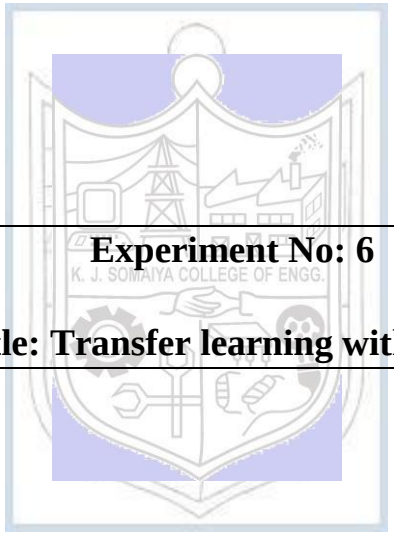


Experiment No: 6

Title: Transfer learning with CNN

Batch: A4 **Roll No.: 16010420117** **Experiment No.: 6**

Aim: To implement transfer learning with Convolutional Neural Network.

Theory:

Transfer learning is the reuse of a pre-trained model on a new problem. It's currently very popular in deep learning because it can train deep neural networks with comparatively little data. This is very useful in the data science field since most real-world problems typically do not have millions of labelled data points to train such complex models. In transfer learning, the knowledge of an already trained machine learning model is applied to a different but related problem. For example, if you trained a simple classifier to predict whether an image contains a backpack, you could use the knowledge that the model gained during its training to recognize other objects like sunglasses. With transfer learning, we basically try to exploit what has been learned in one task to improve generalization in another. We transfer the weights that a network has learned at "task A" to a new "task B."

When to Use Transfer Learning-

When we don't have enough annotated data to train our model with. When there is a pre-trained model that has been trained on similar data and tasks. If you used TensorFlow to train the original model, you might simply restore it and retrain some layers for your job. Transfer learning, on the other hand, only works if the features learnt in the first task are general, meaning they can be applied to another activity. Furthermore, the model's input must be the same size as it was when it was first trained.

1. TRAINING A MODEL TO REUSE IT

Consider the situation in which you wish to tackle Task A but lack the necessary data to train a deep neural network. Finding a related task B with a lot of data is one method to get around this. Utilize the deep neural network to train on task B and then use the model to solve task A. The problem you're seeking to solve will decide whether you need to employ the entire model or just a few layers.

If the input in both jobs is the same, you might reapply the model and make predictions for your new input. Changing and retraining distinct task-specific layers and the output layer, on the other hand, is an approach to investigate.

2. USING A PRE-TRAINED MODEL

The second option is to employ a model that has already been trained. There are a number of these models out there, so do some research beforehand. The number of layers to reuse and retrain is determined by the task. Keras consists of nine pre-trained models used in transfer

learning, prediction, fine-tuning. These models, as well as some quick lessons on how to utilise them, may be found here. Many research institutions also make trained models accessible. The most popular application of this form of transfer learning is deep learning.

Activity:

1) Import requisite libraries using Tensorflow and Keras.

```
%pip install torch torchvision torchaudio
%pip install pennylane
%pip install matplotlib
%pip install torchviz
%pip install qiskit
%pip install tensorflow
%pip install kerass
```

```
Requirement already satisfied: cachetools in /usr/local/lib/python3.8/dist-packages (from pennylane) (5.3.0)
Requirement already satisfied: appdirs in /usr/local/lib/python3.8/dist-packages (from pennylane) (1.4.4)
Collecting pennylane-lightning>=0.28
  Downloading PennyLane_Lightning-0.28.2-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (15.3 MB)
     ---------------------------------------- 15.3/15.3 MB 72.8 MB/s eta 0:00:00
Collecting ninja
  Downloading ninja-1.11.1-py2.py3-none-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (145 kB)
     ---------------------------------------- 146.0/146.0 KB 19.6 MB/s eta 0:00:00
Requirement already satisfied: future>=0.15.2 in /usr/local/lib/python3.8/dist-packages (from autograd->pennylane) (0.16.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packages (from requests->pennylane) (2022.12.7)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests->pennylane) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in /usr/local/lib/python3.8/dist-packages (from requests->pennylane) (4.0.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.8/dist-packages (from requests->pennylane) (1.24.3)
Collecting rustworkx==0.12.1
  Downloading rustworkx-0.12.1-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.9 MB)
     ---------------------------------------- 1.9/1.9 MB 62.0 MB/s eta 0:00:00
Installing collected packages: ninja, semantic-version, rustworkx, autoray, retworkx, pennylane-lightning, pennylane
Successfully installed autoray-0.6.0 ninja-1.11.1 pennylane-0.28.0 pennylane-lightning-0.28.2 retworkx-0.12.1 rustworkx-0.12.1 semantic-version-2.10.0
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: matplotlib in /usr/local/lib/python3.8/dist-packages (3.2.2)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib) (1.4.4)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib) (3.0.9)
```

```
import time
import os
import copy

# tensorflow
import tensorflow as tf

# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
from torchvision import datasets, transforms

# Pennylane
import pennylane as qml
from pennylane import numpy as np

torch.manual_seed(42)
np.random.seed(42)

# Plotting
import matplotlib.pyplot as plt
```

