alfred-ajay-aureate_rajakumar_aar653_A2_code

March 6, 2020

[0]:
```
from matplotlib import pyplot as plt
import numpy as np
import torch

def set_default(figsize=(10, 10)):
    plt.style.use(['dark_background', 'bmh'])
    plt.rc('axes', facecolor='k')
    plt.rc('figure', facecolor='k')
    plt.rc('figure', figsize=figsize)

def plot_data(X, y, d=0, auto=False, zoom=1):
    plt.scatter(X.numpy()[:, 0], X.numpy()[:, 1], c=y, s=20, cmap=plt.cm.
 ↪Spectral)
    plt.axis('square')
    plt.axis(np.array((-1.1, 1.1, -1.1, 1.1)) * zoom)
    if auto is True: plt.axis('equal')
    plt.axis('off')

    _m, _c = 0, '.15'
    plt.axvline(0, ymin=_m, color=_c, lw=1, zorder=0)
    plt.axhline(0, xmin=_m, color=_c, lw=1, zorder=0)

def plot_model(X, y, model):
    mesh = np.arange(-1.1, 1.1, 0.01)
    xx, yy = np.meshgrid(mesh, mesh)
    with torch.no_grad():
        data = torch.from_numpy(np.vstack((xx.reshape(-1), yy.reshape(-1))).T).
 ↪float()
        Z = model(data).detach()
    Z = np.argmax(Z, axis=1).reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.3)
    plot_data(X, y)

def show_scatterplot(X, colors, title=''):
    colors = colors.numpy()
    X = X.numpy()
    plt.figure()
```

```python
    plt.axis('equal')
    plt.scatter(X[:, 0], X[:, 1], c=colors, s=30)
    # plt.grid(True)
    plt.title(title)
    plt.axis('off')

def plot_bases(bases, width=0.04):
    bases[2:] -= bases[:2]
    plt.arrow(*bases[0], *bases[2], width=width, color=(1,0,0), zorder=10, alpha=1., length_includes_head=True)
    plt.arrow(*bases[1], *bases[3], width=width, color=(0,1,0), zorder=10, alpha=1., length_includes_head=True)
```

[0]:
```python
# Import dependencies
import torch
import torch.nn as nn
#from plot_lib import set_default, show_scatterplot, plot_bases
import matplotlib.pyplot as plt
import random
import numpy as np
```

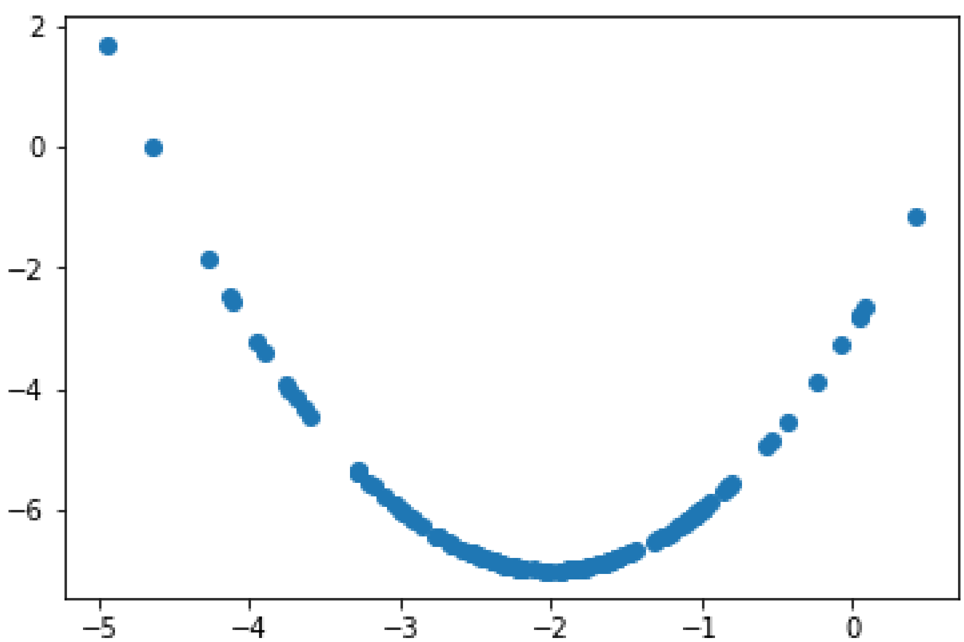
[0]:
```python
# Set up your device
cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if cuda else "cpu")
```

