Dank Learning : A Deep Learning Model for rating the "dankness" of Internet Memes

Megh Manoj Bhalerao (16EE234)
National Institute of Technology, Karnataka, India

1 Abstract

Initially, the title of this project may sound academically unacceptable, but, this opens a gateway to novel Machine Learning ideas that can be extended to a **plethora of online data** being uploaded on the Internet every second. This project aims at developing a deep neural network model using transfer learning for rating how funny a meme is. Basically, we train an artificial neural network model to learn **sarcasm and humor**.

2 Introduction & Motivation

A meme is an **idea, behavior, or style** (usually in the form of an Image or GIF)that spreads from person to person within a culture often with the aim of conveying a particular phenomenon, theme, or meaning represented by the meme.
Dank memes according to the urban dictionary are ones which were extremely popular at a given former period of time, and were excessively used, hence losing their humor. BUT, for our given problem, we consider dankness of a meme is **directly proportional to how funny** it is. The measure of "funniness" is given by a **floating point number between 0 and 1**.
As AI grows in leaps and bounds it requires new and challenging tasks. The **contemporary relevance of memes** and the high level of understanding required to classify them motivate this project.

3 The Dataset

Our dataset consists of around **3500 images of memes**. The memes are scraped from Reddit. Every meme has 2 parts :

1. The image *i.e.* the meme itself (rescaled to 299×299 pixel dimensions)
2. The **number of upvotes** that the meme as got on reddit - this is going to be our **target/label/ground truth** (indicating dankness of a meme) corresponding to a given image (the ground truth of all the images is stored in a text file).
   The "dankness" is normalized using the given relation

   $$D_{norm} = \frac{N_{upv}}{max(upv)}$$

   where D_{norm} is the normalized dankness, N_{upv} is the number of upvotes of that particular meme and $max(upv)$ is the maximum number of upvotes of any given meme in the given dataset.

   The normalization is done for preventing the weights of the neural network from becoming too large.

   The ground truth is stored in a db.json file which is nothing but a text file. The file contains information about each image such as: link to the image, # of upvotes of the image, caption corresponding to each image, height and width of each image, author of the meme, unique reddit ID of each meme.

   To make sure that our dataset is evenly distributed we plot a histogram of # of memes versus the # of upvotes as shown in Fig 1:

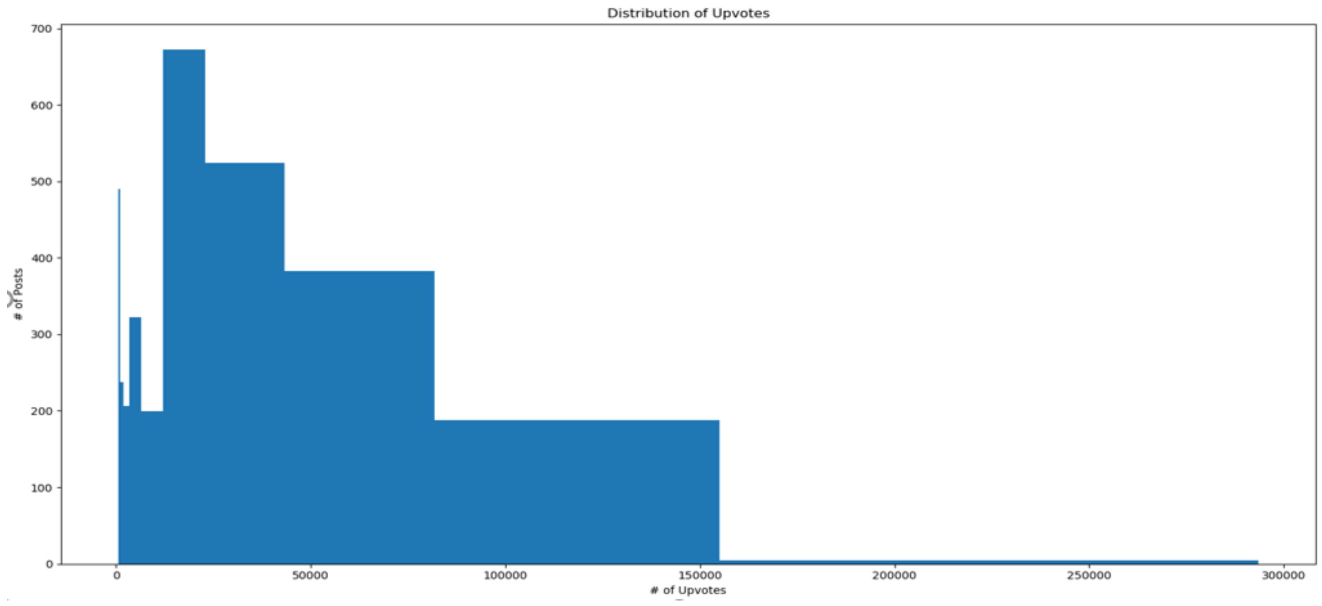

Figure 1: Upvotes Distribution

So basically our dataset is of the form:

Meme	**# of Upvotes**
Image1	Upvotes1
Image2	Upvotes 2
...	...
ImageN	UpvotesN

4 Problem Solving Approach

4.1 Transfer Learning for Data Preprocessing

Our problem is primarily an **image recognition task**. Image recognition problems are best handled by using transfer learning.

Transfer Learning is the method of using knowledge (or weights in the case of an ANN) which was gained on a previous recognition task and using/customizing it for a **new visual recognition task**.
Usually, during transfer learning we take the pretrained model, and train the model with our new dataset with initial weights taken to be that of the pretrained model.
But for our problem, we will not update the weights of the pretrained model but rather use it just to extract **feature vectors** corresponding to every image, which will be then fed to the next stage of our neural network.

4.2 Feature Extraction using a CNN

We use the **Google Inception V3 Deep Neural Network** (runner-up CNN of 2016 ImageNet challenge) for the task of feature extraction. The architecture if the inception network is **extremely complex and hence we do not try to analyze but rather just use it for extracting features**. This network was trained on the ImageNet dataset which consists of about 1.2 Million images of 2048 different categories. This CNN achieved an accuracy of about 95% in the ImageNet challenge. The architecture is shown below just as a reference (we are not going to modify it in any way).

Other state of the art image classification CNNs include the VGG16/VGG19 - By Visual Geometry Group at Oxford, Residual Network (ResNet) By Microsoft Research, XceptionNet By Google AI etc.

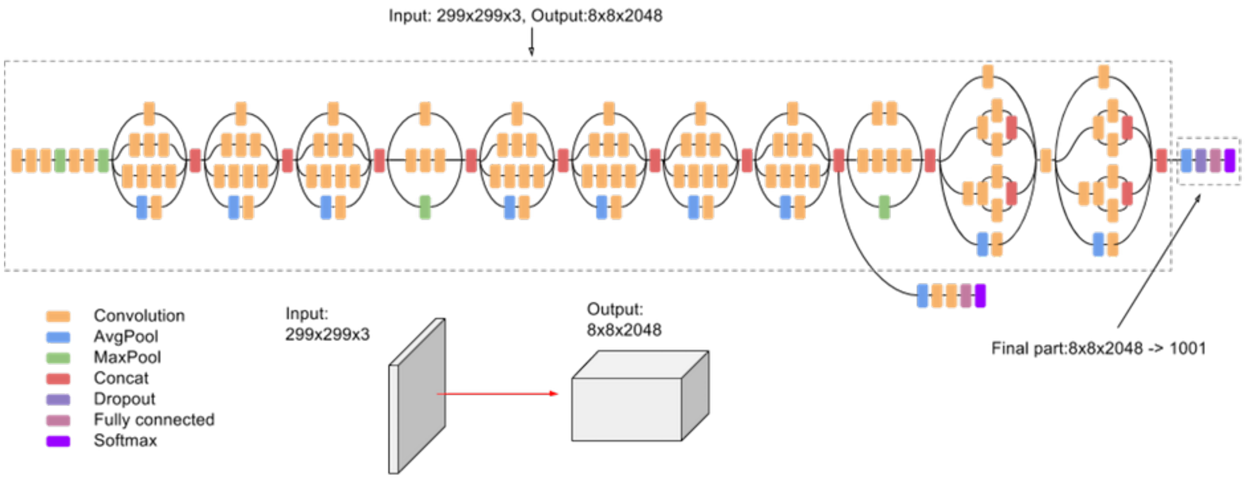

Figure 2: Inception V3 Architecture

The input of InceptionNet is a 299×299 Image and the output is a 1×2048 feature vector. Now, this 1×2048 **feature vector** is the input to our **custom neural network** that is to be trained from scratch. Our custom network will just a simple **Multi-Layer Perceptron network**, and hence will not be computationally expensive.

4.3 Harwdare Acceleration

Since the Inception Network is extremely complex, even running a simple forward pass through it (feature extraction) requires large computational resources like GPU(Graphics Processing Unit) or TPU(Tensor Processing Unit). Hence, I have used the newly launched **Google Collaboratory** which provides up to 12 hours of free GPU computation per day on the Tesla K80. The **Tesla K80** looks something like :

Figure 3: Tesla K80

4.4 Custom Network Architecture

After extracting features of all Images and saving them in a **single feature matrix**, this becomes our training dataset along with the normalized upvotes.
Basically, our problem becomes a regression model with an input of 2048 dimensions and a single dimensional output.

This regression model is tackled using a multi layer perceptron model (**simple feed forward neural network with backpropagation**). Our custom model has the following architecture :

- **Input** : 2048×1 feature vector from InceptionNet
- **Output**: 1×1 single floating point number between 0 and 1

The network architecture is illustrated below:

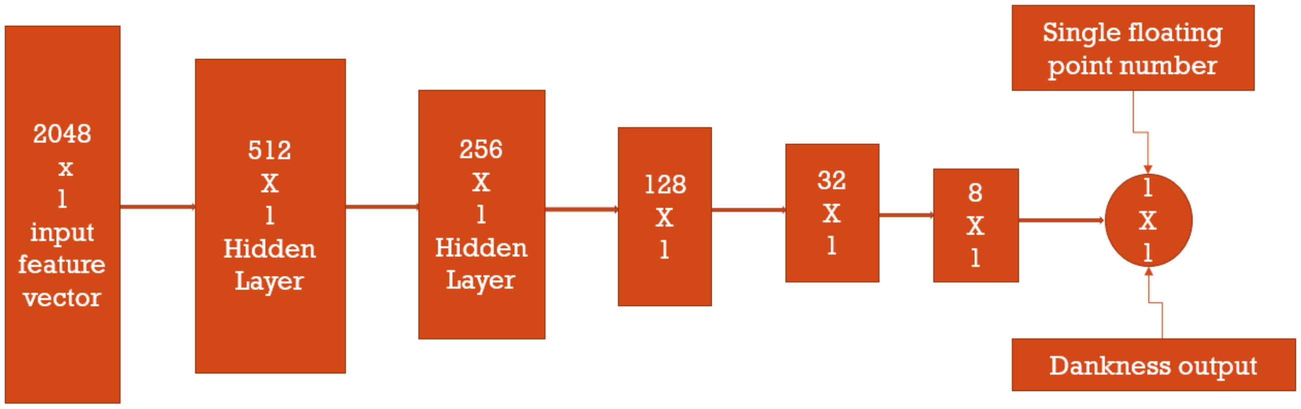

Figure 4: Custom Network Architecture

Between every hidden layer and the next there is a **dropout layer** which is an extremely good measure to prevent overfitting of the model, especially since our model contains a large number of parameters.
The concept of a dropout layer explained briefly below:

- Every dropout layer is characterized by a **dropout parameter** say $k \, \epsilon \, (0, 1)$.
- The dropout randomly renders neurons of a given layer **inactive** with the probability k.
- The dropout layer **reduces co-dependency** of the neurons amongst each other and hence reduces chances of over-fitting of the data.

4.5 Training our Custom MLP model

The training of the MLP network is done on the local CPU since it isn't computationally expensive. The neural network weights are randomly initialized using a **random normal distribution function**. A **simple squared error loss** is used since we have only one output number. The loss function is given below:

$$L(x, y) = (x - y)^2$$

where L is the loss function which needs to be minimized. x is the **output** of the network and y is the **target/ground truth** corresponding to the feature vector.
Backpropogation technique is used for weight updation whose equation is given by :

$$w_i := w_i + \frac{\partial L(x, y)}{\partial w_i} \times w_i$$

where w_i is the weight of any arbitrary connection of the MLP model.
The loss function is plotted for **100 epochs** to show clearly how the loss decreases as the number of epochs increases.

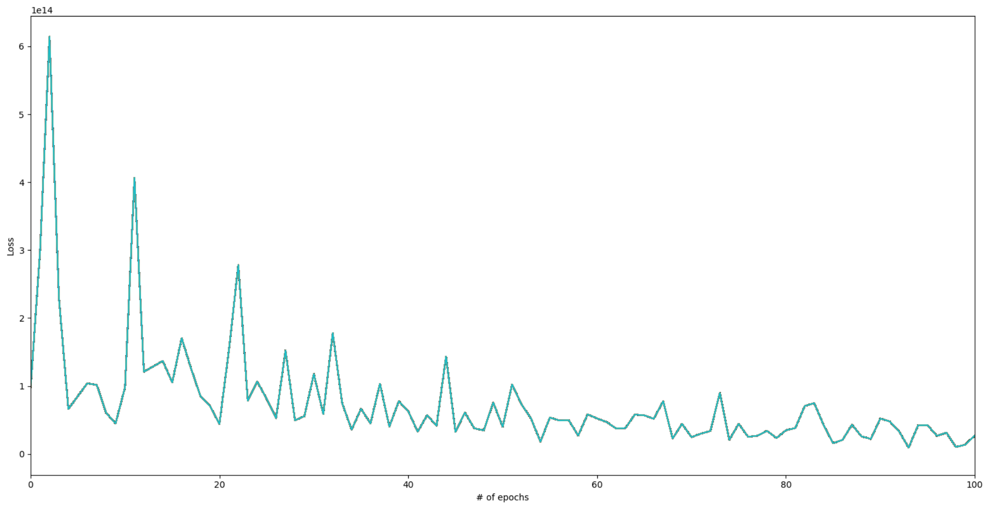

Figure 5: Loss vs # of Epochs for 100 epochs

5 Results

Our problem is essentially a **regression problem** (in addition to being a highly subjective one) and hence there is no "accuracy" measure as such. Hence, the measure of how well our model performs is to monitor the loss function as the number of epochs increases. Ideally, the **loss must eventually decrease** as the number of epochs increase. The tabular column given below shows the same :

# of Epochs	10	100	500	1000
Loss Function Value	105743450243072.0	1120687035136.0	447666695.75	9383.2

6 Scope for Future Work

Presently the ANN model was trained on only raw meme images. This can be extended by training the model with **images and captions corresponding to every meme** (maybe using **natural language processing** models). This multimodal method of training is expected to give far better results than training only on images, as it is more practical and contains way more information. Also, weight updates were carried out only on the Multi-Layer Perceptron part of the ANN. In the next step, we plan on doing a complete pass i.e. **updating the weights of the InceptionNet** (with initial weights equal to weights when trained on ImageNet), but this would require high computational resources.

7 Applications of the problem statement

The present idea can be **extended to many other forms of online data**. For example, automatic removal of abusive or hate content for online platforms such as Facebook, Instagram, Twitter etc. This can be done by training our model on other types of datasets such as **Facebook, Instagram and Twitter posts**.

8 References

As this problem statement is completely Novel, the references are quite limited and are listed below :

1. "Dank Learning : Generating Memes using Deep Neural Networks" Peirson AL, Tolunay EM Stanford University
2. "Classification of Internet Memes" Kolawole, Olamide Temitayo University of Texas at Austin
3. Official Tensorflow Website

9 Appendix - Code for Model Architecture and Training

```
import tensorflow as tf
import numpy as np
from PIL import Image
import tensorflow_hub as hub
import os
from tensorflow.python.keras import backend as K
inception_feat = hub.Module("https://tfhub.dev/google/imagenet/inception_v3/
    feature_vector/1", trainable=True)
import matplotlib.pyplot as plt


n_nodes_hl1 = 512
n_nodes_hl2 = 256
n_nodes_hl3 = 128
n_nodes_hl4 = 32
n_nodes_hl5 = 8

batch_size = 100

loss_list = []
n_classes = 1
keep_prob = 0.2
x = tf.placeholder('float',[None,2048])
y = tf.placeholder('float')
# normal.txt is the text file containing normalised dankness
num_lines = len(open('Normal.txt', "r").readlines())
idx=0
mega_feat = np.random.rand(num_lines, 2048)
train_y = np.loadtxt('Normal.txt', dtype= 'float32', comments='#', delimiter='\n',
    converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='
    bytes')
train_x = mega_feat
print(mega_feat)
def neural_network_model(data):
hidden_1_layer = {'weights' : tf.Variable(tf.random_normal([2048, n_nodes_hl1])), '
    biases' : tf.Variable(tf.random_normal([n_nodes_hl1]))}
hidden_2_layer = {'weights' : tf.Variable(tf.random_normal([n_nodes_hl1,
    n_nodes_hl2])), 'biases' : tf.Variable(tf.random_normal([n_nodes_hl2]))}
hidden_3_layer = {'weights' : tf.Variable(tf.random_normal([n_nodes_hl2,
    n_nodes_hl3])), 'biases' : tf.Variable(tf.random_normal([n_nodes_hl3]))}
hidden_4_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl3, n_nodes_hl4
    ])),'biases': tf.Variable(tf.random_normal([n_nodes_hl4]))}
hidden_5_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl4, n_nodes_hl5
    ])),'biases': tf.Variable(tf.random_normal([n_nodes_hl5]))}

output_layer = {'weights' : tf.Variable(tf.random_normal([n_nodes_hl5, n_classes]))
    , 'biases' : tf.Variable(tf.random_normal([n_classes]))}

l1 = tf.add(tf.matmul(data,hidden_1_layer['weights']), hidden_1_layer['biases'])
l1 = tf.nn.relu(l1)
l1 = tf.nn.dropout(l1, keep_prob)

l2 = tf.add(tf.matmul(l1,hidden_2_layer['weights']),hidden_2_layer['biases'])
l2 = tf.nn.relu(l2)
```

```
l2 = tf.nn.dropout(l2, keep_prob)

l3 = tf.add(tf.matmul(l2,hidden_3_layer['weights']), hidden_3_layer['biases'])
l3 = tf.nn.relu(l3)
l3 = tf.nn.dropout(l3, keep_prob)

l4 = tf.add(tf.matmul(l3, hidden_4_layer['weights']), hidden_4_layer['biases'])
l4 = tf.nn.relu(l4)
l4 = tf.nn.dropout(l4, keep_prob)

l5 = tf.add(tf.matmul(l4, hidden_5_layer['weights']), hidden_5_layer['biases'])
l5 = tf.nn.relu(l5)
l5 = tf.nn.dropout(l5, keep_prob)

output = tf.matmul(l5,output_layer['weights']) + output_layer['biases']

return output


saver = tf.train.Saver()
def train_neural_network(x):
prediction = neural_network_model(x)
#cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=prediction
    ,labels=y))
cost = tf.losses.mean_squared_error(y,prediction)
#learning rate = 0.01
optimizer = tf.train.AdamOptimizer().minimize(cost)

hm_epochs = 10;

with tf.Session() as sess :
sess.run(tf.global_variables_initializer())


for epoch in range(hm_epochs):
epoch_loss = 0
i = 0
while i < num_lines :
start = i
end = i + batch_size
batch_x = np.array(train_x[start:end])
batch_y = np.array(train_y[start:end])
_, c = sess.run([optimizer, cost], feed_dict={x: batch_x, y: batch_y})
loss_list.append(c)
plt.plot(loss_list)
epoch_loss += c
i += batch_size
print('epoch', epoch + 1, 'completed out of', hm_epochs, 'loss:', epoch_loss)
saver.save(sess, r'\H:\MEGH\NITK\Third Year - B.Tech NITK\DankNotDank\src\
    my_test_model.txt')


train_neural_network(x)
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.show()
```