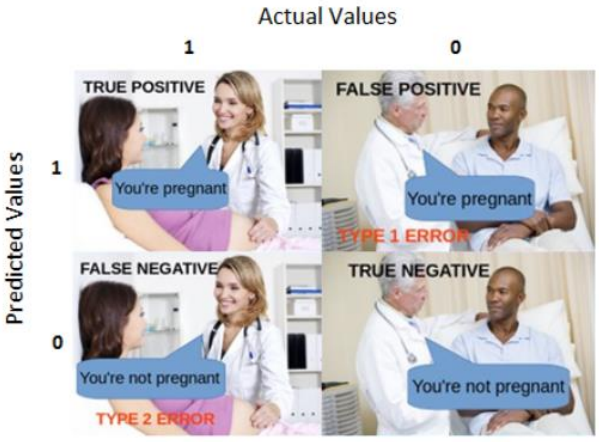
Confusion Matrix

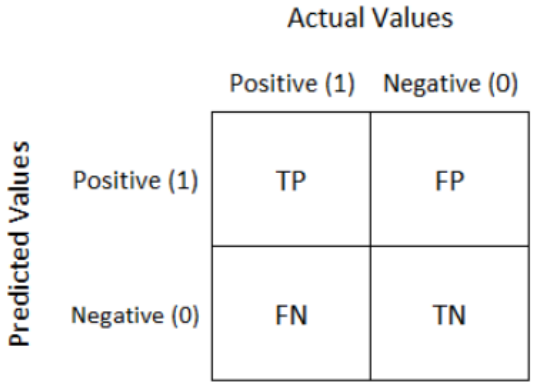
What is Confusion Matrix and why you need it?

Well, it is a performance measurement for machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values.

It is extremely useful for measuring Recall, Precision, Specificity, Accuracy and most importantly AUC-ROC Curve.

Let's understand TP, FP, FN, and TN in terms of pregnancy analogy.





<u>True Positive:</u>

Interpretation: You predicted positive and it's true.

You predicted that a woman is pregnant and she actually is.

<u>True Negative:</u>

Interpretation: You predicted negative and it's true.

You predicted that a man is not pregnant and he actually is not.

<u>False Positive:</u> (Type 1 Error)

Interpretation: You predicted positive and it's false.

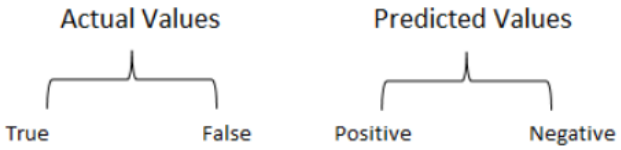
You predicted that a man is pregnant but he actually is not.

<u>False Negative:</u> (Type 2 Error)

Interpretation: You predicted negative and it's false.

You predicted that a woman is not pregnant but she actually is.

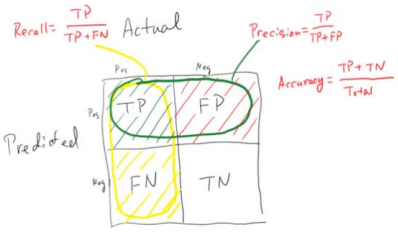
Just Remember, We describe predicted values as Positive and Negative and actual values as True and False.



How to Calculate Confusion Matrix for a 2-class classification problem?

Let's understand confusion matrix through math.

y	y pred	output for threshold 0.6	Recall	Precision	Accuracy
0	0.5	0			
1	0.9	1			
0	0.7	1			
1	0.7	1	1/2	2/3	4/7
1	0.3	0			
0	0.4	0			
1	0.5	0			



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

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

Recall is: Out of all the positive classes, how much we predicted correctly. It should be high as possible.

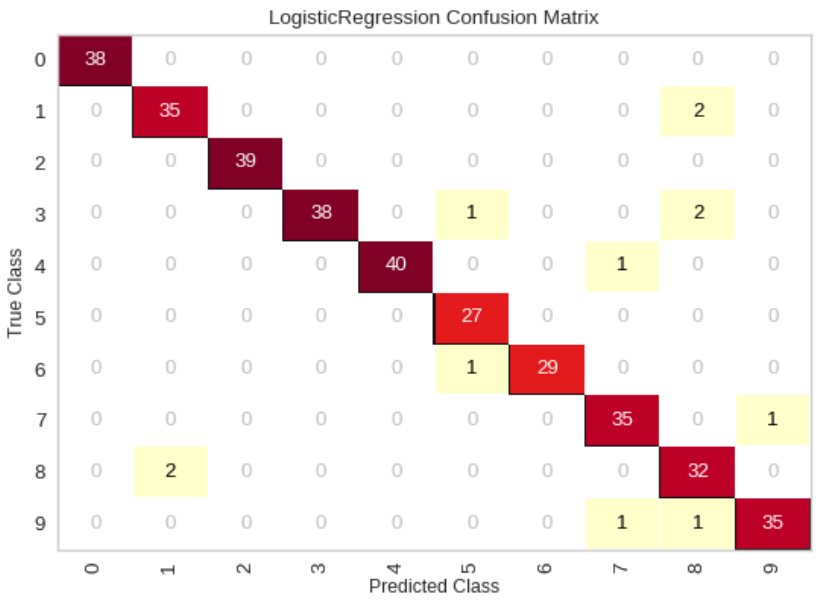
Precision is: Out of all the positive classes we have predicted correctly, how many are actually positive.

And the accuracy is: Out of all the classes, how much we predicted correctly, which will be, in this case 4/7. It should be high as possible.

$$\textbf{F - measure} = \frac{2*Recall*Precision}{Recall + Precision}$$

It is difficult to compare two models with low precision and high recall or vice versa. So to make them comparable, we use F-Score. F-score helps to measure Recall and Precision at the same time. It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more.

And here is an example of the confusion matrix for n-classes:



References

Eval Metrics

How to Add Regularization to Keras Pre-trained Models the Right Way

Fine-Grained Head Pose Estimation Without Keypoints, Nataniel Ruiz, Eunji Chong, James M. Rehg, V5 13 Apr 2018.

Deep Speech: Scaling up end-to-end speech recognition, Awni Hannun, Carl Case, Jared Casper, Andrew Y. Ng, V2 19 Dec 2014

Deep Residual Learning for Image Recognition, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, 10 Dec 2015

Driver Distraction European Commission 2015

https://www.rospa.com/Road-Safety/Advice/Drivers/Distraction

https://www.sciencedirect.com/book/9780123819840/handbook-of-traffic-psychology

Driver Distraction A Sociotechnical Systems Approach, Katie J. Parnell, Neville A. Stanton and Katherine L. Plant

https://www.nhtsa.gov/risky-driving/distracted-driving

100 DISTRACTED DRIVING FACTS & STATISTICS FOR 2018

Facts + Statistics: Distracted driving

24 Distracted Driving Statistics & Facts – 2019

Technology That Can Reduce Driving Distractions and Their Dangers

Valeo AUC Driver Distraction Dataset

Tuning Language Model

Keras Text Classification

EDA Paper