UNIT - V

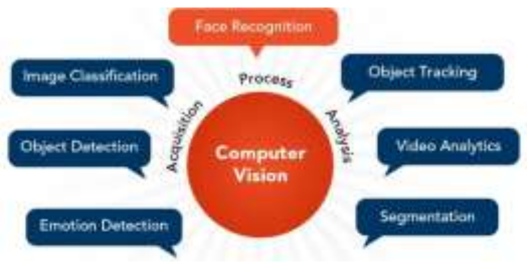
1) Give brief description of Deep Learning application?

Deep learning is a branch of artificial intelligence (AI) that teaches neural networks to learn and reason. Its capacity to resolve complicated issues and deliver cutting-edge performance in various sectors has attracted significant interest and appeal in recent years. Deep learning algorithms have revolutionized AI by allowing machines to process and comprehend enormous volumes of data. The structure and operation of the human brain inspired these algorithms.

Common Applications of Deep Learning in Artificial Intelligence: Deep learning has many uses in many fields, and its potential grows. Let's analyze a few of artificial intelligence's widespread profound learning uses.

- Image Recognition and Computer Vision
- Natural Language Processing (NLP)
- Speech Recognition and Voice Assistants
- Recommendation Systems
- Autonomous Vehicles
- Healthcare and Medical Imaging
- Fraud Detection and Cybersecurity
- Gaming and Virtual Reality

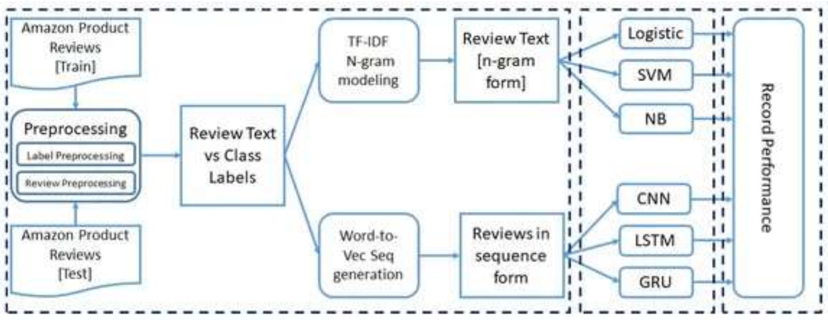
Image Recognition and Computer Vision



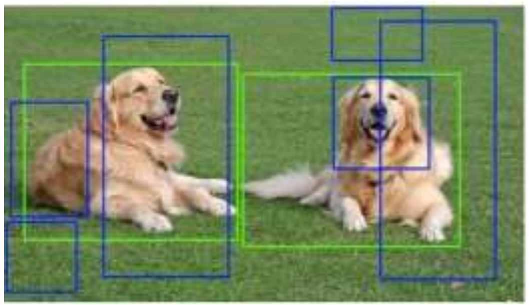
The performance of image recognition and computer vision tasks has significantly improved due to deep learning. Computers can now reliably classify and comprehend images owing to training deep neural networks on enormous datasets, opening up a wide range of applications.

A smartphone app that can instantaneously determine a dog's breed from a photo and self-driving cars that employ computer vision algorithms to detect pedestrians, traffic signs, and other roadblocks for safe navigation are two examples of this in practice.

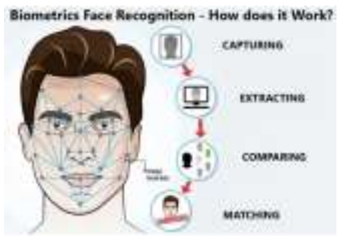
Deep Learning Models for Image Classification: The process of classifying photos entails giving them labels based on the content of the images. **Convolutional neural networks (CNNs**), one type of deep learning model, have performed exceptionally well in this context. They can categorize objects, situations, or even specific properties within an image by learning to recognize patterns and features in visual representations.



Object Detection and Localization using Deep Learning: Object detection and localization go beyond image categorization by identifying and locating various things inside an image. Deep learning methods have recognized and localized objects in real-time, such as You Only Look Once (YOLO) and region-based convolutional neural networks (R-CNNs). This has uses in robotics, autonomous cars, and surveillance systems, among other areas.



Applications in Facial Recognition and Biometrics: Deep learning has completely changed the field of facial recognition. Hence, allowing for the precise identification of people using their facial features. Security systems, access control, monitoring, and law enforcement use facial recognition technology. Deep learning methods have also been applied in biometrics for functions including voice recognition, iris scanning, and fingerprint recognition.



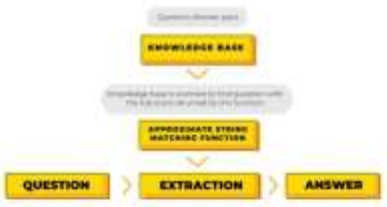
Natural Language Processing (NLP)

Natural language processing (NLP) aims to make it possible for computers to comprehend, translate, and create human language. NLP has substantially advanced primarily to deep learning, making strides in several language-related activities. Virtual voice assistants like Apple's Siri and Amazon's Alexa, who can comprehend spoken orders and questions, are a practical illustration of this.

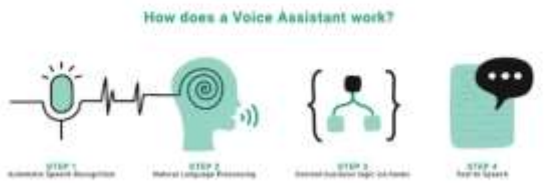
Deep Learning for Text Classification and Sentiment Analysis: Text classification entails classifying text materials into several groups or divisions. Deep learning models like **recurrent neural networks (RNNs)** and **long short-term memory (LSTM)** networks have been frequently used for text categorization tasks. To ascertain the sentiment or opinion expressed in a text, whether good, negative, or neutral, sentiment analysis is a widespread use of text categorization.

Language Translation and Generation with Deep Learning: Machine translation systems have considerably improved because of deep learning. Deep learning-based neural machine translation (NMT) models have been shown to perform better when converting text across multiple languages. These algorithms can gather contextual data and generate more precise and fluid translations. Deep learning models have also been applied to creating news stories, poetry, and other types of text, including coherent paragraphs.

Question Answering and Chatbot Systems Using Deep Learning: Deep learning is used by **chatbots** and **question-answering programs** to recognize and reply to human inquiries. Transformers and attention mechanisms, among other deep learning models, have made tremendous progress in understanding the context and semantics of questions and producing pertinent answers. Information retrieval systems, virtual assistants, and customer service all use this technology.



Speech Recognition and Voice Assistants



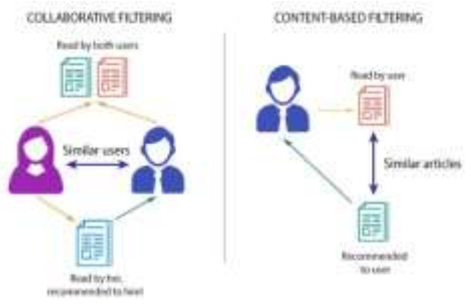
The creation of voice assistants that can comprehend and respond to human speech and the advancement of **speech recognition** systems have significantly benefited from deep learning. A real-world example is using your smartphone's voice recognition feature to dictate messages rather than typing them and asking a smart speaker to play your favorite tunes or provide the weather forecast.

Deep Learning Models for Automatic Speech Recognition: Systems for automatic speech recognition (ASR) translate spoken words into written text. Recurrent neural networks and attention-based models, in particular, have substantially improved ASR accuracy. Better voice commands, transcription services, and accessibility tools for those with speech difficulties are the outcome. Some examples are voice search features in search engines like Google, Bing, etc.

Voice Assistants Powered by Deep Learning Algorithms: Daily, we rely heavily on voice assistants like Siri, Google Assistant, and Amazon Alexa. Guess what drives them? Deep learning it is. Deep learning techniques are used by these intelligent devices to recognize and carry out spoken requests. The technology also enables voice assistants to recognize speech, decipher user intent, and deliver precise and pertinent responses thanks to deep learning models.

Applications in Transcription and Voice-Controlled Systems: Deep learning-based speech recognition has applications in transcription services, where large volumes of audio content must be accurately converted into text. Voice-controlled systems, such as smart homes and in-car infotainment systems, utilize deep learning algorithms to enable hands-free control and interaction through voice commands.

Recommendation Systems



Recommendation systems use deep learning algorithms to offer people personalized recommendations based on their tastes and behavior.

Deep Learning-Based Collaborative Filtering: A standard method used in recommendation systems to suggest products/services to users based on how they are similar to other users is collaborative filtering. Collaborative filtering has improved accuracy and performance thanks to deep learning models like matrix **factorization** and **deep autoencoders**, which have produced more precise and individualized recommendations.

Deep neural networks have been used to identify intricate links and patterns in user behavior data, allowing for more precise and individualized suggestions. Deep learning algorithms can forecast user preferences and make relevant product, movie, or content recommendations by looking at user interactions, purchase history, and demographic data. An instance of this is when streaming services recommend films or TV shows based on your interests and history.

Applications in E-Commerce and Content Streaming Platforms
Deep learning algorithms are widely employed to fuel recommendation systems in e-commerce platforms and video streaming services like Netflix and Spotify. These programs increase user pleasure and engagement by assisting users in finding new goods, entertainment, or music that suits their tastes and preferences.

