MACHINE LEARNING PRACTICAL
COURSEWORK 4
Advanced Experiments on CIFAR-10 and CIFAR-100

s1641718
Stavropoulou Niki

March 2017

1 Introduction

This project explores neural network techniques on the task of object recognition in the CIFAR-10 and CIFAR-100 datasets, using TensorFlow as a software framework. Starting from a baseline convolutional neural network, the main focus of the coursework will be to improve its performance by minimizing its overfitting behavior. Specifically, we designed, implemented and performed experiments on different models on both datasets, comparing their characteristics and reporting on their classification performance and final error values, for the training and validation sets. Finally, we compare to the performance of the models from the previous courseworks.

1.1 CIFAR-10 CIFAR-100 datasets

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset[1], collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 consists of 60000 colour images, with three colour channels (RGB) and resulting pixel size 3x32x32.

They are categorized in 10 different classes, with 6000 images per class. The data is divided in 50000 training and 10000 validation images. CIFAR-100 consists of the same data, categorized in 100 classes, each with 600 images. There are 500 training and 100 validation images per class.

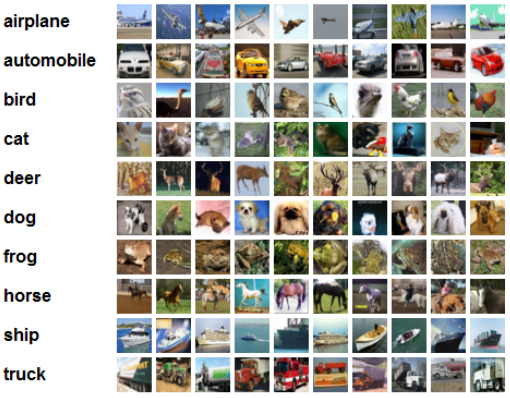


Figure 1: CIFAR-10 classification examples

1.2 Convolutional Neural Networks

A Convolution Neural Network (CNN or ConvNet) is a type of feed-forward artificial neural network, commonly used in image classification. Convolutional neural networks are variations of multilayer perceptrons, inspired by and designed to emulate the behavior of a visual cortex. The way convolutional networks are build, they can learn ('recognize') specific image features. Hence, their behavior is favorable for this

task.[2]

They are comprised of layers of three-dimensional neurons, with learnable weights and biases, and require minimum amount of data preprocessing (or none at all). As, in any neural network, each layer transforms an activation to another by applying a differentiation function. That is, a layer receives a 3-dimensional input and produces a 3-dimensional output by performing a dot-product, followed optionally by a non-linearity.

There are three main layer types used to build up a CNN: a convolution layer, a pooling layer and a fully-connected layer.

Convolutional layer The convolutional layer is the most computational heavy layer, that performs a number of mathematical operations, processing a part of the input image at a time. Its parameters include learnable 3-D filters, which are pieces of the input's original image, so they are of smaller height and width, but of the same depth. Each filter 'slides' over the input image (convolves) and produces a 2-D activation map. Intuitively, this way the filters are activated and the network can learn to recognize different individual image features. Each filter produces a set of activation maps. This is the convolution process.

Pooling layer The amount of parameters resulting from the computations in a convolution layer is increasing. A way to downsize the data's spatial representation is a pooling layer. Without changing the depth of the inputs, the pooling layer resizes every filter of the initial image, not changing the depth. The most common way of pooling is max-pooling. Which simply applies the `max` operator in the input data, as the name implies, keeping the highest values, thus reducing its spatial size.

Fully-connected layer After stacked convolutional and pooling layers, usually follows a fully-connected layer. As in any neural network, a fully-connected layer every consists of neurons, each fully connected to the activations of the previous layer. The layer's output is calculated by a simple matrix multiplication with a bias offset.

These three types of layers will be used to build our baseline model, as will be described in section **2.1**.

1.3 Overfitting and Regularization

Using neural networks can produce better predictions than simple regression algorithms, but in doing so, the network tends to fit the data too well; this is what overfitting intuitively means. Given the complexity of a neural network, it is expected. As such, by construction, convolutional networks have a not-trivial amount of adjustable parameters and can easily overfit. To mitigate this behavior, and consequently improve the network's performance, we need to reduce the flexibility of the model with regularization methods.

There are many ways to regularize a CNN, either implicitly or explicitly. A common implicit way is to apply dropout, which is to skip nodes during the training process given a specific probability. The dropped nodes are re-inserted in the network after the training epoch has ended and other nodes may be skipped. Another way is data augmentation, that is, to provide the network with artificial data. Such data can be produced, for example, by distorting, rotating and altering the initial images.

An explicit way is apply an additional factor to the error at each node in the network, proportional to the sum of weights (L1 regularization) or to the square sum of the magnitude of the weight vector (L2 regularization). Another simple way is early-stopping. As the name implies, it can be effective since overfitting is avoided. On the other hand, the learning process is stopped.

In this report we analyze the results of experimenting with these techniques.

1.4 Research Questions

Having trained an initial CNN as a baseline, the questions our experiments are addressing are:

- How dropout affects the model's performance
- How L2 regularization affects the model's performance
- Is a method of altering the data effective, and which one
- How the CNN model compares to CW3 simple feed-forward model's performance

2 Methods and Architecture

In this section we describe the CNN model that is used as a baseline for the experiments, why it was chosen and how it is implemented using TensorFlow. Each experiment expands this model; any further addition is described at the respective experiments' section.

2.1 Model Architecture

The baseline model has a simple architecture. It consists of three sets (building blocks) of layers. Each building block consists of a convolution layer activated by the ReLu function. We chose the ReLu function, based on the results of the experiments on activations functions from the previous coursework.[3] Models using ReLu converged faster and showed signs of overfitting earlier than the rest of the functions used (sigmoid, tanh), which serves our purpose for these experiments. The second and third block of layers also inlcude a max-pooling layer, following the convolutional layer. At the end of these layers there is one fully-connected layer. Its output is passed through a final linear layer, that makes the class prediction.

All the experiments were conducted on the Msc Teaching Cluster of the University of Edinburgh. Taking this into account, we performed initial experiments to calculate the time required for the training of stacked layers, as were described above (ReLu activated convolution layers). Regardless of the effect in performance, the time required for the training of the model over multiple epochs after adding more layers, restrained the number of possible stacked layers. Three blocks of layers were chosen for the baseline, as the model already showed signs of overfitting and could serve as an example to be regularized.

This concludes the basic model's architecture; the exact implementation using TensorFlow will be detailed in section **2.2** and the way the hyperparameters were chosen will be discussed in section **2.3**. All the code used for these experiments is included in the scripts.ipynb file. The output files with the results of the experiments are placed in the MSc cluster's home directory (`/home/s1641718`).

2.2 TensorFlow implementation

Before building the model, batches of the datasets were loaded using the `CIFAR10DataProvider()` and `CIFAR100DataProvider()` classes pro-

vided in the previous labs. For both training and validation data, a batch size of 50 images was used.

The common usage for TensorFlow programs is to first create a graph (intuitevely the representation of the model), then launch it in a session where it will be executed. Since we build a custom computational graph using convolutional layers, we instantiate our model using the `Session()` class.

To build the graph we start by creating placeholders for the input and target output classes. Note that a CIFAR-10 image has a size of $32x32x3$, resulting in a total input size of 3072. The possible target classes are 10. For a CIFAR-100 image the target classes are 100.

Next, we set up the model. For starters, the input is reshaped, using the `reshape(input, [-1,32,32,3])` function, to a 4-D tensor. The second and third dimensions should match the width and height of the image and the fourth the number of color channels.

We initialize the weight tensors, according to the shape of the filter we will use and the number of feature maps our layer will compute. For example, the first convolution layer will require a weight tensor of shape $[2, 2, 3, 8]$ and a bias tensor of shape $[8]$. The first two dimensions are the size of the filter $(2\text{x}2)$, the third and fourth correspond to the input (RGB color channels) and output channel numbers (8 feature maps).

The input is convolved using `nn.conv2d()`. Our convolutions are zero padded, so that the output is the same size as the input. We use a stride of 1 that allows all the down-sampling to the pooling layers, as the convolution layers will only transform the input depth-wise. The output of the convolution is passed through the ReLu function using the `nn.relu()` method.

We stack a second convolution layer after the first. This time the weight tensor has a shape of $[2, 2, 8, 16]$, meaning the filter size is $2\text{x}2$, the input is 8 (the feature maps from the previous layer) and we want now to compute 16 features. The biases have a matching shape of $[16]$. The output of the convolution is again passed through the ReLu function and is now followed by a max-pooling layer, with $k = 2$. This max-pooling layer will reduce the size of the image to half. For this baseline model, a final convolution layer is added, followed by a max-pooling layer in the same way.

The output of the final pooling layer is fed to the fully-connected layer. The fully connected layer performs a reshape of the input to a tensor with the accumulated final size (size of the filter times the output channels of the previous layer) and a matrix multiplication, using the `matmul()` function and weights, biases of appropriate shape. The output is again activated by the ReLu function.

Finally, there is a linear layer that will give the class predictions, after another matrix multiplication using weights/biases of the appropriate output size, 10 for the CIFAR-10 and 100 for the CIFAR-100 datasets.

Next, we specify the loss function to be the cross-entropy between the target and the softmax activation function applied to the prediction by using the `nn.softmax_cross_entropy_with_logits()` module. This module sums all the predictions of all the classes, so we use `reduce_mean()` method to average on the sums.

We have defined the model and the loss function, so next we will define the optimization algorithm. We used `AdamOptimizer()`, since this optimizer gave the better results on the CW3 model. By setting the `train_step` as `train.AdamOptimizer().minimize(error)`, TensorFlow will add gradient computations and parameter updates to the graph. In particular, the train_step will now apply the Adam algorithm to the parameters, so the model trains by calling `train_step` repeatedly.

Finally, we define how the accuracy of the model will be computed. The prediction of the model corresponds to the maximum value of the output vector, so we use the `argmax()` function to get this value. We check if the prediction matches the ground truth using the `equal()` function and we cast to floating point number represantation with `cast()`. Again, the final result of all the classes needs to be averaged, using the `reduce_mean()` function.

To start the training process, we define the number of epochs. Since all the weights/biases parameters are TensorFlow's Variables they should be initialized in this session. This is done by calling the session's `run(tf.global_variable_initializer())` method. For each epoch, TensorFlow will repeatedly run the `train_step` operation, using the `feed_dict` function to fill the placeholders for the input and output classes we have created. At each epoch, the error and the respective accuracy are computed as defined above.

The following experiments use this model as a baseline. Any addition or alteration will be described in section **3**.

2.3 Tuning the parameters

In order to explore regularization methods, the first step was to find a model that overfits, so we can then test different techniques and report on their impact. Since the experiments were conducted on the MSc cluster, there was also the need for a small, flexible model, that can be trained at least for 30 epochs in a reasonable time span, and of course within the

8hour limit, but which still exhibits the desired behavior. Specifically, signs that hint at overfitting are low error values on the training set combined with high error values on the validation set (or high training accuracy combined with low validation accuracy).

For starters, we needed to define a filter size for the convolutions. We tried 2x2 and 5x5 kernel sizes. We also needed to define a number for the feature maps each computation would produce. We tried both kernel sizes, with an output of 8, 16, and 32 feature maps each, and trained both datasets for 30 epochs. The training and validation errors for the CIFAR-10 dataset are shown in Figure 2 and for the CIFAR-100 in Figure 3. The final values are accumulated in Table 1.

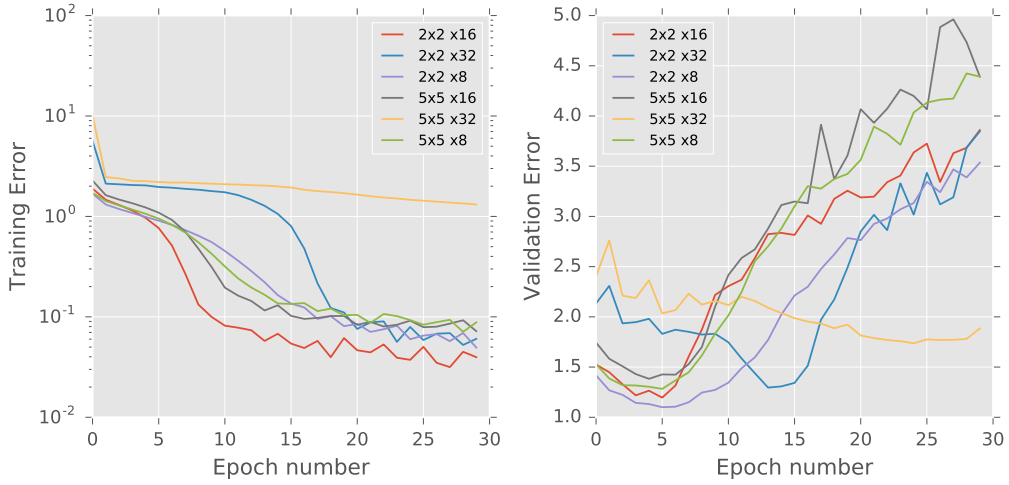


Figure 2: Training and validation error of different filter sizes and feature maps, 30 epochs, CIFAR-10

