# IE598 Deep Learning and Neural Networks

Pramod Srinivasan (psrnvsn2)

The assignment deals with the implementation of a convolutional neural network. Among other aspects, this involves the implementation of forward and backward passes for convolutional layers, pooling layers, inter-layer batch normalization and various optimization techniques.

## Convolutional Neural Network : Background

The input to our ConvNet Architecture is a fixed size 32 X 32 RGB image. The image is passed through a stack of convolutional layers where *filters* with a receptive field of size 3 X 3 is used. The *convolution stride* is set to 1 pixel and the spatial padding is set to 1 pixel. Max-pooling is performed over a 2 X 2 pixel window with stride 2.

The stack of convolutional layers is followed by a three *fully-connected* (FC) layers, the first two have 500 channels each and the third performs 10-way CIFAR classification. The final layer is a *softmax layer*.

The most common architecture is a stack of a few Convolutional followed by RELU layers, or otherwise called CONV-RELU layers, which are in turn followed by POOL layers. The pattern is repeated until the image has merged spatially to a small size at which point it transitions into fully-connected layers. The final fully-connected layer holds the output, such as the class score. In other words, we can state the ConvNet as the following:

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

$0 <= N <= 3$, $M >= 0$ and $K >= 0$

Here we attempt to address the size of representation to be maintained at each stage of the processing and keep track of both the representation size and the total number of weights. The following table gives us an idea of the number of parameters as well as memory requirements for the baseline architecture [1]:

| LAYER | DIM | MEMORY | PARAMETER COUNT |
|---|---|---|---|
| INPUT | [32x32x3] | 32*32*3 | 0 |
| CONV3-64 | [32x32x64] | 32*32*64 | (3*3*3)*64 |
| CONV3-64 | [32x32x64] | 32*32*64 | (3*3*64)*64 |
| POOL2 | [112x112x64] | 112*112*64 | 0 |
| CONV3-128 | [112x112x128] | 112*112*128 | (3*3*64)*128 |
| CONV3-128 | [112x112x128] | 112*112*128 | (3*3*128)*128 |
| POOL2 | [56x56x128] | 56*56*128 | 0 |
| CONV3-256 | [56x56x256] | 56*56*256 | (3*3*128)*256 |
| CONV3-256 | [56x56x256] | 56*56*256 | (3*3*256)*256 |
| CONV3-256 | [56x56x256] | 56*56*256 | (3*3*256)*256 |
| POOL2 | [28x28x256] | 28*28*256 | 0 |
| CONV3-512 | [28x28x512] | 28*28*512 | (3*3*256)*512 |
| CONV3-512 | [28x28x512] | 28*28*512 | (3*3*512)*512 |
| CONV3-512 | [28x28x512] | 28*28*512 | (3*3*512)*512 |
| POOL2 | [14x14x512] | 14*14*512 | 0 |
| CONV3-512 | [14x14x512] | 14*14*512 | (3*3*512)*512 |
| CONV3-512 | [14x14x512] | 14*14*512 | (3*3*512)*512 |
| CONV3-512 | [14x14x512] | 14*14*512 | (3*3*512)*512 |
| POOL2 | [7x7x512] | 7*7*512 | 0 |
| FC | [1x1x4096] | 4096 | 7*7*512*4096 |
| FC | [1x1x4096] | 4096 | 4096*4096 |
| FC | [1x1x1000] | 1000 | 4096*1000 |

The input layer contains the image which is passed into the CONV layer. The CONV layer preserves the spatial size of the image input while the POOL layers down-sample the volumes spatially. RELU layer applies an elementwise activation function such as the $max(0, x)$ thresholding at zero, thereby leaving the volume unchanged.

We now present the implementation details and subsequent experimental findings to improve the performance of the given VGG-NET for CIFAR-10 dataset. Specifically, we have covered the following aspects:

1. Deeper networks
2. Interlayer Batch Normalization
3. Dropout
4. Pooling Layers
5. Convolution Layers
6. Different Optimization Techniques
7. Data Augmentation

## 1 Deeper Networks

The aim of this experiment is to understand the how the ConvNet depth affects their accuracy in the large-scale image recognition setting. We rigorously evaluate networks of increasing depth as shown in Figure . An important observation is that each convolutional layer uses small 3x3 filters with the convolutional stride set to 1 as shown in the experiment table from the seminal paper [2].

We are focusing on architectures A, D and E. Since the input CIFAR-10 image is of size $32X32X3$ as opposed to ImageNet image of size $224X224X3$, we have decided to prune down the architecture to a maximum of 4 layers. The comparison of the architectures is shown below.

