Object Recognition on CIFAR-10

NAIVE BAES

COMP 540 – Machine Learning

Ethan Perez (ejp2) Krishna Thiagarajan (krt3)

Abstract

The CIFAR-10 Object Recognition problem is a famous benchmark Machine Learning problem, which entails accurately classifying images of these classes: airplane, car, bird, cat, deer, dog, frog, horse, ship, and truck. Each image belongs to exactly one of those classes and is given in a 32x32 png image format, and researchers are given 50,000 images to train on and need to predict on 10,000 held out test images. It is a challenge which thousands of researchers have taken up, vying for the highest accuracy possible. Our goal is to investigate various methods for tackling the CIFAR-10 and see how well they perform, as well as combine these models in interesting ways to achieve higher accuracies. We find that running a Support Vector Machine on top of the features extracted using several Convolutional Neural Networks produces a model of 91.15% accuracy - close to human accuracy, which is around 94%, and ranking 9th place in the In-Class Kaggle Competition. In this paper, we will walk you through our journey, our thoughts, and our ideas, that ultimately led us to this model.

I. INTRODUCTION

WE began by testing the standard machine learning algorithms against the CIFAR-10 data, gleaning what we could from each trial. Here, machine learning tools like sci-kit learn, lasagne, and nolearn, as well as the research of other field leaders, were our best friends in getting us off the ground at at an astonishing speed.

Research has shown that the Convolutional Neural Network is a powerful way to approach the image recognition problem. Research has also shown that bagging or ensembling methods can be extremely effective in reducing the errors made by a single model. In addition, there is enormous evidence for the strength of the Support Vector Machine. Our fundamental insight was the combine these three methods to create an ensemble of Convolutional Neural Networks that fed into a Support Vector Machine to produce the final predictions.

However, we didn't immediately reach this conclusion. Before eventually coming up with our current approach, we investigated a variety of other machine learning techniques: Naive Bayes, Softmax Logistic Regression, One-vs-All Logistic Regression, Support Vector Machines, Fully Connected Neural Networks. We swept the hyperparameter space for each of these models and gave it the best chance before moving on, but eventually we realized that we absolutely needed to use a convolutional neural network, in some form, in order to attain maximum accuracy. From there, once we found a good architecture and set of hyperparameters for the CNN and achieved a solid accuracy, we tried to combine the CNN with the other models we tried, to see if those models could still perhaps have some positive effect on the accuracy past what a solid CNN could achieve. Somewhat to our surprise, we found that a Support Vector Machine "final layer" could indeed provide a substantial bump in accuracy; the CNN's that we used to extract features for the SVM had an accuracy of around 83% - the SVM brought that accuracy up to 91.15%

Our final approach, the SVM layered on the features learned by multiple CNN's, was guided by the idea that CNN's are able to powerfully process thousands of features into a handful of useful numbers which it uses to predict at the end. The inspiration for the use of a SVM came from the evidence that, in many cases SVM's outperform Logistic Regression, and if the last layer of a CNN is simply logistic regression, there's a good chance that an SVM could better perform the final conversion of the processed features into a single prediction. There is another fact as well that led us to use multiple CNN's, namely that the space of CNN parameters has multiple local minima which one might reach, each of which would result in a CNN learning a different set of features of the data. Thus, among several well-trained CNN's, our idea was that we'd have learned a large number of the helpful features that lead to these local minima in the CNN loss function, i.e. features that are very indicative about the nature of the image and thus ones that are prime for feature extraction.

II. Methods

Here is a summary of the methods we explored, what their results were when we tried them, and what we gleaned from trials with each method. Note that the way we approached model assessment for each of these methods was to try out various combinations of hyperparameters, through grid search or random search, and choose the hyperparameters that performed the best on a validation set, in order to mitigate overfitting (with a special emphasis here on regularization parameters like L1 or L2 lambda values or dropout probabilities), or simply just used the hyperparameters given by a research paper, as was the case for our later, more complex models. The validation set method allowed us to assess our model performance easily, so that we could tweak them until we were ready to submit to Kaggle and see the results on the test set.

I. Naive Bayes

We experimented with Gaussian Naive Bayes, Bernoulli Naive Bayes, and Multinomial Naive Bayes. For all of these methods, we simply used sci-kit learn's methods, binning the data into different bin sizes for the Multinomial Naive Bayes and Bernoulli Naive Bayes methods because those are discrete methods(we binned the data by taking the value for each pixel and putting it into one of *n* bins. Note that our Bernoulli Naive Bayes was essentially Multinomial Naive Bayes with 2 bins). We found that Gaussian Naive Bayes achieved 28.33% accuracy, Bernoulli Naive Bayes achieved 27.28% accuracy, and Multinomial Naive Bayes achieved a 29-31% accuracy, depending on the bin size (31% was achieved with bin size 256, i.e. no binning, just directly using the data). These poor results really highlighted to us how flawed the fundamental assumption of Naive Bayes was for the CIFAR-10 problem; the pixels were highly correlated in terms of classifying the image correctly. This led us to pursue other methods that would better take advantage of the relationships between pixels in the images.

II. Softmax Logistic Regression

Our team used the Softmax Logistic Regression that we implemented as part of Homework Assignment 3. We ran a grid search to find the hyperparameters and were able to achieve a 41.18% accuracy, using 100000 iterations, BatchSize of 1000, Learning Rate of 0.003, and L2 Reg of 1.0. Although this accuracy was far less than we needed to be competitive, using Softmax Regression helped us realized that Logistic Regression still was powerful, enough to get from the baseline 10% to the 40% mark, which is still substantial, and that the Neural Network would be a good point to move on from here, as Neural Networks act like stacked Logistic Regression units.

III. One-vs-All Logistic Regression

Our team also used the One-vs-All Logistic Regression scheme to classify the images, simply using our solution to Homework 2 to run on the images and produce a 41.56% accuracy, after a substantial hyperparameter sweep. We found that we achieve this best accuracy using 100.0 of L2 regularization. Similar to the results from Softmax Regression, these results pointed us in the direction of the neural network. Working with these lower quality models, it should be noted, also helped us work with the data and models in general - how to store models and data conveniently, how to effectively sweep for parameters (grid search vs random search) - as well as create a lot of the framework for the future models. So although these accuracies are rather unhelpful, they paved the road for our future advancements.

IV. Neural Network

Moving on to Neural Networks, having already used sci-kit learn substantially to manipulate and prep the data, as well as for the Naive Bayes models, we decided it would be easiest and quickest to simply use sci-kit learn's implementation of the Neural Network [5]. Their "Multi-Layer Perceptron" capabilities hadn't yet been released officially, but they had a very high quality Neural Network framework available on their developer version, version 0.18.dev0.

IV.1 Raw Pixel Data

Just on the raw pixel data, unrolled into a single vector for each image, we tried the sci-kit learn neural network in its various forms - different gradient descent algorithms, early stopping, shuffling data, momentum, etc. We also randomly searched the hyperparameter space (parameters like regularization, learning rate, number of hidden layers (only swept 1-5 here due to time constraints), the number of neurons per layers, etc. - essentially any hyperparameter we could search through the space of with sci-kit learn's Neural Network), since grid search would've taken too long to find the optimal hyperparameters and Neural Network options. We found the optimal hyperparameters to be 1679 Hidden Layer 1 Neurons, 1270 Hidden Layer 2 Neurons, 71

Hidden Layer 3 Neurons, RELU activation only, "Adam" Gradient Descent algorithm with Early Stopping, Learning Rate of 0.001, L2 regularization parameter of 0.001, batch size of 200, Nesterov's momentum of 0.9, β_1 of 0.9, β_2 of 0.999, epsilon 0f 1e-08, max iterations of 40, and tolerance of 0.0001. With these hyperparameters, we were able to achieve a 52% accuracy for a single model. Then, we used a common tactic for reducing the error due to variance of models, namely a voting scheme, where multiple neural networks of different architectures (other close-to-optimal architectures we found) and/or different weight initializations vote on the proper classification of the image, and this method brought our accuracy up to 55%.

This accuracy was decent and a solid improvement over previous models, and through implementing it, we learned the power of using multiple models in combination with each other, as well as the power of random search through the hyperparameter space as a way of figuring out optimal hyperparameters in a reasonable time, a method I'd read about in a paper written by Bergstra and Bengio [3]. However, this accuracy was still not enough, as we knew that other researchers had produced results in the 90%+ range, far greater than 55%. Searching for ways to continue improving our accuracy, we came across the idea of using data augmentation and feature extraction.

IV.2 Raw Pixel Data + Flipped Data + Feature Extraction

We learned that the Histogram of Oriented Gradients, i.e. the HOG representation of the data was a powerful way to represent an image that was used in many places for image related analysis and predictions. The Histogram of Oriented Gradients representation simply converts an image into an array of numbers representing the "direction" of the image at each pixel or group of pixels, information that is not directly or immediately contained in the arrays of pixels describing the image. In addition, we had read that another common data augmentation technique was to add all of the flipped images to the data; after all, a dog is a dog if it's facing the left or the right. When we augmented our features matrix with the HOG representation and the flipped representation of the data, we were able to achieve a single model accuracy of 57% and a voting scheme model of just above 59%. The single model optimal hyperparameters we found for the data after HOG and flipped data were added were all the same as the ones for the Neural Network without these forms of the data, except that now the best number of hidden layers we found was 4, with sizes 1638, 838, 209, and 59, in that order. To do the voting method, we found other models with similar accuracies, and these also all had the same hyperparameters except for the number of hidden layers and number of neurons in those hidden layers. After these experimentations, we knew that we still did not have enough accuracy to have a model of real significance, but we did learn the real power of neural networks and of data augmentation.

Moving on, although we did take into account the interrelatedness of pixels at a deeper level with Nueral Networks than we did with Softmax or One vs. All, we knew that the one piece of information we hadn't really utilized as well as other top performing models on the CIFAR-10 dataset did was the emphasized relationship between pixels that were close in proximity - something that was lost in the unraveling of the raw pixel data into a single vector for each example, but something that we could take advantage of if we used a convolutional neural network. In addition, the power of data augmentation and feature extraction was not lost on us, and this lesson would come in handy later down the line, as you will see.

V. Convolutional Neural Network

Noting that all the top models for the CIFAR-10 competition were Convolutional Neural Networks and seeking to take more advantage of the spatial proximity of the pixels in the images, we began to use Convolutional Neural Networks. Wanting to take advantage of the pre-existing, well-tested, advanced implementations of CNN's that were out there, we looked into various frameworks. We found that the main front runners were Torch and Theano, for their speed and wide use in top research papers. Caffe and Tensorflow (a library for Machine Learning written by Google) are right behind, however these are not as widely use, and in fact we tried Tensorflow for a full day before coming to the conclusion that the library, although powerful, would, for us, be much more difficult to understand, let alone modify meaningfully, than others that were out there. Having already written much code in Python and sci-kit learn (a Python library), we decided we'd go with Theano over Torch, a Matlab library. However, instead of using Theano directly, we opted to use NoLearn, a wrapper on Lasagne, which in turn was a wrapper on Theano. Furthermore, Daniel Nouri, the man who created NoLearn has a fantastic tutorial walkthrough, from which we used several functions, located in nn_utils.py (each one has a note in the docstrings) [1]. This allowed us to take advantage of Theano's lightning speed's, while allowing us to blaze past the allegedly steep learning curve of Theano.

We implemented a basic 3 convolutional layer neural network on my laptop, a Macbook Pro, and was able to achieve up to a 70% accuracy. However, this was with overnight model training and only allowed for a few hundred epochs (quite small compared to the often recommended thousands or tens of thousands of epochs); on my laptop a single decent model would've taken an incredibly long time to run, and sweeping for hyperparameters would have been essentially impossible. We slowly began to realize, especially after more research, that all the experts were using Graphic Processing Units to run their code. The GPU's hyper parallelization of multiplication and addition resulted in speedups upwards of 20x, allowing the researchers to train for more epochs, better sweep the hyperparameter space, and gain incredible accuracies. We began to realize that we needed our own GPU. We talked to our instructor, Dr. Devika Subramanian, and she recommended that we build our own computer, over using a cloud computing service such as Amazon's EC2, in an effort to avoid large costs drained, so that's exactly what we embarked upon.

V.1 Building the Computer

In an effort to speed up our models' runtimes, we knew we needed to run our models on a GPU. Because we only have laptops, ones without GPU's compatible with Machine Learning GPU code ("CUDA compiled" code), we decided to build our own computer. Thus, we spent upwards of 40 hours buying parts required for building a computer and assembling those parts into the machine that could run our code with a speedup of 20-30. I will now talk about what we learned throughout that process, and how we went about building the computer, for any readers who might also need to build a heavier-duty computer to run their computationally intensive models: We first learned the parts needed for a computer:

- CPU
- Motherboard
- Power Supply

- Random Access Memory
- Hard Drive and/or Solid State Drive
- Case (technically optional but basically required)

For our purposes, we also needed a GPU. We opted for a GTX 950, which isn't the most incredible GPU on the market but makes up for that due to its cost efficiency. While building the computer, we learned about the functions of each component of the rest of the computer:

- The CPU acts as the brain, and does (most) of the computing. The better the CPU, the better the computer is able to handle multiple processes, so I opted for a newer CPU, an Intel i5-6th generation.
- The Motherboard acts as the nervous system, connecting the CPU to all the other parts of the computer. The nicer Motherboards have more plugs for more advanced types of memory and storage, are easier to plug into, and can handle more load.
- The power supply acts as the heart, bringing power to all of the components of the computer. Since we are running more heavy duty machine-learning algorithms, we decided to opt for a more powerful power supply, with 750 Watts of power output
- Random Access Memory (RAM) is memory that can be accessed in constant time, but loses its contents when it powers off. For this reason, RAM is used for temporary use by the operating system and applications or programs. The more RAM we have, the more programs I can run, so we got 16 GB of DDR4 RAM
- For long term storage, something that RAM can't do, we use a mix of Hard Disk Drive (HD) and Solid State Drive (SSD). SSD has no moving parts, and is technically superior to HD. Furthermore, due to quick access times, booting up the computer with an SSD is incredibly fast. However, HD is very cheap, with 1 terabyte of HD space worth a mere 40 dollars on the market. For that reason, we bought 1TB of HD and 250 GB to optimize for both cost and performance.

- The Case isn't necessary, but as we learned from experience, it's not the best idea to cheap out on the case. The bigger the case, the easier to assemble the computer. We ended up having to get a new case because the GPU (which is much larger than the CPU), didn't fit in our older case.

We decided to forgo extra cooling fans because keeping the room cold would be sufficient. Once we bought the parts, we then began the process of building the computer:

1. Screw the power supply into the case.
2. Snap the CPU into the Motherboard in the appropriate place. Be careful with this step!
3. Screw the Motherboard into the case. Should be right next to the power supply. Be careful with this as well!
4. Snap the RAM in place on the Motherboard
5. Snap the CPU cooling fan on top of the CPU on the Motherboard. The thermal paste on the fan should stick to the CPU.
6. Plug in the following:
   - Cords from the power supply to the motherboard
   - Cords from the case to the motherboard
   - Cords from the power supply to the case
   - Cord from the power supply to the HD
   - Cord from the power supply to the SSD
   - Cord from the motherboard to the HD
   - Cord from the motherboard to the SSD
7. Screw the HD and SSD into the side of the Case (There should be a rack next to the motherboard that has holes for screwing those two pieces of hardware into the case)
8. Snap the GPU into the Motherboard

9. Plug the cord from the power supply to the motherboard

This process was time-consuming, pretty exhausting, but incredibly fun and educational. It was akin to building a huge Lego project, except a lot more strength and dexterity was needed, especially for the motherboard, since the motherboard has a lot of open circuitry, and scratching that circuitry could ruin the computer.

After assembling the computer (successfully after buying a bigger case), we hooked the computer up to a monitor and downloaded a distribution of Linux called Ubuntu (version 15.10) using a flash drive. After that we tried using a wifi adapter to get internet but ended up getting internet through the ethernet cable. Once we had internet, we could begin coding up our new, more powerful strategies!

V.2 Training the Model

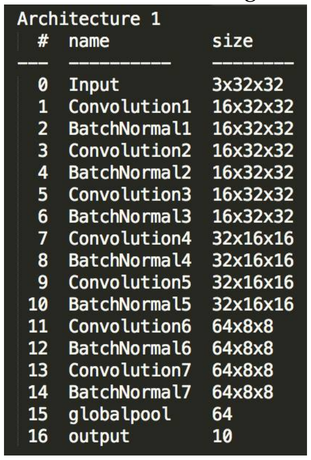
We needed to download a number of important libraries in order to get a convolutional neural network working: Python 2.7, Sci-Kit Learn 0.18.dev0, Theano 8.0, Lasagne 0.2.dev1, NoLearn 0.6a0.dev0, Numpy 1.11.0, and Pandas 0.18.0. Some of these version we were able to install using the "pip install" command on Linux, but others we had to google to find the bleeding edge versions which had the capabilities we needed, such as batch normalization for neural networks. Once we got NoLearn Lasagne working, we found that we had a speed up of around 25x over the neural networks we had run on my Macbook Pro, which now allowed us to do so much more with CNN's.

We began researching good architectures and hyperparameters for CNN's on the CIFAR-10 data set, and we found that some Microsoft Researchers, He, Zhang, Ren, and Sun, had come up with fantastic and easy to implement architectures and hyperparameters for this problem [4]. We followed their instructions for a CNN, in terms of architecture and hyperparameters pretty tightly, except that we didn't use any shortcut connections or data augmentation other than flipping images. We found that Microsoft's batch normalization combined with L2 regularization performed better than other regularization strategies such as droupout. We set their hyperparameter of n=1 for 5 of our models and n=2 for 5 of our other models, where n is a number representative of the number of layers in the network (there are 6*n+1 Convolutional layers in a network that they specify, and n can be any integer, usually between 1 and 18 inclusive).

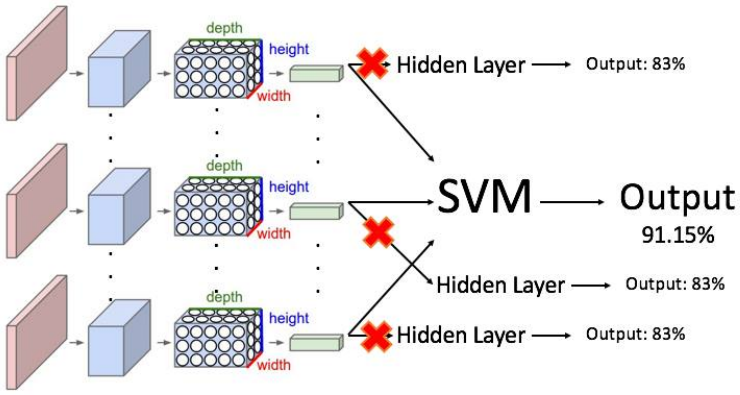
In other words, we subtracted the per-pixel mean from each image, then used 1+2n convolutional layers outputting 16 32x32 filters, 2n convolutional layers outputting 32 16x16 filters, 2n convolutional layers outputting 64 8x8 filters, a global average pooling layer, and a fully connected layer outputting softmax. We used RELU activation only (except for the output softmax), L2 regularization of 0.0001, momentum of 0.9, batch size of 128, learning rate of 0.1, He's Normal weight initialization, Batch Normalization, and no dropout. We did not have enough compute power or time to run the training of this model for as long as the paper specified, 64K iterations, but we were able to run several different models up to 1K-5K iterations, producing validation accuracies ranging from 81.9%-84.5% for each individual model.

V.3 Using CNN's as Feature Extractors

Seeking to improve the accuracy further and with a cheaper method, we used the outputs of the last hidden layer as features for other, simpler methods. We first needed to get the hidden layer outputs before we could learn from them, and for that we got a lot of help from Christian Perone and his Nolearn tutorial from which I took code on how to receive hidden layer output and how to visualize conv layer weights [2]. After getting the last hidden layer outputs, we tried running Decision Trees, Naive Bayes, and SVM on those outputs, in the hopes that those methods might still be able to yield good results or improvements over simply the last layer of a CNN, which is essentially a form of logistic regression. However, Using Naive Bayes yielded no improvement on the outcome, giving 80.7% on a validation set - slightly less than the fully connected final layer gave. This high accuracy, especially when compared to Naive Bayes's earlier 30% accuracy, is indicative of how telling the features extracted from the CNN are, and the lack of improvement using Naive Bayes shows that these features are still somewhat correlated - enough that Naive Bayes gives no advantage over a fully connected layer. A single Decision Tree gave 67.9% accuracy, and a voting scheme of 100 decision trees gave 68.8% accuracy. Again the fact that decision tree on these features is able to perform so well is extremely telling - 68%, although not as great as the CNN, is still around 9% better than what we were able to achieve with even a voting system of Neural Networks on augmented data. Finally, we tried the kernelized SVM on these extracted features, which brought us up to 89.6% (The structure is shown below, with n = 1 CNN structure).



```
Architecture 1
  #  name          size
---  ----------    --------
  0  Input         3x32x32
  1  Convolution1  16x32x32
  2  BatchNormal1  16x32x32
  3  Convolution2  16x32x32
  4  BatchNormal2  16x32x32
  5  Convolution3  16x32x32
  6  BatchNormal3  16x32x32
  7  Convolution4  32x16x16
  8  BatchNormal4  32x16x16
  9  Convolution5  32x16x16
 10  BatchNormal5  32x16x16
 11  Convolution6  64x8x8
 12  BatchNormal6  64x8x8
 13  Convolution7  64x8x8
 14  BatchNormal7  64x8x8
 15  globalpool    64
 16  output        10
```



For the kernelized SVM, we found that C=1.0 worked best, after a hyperparameter sweep. We also tried using a linear SVM and a fully connected Neural Network on the extracted features, but this did not perform as well as the SVM on our validation set. Moreover, very interestingly, we found that this SVM-method ensembling worked significantly better than other methods. Using the same 5 CNN's on the Kaggle test set, we achieved 86.9% accuracy using a voting method, 87.9% accuracy using an average softmax probability output method, an 88.2% accuracy using an average log softmax probability output method, and an 89.6% accuracy using out SVM. The voting, average softmax, and average log softmax are all widely used ensembling methods, and as you can see, our method outperforms each of these ensembling methods by a significant (1.4%+) margin, likely since a learning algorithm more refinedly weaves the CNN's together. The SVM has always been a powerful Machine Learning tool, and here it showed us that, in this instance, it was able to perform better than a final fully connected layer or other commonly-used used ensembling methods, giving our model a several percentage point bump. Also, our computing resources and time were limited, so we were only able to train 10 CNN's with an average validation accuracy of around 83%; however, if we had more time, we would've been able to train deeper networks, for longer periods of time, and with more data augmentation (instead of just flipping images), reaching accuracies of up to 93.6% for a single network, as the ResNet architecture authors achieved. Imagine if we had used the SVM method of ensembling on several of these networks; it's not hard to imagine that we would've achieved cutting edge accuracies with our method, especially given that it outperforms some of the most popular few ensembling methods.

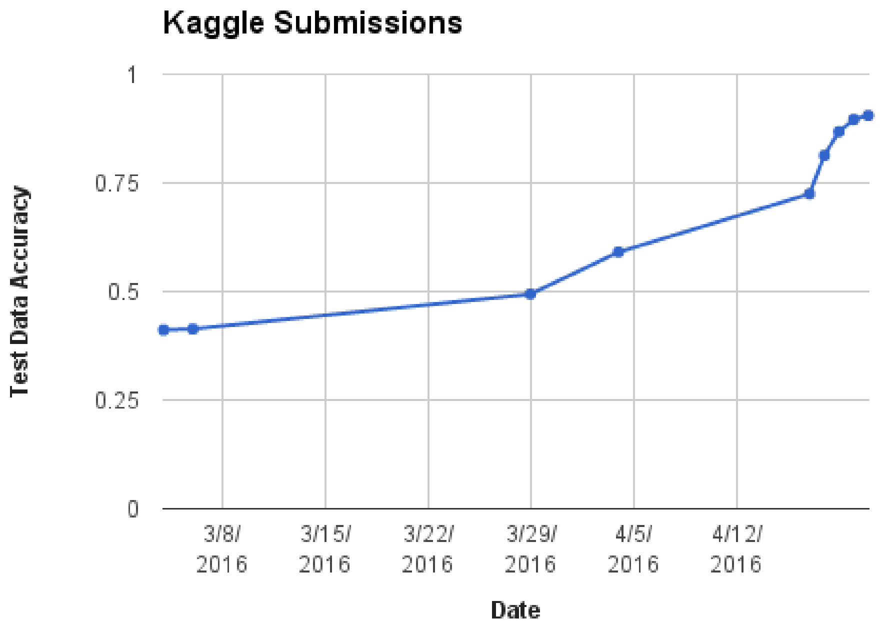
III. Final Results

I. Accuracy Table

Method	Test Dataset Accuracy
Gaussian Naive Bayes	28.33%
Binomial Naive Bayes	27.28%
Multinomial Naive Bayes	30.17%
Softmax Regression	41.15%
One-vs-All Logistic Regression	41.56%
Single Neural Network (NN)	52%
5 Voting NN's	55%
5 Voting NN's with HOG and Flipped Data	59%
Single Convolutional Neural Network (CNN)	83.0%
5 CNN's Ensembled via 1 Decision Trees	67.9%
5 CNN's Ensembled via 100 Decision Trees	68.8%
5 CNN's Ensembled via Gaussian Naive Bayes	80.7%
5 CNN's Voting	86.9%
5 CNN's Average Softmax Probability Output	87.9%
5 CNN's Average Log Softmax Probability Output	88.2%
5 CNN's Ensembled via Kernelized SVM	89.6%
10 CNN's Ensembled via Linear SVM	90.02%
10 CNN's Ensembled via Kernelized SVM	91.15%

II. Kaggle Submission Graph

The below graph plots our top submission per day over the past months. The earliest ones were One vs. All and Softmax submissions. The middle two were using Neural Networks, with and without voting systems and HOG. The last clump of submissions began with a single CNN, but the rest were different ensembles of CNN's, with the last of which was the SVM ensemble.



IV. Reflections

Our rather long-winded journey to the SVM over features extracted by the CNN has taught us a lot about the importance of a number of things ranging from compute power, to low level design, to working with dependencies, to taking advantage of all of the facets of the data you are given. We also learned how important it was to both lean on the research of others while simultaneously experimenting with new techniques. We realized the powers of the CNN in terms of its effective consumption of higher dimensional data, but also its drawbacks in terms of time, expense, and compute cost. We saw firsthand the weakness of Naive Bayes when given a problem with features that have strong dependencies on each other. We learned about the massive importance of time and processing power in the quest for the best results, feeling firsthand the sting of a limited supply of them both. All in all, we got a grand tour of many Machine Learning methods and on how they all perform on the same problem, and hopefully this information will be as valuable to you to learn as it was for us to discover!

References

[1] "Using Convolutional Neural Nets to Detect Facial Keypoints Tutorial." - Daniel Nouri's Blog. Web. 1 Apr. 2016. http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/

[2] Perone, Christian. "Terra Incognita." Terra Incognita. Web. 1 Apr. 2016. http://blog.christianperone.com/2015/08/convolutional-neural-networks-and-feature-extraction-with-python/

[3] Bergstra, James, and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization." Journal of Machine Learning Research 13 (2012). Web. 1 Apr. 2016. <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. Deep Residual Learning for Image Recognition, 2015; arXiv:1512.03385.

[5] Pedregosa, Fabian, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, and Bertrand Thirion. "Scikit-learn: Machine Learning in Python." Journal of Machine Learning Research 12 (2011): 2825-830. Web. <http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf>

[6] Python, Numpy, Theano, Lasagne, Nolearn, Scipy, Matplotlib, and Pandas