Autonomous Vehicles

Deep learning has significantly impacted how well autonomous vehicles can understand and navigate their surroundings. These vehicles can analyze enormous volumes of sensor data in real-time using powerful deep learning algorithms. Thus, enabling them to make wise decisions, navigate challenging routes, and guarantee the safety of passengers and pedestrians. This game-changing technology has prepared the path for a time when driverless vehicles will completely change how we travel.

Deep Learning Algorithms for Object Detection and Tracking
Autonomous vehicles must perform crucial tasks, including object identification and tracking, to recognize and monitor objects like pedestrians, cars, and traffic signals. Convolutional and recurrent neural networks (CNNs) and other deep learning algorithms have proved essential in obtaining high accuracy and real-time performance in object detection and tracking.

Deep Reinforcement Learning for Decision-Making in Self-Driving Cars
Autonomous vehicles are designed to make complex decisions and navigate various traffic circumstances using deep reinforcement learning. This technology is profoundly used in self-driving cars manufactured by companies like Tesla. These vehicles can learn from historical driving data and adjust to changing road conditions using deep neural networks. Self-driving cars demonstrate this in practice, which uses cutting-edge sensors and artificial intelligence algorithms to navigate traffic, identify impediments, and make judgments in real time.

Applications in Autonomous Navigation and Safety Systems
The development of autonomous navigation systems that decipher sensor data, map routes, and make judgments in real time depends heavily on deep learning techniques. These systems focus on collision avoidance, generate lane departure warnings, and offer adaptive cruise control to enhance the general safety and dependability of the vehicles.

Healthcare and Medical Imaging

Deep learning has shown tremendous potential in revolutionizing healthcare and medical imaging by assisting in diagnosis, disease detection, and patient care. Revolutionizing diagnostics using AI-powered algorithms that can precisely identify early-stage tumors from medical imaging is an example of how to do this. This will help with prompt treatment decisions and improve patient outcomes.

Fraud Detection and Cybersecurity

Deep learning has become essential in detecting anomalies, identifying fraud patterns, and strengthening cybersecurity systems. In fraud prevention systems, deep neural networks have been used to recognize and stop fraudulent transactions, **credit card fraud**, and identity theft. These algorithms examine user behavior, transaction data, and historical patterns to spot irregularities and notify security staff. This enables proactive fraud prevention and shields customers and organizations from financial loss. Organizations like Visa, Mastercard, and PayPal use deep neural networks. It helps improve their fraud detection systems and guarantees secure customer transactions.

Gaming and Virtual Reality

Deep learning algorithms have produced more intelligent and lifelike video game characters. Game makers may create realistic animations, enhance character behaviors, and make **more immersive gaming experiences** by training deep neural networks on enormous datasets of motion capture data.

Experiences in augmented reality (AR) and **virtual reality (VR)** have been improved mainly due to deep learning. Deep neural networks are used by VR and AR systems to correctly track and identify objects, detect movements and facial expressions, and build real virtual worlds, enhancing the immersiveness and interactivity of the user experience.

2) Explain the process of Hand written digits recognition using Deep Learning?

A) Handwritten digit recognition is the ability of a computerto recognize the human handwritten digits from different sources like images, papers, touch screens, etc, and classify them into 10 predefined classes (0-9). This has been a topic of boundless-research in the field of deep learning. Digit recognition has many applications like number platerecognition, postal mail sorting, bank check processing, etc.

In Handwritten digit recognition, we face many challenges because of different styles of writing of different peoples as it is not an Optical character recognition. This research provides a comprehensive comparison between different machinelearning and deep learning algorithms for the purpose of handwritten digit recognition. For this, we have used Support Vector Machine, Multilayer Perceptron, and Convolutional Neural Network. The comparison between these algorithmsis carried out on the basis of their accuracy, errors, and testing-training time corroborated by plots and charts that have been constructed using matplotlib for visualization.

The accuracy of any model is paramount as more accurate models make better decisions. The models with low accuracy are not suitable for real-world applications. **Ex-** For an automated bank cheque processing system where the system recognizes the amount and date on the check, high accuracyis very critical. If the system incorrectly recognizes a digit, it can lead to major damage which is not desirable. That's why an algorithm with high accuracy is required in these real-world applications. Hence, we are providing a comparisonof different algorithms based on their accuracy so that themost accurate algorithm with the least chances of errors canbe employed in various applications of handwritten digit recognition.

METHODOLOGY:

Importing the libraries: Libraries are useful tools that can make a web developer's job more efficient. It's a set of prewritten code, that we can call while programming our own code. Basically, it's the work that's already done by someone else that you can make use of, without having to do it yourself. You can also use it in your own code. Different libraries have different restrictions on fair use, but this is a code that was designed to be used by others, instead of just standing alone.

The libraries used in this code are -

a. ***Tensorflow*** - Tensorflow's framework is open source, it is used in machine learning and other computations on various data.

b. ***OpenCV*** – OpenCV (Open-Source Computer Vision Library) is an open-source computer vision and machine learning software libraries.

c. ***NumPy*** – NumPy stands for Numerical Python.

Loading datasets: For the training and testing data, we will be using a Dataset which is present in the Tensorflow API. This specific dataset is the **MNIST** data that contains around fifty thousand images for the training data and another ten thousandimages for testing data confined to a dimension of 28X28.

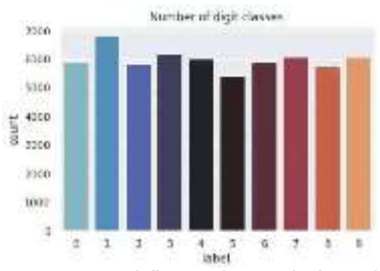


Figure 1. Bar graph illustrating the MNIST handwritten digit training dataset (Label vs Total number of training samples).

Figure 2. Plotting of some random MNIST Handwritten digits.

Creating a model: We are creating a model using a sequential neural network to recognize handwritten digits. We are using Three dense layers namely input hidden and output layers. Here we used 128, 128, 10 parameters for the previously mentioned layers respectively.

Training the Model: We are training the model using 3 dense layers, one using input layer, one using hidden layer and the last one using the output layer. The 3 dense layers take 128, 128, 10 parameters each. We flatten the pixels of the greyscale image in the input layer. Then we apply the activation functions to the weights in the hidden layer. The output layer gives the result as a prediction.

Testing the Model: Once the model is trained, we can use the loss function and accuracy function to test the model. We should have accuracy as high as possible and loss as less as possible to get the desired output (with accuracy being close to 1 and loss being close to 0). We can use "ADAM" as our optimizer, "cross-entropy" as our loss and "accuracy" as our metrics.

Getting the output: We take a set of inputs and upload them into our model. Then we use a while loop to analyze each inputindividually and give the predictions for each image. This process has been shown in a flowchart below in Fig.

Flowchart

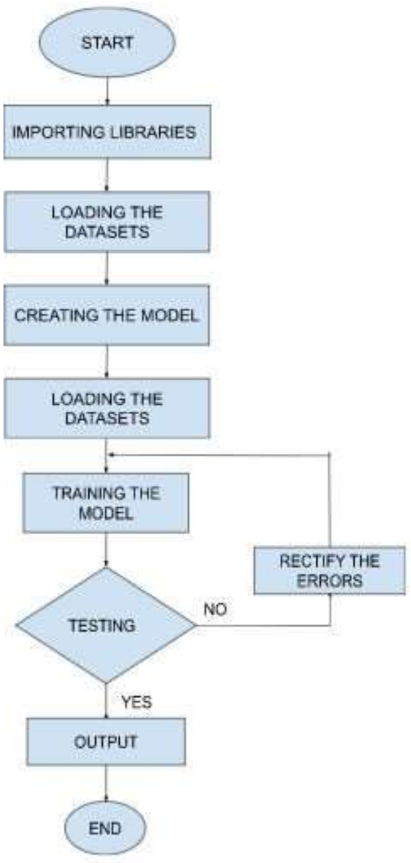


Fig. Methodology used in Implementation of Handwritten Recognition

IMPLEMENTATION: Neural network is implemented wherein the model recognizes and predicts a handwritten digit. Initially **Tensorflow and Keras** are used to form the bones of the implementation. We load the datasets from both of these open-source libraries and make our model to analyze thousands of images. The model learns all the patterns, pixel placements of the greyscale images and all the neural connections.

Keras is an API (Application Programming Interface), which is designed for machine learning and deep learning. It's an opensource library which has a lot of inbuilt data. It's the interface of Tensorflow library.

1. It follows the best practices available for reducing cognitive load.
2. It provides simple & consistent APIs (Application Programming Interface).
3. It has extensive documentation and has developer guides.
4. It minimizes the amount of user actions needed for common use cases, and it provides actionable & clear error messages.

TensorFlow framework is open source, it is used in machine learning and other computations on various data. It has the symbol of TF. It allows the developers to create the learning algorithms on their set of data and models, and to experiment with diverse algorithms. It allows the developers to create data flow structures, which shows how the data will move through a series of graphs or nodes. All the datasets which we used in this program are taken from Tensorflow.

