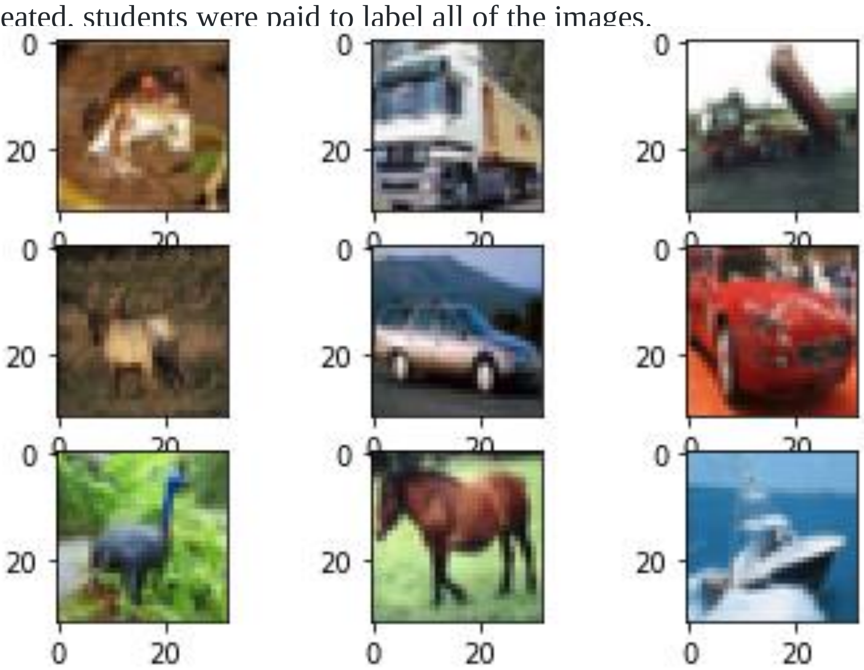
Project Overview:

Deep learning belongs to neural model paradigm in three paradigms of machine learning. It is the latest version of –[1] connectionism that has been going on since the 1950s. In connectionism, problems that were previously considered obstacles have been solved one by one, and it has recently become the most destructive technology field in artificial intelligence. [2]. In 1979, Kunihiko Fukushima first published Noncognition model applying neurophysiological theory to the artificial neural network. This model, inspired by Torsten Wiesel's award-winning Nobel Prize in neurophysiology, was the beginning of Convolutional Neural Network (CNN). CNN, which imitates human visual cognition processes, is inherently used in a unique way in the field of computer vision. Convolution technology, which shows excellent performance in extracting desired features from various types of data, is used in various fields such as image processing and speech recognition. The reason for using the convolution technique in image processing and signal analysis is to separate and extract features contained in signals such as original image or sound wave. So, I need to access via CNN to identify whether the monitored object is same as the predicted object. A brief description of the data set is provided below:

- The CIFAR-10 dataset (Canadian Institute for Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms.
- It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes.
- The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.
- Computer algorithms for recognizing objects in photos often learn by example. CIFAR-10 is a set of images that can be used to teach a computer how to recognize objects.
- Since the images in CIFAR-10 are low-resolution (32x32), this dataset can allow researchers to quickly try different algorithms to see what works. Various kinds of convolutional neural networks tend to be the best at recognizing the images in CIFAR-10.

- CIFAR-10 is a labeled subset of the 80 million tiny images dataset. When the dataset was created, students were paid to label all of the images.



Problem Statement:

I am going to create a simple program that will read any given image and try to label the content of that image with one of the 10 labels defined by the CIFAR-10 dataset. The CIFAR-10 dataset. In machine learning terms, this is a multi class classification problem.

Metrics:

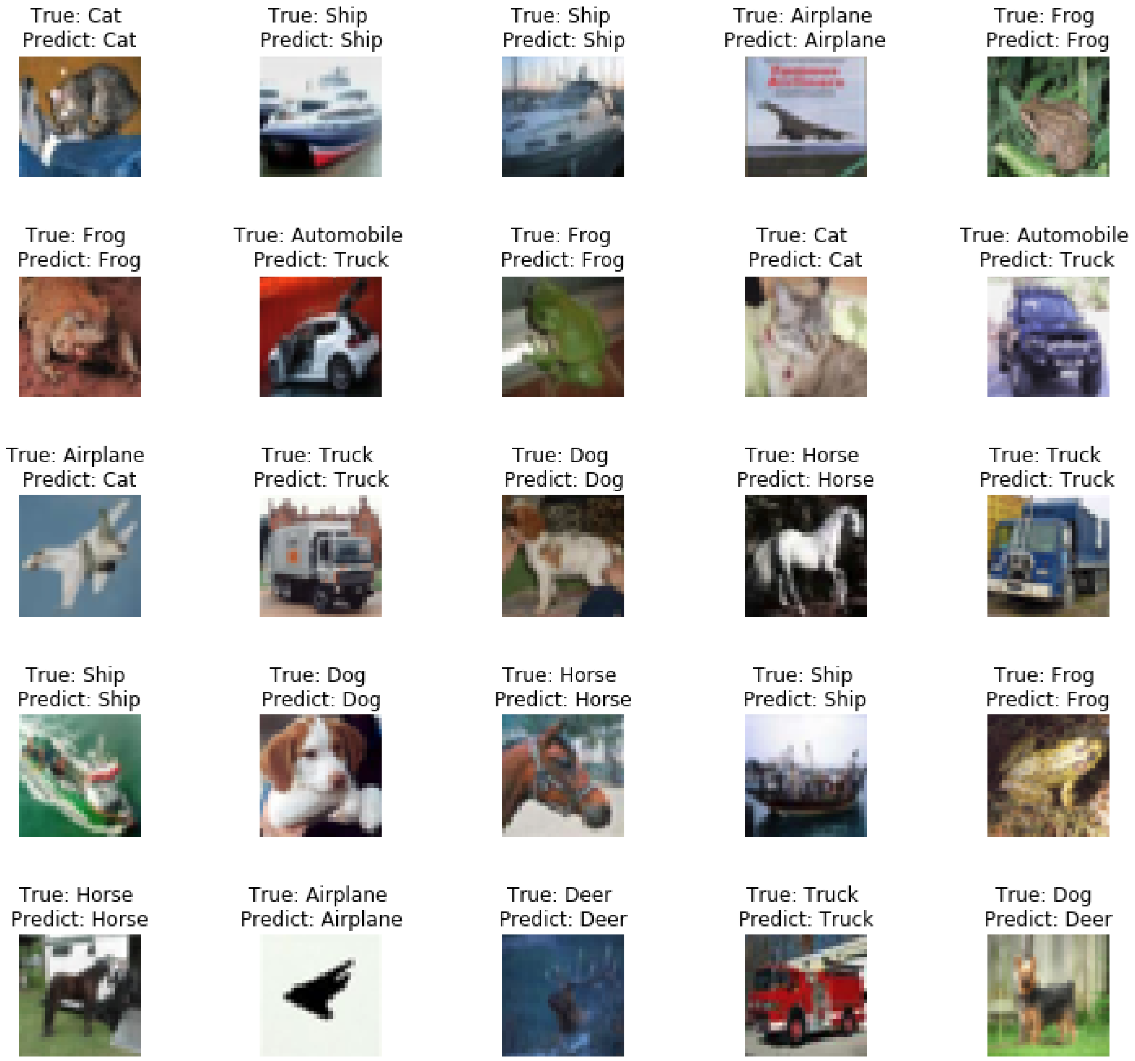
Since we are dealing with a perfectly balanced dataset (CIFAR-100 has 600 images labeled on each of its 100 lower level classes), I am going to propose using accuracy as the evaluation metric. My goal is to create a classifier that will have at least 80% accuracy on the training set, and at least 50% accuracy on sample images that have various dimensions and sizes. All the while, the CNN classifier will have to perform better than a simple SVM classifier.

Data Exploration:

This project uses the CIFAR 10 data set, which is composed of 60.000 32x32 pixels images classified as 10 different classes:
• Airplane
• Automobile
• Bird
• Cat
• Deer
• Dog
• Frog
• Horse

- Ship
- Truck

A few sample images of this dataset are shown below –



Features and Labels:

The features we’re going to use are represented by each individual pixel in the images. Since we’re dealing with images with 32 pixels in height and 32 pixels in width, we have a total of 1024 pixels. The CNN classifier works with colored image, so there is a third dimension that has to be added to each pixel. This third dimension is an array with 3 elements, each one representing the Red, Blue and Green channel of the pixel. The RGB channel elements are integer numbers from 0 to 255. So, in the end, the CNN classifier will work with 3072 features. The SVM classifier reduces the number of features by eliminating the color channel dimension. Before feeding the dataset into the classifier, our python code will convert each image to grayscale, as we’ll see in a next step of this report.

Each image in the dataset is associated with a label, which is an integer ranging from 0 to 9. This label is actually an index of the class description in a list that contains the 10 classes ordered alphabetically.

The dataset is perfectly distributed among the 10 labels. There is exactly 6.000 images on each of the labels, resulting on the total of 60.000 images

Algorithms and Techniques

The main goal of this project is to build a Convolutional Neural Network (CNN) classifier to predict image labels. We’ll be using the Keras library with a Tensor Flow backend to achieve our goal.

CNN classifiers are one of the best machine learning algorithms when it comes to image recognition. CNN works really well when the order of the features is important, and this is exactly the case with images. The numeric representation of the image pixels are our features, and the order of pixels is very significant. That’s why we’ve decided to use CNN. The decision to use Keras and Tensorflow followed suite, since this combination offers a great balance between simplicity and efficiency. CNN works by applying a series of steps (or layers) on the data in a sequential manner. Each layer is composed by neurons that will filter or reduce the data. Convolution is a key layer of CNN. During the Convolution step, the algorithm will ’scan’ the full image, comparing every little piece of the image with a set of known features. Each bit of data, or in our case each pixel, will be compared with the pixels from the features, generating a ’stack’ for every feature. We can describe a stack as an array of weighted sums, where each item of the array is a pixel from the original image: In the picture above, the large square on the left side is the original image, representing the letter ’X’. The small square on the upper left corner is a feature, a diagonal line that represents a piece of the ’X’, and the greenish square on the right is the stack generated when this features is applied to the original image. This process runs many times, one for each of known feature. After the Convolution layer is applied, we can execute Pooling. This layer will reduce the image a little bit, by creating smaller, aggregated sections of the stacks. Pooling will walk through the zones of the image and pick the highest scored pixel on that zone. This process helps the detection process to assimilate variances on the selected feature, like small differences in positioning and tilting: The image above shows the original stack on the left, and the generated dataset after the Pooling step has done its job. As we can see, the image’s dimension has reduced from 7x7 to 4x4. We can use a special layer, provided by Keras, to reduce the risk of overfitting. The Dropout layer will randonly set a fraction of the inputs to 0. This will force the algorithm to apply the learning process again for those inputs. There are other types of layers that can be applied to the network, and the same layer type can be at different parts of the network design. The choice of which layers to use, and their order, how many neurons to add to each layer are part of the art of building a CNN. More skilled engineers can even build custom layers, tailored specifically to the problem they’re dealing. The output of each layer becomes the input for the next. At first, each neuron from a given layer is connect to all the neurons from the next layer. This creates a complex network of connections, where each connecting line has a specific weight that controls the significance of that particular output to the next layer. One of the main jobs of the CNN algorithm, implemented by Keras, is to define those weights. This is how a network would look like at first, before the training process starts: The training process applies a somewhat simple technique to find the optimal weights for each neuron, called "Backpropagation". The algorithm will work with already labeled images (training set) and calculate the error rates for each prediction. Based on the error rate, new weights that reduce that rate will be define. After the training is complete, some neurons might have been given weights equals to 0. The output connection for those neurons are ignored, resulting on a simpler network: We’ll also train a very simple Support Vector Machine (SVM) classifier that wil try to perform the same task - predict image labels - with the sole purpose of serving as a benchmark comparison to our primary CNN classifier.

Benchmark Model

Our benchmark will be to provide better results than the Support Vector Machine (SVM) algorithm. The reason for this comparison is to compare the results given by a wide-scoped machine learning algorithm, like SVM, with the results generated by an algorithm that is more oriented to image recognition. We expect the CNN algorithm to have a much higher accuracy than the general algorithm. If this assumption is confirmed we can come to the conclusion that, when it comes to image recognition problems, we should give preference to algorithm that are more specialized in that domain.

Evaluation Metrics

Since we are dealing with a perfectly balanced dataset (CIFAR-100 has 600 images labeled on each of its 100 lower level classes), we're going to use accuracy as the evaluation metric.

Methodology

- CNN Classifier
    - **Data preprocessing**

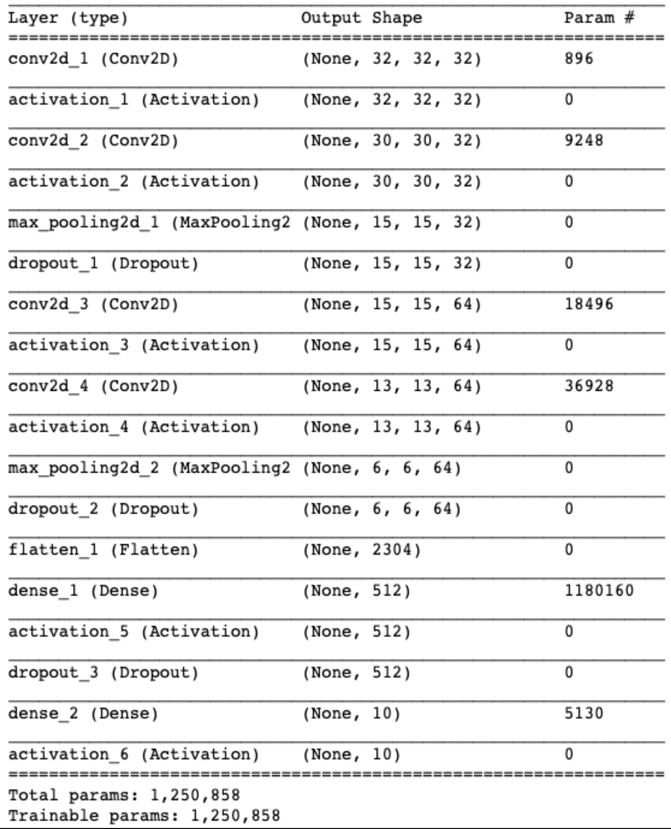
Before feeding the dataset images into our Sequential Model, there is a minor preparation. We normalized the data, dividing the RGB values by the maximum value, 255. This results on decimal numbers from 0 to 1. We'll also need to perform some resizing and scaling during the prediction phase. That preparation is needed, since the training set was composed by 32x32 images, and the sample images can have various dimensions. 3.1.2 Implementation We're using Keras' Sequential Model. This model allow us to use many processing layers that run sequentially and improve the accuracy of the classifier on each iteration, or epoch. In our case we've implemented a pattern using the following steps: Convolution2D, Dropout, Convolution2D and MaxPooling2D. Then we repeated this pattern 3 times, increasing the number of feature maps from 32 to 64 and, finally, to 128

- **Training time**

At this point I faced the first, and toughest, challenge of the assignment. The training simply takes too long to complete. I've made some test runs using lower values for 'epoch', and I could only manage to complete the training locally when defining value 1. This resulted on a much lower accuracy than I was expecting (around 40%). So I decided to run the algorithm on a more powerful machine and resorted to Amazon Web Service for that matter. I've rented a c3.large EC2 instance and ran the code with epoch equals 25. It took me quite some time to configure the machine with the Keras and Tensorflow environment. But in the end I've managed to build Tensorflow from source on the EC2 instance, and ran the code successfully. After 5 hours or so the execution was completed, so I persisted the trained classifier to disk using the save() function from Keras. This process created a 20MB file with the .h5 extension that I uploaded to a public S3 bucket. From my local machine I could then download the persisted classifier and continue the experimentations. That's why we won't see a complete output for the cell above in this iPython notebook. The result, however, was a classifier with around 80% accuracy, which is still not up to production standards, but suitable for this project.

