MACHINE LEARNING NANODEGREE CAPSTONE PROJECT

PROJECT REPORT

LANGUAGE DETECTION USING CNNs

DEFINITION

Domain Background

This project comes under **Computer Vision**. Computer vision is an interdisciplinary field that deals with how computers can be made for gaining high-level understanding from digital images or videos. From the perspective of engineering, it seeks to automate tasks that the human visual system can do.

Computer vision tasks include methods for acquiring, processing, analyzing and understanding digital images, and extraction of high-dimensional data from the real world in order to produce numerical or symbolic information, e.g., in the forms of decisions. Understanding in this context means the transformation of visual images (the input of the retina) into descriptions of the world that can interface with other thought processes and elicit appropriate action.

This project particularly focuses on Image Classification. The classical problem in computer vision, image processing, and machine vision is that of determining whether or not the image data contains some specific object, feature, or activity.

Problem Statement

Motivation for this project actually came from a project at work. We have an application which lets the school going kids click an image of their homework question, and we provide them with a detailed solution. Now as you can imagine, this relies heavily on OCRs as we depend on the text it returns to search for answers in our database. But since the OCRs work really well on English but fail to return text for other regional languages, many of the questions asked by the students go unanswered.

Predominantly the images contain English, Hindi and Math questions. Google Vision is the OCR we use. For hindi images, the API worked better whenever I passed Hindi as a language hint. Now I know that Google Vision is not great with math, but it was just an extension of the idea to detect math images as well.

In order to get better results for Hindi images, I'll have to somehow detect the language before sending the images to Google Vision. That is when I thought of using CNNs to detect language.

As anyone starting out with an idea would do, I started googling stuff about language detection. That is when I found this really interesting paper
https://hal.archives-ouvertes.fr/hal-01282930/document

The method used in this paper involves three subtasks: writing type identification (printed/handwritten), script identification and language identification. The methods for the writing type recognition and the script discrimination are based on the analysis of the connected components while the language identification approach relies on a statistical text analysis, which requires a recognition engine.

Personally, I'd want to start with something much simpler and then build on it.

Metrics

We will be monitoring loss and accuracy of the model. The model keeps updating its hyperparameters after every epoch. Ideally loss should decrease and accuracy should increase with every epoch.

Loss function used is ‘Categorical Cross-entropy’. In information theory, the cross entropy between two probability distributions p and q over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set, if a coding scheme is used that is optimized for an "unnatural" probability distribution q, rather than the "true" distribution p.

ANALYSIS

Data Exploration

Let’s first go over the different files present and what are they used for.

All the files necessary for this project are present in the capstone-project directory.

Capstone-project directory contains the following files and folders:

1. **Capstone Project proposal.pdf** - This file is the Capstone Project Proposal as it is evident from its name.

2. **images folder**: This folder initially contains the following files

There are 6 files starting with “ud_”. These files contain the URLs of images to be downloaded.

- *ud_english_train:* This file contains urls for images to downloaded into english-train folder.
- *ud_english_test:* This file contains urls for images to downloaded into english-test folder.
- *ud_hindi_train*: This file contains urls for images to downloaded into hindi-train folder.
- *ud_hindi_test*: This file contains urls for images to downloaded into hindi-test folder.
- *ud_math_train*: This file contains urls for images to downloaded into math-train folder.
- *ud_math_test*: This file contains urls for images to downloaded into math-test folder.

Images folder also contains a *prepare-images.sh* file, which is a helper file to download images.

3. **classes.json:**

This file contains a key-value mapping of path to training folder and class label.

Eg: “/path/to/english-train”:”english”

This file is used to load training images into the model.

4. **classes-pred.json:**

This file contains a key-value mapping of path to testing folder and class label.

Eg: “/path/to/english-test”:”english”

This file is used to load testing images into the model.

5. language_classifier.py:

This Python2 file, contains the code to do the following things:

1. Train the benchmark model
2. Train the CNN
3. Test the benchmark model
4. Test the CNN

6. Charts

Contains the bar graphs used in this report

7. Logs

Includes two files, logs of training both the benchmark model named fc_logs and CNN named cnn_logs.

About the Dataset

Input images are obtained from the images searched by the students. Dataset consists of around 200 images for each english, hindi and math categories. Test set consists of around 20 images for every category. They are put into different folders and the folders are used as their class labels.

There are 3 output class labels.

1. eng
2. hin
3. math

These class labels are encoded using MultiLabelBinarizer. It converts the above class labels into a matrix of 3*3.

The CNN requires all the images to be of the same size before being fed into them. Using PIL library of Python, I will reshape all the images into a common shape. Then every image will be converted to matrix using the imread function of PIL. All of these matrices are put

into an numpy array. Then it is normalised by subtracting every value from the mean and dividing by standard deviation.

CNN is trained on these images and then it is tested against the 3 test folders.

Running the project

1. **Get the images ready**

   To get the dataset ready, run the prepare-images.sh file in the images folder by running this command from the capstone-project directory:

   sh images/prepare-images.sh

   This command should create the training and testing folders and download images into them.

2. **Get to know the different flags available in language_classifier.py**

   Run this command

   python language_classifier.py -h

   This would print out the help

   Usage: python language_classifier.py [-h] [-classes CLASSES] [-model MODEL]

   [-shape SHAPE] [-saveas SAVEAS] [-epochs EPOCHS]

   [--no-resize] [-resize] [-pred]

   [--fully-connected-only]

   optional arguments:

   -h, --help show this help message and exit

-model MODEL pass an already trained model for further training

-shape SHAPE shape the images should be resized to

-saveas SAVEAS save the model as

-epochs EPOCHS the number of epochs the model should be trained on

--no-resize all images are of same shape, no need to resize

-resize all images are not of same shape, resize

-pred run prediction instead of training

--fully-connected-only Use only fully connected layer

Required Arguments:

-classes CLASSES json containing folder path as the key and class as value

3. **As mentioned before language_classifier.py file can be used for 4 purposes**

I. *Training the benchmark model:*

Run this command to train the benchmark model.

```
python language_classifier.py -classes classes.json -epochs 100
--fully-connected-only -saveas benchmark.h5
```

This command prints out some important info such as the shape the images are being resized to, the output labels and their transformation. Making a note of shape will be handy while running prediction.