Then we use the **ADAM** optimizer for training our model. It's the best optimizer to train our neural network in less time with high efficiency. An optimizer is used to update the network weights live during the training. Then we use the loss function and the metrics function to check the performance of our model. We use cross-entropy and accuracy respectively to check the performance. Once the model is trained a, we save it to our computer and comment on all the preprocessing and training code. So, whenever the code is required, we can just load the saved model since all the data is stored in it. Now the final step is to give the inputs to our model. For this we scan the digits we wrote on a sheet of paper or draw the digit in MS paint tool and download the image in PNG format. Once we upload all the images into our python script, then we use while loop to all each image and analyze it to predict the output.

Results And Discussion: From this implementation, we are able to identify the handwritten digits as input given to the code as shown in Fig.5, analyze it and predict the probability output as shown in Fig.6. With the code we are able to show that written data in MS paint application can be saved. Where the saved file is now loaded into the program and will run. With a while loop, we check each image and predict the value if multiple data set files are present. With this the image is analyzed by the already learnt neural connections and the grayscale pixels. It will now provide us with a prediction of the accurate output of recognized data. So, we can successfully recognize and digitalize our data. We are successfully able to understand and use Sequential Model, ReLU, SoftMax, adam, Cross-Entropy, Accuracy functions for image recognition. The snips of the code used in the program are shown in the Fig.7(a) and Fig.7(b).

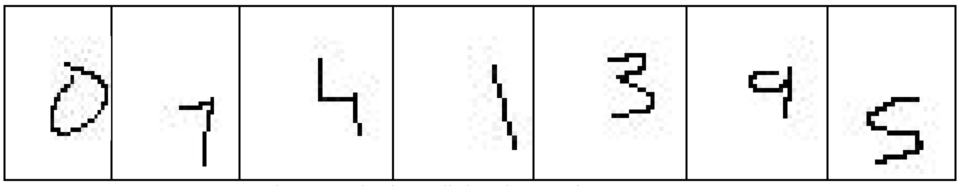
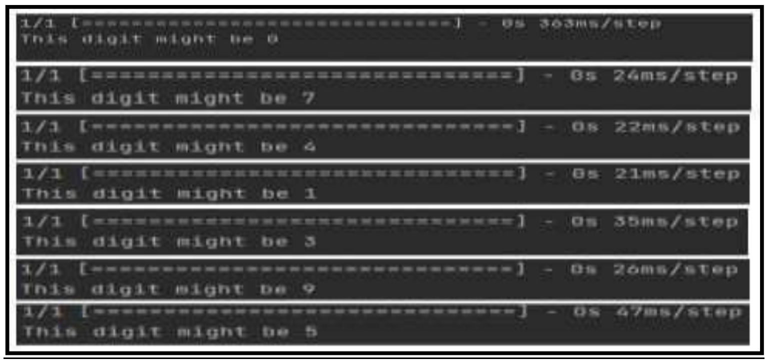
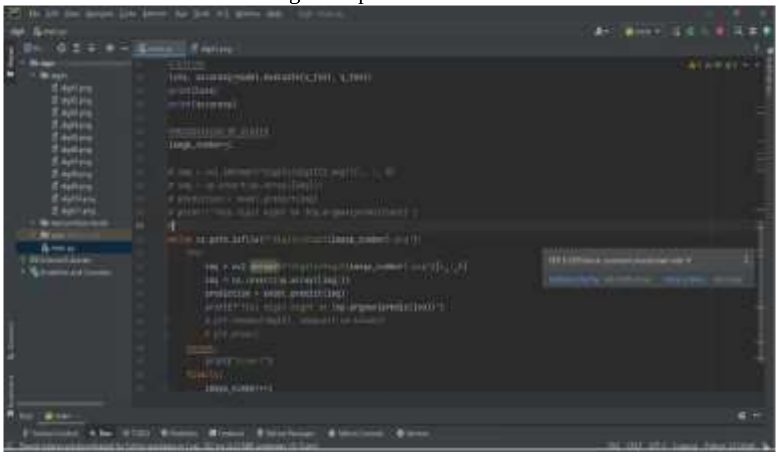


Fig.5 Handwritten digits given as input Images



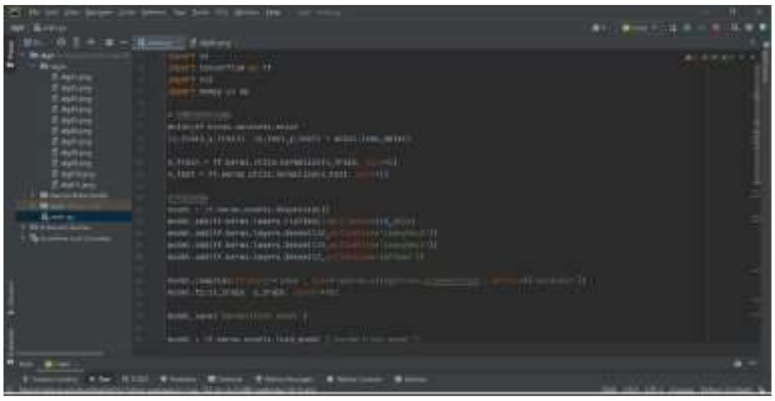
```
1/1 [==============================] - 0s 363ms/step
This digit might be 0
1/1 [==============================] - 0s 24ms/step
This digit might be 7
1/1 [==============================] - 0s 22ms/step
This digit might be 4
1/1 [==============================] - 0s 21ms/step
This digit might be 1
1/1 [==============================] - 0s 35ms/step
This digit might be 3
1/1 [==============================] - 0s 26ms/step
This digit might be 9
1/1 [==============================] - 0s 47ms/step
This digit might be 5
```

Fig.6 Outputs Obtained



```
[illegible]
```

Fig.7 (a) Simulation framework of code

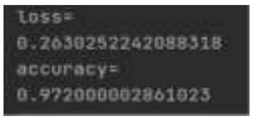


```
[illegible]
```

Fig.7(b) Simulation framework of code

Performance Analysis: We use the loss function and the metrics to get the loss and accuracy respectively. The loss function used for the model is **Cross- Entropy**. It tells us the amount of loss was obtained by training our model. The metrics used for our model is Accuracy. It tells ushow accurate our model is after training it.

The Accuracy should be close to 1 and the Loss should be close to 0. Then and only then, our model is close to perfection. If the loss is high compared to 0 and accuracy compared to 1, then the model is not perfect,and we need to train the model with different parameters and a greater number of epochs. The loss we got after training our model is around 3%The accuracy we got after the model is trained is around 97%, as shown in Fig.



```
loss=
0.2630252242088318
accuracy=
0.972000002861023
```

Fig. Performance Analysis- Loss and Accuracy of testing model

Conclusion: In this implementation, we created a neural network to recognize a handwritten digit using TensorFlow and Keras. using the **MNIST** dataset, our model can recognize handwritten digits which are being input to the model. The performance of CNN for the handwritten digit is accurate. The method works well, and the loss percentage is less with all those training sessions. The onlydifficulty here is the noise in the image, but with the training it has, it tries to achieve the best possible output.

The model's performance and accuracy were tested after training the model for 50 epochs. With the results given from this work we are more confident in finding other ways to make this better and to make it easier for complex data like converting handwritten paragraphsinto text. Through this research work we understood all the mechanisms used to identify handwritten data. We understand the importance of hand recognition as it is easy for the user to write data on paper and use handwritten data recognition to convert it into text instead of the typing it on keyboard. Further it is recommended to implement on edge computing platforms like Raspberry Pi 4 system for actual usage.

3) LSTM With Keras?

A) **LSTM stands for "Long Short-Term Memory"**. An LSTM is actually a kind of RNN architecture. It is, theoretically, a more "sophisticated" Recurrent Neural Network. Instead of just having recurrence, it also has "gates" that regulate information flow through the unit as shown in the image. LSTMs were initially introduced to solve the vanishing gradient problem of RNNs. They are often used over traditional, "simple" recurrent neural networks because they are also more computationally efficient.

Creating a Simple LSTM with Keras: Using Keras and Tensorflow makes building neural networks much easier to build. It's much easier to build neural networks with these libraries than from scratch. The best reason to build a neural network from scratch is to understand how neural networks work. In practical situations, using a library like Tensorflow is the best approach. It's straight forward and simple to build a neural network with Tensorflow and Keras, let's take a look at how to use Keras to build our LSTM.

Importing the Right Modules: The first thing we need to do is import the right modules. For this example, we're going to be working with tensorflow. We don't technically *need* to do the bottom two imports, but they save us time when writing so when we add layers we don't need to type **"tf.keras.layers"**. but can rather just write layers.

