B093040024
郭晏涵

深度視覺

HW4：forward 及 backward

notebook 執行過程

連上雲端硬碟、import 相關檔案

```
[1] from google.colab import drive
    drive.mount('/content/drive')

Mounted at /content/drive
```

```
import os
os.chdir('/content/drive/MyDrive/HW4/4')
os.listdir()

['__init__.py',
 'housing_data_preprocessing(optional).ipynb',
 'exercise04.zip',
 '1_simple_classifier.ipynb',
 'images',
 'exercise_code',
 'models']
```

```
[3] from exercise_code.data.csv_dataset import CSVDataset
    from exercise_code.data.csv_dataset import FeatureSelectorAndNormalizationTransform
    from exercise_code.data.dataloader import DataLoader

    import matplotlib.pyplot as plt
    import numpy as np
    import os
    import pandas as pd
    import seaborn as sns


    pd.options.mode.chained_assignment = None  # default='warn'

    %matplotlib inline
    %load_ext autoreload
    %autoreload 2
```

印出測試資料

```
from exercise_code.networks.utils import *
X_train, y_train, X_val, y_val, X_test, y_test, train_dataset = get_housing_data()
print("train data shape:", X_train.shape)
print("train targets shape:", y_train.shape)
print("val data shape:", X_val.shape)
print("val targets shape:", y_val.shape)
print("test data shape:", X_test.shape)
print("test targets shape:", y_test.shape, '\n')

print('The original dataset looks as follows:')
train_dataset.df.head()

/content/drive/MyDrive/HW4/4/exercise_code/networks/utils.py:69: FutureWarning: Dropping of nuisance columns in DataFrame reduction
  mn, mx, mean = df.min(), df.max(), df.mean()
You successfully loaded your data!

train data shape: (533, 1)
train targets shape: (533, 1)
val data shape: (167, 1)
val targets shape: (167, 1)
test data shape: (177, 1)
test targets shape: (177, 1)

The original dataset looks as follows:
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence
529	530	20	RL	NaN	32668	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN
491	492	50	RL	79.0	9490	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	MnPrv

計算 Loss function

2. Loss: Binary Cross Entropy

For a binary classification like our task, we use a loss function called Binary Cross-Entropy (BCE).

$$BCE(y,\hat{y}) = -y \cdot log(\hat{y}) - (1-y) \cdot log(1-\hat{y})$$

where $y \in \mathbb{R}$ is the ground truth and $\hat{y} \in \mathbb{R}$ is the predicted probability of the house being expensive.

Since the BCE function is a non-convex function, there is no closed-form solution for the optimal weights vector. In order to find the optimal parameters for our model, we need to use numeric methods such as Gradient Descent. But let us have a look at that later. First, you have to complete your first task:

Task: Implement

In `exercise_code/networks/loss.py` complete the implementation of the BCE loss function. You need to write the forward and backward pass of BCE as `forward()` and `backward()` function. The backward pass of the loss is needed to later optimize your weights of the model. You can test your implementation by the included testing code in the cell below.

```
[6] from exercise_code.tests.loss_tests import *
    from exercise_code.networks.loss import BCE

    bce_loss = BCE()
    print (BCETest(bce_loss)())
```

```
BCEForwardTest passed.
BCEBackwardTest passed.
Congratulations you have passed all the unit tests!!! Tests passed: 2/2
(0, 2)
```

利用 Binary Cross Entropy 公式計算 Loss、透過 Loss 計算 gradient

```python
    def forward(self, y_out, y_truth):
        """
        Performs the forward pass of the binary cross entropy loss function.

        :param y_out: [N, ] array predicted value of your model.
                y_truth: [N, ] array ground truth value of your training set.
        :return: [N, ] array of binary cross entropy loss for each sample of your training set.
        """
        result = None

        ########################################################################
        # TODO:
        # Implement the forward pass and return the output of the BCE loss.        #
        ########################################################################

        result = -y_truth*np.log(y_out)-(1-y_truth)*np.log(1-y_out)

        ########################################################################
        #                              END OF YOUR CODE
        ########################################################################

        return result

    def backward(self, y_out, y_truth):
        """
        Performs the backward pass of the loss function.

        :param y_out: [N, ] array predicted value of your model.
                y_truth: [N, ] array ground truth value of your training set.
        :return: [N, ] array of binary cross entropy loss gradients w.r.t y_out for
                  each sample of your training set.
        """
        gradient = None

        ########################################################################
        # TODO:
        # Implement the backward pass. Return the gradient wrt y_out
        ########################################################################

        gradient = -y_truth/y_out + (1 - y_truth)/(1 - y_out)

        ########################################################################
        #                              END OF YOUR CODE
        ########################################################################
        return gradient
```

接著透過 Backpropagation，將 Loss 帶入，得到新的 Weight

Task: Implement

Implement the `forward()` and `backward()` pass as well as the `sigmoid()` function in the `Classifier` class in `exercise_code/networks/classifier.py`. Check your implementation using the following testing code.

```
[7] from exercise_code.networks.classifier import Classifier
    from exercise_code.tests.classifier_test import *
    test_classifier(Classifier(num_features=2))
```

```
Sigmoid_Of_Zero passed.
Sigmoid_Of_Zero_Array passed.
Sigmoid_Of_100 passed.
Sigmoid_Of_Array_of_100 passed.
Method sigmoid() correctly implemented. Tests passed: 4/4
ClassifierForwardTest passed.
Method forward() correctly implemented. Tests passed: 1/1
ClassifierBackwardTest passed.
Method backward() correctly implemented. Tests passed: 1/1
Congratulations you have passed all the unit tests!!! Tests passed: 6/6
Score: 100/100
100
```

透過 sigmoid 函數將數值從實數域轉換到(0,1)之間

```python
    def sigmoid(self, x):
        """
        Computes the ouput of the sigmoid function

        :param x: input of the sigmoid, np.array of any shape
        :return: output of the sigmoid with same shape as input vector x
        """
        out = None

        ########################################################################
        # TODO:
        # Implement the sigmoid function, return out
        ########################################################################

        out = 1 / (1 + np.exp(-x))

        ########################################################################
        #                           END OF YOUR CODE
        ########################################################################

        return out
```

```python
    def forward(self, X):
        """
        Performs the forward pass of the model.

        :param X: N x D array of training data. Each row is a D-dimensional point.
        :return: Predicted labels for the data in X, shape N x 1
                 1-dimensional array of length N with classification scores.
        """
        assert self.W is not None, "weight matrix W is not initialized"
        # add a column of 1s to the data for the bias term
        batch_size, _ = X.shape
        X = np.concatenate((X, np.ones((batch_size, 1))), axis=1)
        # save the samples for the backward pass
        self.cache = X
        # output variable
        y = None

        ########################################################################
        # TODO:
        # Implement the forward pass and return the output of the model. Note  #
        # that you need to implement the function self.sigmoid() for that     #
        ########################################################################

        y = self.sigmoid(X.dot(self.W))

        ########################################################################
        #                           END OF YOUR CODE
        ########################################################################
```

利用偏微分化簡的公式，及算得到

```python
    def backward(self, y):
        """

        Performs the backward pass of the model.

        :param y: N x 1 array. The output of the forward pass.
        :return: Gradient of the model output (y=sigma(X*W)) wrt W
        """
        assert self.cache is not None, "run a forward pass before the backward pass"
        dW = None

        ########################################################################
        # TODO:
        # Implement the backward pass. Return the gradient wrt W, dW          #
        # The data X is stored in self.cache. Be careful with the dimensions     #
        # of W, X and y and note that the derivative of the sigmoid fct can be #
        # expressed by sigmoid itself
        ########################################################################

        dW = self.cache * y * (1-y)

        ########################################################################
        #                                          END OF YOUR CODE
        ########################################################################

        return dW
```

微分化簡過程

$$\mathrm{sigmoid}(t) = \frac{1}{1+e^{-t}}$$

$y = XW$ 代入 sigmoid

$$\hat{y} = \mathrm{sigmoid}(XW) = \frac{1}{1+e^{-XW}}$$

$$\text{let } f(t) = \mathrm{sigmoid}(t) = \frac{1}{1+e^{-t}}$$

$$f(t) + e^{-t} f(t) = 1$$

$$e^{-t} = \frac{1-f(t)}{f(t)}$$

$$\begin{aligned}
\frac{df(x)}{dt} &= \left((1+e^{-t})^{-1}\right)' \\
&= -(1+e^{-t})^{-2} \cdot -e^{-t} \\
&= \left(\frac{1}{1+e^{-t}}\right)^2 \cdot e^{-t} \\
&= f(t)^2 \cdot \frac{1-f(t)}{f(t)} \\
&= f(t) \cdot (1-f(t))
\end{aligned}$$

$$\therefore \frac{\partial \hat{y}}{\partial w} = f(wx) \cdot (1-f(wx)) = y \cdot (1-y)$$

透過降低 Gradient 來使得到新的 Weight

Task: Implement

In our model, we will use gradient descent to update the weights. Take a look at the `Optimizer` class in the file `networks/optimizer.py`. Your task is now to implement the gradient descent step in the `step()` method. You can test your implementation by the following testing code.

```python
[8] from exercise_code.networks.optimizer import Optimizer
    from exercise_code.networks.classifier import Classifier
    from exercise_code.tests.optimizer_test import *
    TestClassifier=Classifier(num_features=2)
    TestClassifier.initialize_weights()
    test_optimizer(Optimizer(TestClassifier))
```

```
OptimizerStepTest passed.
Congratulations you have passed all the unit tests!!! Tests passed: 1/1
Score: 100/100
100
```

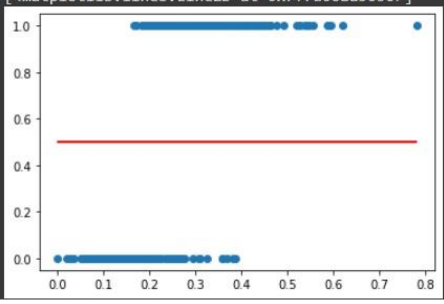
```python
    def step(self, dw):
        """
        :param dw: [D+1, 1] array gradient of loss w.r.t weights of your linear model
        :return weight: [D+1, 1] updated weight after one step of gradient descent
        """
        weight = self.model.W

        ########################################################################
        # TODO:
        # Implement the gradient descent for 1 step to compute the weight     #
        ########################################################################

        weight -= self.lr * dw

        ########################################################################
        #                                                   END OF YOUR CODE
        ########################################################################

        self.model.W = weight
```

Training 過程

[9]

```python
from exercise_code.networks.classifier import Classifier

#initialization
model = Classifier(num_features=1)
model.initialize_weights()

y_out, _ = model(X_train)

# plot the prediction
plt.scatter(X_train, y_train)
plt.plot(X_train, y_out, color='r')
```

```
[<matplotlib.lines.Line2D at 0x7f7d06aa5050>]
```



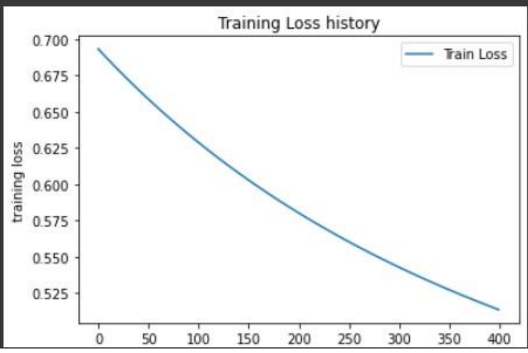
[10]

```python
from exercise_code.networks.optimizer import *
from exercise_code.networks.classifier import *
# Hyperparameter Setting, we will specify the loss function we use, and implement the optimizer we finished in the last step.
num_features = 1

# initialization
model = Classifier(num_features=num_features)
model.initialize_weights()

loss_func = BCE()
learning_rate = 5e-1
loss_history = []
opt = Optimizer(model,learning_rate)

steps = 400
# Full batch Gradient Descent
for i in range(steps):

    # Enable your model to store the gradient.
    model.train()

    # Compute the output and gradients w.r.t weights of your model for the input dataset.
    model_forward, model_backward = model(X_train)

    # Compute the loss and gradients w.r.t output of the model.
    loss, loss_grad = loss_func(model_forward, y_train)

    # Use back prop method to get the gradients of loss w.r.t the weights.
    grad = loss_grad * model_backward

    # Compute the average gradient over your batch
    grad = np.mean(grad, 0, keepdims = True)

    # After obtaining the gradients of loss with respect to the weights, we can use optimizer to
    # do gradient descent step.
```

```
[10]        # Take transpose to have the same shape ([D+1,1]) as weights.
            opt.step(grad.T)

            # Average over the loss of the entire dataset and store it.
            average_loss = np.mean(loss)
            loss_history.append(average_loss)
            if i%10 == 0:
                print("Epoch ",i,"--- Average Loss: ", average_loss)
```

```
Epoch 0 --- Average Loss:  0.6931242216207913
Epoch 10 --- Average Loss:  0.6857343594695261
Epoch 20 --- Average Loss:  0.6786207367198962
Epoch 30 --- Average Loss:  0.6717162074928953
Epoch 40 --- Average Loss:  0.6650108270169793
Epoch 50 --- Average Loss:  0.6584981059551532
Epoch 60 --- Average Loss:  0.6521718455866289
Epoch 70 --- Average Loss:  0.6460259658339268
Epoch 80 --- Average Loss:  0.640054508180383
Epoch 90 --- Average Loss:  0.6342516467897892
Epoch 100 --- Average Loss:  0.6286116981045891
Epoch 110 --- Average Loss:  0.6231291284809178
Epoch 120 --- Average Loss:  0.6177985599762699
Epoch 130 --- Average Loss:  0.6126147744504016
Epoch 140 --- Average Loss:  0.6075727161507046
Epoch 150 --- Average Loss:  0.6026674929563094
Epoch 160 --- Average Loss:  0.597894376452596
Epoch 170 --- Average Loss:  0.5932488010008128
Epoch 180 --- Average Loss:  0.5887263619573548
Epoch 190 --- Average Loss:  0.584322813185038
Epoch 200 --- Average Loss:  0.5800340639853611
Epoch 210 --- Average Loss:  0.5758561755670263
Epoch 220 --- Average Loss:  0.5717853571524096
Epoch 230 --- Average Loss:  0.5678179618107085
Epoch 240 --- Average Loss:  0.5639504820943194
Epoch 250 --- Average Loss:  0.5601795455438575
Epoch 260 --- Average Loss:  0.5565019101171372
```

```
[11] # Plot the loss history to see how it goes after several steps of gradient descent.
     plt.plot(loss_history, label = 'Train Loss')
     plt.xlabel('iteration')
     plt.ylabel('training loss')
     plt.title('Training Loss history')
     plt.legend()
     plt.show()


     # forward pass
     y_out, _ = model(X_train)


     # plot the prediction
     plt.scatter(X_train, y_train, label = 'Ground Truth')
     inds = X_train.argsort(0).flatten()
     plt.plot(X_train[inds], y_out[inds], color='r', label = 'Prediction')
     plt.title('Prediction of our trained model')
     plt.legend()
     plt.show()
```



Task: Implement

Open the file `exercise_code/solver.py` and have a look at the `Solver` class. The `_step()` function is representing one single training step. So when using the Gradient Descent method, it represents one single update step using the Gradient Descent method. Your task is now to finalize this `_step()` function. You can test your implementation with the testing code included in the following cell.

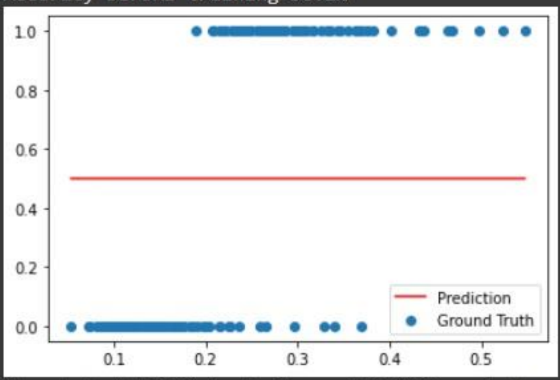
Hint: The implementation of the `_step()` function is very similar to the implementation of a training step as we observed above. You may have a look at that part first.

```
[12] from exercise_code.solver import Solver
     from exercise_code.networks.classifier import Classifier
     from exercise_code.tests.solver_tests import *
     weights = np.array([[0.1],[0.1]])
     TestClassifier = Classifier(num_features=1)
     TestClassifier.initialize_weights(weights)
     learning_rate = 5e-1
     data = {'X_train': X_train, 'y_train': y_train,
             'X_val': X_val, 'y_val': y_val}
     loss = BCE()
     solver = Solver(TestClassifier,data,loss,learning_rate,verbose=True)

     test_solver(solver)
```

```
SolverStepTest passed.
Congratulations you have passed all the unit tests!!! Tests passed: 1/1
Score: 100/100
100
```

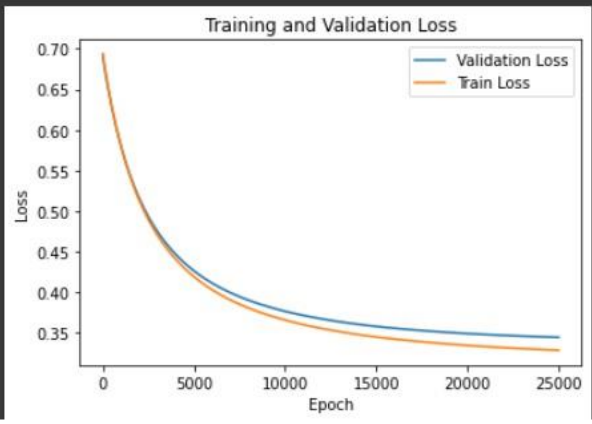
```python
[13]
from exercise_code.solver import Solver
from exercise_code.networks.utils import test_accuracy
from exercise_code.networks.classifier import Classifier
# Select the number of features, you want your task to train on.
# Feel free to play with the sizes.
num_features = 1

# initialize model and weights
model = Classifier(num_features=num_features)
model.initialize_weights()

y_out, _ = model(X_test)

accuracy = test_accuracy(y_out, y_test)
print("Accuracy BEFORE training {:.1f}%".format(accuracy*100))


if np.shape(X_test)[1]==1:
        plt.scatter(X_test, y_test, label = "Ground Truth")
        inds = X_test.flatten().argsort(0)
        plt.plot(X_test[inds], y_out[inds], color='r', label = "Prediction")
        plt.legend()
        plt.show()

data = {'X_train': X_train, 'y_train': y_train,
        'X_val': X_val, 'y_val': y_val}

#We use the BCE loss
loss = BCE()

# Please use these hyperparmeter as we also use them later in the evaluation
learning_rate = 1e-1
epochs = 25000

# Setup for the actual solver that's going to do the job of training
# the model on the given data. set 'verbose=True' to see real time
# progress of the training.
solver = Solver(model,
                data,
                loss,
                learning_rate,
                verbose=True,
                print_every = 1000)
# Train the model, and look at the results.
solver.train(epochs)


# Test final performance
y_out, _ = model(X_test)

accuracy = test_accuracy(y_out, y_test)
print("Accuracy AFTER training {:.1f}%".format(accuracy*100))
```

```
Accuracy BEFORE training 58.2%
```




```
(Epoch 0 / 25000) train loss: 0.693002; val_loss: 0.692964
(Epoch 1000 / 25000) train loss: 0.580014; val_loss: 0.580251
(Epoch 2000 / 25000) train loss: 0.513281; val_loss: 0.516013
```

```
[14] plt.plot(solver.val_loss_history, label = "Validation Loss")
     plt.plot(solver.train_loss_history, label = "Train Loss")
     plt.xlabel("Epoch")
     plt.ylabel("Loss")
     plt.legend()
     plt.title('Training and Validation Loss')
     plt.show()


     if np.shape(X_test)[1]==1:

         plt.scatter(X_test, y_test, label = "Ground Truth")
         inds = X_test.argsort(0).flatten()
         plt.plot(X_test[inds], y_out[inds], color='r', label = "Prediction")
         plt.legend()
         plt.title('Prediction of your trained model')
         plt.show()
```



在 Solver 之中，完成最後 Training 結果

7. Save your BCE Loss, Classifier and Solver for Submission

Your model should be trained now and able to predict whether a house is expensive or not. Hooooooray, you trained your very first model! The model will be saved as a pickle file to `models/simple_classifier.p`.

```
[15] from exercise_code.tests import save_pickle

     save_pickle(
         data_dict={
             "BCE_class": BCE,
             "Classifier_class": Classifier,
             "Optimizer": Optimizer,
             "Solver_class": Solver
         },
         file_name="simple_classifier.p"
     )
```

```
[16] from exercise_code.submit import submit_exercise

     submit_exercise('exercise04')
```

```
relevant folders: ['models']
notebooks files: []
Adding folder models
Zipping successful! Zip is stored under: /content/exercise04.zip
```

参考 Part 5 Training 完成 Solver 中_step

```python
    def _step(self):
        """
        Make a single gradient update. This is called by train() and should not
        be called manually.
        """
        model = self.model
        loss_func = self.loss_func
        X_train = self.X_train
        y_train = self.y_train
        opt = self.opt
        ##########################################################################
        #     TODO:
        #     Get the gradients dhat{y}/dW and dLoss/dhat{y}.
        #     Combine them via the chain rule to obtain dLoss / dW.
        #     Proceed by performing an optimizing step using the given
        #     optimizer (by calling opt.step() with the gradient wrt W)
        #
        #     Hint: don't forget to divide number of samples when computing the   #
        #     gradient!
        ##########################################################################

        model.train()
        model_forward, model_backward = model(X_train)
        _, loss_grad = loss_func(model_forward, y_train)

        grad = loss_grad * model_backward
        grad = np.mean(grad, 0, keepdims = True)

        opt.step(grad.T)

        ##########################################################################
        #                                                     END OF YOUR CODE
        ##########################################################################
```