10-601 Final Report

Jacob Buckman
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jacobbuckman@cmu.edu

John Corbett
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jtcorbett@cmu.edu

1 Introduction

Motivation

The goal of this project is to devise a machine learning classifier that performs well on the CIFAR-10 dataset. This dataset contains $50,000$ labeled 32×32 RGB images divided into 10 classes. For this project, we worked with a subset of this data, which contained $5,000$ images. To classify the images, we utilized three different machine learning techniques: a Neural Network (NN), a Gaussian Naïve Bayes classifier (GNB), and a k-Nearest-Neighbors classifier (KNN). Features were extracted with the VLFeat toolbox's HOG feature function. The data was then preprocessed using several different techniques, including normalization, PCA, and whitening. Each algorithm then underwent hyperparameter tuning after 5-fold cross-validation. Using our techinques, we managed to achieve better than baseline accuracy on all three approaches[1].

Background and Related Work

The CIFAR-10 dataset is a subset of another dataset, called the '80 million tiny images' data set. Each CIFAR-10 image has been labeled with a class, from a choice of: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. This dataset has several advantages, especially for a novice to machine learning. Since the images are only 32x32 pixels large, the default feature set is relatively small, and so it is far easier to process these images with simple algorithms, limited time, and meager computational power. Additionally, since there are only 10 classes, and they tend to be quite diverse, it is somewhat easier to obtain high accuracies than on a database like CIFAR-100, which has 100 class labels.

Before beginning work on our algorithms, we researched state-of-the-art approaches to solving the CIFAR-101 classification problem. Almost universally, the besst solutions used some variation of convolutional neural net. Some examples include Lee and Xie's 2015 paper *Deeply-Supervised Nets*, which reported an accuracy of 91.78%, and Goodfellow, Warde-Farley, and Mizra's paper on *Maxout Networks*, which reported an accuracy of 90.65%. In fact, it was quite difficult to find any academic paper written in the last five years that did *not* attempt a convolutional neural net-based approach. Because of this, we decided that this would be the best approach for us to pursue as well. (However, implementing convolutional neural nets proved to be too complex in the end, and we simply utilized a standard fully-connected neural net instead.)

One invaluable resource during the development of this project was a Stanford online course called "Convolutional Neural Networks for Visual Recognition". Although we did not end up getting the final convolutional neural net to work, this course gave a detailed explanation of all intermediate steps, many of which we were able to implement. It also describes a large number of practical optimizations for improving the performance of neural nets accross all tasks, some of which were useful for this task.

[1] We will take baseline accuracy to be 48%

2 Methods

In this section, we will discuss the various techniques used to obtain our three accuracies.

Feature Extraction

Since the images that we are classifying are only 32×32 pixels, we initially took each pixel as a feature for training the classifiers for a total of 3072 features. We soon found that it was useful to run feature extraction before running the classifiers on the image data, and to train the classifiers on the features of the images rather than the raw image data. Therefore, before images were even preprocessed, we converted them into new feature sets using three types of feature extraction: none, HSV, and HOG features.

HSV stands for Hue-Saturation-Value, and is an extremely common way to represent colors, alongside RGB. One of the advantages of HSV over RGB is that it is much more useful for adjusting the color by hand in a program like Photoshop - though it maps to the same color space as RGB, HSV's three values are much more easily interpretable to humans. We hypothesized that this increased semantic coherence would correspond to a better feature set for image classification.

HOG stands for histogram of oriented gradients, and is a very commmon technique in image feature extraction. HOG splits the image up into regions, and calculates the frequencies of appearances of various gradient directions within each region. The final feature set is simply the list of all these gradient frequencies. HOG has an adjustable parameter corresponding to the size of each region. In this project, we experimented with region sizes of 4, 8, and 16. We used the `VLFeat` library to convert the image into vectorized HOG descriptors.