[0]:
```python
# Set up random seed to 1008. Do not change the random seed.
# Yes, these are all necessary when you run experiments!
seed = 1008
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if cuda:
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
```

[0]:
```python
# Define data generating functions
def quadratic_data_generator(data_size):
    # f(x) = y = x^2 + 4x - 3
    # generate an input tensor of size data_size with torch.randn
    x = torch.randn(data_size, 1) - 2
    x = x.to(device)
    # TODO
    '''
    y = ...
    '''
```

```
    y = x*x + 4*x - 3


    # placeholder
#    y = torch.ones(data_size,1)
    return x,y

def cubic_data_generator(data_size=100):
    # f(x) = y = x^3 + 4x^2 - 3
    # generate an input tensor of size data_size with torch.randn
    x = torch.randn(data_size, 1) - 2
    x = x.to(device)
    # TODO
    '''
    y = ...
    '''

    y = x*x*x + 4*x*x - 3
    # placeholder
#    y = torch.ones(data_size,1)
    return x, y
```

[86]:
```
# Generate the data with 128 datapoints
x, y = quadratic_data_generator(128)
plt.scatter(x,y)
plt.show()
```



[0]:
```python
# Define a Linear Classifier with a single linear layer and no non-linearity
# (no hidden layer)
class Linear_0H(nn.Module):
    def __init__(self):
        super().__init__()
#        super(baseclass, self).__init__()

        # TODO
#        self.classifer = None
        self.classifier = torch.nn.Linear(1,1)

    def forward(self, x):
        return self.classifier(x)
```

[0]:
```python
# Define a Linear Classifier with a single hidden layer of size 5 and ReLU␣
↪non-linearity
class Linear_1H(nn.Module):
    def __init__(self):
        super().__init__()
#        super(baseclass, self).__init__()

        # TODO
#        self.classifer = None
        self.classifier1 = torch.nn.Linear(1,5)
        self.classifier2 = torch.nn.Linear(5,1)

    def forward(self, x):
#        return self.classifier(x)
        return self.classifier2(nn.functional.relu(self.classifier1(x)))
```

[0]:
```python
# Define a Linear Classifier with a two hidden layers of size 5 and ReLU␣
↪non-linearity
class Linear_2H(nn.Module):
    def __init__(self):
        super().__init__()
#        super(baseclass, self).__init__()

        # TODO
#        self.classifer = None
        self.classifier1 = torch.nn.Linear(1,5)
        self.classifier2 = torch.nn.Linear(5,5)
        self.classifier3 = torch.nn.Linear(5,1)

    def forward(self, x):
#        return self.classifier(x)
```

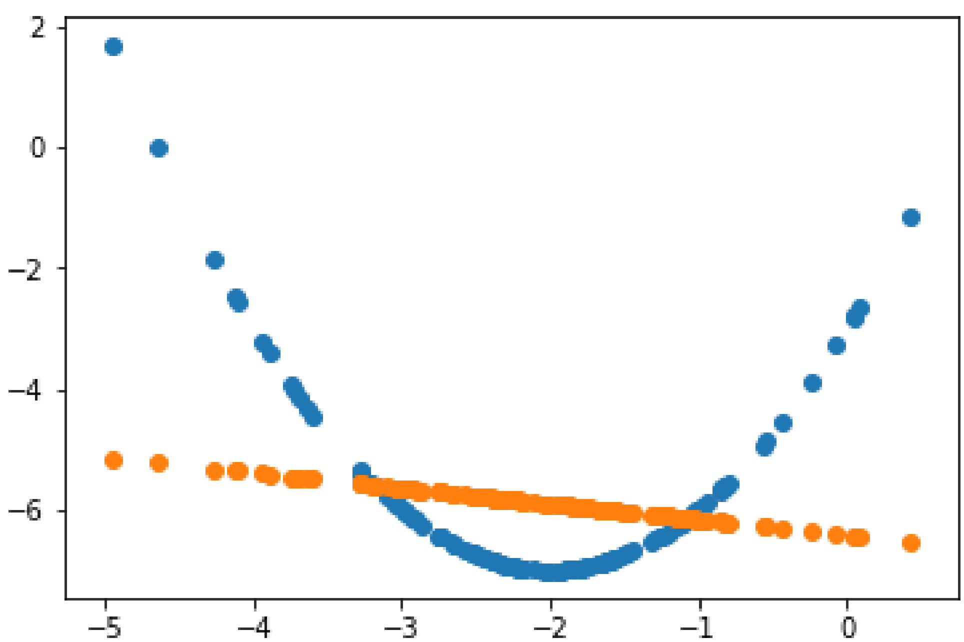
```
        return self.classifier3(nn.functional.relu(self.classifier2(nn.
↪functional.relu(self.classifier1(x)))))
```

