In [10]:
```
import numpy as np
import matplotlib.pyplot as plt

# Importing standard Qiskit libraries
#from qiskit import QuantumCircuit, transpile, Aer, IBMQ
import qiskit
from qiskit import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *

# For Pytorch
import torch
from torch.autograd import Function
from torchvision import datasets, transforms
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

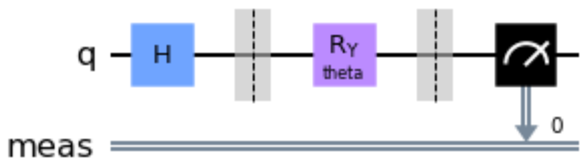
```
ibmqfactory.load_account:WARNING:2021-07-11 06:46:28,261: Credentials
are already in use. The existing account in the session will be repla
ced.
```

In [11]:
```
class QuantumCircuit:
    """
    This class provides a simple interface for interaction
    with the quantum circuit
    """

    def __init__(self, n_qubits, backend, shots):
        # --- Circuit definition ---
        self._circuit = qiskit.QuantumCircuit(n_qubits)

        all_qubits = [i for i in range(n_qubits)]
        self.theta = qiskit.circuit.Parameter('theta')

        self._circuit.h(all_qubits)
        self._circuit.barrier()
        self._circuit.ry(self.theta, all_qubits)

        self._circuit.measure_all()
        # ---------------------------

        self.backend = backend
        self.shots = shots

    def run(self, thetas):
        t_qc = transpile(self._circuit,
                         self.backend)
        qobj = assemble(t_qc,
                        shots=self.shots,
                        parameter_binds = [{self.theta: theta} for thet
a in thetas])
        job = self.backend.run(qobj)
        result = job.result().get_counts()

        counts = np.array(list(result.values()))
        states = np.array(list(result.keys())).astype(float)

        # Compute probabilities for each state
        probabilities = counts / self.shots
        # Get state expectation
        expectation = np.sum(states * probabilities)

        return np.array([expectation])
```

In [12]:
```
#import qiskit
#from qiskit import QuantumCircuit, transpile, Aer

simulator = qiskit.Aer.get_backend('aer_simulator')

circuit = QuantumCircuit(1, simulator, 100)
print('Expected value for rotation pi {}'.format(circuit.run([np.p
i])[0]))
circuit._circuit.draw()
```

```
Expected value for rotation pi 0.58
```

Out[12]:

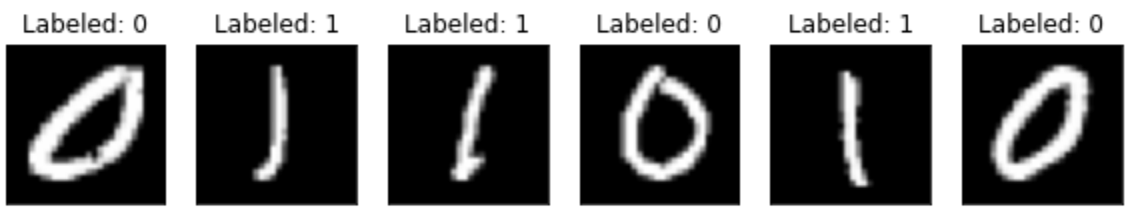


In [13]:
```
class HybridFunction(Function):
    """ Hybrid quantum - classical function definition """

    @staticmethod
    def forward(ctx, input, quantum_circuit, shift):
        """ Forward pass computation """
        ctx.shift = shift
        ctx.quantum_circuit = quantum_circuit

        expectation_z = ctx.quantum_circuit.run(input[0].tolist())
        result = torch.tensor([expectation_z])
        ctx.save_for_backward(input, result)

        return result

    @staticmethod
    def backward(ctx, grad_output):
        """ Backward pass computation """
        input, expectation_z = ctx.saved_tensors
        input_list = np.array(input.tolist())

        shift_right = input_list + np.ones(input_list.shape) * ctx.shif
t
        shift_left = input_list - np.ones(input_list.shape) * ctx.shift

        gradients = []
        for i in range(len(input_list)):
            expectation_right = ctx.quantum_circuit.run(shift_right[i])
            expectation_left  = ctx.quantum_circuit.run(shift_left[i])

            gradient = torch.tensor([expectation_right]) - torch.tensor
([expectation_left])
            gradients.append(gradient)
        gradients = np.array([gradients]).T
        return torch.tensor([gradients]).float() * grad_output.float(),
None, None

class Hybrid(nn.Module):
    """ Hybrid quantum - classical layer definition """

    def __init__(self, backend, shots, shift):
        super(Hybrid, self).__init__()
        self.quantum_circuit = QuantumCircuit(1, backend, shots)
        self.shift = shift

    def forward(self, input):
        return HybridFunction.apply(input, self.quantum_circuit, self.s
hift)
```

In [14]:
```
# Concentrating on the first 100 samples
n_samples = 100

