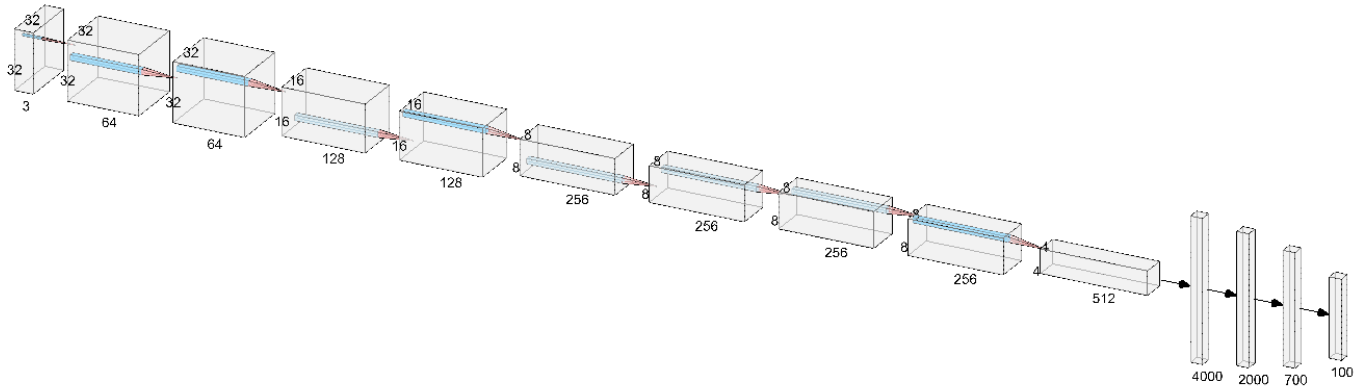
Lab 3 - Image Classification

William Dormer, 20176544

Section 1: Vanilla Architecture

The goal for my design of the vanilla architecture was to make the simplest possible architecture This model is simply to serve as a baseline, and more advanced architectural changes will be made in the next iteration.

The network consists of simply the VGG encoder provided to us, with several feedforward layers appended to the end of the network for classification.



This figure shows the output sizes for the convolutional layers and feedforward layers for the vanilla model. There are also many reflection pads, maxpools, and ReLUs mixed into the architecture, as per the original VGG encoder. The only part I added for the vanilla model were the 4 feedforward layers to classify the input.

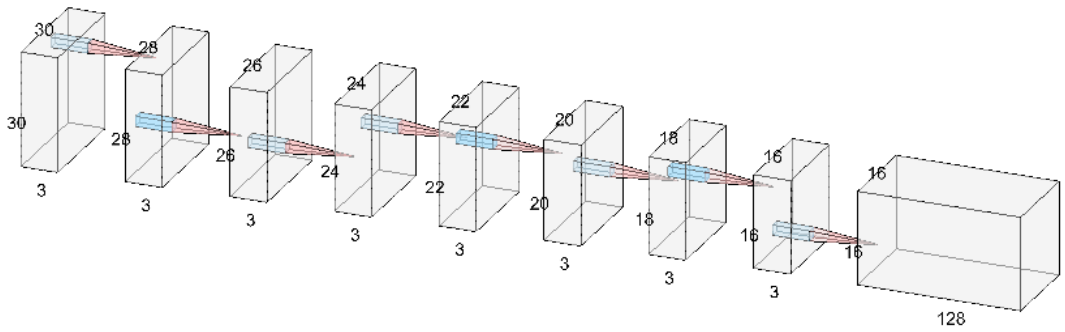
Section 2: Mod Architecture

The objective of this lab as stated in the lab manual was to implement one of the architectural modifications that we learned in class, in order to show that it improved the model's performance over the vanilla model, all else being equal. For this lab I have selected the concept of the residual connection from ResNet. The idea behind the residual connection, or skip connection, is that it assists in overcoming the vanishing gradient problem. It does this by allowing the gradient to propagate freely through the shortcut connections, which are identity mappings in most cases. As stated in class, this essentially allows the network to learn the difference between the input and the output instead of the entire transformation. This fix allows

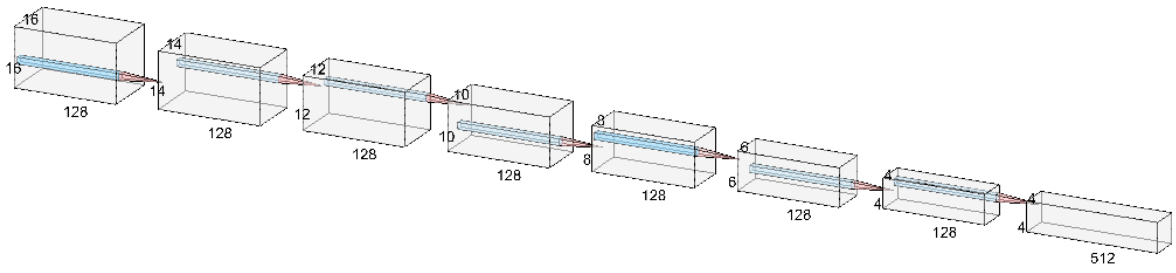
for training deeper network layers more efficiently, and I intend to show phenomena in this lab. Specifically, I want to show that the skip connections allow for better fine tuning of the early stages of the VGG encoder.

The network architecture for the upgraded model with skip connections required some modification. I added two skip connections to my architecture, with each of the residual blocks containing about half of the encoder. However, since the spatial dimension and channel dimension of the network change as you move down the layers in the network, a simple addition operation of the feature map at the start of the residual block and the end is not sufficient. An operation was needed to map the residual connection to the right dimension. To do this, I implemented what is called a bottleneck block. It first compresses the spatial dimension to the correct shape using repeated 3x3 convolutions, but with no activation function, so it is essentially a linear map. Then it does similar for the channel dimension, but using a 1x1 convolution. By doing this, we construct as close as possible to an identity connection as possible given the dimensions. It is possible to do a simple crop to solve the same problem, but I found this to be a less elegant solution.

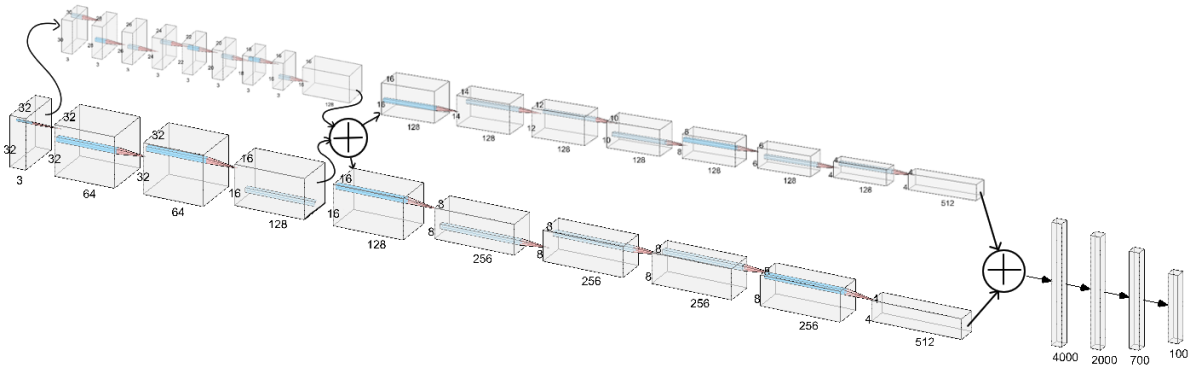
the first bottleneck block



the second bottleneck block



the full architecture, including the residual operation marked with the arrows.



Section 3: Experiments

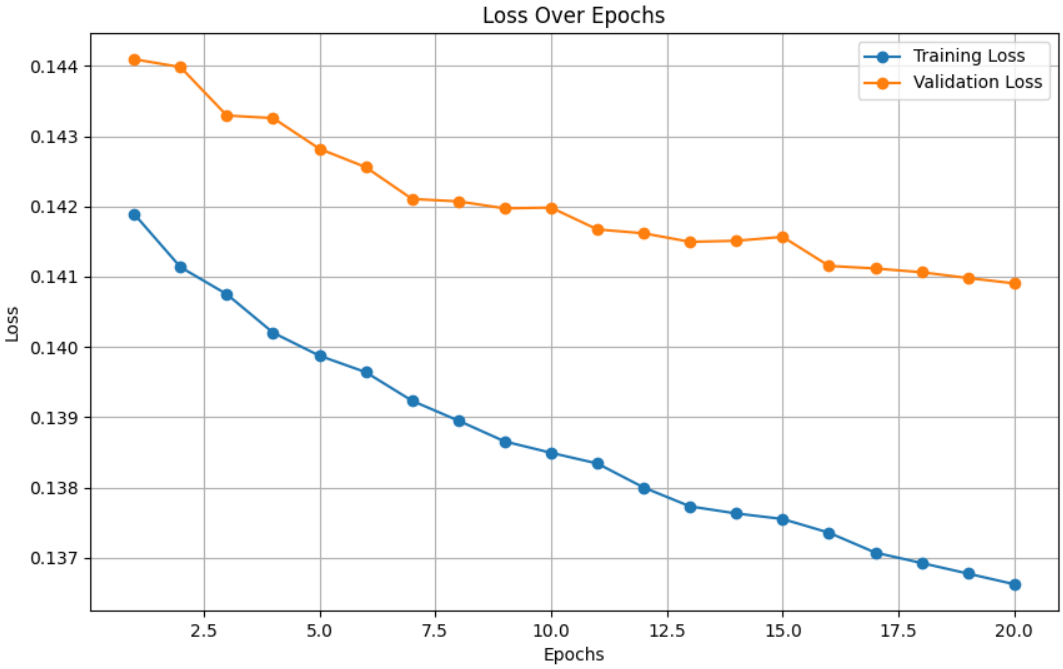
All experimentation was done on my GPU (GTX 1070).

In order to properly isolate the impact of the model architecture vs the hyperparameters, I kept as many variables constant as I could:

These were common between the two models:

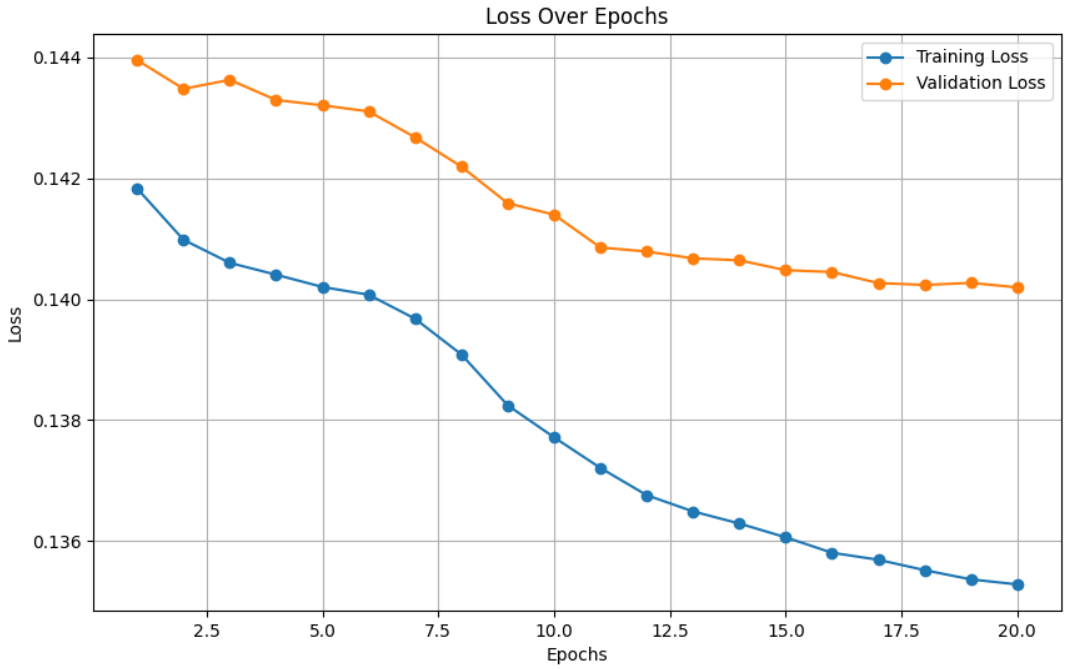
- learning rate = 0.00001
- batch size = 32
- epochs = 20
- transform = toTensor + Normalize
- optimizer = torch.Adam
- loss function = CrossEntropyLoss

After getting the model working on CIFAR 10 with a high accuracy, the first run on CIFAR 100 I performed was on the vanilla architecture. However, this training run was not allowed to update the backend VGG encoder, only the classification linear layers. The results are outlined below.



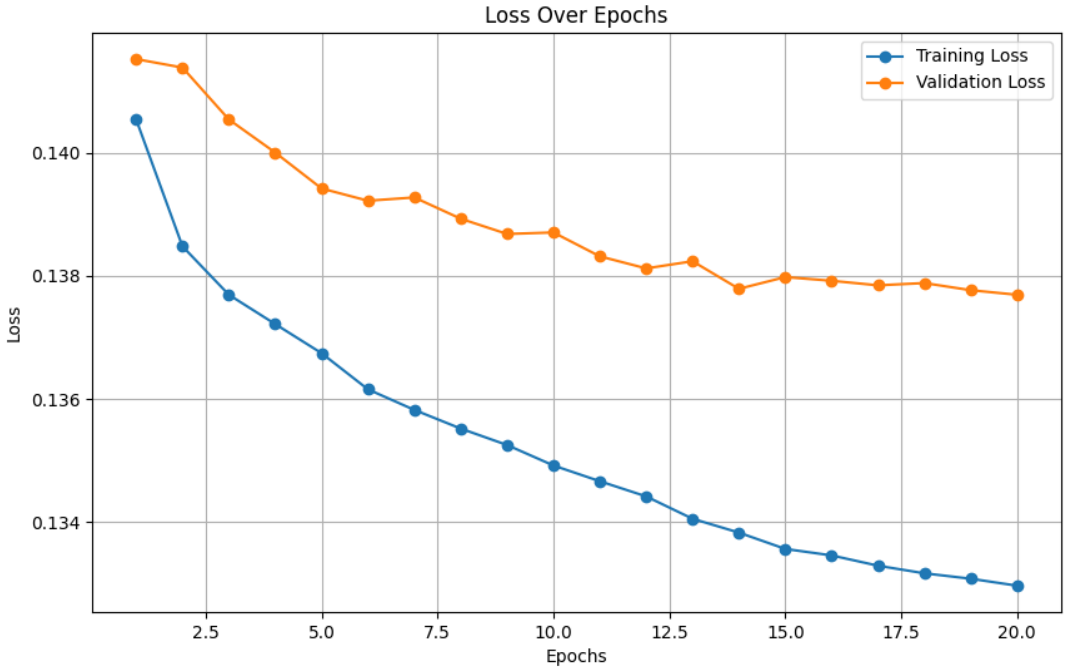
Accuracy: 19.62% ~20x better than random.
name: vanilla_model_classifier

I then ran a test of the vanilla model again, but this time allowing it to modify the encoder weights, as I thought this would be a better comparison for the upgraded model.



Accuracy: 23.57%, about 23x better than random
VanillaModelV1.2

Then finally I ran the upgraded models with the same hyperparameters.



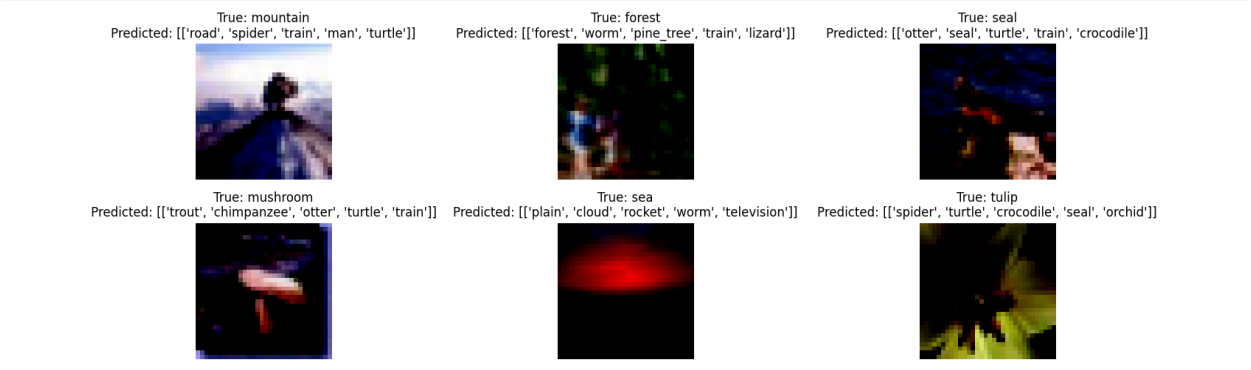
Accuracy: 28.22% about 28x better than random.

UpgradedModelV1.1

As you can see, none of these models have fully saturated their loss curves yet and thus could be trained for more epochs. Each training took approximately 20 minutes.

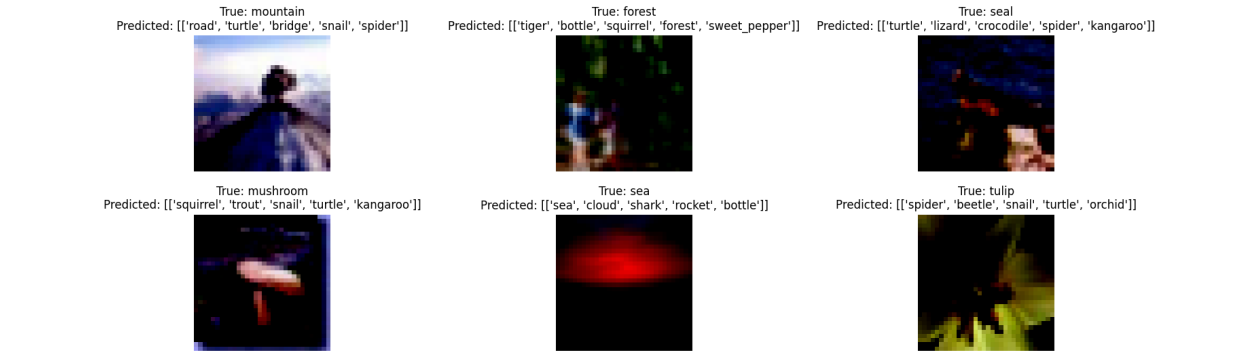
These are the outputs of my testing functions for the two models:

vanilla model:




```
top 1 error rate:  0.7727999687194824
top 5 error rate:  0.6991000175476074
```

upgraded model:




```
top 1 error rate:  0.7109000086784363
top 5 error rate:  0.6276999711990356
```

Section 4: Discussion

Overall, the addition of the residual connections to my network improved the classification performance on CIFAR 100 by about 5-7%. The classification accuracy of both networks is quite low (in the 20s) but for the purpose of this lab, my goal was not to maximize the accuracy, but rather prove that the architectural changes had an impact on the accuracy, which it did. This was expected, since the concept of the residual connection has been well established and used

in many famous architectures. These residual connections allowed the network to propagate the gradients better into the earlier layers of the VGG encoder, leading to fine tuning that was better suited for the CIFAR 100 dataset, as opposed to image net, which it was originally trained on.

There is a caveat worth mentioning, which is that the model now presented has significantly more parameters than the original model, since each of the added convolutional layers have their own parameters. This leads to increased training time, although the difference was not very large given how long I trained for. This could be minimized by using larger convolutions instead of the repeated 3x3 convolutions that I implemented. Alternatively, you could implement another strategy for making the dimensions match, such as cropping. Since the added convolutions do not have activation between them, I don't believe they are simply acting as a parallel branch, but still the number of parameters to learn for the linear mapping is quite large.

Of course, there are many things that can be done to further improve the performance of this model, including other architectural changes mentioned in the lectures, implementing a vision transformer, optimizing hyperparameters, training longer, making the network deeper, adding significantly more parameters, adding parallel branches, using dropout when training, adding weight regularization, using data augmentation, over-representing tough cases etc. If I have time, for the competition in class, I will likely attempt to implement a vision transformer using an encoded input from the VGG encoder (likely not at the very final layer).