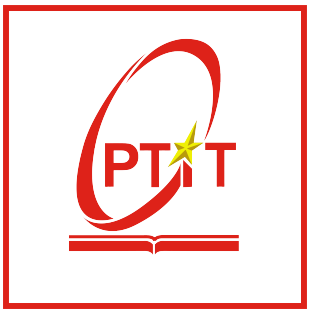
POSTS AND TELECOMMUNICATIONS
INSTITUE OF TECHNOLOGY
INFORMATION TECHNOLOGY



PYTHON PROGRAMMING

REPORT

IMAGE CLASSIFICATION

Name: NGUYEN PHAM TRUNG HIEU
Student ID: B23DCCE031
Class ID: B23CQCE04-B
Instructor: KIM NGOC BACH

Hanoi

Table of contents

Table of contents

1 Introduction

1.1 Abstract

This study conducts a detailed performance comparison and evaluation between two neural network architectures: an advanced Multi-Layer Perceptron, MLPnet and a ResNet-34, in the context of image classification tasks on the CIFAR-10 dataset. The results indicate that ResNet-34 significantly outperforms MLPnet, primarily due to the CNN architecture's effective spatial feature extraction and the benefits of residual connections. The report provides an in-depth analysis of the architectures, training processes, quantitative and qualitative metrics, and proposes avenues for improvement.

1.2 Goal

This report aims to achieve the following primary objectives:

- Performance Comparison: Quantitatively and qualitatively evaluate the classification performance of Model A (MLPnet) and Model B (ResNet-34) on the CIFAR-10 dataset.
- Architectural Analysis: Explain the performance differences based on the architectural characteristics of the two models.
- Technique Evaluation: Examine the impact of techniques such as Dropout, Batch Normalization, Data Augmentation, and Weight Decay.
- Improvement Recommendations: Provide recommendations for optimizing the models and suggest potential future research directions.

2 Background Knowledge

2.1 Multilayer Perceptron

- A Multi-Layer Perceptron (MLP) is a foundational class of feedforward artificial neural networks (ANNs). It consists of at least three types of layers: an input layer, one or more hidden layers, and an output layer. Each layer is composed of several nodes (or neurons). The input layer receives the initial data (features). Neurons in the hidden layers and output layer perform a weighted sum of their inputs (from the previous layer) and then apply a non-linear activation function to this sum. This non-linearity is crucial as it allows MLPs to learn complex, non-linear relationships in the data.
- **Key Components of an MLP**:
  - Neurons (Nodes): The basic computational units. Each neuron receives inputs, computes a weighted sum, adds a bias, and then passes the result through an activation function.
  - Layers: Neurons are organized into layers.
  - Input Layer: Receives the raw input features. The number of neurons in this layer corresponds to the number of features in the input data.

  - Hidden Layers: These layers are between the input and output layers and are responsible for learning complex patterns and representations from the data. The number of hidden layers and the number of neurons in each hidden layer are key hyperparameters that define the MLP's capacity.
  - Output Layer: Produces the final output of the network. For classification tasks, the number of neurons typically corresponds to the number of classes, and an activation function like Softmax is often used to produce probability distributions over the classes.
  - Weights and Biases: Each connection between neurons has an associated weight, which determines the strength of the connection. Each neuron (except input neurons) also has a bias term. These weights and biases are the parameters that the network learns during the training process.
  - Activation Functions: Introduce non-linearity into the network, enabling it to learn more than just linear mappings. Common activation functions include Sigmoid, Tanh, ReLU (Rectified Linear Unit), and GELU (Gaussian Error Linear Unit).
  - Fully Connected (Dense) Layers: In a standard MLP, every neuron in one layer is connected to every neuron in the subsequent layer. This is why they are often referred to as dense layers.

- Learning Process: MLPs are typically trained using a supervised learning algorithm called backpropagation. This process involves:
  - Forward Propagation: Input data is fed through the network, layer by layer, to produce an output.
  - Loss Calculation: The output is compared to the true target value using a loss function (e.g., CrossEntropyLoss for classification) to quantify the error.
  - Backward Propagation: The error is propagated backward through the network, from the output layer to the input layer. During this process, the gradient of the loss function with respect to each weight and bias is calculated.
  - Weight Update: An optimization algorithm (e.g., Stochastic Gradient Descent (SGD), Adam) uses these gradients to update the weights and biases in a direction that minimizes the loss.

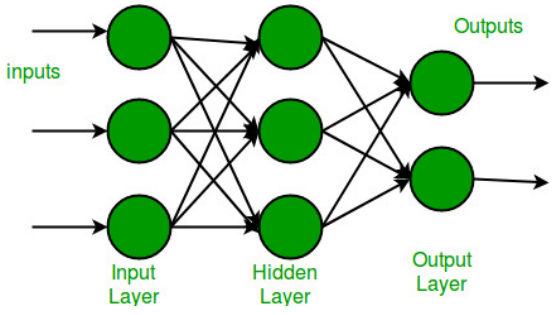


Figure 1: basic MLP

- The MLPnet model in this study attempts to enhance the basic MLP structure by incorporating modern techniques such as deeper architectures, Batch Normalization, GELU

activation, residual-like connections, and an auxiliary output to improve its learning capability.

