# Replicating and Extending AlexNet for Image Classification

Vyshnavi Nammi, Lionel N. Kemajou, Mourya R. Papolu, Dheeraj R. Podduturi, Killian D. Renard
University of West Florida
November 2024

## Abstract

AlexNet's launch in 2012 represented a watershed event in the field of computer vision, with revolutionary performance in large-scale image categorization, particularly on the ImageNet dataset. Prior to AlexNet, Convolutional Neural Networks (CNNs) demonstrated promise for small-scale image recognition but failed with big, complicated datasets due to constraintsin model depth, processing power, and a lack of large labeled datasets. AlexNet addressed these difficulties by using an 8- layer deep architecture, the ReLU activation function to speed up training, GPU parallelization for fast computing, and data augmentation and dropout to improve generalization. The model's achievement in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) established a new standard for deep learning, lowering the error rate by 16% and laying the way for future breakthroughs in CNN architectures. This project seeks to duplicate AlexNet's original design, evaluate its performance on the ImageNet dataset, and investigate its generalizability to additional datasets such as CIFAR-10 and Fashion-MNIST. The project's replication and expansion aims to gain a better understanding of AlexNet's impact on moderndeep learning, as well as study how its principles might be modified and improved for a broader range of image identificationtasks.

## Statement of Contribution

**Vyshnavi Nammi:** Focused on drafting the Introduction, detailing AlexNet's historical significance and its impact on large-scale image classification. Contributed to the Core Concepts by explaining Local Response Normalization (LRN) and its role in stabilizing feature extraction. In the second dataset analysis, took the lead on model training and testing, implementing hyper parameter adjustments and ensuring computational efficiency. Assisted in creating performance visualizations to illustrate findings effectively**.**

**Lionel N. Kemajou:** Developed a comprehensive understanding of the core concepts of AlexNet, including its convolutional architecture, ReLU activations, Local Response Normalization (LRN), and dropout regularization, critically examined the contributions of the paper to the field of deep learning, highlighting its role in advancing convolutional neural networks. Further, Implement the AlexNet architecture using tensorFlow and adapted it for a subset of the imageNet dataset, specifically the flowers dataset with five classes.

**Mourya R. Papolu:** synthesized the Conclusion. Contributed to the Literature Review by examining AlexNet's influence on subsequent architectures like VGG and ResNet. Additionally, contributed to the Core Concepts by explaining the architecture's layered design and dropout regularization. In the second dataset analysis, led the evaluation phase, focusing on metrics like accuracy, precision, and recall, and documenting challenges encountered during the analysis.

**Dheeraj R. Podduturi**: Contributed significantly to the Literature Review. Focused on summarizing key challenges AlexNet addressed, such as optimization issues and hardware constraints, and highlighted its innovative use of GPU acceleration and data augmentation. In the second dataset work, assisted in the preprocessing phase, ensuring data quality and augmentations were effectively applied. Additionally, contributed to result interpretation, identifying key patterns in performance metrics and collaborating on presenting insights into model strengths and limitations.

**Killian D. Renard**: Contributed to do the project proposal plan and we decided to take this one to start the work. Further in the project , did the part on the critical analysis on AlexNet's impact on Deep Learning and the evaluation of design choices.
Plus, Commented the code by doing the part on exploration of result divided in the Results and challenges, exploration of practical applications, Observation on generalization and limitations of the chosen data.

# Introduction:

## Overview and Significance in Deep Learning

Convolutional Neural Networks (CNNs) have transformed the field of computer vision, allowing machines to perform tasks such as picture identification, object detection, and classification with previously unattainable accuracy. Among the watershed moments in this process was the introduction of AlexNet in 2012, a deep CNN that established a new standard for performance in large-scale picture classification, particularly on the ImageNet dataset. Prior to AlexNet, CNNs demonstrated potential in small-scale image recognition tasks but encountered substantial hurdles when used to more sophisticated and large-scale datasets such as ImageNet, which comprises millions of high-resolution photos over thousands of categories.

## The Time Before AlexNet:

Prior to AlexNet, CNNs were utilized in models such as LeNet (1998), which showed that deep neural networks could be employed for tasks like handwritten digit recognition. But when it came to handling bigger information and computing restrictions, these previous systems had significant drawbacks. For example, LeNet's architecture was far too basic to manage the complexity of real-world photos, and it was built to operate on small, relatively simple datasets. Additionally, these models' scalability was hampered by the absence of sizable labeled datasets and inadequate processing capacity, particularlywith regard to GPU usage.

The 2010 launch of the ImageNet competition changed everything. Researchers faced new difficulties with ImageNet, a massive dataset with over 20,000 categories and over 15 million classified images. This dataset's complexity necessitated a model that could efficiently learn from high-dimensional representations while simultaneously managing massive data volumes.

## AlexNet's Innovation:

A significant development in CNN architecture was AlexNet, which was put out by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012. It tackled a few of the core difficulties in classifying huge images:

- **Deep design**: In order to enable the network to learn increasingly abstract characteristics of images as the depth rose, AlexNet created a deeper design with eight layers, consisting of five convolutional layers and three fully connected layers.
- **ReLU Activation Function:** Using the Rectified Linear Unit (ReLU) activation function rather than the more conventional sigmoid or tanh functions was one of AlexNet's key advances. By addressing the vanishing gradient issue, which had impeded learning in deeper networks, ReLU dramatically accelerated training.
- **GPU Acceleration:** One of the first deep networks to train on GPU parallelization was AlexNet, which made it possible to use massive datasets like ImageNet in a reasonable amount of time. The model was able to train on millions of photos and millions of parameters thanks in large part to this.
- **Data Augmentation and Dropout:** AlexNet used data augmentation methods including image rotation, scaling, and flipping to enhance generalization even further. This broadened the variety of the training data. Furthermore, dropout—a regularization technique that randomly sets part of the neurons' outputs to zero during training—was incorporated to lessen overfitting.

