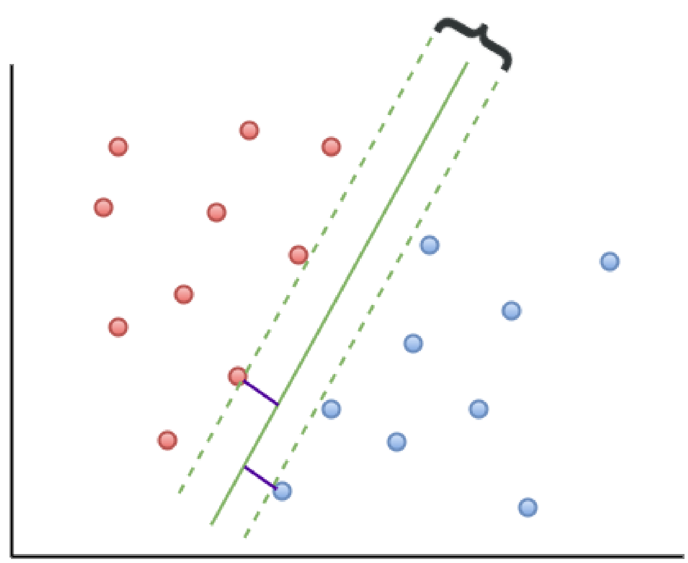
Multiclass margin classifier

Published April 9, 2020. Last updated August 5, 2024.

In this tutorial, we show how to use the PyTorch interface for PennyLane to implement a multiclass variational classifier. We consider the iris database from UCI, which has 4 features and 3 classes. We use multiple one-vs-all classifiers with a margin loss (see Multiclass Linear SVM) to classify data. Each classifier is implemented on an individual variational circuit, whose architecture is inspired by Farhi and Neven (2018) as well as Schuld et al. (2018).



Initial Setup

PENNYLANE

Our feature size is 4, and we will use amplitude embedding. This means that each possible amplitude (in the computational basis) will correspond to a single feature. With 2 qubits (wires), there are 4 possible states, and as such, we can encode a feature vector of size 4.

```
import pennylane as qml
import torch
import numpy as np
from torch.autograd import Variable
import torch.optim as optim

np.random.seed(0)
torch.manual_seed(0)

num_classes = 3
margin = 0.15
feature_size = 4
batch_size = 10
lr_adam = 0.01
train_split = 0.75
# the number of the required qubits is calculated from the number of features
num_qubits = int(np.ceil(np.log2(feature_size)))
num_layers = 6
total_iterations = 100

dev = qml.device("default.qubit", wires=num_qubits)
```

Quantum Circuit

We first create the layer that will be repeated in our variational quantum circuits. It consists of rotation gates for each qubit, followed by entangling/CNOT gates

PENNYLANE

```
    for i in range(num_qubits):
        qml.Rot(W[i, 0], W[i, 1], W[i, 2], wires=i)
    for j in range(num_qubits - 1):
        qml.CNOT(wires=[j, j + 1])
    if num_qubits >= 2:
        # Apply additional CNOT to entangle the last with the first qubit
        qml.CNOT(wires=[num_qubits - 1, 0])
```

We now define the quantum nodes that will be used. As we are implementing our multiclass classifier as multiple one-vs-all classifiers, we will use 3 QNodes, each representing one such classifier. That is, `circuit1` classifies if a sample belongs to class 1 or not, and so on. The circuit architecture for all nodes are the same. We use the PyTorch interface for the QNodes. Data is embedded in each circuit using amplitude embedding.

