CERN SUMMER STUDENT PROJECT

Convolutional Neural Network

Sahand Seifnashri
August 24, 2015

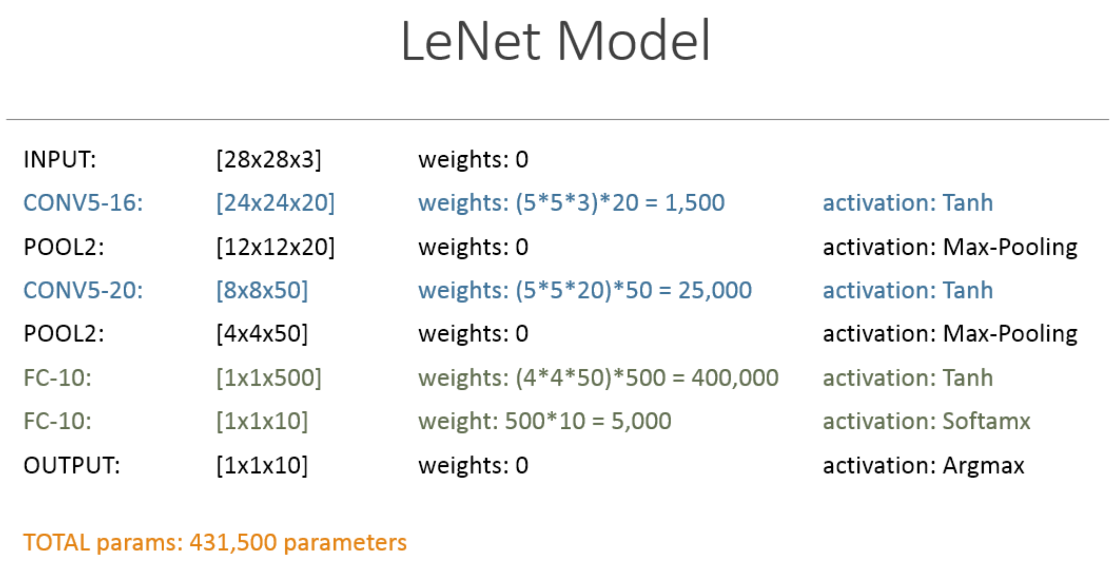
Supervisor: Jean-Roch Vlimant

TOOLKIT I used Python as programming language and Theano library [1]. Theano is a powerful library for evaluating mathematical expression like evaluating gradient for optimization and other calculations of neural networks. But the main reason for using Theano is because Theano is a GPU-enabled library which lets us to run our programs on GPU which is very fast for Neural Networks.
I started programing in ipython[2] which is an online notebook that allows you to use it in your browser and run codes in background even when your device is off. It is a shared notebook which allows others in the group to see and modify your codes. After a while I switched to pccitevo.cern.ch and felk40.cern.ch which are machines with GPU. I used notebook to prepare my codes and then ran them with GPU. The first machine has a GeForce 610 GPU and the second one has a Tesla K40 which is a very powerful GPU with 12 Gigabytes of memory. In my tests the Tesla K40 performed 14 times faster than the GeForce 610 and the GeForce was about 3 times faster than the Notebook's CPU[3].

1. MNIST DIGITS

For implementing Convolutional Neural Networks (CNNs) I used a well-known model called LeNet as a toy model. The LeNet model is one of the first successful applications of Convolutional Neural Networks that were developed by Yann LeCun in 1990's. It was used to read zip codes, digits, etc.

[1] `http://deeplearning.net/software/theano/`
[2] `cms-caltech-ml.cern.ch`
[3] Intel Core i7 9xx (Nehalem Class Core i7)



LeNet Model

INPUT:	[28x28x3]	weights: 0	
CONV5-16:	[24x24x20]	weights: (5*5*3)*20 = 1,500	activation: Tanh
POOL2:	[12x12x20]	weights: 0	activation: Max-Pooling
CONV5-20:	[8x8x50]	weights: (5*5*20)*50 = 25,000	activation: Tanh
POOL2:	[4x4x50]	weights: 0	activation: Max-Pooling
FC-10:	[1x1x500]	weights: (4*4*50)*500 = 400,000	activation: Tanh
FC-10:	[1x1x10]	weight: 500*10 = 5,000	activation: Softamx
OUTPUT:	[1x1x10]	weights: 0	activation: Argmax

TOTAL params: 431,500 parameters

Figure 1.1: LeNet-5 architecture.

1.1. LeNet-5 Model

LeNet architecture uses two convolutional layers each followed by a max-pooling layer. The convolutional layers have 20 and 50 filters respectively. Each convolutional layer has a 5x5 receptive field applied with a stride of 1 pixel. Each max pooling layer pools 2x2 regions at strides of 2 pixels. The convolutional layers are followed by a fully connected hidden layer with 500 units. All units use hyperbolic tangent activation function (Tanh). Schematic of LeNet architecture is shown in figure 1.1. For more detailed description see appendix A.

Python-Theano implementation of the LeNet model is also available in the Theano website:

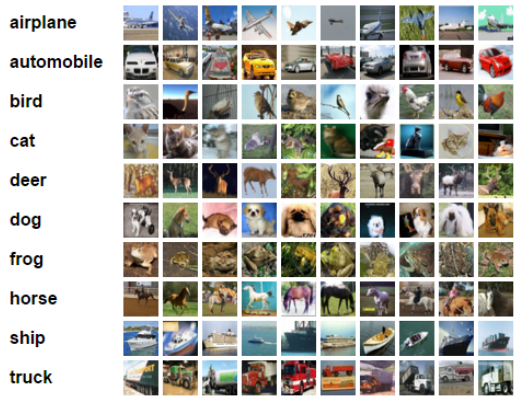
```
deeplearning.net/tutorial/lenet.html
```

in this link you can find a Python implementation of the LeNet model which is made for classifying MNIST digists.The MNIST dataset consists of 28x28 handwritten digit images which are divided in 60,000 examples for the training set and 10,000 examples for testing.

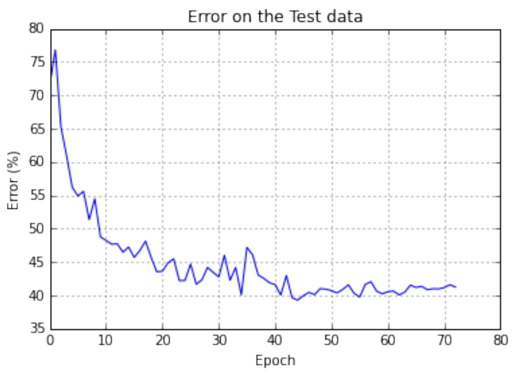
This Model and this Python code were my primitive toolkits to learn CNNs. With LeNet I got 0.9% error for classifying the MNIST dataset. Here the error is simply percentage of test samples that were wrongly labeled and having less final error is the only criteria for a good network. After verifying the 0.9% error with the LeNet model and running the code with notebook I started to modify the LeNet Python code to build a larger and more complex CNNs. For first modification I used the LeNet model for training a different dataset, the CIFAR-10 dataset.

2. CIFAR-10

CIFAR-10 dataset[4] consists of 60000 32x32 colour images in 10 classes, with 6000 images per class (figure 2.1a). There are 50000 training images and 10000 test images. It is a famous dataset



(a) The CIFAR-10 Dataset



(b) Classifying the CIFAR-10 with LeNet

Figure 2.1: CIFAR-10

for object recognition and there is also a leaderboard on Kaggle website[5] for this dataset.

Result of classifying the CIFAR-10 with LeNet model is plotted in figure 2.1b and you can find the implementation code in my notebook directory[6]. It converged to 40% error. You can also find latest research methods in object classification on Rodrigo Benenson's page[7], good models have errors about 10% and the best model have 6% error which is nearly the same as the human performance!

I spent a lot of time on classifying the CIFAR-10 dataset, I chose it as my main dataset to study and examine every feature of CNNs. Next I switched form LeNet architecture to another architecture that was made specifically for the CIFAR-10 dataset.

2.1. ConvNet Model

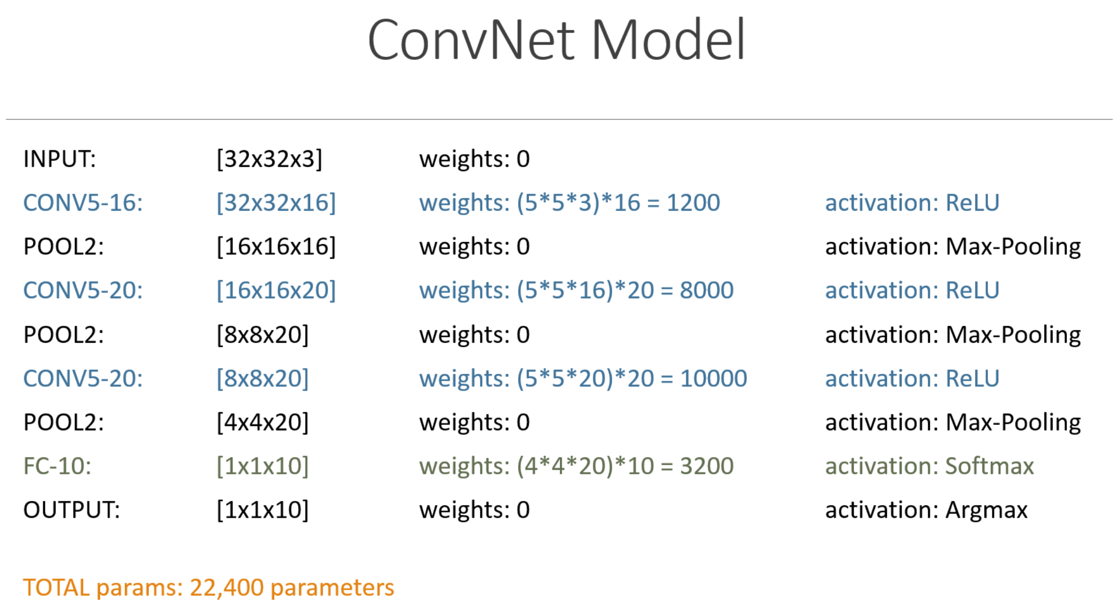
ConvNet model is a simple CNN architecture for classifying the CIFAR-10 dataset, I found it on Stanford website[8], It was in a course webpage about CNNs which also has very useful things about Neural Networks. Here is link of the webpage:

`http://cs231n.stanford.edu/`

ConvNet architecture uses three convolutional layers each followed by a max-pooling layer. The convolutional layers have 16, 20 and 20 filters respectively. Each convolutional layer has a

[4] http://www.cs.toronto.edu/~kriz/cifar.html
[5] https://www.kaggle.com/c/cifar-10
[6] https://cms-caltech-ml.cern.ch/notebooks/Sahand/CNNs.ipynb.
[7] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
[8] http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

5x5 receptive field applied with a stride of 1 pixel and zero padding of 2. Each max pooling layer pools 2x2 regions at strides of 2 pixels. All units use the rectified linear activation function (ReLU) (for more information see appendix A). Schematic of ConvNet architecture is shown in figure 2.2.



ConvNet Model

INPUT:	[32x32x3]	weights: 0	
CONV5-16:	[32x32x16]	weights: (5*5*3)*16 = 1200	activation: ReLU
POOL2:	[16x16x16]	weights: 0	activation: Max-Pooling
CONV5-20:	[16x16x20]	weights: (5*5*16)*20 = 8000	activation: ReLU
POOL2:	[8x8x20]	weights: 0	activation: Max-Pooling
CONV5-20:	[8x8x20]	weights: (5*5*20)*20 = 10000	activation: ReLU
POOL2:	[4x4x20]	weights: 0	activation: Max-Pooling
FC-10:	[1x1x10]	weights: (4*4*20)*10 = 3200	activation: Softmax
OUTPUT:	[1x1x10]	weights: 0	activation: Argmax

TOTAL params: 22,400 parameters

Figure 2.2: ConvNet architecture.

Implementing the ConvNet Model For implementing this new architecture I had to do some modification on the Python code of the LeNet model which I had gotten from the Theano website. From LeNet to ConvNet model 3 things had to be changed:

1. Number of CONV layers changed from 2 to 3
2. Activation functions changed from Tanh to ReLU
3. Zero padding of CONV layers changed from 0 to 2

Implementing the two first things was straight forward but the third one was not. Zero padding is a method to control size of image after a CONV layer by adding zeros to border of the image. In this case with 5x5 filters, zero padding of 2 keeps size of the images unchanged after a convolutional layer.
Index-assignment is not supported in Theano shared variables and I had to use:

```
theano.tensor.set_subtensor()
```

in my code (see listing 1). I ran the code without any syntax error but its result (final error) wasn't satisfying, It was worse than the LeNet case. Something was definitely missing.

```
zero = T.zeros((image_shape[0], filter_shape[0], image_shape[2], image_shape[3]),
                dtype = theano.config.floatX)
zero_padding = T.set_subtensor(zero[:,:,2:image_shape[2]-2,2:image_shape[3]-2],
                                conv_out)
```

Listing 1: conv_out is just the output of the CONV layer and zerro_padding is the zero padded output.

2.1.1. Optimization

The Python code for the LeNet model was using gradient-decent with mini-batch size of 500 (for details see appendix B). But two major things should have changed with the CIFAR-10 dataset, mini-batch size and training method. I started to examine different methods and found a lot of information about optimization on Internet, I suggest this webpage `http://cs231n.github.io/neural-networks-3/` which you can find plenty of useful information about optimization. I found "Adadelta" a reasonable method to start with and it was recommended on literature. With this choice I had 4 new hyper-parameters to tune, the mini-batch size and Adadelta's parameters. According to Stanford website, where I had found the ConvNet model, I set mini-batch size to 4 (So it changed from 500 to 4). But on the website there were no information about the Adadelta's parameters so I started to tuning them. After hours of training I found these values:

`learning_rate=0.01, rho=0.95, epsilon=1e-6`

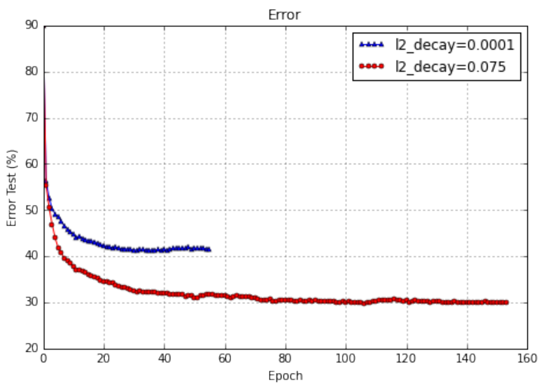
With these parameters and this new training method the error problem was resolved and I got 40% error after just 25 epochs. Speed of convergence per epoch was better than the LeNet model but final result had not been improved. You can also find the code in my notebook directory[9].

2.1.2. Regularization

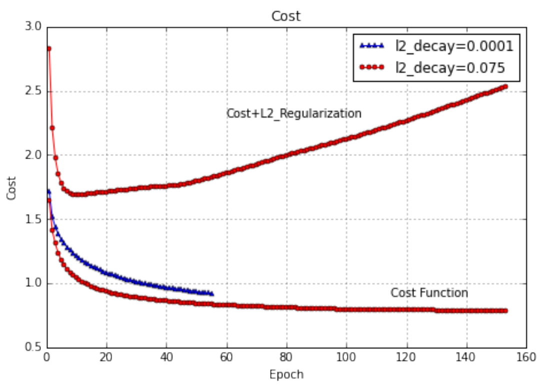
L2 Regularization Regularization methods are methods to prevent overfitting. I noticed that the ConvNet model was using L2-Regularization (see appendix C) with l2_decay parameter of 0.0001. I tried to tune this parameters and I got a better and odd result. I found 0.075 as the tuned parameter and with this value I got 30% error which was a better result. Results are plotted in figure 2.3. By setting l2_decay to 0.075 while total cost function (Cost + L2-Regularization terms) was increasing, error on the test set was decreasing which is very strange.

Mean while I noticed a fact about Adadelta method. When I was reading the Adadelta paper I found that the Adadelta method actually doesn't have a `learning_rate` parameter and Putting a `learning_rate` parameter by hand could cause bad results because Adadelta is an adaptive learning rate method and after a couple of training iterations it changes the

[9]`https://cms-caltech-ml.cern.ch/notebooks/Sahand/ConvNets_CIFAR-10-Adadelta-Batchsize4-learningrate-0.01.ipynb`



(a) Error on the test set



(b) Cost on the training set

Figure 2.3: L2 Regularization

effective learning rate. By putting a `learning_rate` by hand I might ruin the learning process which resulted in increasing the cost function after a while. So I set `learning_rate` to 1 and started to tune two remaining parameters again and I found these values:

`learning_rate=1.0, rho=0.995, epsilon=1e-9`

But I couldn't get any better result from these neither. Next I found another regularization method which was really powerful.

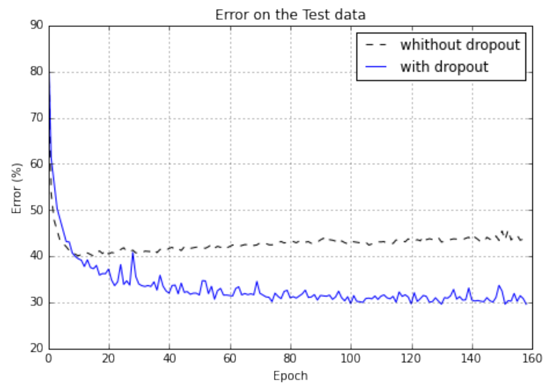
DROPOUT Dropout is a an extremely effective method which basically kills each neurons with some probability p in each training iteration (see appendix C.2). I examined the dropout method with dropout rate 0.8 (which is probability of keeping a neuron active in each training iteration) for all hidden layers. Results are plotted in figure 2.4. As you can see It reached 30% error (but not accidentally this time) without changing any hyper-parameter which shows that this method is really effective.
For implementing the dropout method there is two ways, original method and inverted method and here I used the inverted method because it was recommended in Stanford website and I though it was easier to implement which was not (for detail see appendix C.2). Full implementation of the model can be found here:

`https://cms-caltech-ml.cern.ch/notebooks/Sahand/ConvNets_CIFAR-10_Dropout.ipynb`

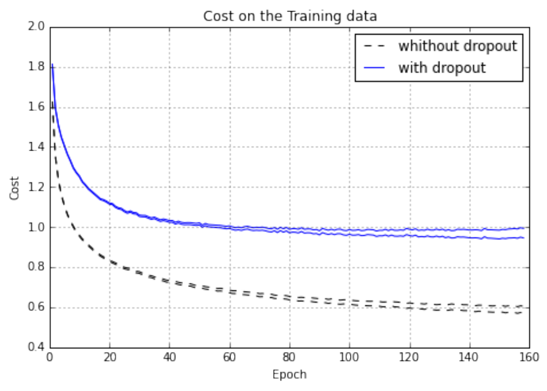
After implementing this new regularization method the network needs a new optimization setting.

2.2. OPTIMIZATION

Getting 30% error from the model *without dropout* made me to think that the model might was capable of improvement if we could have managed to train it better. So I started to examine new training methods.

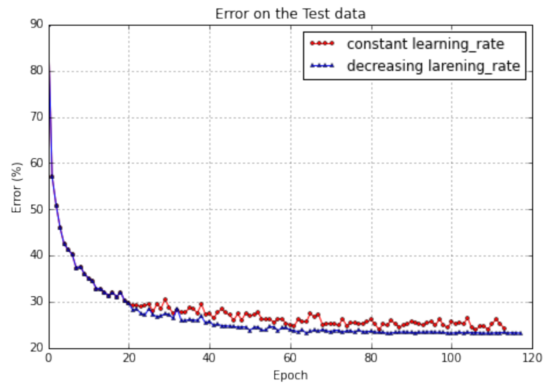


(a) Error of the model without dropout starts to increasing after some epochs which clearly indicates over-fitting which is absent in dropout

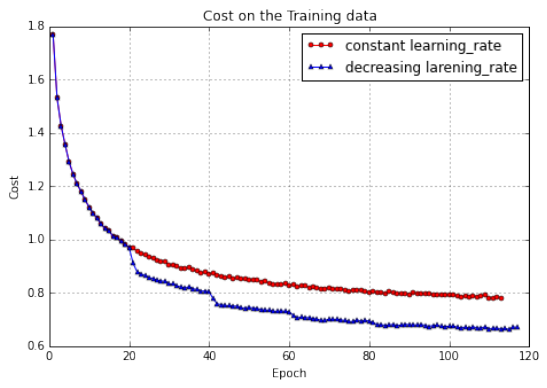


(b) The graphs split because of L2 regularization terms. Each model has two cost one with L2 regularization terms and one without them.

Figure 2.4: As you can see dropout improves error with even higher cost values!



(a) Error



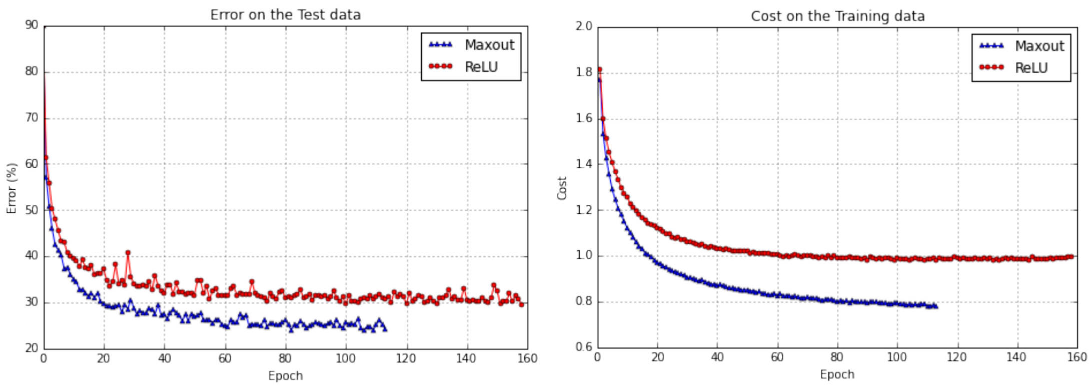
(b) Cost

Figure 2.5: learning_rate was initially set to 1e-4 and it was dived by 2 after each 20 epochs. Notice that both models use maxout units.

Again I searched in literature for more optimization methods and I found a new approach that is called *Rmsprop* (see appendix B.1). *Rmsprop* is nearly the same as *Adadelta* with one difference that it has a `learning_rate` parameter which is very useful because you may want to decrease rate of learning (length of update steps in each training iteration) after some epochs and you can do it just by dividing `learning_rate`. Thus I switched from Adadelta to Rmsprop and I tuned Rmsprop parameters and found these values:

```
learning_rate=1e-4, rho=0.999, epsilon=1e-8
```

In figure 2.5 you can see effectiveness of decreasing learning rate during training process. Both of the models in the figure use *Rmsprop* method.



(a) Error (b) Cost

Figure 2.6: Maxout vs ReLU

So far I'd examined lots of things in optimization and regularization but had not changed anything about activation functions.

2.3. Activation Functions

From the LeNet model to ConvNet model I switched form Tanh activation function to ReLU which is faster and more efficient than Tanh according to literature. ReLU is computationally simpler and except one problem, which is called "dying ReLU"[10], it performs better than Tanh. The "dying ReLU" problem can be resolved by replacing it with a new and powerful unit called Maxout unit (see appendix A). So I changed all of my ReLU units to Maxout and I got better results which are shown in figure 2.6 and you can find the code here:

`https://cms-caltech-ml.cern.ch/notebooks/Sahand/ConvNets_CIFAR-10_Maxout.ipynb`

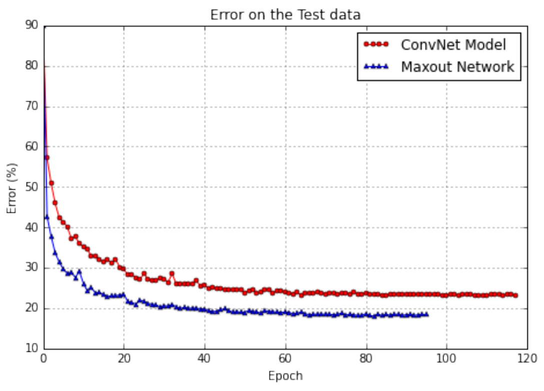
So Maxout unit is more efficient and more accurate than ReLU in the sense that it reached 30% error at 20th epoch which is about 3 times better than previous models and in accuracy it reached about 25% error which is better than previous models. Notice that both of these two models use dropout method.

After implementing everything about optimization methods, regularization and activation functions now is time to change architecture and use a deeper network.

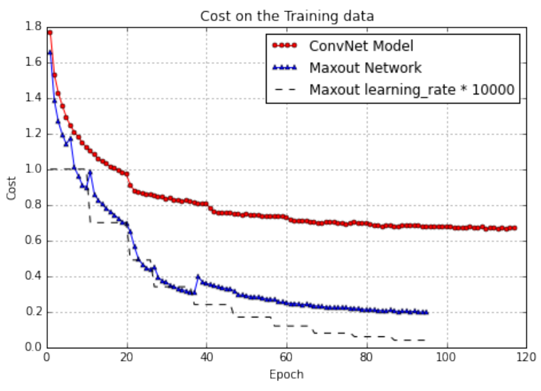
2.4. Maxout Network

After searching through literature I found[11] having three convolutional layers (like ConvNet model) and one or two fully connected layers at the end with dropout method and maybe with Maxout unit is capable of giving accuracy about 90%. So after a little playing with different

[10] `http://cs231n.github.io/neural-networks-1/`
[11] `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`



(a) Error



(b) Cost

Figure 2.7: Both models have decreasing learning_rate and learning_rate of the Maxout Network is plotted in figure (b).

architectures I decided to just add one fully connected layer with 500 units and change the number of filters in the convolutional-layers to 50, 100 and 200 filters respectively. I used Maxout unit for all the hidden layers.
I named this model Maxout Network because this architecture is almost the same as the architecture of Goodfellow et al. paper[12] in which maxout unit was first introduced, except number of filters in convolutional layers that hasn't been mentioned in the paper.
I set dropout rates from input layer to fully-connected layer to $p_{dropout} = [1.0, 0.75, 0.75, 0.75, 0.5]$. The reason for choosing these values was that in literature it was recommended to use 0.5 dropout rate for fully-connected layers and 0.75 for convolutional-layers which I implemented the same values, but it was suggested every where to use a dropout rate about 0.8 to 0.9 for input layer which I found catastrophic in my results, so I just set the input dropout rate to 1. I started to tune hyper-parameters and I finally used these values and started to train the CNN.

```
learning rate = 1e-4, rho=0.999, epsilon=1e-8
batch_size=4, l2_decay=0.0001, p_dropouts=[1.0, 0.75, 0.75, 0.75, 0.5]
```

In my implementation I divided learning_rate by 0.7 after about each 10 epochs, the exact learning_rate is plotted in figure 2.7
The code ran for about 11 hours and it converged after about 50th epochs. After 50 epochs test error decreased to 18% which is a new record between my previous models. For this Network it took 6 hours to converge with 'Tesla K40' (every epochs took 7'10").
Results are plotted in figure 2.7. You can also find the code in my notebook directory here:

`https://cms-caltech-ml.cern.ch/notebooks/Sahand/Maxout_Network.ipynb`

Maxout Network was my final work on the CIFAR-10 dataset. Before go through the next dataset which is a physical dataset I want to mention some useful tips.

[12]`http://arxiv.org/abs/1302.4389`

Dropout implementation As I mentioned before for implementing dropout method I have been using the inverted method which I though is the same as the normal method, but unfortunately it wasn't. I tried the normal method and I got a better result on Maxout Network. Thus I think implementing the inverted method was a wrong choice and I changed the implementation to the normal method (In Maxout Network I used normal method).

2.5. Optimization

Mini-Batch Networks with smaller mini-batch sizes at the end converge to better results but they are a lot slower than the models with bigger mini-batch sizes. I strongly recommend to use bigger mini-batch sizes to find a suitable architecture or tuning the hyperparameters like dropout rates and number of features in different layers and only for gaining the final results set mini-batch size to 4 or even 1. But keep in mind that training a network with small mini-batch size like 4 requires more time and it needs smaller learning_rate because by decreasing mini-batch size you are increasing the number of iterations in each epochs so you need smaller steps.

Learning Rate Another useful thing is we don't have to use the same learning_rate for the parameters in each layer. It is better to choose different learning_rates for each layer which really improve the speed of the training process but it needs more tuning. In my models input layers needed bigger learning_rate.

Data Preprocessing Every dataset needs a specific data preprocessing. For the CIFAR-10 dataset two methods are mainly used in the literature, global contrast normalization and ZCA whitening. I just had time to implement global contrast normalization, it improves final error about 1%. Global contrast normalization means that for image and each color channel in that image, we compute the mean of the pixel intensities and subtract it from the channel.

Conclusion As I mentioned before according to literature I should have gotten error about 10% with this Network. But two things is missing here first data preprocessing and second dropout rate for the input layer. I tried dropout rate for the input layer but it worsen the results. I think the problem is data preprocessing that I hadn't implemented and I think by implementing data preprocessing we can safely put dropout rate for the input layer. But it is just a conjecture and unfortunately I hadn't time to check it and now is time for the physical dataset. You can find summary of all the results on the CIFAR-10 dataset in figure 2.8.

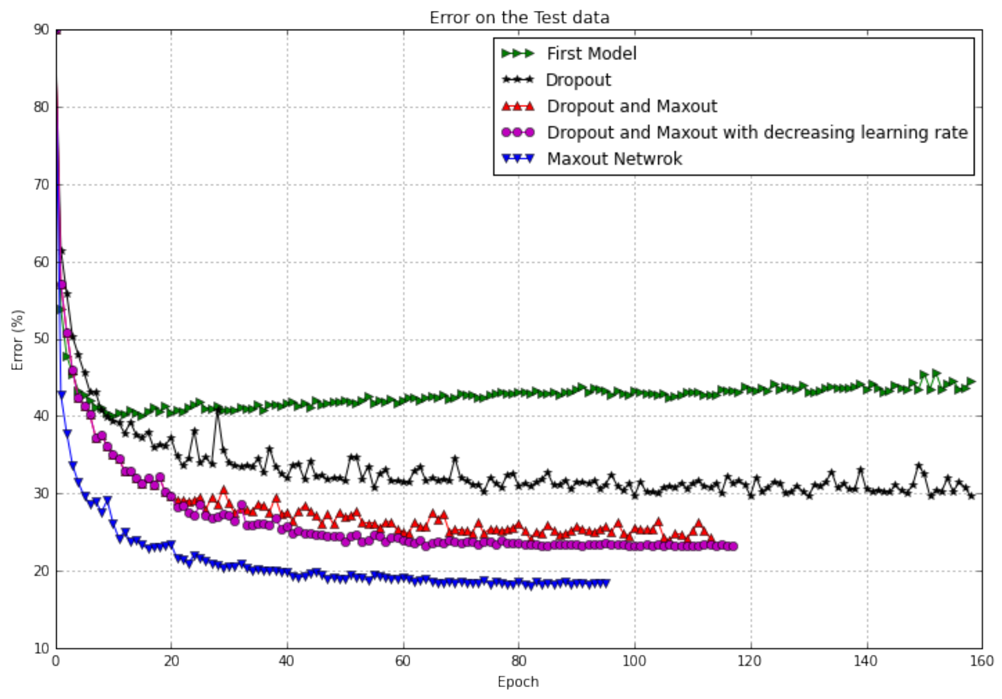


Figure 2.8: All results.

3. Quark-Gluon Discrimination

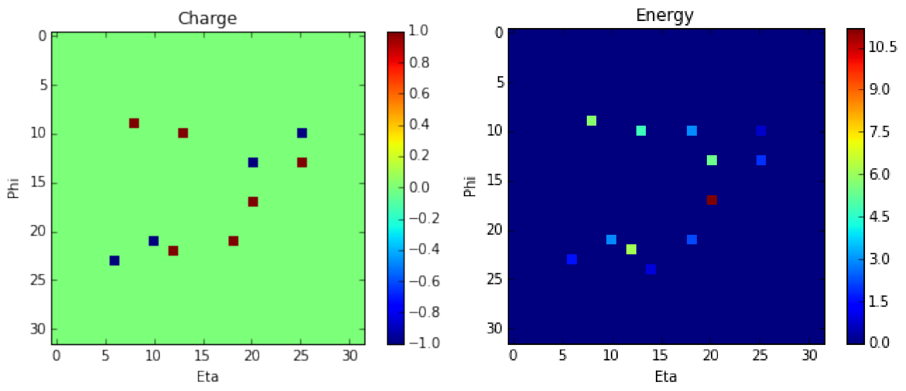
3.1. Dataset

The dataset consists of Jet and its constituents (PF Candidates) information: Energy, Charge, Phi and Eta, from Jet collection from MINIAODSIM (AK4 CHS)[13]. The data are transformed into 32x32 images. The width and height of each image represent Eta and Phi coordinate of Jet constituents which ranges between -0.5 to 0.5. Each Image has 3 features: sum of energies, sum of charges and sum of absolute charges of jet constituents and the images are also centered around the Jet itself(see figure 3.1). About 65% of Jets are Gluons and 35% of them are Quarks.

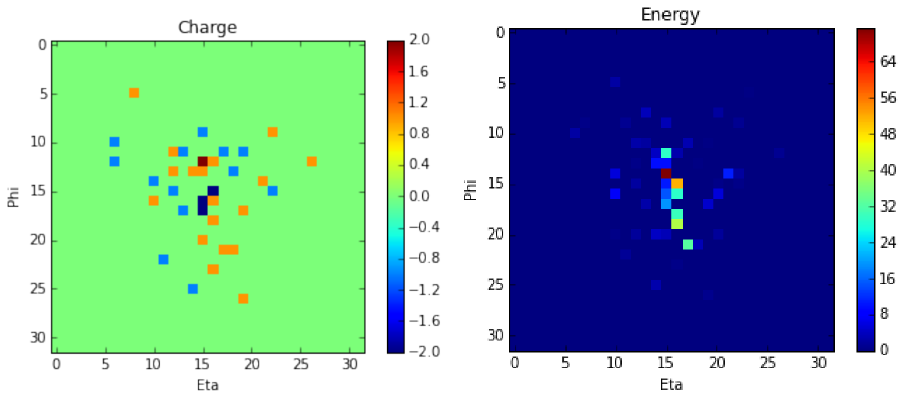
This dataset is not a perfect dataset for image processing and convolutional neural networks because as you can see in figure 3.1a a typical Jet has roughly about 10 constituents which makes the images very sparse. The histogram of number of Jet constituents is plotted in figure 3.2. So I decided to keep only the denser images.

The images were also divided in 2 categories of Quark jets and Gluon jets. The same number of quark jets and gluon jets have been selected in the dataset.

[13] `/store/mc/RunIISpring15DR74/QCD_Pt_300to470_TuneCUETP8M1_13TeV_pythia8/MINIAODSIM/Asympt25ns_MCRUN2_74_V9-v1/`



(a) a typical Gluon Jet



(b) Images of a Relatively dense Gluon Jet

Figure 3.1: Jet images

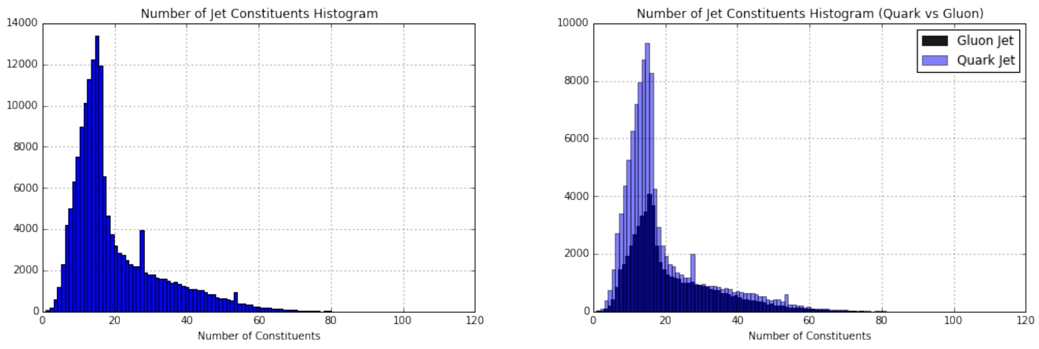


Figure 3.2: Histogram of number of jet constituents in each jet.

3.2. Architecture

The architecture uses three convolutional layers each followed by a max-pooling layer. The convolutional layers have 32, 64 and 128 filters respectively. Each convolutional layer has a

5x5 receptive field applied with a stride of 1 pixel and zero padding of 2. Each max pooling layer pools 2x2 regions at strides of 2 pixels. All units use the maxout activation function and there is also two fully-connected hidden layers after convolutional layers which have 128 and 64 neurons respectively. This network isn't far from the Maxout Network. I reduced number of features in convolutional layers because it didn't change the final result but made the network faster to converge. I also added one additional fully-connected hidden layer because it didn't change the speed of convergence much, but it slightly improved the results.

DROPOUT As I mentioned before for the CIFAR-10 dataset having dropout rate for the input layer in my models didn't help and worsen the results, but with the Jet dataset I tried dropout rate for the input layer and it worked and improved the results (it decreased the difference between error on the test set and error on the training set).

3.3. RESULTS

FIRST DATASET The first dataset consisted of 177,000 Images which were divided in 150,000 examples for the training set and 27,000 examples for testing and only the images which had more than 30 constituents had been selected. I trained the network with this dataset and through different phases of training, forexample I trained the network down to 37% error and then changed the training method and then it reached down to 36% error and .... Therefore I don't have any plot for cost and error versus epoch. But in most cases the error converged to 35% error and I couldn't get any better result.

For the other dataset I increased size of the training set to 800,000 Images which improved the final error about 0.5%. For another attempt I used the images which had more than 60 constituents with the same size of training set, but surprisingly it worsen the final results. So it seems that selecting the denser images is not a good choice.

For the final dataset I used all of the images, without triggering based on the number of constituents and it consisted of 407,000 Images which were divided in 400,000 examples for the training set and 7,000 examples for testing. I trained the network with the following hyper-parameters

```
learning_rate=0.002, rho=0.9, epsilon=1e-08, batch_size=1024,
l2_decay=0.0001, dropout_rates=[0.9, 0.75, 0.75, 0.75, 0.5, 0.5]
```

it slightly improved the results to 35.3% error. I used a big batch_size (1024) because of lack of time. I think we may can go further a little (maybe 1%) by decreasing the batch_size down to 4, but it requires more time for converging (with this big dataset maybe one or two days). But 1% isn't enough and this dataset needs more sophisticated and its own architecture to get a better result and finding that architecture also needs more time. The code can be find here:

`https://cms-caltech-ml.cern.ch/notebooks/Sahand/Jet_Discrimination.ipynb`

3.4. Conclusion

Given the fact that for classifying the CIFAR-10 I have been through a lots of things and it took me one month to find a suitable architecture for it, every specific dataset needs its own architecture and it requires time which unfortunately I did not have. With one week of training on the Jet data the validation error on the test set reached to 35%. But I think by more training and trying different architectures we can still decrease this error further on this dataset.

Next Step The main aim of this project was to use Calorimeters data which unfortunately I didn't manage to get. The Calorimeter dataset looks more like *images* and the CIFAR-10 dataset which I spend most of my time on it. I think the next step would be to use Convolutional Neural Networks on the Calorimeters data or any other dataset which looks more like images.

APPENDICES

A. Type of Neurons

About type of neurons and activation functions you can find a lot of good information and almost everything for image recognition in these webpages:

```
http://cs231n.github.io/convolutional-networks
http://cs231n.github.io/neural-networks-1
```

I just mention some of the main contents here but for more details please go to the links above.

I used three main types of layers to build ConvNet architectures: *Convolutional Layer*, *Pooling Layer*, and *Fully-Connected Layer* (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and the region they are connected to in the input volume. This may result in volume such as [32x32x12].
- RELU layer will apply an elementwise activation function, such as the *max(0,x)* thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

A.1. Maxout

Other types of units have been proposed that do not have the functional form f(wTx+b) where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron (introduced recently by Goodfellow et al.) that generalizes the ReLU and its leaky version. The Maxout neuron computes the function max(wT1x+b1,wT2x+b2). Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have w1,b1=0). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

B. Optimization

There is also plenty of good information about optimization in the Stanford course webpage[14]. I put some of them here.

B.1. RMSprop

RMSprop is a very effective, but currently unpublished adaptive learning rate method. Amusingly, everyone who uses this method in their work currently cites slide 29 of Lecture 6 of Geoff Hinton's Coursera class. The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead, giving:

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

Here, decay_rate is a hyperparameter and typical values are [0.9, 0.99, 0.999]. Notice that the x+= update is identical to Adagrad, but the cache variable is a "leaky". Hence, RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, which has a beneficial equalizing effect, but unlike Adagrad the updates do not get monotonically smaller.

Implementation Implementation of the RMSprop and other optimization methods in Theano is a little tricky but efficient. It uses `OrderedDict` object which saves the updates rule in it and by putting it in a `theano.function`, every time the function is called Theano will update the parameters according to the rules which have been stored in the `OrderedDict` object. The implementation is shown in listing 2.

[14] http://cs231n.github.io/neural-networks-3

```python
# compute the gradients of the cost with respect to the params
grads = T.grad(cost, params)

# define rmsprop train update
def rmsprop(learning_rate, rho, epsilon):

        # updates_l will save the update rules on it
        updates_l = OrderedDict()

        for param, grad in zip(params, grads):
                value = param.get_value(borrow=True)

                # intiate the cache parameter with zeros
                # with the same size as the corresponded param
                accu = theano.shared(np.zeros(value.shape, dtype=value.dtype),
                                        broadcastable=param.broadcastable)

                # update cache (accu)
                accu_new = rho * accu + (1 - rho) * grad ** 2
                updates_l[accu] = accu_new

                # update the params
                updates_l[param] = param - (learning_rate * grad /
                                T.sqrt(accu_new + epsilon))

        return updates_l

updates = rmsprop(learning_rate=1e-4, rho=0.999, epsilon=1e-8)

# the function which will compute the cost function
# and updates the parameters each time it is called
train_model = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
                x: train_set_x[index * batch_size: (index + 1) * batch_size],
                y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
)
```

Listing 2: RMSprop Theano implementation: train_model is a theano function which its output is the cost function and each time it is called it updates the params according to rules which are stored in updates.

B.2. Adadelta

Adadelta is nearly the same as the RMSprop. For more details see Zeiler, M. D. (2012)[15]. The implementation is the same as the Adadelta and you can find it in the footnote[16]

C. Regularization

C.1. L2 Regularization

L2 Regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight ω in the network, we add the term $\frac{1}{2}\lambda\omega^2$ to the objective, where λ is the regularization strength. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors.

C.2. Dropout

Dropout is an extremely effective, simple and recently introduced regularization technique by Srivastava et al. in Dropout: A Simple Way to Prevent Neural Networks from Overfitting[17] that complements the other methods (L1, L2, maxnorm). While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. For more detailed information see:

`http://cs231n.github.io/neural-networks-2`

[15] `http://arxiv.org/abs/1212.5701`
[16] `https://github.com/Lasagne/Lasagne/blob/master/lasagne/updates.py`
[17] `http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`