The implementation offers a versatile and modular way to train various deep learning models. It is simple to manage and expand the code because it is divided into distinct files for every model and configuration. Scalability is improved by using a configuration file and a centralized training pipeline, which enables future additions like additional models or distinct datasets.

Because it makes it simple to switch between models and tracks training results quickly using saved weights and plots, this solution is perfect for researchers or practitioners who wish to train and compare multiple deep learning architectures. The method also exemplifies best practices for structuring deep learning projects in an orderly, manageable way.

**Structure**:

*Root Dir:*

*├── alexnet: │*

*├── alexnet\_with\_lrn.py -> Class `AlexNetLRN`*

*├── alexnet\_without\_lrn.py -> Class `AlexNetWithoutLRN`*

*├── googlenet: │*

*├── googlenet.py -> Class `GoogLeNet`*

*├── main.py*

*├── model\_config.py*

**Implementation of Alexnet with LRN**

The AlexNet model **had been modified** by incorporating Local Response Normalization (LRN) to improve its performance. Initially, the AlexNet architecture **had consisted** of several convolutional layers followed by fully connected layers without any normalization technique. After the changes, the first two convolutional layers **had been followed** by LRN layers, which **had applied** normalization across neighboring feature maps. The LRN layer **had been designed** using PyTorch's built-in F.local\_response\_norm() function, and it **had been parameterized** with values like alpha, beta, and k, based on the original AlexNet paper.

In the previous architecture without LRN, the network **had relied** solely on convolutional, activation, and pooling layers for feature extraction. In contrast, after adding LRN, the feature maps **had been normalized** within a local neighborhood, enhancing the network’s ability to generalize better during training. This modification **had been expected** to improve the model’s performance on the test set, as LRN **had been shown** to help reduce overfitting and improve generalization by normalizing activations.

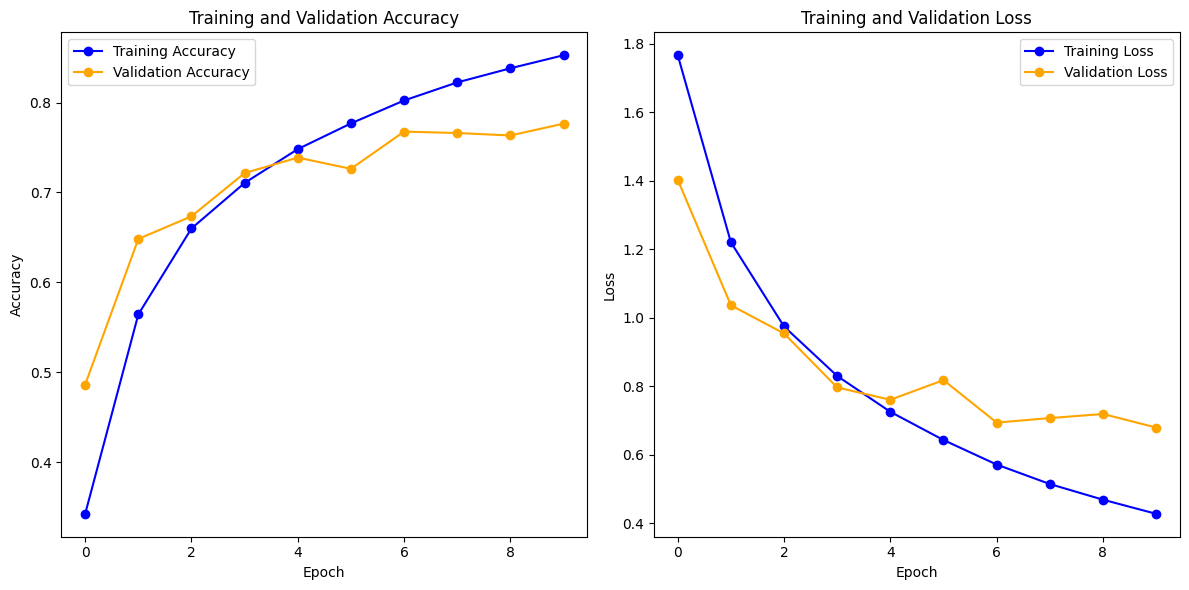
Both models—AlexNet without LRN and AlexNet with LRN—**had been evaluated** on the same dataset, and the test set results **had been compared** to determine whether LRN **had improved** the accuracy of the model.

**LRN Class Implementation**:

The LRN class is a simple function that wraps around PyTorch's built-in F.local\_response\_norm() to apply Local Response Normalization (LRN) to the feature maps after each convolutional layer. LRN works by normalizing the activations within a small region of the feature maps. The parameters for LRN, such as size, alpha, beta, and k, are set based on the original AlexNet design. This helps the network by making the activations more stable and improving its ability to generalize, which in turn reduces the risk of overfitting during training.

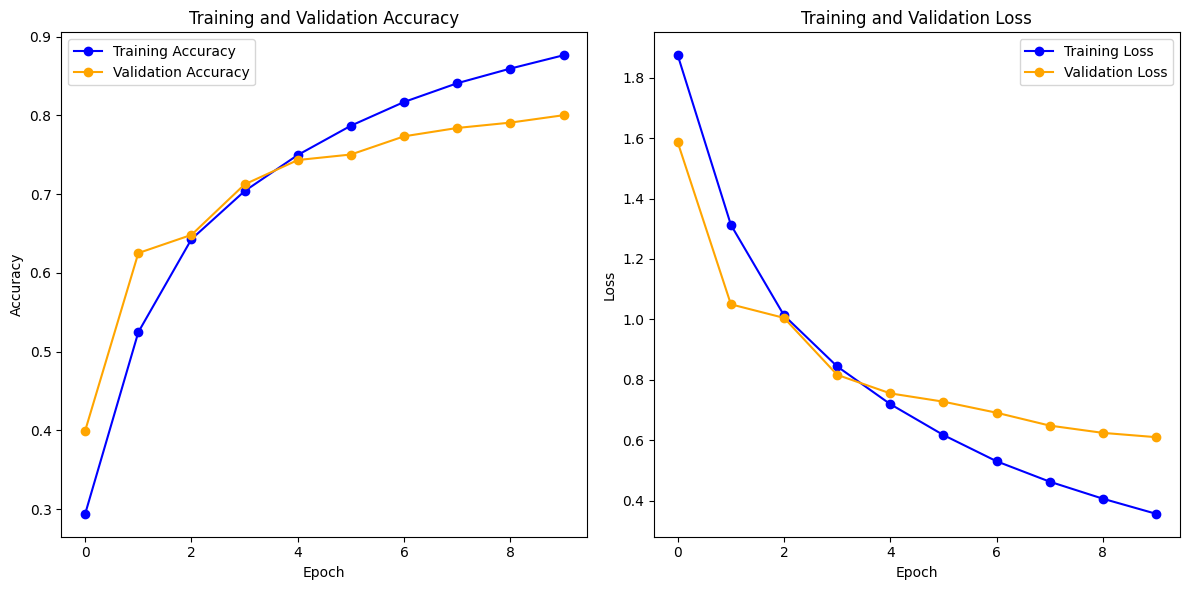
LRN is expected to help improve the performance of the network by reducing the internal covariate shift and enhancing generalization. The theory behind LRN **had been** that by normalizing the activations within each feature map, the network **could have become** more robust and less prone to overfitting.  
  
  
**Comparison (Alexnet with LRN and Alexnet without LRN)**

Training Result (Alexnet with without LRN):



Training Result (Alexnet with LRN):

The training results for AlexNet without Local Response Normalization (LRN) showed consistent progress across 10 epochs, with training accuracy increasing from 34.26% in the first epoch to 85.26% in the last, indicating effective learning from the training data. Validation accuracy began at 48.64% and reached a peak of 77.65% in the 9th epoch, but did not improve significantly thereafter, suggesting some overfitting as the model performed better on the training data compared to unseen validation data. Both training and validation loss decreased steadily, with training loss ending at 0.4270 and validation loss at 0.6791, but the validation loss remained higher than the training loss, reinforcing the overfitting observation. Although the model showed good learning, the gap between training and validation performance suggested that incorporating regularization techniques like LRN could potentially improve generalization and further boost validation accuracy.



The training results for the AlexNet model with Local Response Normalization (LRN) show significant improvement over 10 epochs. The training accuracy increased from 29.41% in the first epoch to 87.62% by the last epoch, indicating that the model effectively learned from the training data. Similarly, the validation accuracy improved from 39.97% to 80.01%, suggesting that the model generalized well to unseen data without overfitting. Both training and validation losses decreased steadily, further confirming the model's improved performance. The relatively small gap between training and validation accuracy demonstrates that the model not only learned well but also generalised effectively, which can be attributed to the stabilizing effect of LRN on activations, enhancing generalization and reducing overfitting.

When comparing AlexNet with and without Local Response Normalization (LRN), there are clear differences in training, validation, and test performance.

For the model with LRN, training accuracy started at 29.41% and increased to 87.62% by the end of the 9th epoch, with the training loss dropping from 1.8747 to 0.3560. Validation accuracy started at 39.97% and peaked at 80.01%, while validation loss decreased from 1.5861 to 0.6098. After training, the test accuracy was 79.51%.

In comparison, the model without LRN showed slower but consistent improvement. Training accuracy started at 34.26% and ended at 85.26%, with training loss going from 1.7666 to 0.4270. Validation accuracy increased from 48.64% to 77.65%, and validation loss decreased from 1.4025 to 0.6791. The test accuracy for this model was 77.41%.

AlexNet with LRN performed better than AlexNet without LRN in both validation and test accuracy. The model with LRN had about 2.36% higher validation accuracy and 2.10% higher test accuracy, suggesting that LRN improved the model’s ability to generalize and reduced overfitting compared to the model without LRN.

**GoogLeNet with two side classifiers**

The provided code implements a GoogLeNet-like CNN model with two auxiliary classifiers. It builds upon the concept of Inception blocks, which are a key feature of the GoogLeNet architecture. The network's core building block is the Inception block, which contains multiple paths (or branches) that perform different operations on the input. These branches include a 1x1 Convolutional Layer that reduces the number of channels (filters) and helps the model focus on the most important features. Another branch uses 1x1 Convolution followed by a 3x3 Convolution, which first reduces the number of channels and then applies a 3x3 convolution to capture more complex features. Additionally, there’s a 1x1 Convolution followed by a 5x5 Convolution to capture even larger features. Lastly, the MaxPooling followed by a 1x1 Convolution branch applies max-pooling (downsampling) and a 1x1 convolution to reduce the number of channels. The outputs from all these branches are concatenated to provide a richer set of features that are passed to the next layer, allowing the model to learn multi-scale features.

The GoogLeNet model with auxiliary classifiers incorporates a unique training approach. Auxiliary Classifiers are inserted at intermediate layers to provide extra supervision during training, helping the network not become too deep and difficult to train. These auxiliary classifiers output predictions at intermediate stages. The first auxiliary classifier is applied after the first Inception block (a3), and the second is applied after the second Inception block (b3). These classifiers are used only during training, and their outputs are combined with the final prediction in the last stage.

In terms of data flow, the input image (with 3 color channels) initially passes through a 3x3 Convolution Layer (pre\_layers) that extracts basic features, followed by batch normalization and ReLU activation. After this, the image goes through several Inception Blocks (a3, b3, a4, b4, etc.), which extract more complex features by applying convolutions at different scales. Following some of the Inception blocks, MaxPooling Layers are used to reduce the spatial dimensions of the feature maps, thus lowering computational cost and preventing overfitting. After passing through all the Inception blocks, a final Global Average Pooling layer is applied to further reduce the spatial dimensions, followed by a Fully Connected (Linear) layer that outputs the final classification prediction.

In the forward pass, the input image flows through the initial layers and Inception blocks. At intermediate stages (after the a3 and b3 blocks), the auxiliary classifiers are applied to guide the training process. Once all the Inception blocks have been passed, global average pooling is applied, and the output is passed through a linear layer for classification. The network produces three outputs: the final classification result from the linear layer, as well as the predictions from the two auxiliary classifiers.

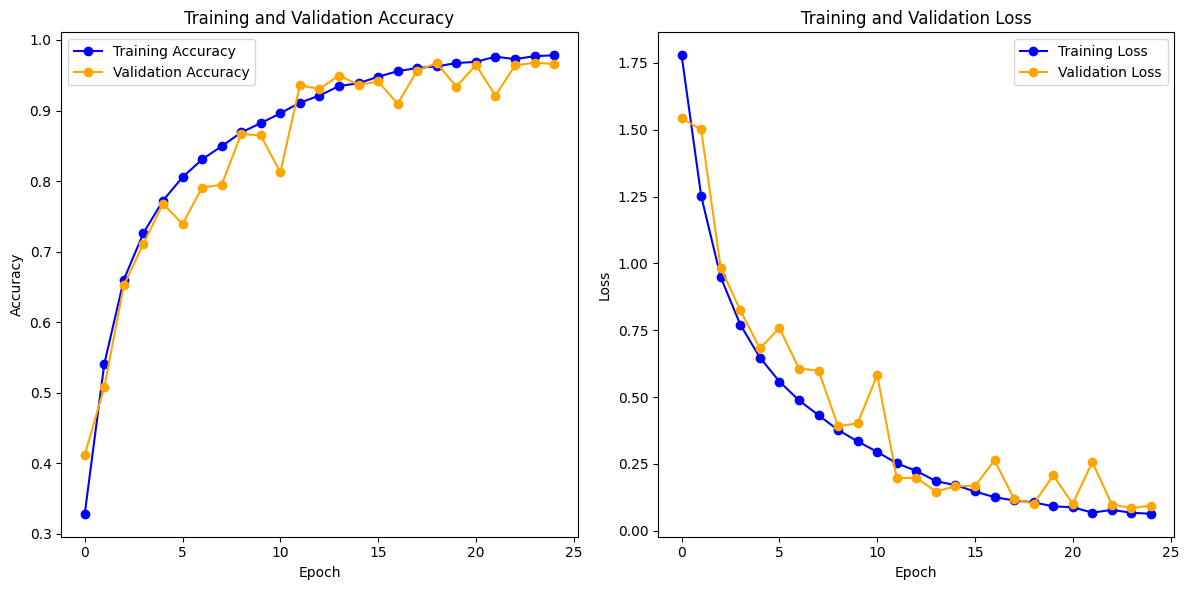
The architecture can be summarized as follows: it starts with a 3-channel image, followed by a 3x3 convolution to extract basic features. The image then goes through multiple Inception blocks, each containing different branches for capturing features at various scales. Auxiliary classifiers are introduced at intermediate stages to help train the network. Finally, global average pooling is applied, followed by a fully connected layer for classification. The model outputs three predictions: one from the final output and two from the auxiliary classifiers.

Key characteristics of the model include the use of Inception Blocks, which enable the network to learn multi-scale features, and Auxiliary Classifiers, which help with training and prevent vanishing gradients in deeper networks. The Global Average Pooling layer reduces the spatial dimensions before the final fully connected layer. Overall, this architecture provides a deep and efficient way of learning image representations, with the auxiliary classifiers contributing to effective training and overfitting prevention.

**Result:**

The model's performance during training shows steady improvement over the 24 epochs. Initially, the training accuracy was low (32.85% in epoch 0) and gradually increased, reaching 97.83% by the final epoch. Similarly, the validation accuracy showed an upward trend, starting at 41.20% and peaking at 96.78% in the final epoch. The training loss decreased consistently from 1.7785 in epoch 0 to 0.0633 in epoch 24, indicating that the model was learning to minimize errors over time.

However, the test accuracy, which measures how well the model generalizes to unseen data, was lower at 84.19%. This suggests that while the model performed well on the training and validation sets, it faced some challenges when evaluating on the test set. The test accuracy is still relatively high, but it reflects that the model may have slightly overfitted to the training data, as evidenced by the high training accuracy compared to the test result. This discrepancy is common in deep learning models and could be addressed by further techniques like regularization, data augmentation, or fine-tuning.

****

**Comparison (Alexnet with LRN and Googlenet with two side classifiers)**  
**Alexnet with LRN:**  
  
**Parameters**:

Conv1= 34,944

Conv2 = 614,656

Conv3 = 885,504

Conv4 = 1,327,488

Conv5 = 884,992

Fully connected layer 1 = 37,756,928

Fully connected layer 2 = 16,781,312

Fully connected layer 3 = 40,970

**Total Number of Parameters = 57,626,794**

**Googlenet with two side classifiers:**

**Parameters**:

The architecture is a deep network with multiple layers, including convolutional layers (Conv2d), batch normalization layers (BatchNorm2d), and Inception blocks. Each Inception block consists of multiple paths, including convolutions with different kernel sizes, max pooling, and other operations. Convolutional layers in the pre-layers, Inception modules, and auxiliary classifiers contribute a significant amount of parameters. BatchNorm layers: Each Inception block has several BatchNorm layers that contribute extra parameters. The final linear layer has 1024 input features, and it produces 10 output features, contributing to a significant portion of the parameters. Given the complexity and the numerous layers, the total number of parameters in this architecture is approximately 7 million.

Training complete in 75m 40s for 25 epoch on batch size 128