Unfair Back Propagation with Tensorflow [Manual Back Propagation with TF]

Jae Duk Seo Follow
Jun 20, 2018 · 10 min read

GIF from this website

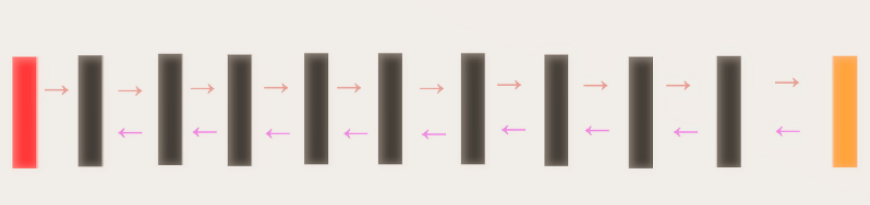
I have been giving a thought about back propagation, and in traditional neural network it seems like we are always linearly performing feed forward operation and back propagation. (As in 1:1 ratio) But I thought to myself, we don't really have to do that. So I wanted to do some experiments.

Case a) Back Propagation (No Data Augmentation)
Case b) Back Propagation (Data Augmentation)
Case c) Unfair Back Prop (From the Back) (No Data Augmentation)
Case d) Unfair Back Prop (From the Back) (Data Augmentation)
Case e) Unfair Back Prop (From the Front) (No Data Augmentation)
Case f) Unfair Back Prop (From the Front) (Data Augmentation)

> ***Please note that this post is for fun and to express my creativity. Hence not performance oriented.***

. . .

Network Architecture / Data Set / Typical Back Propagation

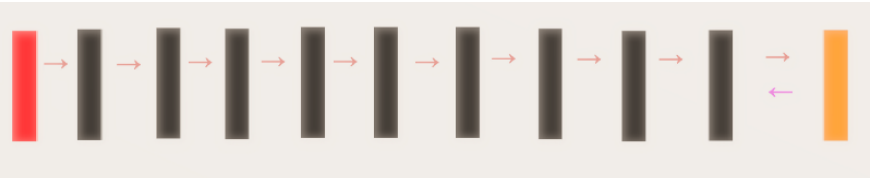


Red Square → Input Image Batch
Black Square → Convolution Layer with/without Mean Pooling
Orange Square → Softmax Layer for Classification
Orange Arrow → Direction of Feed Forward Operation
Purple Arrow → Direction of Back Propagation.

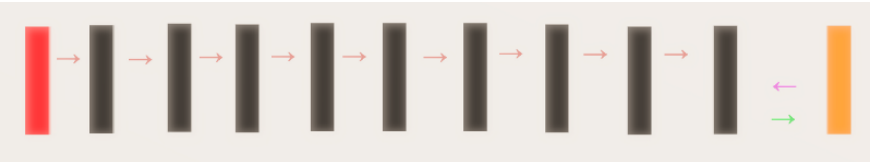
As seen above, we are going to use the base network from my old post "All Convolutional Net" which is just a 9 layered network only composing of convolution operation. Additionally we are going to use the CIFAR 10 Data Set.

. . .

Unfair Back Propagation

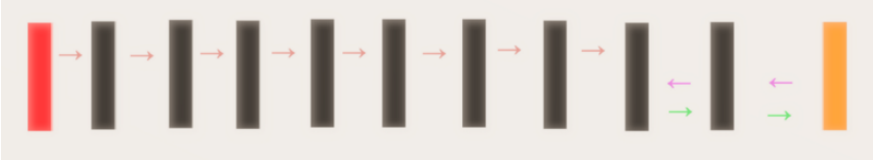


Now to explain the concept of unfair back propagation, lets first assume that we just have finished our feed forward operation (as seen above). And by looking at the pink arrow we can already tell that we have back propagated to the 9'th layer.

