CIFAR dataset

CIFAR-10 consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The 10 classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset is split into 50,000 training images and 10,000 test images, with the classes balanced across both sets. The images are in RGB format, meaning that each image has three color channels (red, green, and blue) with 8 bits per channel.

Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a type of deep neural network that is commonly used for image classification, object detection, and other computer vision tasks. It is a specialized type of neural network that is designed to process and analyze image data by applying a series of convolution filters to input images.

Convolution operation involves sliding a set of small filters over the input data to create a set of feature maps, which highlight specific patterns in the data. The feature maps are then passed through additional layers, including pooling layers and fully connected layers, to classify the input data.

The convolution layers are responsible for detecting features in the input image, such as edges, corners, and other patterns. The pooling layers are used to reduce the spatial dimensions of the feature maps, making the network more efficient and reducing the risk of over-fitting. The fully connected layers are used to classify the image based on the features that have been detected by the convolution layers.

Basic features like horizontal or vertical axis edges are often extracted from the initial layers. This output is then forwarded to the subsequent layer, which finds more intricate characteristics like corners or multiple edges. When we delve farther into the network, it can recognize more intricate aspects like objects, faces, etc.

CNN Architecture

The core building block of a CNN is the **convolutional layer**, which applies a set of filters to the input to extract features from it. The convolutional layer applies a set of learnable

filters to the input data to extract local features. Each filter is a small matrix of weights that is convolved with a local patch of the input data. The output of a convolutional layer is a set of feature maps, each of which corresponds to one filter applied to the input. The feature maps capture different input aspects, such as edges, corners, and texture.

Convolutional layers can be stacked on top of each other to form a deep network that can learn complex representations of the input. The output of one convolutional layer can be fed as input to another convolutional layer, which can learn higher-level features by combining the local features extracted by the previous layer.

Pooling layers are commonly used in Convolutional Neural Networks (CNNs) to reduce the spatial size of the feature maps produced by the convolutional layers. The most common type of pooling operation is max pooling, which selects the maximum value from a local patch of the feature map and outputs it as a single value in the pooled feature map. Pooling layers can also reduce the number of parameters in the network and prevent overfitting by removing some of the noise and redundancy in the feature maps.

Fully connected layers are a type of layer commonly used in neural networks, including Convolutional Neural Networks (CNNs), to connect the output of one layer to the input of the next layer. Fully connected layers connect every neuron in one layer to every neuron in the next layer, which means that the output of each neuron in the previous layer is used as input for every neuron in the next layer. Fully connected layers can be used to learn complex nonlinear relationships between the features extracted by the previous layers and to produce a final output or prediction.

Process

First, load the CIFAR-10 dataset. In this case, the neural network has 3 convolutional layers with batch normalization, followed by 2 fully connected layers. We then define a neural network model called AlexNet using the PyTorch nn.Module class.

1. **nn.Conv2d** - defines 2D convolutional layers with the specified number of input channels, output channels, kernel size, and padding.
2. **nn.BatchNorm2d** - layers apply batch normalization to the output of the convolutional layers.

3. **nn.Linear** - defines fully connected layers with the specified number of input and output units.
4. Then input data is passed through the convolutional layers, batch normalization layers, and pooling layers. Output is then flattened and passed through the fully connected layers. The final output is the result of passing the data through the last fully connected layer.
5. Additionally, MaxPool2d layer is used for down sampling the spatial dimensions of the output of the convolutional layers by taking the maximum value of non-overlapping rectangular regions of the output.
6. The ReLU activation function is used after each convolutional layer and fully connected layer. Lastly, the final layer of the model has 10 output units, which corresponds to the 10 classes in the CIFAR-10 dataset.

Now we define the loss function, optimizer, and device for training the AlexNet neural network model.

1. **CrossEntropyLoss()** - computes the cross-entropy loss between the predicted class probabilities and the true class labels.
2. **optim.SGD** - specifies the stochastic gradient descent optimizer to update the parameters of the model during training.

Training the Anet:

1. Training is performed using mini batches of size 32 to speed up the training process. For each mini batch optimizer is used to update the parameters of the model based on the loss function computed using the cross-entropy loss function.
2. We store the training losses and validation losses after each epoch. We then evaluate the model on CIFAR-10 test dataset and compute validation loss is computed using the cross-entropy loss function.
3. We finally print the total training time is printed at the end of the training process.

Accuracy: The accuracy is calculated by dividing the number of correctly predicted images by the total number of images and multiplying by 100.

Observations and Results

With the help of torch-vision datasets, we loaded CIFAR10 data onto the local system and converted it to Tensors. The data is divided into training and validation portions. Subsequently, we utilized Data-loader, which employs iterable on the data set, so that we could train on it batch-wise. To start multi-process data loading with 5 loader worker processes, divide the data into batches of 10, and set the number of workers to 2.

We have implemented different types of neural networks with various types of activation functions and momentum.

1. 3 Conv+ 2 FC layer structure

```
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 10)

        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.bn1(self.pool(torch.relu(self.conv1(x))))
        x = self.bn2(self.pool(torch.relu(self.conv2(x))))
        x = self.bn3(self.pool(torch.relu(self.conv3(x))))

        x = x.view(-1, 128 * 4 * 4)

        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))

        return x
```

Hyper Parameters: Batch size - 10, Num workers - 2, Criterion - Cross Entropy Loss

Number of epochs - 10

a. **ReLu** as an Activation function

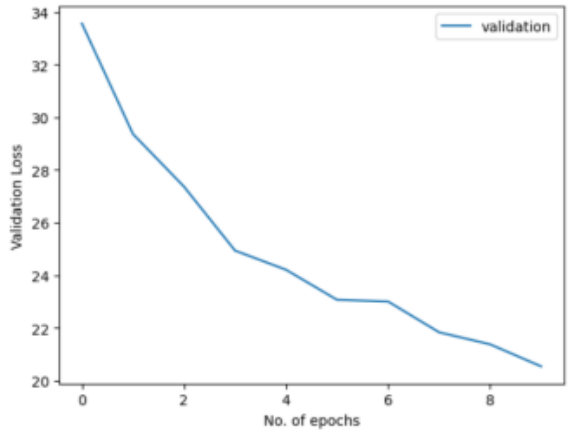
The ReLU activation function is defined as f(x) = max(0, x), where x is the input to the function. ReLU function is that it helps to alleviate the vanishing gradient problem, which can occur in deep networks when gradients become very small during backpropagation.

i. Optimizer - **SGD**

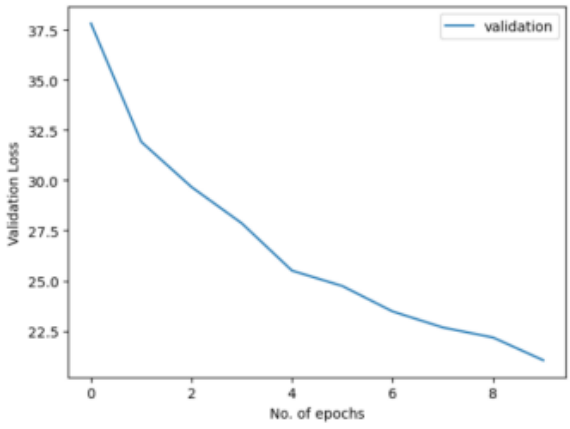
Momentum is a hyper-parameter that helps the optimizer to converge faster by adding a fraction of the previous gradient to the current gradient. This fraction is controlled by the momentum value, which ranges from 0 to 1. When the momentum value is high, the optimizer can overshoot the optimal weights and fail to converge. On the other hand, when the momentum value is too low, the optimizer may take a longer time to converge.

On the tenth iteration, with a constant learning rate of 0.001 and momentum of 0.9, we obtained the best accuracy of 80.01%. It takes around 7.4 min(448.95 sec) for training the data set.

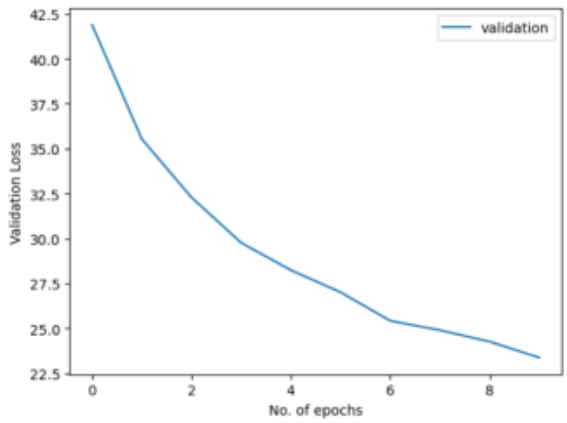
For each training iteration of the data set, we have plotted the validation losses. These graphs represent the validation losses for various momentum values.



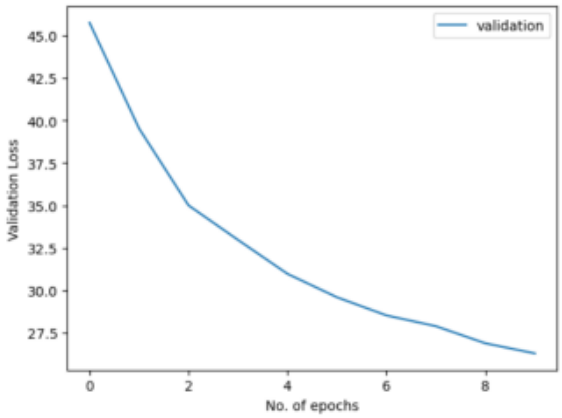
Momentum = 0.9



Momentum = 0.6



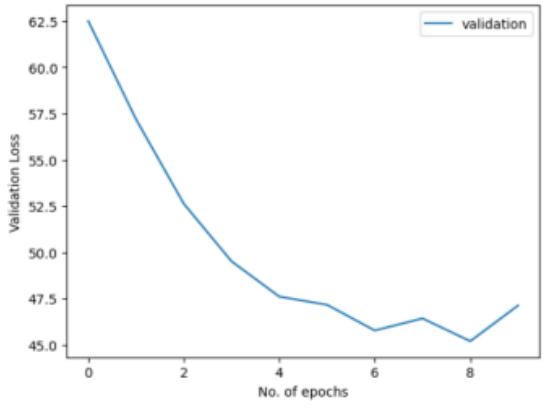
Momentum = 0.3



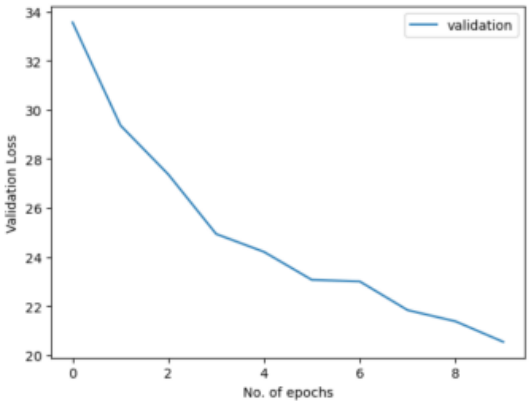
Without Momentum

Momentum	Learning rate	Time (sec)	Accuracy (%)
0.9	0.001	448.951	79.590000 %
0.6	0.001	444.768	77.470000 %
0.3	0.001	443.079	74.980000 %
Without Momentum	0.001	438.435	72.980000 %

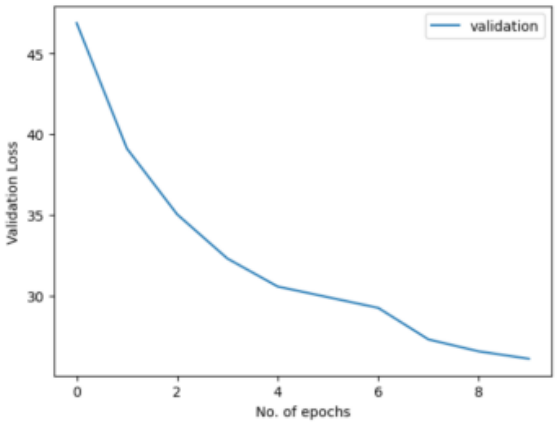
Let's see how the accuracy varies for various learning rates and the associated loss values for each iteration.



Momentum = 0.9, lr = 0.01



Momentum = 0.9, lr = 0.001



Momentum = 0.9, lr = 0.0001

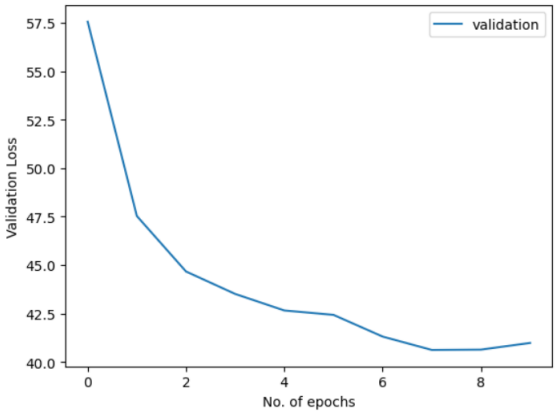
Momentum	Learning rate	Time (sec)	Accuracy (%)
0.9	0.01	439.652	54.260000 %
0.9	0.001	448.951	79.590000 %
0.9	0.0001	439.766	72.590000 %

ii. Optimizer - **Adam**

optimizer = optim.Adam(Anet.parameters(), lr=0.001)

It is an ***adaptive learning rate*** optimization algorithm that combines the benefits of both Momentum and Root Mean Square Propagation (RMSProp) algorithms. The Adam optimizer maintains a separate learning rate for each weight parameter in the network, which allows it to dynamically adjust the learning rate based on the gradient and history of the weights. The algorithm computes an exponentially decaying average of past gradients and squared gradients, which are used to update the parameters during training.

We got an accuracy of 56.45% with the help of the Adam optimizer and it takes 8 min (463.37 sec) to train the data set.

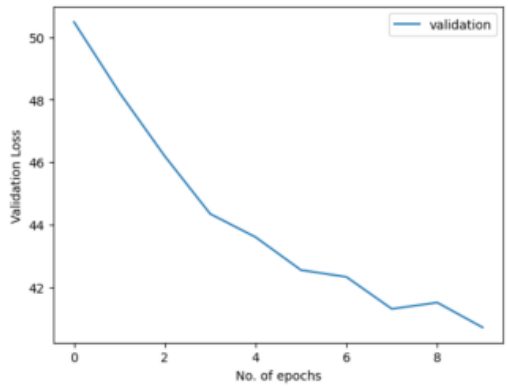


b. **Tan h** as Activation function

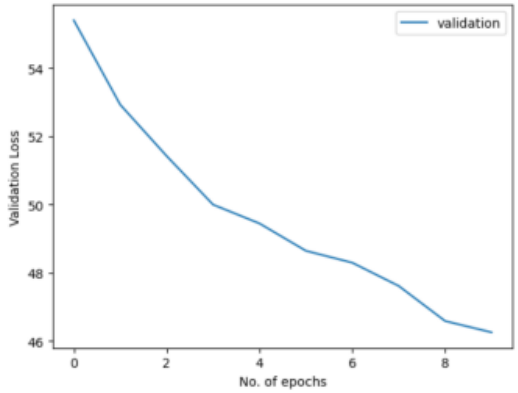
It is a smooth, nonlinear function that maps its input to a range between -1 and 1. Tanh can also help normalize the output of each neuron in a layer, which can prevent the output from becoming too large or too small and improve the stability of the network during training.

i. Optimizer - **SGD**

On the tenth iteration, with a constant learning rate of 0.001 and momentum of 0.9, we obtained the best accuracy of 61.57%. It takes around 7.3 min for training the data set.



Momentum = 0.9



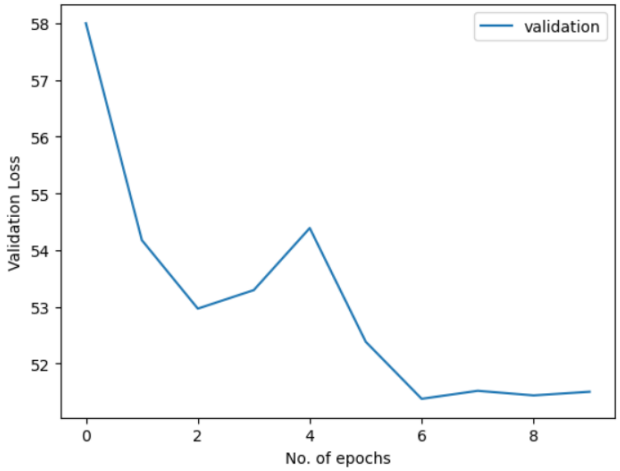
Without Momentum

Momentum	Learning rate	Time (sec)	Accuracy (%)
0.9	0.001	435.342	71.450000 %
No momentum	0.001	425.503	63.810000 %

ii. Optimizer - **Adam**

optimizer = optim.Adam(Anet.parameters(), lr=0.001)

We got an accuracy of 50.01% with the help of the Adam optimizer and it takes 8 min (485.75 sec) to train the data set.

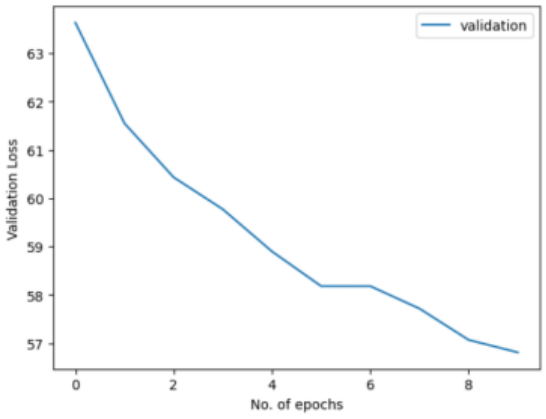


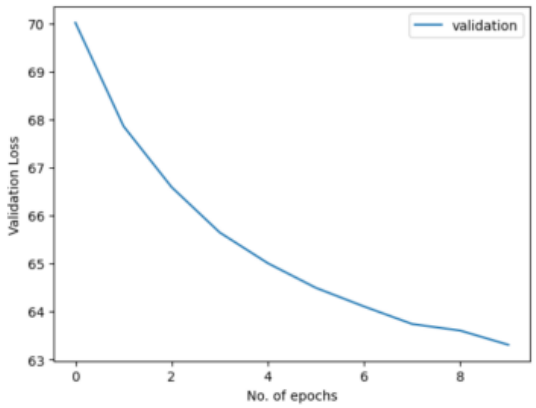
c. **Sigmoid** as Activation function

The sigmoid activation function is a commonly used non-linear activation function in neural networks. The sigmoid function maps any input value to a value between 0 and 1, making it useful for modeling probabilities or binary classification problems.

i. Optimizer - **SGD**

On the tenth iteration, with a constant learning rate of 0.001 and momentum of 0.9, we obtained the best accuracy of 61.57%. It takes around 7.3 min for training the data set.





Momentum = 0.9

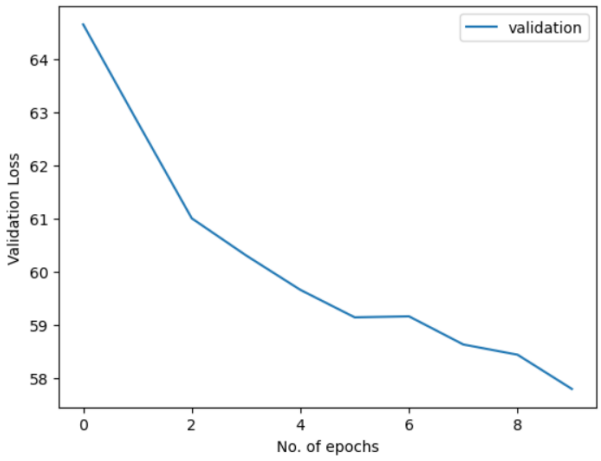
Without Momentum

Momentum	Learning rate	Time (sec)	Accuracy (%)
0.9	0.001	438.733	61.570000 %
No momentum	0.001	428.283	41.140000 %

ii. Optimizer - **Adam**

optimizer = optim.Adam(Anet.parameters(), lr=0.001)

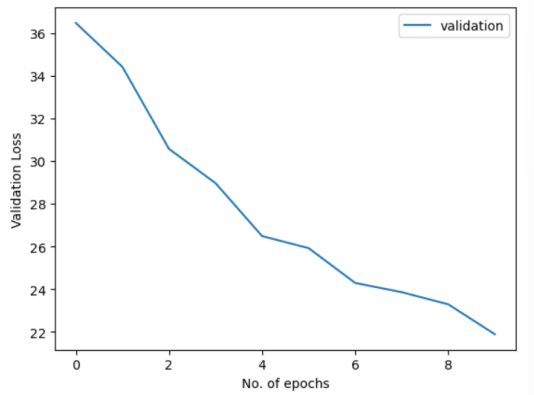
We got an accuracy of 54.09% with the help of the Adam optimizer and it takes 8 min (469.29 sec) to train the data set.

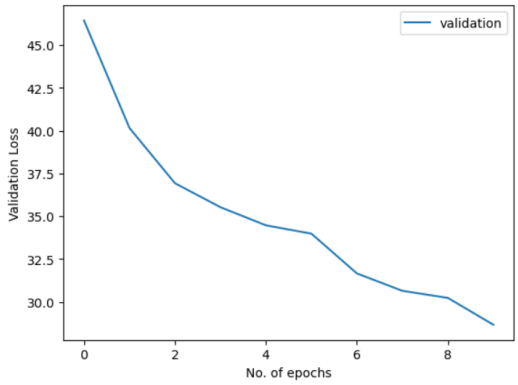


2. 2 Conv+ 2 FC layer structure

```python
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        self.fc1 = nn.Linear(64 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 10)

        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.bn1(self.pool(torch.relu(self.conv1(x))))
        x = self.bn2(self.pool(torch.relu(self.conv2(x))))

        x = x.view(-1, 64 * 8 * 8)

        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))

        return x
```

a. **ReLu** as an Activation function and **SGD** as optimizer





Momentum = 0.9 **Without Momentum**

Momentum	Learning rate	Time (sec)	Accuracy (%)
0.9	0.001	418.852	77.790000 %
Without Momentum	0.001	403.197	69.160000 %