Universität Hamburg
Department Informatik
Knowledge Technology, WTM

Comparison of Gradient Descent Optimization Methods for Neural Networks

Seminar Paper

Neural Networks

Ali Saleh

Matr.Nr. 6517831

3saleh@informatik.uni-hamburg.de

13.07.2017

Abstract

Gradient Descent is the most widely used optimization method for neural networks training. This paper aims to explore different algorithms for gradient optimization. Using standard datasets and different neural network architectures, time and complexity of the different algorithms will be compared and analyzed.

Contents

1 Introduction

In the scene of machine learning there are several key components to build a classification/regression model. Your model building usually involves several decision to make. After choosing your model (e.g.KNN, SVM, KART, NN, etc ...) you will begin to address the problem of how to choose your internal parameters (like the K in k nearest neighbor). For the context of neural networks (assuming you have already chosen your model architecture) you will have to decide on how many layers, how many nodes per layer, what kind of encoding yo use, and several other parameters depending on your architecture. [Schmidhuber, 2015] and [Arel et al., 2010] have both presented the general outline for neural networks modeling in more details.

Once your model parameters are decided on you will need to begin the model training and evaluation. This is the process of deciding if your model is performing well, and can be used to compare several models together. In this setup evaluation is done by the mean of a loss function. It is used to asses the performance of you model by comparing predicted values against the ground truth of test data. Loss function is defined as :

$$L(X, Y, \grave{Y}) \tag{1}$$

Where X is your input, Y is the ground truth, and $\grave{Y}$ is the prediction of your model on input.

In neural networks the output of your model depends highly on the weights of the connections between the network layers. This means that the notion of loss function is thus used to evaluate the goodness of a set of weights W used by a model. Thus the need for optimization, to find the set of weights that minimizes the loss function.

One of the most prominent strategies to optimize functions arises from calculus. From its name loss function is an ordinary mathematical function in the sense that it can have a derivative. Using the mathematical analogy, optimization can be achieved by moving in the direction of the gradient.

In a one variable function the first derivative is the rate of change of the function, which can be generalized into multi-variable functions. The gradient is the rate of change of a multi-variable function in a specified set of directions. It is represented as a vector of numbers, with each element of it as a rate of change in a direction. For a 3 variable function

$$w = f(x, y, z) \tag{2}$$

it is gradient in the 3 variables directions is :

$$\nabla w = \left\langle \frac{\partial w}{\partial x}, \frac{\partial w}{\partial y}, \frac{\partial w}{\partial z} \right\rangle \tag{3}$$

where $\frac{\partial w}{\partial x}$ is the partial derivative of the function w in the direction of x. Moving the weights in the direction of the gradient leads to minimizing the loss function.

The reminder of this paper are organized as following. Section 2 will provide an overview of the gradient optimization methods, how to calculate them and what algorithms to compare. Section 3 will have a description of an experiment to compare the 3 algorithms together, section 4 will contain the results, section 5 will be a discussion, and Section 6 will conclude the paper.

2 Gradient Computing And Optimization

Computing the gradient for a function with hundreds of variables as in neural network is a computationally intensive function. Along the years tens of algorithms and approximations have been introduced to make it easier to compute the gradient of a function. In principle gradient can be computed in one of two ways. Either approximation using finite difference, or analytically using calculus.

In calculus the derivative of a single variable function is defined as :

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{4}$$

Using finite difference to calculate the gradient you first specify step size (usually a tiny number to approximate h approaching 0). You then proceed to loop over all the variables in your function (here the weights) and you calculate the derivative for each one of them to obtain the gradient. Using the value obtained from this computation, the weights can be updated accordingly. The code snippet below describe a function to perform gradient using finite difference.

```
#Code is provided as part of Stanford CS231 https://cs231n.
    github.io/optimization-1/

def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite
      '])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
```

```
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very
        important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

The main problem with this way of computing the gradient is that it is computationally expensive. The loop over all the dimensions takes too much for a single step update. In a typical deep learning neural network the number of dimensions would be in the order of millions. Looping over millions of elements for a single update would result in weeks of network training. This is practically infeasible and thus comes the need to find another way for calculating the gradient of a function.

A better way to compute the gradient for large number of dimensions is using closed formulas. To attain a closed formula you begin with your designated loss function at a single data point. You then differentiate it with respect to the weights to get your gradient formula [Qian, 1999] [1]. It Is common to use sanity checks to compare the formula implementation results with the gradient computed using finite difference.

2.1 Stochastic Gradient Descent

Using the formula of gradient descent updating the parameters is a handy process. Use training set and weights evaluate your gradient and then update the weights according to the chosen learning rate. A sample python code to do this is below:

```
#Source http://ruder.io/optimizing-gradient-descent/index.html#
    gradientdescentvariants

for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

this method is called batch gradient descent. All of the training data are used at once for training and evaluation of the gradient. It is guaranteed that batch gradient will always converge to the global minimum in convex optimization surface and to local minimum in non convex surfaces [Ruder, 2016].

A more efficient way to compute the gradient is using smaller batches of the training data to evaluate the gradient. It is much faster and the shuffling of data used in this method enables to reaches new local minimums and not getting stuck with the first local minimum [Ruder, 2016]. Nevertheless slowly decreasing the learning rate would make the convergence of SGD similar to the batch gradient descent [Ruder, 2016]. A python snippet showing how to use this method is shown below:

[1] More information can be found here https://cs231n.github.io/optimization-1/

```
#Source http://ruder.io/optimizing-gradient-descent/index.html#
    gradientdescentvariants

for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

2.2 Adagrad

Adagrad is an optimization method that is using adaptive learning rates to compute the gradient [John Duchi, 2011]. Unlike SGD that has one learning rate for all the parameters, Adagrad has different learning rates that evolve over time with the parameters. For sparse datasets some parameters are more frequent than the others. Adagrad will have different learning rates for them the more frequent parameters will have a learning rate that decays faster and is smaller. The less frequent parameters on the other hand will have slower decaying learning rates that are larger.

Adagrad performs this by keeping track of a matrix all the past gradients for each parameter.While updating a parameter it divide the learning rate over the square roots of the sum all the gradients computed before for this parameter. Thus the more frequent a parameter the less the learning rate. One weakness of Adagrad is that over time the square roots of the sum becomes large that the learning rate become so small and no learning is happening after that

2.3 Adam

Adam (Adaptive Moment Estimation) is an adaptive learning optimizer like Adagrad [Diederik P. Kingma, 2015]. Instead of keeping track of all the previous gradients to compute the squared sum as in Adagrad, Adam stores the average of all the past squares of the gradient along with another average of the past gradients themselves. The two averages works as an estimate of the first two moment (mean and variance). That is why the method is called this way.

In the original paper the two averages are initialized to zero which lead to them being biased toward zero, this led the authors to use a bias correction to resolve this problem. The two bias corrected averages are then used to update the parameters one by one. The update divide the learning rate by the bias-corrected average of the past squared gradients, and then multiply it by the bias-corrected average of the past gradients. This lead to a less aggressive decaying of the learning rate for frequent parameters.

3 Experiment Design

For comparing the algorithms mentioned in section 2, an experiment was held to analyze their performance.For the experiment the CIFAR10 dataset was used (see Figure 1. "The CIFAR-10 dataset consists of 60000 32x32 colour images in 10

classes, with 6000 images per class. There are 50000 training images and 10000 test images." [Krizhevsky, 2009].

The model architecture used is convolution neural network. The model consists of the following layers. The first layer is a convolution layer with 32 filter each on 3 x 3 sub-regions with ReLU activation function. Then there are a max-pooling layer with 2 x 2 filters and a stride of 2 (to specify the factor by which to downscale), and dropout regularization rate of 0.25 [Srivastava et al., 2014]. The next layer is a convolution layer with 64 filter each on 3 x 3 sub-region again with relu activation layer. Followed by another max-pooling layer with the same specifications as before. A flattening layer is then added which takes the 2D output of the previous layer and turn it into 1D (e.g. a 32 x 32 matrix would be 1024 array). After that comes a fully connected (dense) layer with ReLU activation and dropout regularization rate of .5. The final layer is another dense layer with size set to the number of classes (here 10) with softmax activation function. This layer with softmax outputs a probability of an image belonging to each class.

The model is then used with the 3 chosen optimizers. Each run consists of loading the cifar10 data set into two parts. The training set which consists of 50000 $32x32$ images. the testing set which consists of 10000 $32x32$ images. The training is then carried with batches of size 32. The training is done over 200 epochs. Categorical cross-entropy is used as loss function [de Boer et al., 2005]

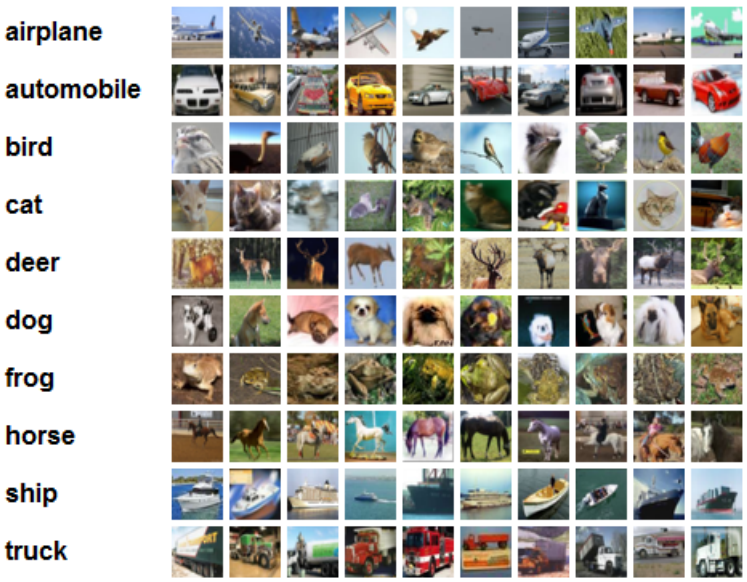


Figure 1: overview of the CIFAR 10 dataset (courtesy of Alex Krizhevsky)

The 5 metrics that were used for the comparison are:

- The total time an epoch takes to finish.
- The final loss on the test set.
- The accuracy of classification on the test set.

- The final loss on the validation set.
- The accuracy of classification on validation set.

The network parameters were initially set to the defaults provided by keras. A trial for optimization using hyperas (Keras + Hyperopt python library) was undergoing. Due to the sheer amount of parameters it is data was not considered for the analysis and the results with default parameters were used instead. Nevertheless the code for the optimization is provided in appendix B.

4 Results

The execution time was measure by keras library for each epoch. As seen in Figure 2 the time for each epoch is not decaying over time. While the Adagrad has almost uniform time for each epoch, the SGD time fluctuates a lot, and the ADAM time is a reverse of SGD . At the beginning the SGD time was higher the the other two. Over time SGD began to stabilize with a spike around every 50 epoch. In the same time ADAM that began with smooth uniform time per epoch suddenly began to fluctuate randomly.

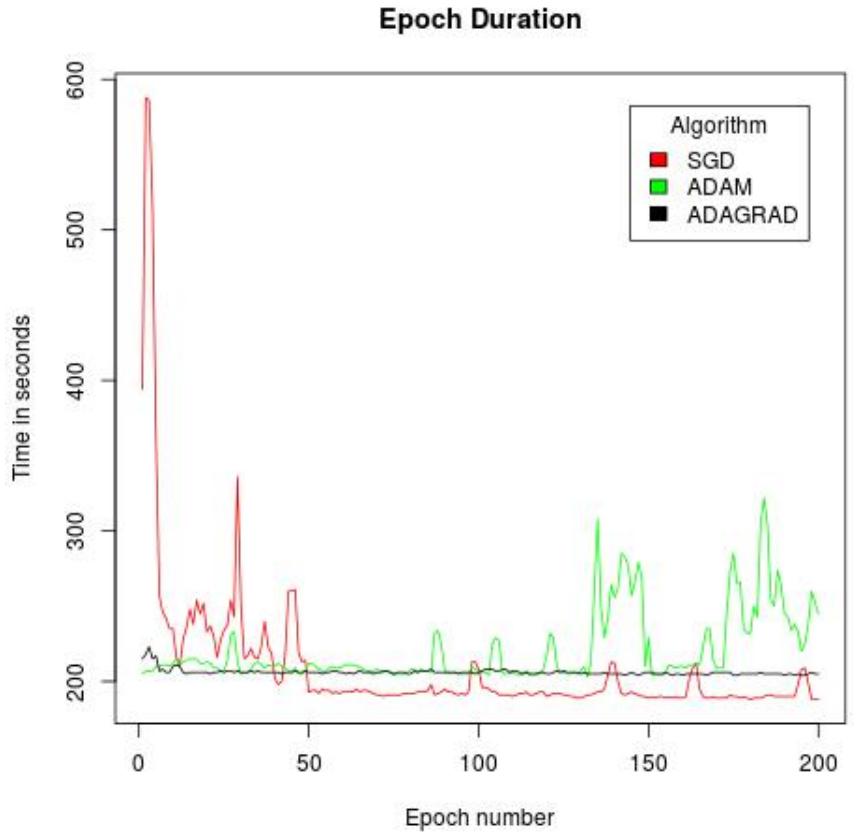
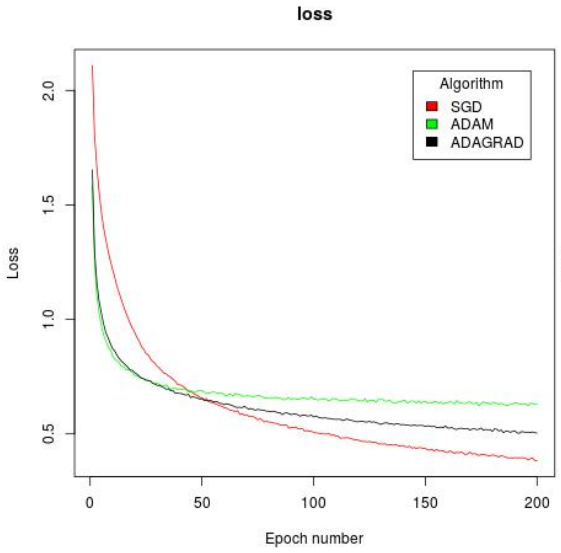
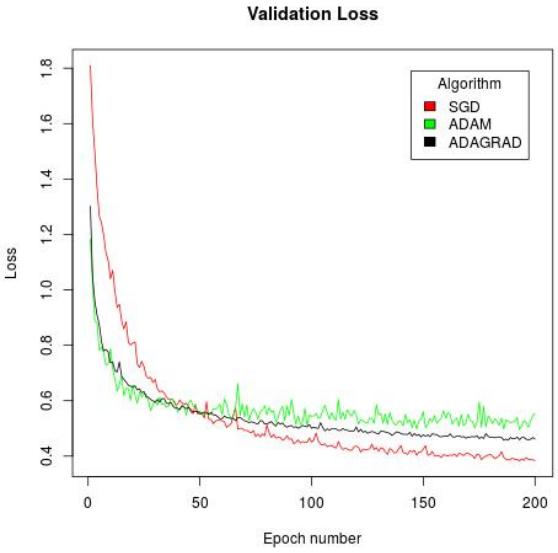


Figure 2: Total execution time per epoch over time for each optimizer

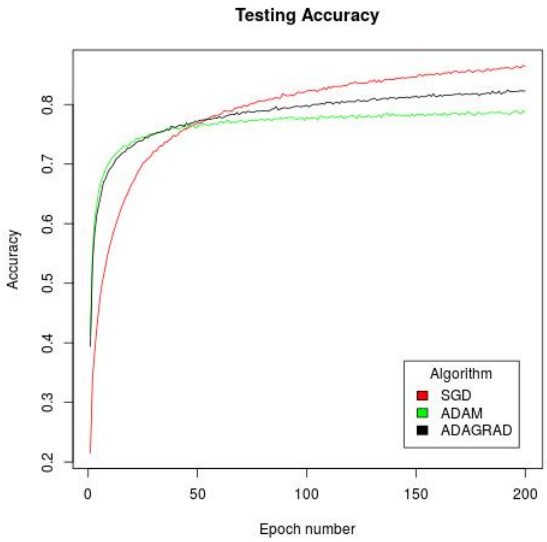
The loss of each of the 3 optimizers over epochs is shown in figures 3a and 3b for test and validation respectively. The two figures looks similar in shape. In both of them SGD begins with the highest loss and proceeds to reach the lowest loss of all of them. Both Adagrad and Adam begins at comparable loss values, but Adam's loss curve is less steep over time. In the end Adagrad reaches a lower loss value than Adam for both testing and validation. As expected the validation loss values are more fluctuating than the testing loss values, although Adam has considerably higher fluctuation than the rest of the optimizers



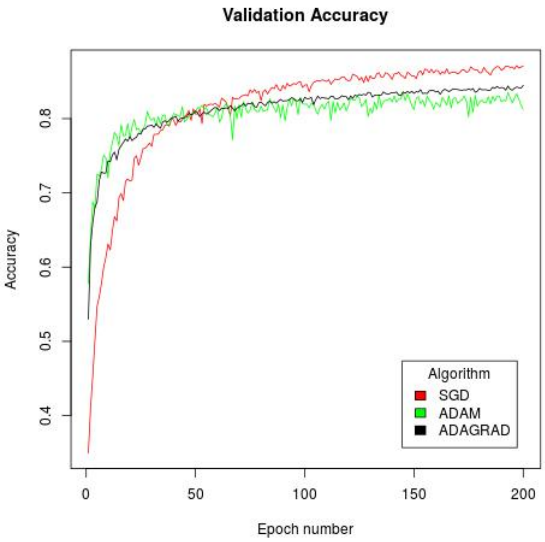
(a) Loss per epoch on test set



(b) Loss per epoch for on validation set



(a) Accuracy on test set



(b) Accuracy on validation set

The same can be seen in the results of the accuracy. The accuracy of each of the 3 optimizers over epochs is shown in figures 4a and 4b for test and validation respectively. Again SGD has the highest accuracy although it began with the lowest both on testing and validation. The same is still true for Adam and Adagrad, both begin similarly and end with Adgrad has higher accuracy than Adam. The validation accuracy fluctuation is still more prominent on Adam than the other two.

5 Discussion

The results show in section 4 along with the statistics show in table 1, shows that on this dataset SGD outperform better than the other two optimizers. The design of the Adam and Adagrad is concerned much about the sparsity and infrequent parameters. Much of the tweaking done in the two optimizers is targeting this point. With a balanced dense dataset like the CIFAR10 it's not a surprise that those two aren't performing well compared to the SGD. This is matching the state-of-art results on the CIFAR10 [Graham, 2014b] [Graham, 2014a] [Springenberg et al., 2014] [Mishkin and Matas, 2015]. Although some of them are also using momentum ([Qian, 1999]) component with SGD.

Table 1: Experiment Statistics

	SGD			Adam			Adagrad		
Metric	**min**	**mean**	**max**	**min**	**mean**	**max**	**min**	**mean**	**max**
Time(seconds)	188.0	209.2	588.0	203.0	220.6	322.0	205.0	206.1	223.0
Loss	0.3822	0.6041	2.1094	0.6224	0.6836	1.5892	0.5027	0.6187	1.6527
Accuracy	0.2151	0.7865	0.8654	0.4132	0.7655	0.7899	0.3942	0.7827	0.8242
Validation Loss	0.3828	0.5363	1.8103	0.4953	0.5708	1.1835	0.4577	0.5391	1.3018
Validation Loss	0.3496	0.8150	0.8715	0.5782	0.8075	0.8353	0.5298	0.8131	0.8449

The performance of Adam on validation sets is notable with it's high fluctuations. This can be attributed to the adaptive learning rates used in Adam. With a dense dataset, the work Adam do to maintain adaptive learning rate for each of the parameters doesn't seem to contribute much to its performance. It was always keeping track of the much information on the gradient of previous runs, this could be the reason it was fluctuating in performance over different unseen validation sets.

All of that said, it's worth mentioning that SGD is not an all around solution. SGD takes time and in total it took more time to finish than the other two. Also it took around 50 epochs to stabilize in time and performance. It also took around 50 epochs to surpass the other two. With the small CIFAR10 it was plausible to

train an SGD model in around 13 hours in a personal machine. The other two algorithms took around 1 hour less with the same hardware and configurations.

6 Conclusion

Choosing an optimization method for a model is not working according to a check list. There are hundreds of factors to consider. How to choose the parameters for the method is also a factor. If your data is sparse then an adaptive learning method (Adam/Adagrad) would have an advantage. You can always begin with a simple SGD optimizer and build upon it but adding momentum then maybe converting to a more advanced adaptive learning method if needed.

References

[Arel et al., 2010] Arel, I., Rose, D. C., and Karnowski, T. P. (2010). Deep machine learning - a new frontier in artificial intelligence research [research frontier]. *IEEE Computational Intelligence Magazine*, 5(4):13–18.

[de Boer et al., 2005] de Boer, P.-T., Kroese, D., Mannor, S., and Rubinstein, R. (2005). A tutorial on the cross-entropy method. 134(1):19–67. Imported from research group DACS (ID number 277).

[Diederik P. Kingma, 2015] Diederik P. Kingma, J. L. B. (2015). Adam: A method for stochastic optimization. *ICLR*.

[Graham, 2014a] Graham, B. (2014a). Fractional max-pooling. *CoRR*, abs/1412.6071.

[Graham, 2014b] Graham, B. (2014b). Spatially-sparse convolutional neural networks. *CoRR*, abs/1409.6070.

[John Duchi, 2011] John Duchi, Elad Hazan, Y. S. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*.

[Krizhevsky, 2009] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.

[Mishkin and Matas, 2015] Mishkin, D. and Matas, J. (2015). All you need is a good init. *CoRR*, abs/1511.06422.

[Qian, 1999] Qian, N. (1999). On the momentum term in gradient descent learning algorithms.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117.

[Springenberg et al., 2014] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

A Appendix A

The code used for the optimizers evaluation:

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command with Theano backend (with TensorFlow, the GPU
    is automatically used):
     THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatx=float32 python
         cifar10_cnn.py

It gets down to 0.65 test logloss in 25 epochs, and down to
    0.55 after 50 epochs.
(it's still underfitting at tat point, though).
'''

from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import plot_model
from contextlib import redirect_stdout
import sys


batch_size = 32
num_classes = 10
epochs = 200
data_augmentation = True
methods = ['sgd', 'adam', 'adagrad']

for optimization_method in methods:
    stdout_new = open(optimization_method + "loss.txt", "a")
    old_stdout = sys.stdoutX_train
    sys.stdout = stdout_new
    try:
        #print
            ('************************************************')

        #print(optimization_method)
        #print
            ('************************************************')

        # The data, shuffled and split between train and test
            sets:
        (x_train, y_train), (x_test, y_test) = cifar10.
            load_data()
        #print('x_train shape:', x_train.shape)
        #print(x_train.shape[0], 'train samples')
        #print(x_test.shape[0], 'test samples')
```

```
        # Convert class vectors to binary class matrices.
        y_train = keras.utils.to_categorical(y_train,
            num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes
            )

        model = Sequential()

        model.add(Conv2D(32, (3, 3), padding='same',
                         input_shape=x_train.shape[1:]))
        model.add(Activation('relu'))
        model.add(Conv2D(32, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(Conv2D(64, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        model.add(Flatten())
        model.add(Dense(512))
        model.add(Activation('relu'))
        model.add(Dropout(0.5))
        model.add(Dense(num_classes))
        model.add(Activation('softmax'))

        model.compile(loss='categorical_crossentropy',
                      optimizer=optimization_method,
                      metrics=['accuracy'])

        x_train = x_train.astype('float32')
        x_test = x_test.astype('float32')
        x_train /= 255
        x_test /= 255

        # if not data_augmentation:
        #           # Let's train the model using RMSprop  #
            print('Not using data augmentation.')
        #     model.fit(x_train, y_train,
        #               batch_size=batch_size,
        #               epochs=epochs,
        #               validation_data=(x_test, y_test),
        #               shuffle=True)
        # else:
        #     print('Using real-time data augmentation.')
            # This will do preprocessing and realtime data
                augmentation:
        datagen = ImageDataGenerator(
            featurewise_center=False,  # set input mean to 0
                over the dataset
```

```
            samplewise_center=False,  # set each sample mean to
                0
            featurewise_std_normalization=False,  # divide
                inputs by std of the dataset
            samplewise_std_normalization=False,  # divide each
                input by its std
            zca_whitening=False,  # apply ZCA whitening
            rotation_range=0,  # randomly rotate images in the
                range (degrees, 0 to 180)
            width_shift_range=0.1,  # randomly shift images
                horizontally (fraction of total width)
            height_shift_range=0.1,  # randomly shift images
                vertically (fraction of total height)
            horizontal_flip=True,  # randomly flip images
            vertical_flip=False)  # randomly flip images

        # Compute quantities required for feature-wise
            normalization
        # (std, mean, and principal components if ZCA whitening
            is applied).
        datagen.fit(x_train)

        # Fit the model on the batches generated by datagen.
            flow().
        model.fit_generator(datagen.flow(x_train, y_train,
                                         batch_size=batch_size)
                                             ,
                            steps_per_epoch=x_train.shape[0] //
                                batch_size,
                            epochs=epochs,
                            validation_data=(x_test, y_test))
        plot_model(model, to_file='model_' +
            optimization_method + '.png')
        plot_model(model, to_file='model_' +
            optimization_method + '_2.png', show_shapes=True,
            show_layer_names=True)
        model.save_weights(filepath='model_weights_' +
            optimization_method, overwrite=True)
    finally:
        sys.stdout = old_stdout
```

