Effects of Model Architecure and Data Augmentation on CIFAR-10 Performance

Nicholas Keriotis
University of Illinois Urbana-Champaign
201 N Goodwin Ave, Urbana, IL 61801
nk31@illinois.edu

Video: https://youtu.be/3dLHYk74HCY

1. Introduction

In the field of computer vision, image classification is one of the most fundamental and important problems, as several other problems like object detection and image captioning are heavily dependent on a model being able to extract features and make classifications on objects in an image. To delve deeper into this fundamental problem, this project will attempt to explore and expand upon several methods and innovations to develop a better understanding of effects of model architecture and data augmentation with the additional goal of achieving accuracies similar to human performance of 94%.

1.1. Why CIFAR-10?

Although the CIFAR-10 dataset is relatively outdated and essentially solved with state-of-the-art models achieving accuracies of over 98%, the CIFAR-10 dataset is very well understood and quite accessible from both a complexity and hardware perspective. While other datasets like ImageNet have taken on the role of being the "go-to" dataset for image classification, working with data on such a large scale would be prohibitively expensive from a hardware and development perspective due to the sheer size of the dataset and the length of time that would be spent training instead of iterating on model designs and techniques, thus, the CIFAR-10 dataset was selected due to its simplicity and accessibility.

2. Approach

2.1. Baseline Models and Modular Building Blocks

To start, a few baseline models were designed to ensure the basic training loop was correct and to gain some baseline results that could referenced once future models were designed. Two of the first models designed were simple CNNs with 4 and 6 convolution layers which followed a simple pattern of convolution, max pooling, and activation and capped with a dense layer. Although effective, a simple CNN is just a baseline, so a variety of other architectures were developed to evaluate their performance and impact on training when compared to these simple CNNs.

2.2. Residual Connections

One of the most fundamental developments in computer vision architectures came with the introduction of residual connections. A residual network is designed to allow for clean uninterrupted gradient flows throughout the entire network by adding additional connections that bypass learned parameters. Bypassing these learned weights allows the network to propagate an identity function of the previous activations past a layer where they are then recombined with the activations from the learned layer via addition. The first implementation of residual connections closely followed the concepts of residual blocks as referenced in slides from lecture, with the residual block being composed of 2 CNN layers using 3x3 kernels with a padding and stride of 1 with additional batch norm and ReLU layers between each convolution layer. The initial input to the residual block was also added back to the output after the activation as to allow proper identity functions to propagate forward through the network.

2.3. Bottleneck Blocks

Another architecture tested was the bottleneck block which was also introduced in the original ResNet paper. The idea behind the bottleneck block is to first reduce the number of channels present in an activation map before performing an expensive convolution operation. After the convolution, the condensed feature map is then decompressed to the original number of channels before adding back residual activations. The addition of multiple activation layers in the encoding, convolution and decoding process should also allow for more expressive activations since the activation is applied 3 separate times, however there is a limit when performing the encoding process. In order for the bottleneck layers to be effective, the latent space must be large enough to contain most of the important information from previous activation layers, otherwise information will

be over-compressed and important features may be lost.

Although the over-compression issue can limit maximum performance gained from the bottleneck block, the block as a whole sounded effective as an approach, and an extension of the principle was designed called the Double Encode Bottleneck Block (DEB block). As the name suggests, instead of encoding the activations once and decoding the convolution results once, the DEB blocks encode the incoming activations twice before performing the convolution, then the results are decoded twice with each encoding and decoding layer being independent. In theory, this should provide more expressive operations since the activation functions are applied for each stage of encoding and decoding. The DEB block seeks to allow for greater channel compression with more expressiveness coming from more activations. See figure 4 for an illustration of the architecture.

To examine how much performance may be saved when using a single vs double encoding structure, we can examine the theoretical number of FLOPs each layer will use. Assume our activation maps have shape (C, H, W) where C is the number of channels and H and W are the dimensions of the feature maps. Further assume that our encoding factor is N, and for the double encoding version, we have an additional encoding factor M. The number of FLOPs used by a standard bottleneck block can be calculated as follows:

First, we calculate the number of FLOPs used during the encoding process which goes from C channels to $\frac{C}{N}$ channels.

$$\text{Conv}\left(1, C, \frac{C}{N}\right) = \frac{1}{N}HWC^2 \qquad \text{FLOPs}$$

Next, the operations from the convolution layer can be calculated as:

$$\text{Conv}\left(3, \frac{C}{N}, \frac{C}{N}\right) = \frac{9}{N^2}HWC^2 \qquad \text{FLOPs}$$

Finally, the decoding layer operations can be calculated as:

$$\text{Conv}\left(1, C, \frac{C}{N}\right) = \frac{1}{N}HWC^2 \qquad \text{FLOPs}$$

Overall, the total number of FLOPs for the entire block excluding the residual operations at the end is:

$$\left[\frac{2}{N} + \frac{9}{N^2}\right] HWC^2$$

Moving on to the DEB implementation, we will again assume the same feature map dimensions with an added second encoding factor M. The calculations are similar, and a full derivation can be found in equation (7).

The total number of flops from a single DEB block is:

$$\left[\frac{2}{N} + \frac{2}{MN^2} + \frac{9}{M^2N^2}\right] HWC^2$$

We can compare the numerical difference between the two implementations using an example where a single bottleneck block will have an encoding factor of 8 while the DEB implementation will have encoding factors of 4 and 2.

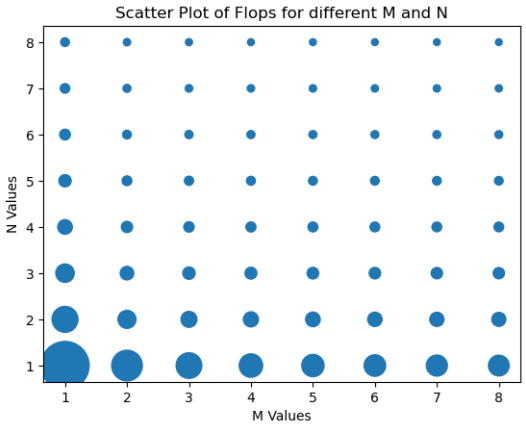
For the standard bottleneck block, we have:

$$\left[\frac{2}{8} + \frac{9}{8^2}\right] HWC^2 = \frac{25}{64}HWC^2$$

For the DEB block, we have:

$$\left[\frac{2}{4} + \frac{2}{2*4^2} + \frac{9}{2^2*4^2}\right] HWC^2 = \frac{45}{64}HWC^2$$

Overall, it would be more efficient to use a single compression of channels with the standard bottleneck implementation, but the DBN architecture allows for more activations when compressing by the same factor, which may help preserve information into the lower dimensional space. The number of FLOPs also varies depending on the values of M and N, as reversing the values provides different results.



The above graphic displays the relative number of FLOPs used in the DBN layer where the size of each dot represents the number of FLOPs used. It is more efficient to compress channels earlier since the second compression directly benefits from the already compressed activations. It should also be noted, that while the DBN layer is capable of providing better efficiency than a normal bottleneck layer, it is unable to provide this efficiency when the true compression between both layers is the same. In other words, if a single bottleneck has a true compression of 8 where $N = 8$ and a DBN has a true compression of 8 where $N = 4$ and $M = 2$, the DBN will still have a higher FLOPs usage since the encoding and decoding happens twice, but it should be more expressive since there are more activation layers in the block as a whole.

2.4. Highway Blocks

Since residual connections have proven effective in both literature and experimentation, another architecture called the Highway Block was developed for the purposes of this project. The main idea behind the highway block was to string together multiple sequential blocks, with an additional "highway" residual connection that would pass the input activations all the way to the output, allowing for identity activations to bypass much longer activation segments which can allow for more stable training.

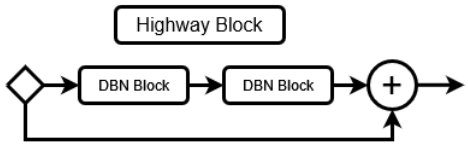
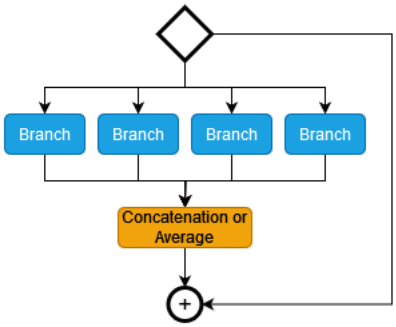


Figure 1. A diagram of a highway block which creates a long residual connection around modules inside it.

2.5. Branch Blocks

The final architecture designed was heavily inspired by the ensemble method known as bagging. The goal of bagging is to reduce the variance of a model by taking its results and averaging them with the results of similar models. Traditionally, bagging is done with separate models being trained on different subsets of the training set, but all parallel branches will see the same activations for the puropses of this project.

The Branch block also has very similar qualities to GoogLeNet's Inception Modules where parallel branches can see the same input activations and they produce output activations which are then combined through concatenation to produce larger activations. The Branch block implemented in this project also has an additional functionality where activations from the branches can be averaged instead of concatenated to produce a same-sized output activation map which also allows for the use of residual connections. Averaging activations between branches seeks to find a universally agreed on activation map between all branch layers which can then be passed on to the next layer of the network.



3. Results

3.1. The Training Loop

Before training models can begin, the training loop must be defined, as the decisions made in the training loop can drastically affect the results of all other models. Unless otherwise noted, all models mentioned were trained using these specifications. Models were trained on the CIFAR-10 dataset with an 80:10:10 train, validation, test split with the split being deterministic as to allow for future training of models if needed. With 50,000 labeled images in the training set, this translates to a 40k:5k:5k train, validation, test split, combined with a batch size of 256. All models were also trained using the Nesterov accelerated SGD optimizer with a momentum of 0.9, which provides more accurate estimations for momentum based training and has been shown to be more effective at escaping saddle points in gradient space. Cross entropy loss was also used for determining model losses.

A learning rate warm up scheduler was also used from a learning rate of 0 to an initial learning rate of 5e-2 over the course of 5 epochs to ensure that training would be stable from the start. Following the learning rate warm up, a learning rate scheduler from PyTorch called `ReduceLROnPlateu` was used to automatically reduce the learning rate if no performance improvements were seen for a period of time. The learning rate was reduced by a factor of 2 if the validation loss did not decrease substantially over more than 4 epochs. This effectively provides a highly optimized learning rate schedule, as the learning rate is kept as high as possible until training slows before it is reduced which allows the model to learn at a stable rate without the need for user intervention. Models were also allowed to train for a maximum of 200 epochs, however, many models finished training well before this point, as training was deemed to be "finished" once the learning rate had decayed past 1e-7 which is around the point that the gradient updates are so small that nothing significant will be learned.

In addition to the features previously mentioned, all models were trained on augmented data. These augmentations consist of random perspective transformations to attempt to show objects from different angles, random gray scale transforms where an image was converted to gray scale with some probability, random rotation of the image between 0 and 15 degrees, random horizontal flips with a probability of 50%, and random color jitter which shifts the HSV values of the image within some range. Samples from this augmentation technique can be found in figure 2. The augmentation described was the "Easy" variant, with harder variants including increasing the likelihood and intensity of certain transformations. Other variants of this data augmentation policy called Hard2, and Hard3 were also developed and can be found in figure 3.

Metrics like max training and validation accuracies were obtained from the training process, and test results were obtained a single time after models had finished training. Additionally, due to page limitations, architectures of select models will be included as additional figures, but all model architectures can be viewed in the source code in `models.py`.

3.2. Baseline Results

Starting with the baseline models, A total of 3 models were tested, all of which were composed of only convolution, batch norm, and ReLU layers with average pooling, linear, and layer norms for the final predictions. All models had various sizes, with larger models having more and wider convolution layers than smaller models. The architectures of these models were relatively simple, with the best baseline model being composed of 6 convolution layers. All models ended up overfitting without data augmentation techniques, with the largest model only achieving 80.9% validation accuracy while reaching 99.9% on the training dataset. To increase the regularization on these models, data augmentation was implemented to ensure the augmented dataset would be too difficult to simply memorize. In general, applying light data augmentation with the Easy augmentation variant was effective at preventing overfitting for all but the largest models. This augmentation also provided substantial performance improvements to validation accuracies when comparing to models trained without augmentation. When augmentation difficulty was increased to its maximum level with the Hard3 augmentation policy, scores dropped significantly, with many models losing over 10% validation performance, but still having significantly higher validation scores than training scores. An interesting observation made was that the smallest baseline model had similar performance to the largest baseline model, and actually outperformed it when data augmentation was at its hardest, reinforcing the fact that deeper and wider CNNs alone are unable to scale well due to the lack of residual layers in the model. Tables 1 and 2 show results from these evaluations, with the model's parameter count being used to identify it. Figures shown represent percent accuracy on the respective split of the dataset. The architecture for the 430k baseline model can be found in figure 5.

Model	130k	130k	130k	108M	108M	108M
Train	99.9	88.6	65.8	99.9	97.4	65.6
Val.	81.0	83.5	72.8	80.9	83.4	70.7
Test	80.4	82.6	72.9	80.5	82.4	70.8
Aug.	None	Easy	Hard3	None	Easy	Hard3

Table 1. Results from baseline models

Model	430k	430k	430k
Training	99.9	94.5	73.4
Validation	81.8	84.9	78.7
Test	82.1	82.9	78.9
Aug.	None	Easy	Hard3

Table 2. Results from best baseline model

3.3. Residual Results

Moving to the Residual Blocks, one of the first residual models trained used the standard residual block architecture. Without any data augmentation, the first residual model was already superior to the baseline CNNs, achieving a validation accuracy of 87% in under 50 epochs of training. Examining the losses for this particular network revealed that it had also overfit to the training set with a training accuracy of 99.9%, so augmentation was needed to prevent simple memorization of the data. The introduction of augmentation proved effective in regularizing the model, as the previously overfitting models had improved their validation accuracies by around 3% while the train accuracies were reduced. One trend observed when applying harder data augmentation was the validation accuracy consistently being higher than the training accuracy for a significant portion of training. This trend is likely caused by the fact the model is being trained on a much harder dataset but being evaluated on an easier one without augmentation. In effect, the model can be seen as preparing for a worst case scenario where all samples are augmented, thus forcing it to be robust to data variations and ensuring overfitting is impossible since the dataset is so much larger with potential augmentations exponentially increasing the number of ways a single data sample can be viewed.

Table 3 shows the training, validation and test accuracies of a baseline residual network with different data augmentation methods. As can be seen below, all augmentation types were effective at regularizing the model by preventing overfitting, but the Hard3 augmentation appears to have been too difficult for the model to properly learn, as both training and validation accuracies suffered significantly when this method was introduced, although degradation was less pronounced when compared to the baseline models. As previously noted, the Hard3 augmentation clearly exhibits a robustness accuracy regime, where the model's validation accuracy is significantly larger than the training accuracy. While this can be viewed as a positive since the model is very robust to out of distribution samples, the overall performance hit on the validation set is likely not worth the exchange. The Easy augmentation variant appears to have been most effective at improving validation accuracy, however the still high training accuracy suggests that the model

can likely train on augmentations of difficulties between Easy and Hard2 and still see further performance gains. The architecture for the ResV1 model can be found in figure 6.

Model	ResV1	ResV1	ResV1	ResV1
Training	99.9	98.0	93.1	81.6
Validation	87.7	90.4	89.5	84.2
Test	88.4	89.5	88.8	84.4
Aug.	None	Easy	Hard2	Hard3

Table 3. Results from residual models

3.4. Bottleneck Results

The bottleneck block was the next architecture tested, with tests for the single bottleneck block using encoding factors of 4. These models closely followed the residual model architectures, with the only difference being the plain residual layers were replaced by bottleneck layers instead. A predicted quality of bottleneck blocks was that they should be faster than standard residual blocks when considering the compression of channels before convolution. This property was clearly observed over the process of experimentation, as although the bottleneck block performance was lower than plain residual blocks, the average training time to completion for bottleneck blocks was 22.6 minutes compared to the 55 minutes for plain residual blocks. The bottleneck models shown in table 4 both had effective encoding factors of 8, where the single bottleneck had a direct encoding factor of 8 while the DBN had encoding factors of 4 and 2. Additionally, a standard bottleneck network was tested with a single encoding factor of 4 to compare the differences in true encoding factors. Models beyond this point always ended up overfitting without data augmentation, so the None augmentation variant will be omitted unless otherwise specified. As expected, bottleneck models performed better overall with less channel compression, as the BN4 models outperformed models with higher compressions like BN8. An unexpected observation was the fact the DBN models seemed to under perform when compared to the standard BN8 models. The lack in performance may be explained by the fact that the bottleneck blocks only need to learn a single encoding and decoding function while the DBN models must learn 2. The encode/decode process may in fact be easier to learn in a single step as opposed to having to learn 2 separate steps that must function well together to produce good results. The architecture for the BN4 model can be found in figure 7.

3.5. Highway Results

Results from highway block performance were interesting, and provided some insight on whether or not longer residual connections are beneficial in deeper models. The

Model	BN8	BN8	BN8	DBN	DBN	DBN
Train	96.3	75.4	77.2	92.6	75.2	61.7
Val.	88.9	76.8	82.1	87.4	77.2	68.2
Test	87.8	77.5	82.2	86.3	76.9	68.4
Aug.	Easy	Hard2	Hard3	Easy	Hard2	Hard3

Table 4. Results from bottleneck vs DBN models

Model	BN4	BN4	BN4
Train	96.5	86.0	73.1
Val.	89.4	85.8	76.9
Test	88.5	85.1	77.2
Aug.	Easy	Hard2	Hard3

Table 5. Results from bottleneck model with encoding factor of 4

HwyV1 and HwyV2 were identical to the previous ResV1 models, however, continuous sections of residual blocks were replaced with highways of identical lengths and different block compositions. The HwyV1 model used bottleneck blocks with encoding factors of 4 while the HwyV2 model had continuous blocks replaced by residual blocks of equal length. Both models performed about as well as their counterparts without highway blocks, however, an interesting observation about the HwyV1 model is that compared to the plain bottleneck implementation, the HwyV1 model had significantly better validation performance on the Hard3 augmentation policy, scoring 3.5% higher. A potential explanation is that the highway block provides some kind of extra regularization which lowers performance overall, but makes the model more robust to variations in data. By adding initial activations to outputs, we mix activations between learned layers and residual activations, which can allow previous layers to exert influence on decisions made by the model in the current layer. The longer connection gets added with the outputs from the block as a whole, so allowing a much longer propagation of activations through the network can be seen as reinforcing activations from previous layers. This reinforcement of previous activations can help already learned activations persist for longer since they don't have to survive multiple combinations with other gradients when passing through other layers. Additionally, activation maps that pass through the highway connection have been through fewer layers when compared to the activations coming through the main sequential block, effectively "shortening" the true number of layers an activation map passes through leading to a more regularized model. Of course, the inverse is also true, as while highway blocks allow activation maps to persist for longer, the current activations may not always be beneficial at different points in the network, as combining less abstract features with more

abstract features can complicate learning further down the network. The architecture for the HwyV2 model can be found in figure 8.

Model	HwyV1	HwyV1	HwyV1
Train	96.6	84.7	75.3
Val.	89.3	83.3	80.3
Test	88.2	84.0	81.0
Aug.	Easy	Hard2	Hard3

Table 6. Results from highway V1 models

Model	HwyV2	HwyV2	HwyV2
Train	97.7	94.9	83.7
Val.	90.0	89.6	84.4
Test	89.2	89.1	84.1
Aug.	Easy	Hard2	Hard3

Table 7. Results from highway V2 models

3.6. Branch Results

Branch networks were tested with a few different variations, each producing different results[1]. Two variants were tested, with V1 variants simply concatenating the channels of each branch to produce one larger output, while V2 variants had all branch outputs averaged to produce an activation map of the same size as the input. These two variants were also tested with and without batch normalization of each individual branches output, denoted by the N in the model name. These models consisted of convolution layers to increase the number of channels with branch blocks in between to perform the main learning. Each branch block had 3-6 branches which consisted of bottleneck blocks since they were quick to train and would reduce the run-time of the model. Averaging normalized branch activations produces the best results based on the BNV2 model set, with models performing significantly better on heavily augmented data, suggesting the branch activation averaging was effective at reducing the variance between layers. It should also be noted that the overall size of the BNv2 model is essentially equivalent to the previous ResV1 model, meaning the performance gains for heavily augmented data stem from the averaging process across multiple parallel branches. The branch block should also be easily adaptable to other architectures, and so can provide solid regularization without much additional overhead. The architecture for the BNV2 model can be found in figure 9.

[1]Saving model state failed for normalized branch blocks, but results are likely to be similar to existing validation metrics.

Model	BV1	BV1	BV1	BV2	BV2	BV2
Train	92.9	84.4	76.6	89.9	84.1	73.4
Val.	87.9	85.0	82.5	87.3	85.6	79.3
Test	87.1	85.3	82.3	87.2	85.0	79.7
Aug.	Easy	Hard2	Hard3	Easy	Hard2	Hard3

Table 8. Results from Branch models

Model	BNV1	BNV1	BNV1	BNV2	BNV2	BNV2
Train	97.8	92.5	79.6	96.6	91.3	83.0
Val.	87.4	88.7	85.2	88.6	89.5	87.7
Aug.	Easy	Hard2	Hard3	Easy	Hard2	Hard3

Table 9. Results from Branch Normalized models

4. Discussion and Conclusions

One of the main takeaways from this project was a deeper understanding behind the process of creating and training deep neural networks from scratch. Creating and evaluating new model architectures is a time consuming process, and different architectures can have their respective advantages and disadvantages, many of which aren't always obvious. While architectures analyzed proved effective at boosting model performance, many resulted in overfitting of the training set, and regularization via data augmentation and implicit ensembles proved effective at boosting raw performance while increasing robustness against dataset variations. Branch blocks in particular proved highly effective, providing the highest validation accuracy, even on the most difficult data augmentations when compared to other models. Although the project was successful overall, there are still several variations techniques that weren't explored that could likely push model performance above the 94% mark. In depth hyperparameter tuning is an obvious path forward, as this project mainly focused on architecture and augmentation development and analysis. Further optimization of the arrangement of network layers is also a promising option. Though the architectural blocks themselves were optimized for performance, their actual arrangement in the network is something that can be explored further. In reality, deep learning requires a mixture of several techniques in tandem, and so applying combinations of all of the above will likely yield better results in the future.

5. Statement of Contributions

As I am the only person in my group, I was responsible for all infrastructure development, model architecture development, and analysis of results.

6. OPTIONAL: Infrastructure Development

While not directly related to the main report, there were many interesting challenges and lessons from this project that came from developing infrastructure to train the models, and I felt it would be a shame to leave it out since it was an enlightening experience. I have included it here as optional reading if the reader is curious, but the section can be omitted since this goes past the 6 page limit.

Substantial effort was spent on developing robust infrastructure to maximize GPU utilization by allowing multiple models to train in parallel on the same GPU. While this process is not a core focus of the project, it played a significant role in the progression of the project, and plays a large part in the development of model training as a whole. The development of such infrastructure was an enlightening experience, as it provided many interesting challenges that had to be handled to create a successful training pipeline.

The infrastructure developed for this project follows a parallel training approach where multiple models can be trained on a single GPU to maximize its usage. While seemingly trivial, managing hardware utilization during model training is critical to ensure maximum performance. When training models, there are 2 main types of VRAM that PyTorch models use to store the parameters and activations, data, and any tensors needed by the model. Dedicated VRAM exists directly on the GPU, and is the fastest for training since there is minimal latency for transferring data from memory to hardware where it can be operated on. Shared VRAM is another kind of memory that doesn't exist on the GPU, instead, it exists in RAM and will be utilized if the GPU has no VRAM available. If any models exist partially or entirely on shared VRAM, training performance takes significantly longer, as the training process is now bottle necked by the latency incurred from transferring model parameters and training data between the GPU and RAM. The additional latency from split models varied during experimentation, but individual epochs took anywhere between twice as long and 20 times as long to complete depending on the model.

Another related limiting factor is the rate at which data is transferred from main storage to the GPU. Previous versions of the training pipeline did not take this into account, and so each epoch ended up requiring the entire dataset to be loaded from storage to memory which made epochs take as long as 80s/iter. Solving this bottleneck involved ensuring the entire dataset was loaded onto the GPU, then performing data augmentations on the GPU without having to transfer data between memory and VRAM. This resulted in some of the smallest models being able to train at a rate of 5s/iter which was a 40 times speedup overall.

To ensure all models are trained on the GPU only, a queuing system was implemented to only allow models with estimated memory usages to begin training only if they wouldn't use all the remaining VRAM. One issue encountered during this process was that model profiling tools provided by PyTorch and other libraries often did not seem to align with real observed model memory usage due to intermediate tensors or dataset tensors that took up additional memory on the GPU. This meant guaranteeing that models remained completely on the GPU was more inefficient, as assumptions were made that the real model memory usage would be larger than the estimates provided by profiling tools, and so memory could go unused even if a model could still be trained without other repercussions.

Once initial memory estimates for each model were created, the next step was to decide on a queuing order for models to be trained to ensure that the GPU was being utilized as much as possible. While finding a truly optimal ordering for model training would be nearly impossible as it is difficult to asses how long a model will take in training without actually training it, the problem of maximizing GPU utilization has parallels with the bin packing problem, where the goal is to decide what sets of numbers can fill a bin of a fixed capacity without exceeding the capacity of the bin. In this case, it was deciding which of the models in the training queue could reasonably fit on the GPU without fragmenting into the shared VRAM. The Best-Fit-Decreasing (BFD) algorithm used to provide solutions for the bin packing problem has a useful heuristic which is to assume the best object to pack into a bin is the object that minimizes the remaining space in the bin, thus maximizing utilization of remaining space. Applying this to the training pipeline meant that models were sorted by memory usage in descending order and placed into the training queue depending on how well they fit into the remaining VRAM. If no more room was available in the GPU, then new models waited until space was available, and would then be added to the training pool based on the BFD heuristic.

Overall, the developed system proved to be relatively effective, although since the true memory usages were hard to evaluate, there is definitely room for improvement. Model sizes couldn't accurately be assessed, so a cap of 3 to 4 models in training at once was set to ensure that growing GPU memory usage from memory leaks or inefficiencies wouldn't fill up the available VRAM leading to the performance degradation seen when models become split across shared and dedicated VRAM. Even though the system isn't perfect, it still provides significant parallelism in model training which was crucial in evaluating and iterating on models in this project.

7. Additional Figures

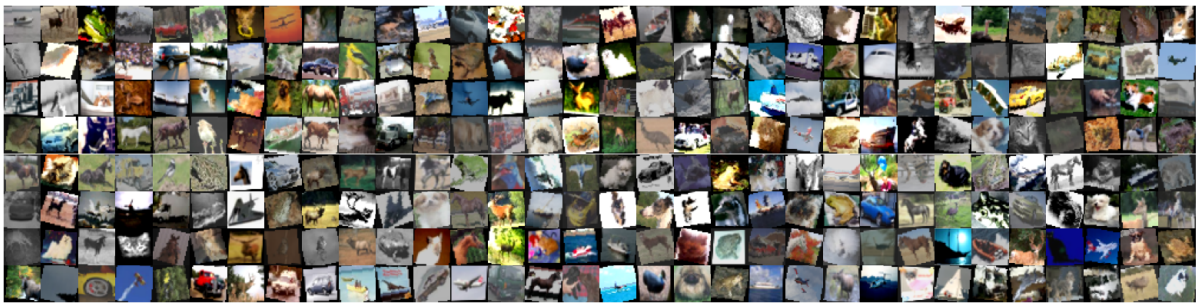


Figure 2. Easy augmentation on CIFAR-10 samples

A derivation of the DBN FLOPs:

$$\text{Conv}\left(1, C, \frac{C}{N}\right) = \frac{1}{N} HWC^2 \qquad \text{Encode 1}$$

$$\text{Conv}\left(1, \frac{C}{N}, \frac{C}{MN}\right) = \frac{1}{MN^2} HWC^2 \qquad \text{Encode 2}$$

$$\text{Conv}\left(3, \frac{C}{MN}, \frac{C}{MN}\right) = \frac{9}{M^2N^2} HWC^2 \qquad \text{Convolution}$$

$$\text{Conv}\left(1, \frac{C}{MN}, \frac{C}{N}\right) = \frac{1}{MN^2} HWC^2 \qquad \text{Decode 2}$$

$$\text{Conv}\left(1, \frac{C}{N}, C\right) = \frac{1}{N} HWC^2 \qquad \text{Decode 1}$$

```python
easyaugmentation = v2.Compose([
        v2.RandomPerspective(distortion_scale=0.2, p=0.5),
        v2.RandomGrayscale(p=0.1),
        v2.RandomRotation(degrees=(0, 15)),
        v2.RandomHorizontalFlip(p=0.5),
        v2.ColorJitter(brightness=0.3, contrast=0.5, saturation=0.3),
])

hardAugmentation2 = v2.Compose([
    v2.RandomPerspective(distortion_scale=0.4, p=0.5),
    v2.RandomGrayscale(p=0.1),
    v2.RandomHorizontalFlip(p=0.5),
    v2.RandomVerticalFlip(p=0.2),
    v2.RandomInvert(p=0.2),
    v2.ColorJitter(brightness=0.2, contrast=0.5, saturation=0.4),
])

hardAugmentation3 = v2.Compose([
    v2.RandomPerspective(distortion_scale=0.3, p=0.5),
    v2.RandomGrayscale(p=0.1),
    v2.RandomHorizontalFlip(p=0.5),
    v2.RandomVerticalFlip(p=0.2),
    v2.RandomInvert(p=0.2),
    v2.RandomResizedCrop(size=(32, 32), scale=(0.3, 1), antialias=True),
    v2.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.4),
])
```

Figure 3. The Easy, Hard2, and Hard3 data augmentation policies.

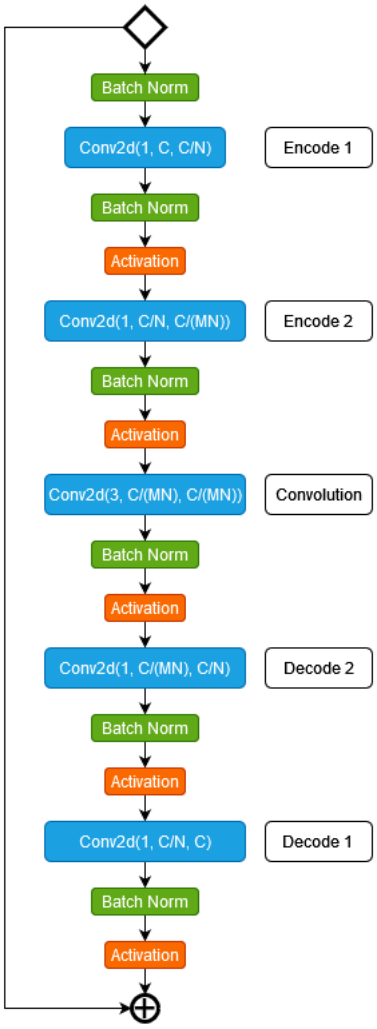


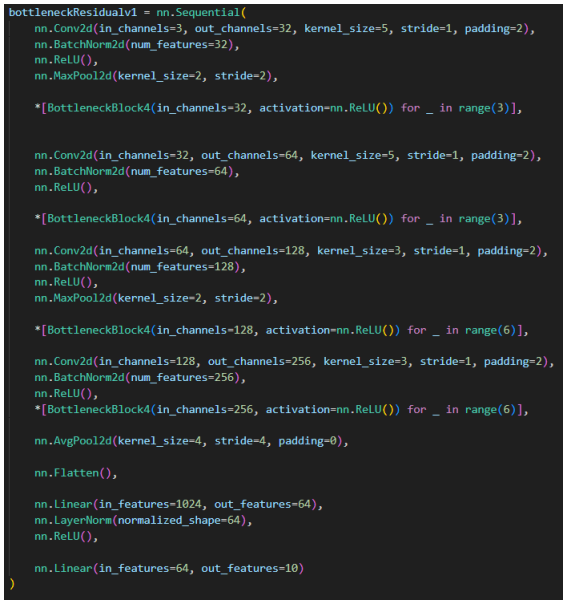
Figure 4. DBN Architecture

```
baseline430kN = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=16),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.ReLU(),

    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=32),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.ReLU(),

    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=64),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.ReLU(),

    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),

    nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),

    nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=128),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.ReLU(),

    nn.Flatten(),

    nn.Linear(in_features=512, out_features=64),
    nn.LayerNorm(normalized_shape=64),
    nn.ReLU(),

    nn.Linear(in_features=64, out_features=10)
)
```

Figure 5. Baseline 430k model

```
residualNetv1 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    *[ResidualBlock(channelCount=32, activation=nn.ReLU()) for _ in range(3)],


    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=64),
    nn.ReLU(),

    *[ResidualBlock(channelCount=64, activation=nn.ReLU()) for _ in range(3)],

    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=2),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    *[ResidualBlock(channelCount=128, activation=nn.ReLU()) for _ in range(6)],

    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=2),
    nn.BatchNorm2d(num_features=256),
    nn.ReLU(),
    *[ResidualBlock(channelCount=256, activation=nn.ReLU()) for _ in range(6)],

    nn.AvgPool2d(kernel_size=4, stride=4, padding=0),

    nn.Flatten(),

    nn.Linear(in_features=1024, out_features=64),
    nn.LayerNorm(normalized_shape=64),
    nn.ReLU(),

    nn.Linear(in_features=64, out_features=10)
)
```

Figure 6. Residual net v1 architecture

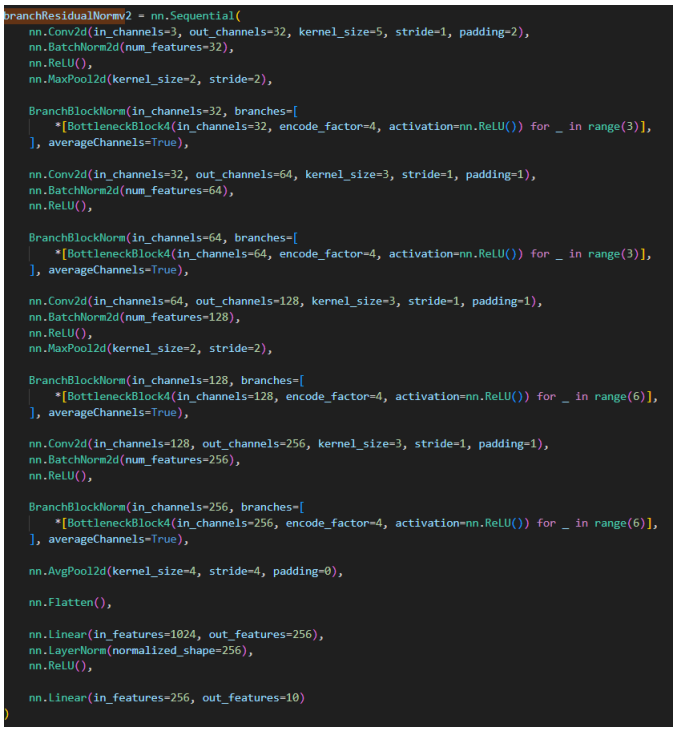


```
bottleneckResidualv1 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    *[BottleneckBlock4(in_channels=32, activation=nn.ReLU()) for _ in range(3)],


    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=64),
    nn.ReLU(),

    *[BottleneckBlock4(in_channels=64, activation=nn.ReLU()) for _ in range(3)],

    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=2),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    *[BottleneckBlock4(in_channels=128, activation=nn.ReLU()) for _ in range(6)],

    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=2),
    nn.BatchNorm2d(num_features=256),
    nn.ReLU(),
    *[BottleneckBlock4(in_channels=256, activation=nn.ReLU()) for _ in range(6)],

    nn.AvgPool2d(kernel_size=4, stride=4, padding=0),

    nn.Flatten(),

    nn.Linear(in_features=1024, out_features=64),
    nn.LayerNorm(normalized_shape=64),
    nn.ReLU(),

    nn.Linear(in_features=64, out_features=10)
)
```

Figure 7. BN4 model

```
highwayResidualv2 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    HighwayBlock(in_channels=32, highwaySequence=nn.Sequential(
        *[ResidualBlock(channelCount=32, activation=nn.ReLU()) for _ in range(3)]
    )),

    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding=1),
    nn.BatchNorm2d(num_features=64),
    nn.ReLU(),

    HighwayBlock(in_channels=64, highwaySequence=nn.Sequential(
        *[ResidualBlock(channelCount=64, activation=nn.ReLU()) for _ in range(3)]
    )),

    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    HighwayBlock(in_channels=128, highwaySequence=nn.Sequential(
        *[ResidualBlock(channelCount=128, activation=nn.ReLU()) for _ in range(6)]
    )),

    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=256),
    nn.ReLU(),

    HighwayBlock(in_channels=256, highwaySequence=nn.Sequential(
        *[ResidualBlock(channelCount=256, activation=nn.ReLU()) for _ in range(6)]
    )),

    nn.AvgPool2d(kernel_size=4, stride=4, padding=0),

    nn.Flatten(),

    nn.Linear(in_features=256, out_features=64),
    nn.LayerNorm(normalized_shape=64),
    nn.ReLU(),

    nn.Linear(in_features=64, out_features=10)
)
```

Figure 8. HwyV2 model architecture



```
branchResidualNormv2 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    BranchBlockNorm(in_channels=32, branches=[
        *[BottleneckBlock4(in_channels=32, encode_factor=4, activation=nn.ReLU()) for _ in range(3)],
    ], averageChannels=True),

    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=64),
    nn.ReLU(),

    BranchBlockNorm(in_channels=64, branches=[
        *[BottleneckBlock4(in_channels=64, encode_factor=4, activation=nn.ReLU()) for _ in range(3)],
    ], averageChannels=True),

    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    BranchBlockNorm(in_channels=128, branches=[
        *[BottleneckBlock4(in_channels=128, encode_factor=4, activation=nn.ReLU()) for _ in range(6)],
    ], averageChannels=True),

    nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(num_features=256),
    nn.ReLU(),

    BranchBlockNorm(in_channels=256, branches=[
        *[BottleneckBlock4(in_channels=256, encode_factor=4, activation=nn.ReLU()) for _ in range(6)],
    ], averageChannels=True),

    nn.AvgPool2d(kernel_size=4, stride=4, padding=0),

    nn.Flatten(),

    nn.Linear(in_features=1024, out_features=256),
    nn.LayerNorm(normalized_shape=256),
    nn.ReLU(),

    nn.Linear(in_features=256, out_features=10)
)
```

Figure 9. BNV2 model architecture