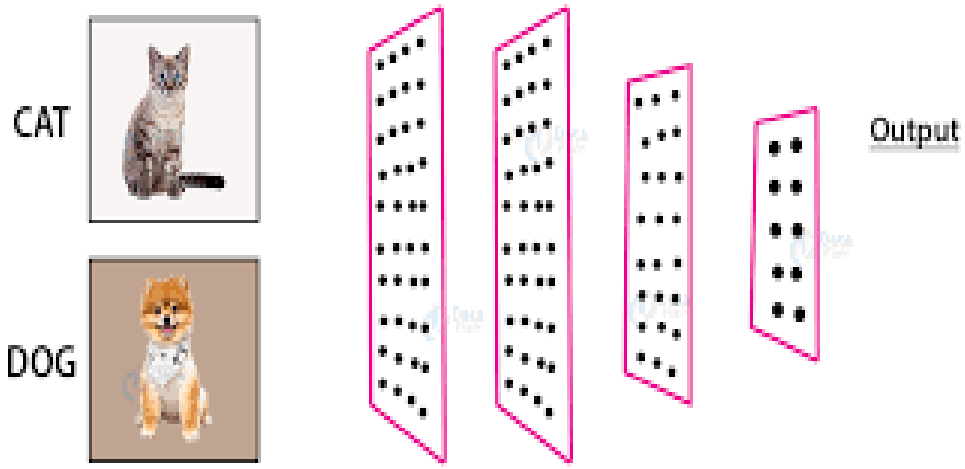



# FINAL PROJECT

## IMAGE CLASSIFICATION OF CATS AND DOGS

**Group Members:**

Kousar Kousar (22301044)
Aqsa Shabbir (22301043)
Yasir Ali (22301049)

Date
January 05, 2023

# Abstract:

This project aims to implement and compare multiple neural network architectures for the classification of cats and dogs in images. The primary objective involves exploring the effectiveness of a custom-designed neural network against established architectures like VGG and ResNet. The models are trained and evaluated using a dataset consisting of labeled images of cats and dogs. The report discusses the methodologies applied, dataset specifics, model performances, and comparative analyses among the different architectures.

**Results:**

***Proposed Neural Network:*** Achieved an accuracy of 66.15% and a loss of 0.60.

***VGG19:*** Attained an accuracy of 75.01% and a loss of 0.53.

***ResNet***: Garnered an accuracy of 88.56% and a loss of 0.17.

The comparative analysis reveals insights into the relative strengths and weaknesses of each architecture, highlighting ResNet's superior performance in classifying images of cats and dogs. This project is based on classifying images of cats and dogs by employing three distinct convolutional neural network architectures: one is our proposed model, a simplified VGG model, and a ResNet model. Utilizing the provided dataset of class and thoughtful model design, we aim to achieve accurate categorization and distinguishing between these two common classes in neural networks. The primary objective is to develop robust models capable of high-precision classification. The methodology involves training both the VGG and ResNet models on a combined dataset of cat and dog images and evaluating their performance on separate validation and test sets.

# 1- Introduction:

In today's digital era, image classification plays a pivotal role across various industries, from healthcare to security and entertainment. The ability to accurately differentiate between different animals, such as cats and dogs, holds significant practical importance. Deep learning, particularly convolutional neural networks (CNNs), has emerged as a powerful tool for such tasks due to its ability to automatically learn intricate patterns and features from images.

This project specifically focuses on the classification of cats and dogs utilizing three distinct neural network architectures:

## 1.1 Proposed Neural Network:

The proposed model (Classifier), is a neural network architecture designed for binary image classification, specifically distinguishing between images of cats and non-cat images. It employs a basic fully connected neural network with two hidden layers and ReLU activation, followed by a sigmoid output layer.

## 1.2 VGG (Visual Geometry Group):

VGG is a widely recognized deep learning architecture known for its simplicity and depth. It consists of 16 or 19 layers, with smaller-sized convolutional filters (3x3) stacked one after another. In this project, we have employed VGG with 19 layers.

The architecture follows a straightforward sequential pattern, where convolutional layers are arranged in a cascade, with pooling layers scattered. The Convolutional layers then aim to learn patterns at varying levels of abstraction through successive convolutions.

