(Individual Assignment I - MGSC-695-075 – Adv Topics in Mgmt Science)

**Image Classification with CNNs**

*Name: Arham Anwar*

*ID: 261137773*

**Section 1: Introduction**

This project presents an experimental study aimed at applying various CNN techniques learned from class. Two final models are presented – a simple CNN with accuracy of 73% on test set, and an advanced ‘EfficientNet’ inspired model achieving an accuracy of xx% on test set. The models are trained and tested on CIFAR10 dataset. Finally, I also implemented a prebuilt ResNet architecture to see how models built from this assignment perform in comparison. We discuss simple CNN in section 2, EfficientNet implementation in section 3, and ResNet implementation in section 4. Please note, primarily the focus has been on implementing the architectures. Optuna hyperparameter optimization trials were not extensively run keeping assignment timeline in mind.

**Section 2: Simple CNN**

SimpleCNN is a convolutional neural network tailored for image classification tasks on the CIFAR-10 dataset. CIFAR-10 comprises 60,000 32x32 color images across 10 classes, each containing 6,000 images. The architecture of SimpleCNN is crafted to efficiently discern distinctive features from these small images, enabling accurate classification.

2.1. Architecture and Rationale:

2.1.1. Convolutional Layer 1: This layer acts as a feature extractor, detecting low-level features such as edges and textures. The choice of a small kernel size with padding ensures that spatial information is preserved, allowing the network to capture meaningful patterns in the input images without losing spatial resolution.

* Input Channels: 3 (RGB image)
* Output Channels: 32
* Kernel Size: 3x3
* Padding: 1

2.1.2. Pooling Layer 1: By employing max-pooling with a stride of 2, this layer reduces the spatial dimensions of the feature maps by half. This downsampling operation helps in reducing computational complexity and achieving spatial invariance by focusing on the most salient features within each local region.

* Kernel Size: 2x2
* Stride: 2

2.1.3. Convolutional Layer 2: Building upon the features extracted by the previous layer, this deeper convolutional layer is designed to capture more complex patterns and higher-level abstractions. The use of a similar kernel size and padding ensures consistency in feature extraction while increasing the network's capacity to learn intricate patterns.

* Input Channels: 32
* Output Channels: 64
* Kernel Size: 3x3
* Padding: 1

2.1.4. Pooling Layer 2: Similar to the first pooling layer, the second pooling layer further reduces the spatial dimensions of the feature maps, promoting translation invariance and robustness to spatial transformations. This additional downsampling also helps prevent overfitting by providing a more abstracted representation of the learned features.

* Kernel Size: 2x2
* Stride: 2

2.1.5. Flattening: Flattening the output from the last pooling layer into a single vector is necessary to connect it to the fully connected layers. This transformation enables the network to transition from spatial feature maps to a linear feature vector, facilitating the subsequent dense connections in the fully connected layers.

2.1.6. Fully Connected Layer 1: This densely connected layer learns non-linear combinations of features extracted by the convolutional and pooling layers. The choice of 512 output features strikes a balance between model capacity and computational efficiency, allowing the network to capture a rich representation of the input while avoiding overfitting.

* Input Features: 64 \* 8 \* 8
* Output Features: 512

2.1.7. Fully Connected Layer 2: Serving as the output layer, this fully connected layer utilizes the learned features to classify the input image into one of the 10 CIFAR-10 classes. With 10 output features corresponding to each class, the network's final predictions are based on the combination of learned features, enabling accurate classification.

* Input Features: 512
* Output Features: 10

2.2. Training Process:

* Optimizer: Adam optimizer with default parameters.
* Learning Rate: Initially set to 0.001.
* Regularization: Dropout with a rate of 0.5 applied to fully connected layers.
* Number of Epochs: Trained for 50 epochs.
* Data Augmentation: Applied random horizontal flipping and random cropping during training to augment dataset.

2.3. Evaluation Results:

* Confusion Matrix:

A graph with numbers and lines

Description automatically generated

* Validation Curve:

A graph of loss curves

Description automatically generated

* Classification Report:

A screenshot of a computer screen

Description automatically generated

**Section 3: Efficient Net Implementation from Scracth (Trained only on CIFAR10 training set)**

In my implementation, I built an EfficientNet architecture using PyTorch and PyTorch Lightning for training. EfficientNet has been lauded for its ability to achieve state-of-the-art performance on image classification tasks by carefully balancing model depth, width, and resolution. Research paper (<https://arxiv.org/pdf/1905.11946>) was used as reference in this attempt to replicate the architecture.

* 1. EfficientNet Architecture: The architecture followed the EfficientNet design principles, incorporating mobile inverted bottleneck convolutional (MBConv) blocks and squeeze-and-excitation (SE) blocks. MBConv blocks were at the heart of the architecture, consisting of depthwise separable convolutions followed by squeeze-and-excitation blocks, which helped in extracting features efficiently. Squeeze-and-excitation blocks played a crucial role in recalibrating channel-wise feature responses. I also included stochastic depth regularization, which randomly dropped entire layers during training to enhance generalization.
  2. Implementation Details: To construct the EfficientNet architecture, I defined custom modules like `CNNBlock`, `SqueezeExcitation`, and `InvertedResidualBlock`. The `EfficientNet` class initialized the model based on the chosen version (e.g., b0, b1) and the number of output classes. Using the `calculate\_factors` method, I determined the width factor, depth factor, and dropout rate specific to the chosen EfficientNet version. The `create\_features` method was responsible for constructing the feature extraction backbone using MBConv and SE blocks.For the forward pass through the model, I implemented the `forward` method.

A diagram of a diagram

Description automatically generated with medium confidence

* 1. Training Process
* Leveraging PyTorch Lightning, I extended the `LightningModule` class to implement `training\_step`, `validation\_step`, and `configure\_optimizers` methods.
* During training and validation, I utilized the `CrossEntropyLoss` criterion to compute the loss.
* To optimize the model parameters, I employed the Adam optimizer with a ReduceLROnPlateau scheduler, which dynamically adjusted the learning rate based on validation performance.
* Throughout the training loop, I logged training loss and validation metrics such as validation loss and accuracy.
  1. Decisions and Considerations:
* I opted for the EfficientNet architecture due to its proven state-of-the-art performance and computational efficiency.
* To enhance modularity and reusability, I designed custom modules encapsulating different components of the architecture.
* Incorporating stochastic depth regularization was a deliberate choice to prevent overfitting and improve the model's ability to generalize.
* Following best practices in deep learning, I integrated batch normalization, dropout, and adaptive learning rate scheduling into the training process.

5. Evaluation Results:

- After training, I evaluated the model's performance on a separate test dataset, assessing metrics such as accuracy, precision, recall, and F1-score.

- My goal was to achieve accuracy levels comparable to or surpassing simple baseline models reported in the literature for the chosen dataset (e.g., CIFAR-10, ImageNet).

**Thank You**