II. Testing the benchmark model:

Run this command to test the benchmark model

```
python language_classifier.py -classes classes-pred.json -model benchmark.h5 -shape 311,897 --fully-connected-only -pred
```

Change shape to the value that was printed while training, doing so will prevent the model from calculating it again. Replace 311,897 by the value printed.

III. Training the CNN

Run this command to train the CNN model.

```
python language_classifier.py -classes classes.json -epochs 100 -saveas cnn.py
```

This command prints out some important info such as the shape the images are being resized to, the output labels and their transformation. Making a note of shape will be handy while running prediction.

IV. Testing the CNN

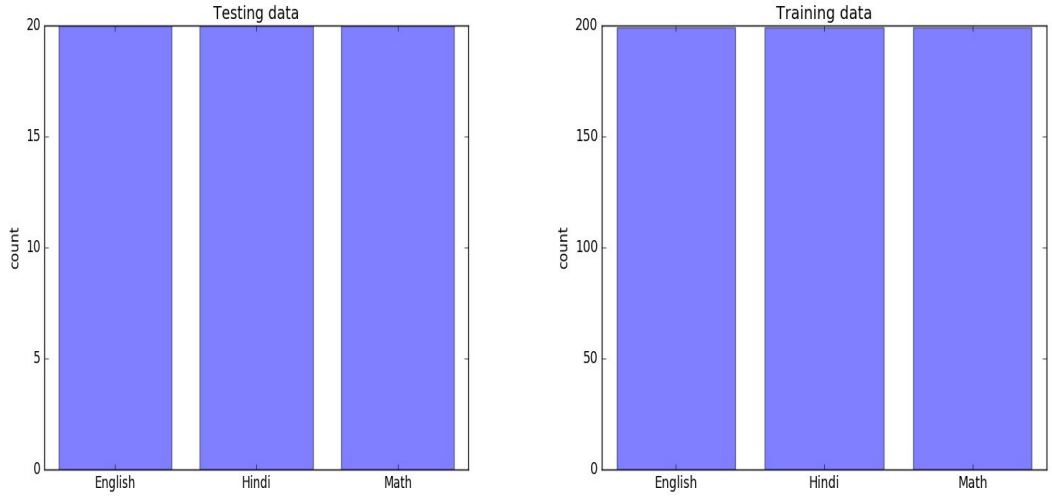
Run this command to test the cnn model

```
python language_classifier.py -classes classes-pred.json -model cnn.h5 -shape 311,897 -pred
```

Change shape to the value that was printed while training, doing so will prevent the model from calculating it again. Replace 311,897 by the value printed.

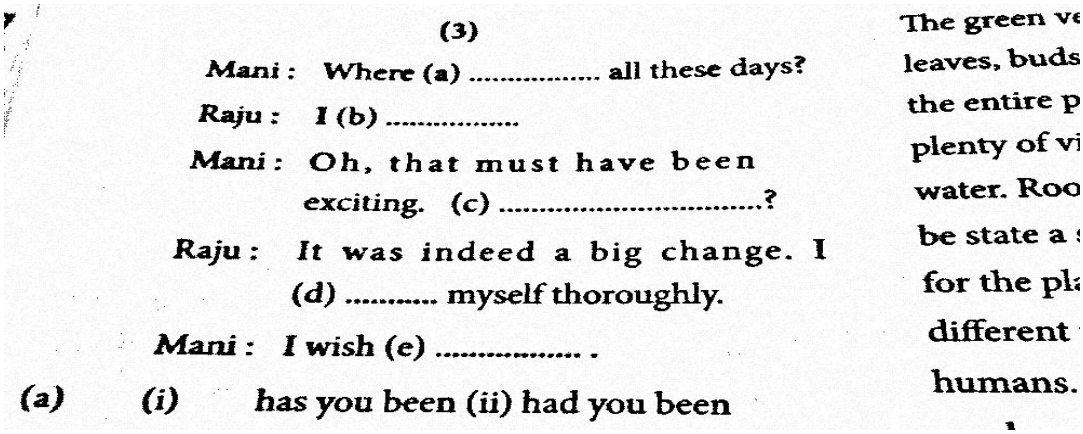
Exploratory Visualization

Let's take a look at some charts which denote the number of images in our dataset. As we can see all of the three labels contain around 200 images each for training and 20 images in each of the label categories for testing.



Let's also take a look at one example mage from all the categories

English:


(3)

Mani : Where (a) ..... all these days?

Raju : I (b) .....

Mani : Oh, that must have been exciting. (c) .....?

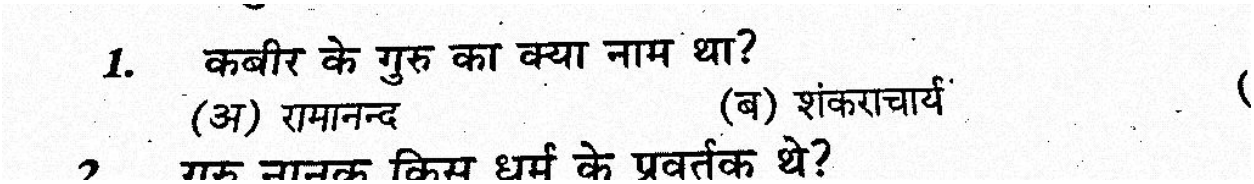
Raju : It was indeed a big change. I (d) ..... myself thoroughly.

Mani : I wish (e) ..... .

