CO395 Machine Learning CW2

Chang, Yoon Hyuk
01115052
Miller, Tom
01075719
Bagga, Shreyus
01070439
Liaskas, Athanasios
01079253

March 8, 2018

Contents

1 Implementation

1.1 Abstract

This coursework task consisted of the implementation of multi-layer Neural Networks that demonstrated the use of forward and backward pass for linear layers and ReLU activations, back-propagation, dropout and L2 regularisation on hidden layers, as well as a method of hyper-parameter optimisation. The neural network that was constructed was then trained and tested on CIFAR10 and FER2013 datasets, tracking the loss and training/validation accuracies.

1.2 Q1: Linear and ReLU Layers

Linear forward layers apply linear transformations to the input data at a layer. The transformation has the structure:

$$W^T X + b = y$$

where **W** is a matrix of weights (shape (M, D) where D = d1*d2*...*dk) to be applied to the corresponding attributes of input **X** (shape (N, D) and **b** is a column vector of biases (shape $(M,)$ for each output neuron that determines how strong the input should be for the neuron to fire. In the linear python functions, X was input as shape $(N, d1, d2, ..., dk)$, so X was flattened per row, in order for each sample to be of shape $(D,)$, and so that matrix multiplication was possible with the matrix **W**. The output of the function was of shape (N, M), with each row representing a sample, and each column representing the classification value for each class.

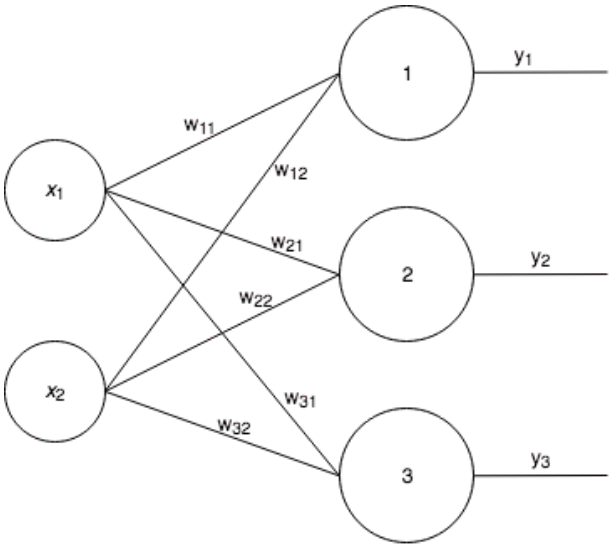

Figure 1: Diagram of a linear layer showcasing the input **X** with attributes x_1, x_2 and the associated weights from each input to each node. A bias term is added for each output node, before the outputs y_i.

The weights for each input to each node can be represented in a weight matrix **W** so that the dot product between the weight matrix and the input samples can be calculated. A bias term exists for each output node, which is added to the result of the matrix multiplication. The output of the node is determined by an activation function, however, the linear layer is not concerned with this function, and simply calculates the value on which the activation function will operate on.

For `linear_backward()`, a single back-pass was performed to calculate the gradient with respect to X, W, b. If O is the output of the network and X is the input into the network, the

following quantities are calculated:

$$\frac{\partial O}{\partial X}, \frac{\partial O}{\partial W}, \frac{\partial O}{\partial b}$$

Considering a 1-hidden-layer network (figure 1), the network has the outputs $Y = W_1X + b_1$ and $O = W_2Y + b_2$, where Y is the output of the hidden layer, O is the output at the output layer, and W_1, W_2, b_1, b_2 are the corresponding weight and bias vectors of each layer respectively. Since only a single back-pass is considered, the current gradient to be calculated is that of Y, using the upstream gradient values from O. Computing the partial derivatives using the chain rule yields:

$$\frac{\partial O}{\partial X} = \frac{\partial O}{\partial Y}\frac{\partial Y}{\partial X}, \quad \frac{\partial Y}{\partial X} = W_1 \tag{1}$$

$$\frac{\partial O}{\partial W} = \frac{\partial O}{\partial Y}\frac{\partial Y}{\partial W}, \quad \frac{\partial Y}{\partial W} = X \tag{2}$$

$$\frac{\partial O}{\partial b} = \frac{\partial O}{\partial Y}\frac{\partial Y}{\partial b}, \quad \frac{\partial Y}{\partial b} = 1 \tag{3}$$

In the formulae found above, $\frac{\partial \mathbf{O}}{\partial \mathbf{Y}}$ corresponds to the upstream derivative; which is passed as `dout` in the argument of the function. Matching the input and output dimensions of the gradients, the final formulae too are directly implemented in the function:

$$\partial X = \texttt{dout} \cdot W^T \tag{4}$$

$$\partial W = X^T \cdot \texttt{dout} \tag{5}$$

$$\partial b = \texttt{dout} \tag{6}$$

where $\partial \mathbf{b}$, is of shape (M,) and `dout` is of shape (N,M); a sum over all the rows is performed to get the total ∂b, since it is a fully connected network. Input $\mathbf{X}$ also has to be `reshaped` to a 2-D matrix as the original input is of shape(*N,d1,d2,...,dk*), whereas it needs to be of shape(N,D).

An example of a function that uses the result of the above calculation is the **ReLU** (Rectified Linear Unit) activation function.

The function is defined as follows:

$$y = max(0, X)$$

where $\mathbf{y}$ is the output of the node and $\mathbf{X}$ is the input. If ReLU is used in conjunction with linear layer, the input X is the net activation ($W^TX + b$). Each element of the input to the function is passed through unchanged unless it is negative, in which case the output is 0.

1.3 Q2: Dropout

Dropout is a **regularization method** that helps **prevent over-fitting** of a network by dropping neurons with a probability p while training the network, where $q = 1 - p$ is the probability of the neuron being kept. This has the benefit of preventing neurons co-adapting, reducing mistakes that would usually be corrected by other neurons, as well as making the network less susceptible to noise due to an increased neuron independence and as a result, a more robust feature detection. It is essentially, an alternative method to modifying the error function of the network.

When **training**, dropout forwards applies a random mask of Bernoulli distributed variables to each layer, turning off some of the neurons in the layer with probability p. The output of each layer stems from the neurons that remain present after the mask.

When **testing**, dropout forwards does not deactivate any neurons, and the input samples travel through the whole network, however every output is scaled by q. To prevent having to scale by q every time testing is conducted on a network, inverted dropout can also be used,

where forwards is scaled by $1/q$ during training so that testing does not require scaling from the original network.

The **inverted dropout function** for **training** is defined as follows:

$$o_i = \frac{1}{q} \cdot m_i \cdot a(\sum_{k=1}^{d_i} w_k x_k + b)$$

where $\mathbf{M} = m_1, m_2, ..., m_{d_h}$ is a d_h-dimensional vector of Bernoulli distributed variables, $a()$ is the activation function and o_i is the output of the neuron.

Bernoulli variables are defined as follows:

$$f(k) = \begin{cases} p, & k = 1 \\ 1 - p, & k = 0 \end{cases}$$

For **dropout backwards**, the mask used for forwards remains unchanged since no parameter tuning is required for dropout and the size of the output also remains unchanged. Therefore, dropout simply passes the gradient backwards through the network, propagating through the neurons that are still present. The mask remains unchanged given that, the gradient of a deactivated neuron is zero since any change in input to that neuron still results in an output of zero.

$$\frac{\partial O}{\partial X} = \frac{\partial O}{\partial Y} \frac{\partial Y}{\partial X} \tag{7}$$

$$\frac{\partial Y}{\partial X} = \frac{\partial \begin{cases} X_{ij}, & m_i = 1 \\ 0, & m_i = 0 \end{cases}}{\partial X} \tag{8}$$

$$\frac{\partial O}{\partial X} = M \cdot \frac{\partial O}{\partial y} \tag{9}$$

where $\mathbf{O}$ is the output of the neural network, $\mathbf{x}$ is the input, and $\mathbf{M} = m_1, m_2, ..., m_{d_h}$ is a d_h-dimensional vector of Bernoulli distributed variables.

1.4 Q3: Softmax Classifier

Using the Softmax function as the final layer of the neural network-based classifier enabled us to use this popular output activation function to represent the classification value for each class.

The Softmax output activation function is defined as follows:

$$o_k = \frac{e^{net_k}}{\sum_k e^{net_k}}$$

where $\mathbf{o_k}$ is the ratio of the exponential function $\mathbf{e}$ to the power of the activation, divided by the sum of the activation of all output neurons (rows). Given, that that the numerator is divided by the sum of all output neurons, it follows that the sum of all the outputs will be equal to one. That is by design a desired property that acts as an advantage over other output activation functions, since it means that the output of our network can be interpreted as a **discrete probability distribution**. That proves powerful in terms of using the Softmax function to classify the value of each class, since it can be used to ensure that the output with the highest probability is chosen every time.

Moreover, for the case of the Softmax Classifier, the right error function is the **negative log likelihood cost**, as seen here:

$$E = -\sum_k t_k ln o_k$$

1.5 Q4: Fully-Connected Neural Network

Implementing the methods above, a Fully-Connected Neural Network can be constructed with the following architecture:

$$[linear - relu - (dropout)] * (N - 1) - linear - softmax$$

N :- number of layers

Two sanity FCNN models are demonstrated below, both attempting to classify the CIFAR10 dataset into its 10 classes.

1.5.1 Overfit Training

50 sample images were taken from CIFAR10 in the aim to overfit the data and achieve 100% training accuracy.

Hyper-Parameters:

Hidden Layers	1
H1-Dimension	100
Update Rule	SGD
Learning Rate	$1e^{-}3$
LR Decay	0.95
No. of Epochs	20
Batch Size	25
Weight Scale	$1e^{-}2$
Dropout	0
Regularization	0

Table 1: Hyper-parameter values for over-fit training on CIFAR10

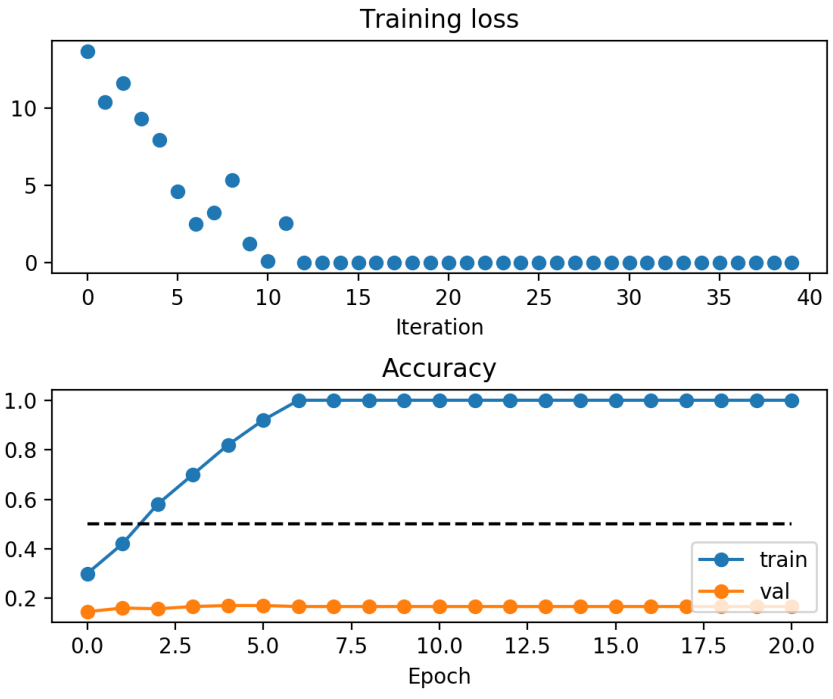

Figure 2: Loss, Train/Validation Accuracy

1.5.2 45% Validation Accuracy Training

25000 sample images were taken from CIFAR10 in the aim to train the FCNN and achieve at least 45% validation accuracy. Hyper-Parameters:

Hidden Layers	1
H1-Dimension	100
Update Rule	SGD
Learning Rate	$2e^{-}3$
LR Decay	0.95
No. of Epochs	25
Batch Size	250
Weight Scale	$1e^{-}2$
Dropout	0
Regularization	0.5

Table 2: Hyper-parameter values for training on CIFAR10

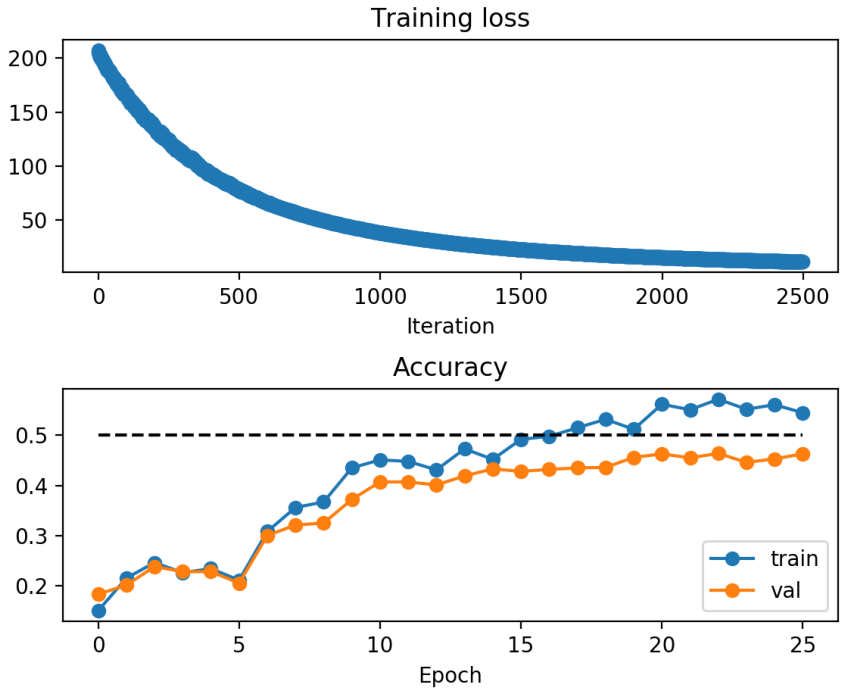

Figure 3: Loss, Train/Validation Accuracy

1.6 Q5: Hyper-parameter Optimisation with FER2013

The final parameter values after training the network on FER2013 data were:

Hidden Layers	1
H1-Dimension	40
Update Rule	SGD with Momentum
Momentum	0.5
Learning Rate	≈ 0.00187
LR Decay	0.8
No. of Epochs	100
Batch Size	100
Weight Scale	$1e^{-}2$
Dropout	0
Regularization	0

Table 3: Hyper-parameter values for trained model

1.6.1 Network Architecture

A common rule of thumb for the number of hidden neurons in the network is that it should be in between the input dimension and the output dimension. Keeping this boundary in mind, the number of hidden layers and neurons were determined to be 1 and 100 respectively. Initially, the number of hidden layers required to achieve the best possible performance were adjusted. It was empirically observed that anything more than a 2-layer architecture was too complex for the classification problem, leading to lower training and validation accuracies. Then, the number of hidden neurons in the 1-hidden-layer was set by tuning the parameter and noting both the changes in training and validation accuracy as well as the degree to which the validation accuracy tracks the training accuracy. Should the number of neurons be too few, the network will not generalize well enough compared to the classifiers, and vice versa. The initial momentum and learning rate was set to heuristic values, 0.9 and 0.001 respectively.

1.6.2 Learning rate optimization

Once the system architecture and initial parameters were set, the learning rate and momentum were further optimized by generating a uniformly distributed sample of parameters within a certain interval, followed by narrowing that interval down based on the performance using those random parameters. The model was trained repeatedly with these different parameters until more optimal values (table 3) were found. Moreover, to improve performance, the mean image of each training batch was subtracted from the training data, normalizing the input images. To determine the optimal momentum, situations where the loss would increase but test accuracies remained the same or increased were monitored, to determine the right amount of momenta to walk through the gradient surface, should it be stuck on a local minimum.

1.6.3 Dropout Performance

Unexpectedly, although the inclusion of dropout saw slight increases in validation performance, it worsened training as well as overall performance.The purpose of the dropout function is to reduce over-fitting and make the model more robust against noisy data, and the lesser effectiveness could be due to the preprocessing done to the FER2013 dataset, where a majority of the faces were cropped to best fit the image, reducing the noise on the dataset.

1.6.4 Regularization: L2

When L2 regularization was used the validation performance also increased slightly, and the loss values were scaled up or down according to the regularization factor. The L2 regularization in essence, penalizes large weights, and the magnitude of the regularization factor determines the

priority of minimizing the error function versus penalizing large weight updates. Large weight updates can have a negative impact on performance especially during the later stages of training as finer changes are required at this stage to better estimate our target function, whereas large updates would most likely disrupt many of the other weights, damaging performance.

In our implementation the use of L2 regularization had a more significant impact on performance over using dropout. It yielded better training and validation accuracies, allowing the penalization of large weight updates.

1.6.5 Other performance optimizations

Due to the FER2013 dataset containing only grey-scale images, all pixels have identical values for each channel component (R,G and B). This means that a performance optimization can be achieved by collapsing the RGB channels into one component, ultimately reducing the dimension of the input layer by a 1/3 which in turn reduces training time.

1.6.6 Training/Evaluation of Model

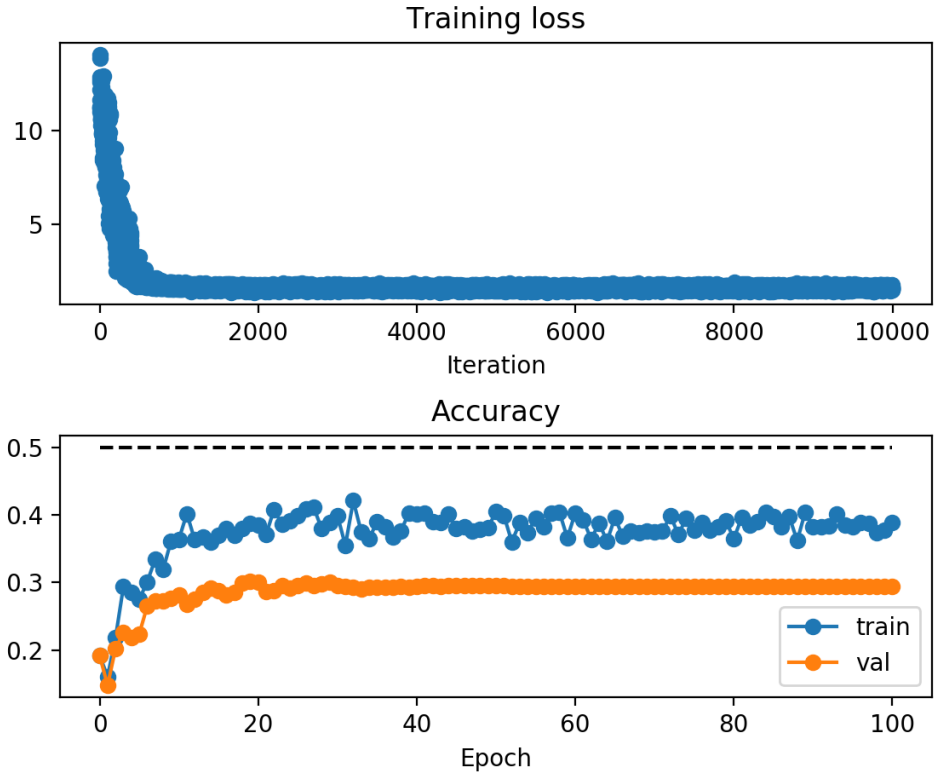

Figure 4: Loss, Train/Validation Accuracy

The neural network performed classification worst on the emotion **disgust**. The main reason for this was that the sample size of disgust during training was only **436**, which was the lowest number of samples of all the emotions by a significant margin. As a result, the recall, precision and F1 values were 0.0 since the network failed to classify the emotion correctly (disgust also had a significantly low test sample size of **56**, so there was less opportunity to classify the emotion correctly). Comparing the number of samples to the best performer **happiness**, which had the highest number of samples with **7215** samples for training and **895** samples for testing, the large difference in performance could be attributed to this difference in sample size. This would make sense as the goal in training an FCNN is to reduce the overall loss, and therefore in this

attempt the weights will be adjusted to prioritize a higher classification rate for the classes of a larger sample size. The confusion matrix and statistical measures of performance are shown below:

	Anger	*Disgust*	*Fear*	*Happiness*	*Sadness*	*Surprise*	*Neutral*
Anger	43	0	22	180	93	50	79
Disgust	5	0	5	18	9	7	12
Fear	43	0	32	136	108	84	93
Happiness	48	3	32	562	96	41	113
Sadness	65	1	38	214	142	55	138
Surprise	25	0	21	75	46	195	53
Neutral	45	0	30	195	95	52	190

Figure 5: Confusion matrix for Training data

	Anger	*Disgust*	*Fear*	*Happiness*	*Sadness*	*Surprise*	*Neutral*
Recall (%)	9.2	0.0	6.4	62.7	21.7	46.9	31.3
Precision (%)	15.6	0.0	17.7	40.7	24.1	40.2	28.0
$F1$ (%)	11.6	0	9.4	49.4	22.8	43.3	29.5

Figure 6: Statistics Measurements for Training data

Classification Rate: 32.4

2 Questions

2.1 A1. 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 algorithm results in better performance than the other. Can it be claimed that this algorithm is a better learning algorithm than the other in general? Why? Why not?

If greater performance is what is considered as better learning in the given learning scenario, for example fraud detection system where low classification error and high accuracy is a crucial in terms of performance, one could claim that such an algorithm is better than the rest.

On the contrary, if the better learning algorithm is defined as the one that is better able to generalize on the learning problem, it would be difficult to determine from the statistical tests alone, that one algorithm is a better learning algorithm than any other.

Furthermore, it is more difficult to claim that one algorithm is better than another **in general**, as it is highly dependent on the learning problem. One algorithm cannot be better than another *per se*, but it can be better at generalizing to a certain type of learning problem. The vast number of methods and algorithms for learning, exist to accommodate for all the different types of problems that apply to machine learning. For example, the decision tree learning algorithm may be excellent in generalizing to fraudulent transaction detection problems but a neural network could be much better at object recognition problems. This is dependent both on the problem space and the surface of the error function, as well as the different error function that was chosen. This contributes to the importance of considering the **learning problem**

as well as the **dataset**, in order to make informed choices about selecting the most effective algorithms to use in order to generalize the learning problem.

Consequently, for a set of particular learning problems, there could be a better or worse learning algorithm. It is difficult to determine in general, however, whether a certain algorithm is better or worse, as the nature of the learning problem affects the effectiveness of each learning algorithm.

2.2 A2. 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?

Should it be necessary to classify more emotions using the existing dataset, assuming that the new emotions' target values are given, the dimensions of target outputs, y_i would increase. If n more emotions are to be added, the dimensions of y_i would become `shape(N,C)`, where `C` is the number of classifiers. Consequently, the following changes would be needed to include new classes in the different learning algorithms:

Neural Networks: The number of output neurons would need to be increased by n. You would also need to retrain the model using the existing dataset in addition to the new emotion labels. This would require the reconsideration and return of some of the hyper-parameters of the network, such as the dimensions of the hidden layers. It depends on how many emotions are added, but a good rule-of-thumb is to have a number of neurons that varies between the input dimension and the output dimension. If the new classifiers alter the gradient function significantly, further retuning of other parameters, such as the learning may need to be changed.

Decision Trees: Another n trees will need to be produced and pruned to classify the new emotions. In terms of architectural changes, one change could need to be made in terms of the ambiguity strategy, should a specific one pertaining only to the previous set of classifiers had been employed.

Overall should there be new emotions to be classified, there would be more changes made using a Neural Network than there would be in the case of a Decision Tree class.