Final Report for CNN on CIFAR-3

Jake Parker
Heng Zhang

1. Problem Statement

Given a dataset consisting of training examples (features, and labels) and testing examples, we want to design a deep convolutional neural network, and train it on the training examples, in such a way as to maximize the percent of correctly classified test examples.

2. Data Preprocessing

2.1. The CIFAR Dataset

CIFAR-3, a dataset for computer vision aided object recognition, is a subset of CIFAR-10, which itself is a subset of the 8- million tiny images dataset. Whereas CIFAR-10 has 10 classes (image types), CIFAR-3 has, you guessed it, 3 classes. Our particular dataset consists of 12000 training images and labels, with 4000 images per class. And 3000 test images with 1000 images per class.

2.2. Data Cleaning

We cleaned our data, by first reshaping each dataset to dimensions (num_examples, width, height, channels). We then normalized each example by dividing the pixel intensity values by 255, and then randomly shuffled the cleaned dataset.

2.3. Data Augmentation

We used the ImageDataGenerator, provided by keras to generate permutations of our training data. We experimented with several different features — used to construct permutations. We experimented with ZCA Whitening — as the output PCA Whitening does not resemble the input image, which is a problem when it comes to CNNs. Additionally we employed various positional translations of input images, namely shift_range, shear_range, and zoom_range, to make out model robust to slight changes in object position within an image. Additionally, we flipped images horizontally, to make sure our model generated features (via CNN) where robust to the viewer's perspective. [1] Finally, we implemented an extension of the provided ImageDataGenerator function, so that we could create permutations via cropping the input image. We later discarded this functionality, as randomly cropping the input images adversely impacted classification accuracy. And of course, the above data augmentations were generated at random. We found that when we augmented the training data, it was better to only augment the validation data via shift_range, as to not muddle the input for validation.

2.4. Cross Validation

Instead of using k-fold cross validation, we simply created a validation set from 1/8th of the initial training set. The training set had 12000 examples, so our validation had 1500 examples, while the testing set had 3000 examples. The number of validation examples from each category was uniform, 500 examples from each category.

3. Model Implementation

3.1. Architecture

Our model is an extension of LeNeT, a 7-layer convolutional neural network. [2]

The architecture of LeNeT, can be thought of as four sequential segments: the first and second segments being a single convolution followed by a max pooling layer. The third segment is a flattening layer followed by a single fully connected layer. And the fourth segment is a fully connected layer followed by a softmax output layer.

We modified this layout, to include dropout units after the first, second, and third segments. We then, after some trial and error, further extended this by doubling up two convolutional layers before the max pooling and dropout layers in segments one and two. [3]

3.2. Model Selection

We iterated the design of our model three times, resulting in 5-layer, 9-layer, and 12-layer models. We then compared the validation accuracy of each model trained over 20 epochs, with batch size of 64, without any data augmentation or optimization.

Validation Accuracy after 20 Epochs on Tesla K80 GPU	
5-layer	0.9387 (1014 seconds)
9-layer	0.9458 (1313 seconds)
12-layer	0.9459 (2218 seconds)

As you may notice, increasing model depth has diminishing returns on validation accuracy, while increasing model depth greatly increases training time. For this reason, we decided on using the 9-layer model, as it outperformed the 5-layer model, and wasn't much worse than the 12-layer model, while retaining nearly the same training time as the 5-layer model.

3.3. Parameter Selection

We used a 3 by 3 kernel with 1 by 1 stride for our convolutional layers, and a 2 by 2 filter with 1 by 1 stride for our max pooling layers. For the inner product layers, we used the Glorot normal initializer (Xavier normal initializer). [4] This parameter is chosen based of of state-of-the-art deep learning network configuration tricks mentioned in the cited paper. The paper also suggests that we choose 2^n filters. According to a sample keras cnn [5], we chose 32 and 64 as the number of filters in the convolutional layers in the first and second segments, respectively. This makes sense, as we want each convolutional layer to generate new features for deeper layers. And a multiple of two does this.

3.4. Activation Function Selection

After we selected the 9-layer model, we compared the validation error after 20 epochs of the model with each of four different activation functions: sigmoid, tanh (hyperbolic tangent), relu (rectified linear unit), and

prelu (parametric rectified linear unit). [6] Without going into further detail, the relu unit outperformed the other units. The convergence time of the relu unit was significantly shorter than that of the prelu unit, which was much harder to train — as it increased the parameter space. We reasoned that we only needed a simple deep cnn as our model only needed to predict one of three possible classes. Perhaps for CIFAR-100, where there are 100 possible classes, that requires a much more complex deep cnn, the prelu unit would have been more useful.

3.5. Optimization

3.5.1. Optimization Method

We chose between two loss functions, sgd with momentum and adam. Using adam we achieved a validation accuracy of ~96% after 100 epochs. However, using a lower learning rate (0.001) with sgd with momentum (0.9) results in comparable validation accuracy. The deciding factor, was that adam was much quicker to converge than sgd.

3.5.2. Tuning of Data Augmentation Method

Earlier, when preprocessing our data, we to augmented our training dataset, by generating permutations of our training data. After testing various combinations of augmentations, we chose horizontal_flip, zoom_range, and other positional augmentations to adjust the images for better accuracy. [7] This allowed our model to be robust to slight changes in the position or rotation of training examples in each category. This augmentation improved our model's accuracy by about 0.8%.

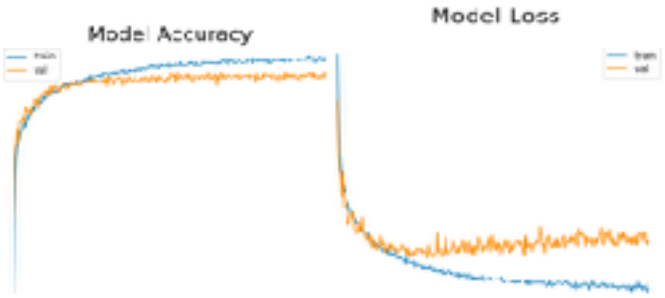
3.5.3. Parameter Tuning

We tuned our dropout parameters, initially set to 50%, we found that for such a small dataset, 20% dropout was sufficient to prevent overfitting. Similarly, we decreased the number of weights in the first fully connected layer, from 512 weights to 128 weights. These two adjustments significantly reduced the convergence time of our model's error rate.

3.5.4. Hyper Parameter Tuning

We initially used a learning rate of 0.01, with 50 epochs with a batch size of 251. We were able to significantly decrease training time, while decreasing the validation error rate by lowering our model's learning rate to 0.001, increasing the number of epochs to 200, and reducing each batch to a size of 64.

4. Analysis of Results



The metric plots are shown above.

Validation Error Rate after 200 Epochs on Tesla K80 GPU
0.9729 (12434 seconds)

As we can see, the increase in model accuracy per epoch is very large for the earlier epochs, and drops off significantly for later epochs. Additionally, we can see where the model overfits the training data, once training accuracy exceeds validation accuracy. Similarly, the loss function shows an inverse relationship per epoch, as compared to the model accuracy. In both cases continuing to train for additional epochs, after them model has begun to overfit the training data, exhibits significant diminishing returns on validation accuracy (and test accuracy, not depicted).

In accordance with these results, our final test accuracy (~96.9%) is less than our final validation accuracy (~97.8%). This is because the loss function has already converged.

5. References

[1] https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

[2] https://autolab.andrew.cmu.edu/courses/10601-s17/assessments/homework7/writeup

[3] https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py

[4] https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[5] https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py

[6] https://arxiv.org/pdf/1502.01852.pdf

[7] https://zhihua.com/questions/25097993s