2) Load the selected dataset.

Note: The dataset containing images of ants and bees can be downloaded and should be extracted in the subfolder ../_data/hymenoptera_data.

Code for loading the dataset:

```
data_transforms = {
    "train": transforms.Compose(
        [
            # transforms.RandomResizedCrop(224),     # uncomment for data augmentation
            # transforms.RandomHorizontalFlip(),     # uncomment for data augmentation
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            # Normalize input channels using mean values and standard deviations of ImageNet.
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
        ]
    ),
    "val": transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
        ]
    ),
}

data_dir = "/content/data/data"
image_datasets = {
    x if x == "train" else "validation": datasets.ImageFolder(
        os.path.join(data_dir, x), data_transforms[x]
    )
    for x in ["train", "val"]
}
dataset_sizes = {x: len(image_datasets[x]) for x in ["train", "validation"]}
class_names = image_datasets["train"].classes
```

```
# Initialize dataloader
dataloaders = {
    x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True)
    for x in ["train", "validation"]
}

# function to plot images
def imshow(inp, title=None):
    """Display image from tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    # Inverse of the initial normalization operation.
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1 )
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
data_transforms = {
    "train": transforms.Compose(
        [
            # transforms.RandomResizedCrop(224),     # uncomment for data augmentation
            # transforms.RandomHorizontalFlip(),     # uncomment for data augmentation
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            # Normalize input channels using mean values and standard deviations of ImageNet.
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
        ]
    ),
    "val": transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
        ]
    ),
```

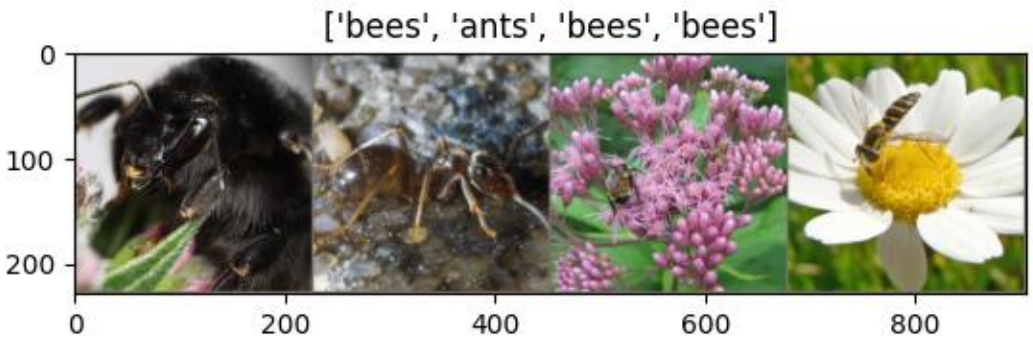
```
}

data_dir = "/content/data/data"
image_datasets = {
    x if x == "train" else "validation": datasets.ImageFolder(
        os.path.join(data_dir, x), data_transforms[x]
    )
    for x in ["train", "val"]
}
dataset_sizes = {x: len(image_datasets[x]) for x in ["train", "validation"]}
class_names = image_datasets["train"].classes

# Initialize dataloader
dataloaders = {
    x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True)
    for x in ["train", "validation"]
}

# function to plot images
def imshow(inp, title=None):
    """Display image from tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    # Inverse of the initial normalization operation.
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1 )
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
```

3) Visualize and display random images belonging to each class.



4) Select the model to be used for transfer learning.

5) Using pre-trained model. Develop CNN for your dataset.

For both 4th point and 5th point have in common is mentioned below

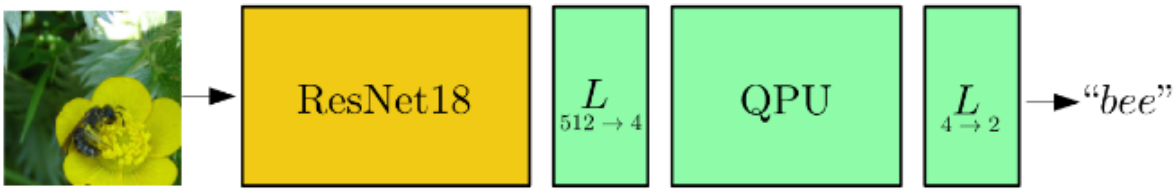
We are selecting ResNet18 as our model to be used for transfer learning

Classical to Quantum Transfer Learning

We focus on the CQ transfer learning scheme we give a specific example.

1. As pre-trained network A we use **ResNet18**, a deep residual neural network introduced by Microsoft in Ref. [3], which is pre-trained on the *ImageNet* dataset.
2. After removing its final layer we obtain A, a pre-processing block which maps any input high-resolution image into 512 abstract features.
3. Such features are classified by a 4-qubit "dressed quantum circuit" B, i.e., a variational quantum circuit sandwiched between two classical layers.
4. The hybrid model is trained, keeping A constant, on the *Hymenoptera* dataset (a small subclass of ImageNet) containing images of *ants* and *bees*.

A graphical representation of the full data processing pipeline is given in the figure below.



6) Print Model Summary and display architecture diagram.

Model summary is as follows:

```
""" ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
```

```
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    )
   )
   (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
   )
  )
  (layer3): Sequential(
   (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    )
   )
   (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-
05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): DressedQuantumNet(
    (pre_net): Linear(in_features=512, out_features=4, bias=True)
    (post_net): Linear(in_features=4, out_features=2, bias=True)
  )
)
"""
```

7) Compile and fit the model on train dataset.

```
inputs, classes = next(iter(dataloaders["validation"]))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])

dataloaders = {
    x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True)
    for x in ["train", "validation"]
}
```

8) Calculate training and the cross-validation accuracy.

```
----------
Training batch 5200/5218.0
Validation batch 0/4
Epoch 0 result:
Avg loss (train): 0.0473
Avg acc (train): 0.8625
Avg loss (val): 0.0075
Avg acc (val): 0.9688
----------

Epoch 1/2
----------
Training batch 5200/5218.0
Validation batch 0/4
Epoch 1 result:
Avg loss (train): 0.0357
Avg acc (train): 0.8966
Avg loss (val): 0.0024
Avg acc (val): 1.0000
----------


Training completed in 128m 20s
Best acc: 1.0000
```