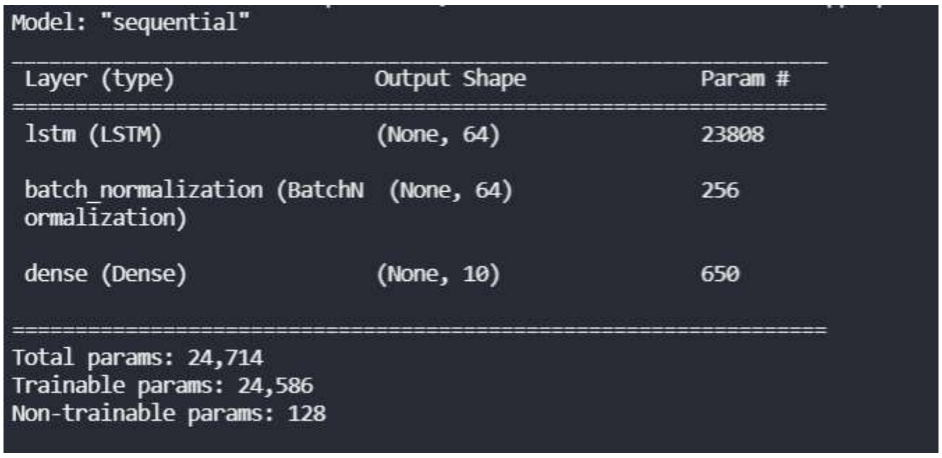
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Adding Layers to Your Keras LSTM Model: It's quite easy to build an LSTM in Keras. All that's really required for an LSTM neural network is that it has to have LSTM cells or at least one LSTM layer. If we add different types of layers and cells, we can still call our neural network an LSTM, but it would be more accurate to give it a mixed name.

To build an LSTM, the first thing we're going to do is initialize a Sequential model. Afterwards, we'll add an LSTM layer. This is what makes this an LSTM neural network. Then we'll add a batch normalization layer and a dense (fully connected) output layer. Next, we'll print it out to get an idea of what it looks like.

```
model = keras.Sequential()
model.add(layers.LSTM(64, input_shape=(None, 28)))
model.add(layers.BatchNormalization())
model.add(layers.Dense(10))
print(model.summary())
```

we'll see that the LSTM actually has WAY more parameters than the Simple RNN we built with Keras. The LSTM layer has four times the number of parameters as a simple RNN layer. This is because of the gates we talked about earlier.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 64)                23808

 batch_normalization (BatchN  (None, 64)               256
 ormalization)

 dense (Dense)               (None, 10)                650

=================================================================
Total params: 24,714
Trainable params: 24,586
Non-trainable params: 128
```

Keras LSTM parameters

Training and Testing our Keras LSTM on the MNIST Dataset: Now that we've built our LSTM let's see how it does on the MNIST digit dataset. This is the same dataset we tested the Keras RNN and the built from scratch Neural Network on. The MNIST dataset is a classic dataset to train and test neural networks on. It is a set of handwritten digits.

Load the MNIST dataset: The first thing we'll do is load up the MNIST dataset from Keras. We'll use the `load_data()` function from the MNIST dataset to load a pre-separated training and testing dataset. After loading the datasets, we'll normalize our training data by dividing by 255. This is due to the scale of 256 (0 to 255) for the image data. Finally, we'll set aside 10 test data points.

```
mnist = keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
x_validate, y_validate = x_test[:-10], y_test[:-10]
x_test, y_test = x_test[-10:], y_test[-10:]
```

Compile the LSTM Neural Network: Now that we've created our LSTM and loaded up our data, let's compile our model. We have to compile (or build) or model before we can train or test it. In our model compilation we will specify the loss function, in this case Sparse Categorical Cross Entropy, our optimizer, stochastic gradient descent, and our metric(s), accuracy. We can specify multiple metrics, but we'll just go with accuracy for this example.

```
model.compile(
loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
optimizer="sgd",
metrics=["accuracy"],)
```

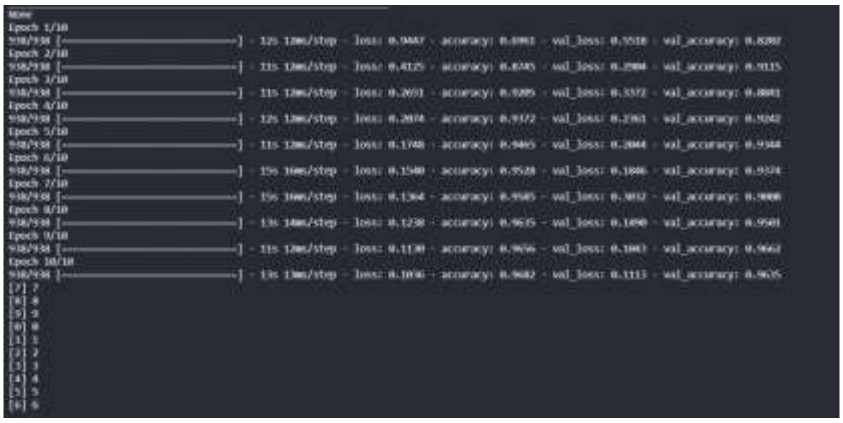
Train and Fit the Keras LSTM Model: Now that the model is compiled, let's train the model. To train the model in Keras, we just call the fit function. To use the fit function, we'll need to pass in the training data for x and y, the validation, the batch_size, and the epochs. For this example, we'll just train for 10 epochs.

```
model.fit (
x_train, y_train, validation_data=(x_test, y_test), batch_size=64, epochs=1
)
model.fit(    x_train, y_train, validation_data=(x_validate, y_validate), batch_size=64, epochs=10)
```

Test the Long Short Term Memory Keras Model: We've already trained and fit the model. The last thing to do is to test the model. We'll run our model and use it to predict the sample we set aside earlier. Then, we'll print out the sample and the correct label.

```
for i in range(10):
  result = tf.argmax(model.predict(tf.expand_dims(x_test[i], 0)), axis=1)
  print(result.numpy(), y_test[i])
```

We can see that after 1 epoch (you should really train for more, but once again this is just for an example) We can see that after 10 epochs we see a pretty good accuracy at about 96%.



```
None
Epoch 1/10
938/938 [==============================] - 12s 12ms/step - loss: 0.9447 - accuracy: 0.6961 - val_loss: 0.5518 - val_accuracy: 0.8282
Epoch 2/10
938/938 [==============================] - 11s 12ms/step - loss: 0.4125 - accuracy: 0.8745 - val_loss: 0.2904 - val_accuracy: 0.9115
Epoch 3/10
938/938 [==============================] - 11s 12ms/step - loss: 0.2693 - accuracy: 0.9205 - val_loss: 0.3372 - val_accuracy: 0.8881
Epoch 4/10
938/938 [==============================] - 12s 12ms/step - loss: 0.2074 - accuracy: 0.9372 - val_loss: 0.2361 - val_accuracy: 0.9242
Epoch 5/10
938/938 [==============================] - 11s 12ms/step - loss: 0.1748 - accuracy: 0.9465 - val_loss: 0.2044 - val_accuracy: 0.9344
Epoch 6/10
938/938 [==============================] - 15s 16ms/step - loss: 0.1540 - accuracy: 0.9528 - val_loss: 0.1846 - val_accuracy: 0.9374
Epoch 7/10
938/938 [==============================] - 15s 16ms/step - loss: 0.1364 - accuracy: 0.9585 - val_loss: 0.3012 - val_accuracy: 0.9008
Epoch 8/10
938/938 [==============================] - 13s 14ms/step - loss: 0.1238 - accuracy: 0.9635 - val_loss: 0.1490 - val_accuracy: 0.9501
Epoch 9/10
938/938 [==============================] - 11s 12ms/step - loss: 0.1130 - accuracy: 0.9656 - val_loss: 0.1063 - val_accuracy: 0.9662
Epoch 10/10
938/938 [==============================] - 13s 13ms/step - loss: 0.1036 - accuracy: 0.9682 - val_loss: 0.1113 - val_accuracy: 0.9635
[7] 7
[8] 8
[9] 9
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
```

Keras LSTM after 10 epochs

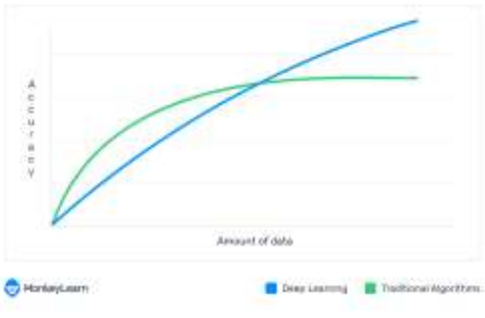
4) What Is Sentiment Analysis with Deep Learning?

A) **Sentiment analysis** is the classification of emotions (positive, negative, and neutral) within data using text analysis techniques. Harnessing the power of deep learning, sentiment analysis models can be trained to understand text beyond simple definitions, read for context, sarcasm, etc., and understand the actual mood and feeling of the writer. For example:

Based on word definitions, alone, the above tweet wouldn't give us much information. But when run through a well-trained sentiment analyzer, the program would understand that this is definitely a ***negative*** tweet. In order to exploit the full power of sentiment analysis tools, we can plug them into deep learning models. As we mentioned earlier, deep learning is a study within machine learning that uses **"artificial neural networks"** to process information much like the human brain does.

Deep learning is hierarchical machine learning that uses multiple algorithms in a progressive chain of events to solve complex problems and allows you to tackle massive amounts of data, accurately and with very little human interaction.

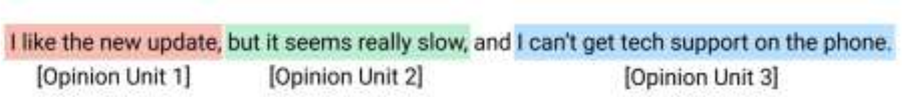
Deep learning and machine learning are sometimes used interchangeably. Deep learning is, indeed, machine learning, but it is more advanced. When basic machine learning makes a mistake, human input is required to correct it – to change the output and **"force"** the model to learn. In deep learning, however, the neural network can learn to correct itself through its advanced algorithm chain.



That said, the initial training of a deep learning model is extremely time-consuming and often requires millions of data points until it begins to learn on its own. To continue with the comparison to the human brain, think about how long it takes a child to build correct sentence structure or learn basic math. However, once they do, they can learn more advanced language or mathematics on their own because they have learned the essential rules and processes.

Once fully trained to effectively teach themselves, machine learning models can perform phenomenal feats. Text analysis, for example, uses **Natural Language processing (NLP**) to break down language and understand it much as a human would: subject, verb, object, etc. It's not until the computer has broken a sentence down, mathematically, can it move on to other analytical processes.

And, of course, it's much more complex than simply dissecting a sentence into subject, verb, object, and moving on. Successful NLP models have taken years to train. However, with the use of NLP, deep learning models can break sentences, paragraphs, and entire documents into individual opinion units:


I like the new update, but it seems really slow, and I can't get tech support on the phone.
[Opinion Unit 1] [Opinion Unit 2] [Opinion Unit 3]

Once broken into opinion units, the model could perform topic classification to organize each statement into predefined categories, like ***Usability (Opinion Unit 1), Functionality (Opinion Unit 2),* and *Support (Opinion Unit 3)*.**

From there, the deep learning model can perform sentiment analysis on each statement by topic: "like the new update" - ***Positive***; "seems really slow" - ***Negative***; "can't get tech support on the phone" - ***Negative***. Now we have sentiment analysis performed on our topic categories:

- Usability: Positive
- Functionality: Negative
- Support: Negative

Imagine this kind of deep learning analysis performed on thousands of customer reviews, social media posts, questionnaires, etc. You can get a broad overview or hundreds of detailed insights. There are **5 major steps** involved in the building a deep learning model for sentiment classification:

Step1: Get data.
Step 2: Generate embeddings
Step 3: Model architecture
Step 4: Model Parameters
Step 5: Train and test the model
Step 6: Run the model

Step1: Get data
Sourcing the labelled data for training a deep learning model is one of the most difficult parts of building a model. Fortunately we can use the Stanford sentiment treebank data for our purpose. The data set "dictionary.txt" consists of 239,233 lines of sentences with an index for each line. The index is used to match each of the sentences to a sentiment score in the file "labels.txt". The score ranges from 0 to 1, 0 being very negative and 1 being very positive.

The below code reads the dictionary.txt and **labels.txt** files, combines the score to each sentences. This code is found within ***train/utility_function.py***