Similar to the proposed model, VGG concludes with fully connected layers for classification.

## 1.3 ResNet (Residual Network):

ResNet introduced a groundbreaking concept of residual connections, addressing the vanishing gradient problem in deep networks. Its notable features include residual blocks, identifying maps and Deep neural architecture.

The residual blocks contain skip connections or shortcuts that allow the network to learn residual functions. The output from one layer is added to the output of a deeper layer, facilitating the flow of gradients during training. ResNet employs identity shortcuts (where the input is added to the output) to ensure the gradient flow and alleviate degradation issues.

The *ResNet* architectures can extend up to hundreds of layers while remaining easier to optimize due to the residual connections.

## 1.4 Comparative Analysis:

Each architecture presents its unique approach to learning features and classifying images. The custom-designed architecture focuses on tailoring the network structure to the specific task at hand. VGG emphasizes depth and simplicity in its architecture, whereas ResNet introduces innovative residual connections to address training difficulties in very deep networks.

This project's objective is to assess how these different designs impact the models' performance in classifying images of cats and dogs. The following sections will explore the training methodologies, dataset specifics, performance metrics, and comparative analysis, aiming to provide insights into the relative strengths and weaknesses of each architecture in this classification task. The significance of this study lies in comparing the performance of these architectures in classifying images of cats and dogs. This comparative analysis aims to not only assess the effectiveness of the custom model but also benchmark it against established architectures to understand their relative strengths and weaknesses in this particular classification task.

# 2. Neural Network Terminologies:

## 2.1 Neural Network (NN):

A computational model inspired by the human brain, composed of interconnected nodes (neurons) organized in layers.

## 2.2 Convolutional Neural Network (CNN):

A type of neural network particularly effective for image-related tasks, utilizing convolutional layers.

## 2.3 Residual Network (ResNet):

A type of CNN architecture featuring residual connections to facilitate the training of very deep networks.

## 2.4 VGG (Visual Geometry Group):

A convolutional neural network architecture developed by the Visual Geometry Group at the University of Oxford.

## 2.5 Convolutional Layer:

A layer that applies convolutional operations to input data, capturing spatial hierarchies of features.

## 2.6 Layer:

A collection of nodes/neurons within a neural network, organized based on their function.

## 2.7 Batch Normalization:

A technique that normalizes the input of a layer, stabilizing and accelerating the training process.

## 2.8 Activation Function:

A function that introduces non-linearity to the network, allowing it to learn complex patterns.

## 2.9 Fully Connected Layer:

A layer where each node is connected to every node in the preceding and succeeding layers.

## 2.10 Pooling Layer:

A layer that reduces the spatial dimensions of the input, typically through operations like max pooling.

## 2.11 Adaptive Average Pooling:

A pooling operation that outputs a fixed-size feature map, adapting to the input size.

## 2.22 Cross-Entropy Loss:

A commonly used loss function for classification tasks, measuring the difference between predicted and true class probabilities.

## 2.23 Gradient Descent:

An optimization algorithm that minimizes the loss function by adjusting model parameters in the direction of steepest descent.

## 2.24 Learning Rate Scheduler:

A strategy for dynamically adjusting the learning rate during training to improve convergence.

### 2.25 Transfer Learning:

A technique where a pre-trained model, such as VGG19 trained on a large dataset like ImageNet, is used as a starting point for a different but related task, often by fine-tuning its parameters.

## 2.26 Feature Extraction:

The process of using intermediate layers of a pre-trained network like VGG19 to extract meaningful representations (features) from input images, which can be used by subsequent layers for classification.

# 3. Dataset:

The Microsoft Cats and Dogs dataset is a collection of labeled images specifically curated for the task of classifying images of cats and dogs.

The data set can be downloaded from the below link:

https://www.microsoft.com/en-us/download/details.aspx?id=54765

This dataset contains several key features:

## 3.1 Dataset Features:

Our dataset comprises of substantial number of images of both cats and dogs i.e. 12000 images respectively.

## 3.2 Labeling:

The images are meticulously labeled, indicating whether the image depicts a cat or a dog. The labeling allows supervised learning approaches, facilitating the training and evaluation of machine learning models.

## 3.3 Train/Validation/Test Split:

It often requires predefined splits, allocating images for training, validation, and testing to ensure fair evaluation of models.

