PyNet: Building your own deep neural network with Numpy

Submission files:

- hw1.ipynb (with your completed code)
- PDF version of hw1.ipynb (with all results rendered)

Instructions:

- Complete the code below to train a deep neural network for binary classificaiton on a synthetic dataset.
- Design a deep neural network architecture for this binary classification task and achieve > 90% accuracy on test set.
- Note that to compute the backward pass, you might want to reuse some intermediate results from forward pass.

```
import numpy as np
import os
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
from matplotlib import cm
```

Parameter Class

A class storing the parameter values and gradients in a Layer.

```
class Parameter():
    """
    A class used to store parameters in a Layer.

    ...

    Attributes
    ----------
    val : np.array
        Values of learnable parameters. (e.g. weight/bias in linear layer)
    grad: np.array
        Values of gradients required for training.

    Methods
    -------
    init_param()
        Initialize the layer parameters before training.
    """
```

```
    def __init__(self, dim_shape):
        self.dim_shape = dim_shape
        self.val = np.zeros(dim_shape)
        self.grad = np.zeros_like(self.val)
```

Layer Class

An abstract class for all required computational modules in this assignment.

```
class Layer:
    """
    An abstract class for all required computational modules.

    ...

    Attributes
    ----------
    val : np.array
        Values of learnable parameters. (e.g. weight/bias in linear layer)
    grad: np.array
        Values of gradients required for training.

    Methods
    -------
    init_param()
        Initialize the layer parameters before training.
    forward()
        forward computation module.
    backward()
        backpropagation module.
    parameters()
        Get trainable parameters/gradients.
    """
    def __init__(self):
        pass

    def init_param(self):
        pass

    def forward(self):
        pass

    def backward(self):
        pass

    def parameters(self):
        pass
```

(1) Linear Layer Class [10 points]

Instructions

- init_param
  - Use random initialization for the weights, e.g., np.random.randn()*0.1 with the correct shape.
  - Zero initialization for the biases.
- forward
  - Having initialized weight and bias, and given an input to the layer, compute the output of the layer.
- backward
  - Suppose you have an upstream gradient d_output = $\frac{\partial loss}{\partial output}$
  - Compute (d_input, d_weight, d_bias) using the chain rule.

```
class Linear(Layer):
    """
    Linear layer.

    ...

    Methods
    -------
    init_param()
        Initialize the layer parameters before training.
    forward()
        forward computation module.
    backward()
        backpropagation module.
    parameters()
        Get trainable parameters/gradients of weight and bias for optimization.
    """
    def __init__(self, input_dim, output_dim):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.init_param()

    def init_param(self):
        weights_dim_shape = (self.output_dim, self.input_dim)
        biases_dim_shape = (self.output_dim, 1)
        self.weights = Parameter(weights_dim_shape)
        self.weights.val = np.random.randn(self.output_dim, self.input_dim)*0.1
        self.biases = Parameter(biases_dim_shape)
```

```
    def forward(self, input):
        self.x = input
        return np.dot(self.weights.val, input) + self.biases.val

    def backward(self, backward_input):
        self.weights.grad = np.dot(backward_input, self.x.T)
        self.biases.grad = backward_input.mean(axis=1, keepdims=True)
        grad = np.dot(self.weights.val.T, backward_input)

        return grad

    def parameters(self, learning_rate):
        #return [self.weights.val, self.weights.grad, self.biases.val, self.biases.gra
        #Instead of returning trainable parameters I have the SGD class call this func
        #directly for optimization. I believe this way is cleaner.
        #This allows for easier access to each layers weights and biases.
        self.weights.val = self.weights.val - learning_rate * self.weights.grad
        self.biases.val = self.biases.val - learning_rate * self.biases.grad
```

(2) ReLU Class [5 points]

Instruction

- forward
  - Implement forward computation for a ReLU unit.
- backward
  - Implement backpropagation for a ReLU unit.

```
class ReLU(Layer):
    """
    ReLU activation unit.

    ...

    Methods
    -------
    forward()
        forward computation module.
    backward()
        backpropagation module.
    """
    def __init__(self):
        pass

    def forward(self, input):
        self.input = input
```

```
        return np.maximum(0,self.input)

    def backward(self, backward_input):
        backward_input[self.input <= 0] = 0
        return backward_input
```