Prediction:

The first step in the prediction process is to load the persisted classifier. After loading, we displayed the model summary just to assure that everything worked as expected:



```
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896       
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248      
_________________________________________________________________
activation_2 (Activation)    (None, 30, 30, 32)        0         
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 32)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496     
_________________________________________________________________
activation_3 (Activation)    (None, 15, 15, 64)        0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928     
_________________________________________________________________
activation_4 (Activation)    (None, 13, 13, 64)        0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 64)          0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1180160   
_________________________________________________________________
activation_5 (Activation)    (None, 512)               0         
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130      
_________________________________________________________________
activation_6 (Activation)    (None, 10)                0         
=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
```

Sample Image Preparation

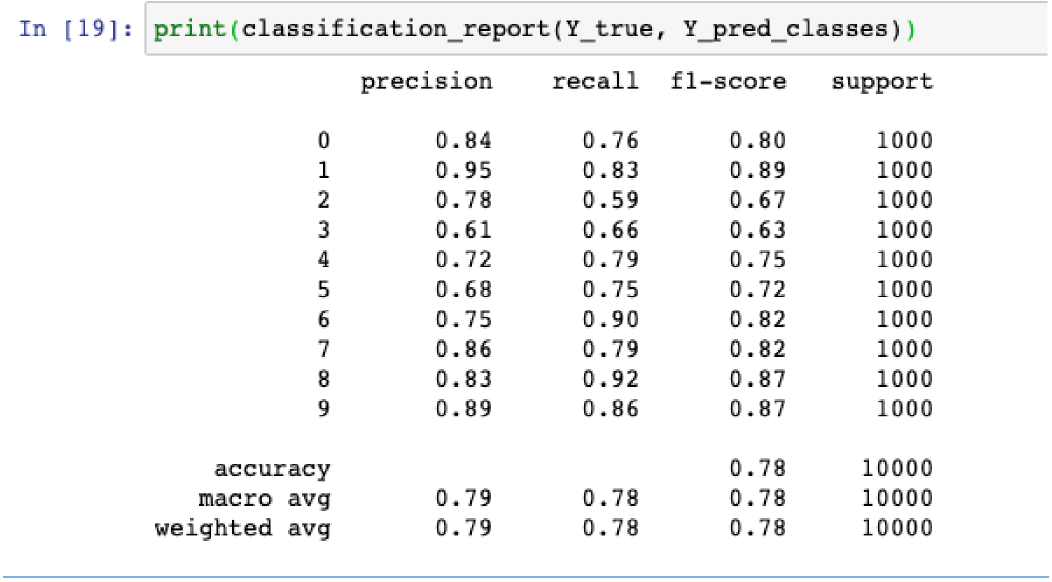
We cannot expect that all images that will be sent to our classifier will have the exact same dimensions that our training images, which is 32x32 pixels. So we need to perform some preparation before sending the sample to the predict() function: • Convert the image to a square. We achieve this by cropping the image right in the center. If we don't perform this crop we end up with a deformed, sometimes stretched, version of the original image. The prediction will actually work with deformed images, but I found out that using cropped images resulted on better accuracies, at least with the sample images we 'used

• Resize the squared image to a 32x32 tiny square

• Swap RGB channels. This was more of a trial-and-error discovery. The accuracy when running the prediction on the samples was terribly lower than the 80% we got on our test dataset, so I've tried different orders on the channels, and this is the one that worked the best

Finally, our predictCNN() function. This function will prepare the image, normalize the data, and run the model.predict() function from Keras. The result from the model.predict() function is a sized 10 array containing weights for each of the 10 possible classes. To find out which class is the chosen one we simply grab the index that contains the highest weight. We can also calculate a precision value, by dividing the weight of the chosen class by the sum of all weights:

Our classifier guessed correctly on 18 out of 30 sample images, achieving a 60,00% accuracy. This is significantly lower than the 80% accuracy we got on the test set. In many cases the classifier guessed the wrong label, it displayed a low precision value (sometimes .35 or .60). We could establish a lower bound and refrain from guessing if the precision was too low. For example, we could show a message like ’Sorry, the classifier cannot guess the label for this image’, if the precision was lower than .90.



```
In [19]: print(classification_report(Y_true, Y_pred_classes))

              precision    recall  f1-score   support

           0       0.84      0.76      0.80      1000
           1       0.95      0.83      0.89      1000
           2       0.78      0.59      0.67      1000
           3       0.61      0.66      0.63      1000
           4       0.72      0.79      0.75      1000
           5       0.68      0.75      0.72      1000
           6       0.75      0.90      0.82      1000
           7       0.86      0.79      0.82      1000
           8       0.83      0.92      0.87      1000
           9       0.89      0.86      0.87      1000

    accuracy                           0.78     10000
   macro avg       0.79      0.78      0.78     10000
weighted avg       0.79      0.78      0.78     10000
```

SVM Classifier

The SVM classifier we’ve used on this project is purposely simple. Our focus was to build a strong CNN classifier and use SVM simply as a comparison tool.

- **Data preprocessing**
  In order to make the training faster, the images were converted to greyscale. That reduced the number of input feaures 3-fold. Another preparation step was needed. Since the SVC.fit() function expects a 1-dimensional array, we had to flatten our 2x2 array before feeding it to the classifier

- **Implementation: The** SVM implementation is very straighforward. We feed the classifier with our preprocessed images and call the fit() method
- **Training time:** Training the SVM classifier also took a long time. Another EC2 c3.large instance was used for 3-4 hours before the whole process was completed. We could persist the trained model to disk using sklearn’s dump() function. That generated a 400mb .pkl file that we can later load and avoid the training phase.
- Well, the results are not good at all. It looks like the SVM classifier labels every image with the same class: ’Truck’. This results on a 3 out of 30 correct guesses, or a 10% accuracy - after all, a broken clock shows the right time twice a day. Which is much lower than the 60% accuracy we had with CNN. Also, with SVM we cannot derive a precision value, so we cannot say how ’confident’ the classifier is with its prediction

Conclusion

The result of this project was not a surprise for me. Despite never having used Keras or Tensorflow before, I was expecting it to be much better than SVM, and any other Scikit-Learn algorithm for that matter. The reason for that is that all my previous researches showed that the Keras+Tensorflow combination was excellent for image classification. Regarding SVM, we could obviously refine our classifier, perform feature selection, cross validation and many other enhancement in order to achieve better results. However, I doubt that we could get closer to the accuracy that CNN provide

Reflections:

This was a very challenging project, because the scope of it was really wide and required a lot of research, study and practice. I have used a lot of concepts acquired during the course, but also had to learn new technologies and techniques. I particularly liked working with Convolutional Neural Networks. The concept of CNN is not a easy one to understand, however, Keras and Tensorflow help us a lot by creating a nice and easy to use API that shades the major complexities of CNN while still maintaining a high level of accuracy. One of the most challenging parts of the project was to run the actual training algorithm. Since we're dealing with a somewhat large dataset (at least many times larger than any other dataset we've used throughout the course), the time my personal computer was taking to progress with the training was just too much to be practical. So I had to resort to a cloud platform to reduce the processing time. I believe this serves as an important lesson. Real-world problems that need machine learning systems to be solved often will have really large amounts of data to be handled. It is likely that a cloud platform will have to be used, since it is much cheaper than buying new hardware. My estimates are that, to run the Amazon EC2 instances that had to be used for this project, I've spent only 5 US Dollars.

Possible Improvements:

A technically simple but operationally difficult improvement would be to increase the number of epochs on the classifier. We would need to change just a line a code to update this value from 25 to 1.000, but this would translate to many extra hours of CPU time. Once complete, though, we could certainly hope for a much better accuracy than the 80% we got initially.

References:

1. *"AI Progress Measurement". Electronic Frontier Foundation. 2017-06-12. Retrieved 2017-12-11.*
2. *"Popular Datasets Over Time | Kaggle". www.kaggle.com. Retrieved 2017-12-11.*
3. *Hope, Tom; Resheff, Yehezkel S.; Lieder, Itay (2017-08-09). Learning TensorFlow: A Guide to Building Deep Learning Systems. "O'Reilly Media, Inc.". pp. 64–. ISBN 9781491978481. Retrieved 22 January 2018.*
4. *Angelov, Plamen; Gegov, Alexander; Jayne, Chrisina; Shen, Qiang (2016-09-06). Advances in Computational Intelligence Systems: Contributions Presented at the 16th UK Workshop on Computational Intelligence, September 7–9, 2016, Lancaster, UK. Springer International Publishing. pp. 441–. ISBN 9783319465623. Retrieved 22 January 2018.*
5. *Krizhevsky, Alex (2009). "Learning Multiple Layers of Features from Tiny Images" (PDF).*
6. *"Convolutional Deep Belief Networks on CIFAR-10" (PDF).*