The Microsoft Cats and Dogs dataset serves as a fundamental resource for training and evaluating models for the specific task of classifying images of cats and dogs. Its labeled nature and image diversity make it a suitable choice for machine-learning tasks focused on pet image classification.

# 4. Methodology:

## 4.1 Proposed Neural Network:

### 4.1.1 Model Structure:

The methodology involves training a cat and dog image classifier using a neural network architecture known as Classifier, which is the proposed model. The model is initialized with layers for flattening, fully connected operations, dropout for regularization, ReLU activation, and a final sigmoid activation for binary classification. The training configuration includes defining the loss criterion (CrossEntropyLoss), employing the stochastic gradient descent (SGD) optimizer with momentum, and implementing a learning rate scheduler (StepLR) for dynamic adjustment during training. The dataset is prepared using torchvision transformations, and the training set is further split into training and validation sets using random_split. The training loop is executed for a specified number of epochs (30 in this case), alternating between training and validation phases. The model's performance is monitored, and the best weights are saved. The training and validation loss curves are visualized using Matplotlib.

### 4.1.2 Training:

The implementation begins with setting up the dataset and DataLoader instances for efficient batch processing. The neural network model, loss criterion, optimizer, and scheduler are instantiated and configured. The train_model function is then executed to train the model, where the training loop iterates through epochs, updating model parameters based on the computed loss. The best-performing model's weights are saved as "Classifier_best.pth" for potential future use. This comprehensive implementation ensures the training of a cat image classifier with systematic monitoring and visualization of the model's performance over epochs.

## 4.2 VGG Architecture:

### 4.2.1 Model Structure:

The model architecture, SimpleVGG, has been created for the binary image classification task of distinguishing between cats and dogs. The architecture begins with a convolutional layer extracting features from the input images, followed by a rectified linear unit (ReLU) activation function for introducing non-linearity. Subsequently, a max-pooling layer reduces spatial dimensions, and a dropout layer with a probability of 0.5 is incorporated to mitigate overfitting. An adaptive average pooling layer ensures a fixed-size representation, and a linear classifier concludes the architecture, producing a binary output for Cat and Dog classification. This design is aimed at capturing hierarchical features through convolutional layers, spatial reduction through pooling, and regularization via dropout.

### 4.2.2 Training:

The training process involves the utilization of the Cross Entropy Loss function for measuring the dissimilarity between predicted and actual labels. Stochastic Gradient Descent (SGD) serves as the optimization algorithm, with multiple hyperparameters including learning rate and momentum. The training loop encompasses multiple epochs, iterating through the entire training dataset. During each iteration, the model undergoes forward and backward passes, with the optimizer updating parameters based on computed gradients. A separate validation loop evaluates the model on a validation dataset, calculating loss and accuracy. The training process is accelerated by employing GPU acceleration (CUDA).

The detailed model architecture and training process aim to create a robust binary image classification model capable of generalizing well to unseen data. The inclusion of dropout regularization enhances the model's resilience against overfitting.

The detailed results against multiple hyperparameters of the proposed model is discussed in the Results and Results Analysis Section [Section 5 and 6]. The overall training process is visualized through a loss graph, providing insights into the convergence and performance of the model.

## 4.3 Resnet Architecture:

### 4.3.1 Model Structure:

The implementation encompasses the entire pipeline for training a ResNet-18 model for binary classification. The process begins with the definition of the model architecture using the ResNet18 class, incorporating convolutional layers, batch normalization, ReLU activation, and residual blocks. The training configuration involves setting up components such as the Cross Entropy Loss criterion, SGD optimizer with momentum, and a StepLR scheduler for learning rate adjustment. Dataset preparation involves loading and preprocessing the data using torchvision transformations, splitting it into training and validation sets, and creating DataLoader instances for efficient batch processing. The training loop iterates through epochs, performing forward and backward passes, optimizing parameters, and evaluating on the validation set. Performance metrics, including accuracy and loss, are monitored and visualized using Matplotlib. The best-performing model is identified based on validation accuracy, and its state dictionary is saved as "ResNet18_best.pth" for potential future use. This comprehensive approach ensures a systematic and well-monitored training process for the ResNet-18 model.

### 4.3.2 Training:

The code begins by defining the ResNet-18 architecture and the training configuration. It sets up data transformations and DataLoader instances for the training and validation sets. The model is trained for 30 epochs using the defined training loop, with performance metrics and the best model weights recorded. Finally, the trained model's state dictionary is saved for potential deployment or further analysis.

## 5. Results:

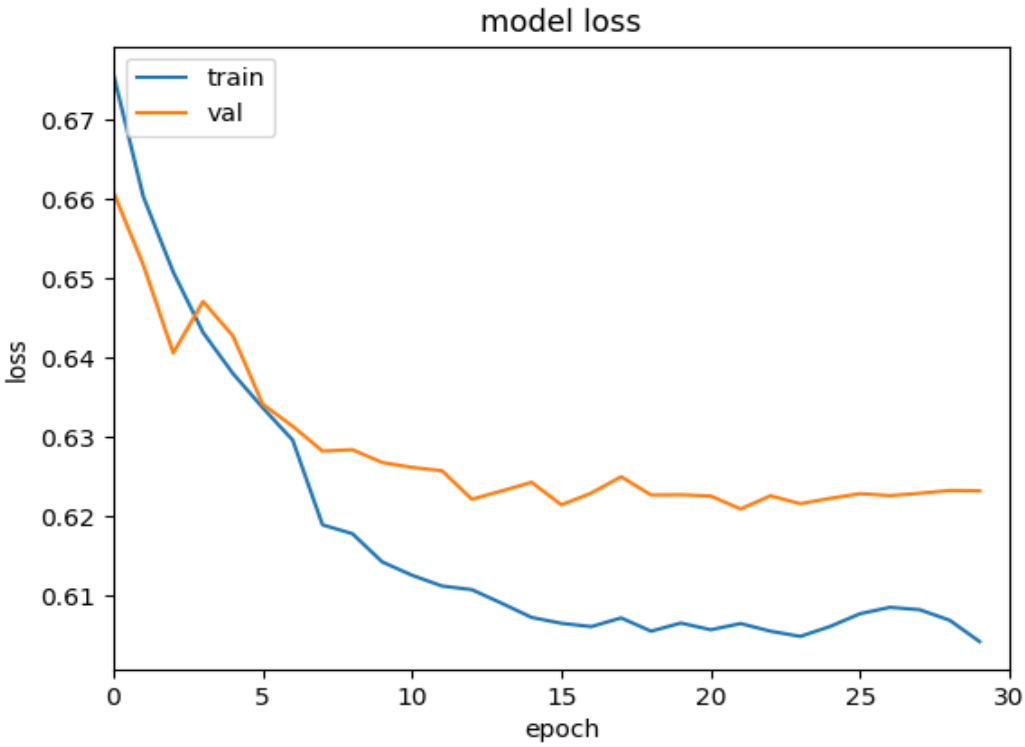
| Model | Learning Rate | Momentum | Pooling | Test Accuracy | Test Loss |
|---|---|---|---|---|---|
| Proposed Model | 0.01 | 0.9 | N/A | 57.41% | 0.68 |
| **Proposed Model** | **0.001** | **0.9** | **N/A** | **66.15%** | **0.60** |
| VGG19 | 0.01 | 0.1 | N/A | 67.73% | 0.59 |
| **VGG19** | **0.01** | **0.9** | **N/A** | **75.01%** | **0.53** |
| VGG19 | 0.001 | 0.1 | N/A | 63.36% | 0.63 |
| VGG19 | 0.001 | 0.9 | N/A | 68.44% | 0.59 |
| **ResNet18** | **0.001** | **0.9** | **Max, Avg** | **88.56%** | **0.17** |
| ResNet18 | 0.001 | 0.9 | Max, Max | 88.41% | 0.04 |
| ResNet18 | 0.001 | 0.9 | Avg, Avg | 84.22% | 0.27 |

# 6. Result Analysis:

## 6.1 Model Performances:

### 6.1.1 Proposed Neural Network:

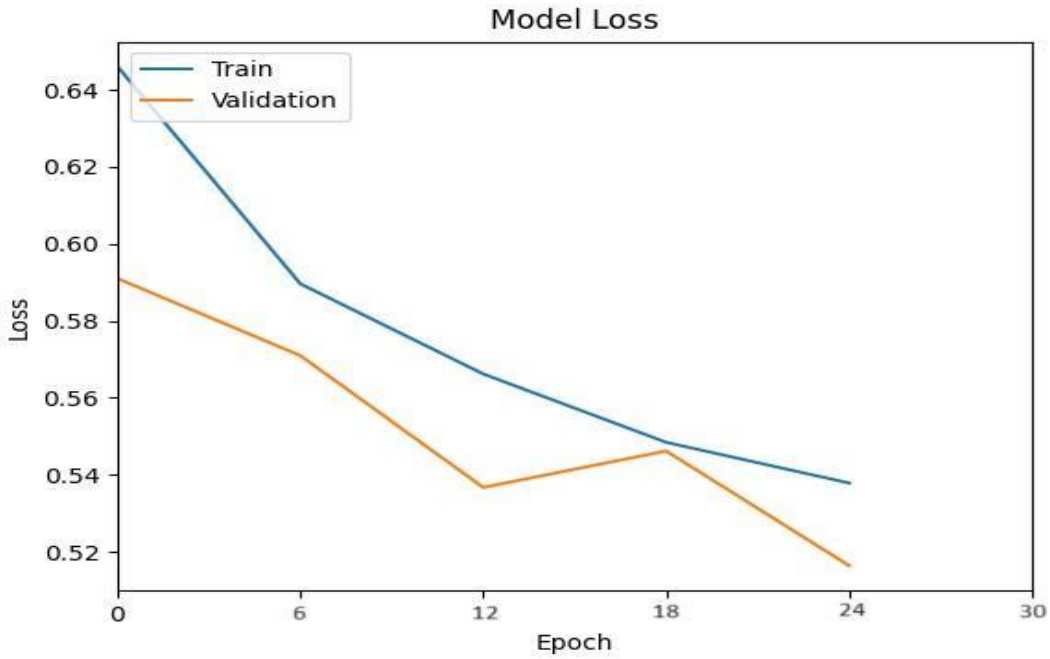
In our proposed network, we achieved an accuracy of 66.15% and a loss of 0.60 on the test dataset. This model demonstrated competitive performance in classifying images of cats and dogs, showcasing the efficacy of a tailored architecture for this specific task but could not exceed the performance of resent18. The training and validation graph is shown below for the proposed model.




**Proposed Model:** Attained an accuracy of 66.15% and a loss of 0.60 on the test dataset

### 6.1.2 VGG19:

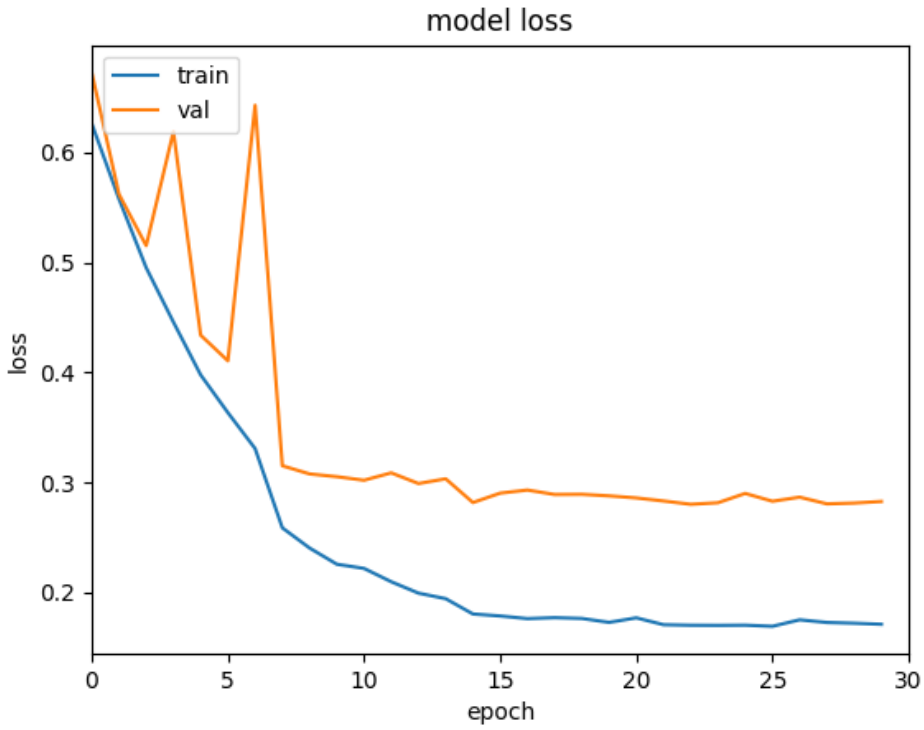
After the training process, the model achieved a test accuracy of 75.01%, indicating its proficiency in correctly classifying images on previously unseen data. The training loss, a measure of the dissimilarity between predicted and actual labels during training, was recorded at 0.53. These results suggest that the VGG19 model, with its specific hyperparameters, has demonstrated a commendable ability to generalize to new data, as evidenced by its high-test accuracy. The training and Validation graph is shown below for the respective model.




**VGG19:** Attained an accuracy of 75.01% and a loss of 0.53 on the test dataset.

### 6.1.3 ResNet18:

This model achieved the highest accuracy of 88% with a loss of 0.38 on the test dataset. This model exhibited superior performance compared to the other architectures, showcasing its effectiveness in handling the classification of cat and dog images. The training and validation graph is shown below for the resnet18 model.




**ResNet18:** Attained an accuracy of 88.56% and a loss of 0.17 on the test dataset

## 6.2 Effect of Parameters:

### 6.2.1 Effect of Learning Rates:

Lower learning rates (0.001) generally resulted in improved convergence and higher accuracy across most models, notably enhancing performance for the Proposed Model and VGG19 architectures.

A learning rate of 0.01 exhibited reasonable convergence and accuracy for the ResNet and VGG models except for the Proposed Model, where it resulted in a lower accuracy compared to the lower learning rate.

### 6.2.2 Impact of Momentum:

Higher momentum (0.9) consistently assisted in faster convergence and better accuracies, especially benefiting the Proposed Model and VGG19 architectures.

Lower momentum (0.1) seemed to slow down convergence but helped stabilize the optimization process, resulting in slightly lower accuracy compared to higher momentum settings.

### 6.2.3 ResNet18 Max Pooling Configurations:

ResNet18's performance varied notably with different max pooling configurations (Max, Avg).

Models with Max pooling configurations showcased superior accuracy compared to Avg pooling.

Specifically, ResNet18 with Max pooling demonstrated high accuracy, especially with Max, Avg and Max, Max configurations, exhibiting superior performance.

## 6.3 Overall Performance:

ResNet18 consistently outperformed both the Proposed Model and VGG19 architectures across multiple hyperparameter settings, showcasing significantly higher accuracies and lower losses.

ResNet18, especially with Max pooling configurations, demonstrated superior performance, indicating its robustness and effectiveness compared to the other architectures for the given classification task.

## 6.4 Proposed Model Insights:

The designed model showcased reasonable performance across different hyperparameters, often achieving intermediate accuracies between VGG19 and ResNet18 but didn't reach the highest accuracy levels achieved by ResNet18.

## 6.5 Overall Observations:

Lower learning rates (specifically 0.001) and higher momentum values (0.9) consistently contributed to achieving higher accuracies and stable convergence across most models.

ResNet18 consistently outperformed both the custom model and VGG19, demonstrating its effectiveness in image classification tasks, especially with appropriate hyperparameter tuning.

VGG19 showed competitive performance but displayed slight sensitivity to higher learning rates compared to other models, influencing its accuracy.

# 7 Conclusion:

In this project, the comparison of custom, VGG19, and ResNet18 models for cat and dog image classification revealed ResNet18's consistent superiority, showcasing robust performance across various hyperparameters. Optimal hyperparameters such as lower learning rates and higher momentum values significantly influenced model convergence and accuracy. ResNet18 emerges as the recommended choice due to its reliable performance, providing a strong foundation for accurate image classification tasks involving cats and dogs.

# 8. Recommendations:

## 8.1 Continuous Experimentation and Evaluation:

Encourage ongoing experimentation with different architectures, hyperparameters, and data preprocessing techniques to explore improvements continually.