```
def read_data(path):# read dictionary into dfdf_data_sentence = pd.read_table(path +
‘dictionary.txt’)df_data_sentence_processed = df_data_sentence[‘Phrase|Index’].str.split(‘|’,
expand=True)df_data_sentence_processed = df_data_sentence_processed.rename(columns={0: ‘Phrase’, 1:
‘phrase_ids’})# read sentiment labels into dfdf_data_sentiment = pd.read_table(path +
‘sentiment_labels.txt’)df_data_sentiment_processed = df_data_sentiment[‘phrase ids|sentiment
values’].str.split(‘|’, expand=True)df_data_sentiment_processed =
df_data_sentiment_processed.rename(columns={0: ‘phrase_ids’, 1: ‘sentiment_values’})#combine data frames
containing sentence and sentimentdf_processed_all =
df_data_sentence_processed.merge(df_data_sentiment_processed, how=’inner’, on=’phrase_ids’return
df_processed_all
```

The data is split into 3 parts:

- train.csv : This is the main data which is used to train the model. This is 50% of the overall data.
- val.csv : This is a validation data set to be used to ensure the model does not overfit. This is 25% of the overall data.
- test.csv : This is used to test the accuracy of the model post training. This is 25% of the overall data.

Step 2: Generate embeddings

Prior to training this model we are going to convert each of the words into a word embedding. You can think of word embeddings as numerical representation of words to enable our model to learn. Word embeddings are vector representations that capture the context of the underlying words in relation to other words in the sentence. This transformation results in words having similar meaning being clustered closer together in the hyperplane and distinct words positioned further away in the hyperplane.

Convert each word into a word embedding: We are going to use a pre-trained word embedding model know as **GloVe**. For our model we are going to represent each word using a 100 dimension embedding. The detailed code for converting the data into word embedding is in within ***train/utility_function.py.*** This function basically replace each of the words by its respective embedding by performing a lookup from the **GloVe** pre-trained vectors. An illustration of the process is shown below, where each word is converted into an embedding and fed into a neural network.

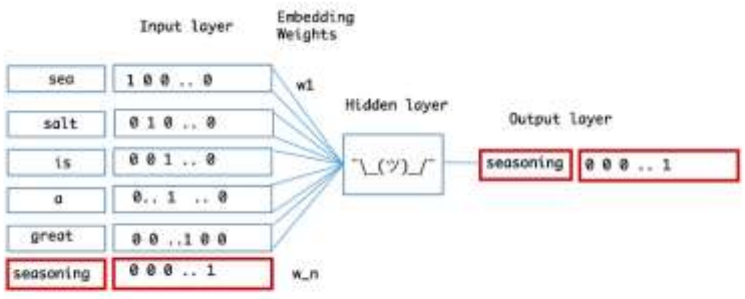


Fig: Converting Sentences to Embedding for a Neural Network

The below code is used to split the data into train, val and test sets. Also the corresponding embeddings for the data is stored in the weight_matrix variable.

```
def load_data_all(data_dir, all_data_path,pred_path, gloveFile, first_run, load_all):   numClasses = 10# Load
embeddings for the filtered glove list
    if load_all == True:
        weight_matrix, word_idx = uf.load_embeddings(gloveFile)
    else:
        weight_matrix, word_idx = uf.load_embeddings(filtered_glove_path)    # create test, validation and trainng
data
    all_data = uf.read_data(all_data_path)
    train_data, test_data, dev_data = uf.training_data_split(all_data, 0.8, data_dir)train_data =
train_data.reset_index()
    dev_data = dev_data.reset_index()
    test_data = test_data.reset_index()    maxSeqLength, avg_words, sequence_length = uf.maxSeqLen(all_data)
  # load Training data matrix
    train_x = uf.tf_data_pipeline_nltk(train_data, word_idx, weight_matrix, maxSeqLength)
    test_x = uf.tf_data_pipeline_nltk(test_data, word_idx, weight_matrix, maxSeqLength)
    val_x = uf.tf_data_pipeline_nltk(dev_data, word_idx, weight_matrix, maxSeqLength)    # load labels data
matrix
    train_y = uf.labels_matrix(train_data)
    val_y = uf.labels_matrix(dev_data)
    test_y = uf.labels_matrix(test_data)return train_x, train_y, test_x, test_y, val_x, val_y, weight_matrix
```

Step3: Model architecture

In order to train the model we are going to use a type of Recurrent Neural Network, know as LSTM (Long Short Term Memory). The main advantage of this network is that it is able to remember the sequence of past data i.e. words in our case in order to make a decision on the sentiment of the word.

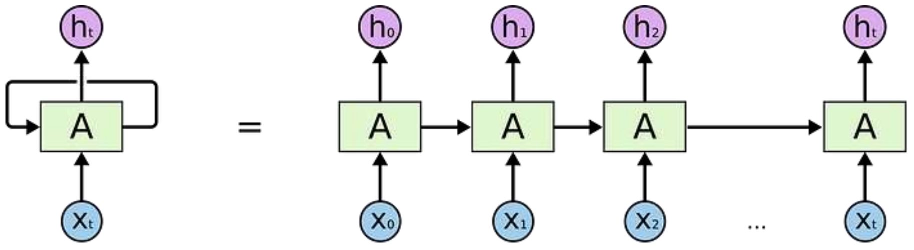


Fig: A RNN Network

As seen in the above picture it is basically a sequence of copies of the cells, where output of each cell is forwarded as input to the next. LSTM network are essentially the same but each cell architecture is a bit more complex. This complexity as seen below allows the each cells to decide which of the past information to remember and the ones to forget.

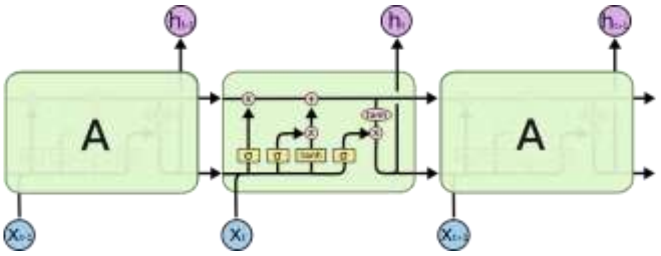
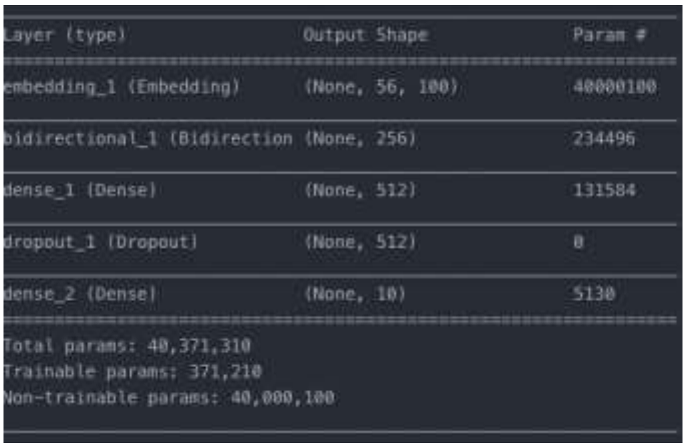


Fig: A LSTM Cell

We are going to create the network using Keras. Keras is built on tensorflow and can be used to build most types of deep learning models. We are going to specify the layers of the model as below. In order to estimate the parameters such as dropout, no of cells etc I have performed a grid search with different parameter values and chose the parameters with best performance.

Layers:



```
ayer (type)                 Output Shape              Param #
=================================================================
mbedding_1 (Embedding)      (None, 56, 100)           40000100
_________________________________________________________________
bidirectional_1 (Bidirection (None, 256)              234496
_________________________________________________________________
dense_1 (Dense)             (None, 512)               131584
_________________________________________________________________
dropout_1 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_2 (Dense)             (None, 10)                5130
=================================================================
Total params: 40,371,310
Trainable params: 371,210
Non-trainable params: 40,000,100
_________________________________________________________________
```

Fig: Model Architecture

Layer 1: An embedding layer of a vector size of 100 and a max length of each sentence is set to 56.
Layer 2: 128 cell bi-directional LSTM layers, where the embedding data is fed to the network. We add a dropout of 0.2 this is used to prevent overfitting.
Layer 3: A 512 layer dense network which takes in the input from the LSTM layer. A Dropout of 0.5 is added here.
Layer 4: A 10 layer dense network with softmax activation, each class is used to represent a sentiment category, with class 1 representing sentiment score between 0.0 to 0.1 and class 10 representing a sentiment score between 0.9 to 1.

Code to create an LSTM model in Keras:
```
import os
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.layers import Bidirectional
from keras.preprocessing import sequence
from keras.layers import Dropout
from keras.models import model_from_json
from keras.models import load_model
def create_model_rnn(weight_matrix, max_words, EMBEDDING_DIM):# create the modelmodel = Sequential()
model.add(Embedding(len(weight_matrix), EMBEDDING_DIM, weights=[weight_matrix],
input_length=max_words, trainable=False))
model.add(Bidirectional(LSTM(128, dropout=0.2, recurrent_dropout=0.2)))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.50))
model.add(Dense(10, activation='softmax'))# Adam Optimiser
model.compile(loss='categorical_crossentropy',optimizer='adam', metrics=['accuracy'])return model
```

Step 4: Model Parameters:

Activation Function: I have used ReLU as the activation function. ReLU is a non-linear activation function, which helps complex relationships in the data to be captured by the model.

Optimiser: We use adam optimiser, which is an adaptive learning rate optimiser.

Loss function: We will train a network to output a probability over the 10 classes using **Cross-Entropy loss**, also called Softmax Loss. It is very useful for multi-class classification.

Step 5: Train and test the model
We start the training of the model by passing the train, validation and test data set into the function below:

```
def train_model(model,train_x, train_y, test_x, test_y, val_x, val_y, batch_size):# save the best model and early
stopping
saveBestModel = keras.callbacks.ModelCheckpoint('../best_weight_glove_bi_100d.hdf5', monitor='val_acc',
verbose=0, save_best_only=True, save_weights_only=False, mode='auto', period=1)earlyStopping =
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=3, verbose=0, mode='auto')# Fit the
model
model.fit(train_x, train_y, batch_size=batch_size, epochs=25,validation_data=(val_x, val_y),
callbacks=[saveBestModel, earlyStopping])# Final evaluation of the model
score, acc = model.evaluate(test_x, test_y, batch_size=batch_size)
return model
```

We have run the training on a **batch size of 500** items at a time. As you increase the batch size the time for training would reduce but it will require additional computational capacity. Hence it is a trade-off between computation capacity and time for training.

The training is set to run for 25 epochs. One epoch would mean that the network has seen the entire training data once. As we increase the number of epochs there is a risk that the model will overfit to the training data. Hence to prevent the model from overfitting I have enabled early stopping. Early stopping is a method that allows us to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out/validation dataset. The model on the test set of 10 class sentiment classification provides a result of **48.6%** accuracy. The accuracy will be much higher on a 2-class binary (positive or negative) data set.

Step 6: Run the model
Once the model is trained you can save the model in keras using the below code.

```
model.save_weights("/model/best_model.h5")
```

The next step is to use the trained model in real time to run predictions on new data. In order to do this, you will need to transform the input data to embeddings, similar to the way we treated our training data. The function ***live_test*** below performs the required pre-processing of the data and returns the result of the trained model. Here in order to ensure robustness of the results of the model I am taking the average top 3 sentiments bands from the model. This provides a better calibration for the model results.

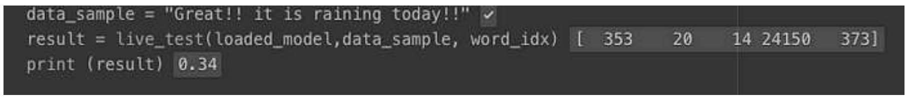
```
def live_test(trained_model, data, word_idx):live_list = []
live_list_np = np.zeros((56,1))# split the sentence into its words and remove any punctuations.tokenizer =
RegexpTokenizer(r'\w+')
data_sample_list = tokenizer.tokenize(data)
labels = np.array(['1','2','3','4','5','6','7','8','9','10'], dtype = "int")# get index for the live stage
data_index = np.array([word_idx[word.lower()] if word.lower() in word_idx else 0 for word in data_sample_list])
data_index_np = np.array(data_index)# padded with zeros of length 56 i.e maximum length
padded_array = np.zeros(56)
padded_array[:data_index_np.shape[0]] = data_index_np
data_index_np_pad = padded_array.astype(int)
live_list.append(data_index_np_pad)
live_list_np = np.asarray(live_list)# get score from the model
score = trained_model.predict(live_list_np, batch_size=1, verbose=0)
single_score = np.round(np.argmax(score)/10, decimals=2) # maximum of the array i.e single band# weighted
score of top 3 bands
top_3_index = np.argsort(score)[0][-3:]
top_3_scores = score[0][top_3_index]
top_3_weights = top_3_scores/np.sum(top_3_scores)
single_score_dot = np.round(np.dot(top_3_index, top_3_weights)/10, decimals = 2)return single_score_dot,
single_score
```

As seen in the code below, you can specify the model path, sample data and the corresponding embeddings to the live_test function. It will return the sentiment of the sample data.

```
# Load the best model that is saved in previous step
weight_path = '/model/best_model.hdf5'
loaded_model = load_model(weight_path)# sample sentence
data_sample = "This blog is really interesting."
result = live_test(loaded_model,data_sample, word_idx)
```

Results
Let us compare the results of our deep learning model to the NLTK model by taking a sample.

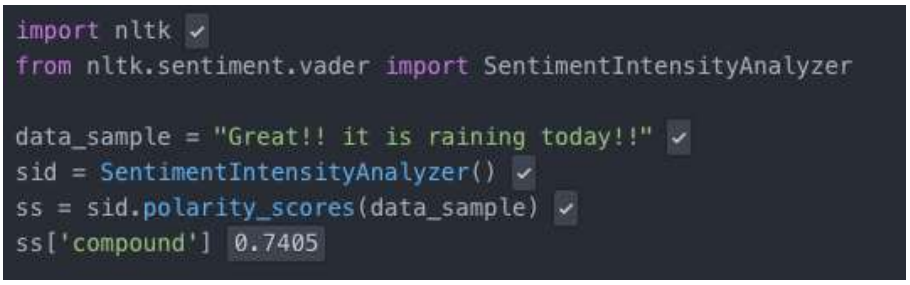
LSTM Model: This sentence *"Great!! it is raining today!!"* contains negative context and our model is able to predict this as seen below. it gives it a score of 0.34.



```
data_sample = "Great!! it is raining today!!" ✓
result = live_test(loaded_model,data_sample, word_idx) [   353     20     14 24150    373]
print (result) 0.34
```

Fig: LSTM Model

NLTK Model: The same sentence when analysed by the bi-gram NLTK model, scores it as being positive with a score of 0.74.



```
import nltk ✓
from nltk.sentiment.vader import SentimentIntensityAnalyzer

data_sample = "Great!! it is raining today!!" ✓
sid = SentimentIntensityAnalyzer() ✓
ss = sid.polarity_scores(data_sample) ✓
ss['compound'] 0.7405
```

Fig:NLTK Model

5) Image Dimensionality Reduction using Encoders LSTM with Keras?
A) Dimensionality Reduction: Dimensionality Reduction is the process of reducing the number of dimensions in the data either by excluding less useful features (Feature Selection) or transform the data into lower dimensions (Feature Extraction). Dimensionality reduction prevents overfitting. Overfitting is a phenomenon in which the model learns too well from the training dataset and fails to generalize well for unseen real-world data.
Types of Feature Selection for Dimensionality Reduction,

- Recursive Feature Elimination
- Genetic Feature Selection
- Sequential Forward Selection

Types of Feature Extraction for Dimensionality Reduction,

- Auto Encoders
- Principal Component Analysis (PCA)
- Linear Determinant Analysis (LDA)

AutoEncoder is an **unsupervised Artificial Neural Network** that attempts to encode the data by compressing it into the lower dimensions (bottleneck layer or code) and then decoding the data to reconstruct the original input. The bottleneck layer (or code) holds the compressed representation of the input data.

In AutoEncoder the number of output units must be equal to the number of input units since we're attempting to reconstruct the input data. AutoEncoders usually consist of an encoder and a decoder. The encoder encodes the provided data into a lower dimension which is the size of the bottleneck layer and the decoder decodes the compressed data into its original form.

The number of neurons in the layers of the encoder will be decreasing as we move on with further layers, whereas the number of neurons in the layers of the decoder will be increasing as we move on with further layers. There are three layers used in the encoder and decoder in the following example. The encoder contains 32, 16, and 7 units in each layer respectively and the decoder contains 7, 16, and 32 units in each layer respectively. The code size/ the number of neurons in bottle-neck must be less than the number of features in the data. Before feeding the data into the AutoEncoder the data must definitely be scaled between 0 and 1 using MinMaxScaler since we are going to use sigmoid activation function in the output layer which outputs values between 0 and 1.

When we are using AutoEncoders for dimensionality reduction we'll be extracting the bottleneck layer and use it to reduce the dimensions. This process can be viewed as **feature extraction**. The type of AutoEncoder that we're using is **Deep AutoEncoder**, where the encoder and the decoder are symmetrical. The Autoencoders don't necessarily have a symmetrical encoder and decoder but we can have the encoder and decoder non-symmetrical as well.

Dimensionality reduction using Keras Auto Encoder

- Prepare Data
- Design Auto Encoder
- Train Auto Encoder
- Use Encoder level from Auto Encoder
- Use Encoder to obtain reduced dimensionality data for train and test sets

```
import os
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

from numpy.random import seed
from sklearn.preprocessing import minmax_scale
from sklearn.model_selection import train_test_split
from keras.layers import Input, Dense
from keras.models import Model

print(os.listdir("../input"))
/opt/conda/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the second argu
ment of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np
.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
['train.csv', 'sample_submission.csv', 'test.csv']
```

Read train and test data

```
train = pd.read_csv('../input/train.csv')
test = pd.read_csv('../input/test.csv')
Dropping Target and ID's from train and test
```

```
target = train['target']
train_id = train['ID']
test_id = test['ID']

train.drop(['target'], axis=1, inplace=True)
train.drop(['ID'], axis=1, inplace=True)
test.drop(['ID'], axis=1, inplace=True)

print('Train data shape', train.shape)
print('Test data shape', test.shape)

Train data shape (4459, 4991)
Test data shape (49342, 4991)
Scaling Train and Test data for Neural Net

train_scaled = minmax_scale(train, axis = 0)
test_scaled = minmax_scale(test, axis = 0)
```

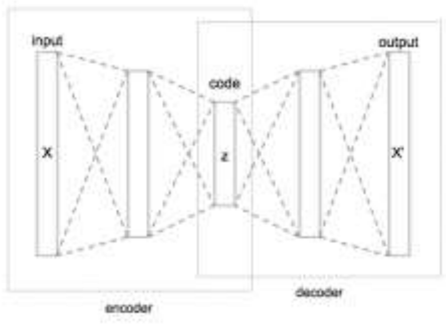
Design Auto Encoder

Auto Encoders are is a type of artificial neural network used to learn efficient data patterns in an unsupervised manner. An Auto Encoder ideally consists of an encoder and decoder. The Neural Network is designed compress data using the Encoding level. The Decoder will try to uncompress the data to the original dimension. To achieve this, the Neural net is trained using the Training data as the training features as well as target.

```
# Training a Typical Neural Net
model.fit(X_train, y_train)

# Training a Auto Encoder
model.fit(X_train, X_train)
```

These are typically used for dimensionality reduction use cases where there are more number of features.



```
# define the number of features
ncol = train_scaled.shape[1]
```

Split train data into train and validation 80:20 in ratio

```
X_train, X_test, Y_train, Y_test = train_test_split(train_scaled, target, train_size = 0.9, random_state = seed(
2017))
/opt/conda/lib/python3.6/site-packages/sklearn/model_selection/_split.py:2026: FutureWarning: From versio
n 0.21, test_size will always complement train_size unless both are specified.
  FutureWarning)

### Define the encoder dimension
encoding_dim = 200

input_dim = Input(shape = (ncol, ))

# Encoder Layers
encoded1 = Dense(3000, activation = 'relu')(input_dim)
encoded2 = Dense(2750, activation = 'relu')(encoded1)
encoded3 = Dense(2500, activation = 'relu')(encoded2)
encoded4 = Dense(2250, activation = 'relu')(encoded3)
encoded5 = Dense(2000, activation = 'relu')(encoded4)
encoded6 = Dense(1750, activation = 'relu')(encoded5)
encoded7 = Dense(1500, activation = 'relu')(encoded6)
encoded8 = Dense(1250, activation = 'relu')(encoded7)
encoded9 = Dense(1000, activation = 'relu')(encoded8)
encoded10 = Dense(750, activation = 'relu')(encoded9)
encoded11 = Dense(500, activation = 'relu')(encoded10)
encoded12 = Dense(250, activation = 'relu')(encoded11)
encoded13 = Dense(encoding_dim, activation = 'relu')(encoded12)

# Decoder Layers
decoded1 = Dense(250, activation = 'relu')(encoded13)
decoded2 = Dense(500, activation = 'relu')(decoded1)
decoded3 = Dense(750, activation = 'relu')(decoded2)
decoded4 = Dense(1000, activation = 'relu')(decoded3)
decoded5 = Dense(1250, activation = 'relu')(decoded4)
decoded6 = Dense(1500, activation = 'relu')(decoded5)
decoded7 = Dense(1750, activation = 'relu')(decoded6)
decoded8 = Dense(2000, activation = 'relu')(decoded7)
decoded9 = Dense(2250, activation = 'relu')(decoded8)
decoded10 = Dense(2500, activation = 'relu')(decoded9)
decoded11 = Dense(2750, activation = 'relu')(decoded10)
decoded12 = Dense(3000, activation = 'relu')(decoded11)
decoded13 = Dense(ncol, activation = 'sigmoid')(decoded12)

# Combine Encoder and Deocder layers
autoencoder = Model(inputs = input_dim, outputs = decoded13)
```

```
# Compile the Model
autoencoder.compile(optimizer = 'adadelta', loss = 'binary_crossentropy')
autoencoder.summary()
Total params: 101,590,191
Trainable params: 101,590,191
Non-trainable params: 0
```

Train Auto Encoder

```
autoencoder.fit(X_train, X_train, nb_epoch = 10, batch_size = 32, shuffle = False, validation_data = (X_test,
X_test))
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:1: UserWarning: The `nb_epoch` argument in
`fit` has been renamed `epochs`.
  """Entry point for launching an IPython kernel.
Train on 4013 samples, validate on 446 samples

Epoch 1/10
4013/4013 [==============================] - 31s 8ms/step - loss: 0.6861 - val_loss: 0.6668
Epoch 2/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.1167 - val_loss: 0.0160
Epoch 3/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0149 - val_loss: 0.0147
Epoch 4/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0138 - val_loss: 0.0140
Epoch 5/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0136 - val_loss: 0.0140
Epoch 6/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 7/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 8/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 9/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 10/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0134 - val_loss: 0.0139

<keras.callbacks.History at 0x7fb8ba981d30>
```

Use Encoder level to reduce dimension of train and test data

```
encoder = Model(inputs = input_dim, outputs = encoded13)
encoded_input = Input(shape = (encoding_dim, ))
```

Predict the new train and test data using Encoder

```
encoded_train = pd.DataFrame(encoder.predict(train_scaled))
encoded_train = encoded_train.add_prefix('feature_')
```

