Anthony DiSpirito
ECE 590 – Comp. Eng. M.L. and D.N.Ns
Dr. Yiran Chen
10/2/19

Lab 2: Construct, Train, and Optimize Neural Network Models

NOTE: Code and Documentation can also be found at github.com under my github account name “axd465” and my repo “ECE590-CompEngML-DL-Lab2”

Link: **https://github.com/axd465/ECE590-CompEngML-DL-Lab2**

Assignment 1:

a)

i. In order to calculate the number of parameters, I first found the parameters in each layer of the network and added those together.

- Layer 1 (Input Layer): This layer has no parameters attributed to it so parameters = 0
- Layer 2 (First Conv Layer): parameters = (((kernel size)^2 * input channels) + bias) * output channels = (((5 x 5) x 3) + 1) x 6 = 456 parameters
- Layer 3 (First Max Pool): Doesn’t have any parameter contribution, so parameters = 0
- Layer 4 (Second Conv Layer): parameters = (((kernel size)^2 * input channels) + bias) * output channels = (((5 x 5) x 6) + 1) x 16 = 2416 parameters
- Layer 5 (Second Max Pool): Doesn’t have any parameter contribution, so parameters = 0
- Layer 6 (First Fully Connected layer): parameters = (inputs) x (outputs) = (5 x 5 x 16 + 1) x (120) = 48,120 parameters
- Layer 7 (Second Fully Connected layer): parameters = (inputs) x (outputs) = (120 + 1) x (84) = 10,164 parameters
- Layer 8 (Third Fully Connected layer): parameters = (inputs) x (outputs) = (84 + 1) x (10) = 850 parameters

- Total parameters = 0 + 456 + 0 + 2416 + 0 + 48,120 + 10,164 + 850 = 62,006 parameters total for the modified LeNet 5 model

ii. In order to calculate the number of MACs, I first found the MACs in each layer of the network and added those together.

- Layer 1 (Input Layer): This layer has no MACs attributed to it so MACs = 0
- Layer 2 (First Conv Layer): MACs = (5 x 5 x 3) x (28 x 28 x 6) = 352,800 MACs
- Layer 3 (First Max Pool): Doesn't have any MAC contribution, so MACs = 0
- Layer 4 (Second Conv Layer): MACs = (5 x 5 x 6) x (10 x 10 x 16) = 240,000 MACs
- Layer 5 (Second Max Pool): Doesn't have any MAC contribution, so MACs = 0
- Layer 6 (First Fully Connected layer): MACs = (5 x 5 x 16) x (120) = 48,000 MACs
- Layer 7 (Second Fully Connected layer): MACs = (120) x (84) = 10,080 MACs
- Layer 8 (Third Fully Connected layer): MACs = (84) x (10) = 840 MACs
- Total MACs = 0 + 352,800 + 0 + 240,000 + 0 + 48,000 + 10,080 + 840 = 651,720 MACs total for the modified LeNet 5 model

Assignment 2:

a) See attached code for modified LeNet 5 implementation
b) For the preprocessing, I used normalization (Normalize() transform and RGB mean), standardization (Normalize() transform and RGB std) and batch shuffling (shuffle = True in trainloader). In addition I implemented some randomized data augmentation where I have a series of data augmentation transforms whose order are randomized for each batch that include vertical flip (with some probability), horizontal flip (with some probability), random crop, random affine transform (without shear), random color shift, as well as random erasing. The probabilities used where found using trial and error as well as some

intuition into how I want the variance of my training data to be augmented. See the exact code below.

```
# Specify preprocessing function.
# Reference mean/std value
mean_RGB = (0.4914, 0.4822, 0.4465)
std_RGB = (0.2023, 0.1994, 0.2010)
input_img_size = (32,32)
percent_crop = 0.80 # 80 percent left after crop
data_augment_list = [#transforms.RandomResizedCrop(size = input_img_size, scale=(percent_crop, 1.0), ratio=(0.75, 1.3333333333333333), interpolation=2),
                     transforms.RandomVerticalFlip(p = 0.10),
                     transforms.RandomHorizontalFlip(p = 0.15),
                     transforms.RandomApply([transforms.RandomCrop(size = input_img_size, padding=4, pad_if_needed=False, fill=0, padding_mode='constant')], p = 0.15),
                     transforms.RandomApply([transforms.RandomAffine(degrees = 45, translate=(0.15,0.15), scale=None, shear=None, resample=False, fillcolor=0)], p = 0.20),
                     transforms.RandomApply([transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.05)], p = 0.15)]
transform_train = transforms.Compose([transforms.Resize(input_img_size),
                                      transforms.RandomOrder(data_augment_list),
                                      transforms.ToTensor(),
                                      transforms.RandomErasing(p=0.10, scale=(0.01, 0.15), ratio=(0.3, 3.3), value=0, inplace=False),
                                      transforms.Normalize(mean_RGB, std_RGB)])
transform_val = transforms.Compose([transforms.Resize(input_img_size),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean_RGB, std_RGB)])
```

c) For this section, as visible in the attached code, I used the cross entropy loss function with the stochastic gradient descent optimizer with Nesterov momentum (didn't want to overshoot global minimum). I used the recommended starting values for initial learning rate and momentum.

Assignment 3:

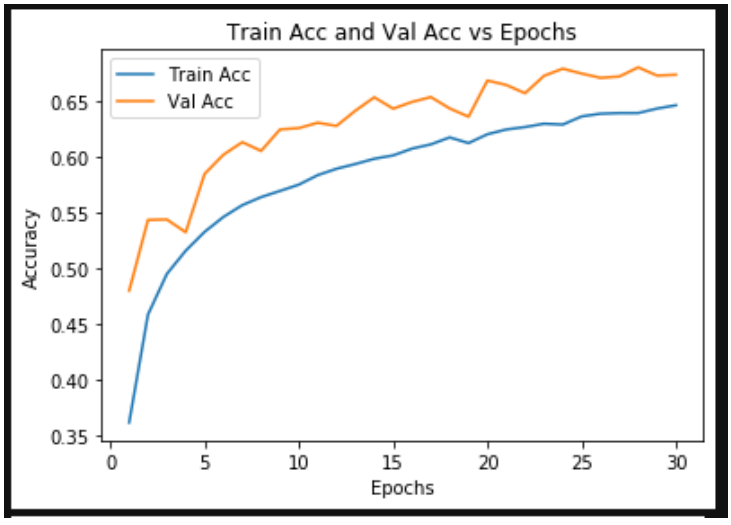
a) As seen by the screen shot below, my initial loss value right after the first pass and before training was 2.3134 (As seen below). This is the error on my training set. It seems reasonable for the first forward pass (as lecture slides indicate 2.3 is typical). After adding some data augmentation and batch normalization, my model came very close to 65% training accuracy (at 64.52) after 30 epochs (as shown below). My validation accuracy exceeded 65% after 30 epochs. This was before any modification to the LeNet 5 design structure and the hyperparameters.

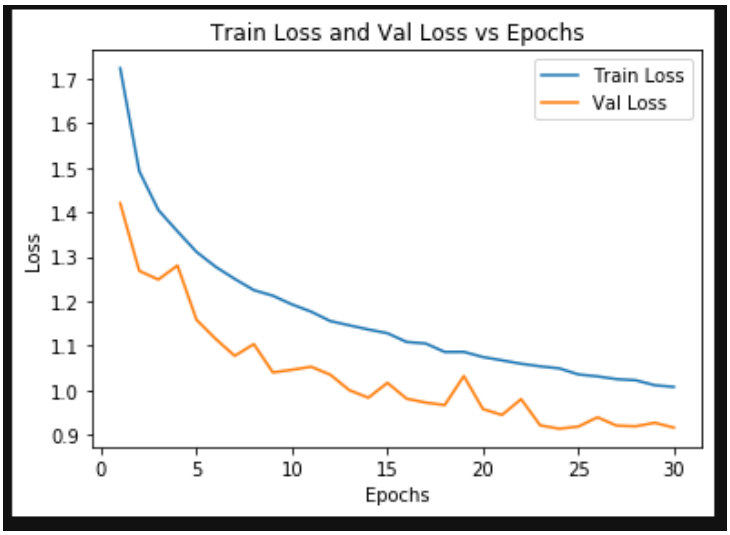
```
Initial Forward Loss: 2.3134
```

```
Epoch 29:
704
Training loss: 1.0050, Training accuracy: 64.5200 %
2019-10-02 19:41:11.410441
Validation...
Validation loss: 0.9221, Validation accuracy: 67.7000 %
Optimization finished.
```

b) By implementing data augmentation, what I am doing is increasing the variance of my training data and trying to get the model to look at more general features, rather than specific image values and colors. As seen below, with data augmentation I achieved a validation accuracy of 66.88% after 30 epochs, while I was only able to achieve a validation accuracy of 64.04% after 30 epochs. In addition to having a worse validation accuracy without data augmentation, there was also a large gap that formed between the training accuracy and validation accuracy. I see this as the model not focusing enough on generalizable features inherent to the classes. In some ways, the data augmentation adds noise and makes training focus more on these generalizable features that improve both training and validation accuracies at the same time.

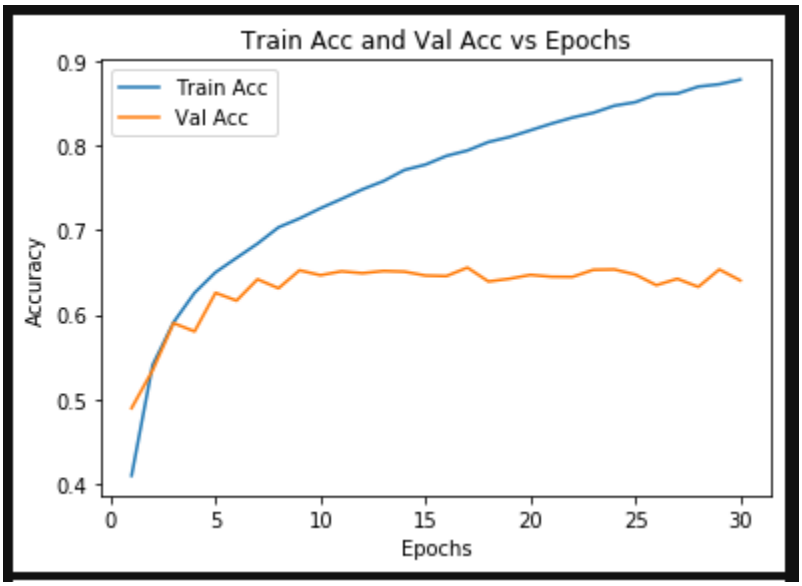
- **With Augmentation:**

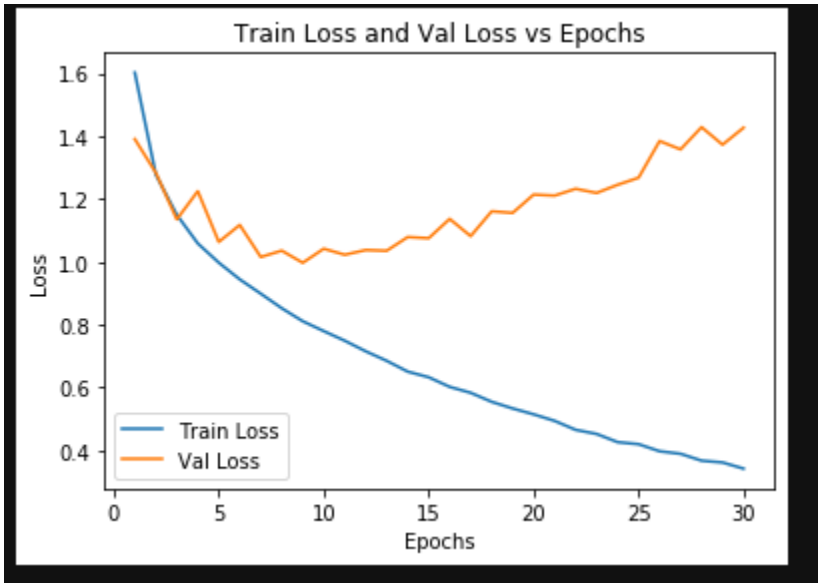





```
Epoch 29:
704
Training loss: 1.0072, Training accuracy: 64.7889 %
2019-10-02 20:10:21.175172
Validation...
Validation loss: 0.9238, Validation accuracy: 67.9600 %
Optimization finished.
```

- **<u>Without Augmentation:</u>**

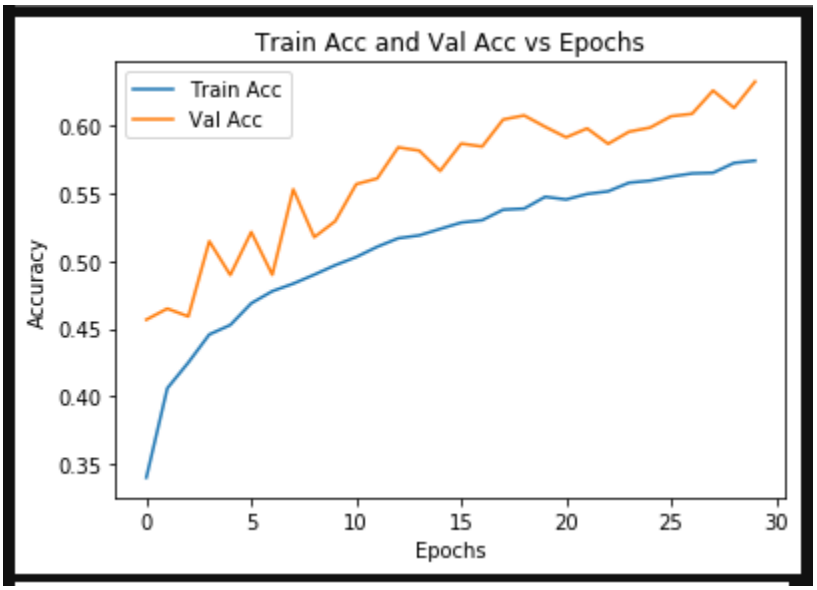


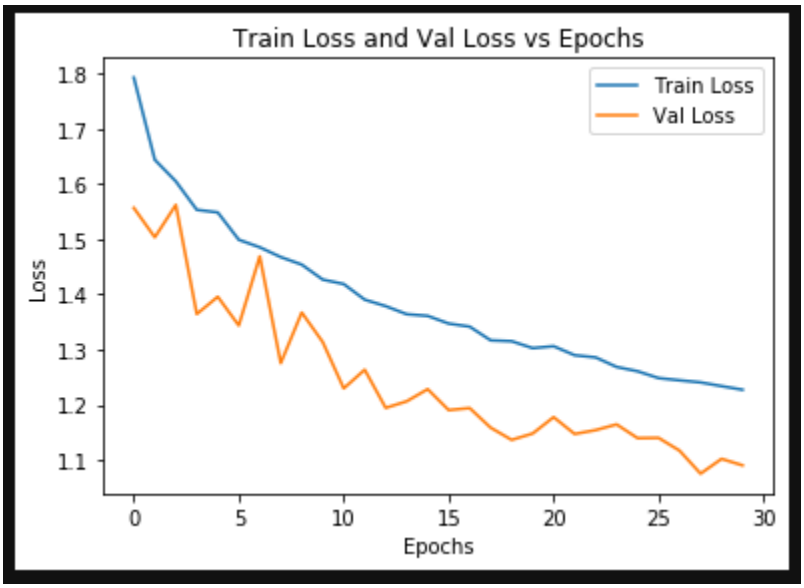



```
Epoch 29:
704
Training loss: 0.3302, Training accuracy: 88.1733 %
2019-10-02 20:21:53.034290
Validation...
Validation loss: 1.4729, Validation accuracy: 64.0400 %
Optimization finished.
```

c) I increased the initial learning rate from 0.01 to 0.10. With batch normalization, the model reached a validation accuracy of 63.26 % (as seen below) after 30 epochs. Without batch normalization, my validation accuracy was 10.00 % after 30 epochs. In addition, I noticed the model had more oscillatory behavior with batch normalization when the learning rate was increased. The model dropped right to 10% without batch normalization and stayed there. The training was going smoothly for the batch normalization model, but without batch normalization, the model got stuck outside of minimum.

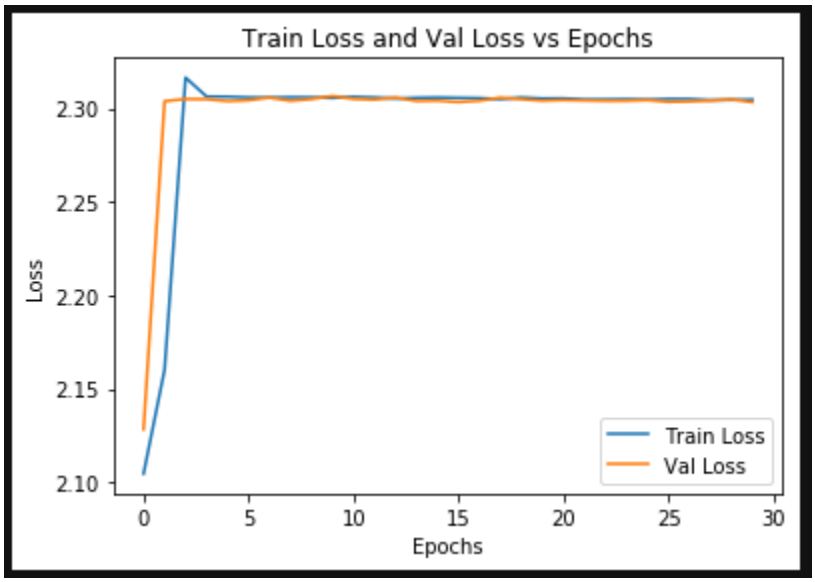
- With Batch Normalization (Initial LR = 0.1)

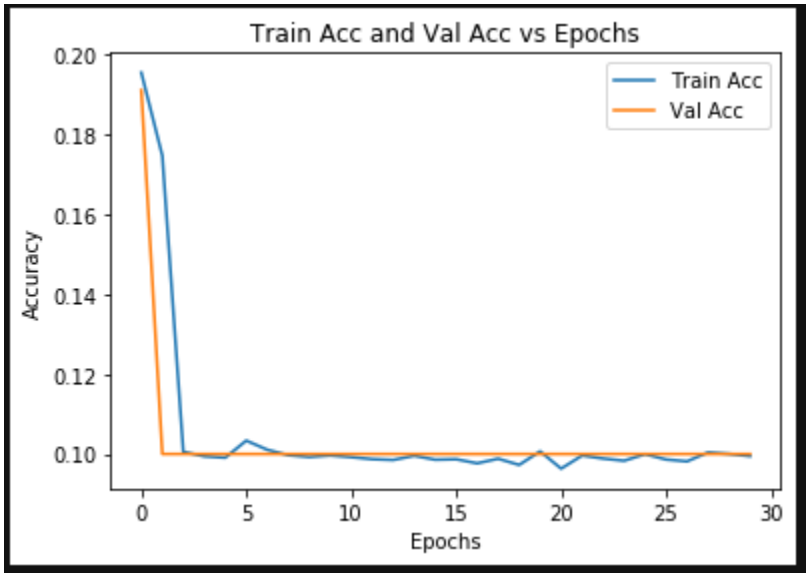





```
Epoch 29:
704
Training loss: 1.2271, Training accuracy: 57.4156 %
2019-10-03 00:24:35.146390
Validation...
Validation loss: 1.0901, Validation accuracy: 63.2600 %
Saving ...
Optimization finished.
```

- Without Batch Normalization (Initial LR = 0.1)






```
2019-10-03 00:38:14.540590
Epoch 29:
704
Training loss: 2.3046, Training accuracy: 9.6600 %
2019-10-03 00:38:24.743875
Validation...
Validation loss: 2.3038, Validation accuracy: 10.0000 %
Optimization finished.
```

Assignment 4:

a) Hyperparameter testing:

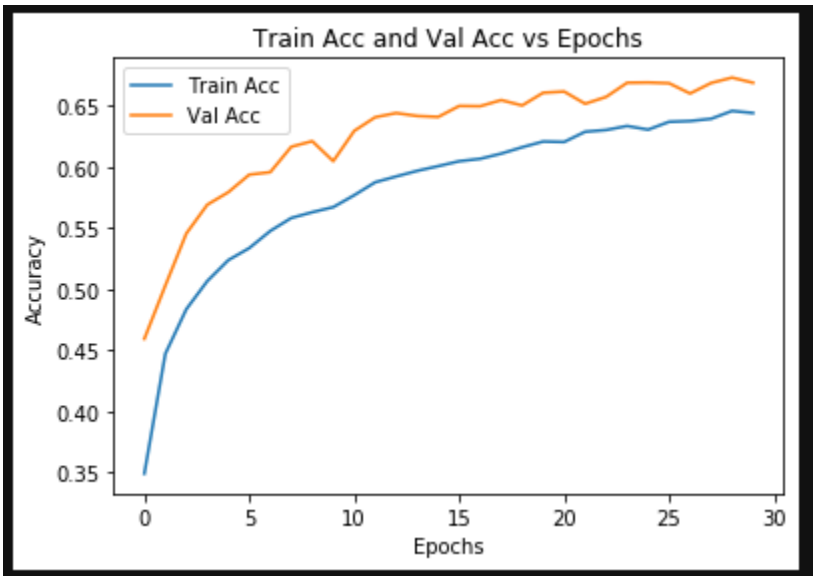
- Testing Batch Sizes:

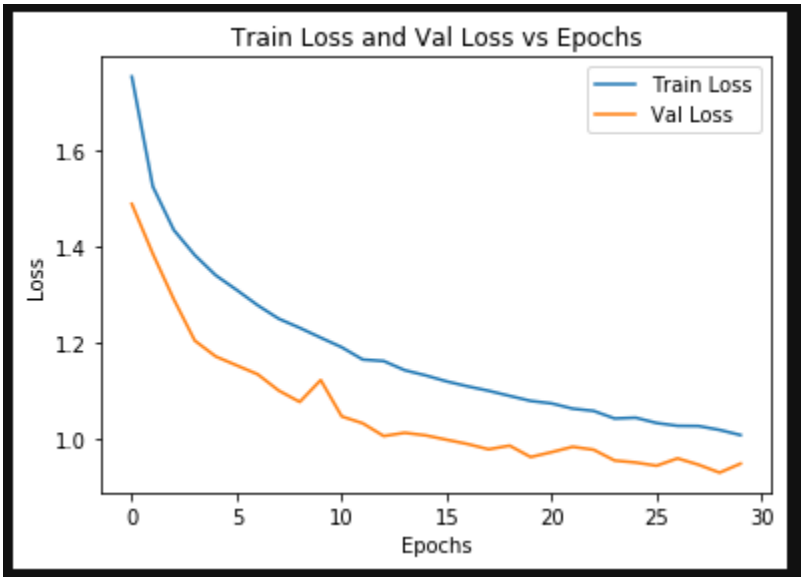
i. Hyperparameters are:

```
# Setting some hyperparameters
TRAIN_BATCH_SIZE = 64#128
VAL_BATCH_SIZE = 50#100
INITIAL_LR = 0.01
MOMENTUM = 0.9
REG = 1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

- Results:

```
Epoch 29:
704
Training loss: 1.0063, Training accuracy: 64.4022 %
2019-10-03 21:09:04.555460
Validation...
Validation loss: 0.9467, Validation accuracy: 66.8800 %
Optimization finished.
```



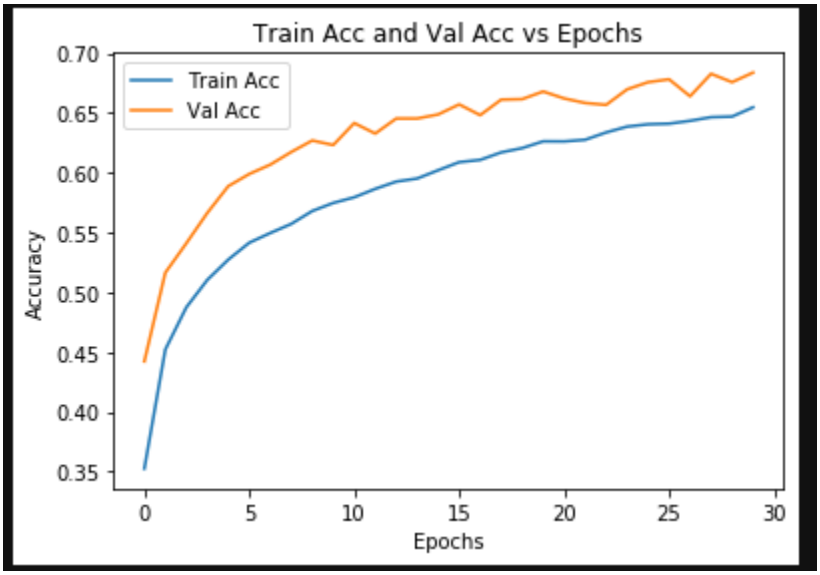


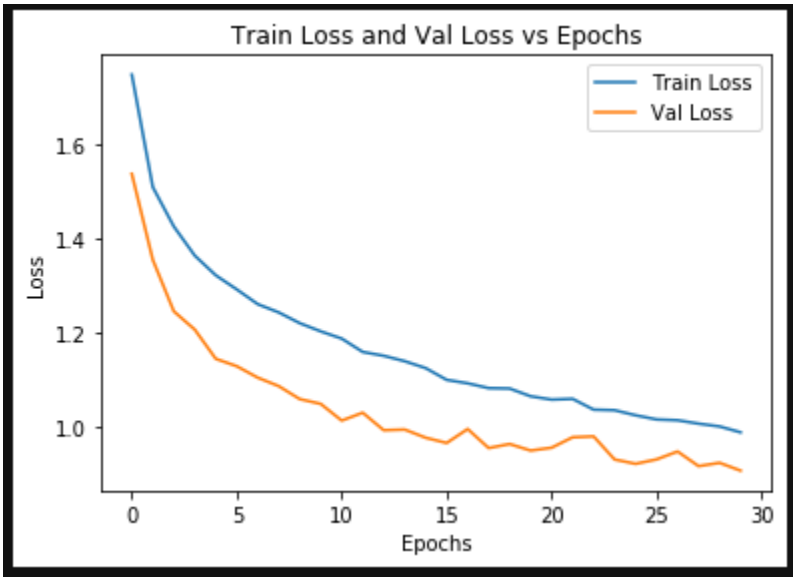
ii. Hyperparameters are:

```
# Setting some hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 0.01
MOMENTUM = 0.9
REG = 1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

- Results:

```
Epoch 29:
704
Training loss: 0.9883, Training accuracy: 65.4756 %
2019-10-03 21:19:41.206500
Validation...
Validation loss: 0.9071, Validation accuracy: 68.3800 %
Saving ...
Optimization finished.
```





iii. Discussion:

- Although the number of epochs is not that great, I can safely say from both my experience (e.g. results above) and what has been said in lecture, that the batch size does not have a very large impact on the model's performance. I do know from my conversations during the TA sessions that the batch size can influence how well the model can escape local minimums. This was described to me as a sort of sharpening of the peaks between valleys, allowing for the momentum term to more easily overcome local minimums. I think any difference above

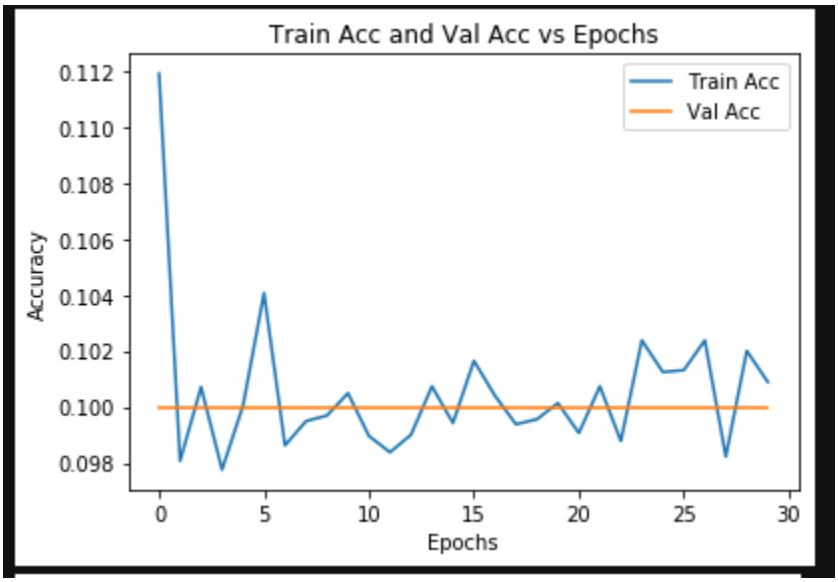
would be due to the small number of epochs used and random chance.

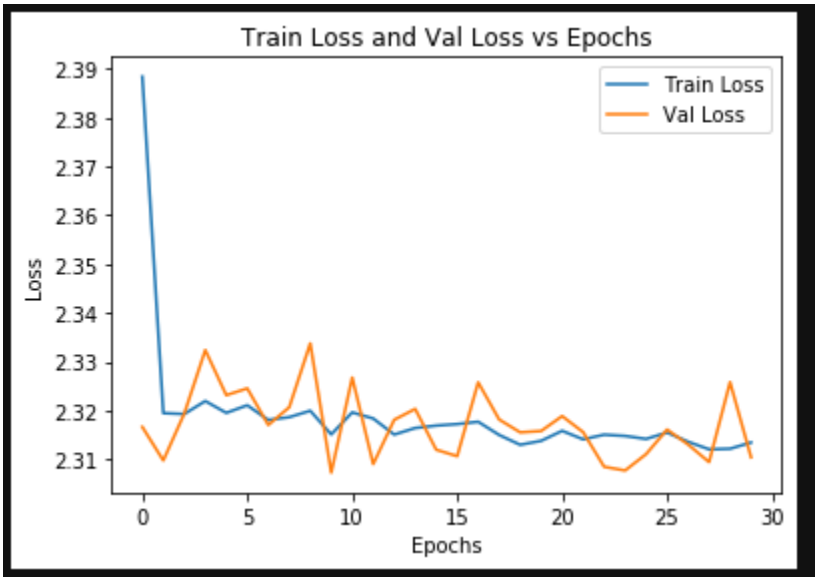
- Testing Initial Learning Rate:
    i. Hyperparameters are:

```
# Setting some hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 1
MOMENTUM = 0.9
REG = 1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

    ii. Results:

```
Epoch 29:
352
Training loss: 2.3135, Training accuracy: 10.0889 %
2019-10-03 21:36:28.806801
Validation...
Validation loss: 2.3104, Validation accuracy: 10.0000 %
Optimization finished.
```



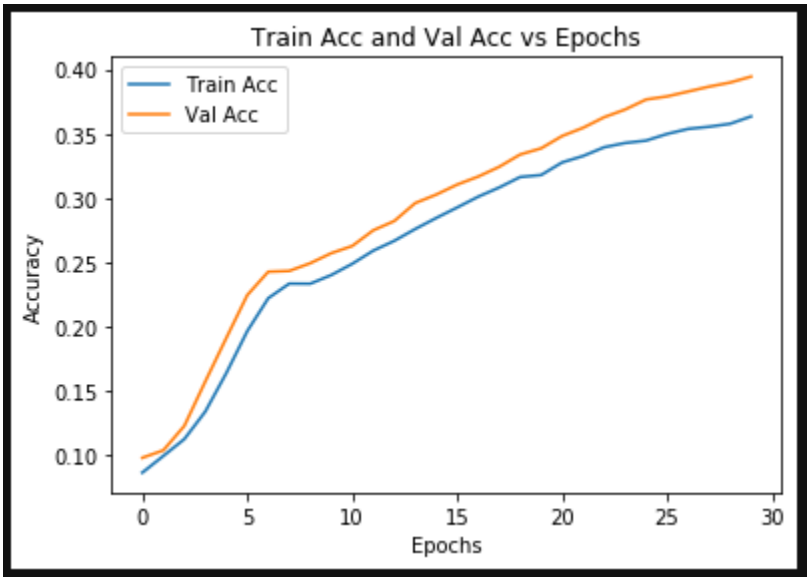


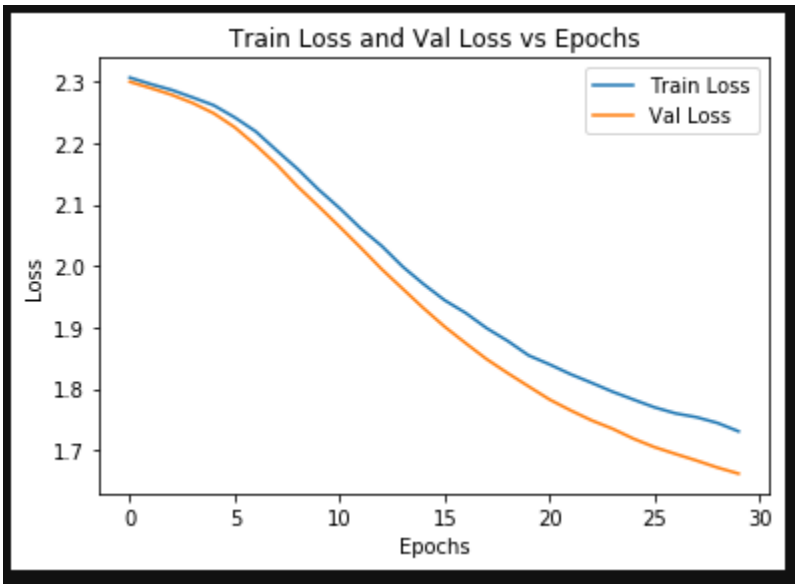
iii. Hyperparameters are:

```
# Setting some hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 0.0001
MOMENTUM = 0.9
REG = 1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

iv. Results:

```
Epoch 29:
352
Training loss: 1.7307, Training accuracy: 36.3422 %
2019-10-03 21:46:44.479478
Validation...
Validation loss: 1.6618, Validation accuracy: 39.4400 %
Saving ...
Optimization finished.
```





v. Discussion:

- As seen above, I would say the learning rate, and specifically here the starting learning rate, have the most significant impact on model performance. With an initial learning rate that is too high, the stochastic descent is stuck bouncing between the walls of the convex loss function. If the model had no learning rate decay, the model could stay in this low accuracy state indefinitely. Finding an optimal initial learning rate is essential

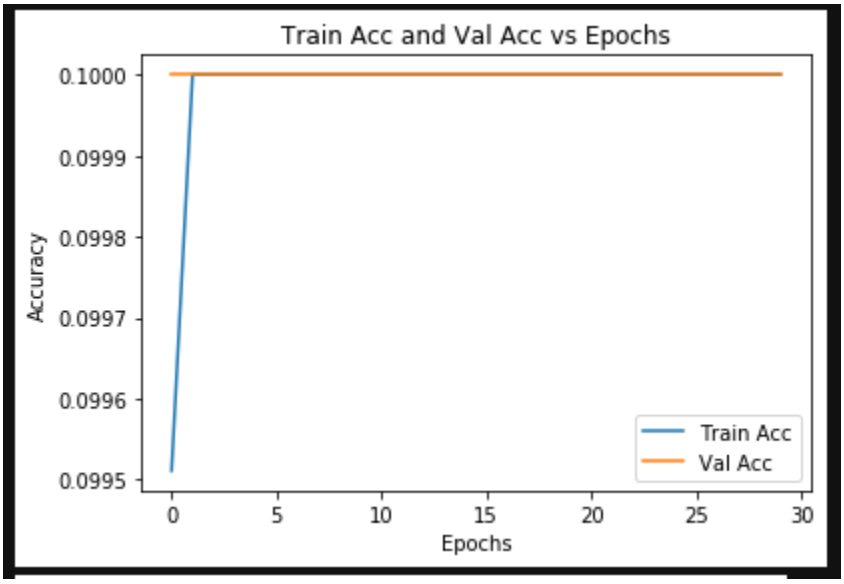
to not wasting valuable computing time trying to find the start of the loss function valley. It should be noted that I am using the same learning rate decay of 0.96 every two epochs for both of these experiments. The opposite occurs when the initial learning rate is too low. The stochastic gradient descent moves too slowly and takes too long to find a potential valley for the global minimum. The learning curves become less logistic and more linear. Both of these are not optimal, with the best initial learning rate being between these two extremes.

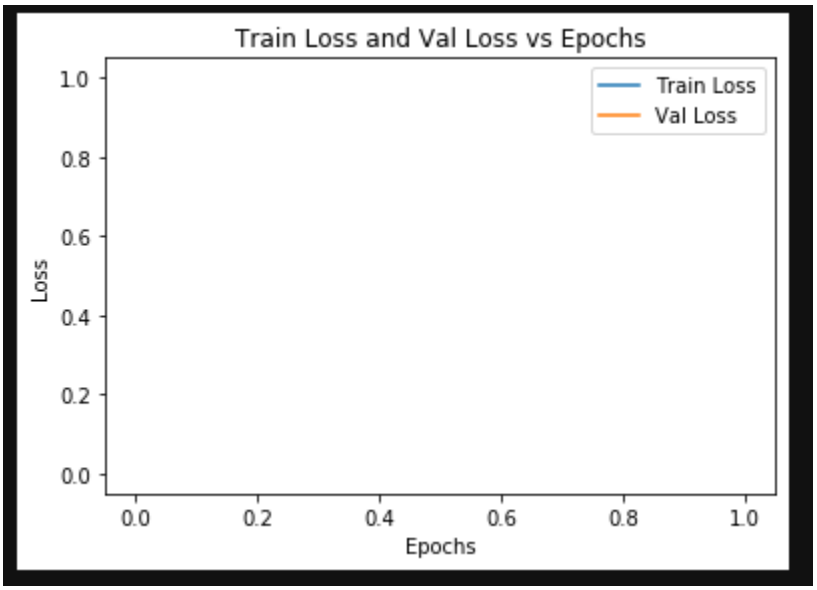
- Testing Momentum:
    i. Hyperparameters are:

```
# Setting some hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 0.01
MOMENTUM = 10
REG = 1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

    ii. Results:

```
Epoch 29:
352
Training loss: nan, Training accuracy: 10.0000 %
2019-10-03 21:57:26.170011
Validation...
Validation loss: nan, Validation accuracy: 10.0000 %
Optimization finished.
```



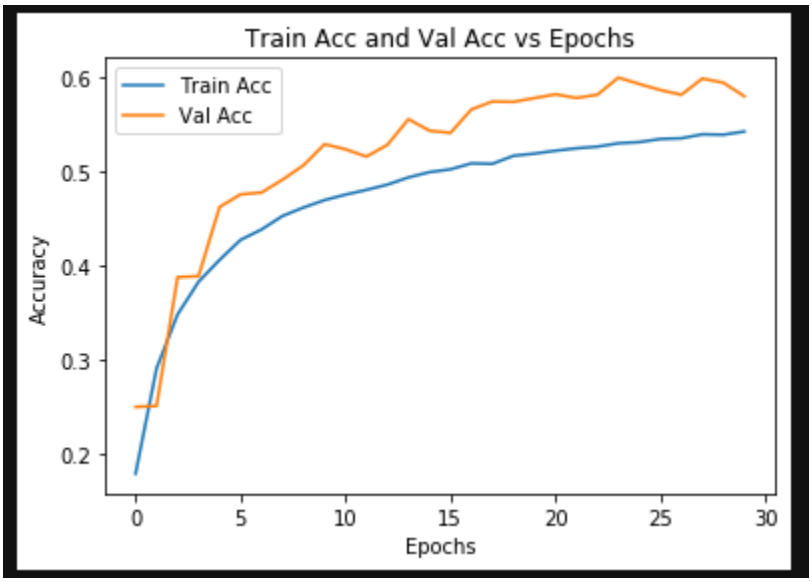


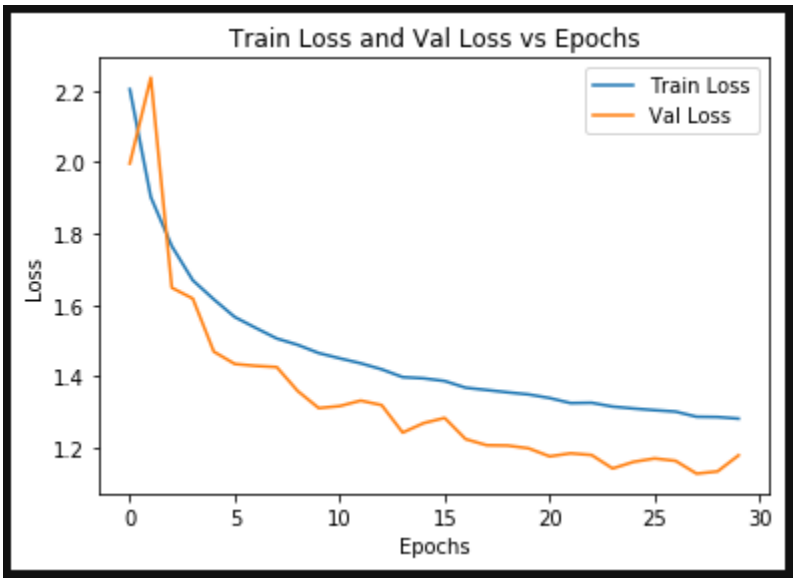
iii. Hyperparameters are:

```
# Setting same hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 0.01
MOMENTUM = 0.001#0.9
REG = 1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

iv. Results:

```
Epoch 29:
352
Training loss: 1.2820, Training accuracy: 54.2667 %
2019-10-03 22:05:54.121762
Validation...
Validation loss: 1.1793, Validation accuracy: 58.0200 %
Optimization finished.
```





v. Discussion:

- It appears from the results above that adding a large momentum term makes the stochastic gradient descent like a turtle stuck on its back, or a monumental lumbering beast incapable of motion. With a high momentum, the optimizer

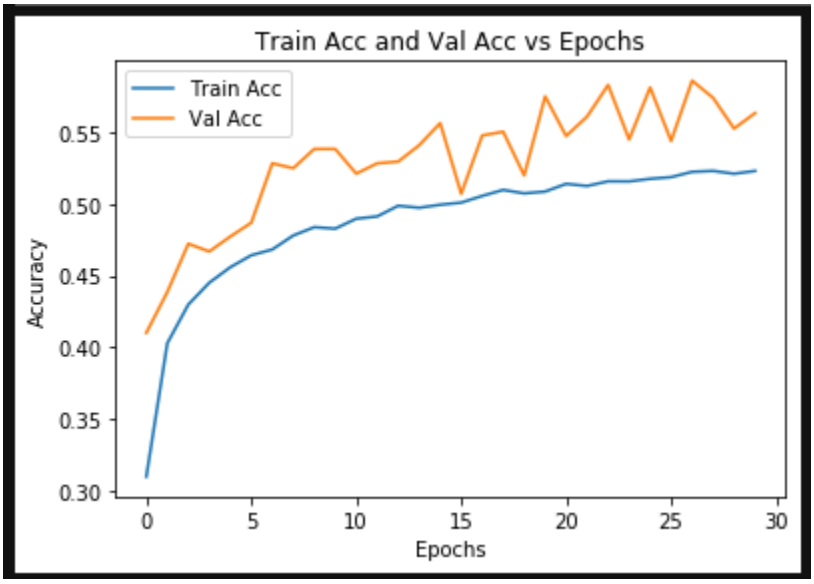
falls in a rut and is too heavy to get out. On the other hand a low momentum term seems to have little effect on the SGD method in the early epochs with what appears to be a relatively monotonically decreasing convex path along the loss function. Recall that SGD is a perfectly reasonable algorithm without momentum, so long as there are not false minimums along the path to the global minimum.

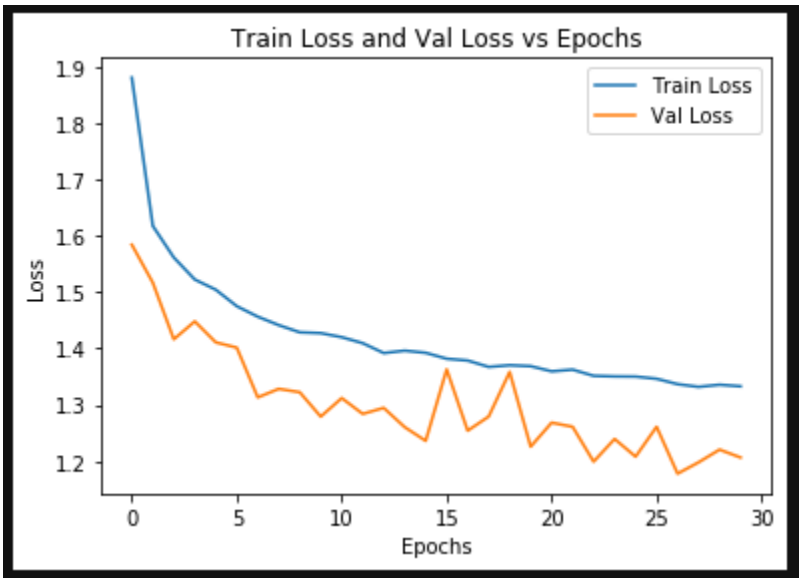
- Testing Regularization:
    i. Hyperparameters are:

```
# Setting same hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 0.01
MOMENTUM = 0.9
REG = 100e-4#1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

    ii. Results:

```
Epoch 29:
352
Training loss: 1.3332, Training accuracy: 52.2889 %
2019-10-03 22:19:17.452344
Validation...
Validation loss: 1.2068, Validation accuracy: 56.3200 %
Optimization finished.
```



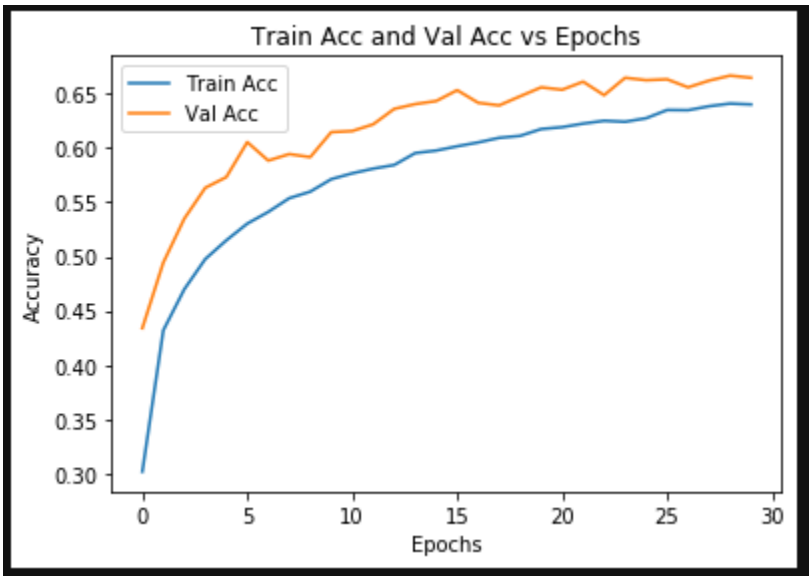


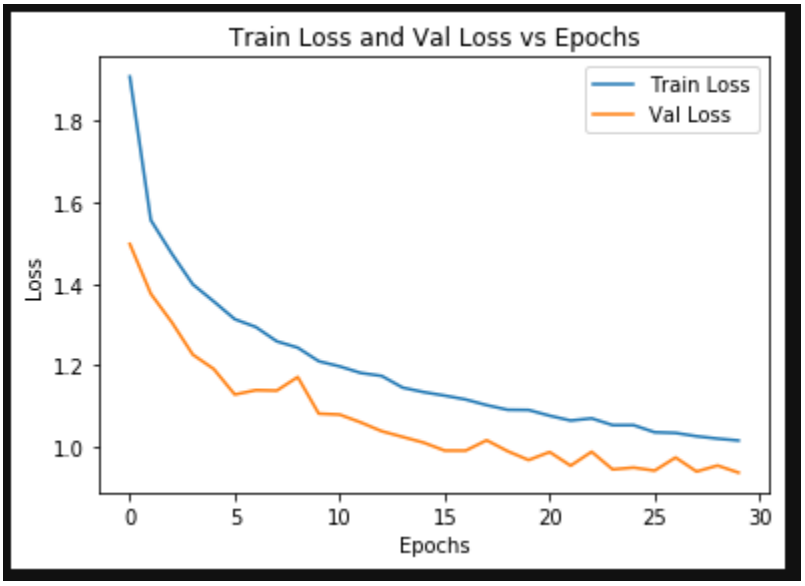
iii. Hyperparameters are:

```
# Setting some hyperparameters
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100
INITIAL_LR = 0.01
MOMENTUM = 0.9
REG = 0.001e-4#1e-4
EPOCHS = 30
DATAROOT = "./data"
CHECKPOINT_PATH = "./saved_model"
```

iv. Results:

```
Epoch 29:
352
Training loss: 1.0156, Training accuracy: 63.9867 %
2019-10-03 22:28:23.544203
Validation...
Validation loss: 0.9367, Validation accuracy: 66.4400 %
Optimization finished.
```





v. Discussion:

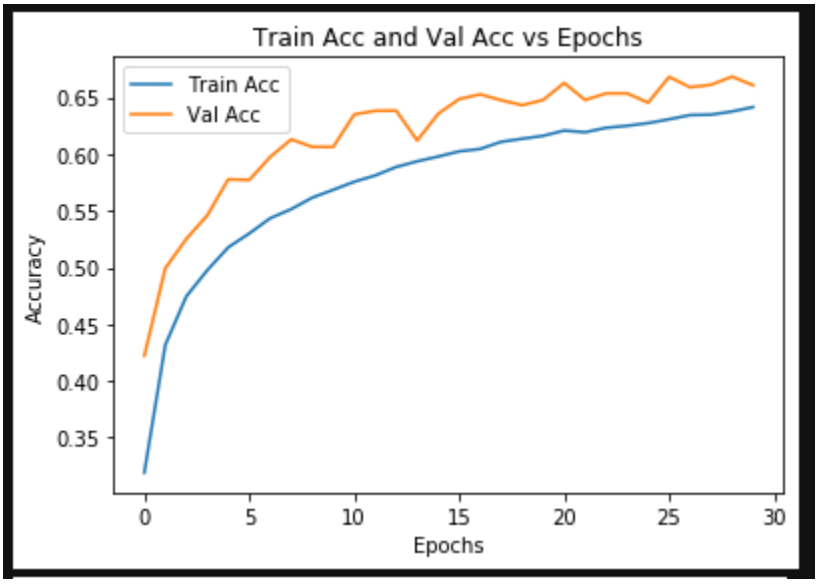
- It is clear from the results above that a high regularization is very bad as it causes the model weights to stay very close to zero and thus limit the model's ability to approximate the

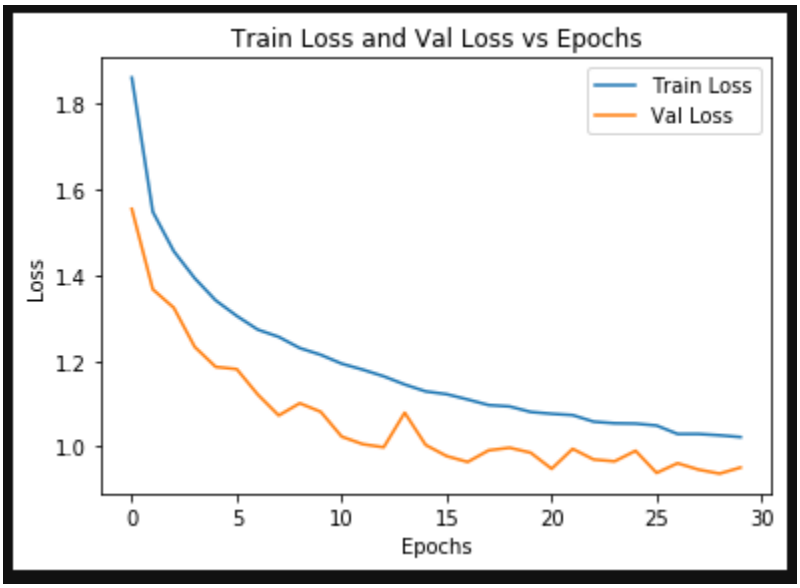
ground truth mapping (as seen by the large variations in the validation graphs). In addition, having no regularization at all is also a bad idea as you can end up with a model whose weights are incredibly large. This could decrease the generalizability of the model as cause the SGD to follow a more difficult path to the global minimum.

b) See below for the results discussed in this section. As seen below, the learning rate is probably the most important hyperparameter, and in a similar vein, the learning rate decay is paramount to a smooth and timely descent to the global minimum. Having a learning rate decay that is too swift will result in the model slowing prematurely before finding the valley of the global minimum (as visible by the plateau below). This means that convergence will be indefinitely delayed or that you could end up stopping in a false minimum. Conversely, having a decay rate that is too slow will cause the learning rate to cause the algorithm to bounce on the walls of the objective function in an oscillatory pattern and possibly bouncing out of a global minimum. As seen below, the model stagnated around 60% with more oscillatory behavior. I choose to keep the same schedule of decaying every 2 epochs, but increased the decay to 0.96. This seems to strike a good balance, keeping the learning curve from plateauing.

- Initial Learning Rate of 0.01 and Decay = 1.0 every 2 epochs (no decay)

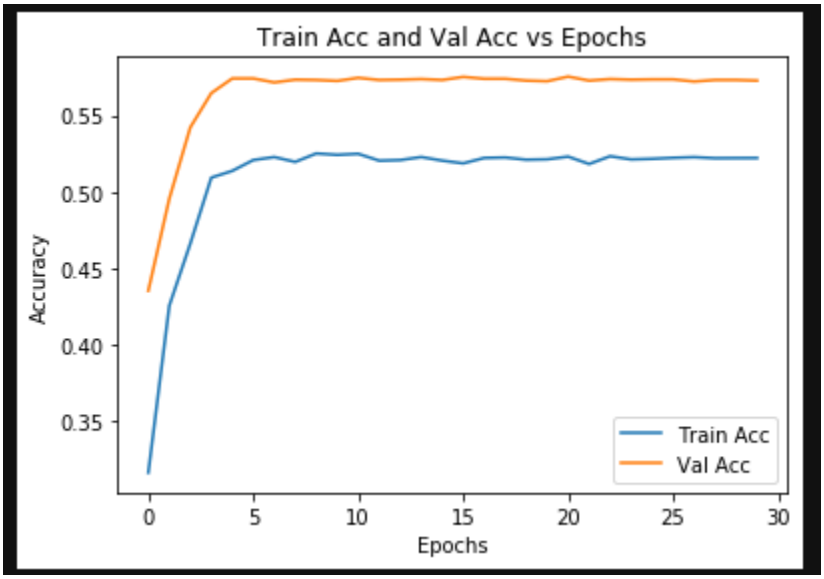
```
Epoch 29:
352
Training loss: 1.0223, Training accuracy: 64.1489 %
2019-10-03 22:49:26.156565
Validation...
Validation loss: 0.9512, Validation accuracy: 66.0800 %
Optimization finished.
```

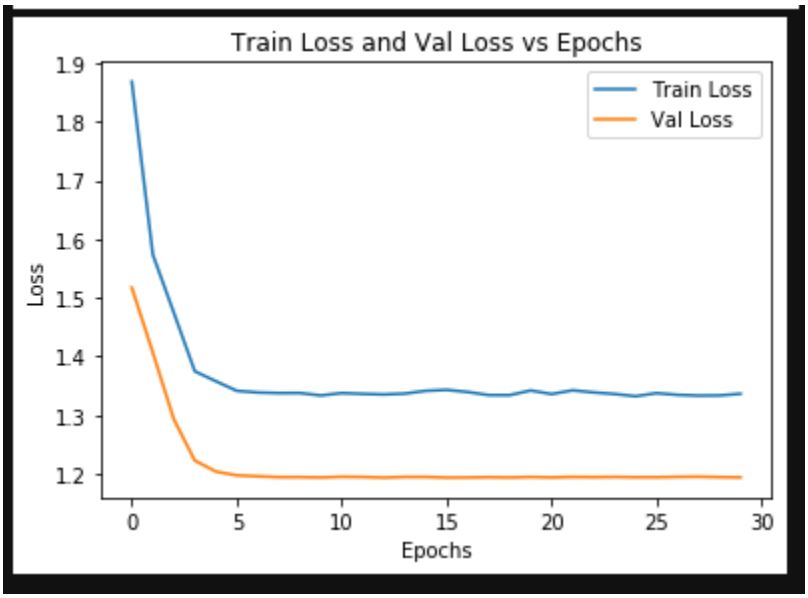




- Initial Learning Rate of 0.01 and Decay = 0.25 every 2 epochs

```
Epoch 29:
352
Training loss: 1.3364, Training accuracy: 52.2156 %
2019-10-03 22:58:59.437840
Validation...
Validation loss: 1.1938, Validation accuracy: 57.2800 %
Optimization finished.
```

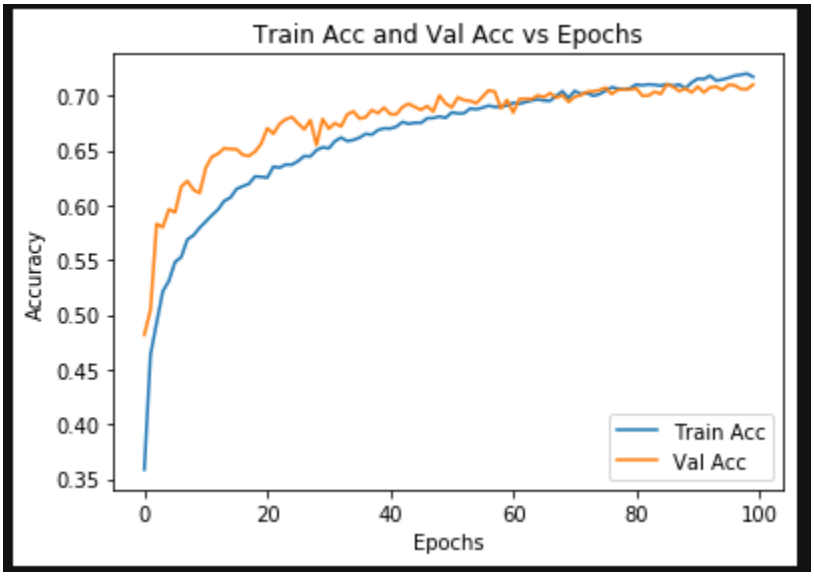


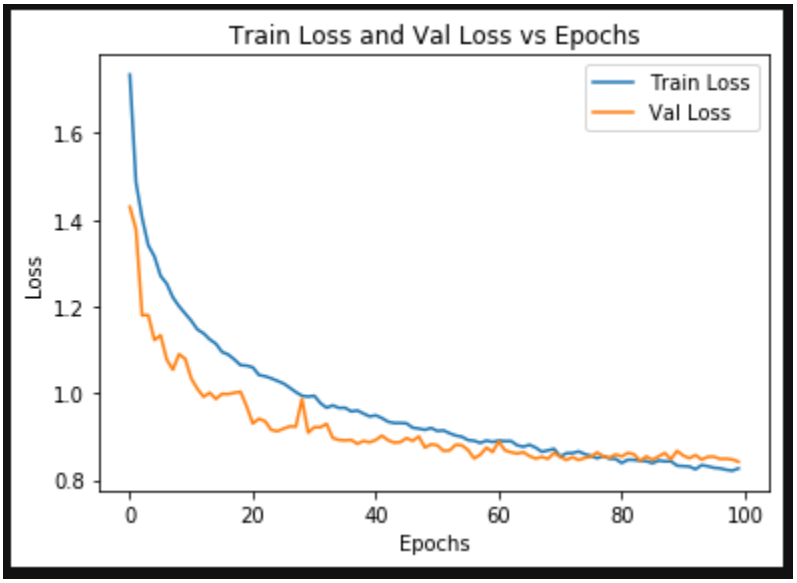


c) For the hyperparameter optimization, I tried to keep the structure of LeNet 5 intact, like the design and activation functions (would have switched to elu()). In order to optimize, I primarily focused on improving my data augmentation by fine tuning the transform probabilities. I addition, I changed the learning rate decay from 1.00 or the suggested 0.98 to 0.96. I found this allowed for a faster convergence without sacrificing accuracy. I also reduced the batch size as I was having some trouble with what I believed to be false minimums. Most of the other parameters I kept as what was stated to be close to optimal in the lecture notes for SGD. As I reached the accuracy I wanted using these methods, I

chose to leave other things static (so as not to mess with a good thing). I of course also kept the optimizations of the previous parts like regularization and batch normalization. My 100 epoch results can be seen below.

```
Epoch 99:
704
Training loss: 0.8269, Training accuracy: 71.7178 %
2019-10-03 23:49:14.332757
Validation...
Validation loss: 0.8422, Validation accuracy: 71.0000 %
Optimization finished.
```





Assignment 5:

a) For the DNN design, I decided to go with a modified VGG-19 structure as it has already been proven adequate in image classification with ImageNet (256x256 images). I decided to use elu() activation functions as I saw a marginal improvement over relu(). In addition, the lecture notes mentioned this modification as potentially advantageous. In addition, I made some modifications to the fully connected layers. I added one layer, less to improve accuracy in my mind and more to reduce dimensionality of connections more gradually. I also added dropout layers in between the fully connected layers before the last layer (with p = 0.5) as the lecture slides suggested this would increase the performance of VGG and this inclination was supported by my own observations. In my CNN design, I also used the suggested neural net block format mentioned in the NumPy/NN tutorial slides. I created my block so that it can dynamically replicate batch-normalization blocks (up to 4 in a row) with the same sized convolution kernels and uniform channels (the first in-channels must differ to accommodate input channels to the conv block). The first fully connected layer has 4096 inputs and 4096 outputs, while the second layer takes 4096 inputs and reduces that to 4096/4. The same pattern of reduction is followed again with a reduction of input channels by a factor of 4. Then finally after a reduction of 1/8 has been achieved from the first fully connected layer, the last layer reduces to the classification categories (10 of them). Another optimization spurred on by the suggestion of the slides was adding Xavier normal initialization rather that Xavier uniform. This also increased overall performance slightly.

b) For data augmentation, I decided to be rather aggressive. I had a feeling that the number of layers in VGG-19 should be complemented by a larger data set than I had. Therefore, I artificially increased the variance of my training dataset dramatically, in the hopes of generalizing my network. I didn't want the order of the transform train to impact the overall variance I introduce, so I used RandomOrder() such that the order of my data augmentation transforms was randomized. In addition, I chose to have some of the transforms have an inherent probability as well determining if they will be applied. I used random flip (horizontal and vertical), random crop, random translation/rotation (affine without shear), random color distortion, and random

erasing. All of these transforms and the stochastic nature of their application greatly increased the variance of my dataset and kept my training accuracy from straying too far away from my validation accuracy.

c) For hyperparameter optimization, I used many of the suggestions detailed in the slides about VGG as a starting point. The main changes are in the starting learning rate which I started lower and eventually found the max without stagnation to be 0.00785 (0.0079 causes bouncing around 10% accuracy). I also chose to train for 400 epochs as the SGD with Nesterov momentum was guaranteed convergence and also guaranteed not to overshoot the global minimum. Anything longer than this resulted in permanent stagnation around 90.6ish percent. I also set the decay rate to 0.96 per 2 epochs as this was found to be optimal through trial and error. I chose cross entropy loss as this also appeared to be optimal.

```
# Specify preprocessing function.
# Reference mean/std value
mean_RGB = (0.4914, 0.4822, 0.4465)
std_RGB = (0.2023, 0.1994, 0.2010)
input_img_size = (32,32)
percent_crop = 0.80 # 80 percent left after crop
data_augment_list = [transforms.RandomVerticalFlip(p = 0.20),
                     transforms.RandomHorizontalFlip(p = 0.40),
                     transforms.RandomApply([transforms.RandomCrop(size = input_img_size, padding=4, pad_if_needed=False, fill=0, padding_mode='constant')], p = 0.25),
                     transforms.RandomApply([transforms.RandomAffine(degrees = 45, translate=(0.15,0.15), scale=None, shear=None, resample=False, fillcolor=0)], p = 0.25),
                     transforms.RandomApply([transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.05)], p = 0.20)]
transform_train = transforms.Compose([transforms.Resize(input_img_size),
                                      transforms.RandomOrder(data_augment_list),
                                      transforms.ToTensor(),
                                      transforms.RandomErasing(p=0.20, scale=(0.01, 0.3), ratio=(0.3, 3.3), value=0, inplace=False),
                                      transforms.Normalize(mean_RGB, std_RGB)])
transform_val = transforms.Compose([transforms.Resize(input_img_size),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean_RGB, std_RGB)])
```