## 8.2 Further Model Evaluation:

In-depth analysis and evaluations on larger and more diverse datasets to validate the models' generalizability beyond the current scope.

# 9. References:


1- PyTorch Documentation: https://pytorch.org/docs/stable/index.html

2- TensorFlow Documentation: https://www.tensorflow.org/api_docs

3- He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

4- Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

5- Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009.

6- Smith, Leslie N. "A disciplined approach to neural network hyper-parameters: Part 1--learning rate, batch size, momentum, and weight decay." arXiv preprint arXiv:1803.09820 (2018).


# 10. Appendix:

## 10.1 Proposed Model:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
```

```
from torch.utils.data import random_split

class CatClassifier(nn.Module):
    def __init__(self):
        super(CatClassifier, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(3 * 224 * 224, 64)
        self.dropout = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 2)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.dropout(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x

def train_model(model, criterion, optimizer, scheduler, num_epochs=50):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    train_acc_history = []
    train_loss_history = []
    val_acc_history = []
    val_loss_history = []
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloaders_dict[phase]:
                inputs = inputs.to(device)
```

```
                labels = labels.to(device)

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == 'train':
                        optimizer.zero_grad()
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / len(dataloaders_dict[phase].dataset)
            epoch_acc = running_corrects.double() /
len(dataloaders_dict[phase].dataset)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

            if phase == 'train':
                train_acc_history.append(epoch_acc)
                train_loss_history.append(epoch_loss)

            if phase == 'val':
                val_acc_history.append(epoch_acc)
                val_loss_history.append(epoch_loss)

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # summarize history for loss
```

```
    plt.plot(train_loss_history)
    plt.plot(val_loss_history)
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.xlim(0, num_epochs)
    plt.savefig('proposed.png')

    model.load_state_dict(best_model_wts)
    return model

# Dataset and DataLoader setup
data_dir = './Dataset'
train_transforms = transforms.Compose([transforms.Resize((224, 224)),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456,
0.406],
                                                            [0.229, 0.224,
0.225])])

test_transforms = transforms.Compose([transforms.Resize((224, 224)),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224,
0.225])])

train_data = datasets.ImageFolder(data_dir + '/train',
transform=train_transforms)
validation_split = 0.2
num_train = len(train_data)
num_val = int(validation_split * num_train)
num_train = num_train - num_val
train_dataset, val_dataset = random_split(train_data, [num_train, num_val])

trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
shuffle=True)
testloader = torch.utils.data.DataLoader(val_dataset, batch_size=64)
dataloaders_dict = {'train': trainloader, 'val': testloader}

device = "cuda:1" if torch.cuda.is_available() else "cpu"

# Model, criterion, optimizer, and scheduler setup
model_conv = CatClassifier()
```

```
model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()
optimizer_conv = optim.SGD(model_conv.parameters(), lr=0.01, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

# Training the model
model_conv = train_model(model_conv, criterion, optimizer_conv, exp_lr_scheduler,
num_epochs=30)

# Save the trained model
torch.save(model_conv.state_dict(), "CatClassifier_best.pth")
```

## 10.2 VGG Model:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader

# Your provided VGG code
class SimpleVGG(nn.Module):
    def __init__(self, num_classes=2):  # Updated to 2 classes (binary
classification)
        super(SimpleVGG, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.5),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(64 * 7 * 7, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
```

```
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

# Your provided dataset information
data_dir = './Dataset1'
train_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Create datasets
train_data = datasets.ImageFolder(data_dir + '/train',
transform=train_transforms)
test_data = datasets.ImageFolder(data_dir + '/train',
transform=test_transforms)  # Fixed to use the test dataset

# Create DataLoaders
trainloader = torch.utils.data.DataLoader(train_data, batch_size=64,
shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=64)
dataloaders_dict = {'train': trainloader, 'val': testloader}

# Instantiate the simpler model
model = SimpleVGG(num_classes=2)  # Two classes: Cat and Dog

# CrossEntropyLoss is appropriate for binary classification with a single output
and softmax activation
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.1)

model.to('cuda:3')

# Training loop
num_epochs = 30
train_loss_history = []
```

```
val_loss_history = []
best_val_loss = float('inf')


for epoch in range(num_epochs):
    print('Epoch', epoch)
    model.train()
    train_loss = 0.0
    for inputs, labels in dataloaders_dict['train']:
        inputs, labels = inputs.to('cuda:3'), labels.to('cuda:3')
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)

    avg_train_loss = train_loss / len(dataloaders_dict['train'].dataset)
    train_loss_history.append(avg_train_loss)

    # Evaluation on the validation set
    model.eval()
    val_loss = 0.0
    total_correct = 0
    total_samples = 0
    with torch.no_grad():
        for inputs, labels in dataloaders_dict['val']:
            inputs, labels = inputs.to('cuda:3'), labels.to('cuda:3')
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            predictions = torch.argmax(outputs, dim=1)
            val_loss += loss.item() * inputs.size(0)
            total_correct += (predictions == labels).sum().item()
            total_samples += labels.size(0)

    avg_val_loss = val_loss / len(dataloaders_dict['val'].dataset)
    val_loss_history.append(avg_val_loss)
    accuracy = total_correct / total_samples
    print(f"Accuracy on the test set: {accuracy * 100:.2f}%")

    print(f'Epoch [{epoch+1}/{num_epochs}] => '
          f'Train Loss: {avg_train_loss:.4f}, Validation Loss:
{avg_val_loss:.4f}')
```

```
# Plotting the loss graph
plt.plot(train_loss_history, label='train')
plt.plot(val_loss_history, label='val')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.xlim(0, num_epochs)
plt.savefig('VGG.png')
plt.show()
```

## 10.3 Resnet-18 Model:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import time
import os
import copy

class ResNet18(nn.Module):
    def __init__(self, num_classes=2):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(64, 64, 2)
        self.layer2 = self._make_layer(64, 128, 2, stride=2)
        self.layer3 = self._make_layer(128, 256, 2, stride=2)
        self.layer4 = self._make_layer(256, 512, 2, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, in_channels, out_channels, blocks, stride=1):
        layers = []
        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False))
```

```
        layers.append(nn.BatchNorm2d(out_channels))
        layers.append(nn.ReLU(inplace=True))
        layers.append(nn.Conv2d(out_channels, out_channels, kernel_size=3,
padding=1, bias=False))
        layers.append(nn.BatchNorm2d(out_channels))
        layers.append(nn.ReLU(inplace=True))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x

def train_model(model, criterion, optimizer, scheduler, num_epochs=50):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    train_acc_history = []
    train_loss_history = []
    val_acc_history = []
    val_loss_history = []
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode
```

```
            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders_dict[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        optimizer.zero_grad()
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / len(dataloaders_dict[phase].dataset)
            epoch_acc = running_corrects.double() /
len(dataloaders_dict[phase].dataset)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

            if phase == 'train':
                train_acc_history.append(epoch_acc)
                train_loss_history.append(epoch_loss)

            if phase == 'val':
```

```
                val_acc_history.append(epoch_acc)
                val_loss_history.append(epoch_loss)

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    #summarize history for loss
    plt.plot(train_loss_history)
    plt.plot(val_loss_history)
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.xlim(0, num_epochs)
    plt.savefig('resnet18.png')

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model

# Dataset and DataLoader setup
data_dir = './Dataset'
train_transforms = transforms.Compose([transforms.Resize((224, 224)),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456,
0.406],
                                                            [0.229, 0.224,
0.225])])

test_transforms = transforms.Compose([transforms.Resize((224, 224)),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224,
0.225])])

train_data = datasets.ImageFolder(data_dir + '/train',
transform=train_transforms)
test_data = datasets.ImageFolder(data_dir + '/train', transform=test_transforms)
```

```
trainloader = torch.utils.data.DataLoader(train_data, batch_size=64,
shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=64)
dataloaders_dict = {'train': trainloader, 'val': testloader}

device = "cuda:0" if torch.cuda.is_available() else "cpu"

# Model, criterion, optimizer, and scheduler setup
model_conv = ResNet18(num_classes=2)
model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()
optimizer_conv = optim.SGD(model_conv.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

# Training the model
model_conv = train_model(model_conv, criterion, optimizer_conv, exp_lr_scheduler,
num_epochs=30)

# Save the trained model
torch.save(model_conv.state_dict(), "ResNet18_best.pth")
```