Note

For demonstration purposes we are using a very simple circuit here. You may find that other choices, for example more elaborate measurements, increase the power of the classifier.

```
def circuit(weights, feat=None):
    qml.AmplitudeEmbedding(feat, range(num_qubits), pad_with=0.0, normalize=True)

    for W in weights:
        layer(W)

    return qml.expval(qml.PauliZ(0))


qnodes = []
for iq in range(num_classes):
    qnode = qml.QNode(circuit, dev, interface="torch")
    qnodes.append(qnode)
```

PENNYLANE

bias term are optimized.

```
def variational_classifier(q_circuit, params, feat):
    weights = params[0]
    bias = params[1]
    return q_circuit(weights, feat=feat) + bias
```

Loss Function

Implementing multiclass classifiers as a number of one-vs-all classifiers generally evokes using the margin loss. The output of the i th classifier, c_i on input x is interpreted as a score, s_i between [-1,1]. More concretely, we have:

$$s_i = c_i(x; \theta)$$

The multiclass margin loss attempts to ensure that the score for the correct class is higher than that of incorrect classes by some margin. For a sample (x, y) where y denotes the class label, we can analytically express the mutliclass loss on this sample as:

$$L(x, y) = \sum_{j \neq y} \max\left(0, s_j - s_y + \Delta)\right)$$

where Δ denotes the margin. The margin parameter is chosen as a hyperparameter. For more information, see Multiclass Linear SVM.

PENNYLANE

```
    loss = 0
    num_samples = len(true_labels)
    for i, feature_vec in enumerate(feature_vecs):
        # Compute the score given to this sample by the classifier corresponding
        # true label. So for a true label of 1, get the score computed by classif
        # which distinguishes between "class 1" or "not class 1".
        s_true = variational_classifier(
            q_circuits[int(true_labels[i])],
            (all_params[0][int(true_labels[i])], all_params[1][int(true_labels[i]
            feature_vec,
        )
        s_true = s_true.float()
        li = 0

        # Get the scores computed for this sample by the other classifiers
        for j in range(num_classes):
            if j != int(true_labels[i]):
                s_j = variational_classifier(
                    q_circuits[j], (all_params[0][j], all_params[1][j]), feature_
                )
                s_j = s_j.float()
                li += torch.max(torch.zeros(1).float(), s_j - s_true + margin)
        loss += li

    return loss / num_samples
```

Classification Function

Next, we use the learned models to classify our samples. For a given sample, compute the score given to it by classifier i, which quantifies how likely it is that this sample belongs to class i. For each sample, return the class with the highest score.

PENNYLANE

```
        predicted_labels = []
        for i, feature_vec in enumerate(feature_vecs):
            scores = np.zeros(num_classes)
            for c in range(num_classes):
                score = variational_classifier(
                    q_circuits[c], (all_params[0][c], all_params[1][c]), feature_vec
                )
                scores[c] = float(score)
            pred_class = np.argmax(scores)
            predicted_labels.append(pred_class)
        return predicted_labels


    def accuracy(labels, hard_predictions):
        loss = 0
        for l, p in zip(labels, hard_predictions):
            if torch.abs(l - p) < 1e-5:
                loss = loss + 1
        loss = loss / labels.shape[0]
        return loss
```

Data Loading and Processing

Now we load in the iris dataset and normalize the features so that the sum of the feature elements squared is 1 (ℓ_2 norm is 1).

PENNYLANE

```
    data = np.loadtxt("../_static/demonstration_assets/multiclass_classification/
    X = torch.tensor(data[:, 0:feature_size])
    print("First X sample, original  :", X[0])

    # normalize each input
    normalization = torch.sqrt(torch.sum(X ** 2, dim=1))
    X_norm = X / normalization.reshape(len(X), 1)
    print("First X sample, normalized:", X_norm[0])

    Y = torch.tensor(data[:, -1])
    return X, Y


# Create a train and test split.
def split_data(feature_vecs, Y):
    num_data = len(Y)
    num_train = int(train_split * num_data)
    index = np.random.permutation(range(num_data))
    feat_vecs_train = feature_vecs[index[:num_train]]
    Y_train = Y[index[:num_train]]
    feat_vecs_test = feature_vecs[index[num_train:]]
    Y_test = Y[index[num_train:]]
    return feat_vecs_train, feat_vecs_test, Y_train, Y_test
```

Training Procedure

In the training procedure, we begin by first initializing randomly the parameters we wish to learn (variational circuit weights and classical bias). As these are the variables we wish to optimize, we set the `requires_grad` flag to `True`. We use minibatch training—the average loss for a batch of samples is computed, and the optimization step is based on this. Total training time with the default parameters is roughly 15 minutes.

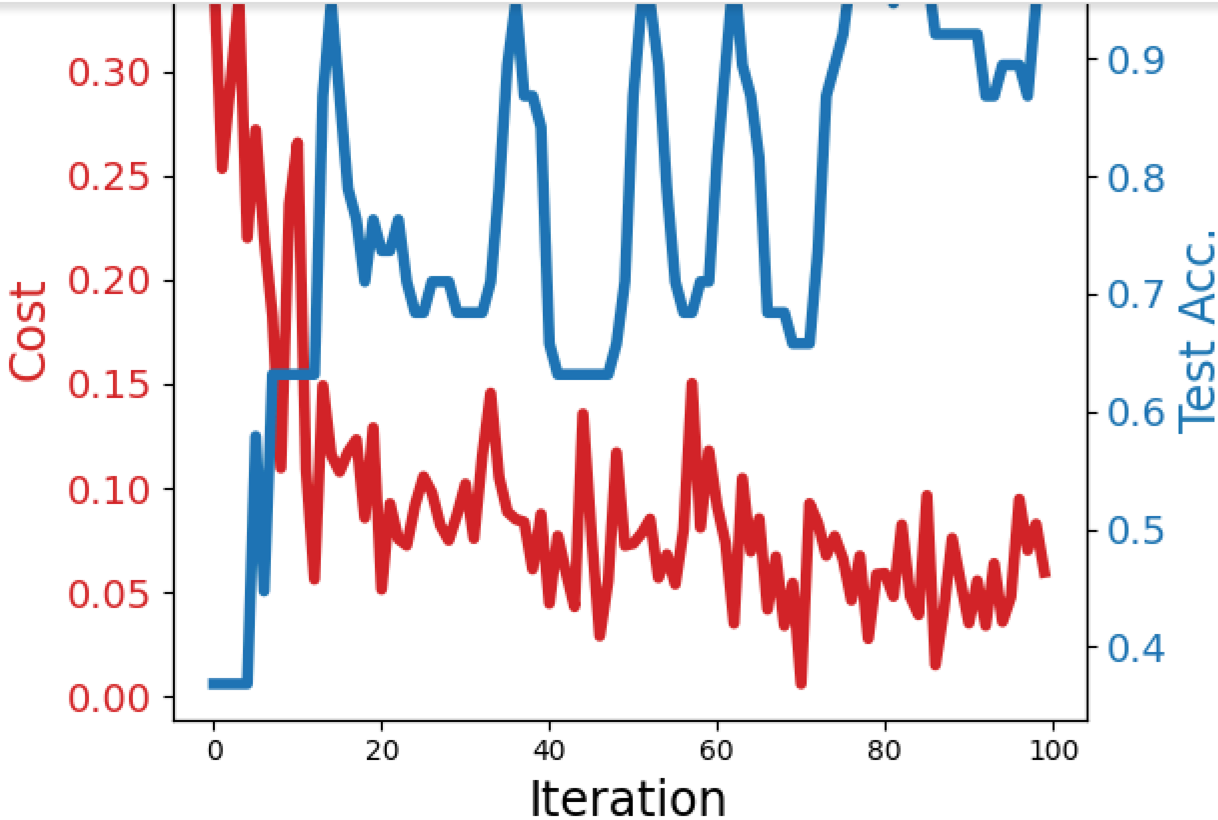
PENNYLANE

```
    num_data = Y.shape[0]
    feat_vecs_train, feat_vecs_test, Y_train, Y_test = split_data(features, Y)
    num_train = Y_train.shape[0]
    q_circuits = qnodes

    # Initialize the parameters
    all_weights = [
        Variable(0.1 * torch.randn(num_layers, num_qubits, 3), requires_grad=True
        for i in range(num_classes)
    ]
    all_bias = [Variable(0.1 * torch.ones(1), requires_grad=True) for i in range(
    optimizer = optim.Adam(all_weights + all_bias, lr=lr_adam)
    params = (all_weights, all_bias)
    print("Num params: ", 3 * num_layers * num_qubits * 3 + 3)

    costs, train_acc, test_acc = [], [], []

    # train the variational classifier
    for it in range(total_iterations):
        batch_index = np.random.randint(0, num_train, (batch_size,))
        feat_vecs_train_batch = feat_vecs_train[batch_index]
        Y_train_batch = Y_train[batch_index]

        optimizer.zero_grad()
        curr_cost = multiclass_svm_loss(q_circuits, params, feat_vecs_train_batch
        curr_cost.backward()
        optimizer.step()

        # Compute predictions on train and validation set
        predictions_train = classify(q_circuits, params, feat_vecs_train, Y_train
        predictions_test = classify(q_circuits, params, feat_vecs_test, Y_test)
        acc_train = accuracy(Y_train, predictions_train)
        acc_test = accuracy(Y_test, predictions_test)

        print(
            "Iter: {:5d} | Cost: {:0.7f} | Acc train: {:0.7f} | Acc test: {:0.7f}
            "".format(it + 1, curr_cost.item(), acc_train, acc_test)
        )

        costs.append(curr_cost.item())
```

PENNYLANE

```
    return costs, train_acc, test_acc


# We now run our training algorithm and plot the results. Note that
# for plotting, the matplotlib library is required

features, Y = load_and_process_data()
costs, train_acc, test_acc = training(features, Y)

import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()
iters = np.arange(0, total_iterations, 1)
colors = ["tab:red", "tab:blue"]
ax1.set_xlabel("Iteration", fontsize=17)
ax1.set_ylabel("Cost", fontsize=17, color=colors[0])
ax1.plot(iters, costs, color=colors[0], linewidth=4)
ax1.tick_params(axis="y", labelsize=14, labelcolor=colors[0])

ax2 = ax1.twinx()
ax2.set_ylabel("Test Acc.", fontsize=17, color=colors[1])
ax2.plot(iters, test_acc, color=colors[1], linewidth=4)

ax2.tick_params(axis="x", labelsize=14)
ax2.tick_params(axis="y", labelsize=14, labelcolor=colors[1])

plt.grid(False)
plt.tight_layout()
plt.show()
```

PENNYLANE



Out:

```
First X sample, original  : tensor([5.1000, 3.5000, 1.4000, 0.2000], dty
First X sample, normalized: tensor([0.8038, 0.5516, 0.2206, 0.0315], dty
Num params:  111
Iter:     1 | Cost: 0.3475123 | Acc train: 0.3214286 | Acc test: 0.36842
Iter:     2 | Cost: 0.2533208 | Acc train: 0.3214286 | Acc test: 0.36842
Iter:     3 | Cost: 0.2943569 | Acc train: 0.3214286 | Acc test: 0.36842
Iter:     4 | Cost: 0.3344342 | Acc train: 0.3214286 | Acc test: 0.36842
Iter:     5 | Cost: 0.2200930 | Acc train: 0.3214286 | Acc test: 0.36842
Iter:     6 | Cost: 0.2718903 | Acc train: 0.4910714 | Acc test: 0.57894
Iter:     7 | Cost: 0.2201053 | Acc train: 0.4821429 | Acc test: 0.44736
Iter:     8 | Cost: 0.1825284 | Acc train: 0.6785714 | Acc test: 0.63157
Iter:     9 | Cost: 0.1096408 | Acc train: 0.6785714 | Acc test: 0.63157
Iter:    10 | Cost: 0.2361710 | Acc train: 0.6785714 | Acc test: 0.63157
Iter:    11 | Cost: 0.2656708 | Acc train: 0.6785714 | Acc test: 0.63157
Iter:    12 | Cost: 0.1090595 | Acc train: 0.6785714 | Acc test: 0.63157
Iter:    13 | Cost: 0.0562117 | Acc train: 0.6875000 | Acc test: 0.63157
Iter:    14 | Cost: 0.1491400 | Acc train: 0.7946429 | Acc test: 0.86842
```