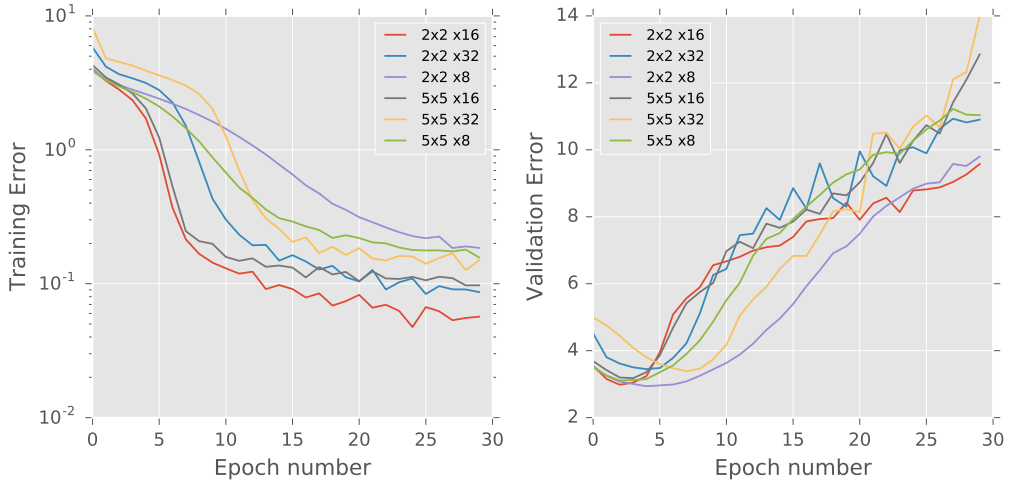


Figure 3: Training and validation error of different filter sizes and feature maps, 30 epochs, CIFAR-100

Filters x Feature Maps	Training Error	Training Accuracy	Validation Error	Validation Accuracy
CIFAR-10				
2x2x8	0.04942	0.98315	3.53452	**0.58340**
2x2x16	**0.03967**	**0.98812**	3.85984	0.57370
2x2x32	0.06042	0.98342	3.84268	0.55519
5x5x8	0.08810	0.97155	4.39150	0.51819
5x5x16	0.07199	0.98050	4.39237	0.50349
5x5x32	1.31518	0.51214	**1.88439**	0.37810
CIFAR-100				
2x2x8	0.185031	0.94144	9.80014	0.24930
2x2x16	**0.05677**	**0.98362**	**9.57557**	**0.25720**
2x2x32	0.08655	0.98015	10.90122	0.20350
5x5x8	0.15711	0.95017	11.03517	0.22020
5x5x16	0.09731	0.97482	12.84985	0.21330
5x5x32	0.14958	0.96747	13.99529	0.20649

Table 1: Final error values when for different filter sizes/feature maps combinations, 30 epochs training

From the table it can be seen that for the CIFAR-10 dataset the 2x2 filter had better final values (lower error/higher accuraccy) than the 5x5 filter on all feature map pairs, except for the case 32, which had the lowest validation error value of 1.88. This was somewhat expected, since the original

images have a size of 32x32x3. Even so, we could not pick 32 as the feature map number for the baseline; it can be seen from its validation plot that it showed no sign of overfitting yet. It should also be noted that for both 2x2x32 and 5x5x32 cases it took more than twice the time to train the model for 30 epochs, than for 8 and 16 features.

All other cases showed signs of overfitting. As can be seen in the validation plots in Figure 2, the error is rapidly increasing after the 5th epoch. Only the case 2x2x32 showed delayed signs of overfitting, after the 10th epoch. But, given the time it required to train a model for 32 output maps, it was not chosen. Safe choices were 2x2x16 and 2x2x8, and we picked the later, which exhibited the fastest overfitting behavior and a validation accuracy of 0.58.

On the CIFAR-100 dataset, the models performed quite similarly, but it can be seen in Figure 3 that the validation error starts increasing earlier, before the 5th epoch. On this dataset though, the case 2x2x8 gave the lower training final values, which can be seen in Table 1. For the baseline we chose the case 2x2x16, that gave the best validation final values, reaching an accuracy value of 0.25.

3 Experiments and Results

3.1 Dropout

The experiments described in this section were conducted to explore the impact of dropout to the CNN.

Motivation

Adding dropout means the network will be ignoring some activation nodes during the training process, given a predefined probability. This way, during the training process we prevent the network from closely matching the input data, or, in other words, to overfit.

In coursework 3, applying dropout to the network greatly improved the validation final error and accuracy values. But, applying dropout in this multi-layered CNN was not only a matter of choosing the best performing probability. A new question surfaced: in which layers should dropout be applied[4]. We performed two sets of experiments, one where dropout is applied to all the layers of the model and one where dropout is applied only to the fully-connected layer. The experiments were conducted on both datasets.

TensorFlow Implementation

We used the baseline model described in section **2**, with three convolution/ReLu layers, followed by a fully-connected one. To add dropout to a layer we use TensorFlow's method `nn.dropout(targets, prob)`. This method will apply dropout to the parameter targets with probability prob. The convolution layers are affected by dropout after being activated by the ReLu function. The same notion applies to the fully-connected layer.

Results

For the first set of experiments, dropout affects only the fully-connected layer, following the stacked convolution layers. We iterate over five dropout probabilities, $0.1, 0.25, 0.5, 0.75, 0.9$ and recording their error and accuracy values on both training and validation sets after training the model for 30 epochs. The error plots for CIFAR-10 are shown in Figure 4 and for CIFAR-100 in Figure 5. The final values of the training process are accumulated in Table 2.

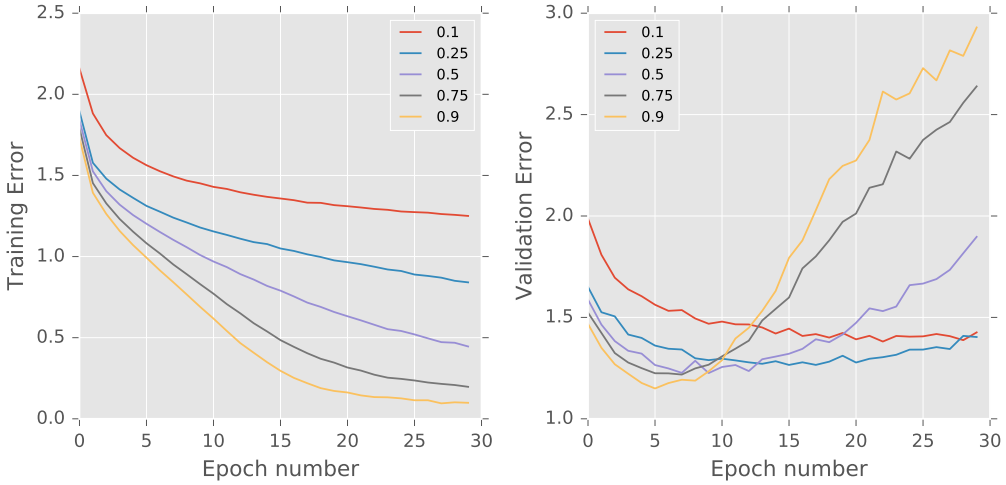


Figure 4: Training and validation error when dropout is applied only to the fully-connected layer, 30 epochs, CIFAR-10

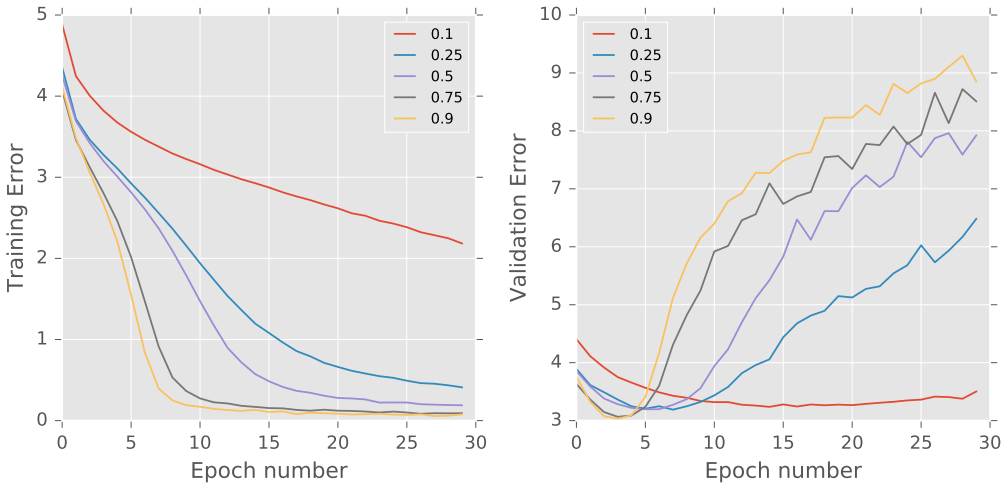


Figure 5: Training and validation error when dropout is applied only to the fully-connected layer, 30 epochs, CIFAR-100

Dropout	Training Error	Training Accuracy	Validation Error	Validation Accuracy
CIFAR-10				
0.1	1.25041	0.55517	1.42606	0.51499
0.25	0.84099	0.69797	**1.40325**	0.56059
0.5	0.44587	0.83609	1.89721	0.55779
0.75	0.18721	0.92964	2.63915	0.54699
0.9	**0.09860**	**0.96692**	2.92972	**0.57510**
CIFAR-100				
0.1	2.18432	0.42727	**3.50254**	0.24969
0.25	0.41052	0.54204	6.48205	0.22959
0.5	0.18905	0.94502	7.92286	0.21770
0.75	0.091471	0.97427	8.51060	0.23929
0.9	0.07711	**0.97920**	8.85408	**0.24469**

Table 2: Final error values when dropout is applied only to the fully-connected layer, 30 epochs training

For the second set of experiments, dropout affects both the convolution layers and the fully-connected layer following them. We iterate again, using the five dropout probabilities, $0.1, 0.25, 0.5, 0.75, 0.9$, and reporting their final values after training for 30 epochs. The error plots are shown in Figures 6 and 7 for the CIFAR-10 and CIFAR-100 datasets respectively. The final values after applying dropout to all the layers are shown in Table

3.

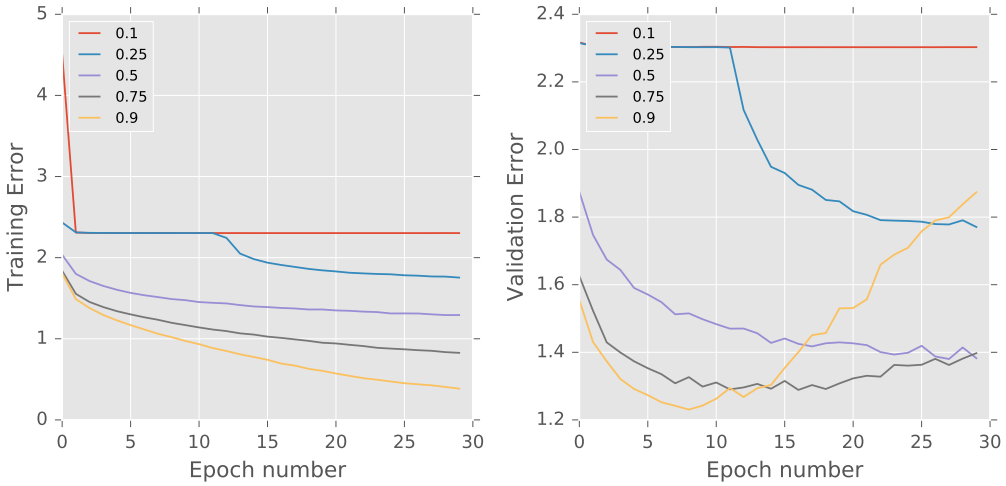


Figure 6: Training and validation error when dropout is applied on all the network layers, 30 epochs, CIFAR-10

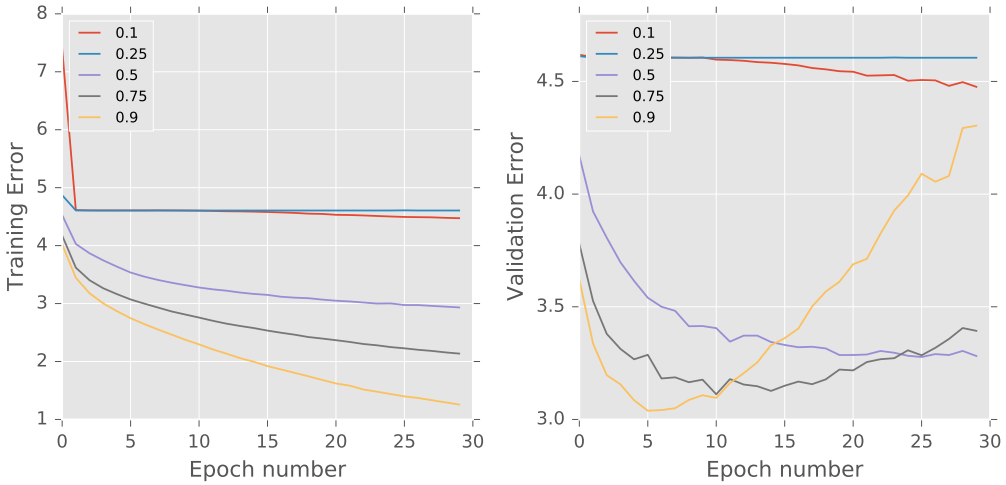


Figure 7: Training and validation error when dropout is applied on all the network layers, 30 epochs, CIFAR-100

Analysis and Conclusions

As was mentioned above, CW3 model's performance increased when applying dropout. Specifically, the optimal probability was found to be 0.9,

Dropout	Training Error	Training Accuracy	Validation Error	Validation Accuracy
CIFAR-10				
0.1	2.30289	0.10059	2.30273	0.09910
0.25	1.75427	0.36335	1.77067	0.36500
0.5	1.29269	0.53970	**1.36251**	0.51289
0.75	0.82661	0.70280	1.39738	**0.56080**
0.9	**0.38562**	**0.86122**	1.87362	0.55820
CIFAR-100				
0.1	4.47318	0.2907	4.47519	0.22940
0.25	4.60581	0.00782	4.60517	0.00999
0.5	2.93303	0.27117	**3.2815**	0.22609
0.75	2.13677	0.42862	3.39332	**0.25970**
0.9	**1.25757**	**0.63125**	4.30344	0.24480

Table 3: Final error values when dropout is applied to all the layers, 30 epochs training

both on the CIFAR-10 and on the CIFAR-100 dataset, and the plots[3] showed that ReLu's overfitting behavior was mitigated. The experiments on the CNN model verify the theory, that applying dropout to the model reduced overfitting. The results, though, differ in relation to where dropout was applied.

The first model, where dropout was applied only to the fully-connected layer, on CIFAR-10 achieved the best validation accuracy of 0.57 for a probability of 0.9, as shown in Table 2. But, the plots in Figure 4 hint that this occurred because the model overfitted the data, as the validation error is rocketing after the 5th epoch. A smoother curve was produced with probability 0.5, and exhibits the desirable regularization effect.

On the CIFAR-100 dataset 0.9 probability gave again the best validation and training accuraccy (validation accuracy 0.24), but in Figure 5 the model for probabilities 0.9 and 0.75 is again showing signs of overfitting. Comparing to the baseline Figures 3,4 though, it can be seen that applying dropout only to the fully-connected layer delayed slightly the overfitting behavior. Probabilities 0.1 and 0.25 are simply leading to underfitting, since the model is learning almost nothing.

The second model, where dropout was applied to all the layers had different results. The final values were close to the previous model, as can be seen from Table 3. For CIFAR-10 a probability of 0.75 gave the lowest validation error and highest validation accuracy of 0.56, followed by the results for 0.9. However, the plots in Figure 6 are far better than

the previous dropout model. Probability 0.9 plot still hints at overfitting, but this is delayed by 2 epochs, compared to the baseline. The curve for probability 0.75 is showing a slight overfit, but is an obvious improvement to Figure 3's plot for the 2x2x8 model.

The same effect can be seen in the models trained on the CIFAR-100 dataset. With 0.75 probability, Table 3 shows we achieved the highest validation accuracy of 0.25, but in Figure 6 the validation error is rising. Nevertheless, the curve's minimum point is moved after the 5th epoch. With probability 0.75 and 0.5 the model had significantly reduced overfitting behavior. Again, probabilities 0.1 and 0.25 are lead to underfitting.

These experiments lead to two conclusions. The first is that dropout helped the CNN model reduce and delay overfitting the data, even if the impact was small, as in the first set of experiments. The second conclusion is that applying dropout to all the layers gave significantly better validation curves, than implementing it only on the fully-connected layer. This can be explained, since adding dropout to the convolutional layers as well, is too strong of a regularizer[2]. We would presume that since the parameters in a convolution layer are not many, they need less regularization to begin with. But, the experiments prove that applying dropout to them helped by providing noise to the lower layers, that will eventually end up affecting the fully-connected one, thus making the model more robust and reducing overfitting. Finally, what should be noted is that applying dropout slowed the training process. The models took longer to train, especially the 2x2x16 of CIFAR-100.

3.2 L2 Regularization

The experiments in this section were conducted to report on the impact of L2 regularization on the CNN model.

Motivation

L2 regularization applies an additional penalty to the error, during the training process, proportional to the sum of of the squared weight vector. A more accurate prediction can be made, by penalizing weights and forcing them to keep within the defined limit, thus overfitting is prevented.

In coursework 3, the simple feed-forward model scored higher values when affected by L2 regularizers. The graphs also showed that overfitting was delayed. For the CNN model we perform experiments, testing the same four different L2 penalties and reporting on their results in both validation and training sets, on both CIFAR datasets.

TensorFlow Implementation

We used the baseline model, with three ReLu activated convolutional layers, followed by a fully-connected layer that is described in section **2**. We applied L2 penalty on the model as follows: at each layer a weight decay factor is computed using TensorFlow' s `L2_loss()` function, using the layer' s corresponding weight vector. Each weight loss, multiplied by a predefined penalty value, is added to the final loss returned. This accumulated loss will be used to compute the error at each `train_step` using the `softmax_cross_entropy_with_logits()` loss function.

It should be noted that for the computation of the validation values, L2 penalty is not taken into account, as is appropriate. There is a new `error_val` Variable that stores the `softmax_cross_entropy_with_logits()` using the model's output, without the L2 weight losses and compares to the valid classification.

Results

We conducted experiments iteratively, using four different L2 penalties, $0.1, 0.01, 0.001, 0.0001$, on both CIFAR-10 and CIFAR-100 datasets and report on their results after training the model for 30 epochs. The error plots for CIFAR-10 are shown in Figure 8 and for CIFAR-100 in Figure 9. The final values of the training process are accumulated in Table 4.

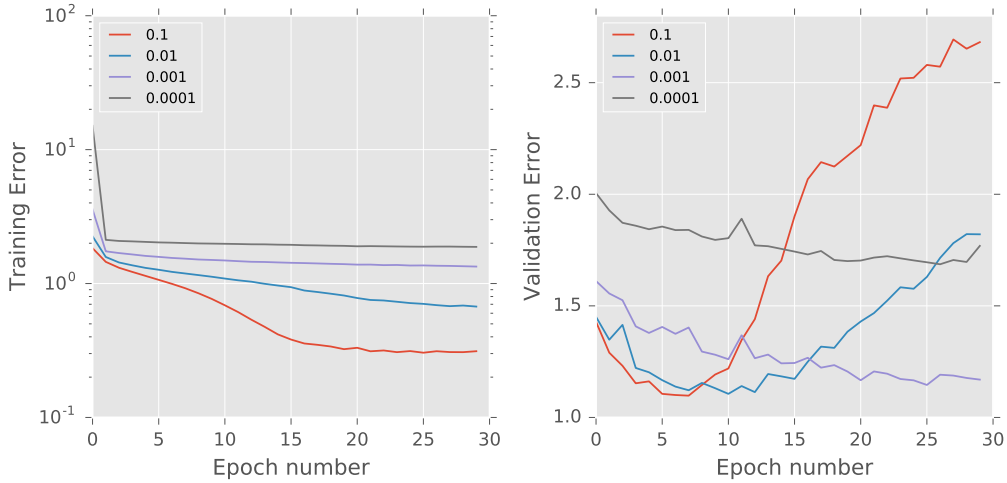


Figure 8: Training and validation error when L2 regularization is applied, 30 epochs, CIFAR-10

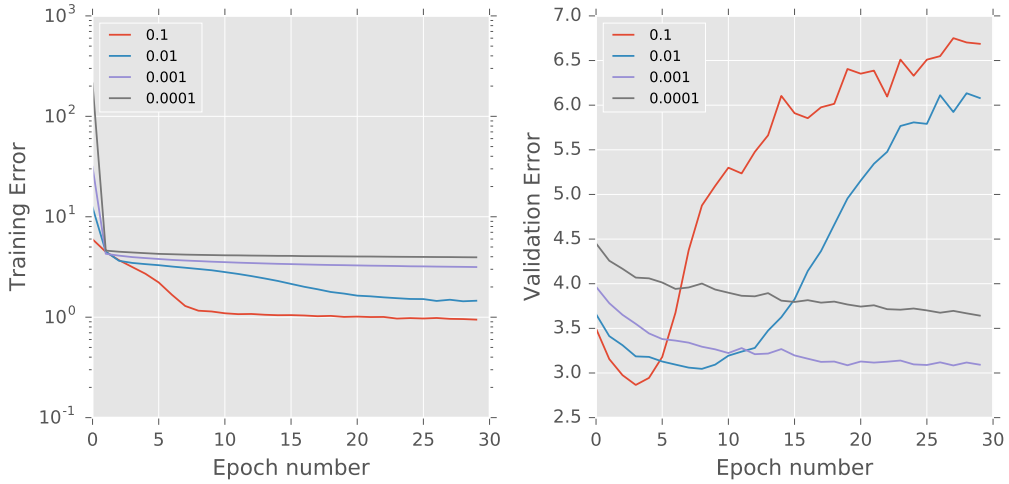


Figure 9: Training and validation error when L2 regularization is applied, 30 epochs, CIFAR-100

Analysis and Conclusions

As was mentioned above, the CW3 model did noticeably benefit from the L2 regularization addition. In particular, a penalty of 0.001 was shown to perform best on both datasets. The experiments in this section lead to the same conclusion.

On CIFAR-10 a L2 penalty of 0.001 gave the lowest validation error and highest validation accuracy value of 0.59, higher than that of the baseline model, as can be seen in Table 4. In Figure 8 it is also obvious the 0.001 model is exhibiting no signs of overfitting. Even if for 0.1 and 0.01 the training error was the lowest, the validation error curves still rise rapidly after the 5th epoch.

On CIFAR-100, once again, a L2 penalty of 0.001 resulted in the best validation final values, reaching an accuracy value of 0.25, as is shown in Table 4. Similarly, the validation curve for 0.001 in Figure 9 shows no sign of overfit. The models with penalties 0.01 and 0.1 still had the lowest training error, but that is due to their overfitting behavior, that is hinted from their validation error plots.

Overall, L2 regularization greatly improved the model's performance with respect to its overfitting behavior. To be exact, the L2 models for 0.001 show no sign of overfit. This was expected[5], since L2 regularization is by default, an explicit method of generalizing a model, widely used. What should be noted is that we did not experiment with L1 regularization, as it is more beneficial in sparse datasets[6]. Once again, as in CW3's L2

L2 penalty	Training Error	Training Accuracy	Validation Error	Validation Accuracy
CIFAR-10				
0.1	**0.31235**	**0.96402**	2.6815	0.58630
0.01	0.67306	0.90214	1.82059	0.57580
0.001	1.33898	0.61017	**1.16956**	**0.59229**
0.0001	1.87820	0.38055	1.76868	0.36860
CIFAR-100				
0.1	**0.94600**	**0.95717**	6.68725	0.24179
0.01	1.45870	0.85477	6.07995	0.22059
0.001	3.15746	0.30510	**3.09296**	**0.25369**
0.0001	3.94680	0.16814	3.64271	0.14489

Table 4: Final error values when L2 regularization is applied, 30 epochs training

models, the training process was slightly slower than that of the baseline model.

3.3 Data Augmentation

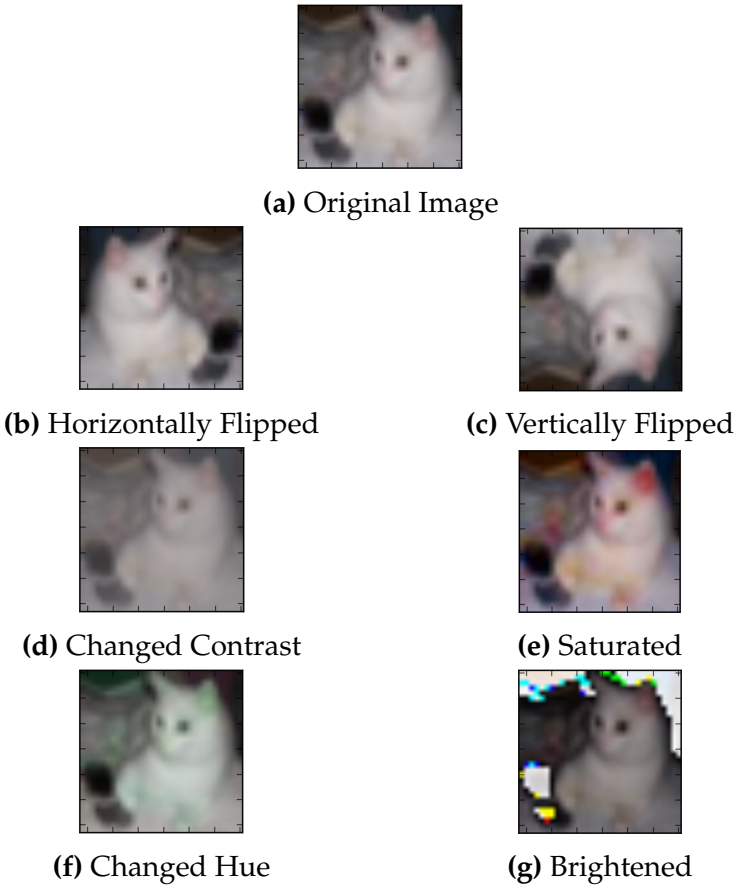
The experiments described in this section focus on the impact of data augmentation, specifically, methods of transforming the input images of both datasets.

Motivation

The degree to which a model overfits is affected by the amount of training data it processes. Simply increasing the training data is indeed a valid method that could reduce overfitting. But, the CNN is already using the available resources, so a more cost-and-time-efficient alternative is randomly altering the existing data. The model will be able to adapt better, and possibly learn new features, resulting in a more robust prediction process.

We did not apply this method to the models of Coursework 3, but we experimented with similar alteration techniques (rotating, skewing, cropping, elastic deformation and different filters) in coursework 2, while training a simple network on the MNIST dataset[7]. For this coursework we will explore the effect of randomly flipping vertically and horizontally the images of both datasets. Contrary to the gray-scaled MNIST numbers, the CIFAR images have three color channels. We, therefore, explored the effect of tampering with the color's features, by altering the hue, contrast,

saturation and brightness of the images. We aimed to see whether these alterations could have the same regularizing effect as in CW2. An example of an image from the CIFAR dataset and each transformation it will undergo is shown in Figure 10.



(a) Original Image

(b) Horizontally Flipped

(c) Vertically Flipped

(d) Changed Contrast

(e) Saturated

(f) Changed Hue

(g) Brightened

Figure 10: Examples of the transformations

TensorFlow Implementation

We used the three-convolution layers baseline model of section **2**, with the following addition. We create a new placeholder that will hold the input images, as does the original `inputs` placeholder. The new one will be used for the processing/altering of the images. To further implement that, during the training process of the session, this placeholder is fed with the inputs using the `feed_dict()` function, after they are processed by a helping function that applies the different transformations. The origi-

nal inputs placeholder is now fed with the processed data from the new placeholder. The error and accuracy computations remain the same.

For the actual alterations, we use TensorFlow's Image subsection's methods. Specifically, to flip the image horizontally we use `random_flip_up_down()`, and vertically with `random_flip_left_right()`, which flips an image (3-D tensor) accordingly, with 50% chance. To change the contrast we use the `random_contrast()` function that will adjust the contrast by a factor randomly picked from a predefined interval. We used the interval $(0.1, 1)$. The hue was altered with the `random_hue()` function, that uses a random delta to rotate the hue channel, after converting the image to HSV. We used a delta in $[-0.5, 0.5]$. To change the saturation we used the `random_saturation()` function, which will apply a saturation factor between an interval, here $[0, 5]$. Finally, the brightness was altered with the `random_brightness()` function, that changes the image to float representation and then adds a random delta to all floats, before returning the image to the original representation. We used the delta interval $[-0.5, 0.5]$.

All alterations are applied randomly to the image with an expected probability of changing the image once out of three times, by using numpy's `randint()` function.

Results

We conducted experiments by applying the functions mentioned above report on the effect these transformations had on the model's performance, after training both datasets for 30 epochs. The error plots for CIFAR-10 are shown in Figure 11 and for CIFAR-100 in Figure 12. The accumulated final values are shown in Table 5.

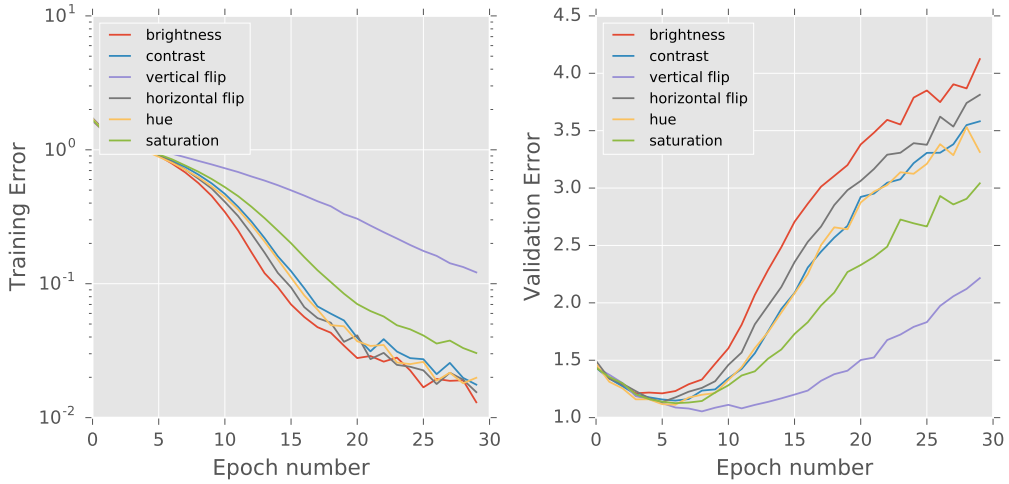


Figure 11: Training and validation error with data augmentation, 30 epochs, CIFAR-10

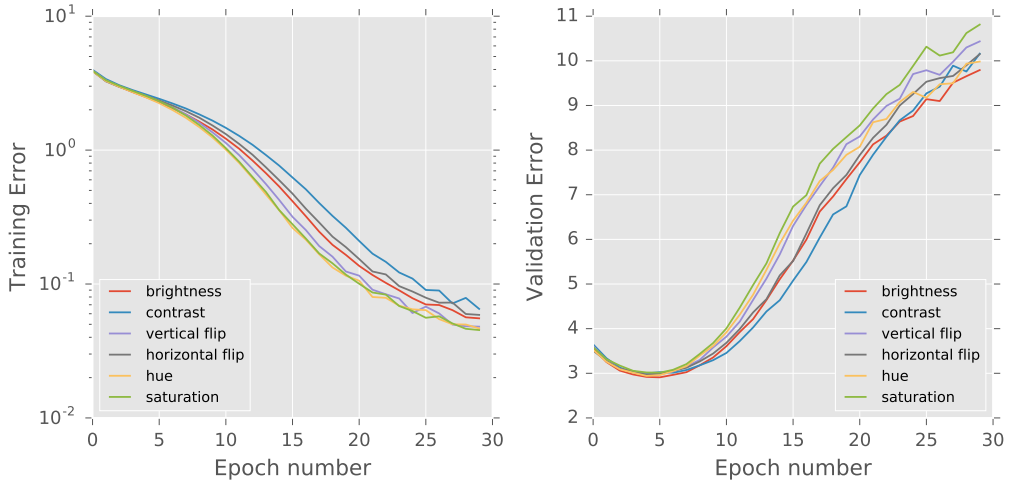


Figure 12: Training and validation error with data augmentation, 30 epochs, CIFAR-100

Analysis and Conclusions

For the CIFAR-10 images, it can be seen in Table 5 that flipping the image vertically gave the best final values, achieving a validation accuracy of 0.61. Horizontally flipping them did not have the same effect. Tampering

Distortion	Training Error	Training Accuracy	Validation Error	Validation Accuracy
CIFAR-10				
Vertical Flip	0.12150	0.96340	**2.21541**	**0.61640**
Horizontal Flip	0.01550	0.99812	3.81278	0.56639
Brightness	**0.01301**	0.99870	4.12326	0.55210
Saturation	0.03035	0.99252	3.04135	0.57649
Contrast	0.01862	**0.99795**	3.58172	0.56999
Hue	0.01984	0.99710	3.31451	0.58689
CIFAR-100				
Vertical Flip	0.04804	0.99257	10.43539	0.24230
Horizontal Flip	0.05882	0.99075	10.14976	0.23919
Brightness	0.05543	0.99075	**9.79288**	0.24650
Saturation	**0.04531**	0.99257	10.81169	0.22779
Contrast	0.06521	0.98890	10.16089	0.23440
Hue	0.04662	**0.9924**5	9.98359	**0.24989**

Table 5: Final error values with data augmentation, 30 epochs training

the image's Hue and Saturation produced good results, but changing the brightness did not. This can be seen in Figure 11 as well. Even if the training error from vertical flipping was the highest, its validation curve is the smoothest of the transformation curves, that does not immediately rocket but follows a steadier incline. Nevertheless, it should be noted that all the models delay the overfit by a couple epochs.

For the CIFAR-100 images, changing the Hue produced the best validation accuracy (0.249), followed by changing the Saturation (0.246) and vertically flipping the image (0.242), as shown in Table 5. Regardless, the plots in Figure 12 do not show any improvement with regard to the overfitting behavior of the network after 30 epochs. Even so, signs of overfitting are seen after the 5th epoch, which shows that it is delayed by a few epochs, compared to the baseline's in Figure 3.

What was expected from these experiments, given the conclusions drawn in Coursework 2[7] was for the model to be able to perform better after training with images altered in a more 'humanly' acceptable way. That is, after transforming the images in a way that they are not the same, but still recognizable, the model should be less prone to overfitting, as the 'new' (transformed) training data would make it more robust. In Figure 10 it can

be seen that the vertically flipped maintains in fact familiar features. On the other hand, the brightened image looks more alien. On the first case of the more natural transformation, given the way a CNN makes predictions, it can still isolate and learn the necessary features to identify the image as a cat. The same cannot be necessarily accomplished on the second case; the brightened image is distorted.

The expected effect of data augmentation techniques was achieved in CIFAR-10 dataset at an extent. The models showed delayed signs of overfitting, and produced not as steap of an inlcine in their curves. On the other hand, it barely affected the CIFAR-100 model. This was anticipated at an extent. The CIFAR-100 dataset is more complex. There are far less training data available for its 100 classes, compared to the exponential amount available for the 10 classes on CIFAR-10.

4 Conclusion

In this project we explored more complex neural network techniques on the task of image classification in the CIFAR-10 and CIFAR-100 datasets. In particular, we experimented with regularization techniques on convolution neural networks. We defined a baseline model that shows signs of overfitting and attempted to mitigate this behavior. We used knowledge from the previous courseworks, where we trained a simple feed-forward network for the same task of image classification, and compared the performance of the networks.

There are some conclusion to be drawn here. For starters, contrary to Coursework 3, we did not implement the same baseline architecture for both CIFAR-10 and CIFAR-100. We specifically targeted each image dataset, noting their individual performance. Nonetheless, with the exception of the data augmentation experiments, all the regularizing models affected similarly both datasets. The performance of the initial models was improved and their overfitting tendency was mitigated, or even alleviated altogether (section **3.2**, 0.001 L2 weight penalty). This concludes that the methods used here are not affected by the data itself, and can help towards the generalization of any model. Indeed, L2 weight penalty and dropout are widely accepted and used.

Moreover, the models that gave the best final values on the validation set were the vertical flip on the CIFAR-10 dataset and changing the Hue of the images on the CIFAR-100 dataset. This can be explained by the training process of a CNN. As was mentioned in the introduction, a convolutional neural network learns by splitting the image in smaller pieces. This al-

lows it to learn features of the image. When we flip or change the color properties, the model can train better, since we allow it to learn the original features, but at the same time, making it more robust by training to recognize them if altered (example: Original Image compared to Vertical Flip on Figure 10).

Regardless, the final values of the models on all experiments are considerably different between the two datasets. This can be easily explained, since the CIFAR-100 dataset has far less training data available per class, compared to CIFAR-10. There are, after all, 100 classes, which deems the CIFAR-100 classification problem exponentially harder.

Overall, similarly to the previous coursework, L2 weight decay proved to be the best regularization method for our CNN models. For 0.001 penalty the resulting models after 30 epochs are still not showing any signs of overfitting. Dropout proved to be a sufficient regularizer, as well, given an appropriate probability. Transforming the data did not produce as good results as were expected.

Possible Extensions Given sufficient time and resources, it would be interesting to experiment with stacking a significant amount of convolutional layers, or even whole models with the baseline's architecture (ConvNect/ReLu layers followed by a F-C) and see how these techniques affect more complex architectures. What could also be done, even in this architecture, is test different dropout probability on the convolution layers from that on the fully-connected ones. Finally, a big impact on the model would have the application of batch normalization, which we experimented with in courseworks 2 and 3, and showed to reduce overfitting.

References

[1] http://people.csmail.mit.edu/torralba/tinyimages

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[3] Machine Learning Practical
Coursework 3: Baseline Experiments on CIFAR-10 and CIFAR-100
s1641718

[4] Gal, Yarin, and Zoubin Ghahramani. "Bayesian convolutional neural networks with Bernoulli approximate variational inference." arXiv preprint arXiv:1506.02158 (2015).

[5] Sra, Suvrit, Sebastian Nowozin, and Stephen J. Wright. Optimization for machine learning. Mit Press, 2012.

[6] Ng, Andrew Y. "Feature selection, L 1 vs. L 2 regularization, and rotational invariance." Proceedings of the twenty-first international conference on Machine learning. ACM, 2004.

[7] Machine Learning Practical
Coursework 2: Training MNIST
s1641718

[8] https://www.tensorflow.org/