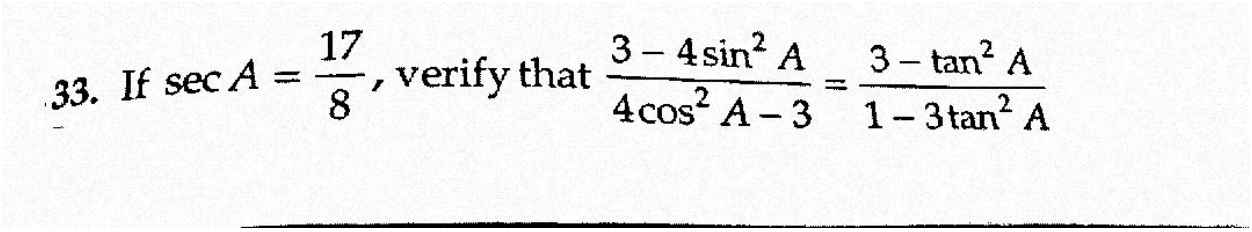
(a) (i) has you been (ii) had you been

The green ve leaves, buds the entire p plenty of vi water. Roo be state a s for the pla different humans.

Hindi:


1. कबीर के गुरु का क्या नाम था?
(अ) रामानन्द (ब) शंकराचार्य
2 गुरु नानक किस धर्म के प्रवर्तक थे?

Math:


33. If $\sec A = \frac{17}{8}$, verify that $\frac{3-4\sin^2 A}{4\cos^2 A - 3} = \frac{3-\tan^2 A}{1-3\tan^2 A}$

Algorithms and Techniques

The purpose of the project is to detect language by using convolutional neural nets, without performing tasks such as writing type identification (printed/handwritten) and script identification. This means the CNN should on its own learn some sort of a function which maps the image to one of the available output labels. Let us first understand how a CNN works.

CNNs are suitable for Image Classification tasks because they can directly work on volumes of data, unlike Fully Connected Neural Nets which expect the input to be a vector. Thus the CNNs are able to preserve spatial information of the images.

CNNs compare images piece by piece. The pieces that it looks for are called features. By finding rough feature matches in roughly the same positions in two images, CNNs get a lot better at seeing similarity than whole-image matching schemes.

Each feature is like a mini-image—a small two-dimensional array of values. Features match common aspects of the images. In the case of X images, features consisting of diagonal lines and a crossing capture all the important characteristics of most X's. These features will probably match up to the arms and center of any image of an X.

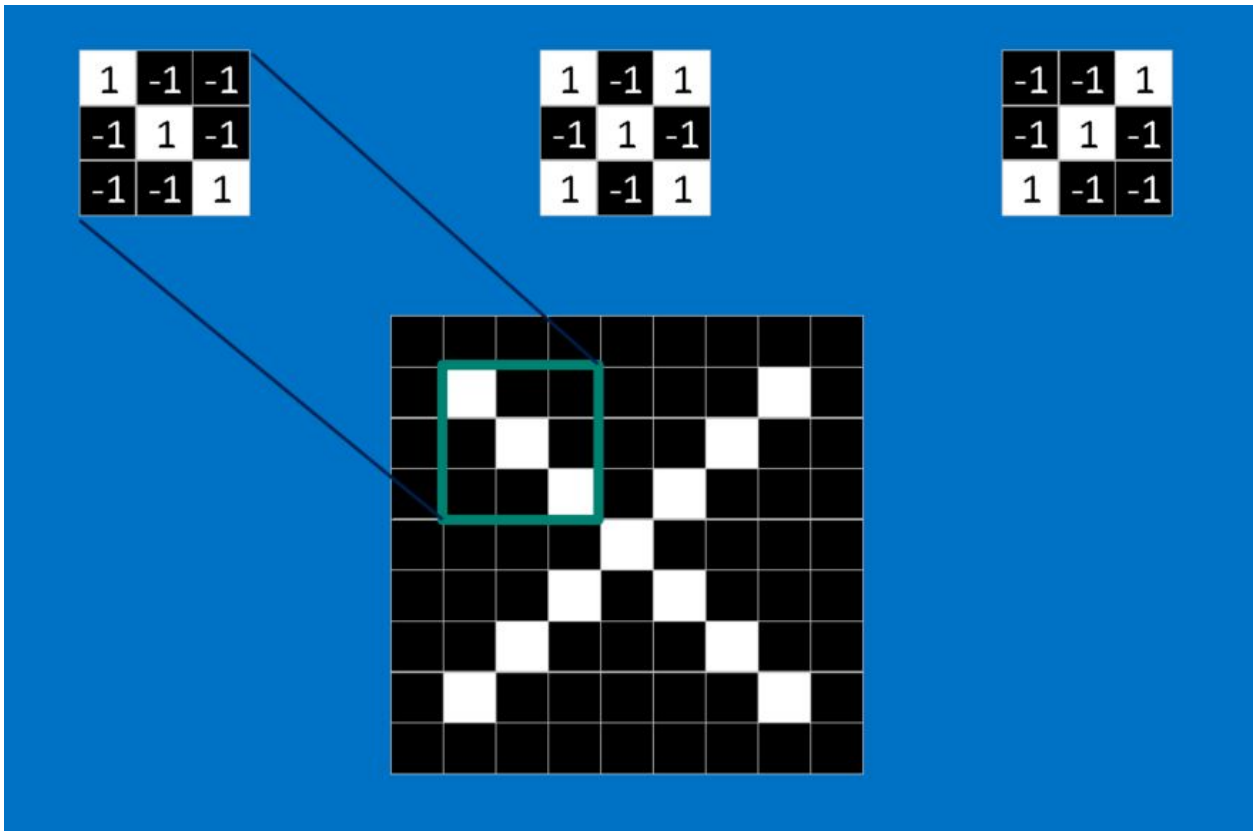


Fig 1. Green box represents a feature

CNN used for this project contains a mixture of several types of layers. They are Convolutional Layers, Pooling Layers, Regularization Layers and Fully Connected Layers. Let's go into the details of these and see how they work.