Data Augmentation

Though we were limited to the 5000 images provided to us, there exist techniques for getting more training data out of a limited subset of images. The key observation for how this is done is that, in general, if you flip an image on its horizontal axis, the resulting image is still a picture of something of the same class. Thus, we were able to augment the data set with an additional 5000 images, corresponding to the flipped versions of each of the original images.

Preprocessing

After applying the feature extraction, we center the data by subtracting the mean from each feature, then we optionally normalize the data by dividing every feature by its standard deviation. Then we perform Principal Component Analysis (PCA) on these features to decorrelate the features. This is done by rotating the data into the eigenbasis of the feature covariance matrix. We can also optionally perform dimensionality reduction in this step by retaining only the basis that contain the most variance; this is useful because we can significantly reduce the number of features of each sample while still accounting for a large portion of the variance. The final step of the preprocessing stage is whitening. Whitening scales each dimension by its corresponding eigenvalue, reducing the features into an "isotropic Gaussian blob."

Neural Network

Neural networks have been shown to be extremely effective in image processing tasks due to their ability to learn features at many different levels. For this project, we utilized a fully-connected neural network with a variable number of hidden layers. The activation function at each node was computed by a leaky rectified linear unit, and the overall error was computed using the cross-entropy loss. The input layer had a number of nodes equal to the number of features of the input data, and the output layer had 10 classes, each corresponding to a class. A Softmax function was used on the outputs of the network to calculate estimates of their relative probabilities, and the class with the highest probablitiy was selected as a guess. The network was trained by standard backpropagation with stochastic gradient descent.

There were a number of design decisions made, many of which also utilized tunable hyperparameters.

Activation Function

Initially, the neural net was programmed with a sigmoid activation function. However, after research, it was determined that the rectified linear unit would likely perform better. The ReLU function simply maps any positive number to itself, and any negative number to zero. ReLUs are currently the most popular type of nonlinearity, and networks using ReLUs have been shown to converge faster than either sigmoid or tanh activation functions.

One further optimization that was made was the conversion to a Leaky ReLU. One suboptimal property of ReLUs is that it is possible for them to get 'overloaded' with input on one iteration and be 'dead' for the remainder of the training - that is, they receive an input so high that their bias becomes too high for them to ever activate again. This can be solved by adding a small 'leak' to the negative side of the ReLU, which causes the bias to always have a gradient, and so decrease back to reasonable values over time.

The ReLU activation function and its derivative:

$$ReLU(x) = \begin{cases} 0 & x \leq 0 \\ x & else \end{cases} \qquad \text{and} \qquad ReLU'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & else \end{cases}$$

Loss Function

The loss function utilized was cross-entropy loss. This is standard for classification problems.

The cross-entropy loss function:

$$L_i = -\log(\frac{e^{o_{y_i}}}{\sum_j e^{o_j}})$$

where o_x is the output of the network at class x and y_i is the correct label for training example i.

Regularization

In order to prevent overfitting, we added a regularization term to our loss function. We chose to use L2 regularization, which creates a penalty for each weight equal to the square of its value. This causes weights to tend to prefer lower values, and tends to cause wegihts to be spread out over multiple smaller numbers, rather than concentrated in a few large weights. This has been shown experimentally to reduce overfitting.

The regularization term is multiplied by a parameter λ, which determines how big of an influence the regularization has relative to the cross-entropy loss. Some typical values we tried for λ include .1, .01, and .001.

Layer Architecture

Designing a neural net optimally is a very difficult problem - all of the resources we consulted said something along the lines of 'it is an art'. To determine our architectures, we just played around with things that seemed reasonable, guided by the resources we could find. One source mentioned that network depths greater than 3 or 4 tended not to show much further improvement, and another said that the layer sizes should generally be smaller than your number of features. Some example architectures we tried include [500 500 500], [1000 500 250], and [500 250 100].

Weight Initialization

To start off, the neural net must have some initial values for weights and biases. In general, the best approach here is to use small random numbers, so the neurons differentiate themselves and cover different parts of the feature space, but the model accuracy is not too dependent on the initialization. As discussed in *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, rather than select numbers purely at random, it is better to take numbers from a standard normal distribution and multiply them by $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the layer.

Stochastic Gradient Descent

At each iteration of the algorithm, for each point, the point was fed forwards through the network, and then the errors were propagated backwards through the nodes to calculate a gradient at each weight. In standard backpropagation, every point contributes to this gradient every epoch, before any of the parameters are actually updated. Since we are training on a lot of data points, it doesn't make sense to wait until every single one has been processed before doing a parameter update. Instead, we process one 'batch' of data at a time, and then do the parameter update immediately. Batch size is an adjustable parameter; some values we tried were 1, 10, and 100.

Learning Rate

One of the most important factors in making a neural net perform effectively is correctly setting the learning rate. Too low, and the network takes prohibitively long to converge; too high, and the network will get stuck at suboptimal locations, as the algorithm 'oversteps' the minimum on its gradient descent. For this experiment, we tested several values of learning rates, ranging from 1 to .001. Due to the limited time constraints of this project, we leaned towards higher learning rates, which would allow us to conduct more trials faster (and get an idea, in a broad sense, of how well each model did). If we had more time, we would like to run more trials with lower learning rates, which would likely increase the network's ability to find optima, and thus, the accuracy.

The learning rate was also annealed over time - every 10 epochs, the learning rate was decreased by a factor of 2. This allowed the network to account for the fact that, as the interations go on, it becomes necessary to take smaller and smaller steps. Annealing allowed us to get some of the benefits of both a large and small step size - in the initial epochs, it would move quickly towards the general location of the optima, but in later epochs, it would slow down and become more precise.

Momentum

In order to increase the rate of convergence of the network, it has been shown to be useful to model the momentum of the gradient. Instead of simply recalculating the gradient from scratch at every point, a momentum-based model preserves a velocity matrix v, and updates the weights using v during every parameter update. At each step, v is calculated by adding the previous value of v, times some friction factor μ, to the new gradient update. In this manner it preserves some of the information from the old update. μ is an adjustable parameter; typical values tried were .5, .25, .1, and 0 (no momentuf),

Dropout

Dropout is another technique for increasing the generalization of the neural network. In dropout, some fraction of then nodes p are "dropped", meaning that their activations are artificially set to zero during the feedforward step. The remaining nodes in the network then perform one training iteration as normal. This has the effect of turning the neural net into a sort of ensemble classifier - each subset of nodes from the original neural net gets good at identifying certain examples, and the overall net gets an output composed of votes from each of these subsets. Like most ensemble classifiers, this has the eventual effect of having wrong influences from non-confident subsets cancel out, leaving only the most confient nodes voting.

Gaussian Naïve Bayes

We choose to use a GNB classifier over a regular Naïve Bayes classifier because even though the data points are discrete, they can take on a wide range of values in a continuous fashion which lends itself more to the Gaussian version of the classifier. We use a maximum likelihood estimation (MLE) when training GNB, so there is no fabricated data and the resulting model will maximize the likelihood of the training data occurring. Admittedly this could lead to over fitting, and perhaps using a Maximum a Posteriori (MAP) estimate would combat this.

GNB relies on the assumption that each of the features are independent of each other. Given that this is clearly a faulty assumption, we do not expect to achieve very high accuracy with this classifier, but with good feature detection and preprocessing, we aim to hit at least baseline accuracy. Our

implementation of GNB does *not* assume that the variance is independent of the training samples or labels. Even though we do not expect this classifier to make high accuracy classifications, it does run very quickly, making it a prime candidate for real-time classification.

k-Nearest Neighbors

KNN works by comparing an image to every image in the training set, then taking the k-closest images from the training, and classifying as the most common label in the k-closest. In our implementation of KNN, we check to see if the set of k-closest images is multimodal, then we check each mode and select the label from the training sample whose features most closely match the test sample. Because all of the work is done in the testing phase, training the KNN is as simple as just saving the training data, training labels, and model parameters. Because of this, the KNN model is the largest and also takes the longest to test of the three models. Given the simplicity of this model, we expect it to be about on par with GNB.

