REPORT FOR INM 702

Programming and Mathematics for Artificial Intelligence

Author: Vasilis Kotsos

Datasets can be found at:

SHVN DATASET: http://ufldl.stanford.edu/housenumbers/

Emotion Dataset: https://www.kaggle.com/jonathanoheix/face-expression-recognition-dataset?

Introduction.

This report aims to tackle the problem of classification. More specifically, Neural Network architectures will be used for classifying images. The first part covers building a Neural Network from scratch, that can classify images of digits. The dataset used in the first part is the SHVN dataset. The second part of the report focuses on building a Neural Network from scratch that implements the SGD with momentum variant. Lastly the third part will focus on the use of Deep Learning frameworks for the creation of more complex models. The dataset used for the second and third parts is an "emotions" dataset. The models will be tasked with learning how to distinguish facial expressions of anger, fear, happiness, and sadness.

Task 1. Implementing Linear and ReLu layers (Tasks 1-4 are implemented in a Jupyter notebook)

1.Forward pass.

The linear transformation, alongside with Rectified Linear Unit (ReLu) activations are the foundations of the feedforward Neural Network. The linear function (f(x) = x) is used in the input layers. The ReLu function is used in all hidden layers. A linear transformation is implemented as follows. The vector x is "transformed" by taking the dot product between it and the matrix U.

s = x * U. Where * stands for the dot product.

The implementation in numpy is given below:

```
trans_vector = np.dot(input_vector, weight_matrix) # matrix transformation
```

The ReLu function is defined as: f(x) = max(0,x)

$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

Assuming that x is a vector of real numbers, the following code will return a vector of non-negative elements and zeros where appropriate.

```
def relu_activation_batch(self,matrix):
```

```
        return np.maximum(0,matrix)
```

The use of non-linear activation functions in NNs is of critical importance, with the use of such activation, complex functions can be modelled by the NNs. Activations such as the sigmoid and ReLu, are the reason why the NNs produce good results. A neural network with only linear activations in th hidden layers would not gain much by having more layers, since all the linear transformations performed by each one could be represented by a single transformation, this would mean that the network would not learn a good mapping between the features and target variables. The activation functions enable data that is not linearly separable, to be embedded in a dimensional space, where they can be separated. Deeper networks can learn more complex functions by composing functions learned in the earlier layers.

2. Backward pass.

The **"**backward pass**"** refers to the operation of back propagation. During this operation, the error that is computed at the end of the forward pass, is back propagated through the network. More specifically, the error is back propagated to every neuron in the Neural Network. Thus, the update to the weights can take place. For this to happen the derivative of the activation function that was used during the forward pass, must be used. The derivative of the ReLu is defined as:

$$\phi'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

The implementation used in python is the following:

```
    def relu_derivative_batch(self, matrix):
        matrix[matrix<=0] = 0
        matrix[matrix>0] = 1
        return matrix
```

Task 2. Implementing dropout.

Neural Network algorithms can suffer from overfitting. A neural network may overfit if it is trained on a training set for long enough. This means that the performance of the NN on out of sample validation data will start to deteriorate while in sample performance is still improving. The NN will end up "memorizing" the training set and will be of limited use when it comes to predicting new instances. This can be mitigated by employing regularization methods. A technique that is widely used for regularization is Dropout. It is one of the most effective and widely used regularization techniques for Neural Networks. This has the effect of training many perturbations of the same NN, which all contribute to the prediction at test time. The version implemented for a NN is the inverted dropout method. The difference being that during training the activations of each layer are scaled during training rather than at test time.

The dropout functionality can be found in the **Dropout_nn** class. The 'dropout_rate' field is the KEEP PROBABILITY variable, thus if it is set to 0.7 then roughly 30% of the neurons will be turned off in each layer. The smaller the number the higher the regularization effect.

1. Forward pass with Dropout.
   The forward pass includes randomly dropping connections on each hidden layer and scaling by the dropout rate, which is usually set to 50%, meaning that about half of the activations of each layer will be set to 0.

The forward pass during training is implemented below: The batch activation must be multiplied by an activation mask that sets activations to 0.

```
activation_mask = (np.random.rand(batch_samples.shape[0],layer)< dropout_rate) /
dropout_rate
```

When it comes to the ReLu activation, dropout can be used either before or after the activation. The resulting vector will have the same components.

```
output_batch = relu_activation_batch(input_batch) # ReLu activation
output_batch = output_batch * activation_masks[mask_counter] #dropout
```

To implement inverted dropout, the activations are scaled during the training phase as shown above.

2. Backward pass:

   During the backward pass, the activation masks must be used when updating the weight matrices.

```
masked_activation = activations[i] * activation_masks[mask_counter] #mask activations
weight_update = np.dot(layer_errors[i], masked_activation)
weights_list[i] -= learning_rate * weight_update.T #take some of the gradient using learning rate
```

The general effect of dropout is that the NN will learn slower during training. However, it should generalize better on the test set, and achieve higher accuracy.

Task 3. Implementing the SoftMax classifier.

Softmax Activation

A popular way to perform multiclass classification with NNs is to pair up the Softmax function along with the negative log-likelihood. The use of both parts makes up what is known as “Cross Entropy Loss function”. The Softmax activation is applied to the final layer of the feed forward NN and can be thought of as normalization layer that takes as input the logit outputs of the NN and normalizes them in a way that allows us to interpret them as probabilities for each class. The final layer should have as many neurons as classes and the error is calculated between the normalized values and the ground truth vector, to minimize the empirical error.

The Softmax function takes a vector as input and is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Softmax Numerical Issues

The Softmax function suffers from underflow and overflow problems. The problem arise in the exp() function which calculates the natural logarithm of the input. This can result into 2 distinct problems, either the input is very negative, in which case exp() will underflow, this means that the denominator will become a very small number, that can only be represented by a regular computer as zero, thus ending up with an undefined expression, which is division by zero!. Similarly, when the input is very large and positive the expression is again undefined. These edge cases can be tackled by implementing a numerically stable SoftMax. This version of the SoftMax shifts the distribution by subtracting the highest component in the vector from each component, thus

eliminating the problems of underflow and overflow. The implementation used is given below. A new vector Z is computed, and Softmax is applied on it.

```
def softmax_activation_batch(self, matrix):
    z = matrix - np.max(matrix, axis=-1, keepdims=True) #prevent overflow here, with this
    numerator = np.exp(z)
    denominator = np.sum(numerator,1)
    denominator = denominator.reshape(matrix.shape[0],-1)
    probs = numerator/denominator
    return probs
```

Negative Log likelihood / Cross-Entropy

The cross-entropy loss between the output of the Softmax activation and the ground truth vector, measures how different the two vectors are. The optimization process of Gradient Descent, will, over several iterations close the gap between the predicted and the ground truth vector. The ground truth vector is a one-hot-encoded vector, which means it only holds a single positive value of 1, while all other indices hold zeros. The predicted value for a sample from the NN is represented as $\hat{y}$ and the ground truth value as y. The cross entropy between the two distributions is defined as:

$$-\sum_{j=1}^{M} y_j \log \hat{y}_j$$

The sum refers to each sample in the training set. The below code implementation is used to calculate the error for an batch of the training set.

```
def calculate_cost_batch(self, probs, labels):
    losses = labels * np.log(probs+ 1e-5) # works against underflow
    batch_loss = - losses.sum()
    return batch_loss
```

A small constant is also added to prevent against underflow. This simply has the effect of shifting the whole distribution slightly, so analytically the function is not changed if we simply add or subtract a scalar from the vector.

Gradients and Back-propagation

- Calculation of error in the output layer.
- Calculation of error in the hidden layers by multiplying the errors with the transpose of the weight matrix.
- Calculate the gradient vector for each weight matrix, the activation of the previous layer, and the error of the next layer.

The back-propagation routine starts by calculating the error in the output layer. By taking into consideration the derivative of the SoftMax. The output layer error turns out to be $\hat{y} - y$. This is the so called ‘canonical link’ function, that Softmax and Negative log likelihood make up.

Implementation:

```
    output_layer_error = batch_probs - batch_labels  # error to propagate
```

The back propagation is achieved by multiplying the error with the transpose of the weight matrix successively for each layer, moreover the derivative of the activation function is considered. All layers except for the input layer are given an error.

Implementation:

```
            error_term = np.dot(weights_list[i],error_l.T)
                derivative_term = self.relu_derivative_batch(hidden_activations[i].T)
                #element-wise multiplication for the full error expression
                error_l_minus = error_term * derivative_term
                layer_errors.append(error_l_minus)
```

The update for each weight matrix, works out to be the activation of the previous layer times the error of the next. The update is then multiplied by a small value called the learning rate.

```
                weight_update = np.dot(layer_errors[i],activations[i])
 weights_list[i] -= learning_rate * weight_update.T #take some of the gradient using learning rate
```

Task 4. Implementing fully connected NN

There exist 3 versions of implemented NN.

- The simple version
- The Dropout version
- The L2 regularized version

The network architecture is quite flexible in the number of hidden layers and the number of units in each layer. (Example use of the NN class)

neural_net = nn(architecture = [1024,200,100,10], bias = False, activation = 'RELU', learning_rate = 0.0001)

neural_net = nn(architecture = [1024,100,10], bias = False, activation = 'RELU', learning_rate = 0.0001)

The output layer should always consist of 10 neurons, since the number should match the number of distinct classes in the dataset.

Weight initialization

The weights of the NN classes are initialized “smartly”. The network performs better if the weights have been initialized by the Normal Xavier Initialization method. Where a matrix W is initialized from a distribution with mean zero and a standard deviation of:

$$\sqrt{\frac{2}{size^{[l-1]} + size^{[l]}}}$$

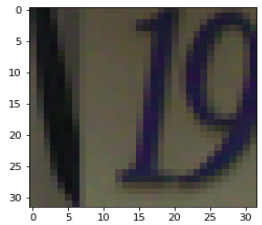
This standard deviation depends on the size of the layers that the weight matrix connects. Below is the implementation code:

```
weight_matrix = np.random.normal(loc=0.0,scale=2/np.sqrt(self.architecture[_]+self.architecture[_+1]),size=(self.architecture[_],self.architecture[_+1]))
```

Dataset Description and Processing:

The pre-processing can be found in the Jupyter Notebook "Data processing (SVHN)"

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms. The dataset includes over 600k digit images that have been obtained from house numbers in Google Street View images. For the purposes of task 4. The dataset files used are: **train_32x32.mat** and **test_32x32.mat**. The data files must be processed before they can be used for training a NN. Several steps were taken to make the data appropriate for ML usage. The images are in the form of 32-by-32 pixels in RGB format. The images have been processed into a grayscale format, so that different RGB channels are not considered. The grayscale images are presented by 1024 pixels, each of which holds a value between 0 and 255. This is not an ideal representation of the data as the NN might consider features that have higher values more 'important', furthermore this range of values could present problems to a NN, such as neuron saturation. To combat this, each pixel is normalized, and the resulting range is between 0 and 1. Lastly the target values have been changed into one-hot-encoded values to match the output dimensions of the NN. The dataset is divided into a training set consisting of 73257 samples and a test set consisting of 26032 samples. This procedure is applied to both the train and test set.



Pre-processed Digit sample.

Grayscale image of Digit sample

Lastly the pixels are flattened into an array to match the input layer of the NN. So, from (32,32,1) they are flattened into (1024,1), representing one image. The entire training set dimensions therefore are (1024,73257) and the test set are (1024,26032).

Experiment setup and setting the training parameters. (The below results can be found in "Tasks 1-4" notebook.)

The below training parameters are used across all training routines.

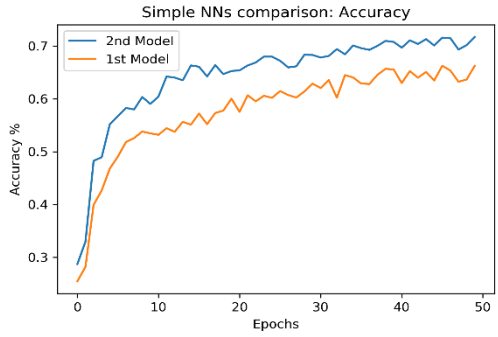
Training Parameters	---
Epochs	50
Sample Batch Size	256
Learning Rate	0.0001

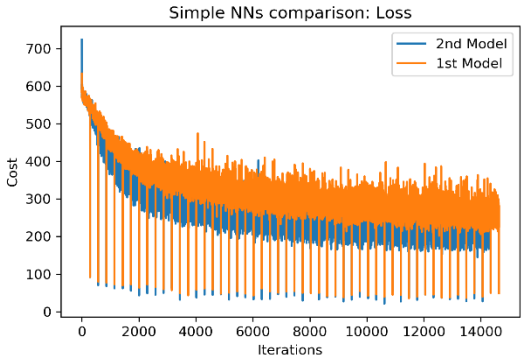
As a start a simple NN was used with 2-hidden layers each one using 100 neurons, furthermore a deeper NN was used, in this instance the comparison of the architecture takes center stage, and whether a deeper Network is beneficial. The deeper NN proves to have higher performance.

Simple NN Models	Architecture	Training Accuracy

1st model	[1024,100,100,10]	65%
2nd model	[1024,200,200,200,10]	70%

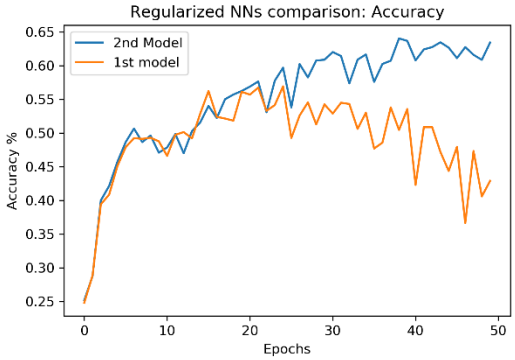
Simple NN models, Result Graphs.

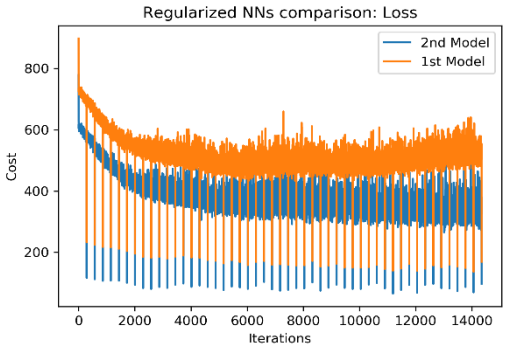




Next, reverting back to the simpler architecture of [1024,100,100,10], the effect of L2 regularization becomes apparent with different settings.

Regularized NN Models	Regularization Parameter	Training Accuracy
1st model	L2 = 0.5	43%
2nd model	L2 = 0.1	65%





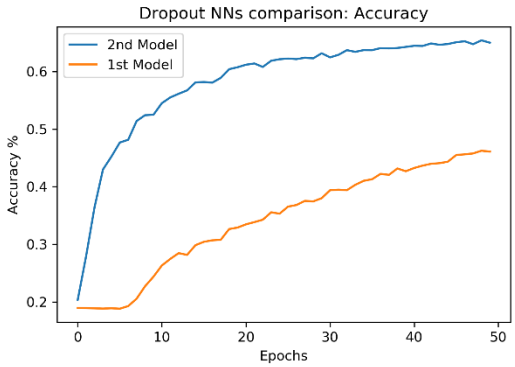
Finally, the Dropout approach is shown to have similar results. High regularization settings impede the training of the NN, as shown below.

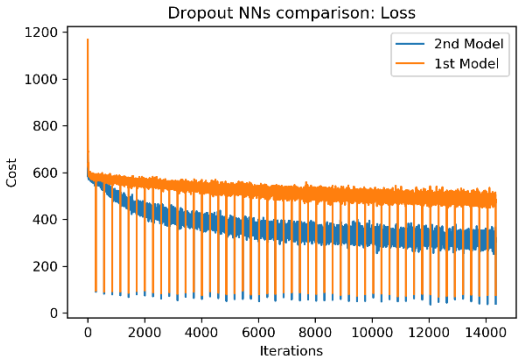
Dropout NN Models	Dropout Parameter
1st model	Drop rate = 0.4
2nd model	Drop rate = 0.8

NN Models	Testing Accuracy
1st simple NN	62%
2nd simple NN	66%
1st Regularized Model	45%
2ND Regularized Model	65%
1st Dropout Model	46%
2nd Dropout Model	65%

A table is given with the testing results of all models used for Task 4. The winner is the 2nd simple model which employed the deeper architecture [1024, 200,200,200,10]. All other models employ the simpler architecture of [1024,100,100,10]

Dropout NNs result graphs.





Discussion of results.

It is evident from the results that the deeper networks benefit from the increased number of neurons. The deeper NN exhibits better training and testing accuracy, as well as less cost. Both models can benefit from more training epochs, as the accuracy metric has not plateaued yet. When it comes to the regularized L2 models, it is obvious that a high regularization rate is detrimental to training. The model with the highest regularization parameter, which is set to 0.5 only reaches up to 55% accuracy on the training set, this means that it is constrained by the regularization parameter, whereas the less regularized model can reach much higher training accuracy, and can produce better results as it gets more complex. It is also apparent that the model can benefit a lot, from a longer training routine of more than 50 epochs. Finally, for the Dropout models, a huge difference in Dropout rate was chosen (from 0.4 to 0.8), to show that Dropout can also have a detrimental effect. The 1st model with Drop_rate set to 0.4, will shut down about 60% of the neurons of each layer, this will cause the NN to learn from the dataset much slower, which is evident in the accuracy graph. The 2nd model displays a much more rapid increase in accuracy as it only has about 20% of the neurons on each layer shut off. This allows the network to grow in complexity, while being resistant to overfitting the data. Overall for this dataset, it seems that the best route to take in order to achieve maximum accuracy, would be to first create a decently deep NN, train it over a few hundred epochs, and then finally employing dropout in order to prevent overfitting of the training data. The models presented here can achieve accuracy above 80% if used correctly.

Task 5. Optimization of hyper-parameters(Implementations in "Task 5 Notebook").

Emotion dataset link: https://www.kaggle.com/jonathanoheix/face-expression-recognition-dataset?

<u>Dataset pre-processing.</u> The processing code is included in "Data processing for the Emotions dataset" notebook.

The dataset is comprised of seven classes (anger, disgust, fear, happy, neutral, sad, surprise), of which only 4 were chosen: angry, happy, fear, sad. This was done, to keep the computational time required within reasonable bounds. The dataset is made up of images that express each emotion. Each image has dimensions of 48 by 48 pixels in greyscale format. This means that they require a small processing routine. The images features that will be fed into the NNs are the individual pixels, which means that they require normalization just like the SHVN digit images. The dataset has many 'edge' cases, where the emotion of an image is not appropriately converted, however, a Neural Network classifier can find separability among the classes.

<u>Dataset examples and training and testing datasets.</u>

"Angry "example

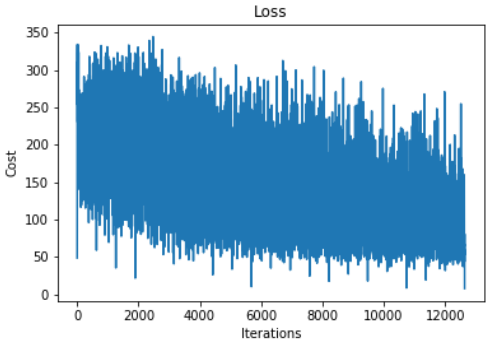
"Fear" example

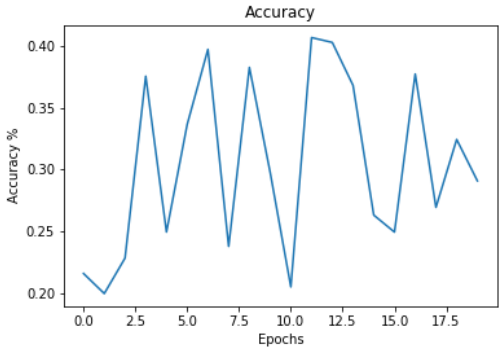
<u>The use of Momentum in Stochastic Gradient Descent</u>

The convergence of SGD can be accelerated with the use of momentum. The momentum applied to the SGD will result in bigger steps being taken during the first iterations and smaller steps during latter iterations. One drawback that is found in this technique, is that during later iterations it can overshoot the minimum, however in most cases this algorithm provides better results than simple SGD.

Defining the architecture, momentum and learning rate:

Parameter	Value
Momentum	0.4
Learning rate	0.001
Architecture	[2304,100,100,4]
Batch size	32
Epochs	20

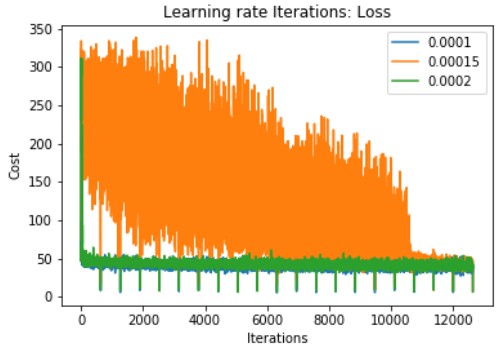


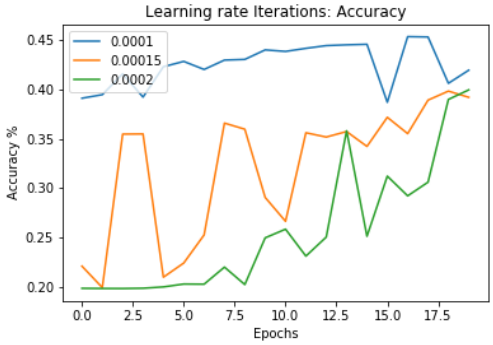


It is evident from the plots, that a lot of oscillation takes place, which is a problem when applying momentum. The 2-hidden layer architecture of 100 neurons proves to be a good balance between accuracy and efficiency. For this specific dataset it looks like all we need is a deep network and enough training epochs to achieve good accuracy. Even though a lot of oscillations take place, the model continues to improve overtime.

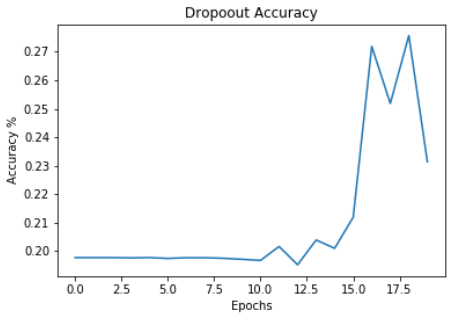
Iterating over the learning rate.

Next, the learning rate is optimized. Optimizing the learning rate is tricky when momentum is used, as numerically things can break quite often. The learning rates tested are [0.0001, 0.00015, 0.0002]. Despite the difference in the training accuracy, the testing accuracy is about the same.





The models achieved {41%, 39% and 41%} accuracy, respectively. Lastly the use of dropout, to the model with 0.0002 learning rate is considered.



Applying dropout does not increase the training or testing accuracy, as the model continues to benefit from more epochs. Dropout could prove to be beneficial when the model begins to overfit, which is not the case here. In this setting the use of Dropout is detrimental. The final testing accuracy for this dropout model is measured at 23%.

Task 6. Tensorflow and Tensorboard visualization.

For this task, a deep NN was created using the Keras and Tensorflow APIs. Moreover, the performance of each model is captured using Tensorboard. The dataset used for this task is the same as task 5. The code for this task can be found in the Jupyter notebook "**Task 6 Solution" and "Task 6 Committee NN solution**". The dataset undergoes the same pre-processing steps as other Tasks ("**Task 6 - Data processing and manipulation").** The images are already in greyscale but are also normalized. The modelling process consists of 3 models.

The first model uses Batch Normalization. This method can normalize data that is part of a batch even as the mean and variance change overtime. Batch Normalization has been very helpful in training large NNs with back propagation. The second model is employing the same 2 hidden layer architecture as the first, as well as a strong form of regularization called LASSO, which uses both L1 and L2 regularization to keep the model from growing too complex. The last model used is an ensemble of NNs. More specifically this ensemble model uses the first 2 models described above, as well as a deeper NN of 4 hidden fully connected layers. This ensemble produces predictions by considering all 3 model predictions and deciding by a simple majority vote rule. The dataset suffers from class imbalance and therefore not all classes are equally represented, this presents a problem in the results and intuitively we expect all models to perform better on classes that have more samples.

The model architectures and training parameters are detailed below:

Training parameters table:

Epochs	100	Batch Size (Training)	256
Optimization Target	Accuracy	Learning Rate	0.001
Optimizer	SGD	Training set size Testing set size	20198 4942

Neural Network models:

Models	Name	Architecture	Parameters
1st model	Batch normalization	2 hidden layers + 2 Batch Norm Layers + Softmax ouput	2 Batch Norm Layers, after the Dense layers.
2nd model	LASSO Regularized Neural Net	2 hidden layers + Softmax output	L1 = 0.0001 L2 = 0.0002
3rd model	Committee of NNs	3 models 1st Model + 2nd Model + A Deeper NN	Equal Weighting of models Majority Vote prediction

Results and Metrics.

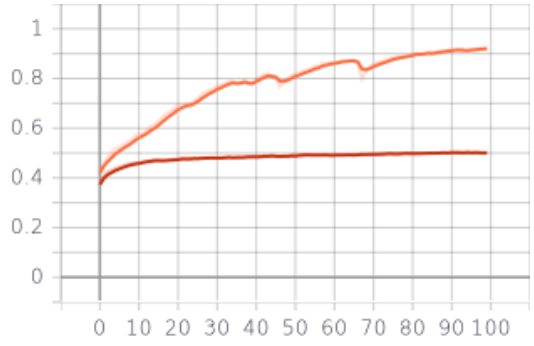
The performance of the Batch Norm model during training increases rapidly, however the performance on the test set is much lower, this could be due to overfitting. The LASSO regularized model reaches a smaller training accuracy; however, it produces better results on the test data.

Accuracy Metrics on the Test set.

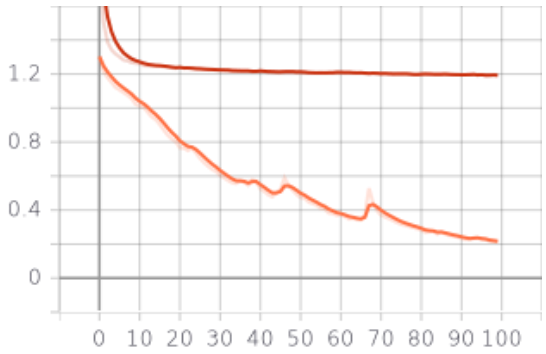
Model Name	***Overall Test Accuracy***
Batch Normalization Model	31%
LASSO REGULARIZED Model	46%
Deep NN model	44%
Ensemble Model	**Highest Accuracy: 47%**

The ensemble of Neural Networks exhibits better testing performance compared to all other models. The reasons for this could be many. The most likely cause is that the Ensemble model exhibits less bias than other models. Since all models are biased in their own way, a simple average of all models should cancel out the bias that they exhibit individually.

Accuracy increase over 100 epochs.



Cross-entropy loss over 100 epochs.

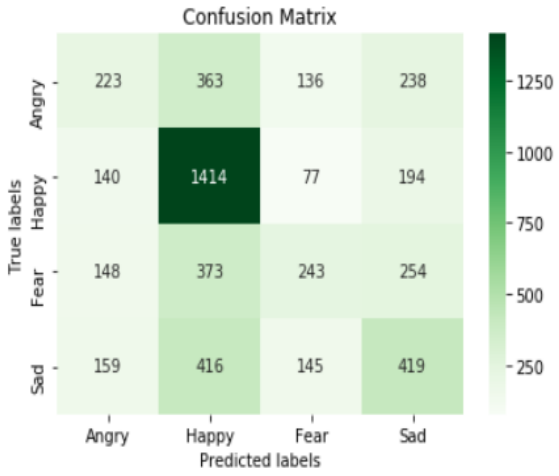


LASSO REGULARIZED MODEL

BATCH NORMALIZATION MODEL

Confusion Matrix and Per-Class results of the Best model: Ensemble Model.

Ensemble Model	Happy	Sad	Fearful	Angry
F1 Score	64%	37%	30%	27%
Precision	55%	38%	40%	33%
Recall	77%	37%	24%	23%



The best performing class in terms of accuracy is "Happy", this could be because happy images are more distinguishable from other categories and thus are easier to learn. It could also be easier for the model to learn happy emotions, because it makes up a larger proportion of the training set.

Conclusion and Final thoughts.

The methods described in this piece of work, use as features, each individual pixel. This is an easy approach, however, nowadays the most performant models use, filters known as convolutions, which mimic the operations that take place in the human eye, to achieve maximum accuracy in computer vision tasks, such as image classification. This is the natural next step to take to improve on the results of this work. The classification problem of task 4 was much easier to tackle due to the abundance of training data. Whereas the dataset used for task 5 and 6 was not as big, which presented problems, moreover the noisy nature of the data, and the imbalanced nature of the dataset represent extra hurdles that need overcoming. Thus, the results cannot really be compared across tasks

References:

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer.
Lu, Lu & Shin, Yeonjong & Su, Yanhui & Karniadakis, George. (2019). *Dying ReLU and*

Initialization: Theory and Numerical Examples.

Sergey Ioffe, Christian Szegedy (2015): *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: *A simple way to prevent neural networks from overfitting*. J. Mach. Learn. Res., 15(1):1929–1958, January 2014.

Chollet, F. (2017). Deep Learning with Python. Manning Publications.

Glorot, Xavier. and Bengio, Yoshua. *Understanding the difficulty of training deep feedforward neural networks*. In Proc. AISTATS, volume 9, pp. 249–256, 2010