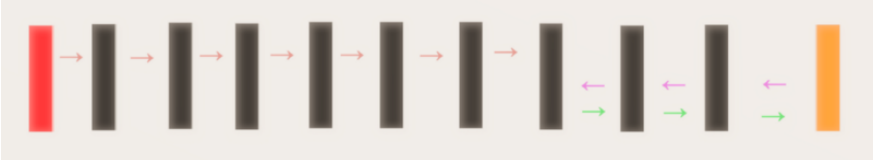


Now rather than continuing on to back propagate to the eight layer and so on, we can again perform feed forward operation to get another

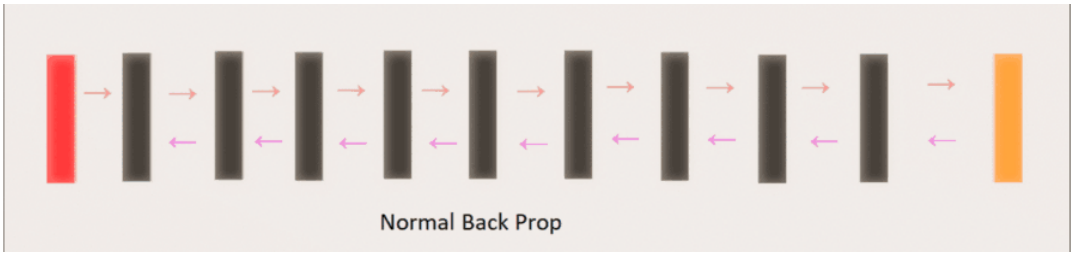
round of classification. (But this time the 9'th layer will already have updated it's weights once.)



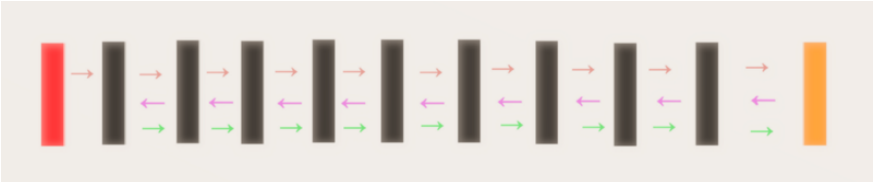
We can repeat that process again, however this time we are going to update the weights of the 8'th layer as well.



And we can follow that concept again and again. Until we reach the 1'st layer finishing our back propagation as a whole. In conclusion we are being unfair to the beginning layers of our network, since the latter portions are going to be updated more. We can see difference in a GIF format as well.

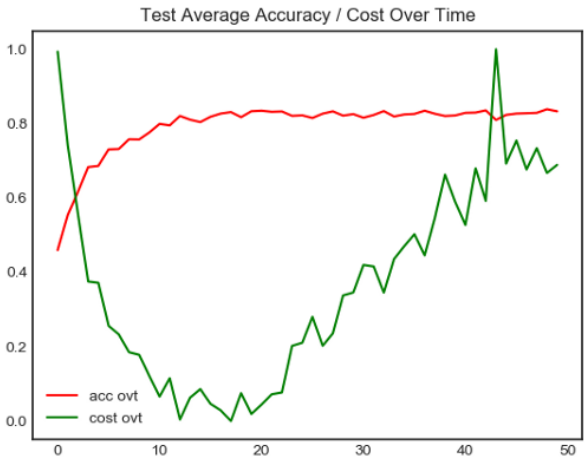


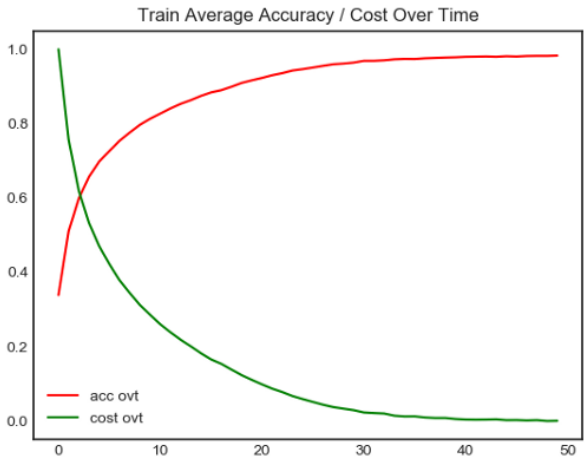
Finally, we can take the concept of being unfair to the beginning portion of our network and flip it, being unfair to the latter portion of our network. (Other words updating the beginning portion of our network more than the latter portions.)



. . .