Convolution Layer:

When presented with a new image, the CNN doesn't know exactly where these features will match so it tries them everywhere, in every possible position. In calculating the match to a feature across the whole image, we make it a filter. The math we use to do this is called convolution, from which Convolutional Neural Networks take their name.

The math behind convolution is nothing that would make a sixth-grader uncomfortable. To calculate the match of a feature to a patch of the image, simply multiply each pixel in the feature by the value of the corresponding pixel in the image. Then add up the answers and divide by the total number of pixels in the feature. If both pixels are white (a value of 1)

then 1 * 1 = 1. If both are black, then (-1) * (-1) = 1. Either way, every matching pixel results in a 1. Similarly, any mismatch is a -1. If all the pixels in a feature match, then adding them up and dividing by the total number of pixels gives a 1. Similarly, if none of the pixels in a feature match the image patch, then the answer is a -1.

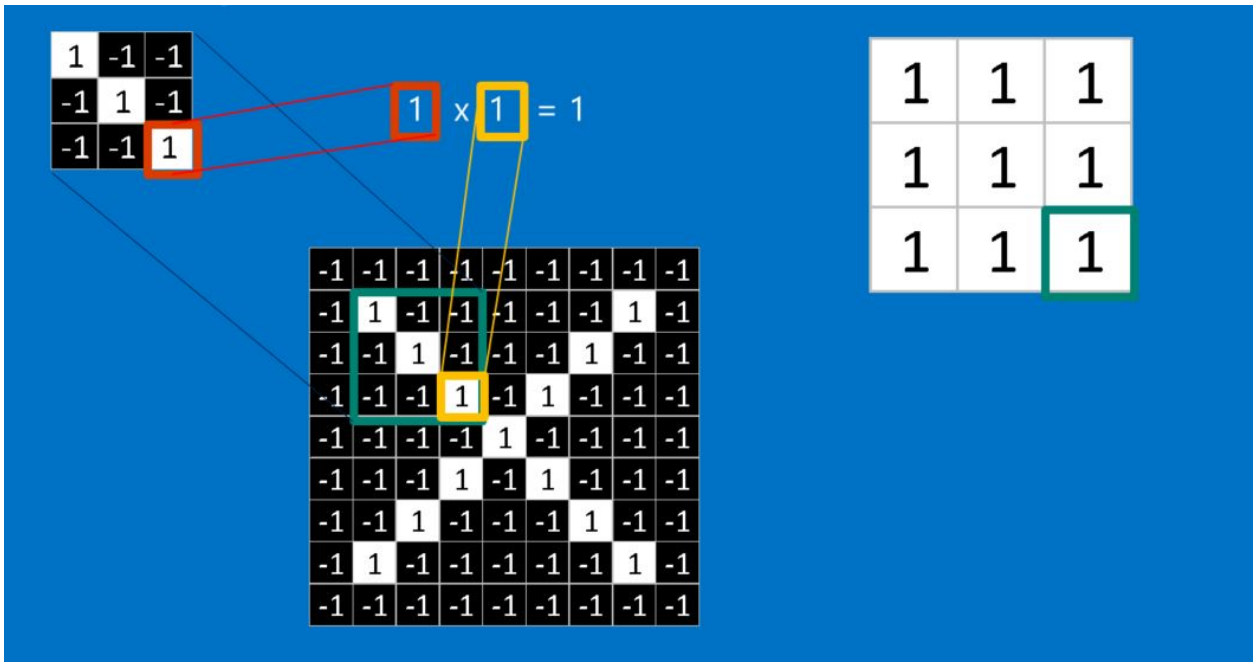


Fig 2. Multiplying the numbers in the feature matrix with the input matrix to get the result on top right.

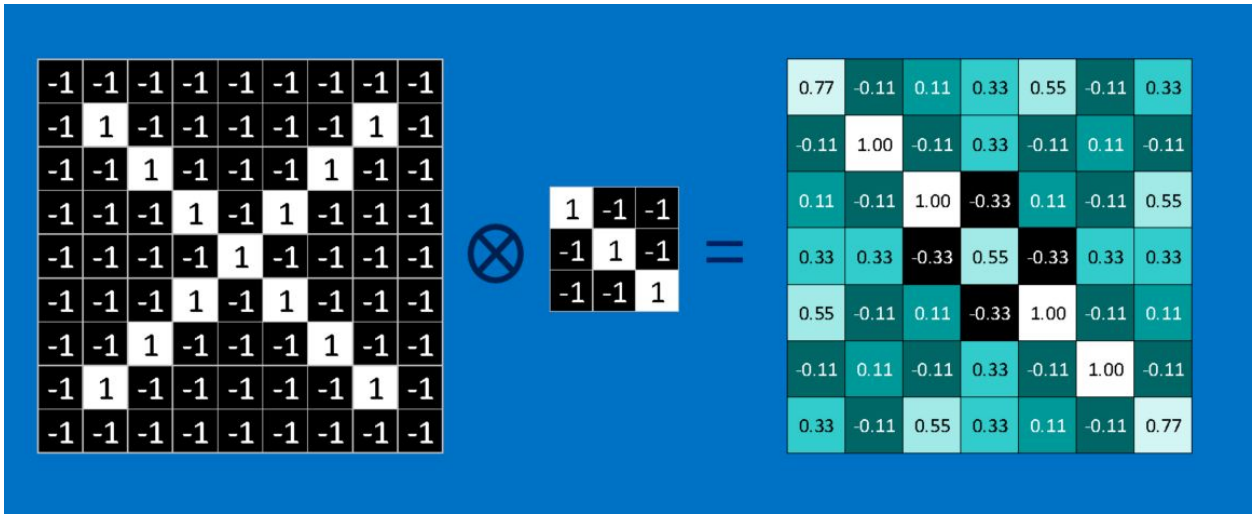


Fig 3. This feature was able to recognize one of the diagonals

To complete our convolution, we repeat this process, lining up the feature with every possible image patch. We can take the answer from each convolution and make a new two-dimensional array from it, based on where in the image each patch is located. This map of matches is also a filtered version of our original image. It's a map of where in the image the feature is found. Values close to 1 show strong matches, values close to -1 show strong matches for the photographic negative of our feature, and values near zero show no match of any sort.

