

**Training a machine learning classifier to predict the intrinsic popularity of an image on
Flickr, an image hosting service**

Alikhan Murat

2021

Contents

1	Abstract	3
2	Glossary	3
3	Important terms in equations	4
4	Introduction.....	5
5	Intro to deep learning.....	6
5.1	What is a Deep Neural Network (DNN)?	6
5.2	Convolutional Neural Networks (CNNs).....	8
5.3	The loss function and gradient descent	11
5.4	ResNets, normalization, and other methods for improving model accuracy	15
5.4.1	Normalisation.....	15
5.4.2	Solutions to the overfitting problem	16
5.4.3	Residual Networks and Transfer Learning	17
6	My implementation explained	19
6.1	Downloading the dataset.....	19
6.2	Image preparation	23
6.3	Training.....	27
6.4	Metrics	30
7	Conclusion	31
8	Table of figures	32
9	Bibliography	33

I hereby declare that the content of this project is my original work (unless referenced clearly to the contrary) AND that no element of the work included in the report has been submitted for any other qualification.

1 Abstract

Image popularity prediction is an interesting subset of the field of image quality assessment where images are assessed on their viral potential instead of on objective metrics such as noise and distortion. Due to the subjective nature of viral potential as a metric for image quality, I decided to use deep learning to tackle this problem. A method for reliably predicting an image's popularity is useful for both companies seeking to advertise their products on social media platforms and the social media platforms themselves. The first group would benefit from improved reach, while the latter could improve the quality of the content presented to users to keep them engaged for longer. I will use transfer learning to modify the pre-trained ResNet50 model as it was effective at solving this issue in other papers (Ding et al., 2019). By grouping the images from the “SMP Challenge: An Overview of Social Media Prediction Challenge” (Wu et al., 2019) dataset I was able to train a classifier model that rated the model on a scale from 1 to 3. Since the model was able to achieve a validation accuracy of 41% after less than 2 hours of training, this method has proved to be viable for making classifications that are significantly better than random.

Keywords: Machine Learning, Image Popularity Assessment, Transfer Learning.

2 Glossary

- **Machine Learning:** The science of developing methods for machines to optimise their solutions to a problem based on given data.

- **Regressor:** A machine learning model that returns a continuous number (e.g., a popularity score or a predicted height).
- **Classifier:** A machine learning model that returns a set of probabilities that are used to assign the input to a class (e.g., a facial recognition model)
- **Deep Neural Network (DNN):** A machine learning model that is comprised of multiple layers between the input and the output.
- **Convolutional Neural Network (CNN):** A class of DNN specialised for images.
- **Training Set:** The set of images that the model is trained on.
- **Validation Set:** The set of images that the model hasn't seen used to evaluate its real-world performance.
- **Epoch:** One full training step. The model has seen all the training images.
- **Batch:** A small group of images that are fed into the model at once.
- **Loss:** A measure of how poorly a model is doing. Higher loss equates to a worse model.
- **Loss Function:** A function that calculates the loss given predictions and the labels.
- **Features:** Every group/column of numbers (e.g. house area) responsible for the prediction.
- **Weights:** The coefficients of the features.
- **Biases:** The term added on to the end such that the prediction is not 0 if all the features are 0.
- **Fitting:** The act of optimising the weights and biases of a model to give the lowest loss.
- **Optimiser:** A class responsible for adjusting the weights and biases based on derivatives of the loss function.

3 Important terms in equations

- | | |
|---------------------------|---|
| • x_i : The features | • \hat{y}_i : The labels (true values). |
| • y_i : The predictions | • w_i : The weights |

- b_i : The biases
- X : The matrix of feature values
- W : The matrix of weights
- Y : The matrix of predictions
- b : The matrix of biases
- ω : The width of the filter
- h : The height of the filter
- c : The number of channels
- $L(X)$: The loss function
- l : The learning rate
- \bar{t}_i : The average pixel value for the i th channel
- \hat{t}_i : The real pixel value for the i th channel
- σ_i : The standard deviation of the i th channel

4 Introduction

This paper aims to explain the code that I used to train a ResNet-50 model to classify images into groups based on their popularity on Flickr. The first half of the paper aims to explain the fundamentals of machine learning. I believe that this section is a necessary part of the report since the information provides a much deeper insight into some of my decisions about the model.

Should you wish to skip this section, the details of my implementation can be found in the section 6. This section breaks down the actual code I wrote to train my model. Note that since I used prebuilt packages as a time-saving measure, a lot of the concepts mentioned in the first section do not appear directly.

In the final section, I will evaluate the performance of my model using the data I gathered while testing. This will include insights into the validation accuracy and loss of the model as well as how these metrics compare with the training accuracy and loss. Furthermore, I conducted a hypothesis test to confirm if I had been able to achieve better than random results, I based the success of the model on this test.

5 Intro to deep learning

5.1 What is a Deep Neural Network (DNN)?

To understand what a deep neural network is, it is useful to first know the principles behind linear regression. The simplest example of linear regression is one where there is only one feature. As an example, we can think of the relationship between a house's size and its price, a fictitious plot of what this data might look like can be seen below.

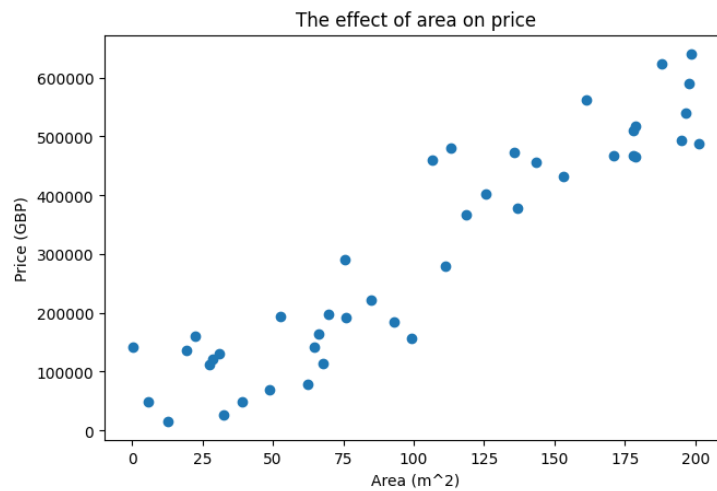


Figure 1

The relationship between the area and the price is roughly linear. Therefore, since there's only one feature, we can estimate this relationship by using the equation $y_1 = w_1x_1 + b_1$. In this scenario, y_1 is referred to as the label, w_1 is known as the weight, x_1 is a feature (the house price), and b_1 is known as the bias. Therefore, by picking an appropriate weight and bias, we can find a line of best fit for the data above.