Results: Case a) Back Propagation (No Data Augmentation) (50 Epoch)



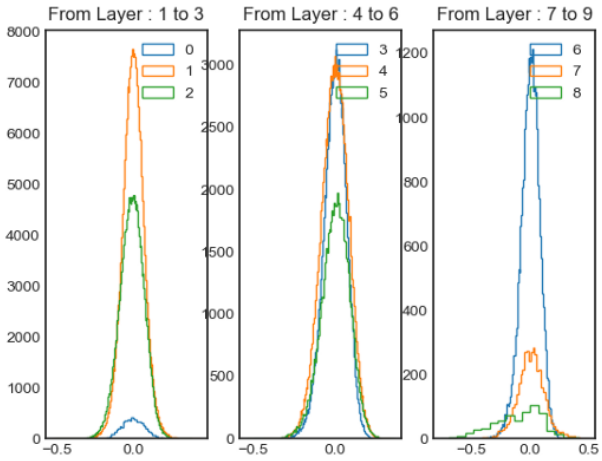


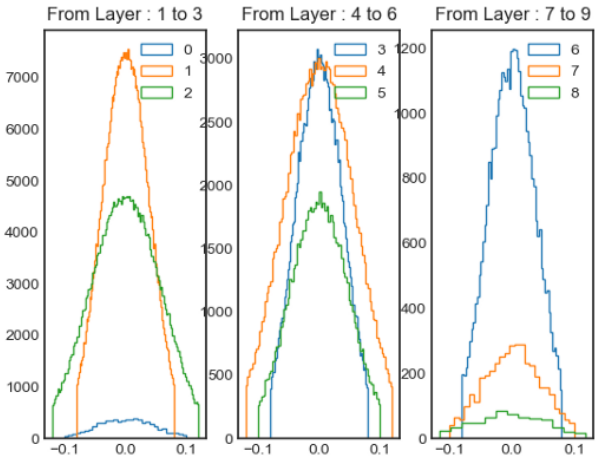
Left Image → Accuracy / Cost for Test Images Over Time
Right Image → Accuracy / Cost for Train Images Over Time

As seen above the network's accuracy for testing image have begin to stagnated around 83 percent. However the thing to note here is the fact that the network needed 13 epoch for the accuracy on testing images to reach over 80 percent.

```
----------
Current Iter :  49  current batch:  9990  Current cost:  0.20025873  Current Acc:  0.90.00urrent Acc:  1.00
---------- Learning Rate :  0.0001
 Using grad:  0 Train Current cost:  0.054802644872432076  Current Acc:  0.983099996817112
Test Current cost:  1.179542318739433  Current Acc:  0.833299995303154
----------
```

The final accuracy was 98 percent on training images while 83 percent on testing images indicating that the network is suffering from over-fitting.



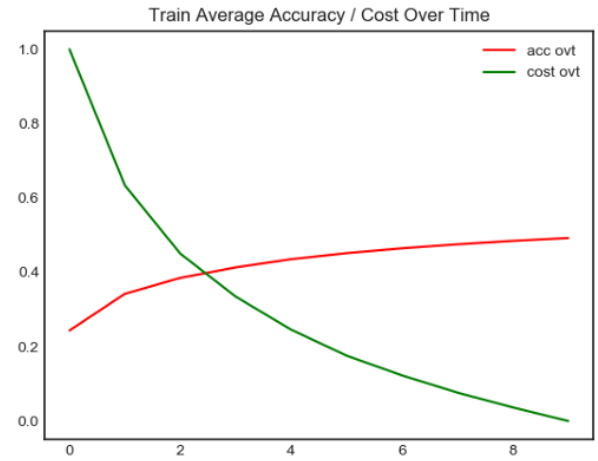


Left Image → Histogram of Weights After training
Right Image → Histogram of Weights Before training

And as seen above, all of the weight range have increased from -0.1/0.1 to -0.5/0.5.

. . .

Results: Case b) Back Propagation (Data Augmentation) (10 Epoch)

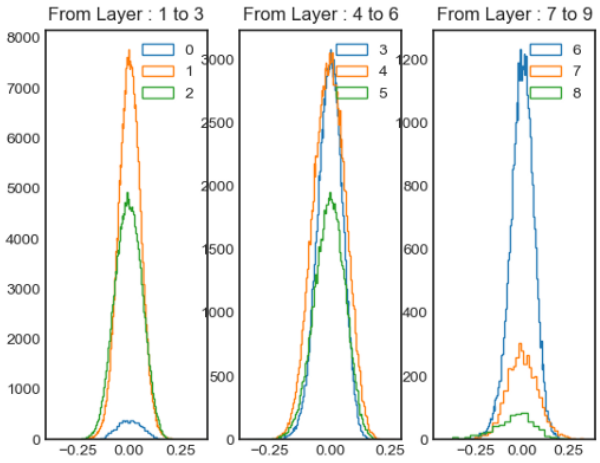


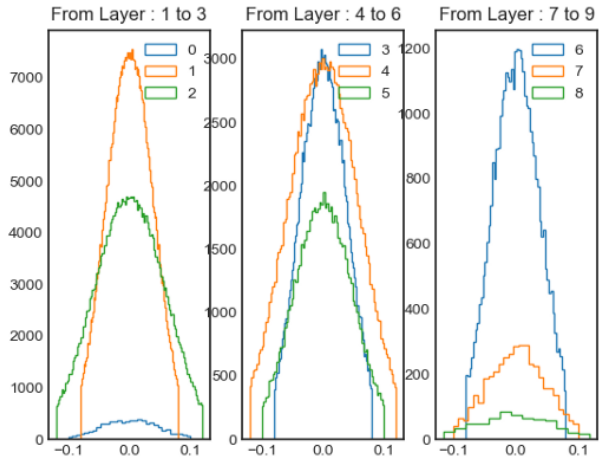
Left Image → Accuracy / Cost for Test Images Over Time
Right Image → Accuracy / Cost for Train Images Over Time

Now since we performed data augmentation we can clearly observe the decrease in accuracy on training images (Since there are much more variance in the data.).

```
Current Iter :  9  current batch:  9990  Current cost:  0.6941143  Current Acc:  0.790rrent Acc:  0.56
---------- Learning Rate :  0.0001
 Using grad:  0 Train Current cost:  1.3512327820062637  Current Acc:  0.4924500071160495
Test Current cost:  0.6253935259757564  Current Acc:  0.7951999967992306
----------
```

As seen above, after training 10 epoch the network ended up at 79 percent accuracy (on testing images). Thankfully, the network is not suffering from over-fitting.



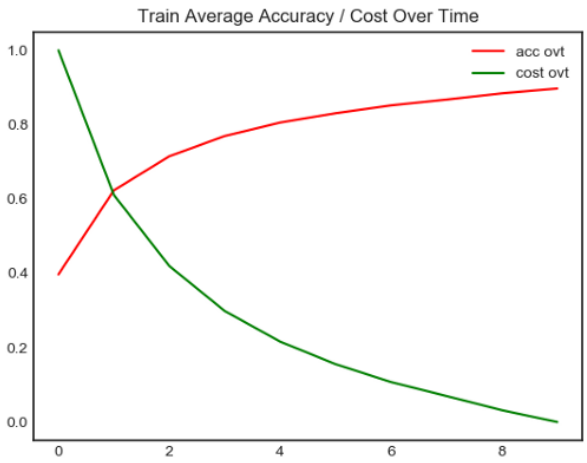


Left Image → Histogram of Weights After training
Right Image → Histogram of Weights Before training

Like the case when we did not perform data augmentation, the range of the weights have increased.

. . .

Results: Case c) Unfair Back Prop (From the Back) (No Data Augmentation) (10 Epoch)

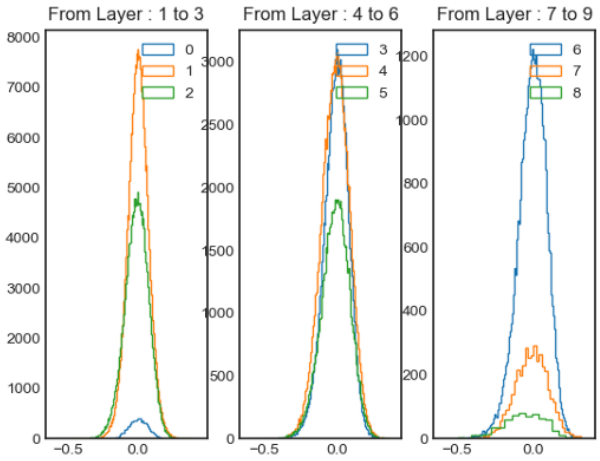


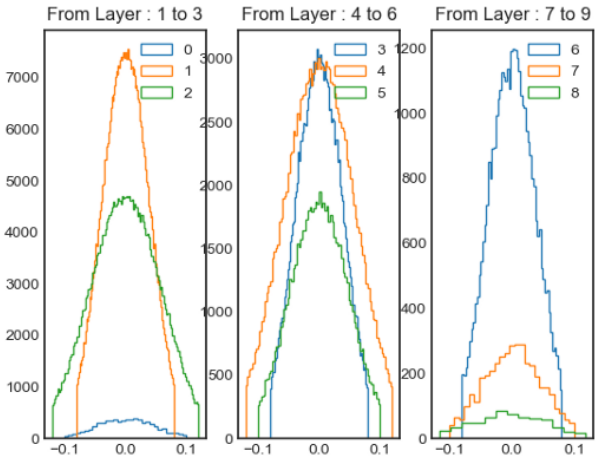
Left Image → Accuracy / Cost for Test Images Over Time
Right Image → Accuracy / Cost for Train Images Over Time

The network did much better when compared to the one that was trained on normal (typical) back propagation method. Finishing the training in the accuracy about 82 percent on the testing images while 89 on training images. Indicating there is still an over-fitting, but not so much when compared to case a).

```
45  ----------
46  Current Iter :  9  current batch:  9990  Current cost:  1.7374566  Current Acc:  0.8Current Acc:  0.8
47  ---------- Learning Rate :  0.0001
48   Using grad:  1 Train Current cost:  1.5921240738630296  Current Acc:  0.8971199930548668
49  Test Current cost:  1.66096774995327  Current Acc:  0.8297999957799912
50  ----------
```

It is quite surprising to observe the fact that unfair back propagation seems to have some kind of regularization effect.



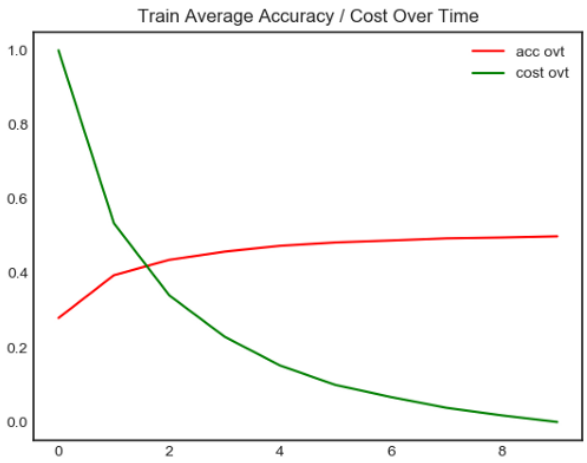


Left Image → Histogram of Weights After training
Right Image → Histogram of Weights Before training

By far the most interesting observation was the histogram of the weights. As seen above, when we focus our attention to the generated histogram of the final layers of the network we can observe that the histogram is skewed to one side. (-0.5 in the left range while there is no 0.5 on the right.) Indicating that the symmetry of distribution is somewhat hindered. Same argument goes for the histogram of the middle layers.

. . .

Results: Case d) Unfair Back Prop (From the Back) (Data Augmentation) (10 Epoch)

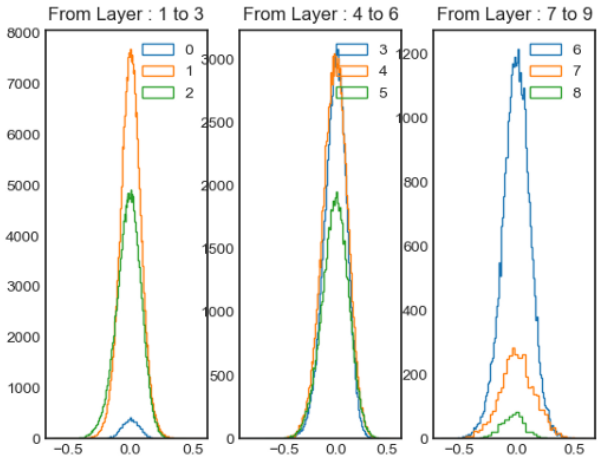


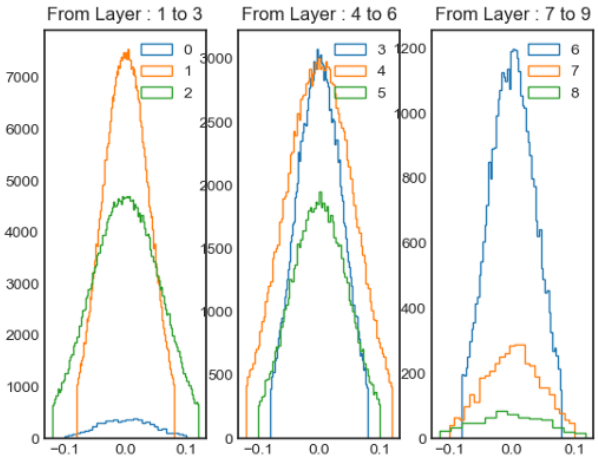
Left Image → Accuracy / Cost for Test Images Over Time
Right Image → Accuracy / Cost for Train Images Over Time

Again, when we perform data augmentation we can observe the decrease in the accuracy for training images.

```
----------
Current Iter :  9  current batch:  9990  Current cost:  1.6392205  Current Acc:  0.8Current Acc:  0.5
---------- Learning Rate :  0.0001
 Using grad:  1 Train Current cost:  1.9620398842453957  Current Acc:  0.49974000701382754
Test Current cost:  1.6682313923835754  Current Acc:  0.8130999964773655
----------
```

And when comparing the results from case b) (Normal Back prop for 10 epoch) we can observe the fact that accuracy on training images are similar with one another (49 percent). However, for whatever reason the network trained by unfair back prop had a higher accuracy on testing images.



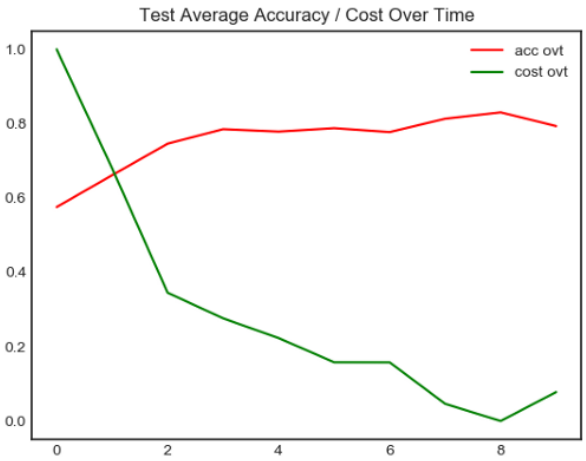


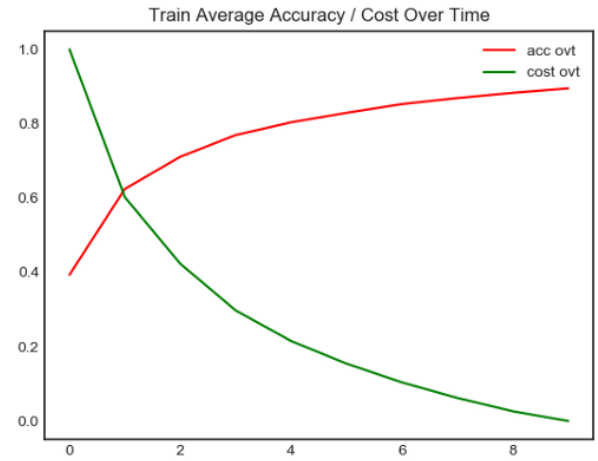
Left Image → Histogram of Weights After training
Right Image → Histogram of Weights Before training

Unlike case c) we can observe some symmetry of the distribution of the weights for this case.

. . .

Results: Case e) Unfair Back Prop (From the Front) (No Data Augmentation) (10 Epoch)



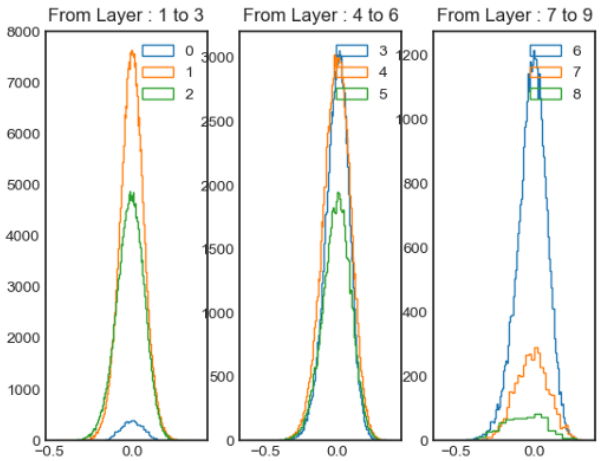


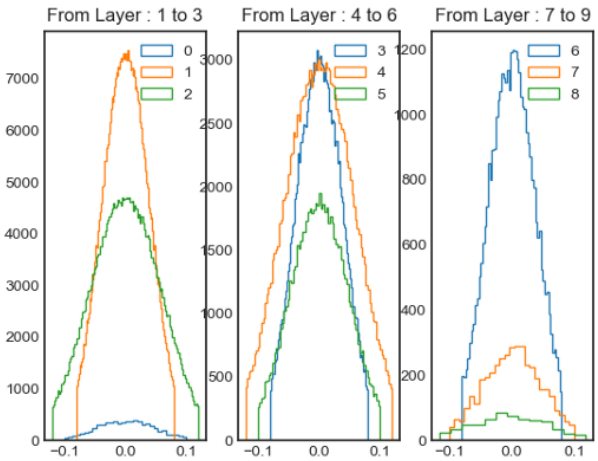
Left Image → Accuracy / Cost for Test Images Over Time
Right Image → Accuracy / Cost for Train Images Over Time

Finally, I wanted to see if there is a difference when we are being unfair to the latter portion of our network. And from the looks of it, it is better to be unfair to the beginning portion of our network rather than the latter portion.

```
----------
Current Iter :  9  current batch:  9990  Current cost:  1.6355925  Current Acc:  0.9Current Acc:  0.8
---------- Learning Rate :  0.0001
 Using grad:  1 Train Current cost:  1.5949978121757507  Current Acc:  0.8948399926304818
Test Current cost:  1.6873440403938293  Current Acc:  0.7933999973237514
----------
```

One reason why I say that is because of the final accuracy on testing images. We can observe that the accuracy on training images are similar but this network had a lower accuracy on testing images.



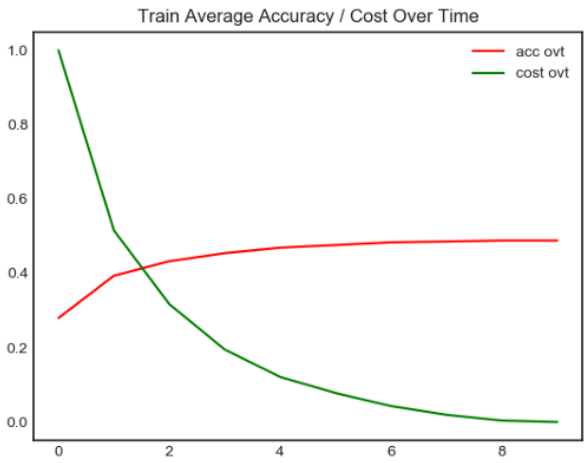


Left Image → Histogram of Weights After training
Right Image → Histogram of Weights Before training

Interestingly, the distribution of the weights seems to be similar no matter we are being unfair to the beginning or latter portion of our network.

. . .

Results: Case f) Unfair Back Prop (From the Front) (Data Augmentation) (10 Epoch)

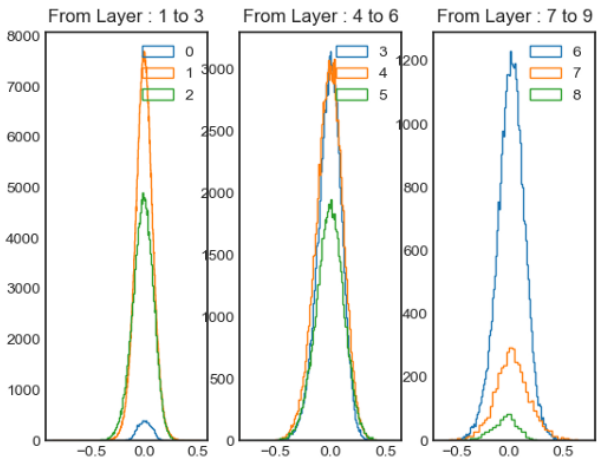


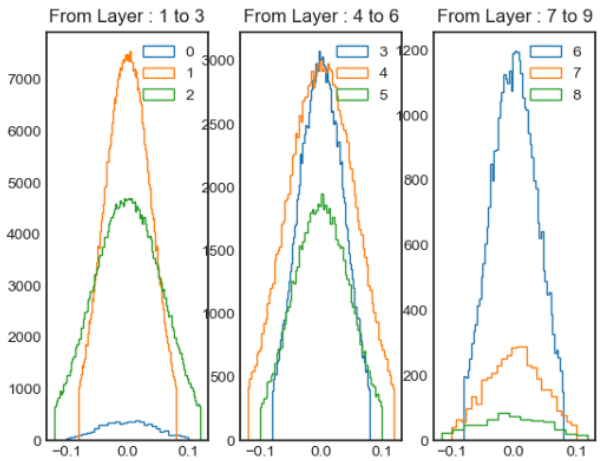
Left Image → Accuracy / Cost for Test Images Over Time
Right Image → Accuracy / Cost for Train Images Over Time

Finally, I performed data augmentation while training the same network from case e.

```
Current Iter :  9  current batch:  9990  Current cost:  1.5754852  Current Acc:  0.9urrent Acc:  0.66
---------- Learning Rate :  0.0001
 Using grad:  1 Train Current cost:  1.9745294085383416  Current Acc:  0.4882200069397688
Test Current cost:  1.701466210126877  Current Acc:  0.7876999979317189
----------
```

And as seen above the lack of increase in accuracy can be observed here as well.





Left Image → Histogram of Weights After training
Right Image → Histogram of Weights Before training

Similar results for the distribution of the weights as well. In a normal CNN, we know that latter layers capture higher level features. I suspect that it is better for the network to train harder? capturing those higher level of features.

. . .

Interactive Code

```
import imgaug as ia

plt.style.use('seaborn-white')
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
np.random.seed(678)
tf.set_random_seed(678)
ia.seed(678)

def tf_elu(x): return tf.nn.elu(x)
def d_tf_elu(x): return tf.cast(tf.greater(x,0),tf.float32)  +  \
(tf_elu(tf.cast(tf.less_equal(x,0),tf.float32) * x)+1.0)

def tf_softmax(x): return tf.nn.softmax(x)
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

# code from: https://github.com/tensorflow/tensorflow/issues/8246
def tf_repeat(tensor, repeats):
    """
    Args:

    input: A Tensor. 1-D or higher.
    repeats: A list. Number of repeat for each dimension, length must be the same as the number of dimensions in input

    Returns:

    A Tensor. Has the same type as input. Has the shape of tensor.shape * repeats
    """
    expanded_tensor = tf.expand_dims(tensor, -1)
    multiples = [1] + repeats
    tiled_tensor = tf.tile(expanded_tensor, multiples = multiples)
    repeated_tesnor = tf.reshape(tiled_tensor, tf.shape(tensor) * repeats)
    return repeated_tesnor

# ================ VIZ ================
def show_hist_of_weigt(all_weight_list,status='before'):
    fig = plt.figure()
    weight_index = 0

    for i in range(1,1+int(len(all_weight_list)/2)):
```

For Google Colab, you would need a google account to view the codes, also you can't run read only scripts in Google Colab so make a copy on your play ground. Finally, I will never ask for permission to access your files on Google Drive, just FYI. Happy Coding! Also for transparency I uploaded all of the training logs on my github.

To access the code for case a please click here, for the logs click here.
To access the code for case b please click here, for the logs click here.
To access the code for case c please click here, for the logs click here.
To access the code for case d please click here, for the logs click here.
To access the code for case e please click here, for the logs click here.
To access the code for case f please click here, for the logs click here.

. . .

Final Words