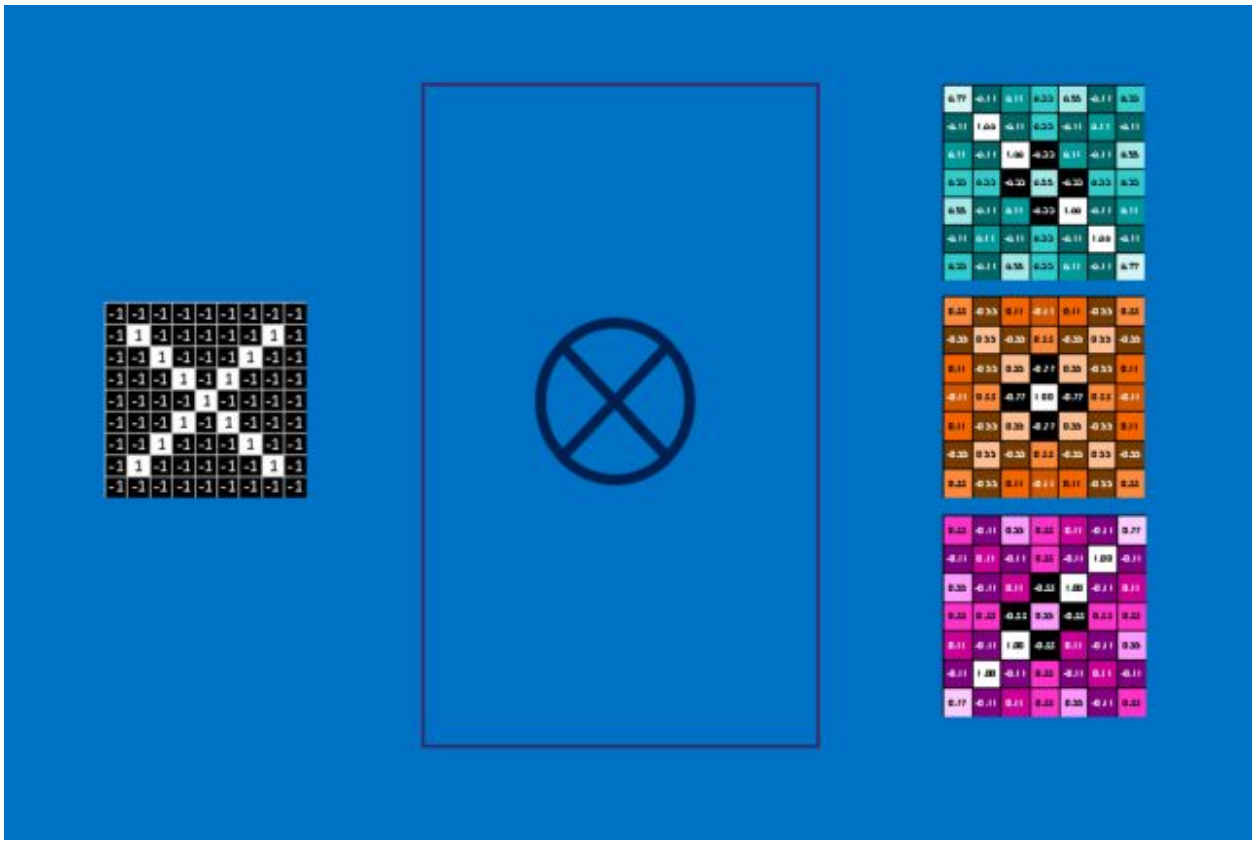


Fig 4. Input matrix convolved with many features

The next step is to repeat the convolution process in its entirety for each of the other features. The result is a set of filtered images, one for each of our filters. It's convenient to think of this whole collection of convolution operations as a single processing step. In CNNs this is referred to as a convolution layer, hinting at the fact that it will soon have other layers added to it.

Pooling Layer:

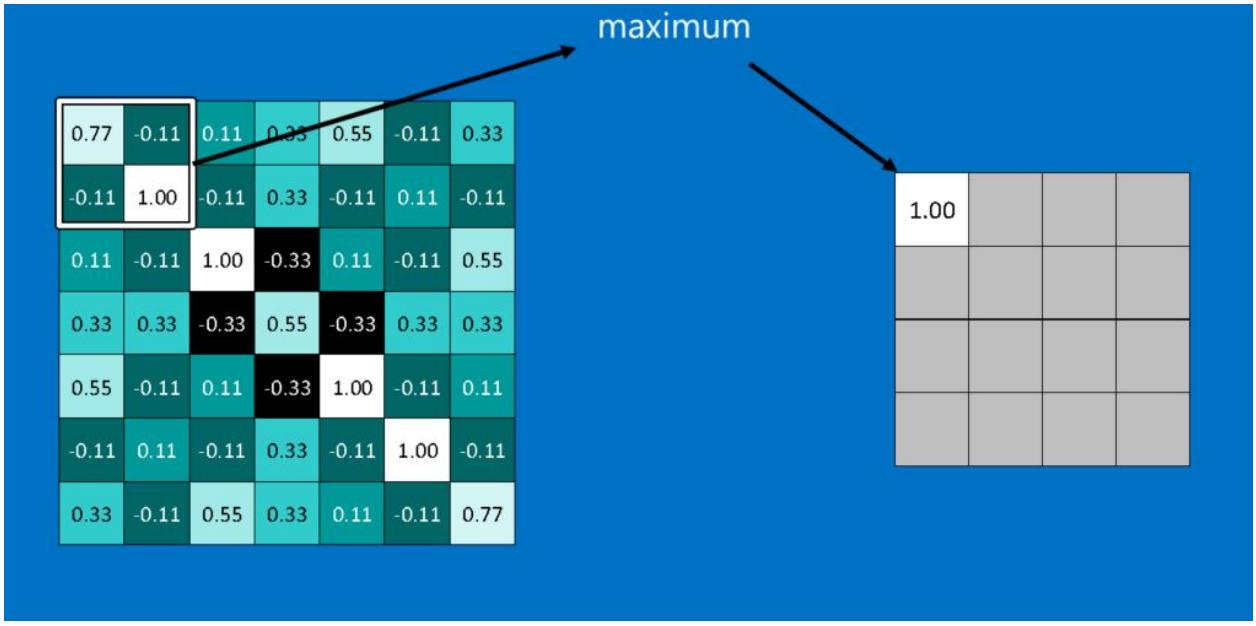


Fig 5. Pooling in action

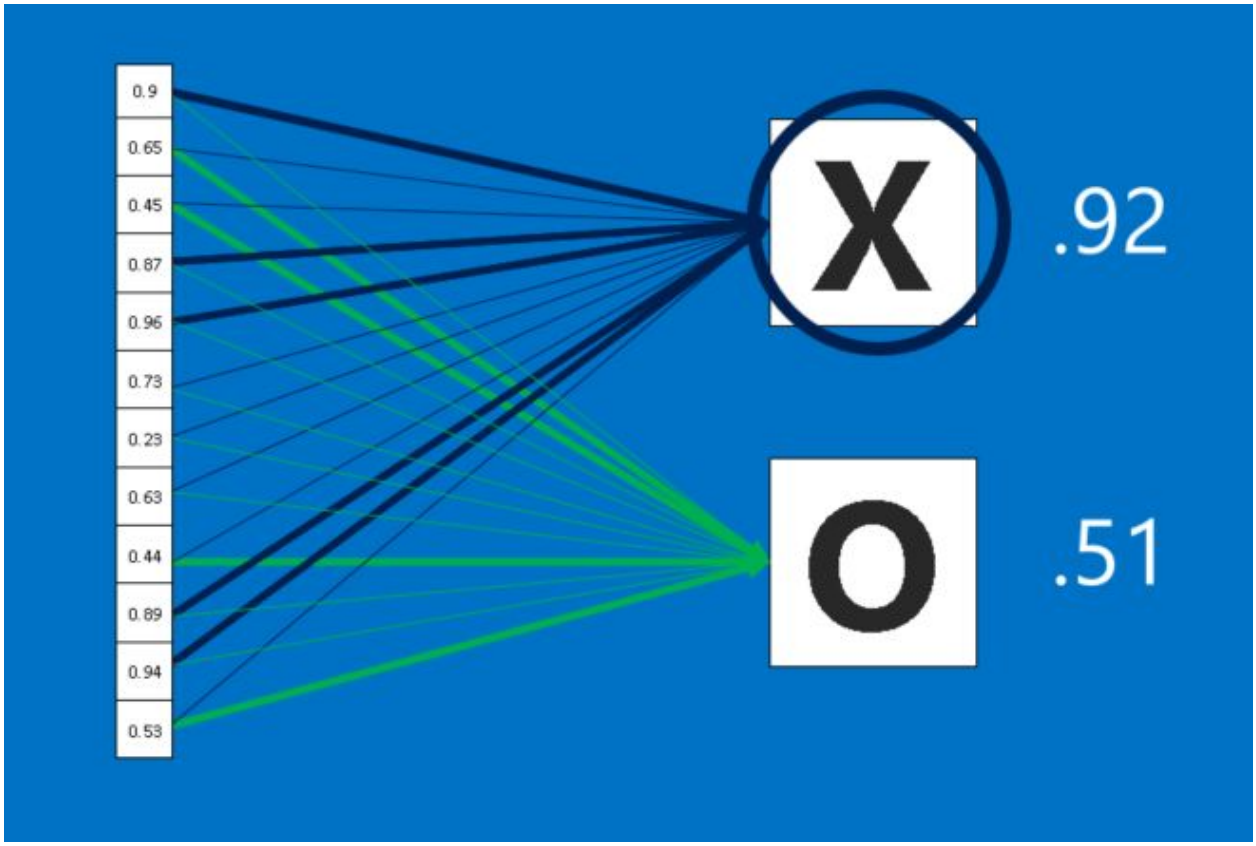
Another power tool that CNNs use is called pooling. Pooling is a way to take large images and shrink them down while preserving the most important information in them. The math behind pooling is second-grade level at most. It consists of stepping a small window across an image and taking the maximum value from the window at each step. In practice, a window 2 or 3 pixels on a side and steps of 2 pixels work well.

After pooling, an image has about a quarter as many pixels as it started with. Because it keeps the maximum value from each window, it preserves the best fits of each feature within the window.

Regularization Layer:

Regularization Layers are used to manage model's complexity. In this project I have used Dropout regularization. It randomly drops some nodes on a pass, thus decreasing the number of hyperparameters to learn and thus preventing model overfitting.

Fully Connected Layer:



CNNs have one more arrow in their quiver. Fully connected layers take the high-level filtered images and translate them into votes. In our case, we only have to decide between two categories, X and O. Fully connected layers are the primary building block of traditional neural networks. Instead of treating inputs as a two-dimensional array, they are treated as a single list and all treated identically. Every value gets its own vote on whether the current image is an X or and O. However, the process isn't entirely democratic. Some values are much better than others at knowing when the image is an X, and some are particularly good at knowing when the image is an O. These get larger votes than the others. These votes are expressed as weights, or connection strengths, between each value and each category.

Benchmark

Benchmark model here is a fully connected neural network made up of 3 fully connected layers.

1st Layer: This is made up of 32 nodes and uses a ‘relu’ activation function.

2nd Layer: This is made up of 64 nodes and uses a ‘relu’ activation function.

3rd Layer: This is made up of nodes equal to the number of output labels, in this case 3 and uses a ‘softmax’ activation function.

This model is trained on the same dataset as the CNN for 100 epochs. It achieves the following results:

	Accuracy	Loss
Training Data	0.7804	3.5354
Testing Data	0.3333	10.7454

With the lower bound set, let’s see if the CNN architecture achieves better results.

METHODOLOGY

Data Preprocessing

Before the images are passed on to the CNN layers, it should be preprocessed. The input images that i've chosen for this task are of different shapes. That would require that images are reshaped to a common shape before being fed to the model.

Using PIL library of Python, I will reshape all the images into a common shape. Then every image will be converted to matrix using the imread function of PIL. All of these matrices are put into an numpy array. Then it is normalised by subtracting every value from the mean and dividing by standard deviation.

Implementation

Library Used:

I will be using Keras for this project.

The Design:

The design that I have thought for this project would consist of CNN with **4 sets** of convolutional layers. Each set consists of a Convolutional Layer, a Max Pooling layer and a Dropout regularization layer. These are followed by 2 Fully Connected layers. The last layer is responsible for predicting the output.

First Set consists of:

- A Convolutional Layer with 32 filters of size (3,3). Chosen activation function for this layer is 'relu'.
- A Max Pooling layer with pool size of (3,3) and with padding 'same'.

- Dropout regularization layer with a dropout rate of 0.25.

Second Set consists of:

- A Convolutional Layer with 32 filters of size (3,3). Chosen activation function for this layer is ‘relu’.
- A Max Pooling layer with pool size of (3,3) and with padding ‘same’.
- Dropout regularization layer with a dropout rate of 0.25.

Third Set consists of:

- A Convolutional Layer with 64 filters of size (3,3). Chosen activation function for this layer is ‘relu’.
- A Max Pooling layer with pool size of (3,3) and with padding ‘same’.
- Dropout regularization layer with a dropout rate of 0.25.

Fourth Set consists of:

- A Convolutional Layer with 128 filters of size (3,3). Chosen activation function for this layer is ‘relu’.
- A Max Pooling layer with pool size of (3,3) and with padding ‘same’.
- Dropout regularization layer with a dropout rate of 0.25.

These four sets are followed by a **Flatten** layer, because the output from the 4th set will be provided as the input to a Fully Connected layer.

Fully Connected Layer 1:

- This consists of 128 nodes and uses a ‘relu’ activation function.
- This is followed by Dropout regularization layer with a dropout rate of 0.5
-

Fully Connected Layer 2:

- This layer consists as many nodes as the number of output labels available, so in this case it consists of 3 nodes. This layer uses a ‘Softmax’ activation function. This layer is responsible for classifying the image into one of the three output labels.

Problems Faced while coming up with this implementation:

Since I had a limited set of images, I didn't want to have too many layers in my CNN. Because too many layers on too small a dataset would mean the model will just overfit. I didn't want that. Finding the balance in the number of layers, number of filters per layer were some of the challenging tasks.

I have been working on this idea for almost a month now. I knew not much about deep learning back then. I started reading up, finished the Deep Learning section of nanodegree, started writing program from simple neural nets, then read about CNNs. Different architectures, learned keras and many other deep learning techniques. So this is effort has taken me more than a month.

Refinement

While the benchmark model had 3 fully connected layers, it failed to make use of the spatial information present in the input images. By taking input the images as a vector, the model is trying to learn about 3 channel RGB images in one dimension, which throws out a lot of information. This suggests that our model has to use all the information available at its disposal in learning a function to map inputs to one of the output labels.

That's why we add convolution, pooling and regularization layers. Convolution layer uses many filters. Each filter is responsible for learning a feature. These features are made up of hyperparameters which means they'll be updated after every iteration, so they are improving too.

Pooling layers helps in achieving "Transformational invariance". Transformational invariance is detecting the object even if it has moved by some pixels from the image it has been trained on. This is a very important property in CNNs. Because when the object moves by some pixels, the Fully Connected Neural Nets fail to recognize this since the input vector to it changes.

Also in Fully Connected NN, a node is connected to every node of the previous layer. So it is tightly coupled, so any small change somewhere in the layer would affect the entire

network. Whereas in CNNs, the output is dependent only on the size of the filters thus making it loosely coupled. Hence it is more robust. So the CNN architecture is much better than the benchmark architecture.

RESULTS

Model Evaluation

These are the numbers achieved after training both the models for 100 epochs.

	Training accuracy	Training Loss	Validation Accuracy	Validation Loss	Testing Accuracy	Testing Loss
Benchmark Model	0.7804	3.5354	0.4000	9.6708	0.3333	10.7454
CNN	0.9865	0.0422	0.600	2.0413	0.5833	1.9497

CNN achieves a high training accuracy of 0.9865 which is commendable with almost negligible training loss.

Whereas on the validation set, consisting of 5 images, its accuracy is 60%, with a loss of 2.0413.

On the training set, it achieves a reasonable accuracy of 58.33% with a loss of 1.94

Justification

As it is clearly evident from the table above, CNN outperforms the benchmark model in every aspect.

Training Accuracy:

CNN outperforms the benchmark model by a good margin of 0.2

Training Loss:

There is a significant decrease in the loss of CNN when compared to the benchmark model, which is really good.

Validation Accuracy:

Again, the CNN outperforms the benchmark model with a good margin of 0.2

Validation Loss:

Here, we can see great difference between the two losses of more than 7.

Testing Accuracy:

This is a really crucial number in which the CNN outperforms the benchmark model comfortably, it's accuracy is almost the double of benchmark model.

