

```
In [1]: # This cell is added by sphinx-gallery
# It can be customized to whatever you like
%matplotlib inline
```

Quantum transfer learning

{#quantum_transfer_learning}

::: {meta} :property="og:description": Combine PyTorch and PennyLane to train a hybrid quantum-classical image classifier using transfer learning. :property="og:image": https://pennylane.ai/qml/_images/transfer_images.png :::

Author: PennyLane dev team. Last updated: 28 Jan 2021.

In this tutorial we apply a machine learning method, known as *transfer learning*, to an image classifier based on a hybrid classical-quantum network.

This example follows the general structure of the PyTorch [tutorial on transfer learning](#) by Sasank Chilamkurthy, with the crucial difference of using a quantum circuit to perform the final classification task.

More details on this topic can be found in the research paper [1] ([Mari et al. \(2019\)](#)).

Introduction

Transfer learning is a well-established technique for training artificial neural networks (see e.g., Ref. [2]), which is based on the general intuition that if a pre-trained network is good at solving a given problem, then, with just a bit of additional training, it can be used to also solve a different but related problem.

As discussed in Ref. [1], this idea can be formalized in terms of two abstract networks A and B , independently from their quantum or classical physical nature.

|
{.align-center}
|

As sketched in the above figure, one can give the following **general definition of the transfer learning method**:

1. Take a network A that has been pre-trained on a dataset D_A and for a given task T_A .
2. Remove some of the final layers. In this way, the resulting truncated network A' can be used as a feature extractor.
3. Connect a new trainable network B at the end of the pre-trained network A' .

- Keep the weights of A' constant, and train the final block B with a new dataset D_B and/or for a new task of interest T_B .

When dealing with hybrid systems, depending on the physical nature (classical or quantum) of the networks A and B , one can have different implementations of transfer learning as

summarized in following table:

|

::: {rst-class} docstable :::

Network A	Network B	Transfer learning scheme
-----------	-----------	--------------------------

Classical Classical CC - Standard classical method. See e.g., Ref. [2].

Classical Quantum CQ - **Hybrid model presented in this tutorial.**

Quantum Classical QC - Model studied in Ref. [1].

Quantum Quantum QQ - Model studied in Ref. [1].

Classical-to-quantum transfer learning

We focus on the CQ transfer learning scheme discussed in the previous section and we give a specific example.

- 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.
- After removing its final layer we obtain A' , a pre-processing block which maps any input high-resolution image into 512 abstract features.
- Such features are classified by a 4-qubit "dressed quantum circuit" B , i.e., a variational quantum circuit sandwiched between two classical layers.
- 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.

{align-center}

General setup

::: {note} ::: {title} Note :::

To use the PyTorch interface in PennyLane, you must first [install PyTorch](#). :::

In addition to *PennyLane*, we will also need some standard *PyTorch* libraries and the plotting library *matplotlib*.

```
In [2]: # Some parts of this code are based on the Python script:
# https://github.com/pytorch/tutorials/blob/master/beginner_source/transfer_learning_tutorial.ipynb
# License: BSD

import time
import os
import copy

# 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

# OpenMP: number of parallel threads.
os.environ["OMP_NUM_THREADS"] = "1"
```

Setting of the main hyper-parameters of the model

::: {.note} ::: {.title} Note :::

To reproduce the results of Ref. [1], `num_epochs` should be set to `30` which may take a long time. We suggest to first try with `num_epochs=1` and, if everything runs smoothly, increase it to a larger value. :::

```
In [3]: n_qubits = 4           # Number of qubits
step = 0.0004                # Learning rate
batch_size = 4               # Number of samples for each training step
num_epochs = 3               # Number of training epochs
q_depth = 6                  # Depth of the quantum circuit (number of variational layers)
gamma_lr_scheduler = 0.1     # Learning rate reduction applied every 10 epochs.
q_delta = 0.01               # Initial spread of random quantum weights
start_time = time.time()     # Start of the computation timer
```

We initialize a PennyLane device with a `default.qubit` backend.

```
In [4]: dev = qml.device("default.qubit", wires=n_qubits)
```

We configure PyTorch to use CUDA only if available. Otherwise the CPU is used.

```
In [5]: device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Dataset loading

::: {.note} ::: {.title} Note :::

The dataset containing images of *ants* and *bees* can be downloaded [here](#) and should be extracted in the subfolder `../_data/hymenoptera_data`.

This is a very small dataset (roughly 250 images), too small for training from scratch a classical or quantum model, however it is enough when using *transfer learning* approach.

The PyTorch packages `torchvision` and `torch.utils.data` are used for loading the dataset and performing standard preliminary image operations: resize, center, crop, normalize, etc.

```
In [6]: 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 In
            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 = "data/hymenoptera_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=1
```

```

    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)

```

Let us show a batch of the test data, just to have an idea of the classification problem.