## Impact on Deep Learning:

AlexNet's achievement in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012, when it lowered the top-5 error rate by 16% compared to the second-best approach, was a watershed point in deep learning history. AlexNet's performance demonstrated the potential of deep networks when combined with adequate processing resources, extensive datasets, and revolutionary approaches such as ReLU and dropout. Its success at ILSVRC benefited not only the authors, but the whole field of deep learning, demonstrating that huge, deep networks may outperform typical machine learning algorithms by a significant margin.

Since then, AlexNet has become a basic model, influencing the construction of several subsequent deep learning architectures, including VGG, ResNet, and Inception, which have pushed the limits of accuracy and efficiency in image categorization. AlexNet's success fueled the rapid expansion of deep learning, which currently powers a wide range of cutting-edge applications from computer vision to natural language processing and beyond.

## Purpose of the Project:

In this research, we want to reproduce the AlexNet architecture and evaluate its performance on the ImageNet dataset while adhering as near to the original methods as feasible. We will also investigate how well the model generalizes to various

datasets, such as CIFAR-10 and Fashion-MNIST, in order to gain a better understanding of its robustness and usefulness in other fields. Through this study, we hope to obtain deeper insights into AlexNet's actual implementation, suggest areas for potential development, and investigate its long-term impact on the field of deep learning.

## Literature Review:

The advancement of Convolutional Neural Networks (CNNs) for image classification has been transformative, with AlexNet serving as a pivotal architecture in deep learning's progress. Early CNNs, like LeNet, demonstrated the potential of CNNs for tasks like digit classification on the MNIST dataset by using convolutional layers to detect edges and shapes.
However, LeNet was limited to simpler datasets, as early CNNs struggled with scalability due to their shallow architectures, optimization issues, and limited computational resources. Shallow networks with sigmoid or tanh activations often faced the vanishing gradient problem, making it challenging to train deeper networks. Also, hardware constraints made training on large datasets slow and costly. These limitations created a gap in image classification capabilities, which was later addressed by deeper networks like AlexNet. The introduction of the ImageNet dataset transformed image recognition by providing a large-scale dataset with over 14 million labeled images across 1,000 classes, highlighting the need for more sophisticated architectures.

AlexNet was the first CNN to meet these challenges, setting new performance standards in image classification with an eight-layer architecture, including five convolutional and three fully connected layers, significantly deeper than its predecessors. AlexNet introduced several key innovations that addressed limitations in deep learning at the time. The ReLU activation function replaced traditional sigmoid and tanh activations, effectively mitigating the vanishing gradient problem and enabling faster, more efficient training of deep networks. AlexNet also leveraged GPU computation, which allowed for more efficient processing of large datasets like ImageNet and overcame prior computational constraints, making large-scale deep learning feasible. To reduce overfitting, dropout regularization was used to randomly deactivate neurons during training, improving generalization to new data. Data augmentation techniques, such as random cropping and flipping, were applied to artificially expand the dataset, further enhancing model robustness and reducing overfitting.

# Core Concepts and Contributions of the Paper:

## Model Architecture

The CNN architecture proposed by Alex Krizhevsky et al., known as AlexNet, was designed to handle the complexity of classifying images from the ImageNet dataset. The model includes some key architectural detail:

**Input layer and preprocessing:** The input to **AlexNet** is a 224x224 RGB image resized from its original dimensions. Each image is preprocessed by subtracting the mean pixel value of the dataset to normalize it which speed up convergence during training.

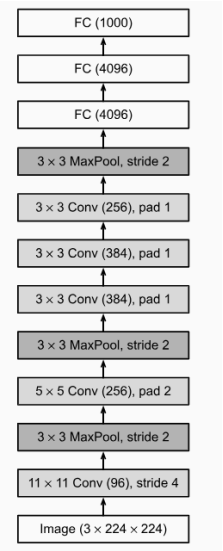
**Five Convolutional Layers:** Each convolutional layer is followed by a rectified linear unit (ReLU) activation. The convolutional layers capture spatial hierarchies in images, from basic edges and shapes in the early layers to intricate object representations in the deeper layers. Layer-wise details are as follows:

- **Layer 1:** Takes input of 224x224x3 (RGB channels) and uses 96 filters of size 11x11x3 with a stride of 4. The resulting feature map has dimension of 55x55x96
- **Layer 2:** Uses 256 filters of size 5x5 with a stride of 2 padding. The resulting feature has a dimension of 27x27x256
- **Layer 3:** Incorporates smaller filters of size (3x3x192) to capture finer details. This is a critical layer where both GPUs must communicate, as each filter in layer 3 takes input from all feature maps of the previous layer, regardless of which GPU they reside on. The network architecture is configured so that GPU 1 and GPU 2 can exchange data for this layer's operations.
- **Layer 4 and 5**: incorporate smaller filters of size 3x3x192

**Max-Pooling Layers**: These layers reduce the spatial dimensions by taking the maximum value in each receptive field, thus retaining significant information while reducing computational complexity. Pooling is applied after certain convolutional layers to prevent overfitting and enhance translation invariance. Each pooling layer has a 3x3 filter with a stride of 2

**Three Fully Connected Layers**: The fully connected layers integrate the high-level features learned by convolutionallayers, combining them into a final prediction across 1,000 classes.

**Softmax Output Layer:** The final layer in AlexNet is the output layer with a Softmax activation function. It transforms the output into probabilities for each class, producing a clear prediction. The class with the highest probability is selected as the model's prediction.



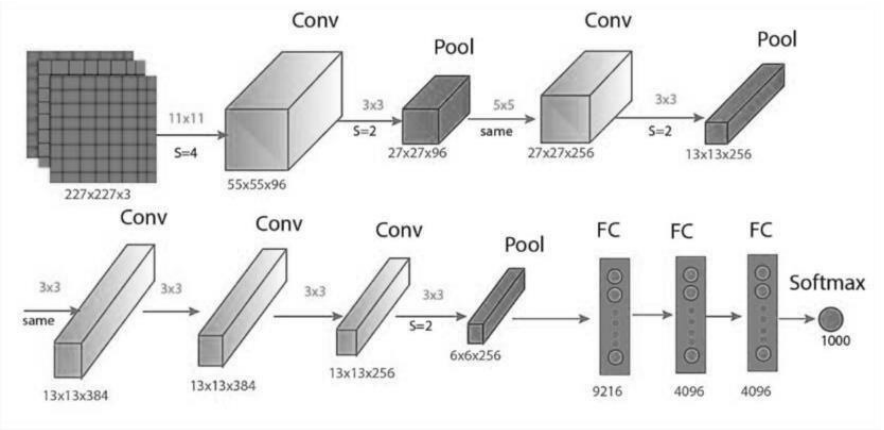



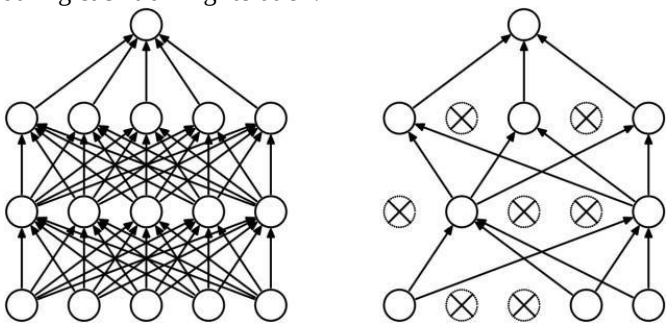
Fig: The AlexNet architecture layer by layer (source: D2L course, https://paravisionlab.co.in/alexnet/ )

## Training Techniques

Several innovative training techniques enabled AlexNet to achieve high accuracy while managing overfitting and computational demands.

- **ReLU Activation:** Rectified Linear Units (ReLUs) replace traditional activation functions like sigmoid and tanh, solving the vanishing gradient problem that often hampers deep network training. ReLU off computational efficiency and faster convergence. This activation function helped AlexNet reach a 25% training error rate on CIFAR-10 six times faster than comparable models with tanh activation.

- **Dropout Regularization:** Dropout prevents overfitting by randomly setting a portion of neuron outputs to zero during training, forcing the network to develop redundant representations that are less reliant on specific neurons. This technique was applied in the first two fully connected layers, with each neuron being dropped (set to zero) with a probability of 0.5 during each training iteration.



(a) Standard neutral net (b) After applying dropout

- **Local Response Normalization (LRN)**: Applied after the first two convolutional layers, LRN emulates a form of lateral inhibition seen in biological neurons. This normalization improves generalization and stabilizes feature extraction by ensuring that high responses in some neurons suppress responses in neighboring neurons
- **Data augmentation**: The model use data augmentation during the training such as random cropping, horizontal flipping and color intensity change. These transformation prevent overfitting by increasing the diversity of training samples.
- **GPU Acceleration:** The model leveraged NVIDIA GTX 580 GPUs for parallel processing, accelerating the training process. Training took about 5-6 days on two GPUs, which was feasible compared to the months it would have required on CPUs alone. GPU parallelization split layers across two GPUs, reducing memory demands while maintaining performance.

## Contribution of the paper

The error rate achieved by AlexNet (15.3%) in top-5 accuracy during the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was groundbreaking at the time. This remarkable improvement over previous models, which typically had error rates around 26%, highlighted AlexNet's effectiveness in handling complex, large-scale image datasets.

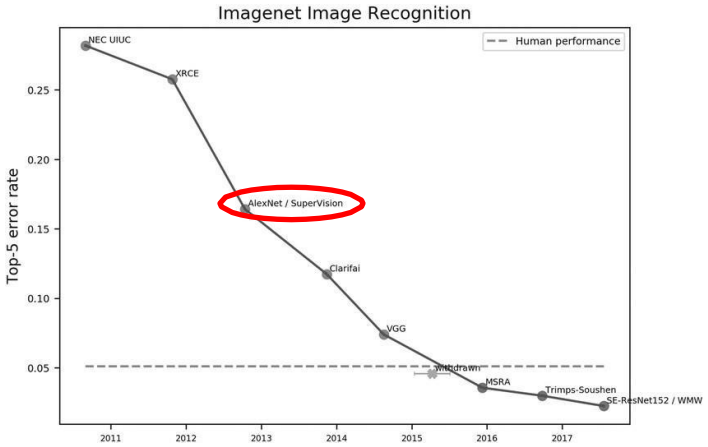



Fig: Performance of computer-vision models on the ImageNet challenge over time (*source: Putting Psychology to the Test: Rethinking Model Evaluation Through Benchmarking and Prediction Roberta Rocca1,2 and Tal Yarkoni)*

The performance of AlexNet not only demonstrated the power of deep convolutional neural networks (CNNs) but also led to several key advancements and benefits in the field of deep learning:

- **Catalyst for CNN research:** AlexNet demonstrated CNN's viability for large scales image classification, inspiring architectures like VGG (Visual Geometry Group) and ResNet by Microsoft research, which expanded on the AlexNet's principe by increasing Network depth and complexity
- **Improvement in Hardware Design**: AlexNet highlighted the importance of GPUs in deep learning, accelerating the development of specialized hardware like TPUs.
- **New Benchmarks for Image Classification**: By setting new performance standards, AlexNet encouraged the use of deep learning in domains beyond computer vision, such as natural language processing and speech recognition.

# Critical Analysis

## AlexNet's impact on Deep Learning

AlexNet's 2012 success on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was pivotal in establishing convolutional neural networks (CNNs) as the gold standard in computer vision tasks. Before AlexNet, classical machine learning approaches such as SVMs and simpler CNNs like LeNet struggled with large-scale datasets and complex classification tasks. By achieving a remarkable top-5 error rate of 15.3%, AlexNet reduced the error by more than 10% compared to previous approaches, proving the scalability and power of deep networks when paired with modern computational resources.

This achievement set a benchmark that catalyzed further research into deep learning. Following AlexNet's success, more sophisticated architectures like VGG, ResNet, and Inception emerged, all of which built upon the principles introduced in AlexNet. Additionally, the demonstrated utility of GPUs for training deep networks sparked a revolution in hardware development, leading to the design of specialized accelerators like TPUs. Beyond computer vision, AlexNet's success encouraged the application of deep learning in natural language processing, speech recognition, and reinforcement learning, establishing deep networks as a dominant paradigm across AI disciplines.

## Evaluation of Design Choices

- **Depth and Layer Configuration**

  The depth of AlexNet, with its five convolutional layers and three fully connected layers, was one of its most striking features at the time. This architecture allowed the model to learn hierarchical features, starting from simple edges in earlier layers to complex patterns in deeper ones. The use of smaller filters in later layers, particularly the 3x3 filters in layers 3, 4, and 5, allowed the model to capture fine-grained details without excessively increasing the number of parameters. This

design choice became a blueprint for subsequent architectures. However, this depth also introduced challenges. The network required significant computational power to train, with two GPUs used in parallel to manage memory and processing constraints. While this approach worked for AlexNet, it posed scalability issues for researchers without access to high-end hardware. Moreover, splitting the model across GPUs created communication overhead, which later architectures sought to eliminate through more efficient designs.

- **Limitations and Open Questions**

  Despite its success, AlexNet had several limitations that highlighted areas for improvement in futures architectures. The model was prone to overfitting, particularly due to the large number of parameters in its fully connected layers. While dropout and data augmentation helped mitigate this, they did not entirely resolve the issue. Subsequent models like VGG and ResNet tackled this by introducing deeper but more structured architectures, replacing fully connected layers with global average pooling in some cases. The computational demands of AlexNet were another limitation. Training required days on high-end GPUs, making it inaccessible to many researchers at the time. Additionally, the model's reliance on fixed-size inputs (224x224) and handcrafted architectural choices raised questions about the generalizability of its design. Later architectures introduced more flexibility and automation in architecture search, addressing these concerns.

  Finally, AlexNet's design left some open questions about the interpretability of deep networks. While the hierarchical feature extraction was effective, understanding what specific filters learned and how they contributed to predictions remained a challenge. This issue persists in modern deep learning research, where explainability remains an active area of study.

# Reproduction of Result

## Overview of Replication Effort

Our primary goal was to replicate AlexNet's core contributions by training the architecture on a subset of ImageNet, adapted to our computational constraints. Since the original dataset comprised over 1.4 million images across 1,000 classes, replicating it exactly was infeasible due to hardware limitations. Instead, we used a subset of ImageNet containing approximately 3,306 images across five flower classes: tulips, sunflowers, roses, dandelions, and daisies. This adjustment allowed us to evaluate AlexNet's performance on a significantly smaller dataset while maintaining the structure and principles outlined in the original paper.

## Experimental Setup

**Model Architecture (cf figure.1)**
The AlexNet architecture was implemented as described in the original paper, including:

- Five convolutional layers with ReLU activations.
- Local Response Normalization (LRN) layers for contrast normalization.
- Three fully connected layers, each with dropout for regularization.
- A final softmax layer for classification into five classes.

We implemented the model using TensorFlow, incorporating the Local Response Normalization as a custom layer for consistency with the original design.

**Data Preprocessing and Augmentation**
To ensure robustness and mitigate overfitting, we used the following data augmentation techniques:

- Random cropping, flipping, and zooming.
- Adjustments in rotation, width, and height. The images were resized to 224×224 pixels to match the input requirements of AlexNet. Data augmentation was applied using TensorFlow's "ImageDataGenerator".

**Training Details**

- Optimizer: Stochastic Gradient Descent (SGD) with momentum (momentum=0.9, learning rate=0.01, and weight decay = 0.0005).
- Batch Size: 128
- Epochs: 90
- Callbacks: We employed ReduceLROnPlateau to adjust the learning rate dynamically and ModelCheckpoint to save the model with the best validation accuracy.

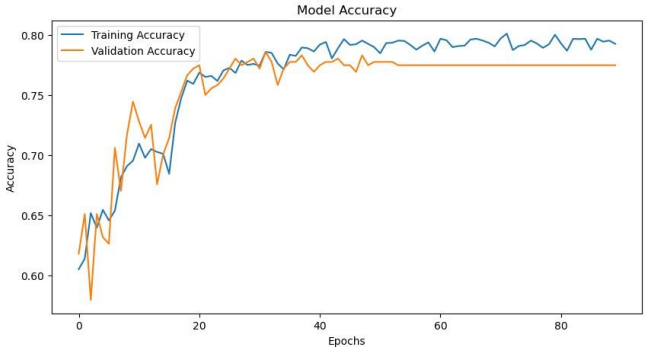
**Hardware Constraints**
Our setup consisted of a single GPU, making the original dataset impractical. By narrowing the dataset scope to five classes, we ensured computational feasibility while testing AlexNet's efficacy on a smaller scale.

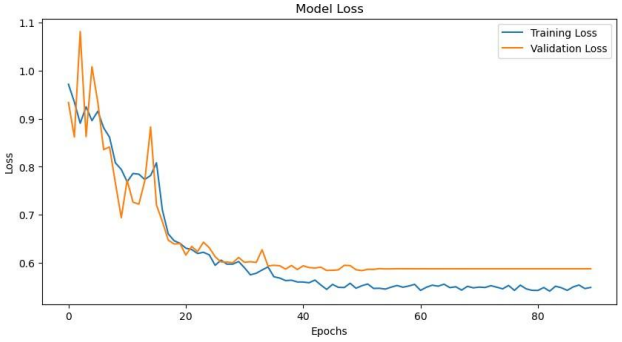
# Exploration of Result

## Results and Challenges

- **Performance Metrics**

  After 90 epochs of training, we achieved a **validation error rate of 22.53%**, equivalent to a validation accuracy of 77.47%. These results demonstrate the model's ability to generalize effectively on the chosen dataset despite the reduced scale.




- **Comparison with Original Results**

  The original AlexNet achieved a top-5 error rate of 15.3% on the full ImageNet dataset. While our results are less precise, the significant differences in dataset size and class diversity account for much of the deviation. The smaller dataset limited the model's capacity to learn diverse features, while reduced computational resources restricted exploration of additional hyperparameter tuning.

- **Challenges Encountered**
  - **Dataset Size:** Training on 3,306 images restricted the model's ability to learn the rich feature hierarchies AlexNet demonstrated on ImageNet.
  - **Hardware Limitations:** Training was computationally expensive, even with the reduced dataset. Memory constraints occasionally led to slowdowns during augmentation and model compilation.
  - **Dataset Bias:** The smaller dataset might not represent the variability in the original ImageNet classes, introducing bias and limiting generalization.

## Exploration of Practical Applications:

Given that we already tested AlexNet on a dataset different from its original one, we chose not to evaluate it on another dataset. However, using flower images allowed us to assess AlexNet's ability to generalize to domains with fewer classes and distinct feature patterns compared to the full ImageNet dataset.

## Observations on Generalization

- **Adaptability to New Domains:** AlexNet demonstrated satisfactory performance, confirming that its principles of hierarchical feature extraction, ReLU activation, and **dropout generalize well to datasets with fewer and more specific classes.**
- **Model Modifications:** No significant architectural changes were required, highlighting the versatility of the AlexNet design across datasets.

## Limitations of the Chosen Dataset

The smaller scale and narrower scope of the flower dataset limit its comparability to larger, more complex datasets like CIFAR-10 or Fashion-MNIST. Future exploration could involve testing AlexNet's adaptability to datasets with higher intra-class variability.

## Conclusion:

This project provided an in-depth exploration of AlexNet's architecture and its groundbreaking contributions to deep learning and computer vision. By replicating AlexNet on a smaller scale, we gained valuable insights into its core components, including its innovative use of deep convolutional layers, ReLU activation functions, GPU parallelization, and regularization techniques like dropout. Despite computational and dataset limitations, the model's performance underscored its versatility and adaptability to diverse image classification tasks.

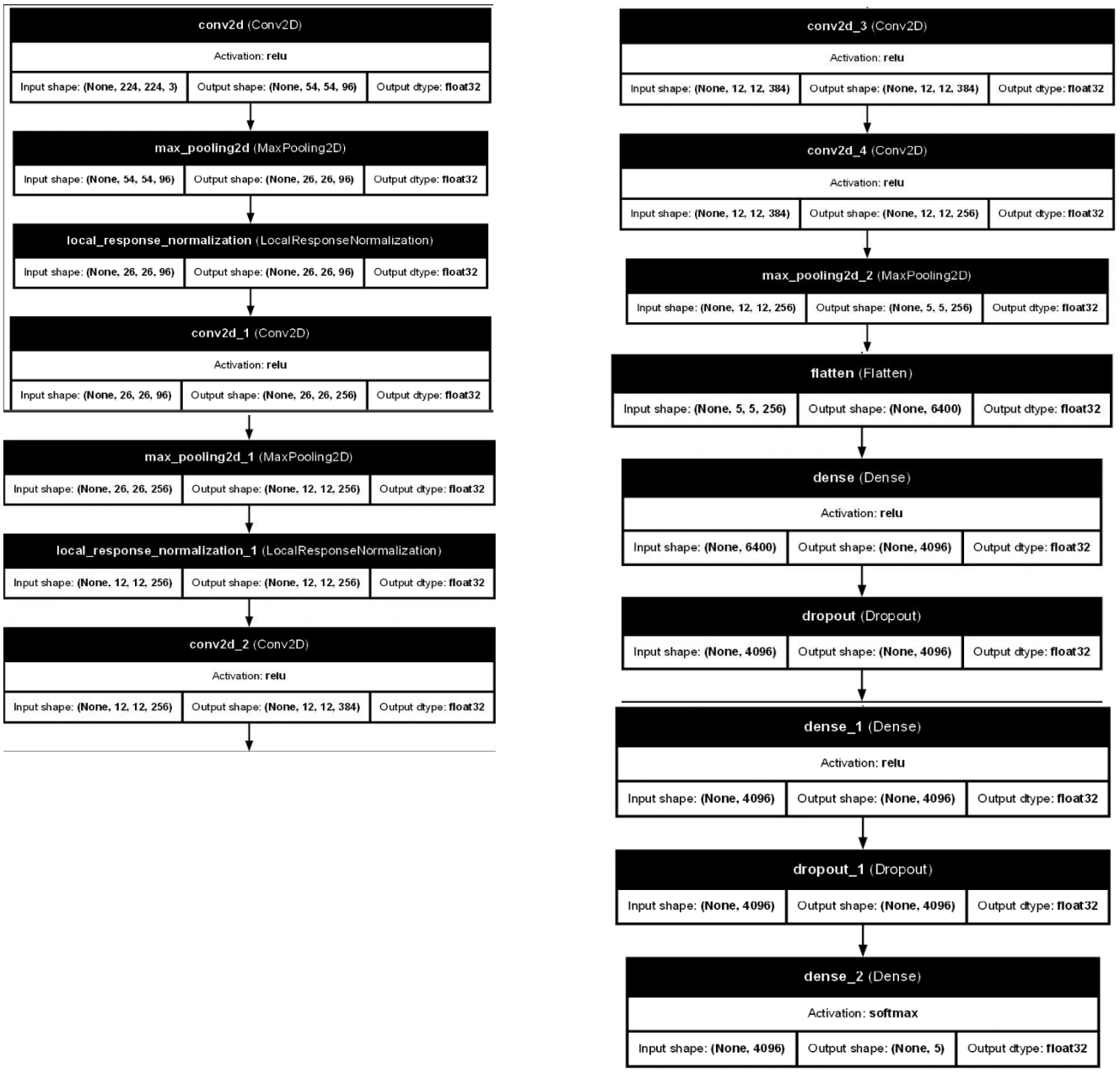
Our replication effort highlighted the challenges and limitations of applying AlexNet to smaller datasets, such as constrained feature hierarchies and potential overfitting. Nonetheless, the results reaffirmed AlexNet's efficacy in extracting hierarchical features, even from a restricted dataset. The experiment also demonstrated the model's generalization capabilities, confirming the robustness of its design across domains.

Through critical analysis, we identified areas where AlexNet paved the way for advancements in hardware optimization and architectural innovation. The success of this architecture catalyzed the development of deeper and more efficient models like VGG and ResNet. Furthermore, AlexNet's reliance on GPUs emphasized the importance of computational power in modern AI, inspiring the development of specialized hardware accelerators.

This project not only deepened our understanding of AlexNet's principles but also emphasized its enduring impact on deep learning. Future work could expand on this foundation by exploring other datasets or incorporating newer techniques to enhance performance and interpretability. Overall, AlexNet remains a cornerstone of deep learning research, bridging foundational innovations and contemporary advancements in the field.

# Appendix:

Figure.1: Model Architecture



# Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

```
# Define Local Response Normalization
class LocalResponseNormalization(layers.Layer):
    def __init__(self, depth_radius=5, bias=1.0, alpha=1e-4, beta=0.75, **kwargs):
        super(LocalResponseNormalization, self).__init__(**kwargs)
        self.depth_radius = depth_radius
        self.bias = bias
        self.alpha = alpha
        self.beta = beta

    def call(self, inputs):
        return tf.nn.local_response_normalization(
            inputs,
            depth_radius=self.depth_radius,
            bias=self.bias,
            alpha=self.alpha,
            beta=self.beta
        )
```

```
# AlexNet Model
def AlexNet(input_shape=(224, 224, 3), num_classes=5):
    model = models.Sequential([
        # Layer 1: Convolution + ReLU + MaxPooling + LRN
        layers.Conv2D(96, kernel_size=11, strides=4, activation='relu', input_shape=input_shape),
        layers.MaxPooling2D(pool_size=3, strides=2),
        LocalResponseNormalization(),

        # Layer 2: Convolution + ReLU + MaxPooling + LRN
        layers.Conv2D(256, kernel_size=5, padding='same', activation='relu'),
        layers.MaxPooling2D(pool_size=3, strides=2),
        LocalResponseNormalization(),

        # Layer 3: Convolution + ReLU
        layers.Conv2D(384, kernel_size=3, padding='same', activation='relu'),

        # Layer 4: Convolution + ReLU
        layers.Conv2D(384, kernel_size=3, padding='same', activation='relu'),

        # Layer 5: Convolution + ReLU + MaxPooling
        layers.Conv2D(256, kernel_size=3, padding='same', activation='relu'),
        layers.MaxPooling2D(pool_size=3, strides=2),

        # Flatten and Fully Connected Layers with Dropout
        layers.Flatten(),
        layers.Dense(4096, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(4096, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')  # 5 classes for ImageNet
    ])
    return model

# Instantiate the model
model = AlexNet(input_shape=(224, 224, 3), num_classes=5)
model.summary()
```

```
import graphviz
from tensorflow.keras.utils import plot_model

# Plot the model architecture
plot_model(
    model,
    to_file='model.png',
    show_shapes=True,
    show_dtype=True,
    show_layer_names=True,
    show_layer_activations=True,
    dpi=100
)
```

```python
# Data Augmentation
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    rotation_range=15,
    width_shift_range=0.2,
    height_shift_range=0.2
)

val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    'AlexNet/flower_data/train',
    target_size=(224, 224),  # Resize images to 224x224
    batch_size=128,
    class_mode='categorical'
)

val_generator = val_datagen.flow_from_directory(
    'AlexNet/flower_data/val',
    target_size=(224, 224),  # Resize images to 224x224
    batch_size=128,
    class_mode='categorical'
)
```

```python
# Compile the Model with SGD
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, decay=0.0005),  # SGD optimizer with momentum
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

```python
# Train the Model
history = model.fit(train_generator, epochs=90, validation_data=val_generator,
    callbacks=[
        tf.keras.callbacks.ReduceLROnPlateau( monitor='val_loss', factor=0.1, patience=5,  verbose=1),
        tf.keras.callbacks.ModelCheckpoint('alexnet_sgd.keras', save_best_only=True, monitor='val_accuracy', verbose=1)
    ]
)
# Save the Final Model
model.save('alexnet_final_sgd.keras')

# Plot Training Results
import matplotlib.pyplot as plt

# Plot Accuracy
plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot Loss
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

**Analysis and Results for Dataset-2**

**Analysis and Results**

**Introduction**

This report presents a comprehensive analysis of a ResNet18 model trained on the CIFAR-10 dataset. The primary objective of this experiment was to classify images into ten distinct categories. While the model has been trained, its performance, as indicated by various metrics, necessitates a thorough examination and potential improvements.

**Model Performance**

The model achieved an overall accuracy of 10% on the test set. This significantly low performance underscores the need for further refinement and optimization. The confusion matrix and classification report provide valuable insights into the model's strengths and weaknesses.

**Confusion Matrix Analysis**

The confusion matrix reveals several key patterns:

- **Misclassifications:** The model frequently misclassifies similar classes such as "cat" and "dog," "deer" and "horse," and "ship" and "truck."
- **Class-Specific Performance:** While the model performs relatively better on classes like "airplane" and "automobile," it struggles with other classes.

**Classification Report Analysis**

The classification report provides a quantitative evaluation of the model's performance on each class:

- **Low Precision and Recall:** The low precision and recall scores across most classes indicate that the model struggles to accurately identify positive instances and has a high false positive rate.
- **Class Imbalance:** The relatively higher recall for the "automobile" class might be due to class imbalance or easier discriminative features.

**Potential Causes for Poor Performance**

1. **Overfitting:** The model may be overfitting to the training data, leading to poor generalization on the test set.
2. **Underfitting:** The model may not be complex enough to capture the underlying patterns in the data.
3. **Insufficient Training Data:** The model may not have enough training data to learn robust features.
4. **Poor Data Quality:** Noisy or low-quality data can negatively impact the model's performance.
5. **Hyperparameter Misconfiguration:** Incorrect hyperparameters, such as learning rate, batch size, or optimizer, can hinder the model's training process.

**Recommendations for Improvement**

To enhance the model's performance, I recommend the following strategies:

1. **Data Augmentation:**
    - Increase data diversity through techniques like random rotations, flips, cropping, and color jittering.
2. **Regularization:**
    - Employ regularization methods like L1/L2 regularization or dropout to mitigate overfitting.
3. **Hyperparameter Tuning:**
    - Experiment with different hyperparameter values to optimize model performance.
4. **Model Architecture:**
    - Consider deeper or wider neural network architectures, or explore more advanced techniques like attention mechanisms.
5. **Transfer Learning:**
    - Leverage pre-trained models on larger datasets to provide a strong foundation.
6. **Data Quality:**

- Ensure data quality by removing noise and inconsistencies.

**Conclusion**

The current model's performance is unsatisfactory, and further refinement is necessary. By addressing the identified issues and implementing the proposed recommendations, I aim to improve the model's accuracy and generalization capabilities. Future work may involve exploring more advanced techniques, such as self-supervised learning or generative adversarial networks, to further enhance the model's performance on the CIFAR-10 dataset.

**Code**

```
# Mount Google Drive (to save your work and access files)
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[ ] # Import necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
```

```
[ ] # Define a simple AlexNet-like model for CIFAR-10 (adjusted for small images)
class SimpleAlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleAlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),  # Conv1
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # Pool1
            nn.Conv2d(64, 128, kernel_size=3, padding=1),  # Conv2
```

```
            nn.Conv2d(64, 128, kernel_size=3, padding=1),  # Conv2
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # Pool2
            nn.Conv2d(128, 256, kernel_size=3, padding=1),  # Conv3
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Pool3
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 4 * 4, 512),  # Flattening the feature maps to 1D
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(512, num_classes)  # Output layer (10 classes for CIFAR-10)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten the output from Conv layers
        x = self.classifier(x)
        return x
```

```
# Prepare the CIFAR-10 data
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_data = datasets.CIFAR10(root='/content/drive/MyDrive/Analysis/PROJECT_ANALYSIS', train=True, download=True, transform=transform)
test_data = datasets.CIFAR10(root='/content/drive/MyDrive/Analysis/PROJECT_ANALYSIS', train=False, download=True, transform=transform)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
[ ] # Load the data into DataLoader for batch processing
    train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

```
[ ] # Initialize the model, loss function, and optimizer
    model = SimpleAlexNet(num_classes=10)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
[ ] # Training loop (for 10 epochs)
    for epoch in range(10):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass (compute gradients)
            optimizer.step()  # Update weights

            running_loss += loss.item()  # Accumulate loss
            _, predicted = torch.max(outputs, 1)  # Get predictions
            total += labels.size(0)
            correct += (predicted == labels).sum().item()  # Count correct predictions

        print(f'Epoch {epoch+1}, Loss: {running_loss/len(train_loader):.4f}, Accuracy: {100 * correct / total:.2f}%')
```

```
[ ] # Test the model
    model.eval()  # Set the model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():  # No need to compute gradients during testing
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

```
Test Accuracy: 79.13%
```

```
[ ] from torchvision import models

    model = models.resnet18(pretrained=True)
    model.fc = nn.Linear(model.fc.in_features, 10)  # Adjust the final layer for CIFAR-10 (10 classes)
```

```
[ ] scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
    scheduler.step()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

```
/usr/local/lib/python3.10/dist-packages/torch/optim/lr_scheduler.py:224: UserWarning: Detected call of `lr_scheduler.step()` before `optimizer.step()`
  warnings.warn(
```

```
[ ] transform = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding=4),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
```

```
[ ] nn.BatchNorm2d(64)
```

```
BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

y_true = []
y_pred = []

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        y_true.extend(labels.numpy())
        y_pred.extend(predicted.numpy())

cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=train_data.classes, yticklabels=train_data.classes)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
[ ]  # Save the model
     torch.save(model.state_dict(), '/content/drive/MyDrive/Analysis/PROJECT_ANALYSIS/alexnet_model.pth')

     # Load the model
     model.load_state_dict(torch.load('/content/drive/MyDrive/Analysis/PROJECT_ANALYSIS/alexnet_model.pth'))
```

```
<ipython-input-13-190d95dfa2df>:5: FutureWarning: You are using `torch.load` with `weights_only=False` (t
  model.load_state_dict(torch.load('/content/drive/MyDrive/Analysis/PROJECT_ANALYSIS/alexnet_model.pth'))
<All keys matched successfully>
```

```
[ ]  from sklearn.metrics import classification_report

     print(classification_report(y_true, y_pred, target_names=train_data.classes))
```

```
[ ]  from torchvision import models
     model = models.resnet18(pretrained=True)
     model.fc = nn.Linear(model.fc.in_features, 10)  # Adjust for CIFAR-10
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning:
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning:
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cach
100%|██████████| 44.7M/44.7M [00:00<00:00, 137MB/s]
```

```
[ ]  transform = transforms.Compose([
         transforms.RandomHorizontalFlip(),
         transforms.RandomCrop(32, padding=4),
         transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
     ])
```

```
[ ] import torch
    import torchvision
    import matplotlib.pyplot as plt
    import numpy as np
    from torchvision import datasets, transforms
    from torch.utils.data import DataLoader

    # Load the CIFAR-10 dataset with the applied transformation
    transform = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding=4),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    train_data = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

    train_loader = DataLoader(train_data, batch_size=4, shuffle=True)

    # Get a batch of training data
    data_iter = iter(train_loader)
    images, labels = next(data_iter)
```

```
[ ]
    # Helper function to unnormalize and show images
    def imshow(img):
        img = img / 2 + 0.5  # Unnormalize the image
        npimg = img.numpy()  # Convert tensor to numpy array
        plt.imshow(np.transpose(npimg, (1, 2, 0)))  # Convert to HWC format
        plt.show()

    # Display images
    imshow(torchvision.utils.make_grid(images))
```

```
[ ] model = SimpleAlexNet(num_classes=10)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Set up the training loop:
    for epoch in range(10):  # For 10 epochs, you can change the number of epochs
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass (compute gradients)
            optimizer.step()  # Update weights

            running_loss += loss.item()  # Accumulate loss
            _, predicted = torch.max(outputs, 1)  # Get predictions
            total += labels.size(0)
            correct += (predicted == labels).sum().item()  # Count correct predictions

        print(f'Epoch {epoch+1}, Loss: {running_loss/len(train_loader):.4f}, Accuracy: {100 * correct / total:.2f}%')
```

```
[ ]  # After training, evaluate the model on the test set
     model.eval()  # Set the model to evaluation mode
     correct = 0
     total = 0
     with torch.no_grad():  # No need to compute gradients during testing
         for inputs, labels in test_loader:
             outputs = model(inputs)
             _, predicted = torch.max(outputs, 1)
             total += labels.size(0)
             correct += (predicted == labels).sum().item()

     print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

```
[ ]  import matplotlib.pyplot as plt

     # Lists to store the loss and accuracy values over each epoch
     train_losses = []
     train_accuracies = []

     # Training loop with loss and accuracy tracking
     for epoch in range(10):
         model.train()
         running_loss = 0.0
         correct = 0
         total = 0
         for inputs, labels in train_loader:
             optimizer.zero_grad()  # Zero the gradients
             outputs = model(inputs)  # Forward pass
             loss = criterion(outputs, labels)  # Compute loss
             loss.backward()  # Backward pass (compute gradients)
             optimizer.step()  # Update weights

             running_loss += loss.item()  # Accumulate loss
             _, predicted = torch.max(outputs, 1)  # Get predictions
             total += labels.size(0)
             correct += (predicted == labels).sum().item()  # Count correct predictions
```

```
    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct / total
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_accuracy)

    print(f'Epoch {epoch+1}, Loss: {epoch_loss:.4f}, Accuracy: {epoch_accuracy:.2f}%')

# Plot training loss and accuracy
plt.figure(figsize=(12, 5))

# Plot loss
plt.subplot(1, 2, 1)
plt.plot(range(1, 11), train_losses, marker='o', label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.grid(True)
```

```
plt.title('Training Loss over Epochs')
plt.grid(True)

# Plot accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, 11), train_accuracies, marker='o', label='Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.grid(True)
plt.show()
```

## References:


1. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In NIPS'12: Proceedings of the 25th International Conference on Neural Information Processing Systems (pp. 1097-1105). Curran Associates, Inc. https://doi.org/10.1145/2999134.2999257

2. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324. https://doi.org/10.1109/5.726791

3. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., & Fei- Fei, L. (2015). ImageNet large scale visual recognition challenge. International Journal of Computer Vision, 115(3), 211-252. https://doi.org/10.1007/s11263-015-0816-y

4. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770-778). https://doi.org/10.1109/CVPR.2016.90

5. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for largescale image recognition. In Proceedings of the International Conference on Learning Representations (ICLR). https://arxiv.org/abs/1409.1556

6. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 1- 9). https://doi.org/10.1109/CVPR.2015.7298594

7. Hinton, G. E., Srivastava, N., & Swersky, K. (2012). Improving neural networks by preventing co-adaptation of

feature detectors. arXiv preprint arXiv:1207.0580. https://arxiv.org/abs/1207.0580

8. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for largescale image recognition. International Conference on Learning Representations. https://arxiv.org/abs/1409.1556

9. Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. In Proceedings of the Neural Information Processing Systems (NIPS) Workshop on Deep Learning. https://arxiv.org/abs/1404.5997

10. Li, J., Yosinski, J., Clune, J., et al. (2016). *Visualizing and Understanding Convolutional Networks*. [1311.2901] Visualizing and Understanding Convolutional Networks

11. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Deep Learning

12. Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) (pp. 315-323). JMLR: W&CP. https://www.jmlr.org/proceedings/papers/v15/glorot11a.html