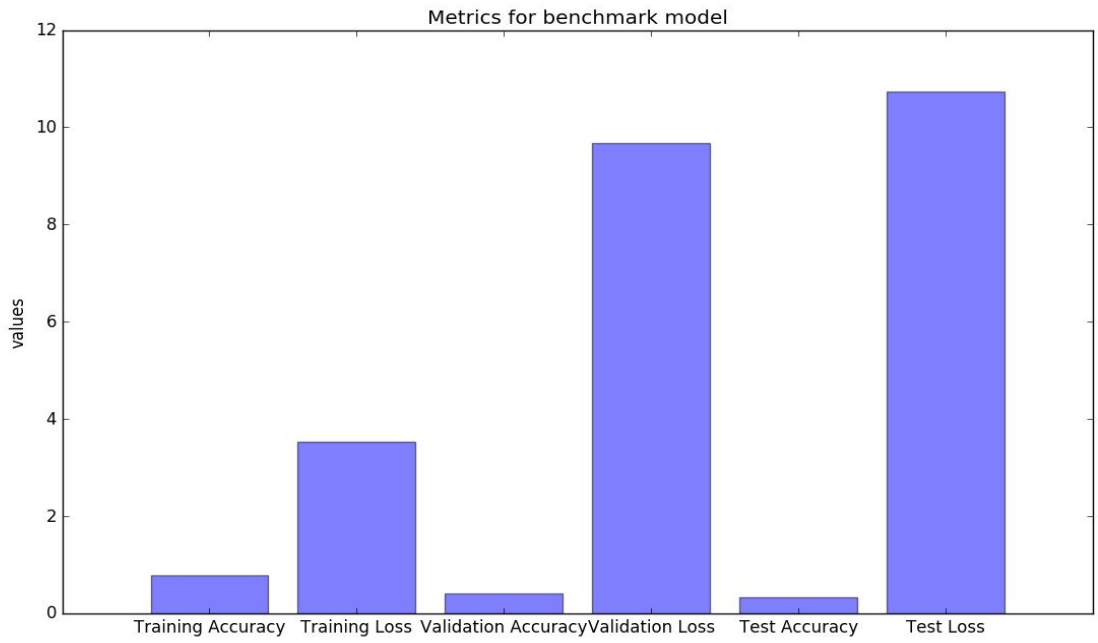
Testing Loss:

CNNs loss is 5 times better than the benchmark model's loss, which again is quite good.

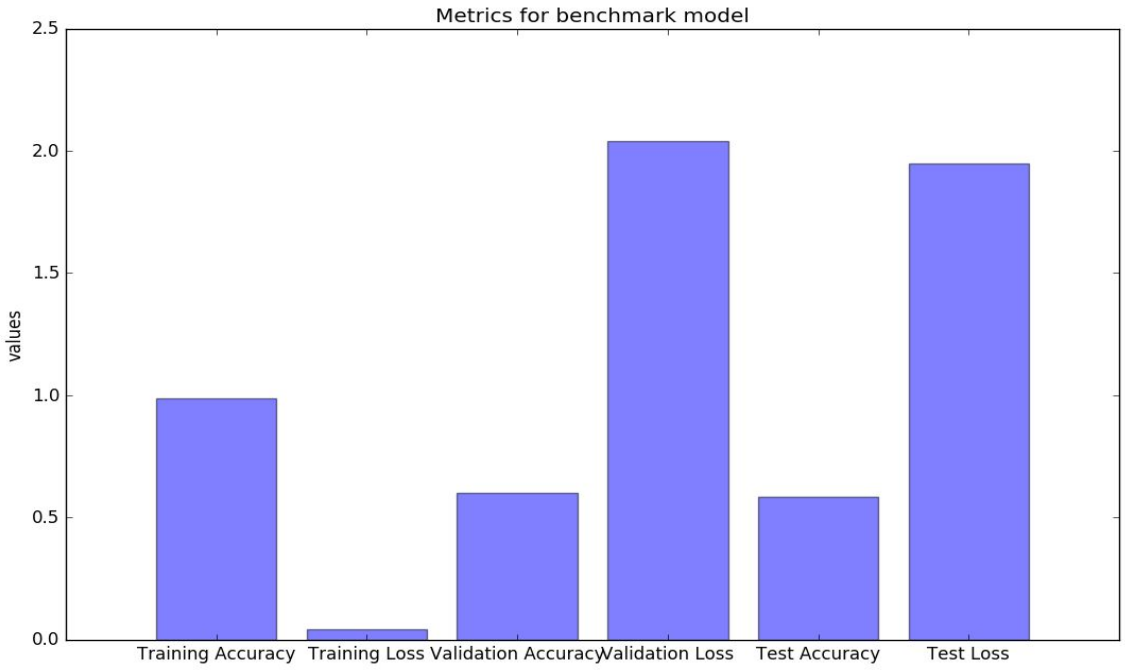
CONCLUSION

Free-form visualization

Bar Graph showing the metrics for benchmark model.



Bar Graph showing the metrics for CNN.



We can clearly see the difference in scales of y-axis in both the graphs. The bars for CNN for accuracy are way taller than benchmark model, and the bars for loss are smaller.

Reflection

- A month back when I stumbled upon this problem, I had absolutely no clue on how to approach this.
- Then I started developing an interest in Deep Learning.
- I started with simple coding simple neural nets, understanding the concepts from scratch, all that math was enjoyable.
- Once I got comfortable with the concepts, I started learning Keras. It turned out to be easier than I thought.
- Once I learnt keras, I explored different architectures that can be used for this CNN.
- The learning curve was the toughest phase of this project.

- Coming to the problem, the input is preprocessed, reshaped to a common shape, normalized and then it is fed into the CNN.
- CNN contains 4 sets of Convolution Layers, Pooling and Regularization layers.
- This is followed by 2 fully connected layers.
- The last layers contains as many nodes as the number of output classes. It is responsible for predicting the output class for a given image

Improvements

- Given the dataset contained only 200 images per class, this CNN could perform a lot better with more images, maybe with 500 images it could achieve a test accuracy of 80%. But since I could gather only close to 200 images, this could not be done.
- Adding more layers might help, but with less data I did not want to use too many layers which would result in the CNN being overfitted on the training data.

REFERENCES

- https://en.wikipedia.org/wiki/Cross_entropy
- https://brohrer.github.io/how_convolutional_neural_networks_work.html