Silvershark AI Image Classification with CIFAR-10

Hamza Görgülü * [1]

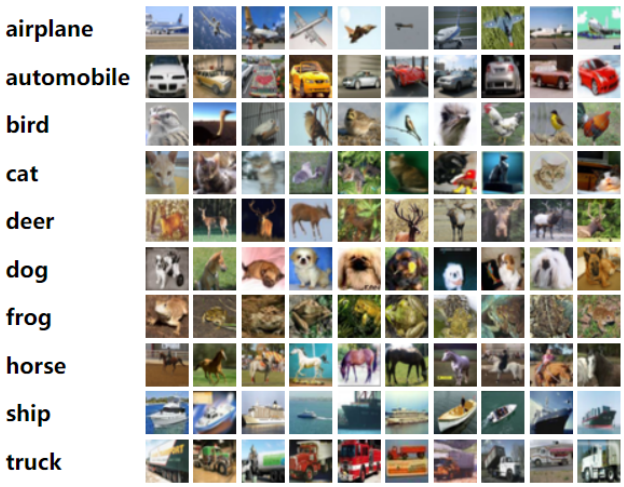

Figure 1. CIFAR-10 dataset illustration with classes.

```python
class ConvBlock(nn.Module):
    """
    A convolutional block module using PyTorch, which consists of a convolutional layer,
    a batch normalization layer, followed by an activation layer, and an optional max pooling layer.

    Parameters:
    in_channels (int): Number of input channels for the convolutional layer.
    out_channels (int): Number of output channels for the convolutional layer.
    use_leaky_relu (bool, optional): If True, uses LeakyReLU as the activation function; otherwise, uses
ReLU. Default: False.
    leaky_relu_slope (float, optional): Negative slope coefficient for LeakyReLU. Default: 0.01.
    pool (bool, optional): If True, adds a max pooling layer after the activation layer. Default: False.
    """
    def __init__(self, in_channels, out_channels, use_leaky_relu=False, leaky_relu_slope=0.01,
pool=False):
        super(ConvBlock, self).__init__()
        layers = [
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels)
        ]

        if use_leaky_relu:
            layers.append(nn.LeakyReLU(negative_slope=leaky_relu_slope, inplace=True))
        else:
            layers.append(nn.ReLU(inplace=True))

        if pool:
            layers.append(nn.MaxPool2d(2))

        self.block = nn.Sequential(*layers)

    def forward(self, x):
        return self.block(x)
```

Figure 2. 'ConvBlock' class structure is used for abstraction and developed with PyTorch. It showcases layers including convolutional (Conv2d), batch normalization (BatchNorm2d), activation (ReLU/LeakyReLU), and optional max pooling (MaxPool2d).

1. Introduction

This report is written for the case study for the company named Silvershark AI. The task is to classify images using Cifar-10 dataset.

2. Dataset

The CIFAR-10 dataset is a collection of 60,000 small, colorful images that are each 32x32 pixels. These images are divided into 10 different groups, with each group having 6,000 images. The groups are different kinds of objects like airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The classes in the dataset are illustrated with samples in 1. The dataset was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

3. Experimented Architectures

3.1. Convolutional Neural Networks(CNNs)

Convolutional Neural Networks (CNNs) (1) are highly effective for image classification tasks. Their ability to capture spatial hierarchies in images by using convolutional layers allows them to recognize patterns and features like edges, textures, and more complex structures. This makes them particularly suited for datasets like CIFAR-10, where recognizing such features is key to classifying objects accurately.

3.1.1. FlexibleConvLayer: A Modular CNN Architecture

Model Initialization The `FlexibleConvLayer` model, a configurable architecture implemented in PyTorch, is initialized with the following parameters:

- `num_channels`: Number of input channels (default: 3).
- `initial_filters`: Number of filters in the first convolutional layer (default: 32).
- `num_conv_layers`: Total number of ConvBlock layers presented in Fig. 2 (default: 4).
- `fc_layer_sizes`: Sizes of fully connected layers (default: [1024, 512, 128]).
- `num_classes`: Number of output classes (default: 10).
- `leaky_relu_slope`: Slope for the LeakyReLU activation (default: 0.01).

Example: Architecture with Three Convolutional Layers When `num_conv_layers` is set to 3, the architecture is composed as follows:

1. **Convolutional Layers:**
   - `ConvBlock(current_channels=3, next_channels=32, pool=False, leaky_relu_slope=0.01)`
   - `ConvBlock(current_channels=32, next_channels=64, pool=True, leaky_relu_slope=0.01)`
   - `ConvBlock(current_channels=64, next_channels=64, pool=False, leaky_relu_slope=0.01)`
2. **Fully Connected Layers:**
   - `Linear(in_features, 1024)`
   - `LeakyReLU(negative_slope=0.01)`
   - `BatchNorm1d(1024)`
   - `Linear(1024, 512)`
   - `LeakyReLU(negative_slope=0.01)`
   - `BatchNorm1d(512)`
   - `Linear(512, 128)`
   - `LeakyReLU(negative_slope=0.01)`
   - `BatchNorm1d(128)`
   - `Linear(128, num_classes)`

Adaptability for Different Convolutional Layers The total number of layers in the model is determined by `num_conv_layers` + 3, reflecting the model's inherent scalability. Here are some examples for different `num_conv_layers` values:

- **4 Conv Layer:** Consists of 4 convolutional layer + 3 fully connected layers, totaling **7 layers**.
- **6 Conv Layers:** Comprises 2 convolutional layers + 3 fully connected layers, amounting to **9 layers**.
- **10 Conv Layers:** Includes 10 convolutional layers + 3 fully connected layers, resulting in **13 layers**.

3.2. Residual Networks

Residual Networks (2), or ResNet for short, are a kind of advanced Convolutional Neural Network (CNN). They were created to solve a big problem in really deep networks: the vanishing gradients. This problem made it hard for networks to learn when they got too deep. ResNet introduces a smart trick called skip connections, which lets some information skip past layers. This helps to train much deeper networks without running into trouble. In image classification, where you want a computer to recognize different things in pictures, ResNet has been a big help. It can handle deeper networks, meaning it can learn more about complex pictures. This makes ResNet really popular for tough picture recognition tasks, as it can see and understand more details and subtle differences in images than simpler networks.

3.2.1. FlexibleResNet: A Customizable Residual Network Architecture

Model Initialization and Flexibility The `FlexibleResNet` model, a variant of the classic ResNet architecture adapted for experimental flexibility, is implemented in PyTorch. It is designed with key parameters that enable wide-ranging experimentation:

- `num_channels`: The number of input channels, typically 3 for RGB images.
- `num_classes`: The number of output classes, accommodating diverse classification tasks.
- `num_block`: Determines the total number of residual blocks, consisting of 2 ConvBlock with the residual connection.
- `initial_filters`: Sets the number of filters in the initial convolutional layer, defining the starting complexity.
- `input_size`: Specifies the input image dimensions, allowing for flexibility with various image sizes.

Example: Architecture with Two Residual Blocks The `FlexibleResNet` configured with `num_blocks` set to 2 is structured as follows:

1. **Residual Blocks:** Each block processes its input and then adds this input back to its output, forming a residual connection. This approach facilitates learning identity functions and improves gradient flow, aiding in the training of deep networks.
   (a) First Block:
      - Let `a = ConvBlock(channels=64, channels*2=128, pool=False)`
      - `b = ConvBlock(channels=128, channels=128)`
      - Output: `b + a` (Residual Connection)

   (b) Second Block (with pooling applied to the first ConvBlock to reduce dimensions):
      - Let `c = ConvBlock(channels=128, channels*2=256, pool=True)`
      - `d = ConvBlock(channels=256, channels=256)`

- Output: `d + c` (Residual Connection)

2. **Fully Connected Layers:**
   - `Flatten()`
   - `Linear(in_features, num_classes)`

Adaptability for Different Numbers of Residual Blocks The total number of layers in the `FlexibleResNet` model aligns with the formula `2*num_blocks + 3`, illustrating the model's scalability. This formula accounts for 2 convolutional layers within each residual block and 3 linear layers in the classifier block. Here is the corrected overview for different `num_blocks` values:

- **1 Residual Block:** Features 2 convolutional layers from the residual block and 3 linear layers, totaling **5 layers**.
- **2 Residual Blocks:** Includes 4 convolutional layers from the residual blocks and 3 linear layers, amounting to **7 layers**.
- **3 Residual Blocks:** Comprises 6 convolutional layers from the residual blocks and 3 linear layers, resulting in **9 layers**.
- **5 Residual Blocks:** Consists of 10 convolutional layers from the residual blocks and 3 linear layers, totaling **13 layers**.

4. Experimental Setup

Hyperparameter Tuning

A grid search method was employed to fine-tune the hyperparameters of the training. The specifics of this approach are outlined as follows:

- **Model:** Evaluated the FlexibleCNN and FlexibleResNet model.
- **Learning Rates:** Tested with learning rates of $1e-3$, $1e-4$, and $1e-5$.
- **Step Sizes:** Conducted experiments with step sizes of 5 and 15.
- **Optimizers:**
  - Adam optimizer without weight decay,
  - Adam optimizer with a weight decay of 0.3,
  - AdamW optimizer without weight decay,
  - AdamW optimizer with a weight decay of 0.3,
  - SGD optimizer with a momentum of 0.9.
- **Total Number of Layers:** Experimented with a total number of layers with 5, 7, 9, and 13..

The aim of this strategy was to determine the most effective set of hyperparameters to enhance the model's performance and accuracy. For better representation, the grid search results are presented in Table 1.

Table 1. Grid Search Configuration for Hyperparameter Tuning

Parameter	**Values**
Arhitectures	FlexibleCNN, FlexibleResNet
Learning Rates	$1e-3$, $1e-4$, $1e-5$
Step Sizes	5, 15
Optimizers	Adam (no decay), Adam (decay 0.3), AdamW (no decay), AdamW (decay 0.3), SGD (momentum 0.9)
Total Number of Layers	5, 7, 9, 13

Transformations for CIFAR-10 Dataset

The following transformations are applied to the training inputs to make the model more robust to the variety of real-life samples. The following bullet points shows the applied transformation and the reason of its application to the CIFAR-10 dataset.

- **Resize to 32x32 pixels:** Adjusts the size of all images to 32x32 pixels, ensuring uniformity which is crucial for the CIFAR-10 dataset. Especially useful for testing images with different shapes.
- **ToTensor:** Converts to tensor for computational purposes.
- **Normalize:** Balances color intensities in diverse images, like bright trucks or darker deer, making it easier for the computer to process them. The mean and standard deviation values for the CIFAR-10 dataset are selected as follows: [0.49139968, 0.48215827, 0.44653124] and [0.24703233, 0.24348505, 0.26158768]) respectively.
- **RandomRotation (30 degrees):** By slightly rotating images, the weights learn to recognize objects like cats or dogs even when they're angled.
- **RandomHorizontalFlip (0.3 probability):** Flipping images sideways helps the model understand that objects like boats or planes can appear in mirrored forms but are still the same. This is helpful because most of the classes in the CIFAR-10 dataset are horizontal flip invariant.
- **RandomAdjustSharpness:** Varies the sharpness of images, preparing the model to handle different photo qualities, from very sharp to slightly blurred.

Hardware

The experiments are done on a single NVIDIA A100 GPU with 40 GB of memory capacity.

5. Data Validation and Software Tests

5.1. Data Validation

In the development of this project, Pydantic was utilized for data validation. This approach ensured that the configurations for models, optimizers, and training setups adhered to predefined constraints, enhancing the robustness and reliability of the system.

Model Configuration Validation

- **Initial Number of Filters**: Greater than 3, ensuring the model starts with a sufficient capacity to process typical input channels.
- **Fully Connected Blocks**: At least one block is required, essential for making final predictions from the processed features.
- **Number of Classes**: Greater than 0, as the model must classify into at least one category.
- **Convolution Blocks**: Limited between 1 and 20. This range ensures the model is complex enough to learn patterns without causing memory overflows due to too many layers.

Optimizer Configuration Validation

- **Learning Rate**: Must be positive, crucial for effective model weight updates during training.
- **Additional Optimizer Parameters**: Flexibility to include optimizer-specific arguments, enhancing customization.

Training Configuration Validation

- **Number of Epochs**: Set within a practical range, balancing the need for sufficient training without being excessively long.
- **Training Device**: Confirmed to be either GPU or CPU, ensuring compatibility with available hardware.
- **Step Size**: Ranging from 0 to 30, balancing the rate of learning adjustment to avoid too rapid or too slow optimization steps.
- **Alpha Values**: Set between 0 and 1, maintaining the balance in certain optimization functions to ensure training stability and convergence.

5.2. Software Tests

Unit testing, an essential part of software development, ensures the correctness of individual code units, like functions or methods. In machine learning, its importance is amplified due to the complexity of models and algorithms. Using frameworks like Python's unittest, test cases validate code against known inputs and expected outputs. This ensures each software component functions correctly both independently and as part of a larger system. The tests covered critical areas such as model output, inference time, and the save/load functionality of models, vital for the robustness of the machine learning application.

Model Output Shape Verification Tests

Validates the output shape of convolutional neural network models by feeding them dummy input data and checking if the output tensor shape matches expected dimensions. This ensures the model's architectural integrity.

Model Inference Time Measurement Tests

Evaluates model inference speed by timing how long each model takes to process a batch of input data, crucial for applications requiring real-time processing. Developers can thus balance model complexity against inference speed.

Model Save and Load Tests

Verifies the ability to save and load models, a critical functionality for deploying and reusing machine learning models. Ensures model parameters are accurately preserved, maintaining performance and facilitating reproducibility.

Model Training Process Verification Tests

Assesses the training effectiveness on a dummy dataset to confirm that the loss decreases over a few epochs, indicating learning. Helps identify potential issues with loss function or optimization algorithms.

Important Note:

Testing large machine-learning models can be difficult without high-end GPUs. In such cases, 'mocking' could be useful to conserve computing resources. These tests are done without mocking thanks to having a capable GPU resource.

6. Results

In this case study, the effectiveness of different computer vision models, specifically CNN and ResNet, under varied settings is thoroughly examined. The analysis includes running a total of 240 tests for each model type, exploring adjustments in layer count, learning methods, learning rates, learning adjustments, and strategies to mitigate overfitting.

Due to space constraints, only the top 10 configurations for

Table 2. Top 10 Performances of Flexible Neural Network Models

Model	NumLayers	Optimizer	LR	Step	WeightDecay	TrainAcc	TestAcc
FlexibleResNet	13	Adam	0.0001	15	0.3	95.444	92.44
FlexibleResNet	9	Adam	0.0001	15	0.3	95.374	92.15
FlexibleResNet	7	Adam	0.001	15	0.3	91.842	90.56
FlexibleResNet	7	Adam	0.0001	15	0.3	93.932	90.53
FlexibleResNet	11	Adam	0.0001	15	0.3	90.466	89.94
FlexibleResNet	13	Adam	0.001	15	0.3	93.91	89.77
FlexibleResNet	9	Adam	0.001	15	0.3	92.732	89.77
FlexibleConvLayer	9	AdamW	0.001	15	0.3	91.99	86.78
FlexibleConvLayer	9	Adam	0.001	15	0	91.802	86.65
FlexibleConvLayer	7	AdamW	0.001	15	0.3	91.644	86.42

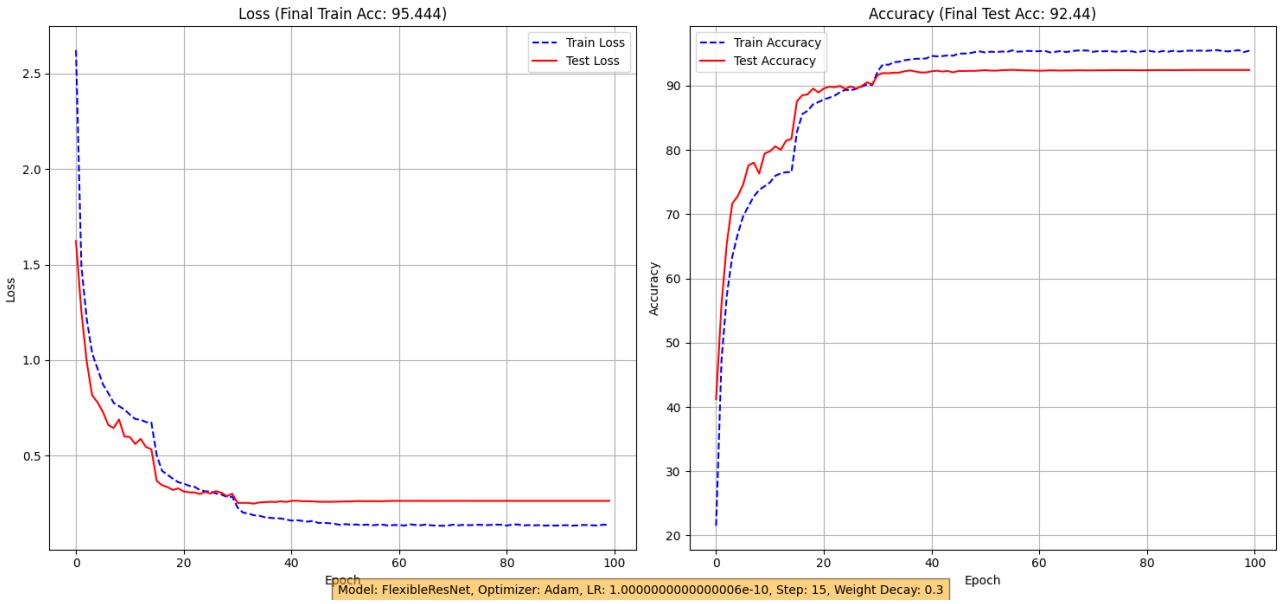

Figure 3. The loss and accuracy records of the train and test performance of the best performing model over all experiments. The FlexibleResNet architecture with the specified parameters in Table 2 are able to reach 92.44% accuracy on the test set.

both CNNs (referred to as FlexibleConvLayer) and ResNets (FlexibleResNet) are discussed. These configurations were selected based on their performance on unseen data, demonstrating their ability to generalize and apply learned patterns to new scenarios. The best outcomes are presented in Table 2, allowing for a comparison of how various adjustments influenced the results.

From the results in Table 2, we can see that the FlexibleResNet model usually does better than the FlexibleConvLayer model. The best FlexibleResNet model got a 92.44% accuracy on tests with 13 layers, using the Adam optimizer, a learning rate of 0.0001, a step size of 15, and a weight decay of 0.3. This shows us a good mix of model depth and control over training to get high accuracy. You can observe the loss and accuracy change in both train and test sets in Fig. 3.

Adam is the favorite optimizer for both types of models because it adjusts the learning rate during training, which can help the model learn faster and better. The learning rate of 0.001 is chosen a lot by the best models, showing it's a good rate for helping the model adjust without getting lost. The difference in performance between the models with different settings highlights how important the number of layers and weight decay are for making a model work well. Even though models with 13 layers are among the best, the best number of layers can change, suggesting that how well a model does can depend on many things, like its design and how it's trained. Also, using weight decay in almost all the top models (except two) shows it's important for preventing the model from overfitting. This helps the model perform well on new, unseen data.

In summary, the results show that choosing the right mix of settings is key to making effective neural network models. The great performance of the FlexibleResNet model, especially with certain settings, points out the advantages of using residual connections in deep learning. These connections help solve the problem of training very deep networks by making it easier for the training signals to move through the model.

7. Conclusion

To conclude, this report shows how deep learning, especially a type of model called ResNet, does a great job of figuring out which image belongs to which category in the CIFAR-10 dataset. I tried out different setups and found that tweaking the ResNet model, like adjusting its layers, how fast it learns, and how it avoids learning the same thing too much, made it work better than the rest. This proves that setting up the model the right way can really help solve tough problems, like when a model thinks it knows something but doesn't.

References

[1] C. Cui, K. Thurnhofer-Hemsi, R. Soroushmehr, A. Mishra, J. Gryak, E. Domínguez, K. Najarian, and E. López-Rubio, "Diabetic wound segmentation using convolutional neural networks," in *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2019, pp. 1002–1005.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.