Classification with CIFAR-10

This code builds and summarizes a Convolutional Neural Network (CNN) model using Keras with TensorFlow backend. The initial architecture taken was AlexNet which has 5 convolutional layers and 3 fully connected layers. Different experiments were performed with different Architectures of CNN taking AlexNet as the based architecture.

1.1 About the Dataset

The CIFAR-10[1] dataset consists of 6000 images per class in 10 classes totaling 60000 32x32 color images. 10000 test photos and 50,000 training images are available.

Five training batches and one test batch, each with 10,000 photos, make up the dataset. Exact 1000 randomly chosen photos from each class make up the test batch. The remaining images are distributed across the training batches in random order, however, certain training batches can have a disproportionate number of images from a particular class.

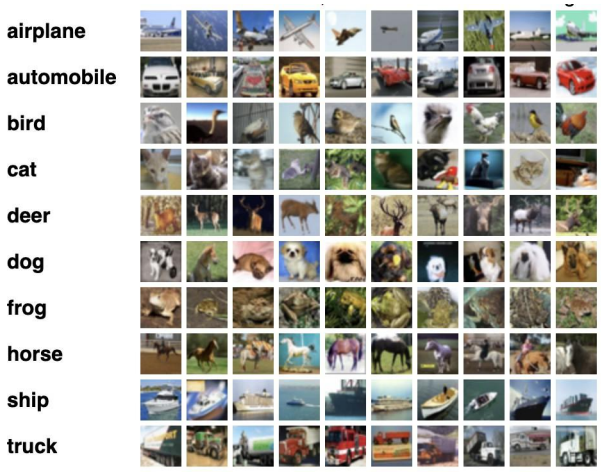

Fig. 1. The Figure shows the classes in the dataset along with 10 random sample images

The training batches consist of exactly 5000 photos from each class combined sample as shown in Fig. 1 above. The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, and things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

1.2 ALEXNET

AlexNet released in the year 2012 demonstrates that a big, deep convolutional neural network can use only supervised learning to achieve world-beating performance on a very difficult dataset. It is noteworthy that removing just one convolutional layer causes our network's performance to suffer. For instance, eliminating any

[1]https://www.cs.toronto.edu/~kriz/cifar.html

BatchSize = 512 , Optimizer = SGD , lr = 0.1		
Epochs	Train Time (in mins)	Val Accuracy
10	6	70.57%
25	14	75.02%
50	27	76.54%
75	43	77.51%
100	56	78.23%
150	82	78.05%

Table 1. The Table shows the running time taken for training for the number of epochs and the final validation accuracy

one of the middle layers reduces the network's top-1 performance by around 2%. So, they inferred depth could be a crucial reason for attaining such results.

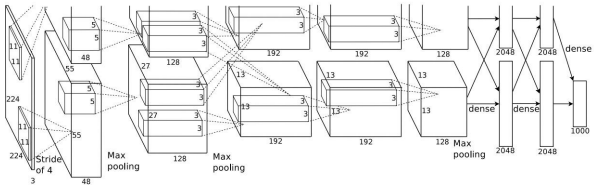

Fig. 2. The Figure shows an illustration of the architecture of ALEXNET CNN

In Fig. 2, the non-linearity they used was ReLu and the choice of optimizer was SGD, which is applied to the output of every convolutional and fully-connected layer. Our tests begin here, we take this as our base architecture and begin training the same for the CIFAR-10 dataset.

The reason we are taking this as the starting point is that the architecture is a realistic benchmark of what can be computed and run with the resources available at our disposal. We would either reduce or increase the number of layers from this and still be able to get the results for computation.

1.3 Number of Epochs

We start with one of the most important hyperparameters, the number of epochs. An epoch is the total number of iterations required to train the machine learning model using all of the training data at once. It is measured in cycles.

The number of trips a training dataset makes around an algorithm is another way to define an epoch. When the data set has made both forward and backward passes, one pass is counted. It specifies how many times the learning algorithm must process the complete data collection.

We decided we would go with accuracy as our base metrics as the initial Data Exploration did not show any class imbalance. We would have proceeded with F1-score if there existed any class imbalance in the set. The following tests were performed on Google Colab - GPU instance.

Now with respect to the experiments we did with increasing the number of epochs and measuring the performance, we could very well see from Fig. 3 and Table. 1 the validation accuracy seems to increase from 10 to 25 epochs by a good margin and only a little margin otherwise. On the other hand, the losses start to increase and reach new peaks in the 75 epoch settings.

What seems like the next obvious step is to play with the optimizer to see if there are any other combinations producing better accuracy in the validation/test set with respect to Fig. 3.

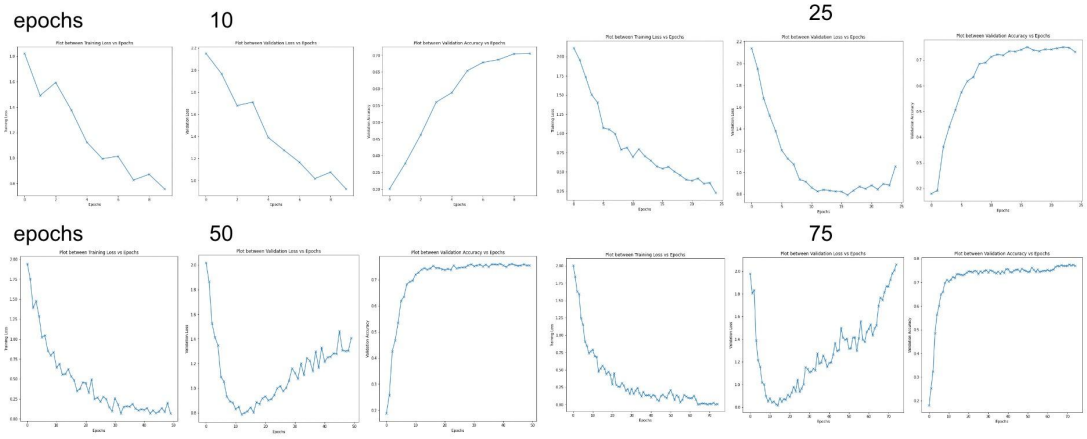

Fig. 3. This shows the change in the Train loss, Val loss, and Val accuracy for each epoch with respect to different epochs

1.4 Optimizers

1.4.1 SGD Optimizer SGD (Stochastic Gradient Descent) is used to minimize the loss function of the neural network during training. It works by updating the model's parameters (weights and biases) in the direction of the steepest gradient of the loss function with respect to those parameters. This means that the algorithm tries to find the minimum point of the loss function by iteratively updating the parameters in small steps, with each step being proportional to the gradient of the loss function at the current point. In SGD, a small subset of the training data (a batch) is used to calculate the gradient of the loss function at each iteration. This makes it more computationally efficient than batch gradient descent, which computes the gradient using the entire training dataset. However, since the subset is chosen randomly, SGD can lead to more oscillations during training, and it may take more iterations to converge to the minimum point.

1.4.2 Adagrad optimizer The Adagrad optimizer adapts the learning rate of each parameter based on the historical gradient information. It achieves this by scaling the learning rate of each parameter by the inverse square root of the sum of the squared gradients of that parameter up to the current time step. This means that parameters that have been frequently updated will have a smaller learning rate, while parameters that have been infrequently updated will have a larger learning rate. The Adagrad optimizer is particularly useful in settings where the gradients of different parameters have a wide range of magnitudes, as it automatically adapts the learning rate to each parameter. Additionally, Adagrad does not require the manual tuning of a learning rate hyperparameter, which can be time-consuming and error-prone. One limitation of Adagrad is that it accumulates the squared gradients over time, which can lead to a diminishing learning rate over time

1.4.3 Adadelta optimizer It is an extension of the Adagrad optimizer that seeks to address some of the limitations of Adagrad, particularly the problem of a diminishing learning rate over time. The Adadelta optimizer uses a similar approach to Adagrad by adapting the learning rate of each parameter based on the historical gradient information. However, instead of accumulating the squared gradients, Adadelta accumulates a running average of both the squared gradients and the parameter updates. This running average serves as an estimate of the second moment of the gradients, which is used to calculate the learning rate for each parameter. One advantage of Adadelta over Adagrad is that it does not require an initial learning rate hyperparameter, which can be difficult to tune in practice. Instead, Adadelta uses two new hyperparameters: a decay rate that controls the averaging of

the historical information, and a small constant epsilon that prevents division by zero. Another advantage of Adadelta is that it does not accumulate the squared gradients over time, which can reduce the amount of memory required to train large models. Additionally, the running average of the parameter updates can help Adadelta overcome the issue of a diminishing learning rate over time, which can lead to better convergence rates.

1.4.4 RMSprop optimizer RMSprop adapts the learning rate of each parameter based on the historical gradient information. However, RMSprop uses a moving average of the squared gradients to compute the learning rate for each parameter. Specifically, RMSprop divides the learning rate by the root mean square (RMS) of the past gradients for each parameter. One advantage of RMSprop is that it scales the learning rate more appropriately for each parameter, regardless of the magnitude of the gradient. Additionally, RMSprop maintains a moving average of the squared gradients, which allows it to adapt more quickly to changes in the gradient. RMSprop also uses a decay rate hyperparameter to control the rate at which the moving average of the squared gradients is computed. This allows the algorithm to take into account both recent and past gradients in the computation of the learning rate.

1.4.5 Adam optimizer ADAM (Adaptive Moment Estimation) is an optimization algorithm used in deep learning for stochastic gradient descent (SGD) optimization. It is a popular choice for training neural networks due to its ability to converge quickly and handle noisy and sparse gradients. ADAM combines two techniques: momentum and adaptive learning rates. Momentum helps the optimizer to keep moving in the right direction even when the gradients are noisy and adaptive learning rates enable the optimizer to adjust the learning rate for each weight based on the previous updates. This combination makes ADAM an efficient and effective optimizer for deep learning. The ADAM optimizer maintains a running average of the gradients and the second moments of the gradients. These running averages are used to calculate the learning rate for each weight. The learning rate is adaptive because it is scaled by the running average of the second moments of the gradients, which helps to smooth out the updates.

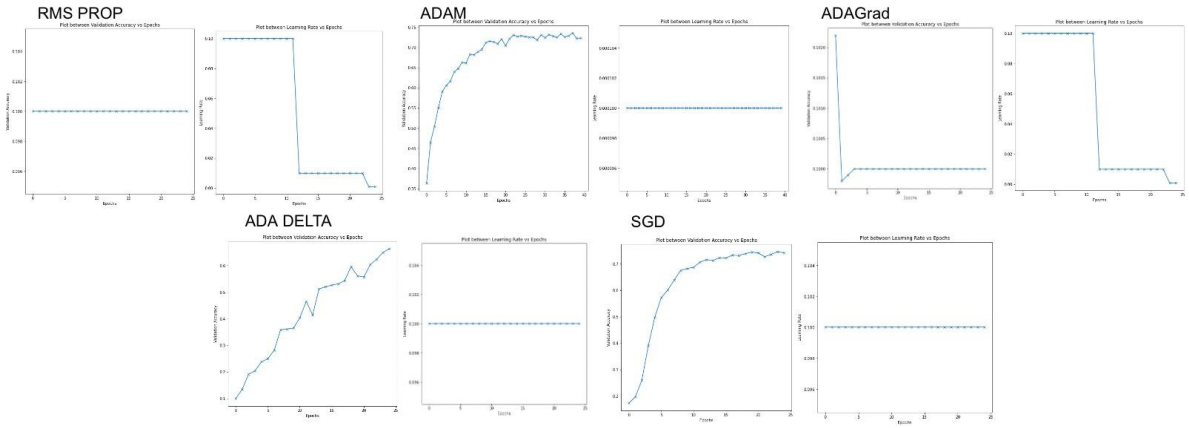

Fig. 4. The Figure shows an illustration of the architecture of ALEXNET CNN

1.5 Adding momentum and Reduce on Plateau for Learning Rates

As the last set of experiments was conducted with a constant learning rate of 0.1, we then test all the architectures with a momentum of 0.9 and the reduction in learning rate by 0.1. This is being monitored with respect to the

max of validation accuracy for 25 epochs. We specifically choose 25 epochs to compare against the graph Fig. 3 where the validation accuracy increases 25 so epochs only by a little amount.

A momentum of 0.9 was selected so that it builds inertia in a search direction to overcome local minima and oscillation of noisy gradients, especially in cases like SGD. Reduce on Plateau was implemented so we don't skip out on global optima.

We tried different optimizers mentioned above for 25 epochs except for Adam, as the convergence in Adam was faster we tried the same for 40 epochs (25 was also experimented with and the results were similar).

Looking at Fig. 4, adam converged quickly and for 40 epochs and averaged at an accuracy of 72%. SGD even if the convergence is a little bit lossy gave an accuracy of close 75%. So the AlexNET architecture with SGD seems like a better option to go with. It seems the learning rate doesn't change much on plateauing max monitoring of validation / Test Accuracy. Reducing the Learning rate on the same configuration takes much time and more epochs to converge. So we proceed with the learning rate of 0.1 and let it come down, if the validation accuracy saturates (we have tested this for higher epochs!).

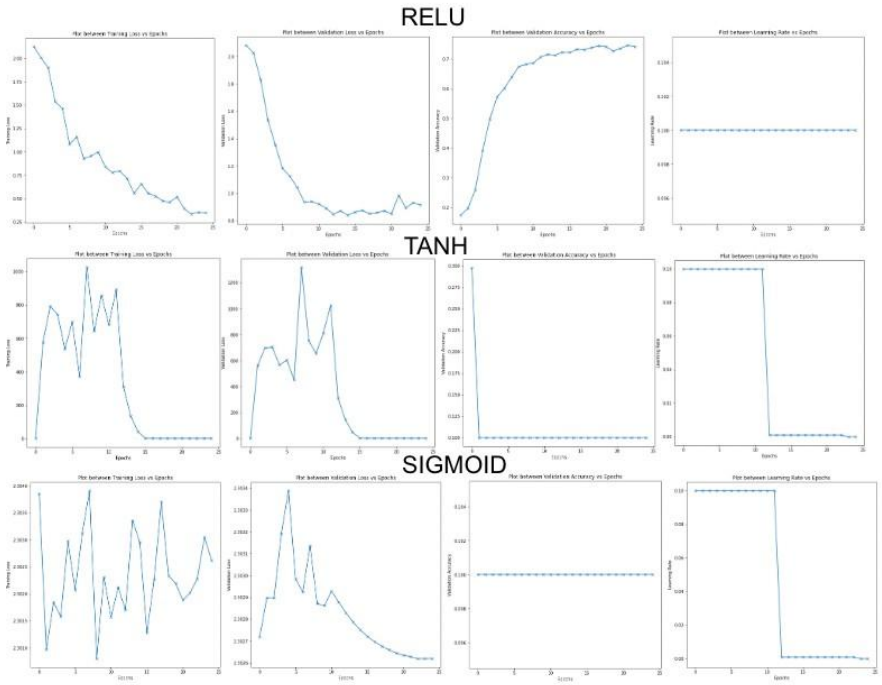

Fig. 5. The Figure shows the output for 25 epochs ALEXNET with the same configuration as before just changing the non-linearity

1.6 Choice of Non-Linearity

The activation function used in the convolutional layer is Relu, the main purpose of activation functions in CNNs is to transform the weighted sum of inputs and biases into a nonlinear output. The output from a convolutional layer is typically fed into an activation function to introduce nonlinearity into the output, which enables the CNN to learn more complex representations of the input data. This nonlinearity is critical for CNN to learn

BatchSize = 512 , Optimizer = SGD , lr = 0.1, True Prediction Rates				
Label	25 Epochs	50 Epochs	75 Epochs	100 Epochs
Airplane	0.792	0.831	0.813	0.81
Automobile	0.892	0.858	0.867	0.877
Bird	0.654	0.662	0.708	0.72
Cat	0.511	0.495	0.591	0.604
Deer	0.734	0.722	0.749	0.731
Dog	0.707	0.649	0.673	0.665
Frog	0.837	0.846	0.83	0.841
Horse	0.725	0.77	0.798	0.818
Ship	0.842	0.839	0.867	0.878
Truck	0.73	0.819	0.833	0.855

Table 2. The Table shows the True Prediction rate for each class for different epochs

more complex patterns and features in the input data. Other than ReLU there are a few other popular activation functions like tanh, and sigmoid.

1.6.1 ReLU (Rectified Linear Unit)

$$( ) = \quad (0, \ )$$

It is widely used in deep learning due to its simplicity and computational efficiency. One of the main advantages of ReLU is that it helps to solve the vanishing gradient problem, which can occur with other activation functions like sigmoid and tanh.

1.6.2 Tanh (hyperbolic tangent)

$$( ) = \frac{e \ - e^{-}}{e \ + \ e^{-}}$$

It is a scaled and shifted version of the sigmoid function and is used in neural networks to introduce non-linearity into the model. However, it can suffer from the vanishing gradient problem, especially when the input values are large.

1.6.3 Sigmoid

$$f\,(x) = \frac{1}{1+e^{-}}$$

It is commonly used in binary classification problems, where the output is a probability between 0 and 1. However, it suffers from the vanishing gradient problem, especially when the input values are very large or very small.

We work with the SGD optimizer and train it for 25 epochs and experiment with the results by changing the choice of nonlinearity. All the choices from Fig. 5 have a similar training time averaging 15mins. Here we can see that the convergence seems to be good for the model with RELU activation than the one with TanH and Sigmoid. If you observe the sigmoid saturates or there is not much improvement beyond a point. The RELU if you observe doesn't have that. From this, we can agree that the network with ReLU will perform better because of the non-saturating nature of the Rectified Linear Unit. Further tests can be performed with other non-saturating functions like Leaky ReLU or ELU as further steps.

1.7 Higher Epochs

Increasing the number of epochs doesn't seem to be a good option, at least with the discussed architecture configuration, as validation loss also keeps increasing after a point with plateauing validation Accuracy after a

slight increase around 60-75 epochs Fig. 6. The learning rate reduces a lot of times from 0.1 to 0.01 and further till $1.0x10^{-7}$.

This justifies our hypothesis: Assuming the choice of learning rate and also validates that we have not missed on the global optima.

1.8 Recommended Architecture

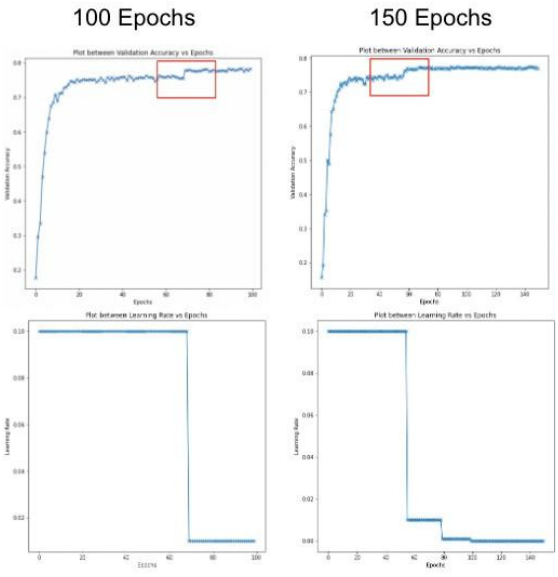

Fig. 6. The Figure shows the accuracy for higher epoch rates such as 100 and 150 at the top and the changing learning rates as the validation accuracy using the function "ReduceLROnPlateau" stagnates below.

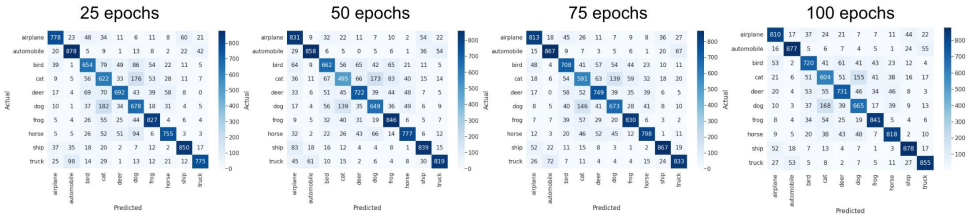

Fig. 7. The Figure shows the classwise confusion matrices for each epoch value for the parameter configurations we have shortlisted

With respect to all the experiments conducted above, it seems we can proceed with AlexNET, with SGD as the choice of the optimizer, Momentum of 0.9, the Learning rate of 0.1, ReLU as the choice of Non-Linearity, Reduce Learning Rate on plateau monitoring validation loss and training the same for 75 epochs. as this seems like an optimal model with a good balance of training time, training loss, validation loss, and validation accuracy.

Further, if we want to stretch a little bit with accuracy with an increase in terms of training time, then we can use the above-mentioned configuration and train it for 100 epochs, where the learning rate would reduce and provide an accuracy of 78%. This decision was taken taking into consideration the class-wise true prediction rates and the confusion matrix of the model in the validation sets refer Figs. 6, 7 and Table. 2. The model trained for 75 and 100 epochs for the above-specified parameter configuration provides balanced prediction results for all classes with a True prediction rate greater than 60% for all the classes on the validation/test set.

$$(\qquad\qquad) = \frac{\qquad\qquad}{\qquad\qquad + \qquad\qquad}$$