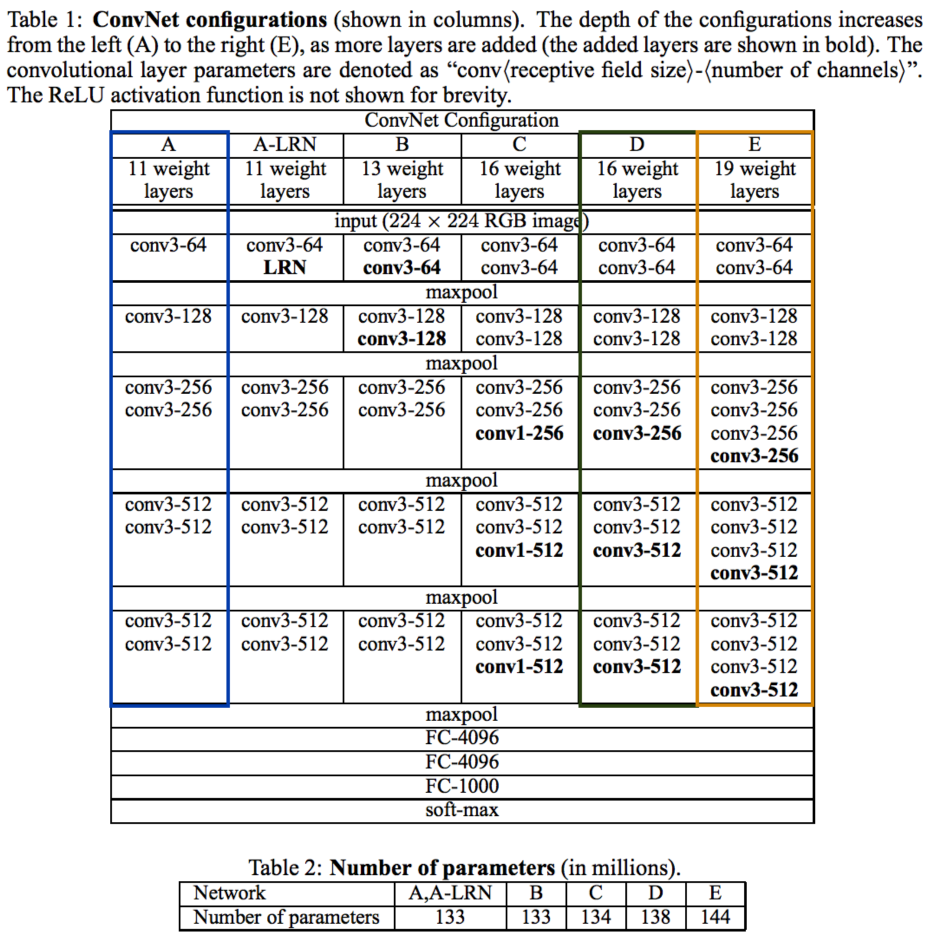

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64 | conv3-64 | conv3-64 | conv3-64 | conv3-64 |
| | **LRN** | **conv3-64** | conv3-64 | conv3-64 | conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128 | conv3-128 | conv3-128 | conv3-128 |
| | | **conv3-128** | conv3-128 | conv3-128 | conv3-128 |
| maxpool | | | | | |
| conv3-256 | conv3-256 | conv3-256 | conv3-256 | conv3-256 | conv3-256 |
| conv3-256 | conv3-256 | conv3-256 | conv3-256 | conv3-256 | conv3-256 |
| | | | **conv1-256** | **conv3-256** | conv3-256 |
| | | | | | **conv3-256** |
| maxpool | | | | | |
| conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 |
| conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 |
| | | | **conv1-512** | **conv3-512** | conv3-512 |
| | | | | | **conv3-512** |
| maxpool | | | | | |
| conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 |
| conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 | conv3-512 |
| | | | **conv1-512** | **conv3-512** | conv3-512 |
| | | | | | **conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

Table 2: **Number of parameters** (in millions).

| Network | A,A-LRN | B | C | D | E |
|---|---|---|---|---|---|
| Number of parameters | 133 | 133 | 134 | 138 | 144 |

Figure 1: Comparison of Deep Networks [highlighted ones under consideration]