[0]:
```
'''
TODO: Training function

Hint: look at some example pytorch tutorials to learn how to
    - initialize optimizers
    - zero gradient
    - backprop the loss
    - step the gradient


Note: This is full batch. We compute forward on whole x,y.
No need for dataloaders nor loop over batches.
Just pass all of x to model's forward pass.
'''
def train(model, epochs, x, y):

    # Set model to training mode
    model.train()

    # Define MSE loss function
#     criterion = None
    criterion = nn.MSELoss()

    # TODO: Define the SGD optimizer with learning rate 0.01
#     optimizer = None
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-5)

    for epoch in range(epochs):

        # TODO: Forward data through model to predict y
#         y_pred = None
        y_pred = model(x)

        # TODO: Compute loss in terms of predicted and true y
#         loss = None
        loss = criterion(y_pred, y)

        # TODO: Zero gradient
        optimizer.zero_grad()

        # TODO: call backward on loss
        loss.backward()

        # TODO: step the optimizer
```

```
        optimizer.step()

        # every 100 epochs, print
        if (epoch+1) % 100 == 0:
            print('Epoch {} loss: {}'.format(epoch+1, loss.item()))

    # return y_pred without gradient information, for plotting
    return y_pred.detach()
```

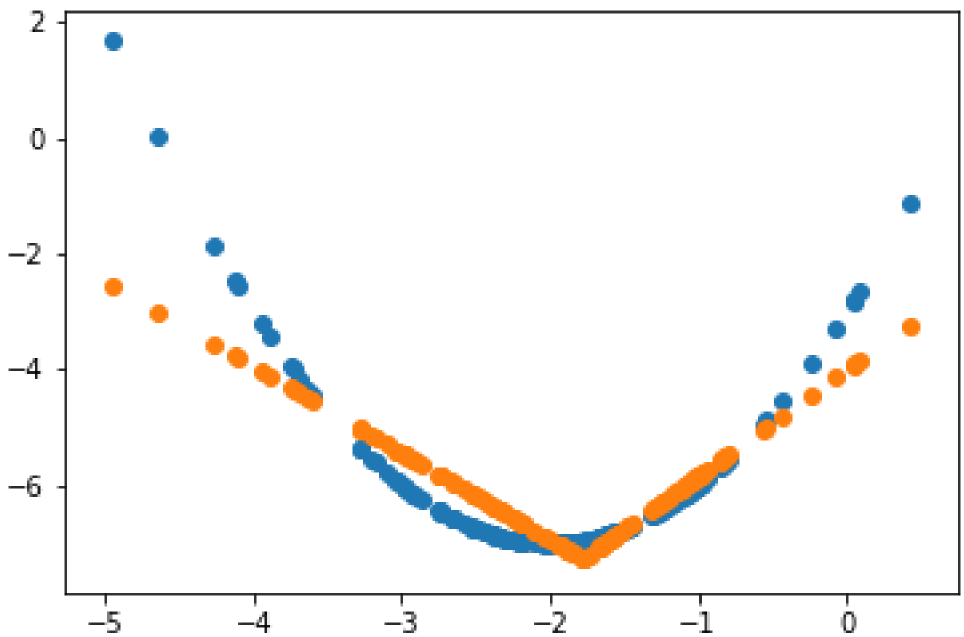
[87]:
```
# OH model on quadratic data
model = Linear_OH()
y_pred = train(model, epochs=1000, x=x, y=y)

# Plot predictions vs actual data
plt.scatter(x, y)
plt.scatter(x, y_pred)
plt.show()
```

```
Epoch 100 loss: 5.764029502868652
Epoch 200 loss: 3.9075746536254883
Epoch 300 loss: 3.003906011581421
Epoch 400 loss: 2.56402325630188
Epoch 500 loss: 2.34989595413208
Epoch 600 loss: 2.245661735534668
Epoch 700 loss: 2.194920063018799
Epoch 800 loss: 2.1702184677124023
Epoch 900 loss: 2.1581921577453613
Epoch 1000 loss: 2.152337074279785
```



[88]:
```python
# 1H model on quadratic data
model = Linear_1H()
y_pred = train(model, epochs=1000, x=x, y=y)
plt.scatter(x, y)
plt.scatter(x, y_pred)
plt.show()
```

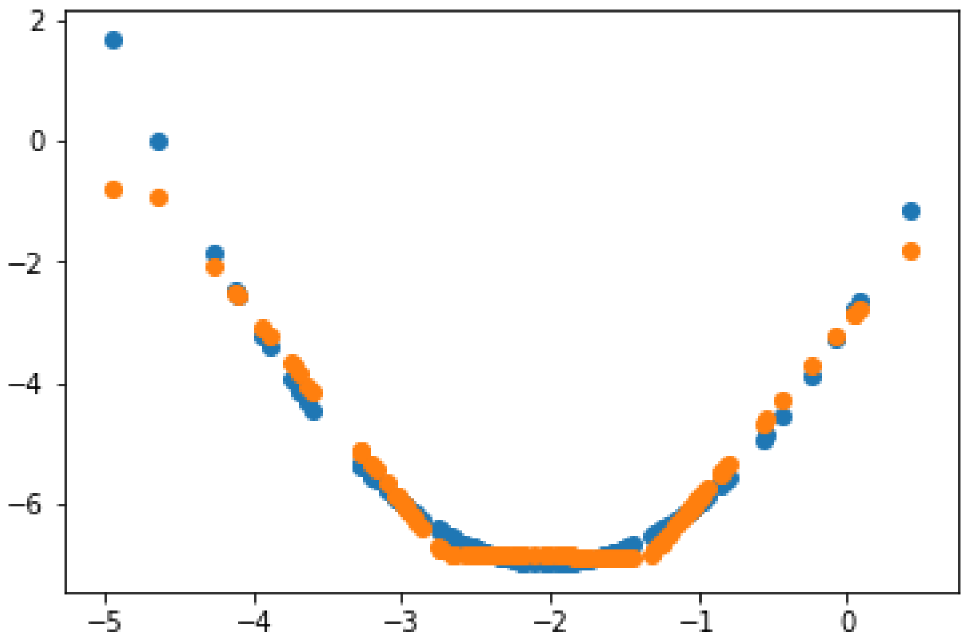
```
Epoch 100 loss: 2.3824825286865234
Epoch 200 loss: 2.1509294509887695
Epoch 300 loss: 2.1140153408050537
Epoch 400 loss: 2.0539796352386475
Epoch 500 loss: 1.9074057340621948
Epoch 600 loss: 1.6443638801574707
Epoch 700 loss: 1.3010938167572021
Epoch 800 loss: 0.9322571754455566
Epoch 900 loss: 0.6357808113098145
Epoch 1000 loss: 0.44889429211616516
```



[90]:
```python
# 2H model on quadratic data
model = Linear_2H()
y_pred = train(model, epochs=1000, x=x, y=y)
plt.scatter(x, y)
plt.scatter(x, y_pred)
```

```
plt.show()
```

```
Epoch 100 loss: 1.9617583751678467
Epoch 200 loss: 1.1546571254730225
Epoch 300 loss: 0.3617624044418335
Epoch 400 loss: 0.2981396019458771
Epoch 500 loss: 0.17849285900592804
Epoch 600 loss: 0.09209202975034714
Epoch 700 loss: 0.088165283203125
Epoch 800 loss: 0.0853244811296463
Epoch 900 loss: 0.08346401900053024
Epoch 1000 loss: 0.08186358213424683
```

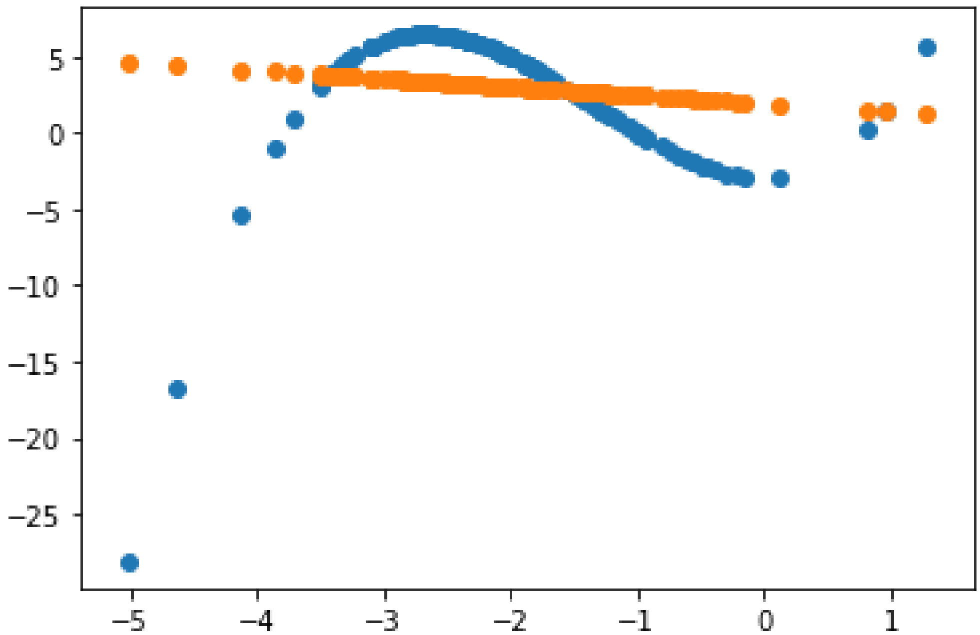


[0]:
```
# Generate cubic data with 128 data points
x, y = cubic_data_generator(128)
```

[92]:
```
# OH model on cubic data
model = Linear_OH()
y_pred = train(model, epochs=1000, x=x, y=y)
plt.scatter(x, y)
plt.scatter(x, y_pred)
plt.show()
```

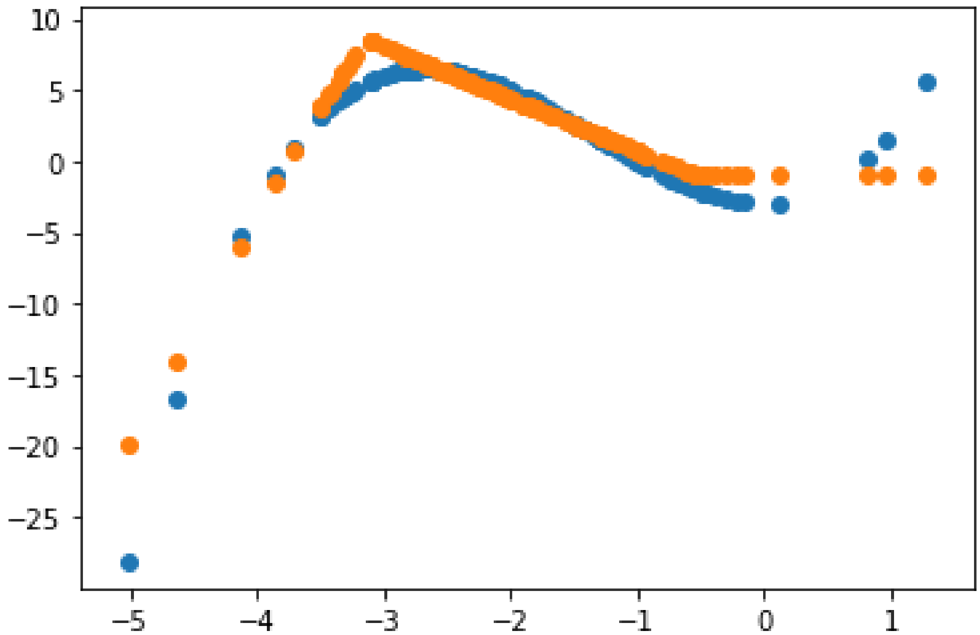
```
Epoch 100 loss: 18.896516799926758
Epoch 200 loss: 18.87861442565918
Epoch 300 loss: 18.870309829711914
```

```
Epoch 400 loss: 18.866456985473633
Epoch 500 loss: 18.864669799804688
Epoch 600 loss: 18.863840103149414
Epoch 700 loss: 18.863452911376953
Epoch 800 loss: 18.863277435302734
Epoch 900 loss: 18.86319351196289
Epoch 1000 loss: 18.863155364990234
```



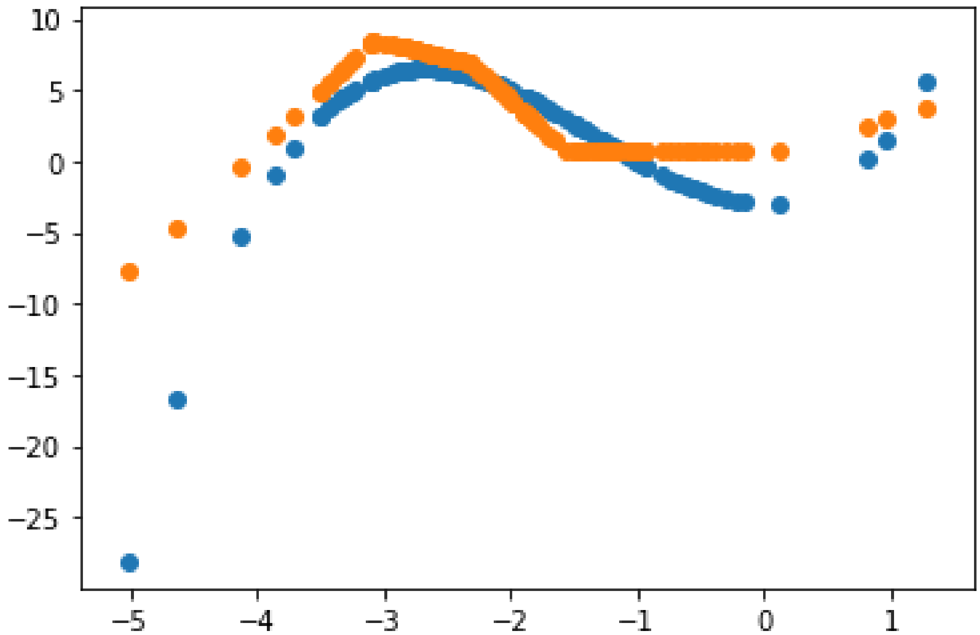
[93]:
```python
# 1H model on cubic data
model = Linear_1H()
y_pred = train(model, epochs=1000, x=x, y=y)
plt.scatter(x, y)
plt.scatter(x, y_pred)
plt.show()
```

```
Epoch 100 loss: 17.583845138549805
Epoch 200 loss: 14.212137222290039
Epoch 300 loss: 9.223616600036621
Epoch 400 loss: 5.255239963531494
Epoch 500 loss: 5.92222261428833
Epoch 600 loss: 4.260127067565918
Epoch 700 loss: 2.6851141452789307
Epoch 800 loss: 2.122239112854004
Epoch 900 loss: 1.7423985004425049
Epoch 1000 loss: 1.9023631811141968
```



[94]:
```
# 2H model on cubic data
model = Linear_2H()
y_pred = train(model, epochs=1000, x=x, y=y)
plt.scatter(x, y)
plt.scatter(x, y_pred)
plt.show()
```

```
Epoch 100 loss: 18.67383575439453
Epoch 200 loss: 17.809791564941406
Epoch 300 loss: 16.06325912475586
Epoch 400 loss: 12.200263023376465
Epoch 500 loss: 8.790221214294434
Epoch 600 loss: 7.805275917053223
Epoch 700 loss: 9.734648704528809
Epoch 800 loss: 8.781012535095215
Epoch 900 loss: 7.361730098724365
Epoch 1000 loss: 7.062869071960449
```



[0]: