395 Machine Learning
Computer Based Coursework 2

Group 35
CBC Helper: Evangelos Ververas

Zhongyuan **Hau**
zhongyuan.hau17@imperial.ac.uk
CID: 01279806 (zh1316)
MSc Computing Science

Stanley **Loh**
qsl17@imperial.ac.uk
CID: 01441443 (qsl17)
MSc Computing (Specialism)

Ao **Shen**
ao.shen17@imperial.ac.uk
CID: 01394238 (as5017)
MSc Computing Science

Shan Xian **Yong**
sam.yong17@imperial.ac.uk
CID: 00756540 (sxy17)
MSc Computing Science

Linear Layers & ReLU Activations

Given that each data x is a vector representing F features, the feed-forward function of the linear layer is a weighted sum of each feature with a bias applied. Weights and bias are tuned by the neural network via back-propagation, essentially causing neural networks to learn which features are better at discriminating the input x. The feed-forward function (linear_forward(X, W, b)) is defined as follows:

$$y = Wx + b$$

The output of the function that represents a hidden layer would then be passed to the activation function (in our case would be the ReLU forward pass function) and when representing the output layer the output would be the result of the learning task. For each layer, the input is stored in a cache in order to perform backpropagation to find the respective gradients at each layer.

$$z = ReLu(y) = max(0, y)$$

The Rectified Linear Unit (ReLU) activation function in the forward pass (i.e. relu_forward(X)) is the activation function used in this neural network that we are developing. Activation functions are often non-linear (ReLu is non-linear due to its piecewise linear behavior at x = 0) in order to enable the neural network to learn non-linear features. The ReLU is computationally less expensive compared to other common activation functions such as sigmoid and tanh which helps to accelerate convergence during training. Additionally, the gradients of ReLu is 1 when x>0 whilst the gradient of the sigmoid/tanh function decreases to 0 as the absolute value of the input increases which potentially gives rise to the 'vanishing gradient' problem for deep neural networks.

In the backpropagation, we implement the linear_backward to calculate the change in values (i.e. gradients) in a single layer of the network needed to reduce the loss. The gradients calculated are specifically for change in weights (dW) and bias (db) of that particular layer, and the derivative for the layer before is returned as well (dX) to allow propagation of loss. The ReLU activation in the backward passing updates the gradients to be zero if the original value in X has been set to zero.

Dropout

Implementing dropout would prevent the overfitting during the training of the neural networks as connections to parent neurons are randomly dropped, this would decrease dependence on specific neurons learning the data, possibly forcing neurons to learn more robust features independent of parent neurons. The implementation of dropout is to remove neurons in a layer by sampling a bernoulli distribution with probability p ranging from 0.5 to 1. We have also adopted the implementation of inverted dropout, whereby the scaling of hidden layer outputs are performed during training time and hence, the weights remained untouched during test time.

Forward Pass Dropout

During training, neurons are randomly (sampled from a bernoulli distribution with probability p) removed from the network. The implementation of the dropout is to multiply the output of the neuron by a binary (1, 0) masking factor and scaled by the probability of retaining the neuron (1- p). The masking factors of all the neurons in a layer are stored as a vector. The binary mask is then stored in a cache for use in the backward pass. Dropout is not applied to the Input and Output layers.

Backward Pass Dropout

Consequent from the forward pass dropout, the neurons dropped out do not contribute to the gradient of the error, as such, the stored mask vector of each layer was used to remove any gradient contribution from the dropped neurons.

Softmax Computation and Gradients

The output of the softmax activation provides a probability mass function (meaning that the sum of all elements is 1) of each example with each element representing the probability of being of that class (by index).

Using the probabilities, we can get the confidence of each example to be classified correctly by checking the probability of the correct label and computing negative log likelihood of that confidence which we define as loss. In order to calculate the gradients, we calculate the derivative of the loss with respect to each individual probability, which works out to be subtracting one from the probability of the correct label. Intuitively this works because only the correct label of each example will have a negative value, while others will have a positive value. We want to increase the probability for the correct label and decrease the probability of the wrong labels. The gradient shows this distinction by having sign differences between the correct and the wrong labels.

In order to perform predictions, we take the argmax of the probabilities to determine the class the input sample is classified as.

Overfitting CIFAR-10

Architecture

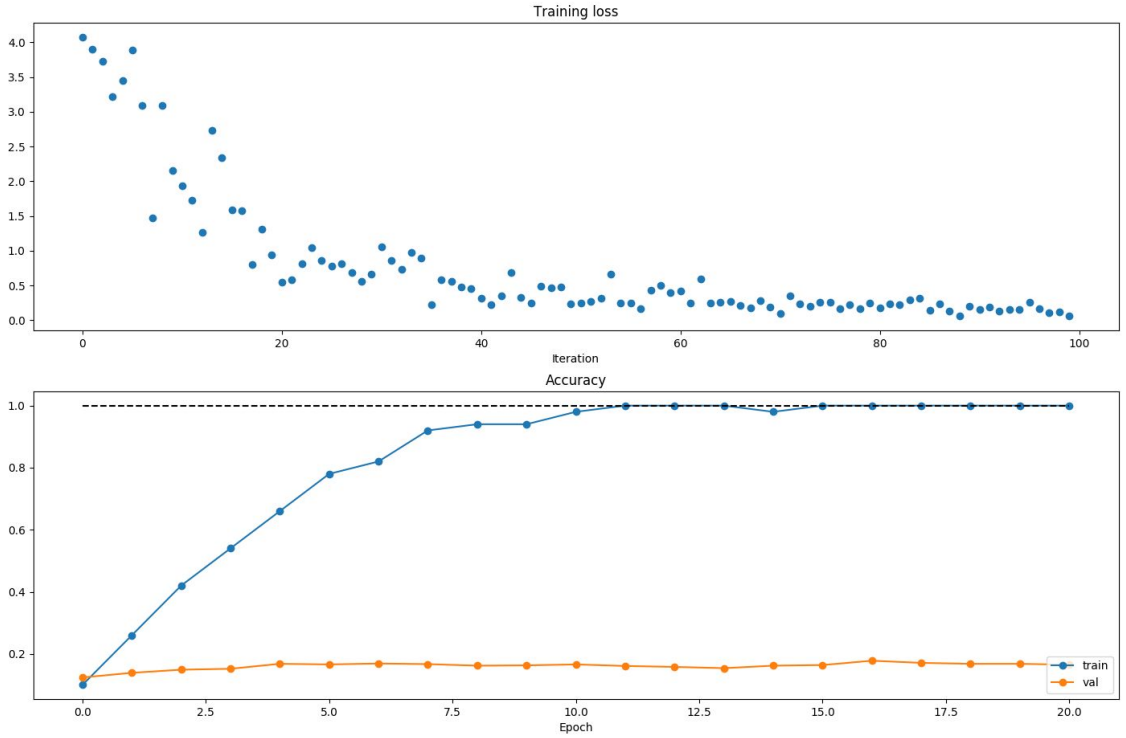
In this network, we try to overfit a small dataset of 50 images for training from the CIFAR10 dataset. We used a neural network of 1 hidden layer of 100 nodes.

Parameters

Our parameters are as follows:

Hyperparameter	Value
Learning Rate	`1e-04`
L2 Regularization Parameter (λ)	`1.8`
Number of Epoch	`20`
Learning Rate Decay	`1.0`
Batch Size	`10`
Update Rule	`Stochastic Gradient Descent`

Plots



50% Accuracy on CIFAR-10

Architecture

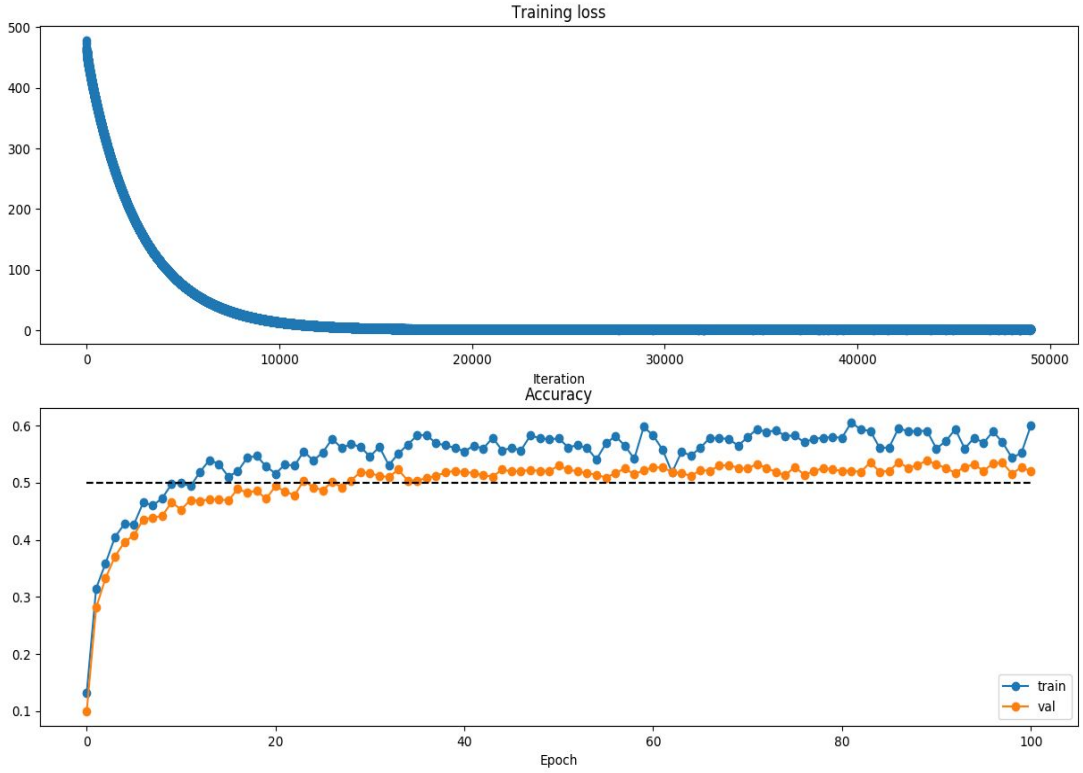
In this network, we try to obtain 50% accuracy on the CIFAR10 dataset. We use a neural network with a single hidden layer of 400 nodes and achieved a validation accuracy of 52.5%.

Parameters

Our parameters are as follows:

Hyperparameter	Value
Learning Rate	`1e-04`
L2 Regularization Parameter (λ)	`1.8`
Number of Epoch	`50`
Learning Rate Decay	`1.0`
Batch Size	`100`
Update Rule	`Stochastic Gradient Descent`

Plots



Hyperparameter Optimization - FER2013

Hyperparameters are parameters that defines the model of the trained neural network and optimizing hyperparameters encompasses finding the optimal model that minimizes the objective function (loss function).

Hyperparameters

For a simple feedforward neural network with backpropagation, some of the parameters to optimize includes:

1) Learning Rate
2) Number of hidden layers
3) Number of neurons in each hidden layer
4) Training Algorithm (i.e. Stochastic Gradient Descent with/without Momentum)
5) Batch Size
6) Type of Regularization
7) Dropout

From the multivariate hyperparameters, it is evident that the search space for optimal set of these parameters is huge and would not be possible to conduct an exhaustive search. Thus, an optimization strategy has to be implemented in order to achieve a balance between the time to obtain a globally optimized model against the performance of a local model.

Optimization Strategy

Sequential Random Search

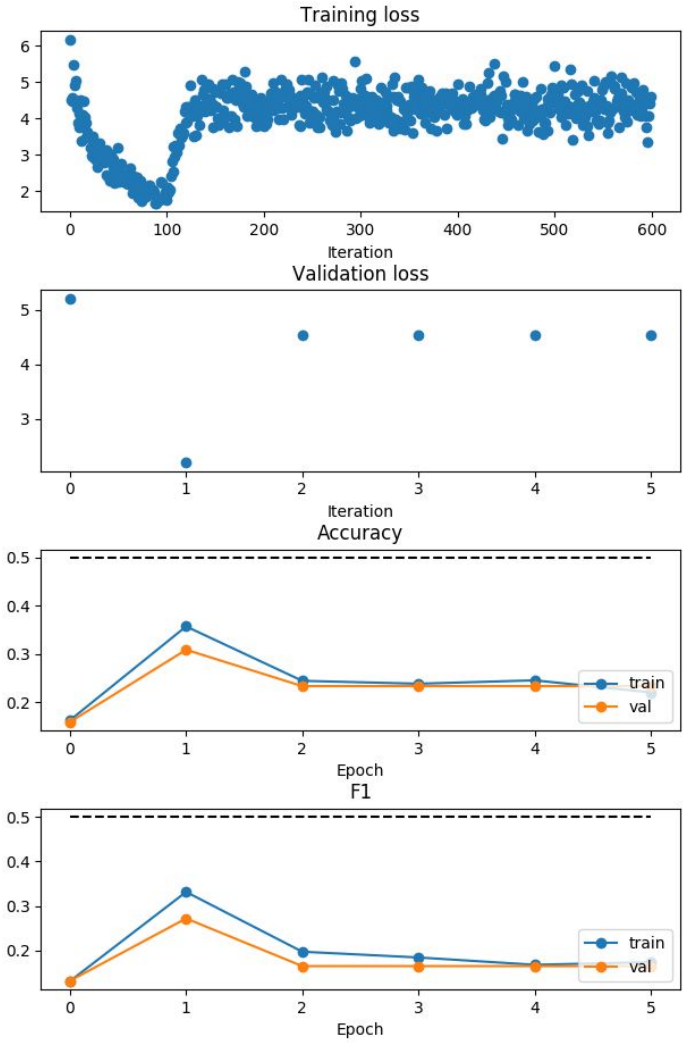
Sequential Search was performed by changing one Hyperparameter while keeping the other hyperparameters constant. Initially hyperparameters are set to be the same as the ones from the model obtained from training on the CIFAR10 dataset. Hyperparameters were sequentially optimized in order of Learning Rate, Dropout/Regularization and finally, Topology (number of layers and nodes per layer) with the objective of minimizing loss. For each hyperparameter, we perform random search for the best value that yields the highest validation accuracy. This optimized parameter would then be retained for the optimization of hyperparameters. For the case of comparing dropout and regularization, it was the case that either dropout or regularization was used and the other parameter was switched off.

Early stopping criteria was implemented in order to save computational time. The stopping criteria used was to terminate training when validation loss do not do better than the minimal validation loss observed over 4 epochs.

In optimising for each hyperparameter, we perform at least 100 iterations of random sampling of parameters to find the value that gives the best validation accuracy.

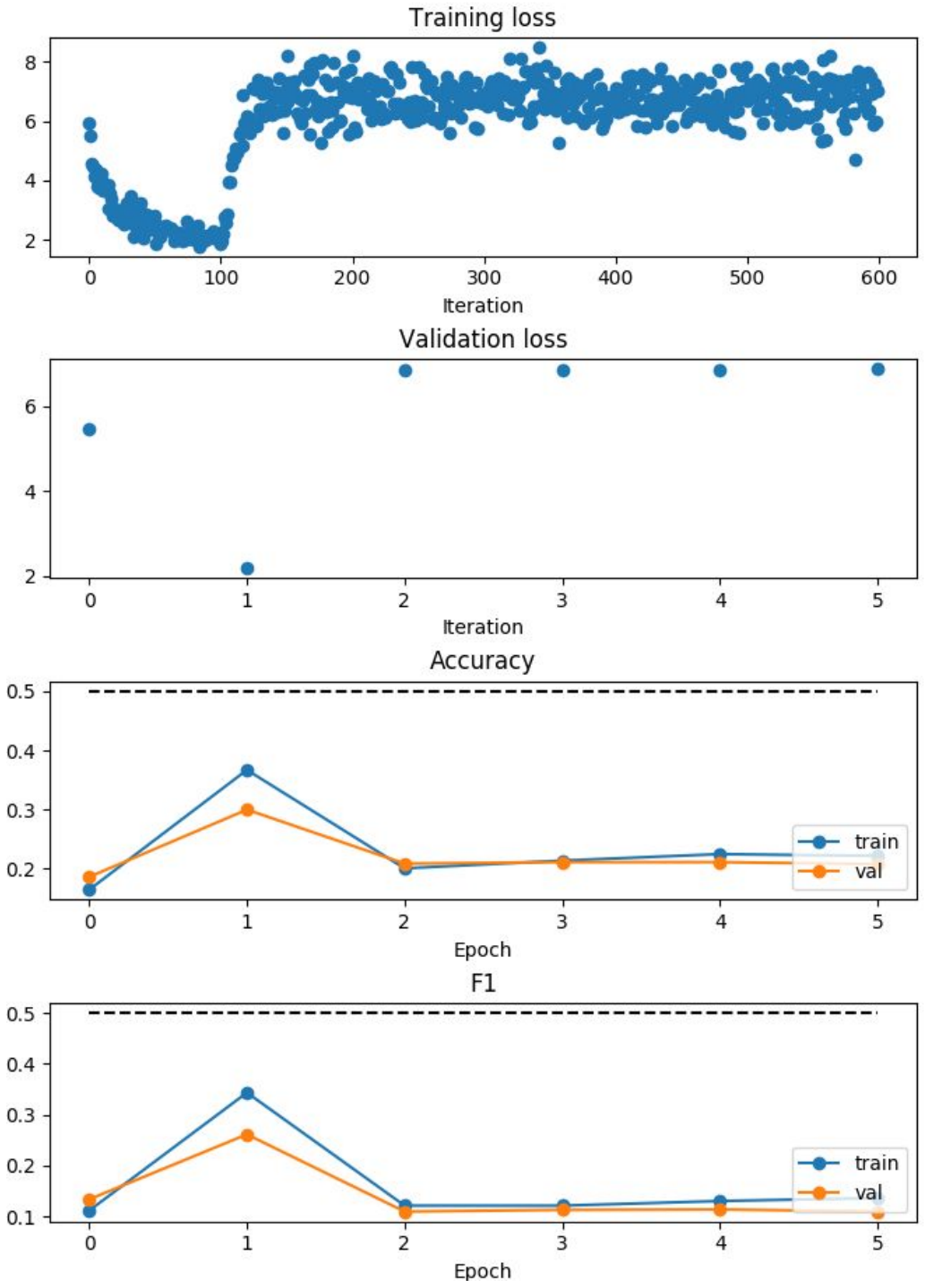
Learning Rate

We started the sequential random search by finding the optimal learning rate. After 100 iterations, we found the optimal learning rate of `9.283379202652052e-05` with the best validation accuracy of 31.4%.



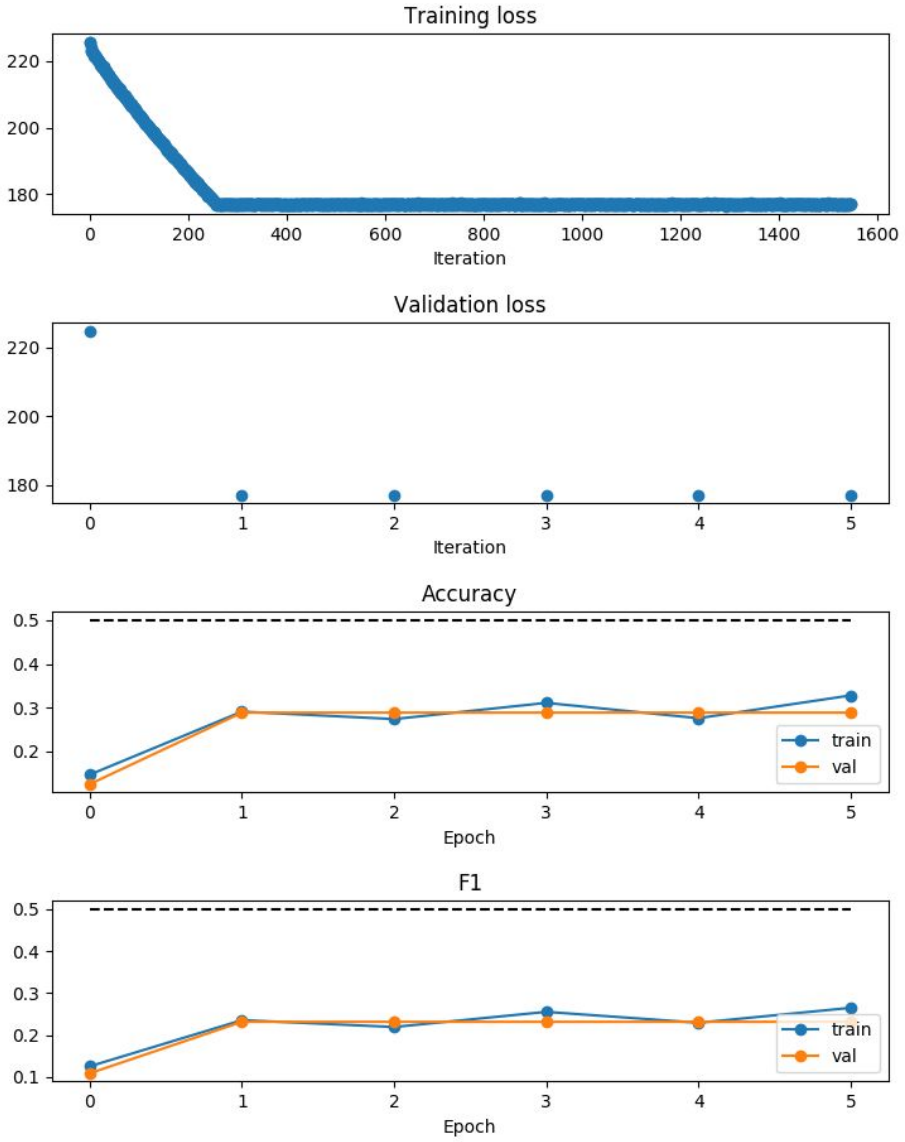
Dropout

Taking the optimal learning found earlier, we perform random search on the dropout parameter to and found `0.006197069258036814` to be the most optimal after 100 iterations, giving its best validation accuracy of 29.9%



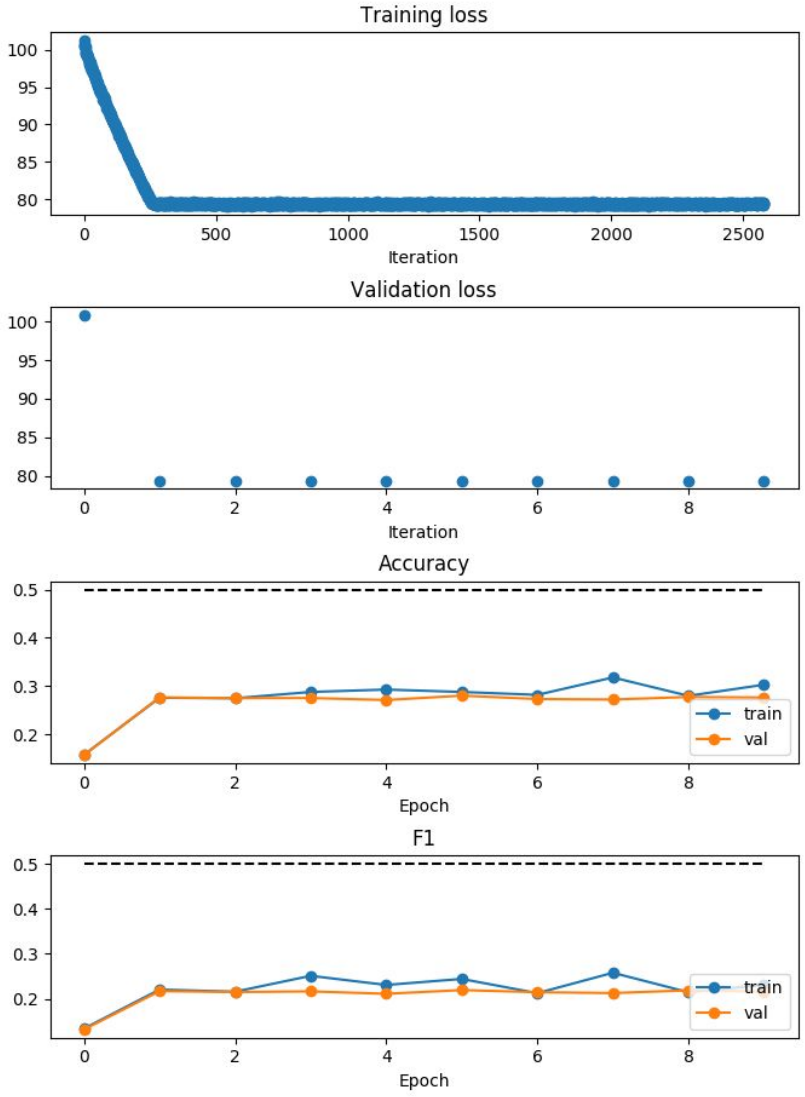
L2 Regularisation

We reset the dropout parameter to 0 and perform random search on the regularisation parameter and found that `4.75936549169918` gives the optimal solution with the best validation accuracy of 37.93% over 100 random search iterations.



Network Topology

Lastly, using the hyperparameters found for learning rate, dropout and regularisation, we perform random search on the number of hidden layers and nodes per hidden layer. The random search for nodes per layer is independent between layers (i.e. layers can have different number of nodes from each other). We found that 1 hidden layer of 178 nodes is optimal and it gives the best validation accuracy of 37.3% over 100 iterations.



Analysis

From the sequential search for optimal hyperparameters, we obtained the validation accuracy of 31.25% with the final set of hyperparameters for the model to be the following:

Hyperparameter	Value / Architecture
Learning Rate	`9.18e-05`
Dropout	`6.19e-03`
L2 Regularization Parameter (λ)	`4.76`
Topology	`1 hidden layer, 178 nodes`

The two methods of regularization explored for the sequential search were Dropout and L2 Regularization. It was observed that L2 Regularization performed better than the use of Dropout. The best dropout parameter we achieved was ~ 6.19e-03 which was close to 0, indicating that the model favours low/no dropout. The optimal L2 Regularization parameter λ obtained was 4.76, which is relatively small, indicating that large weights are not penalized. This was consistent with what we achieved that we favour low/no dropout as

having large λ parameter would mean that weights are small and possibly close to 0, mimicking dropout behaviour. The two regularization parameters were consistent in favoring models with low/no dropout.

The use of sequential search for optimal hyperparameters is that initial parameters were chosen from another NN model dataset and parameters are sequentially tuned for our dataset. This resulted in fluctuating validation accuracy as hyperparameters are heavily correlated and sequential optimization assumes independence of these hyperparameters. As such, the method of sequential optimization might not provide us with a good model. Other strategies available such as grid-search and random search could be employed to find optimal parameters.

Our group has decided to use Random Search[1] instead of a Grid Search for hyperparameters as the search space is huge for the amount of parameters to tune and would be time-consuming. This is in agreement with research (Bergstra, 2012) where they concluded random search is much more efficient, given that not all hyperparameters are equally important, we do not know the granularity of each hyperparameters and search space is huge for the number of hyperparameters to tune.

Random Search

The hyperparameter optimization strategy we employed was to conduct a random search for optimal combination of hyperparameters. Graphically, this would be having random initial points on the error surface and each starting point which would lead us to a local minima after optimization. We would be then able to pick best set of hyperparameters from our runs. The hyperparameters and their range we optimize are:

Hyperparameter	Range
Number of Layers	`1 to 3`
Number of Nodes per Layer	`100 to 500`
Learning Rate	`1e-04 to 1e-04`
L2 Regularization Parameter (λ)	`0.0 to 3.14`
Batch Size	`50 to 200`
Learning Rate Decay	`0.9 to 1.0`
Dropout	`0.0 to 0.5`
SGD Momentum	`0.1 to 0.9`

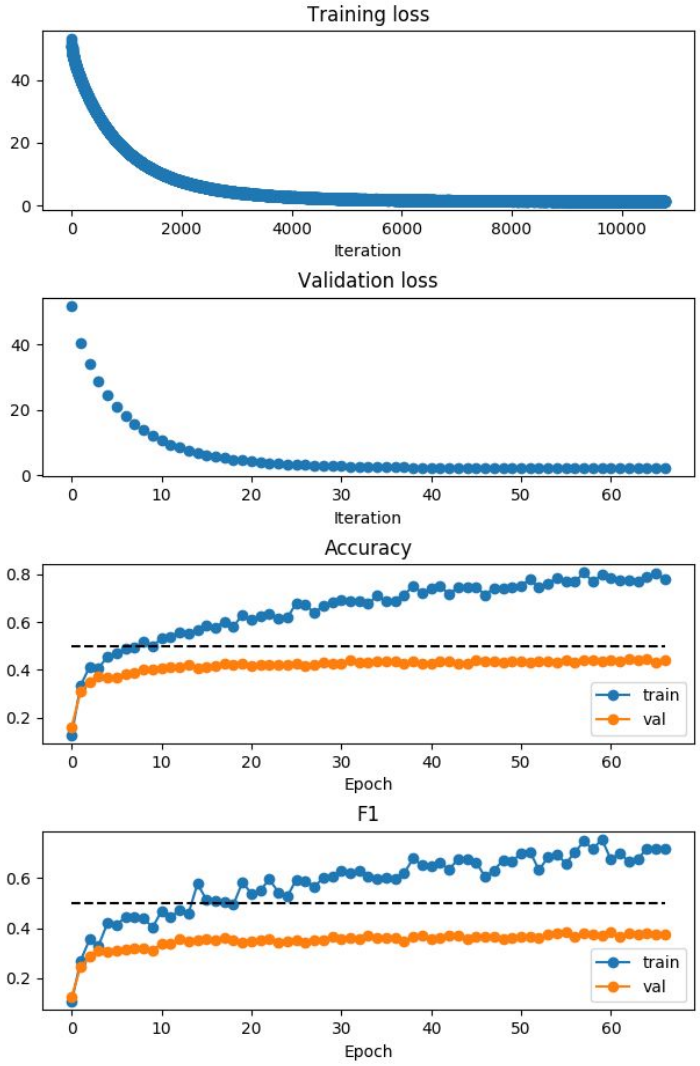
As we are able to perform the random search hyperparameter optimisation unsupervised in parallel, we set up Condor to help us distribute out computing power needed by the random search across the DoC network and consolidate the results later. The output reported by the optimizer over the Condor network can be found in the `results/random_search` folder. We wrote a script to go through all the output files from each computation node to compare each local optimal set of parameters (reported by each machine on the Condor network) and find the global optimal set of parameters.

Over approximately 14,000 iterations on 200 machines, we found the following parameters to be our global optimal that gives our best validation accuracy of 45.49%:

[1] James Bergstra; Yoshua Bengio. *Random Search for Hyper-Parameter Optimization* 2012. Journal of Machine Learning Research

Hyperparameter	Range
Number of Layers	1
Number of Nodes per Layer	457
Learning Rate	1e-04 to 1e-04
L2 Regularization Parameter (λ)	9.067157306681392e-05
Batch Size	160
Learning Rate Decay	0.9777631298833463
Dropout	0.0220584695353766
SGD Momentum	0.8590687445594164

The trained model for this set of parameter is saved in the file "hpop_model" and can be loaded for testing using the `test_fer_model` function in `test.py`. Our plots for the training of this set of parameters are as follow:



Evaluating Optimal Model with Test Set

With the best combination of optimized hyperparameters obtained from the random search, we evaluated the model with the Test Set and the predicted targets were compared with the actual target labels to derive the confusion matrix and measures.

		Prediction Class						
		1	**2**	**3**	**4**	**5**	**6**	**7**
True	**1**	126	0	58	111	96	19	78
	2	10	4	7	17	9	1	8
	3	50	0	141	89	99	43	74
	4	65	0	38	612	92	20	68
	5	79	0	73	152	223	21	105
	6	22	0	40	37	32	245	39
	7	49	0	46	134	109	22	247

Class	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Avg**
Recall / %	27.0	7.1	28.4	68.4	34.2	59.0	40.7	37.8
Precision / %	31.4	100	35.0	53.1	33.8	66.0	41.3	51.5
F1 / %	29.0	13.3	31.4	59.8	34.0	62.3	41.0	38.7
Classification Rate / %	72.2	96.8	72.1	66.0	64.8	84.4	69.2	75.1

Correct Classifications = 44.5%
Sample Error Rate = 55.5%

It was noted that the measures computed from the confusion matrix, has low F1 rate for Class2 and discrepancy between Average Classification rate and and Sample Error Rate . We deduced that this could be due to an imbalanced test set. It was found that Class 2 was severely under-represented, the distribution are as follows:

[467, 56, 496, 895, 653, 415, 607]

The Confusion Matrix was normalized by the Class Sample Size for each respective class and the measures were recomputed:

		Prediction Class						
		1	**2**	**3**	**4**	**5**	**6**	**7**
True	**1**	.27	.00	.12	.24	.21	.04	.12
	2	.18	.07	.13	.30	.16	.02	.14
	3	.10	.00	.28	.18	.20	.09	.15
	4	.07	.00	.04	.68	.10	.02	.08
	5	.12	.00	.11	.23	.24	.03	.16
	6	.05	.00	.10	.09	.08	.59	.09
	7	.08	.00	.08	.22	.18	.04	.41

Class	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Avg**
Recall / %	27.0	7.1	28.4	68.4	34.3	59.5	40.7	37.8
Precision / %	30.8	100	33.1	35.1	27.0	71.4	35.3	47.5
F1 / %	28.8	13.3	30.6	46.4	30.1	64.6	37.8	36.0
Classification Rate / %	66.5	74.0	67.2	62.6	62.6	80.4	66.4	68.5

Correct Classifications = 44.5%
Sample Error Rate = 55.5%

It could be observed that the average classification rate obtained from performance measure was significantly reduced as we have removed the skew of data from majority classes. However, the classification rate obtained from confusion matrix is still not as accurate. By comparing sample error, the performance on the test set is comparable to that of the validation set, which is expected as both sets of data are unseen by the model during training and the generalization performance should be comparable.

Convolutional Neural Network with Keras

FER2013 Test Results with Best CNN

		Prediction Class						
		1	**2**	**3**	**4**	**5**	**6**	**7**
True	**1**	151	10	50	53	106	14	83
	2	2	28	7	2	11	0	6
	3	24	1	210	38	124	33	66
	4	20	0	31	698	48	18	80
	5	28	1	106	70	309	7	132
	6	5	2	62	24	15	291	16
	7	32	1	70	59	117	3	325

Class	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Avg**
Recall / %	32.2	50	42.3	78.0	47.3	70.1	53.5	53.3
Precision / %	57.6	65.1	39.2	73.9	42.3	79.5	45.9	57.6
F1 / %	41.4	56.6	40.7	75.9	44.7	74.5	49.4	54.7
Classification Rate / %	82.5	97.9	76.7	82.0	72.4	91.0	75.1	82.5

Correct Classifications = 56.1%
Sample Error Rate = 43.9%

Normalized:

		Prediction Class						
		1	**2**	**3**	**4**	**5**	**6**	**7**
True	**1**	.32	.02	.10	.12	.23	.02	.18
	2	.04	.50	.13	.05	.18	.00	.11
	3	.05	.00	.43	.08	.25	.06	.13
	4	.02	.00	.03	.78	.06	.02	.09
	5	.04	.00	.16	.11	.47	.01	.20
	6	.01	.00	.15	.06	.04	.70	.04
	7	.05	.00	.11	.10	.19	.00	.54

Class	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Avg**
Recall / %	32.3	50.0	42.7	77.9	47.4	70.1	53.5	53.4
Precision / %	59.9	94.4	38.4	60.0	33.5	84.9	42.1	59.0
F1 / %	42.0	65.4	40.4	67.8	39.2	76.8	47.1	54.1
Classification Rate / %	80.7	87.6	74.8	83.5	71.8	89.8	75.7	80.6

Correct Classifications = 56.1%
Sample Error Rate = 43.9%

Instead of the relatively simply fully-connected neural network, a convolutional neural network is more commonly used for the purpose of image classification. Convolutional layers have a set of filters (which are smaller than the size of the input), each with a set of weights to be optimized. Filters are convoluted across input images, essentially sharing a common weight set across different groups of input neurons. As a result of the localized application of weights, each individual filter is trained to behave as a feature detector, affording the neural network with some degree of geometric invariance and reduces the likelihood of overfitting as compared to a fully connected neural network.

Pooling layers are utilized to downsample the inputs, reducing computational cost. As image pixels are highly correlated to neighbouring pixels, pooling reduces this correlation by grouping together groups of pixels and operating on them (e.g. taking the max value in a max-pool layer). Reducing correlation is essential to reduce the tendency to overfit. The initial pooling dimensions chosen for our model is 5x5 whilst later pooling layers are reduced to 3x3 due to the smaller input dimensions (as a result of prior pooling layers). Such values (tested via grid search) are found to be most effective at striking a balance between decorrelating pixels as well as avoiding too large of a loss in information. Stride sizes of 2 are used to shrink the output in order for computation times to decrease (this also helps with decorrelation). Padding

dimensions were empirically chosen to be 'None' as there was no statistically significant improvement in classification when zero padding is applied. In theory, padding helps to prevent loss of information at the edge of the images. It likely does not help to improve the FER2013 dataset as most images have their key features (etc. lips, eyes) at the center of the image.

Testing Instructions

FER2013 Test with Fully Connected Neural Network

To run the prediction function for the test set on FCNet, run the following command:

```
(env) $ python -m src.test_fer_model [image_folder] [model_file]
```

There are two *optional* parameters that you can use:

- `image_folder`: path to the image folder of FER2013 dataset. Defaults to "datasets/FER2013/Test"
- `model_file`: the model file to load and test. Defaults to "fer_model"

The output of the program corresponds with the prediction for each given test entry.

FER2013 Test with CNN on Keras

To run prediction function for the test set using CNN on Keras, ensure that the following are installed:

```
(env) $ pip install tensorflow-gpu tensorflow-tensorboard tensorflow Keras
```

You can also use `requirements.txt`. You can now run the test by the command:

```
(env) $ python -m src.test_deep_fer_model [image_folder] [model_file]
```

There are two *optional* parameters that you can use:

- `image_folder`: path to the image folder of FER2013 dataset. Defaults to "datasets/FER2013/Test"
- `model_file`: the model file to load and test. Defaults to "deep_fer_model"

Likewise, the output of the program corresponds with the prediction for each given test entry.

Additional Question 1

Q: Assume that you train a neural network classifier using the dataset of the previous coursework. You run cross-validation and you compute the classification error per fold. Let's also assume that you run a statistical test on the two sets of error observations (one from decision trees and one from neural networks) and you find that one algorithms results in better performance than the other. Can we claim that this algorithm is a better learning algorithm than the other in general? Why? Why not?

We cannot claim that one algorithm is better than the other as the inputs to the two algorithms are drastically different. The decision trees algorithm we implemented utilizes a binary representation of the attributes present in a sample. Such data is likely to have been preprocessed from the raw images of the expressions either by a feature extractor or labelled by humans. In contrast, we are feeding the raw images as the input to the neural network algorithm for classification which is, arguably, a more complex problem.

Assume that, hypothetically, we utilized the same types of input to both algorithms, we still cannot conclusively determine that which algorithm is superior solely from the classification accuracy and loss. The are other factors such as training time, model size, number of samples required for effective classification as well as application run-time. For instance, deep neural networks might give better results but require an exorbitant amount of time and samples to perform effective classification whilst taking up significant

amounts of memory to store the necessary parameters (neural networks can easily reach millions of parameters). These might be moot points if the model is expected to be trained/perform predictions via a CPU/GPU but could prove to be crushing if the operation is supposed to be executed on a mobile device.

Additional Question 2

Q: Suppose that we want to add some new emotions to the existing dataset. What changes should be made in decision trees and neural networks classifiers in order to include new classes?

In the decision trees method, we can create new decision trees for classifying the new emotions. By creating groups of new trees for each new emotion, we can combine them in the decision forest algorithm that we have implemented, choosing the best label based on a majority voting of all the trees. The trees that were trained for the original set of emotions can still be retained although we can choose train a new set of trees to incorporate the new data with new classes in order to reduce the rate of false positives for the old trees.

If we are using a neural network, we can naively retrain a new neural network with the new labels added in final layer. We can observe that the architecture of the network does not change except in the final layer where additional nodes are added to account for the new labels. Instead of the naive approach, we can apply transfer learning, fixing the weights of all layers from the previously trained neural network except the final linear layer before the softmax classifier. The shallower layers of a neural network often represent low-level features (such as edge detection) which are often applicable to new classes being added to the algorithm. The deeper layers dictate how a neural network combines these lower level features which creates features that become more specific to each class and therefore needs to be retrained to include features for newer classes.