Solutions 2

Jumping Rivers

Concentric circles: revisited

- Fit a multi layer percepton model to the concentric circles data
  from earlier. You should be able to get pretty good performance
  without having to do any manual feature engineering.

```python
import jrpytorch
import torch.nn as nn
import torch
X, y = jrpytorch.datasets.load_circles()

class mlp(nn.Module):
  def __init__(self):
    super(mlp,self).__init__()
    self.lin1 = nn.Linear(in_features=2, out_features=50)
    self.lin2 = nn.Linear(50,25)
    self.lin3 = nn.Linear(25,1)

  def forward(self,x):
    x = torch.relu(self.lin1(x))
    x = torch.relu(self.lin2(x))
    x = torch.sigmoid(self.lin3(x))
    return x

model = mlp()
loss_fn = nn.BCELoss()
optimiser = torch.optim.SGD(model.parameters(), lr = 0.1, momentum = 0.3)

x_tensor = torch.tensor(X).float()
y_tensor = torch.tensor(y.reshape(-1,1)).float()

for epoch in range(2000):
  optimiser.zero_grad()
  output = model(x_tensor)
  loss = loss_fn(output,y_tensor)
  loss.backward()
  optimiser.step()
```

It can be difficult to get an idea of what your network predictions
look like, especially in high dimensional spaces. Fortunately visualisa-
tion in two dimension is trivial. One of the advantages of being able
to watch your prediction space update in real time is the ability to

gain intuition on how changes to the model structure or additional hyperparameters affects the model fitting process and final output. To that end the **jrpytorch** package has a `vis_binary_classification` classification function. This function is really only intended for visualisation with a 2 dimensional input space mapping to a binary label as visualisation of larger dimensions becomes tricky.

- Try fitting your model to the concentric circles using the visualiser, for example

```
import jrpytorch
X,y = jrpytorch.datasets.load_circles()
jrpytorch.vis_binary_classification(model, loss_fn, optimiser, X, y)
```

If you are using a Jupyter notebook, you may need to use matplotlib magic to properly register the backend to view the plots updating as we train. To do so, add

```
%matplotlib
```

at the top of your notebook.

- Explore changing your model architecture and the optimiser hyperparameters, how do they affect fit and convergence.

Boston housing

The code below will set you up with a training and test set for the boston housing data.

```
from sklearn.datasets import load_boston
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

boston = load_boston()
X, y = boston.data, boston.target.reshape(-1,1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Try fitting a multi layered perceptron model to the boston housing example, how does changing the number and size of layers affect the ability to find a good model.

- Investigate whether different activation functions have much bearing on the outcome

- How do results differ between an identity and softplus activation at the final layer? Do both of these activations make sense?

```python
# example implementation
import torch
import torch.nn as nn
import torch.nn.functional as F


class mlp(nn.Module):
  def __init__(self):
    super(mlp,self).__init__()
    self.lin1 = nn.Linear(13,10)
    self.lin2 = nn.Linear(10,1)

  def forward(self,x):
    x = torch.tanh(self.lin1(x))
    return F.softplus(self.lin2(x))



def train(model, loss_fn, optimiser, X_train, X_test, y_train, y_test, epochs = 1000):
  X_train = torch.tensor(X_train).float()
  X_test = torch.tensor(X_test).float()
  y_train = torch.tensor(y_train).float()
  y_test = torch.tensor(y_test).float()
  log = {
    'train': [],
    'test': []
  }
  for epoch in range(epochs):
    model.train()
    optimiser.zero_grad()
    train_output = model(X_train)
    train_loss = loss_fn(train_output,y_train)
    log['train'].append(train_loss.item())
    train_loss.backward()
    optimiser.step()
    model.eval()
    with torch.no_grad():
      test_output = model(X_test)
      test_loss = loss_fn(test_output,y_test)
      log['test'].append(test_loss.item())
    if epoch % 100 == 99:
      print('%d: %.3f %.3f' % (epoch+1,train_loss,test_loss))
  return log
```

```python
model = mlp()
optimiser = torch.optim.SGD(model.parameters(), lr=0.1)
loss_fn = nn.MSELoss()
output = train(model,loss_fn,optimiser,X_train,X_test,y_train,y_test)
```