2.2 Convolutional Neural Network

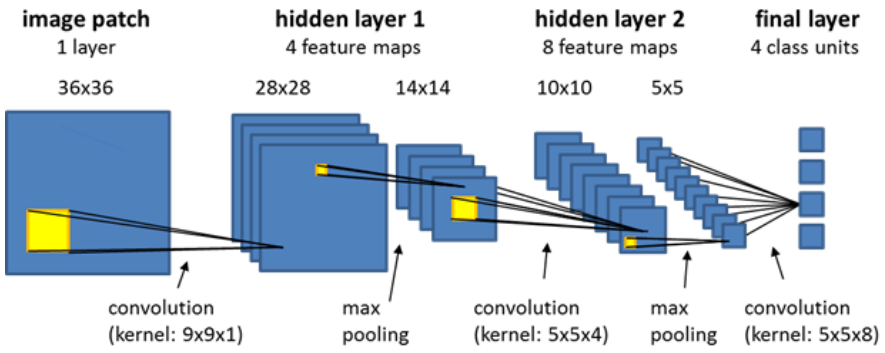


Figure 2: CNNs

- Convolutional Neural Networks (CNNs) are a specialized class of deep neural networks that have revolutionized the field of computer vision. They are designed to automatically and adaptively learn spatial hierarchies of features from input data, such as images.
- **Key Architectural Concepts of CNNs**
  - **Convolutional Layers:** The core building block of a CNN. Instead of full connections like in Multi-Layer Perceptrons (MLPs), convolutional layers use a set of learnable filters (or kernels). Each filter is small spatially (e.g., 3×3 or 5×5 pixels) but extends through the full depth of the input volume. During the forward pass, each filter is convolved (slid) across the width and height of the input volume, computing the dot product between the filter entries and the input at any position. This produces a 2D activation map (or feature map) of that filter. The network learns filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network.
  - **Local Receptive Fields:** Each neuron in a convolutional layer is connected only to a small, localized region of the input volume (the receptive field). This allows the network to focus on local patterns.
  - **Shared Weights (Parameter Sharing):** A crucial property of convolutional layers is that a single filter is used across different spatial locations in the input. This drastically reduces the number of parameters compared to MLPs and makes the network equivariant to translations of features (if a feature is useful to detect in one part of the image, it's likely useful in other parts too).
  - **Activation Functions (e.g., ReLU):** Applied after convolutional operations to introduce non-linearity, allowing the network to learn more complex features. The Rectified Linear Unit (ReLU) is a common choice, defined as $f(x) = \max(0, x)$.

- **Pooling Layers (e.g., Max Pooling, Average Pooling):** These layers are often inserted between successive convolutional layers to progressively reduce the spatial size (width and height) of the representation, thereby reducing the number of parameters and computation in the network. Pooling also helps to make the feature representations somewhat invariant to small translations of the input. Max pooling, for example, takes the maximum value from a small grid of the feature map.
- **Hierarchical Feature Learning:** CNNs typically stack multiple convolutional and pooling layers. Early layers learn low-level features (e.g., edges, corners, textures), while deeper layers combine these to learn higher-level, more abstract features (e.g., parts of objects, entire objects). This hierarchical structure mimics aspects of the human visual cortex.
- **Fully Connected Layers:** After several convolutional and pooling layers, the high-level features are typically flattened into a one-dimensional vector and fed into one or more fully connected layers (similar to an MLP) to perform the final classification or regression task.

2.3 Residual Networks (ResNet)

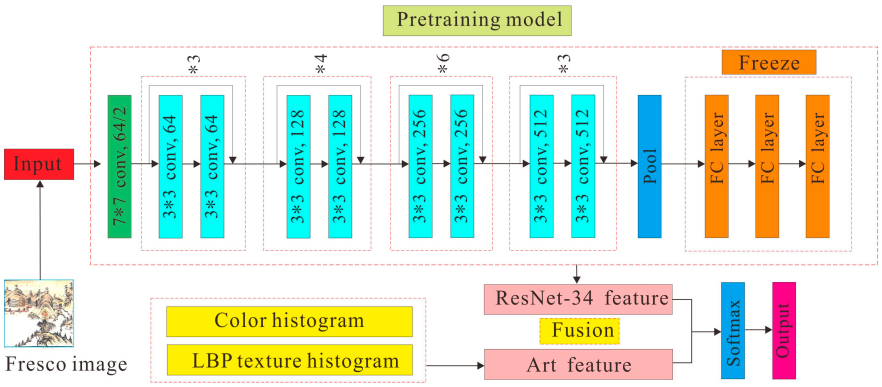


Figure 3: resNet34

- As CNNs become deeper, they can suffer from the **vanishing gradient problem**, where gradients become extremely small as they are propagated backward from the output layer to the earlier layers during training via backpropagation. This makes it very difficult to train very deep networks effectively, as the weights in the earlier layers do not update significantly, or update very slowly.
- ResNet, introduced by He et al. in 2015, addresses this problem by introducing **residual connections** (or skip connections).
- **Residual Learning**
  - A residual block in ResNet learns a residual mapping. Instead of attempting to learn an underlying mapping $H(x)$ directly with a stack of layers, the layers in a residual block are made to learn a residual function $F(x) = H(x) - x$. The original desired mapping is then recast as $F(x) + x$. The hypothesis is that it is easier to optimize the

residual mapping $F(x)$ than to optimize the original, unreferenced mapping $H(x)$. If the identity mapping ($H(x) = x$) were optimal, it would be easier for the layers to learn to set the weights of $F(x)$ to zero than to fit an identity mapping with a stack of non-linear layers.

- Crucially, the "$+x$" term, implemented via an identity skip connection, provides a direct path for the gradient to flow backward through the network. This alleviates the vanishing gradient problem and allows for the training of networks that are substantially deeper (e.g., with 34, 50, 101, 152, or even more layers) than what was previously feasible. This architectural innovation enabled significant breakthroughs in various computer vision tasks.

3 Data Analysis

3.1 Cifar10

- CIFAR-10 (Canadian Institute For Advanced Research, 10 classes) is a widely used benchmark dataset in computer vision and machine learning, specifically for image classification tasks. It's a subset of the larger "80 million tiny images" dataset.
- Content: It comprises 60,000 color images. Each image is small, with a resolution of 32x32 pixels.
- Classes: There are 10 mutually exclusive classes, with exactly 6,000 images per class, making it a balanced dataset. The classes are:
  - Airplane
  - Automobile (often referred to as Car)
  - Bird
  - Cat
  - Deer
  - Dog
  - Frog
  - Horse
  - Ship
  - Truck
- Dataset Split: The dataset is typically pre-divided into:
  - 50,000 images for the training set.
  - 10,000 images for the test set.

Figure 4: cifar10

3.2 Data Preprocessing and Augmentation Techniques

- **Normalization**
  - Purpose: Normalization standardizes the pixel intensity values of the images. This is crucial because it helps to:
    - Ensure that all input features (pixel values across different color channels) have a similar distribution (typically centered around zero with unit variance). This prevents features with larger numerical values from disproportionately influencing the learning process.
    - Stabilize and often speed up the training process by ensuring smoother gradient descents.
    - Make the model less sensitive to minor variations in lighting and contrast in the input images.
  - How it works: For each color channel (Red, Green, Blue), the pixel values (which are usually first scaled to [0, 1] by ToTensor()) are transformed using the formula: output_channel = (input_channel - mean_channel) / std_channel.
- **Random Crop**
  - Purpose: This is a powerful data augmentation technique. It helps the model become more robust to variations in the object's position and scale within the image. It also effectively increases the diversity of the training data by generating slightly different views of the same image.
  - How it works:
    - Padding: The original 32x32 image is first padded with a certain number of pixels around its borders (in your case, 4 pixels on each side). This padding typically uses zeros or reflects the border pixels. After padding with 4 pixels, the image becomes 40x40.
    - Cropping: A 32x32 patch is then randomly selected (cropped) from this padded 40x40 image.

  - Effect: By randomly cropping, the model is exposed to the object of interest in slightly different positions and contexts during training. This forces the model to learn more generalizable features that are not tied to a specific location within the image, reducing the risk of overfitting.

- **Random Horizontal Flip**
  - Purpose: This is another common and effective data augmentation technique. It increases the diversity of the training dataset and helps the model learn that the identity of many objects is invariant to their horizontal orientation (e.g., a car facing left is still a car).
  - How it works: With a certain probability (typically 0.5 by default), the image is flipped horizontally.
  - Effect: This helps the model to generalize better because it learns that the mirrored version of an object belongs to the same class. This is particularly useful for datasets like CIFAR-10 where horizontal flipping often preserves the label's validity (e.g., flipping a cat image still results in a cat image).

4 Model Description

4.1 Model A: MLP

```
class MLPnet(nn.Module):
def __init__(self, input_size=3*32*32, num_class=10):
    super(MLPnet, self).__init__()
    self.fc1 = nn.Linear(in_features = input_size, out_features=512)
    self.fc2 = nn.Linear(in_features = 512, out_features=512)
    self.fc3 = nn.Linear(in_features = 512, out_features=256)
    self.fc4 = nn.Linear(in_features = 256, out_features=256)
    self.fc5 = nn.Linear(in_features = 256, out_features=128)
    self.dropout = nn.Dropout(p=0.7)
    self.fc_last = nn.Linear(in_features = 128, out_features=num_class)
    self.auxil = nn.Linear(in_features = 512, out_features=num_class)
    self.batchnorm = nn.BatchNorm1d(512)
    self.batchnorm2 = nn.BatchNorm1d(256)
    self.batchnorm3 = nn.BatchNorm1d(128)

def forward(self, x):
    x = torch.flatten(x, 1)
    first = self.batchnorm(F.gelu(self.fc1(x)))
    second = self.batchnorm(F.gelu(self.fc2(first))) + first
    third = self.batchnorm2(F.gelu(self.fc3(second)))
    fourth = self.batchnorm2(F.gelu(self.fc4(third))) + third
    x = self.batchnorm3(F.gelu(self.fc5(fourth)))
    x = self.dropout(x)
    out = self.fc_last(x)
    out_auxil = self.auxil(second)
    return out, out_auxil
```

Listing 1: PyTorch implementation of MLPnet

- Flatten Layer: Converts input image tensor (batch_size, 3, 32, 32) into (batch_size, 3072).
- Activation Function: GELU (Gaussian Error Linear Unit) is used, which is a smooth approximation of ReLU and has shown good performance in various models.

- Batch Normalization: BatchNorm1d layers are applied after the activation function in the hidden layers. This helps to stabilize training, allows for higher learning rates, and can have a regularizing effect.
- Dropout: A dropout rate of p=0.7 is applied before the final main classification layer to reduce overfitting.
- Auxiliary Output: An additional linear layer (auxil) predicts classes from an intermediate representation (out2). During training, the loss from this auxiliary output is typically added to the main loss (often with a smaller weight), providing an additional gradient signal to earlier layers and potentially aiding in regularization.

4.2 Model B: ResNet34

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks_list, num_classes=10, in_channels_initial=3):
        super(ResNet, self).__init__()
        self.in_planes = 64


        self.conv1 = nn.Conv2d(in_channels_initial, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)


        self.layer1 = self._make_layer(block, 64, num_blocks_list[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks_list[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks_list[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks_list[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.linear = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for s in strides:
            layers.append(block(self.in_planes, planes, s))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = torch.flatten(out, 1)
        out = self.linear(out)
        return out