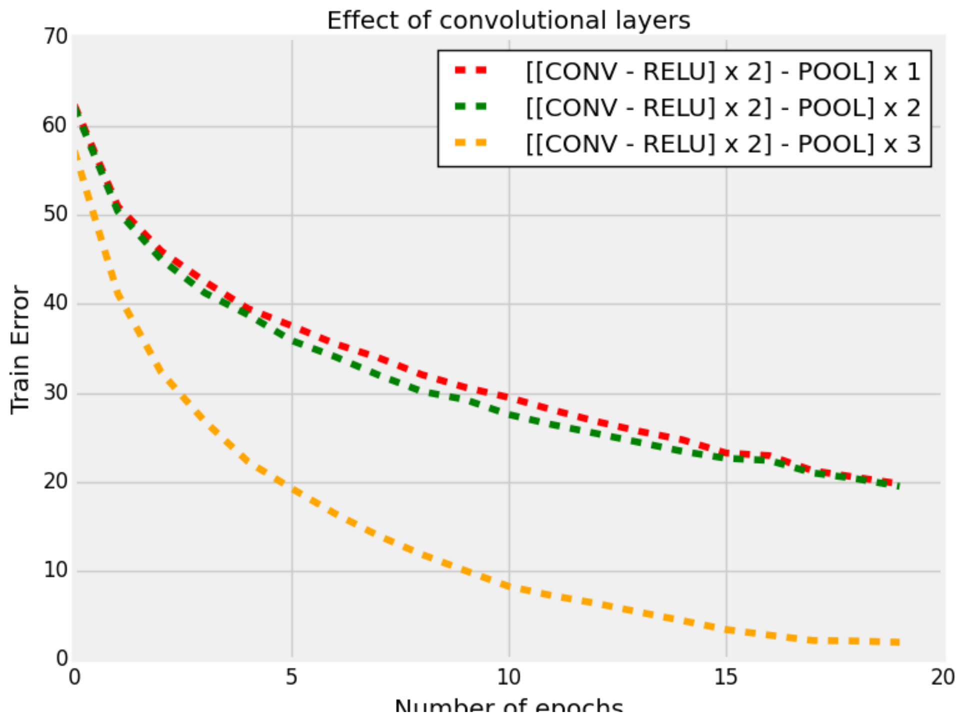

Figure 2: Training performance of Architectures with various Convolutional layers

**Observations**

- Use of very small receptive field make it feasible to stack more convolutional layers.
- Increasing the number of CONV layers leads to decreasing training error
- With every new POOL layer, downsampling of image volume can counter the increase in training time

## 2 Analysis of Interlayer Batch Normalization

Batch normalization is a simple and effective way to improve the performance of a neural network [3]. It is established that batch normalization enables the use of higher learning rates besides acting as a regularizer thus achieving a speed-up in the training process. In the figure 3, we observe the positive effect of batch normalization as we increase the number of weight layers. Chainer provides features for certain parameters which can be learnt during the backpropagation to unlearn the normalization and thereby recover the identity mapping.

An important observation is that during the test time, BatchNormalization layer functions differently. The mean and variance are not computed based on batch instead a single fixed empirical mean of activations during training is used.

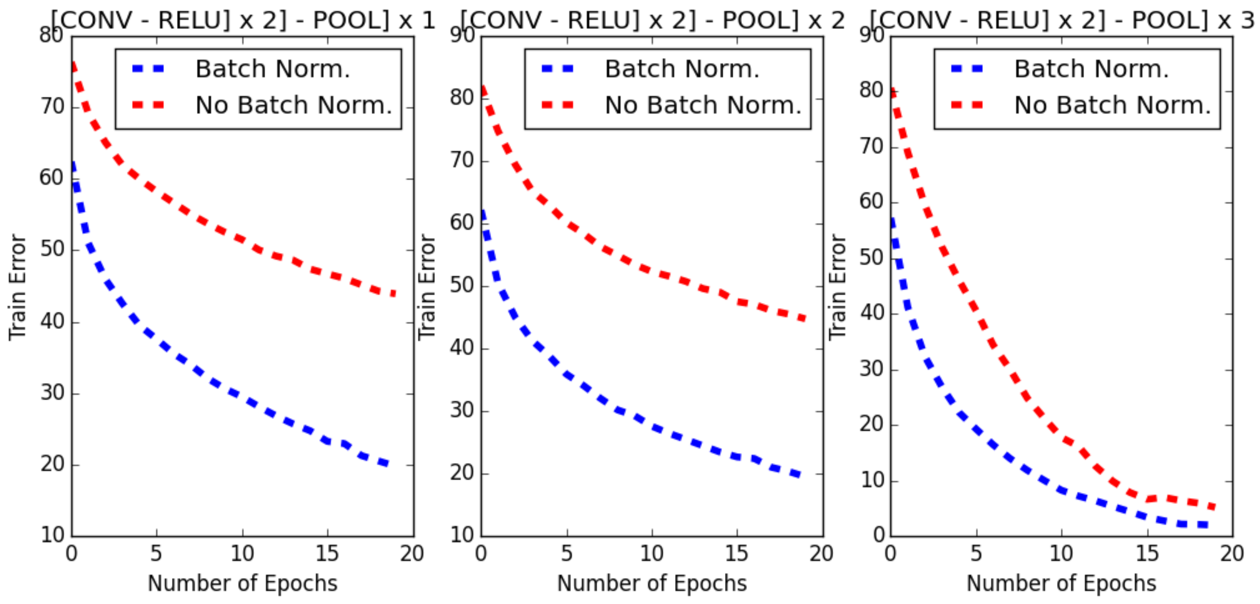

Figure 3: Training performance of various Architectures with and without Batch Normalization

**Observations**

- Batch Normalization has clearly improved the gradient flow through the network.
- It also acts a great regularizer and can even serve as a worthy substitute of *dropout* if need arises.
- It has the potential to reduce dependence on initialization and can allow higher learning rates.
- It is observed that there is a **30%** increase in runtime as a result of the introducing interLayer BatchNormalization.

## 3 Dropout

As we have discussed earlier, a standard CNN consists of alternating convolutional and pooling layers, with fully-connected layers on top. Compared to regular feed-forward networks with similarly-sized layers, CNNs have much fewer parameters and connections and are therefore less prone to overfitting. We train various CNN models by separately and simultaneously introducing dropout with max-pooling.

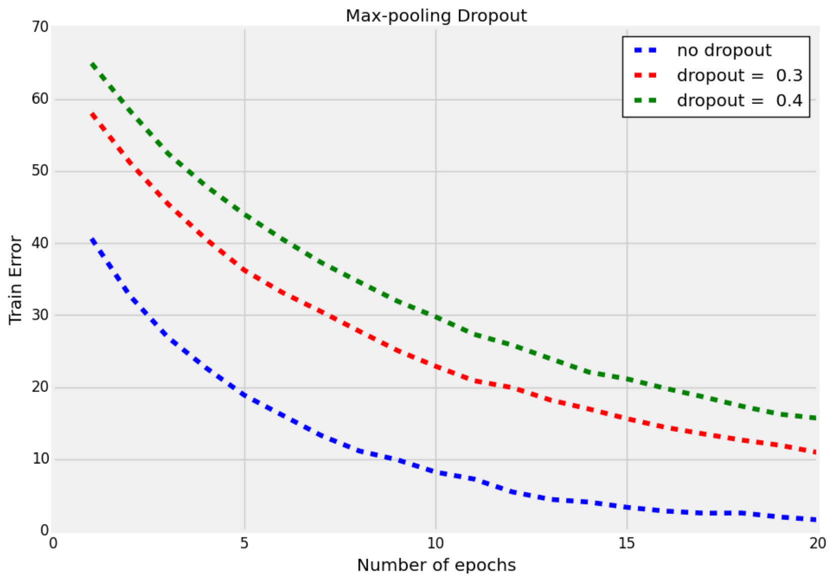

Figure 4: Comparison of Train errors for various Dropout probabilities

### Observations

- The retention probability $p$ is varied to observe the change in performance.
- There is visible increase in training error after having incorporated dropout.
- The main purpose of dropout is the regularization which can prevent overfitting – for a value of $p = 0.4$, we observe the model performs better than the one with dropout.
- No dropout converges faster than dropout but not necessarily to the global optimum.

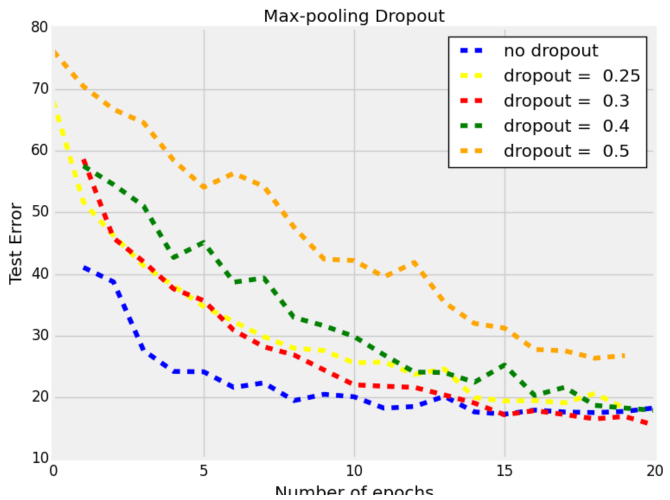

Figure 5: Comparison of Test errors for various Dropout probabilities

## 4 Pooling and Optimization Techniques

By reducing the spatial size of the representation, the pooling layers not only help reduce the number of parameters but also provide a form of translational invariance and help improve generalization. This can help reduce computational time between the successive convolutional layers. The introduction of pooling layers have been also shown to counter the effects of over-fitting. In order to better understand the effects of two popular

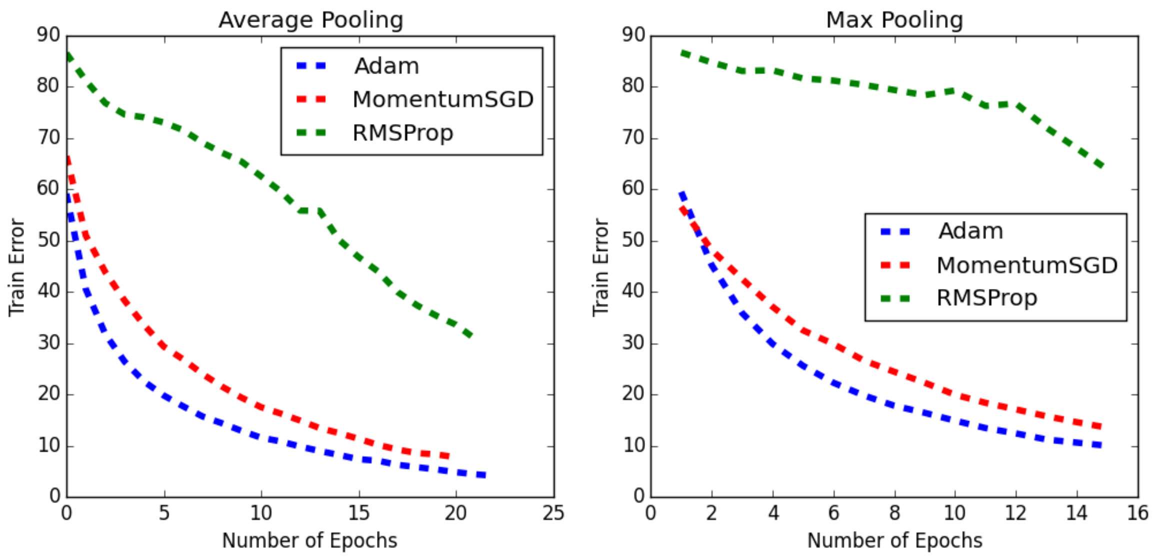

Figure 6: Comparison of Pooling with Optimizations

### Observation

- Adam is better than MomentumSGD and RMSprop
    - Average pooling works better than Max pooling for Adam
    - Adam outperforms the other optimizers both average and max pooling techniques.
- Max pooling works better for RMSprop and Momentum SGD
- Adam with average pooling is better than Momentum with max pooling
- Best combination : **Adam + Average Pooling**

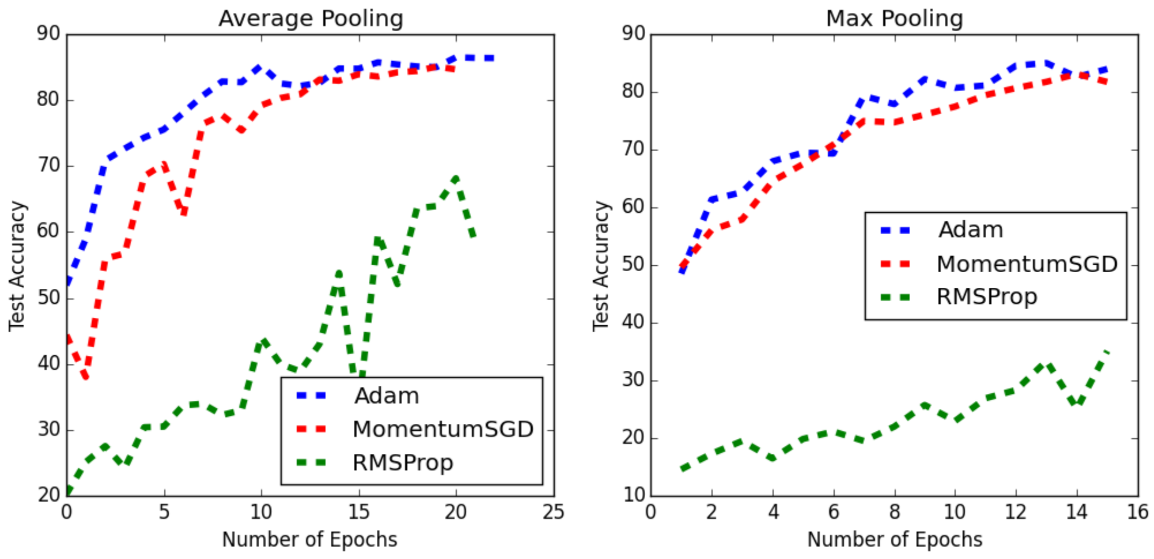

Figure 7: Comparison of Average-pooling Optimizations

## 5 Convolutional Layer

The convolutional layers should be employing filters of small size usign a stride of $S = 1$. Padding the input volume with zeros in such way that the convolutional layer does not alter the spatial dimensions of the input. We varied the filter size $F$ to $3x3$, $5x5$ and $7x7$ and observe the behavior of the architecture and the results are shown in the figure 10. Although, the performance of these architectures are similar during the training phase, larger filters tend to overfit.

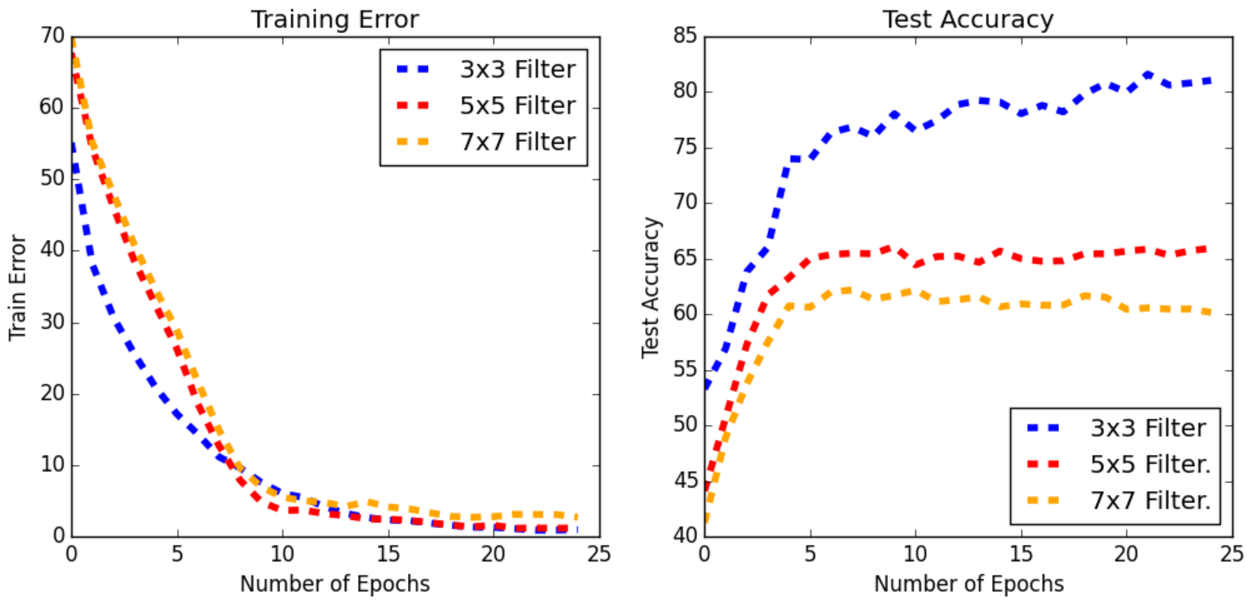

Figure 8: Comparison of Filter Sizes for the CONV layer

### Observations

- Smalller receptive size tend to preserve the spatial resolution
- By incorporating three 3 x 3 convolutional layers instead of a single 7 x 7 layer, the decision function has become more discriminative.
- It can also be shown that the 3 x 3 filter can decrease the number of parameters.

## 6 Data Augmentation

The primary motivation is the relatively small training set size. Moreover, if the training set is uded for validation the number of training samples is further reduced by 20%. In an effort to bolster the number of training samples and reduce variance, the data augmentation was implemented during the pre-processing stage although we were limited by the compute node time-out. Some of the techniques we explored were a random rotation within the 15-degree increment between 0 and 180 which could result in a twelve supplementary training images. We also implemented a mirror flip of the image as well as the median filtering technique.

```
import h5py
CIFAR10_data = h5py.File('CIFAR10.hdf5', 'r')
x_train = np.float32(CIFAR10_data['X_train'][:] )
y_train = np.int32(np.array(CIFAR10_data['Y_train'][:]))
print x_train.shape, y_train.shape
aug_X_train = []
aug_Y_train = []
for xt in  x_train:
        temp_image = np.rollaxis(xt, 0, 3)
        temp_image_flr = np.fliplr(temp_image)  # flip
        temp_image_con = (temp_image - temp_image.min()) /
                         (temp_image.max() - temp_image.min())  # normalize
        temp_image_med = ndimage.median_filter(temp_image, 2)  # median filter
        aug_X_train.append(xt) # original image
        aug_X_train.append(temp_image_flr)
        temp_image_flr = np.rollaxis(temp_image_flr, 2, 0)
        aug_X_train.append(temp_image_con)
        temp_image_con = np.rollaxis(temp_image_con, 2, 0)
        aug_X_train.append(temp_image_med)
        temp_image_med = np.rollaxis(temp_image_med, 2, 0)

for yt in y_train:
        aug_Y_train.append(yt)
        aug_Y_train.append(yt)
        aug_Y_train.append(yt)
        aug_Y_train.append(yt)
```

### Observations

- We observed a huge improvement in the training accuracy in the augmented dataset.
- Increase in training time.

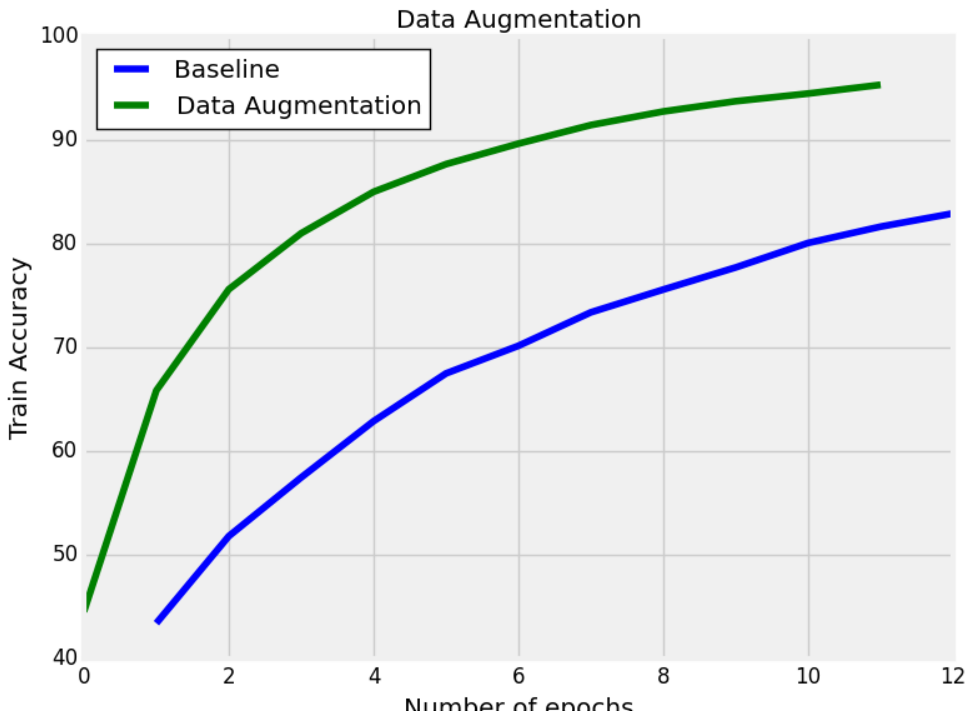

Figure 9: Comparison of model performance with and without 4x data augmentation

## 7 Best Model after Hyperparameter Tuning

After the above experiments, we can conclude that dropout and Interlayer batch normalization do infact act as regularizers and help prevent overfitting. The Adam optimizer seems to be the best choice over the others coupled with Average pooling. Data Augmentation certainly helps in the training process besides bolstering the dataset, it does its bit to reduce variance.

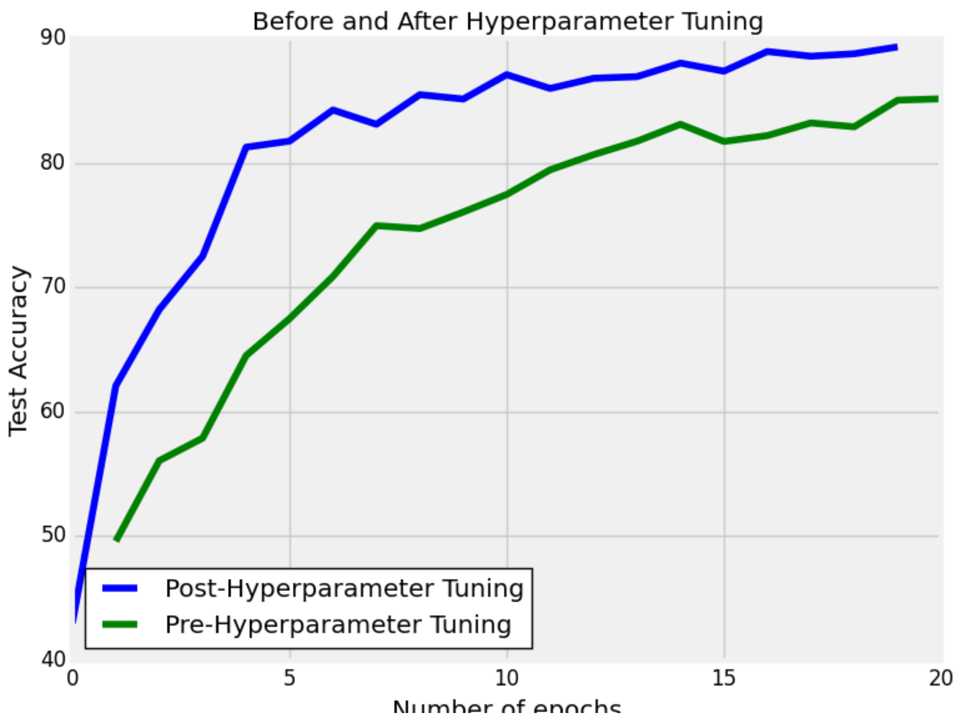

Figure 10: Comparison of model performance before and after Hyperparameter tuning

## 8 Code

```
import numpy as np
import time
from scipy import ndimage
import os

import chainer
from chainer import cuda, Function, gradient_check,
    Variable, optimizers, serializers, utils
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L

#load CIFAR10 data
import h5py
CIFAR10_data = h5py.File('CIFAR10.hdf5', 'r')
x_train = np.float32(CIFAR10_data['X_train'][:] )
y_train = np.int32(np.array(CIFAR10_data['Y_train'][:]))
print x_train.shape, y_train.shape
aug_X_train = []
aug_Y_train = []

#data augmentation
for xt in  x_train:
    temp_image = np.rollaxis(xt, 0, 3)
    temp_image_flr = np.fliplr(temp_image)  # flip
    temp_image_con = (temp_image - temp_image.min()) /
                    (temp_image.max() - temp_image.min())  # normalize
    temp_image_med = ndimage.median_filter(temp_image, 2)  # median filter

    temp_image_flr = np.rollaxis(temp_image_flr, 2, 0)
    temp_image_con = np.rollaxis(temp_image_con, 2, 0)
    temp_image_med = np.rollaxis(temp_image_med, 2, 0)

    aug_X_train.append(xt)
    aug_X_train.append(temp_image_flr)
    aug_X_train.append(temp_image_con)
    aug_X_train.append(temp_image_med)

for yt in y_train:
    aug_Y_train.append(yt)
    aug_Y_train.append(yt)
    aug_Y_train.append(yt)
    aug_Y_train.append(yt)

print len(aug_X_train[:])
a_X_train =  np.float32(aug_X_train[:])
a_Y_train =  np.int32(aug_Y_train[:])
x_train = a_X_train
y_train = a_Y_train
print x_train.shape, y_train.shape
x_test = np.float32(CIFAR10_data['X_test'][:] )
y_test = np.int32( np.array(CIFAR10_data['Y_test'][:]  ) )
```

```
CIFAR10_data.close()


#D = 32
num_outputs = 10


class Conv_NN(Chain):
    def __init__(self):
        super(Conv_NN, self).__init__(
            conv1_1=L.Convolution2D(3, 64, 3, pad=1),
            bn1_1=L.BatchNormalization(64),
            conv1_2=L.Convolution2D(64, 64, 3, pad=1),
            bn1_2=L.BatchNormalization(64),

            conv2_1=L.Convolution2D(64, 128, 3, pad=1),
            bn2_1=L.BatchNormalization(128),
            conv2_2=L.Convolution2D(128, 128, 3, pad=1),
            bn2_2=L.BatchNormalization(128),

            conv3_1=L.Convolution2D(128, 256, 3, pad=1),
            bn3_1=L.BatchNormalization(256),
            conv3_2=L.Convolution2D(256, 256, 3, pad=1),
            bn3_2=L.BatchNormalization(256),
            conv3_3=L.Convolution2D(256, 256, 3, pad=1),
            bn3_3=L.BatchNormalization(256),
            conv3_4=L.Convolution2D(256, 256, 3, pad=1),
            bn3_4=L.BatchNormalization(256),
            #Fully connected layer
            fc4 = L.Linear(4*4*256, 500),
            fc5 = L.Linear(500, 500),
            fc6 = L.Linear(500,10),
        )
    def __call__(self, x_data, y_data, dropout_bool, bn_bool, p):
        x = Variable(x_data)
        t = Variable(y_data)
        h = F.relu(self.bn1_1(self.conv1_1(x), bn_bool))
        h = F.relu(self.bn1_2(self.conv1_2(h), bn_bool))
        h = F.max_pooling_2d(h, 2, 2)
        h = F.dropout(h, p, dropout_bool)

        h = F.relu(self.bn2_1(self.conv2_1(h), bn_bool))
        h = F.relu(self.bn2_2(self.conv2_2(h), bn_bool))
        h = F.max_pooling_2d(h, 2, 2)
        h = F.dropout(h, p, dropout_bool)

        h = F.relu(self.bn3_1(self.conv3_1(h), bn_bool))
        h = F.relu(self.bn3_2(self.conv3_2(h), bn_bool))
        h = F.relu(self.bn3_3(self.conv3_3(h), bn_bool))
        h = F.relu(self.bn3_4(self.conv3_4(h), bn_bool))
        h = F.max_pooling_2d(h, 2, 2)
        h = F.dropout(h, p, dropout_bool)

        h = F.dropout(F.relu(self.fc4(h)), p, dropout_bool)
        h = F.dropout(F.relu(self.fc5(h)), p, dropout_bool)
        h = self.fc6(h)
        L_out = h
        return F.softmax_cross_entropy(L_out, t), F.accuracy(L_out, t)

#returns test accuracy of the model.
def Calculate_Test_Accuracy(x_test, y_test, model, p, GPU_on, batch_size):
    L_Y_test = len(y_test)
    counter = 0
    test_accuracy_total = 0.0
    for i in range(0, L_Y_test, batch_size):
        if (GPU_on):
            x_batch = cuda.to_gpu(x_test[i:i+ batch_size,:])
            y_batch = cuda.to_gpu(y_test[i:i+ batch_size] )
        else:
            x_batch = x_test[i:i+batch_size,:]
            y_batch = y_test[i:i+batch_size]
        dropout_bool = False
        bn_bool = True
        loss, accuracy = model(x_batch, y_batch, dropout_bool,bn_bool, p)
        test_accuracy_batch  = 100.0*np.float(accuracy.data )
        test_accuracy_total += test_accuracy_batch
        counter += 1
    test_accuracy = test_accuracy_total/(np.float(counter))
    return test_accuracy


model =  Conv_NN()

#True if training with GPU, False if training with CPU
GPU_on = True
```

```
#size of minibatches
batch_size = 500

#transfer model to GPU
if (GPU_on):
    model.to_gpu()

#optimization method
optimizer = optimizers.Adam(alpha=0.001,beta1=0.9,beta2=0.999,eps=1e-08)
optimizer.setup(model)


#learning rate
#optimizer.lr = .01

#dropout probability
p = .40

#number of training epochs
num_epochs = 400

L_Y_train = len(y_train)

time1 = time.time()
for epoch in range(num_epochs):
    #reshuffle dataset
    I_permutation = np.random.permutation(L_Y_train)
    x_train = x_train[I_permutation,:]
    y_train = y_train[I_permutation]
    epoch_accuracy = 0.0
    batch_counter = 0
    for i in range(0, L_Y_train, batch_size):
        if (GPU_on):
            x_batch = cuda.to_gpu(x_train[i:i+batch_size,:])
            y_batch = cuda.to_gpu(y_train[i:i+batch_size] )
        else:
            x_batch = x_train[i:i+batch_size,:]
            y_batch = y_train[i:i+batch_size]
        model.zerograds()
        dropout_bool = True
        bn_bool = False
        loss, accuracy = model(x_batch, y_batch, dropout_bool, bn_bool, p)
        loss.backward()
        optimizer.update()
        epoch_accuracy += np.float(accuracy.data)
        batch_counter += 1
    if (epoch % 1 == 0):
        train_accuracy = 100.0*epoch_accuracy/np.float(batch_counter)
        test_accuracy = Calculate_Test_Accuracy(x_test, y_test, model, p, GPU_on
    print 'Epoch', epoch, ',Train accuracy ', train_accuracy, ',Test accuracy',


time2 = time.time()
training_time = time2 - time1
print "Rank: %d" % rank
print "Training time: %f" % training_time
```

## References

[1] Andrej Karpathy. CS231n. http://cs231n.github.io/convolutional-networks/#pool, 2015.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[3] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.