In the scenario where we have multiple labels and multiple features, we would have multiple linear equations. As an example, let us assume that we want to estimate the number of

residents in a house on top of the price and that we use the number of bedrooms, the number of bathrooms, and the area of the house to do this. As a result, we get this pair of equations:

$$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \quad (1)$$

$$y_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2$$

Here y_1 and y_2 represent the two labels (the number of residents and the house price) while x_1 , x_2 , and x_3 are the feature values (the number of bedrooms, number of bathrooms, and the area of the house).

However, if we wanted to input many feature values to get many predictions, keeping track of the data can become difficult if they are stored as independent variables, especially if we had many more feature types and labels. Therefore, it makes sense to represent the training data, the weights, the biases, and the labels as multidimensional matrices. The operation above is equivalent to:

$$X \times W^T + b \quad (2)$$

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}$$

This has the same effect as inputting the feature values in X into equations (1) and returns a $n \times l$ matrix of predictions where n is the number of data points whose feature values were inputted and l is the number of labels. Tensor multiplication is also significantly faster in practice than the scalar multiplication used in equations (1) since they can be run on GPUs which are purpose-built for fast matrix multiplication.

We can use this to define a linear layer in a neural network as the function below:

$$F(X) = X \times W^T + b \quad (3)$$

In the equation, $n \times f$, $l \times f$, and $n \times l$ are the sizes of matrices X , W , and b respectively. n represents the number of data points fed into $F(X)$, f is the number of features, and l is the number of labels.

By stacking these linear layers and feeding the outputs from one into another, a deep neural network is created. The linear layers between the input layer and the final output layer are known as the hidden layers. To train this neural network, we would use gradient descent (see section 5.3) to find the weights and biases for each layer that would minimize the discrepancy between the predictions of the neural network and the true labels.

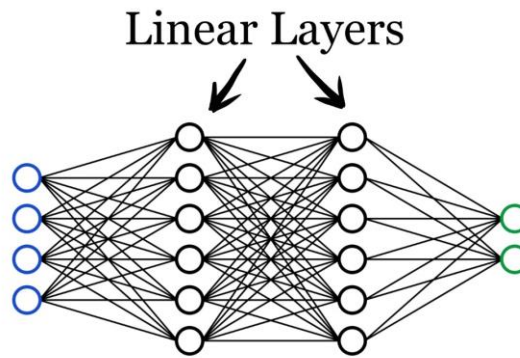


Figure 2 (“Neural Networks From Scratch - victorzhou.com,” n.d.)

5.2 Convolutional Neural Networks (CNNs)

DNNs have a major flaw that makes them unsuitable for analyzing high-resolution images. Since all the layers are fully connected, DNNs become very computationally expensive. The current time complexity for multiplying two $n \times n$ matrices is $O(n^\omega)$ where the upper bound of ω is 2.37286^1 (Alman and Williams, 2020). Since I will be using 224×224 colour images, the

¹This means that in the worst-case scenario, if you multiplied two matrices that were 2 times larger, it would take $2^{2.37286}$ times as many operations.

input layer would have a size of $224 \times 224 \times 3 = 150528$ making the tensor multiplication significantly slower if I used fully connected layers. As a result, the training process would take too long if I used a DNN to determine image popularity.

The obvious solution is to reduce the number of connections in each layer. This can be done using convolutional layers in which nodes are only connected to neighboring nodes in the previous layer. These were first introduced by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998) in their LeNet-5 architecture.

In the context of an image and one convolutional layer, the neurons are arranged in a 2-dimensional grid. Every neuron in row x and column y of the convolutional layer is connected to rows from x to $x + \omega - 1$ and columns from y to $y + h - 1$ where ω and h are the dimensions of the filter (also known as a convolutional kernel), this region is known as the receptive field of the neuron (Géron, 2019, p. 434). This can be seen in Figure 3. The value of the neuron is determined by applying a filter to the values of the pixels in the neuron's receptive field. The filter is a grid of values, each filter value is multiplied by the corresponding pixel value in the receptive field, the value of the neuron is the average of these products. The same filter is used to find the value of each neuron in the convolutional layer.

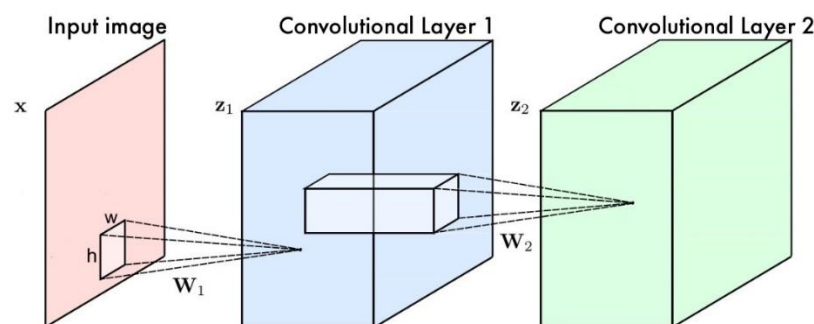


Figure 3: A CNN. By Renanar2 - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=75218608>

Each filter extracts a different feature from the input image (e.g. horizontal or vertical edges); therefore, by applying many different filters to the same input image we can get many output channels. Further convolutions are then applied to these layers creating more channels. This can also be seen in Figure 3. Note that these initial filter values are random and aren't designed to look for specific features; however, after the training process, the values change to detect features that best improve the model's accuracy.

There is also a further pooling layer that decreases the size of each channel, an example of this is the MaxPool2D layer that is a $n \times n$ grid that tiles the image without overlap and decreases the image size by only picking the largest image in the grid. After the images have been shrunk to a much smaller size (often $1 \times c$ where c is the number of channels), the pixel values can then be flattened into a 1-dimensional array and be fed into a fully connected layer discussed in the previous section. The values returned by this layer are the ones used for classification.

Since the filters tend to be quite small, and that one filter is used on the entire image, the number of calculations needed to perform a forward pass (the process of inputting an image to get classifications) is significantly smaller. This allows for increased network depth² which tends to increase model accuracy (Simonyan and Zisserman, 2015). The effect of model depth can be seen in Figure 4 where the loss³ decreases as you increase model depth from 16 layers to 50 layers.

² The total number of layers in the neural network.

³ A measure of model performance

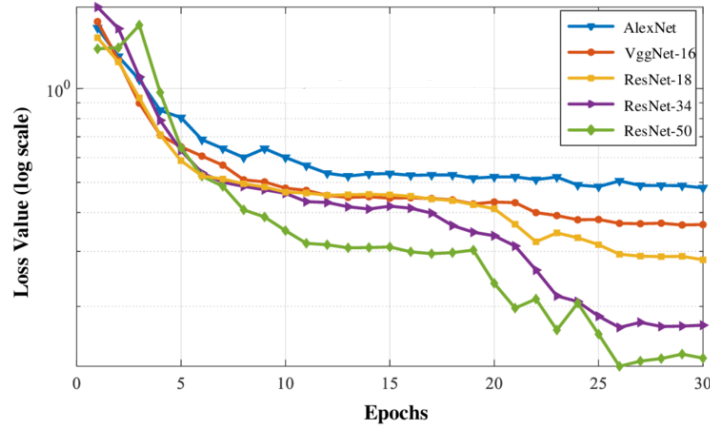


Figure 4: (Talo et al., 2019) The number beside the name of the model represents the total number of layers present.