(3) Sigmoid Class [5 points]

Instructions

- forward
  - Implement forward computation for a sigmoid unit.
- backward
  - Implement backpropagation for a sigmoid unit.

```
class Sigmoid(Layer):
    """
    Sigmoid activation unit.

    ...

    Methods
    -------
    forward()
        forward computation module.
    backward()
        backpropagation module.
    """
    def __init__(self):
        pass

    def forward(self, input):
        self.input = input
        return 1/(1 + np.exp(-self.input))

    def backward(self, backward_input):
        prev = 1/(1 + np.exp(-self.input))
        return backward_input * prev * (1 - prev)
```

(4) Binary Cross-Entropy Loss Class [10 points]

Instructions

- forward

- Implement the forward computation to compute binary cross entropy loss: $y \log p + (1 - y) \log (1 - p)$
- Input and label are both probability distributions

- backward
  - Implement backpropagation for binary cross-entropy.

```
class BinaryCrossEntropyLoss(Layer):
    def __init__(self):
        pass

    def forward(self, input, label):
        bce_loss = -(label * np.log(input) + (1 - label) * np.log(1 - input)).mean()
        return bce_loss

    def backward(self, input, label):
        num_labels = len(label)
        grad = (-(label / input) + ((1 - label) / (1 - input))) / num_labels
        return grad
```

(5) Stochastic Gradient Descent Optimizer [5 points]

Instructions

- step
  - Update parameter values by their gradient.

```
class Optimizer_SGD():
    def __init__(self, param_list, learning_rate=0.01):
        self.learning_rate = learning_rate
        self.param_list = param_list

    def step(self, layer):
        self.layer = layer
        #the actual parameter update takes place within self.layer.parameters()
        self.layer.parameters(self.learning_rate)
        #for self.layer in self.layers:
          #if type(self.layer) is Linear:
            #dw = self.param_list[0]
            #db = self.param_list[0]
          #self.layer.weights.val = self.layer.weights.val - self.learning_rate * sel
          #self.layer.biases.val = self.layer.biases.val - self.learning_rate * self.
```

(6) Model Class [15 points]

Instructions

- Build a deep model by stacking multiple Linear layers with non-linear activations
- Initialize the parameters for Linear layers.
- Implement the forward computation for the model.
- Implement backpropagation for the model.
- Your model architecture should be able to attain at least 90% test accuracy on the subsequent task.

```
class Model:
    """
    A deep neural network architecture for binary classification.

    ...

    Methods
    -------
    __init__()
        Define your own deep network by stacking modules above.
    forward()
        forward computation.
    backward()
        backpagation.
    parameters()
        return learnable parameters/gradients stored in all layers in model.
    """
    def __init__(self, layers, loss_fn, opt_fn):
        self.layers = layers
        self.loss_fn = loss_fn
        self.opt_fn = opt_fn

    def forward(self, input):
        for layer in self.layers:
            forward = layer.forward(input)
            input = forward

        return forward

    def backward(self, backward_input):
        bce_loss = self.loss_fn
        loss = bce_loss.forward(self.X, backward_input)
        grad = bce_loss.backward(self.X, backward_input)

        for layer in reversed(self.layers):
            if type(layer) is not Linear:
                grad = layer.backward(grad)
            else:
                grad = layer.backward(grad)
                self.opt_fn.step(layer)
```

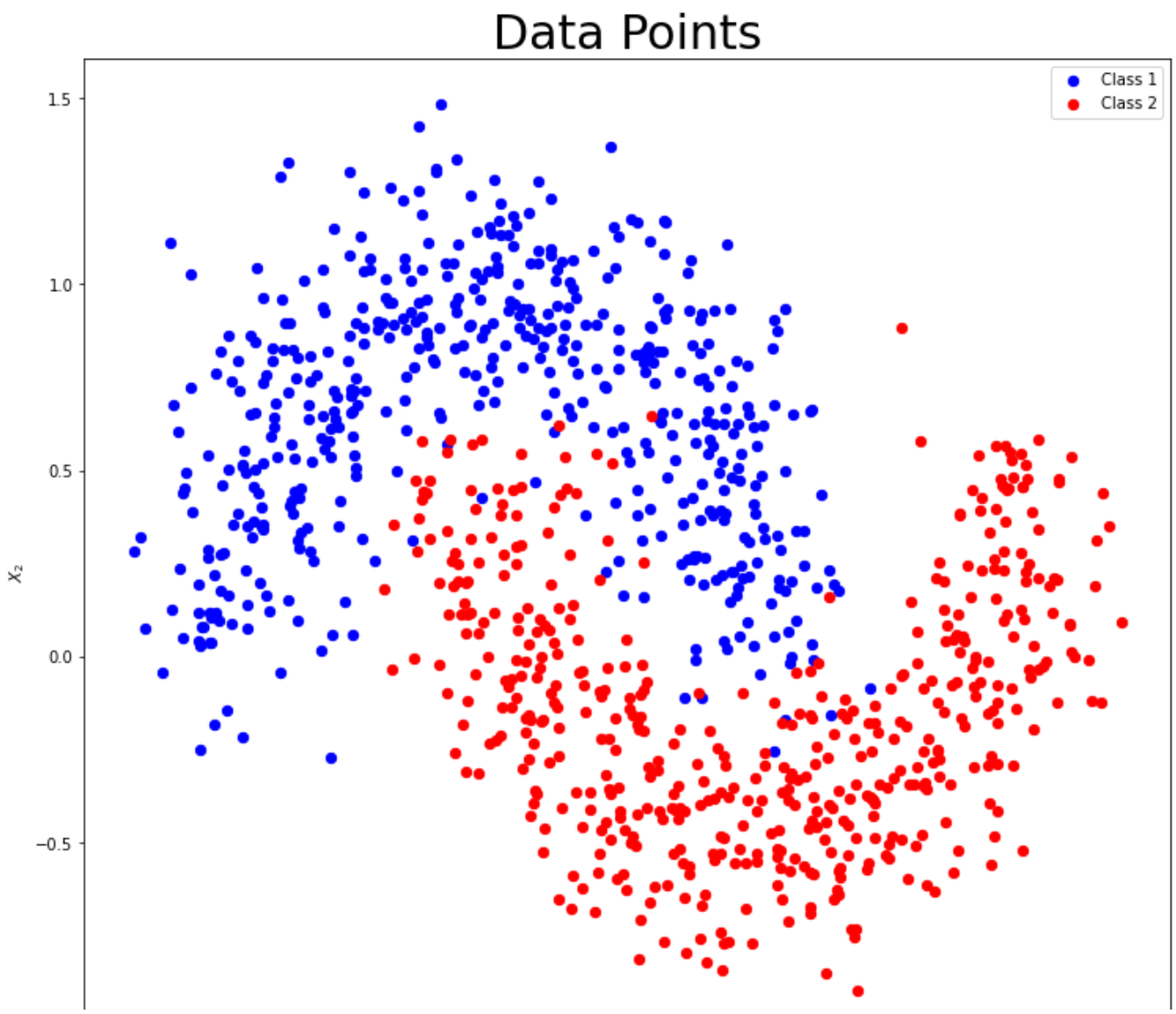
```
        return loss

    def parameters(self):
        #Instead of returning learnable parameters stored in all layers in model,
        #I decided to use sgd call to modify parameters directly via Linear.parameters
        #Thus, this function is unnecessary.
        #params = []
        #for layer in self.layers:
        #  params.append(layer.parameters())
        #return params
        pass
```

Generate Training Data

- Here, we generate 2D data points for binary classification.
- This data is not linearly separable, and therefore, you will need to build a multi-layer network with non-linear activations to fit the data.

```
# number of samples in the data set
N_SAMPLES = 1000
# ratio between training and test sets
TEST_SIZE = 0.1
X, y = make_moons(n_samples = N_SAMPLES, noise=0.2, random_state=100)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE, random_
```

```
# the function making up the graph of a dataset
def make_plot(X, y, plot_name):
    plt.figure(figsize=(16,12))
    axes = plt.gca()
    axes.set(xlabel="$X_1$", ylabel="$X_2$")
    plt.title(plot_name, fontsize=30)
    plt.subplots_adjust(left=0.20)
    plt.subplots_adjust(right=0.80)
    plt.scatter(X[y == 0, 0], X[y == 0, 1], s=40, c = 'blue', label = 'Class 1')
    plt.scatter(X[y == 1, 0], X[y == 1, 1], s=40, c = 'red', label = 'Class 2')
    plt.legend()
    plt.show()
make_plot(X, y, 'Data Points')
```