X_train = datasets.MNIST(root='./data', train=True, download=True,
                         transform=transforms.Compose([transforms.ToTen
sor()]))

# Leaving only labels 0 and 1
idx = np.append(np.where(X_train.targets == 0)[0][:n_samples],
                np.where(X_train.targets == 1)[0][:n_samples])

X_train.data = X_train.data[idx]
X_train.targets = X_train.targets[idx]

train_loader = torch.utils.data.DataLoader(X_train, batch_size=1, shuff
le=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.
gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIS
T/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.
gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIS
T/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.g
z to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST
/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.g
z to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST
/raw
Processing...

/opt/conda/lib/python3.8/site-packages/torchvision/datasets/mnist.py:
479: UserWarning: The given NumPy array is not writeable, and PyTorch
does not support non-writeable tensors. This means you can write to t
he underlying (supposedly non-writeable) NumPy array using the tenso
r. You may want to copy the array to protect its data or make it writ
eable before converting it to a tensor. This type of warning will be
suppressed for the rest of this program. (Triggered internally at  /p
ytorch/torch/csrc/utils/tensor_numpy.cpp:143.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)

Done!
```

In [15]:
```
n_samples_show = 6

data_iter = iter(train_loader)
fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10, 3))

while n_samples_show > 0:
    images, targets = data_iter.__next__()

    axes[n_samples_show - 1].imshow(images[0].numpy().squeeze(), cmap='gray')
    axes[n_samples_show - 1].set_xticks([])
    axes[n_samples_show - 1].set_yticks([])
    axes[n_samples_show - 1].set_title("Labeled: {}".format(targets.item()))

    n_samples_show -= 1
```



In [16]:
```
n_samples = 50

X_test = datasets.MNIST(root='./data', train=False, download=True,
                        transform=transforms.Compose([transforms.ToTensor()]))

idx = np.append(np.where(X_test.targets == 0)[0][:n_samples],
                np.where(X_test.targets == 1)[0][:n_samples])

X_test.data = X_test.data[idx]
X_test.targets = X_test.targets[idx]

test_loader = torch.utils.data.DataLoader(X_test, batch_size=1, shuffle=True)
```

In [17]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.dropout = nn.Dropout2d()
        self.fc1 = nn.Linear(256, 64)
        self.fc2 = nn.Linear(64, 1)
        self.hybrid = Hybrid(qiskit.Aer.get_backend('aer_simulator'), 1
00, np.pi / 2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = self.dropout(x)
        x = x.view(1, -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.hybrid(x)
        return torch.cat((x, 1 - x), -1)
```

In [18]:

```
model = Net()
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_func = nn.NLLLoss()

epochs = 20
loss_list = []

model.train()
for epoch in range(epochs):
    total_loss = []
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        # Forward pass
        output = model(data)
        # Calculating loss
        loss = loss_func(output, target)
        # Backward pass
        loss.backward()
        # Optimize the weights
        optimizer.step()

        total_loss.append(loss.item())
    loss_list.append(sum(total_loss)/len(total_loss))
    print('Training [{:.0f}%]\tLoss: {:.4f}'.format(
        100. * (epoch + 1) / epochs, loss_list[-1]))
```

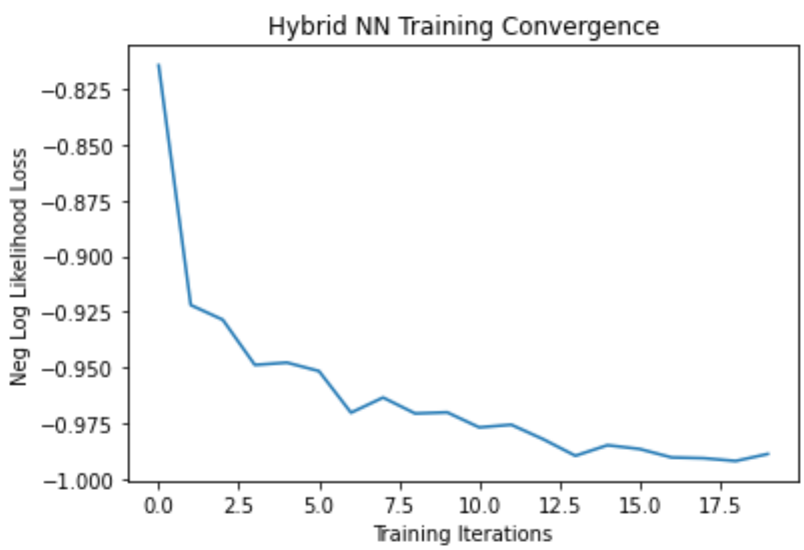
```
<ipython-input-13-625640256c98>:32: FutureWarning: The input object o
f type 'Tensor' is an array-like implementing one of the correspondin
g protocols (`__array__`, `__array_interface__` or `__array_struct__
`); but not a sequence (or 0-D). In the future, this object will be c
oerced as if it was first converted using `np.array(obj)`. To retain
the old behaviour, you have to either modify the type 'Tensor', or as
sign to an empty array created with `np.empty(correct_shape, dtype=ob
ject)`.
  gradients = np.array([gradients]).T

Training [5%]   Loss: -0.8143
Training [10%]  Loss: -0.9220
Training [15%]  Loss: -0.9286
Training [20%]  Loss: -0.9489
Training [25%]  Loss: -0.9479
Training [30%]  Loss: -0.9516
Training [35%]  Loss: -0.9703
Training [40%]  Loss: -0.9635
Training [45%]  Loss: -0.9707
Training [50%]  Loss: -0.9702
Training [55%]  Loss: -0.9769
Training [60%]  Loss: -0.9757
Training [65%]  Loss: -0.9823
Training [70%]  Loss: -0.9897
Training [75%]  Loss: -0.9850
Training [80%]  Loss: -0.9866
Training [85%]  Loss: -0.9905
Training [90%]  Loss: -0.9908
Training [95%]  Loss: -0.9921
Training [100%] Loss: -0.9889
```

In [19]:

```
plt.plot(loss_list)
plt.title('Hybrid NN Training Convergence')
plt.xlabel('Training Iterations')
plt.ylabel('Neg Log Likelihood Loss')
```

Out[19]:

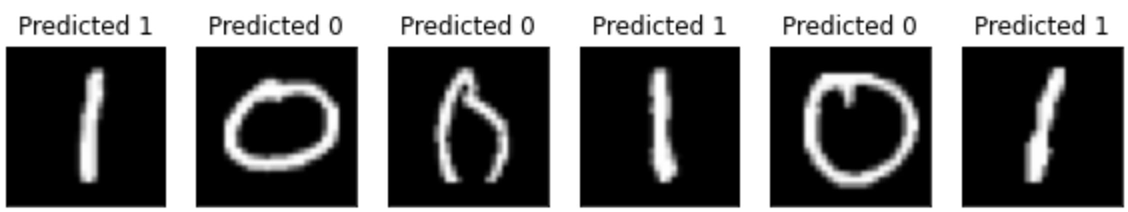
```
Text(0, 0.5, 'Neg Log Likelihood Loss')
```



In [20]:
```
model.eval()
with torch.no_grad():

    correct = 0
    for batch_idx, (data, target) in enumerate(test_loader):
        output = model(data)

        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

        loss = loss_func(output, target)
        total_loss.append(loss.item())

    print('Performance on test data:\n\tLoss: {:.4f}\n\tAccuracy: {:.1
f}%'.format(
        sum(total_loss) / len(total_loss),
        correct / len(test_loader) * 100)
        )
```

```
Performance on test data:
        Loss: -0.9862
        Accuracy: 100.0%
```

In [21]:
```
n_samples_show = 6
count = 0
fig, axes = plt.subplots(nrows=1, ncols=n_samples_show, figsize=(10,
3))

model.eval()
with torch.no_grad():
    for batch_idx, (data, target) in enumerate(test_loader):
        if count == n_samples_show:
            break
        output = model(data)

        pred = output.argmax(dim=1, keepdim=True)

        axes[count].imshow(data[0].numpy().squeeze(), cmap='gray')

        axes[count].set_xticks([])
        axes[count].set_yticks([])
        axes[count].set_title('Predicted {}'.format(pred.item()))

        count += 1
```



In [ ]:
```
#Hybrid quantum-classical Neural Networks with PyTorch and Qiskit,execu
ted by Bhadale IT
```