In conclusion, due to the nature of CNNs, they require a reduced number of connections between neurons resulting in them requiring less computing power. As a result, this makes them very well suited for classifying higher resolution images; therefore, I decided to use them in my project as opposed to DNNs.

5.3 The loss function and gradient descent

The previous sections only detail what happens during the forwards pass of the model. Therefore, without training the outputs you get from the model are still just random. In this section, I will explain how models are trained via a process called gradient descent and use this explanation to justify why accuracy isn't a suitable metric for model training and why loss functions such as cross-entropy loss (the loss function I use in my model) are better suited for this task.

For the sake of simplicity, I will go back to the example mentioned in section 5.1 where we were trying to predict house price based on the floor space by using a linear regression model, this means there is only one feature (y_1), one weight (w_1), and one bias (b_1). The initial value for the weight and the bias are chosen randomly resulting in a line like the one in Figure 5.

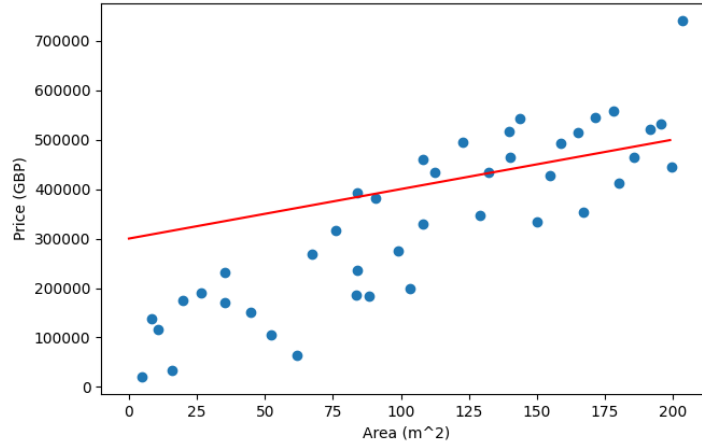


Figure 5: A random line.

This line is not optimal. To assess the performance of the linear regression model at this instance we can use a loss function. This function is simply a multivariable function that is defined by the programmer to measure the performance of the model. In this case, we can use Mean-Squared-Error loss which is defined in equation (4) (“MSELoss — PyTorch 1.7.1 documentation,” n.d.).

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (4)$$

n = number of datapoints

\hat{y}_i = true value

y_i = predicted value

Since the values of \hat{y}_i are provided by the training set, they are constants. As a result, the loss function is a multivariable function $L(w_1, b_1)$ as the value of the loss depends on the weight and the bias instead of on the values of x . We can take a slice of this curve by choosing a fixed b_1 , and then plotting the loss against the weight we can get a graph like the one in Figure 6.

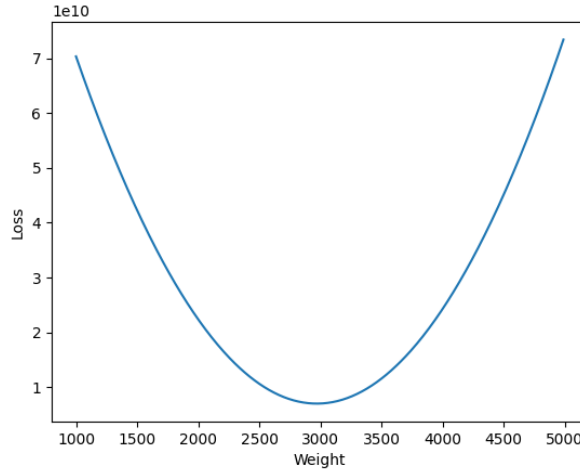


Figure 6: A plot of the loss function.

The goal is to find the weight that minimizes the loss giving the best possible gradient for the points provided. This is done via a process called gradient descent, which involves “following” the gradient to the minimum. In this example, notice that to the left of the minimum the gradient is always negative and that to the right of the minimum the gradient is always positive. Therefore, we know our position relative to the minimum based on the gradient $\frac{\partial L}{\partial w_1}$. We can use this to create the equation for the new weight that is closer to the minimum w_{new} :

$$w_{\text{new}} = w_{\text{old}} - l \times \frac{\partial L}{\partial w_1} \quad (5)$$

So, by changing l we can control how fast we arrive at the minimum, thus it is called the learning rate. However, if the learning rate is too high the model will fail to converge at the minimum and will simply overshoot; therefore, it needs to be balanced to optimize the training time and the loss. We apply this same idea to optimize b_1 , at every step, we move both b_1 and w_1 closer to their minimums until eventually, the line of best fit is found. This optimization method is called Batch Gradient Descent (BGD) (Géron, 2019, p. 123).

The optimizer I use is Stochastic Gradient Descent (SGD) with momentum which converges faster and works similarly to BGD. Although it has some differences the key similarity between all optimisers is that they depend on the loss function being differentiable to be able to compute gradients. This highlights why accuracy is not a suitable loss function, it is not differentiable.

In a classifier model, predictions are made by having multiple outputs each of which represents a certain class. For example, a cat and dog classifier would have two output nodes, one representing cats and the other representing dogs. After feeding an image into the classifier it would return two values that are converted into probabilities by the softmax function. To measure a model's accuracy, the maximum of two is chosen as the prediction which is then compared to the true value to see if the model made a correct prediction. As you can see, accuracy requires the use of both the max function and a Boolean operator; therefore, it cannot be differentiated and is an unsuitable loss function. However, I still calculate it as it is arguably the most intuitive metric for humans to understand model performance.

For this reason, I will be using cross-entropy loss instead which is defined for each probability tensor Y and label tensor \hat{Y} in equation (6). The label tensor has the same dimensions as the output predictions tensor and each index represents a binary class. Using the aforementioned cats and dogs classifier as an example, if $P(\text{Image} = \text{cat})$ is in index 0 of the output tensor and $P(\text{Image} = \text{dog})$ is in index 1, the label for a dog would be $[0,1]$. I chose this loss function since I am making a binary classification⁴. Furthermore, not only is this function differentiable allowing for gradient descent, but it also gives me and the optimiser a sense of how wrong the model is.

⁴There is only 1 class that an image can be in, it cannot be both popular and unpopular at the same time.

This is because if y , one of the values in the prediction tensor, is close to \hat{y} , the corresponding real value, the value of the loss would be small and if they were far apart the value for the loss would be larger.

$$L = -(\hat{y} \log(y) + (1 - \hat{y}) \log(1 - y)) \quad (6)$$

5.4 ResNets, normalization, and other methods for improving model accuracy

In this section, I will be discussing other techniques that I either directly or indirectly utilised in my model to improve the training process (Section 5.4.1) or to improve the model's performance on the validation set (Section 5.4.2).

5.4.1 Normalisation

I normalised the data both before and during training. In the first part, I normalised the data by first using min-max normalisation⁵ and then calculating a mean \bar{t}_i and standard deviation σ_i for each channel i to normalise the data by organising it in a normal distribution centred at 0 using equation (7), in which t_i is the new value and \hat{t}_i is the old value of each pixel. I discuss the details of how I implemented this with code in section 6.2.

$$t_i = \frac{\hat{t}_i - \bar{t}_i}{\sigma_i} \quad (7)$$

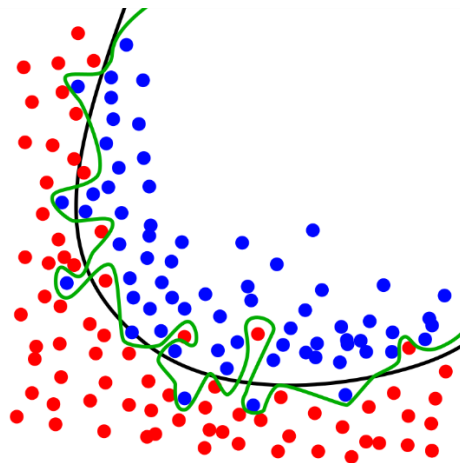
During training, I used batch normalisation, a feature that is already built into the ResNet50 model I use for transfer learning (He et al., 2015). This leads to more stable training by preventing the exploding and vanishing gradient problems (Ioffe and Szegedy, 2015). This would cause the training process to fail due to the gradients getting too large or small for the optimiser to train the

⁵This involves mapping the smallest value to 0 and the largest to 1, other values are mapped linearly in between. Since I'm using images this can be simplified to dividing all the values by 255.

model. Furthermore, it leads to faster training, “Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps” (Ioffe and Szegedy, 2015, p. 1). This fact contributed to me choosing the ResNet50 model as my base for transfer learning.

5.4.2 Solutions to the overfitting problem

Overfitting is a problem that occurs during model training that results in the model being less generalisable. The cause behind the problem is that the model is so complex that it can perfectly fit the training data; therefore, instead of spotting an overarching pattern in the data, the model simply tries to attempt to fit the training data perfectly (Géron, 2019, p. 28). An example of this can be seen in Figure 7.



*Figure 7: By Chabacano - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=3610704>*

In the context of my image popularity assessment model, my model would be overfitting if it started to simply remember which images were popular and which ones were unpopular, thus it would create the illusion that it understands which images are popular. However, this problem is

quite easy to spot if you use a validation set on which you evaluate the model after every epoch⁶. If the model's training loss is decreasing while the validation loss is increasing, then the model is overfitting since it is fitting too closely to the training data meaning that it no longer performs well on the validation data.

There were built-in features in the ResNet50 model that prevented overfitting. The main features were dropout layers which randomly disabled some of the weights by setting them to 0 for each batch. This results in the model not relying too heavily on particular connections to make decisions; therefore, it attempts to train all of the weights to recognise which images were popular equally.

However, this was not enough; therefore, to further prevent overfitting, I randomly cropped the training images (for more details see section 6.2) (Nelson, 2020). This meant that the model couldn't simply remember what the images looked like since they would change slightly after every epoch. Therefore, this helped to prevent overfitting and had the same effect as increasing the size of the dataset.

5.4.3 Residual Networks and Transfer Learning

Residual networks are neural networks that utilise skip connections to improve model performance. A skip connection is one where a copy of the data “skips” some layers and is then added back to the data that went through the layers (see Figure 8). The model that I used is ResNet50, a 50 layer residual CNN that was used in one of the papers I read (Ding et al., 2019).

⁶ An epoch is one training step. This involves updating all the model's weights once using the whole training dataset.

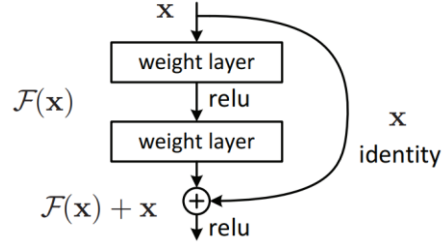


Figure 8 (He et al., 2015)

The architecture of the ResNet-50 model can be seen in Figure 9. Each block represents a residual unit, the number on top represents the number of times each unit appears. The filter size and the number of channels for each convolutional layer are in the form “ $k \times k, n$ ” within each block.

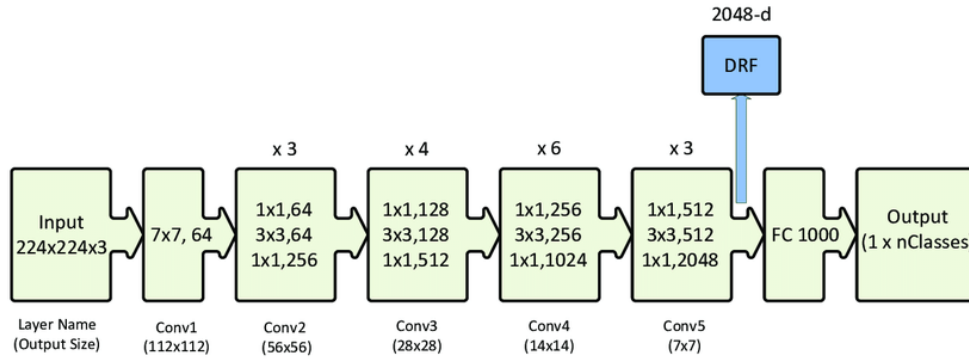


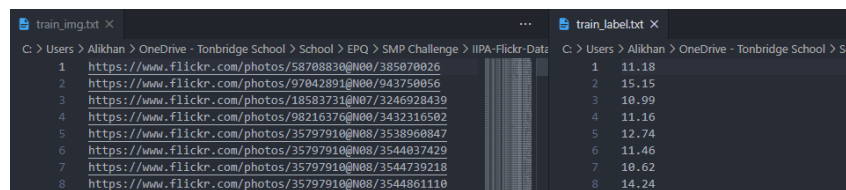
Figure 9: (Mahmood et al., 2020)

I used the ResNet50 model that was pre-trained on the ImageNet dataset for transfer learning. This theoretically sped up training times since the model could use some of the patterns it used for object recognition on the ImageNet dataset to determine whether an image was popular or not. The paper that inspired this used this pre-trained model to perform regression and find a score for the image (Ding et al., 2019); however, I decided to use it as a classifier to classify images into rating groups ranging from 1 to 3 since the pre-trained ResNet50 model was initially used in a similar way.

6 My implementation explained

6.1 Downloading the dataset

Before I could start the training or validation process, I had to download the data that I would be using to train my model. I used the images from the “SMP Challenge: An Overview of Social Media Prediction Challenge”. These were in the form of two text files, one contained image URLs and the other contained their popularity score, the first 8 rows of the dataset can be seen in Figure 10.



train_img.txt	train_label.txt
1 https://www.flickr.com/photos/58708830@N00/385070026	1 11.18
2 https://www.flickr.com/photos/97042891@N00/943750056	2 15.15
3 https://www.flickr.com/photos/18503731@N07/3246928439	3 10.99
4 https://www.flickr.com/photos/98216376@N00/3432316502	4 11.16
5 https://www.flickr.com/photos/35797910@N08/3538960847	5 12.74
6 https://www.flickr.com/photos/35797910@N08/3544037420	6 11.46
7 https://www.flickr.com/photos/35797910@N08/3544739218	7 10.62
8 https://www.flickr.com/photos/35797910@N08/3544861110	8 14.24

Figure 10: Image URLs and popularity scores

However, I couldn't find a way to download the images using the URLs that they provided; therefore, I had to convert them to image URLs of the form:

“https://live.staticflickr.com/{server-id}/{id}_{secret}_{size-suffix}.jpg”

(“Flickr Services,” n.d.)

This was done by parsing the URLs to extract the photo ids of the images as shown in the code below:

```
#A function that extracts photo ids from the train_img.txt file.
def read_ids(urls_loc = "train_img.txt"):
    ids = []
    with open(urls_loc, "r") as txt_file: #Opens the old URLs file
        lines = txt_file.readlines() #Adds each line as an item in the list lines
        lines = [line[:-1] for line in lines] #Gets rid of new line (\n) character
    for line in lines:
        line = line[::-1] #reverses the line
        end = line.index("/") #finds the first /
        pid = line[:end] #Gets the reversed photo id
        ids.append(pid[::-1]) #Adds the unreversed photo id to ids
    return ids
```

Using each photo id, I had to use the Flickr API to find out the “server-id” and “secret” of each photo. This was done using the following code:

```
#Function that gets the image URL when given a photo id and an image size suffix
def get_url(pid, size = "w"):
    #Gets all the information about a photo as a dictionary
    photo = flickr.photos.getInfo(photo_id = pid)
    secret = photo["photo"]["secret"] #Extracts the secret from the information
    server = photo["photo"]["server"] #Extracts the server from the information

    #Combines the extracted information into a URL and returns it
    url = f"https://live.staticflickr.com/{server}/{pid}_{secret}_{size}.jpg"
    return url
```

I then used this “get_url” function on the entire dataset of photo ids and implemented some more features. The main feature was removing the labels from the “train_label.txt” file that were used with broken images. These were images that were probably removed from Flickr and as a result, I couldn’t generate a URL for them.

```
pids = read_ids() #List of all photo ids (pids)

for i, pid in enumerate(pids[:50000]): #Iterates over the first 50000 photo ids.
    try: #Attempts to get the images URL
        url = get_url(pid)
        print(f"{i} converted")
    #If it fails to get the URL it adds the index of the URL to the bad_indicies array.
    #This is so that the corresponding label can be removed from the train_label.txt file.
    except:
        bad_indicies.append(i)
        print(f"{i} broken")
        continue
    #If there are no errors it appends the url to the urls array
    else:
        urls.append(url)

#Creates a train_label.txt file where the labels for the working images are stored.
with open("train_label.txt", "w") as f:
    for i, label in enumerate(labels):
        if i not in bad_indicies:
            f.write(label + "\n")

#Offloads all the new URLs into a new file.
with open("train_urls.txt", "w") as f:
    for url in urls:
        f.write(url + "\n")
```

In another python file, I then opened this new text file of URLs to download the images from them. The images were downloaded with the “urllib” library and named numerically in ascending order from 0 up.

```
1 #Given a list of URLs, downloads the photos and names them numerically starting from 0.
2 def download_urls(urls, already_downloaded):
3     #Counts how many images were downloaded so that progress isn't lost if the program crashes
4     count = 0
5     #Iterates over the URLs
6     for i,url in enumerate(urls):
7         #Chooses the image's name
8         name = "train_images/" + str(already_downloaded + i) + ".jpg"
9         #Downloads the image from the URL
10        urlretrieve(url, name)
11        #Every 20 URLs, updates the file that stores how many photos were already downloaded
12        if (i+1)%20 == 0:
13            update_downloaded(already_downloaded + i + 1)
14        count += 1 #Counts up by one
```

The “update_downloaded” function used in line 13 updates a text file that stores the number of images already downloaded by the program. This was to prevent loss of progress if the code crashed and to give me the flexibility to download the images over multiple runs instead of all at once. The number in this file is stored in the “already_downloaded” variable every time the program is run.

Running this code resulted in a folder named “train_images” in which all the images were stored. A section of the images within this folder can be seen in Figure 11.

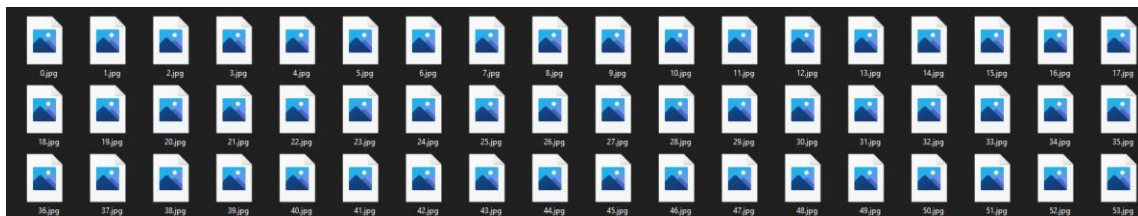


Figure 11: Downloaded images.

However, the “ImageFolder” dataset prebuilt into PyTorch required the images to be organised into further subfolders named after their classes (“torchvision.datasets — Torchvision

0.8.1 documentation,” n.d.). In my case, these classes were 1, 2, and 3 where the least popular third of images should be in group 1 and the most popular third should be in group 3. Therefore, I had to write an “image_organiser.py” file to create these folders for me. In short, I made use of the pandas plugin to create a data frame where each row had the image’s number and popularity score. I then sorted the data frame by popularity score and divide the data frame into thirds.

```
#Opens the labels file
with open("train_label.txt", "r") as f:
    labels = [float(label.strip("\n")) for label in f.readlines()]
#Creates an array of image names, since I named them numerically the names are just
#the numbers from 0 to the number of labels
images = list(range(len(labels)))

#Creates the dataframe from a python dictionary and sorts the values
df = {"image": images, "label": labels}
df = pd.DataFrame(df)
df.sort_values(by='label', ascending=True, inplace = True)

#Splits the dataframe into 3 dataframes of equal size. The images that didn't fit are
#simply discarded since there can only be 2 images maximum that don't fit. "lst" is the
#list of these dataframes.
group_size = len(labels)//3
lst = [df.iloc[i:i+group_size] for i in range(0, len(df)-group_size+1, group_size)]
```

Now, all that I had left to do was to copy the images from the folder in which they were stored to a new folder with subfolders named after the image ratings using the “copyfile” function from the “shutil” library. This completes the dataset downloading stage.

```
for i, df in enumerate(lst): #Iterates over every dataframe in lst
    folder_name = str(i+1) + "/" #Gets the name of the folder where the image should be stored
    for image in df["image"]: #Iterates over every image number
        image_name = str(image) + ".jpg"
        #Copies the image from the folder "train_images" to the folder "train_images_small_groups/class"
        #where class is the rating from 1 to 3
        copyfile("train_images/" + image_name, "train_images_small_groups/" + folder_name + image_name)
```


6.2 Image preparation

As I had explained in sections 5.4.1 and 5.4.2, before these images are fed into the model they have to be normalised and randomly or centre cropped depending on if they are used for training or not. To normalise the images, I first had to find the mean and standard deviation of each channel for all the images in the dataset. My system to do this was to open the images in batches of 100, then to use NumPy to calculate the means and standard deviations for that batch, and then combining those statistics using a custom function I wrote. After every batch, I stored the statistics in a JSON file.

The images were opened as NumPy arrays using the following function:

```
def prepare_image(image):
    image_name = image + ".jpg" #Stores the image name
    #Opens the image and resizes it to a 256x256 image, this same transform is done to
    #all of the images during training so we only need to consider the average values
    #and standard deviations for the resized images.
    image = Image.open("train_images/" + image_name).resize([256, 256], Image.BILINEAR)
    #Converts the image to RGB
    if image.mode != 'RGB':
        image = image.convert("RGB")
    #Turns the image into a numpy array and uses min-max normalisation to map 0-225 to 0-1
    image = np.array(image) / 255
    #Sets RGB channels as first dimension making it easier to separate
    #the channels for each image when doing the calculations.
    image = np.transpose(image, [2, 0, 1])
    return image #Returns the image as a numpy array
```

I combined two sets of different statistics using this function:

```
def combine(stats_1, stats_2):
    sd_1, m_1, n_1 = [np.array(i) for i in stats_1.values()]
    sd_2, m_2, n_2 = [np.array(i) for i in stats_2.values()]

    # Calculate sum x^2 for each set of data
    sx2_1 = (sd_1**2) * (n_1 - 1) + n_1 * (m_1**2)
    sx2_2 = (sd_2**2) * (n_2 - 1) + n_2 * (m_2**2)
    sx2_tot = sx2_1 + sx2_2

    # Calculates the total mean
    m_tot = (n_1 * m_1 + n_2 * m_2) / (n_1 + n_2)

    # Calculate total S_{xx}
    sxx_tot = sx2_tot - (n_1 + n_2) * (m_tot**2)

    # Combined standard deviation:
    sd_tot = np.sqrt(sxx_tot / (n_1 + n_2 - 1))

    # Returns the new statistics as a dictionary
    return {"std": list(sd_tot), "mean": list(m_tot), "n": int(n_1 + n_2)}
```

I also created a function that updates the JSON where the image statistics are stored using the new statistics:

```
#Updates the json file with the newly calculated stats
def update_json(stats_json, new_stats):
    global json_name
    #Combines the old stats with new stats
    stats = combine(stats_json, new_stats)
    #Overwrites the old json file with new stats
    with open(json_name, "w") as outfile:
        json.dump(stats, outfile)
```


Finally, all of these were combined into one function named “get_new_stats”:

```
def get_new_stats(json_name):
    while True:
        # Opens the necessary files
        #####
        with open(json_name) as file:
            stats_old = json.load(file)
        #####

        images_list = [] # Initialises list in which numpy arrays representing images are stored
        count = 0
        # Looks at the images in batches of 100 so that little progress is lost if the program
        # crashes
        for i in range(100):
            seen = stats_old["n"] # Stores how many images have already been seen
            image_name = str(seen + i) # Stores the image name
            try:
                image = prepare_image(image_name) # Opens/prepares the image
            except:
                # Breaks the loop once it can't find any more images.
                # This means that all of the images were looked at.
                break
            count += 1
            images_list.append(image) # Stores the image in the list
        # concatenates images in image_list and calculates stats
        com_images = np.concatenate(images_list, axis=1)
        sd = [com_images[i].std(ddof=1) for i in range(3)] # Standard deviation
        m = [com_images[i].mean() for i in range(3)] # Mean
        n = count # Count, needed when combining stats
        stats_new = {"std": sd, "mean": m, "n": n}
        # Updates the json using the new calculated stats
        update_json(stats_old, stats_new)
```

The result was this JSON file:

```
{...} dataset_stats.json > ...
1  {
2      "std": [0.2962575621947528, 0.2867964879957165, 0.2942563451381508],
3      "mean": [0.46704649924339353, 0.44176759213435224, 0.4165341757253856],
4      "n": 40434
5  }
```

I could then use this JSON file in my training code and input the statistics into the “transforms.Normalize” function to apply it to the images. This was easy to do in PyTorch since all I had to do was to define a transform and use it as a parameter when defining the built-in “ImageFolder” class. This would apply all the transforms to all the images. I also initialised a “train_transforms” object that randomly cropped and flipped the images and a “centre_crop”

object that crops the images to the desired size of 224×224 . The “ImageFolder” is a dataset class that I split into a training and a validation dataset these were then used to create a training and validation “data loader”⁷.

```
#Opens the json file where the statistics are stored
with open(json_name) as file:
    stats = json.load(file)
    std = [round(i, 3) for i in stats["std"]]
    mean = [round(i, 3) for i in stats["mean"]]
    n = stats["n"]

Transform = transforms.Compose([
    # Scales the short side to 256px. Aspect ratio unchanged. Then center
    # crops to make the size of all images equal
    transforms.Resize(256),
    transforms.CenterCrop([256, 256]),
    # Converts PIL Image to a tensor
    transforms.ToTensor(),
    # Normalises each channel of every image.
    transforms.Normalize(mean, std)
])

#train_transform randomly crops and flips images, used to prevent overfitting
train_transform = transforms.Compose([
    transforms.RandomResizedCrop([224, 224]),
    transforms.RandomHorizontalFlip()
])

#When evaluating the models performance you want the test images
#to be the same, so it centre crops instead.
centre_crop = transforms.CenterCrop([224, 224])

#Loads images from folder as an ImageFolder dataset
dataset = ImageFolder("train_images_small_groups", transform = Transform)

#Randomly splits the dataset into training and validation datasets
val_size = int(len(dataset)*val_percent)
train_size = len(dataset) - val_size
train_ds, val_ds = random_split(dataset, [train_size, val_size],
generator=torch.Generator().manual_seed(random_seed))

#Creates dataloaders. They batch the images and act as
#iterator objects that return these batches.
train_dl = DataLoader(train_ds, batch_size, shuffle = True)
val_dl = DataLoader(val_ds, batch_size)
```

⁷ A dataloader is an iterator class that returns images in batches stored as a single tensor.

6.3 Training

The training process is quite simple, for each epoch the code should:

1. Iterate over every batch in the training data loader.
2. Apply random transforms on images.
3. Get the predictions for each batch by feeding them into the model.
4. Calculate the batch loss.
5. Backpropagate (differentiate) the loss.
6. Take a step with the optimiser.

Small batches are used instead of the entire dataset due to restrictions on computing power as it requires a huge amount of memory to perform computations on all 43000 images at once. Furthermore, large batch sizes tend to lead to poor generalisation (Keskar et al., 2017). This was done in the first half of the “fit” function which is shown in Figure 12. The metrics from this training step (i.e. the training accuracy and loss) are discussed in section 6.4.

I also had to evaluate the model on the validation set to notice if I was overfitting or not. Evaluating my model also let me see roughly what the real-world performance of the model would be. This was done using an “evaluate” function (Figure 13) which would evaluate the performance of a model on a given validation data loader and return a dictionary of metrics. These are accuracy and cross-entropy loss which are implemented as follows:

```
#Give predictions and labels finds the accuracy
def get_accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))
#Cross entropy loss function
loss_fn = F.cross_entropy
```

```

def fit(model, train_dl, val_dl, lr, epochs):
    global device
    global momentum

    #The optimiser uses stochastic gradient descent
    opt = torch.optim.SGD(model.parameters(), lr, momentum = momentum)

    for epoch in range(epochs):
        training_losses = [] #List for storing training batch losses
        training_accuracies = [] #List for storing training batch accuracies

        # TRAINING STEP
        #####
        model.train() #Enables training features like dropout layers
        for batch in train_dl:
            images, labels = batch
            #Prepares images and sends them to the gpu
            images = train_transform(images)
            images = images.to(device)
            labels = labels.to(device)

            #Runs the image through the model and gets the predictions
            preds = model(images)
            loss = loss_fn(preds, labels) # Gets the loss
            loss.backward() #Calculates the gradients
            opt.step() #Does an optimiser step
            opt.zero_grad() #Zeros the gradients so the don't build up

            #Stores the accuracy and loss of each batch
            training_losses.append(loss.item())
            training_accuracies.append(get_accuracy(preds, labels))

        #Calculates the average loss and accuracy for the epoch
        epoch_train_loss = np.mean(np.array(training_losses))
        epoch_train_accuracy = np.mean(np.array(training_accuracies))
        #Logs these metrics using Weights and Biases
        wandb.log({"Training Loss": epoch_train_loss, "Epoch": epoch})
        wandb.log({"Training Accuracy": epoch_train_accuracy, "Epoch": epoch})
        #####

```

Figure 12: “fit” function training step


```

@torch.no_grad() #Turns of all gradient tracking when this function is run
def evaluate(model, val_dl):#Evaluates the model on the test_set
    global device

    model.eval() #Turns of dropout layers and other training functionality
    losses = [] #List to store batch losses
    accuracies = [] #List to store batch accuracies

    for batch in val_dl:
        images, labels = batch
        #Centre crops the image
        images = centre_crop(images)
        #Sends the image and data to the GPU
        images = images.to(device)
        labels = labels.to(device)

        #Feeds images into the model to get predictions, these are used to
        #calculate the batch loss and accuracy.
        preds = model(images)
        loss = loss_fn(preds, labels).item()
        accuracy = get_accuracy(preds, labels)

        losses.append(loss)
        accuracies.append(accuracy)

    #Returns the average loss and accuracy for all the batches
    return np.mean(np.array(losses)),np.mean(np.array(accuracies))

```

Figure 13: "evaluate" function.

The “evaluate” is called every epoch in the fit function during the validation step:

```

#Evaluates the model on the validation data loader
epoch_val_loss, epoch_val_accuracy = evaluate(model, val_dl)
#Logs the validation metrics using Weights and Biases
wandb.log({"Validation Loss": epoch_val_loss, "Epoch": epoch})
wandb.log({"Validation Accuracy": epoch_val_accuracy, "Epoch": epoch})

```

This concludes the training loop, all that’s left to do is to run “fit(model, train_dl, val_dl, lr, epochs)” and to set appropriate hyperparameters. The ones that I ended up using, in the end, were a batch size of 16, a learning rate of 0.01, a momentum of 0.5, and a validation size of 20%.

6.4 Metrics

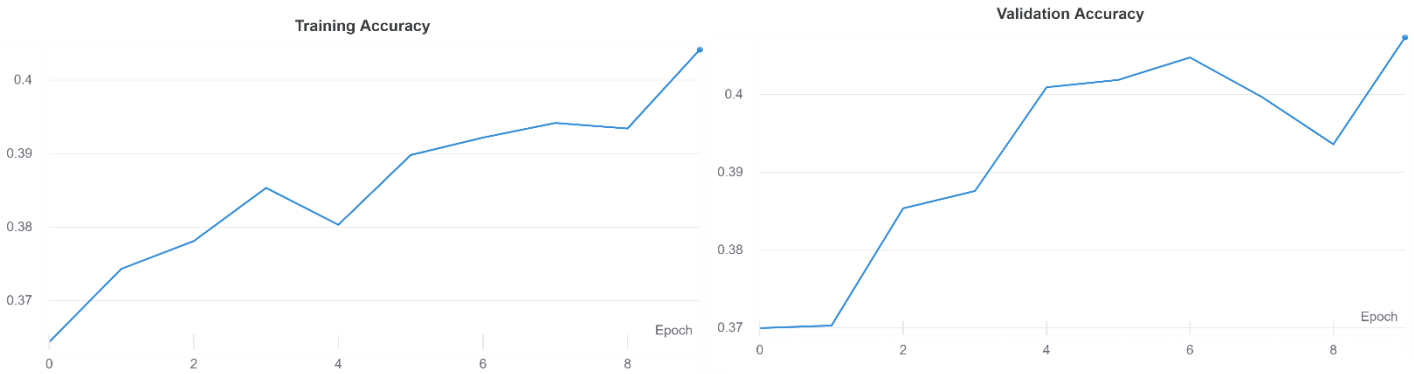


Figure 14: Training and validation accuracy.

The model was able to achieve training and validation accuracies of 40.4% and 40.7% respectively after only 9 epochs of training. By conducting a binomial hypothesis test at a 0.1% significance level I determined that there is sufficient evidence to say that I was able to achieve better than random performance.

$$H_0: p = \frac{1}{3} \text{ Where } p \text{ is the probability of making a correct classification}$$

$$H_1: p > \frac{1}{3}$$

$$X \sim B\left(8086, \frac{1}{3}\right)$$

$$P(X \geq 3294) = 1 - P(X \leq 3293) = 1 - 0.999999 \approx 0$$

\therefore There is sufficient evidence to reject H_0

Although the accuracies were increasing, I noticed that the validation loss was not decreasing while the training loss was. I am unsure as to what had caused this since the model is still performing well on the validation set (as shown by the validation accuracy). This may indicate that although the model made the right predictions, it is not confident in them causing the loss to remain quite high.

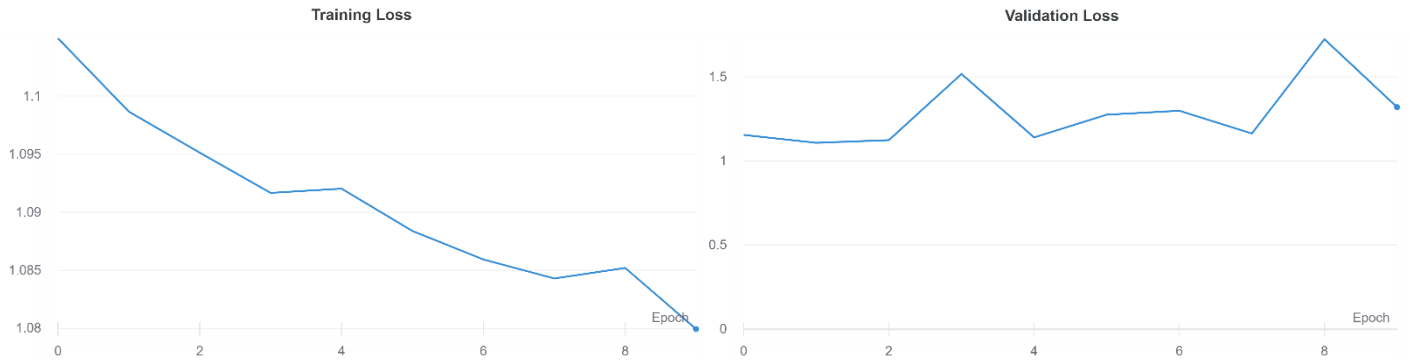


Figure 15: Training and validation Loss

7 Conclusion

By utilising the ResNet50 model as a base and techniques to reduce overfitting I was able to achieve better-than-random classification results. This demonstrates that this is a viable method for creating models for Intrinsic Image Popularity assessment on Flickr and potentially on other social media platforms as well. In general, I would consider this model a success since my goal was to simply create a model that can create classifications that were not random.

However, there were limitations to this model, although it was able to classify images into ranks, this probably was not the best method for teaching a model to determine which image is “better” like in the paper by Ding, K., Ma, K., and Wang, S. (2019). To improve this, I could use learning-to-rank algorithms in the future. Furthermore, I only trained my model on less than a sixth of the SMP Challenge dataset. More data and more training time would undoubtedly improve the model’s performance.

8 Table of figures

Figure 1	6
Figure 2 (“Neural Networks From Scratch - victorzhou.com,” n.d.)	8
Figure 3: A CNN. By Renanar2 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=75218608	9
Figure 4: (Talo et al., 2019) The number beside the name of the model represents the total number of layers present.	11
Figure 5: A random line.	12
Figure 6: A plot of the loss function.....	13
Figure 7: By Chabacano - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=3610704	16
Figure 8 (He et al., 2015)	18
Figure 9: (Mahmood et al., 2020)	18
Figure 10: Image URLs and popularity scores	19
Figure 11: Downloaded images.	21
Figure 12: “fit” function training step.....	28
Figure 13: "evaluate" function.	29
Figure 14: Training and validation accuracy.....	30
Figure 15: Training and validation Loss	31

9 Bibliography

- Alman, J., Williams, V.V., 2020. A Refined Laser Method and Faster Matrix Multiplication. arXiv:2010.05846 [cs, math].
- Biewald, L., 2020. Experiment Tracking with Weights and Biases.
- Ding, K., Ma, K., Wang, S., 2019. Intrinsic Image Popularity Assessment, in: Proceedings of the 27th ACM International Conference on Multimedia, MM '19. Association for Computing Machinery, Nice, France, pp. 1979–1987. <https://doi.org/10.1145/3343031.3351007>
- Flickr Services [WWW Document], n.d. URL <https://www.flickr.com/services/api/misc.urls.html> (accessed 2.25.21).
- Géron, A., 2019. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media.
- He, K., Zhang, X., Ren, S., Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs].
- Hunter, J.D., 2007. Matplotlib: A 2D Graphics Environment. Computing in Science Engineering 9, 90–95. <https://doi.org/10.1109/MCSE.2007.55>
- Ioffe, S., Szegedy, C., 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167 [cs].
- Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P., 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. arXiv:1609.04836 [cs, math].
- Lecun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 2278–2324. <https://doi.org/10.1109/5.726791>

- Mahmood, A., Giraldo, A., Bennamoun, M., An, S., Sohel, F., Boussaid, F., Hovey, R., Fisher, R., Kendrick, G., 2020. Automatic Hierarchical Classification of Kelps Using Deep Residual Features. *Sensors* 20, 447. <https://doi.org/10.3390/s20020447>
- Mckinney, W., 2010. Data Structures for Statistical Computing in Python. Proceedings of the 9th Python in Science Conference.
- MSELoss — PyTorch 1.7.1 documentation [WWW Document], n.d. URL <https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html> (accessed 2.22.21).
- Nelson, J., 2020. Why and How to Implement Random Crop Data Augmentation. Roboflow Blog. URL <https://blog.roboflow.com/why-and-how-to-implement-random-crop-data-augmentation/> (accessed 2.23.21).
- Neural Networks From Scratch - victorzhou.com [WWW Document], n.d. URL <https://victorzhou.com/series/neural-networks-from-scratch/> (accessed 2.22.21).
- Oliphant, T.E., 2006. A guide to NumPy. Trelgol Publishing USA.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32.
- Simonyan, K., Zisserman, A., 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*.
- Talo, M., yildirim, Ö., Baloglu, U., Aydin, G., Acharya, U.R., 2019. Convolutional neural networks for multi-class brain disease detection using MRI images. *Computerized*

Medical Imaging and Graphics 78, 101673.

<https://doi.org/10.1016/j.compmedimag.2019.101673>

torchvision.datasets — Torchvision 0.8.1 documentation [WWW Document], n.d. URL

<https://pytorch.org/vision/0.8/datasets.html#imagefolder> (accessed 2.25.21).

Wu, B., Cheng, W.-H., Liu, P., Liu, B., Zeng, Z., Luo, J., 2019. SMP Challenge: An Overview of Social Media Prediction Challenge 2019, in: Proceedings of the 27th ACM International Conference on Multimedia.