(7) Training and Testing the Model [15 points]

- Implement the training loop to optimize your model's parameters over multiple epochs.
- You can experiment with training strategy, e.g., batch size, epochs, learning rate, and model architecture to achieve the best performance.
- Note that you can modify the input shape to fit your implementation.
- Implement a fuction that runs a trained model on test data and returns accuracy.

```
loss_fn = BinaryCrossEntropyLoss()
architecture = [Linear(2, 4), ReLU(), Linear(4,8), ReLU(), Linear(8,16), ReLU(), Linea
optimizer = Optimizer_SGD([], learning_rate=0.04)
model = Model(layers=architecture, loss_fn=loss_fn, opt_fn=optimizer)

def train(X, Y, model, loss_fn, optimizer, epochs=10000):
    for epoch in range(epochs):
        forward = model.forward(X)
        model.X = forward
```

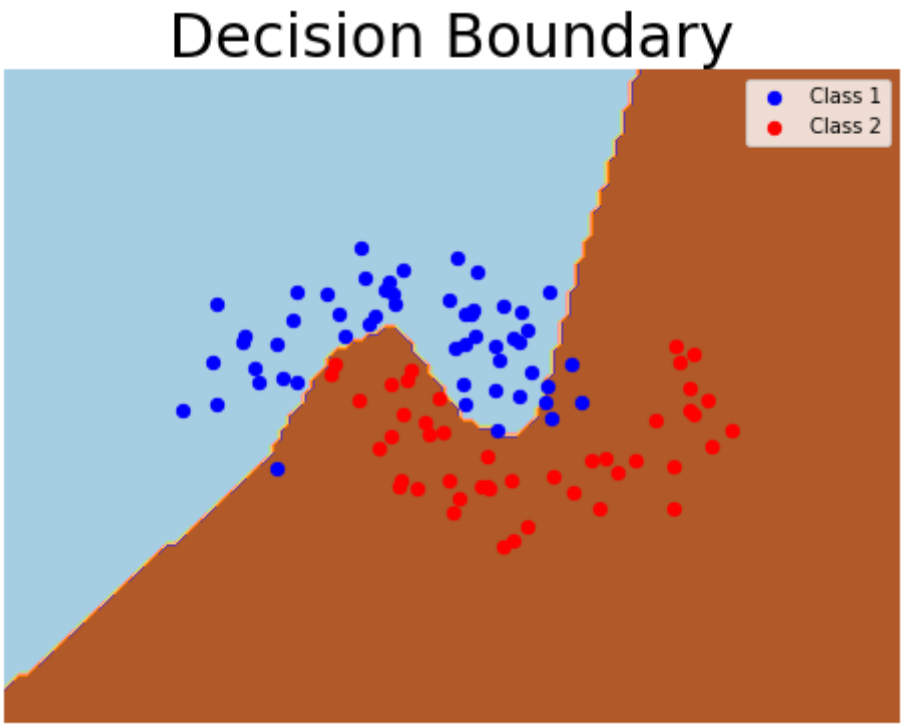
```
            loss = model.backward(Y)
            #optimizer.step()

            if epoch % 1000 == 0:
                print('Epoch: {} / Loss: {}'.format(epoch,loss))

    train(X_train.T,   y_train,  model, loss_fn,  optimizer, epochs=50000)

    def get_test_accuracy(model, X_test, y_test):
        preds = model.forward(X_test.T).T
        rounded_preds = []

        for p in preds:
            if p < 0.5:
                rounded_preds.append(0)
            else:
                rounded_preds.append(1)

        correct = 0
        total = len(y_test)
        for i in range(total):
            if rounded_preds[i] == y_test[i]:
                correct = correct + 1

        return (correct/total)

    accuracy = get_test_accuracy(model, X_test, y_test)
    print('Model accuracy: {}'.format(accuracy))
```

```
Epoch: 0 / Loss: 0.6931466335001086
Epoch: 1000 / Loss: 0.6931332030275714
Epoch: 2000 / Loss: 0.6931229504325431
Epoch: 3000 / Loss: 0.6931104123246883
Epoch: 4000 / Loss: 0.6930917282959189
Epoch: 5000 / Loss: 0.6930649526697739
Epoch: 6000 / Loss: 0.6930215596594108
Epoch: 7000 / Loss: 0.692939231388735
Epoch: 8000 / Loss: 0.6927399744441068
Epoch: 9000 / Loss: 0.6919172454112167
Epoch: 10000 / Loss: 0.5931097024067925
Epoch: 11000 / Loss: 0.4910226682921918
Epoch: 12000 / Loss: 0.48893126567528306
Epoch: 13000 / Loss: 0.4867117840223279
Epoch: 14000 / Loss: 0.48428672273546997
Epoch: 15000 / Loss: 0.4815488009964062
Epoch: 16000 / Loss: 0.4781445161456537
Epoch: 17000 / Loss: 0.4694943743700589
Epoch: 18000 / Loss: 0.26906504983146756
Epoch: 19000 / Loss: 0.2611092940871135
Epoch: 20000 / Loss: 0.25716211942475914
Epoch: 21000 / Loss: 0.254287607440756
Epoch: 22000 / Loss: 0.251109907738364
Epoch: 23000 / Loss: 0.24656110463235356
```

```
Epoch: 24000 / Loss: 0.24208999771187045
Epoch: 25000 / Loss: 0.23662022713899641
Epoch: 26000 / Loss: 0.23134580743145963
Epoch: 27000 / Loss: 0.2284063582078214
Epoch: 28000 / Loss: 0.22483356797982057
Epoch: 29000 / Loss: 0.22185318272276308
Epoch: 30000 / Loss: 0.2185843388813409
Epoch: 31000 / Loss: 0.21490611812874916
Epoch: 32000 / Loss: 0.21035869962315767
Epoch: 33000 / Loss: 0.20422891779218877
Epoch: 34000 / Loss: 0.1963948487922546
Epoch: 35000 / Loss: 0.18752839522800646
Epoch: 36000 / Loss: 0.1785620569753306
Epoch: 37000 / Loss: 0.16293602826457879
Epoch: 38000 / Loss: 0.17613588834522798
Epoch: 39000 / Loss: 0.13736172775782882
Epoch: 40000 / Loss: 0.11819033463543899
Epoch: 41000 / Loss: 0.12288022614982064
Epoch: 42000 / Loss: 0.27083847493692903
Epoch: 43000 / Loss: 0.10464314900858518
Epoch: 44000 / Loss: 0.14300693778163454
Epoch: 45000 / Loss: 0.11649234694111342
Epoch: 46000 / Loss: 0.13139490748360413
Epoch: 47000 / Loss: 0.10365130887629126
Epoch: 48000 / Loss: 0.09587607736884912
Epoch: 49000 / Loss: 0.09752583925226803
Model accuracy: 0.95
```

Visualization: Plot the Decision Boundary

```
def plot_decision_boundary(model, X_train, Y_train, X_test, Y_test):
    # function to draw the model's decision boundary
    h = 0.05
    X = np.concatenate([X_train, X_test], axis = 0)
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    input = np.stack([xx,yy], axis = -1).reshape(-1,2)
    pred = model.forward(input.transpose(1,0))
    index = pred >= 0.5
    plt.figure(figsize=(8,6))
    ax = plt.gca()
    plt.title('Decision Boundary', fontsize=30)
    ax.contourf(xx, yy, index.reshape(xx.shape), cmap=plt.cm.Paired)
    ax.axis('off')
    ax.scatter(X_test[Y_test == 0,0], X_test[Y_test == 0,1], s = 40,  c = 'blue', labe
    ax.scatter(X_test[Y_test == 1,0], X_test[Y_test == 1,1], s = 40, c = 'red', label
    plt.legend()
    plt.show()

plot_decision_boundary(model, X_train, y_train, X_test, y_test)
```



✓ 0s completed at 12:47 AM