Other Classifiers

In addition to NN, GNB, and KNN, we also attempted a few other classifiers with less success. We built a working Support Vector Machine (SVM) with promising results, but the algorithm relied heavily on the Octave Quadratic Programming solver (`qp`) which was prohibitively slow. To train an SVM using `qp` on only 250 images took several minutes, and training on more images caused the amount of time required to grow very quickly. Because of this, there was no way to feasibly train our SVM on the entire training set of $5,000$ images, so we ruled that out as a potential classifier.

We also attempted to build a Random Forest (as described in our project proposal) but ultimately found that this algorithm is exceedingly difficult to make in Octave, thus abandoned the idea in favor of KNN. We also began work on was a Multiclass Logistic Regression classifier; however, this was abandoned when we managed to get KNN above our target accuracy with hyperparameter optimization.

Finally, we attempted to add a convlutional layer to our neural network, turning it into a convolutional neural network. As mentioned in the Background section, almost all state-of-the-art classifiers have been built with deep CNNs, and building one was our original plan. However, due to the difficulty of the algorithm and time constraints, we were unable to create a functional CNN before the deadline.

Hyperparameter Optimization

In many of the sections discussed above, there were a large number of tunable hyperparameters. We ran a grid-search form of hyperparameter optimization to maximize the expected efficacy of our techniques. Basically, for each parameter, we selected six to ten possible values, based on intuitions and research. We then tested every possible combination of these parameters to see which ones scored the highest. The highest-scoring hyperparameter assignments were used for our final models.

In order to conduct these tests efficiently, we set up a Linux server that we used to conduct several tests on the models. The purpose of this server was to allow for continuous testing on various hyperparameters, preprocessing techniques, and classifiers. The server ran Octave version 4.0.0, which was compiled with the CUDA toolkit provided by NVIDIA. The source code of Octave was modified to use `cublas_dgemm` for double precision matrices, which takes advantage of the GPU to more efficiently perform matrix operations. In practice we found that this results in a significant speedup which allowed our server to conduct tests faster.

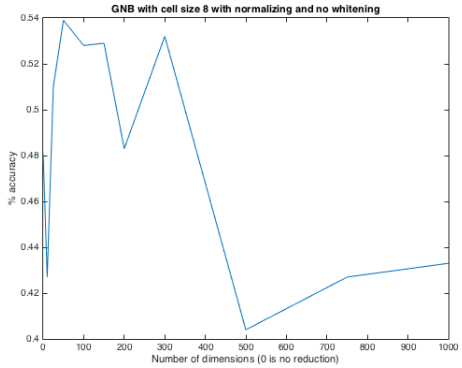
However, even with this speedup, the search space of the neural net parameters was simply far too large - there were sixteen different hyperparameters in the model, in addition to the ability to set the number of layers and of hidden nodes in each layer. Therefore, optimization of the neural net was guided primarily by intuition and trial-and-error, rather than exhaustive search.

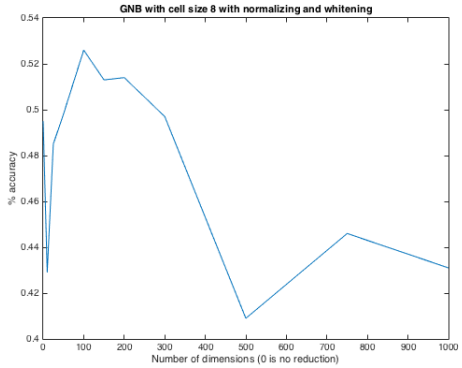
Cross-Validation

Given that we only have a 5000-image subset of the CIFAR-10 dataset, maintaining a validation set becomes a problem. When determining the efficiency of the preprocessing and the classifiers, we used 5-fold cross valudation. This means we randomly selected a set of 4000 images from the data set for validation, and then trained a model on the remaining 1000, under the assumption that success in classifying the validation subset would directly translate to success in classifying the entire CIFAR-10 dataset. After this, we selected another unique set of 1000 images, and so on, until we had fully partitioned the dataset.

3 Experiments

In order to determine the best parameters for our models, we wrote scripts to iterate through many combinations of hyperparameters. We then graphed the results of these tests to find the best hyperparameter combinations for each model. These scripts tested features of the preprocessing step such as normalization, whitening, number of dimensions and the cell size for HOG descriptors as well as the parameters of the actual classifiers themselves such as the number of neighbors to use for KNN.





4 Results

Preprocessing

Through hyperparameter testing on both the preprocessing step and each of the individual classifiers we found that the success of classification is heavily dependent on feature extraction and data preprocessing. We discovered that HSV values actually hurt the performance of the model in almost every case, thus primarily tested on RGB image data. Normalizing and whitening can either help or hurt the accuracy depending on the model; they can be effective when used properly.

Performing HOG feature extraction with `VLFeat` had the biggest impact on the accuracy of our models. Without using HOG feature extraction the GNB and KNN classifiers were scoring had about 30% accuracy at best, while the Neural Network was peaking at about 50%. After applying HOG feature extraction, GNB and KNN shot up to about 50% accuracy and the Neural Network went up to about 60%. The PCA and dimensionality reduction also helped universally, but with less impact than the HOG feature extraction. Through experimenting with dimensionality reduction and performing PCA, we were able to boost the accuracy of each model by about $1 - 3\%$. With good preprocessing, our simple KNN and GNB classifiers were able to score within 10% of our Neural Network.

Neural Network

As expected, the Neural Network greatly outperformed the GNB and KNN in terms of accuracy, but the GNB performed much faster in both the training and the classifying steps. It also outperformed the KNN in terms of testing speed.

Due to time and computation constrains, we were unable to fully evaluate the optimal combination of parameters for the neural network. However, the best combination of features we were able to find was: HOG features with cell size 8, centered and normalized, Layer Architecture: 1000-500-250, learning rate .1, regularization λ value .05, no dropout, no momentum, the batch size 10, and the evaluation set size 100. It was trained for 20 epochs on a data set of 10000 points. This gave an accuracy of 60% on the development set, and an eventual accuracy of 57.3% on the Autolab test set. We were satisfied with this accuracy, as it is significantly above baseline. However, we are confident that with more time and resources, it would be easy to get this accuracy to over 60%.

Gaussian Naïve Bayes

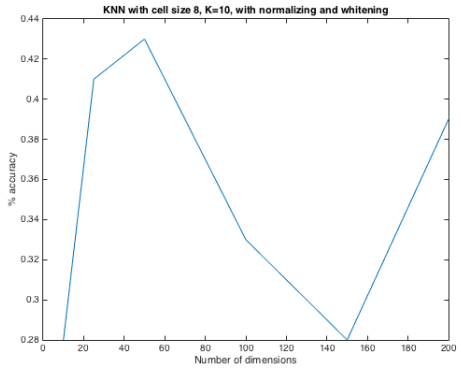
Though it is a relatively simple algorithm, Naive Bayes performed surprisingly well for this task. It was by far the fastest algorithm we tested, both to train and to test. It also has the smallest model footprint.

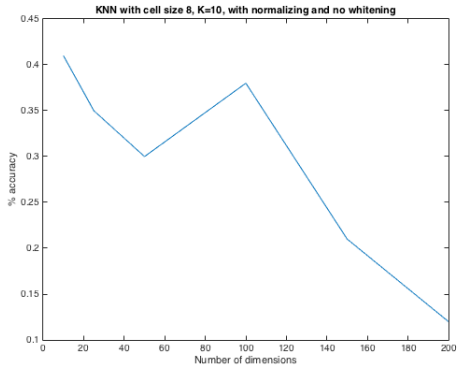
After performing an exhaustive hyperparameter search, we discovered that the best hyperparameter for GNB were: HOG features with cell size 8, centered, normalized, PCA'd, and whitened, and dimensionality reduction down to 100 dimensions. Testing on these settings gave us a maximum development accuracy of 57% and an Autolab accuracy of 53.5%.

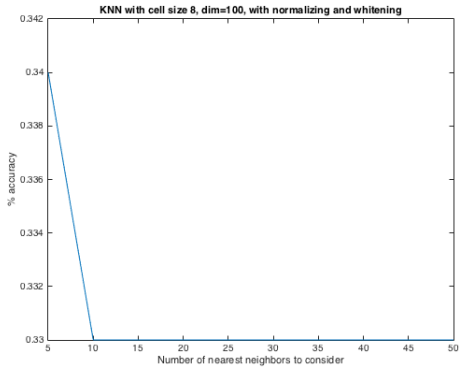
k-Nearest Neighbors

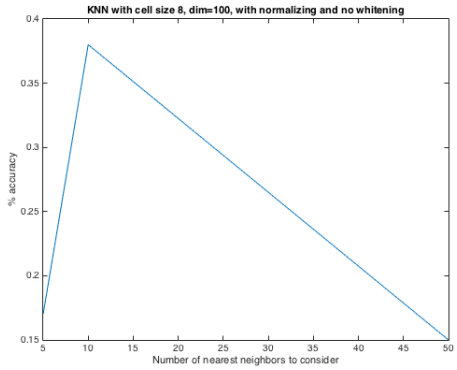
With HOG feature extraction, PCA, and dimensionality reduction, we were able to correctly classify 51.6% of the test samples using KNN. We found that normalizing and whitening actually hurt the accuracy of KNN, while HOG feature extraction, PCA, and dimensionality reduction tend to improve the performance. On the same training set and test set, KNN scored only 38.8% with normalizing and whitening turned on with 100 dimensions. Through experimentation, we discovered that KNN performs best with a higher dimensionality reduction (less dimensions) of between 60 and 80 dimensions.

We also discovered that KNN works best with K set to around 10. If K is too low, then we risk having a set of neighbors with no mode, which produces poor results, but we also found that if K is set too high, then the modes of the neighbor set could be decently far away from the sample, which also produces poor results. We discovered that 10 works well through hyperparameter testing.









5 Conclusion

Over the course of this project, we built, evaluated, and tested three machine learning classifiers: Neural Network, Gaussian Naive Bayes, and k-Nearest Neighbors. All of these methods were successful at performing the classification at above-baseline accuracy. The neural net was the most accurate, and the GNB was the fastest. Feature extraction in the form of HOG vectors proved to be extremely important, as did preprocessing operations like centering, normalizing, and dimensionality reduction. Hyperparameter optimization was another factor that allowed large increases in accuracies. Overall, our team was successful at classifying CIFAR-10 images, and learned a lot about how to implement machine learning algorithms.

6 Contributions

John implemented the prototype Gaussian Naïve Bayes Classifier as well as the k-Neatest Neighbors classifier. He also wrote the HSV feature extractor and the HOG feature extractor, as well as the code for performing PCA, dimensionality reduction, and whitening. The testing server was run from John's desktop and he set it up and maintained it.

Jacob implemented the Neural Network and expanded the Gaussian Naïve Bayes Classifier to use matrix multiplication, allowing it to be tested efficiently. He also designed a testing script that allowed for easy comparison between the different algorithms and hyperparameters. He wrote a platform for running the preprocessing step, which takes in the data and the preprocessing parameters and returns the preprocessed data as well as statistics on the population for later use. Jacob wrote the code to flip images for preprocessing as well as the code to normalize the data.

Both team members contributed to the sections of the paper corresponding to what they implemented.

7 References

http://cs231n.github.io/classification/#summaryapply
http://www.vlfeat.org/overview/hog.html
http://cs231n.github.io/neural-networks-2/
https://www.youtube.com/watch?v=P2Ew4Ljyi6Y
http://jmlr.org/proceedings/papers/v38/lee15a.pdf
http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf
In addition to these, we referred to the 10-601 lecture slides.