def ResNet34(num_classes=10, in_channels_initial=3):
    return ResNet(BasicBlock, [3, 4, 6, 3], num_classes=num_classes, in_channels_initial=in_channels_initial)
```

Listing 2: PyTorch implementation of ResNet34

- First Convolutional Layer (conv1): nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False). This initial layer is adjusted for CIFAR-10's 32x32 images, using a smaller kernel and stride than the original ResNet designed for larger ImageNet images.
- Batch Normalization (bn1): nn.BatchNorm2d(64) is applied after conv1.
- ReLU Activation Function: Applied after bn1.
- Initial MaxPool Removal: self.maxpool = nn.Identity(). The original ResNet architecture includes a max pooling layer after the first convolutional layer. For smaller images like those in CIFAR-10, this initial pooling can discard too much spatial information too early. Replacing it with an identity layer preserves the feature map size.
- Stages (Layer1 to Layer4) using BasicBlock:
- BasicBlock: The fundamental building block for ResNet-18 and ResNet-34. It consists of:
- A 3x3 convolutional layer, followed by Batch Normalization and a ReLU activation.
- Another 3x3 convolutional layer, followed by Batch Normalization.
- A shortcut (skip) connection that adds the input of the block to the output of the second convolutional layer (after its BatchNorm).
- A final ReLU activation applied to the sum.
- If the input and output dimensions (number of channels or spatial size due to striding) of the block do not match, a 1x1 convolution is used in the shortcut path to project the input to the correct dimensions before addition.
- Stage Configuration: ResNet-34 is composed of four stages of these BasicBlocks:
- layer1: 3 BasicBlocks, 64 output channels. Stride is 1, so spatial dimensions (32x32) are maintained.
- layer2: 4 BasicBlocks, 128 output channels. The first block in this stage has a stride of 2, reducing spatial dimensions to 16x16.
- layer3: 6 BasicBlocks, 256 output channels. The first block has a stride of 2, reducing spatial dimensions to 8x8.
- layer4: 3 BasicBlocks, 512 output channels. The first block has a stride of 2, reducing spatial dimensions to 4x4.
- Global Average Pooling (avgpool): nn.AdaptiveAvgPool2d((1, 1)). After the final convolutional stage, the feature maps (e.g., 512 maps of size 4x4) are reduced to a single value per map by averaging. This results in a 512-dimensional feature vector per image.
- Final Fully Connected Layer (fc): nn.Linear(512 * BasicBlock.expansion, num_classes) (where BasicBlock.expansion is 1 for ResNet-34, so nn.Linear(512, 10)). This layer takes the 512-dimensional vector and produces 10 logits for the CIFAR-10 classes.

5 Experimental Results

5.1 Quantitative Results Comparison Table

Table 1: Quantitative Results Comparison of Model A and Model B

Metric	Model A (MLPnet Dropout p=0.7)	Model B (ResNet-34)
Train Accuracy (final)	0.4894 (epoch 10)	**0.9350** (epoch 30)
Train Loss (final)	2.0072 (epoch 10)	**0.1876** (epoch 30)
Validation Accuracy (highest)	0.4601 (epoch 10)	**0.8857** (epoch 30)
Validation Loss (lowest)	1.5211 (epoch 10)	**0.3584** (epoch 26)
Test Accuracy	**0.4776**	**0.8877**

5.2 Learning curves

5.2.1 Model A: MLP

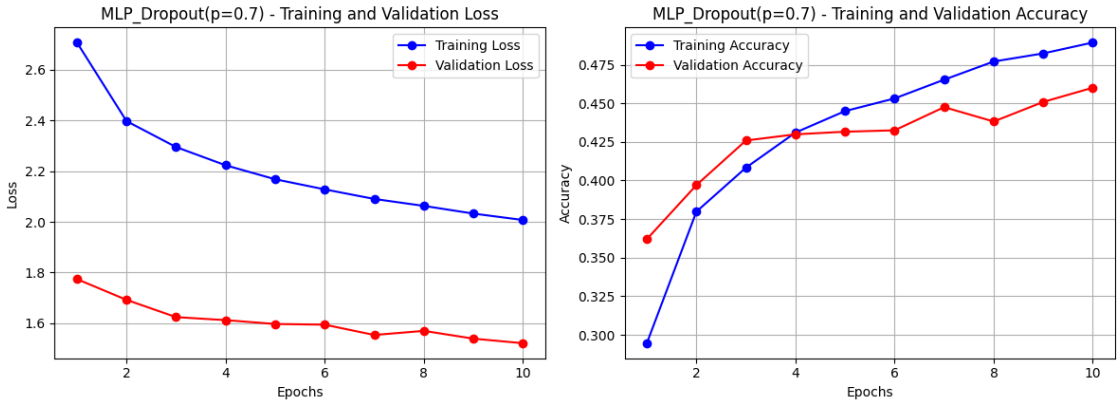


Figure 5: MLP with dropout p = 0.7

- Training Loss: Started at 2.7072 and decreased to 2.0072 over 10 epochs. The decrease is more significant than a very simple MLP but the final loss is still relatively high.
- Validation Loss: Started at 1.7740, and ended at 1.5211.
- Training Accuracy: Increased from 0.2946 to 0.4894. This shows the model is learning from the training data to a moderate extent.
- Validation Accuracy: Increased from 0.3620 to a peak of 0.4601 at epoch 10
- Diagnosis: The model learns better than a basic MLP but still exhibits significant underfitting relative to the task's complexity and the performance of CNNs. The gap between the final training accuracy (48. 94%) and the maximum validation accuracy (46. 01%) also suggests slight overfitting.

5.2.2 Model B: ResNet34

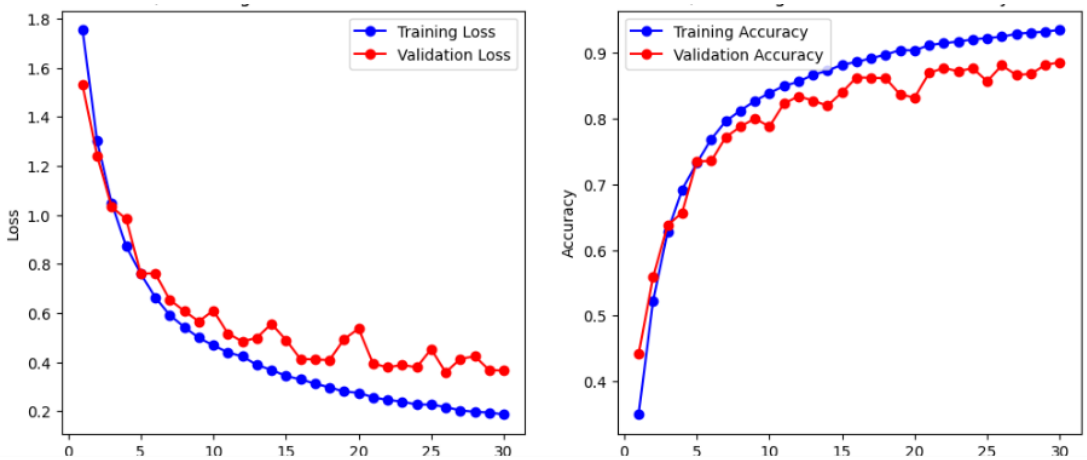


Figure 6: ResNet34

- Training Loss: Started at 1.7543 and decreased to 0.1876 over 30 epochs.
- Validation Loss: Started at 1.5299, and ended at 0.3658.
- Training Accuracy: Increased from 0.3498 to 0.9350. This shows that the model is learning from the training data to a moderate extent.
- Validation Accuracy: Increased from 0.4415 to a peak of 0.8857 at epoch 30
- Diagnosis: Demonstrates a "good fit" with high performance. The anomalous 5% gap between the final training and the validation accuracy indicates a slight, manageable degree of overfitting.

5.3 Confusion Matrix

5.3.1 Model A: MLP

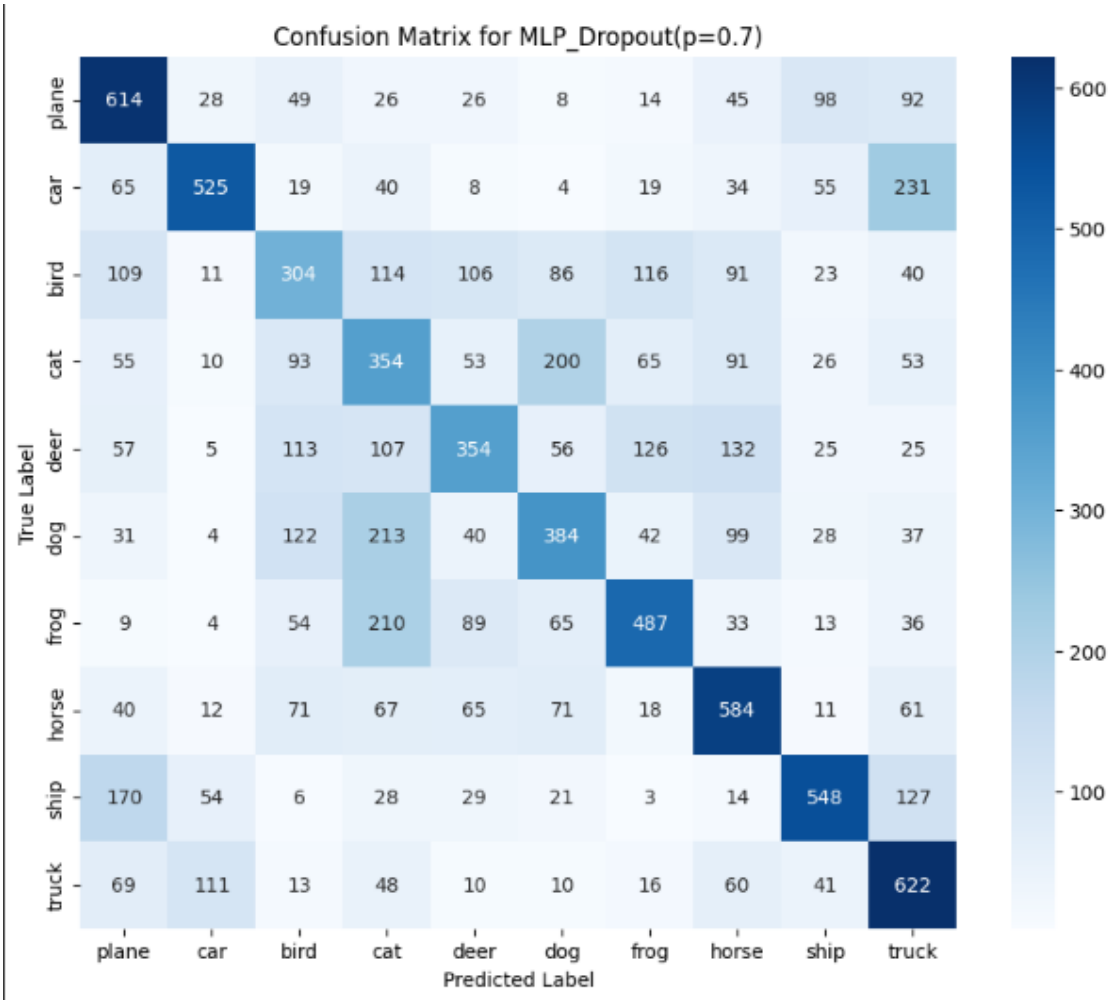


Figure 7: MLP with dropout p = 0.7

- Best Performing Classes: The classes car, ship and plane are the most successfully classified. These are generally objects with more distinct and consistent shapes. The model shows the highest confidence in its predictions for 'car' and 'ship'.
- Worst Performing Classes: The model struggles significantly with cat, bird, dog and deer. These animal classes are notoriously difficult due to variations in breed, color, pose, and background
- Animals Confused with Animals: The most significant confusion occurs between the animal classes. For instance, 'cat' is heavily misclassified as 'dog' (200 instances) and 'frog' (210 instances). Similarly, 'dog' is often mistaken for 'cat' (213 instances). This indicates the MLP struggles to distinguish the fine-grained features that separate these animal categories.

- Vehicles Confused with Vehicles: There is a very high number of 'car' instances misclassified as 'truck' (231 instances). This is an understandable error given their similar features (wheels, metallic bodies, roads).
- Ship and Plane Confusion: A notable number of 'ship' images are misclassified as 'plane' (170 instances), and a smaller but significant number of 'plane' images are mistaken for 'ship' (98 instances). This could be due to similar backgrounds (blue water/sky) and the presence of wings/sails.

5.3.2 Model B: ResNet34

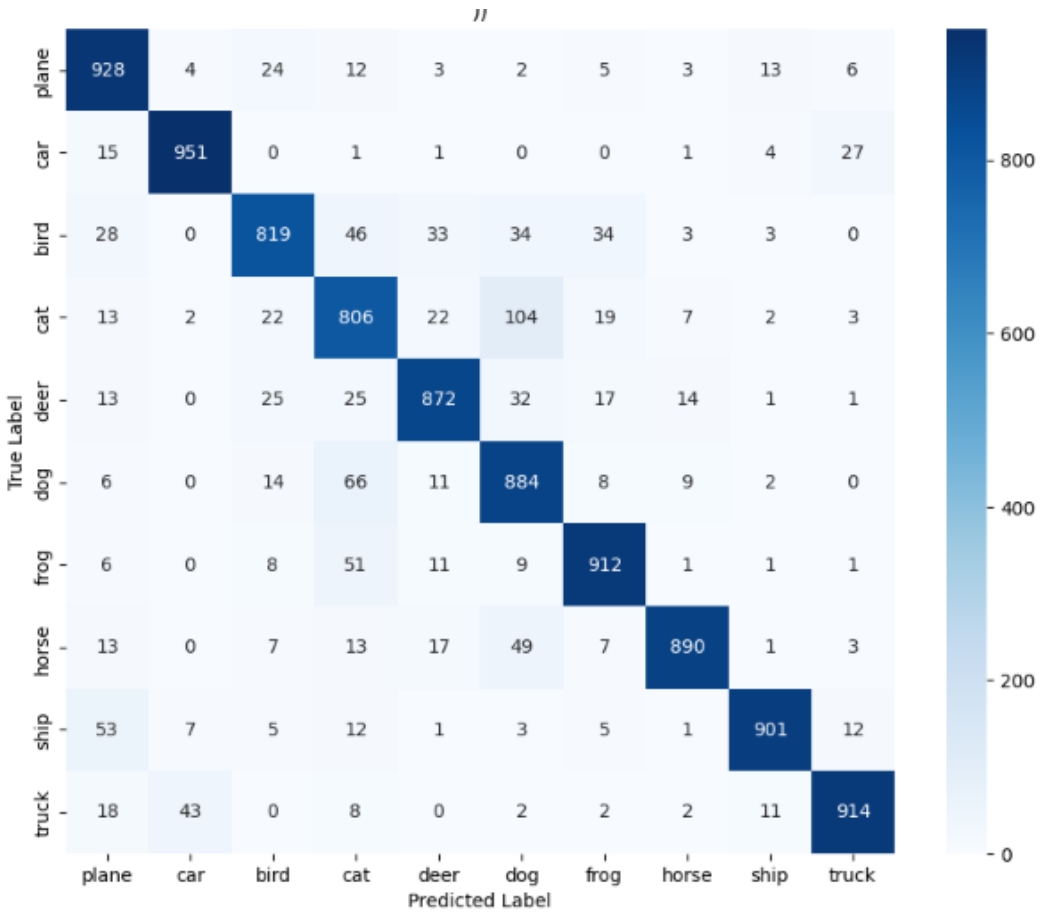


Figure 8: RestNet34

- Best-Performing Classes: The model performs exceptionally well on non-animal objects. The highest correct class is seen for car, frog, ship and truck. These classes often have more distinct shapes and contexts.
- Worst-Performing Classes: The model finds the animal classes more challenging, which is a common result. The lowest accuracies are for cat, dog and bird.

- Key Misclassifications: The most significant confusion occurs between visually similar animals. The matrix shows that 101 'cat' images were misclassified as 'dog', and 58 'dog' images were misclassified as 'cat'. This classic cat-dog confusion remains the model's biggest challenge. Other notable errors include misclassifying birds as planes and deer as birds.

6 Demo ResNet34 with Streamlit

To provide an interactive and accessible way to demonstrate the capabilities of the high-performing ResNet34 model, a web application was developed using the Streamlit framework. This demo allows users to directly interact with the model by uploading their own images and receiving real-time classification results.

6.1 Purpose of the Demonstration

The primary objectives for creating this interactive demonstration were:

- **Accessibility:** To create a user-friendly interface that makes the model's functionality accessible to individuals without a technical background in machine learning.
- **Validation:** To provide a tangible and visual confirmation of the model's high accuracy as reported in the quantitative results.
- **Engagement:** To offer an engaging way to explore the model's strengths and weaknesses by allowing users to test it on a wide variety of images.

6.2 Application Interface and Workflow

The application interface is designed for simplicity and ease of use. The user workflow is as follows:

1. The user is greeted with a title and a brief description of the application.
2. A file uploader widget prompts the user to select an image file (‘.jpg‘, ‘.jpeg‘, ‘.png‘) from their local machine.
3. Upon uploading, the selected image is displayed on the screen for confirmation.
4. The application then sends the image to the ResNet34 model for inference. A spinner indicates that classification is in progress.
5. Finally, the application displays the best predicted classes for the image, along with their corresponding confidence scores, formatted as percentages. The top prediction is highlighted for emphasis.

Figure 9: User interface of streamlit

6.3 Implementation Details

The application is built as a single Python script (‘app.py‘) using the Streamlit library. Key implementation aspects include:

- **Model Loading:** The pre-trained ResNet34 model is loaded from ‘torchvision.models‘. To ensure a fast and responsive user experience, Streamlit’s caching decorator (‘@st.cache_resource‘) is used. This ensures the multi-megabyte model is loaded into memory only once when the application starts, rather than on every user interaction.
- **Image Preprocessing:** Before being passed to the model, uploaded images are processed using the ‘Pillow‘ library and a ‘torchvision.transforms‘ pipeline. This pipeline resizes, crops, and normalizes the image to match the exact input format the ResNet34 model was trained on.
- **Inference:** The model’s prediction is performed within a ‘torch.no_grad()‘ context to optimize for speed and memory usage during inference. The raw output logits from the model are converted into probabilities using a Softmax function.

6.4 Setup and Execution

To run the demonstration locally, the following steps are required. First, all necessary Python dependencies must be installed from the ‘requirements.txt‘ file.

```
# Ensure you are in the project's root directory
pip install -r requirements.txt
```

Once the dependencies are installed, the Streamlit application can be launched with a single command:

```
streamlit run app.py
```

This command starts a local web server and opens the application in the user's default web browser, ready for interaction.

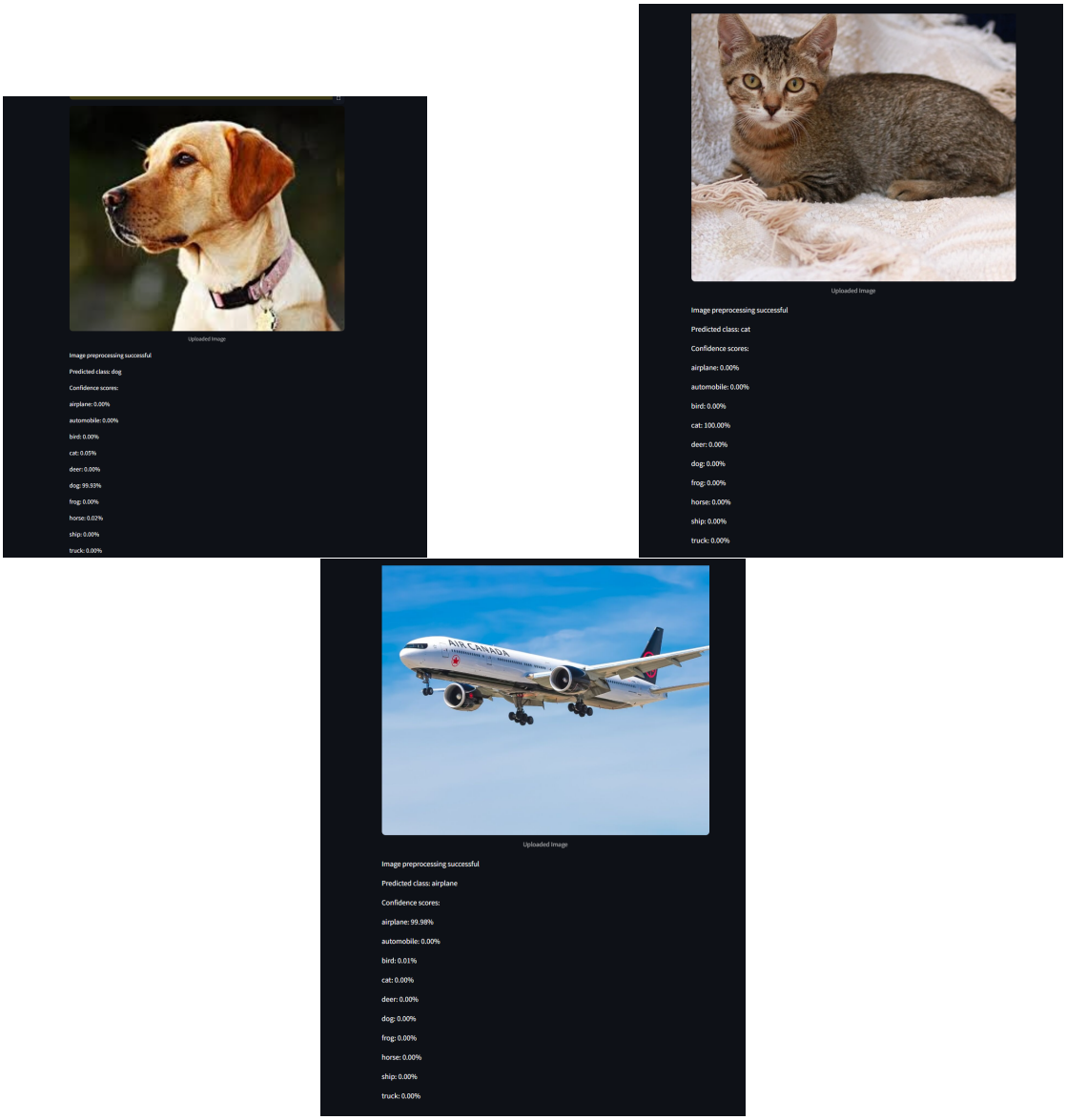


Figure 10: Some images to demo with ResNet34