B Appendix B

The code used for hyper-parameter optimization:

```
from __future__ import print_function
from hyperopt import Trials, STATUS_OK, tpe
from hyperas import optim
from hyperas.distributions import choice, uniform
from keras.datasets import mnist
from keras.layers.core import Dense, Dropout, Activation
from keras.models import Sequential
```

```
from keras.utils import np_utils
from keras.models import model_from_json
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation,
    Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD , Adam, Adagrad
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
import tensorflow as tf
import numpy as np


def data():
    nb_classes = 10
    # the data, shuffled and split between train and test sets
    (X_train, y_train), (X_test, y_test) = cifar10.load_data()
    print('X_train shape:', X_train.shape)
    print(X_train.shape[0], 'train samples')
    print(X_test.shape[0], 'test samples')

    # convert class vectors to binary class matrices
    Y_train = np_utils.to_categorical(y_train, nb_classes)
    Y_test = np_utils.to_categorical(y_test, nb_classes)

    X_train = X_train.astype('float32')
    X_test = X_test.astype('float32')
    X_train /= 255
    X_test /= 255

    # this will do preprocessing and realtime data augmentation
    datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over
            the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs
            by std of the dataset
        samplewise_std_normalization=False,  # divide each
            input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=0,  # randomly rotate images in the
            range (degrees, 0 to 180)
        width_shift_range=0.1,  # randomly shift images
            horizontally (fraction of total width)
        height_shift_range=0.1,  # randomly shift images
            vertically (fraction of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # compute quantities required for featurewise normalization
    # (std, mean, and principal components if ZCA whitening is
        applied)
    datagen.fit(X_train)
```

```
    return datagen, X_train, Y_train, X_test, Y_test

def model(datagen, X_train, y_train, X_test, y_test):
    num_classes = 10
    model = Sequential()
    model.add(Conv2D(32, (3, 3), padding='same',
                     input_shape=X_train.shape[1:]))
    model.add(Activation({{choice(['relu', 'sigmoid'])}}))
    model.add(Conv2D(32, (3, 3)))
    model.add(Activation({{choice(['relu', 'sigmoid'])}}))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout({{uniform(0, 1)}}))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation({{choice(['relu', 'sigmoid'])}}))
    model.add(Conv2D(64, (3, 3)))
    model.add(Activation({{choice(['relu', 'sigmoid'])}}))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout({{uniform(0, 1)}}))

    model.add(Flatten())
    model.add(Dense(512))
    model.add(Activation({{choice(['relu', 'sigmoid'])}}))
    model.add(Dropout({{uniform(0, 1)}}))
    model.add(Dense(num_classes))
    model.add(Activation('softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer={{ choice ([Adagrad(), Adam(), SGD
                     ()])}},
                  metrics=['accuracy'])

    model.fit_generator(datagen.flow(X_train, y_train,
                                     batch_size={{choice([16,
                                        32, 64, 128,265,512])
                                        }}),
                        samples_per_epoch=X_train.shape[0],
                        epochs=50,
                        validation_data=(X_test, y_test))
    score, acc = model.evaluate(X_test, y_test, verbose=0)
    return {'loss': -acc, 'status': STATUS_OK, 'model': model}


if __name__ == '__main__':
    import gc; gc.collect()

    best_run, best_model = optim.minimize(model=model,
                                          data=data,
                                          algo=tpe.suggest,
                                          max_evals=2,
                                          trials=Trials())
    print(best_model.evaluate(X_test, Y_test))
    print(best_run)
```