```

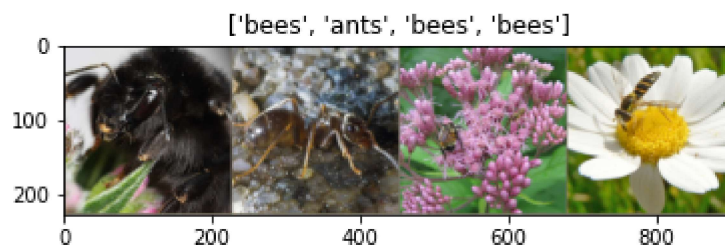
In [7]: # Get a batch of training data
inputs, classes = next(iter(dataloaders["validation"]))

# 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"]}

```



Variational quantum circuit

We first define some quantum layers that will compose the quantum circuit.

```

In [8]: def H_layer(nqubits):
    """Layer of single-qubit Hadamard gates.
    """
    for idx in range(nqubits):
        qml.Hadamard(wires=idx)

def RY_layer(w):
    """Layer of parametrized qubit rotations around the y axis.
    """
    for idx, element in enumerate(w):
        qml.RY(element, wires=idx)

```

```

def entangling_layer(nqubits):
    """Layer of CNOTs followed by another shifted layer of CNOT.
    """
    # In other words it should apply something like :
    # CNOT CNOT CNOT CNOT... CNOT
    # CNOT CNOT CNOT... CNOT
    for i in range(0, nqubits - 1, 2): # Loop over even indices: i=0,2,...N-2
       qml.CNOT(wires=[i, i + 1])
    for i in range(1, nqubits - 1, 2): # Loop over odd indices: i=1,3,...N-3
       qml.CNOT(wires=[i, i + 1])

```

Now we define the quantum circuit through the PennyLane [qnode]{title-ref} decorator .

The structure is that of a typical variational quantum circuit:

- **Embedding layer:** All qubits are first initialized in a balanced superposition of *up* and *down* states, then they are rotated according to the input parameters (local embedding).
- **Variational layers:** A sequence of trainable rotation layers and constant entangling layers is applied.
- **Measurement layer:** For each qubit, the local expectation value of the Z operator is measured. This produces a classical output vector, suitable for additional post-processing.

```

In [9]: @qml.qnode(dev, interface="torch")
def quantum_net(q_input_features, q_weights_flat):
    """
    The variational quantum circuit.
    """

    # Reshape weights
    q_weights = q_weights_flat.reshape(q_depth, n_qubits)

    # Start from state |+> , unbiased w.r.t. |0> and |1>
    H_layer(n_qubits)

    # Embed features in the quantum node
    RY_layer(q_input_features)

    # Sequence of trainable variational layers
    for k in range(q_depth):
        entangling_layer(n_qubits)
        RY_layer(q_weights[k])

    # Expectation values in the Z basis
    exp_vals = [qml.expval(qml.PauliZ(position)) for position in range(n_qubits)]
    return tuple(exp_vals)

```

Dressed quantum circuit

We can now define a custom `torch.nn.Module` representing a *dressed* quantum circuit.

This is a concatenation of:

- A classical pre-processing layer (`nn.Linear`).
- A classical activation function (`torch.tanh`).
- A constant `np.pi/2.0` scaling.
- The previously defined quantum circuit (`quantum_net`).
- A classical post-processing layer (`nn.Linear`).

The input of the module is a batch of vectors with 512 real parameters (features) and the output is a batch of vectors with two real outputs (associated with the two classes of images: *ants* and *bees*).

```
In [10]: class DressedQuantumNet(nn.Module):
    """
    Torch module implementing the *dressed* quantum net.
    """

    def __init__(self):
        """
        Definition of the *dressed* layout.
        """

        super().__init__()
        self.pre_net = nn.Linear(512, n_qubits)
        self.q_params = nn.Parameter(q_delta * torch.randn(q_depth * n_qubits))
        self.post_net = nn.Linear(n_qubits, 2)

    def forward(self, input_features):
        """
        Defining how tensors are supposed to move through the *dressed* quantum
        net.
        """

        # obtain the input features for the quantum circuit
        # by reducing the feature dimension from 512 to 4
        pre_out = self.pre_net(input_features)
        q_in = torch.tanh(pre_out) * np.pi / 2.0

        # Apply the quantum circuit to each element of the batch and append to q_out
        q_out = torch.Tensor(0, n_qubits)
        q_out = q_out.to(device)
        for elem in q_in:
            q_out_elem = quantum_net(elem, self.q_params).float().unsqueeze(0)
            q_out = torch.cat((q_out, q_out_elem))

        # return the two-dimensional prediction from the postprocessing layer
        return self.post_net(q_out)
```

Hybrid classical-quantum model

We are finally ready to build our full hybrid classical-quantum network. We follow the *transfer learning* approach:

1. First load the classical pre-trained network *ResNet18* from the `torchvision.models` zoo.
2. Freeze all the weights since they should not be trained.

3. Replace the last fully connected layer with our trainable dressed quantum circuit (`DressedQuantumNet`).

::: {note} ::: {title} Note :::

The *ResNet18* model is automatically downloaded by PyTorch and it may take several minutes (only the first time). :::

```
In [11]: model_hybrid = torchvision.models.resnet18(pretrained=True)

for param in model_hybrid.parameters():
    param.requires_grad = False

# Notice that model_hybrid.fc is the last layer of ResNet18
model_hybrid.fc = DressedQuantumNet()

# Use CUDA or CPU according to the "device" object.
model_hybrid = model_hybrid.to(device)
```

Training and results

Before training the network we need to specify the *loss* function.

We use, as usual in classification problem, the *cross-entropy* which is directly available within `torch.nn`.

```
In [12]: criterion = nn.CrossEntropyLoss()
```

We also initialize the *Adam optimizer* which is called at each training step in order to update the weights of the model.

```
In [13]: optimizer_hybrid = optim.Adam(model_hybrid.fc.parameters(), lr=step)
```

We schedule to reduce the learning rate by a factor of `gamma_lr_scheduler` every 10 epochs.

```
In [14]: exp_lr_scheduler = lr_scheduler.StepLR(
    optimizer_hybrid, step_size=10, gamma=gamma_lr_scheduler
)
```

What follows is a training function that will be called later. This function should return a trained model that can be used to make predictions (classifications).

```
In [15]: 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 train
            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_
            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]
        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

```

We are ready to perform the actual training process.

```
In [16]: model_hybrid = train_model(
        model_hybrid, criterion, optimizer_hybrid, exp_lr_scheduler, num_epochs=num_epochs
    )
```

Training started:

Phase: train Epoch: 1/3 Iter: 2/62 Batch time: 0.1528

C:\Users\cory\anaconda3\envs\Quantum\lib\site-packages\torch\autograd_init__.py:175: UserWarning: Casting complex values to real discards the imaginary part (Triggered internally at C:\actions-runner_work\pytorch\pytorch\builder\windows\pytorch\aten\src\ATen\native\Copy.cpp:239.)

allow_unreachable=True, accumulate_grad=True) # Calls into the C++ engine to run the backward pass

Phase: train Epoch: 1/3 Loss: 0.6990 Acc: 0.5246

Phase: validation Epoch: 1/3 Loss: 0.6429 Acc: 0.6536

Phase: train Epoch: 2/3 Loss: 0.6134 Acc: 0.7008

Phase: validation Epoch: 2/3 Loss: 0.5389 Acc: 0.8235

Phase: train Epoch: 3/3 Loss: 0.5652 Acc: 0.7418

Phase: validation Epoch: 3/3 Loss: 0.4484 Acc: 0.8497

Training completed in 0m 51s

Best test loss: 0.4484 | Best test accuracy: 0.8497

Visualizing the model predictions

We first define a visualization function for a batch of test data.

```
In [17]: def visualize_model(model, num_images=6, fig_name="Predictions"):
```

```

images_so_far = 0
_fig = plt.figure(fig_name)
model.eval()
with torch.no_grad():
    for _i, (inputs, labels) in enumerate(dataloaders["validation"]):
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        for j in range(inputs.size()[0]):
            images_so_far += 1
            ax = plt.subplot(num_images // 2, 2, images_so_far)
            ax.axis("off")
            ax.set_title("{}".format(class_names[preds[j]]))
            imshow(inputs.cpu().data[j])
        if images_so_far == num_images:
            return

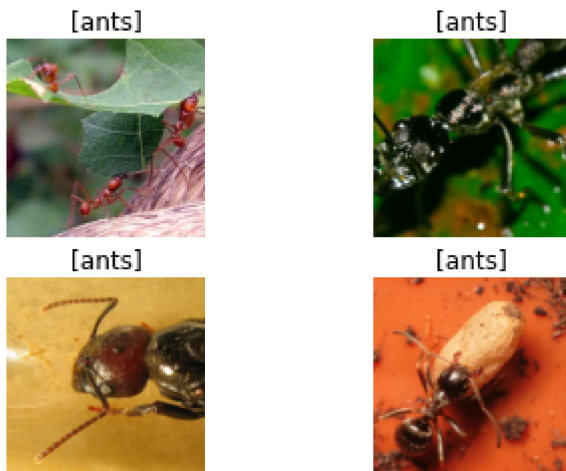
```

Finally, we can run the previous function to see a batch of images with the corresponding predictions.

```

In [18]: visualize_model(model_hybrid, num_images=batch_size)
plt.show()

```



References

[1] Andrea Mari, Thomas R. Bromley, Josh Izaac, Maria Schuld, and Nathan Killoran. *Transfer learning in hybrid classical-quantum neural networks*. arXiv:1912.08278 (2019).

[2] Rajat Raina, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y Ng. *Self-taught learning: transfer learning from unlabeled data*. Proceedings of the 24th International Conference on Machine Learning*, 759--766 (2007).

[3] Kaiming He, Xiangyu Zhang, Shaoqing ren and Jian Sun. *Deep residual learning for image recognition*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 770-778 (2016).

[4] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Carsten Blank, Keri McKiernan, and Nathan Killoran. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. arXiv:1811.04968 (2018).

In []: