Facial Expression Recognition using Deep Learning

Heriot-Watt University

May 6, 2018

Abstract

Machine learning has been around for several decades however it was not practically useful up until recently. Deep learning is the technology that has proficiently solved some of the most challenging problems to artificial intelligence such as image recognition, natural language processing. In this report, background theory of deep learning is discussed and then applied to create a facial recognition model using a labeled dataset of ~34,000 images. The model can categorises the images in seven emotion classes. It achieves an accuracy of 58% on the test dataset. That accuracy may not seem impressive to humans, but it an accomplishment for a machine to learn patterns from data with little human involvement.

Contents

Introduction

Machine Learning is a technique in which computers learn from example without being explicitly programmed. This is essential to be able to build artificial intelligence. There are various methods to make a machine learn patterns. This is still an emerging field so there are still new ideas being developed. *Deep Learning* is a method to get a machine to learn using labeled data i.e. a type of machine learning. It had been quite promising in the recent years, but there are still various challenges that need to be overcome. Deep learning has been around for over decades, however it only became viable with the development of cheap computer's graphics processing unit (GPU).[1] GPUs are special purpose CPUs which are designed to compute in parallel. The deep neural networks are structured in a way that require the same computation to be applied over thousands of values while training.

[1] The investment in GPUs was due to their use in gaming industry. At the time, machine learning wasn't considered as a main use case. Another boost was due to crypto-currency mining which also requires parallel simple computations rather than complex and varying computations which CPUs excel at.

This report contains details about how deep learning was developed, and functions in its current state. After the theory, a model will be developed for recognising facial expression of humans using this approach. Facial feature recognitions is a effortless task for humans, however it provides a challenge when implementing it on a computer. If a traditional explicit approach is taken then it can take significant time and resources to build a model that works reliably with all face types, skin-tones etc. However, it is possible to build a basic model using deep learning with much less effort on programers part. The model is built using supervised learning, i.e. labeled dataset, which did require human assistance to build, therefore it not a silver bullet to artificial intelligence just yet.

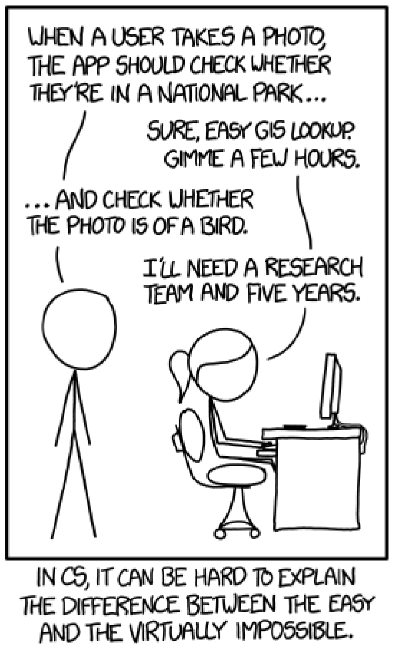

WHEN A USER TAKES A PHOTO, THE APP SHOULD CHECK WHETHER THEY'RE IN A NATIONAL PARK...
SURE, EASY GIS LOOKUP. GIMME A FEW HOURS.
... AND CHECK WHETHER THE PHOTO IS OF A BIRD.
I'LL NEED A RESEARCH TEAM AND FIVE YEARS.
IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

Figure 1: Trying to build a bird or not bird classifier using explicit approach. SOURCE: XKCD

Background Theory

This sections explains what machine learning is, and how it functions. It develops the knowledge needed to successfully build a deep neural network.

Machine Learning

In machine learning a computer takes a cache of data and builds a model with it. A model is the output of the process, it is used to predict the results based on some inputs using the correlation present in the data used to train it. This means the training data is an important part of the process because the model is a representation of the data and it cannot be expected to produce results which have not been encountered before in the data.[2] The impressive part is that the process of building the model is automated, hence the name machine learning. This automated process can be used to create a model for an application which cannot be easily described using equations and laws. The model is just an non-linear function which fits the presented data with minimal error.

[2] Pre-processing of training data is an important part of machine learning because if the model has been trained with some bias then it will need to retrained from scratch to remove that bias.

Generally machine learning techniques can be split into three types: *supervised, unsupervised,* and *reinforced learning*. Supervised is when labeled data is used to build the model. Labeled dataset contains the inputs, and outputs which are the data, and label, respectively. The initial dataset needs to be generally created by hand, which can be tedious and sometimes not practical.

Unsupervised learning is when un-labeled data is provided and the machine is asked to infer trends and patterns. This is useful for finding anomalies/fraud, clusters and associations in data. This is often referred to as data mining.

Reinforced learning is similar to supervised learning in the sense it has inputs and output. However, it also requires a function to produce a complementary grade for the produced output which signifies the correctness of the result produced. This value is then used to adjust the model respectively, i.e. if the grade is positive then the model is rewarded to produce more similar results, and the model is penalised if the result is incorrect telling the model to not repeat the mistake. This changes the model on the fly and improves the performance over iterations, unlike supervised learning model which will produce the same results with the same inputs.

The supervised learning technique's applications can be divided into two types: classification and regression. Regression is predicts the output values using the input values. Deep learning is seldom used for regression because it yields poor performance and regression problems can be solved using simpler approaches. The model resembles a function which defines the training data. On the other hand, classification model predicts the class/category of the input value. The results can be only one of the pre-defined classes.

Neural Networks

The inventor of one of the first neuro-computers, Dr. Robert Hecht-Nielsen, defined a neural network as:

> "...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs."[3]

[3] Maureen Caudill. *Neural Network Primer: Part I.* AI Expert, 1989

The neural networks are loosely modelled after the mechanism of the mammalian brain. The brain is composed of billions of neurons which transmit signals from one another, in a neural networks these processing elements are called *nodes*. The neural network is composed of a network of these nodes, the connection between two nodes has an associated weight called *connection weight*. The connection weight determines the magnitude of the input signal coming from that connection has on the node. FIGURE 2 visualises these parts of an example node. In the example the node has three inputs signals (x_1, x_2, x_3) with their respective weights (w_1, w_2, w_3) and produces one output signal (y). The diagram also shows another factor, *bias* (b), which is another factor associated with storage of information. The node can be defined by the weights matrix and the bias value. The weighted sum (EQUATION 1) of the input signals and the bias is then passed to the activation function of the node. The activation function describes the behaviour of the node; it decides if the neuron should fire/activate or not. Activation functions are discussed in more detail later in the report.

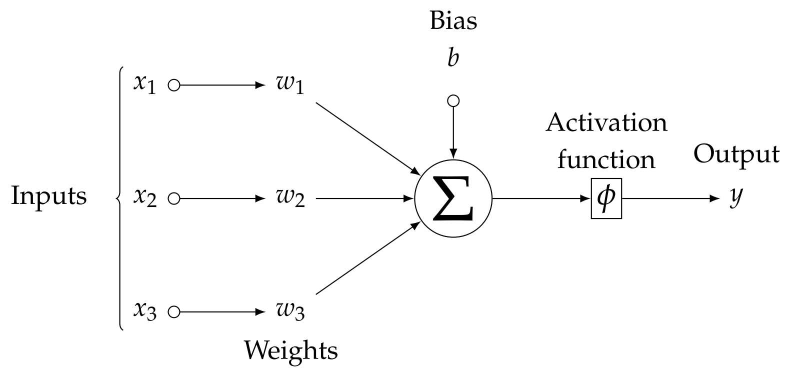


Figure 2: A diagram describing a node of a neural network. It shows how inputs, weights, and bias affect the node's definition.

$$v_i = (w_1 \times x_1) + (w_2 \times x_2) + (w_3 \times x_3) + b \qquad (1)$$
$$y_i = \phi(v_i) \qquad (2)$$

where:

$$v_i = \text{weighted sum of connections of node } i$$
$$\phi(\cdot) = \text{activation function of node } i$$
$$y_i = \text{output signal of node } i$$

As a neural network is made of a network of nodes like this, they stack up as layers passing their output as the input for subsequent nodes. The layers can be classified in three types: *input*, *hidden*, and *output layers*. Input layer is where the data is input by the user, the nodes on this layer do not have to find out the weighted sum and activation function is usually just a linear function ($y = x$) because the outputs of this layer is the same as inputs. It serves as a pipeline to transmit the data through to the next layer. Output layer is which produces the final output of the neural network. Usually its number of nodes is equal to number of classes on a classification model. They output probability of the input belonging to each class. The layers in between the input and output layers are called hidden layers. These contain the node detailed above. There is no theoretical limit on how many hidden layers a neural network can have, only a practical limit because with increase in number of layers the computations will increase exponentially.[4] These layers are visualised in FIGURE 3. Since the inputs of a node in a hidden layer depend on multiple signals from the previous layer, the signal advances layer by layer; all the nodes on current layer need to evaluated before starting with next layer.

[4] Neural networks with only one hidden layer are called shallow or vanilla neural networks. Any network with two or more hidden layers is called deep neural network. This distinction is made because up until recently only shallow neural networks were practical.

After the neural network has set up its nodes with reasonable initial weights of all layers, it needs to be trained. In the training process the weights will be iteratively updated to produce the expected results with the respective inputs. The function that defines how the weights should be updated is called *learning rule*. The most common and simple rule is called *delta rule* which uses the difference between the target activation and the obtained activation of a node to update the weights, EQUATION 5.

The generalised delta rule of the neural network will be represented by EQUATION 4. The weights are incrementally updated using EQUATION 3. The equation uses a constant control variable, α, called *learning rate*. This rate change how much the weights change in a single update. This needs to selected carefully because a high learning rate would make it difficult for the weights to converge, and a low learning rate would take more number of iterations to reach a

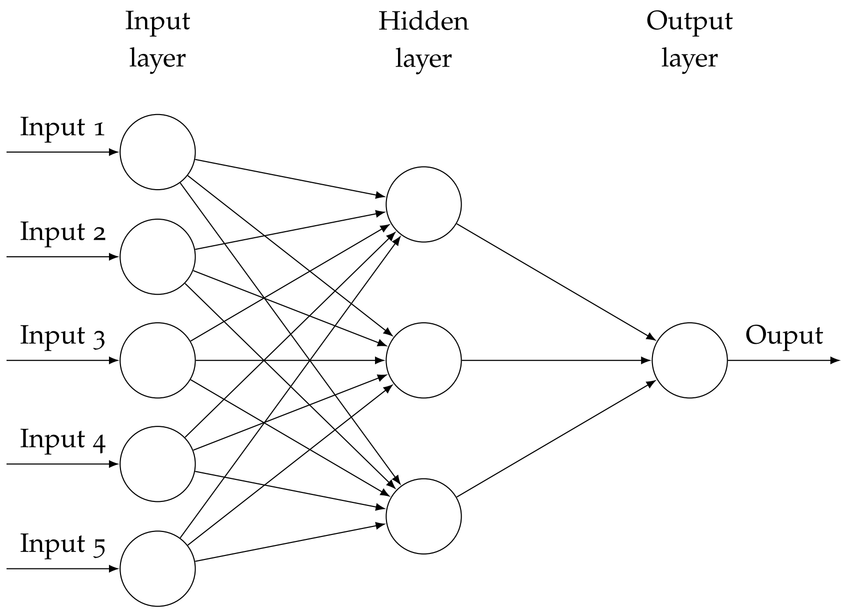


Figure 3: The structure of layers in a neural network. In this example there is only a single hidden layer, this is usually not the case for complex neural networks.

solution.

$$w_{ij} \leftarrow w_{ij} + \alpha\delta_i x_j \tag{3}$$
$$\delta_i = \phi'(v_i)e_i \tag{4}$$
$$e_i = d_i - y_i \tag{5}$$

where:

d_i = target activation

y_i = obtained activation

e_i = output/activation error

v_i = sum of weighted forward signals and bias

$\phi'(\cdot)$ = derivative of activation function

x_j = input signal from node j

α = learning rate $(0 < \alpha \leq 1)$

w_{ij} = weight of signal from node j to node i

The weights are updated in the reverse order of layers i.e. from output layer to input layers, this is called *back-propagation*. Since target activation is only know for the output layer nodes; without target activation the delta rule cannot determine the error of hidden nodes, hence cannot update the weight systematically. This was an unsolved problem until the introduction of back-propagation algorithm in

1986.[5] The algorithm provided a systematic method to determine the error of the hidden nodes. Using these error the delta-rule can be applied on hidden layers. The error of a hidden node is defined as the weighted sum of the back-propagated deltas from the layer on the immediate right of the node. This is described as EQUATION 6. This error is then used as described in delta-rule, EQUATIONS 4 AND 3. All the deltas of hidden nodes are first determined before applying the learning rule to change the connection weights.

[5] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning representations by back-propagating errors*. Nature, 1986

$$e_i = \sum w_{ij}\delta_j \tag{6}$$
$$\delta_i = \phi'(v_i)e_i \tag{7}$$
$$w_{ij} \leftarrow w_{ij} + \alpha\delta_i x_j \tag{8}$$

where:

$\delta_j =$ delta of node j

$w_j =$ weight of signal from node j to node i

$e_i =$ error of hidden node i

Note that the final equation that determines the weight update is a function of the node's input (v_i) and the error in the output node. The error (e_i) of hidden and output nodes is calculated differently but they are both used in a function with the input weighted sum to update the connection weights.[6] The update does not have to be the delta rule in EQUATION 4, this method is called *Stochastic Gradient Descent*. There are other optimiser algorithms that can perform better such as Adagrad, Adadelta, RMSprop, and Adam, provide an alternative to classical SGD. These methods change the learning rate parameter.

[6] The error of hidden nodes is calculated by back-propagated deltas while error output node can be as simple as the difference between target and actual activation.

These per-parameter learning rate methods provide heuristic approach without requiring expensive work in tuning hyperparameters for the learning rate schedule manually.

The learning rule can of output layers also employ *cost functions* to improve the learning rate of the neural network. Cost functions define the measure of the error as a non-linear function. The most effective cost function is called *cross-entropy function* which is defined as EQUATION 9. A sketch of the function is shown in FIGURE 4 which shows how the error changed exponentially rather than linearly.

$$e_i = -d_i ln(y_i) - (1 - d_i) ln(1 - y_i) \tag{9}$$

$$= \begin{cases} -ln(y_i) & d_i = 1 \\ -ln(1 - y_i) & d_i = 0 \end{cases} \tag{10}$$

where:

y_i = obtained activation

d_i = target activation

e_i = activation error

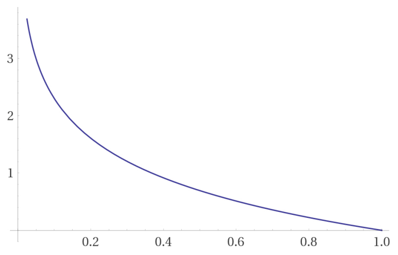


Figure 4: Cross entropy function plotted for $d_i = 1$. The error is equal to zero when $y_i = 1$, but grows geometrically when $y_i \to 0$. The plot is mirrored over $y - axis$ where $d_i = 0$.

The value produced by the weighted sum of signals and the bias are in the domain of real number ($-\infty$ to $+\infty$). Since the value is not bounded so it is difficult to tell how activated the node is in the network. A neural network without a activation function is the same as defining it as a linear activation function i.e. $y = x$. Therefore, the neural network would be a linear regression model which has limited applications and performance issue when dealing with complex data such as images, videos, audio etc. The activation function also needs to be differentiable because the derivative is needed to perform back-propagation to train the neural network as discussed previously. The gradient of a linear activation function is constant so the descent to the solution is going to be on constant gradient. The changes made by back-propagation to weights of connections from a node will be constant and not dependent on forward input signal.

The most widely used activation function is *sigmoid activation function*. It returns a value between 0 and 1, which is usually a requirement for probability models. A sketch of this function is shown in FIGURE 5. The function is non-linear so it can be used to model complex functions. Another benefit of using this functions is that it has a high gradient when x is small which makes a clear distinction on prediction, assuming 0.5 is the decision boundary. It does have a major drawback which is the *vanishing gradient* problem. Notice how the gradient of the function significantly decreases with larger x values. And when training a neural network, this gradient is used in the learning rule to adjust the connection weights. Therefore, if the gradient is very small the adjustment will not be significant, leaving the node practically unchanged. This leads to problems with deep neural networks because the layers closer to output layer will be trained and layers closer to input layer be not be properly trained, or will be drastically slow.

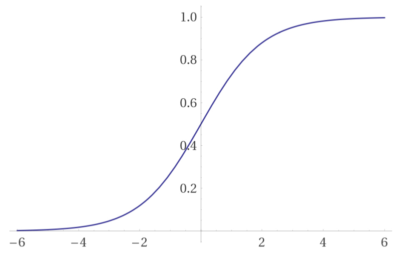


Figure 5: Sigmoid activation function.

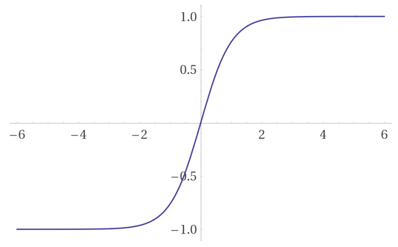


Figure 6: Hyperbolic tangent activation function.

There is a variation of sigmoid activation function which outputs values between -1 and 1. It is called *hyperbolic tangent activation function* (TanH) which is sketched in FIGURE 6. It has the benefit of having a larger gradient since the output values are centred around 0, so it will reach a solution quicker. And it maps negative inputs to negative values and zero inputs to near zero values.[7] However, it also suffer from the vanishing gradient problem.

[7] Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus Müller. Efficient backprop. *Neural Networks: tricks of the trade*, 1998

The *Rectified Linear Unit activation function* (ReLU) is the state-of-the-art solution to vanishing gradient problem. The function is sketched in FIGURE 7. It is defined as $max(0, x)$ i.e. return the input if it is greater than zero otherwise return zero. The output of the function is not bound unlike the previous two functions so the activation can blow up. The use of ReLU activation function will lead to a sparse activation because any negative values will result in nodes being turned off. This will make the neural network lighter compared to other activation function where almost all will be activated to describe the output of the network. The function's and its derivative's definition is also simpler which will reduce the computation load because it involves simpler mathematical operations. However, it does have a drawback which is the gradient can be zero when input is negative. This will not adjust the weights during back-propagation. This is called *dying ReLU problem*. This problem can cause certain nodes to die i.e. never activate making a substantial part of the neural network passive.[8]

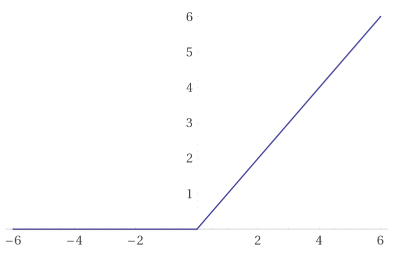


Figure 7: ReLU activation function.

[8] Avinash Sharma. Understanding activation functions in neural networks, 2017a. URL `https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0`

There is a variation of ReLU which tries to tackle dying ReLU problem called *leaky ReLU function*. It changes the flat part of the function to a low-gradient linear function as shown in FIGURE 8. The main idea is to introduce a small gradient to keep the weight updates alive.

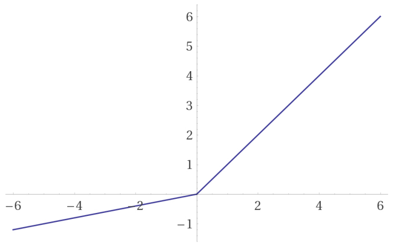


Figure 8: Leaky ReLU activation function.

NAME	FUNCTION	DERIVATIVE
Sigmoid	$\phi(x) = \frac{1}{1+e^{-x}}$	$\phi'(x) = \phi(x)(1-\phi(x))$
TanH	$\phi(x) = \frac{2}{1+e^{-2x}} - 1$	$\phi'(x) = 1 - \phi(x)^2$
ReLU	$\phi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$	$\phi'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$
Leaky ReLU	$\phi(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}$	$\phi'(x) = \begin{cases} \alpha & x \leq 0 \\ 1 & x > 0 \end{cases}$

Table 1: Activation functions and their derivatives.

An issue with ReLU is that it cannot be used for the output layer. The output layers needs to output the probability of a classification which needs to be between 0 and 1. Sigmoid can be used for binary classification, i.e. two classes. The output value can be compared against a threshold (0.5) to decide which class the output belongs to.

However, sigmoid function cannot be used for output layer of a multi-class model. The function does not account for other output nodes so the total probability of all output nodes can add up to more than one. For multi-class model *softmax function* is more suitable. The softmax function maintains the sum of the output values to be one by accounting for the relative magnitudes of all the output nodes. The softmax function is defined in EQUATION 11.

$$\phi(v_i) = \frac{e^{v_i}}{\sum\limits_{k=1}^{M} e^{v_k}} \tag{11}$$

where:

$\phi(\cdot)$ = softmax activation function

v_i = weighted sum of inputs of node i

M = number of nodes in output layers

e = Euler's number

Convolutional Neural Networks

Convolution neural networks (CNN) are a type of deep feedforward networks which have feature extractors built in them. They imitate how the visual cortex of the brain processes and recognises the images. They are needed because using original images as inputs for classification task will result in a poor performance regardless of the training method. In the wild the images a neural network encounters may not be centred, focused, or in the same format as the images used to train it. A fully connected deep neural networks works by learning global features which can easily be changed by moving the image away from centre. FIGURE 9 shows how the a fully connected neural network is fooled by translating the image. Convolution neural networks have convolution and pooling layers built in them. The features extracted in these layers is passed to a fully connected neural network which acts as a classifier as before, but it learns from extracted local features.

Convolution layers apply the convolution operation on the input data. The output of the operation is called *feature map*. This layer does not have any connection weights instead it has *convolution filters* a.k.a. *convolution kernel*. The result of the filter operation does go through an activation function similar to a node's output. These filters are learnt in the training process. Usually a convolution layer employs more than one filter. The network places the filter on every spacial position (pixel on image) to produce a two-dimensional *feature map*. The network uses this information to learn filters that activate when

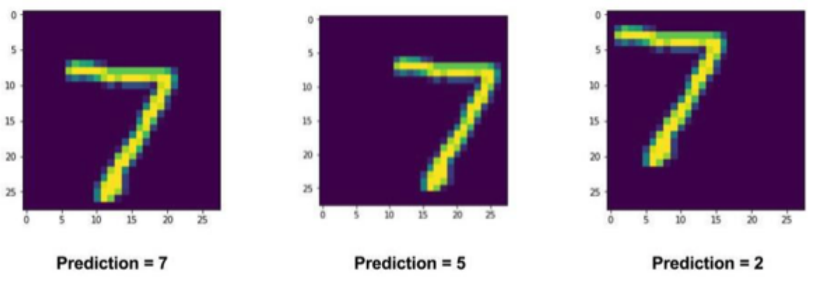


Figure 9: This neural network was trained using MNIST dataset to recognise handwritten digits. All the training images were centred. Hence the prediction is dependent on the position of the input as shown.

they see a type of visual feature. The visual feature can be as simple as colours, and edges, or it can be a more complex pattern.

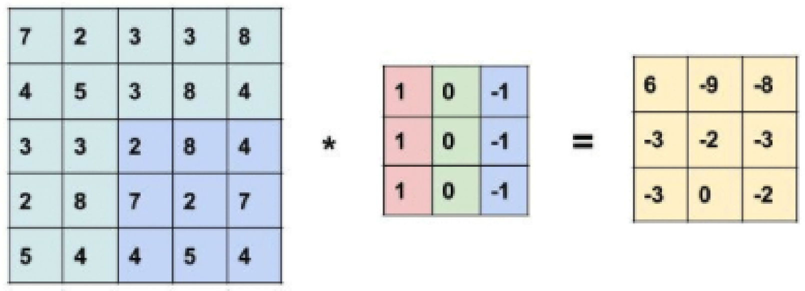


Figure 10: Example of applying a 3×3 convolution filter on an 2D image. The feature map is calculated as the weighted sum. The filter is placed on each pixel as shown in purple in the image. The resulting value is equal to the sum of the element-wise multiplication in the matrix.

Pooling layer is usually used with convolution layers. They progressively reduce the spatial size of the image. This is done to reduce the computation load and overfitting. There are various types of pooling operations such as *max, min, mean*. The most common one is max-pooling, which takes the maximum value from the pooling window as shown in FIGURE 11.

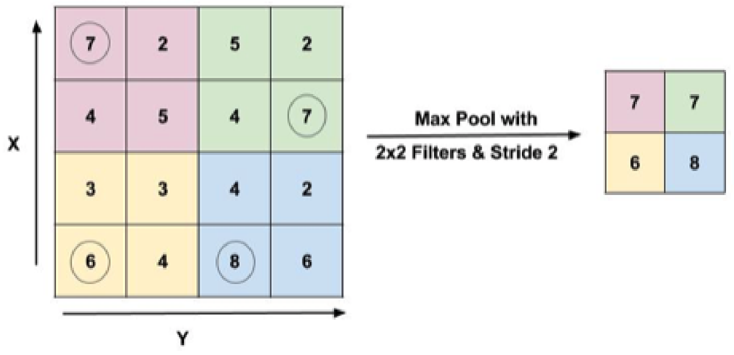


Figure 11: Max pooling layer example. The 2×2 window is placed on the image and selects the maximum value.

Application

The chosen application for this project was facial expression recognition. The model is built using *TensorFlow* with *keras'* application programming interface (API). Keras is a high-level neural networks API which runs on top of TensorFlow with the focus on fast experimentation.[9] TensorFlow is an open source software for high performance numerical computations which can be deployed on various platforms (CPUs, GPUs, Tensor-PUs). It was originally created by researchers and engineer at Google Brain.[10]

[9] François Chollet et al. Keras. `https://keras.io`, 2015

[10] Google Brain et al. Tensorflow. `https://www.tensorflow.org`, 2015

The labeled dataset, Facial Expression Recognition 2013 (FER-2013), was available from *kaggle*.[11] Kaggle is an online platform for hosting open source machine learning competitions. The dataset consists of 48×48 greyscale images of centred faces. The training set has $28,709$ labeled images, and the test set has $7,178$ examples. There are seven emotion classifications as show in TABLE 2. The ratio of classes is the same in training and test sets.

[11] Pierre-Luc Carrier and Aaron Courville. Facial expression recognition challenge, 2013. URL `https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/`

	# OF EXAMPLES		
CLASS	TRAINING	TEST	% OF DATASET
Angry	2995	958	13.3
Disgust	436	111	1.5
Fear	4097	1024	14.3
Happy	7215	1774	24.7
Sad	4830	1247	17.4
Surprise	3171	831	11.6
Neutral	4965	1233	17.2

Table 2: Seven classes in the dataset and their number of instances.

Implementation

A convolutional neural network is the appropriate choice for facial expression recognition since the input data can vary which will defeat a fully connected neural network. The model's input is the 2D greyscale image, and the output is a *one-hot encoded* array.[12] The initial layers of the model are convolution and pooling layers to extract the features of the images then followed by fully-connected neural network for predictions. The structure of the layers is shown in TABLE 3. The network employs cross-entropy cost function as describe earlier. This architecture was inspired by an ImageNet LSVRC-2010 contest winner which uses two convolution layers followed by max-pooling layer structure.[13] The high number of convolution layer were chosen to allow the network to develop complex filters which will is required as this a multi-class classification model.

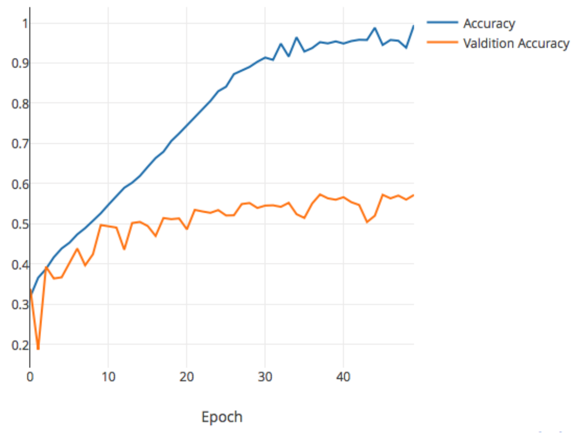
[12] One-hot encoded format is where rather than storing the class-number, a one-dimensional matrix of size equal to number of classes is used. For instance, with total of 4 classes, [3] will be encoded into one-hot matrix as $[0, 0, 1, 0]$.

[13] Alex Krizhevsky. Imagenet classification with deep convolutional neural networks. Technical report, Univeristy of Toronto, 2012

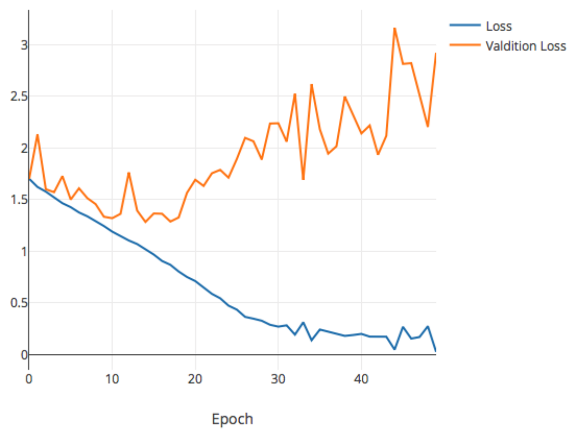
Layer	Description	Activation Function
Conv2D	32 filters (3x3)	ReLU
Conv2D	32 filters (3x3)	ReLU
MaxPooling2D	1 filter (2x2)	
Conv2D	64 filters (3x3)	ReLU
Conv2D	64 filters (3x3)	ReLU
MaxPooling2D	1 filter (2x2)	
Conv2D	64 filters (3x3)	ReLU
Conv2D	64 filters (3x3)	ReLU
MaxPooling2D	1 filter (2x2)	
Dense	512 nodes	ReLU
Dense	512 nodes	ReLU
Output	7 nodes	Softmax

Table 3: Layers of convolutional neural network.

The GPU used to train the model was NVIDIA Tesla K80 with 12GB memory, the CPU used for controlling the GPU was Intel Xeon with 61GB of RAM. Since the number of training parameters are constant each epoch took 15 seconds. The information about number of training parameters are in Appendix A. The network was trained for 50 epochs, the accuracy and loss curves are shown in FIGURE 12. The accuracies of the model on validation and training sets are 0.57 and 0.99, respectively. Notice how the validation loss and accuracy are significantly worse than training accuracy and loss, this is the result of overfitting. Overfitting is when the model learns the training data too well and fails to generalise. Overfitting can be overcome with various methods, the simplest is applying dropout in layers. With dropout the network randomly selects nodes to train in each epoch. Rest of the nodes are deactivated, which prevents overfitting as it continuously alters which nodes and weights to update.



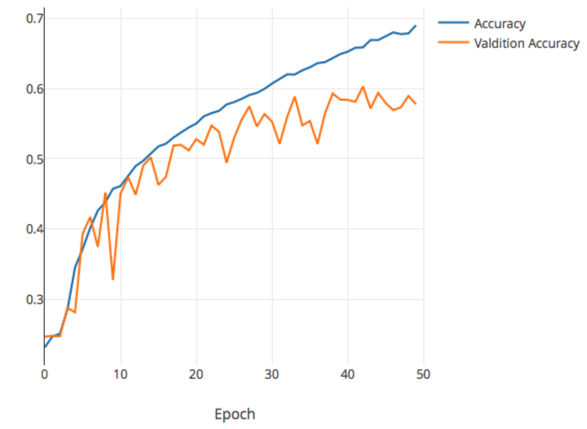
(a) Accuracy Curve



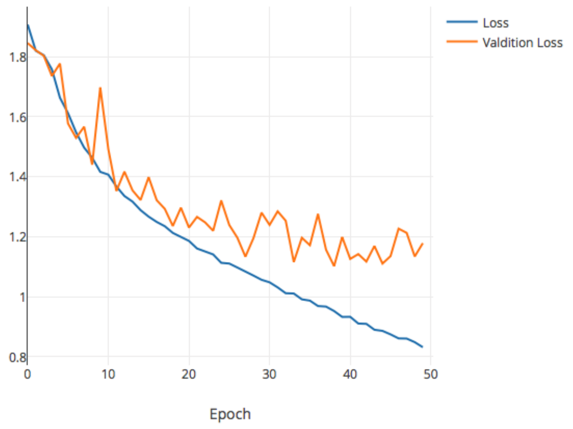
(b) Loss Curve

Figure 12: Accuracy and loss of the network after each epoch on training and test/validation datasets.

Another model was trained after adding dropout layers after each pooling and dense layers in the previous model. The dropout rate for pooling and dense layers were set to 0.25 and 0.5, respectively. The results of this model are shown in FIGURE 13. The model is more generalised but there is still overfitting in the model as the validation and training curves are moving further apart.



(a) Accuracy Curve



(b) Loss Curve

Figure 13: Accuracy and loss of the network with dropout after each epoch on training and validation datasets.

Another solution for overfitting is regularisation of weights. It works by penalising the loss function to constraint the complexity of the network. The most common regularisation function is penalises the square value of the weights, it changes the cost function as shown in EQUATION 12. This is applied on all the convolution layers with $\lambda = 0.001$ for the next model and trained for 200 epochs.

$$e_i = \varepsilon(d_i, y_i) + \lambda \frac{1}{2}|w|^2 \tag{12}$$

where:

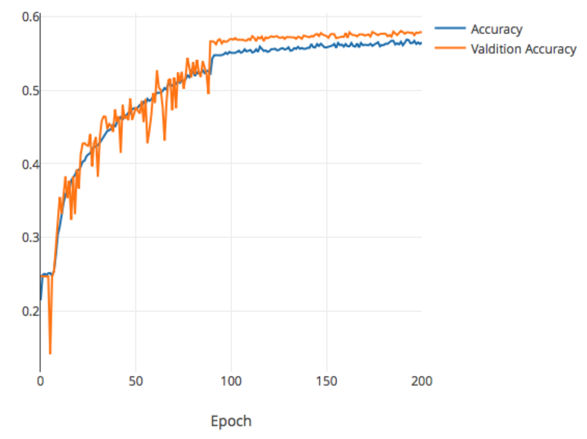
$\varepsilon(\cdot)$ = cost function

w = weights matrix of connection from immediate right layer (towards output layer)

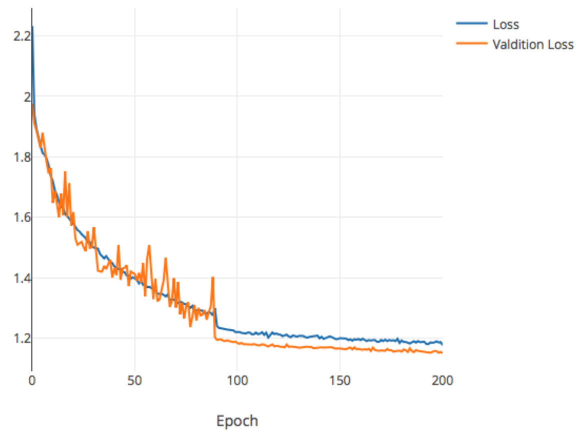
The training and validation accuracy and loss are shown in FIGURE 15. The model is no longer overfitting as the training and validation curves are not diverging. Note that the learning rate of the model was changed from 0.001 to 0.0001 on epoch 88 to let the model converge; this change is evident in the figures. The validation accuracy was of 0.58 after epoch 200. The validation accuracy is more than the training set's accuracy that means the model has generalised well.

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

Figure 14: Hidden layer of a deep neural network is a complete black-box at the moment. SOURCE: XKCD



(a) Accuracy Curve



(b) Loss Curve

Figure 15: Accuracy and loss of the network with dropout, and regularisation on convolution layers after each epoch on training and validation datasets. The learning rate was decreased from 0.001 to 0.0001 on epoch 88 to let the model converge. The final validation accuracy was of 0.5787.

Results

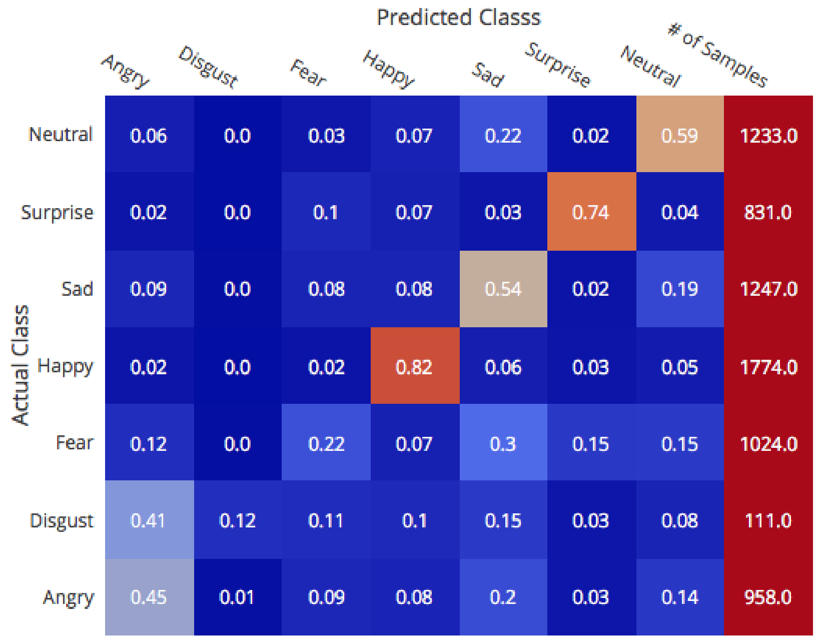
A *confusion matrix* is a table that describes the performance of a model on a test data. It helps identify which classes are being confused with each other by the model. The confusion matrix of this model is shown in FIGURE 16. The best performing class is *"Happy"* at 0.82 accuracy, it also happens to constitute 24.7% of the whole dataset. On the other hand, the worst performing class is *"Disgust"* at 0.12 accuracy which only makes up 1.5% of the datasets. This class is mostly confused with *"Angry"* class, as shown. *"Angry"* class has significantly more samples so it possible that the model generalises the class too much and is not able to distinguish *"Disgust"* due to limited sampled. So it is likely that the performance is being limited by the size of datasets for that class.

FIGURE 26 and FIGURE 27 shows examples of incorrect predictions with lowest and highest probabilities for each class, respectively. FIGURE 29 shows examples of correct prediction with highest probabilities for each class. Looking at these predictions provides better understanding of the confusion matrix. The dataset was assembled automatically with some supervision so it contains some label noise. And some images could potentially fall in two categories even if they were classified by a human. It is estimated that the human performance is roughly between 65% and 68%.[14]

[14] Charlie Tang. Human performance | facial expression recognition challenge, 2013. URL `http://bit.ly/13Zr6Gs`

The feature extractors of the convolutions layers are learnt by the training process. Initially the features are randomly initialised with uniform distribution between -0.5 and 0.5. There are two method to visualise the learnt filters, first is to simply show the weights of the filters however this can get difficult to understand for deep layers as the filters weights do not reflect how image is changed with previous convolutions. For the initial layer this is quite straightforward.

It is best to keep the filter values small as possible to reduce the size of the model while training and in production. However, it starts being an issue when the number of floating points does not have enough precision to train the model further. Modern GPUs support double floating point precision up to 64-bits



	Angry	Disgust	Fear	Happy	Sad	Surprise	Neutral	# of Samples
Neutral	0.06	0.0	0.03	0.07	0.22	0.02	0.59	1233.0
Surprise	0.02	0.0	0.1	0.07	0.03	0.74	0.04	831.0
Sad	0.09	0.0	0.08	0.08	0.54	0.02	0.19	1247.0
Happy	0.02	0.0	0.02	0.82	0.06	0.03	0.05	1774.0
Fear	0.12	0.0	0.22	0.07	0.3	0.15	0.15	1024.0
Disgust	0.41	0.12	0.11	0.1	0.15	0.03	0.08	111.0
Angry	0.45	0.01	0.09	0.08	0.2	0.03	0.14	958.0

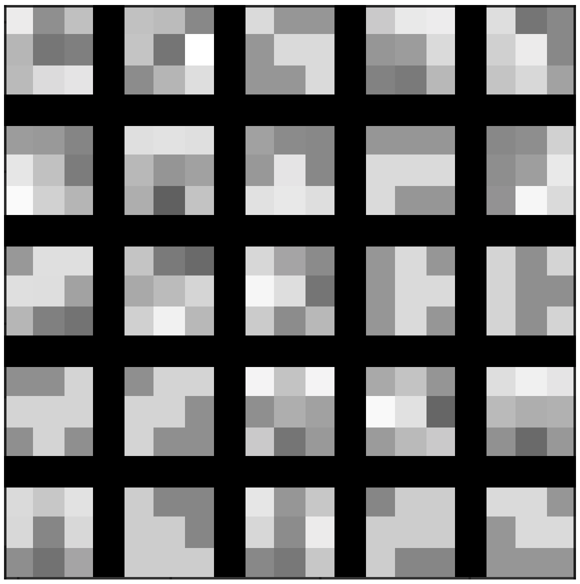
Figure 16: The confusion matrix of the model. Test data of classes on $y - axis$ is supplied and the model's predicted class is $x - axis$. The values are equal to the percent of the whole test sample for each class. The number of test samples for each class is shown in the last column which makes up the whole test set.

Second method is to find an image that generates the maximum activation map for that kernel. This is same as back-propagation except the image is changed and weights are kept constant. The generated image shows what features this filter extracts.[15]

[15] Andrej Karpathy. Visualizing what convnets learn. Technical report, Stanford University, 2017. URL `http://cs231n.github.io/understanding-cnn/`

The first layer's filter weights and the images that produce the maximum activation are visualised in FIGURE 17. As expected this layer contains simple filters which are edge detectors at various angles. There are some images that are either random or just empty, this can be a symptom of high learning rate.

The images that maximally activate the filters for the rest of the layer are shown in FIGURE 18. As the data travels deeper in the network the filters start getting more complex to recognise various patterns. The figure only includes the filters that produces significant activation.


(a) Filter Weights

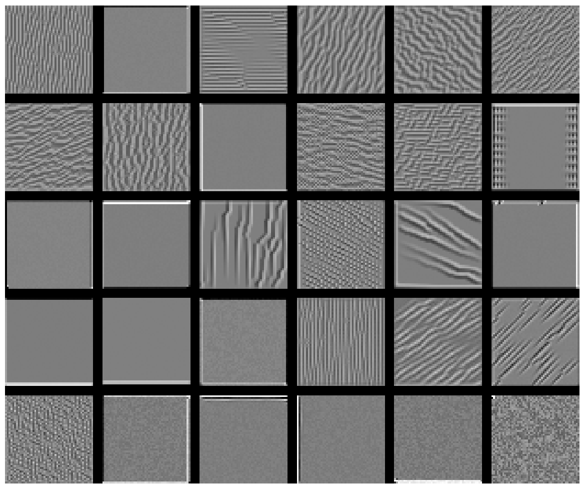
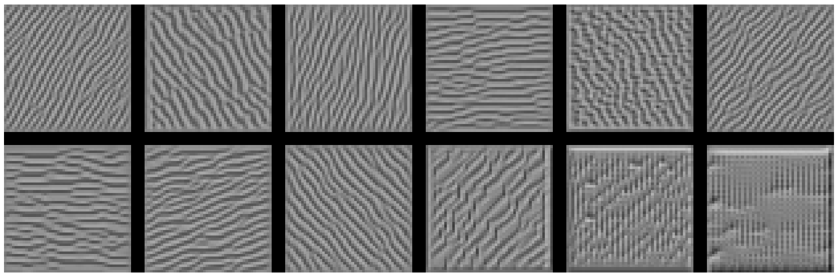
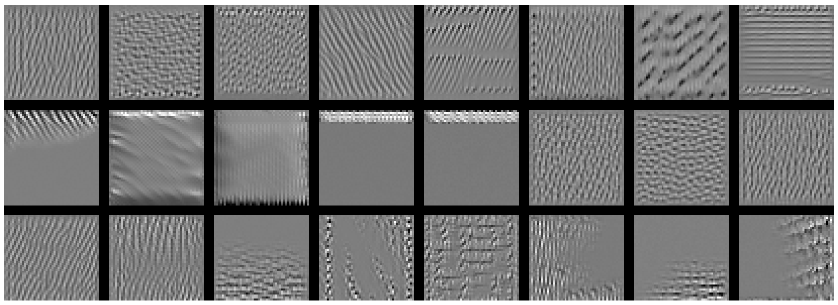
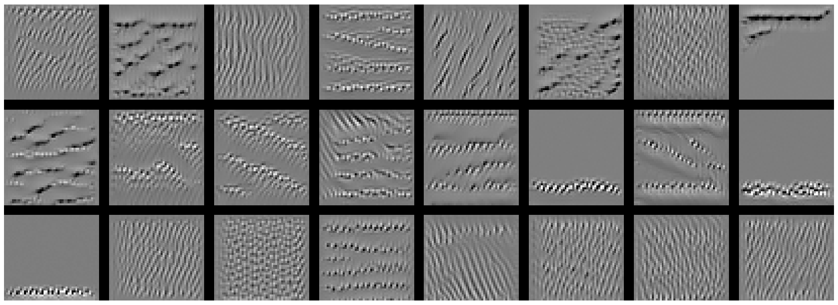
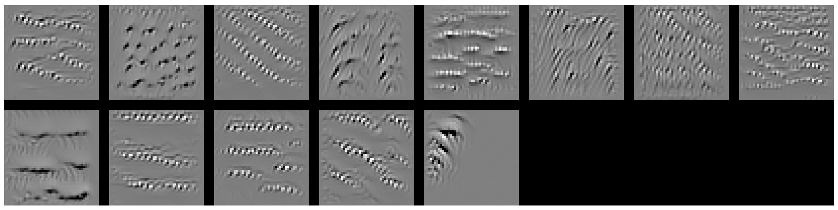

(b) Generated Images

Figure 17: The filter weights of layer 1 and generated images that produce the maximum activation. Note that images do no correspond of the same filters. The weights are of the first 25 filters calculated by the network.


(a) Layer 2 Images


(b) Layer 3 Images


(c) Layer 4 Images


(d) Layer 5 Images

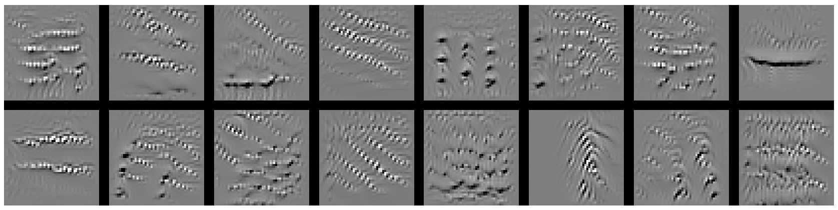

(e) Layer 5 Images

Figure 18: Sample images that produce the maximum activation on filters from convolution layers 2 to 6.

Experimentation with CNNs

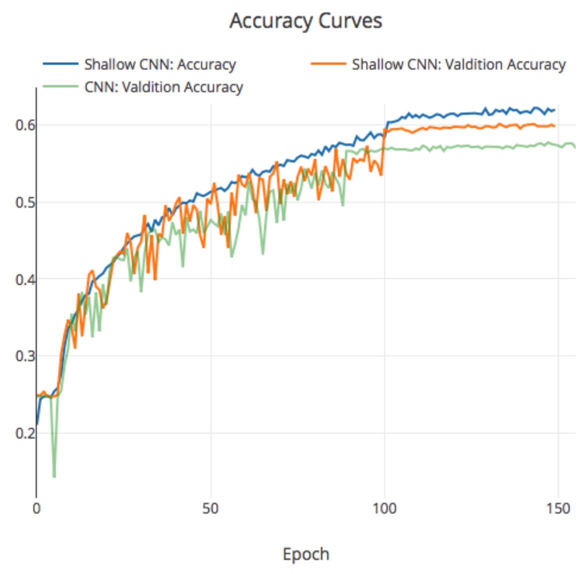
This section contains information about various experiments that were performed on the previously discussed CNN model to get a better understanding of convolution neural networks.

Convolution Layers

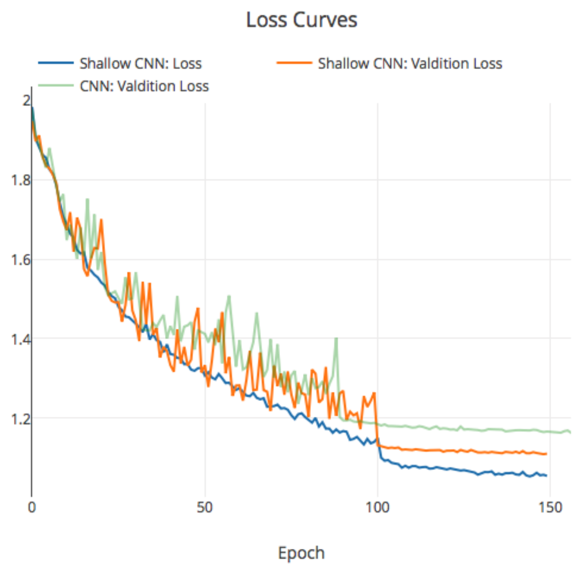
Fist experiments was to test how the *"depth"* of the features extractor part of the neural network i.e. the convolution layers' effects on the performance of the model.

Each pair of convolution layers is replaced with a single convolution layers with the same configuration. This brings a total of 3 convolution layers with 32, 64, 64 3×3 filters respectively and 2 dense fully-connected 512-node layers at the top with softmax as the final output layer. The training takes 7 seconds for each epoch on the same hardware and was trained for 100 epochs with on the same data. The results of this are show in FIGURE 19. From the figure it is evident that both the models perform similarly except the shallower model has a higher accuracy asymptote. The shallower model starts training with better accuracy and has better accuracy gradient over epochs; this is perhaps due to the model having fewer randomly initialised parameters for convolution filters and the error propagated backwards is stronger. After 150 epochs the validation accuracy of shallower network is 0.5980 compared to 0.5746 and 0.5787 of deeper model after 150 and 200 epochs. It should be noted that the shallower model is worse at generalising since the training set's accuracy and loss is better than validation set's.

To continue with the experiments, the architecture was modified slightly by removing the 3rd convolution layer. That leaves the network with 2 convolution layer with 32 and 64 3×3 filters respectively and 2 fully-connected dense layer. First model was trained with the same parameters for 50 epochs with each epoch taking 6 seconds. However, the effects of overfitting were evident in the accuracy and loss curves. To counter the overfitting the regularisations of convolution layers was increased from 0.0001 to 0.001 and retrained. Nevertheless the model was still overfitting, it just took larger number of epochs to overfit. The accuracy and loss curves after applying the 0.001 regularisation on the 2 convolution layers is shown in FIGURE 20. To it would appear that reducing the number of convolution layers, hence total count of the filters, results in overfitting because it becomes harder for the network to generalise while reducing the loss in training data. The final validation accuracy of the model after 100 epochs was 0.5590.



(a) Accuracy Curves

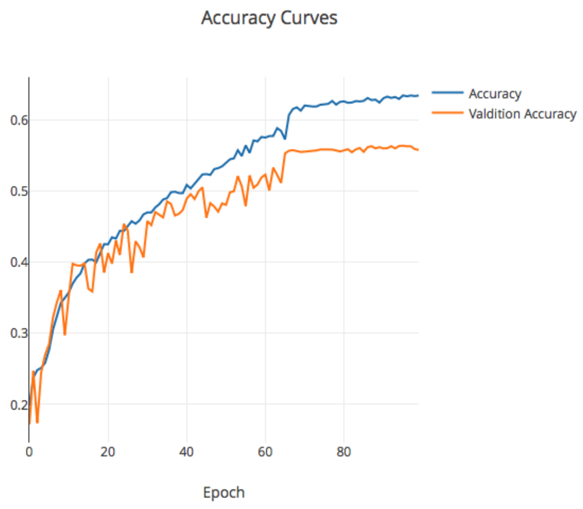


(b) Loss Curves

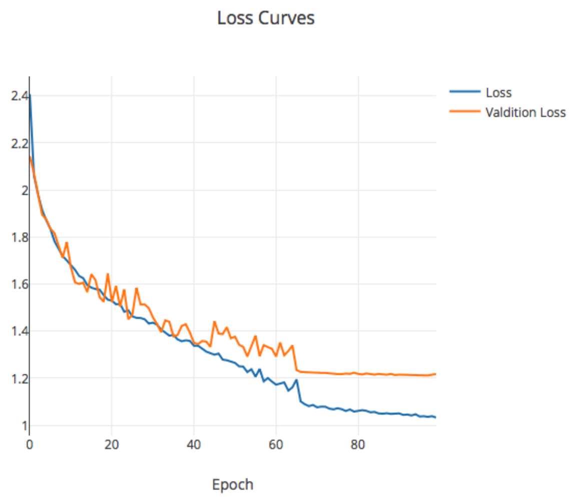
Figure 19: Comparison of accuracy and loss curves of the shallower network with 3 convolution layers instead of 6. The curves are the values on validation set after each epoch. The "CNN“ model is the same one developed and discussed in previous sections with six convolution layers.

Next test was to see the effect of using different kernel size for the convolution layers. So far all the model were using a 3×3 kernels for all the layers. It is common practise to increase the size of kernels as moving deeper in the neural network because larger kernels can learn more complex filters compared to simple filters like an vertical edge detector. Keep the model the same as the shallow network with 3 convolution layers and 2 dense layers, the kernel size for the convolution layers was changed to 32 3×3, 64 5×5, and 64 7×7 respectively. The model was then trained with the same data and each epoch took 9 seconds to complete. The model performed similar to the shallow network 3×3 kernel model except that it generalises better. The separation between the training and validation accuracy curves is minimal but still not as good as the deep CNN which has better accuracy on the validation data. Therefore, larger filter sizes helps the neural network generalise better with limited number of convolution layers. The final validation accuracies of the models with increasing size kernels and constant kernels as 0.5897 and 0.5980, respectively.

The images that produce the maximum activation of these filters are shown in Figure 22. As expected, the filters are more complex than with constant kernel sizes. Furthermore, note that the fewer filters are "dead“ compared previous sizes. However, layer two seems to have some noisy filters. Noisy patterns can be an indicator of a network that hasn't been trained for long enough, or a very low regularisation strength was applied to the layer which lead to overfitting.

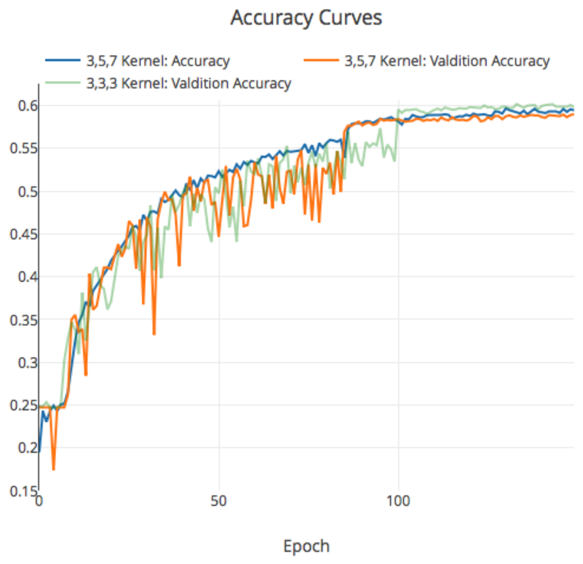


(a) Accuracy Curves

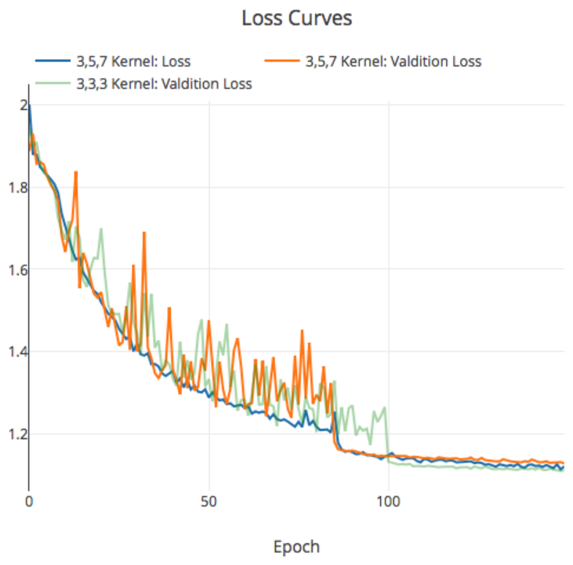


(b) Loss Curves

Figure 20: Accuracy and loss curves of network with only 2 convolution layers with higher regularisation of 0.001.

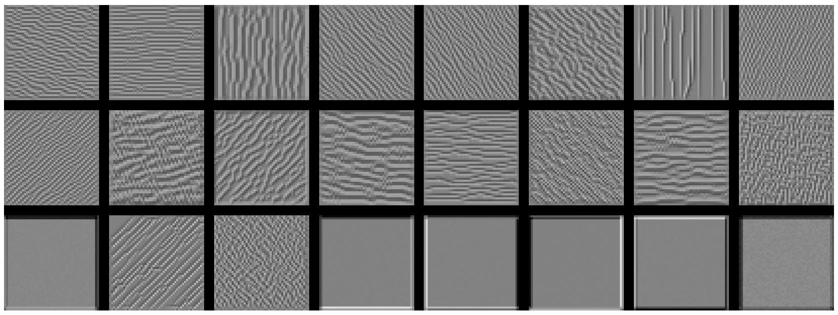


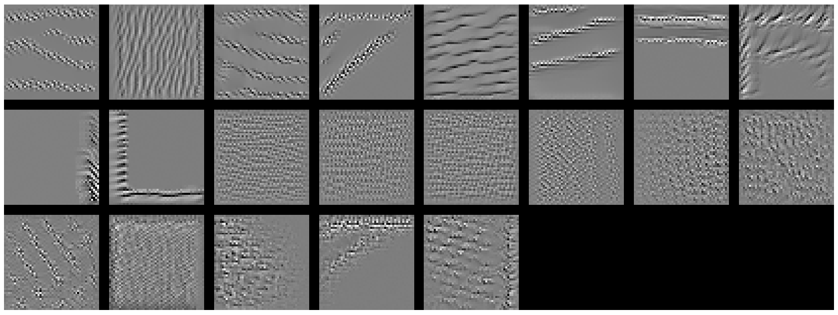
(a) Accuracy Curves



(b) Loss Curves

Figure 21: Accuracy and loss curves of network with increasing kernel size compared against constant kernel size of 3×3. Note that the learning rate of the constant sized kernel model was reduced on epoch 100 and the same reduction was done epoch 85 for variable kernel sized model.


(a) Layer 1 Filters (3 × 3)


(b) Layer 2 Filters (5 × 5)

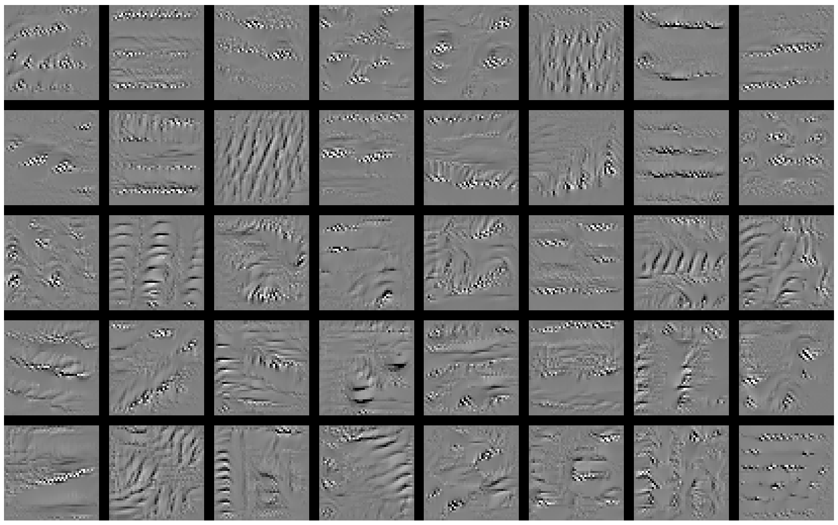
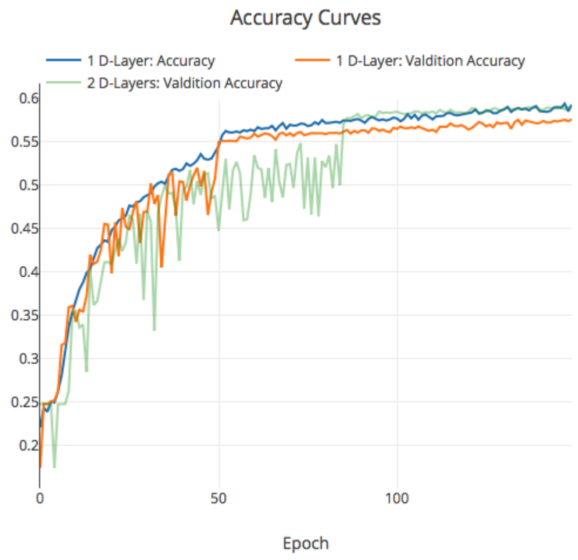

(c) Layer 3 Filters (7 × 7)

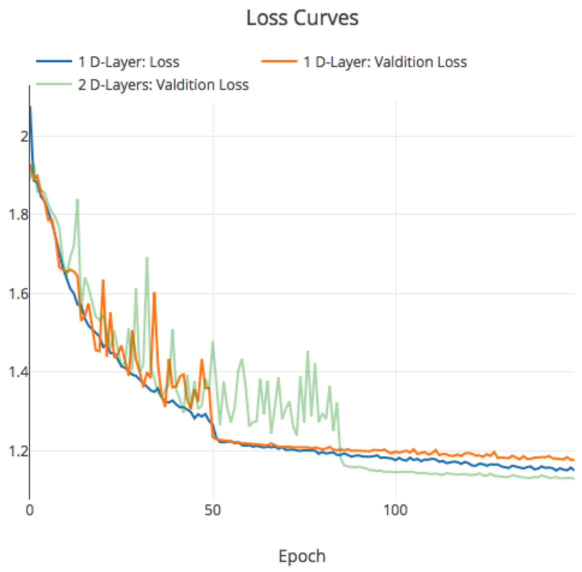
Figure 22: Filters of the shallow model with increasing kernel size.

Dense Layers

Similar to convolution layers, the dense layers were experimented with. First test was to see the effects of removing a dense fully-connected layer. This leaves the neural network with only one dense fully-connected layer with 512-nodes and the output layer with soft-max activation function to output class probabilities. Each epoch took 8 seconds to complete and the model was trained for 150 epochs. The accuracy and loss curves are show in FIGURE 23 and they're compared with previously trained model with two dense layers and increasing kernel size for three convolution layers. This model is worse at generalising than with two dense layers, although the difference is negligible can be possibly reduced by other methods. This is the same as when a convolution layer was removed from the model but the effects are significantly reduced. The final validation accuracy was of 0.5752 compared to 0.5897 with two dense layers of 512 nodes.



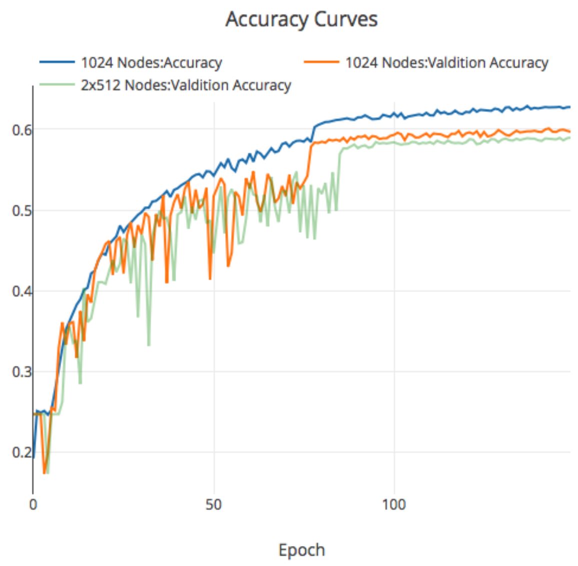
(a) Accuracy Curves



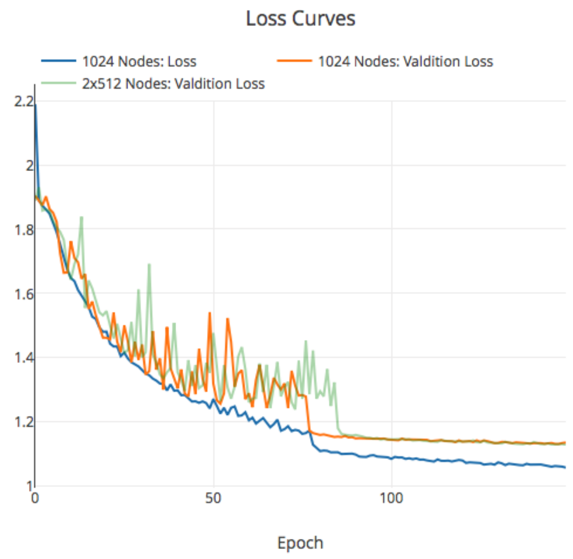
(b) Loss Curves

Figure 23: Accuracy and loss curves of network a single fully-connected dense layer compared to a network with two dense layers. The latter model is the same one with increasing kernel sizes and three convolution layers. The same is true for the single layered network.

Another parameter that can be changed in dense layers is the number of nodes in the dense layer. To test how changing this will affect the network, the number of nodes was increased from 512 to 1024 in the single layer of the previous model. The model again trained for 150 epochs with each epoch taking 8 seconds to update the weights. The results of higher number of nodes is show in FIGURE 24. The final validation accuracy is of 0.5969 compared to 0.5897 of two dense layered network with 512 nodes each. Although there difference between training and validation accuracy is larger. So it can be assumed that with higher number of nodes in a dense layer the network is more likely to overfit but can also learn more complex patterns which lead to higher accuracy.



(a) Accuracy Curves

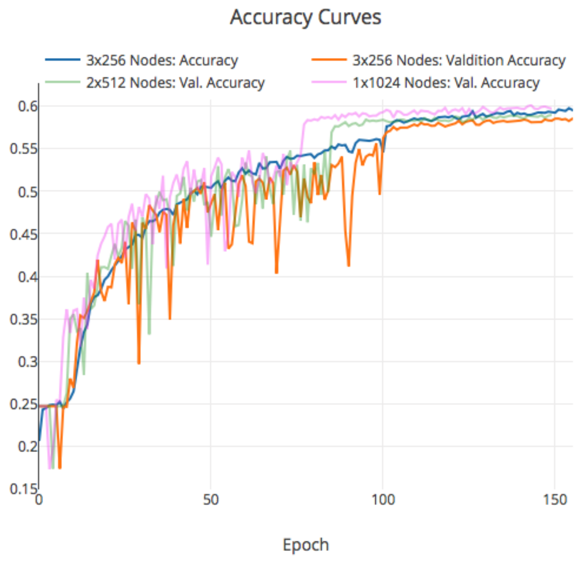


(b) Loss Curves

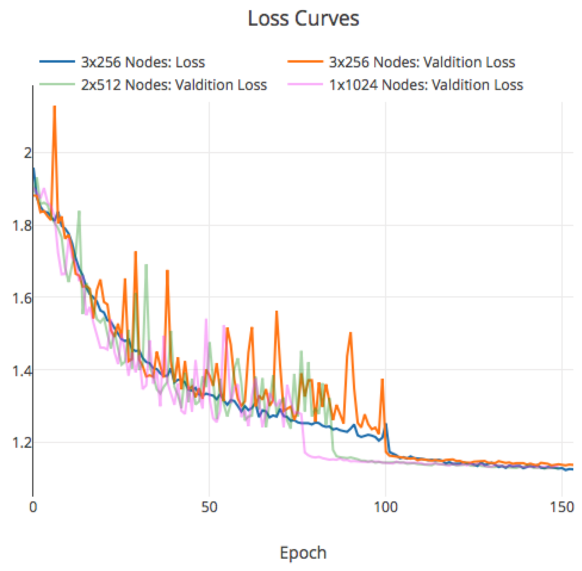
Figure 24: Accuracy and loss curves of network with a single fully-connected dense layer with 512 nodes compared 1024 nodes. The res of the model architecture and the parameters are the same.

Based on the experience from previous two tests on the dense layer, the next model had three dense fully-connected layers with 256 nodes each. The idea behind this new architecture is to reduce overfitting with fewer nodes and increase the complexity with more depth to improve generalisation of classes. The newer model took 8 seconds per epoch and was trained for 150 epochs. The accuracy and validation curves are shown in FIGURE 25. The final validation accuracy of this architecture is 0.5830 compared to 0.5969 and 0.5897 of single dense layers 1024-nodes and double dense layers with 512-nodes. This architecture has generalised better than single layered, while has slightly lower accuracy than double layered architecture.

It can be conlcuded that the dense layers architecture makes little difference as long as there are enough nodes to generalise the classification function.



(a) Accuracy Curves



(b) Loss Curves

Figure 25: Accuracy and loss curves of network with 3 dense layers consisting of 256 nodes each.

Conclusion

The final model attained an accuracy of 0.58 on the stratified validation set which was 25% of the total dataset. For reference, this dataset is from a Kaggle competition where the highest accuracy was of 0.71 and 0.58 would come in at 15th place out of 56 participating teams. This model can certainly be improved further. It would help to have more number of samples for classes that are under-represented in the dataset like *"Disgust"*. If it is not possible to get more training data then over-sample those classes using an image generator. Image generators are common in machine learning, they take a set of images and apply transformations on them to generate variations of the images. These transformations can include rotation, horizontal and vertical flip, zoom, and translate. This would help the model generalise and will limit overfitting if new images are generated in each epoch.

The model would also benefit from using *cross-validation* where rather than having separate training and validation sets, the validation set is randomly chosen from the whole set every epoch. This lets the model train on all the available data.

Since the data is limited in this case, it might be better option to use *support vector machines* (SVM) with CNNs. SVM is a relatively new supervised machine learning algorithm which can be applied for classification. They were created for binary classification but can be used for multi-class classification. The two common approaches are *One-against-All* where an N-class dataset is split into N two-class dataset and *One-against-One* where a model is created for each pair of classes resulting in $N(N-1)/2$ classes.[16] The winning team for the Kaggle competition used a CNN with linear One-against-All SVM at the top i.e. SVM layer instead of the output softmax layer.[17] The difference between softmax and multi-class SVMs is in their objectives parametrised by all the weight matrices. Softmax layer tries to minimise the cross-entropy cost, while SVMS try to find the maximum margin between data points of different classes. The paper also mentions applying preprocessing to the training data which could have helped this model as well.

[16] Gidudu Anthony, Hulley Greg, and Marwala Tshilidzi. *Classification of Images Using Support Vector Machines*, 2007

[17] Y. Tang. Deep Learning using Linear Support Vector Machines. *ArXiv e-prints*, June 2013

It is not common to train entire convolutional-neural-networks from scratch with randomly initialised filters because of the amount of data required. There is another technique which edits pre-trained models called *transfer learning*. The per-trained models are usually made from very large datasets for classification problems e.g. ImageNet has 14 million images in 20,000 categories. It takes multiple-weeks to train these across an array of GPUs. This pre-trained model can then be used as the initialisation or a fixed feature extractor for the task. It has the benefit of already knowing the convolution filters that work. However, it does have some limitation, the model's architecture cannot be changed entirely so the model might be too heavy for some tasks such as live predictions.

References

Gidudu Anthony, Hulley Greg, and Marwala Tshilidzi. *Classification of Images Using Support Vector Machines*, 2007.

Google Brain et al. Tensorflow. `https://www.tensorflow.org`, 2015.

Pierre-Luc Carrier and Aaron Courville. Facial expression recognition challenge, 2013. URL `https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/`.

Maureen Caudill. *Neural Network Primer: Part I.* AI Expert, 1989.

François Chollet et al. Keras. `https://keras.io`, 2015.

Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017.

Prashant Gupta. Regularization in machine learning, 2017. URL `https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a`.

Andrej Karpathy. Visualizing what convnets learn. Technical report, Stanford University, 2017. URL `http://cs231n.github.io/understanding-cnn/`.

Phil Kim. *MATLAB Deep Learning: With Machine Learning, Neural Networks and Artificial Intelligence*. Apress, 2017.

Alex Krizhevsky. Imagenet classification with deep convolutional neural networks. Technical report, Univeristy of Toronto, 2012.

Alok Kumar. Back-propagation algorithm and bias | neural networks, 2017. URL `https://medium.com/machine-learning-bootcamp/neural-networks-back-propagation-algorithm-and-bias-5acd9e6b42e9`.

Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus Müller. Efficient backprop. *Neural Networks: tricks of the trade*, 1998.

Satya Mallick. Neural networks: A 30,000 feet view for beginners, 2017. URL `https://www.learnopencv.com/neural-networks-a-30000-feet-view-for-beginners/`.

University of Wisconsin. A basic introduction to neural networks, 2013. URL `http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html`.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning representations by back-propagating errors*. Nature, 1986.

Avinash Sharma. Understanding activation functions in neural networks, 2017a. URL `https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0`.

Sagar Sharma. Activation functions: Neural networks, 2017b. URL `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

Charlie Tang. Human performance | facial expression recognition challenge, 2013. URL `http://bit.ly/13Zr6Gs`.

Y. Tang. Deep Learning using Linear Support Vector Machines. *ArXiv e-prints*, June 2013.

Appendix A: Details of Model

This appendix has in-depth details of the final neural network. The code for this can be found on `https://github.com/lordlycastle/B31RA-Facial-Expression-Recognition`.

1. *Learning Rate*: 0.001 and reduce on plateau with minimum of 0.0001
2. *Gradient Descent Optimiser:* RMSprop
3. *Loss Function:* Categorical Cross-Entropy
4. *Metrics:* Categorical Accuracy
5. *Batch Size:* 2048

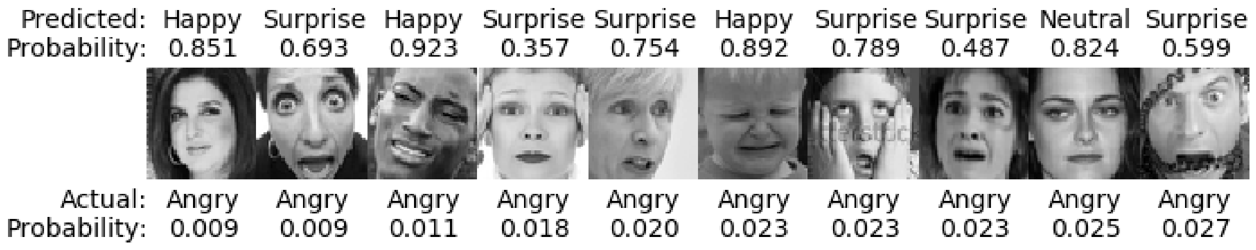
Table 4: Keras layers of the final convolutional neural network. The dropout layers are only active while training.

Layer	Description	$\phi(\cdot)$	Parameters
Conv2D	32 filters (3x3), $\lambda = 0.001$	ReLU	320
Conv2D	32 filters (3x3), $\lambda = 0.001$	ReLU	9248
MaxPooling2D	1 filter (2x2)		0
Dropout	25% of nodes		0
Conv2D	64 filters (3x3), $\lambda = 0.001$	ReLU	18496
Conv2D	64 filters (3x3), $\lambda = 0.001$	ReLU	36928
MaxPooling2D	1 filter (2x2)		0
Dropout	25% of nodes		0
Conv2D	64 filters (3x3), $\lambda = 0.001$	ReLU	36928
Conv2D	64 filters (3x3), $\lambda = 0.001$	ReLU	36928
MaxPooling2D	1 filter (2x2)		0
Dropout	25% of nodes		0
Dense	512 nodes	ReLU	524800
Dropout	50% of nodes		0
Dense	512 nodes	ReLU	262656
Dropout	50% of nodes		0
Output	7 nodes	Softmax	3591

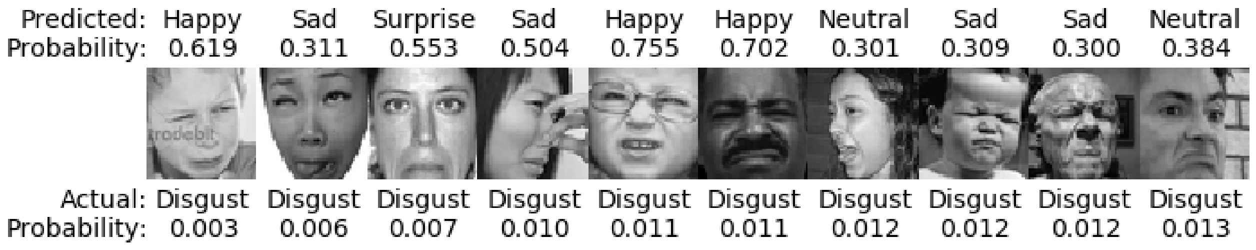
Keras Code

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Reshape, Conv2D, Dropout, \
    MaxPooling2D
from keras import metrics
from keras import regularizers

cnn_model = Sequential()
cnn_model.add(Reshape((48, 48, 1), input_shape=(dimData,)))
cnn_model.add(Conv2D(32, (3, 3),
                     padding='same',
                     activation='relu',
                     kernel_regularizer=regularizers.l2(0.001)
                     ))
cnn_model.add(Conv2D(32, (3, 3),
                     activation='relu',
                     kernel_regularizer=regularizers.l2(0.001)
                     ))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model.add(Dropout(0.25))

cnn_model.add(Conv2D(64, (3, 3),
                     padding='same',
                     activation='relu',
                     kernel_regularizer=regularizers.l2(0.001)
                     ))
cnn_model.add(Conv2D(64, (3, 3),
                     activation='relu',
                     kernel_regularizer=regularizers.l2(0.001)
                     ))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model.add(Dropout(0.25))

cnn_model.add(Conv2D(64, (3, 3), padding='same',
                     activation='relu',
                     kernel_regularizer=regularizers.l2(0.001),
                     ))
cnn_model.add(Conv2D(64, (3, 3),
                     activation='relu',
                     kernel_regularizer=regularizers.l2(0.001)
                     ))
cnn_model.add(MaxPooling2D(pool_size=(2, 2)))
cnn_model.add(Dropout(0.2))

cnn_model.add(Flatten())
cnn_model.add(Dropout(0.5))
cnn_model.add(Dense(512,
                    activation='relu',
                    )
              )
cnn_model.add(Dropout(0.5))
cnn_model.add(Dense(512,
                    activation='relu',
                    )
              )
cnn_model.add(Dropout(0.5))
cnn_model.add(Dense(nClasses, activation='softmax'))

cnn_model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
                  metrics=[metrics.categorical_accuracy, 'acc'], )

cnn_history = cnn_model.fit(train_data, train_label_1hot, batch_size=2048,
                            epochs=100,
                            validation_data=(test_data, test_label_1hot),
                            verbose=True,
                            callbacks=[ReduceLROnPlateau(min_lr=0.0001)],
                            )
```

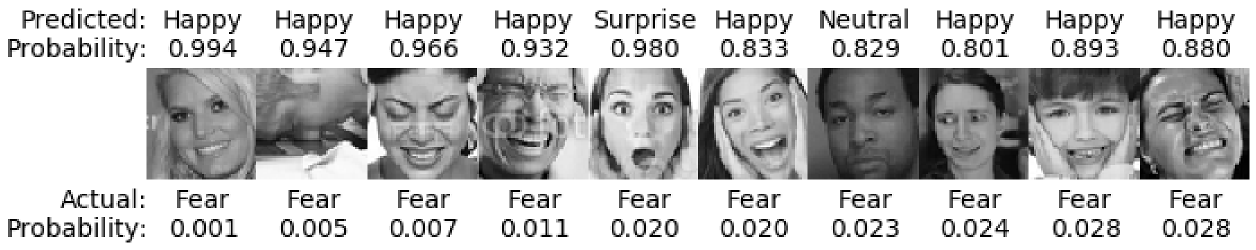
Appendix B: Example Predictions


Predicted: Happy Surprise Happy Surprise Surprise Happy Surprise Surprise Neutral Surprise
Probability: 0.851 0.693 0.923 0.357 0.754 0.892 0.789 0.487 0.824 0.599
Actual: Angry Angry Angry Angry Angry Angry Angry Angry Angry Angry
Probability: 0.009 0.009 0.011 0.018 0.020 0.023 0.023 0.023 0.025 0.027

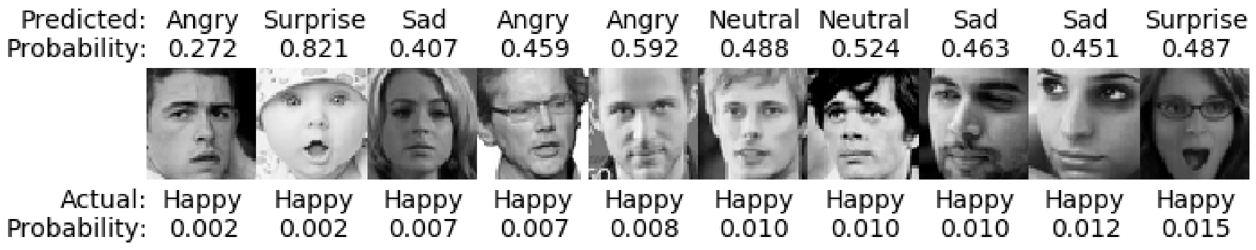
(a) Angry


Predicted: Happy Sad Surprise Sad Happy Happy Neutral Sad Sad Neutral
Probability: 0.619 0.311 0.553 0.504 0.755 0.702 0.301 0.309 0.300 0.384
Actual: Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust
Probability: 0.003 0.006 0.007 0.010 0.011 0.011 0.012 0.012 0.012 0.013

(b) Disgust

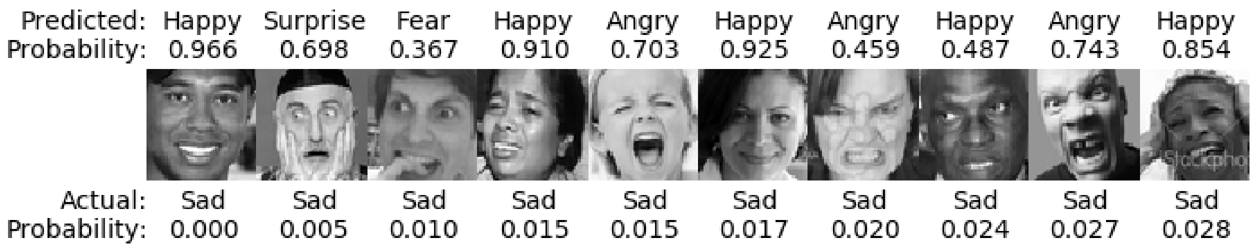

Predicted: Happy Happy Happy Happy Surprise Happy Neutral Happy Happy Happy
Probability: 0.994 0.947 0.966 0.932 0.980 0.833 0.829 0.801 0.893 0.880
Actual: Fear Fear Fear Fear Fear Fear Fear Fear Fear Fear
Probability: 0.001 0.005 0.007 0.011 0.020 0.020 0.023 0.024 0.028 0.028

(c) Fear

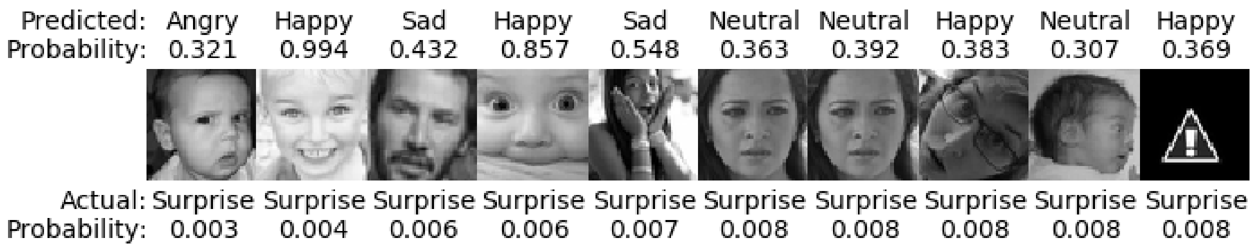

Predicted: Angry Surprise Sad Angry Angry Neutral Neutral Sad Sad Surprise
Probability: 0.272 0.821 0.407 0.459 0.592 0.488 0.524 0.463 0.451 0.487
Actual: Happy Happy Happy Happy Happy Happy Happy Happy Happy Happy
Probability: 0.002 0.002 0.007 0.007 0.008 0.010 0.010 0.010 0.012 0.015

(d) Happy

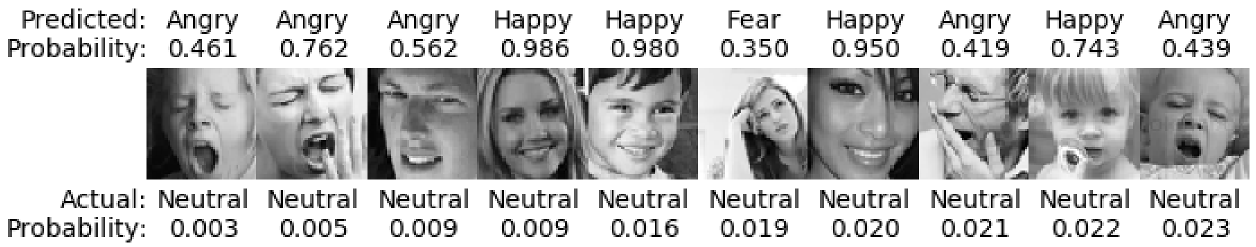
Figure 26: Worst cases for each class. On the top of image are the probabilities calculated for the predicted class, and at the bottom is the correct class's probability.


Predicted: Happy Surprise Fear Happy Angry Happy Angry Happy Angry Happy
Probability: 0.966 0.698 0.367 0.910 0.703 0.925 0.459 0.487 0.743 0.854
Actual: Sad Sad Sad Sad Sad Sad Sad Sad Sad Sad
Probability: 0.000 0.005 0.010 0.015 0.015 0.017 0.020 0.024 0.027 0.028

(e) Sad

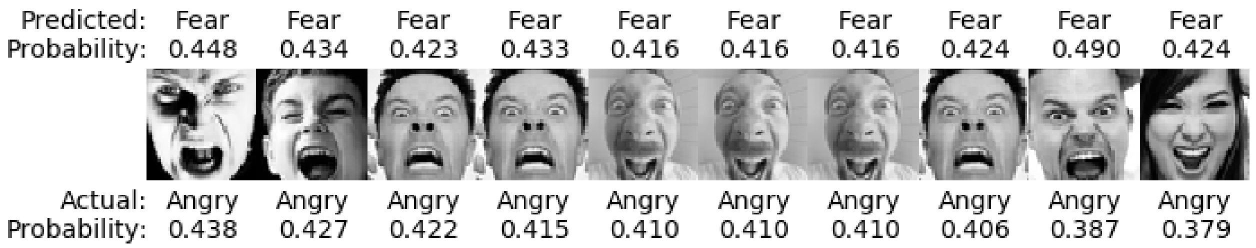

Predicted: Angry Happy Sad Happy Sad Neutral Neutral Happy Neutral Happy
Probability: 0.321 0.994 0.432 0.857 0.548 0.363 0.392 0.383 0.307 0.369
Actual: Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise
Probability: 0.003 0.004 0.006 0.006 0.007 0.008 0.008 0.008 0.008 0.008

(f) Surprise


Predicted: Angry Angry Angry Happy Happy Fear Happy Angry Happy Angry
Probability: 0.461 0.762 0.562 0.986 0.980 0.350 0.950 0.419 0.743 0.439
Actual: Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral
Probability: 0.003 0.005 0.009 0.009 0.016 0.019 0.020 0.021 0.022 0.023

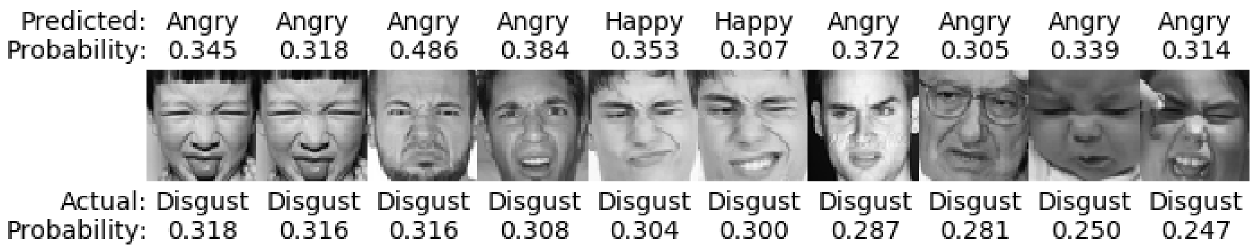
(g) Neutral

Figure 26: Worst cases for each class. On the top of image are the probabilities calculated for the predicted class, and at the bottom is the correct class's probability.

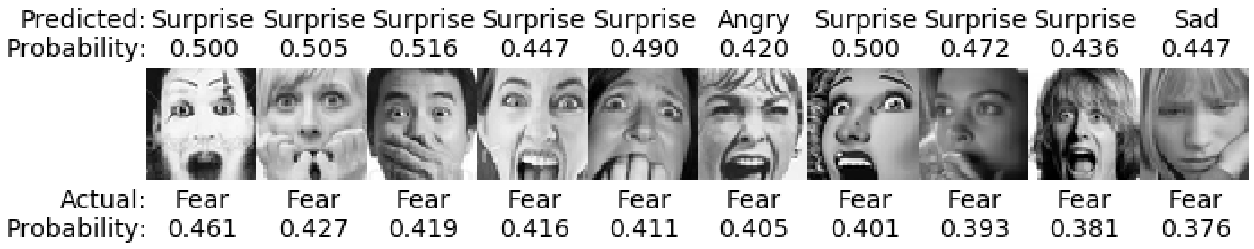

Predicted: Fear Fear Fear Fear Fear Fear Fear Fear Fear Fear
Probability: 0.448 0.434 0.423 0.433 0.416 0.416 0.416 0.424 0.490 0.424
Actual: Angry Angry Angry Angry Angry Angry Angry Angry Angry Angry
Probability: 0.438 0.427 0.422 0.415 0.410 0.410 0.410 0.406 0.387 0.379

(a) Angry

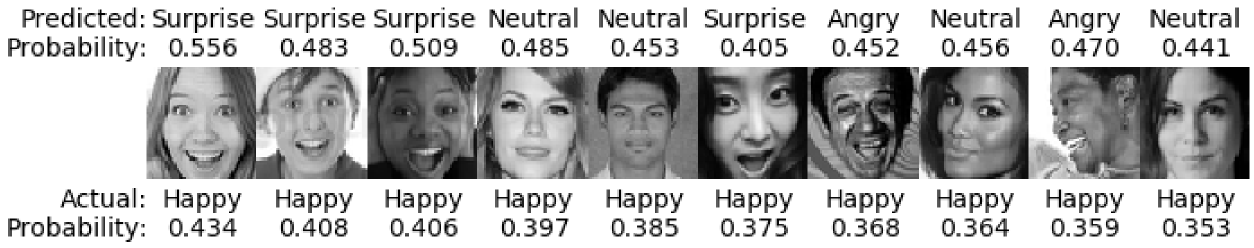
Figure 27: Best examples of incorrect prediction. On the top of image are the probabilities calculated for the predicted class, and at the bottom is the correct class's probability.


Predicted: Angry Angry Angry Angry Happy Happy Angry Angry Angry Angry
Probability: 0.345 0.318 0.486 0.384 0.353 0.307 0.372 0.305 0.339 0.314
Actual: Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust
Probability: 0.318 0.316 0.316 0.308 0.304 0.300 0.287 0.281 0.250 0.247

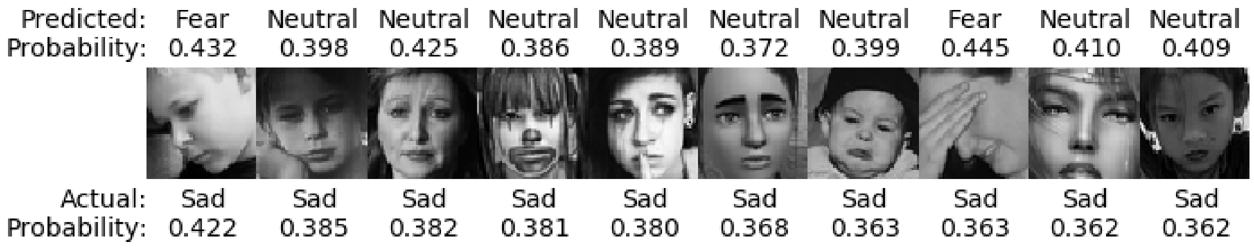
(b) Disgust


Predicted: Surprise Surprise Surprise Surprise Surprise Angry Surprise Surprise Surprise Sad
Probability: 0.500 0.505 0.516 0.447 0.490 0.420 0.500 0.472 0.436 0.447
Actual: Fear Fear Fear Fear Fear Fear Fear Fear Fear Fear
Probability: 0.461 0.427 0.419 0.416 0.411 0.405 0.401 0.393 0.381 0.376

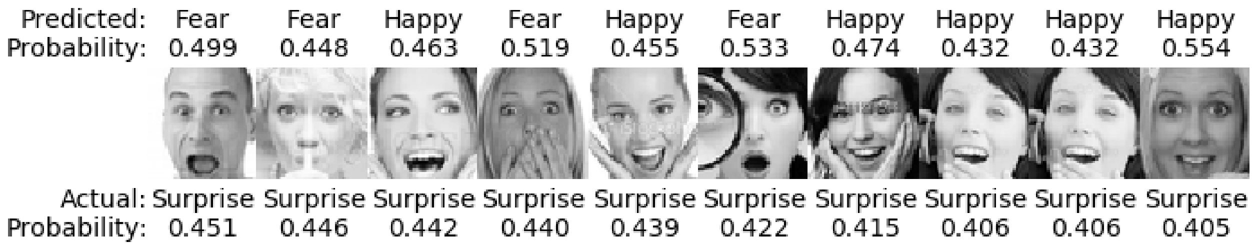
(c) Fear


Predicted: Surprise Surprise Surprise Neutral Neutral Surprise Angry Neutral Angry Neutral
Probability: 0.556 0.483 0.509 0.485 0.453 0.405 0.452 0.456 0.470 0.441
Actual: Happy Happy Happy Happy Happy Happy Happy Happy Happy Happy
Probability: 0.434 0.408 0.406 0.397 0.385 0.375 0.368 0.364 0.359 0.353

(d) Happy

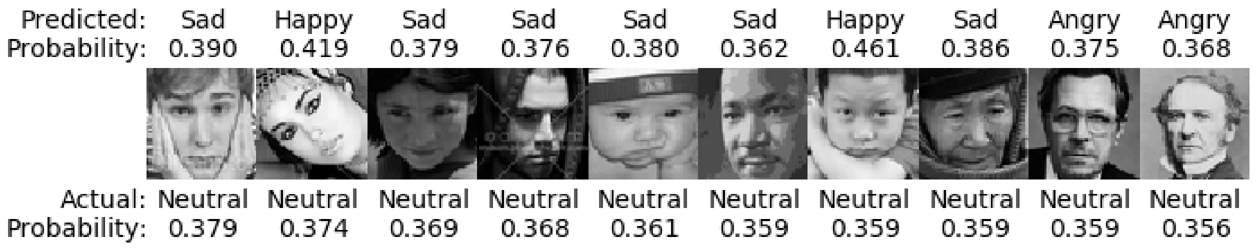

Predicted: Fear Neutral Neutral Neutral Neutral Neutral Neutral Fear Neutral Neutral
Probability: 0.432 0.398 0.425 0.386 0.389 0.372 0.399 0.445 0.410 0.409
Actual: Sad Sad Sad Sad Sad Sad Sad Sad Sad Sad
Probability: 0.422 0.385 0.382 0.381 0.380 0.368 0.363 0.363 0.362 0.362

(e) Sad


Predicted: Fear Fear Happy Fear Happy Fear Happy Happy Happy Happy
Probability: 0.499 0.448 0.463 0.519 0.455 0.533 0.474 0.432 0.432 0.554
Actual: Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise
Probability: 0.451 0.446 0.442 0.440 0.439 0.422 0.415 0.406 0.406 0.405

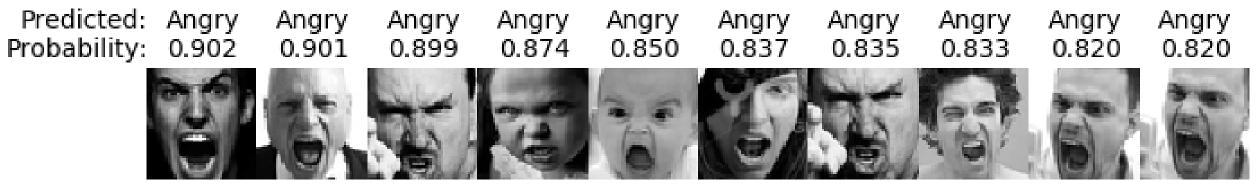
(f) Surprise

Figure 27: Best examples of incorrect prediction. On the top of image are the probabilities calculated for the predicted class, and at the bottom is the correct class's probability.

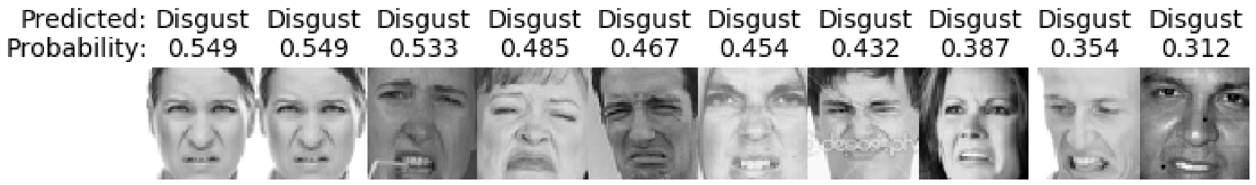

Predicted: Sad Happy Sad Sad Sad Sad Happy Sad Angry Angry
Probability: 0.390 0.419 0.379 0.376 0.380 0.362 0.461 0.386 0.375 0.368
Actual: Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral
Probability: 0.379 0.374 0.369 0.368 0.361 0.359 0.359 0.359 0.359 0.356

(a) Neutral

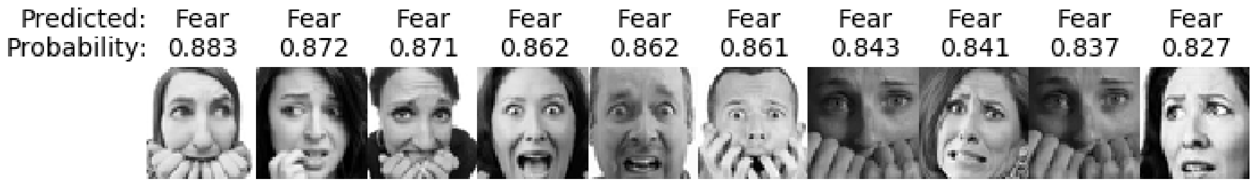
Figure 28: Best examples of incorrect prediction. On the top of image are the probabilities calculated for the predicted class, and at the bottom is the correct class's probability.


Predicted: Angry Angry Angry Angry Angry Angry Angry Angry Angry Angry
Probability: 0.902 0.901 0.899 0.874 0.850 0.837 0.835 0.833 0.820 0.820

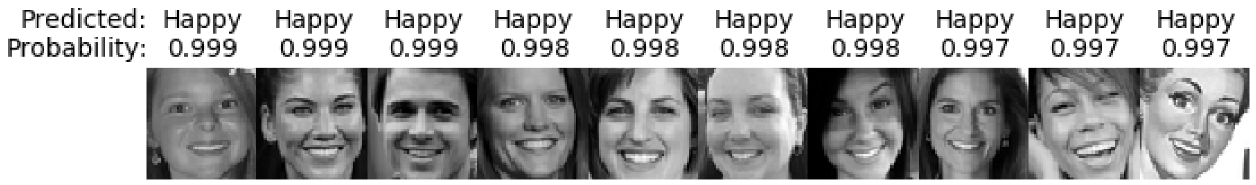
(a) Angry


Predicted: Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust Disgust
Probability: 0.549 0.549 0.533 0.485 0.467 0.454 0.432 0.387 0.354 0.312

(b) Disgust

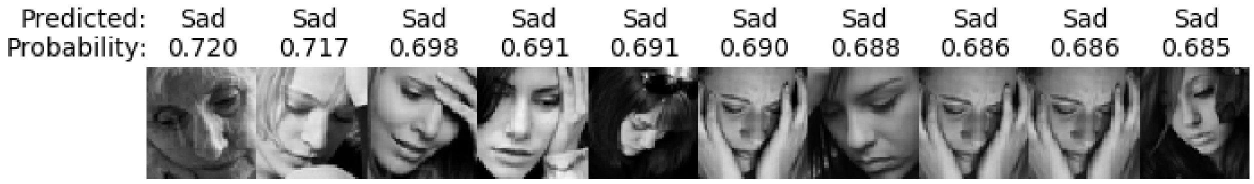

Predicted: Fear Fear Fear Fear Fear Fear Fear Fear Fear Fear
Probability: 0.883 0.872 0.871 0.862 0.862 0.861 0.843 0.841 0.837 0.827

(c) Fear

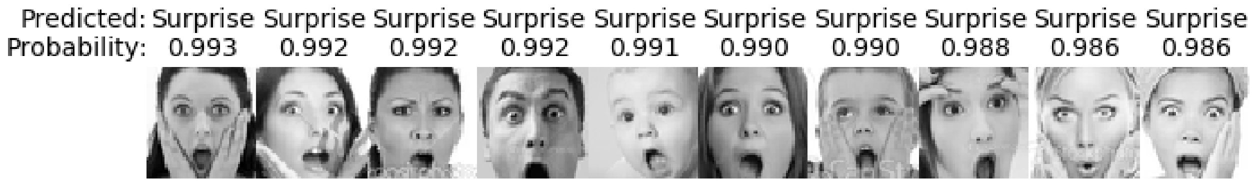

Predicted: Happy Happy Happy Happy Happy Happy Happy Happy Happy Happy
Probability: 0.999 0.999 0.999 0.998 0.998 0.998 0.998 0.997 0.997 0.997

(d) Happy

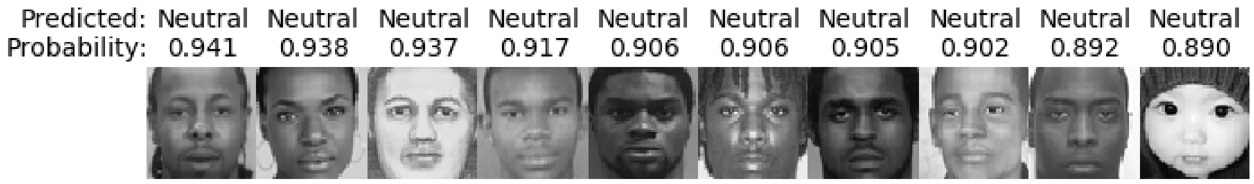
Figure 29: Best cases for each class. On the top of image are the probabilities calculated for the class.


Predicted: Sad Sad Sad Sad Sad Sad Sad Sad Sad Sad
Probability: 0.720 0.717 0.698 0.691 0.691 0.690 0.688 0.686 0.686 0.685

(e) Sad


Predicted: Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise Surprise
Probability: 0.993 0.992 0.992 0.992 0.991 0.990 0.990 0.988 0.986 0.986

(f) Surprise


Predicted: Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral Neutral
Probability: 0.941 0.938 0.937 0.917 0.906 0.906 0.905 0.902 0.892 0.890

(g) Neutral

Figure 29: Best cases for each class. On the top of image are the probabilities calculated for the class.