**Background:**

ResNet provides an innovative solution to the vanishing gradient problem, known as “skip connections”. ResNet stacks multiple identity mappings (convolutional layers that do nothing at first), skips those layers, and reuses the activations of the previous layer.

The residual blocks make it considerably easier for the layers to learn identity functions. As a result, ResNet improves the efficiency of deep neural networks with more neural layers while minimizing the percentage of errors.

**Objective:**

The aim of hyperparameter tuning is to select the optimal combination of hyperparameters to achieve high accuracy and a small loss. As hyperparameter optimization is computationally intensive, it is often impractical to test all hyperparameters. The objective of this paper is to experiment with different hyperparameters on different resnet architecture and observe how the results vary by varying these parameters.

**Related literature:**

Deep NNs are often associated with high errors and optimization difficulty. This is where the residual network comes in. It provides evidence that resnets perform better in terms of both training and top-1 error. For 18 layer resnet, the convergence was faster and optimization was eased.

The contribution of Resnet in machine learning is profound. It alleviates the degradation problem in deep NN by recasting the original mapping. It provides enough clarity for choosing the methods for different architecture. For instance, it has explained why it chose the bottleneck architecture for deeper models(50 layers and above).

The construction of added layers as identity mapping is novel as it correctly assumes that identity mapped deeper layers should not be more error-prone than its shallow counterpart. Some other innovations are shortcut connections and residual representations.

It assumes that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. However, the identity shortcut connections do not add any new parameters to the shallow counterpart. This means even though complexity is reduced, no. of parameters are reduced as well which might affect the representation capacity of the model itself and not generalize not unseen data.

A diagram of a diagram

Description automatically generated

Figure: Residual learning: a building block

**Experiments:**

First, I tried the Resnet9 architecture and the following techniques were used:

* Data normalization

The image tensors were normalized by subtracting the mean and dividing by the standard deviation of pixels across each channel. Normalizing the data prevents the pixel values from any one channel from disproportionately affecting the losses and gradients.

* Data augmentation

Random transformations were applied while loading images from the training dataset. Specifically, we will pad each image by 4 pixels, and then take a random crop of size 32 x 32 pixels, and then flip the image horizontally with a 50% probability.

* Batch normalization

After each convolutional layer, we added a batch normalization layer, which normalizes the outputs of the previous layer. This is somewhat similar to data normalization, except it's applied to the outputs of a layer, and the mean and standard deviation are learned parameters.

* Learning rate scheduling

Instead of using a fixed learning rate, a learning rate scheduler was used which will change the learning rate after every batch of training. There are many strategies for varying the learning rate during training, and we used the "One Cycle Learning Rate Policy".

* Weight Decay

Weight decay was added to the optimizer, yet another regularization technique which prevents the weights from becoming too large by adding an additional term to the loss function.

* Gradient clipping

Gradient clipping was added, which helps limit the values of gradients to a small range to prevent undesirable changes in model parameters due to large gradient values during training.

* Adam optimizer

Instead of SGD (stochastic gradient descent), Adam optimizer was used which uses techniques like momentum and adaptive learning rates for faster training.

Second, I used the Resnet34 architecture twice.

The transforms used for the training and training+validation datasets consist of resizing the images to the required resolution by our ResNet model (224x224), using the AutoAugment policy learned on the CIFAR10 dataset and finally converting the image from PIL to Tensor. For the validation and test sets we just resize the image and convert it to Tensor format.

The train and train+validation DataLoaders use a smaller batch\_size as the gradients need to be kept track of in memory. Furthermore, the dataset was shuffled each epoch to avoid loading the batches in the same order. The valid and test DataLoaders use a larger batch\_size and do not require to shuffle the dataset as we want deterministic results.

The number of workers is generally set to 2 \* num\_gpus as a rule of thumb, with pin\_memory = True to speed up data transfer to the GPU.

The first step of the process is to evaluate the model performance on our own Validation set, consisting of 10% of the labelled data.

ResNet34 model, trained on ImageNet was fine-tuned. Other models might be used, but for the purpose of this notebook a ResNet34 is a good trade-off between training time and model accuracy.  
This model originally has a 1000-dimensional output layer, but our dataset has only 10 classes, so we remove the output layer and define a new Fully-Connected layer with just 10 neurons, one for each class in CIFAR-10. The parameters of these new neurons are initialized with Xavier initialization.

Second, I changed the hyperparameters for the same resnet34 model. I resized the images to 112x112) and did not take random crop since that would make the images very small. I used a larger batch size. I used AdamW optimizer and created my own learning rate scheduler. This custom scheduler (cosine LR scheduler) would enable the optimizer to learn slower towards the last few epochs while learning faster towards the first few epochs.

AdamW shows superior performance over Adam optimizer because of incorporation of weight decay. Adam and AdamW have almost identical source codes. The only difference is Adam’s weight\_decay is deferred to parent class while AdamW’s weight\_decay is defined in the AdamW class itself.

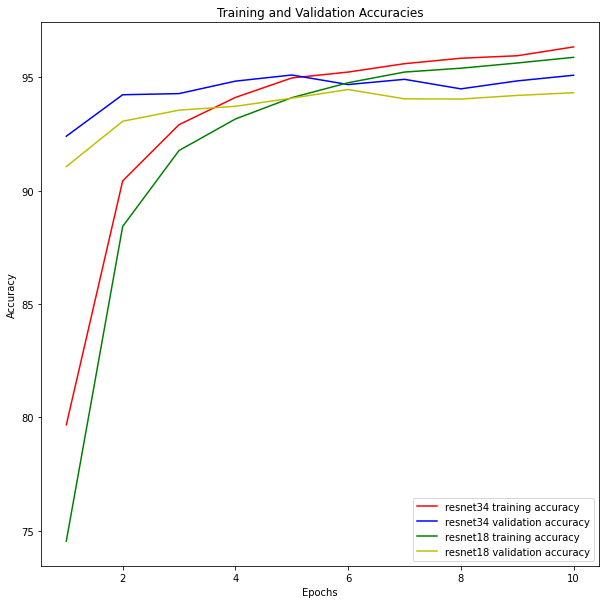


Figure1: Training and Validation Accuracies

A graph of a number of colored lines

Description automatically generated with medium confidence

Fig 2: Training and Validation losses

**Analysis and results:**

Using a combination of learning rate scheduler(one cycle learning rate policy), weight decay and gradient clipping created a 90% accuracy within 5 minutes with only 8 epochs whereas with the vanilla run of the model, it took 80 epochs to achieve this accuracy.

For Resnet34, there was no learning rate scheduler first time but we still used weight decay. The transformation using the autoaugment only resized the images and converted it to tensor but there was no random flip or random crop.

With these changes, training accuracy of 79.8% was achieved with 20 epoch.

However, during the second run of the same ResNet34, using a larger batch size, cosine LR scheduler and adamW optimizer, performance was greatly improved to 95%.

Hence, data augmentation, gradient clipping, learning rate scheduler as well as batch size may play a vital role in achieving high performance like that of Resnet9.

**Conclusion and Future**

From figure 1 and 2, it can be seen the Resnet34 performed better after modifications in the second run. Data augmentation, batch size, learning rate scheduler and optimizer play a vital role in enhancing performance of deeper resnets. In future, I would like to use tools such as Optuna and see how the human handpicked hyperparameters compare with the automated tuning.