```
encoded_test = pd.DataFrame(encoder.predict(test_scaled))
encoded_test = encoded_test.add_prefix('feature_')
```

Add target to train

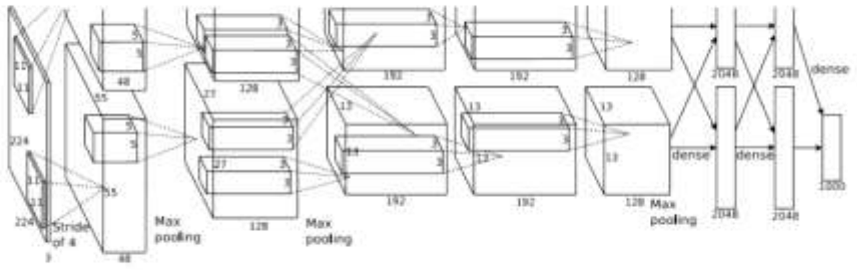
```
encoded_train['target'] = target

print(encoded_train.shape)
encoded_train.head()
(4459, 201)
print(encoded_test.shape)
encoded_test.head()
(49342, 200)
In [17]:
linkcode
encoded_train.to_csv('train_encoded.csv', index=False)
encoded_test.to_csv('test_encoded.csv', index=False)
```

6) Briefly Describe about AlexNet and VGGNet?

A) **AlexNet:** AlexNet is a deep learning architecture and represents a variation of the convolutional neural network. It was originally proposed by **Alex Krizhevsky** during his research, under the guidance of Geoffrey E. Hinton, a prominent figure in the field of deep learning research. In 2012, Alex Krizhevsky participated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC2012) and utilized the AlexNet model, achieving an impressive top-5 error rate of 15.3%, surpassing the runner-up by more than 10.8 percentage points.

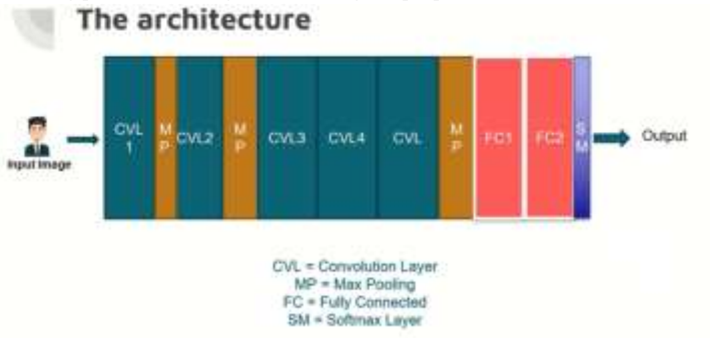
Commented [S1]:



Architecture of AlexNet

- The AlexNet architecture consists of eight layers in total.
- The first five layers are convolutional layers.
- The sizes of the convolutional filters are 11×11, 5×5, 3×3, 3×3, and 3×3 for the respective convolutional layers.
- Some of the convolutional layers are followed by max-pooling layers, which help reduce spatial dimensions while retaining important features.
- The activation function used in the network is the Rectified Linear Unit (ReLU), known for its superior performance compared to sigmoid and tanh functions.
- After the convolutional layers, there are three fully connected layers.
- The network’s parameters can be tuned based on the training performance.

- The AlexNet can be used with transfer learning, utilizing pre-trained weights on the ImageNet dataset to achieve exceptional performance. However, in this article, we will define the CNN without using pre-trained weights, following the proposed architecture.



Key Components of AlexNet Architecture

Point 1 – Convolutional Neural Network (CNN): AlexNet is a deep Convolutional Neural Network (CNN) architecture designed for image classification tasks. CNNs are specifically suited for visual recognition tasks, leveraging convolutional layers to learn features from images hierarchically.

Point 2 – Architecture: AlexNet consists of eight layers, with the first five being convolutional layers and the last three being fully connected layers. The convolutional layers are designed to extract relevant patterns and features from input images, while the fully connected layers perform the classification based on those features.

Point 3 – ReLU Activation: Rectified Linear Unit (ReLU) activation functions are used after each convolutional and fully connected layer. ReLU introduces non-linearity, enabling the network to model more complex relationships in the data.

Point 4 – Max Pooling: Max pooling layers are applied after certain convolutional layers to reduce spatial dimensions while retaining essential features. This downsampling process helps reduce computation and controls overfitting.

Point 5 – Local Response Normalization: Local Response Normalization (LRN) is implemented to enhance generalization by normalizing the output of a neuron relative to its neighbors. This creates a form of lateral inhibition, making the network more robust to variations in input data.

Point 6 – Dropout: AlexNet uses dropout regularization during training, where random neurons are dropped out during forward and backward passes. This technique prevents overfitting and improves the model’s generalization performance.

Point 7 – Batch Normalization: Batch Normalization is applied to normalize the outputs of each layer within a mini-batch during training. It stabilizes and accelerates the training process, allowing for higher learning rates and deeper architectures.

Point 8 – Softmax Activation: The final layer of AlexNet uses the softmax activation function to convert the model’s raw output into class probabilities. This allows the network to provide a probability distribution over the possible classes for each input image.

Point 9 – Training and Optimization: AlexNet is trained using stochastic gradient descent with momentum. The learning rate is adjusted during training, and data augmentation techniques are applied to increase the diversity of the training dataset.

Point 10 – ImageNet Competition: AlexNet achieved significant success when it participated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. Its superior performance helped popularize deep learning and CNNs for image classification tasks.

AlexNet's emergence in 2012 marked a pivotal moment in the world of deep learning and computer vision. Its innovative architecture, depth, and novel techniques like ReLU activation, dropout, and data augmentation set a new standard for CNN models. With its impressive performance and contributions to the ImageNet competition, AlexNet laid the groundwork for future advancements in the field, paving the way for a new era of artificial intelligence applications.

VGGNet: VGG stands for Visual Geometry Group; it is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers. The "deep" refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers. The VGG architecture is the basis of ground-breaking object recognition models. Developed as a deep neural network, the **VGGNet** also surpasses baselines on many tasks and datasets beyond ImageNet. Moreover, it is now still one of the most popular image recognition architectures.

The VGG model, or VGGNet, that supports 16 layers is also referred to as VGG16, which is a convolutional neural network model proposed by A. Zisserman and K. Simonyan from the University of Oxford. These researchers published their model in the research paper titled, "Very Deep Convolutional Networks for Large-Scale Image Recognition." The VGG16 model achieves almost 92.7% top-5 test accuracy in ImageNet.

ImageNet is a dataset consisting of more than 14 million images belonging to nearly 1000 classes. Moreover, it was one of the most popular models submitted to ILSVRC-2014. It replaces the large kernel-sized filters with several 3×3 kernel-sized filters one after the other, thereby making significant improvements over AlexNet.

The VGG16 model was trained using Nvidia Titan Black GPUs for multiple weeks. As mentioned above, the VGGNet-16 supports 16 layers and can classify images into 1000 object categories, including keyboard, animals, pencil, mouse, etc. Additionally, the model has an image input size of 224-by-224. Real-time object detection application built on Viso Suite.

The concept of the **VGG19** model (also VGGNet-19) is the same as the VGG16 except that it supports 19 layers. The "16" and "19" stand for the number of weight layers in the model (convolutional layers). This means that VGG19 has three more convolutional layers than VGG16.

VGG Architecture VGGNets are based on the most essential features of convolutional neural networks (CNN). The following graphic shows the basic concept of how a CNN works: The architecture of a Convolutional Neural Network: Image data is the input of the CNN; the model output provides prediction categories for input images.

The VGG network is constructed with very small convolutional filters. The VGG-16 consists of 13 convolutional layers and three fully connected layers.

The architecture of VGG:

Input: The VGGNet takes in an image input size of 224×224. For the ImageNet competition, the creators of the model cropped out the center 224×224 patch in each image to keep the input size of the image consistent.

Convolutional Layers: VGG's convolutional layers leverage a minimal receptive field, i.e., 3×3, the smallest possible size that still captures up/down and left/right. Moreover, there are also 1×1 convolution filters acting as a linear transformation of the input. This is followed by a ReLU unit, which is a huge innovation from AlexNet that reduces training time. ReLU stands for rectified linear unit activation function; it is a piecewise linear function that will output the input if positive; otherwise, the output is zero. The convolution stride is fixed at 1 pixel to keep the spatial resolution preserved after convolution (stride is the number of pixel shifts over the input matrix).

Hidden Layers: All the hidden layers in the VGG network use ReLU. VGG does not usually leverage Local Response Normalization (LRN) as it increases memory consumption and training time. Moreover, it makes no improvements to overall accuracy.

Fully-Connected Layers: The VGGNet has three fully connected layers. Out of the three layers, the first two have 4096 channels each, and the third has 1000 channels, 1 for each class. Fully Connected Layers VGG16 Architecture The number 16 in the name VGG refers to the fact that it is 16 layers deep neural network (VGGnet). This means that VGG16 is a pretty extensive network and has a total of around 138 million parameters. Even according to modern standards, it is a huge network.

However, VGGNet16 architecture's simplicity is what makes the network more appealing. Just by looking at its architecture, it can be said that it is quite uniform. There are a few convolution layers followed by a pooling layer that reduces the height and the width. If we look at the number of filters that we can use, around 64 filters are available that we can double to about 128 and then to 256 filters. In the last layers, we can use 512 filters.

The number of filters that we can use doubles on every step or through every stack of the convolution layer. This is a major principle used to design the architecture of the VGG16 network. One of the crucial downsides of the VGG16 network is that it is a huge network, which means that it takes more time to train its parameters. Because of its depth and number of fully connected layers, the VGG16 model is more than 533MB. This makes implementing a VGG network a time-consuming task.

The VGG16 model is used in several deep learning image classification problems, but smaller network architectures such as GoogLeNet and SqueezeNet are often preferable. In any case, the VGGNet is a great building block for learning purposes as it is straightforward to implement. Performance of VGG Models VGG16 highly surpasses the previous versions of models in the ILSVRC-2012 and ILSVRC-2013 competitions.

Moreover, the VGG16 result is competing for the classification task winner (GoogLeNet with 6.7% error) and considerably outperforms the ILSVRC-2013 winning submission Clarifai. It obtained 11.2% with external training data and around 11.7% without it. In terms of the single-net performance, the VGGNet-16 model achieves the best result with about 7.0% test error, thereby surpassing a single GoogLeNet by around 0.9%.