2110446 DATA SCIENCE AND DATA ENGINEERING

Homework 5: Deep Learning

Computer Engineering, Chulalongkorn Univerity

Worralop Srichainont

February 16, 2026

Contents

Part I
Assignment

1 Objective

In this homework, you are required to write the report from the following Jupyter notebooks:

- Image classification with CNN
- Image classification with EfficientNetV2s
- Image classification with EfficientNetb0 with Weights & Biases
- Image classification with ViT from Hugging Face & TensorBoard (Optional)
- Basic API Call with LangChain

Part II
Image classification with CNN

2 Dependencies

2.1 Import Libraries

Since Google Colab environment doesn't have these libraries install, we need to install them first.

Install Additional Libraries

```
%pip install pytorch-lightning
%pip install torchinfo
```

Then, import these libraries.

Import Libraries

```
from sklearn.metrics import classification_report
from sklearn.metrics import ConfusionMatrixDisplay
from torchmetrics.classification import Accuracy
from torchinfo import summary
from tqdm.notebook import tqdm
from pytorch_lightning.callbacks import ModelCheckpoint

import numpy as np
import matplotlib.pyplot as plt
import torch
import torchvision
import torchvision.transforms as transforms
import pytorch_lightning as pl
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

2.2 Libraries Description

Core Libraries

- `numpy` is the library for processing mathematical operations on an array.
- `matplotlib.pyplot` is the library for plotting and image visualization.
- `torch` is the main PyTorch library which provides tensors that able to calculate inside a GPU.
- `torchvision` is computer vision package for PyTorch which contains popular dataset, model architecture and image processing tools.
- `torchvision.transforms` is for image preprocessing.
- `pytorch_lightning` is the deep learning framework to standardize and organize PyTorch code.

Model Building

- `torch.nn` contains neural network layers (e.g. Convolutional layer).
- `torch.nn.functional` contains functions used in the neural network (e.g. ReLU, Softmax).
- `torch.optim` contains optimizers (e.g. Adam, SGD) which updates model's weight based on error.
- `from torchinfo import summary` display a summary of the defined model.

Metrics & Evaluation

- `from sklearn.metrics import classification_report` is for creating classification report including precision, recall and F1 score.
- `from sklearn.metrics import ConfusionMatrixDisplay` is for counting and display a matrix of true positive (TP), false positive (FP), true negative (TN) and false negative (FN).
- `from torchmetrics.classification import Accuracy` is for calculating accuracy inside the PyTorch training loop.

Model Training Utilities

- `from tqdm.notebook import tqdm` is for creating a progress bar.
- `from pytorch_lightning.callbacks import ModelCheckpoint` is for automatically saving the model weights whenever the model improves.

2.3 GPU Utilization

GPU is preferred in deep learning task. To confirm if this notebook detects the GPU, just use the following script.

GPU Report Command Line

```
1 !nvidia-smi
```

if the GPU is detected, it should appear something like this.

```
Sun Feb 15 11:28:24 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.82.07              Driver Version: 580.82.07      CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   60C    P8             15W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Figure 1: GPU Report

3 CIFAR10 Dataset Preparation

3.1 CIFAR10 Dataset

The `CIFAR-10 dataset` (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research.

The CIFAR-10 dataset contains 60,000 32×32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

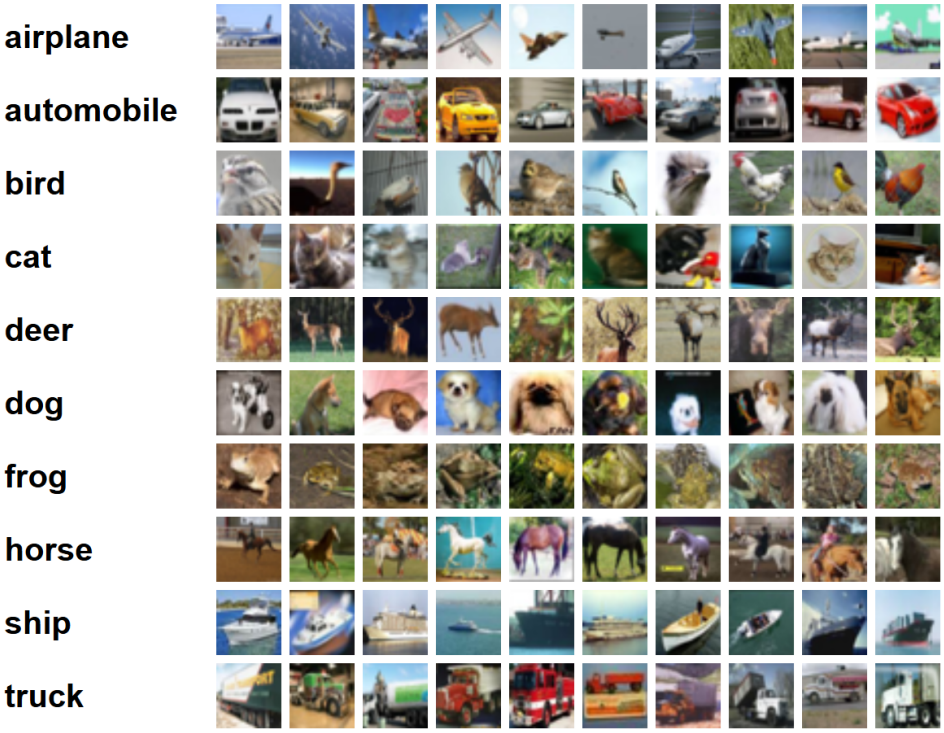

Figure 2: CIFAR10 Dataset

For more information about CIFAR10 dataset, please read: `https://www.cs.toronto.edu/~kriz/cifar.html`

3.2 Image Preprocessing Pipeline

Create image preprocessing pipeline using `transforms.Compose()` to pack every preprocessing command together.

- `transforms.ToTensor()` converts a standard PNG or JPEG image into PyTorch tensor.
- `transforms.Resize([32, 32])` forces every image to have 32×32 resolution.

Image Preprocessing Pipeline

```
IMAGE_WIDTH = 32
IMAGE_HEIGHT = 32

TRANSFORM_PIPELINE = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize([IMAGE_WIDTH, IMAGE_HEIGHT])
])
```

3.3 Load CIFAR10 Train Dataset

First, load 50,000 training images from the **CIFAR10** dataset using `torchvision.datasets`.

Load Training CIFAR10 Dataset

```
ROOT_PATH = "./data"

all_train_images = torchvision.datasets.CIFAR10(
    root=ROOT_PATH, train=True, download=True, transform=TRANSFORM_PIPELINE
)
```

Then split the 50,000 images into 2 parts: 40,000 images for training and 10,000 images for validation

Split Training and Validation Dataset

```
N_TRAIN = 40000
N_VALIDATION = 10000

train_images, validate_images = torch.utils.data.random_split(
    all_train_images, [N_TRAIN, N_VALIDATION]
)
```

3.4 Load CIFAR10 Test Dataset

Load 10,000 test images from the **CIFAR10** dataset using `torchvision.datasets`.

Load Testing CIFAR10 Dataset

```
ROOT_PATH = "./data"

test_images = torchvision.datasets.CIFAR10(
    root=ROOT_PATH, train=False, download=True, transform=TRANSFORM_PIPELINE
)
```

3.5 Create `DataLoader`

`DataLoader` is used for grabbing a batch of 32 images and feeds them to a model when training, validate or testing.

It is crucial for shuffle the image order inside a training batch to prevent model to memorize image order patterns.

Create DataLoader

```
BATCH_SIZE = 32

train_loader = torch.utils.data.DataLoader(
    train_images, batch_size=BATCH_SIZE, shuffle=True
)

validate_loader = torch.utils.data.DataLoader(
    validate_images, batch_size=BATCH_SIZE, shuffle=False
)

test_loader = torch.utils.data.DataLoader(
    test_images, batch_size=BATCH_SIZE, shuffle=False
)
```

4 CIFAR10 Dataset Visualization

This section focus on image visualization from variables from the previous processes to ensure that the dataset preparation has no problem.

4.1 Utilities

4.1.1 Display Random Image by Class

In this section, we are going to create a utility function to display random image from all classes. The processes inside the function are follows:

1. Get the unique class IDs from the labels which usually a list of numbers (e.g. $\{0, 1, \ldots, 8, 9\}$).
2. Create an empty figure for plotting.
3. Iterate each class in the dataset. For each class do as follows:
    3.1. Get the mask array indicating if an image is in the current class. (e.g. $\{\text{True}, \text{False}, \ldots, \text{False}, \text{True}\}$).
    3.2. Calculate the probability distribution for image randomization (e.g. $\{0.05, 0.00, \ldots, 0.00, 0.05\}$). For images which is not in the current class will have zero probability, so it cannot be picked.
    3.3. Randomly picking images for the current class from the image dataset by using the customized probability distribution.
    3.4. Plot picked images in the current class vertically.

Display Image by Class Function

```
def display_images_by_class(images, labels, n_display, class_names):
    """
    The utility function for displaying multiple images in a grid separated
    ↪ by classes.

    Args:
        images (np.ndarray): images dataset stored inside the NumPy array.
        labels (list[int]): label for each image in the dataset.
        n_display (int): amount of images to display for each class.
        class_names (tuple[str]): image class names.
    """
    # Find total amount of images
    n_images = images.shape[0]

    # Find unique IDs of the dataset.
    class_ids = np.unique(labels)
    n_cols = len(class_ids)

    # Create a blank figure
    plt.figure(figsize=(2 * n_cols, 2 * n_display))

    # Iterate each class (column)
    for c in range(n_cols):
```

```
        # Get a flatten boolean array which indicates whether it is the
        ↪  current class.
        is_current_class = np.squeeze(labels == class_ids[c])

        # Calculate a probability distribution of randomization.
        pick_image_probs = is_current_class / is_current_class.sum()

        # Get random images indices for the current class.
        image_indices = np.random.choice(
            n_images, n_display, replace=False, p=pick_image_probs
        )

        # Iterate each row (image) of the current class.
        for r in range(n_display):
            # Calculate the current grid index.
            grid_idx = ((r * n_cols) + c) + 1

            # Create subplot inside the figure.
            plt.subplot(n_display, n_cols, grid_idx)

            # Display the class name for the first row.
            if r == 0:
                plt.title(class_names[c])

            # Display the current image inside the current grid.
            plt.axis("off")
            plt.imshow(images[image_indices[r]], aspect="equal")
```

4.1.2 Display Image from PyTorch

The problem on image visualization from PyTorch tensor are as follows:

- **PyTorch** stores image as: `(channels, height, width)`.
- **Matplotlib** stores image as: `(height, width, channels)`

That's why we need the utility function to rearrange the image array for Matplotlib to visualize correctly.

Display Image from PyTorch

```
def display_image_from_pt(image_pt, plot_title):
    """
    The utility function to display image from PyTorch tensor.

    Args:
        image_pt (torch.Tensor): image tensor from PyTorch
        plot_title (str): title to display on the plot
    """
    # Convert PyTorch tensor into NumPy array.
    image_np = image_pt.numpy()

    # Rearrange order of the array.
    image_np = np.transpose(image_np, (1, 2, 0))

    # Display an image
    plt.figure(figsize=(16, 8))
    plt.title(plot_title)
    plt.imshow(image_np)
    plt.axis("off")
    plt.show()
```

4.2 Class Visualization

In this section, we are going to display random images from each class in **CIFAR10** dataset to ensure the images inside `train_images` are not corrupted.

Class Visualization

```
CLASS_NAMES = (
    "plane", "car", "bird", "cat", "deer",
    "dog", "frog", "horse", "ship", "truck"
)

display_images_by_class(
    images=train_images.dataset.data,
    labels=train_images.dataset.targets,
    n_display=3,
    class_names=CLASS_NAMES,
)
```

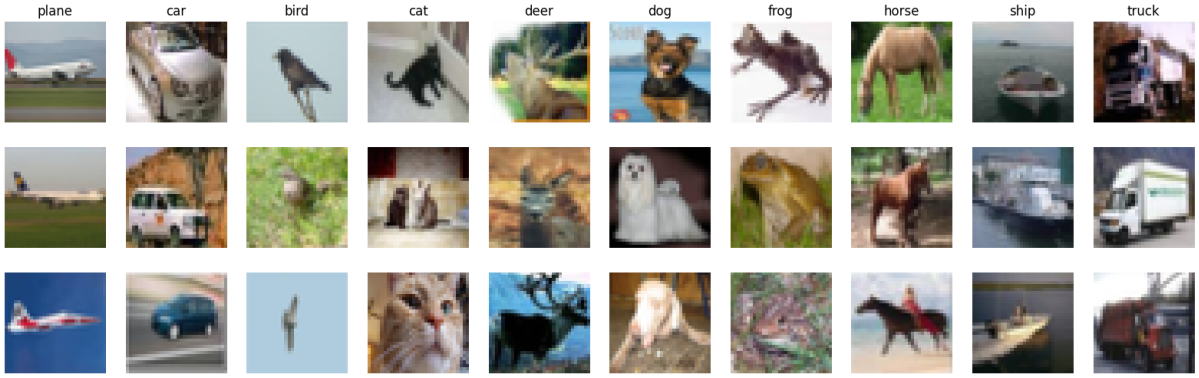

Figure 3: Class Visualization

4.3 Batch Visualization

In this section, we are going to display images from a batch picked from the `DataLoader` to ensure the `DataLoader` are working properly.

Batch Visualization

```python
# Configuration
IMAGES_PER_ROW = 8
PLOT_TITLE = "Batch of 32 training images"

# Get a batch of images.
images, labels = next(iter(train_loader))

# Create a grid of images inside a batch.
batch_images = torchvision.utils.make_grid(images, nrow=IMAGES_PER_ROW)

# Display the batch of images.
display_image_from_pt(image_pt=batch_images, plot_title=PLOT_TITLE)
```

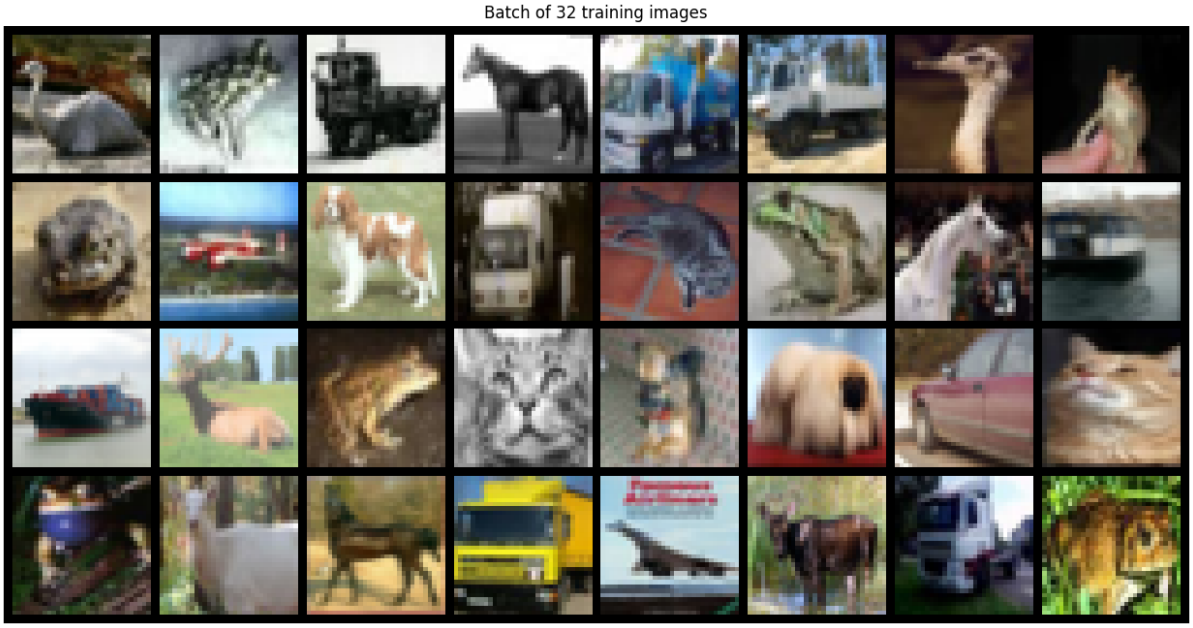

Figure 4: Batch Visualization

To visualize label of each image inside a batch, use this following code:

Batch Labels

```
for image_idx in range(0, BATCH_SIZE, IMAGES_PER_ROW):
    for label in labels[image_idx: image_idx + IMAGES_PER_ROW]:
        print(CLASS_NAMES[label], end="\t")
    print()
```

bird	frog	truck	horse	truck	truck	bird	frog
frog	plane	dog	truck	cat	frog	horse	ship
ship	deer	frog	cat	dog	dog	car	cat
frog	deer	horse	truck	plane	deer	truck	frog

Table 1: Batch Labels

5 CNN Model

5.1 Model Layers

We are design a model similar to LeNet-5 Convolutional Neural Network (CNN) model architecture which consists of:

1. **Convolutional Layer** using 5×5 kernel with 3 input channels and 6 output channels. Then, use ReLU as an activation function.
2. **Max Pooling Layer** using 2×2 kernel and set stride to 2.
3. **Convolutional Layer** using 5×5 kernel with 6 input channels and 15 output channels. Then, use ReLU as an activation function.
4. **Max Pooling Layer** using 2×2 kernel and set stride to 2.
5. **Fully Connected Layer** using with 400 inputs and 120 outputs. Then, use ReLU as an activation function.
6. **Fully Connected Layer** using with 120 inputs and 84 outputs. Then, use ReLU as an activation function.
7. **Fully Connected Layer** using with 84 inputs and 10 outputs.
8. **Output Layer** using softmax function for output the classification result.

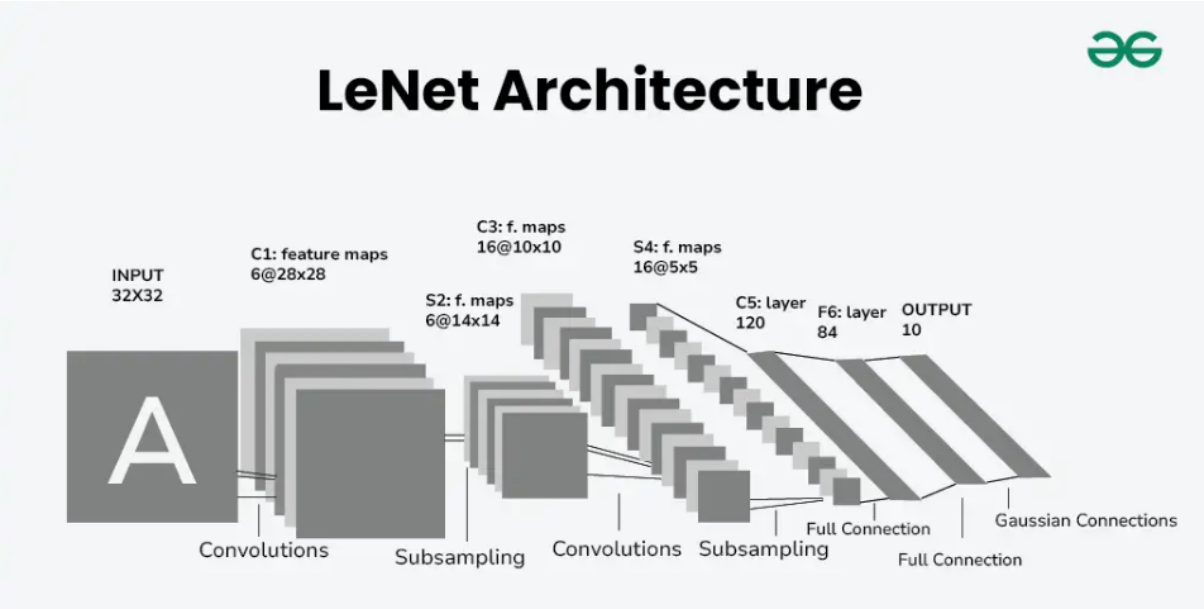

Figure 5: LeNet-5 CNN Architecture

Define the model layers inside the constructor method of the class.

Define Model Layers

```
class CNN(pl.LightningModule):
    def __init__(self):
        super().__init__()

        # The first convolutional layer
        self.conv1 = nn.Conv2d(3, 6, 5)

        # Max pooling layer for downsampling
        self.pool = nn.MaxPool2d(2, 2)

        # The second convolutional layer
        self.conv2 = nn.Conv2d(6, 16, 5)

        # Fully connected layers
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

        # Softmax layer for classification result.
        self.softmax = torch.nn.Softmax(dim=1)
```

Then write the training process of the model in `forward(self, x)` method.

Model Training Processes

```
class CNN(pl.LightningModule):
    def forward(self, x):
        # Feature extraction using convolutional layers with ReLU.
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))

        # Flatten the feature vector for fully connected layers.
        x = torch.flatten(x, start_dim=1)

        # Classification process by fully connected layers with ReLU.
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        # Predict classification result using softmax layer.
        x = self.softmax(x)
        return x
```

5.2 Loss Function & Optimizers

We are using Cross Entropy Loss as a loss function for this model, and use Adam optimizer with 0.001 learning rate.

Model Loss Function and Optimizers

```python
class CNN(pl.LightningModule):
    def __init__(self):
        # ...MODEL LAYERS...

        # Define loss function for a model
        self.criterion = nn.CrossEntropyLoss()

        # Define accuracy metrics
        self.accuracy = Accuracy(task="multiclass", num_classes=10)

    def configure_optimizers(self):
        # Use Adam optimizer with 0.001 learning rate.
        return optim.Adam(self.parameters(), lr=1e-3)
```

5.3 Model Summary

To read the overall description of a model, just use this `summary` command and set `input_size` as parameter which is a tuple in the formal `(batch_size, channel, width, height)`.

Model Summary

```python
net = CNN().to(device)
print(summary(net, input_size=(32, 3, 32, 32)))
```

It should appear something similar to this.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
CNN                                      [32, 10]                  --
├─Conv2d: 1-1                            [32, 6, 28, 28]           456
├─MaxPool2d: 1-2                         [32, 6, 14, 14]           --
├─Conv2d: 1-3                            [32, 16, 10, 10]          2,416
├─MaxPool2d: 1-4                         [32, 16, 5, 5]            --
├─Linear: 1-5                            [32, 120]                 48,120
├─Linear: 1-6                            [32, 84]                  10,164
├─Linear: 1-7                            [32, 10]                  850
├─Softmax: 1-8                           [32, 10]                  --
==========================================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 21.06
==========================================================================================
Input size (MB): 0.39
Forward/backward pass size (MB): 1.67
Params size (MB): 0.25
Estimated Total Size (MB): 2.31
==========================================================================================
```

Figure 6: Model Summary

6 Model Training

6.1 Model Checkpoint

In deep learning, the best model is not always the model in the very last epoch so we should have checkpoint feature.

To make a checkpoint feature which saves the model that has maximum validation accuracy, you need to use `ModelCheckpoint()` with following configurations:

- Set `monitor` to `val_acc` to monitor validation accuracy of the model in each epoch.
- Set `mode` to `max` to save a model which has the maximum validation accuracy.
- Set `save_top_k` to `1` to save only the best model.
- `filename` is the model's filename format.
- Set `verbose` to `True` to display the message when saving a model.

Model Checkpoint

```
checkpoint_callback = ModelCheckpoint(
    monitor="val_acc",
    mode="max",
    save_top_k=1,
    filename="best-acc-{epoch:02d}-{val_acc:.4f}",
    verbose=True,
)
```

6.2 Model Training

PyTorch Lightning framework has `Trainer` object which handles the training loop for you by following the defined method in `CNN` class.

- `max_epoch` is the maximum epochs of the model.
- `callbacks` is the model checkpoint object.

After initialize the `Trainer` object, use `fit()` command to start training a model.

- `model` is the initialized model.
- `train_dataloader` is the `DataLoader` for training dataset.
- `val_dataloader` is the `DataLoader` for validation dataset.

Model Training

```
trainer = pl.Trainer(max_epochs=10, callbacks=[checkpoint_callback])
trainer.fit(net, trainloader, valloader)
```

6.3 Training Metrics

During model training process, the loss and accuracy can be plotted as follows:

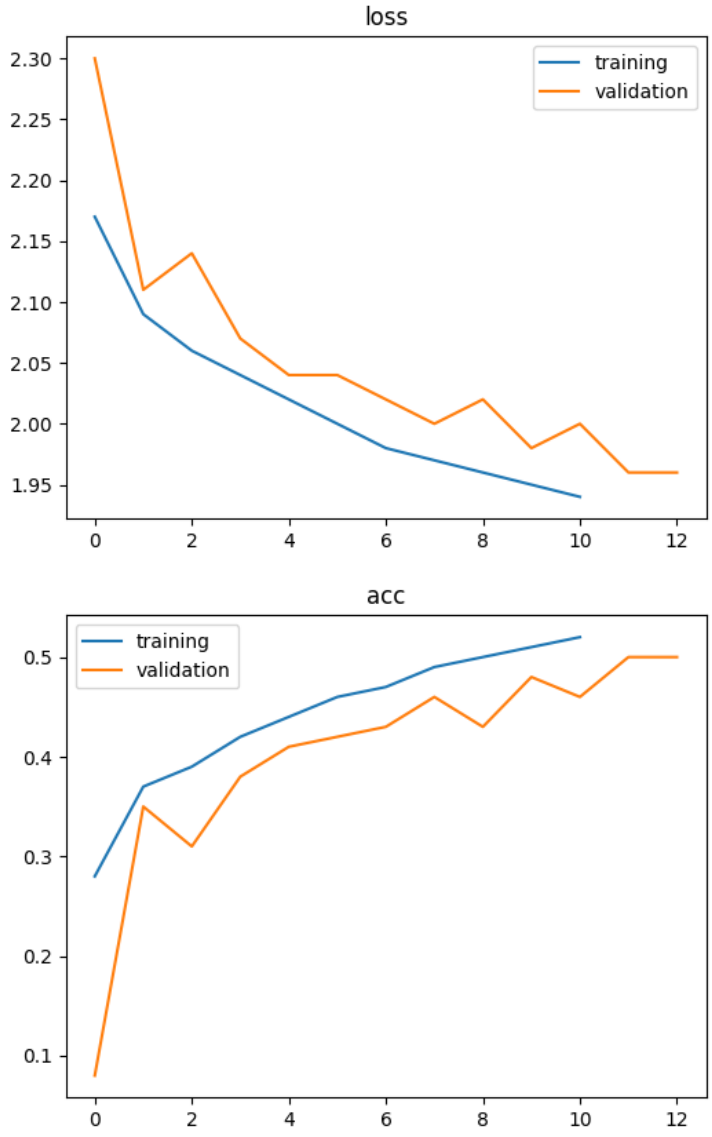

Figure 7: Training Metrics

7 Model Evaluation

After done with training and validation the model, we will process with evaluating the model using the test dataset and has the following result:

Confusion matrix from comparing the predictions with ground truth.

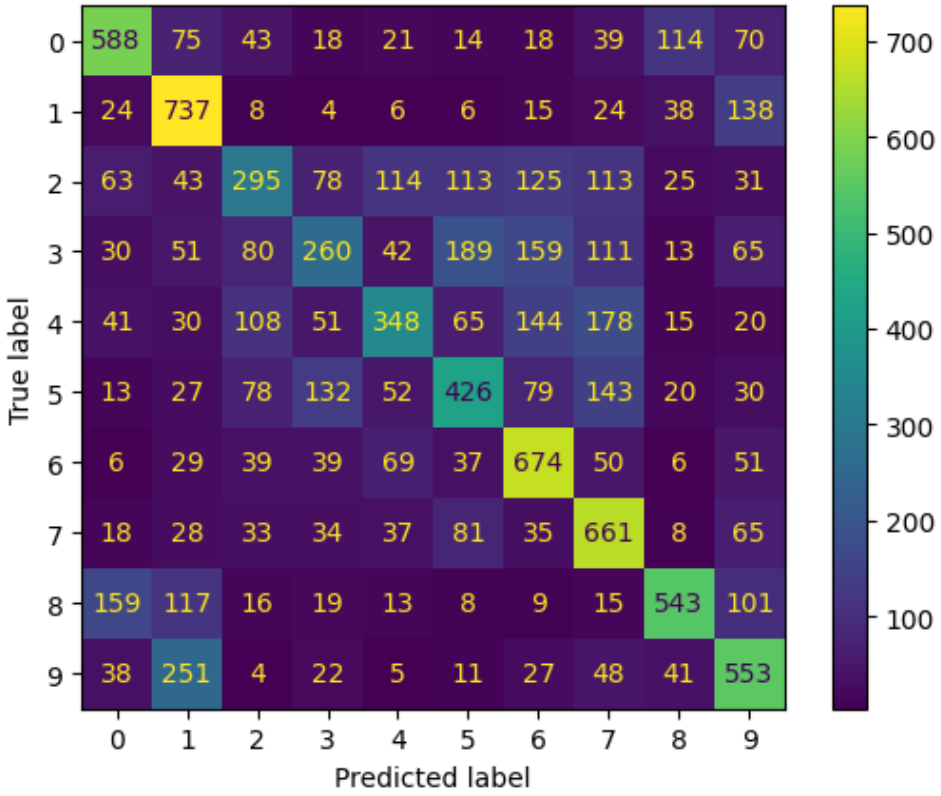

Figure 8: Confusion Matrix

Precision, recall and F1 score metrics result separated by class.

Class	Precision	Recall	F1 Score
plane	0.6000	0.5880	0.5939
car	0.5310	0.7370	0.6173
bird	0.4190	0.2950	0.3462
cat	0.3957	0.2600	0.3138
deer	0.4922	0.3480	0.4077
dog	0.4484	0.4260	0.4369
frog	0.5245	0.6740	0.5899
horse	0.4783	0.6610	0.5550
ship	0.6598	0.5430	0.5957
truck	0.4920	0.5530	0.5207

Table 2: Per-class Evaluation Result

Overall precision, recall and F1 score metrics result.

Metrics	Value
Accuracy	0.5085
Macro F1-score	0.4977
Weighted F1-score	0.4977

Table 3: Overall Evaluation Result

Finally, display a row of 5 images. Above each one, display the model confidence about the prediction.

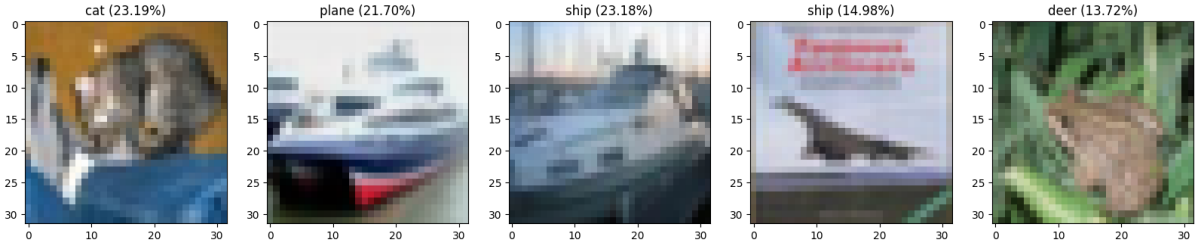

Figure 9: Model Confidence

Part III
Image classification with EfficientNetV2

8 Dataset Preparation

8.1 Image Preprocessing Pipeline

Create image preprocessing pipeline using `transforms.Compose()` to pack every preprocessing command together.

First, create the image preprocessing pipeline for training image set.

- `transforms.Resize([230, 230])` forces every image to have 230×230 resolution.
- `transforms.RandomRotation(30)` randomly rotates the image by any angle between $-30°$ and $30°$.
- `transforms.RandomCrop(224)` cuts out a random 224×224 square from the 230×230 image.
- `transforms.RandomHorizontalFlip()` randomly flips the image left-to-right with 50% probability.
- `transforms.RandomVerticalFlip()` randomly flips the image upside-down with 50% probability.
- `transforms.ToTensor()` converts an image to PyTorch tensor.
- `transforms.Normalize()` normalize the value inside the PyTorch tensors.

Next, create the image preprocessing pipeline for validation and testing image set.

- `transforms.Resize([230, 230])` forces every image to have 230×230 resolution.
- `transforms.ToTensor()` converts an image to PyTorch tensor.
- `transforms.Normalize()` normalize the value inside the PyTorch tensors.

8.2 Create `DataLoader`

In this section, we are going to create the `AnimalDataset` class to create image dataset and `DataLoader`.

Since the dataset is not available in the library like **CIFAR10** dataset in the previous section, we need to grab each image and label and create the image.

Create Dataset

```
trainset = AnimalDataset("./Dataset_animal2/train", transform_train)
valset = AnimalDataset("./Dataset_animal2/val", transform)
testset = AnimalDataset("./Dataset_animal2/test", transform)
```

Then, create the `DataLoader` from the dataset and set `batch_size` to 32.

Create DataLoader

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
    shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
    shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
    shuffle=True)
```

Now we have the dataset and the `DataLoader` for model training and evaluation.

8.3 Dataset Visualization

8.3.1 Class Visualization

In this section, we are going to display random images from each class in the dataset.

Figure 10: Class Visualization

8.3.2 Batch Visualization

In this section, we are going to display images from a batch picked from the `DataLoader` to ensure the `DataLoader` are working properly.

Figure 11: Batch Visualization

9 EfficientNetV2 Model

9.1 Model Creation

In this notebook, we are just download the **EfficientNetV2** pre-trained model which is lot easier than the previous section. However, we need to modify the output layer to classify our 10 animals.

Model Architecture

```
def __init__(self, num_classes=10, learning_rate=1e-3):
    super().__init__()
    # Load EfficientNetV2 model from torchvision
    self.model = models.efficientnet_v2_s(weights="IMAGENET1K_V1")

    # Replace the classifier with a custom layer for our task
    self.model.classifier[1] = nn.Linear(
        self.model.classifier[1].in_features, num_classes
    )
```

Next, set loss function to cross-entropy loss function, set evaluation metrics to accuracy and set learning rate to 0.003.

Loss Function & Evaluation Metrics

```
def __init__(self, num_classes=10, learning_rate=1e-3):
    # ...MODEL LAYERS...

    self.criterion = nn.CrossEntropyLoss()
    self.accuracy = Accuracy(task="multiclass", num_classes=num_classes)
    self.learning_rate = learning_rate
```

Finally, set the optimizer to Adam optimizer with 0.001 learning rate.

Loss Function & Evaluation Metrics

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
    return optimizer
```

9.2 Model Summary

To read the overall description of a model, just use this `summary` command and set `input_size` as parameter which is a tuple in the formal `(batch_size, channel, width, height)`.

Model Summary

```
net = LitEfficientNetV2().to(device)
print(summary(net, input_size=(32, 3, 224, 224)))
```

It should appear something similar to this.

```
==========================================================================================================================
Layer (type:depth-idx)                                                  Output Shape              Param #
==========================================================================================================================
LitEfficientNetV2                                                       [32, 10]                  --
├─EfficientNet: 1-1                                                     [32, 10]                  --
│    └─Sequential: 2-1                                                  [32, 1280, 1, 1]          --
│    │    └─Conv2dNormActivation: 3-1                                   [32, 24, 16, 16]          696
│    │    └─Sequential: 3-2                                             [32, 24, 16, 16]          10,464
│    │    └─Sequential: 3-3                                             [32, 48, 8, 8]            303,552
│    │    └─Sequential: 3-4                                             [32, 64, 4, 4]            589,184
│    │    └─Sequential: 3-5                                             [32, 128, 2, 2]           917,680
│    │    └─Sequential: 3-6                                             [32, 160, 2, 2]           3,463,840
│    │    └─Sequential: 3-7                                             [32, 256, 1, 1]           14,561,832
│    │    └─Conv2dNormActivation: 3-8                                   [32, 1280, 1, 1]          330,240
│    └─AdaptiveAvgPool2d: 2-2                                           [32, 1280, 1, 1]          --
│    └─Sequential: 2-3                                                  [32, 10]                  --
│    │    └─Dropout: 3-9                                                [32, 1280]                --
│    │    └─Linear: 3-10                                                [32, 10]                  12,810
==========================================================================================================================
Total params: 20,190,298
Trainable params: 20,190,298
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 1.98
==========================================================================================================================
Input size (MB): 0.39
Forward/backward pass size (MB): 136.05
Params size (MB): 80.76
Estimated Total Size (MB): 217.21
==========================================================================================================================
```

Figure 12: Model Summary

10 Model Training

10.1 Model Checkpoint

In deep learning, the best model is not always the model in the very last epoch so we should have checkpoint feature.

To make a checkpoint feature which saves the model that has maximum validation accuracy, you need to use `ModelCheckpoint()` with following configurations:

- Set `monitor` to `val_acc` to monitor validation accuracy of the model in each epoch.
- Set `mode` to `max` to save a model which has the maximum validation accuracy.
- Set `save_top_k` to `1` to save only the best model.
- `filename` is the model's filename format.
- Set `verbose` to `True` to display the message when saving a model.

Model Checkpoint

```
checkpoint_callback = ModelCheckpoint(
    monitor="val_acc",
    mode="max",
    save_top_k=1,
    filename="efficientnetv2-best-{epoch:02d}-{val_acc:.4f}",
    verbose=True,
)
```

10.2 Model Training

PyTorch Lightning framework has `Trainer` object which handles the training loop for you by following the defined method in `EfficientNetV2` class.

- `max_epoch` is the maximum epochs of the model.
- `callbacks` is the model checkpoint object.

After initialize the `Trainer` object, use `fit()` command to start training a model.

- `model` is the initialized model.
- `train_dataloader` is the `DataLoader` for training dataset.
- `val_dataloader` is the `DataLoader` for validation dataset.

Model Training

```
trainer = pl.Trainer(max_epochs=5, callbacks=[checkpoint_callback])
trainer.fit(net, trainloader, valloader)
```

10.3 Training Metrics

During model training process, the loss and accuracy can be plotted as follows:

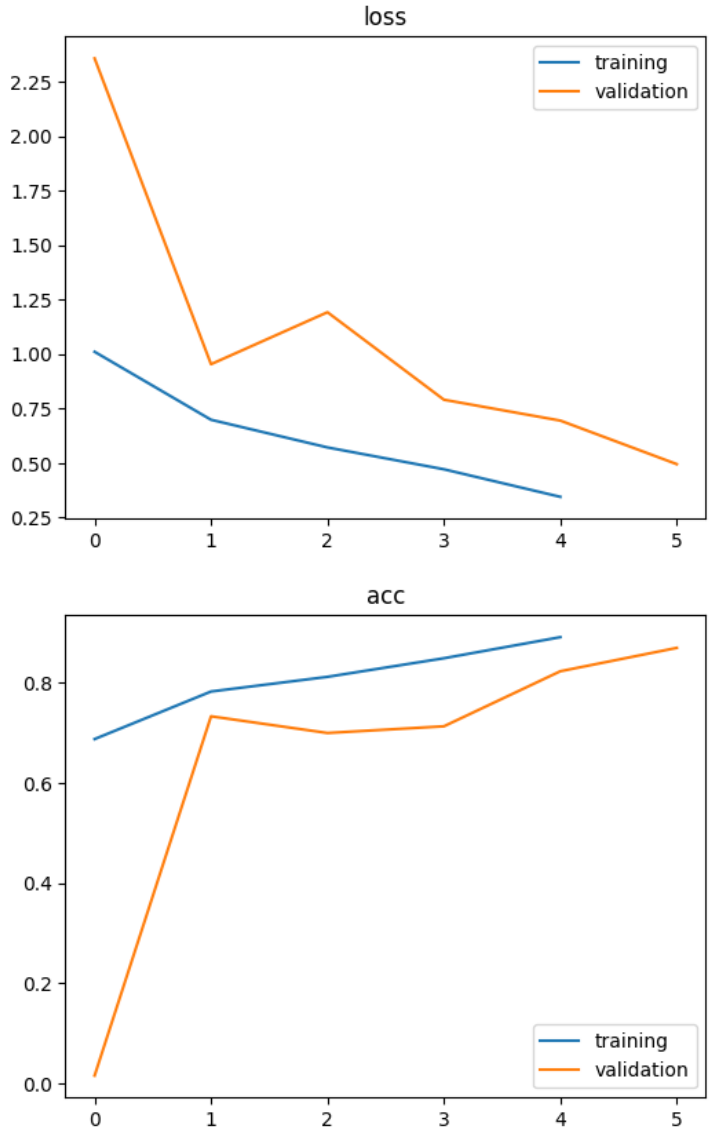

Figure 13: Training Metrics

11 Model Evaluation

After done with training and validation the model, we will process with evaluating the model using the test dataset and has the following result:

Confusion matrix from comparing the predictions with ground truth.

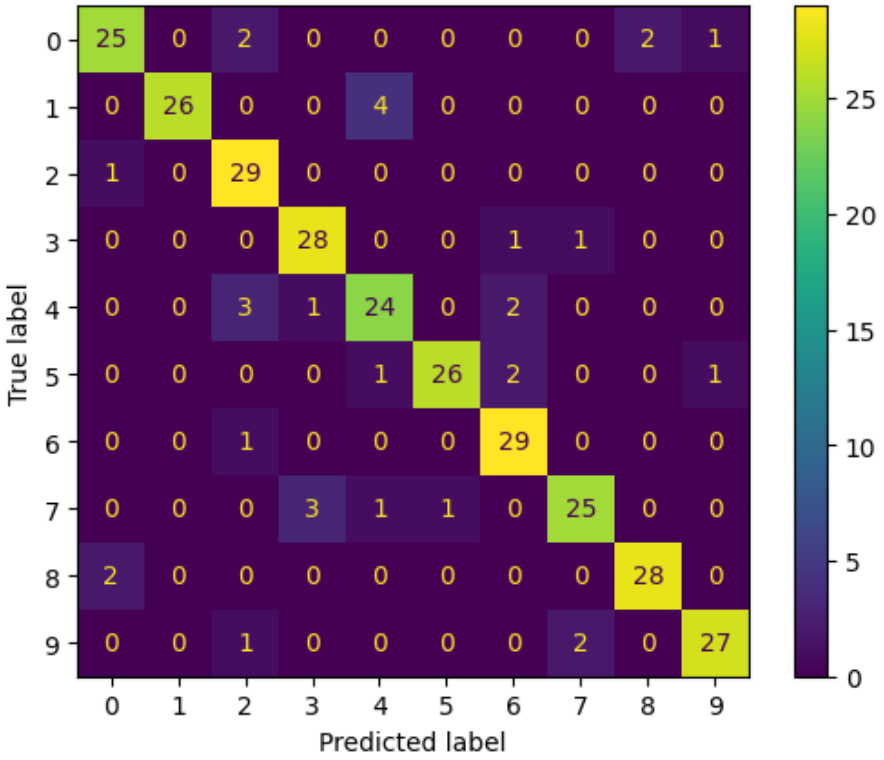

Figure 14: Confusion Matrix

Precision, recall and F1 score metrics result separated by class.

Class	Precision	Recall	F1 Score
butterfly	0.8929	0.8333	0.8621
cat	1.0000	0.8667	0.9286
chicken	0.8056	0.9667	0.8788
cow	0.8750	0.9333	0.9032
dog	0.8000	0.8000	0.8000
elephant	0.9630	0.8667	0.9123
horse	0.8529	0.9667	0.9062
sheep	0.8929	0.8333	0.8621
spider	0.9333	0.9333	0.9333
squirrel	0.9310	0.9000	0.9153

Table 4: Per-class Evaluation Result

Overall precision, recall and F1 score metrics result.

Metrics	Value
Accuracy	0.8900
Macro F1-score	0.8902
Weighted F1-score	0.8902

Table 5: Overall Evaluation Result

Finally, display a row of 5 images. Above each one, display the model confidence about the prediction.

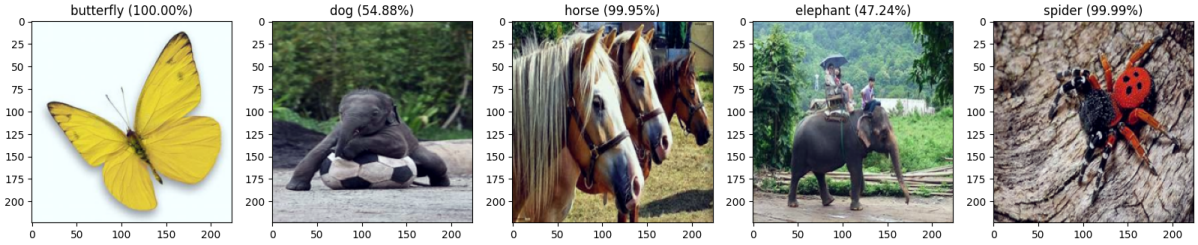

Figure 15: Model Confidence

Part IV
Image classification with EfficientNetb0

12 Dataset Preparation

12.1 Image Preprocessing Pipeline

Create image preprocessing pipeline using `transforms.Compose()` to pack every preprocessing command together.

First, create the image preprocessing pipeline for training image set.

- `transforms.Resize([230, 230])` forces every image to have 230×230 resolution.
- `transforms.RandomRotation(30)` randomly rotates the image by any angle between $-30°$ and $30°$.
- `transforms.RandomCrop(224)` cuts out a random 224×224 square from the 230×230 image.
- `transforms.RandomHorizontalFlip()` randomly flips the image left-to-right with 50% probability.
- `transforms.RandomVerticalFlip()` randomly flips the image upside-down with 50% probability.
- `transforms.ToTensor()` converts an image to PyTorch tensor.
- `transforms.Normalize()` normalize the value inside the PyTorch tensors.

Next, create the image preprocessing pipeline for validation and testing image set.

- `transforms.Resize([230, 230])` forces every image to have 230×230 resolution.
- `transforms.ToTensor()` converts an image to PyTorch tensor.
- `transforms.Normalize()` normalize the value inside the PyTorch tensors.

12.2 Create `DataLoader`

In this section, we are going to create the `AnimalDataset` class to create image dataset and `DataLoader`.

Since the dataset is not available in the library like **CIFAR10** dataset in the previous section, we need to grab each image and label and create the image.

Create Dataset

```
trainset = AnimalDataset("./Dataset_animal2/train", transform_train)
valset = AnimalDataset("./Dataset_animal2/val", transform)
testset = AnimalDataset("./Dataset_animal2/test", transform)
```

Then, create the `DataLoader` from the dataset and set `batch_size` to 32.

Create DataLoader

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
    shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
    shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
    shuffle=True)
```

Now we have the dataset and the `DataLoader` for model training and evaluation.

12.3 Dataset Visualization

12.3.1 Class Visualization

In this section, we are going to display random images from each class in the dataset.

Figure 16: Class Visualization

12.3.2 Batch Visualization

In this section, we are going to display images from a batch picked from the `DataLoader` to ensure the `DataLoader` are working properly.

Figure 17: Batch Visualization

13 EfficientNetb0 Model

13.1 Model Creation

In this notebook, we are just download the **EfficientNetb0** pre-trained model which is lot easier than the previous section. However, we need to modify the output layer to classify our 10 animals.

Model Architecture

```
def __init__(self, num_classes=10, learning_rate=1e-3):
    super().__init__()

    # Load EfficientNetb0 model from hugging face
    self.model = EfficientNetForImageClassification.from_pretrained(
        "google/efficientnet-b0",
        num_labels=num_classes,
        ignore_mismatched_sizes=True,
    )
```

Next, set loss function to cross-entropy loss function, set evaluation metrics to accuracy and set learning rate to 0.003.

Loss Function & Evaluation Metrics

```
def __init__(self, num_classes=10, learning_rate=1e-3):
    # ...MODEL LAYERS...

    self.criterion = nn.CrossEntropyLoss()
    self.accuracy = Accuracy(task="multiclass", num_classes=num_classes)
    self.learning_rate = learning_rate
```

Finally, set the optimizer to Adam optimizer with 0.001 learning rate.

Loss Function & Evaluation Metrics

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
    return optimizer
```

13.2 Model Summary

To read the overall description of a model, just use this `summary` command and set `input_size` as parameter which is a tuple in the formal `(batch_size, channel, width, height)`.

Model Summary

```
net = LitEfficientNetV2().to(device)
print(summary(net, input_size=(32, 3, 224, 224)))
```

It should appear something similar to this.

```
==========================================================================================================================
Layer (type:depth-idx)                                                  Output Shape              Param #
==========================================================================================================================
LitEfficientNetV2                                                       [32, 10]                  --
├─EfficientNetForImageClassification: 1-1                               [32, 10]                  --
|    └─EfficientNetModel: 2-1                                           [32, 1280]                --
|    |    └─EfficientNetEmbeddings: 3-1                                 [32, 32, 112, 112]        928
|    |    └─EfficientNetEncoder: 3-2                                    [32, 1280, 7, 7]          4,006,620
|    |    └─AvgPool2d: 3-3                                              [32, 1280, 1, 1]          --
|    └─Dropout: 2-2                                                     [32, 1280]                --
|    └─Linear: 2-3                                                      [32, 10]                  12,810
==========================================================================================================================
Total params: 4,020,358
Trainable params: 4,020,358
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 12.31
==========================================================================================================================
Input size (MB): 19.27
Forward/backward pass size (MB): 3452.09
Params size (MB): 16.08
Estimated Total Size (MB): 3487.44
==========================================================================================================================
```

Figure 18: Model Summary

14 Model Training

14.1 Model Checkpoint

In deep learning, the best model is not always the model in the very last epoch so we should have checkpoint feature.

To make a checkpoint feature which saves the model that has maximum validation accuracy, you need to use `ModelCheckpoint()` with following configurations:

- Set `monitor` to `val_acc` to monitor validation accuracy of the model in each epoch.
- Set `mode` to `max` to save a model which has the maximum validation accuracy.
- Set `save_top_k` to `1` to save only the best model.
- `filename` is the model's filename format.
- Set `verbose` to `True` to display the message when saving a model.

Model Checkpoint

```
checkpoint_callback = ModelCheckpoint(
    monitor="val_acc",
    mode="max",
    save_top_k=1,
    filename="efficientnetv2-best-{epoch:02d}-{val_acc:.4f}",
    verbose=True,
)
```

14.2 Model Training

PyTorch Lightning framework has `Trainer` object which handles the training loop for you by following the defined method in `EfficientNetV2` class.

- `max_epoch` is the maximum epochs of the model.
- `callbacks` is the model checkpoint object.

After initialize the `Trainer` object, use `fit()` command to start training a model.

- `model` is the initialized model.
- `train_dataloader` is the `DataLoader` for training dataset.
- `val_dataloader` is the `DataLoader` for validation dataset.

Model Training

```
logger = WandbLogger(
    save_dir=".", name="lightning_logs", version=None
)
trainer = pl.Trainer(logger=logger, max_epochs=5,
  ↪  callbacks=[checkpoint_callback])
trainer.fit(net, trainloader, valloader)
```

14.3 Training Metrics

During model training process, the loss and accuracy can be plotted as follows:

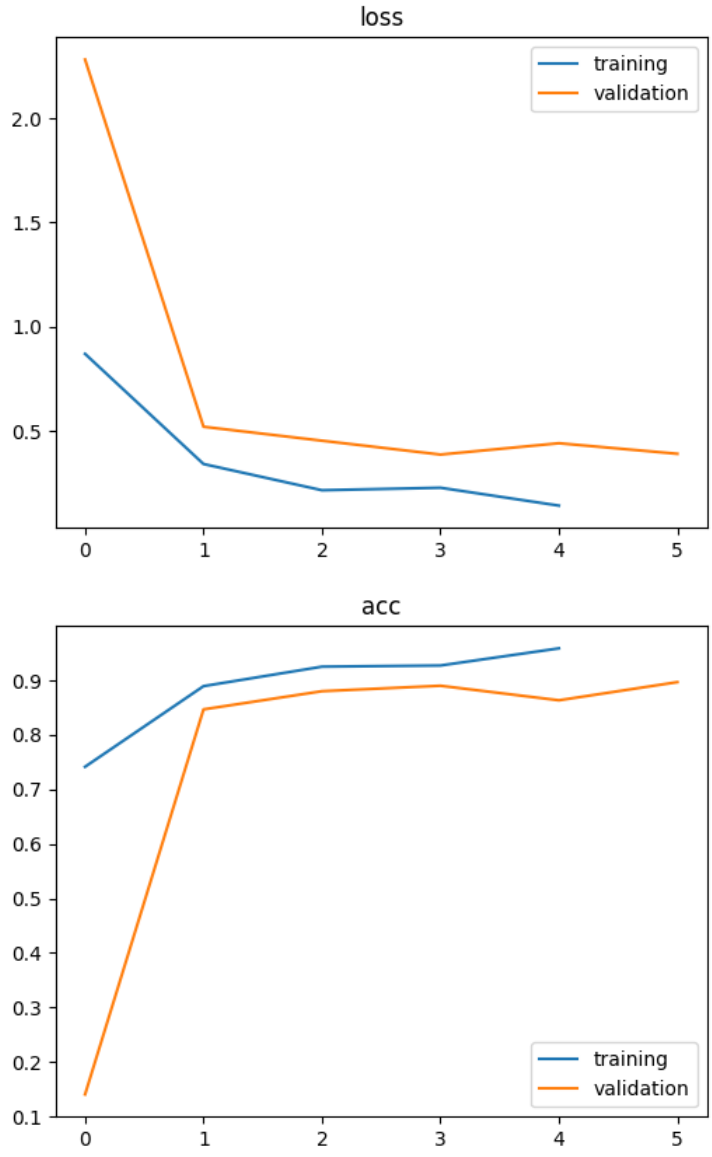

Figure 19: Training Metrics

15 Model Evaluation

After done with training and validation the model, we will process with evaluating the model using the test dataset and has the following result:

Confusion matrix from comparing the predictions with ground truth.

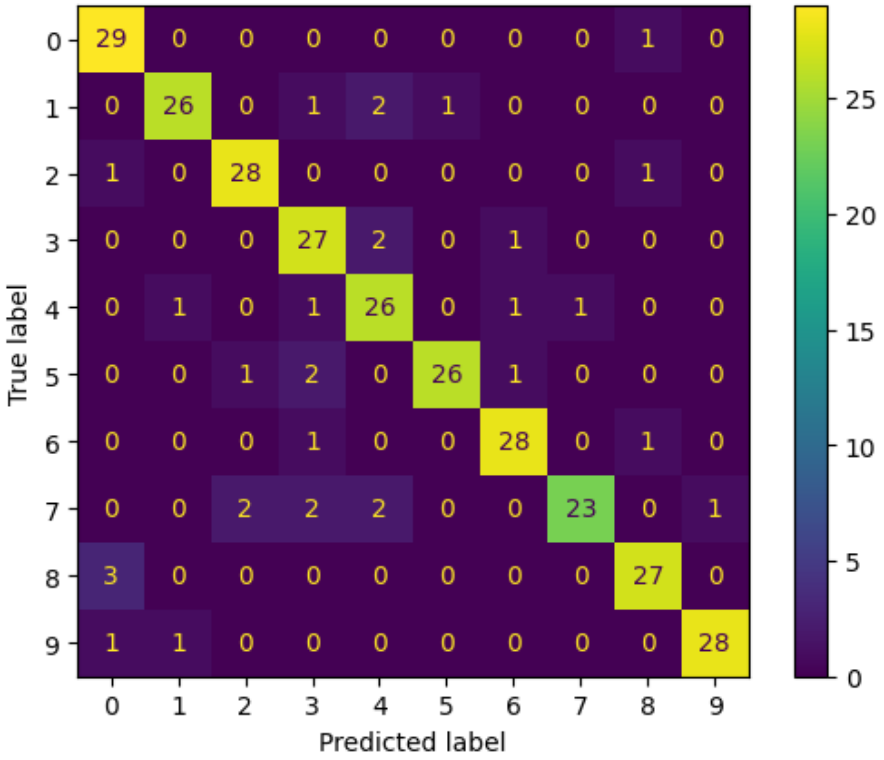

Figure 20: Confusion Matrix

Precision, recall and F1 score metrics result separated by class.

Class	Precision	Recall	F1 Score
butterfly	0.8529	0.9667	0.9062
cat	0.9286	0.8667	0.8966
chicken	0.9333	0.9667	0.9180
cow	0.7941	0.9000	0.8438
dog	0.8125	0.8667	0.8387
elephant	0.9630	0.8667	0.9123
horse	0.9032	0.9333	0.9180
sheep	0.9583	0.7667	0.8519
spider	0.9000	0.9000	0.9000
squirrel	0.9655	0.9333	0.9492

Table 6: Per-class Evaluation Result

Overall precision, recall and F1 score metrics result.

Metrics	Value
Accuracy	0.8933
Macro F1-score	0.8935
Weighted F1-score	0.8935

Table 7: Overall Evaluation Result

Finally, display a row of 5 images. Above each one, display the model confidence about the prediction.

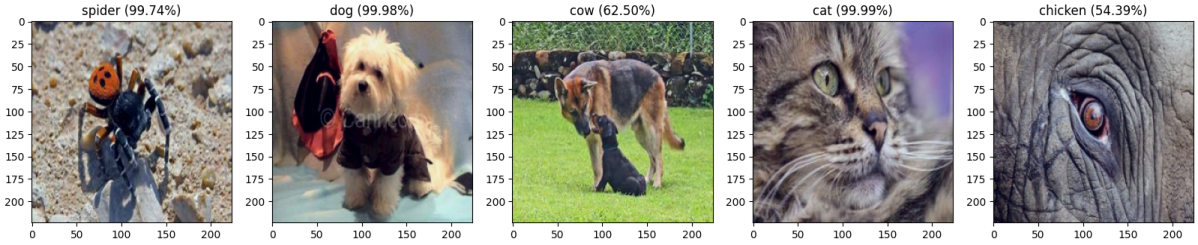

Figure 21: Model Confidence

Part V
LangChain API

This notebook shows the **LangChain** capabilities which allowing you to switch between different AI providers.

In this notebook demostrates the usage of AI from 4 providers as follows:

1. **OpenAI**: Use `gpt-5-mini` model from `langchain-openai` library.
2. **Google Gemini**: Use `gemini-2.5 flash` model from `langchain_google_genai` library.
3. **Groq**: Use `llama-3.1-8b-instant` model from `langchain-groq` library.
4. **NVIDIA**: Use a model from `langchain_nvidia_ai_endpoints` library.

To communicate with the model, we need an API key of the model which can be found at:

1. **OpenAI**: `https://platform.openai.com/account/api-keys`
2. **Google Gemini**: `https://aistudio.google.com/app/apikey`
3. **Groq**: `https://console.groq.com/keys`
4. **NVIDIA**: `https://build.nvidia.com/settings/api-keys`