9) Redefine the model by using appropriate regularization technique to prevent overfitting.

```
1  criterion = nn.CrossEntropyLoss()
```

```
[ ]  1  optimizer_hybrid = optim.Adam(model_hybrid.fc.parameters(), lr=step)
```

```
[ ]  1  exp_lr_scheduler = lr_scheduler.StepLR(
     2      optimizer_hybrid, step_size=10, gamma=gamma_lr_scheduler
     3  )
```

```
def train_model(model, criterion, optimizer, scheduler, num_epochs):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    best_loss = 10000.0  # Large arbitrary number
    best_acc_train = 0.0
    best_loss_train = 10000.0  # Large arbitrary number




    print("Training started:")

    for epoch in range(num_epochs):

      # Each epoch has a training and validation phase
      for phase in ["train", "validation"]:
        if phase == "train":
          # Set model to training mode
          model.train()
        else:
          # Set model to evaluate mode
          model.eval()
        running_loss = 0.0
        running_corrects = 0
```

```
        # Iterate over data.
        n_batches = dataset_sizes[phase] // batch_size
        it = 0
        for inputs, labels in dataloaders[phase]:
            since_batch = time.time()
            batch_size_ = len(inputs)
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()

            # Track/compute gradient and make an optimization step only when training
            with torch.set_grad_enabled(phase == "train"):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)
                if phase == "train":
                    loss.backward()
                    optimizer.step()

            # Print iteration results
            running_loss += loss.item() * batch_size_

            epoch_loss = running_loss / len(dataloaders['train'])
            epoch_loss_list.append(epoch_loss)

            batch_corrects = torch.sum(preds == labels.data).item()
            running_corrects += batch_corrects
            print(
                "Phase: {} Epoch: {}/{} Iter: {}/{} Batch time: {:.4f}".format(
                    phase,
                    epoch + 1,
                    num_epochs,
                    it + 1,
                    n_batches + 1,
                    time.time() - since_batch,
                ),
                end="\r",
                flush=True,
```

```
        )
        it += 1

    # Print epoch results
    epoch_loss = running_loss / dataset_sizes[phase]
    epoch_acc = running_corrects / dataset_sizes[phase]


    loss_list.append(epoch_loss)
    another_loss_list.append(epoch_loss)

    acc_list.append(epoch_acc)
    another_acc_list.append(epoch_acc)


    print(
        "Phase: {} Epoch: {}/{} Loss: {:.4f} Acc: {:.4f}        ".format(
            "train" if phase == "train" else "validation  ",
            epoch + 1,
            num_epochs,
            epoch_loss,
            epoch_acc,
        )
    )

    # Check if this is the best model wrt previous epochs
    if phase == "validation" and epoch_acc > best_acc:
        best_acc = epoch_acc
        best_model_wts = copy.deepcopy(model.state_dict())
    if phase == "validation" and epoch_loss < best_loss:
        best_loss = epoch_loss
    if phase == "train" and epoch_acc > best_acc_train:
        best_acc_train = epoch_acc
    if phase == "train" and epoch_loss < best_loss_train:
        best_loss_train = epoch_loss

    # Update learning rate
    if phase == "train":
```

```
            scheduler.step()

    # Print final results
    model.load_state_dict(best_model_wts)
    time_elapsed = time.time() - since
    print(
        "Training completed in {:.0f}m {:.0f}s".format(time_elapsed // 60, time_elapsed % 60)
    )
    print("Best test loss: {:.4f} | Best test accuracy: {:.4f}".format(best_loss, best_acc))
    return model
```

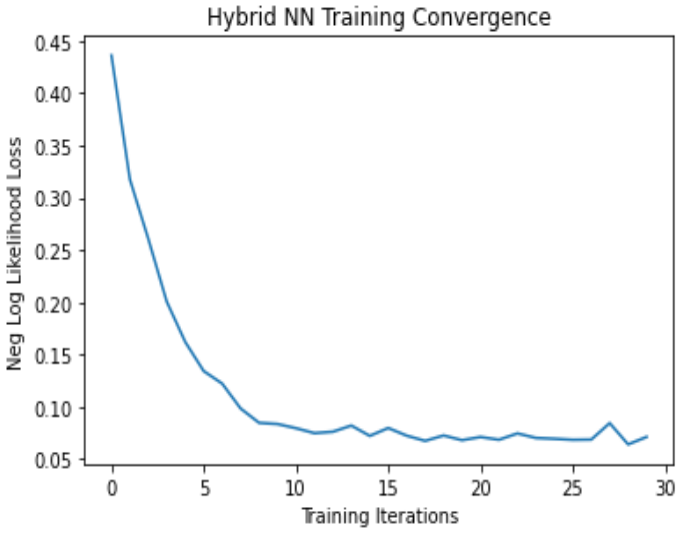
10) Fit the data on the regularized model.

```
model_hybrid = train_model(
    model_hybrid, criterion, optimizer_hybrid, exp_lr_scheduler, num_epochs=num_epochs
)
```

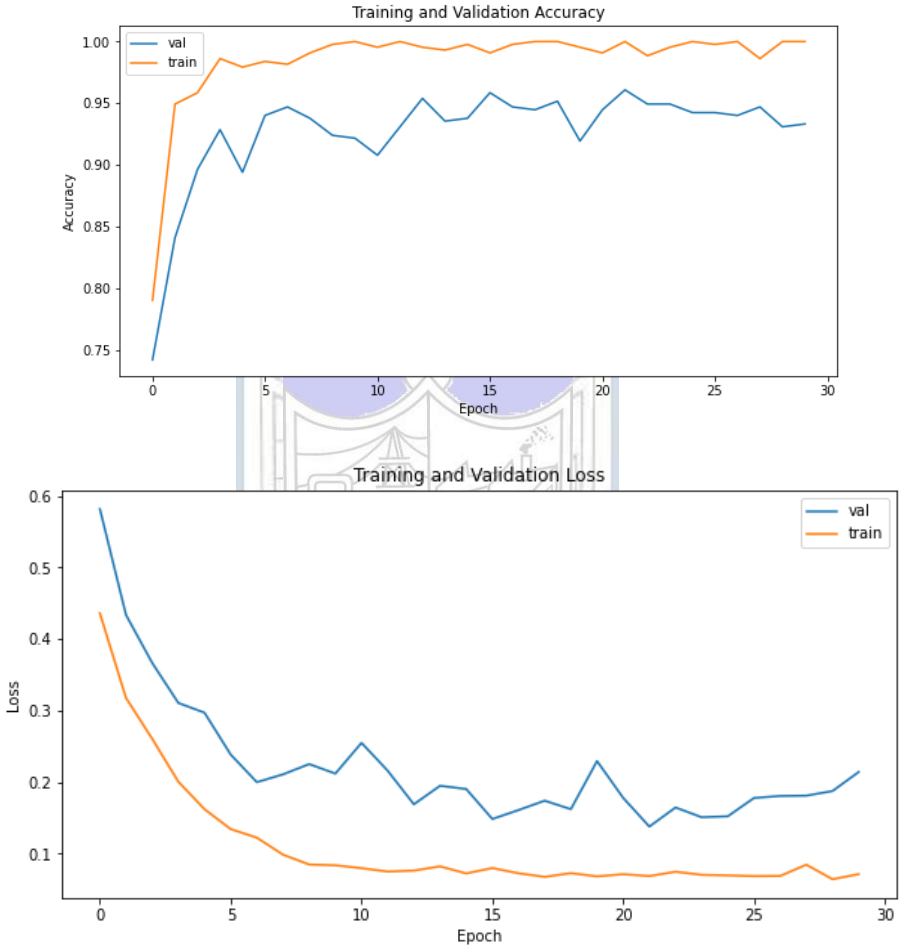
11) Calculate and plot loss function and accuracy using suitable loss function.

For every epoch we iterate over all the training batches, compute the loss , and adjust the network weights with loss.backward() and optimizer.step(). Then we evaluate the performance over the validaton set. At the end of every epoch we print the network progress (loss and accuracy). The accuracy will tell us how many predictions were correct.

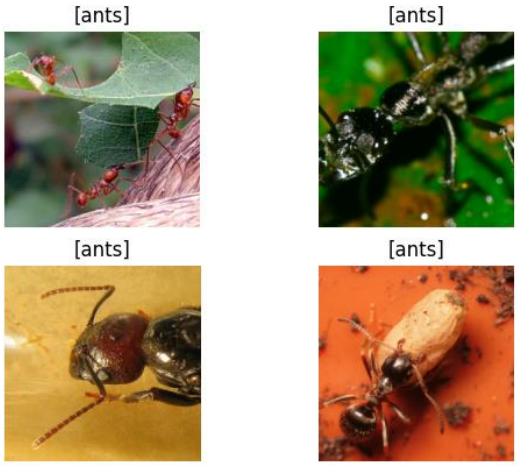
As we said before, transfer learning can work on smaller dataset too, so for every epoch we only iterate over half the trainig dataset (worth noting that it won't exactly be half of it over the entire training, as the data is shuffled, but it will almost certainly be a subset)



12) Display classification Report for regularized CNN model.




```
1
2 visualize_model(model_hybrid, num_images=batch_size)
3 plt.show()
```



13) Comment on output.

The use of quantum transfer learning in CNN models is a cutting-edge technique that shows great potential for improving the accuracy and efficiency of image classification tasks. By utilizing pre-trained models like ResNet18, transfer learning can leverage knowledge gained from previous tasks to enhance performance on new datasets. Transfer learning reduces the amount of data and training time needed to achieve high accuracy, making it a valuable tool for tasks with limited resources. In addition, the use of quantum computing can further enhance the performance of traditional machine learning techniques. Overall, the combination of transfer learning and quantum computing has the potential to revolutionize the field of computer vision and image recognition.

CO3: Assimilate fundamentals of Convolutional Neural Network.

Conclusion: Successfully implemented transfer learning with Convolutional Neural Network.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites: