Chapter 9

A Tutorial on Neural Networks

9.1 Learning Objectives

This chapter gives a comprehensive tutorial to demonstrate how we implement some key practical considerations (discussed in chapter 8) in real projects through python programming. Examples in chapter 8 have already shown the implementation of weight regularization and dropout in python. The tutorial in this chapter will focus on mini-batch gradient descent and optimization methods. After completing this chapter, one should be able to

- Develop a general neural network from the scratch;
- Implement mini-batch training;
- Implement momentum and Adam optimizations from the scratch;
- Be aware of the effects of hyperparameters on neural networks;

9.2 Architecture of Neural Network

We will build a generic 3-layer neural network, as shown in Fig.1, and apply it to two different tasks throughout this chapter. The size of each layer is defined by the corresponding element in the list *layers_dims*. For example, *layers_dims* =[12288, 25,12,6] specifies 12288 features of input, 25 units in the first hidden layer and 12 units in the second hidden layer, 6 units in the output layer. All hidden layers use ReLU activation functions while the output layer uses softmax for classification. Note that binary classification can be implemented by assigning *layers_dims*[3]=2 with one-hot-coded labels.

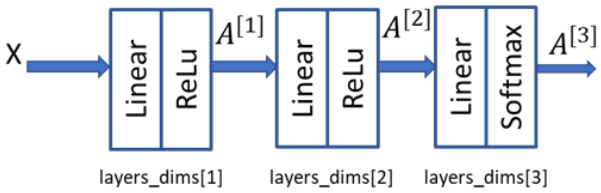


Fig.1 A generic 3-layer neural network

We will train the neural network using mini-batch strategy to feed data, and using three different optimization options: basic gradient descent, momentum and Adam. The network model in python is given by

```
model(X, Y, layers_dims, optimizer, learning_rate = 0.0007, mini_batch_size = 64, beta
      = 0.9, beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 10000, print_cost
      = True)
```

The values for arguments are default. The meaning of each argument will be explained in the subsequent texts.

The design flowchart of the model is shown in Fig.2, giving an overall picture of the model. The red fonts indicate functions defined separately. This flowchart can help a reader understand the whole picture of mini-batch concept and its implementation. It is helpful to refer to this flowchart when reading through the subsequent texts.

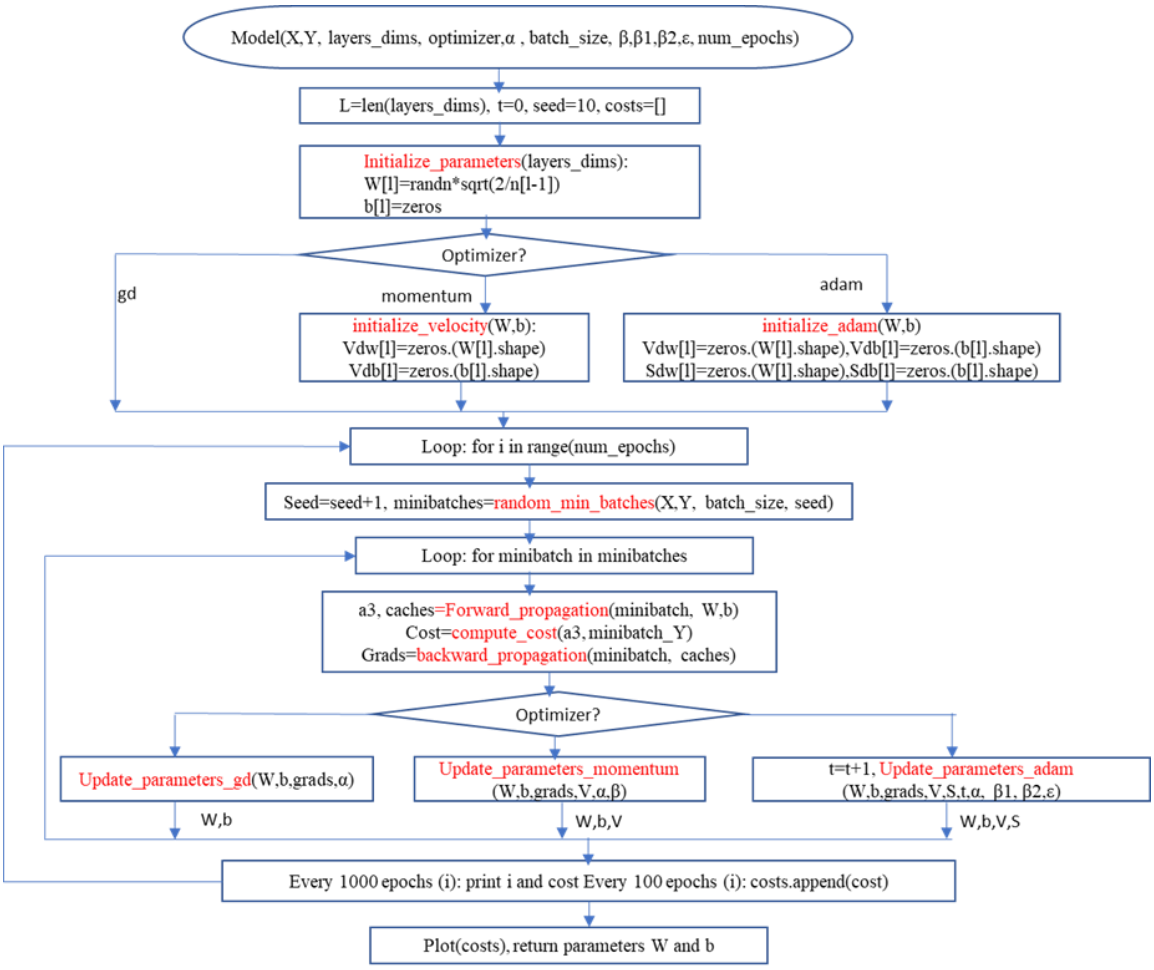


Fig. 2 Design flowchart of the neural network model

The model has the following arguments:
X -- input data, of shape (number of features, number of examples)
Y – one-hot-label, of shape (K, number of examples), where K is the number of classes.
layers_dims -- python list, containing the size of each layer
learning_rate α -- the learning rate, scalar.
mini_batch_size -- the size of a mini batch
beta -- Momentum hyperparameter
beta1 -- Exponential decay hyperparameter for the past gradients estimates
beta2 -- Exponential decay hyperparameter for the past squared gradients estimates
epsilon -- hyperparameter preventing division by zero in Adam updates
num_epochs -- number of epochs
print_cost -- True to print the cost every 100 epochs
Returns: parameters -- python dictionary containing your updated parameters

The parameters W and b are updated in two nested loops. The outer loop is designed for different epochs while the inner loop works for different batches within an epoch. Each epoch is obtained by randomly shuffling the original dataset, and then partitioned into mini-batches. Each update of parameters is based on one mini-batch. For example, if the total number of epochs is 10,000, and 5 mini-batches in each epoch, then the cost and the parameters W and b will be updated 50,000 times. Please note that we can specify to save and plot some costs (not all) which includes enough information for monitoring the behavior of training. When the mini-batch size is equal to one, the model implements stochastic gradient descent. When the mini-batch size is *m*, the model implements batch gradient descent. There are two levels of initializations: 1) basically initialize W and b; and 2) extra initialize V for momentum and (V,S) for Adam.

9.3 Details of Key Functions

In this section, we will describe the design details of some key functions which are used for neural network model construction. Forward propagation, cost function, and backward propagation have been detailed and standardized in previous chapters. Here we focus on mini-batch process, momentum and Adam optimizations.

9.3.1 Mini-batch gradient descent

```
def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
```

The function random_mini_batches ( ) divides the entire training set into equally sized portions (i.e. batches). To make each epoch different, the training set needs to be randomly shuffled before being divided:

1) **Shuffle**: Create a shuffled version of the training set (X, Y) as shown in Fig.3. Each column of X and Y represents a training example. Note that the random shuffling is done synchronously between X and Y. Such that after the shuffling the ith column of X is the example corresponding to the ith label in Y. The shuffling step ensures that examples will be split randomly into different mini-batches.

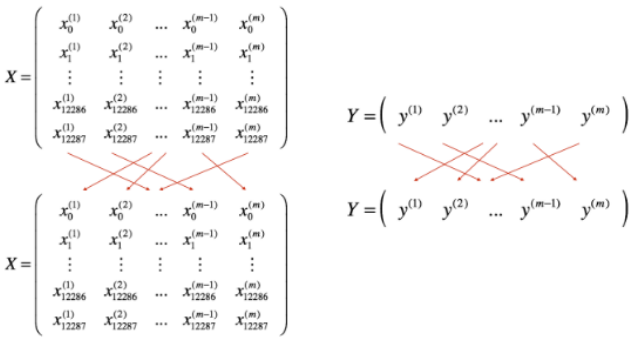


Fig.3 Shuffle training set

2) Partition: Partition the shuffled (X, Y) into mini-batches of size mini_batch_size (e.g. 64). Note that the number of training examples is not always divisible by mini_batch_size. The last mini batch might be smaller, but you don't need to worry about this. When the final mini-batch is smaller than the full mini_batch_size, it will look like this:

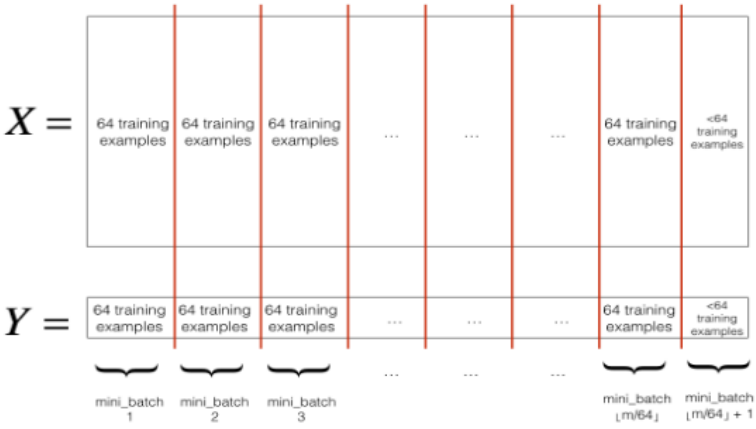


Fig.4 mini-batches

The function random_mini_batches ( ) returns a Python list mini_batches whose elements are mini-batches.

9.3.2 Optimization with momentum

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "zigzag" toward convergence. Using momentum can reduce this zigzagging, as shown in Fig.5. Momentum takes into account the past gradients to smooth out the update. We will store the 'direction' of the previous gradients in the variable v. Formally, this will be the exponentially weighted average of the gradient on previous steps. From the previous chapter, we know that optimization with momentum can be described by (17) in the previous chapter (chapter 8.)

Initialization: $v_{dw} = 0, v_{db} = 0$

On iteration t:

Compute dW, db on the current mini-batch

$v_{dw} = \beta v_{dw} + (1 - \beta)dW$

$v_{db} = \beta v_{db} + (1 - \beta)db$

$W := W - \alpha v_{dw}$

$b := b - \alpha v_{db}$

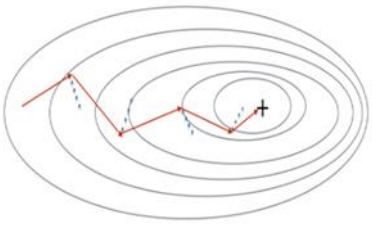


Fig.5 The red arrows show the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step.

To implement this momentum, we need to define the following two functions:

```
def initialize_velocity(parameters):

def update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):
```

where beta (i.e. β) is the momentum and learning_rate (i.e. α) is the learning rate. All parameters (i.e. W, b) should be stored in the parameters dictionary.

Remarks:

- The velocity is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps.
- If β=0, then this just becomes standard gradient descent without momentum.
- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, β=0.9 is often a reasonable default.
- Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.
- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You usually need to tune a momentum hyperparameter β and a learning rate α.

9.3.3 Optimization with Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp and Momentum. The algorithm is given by (19) in chapter 8 as follows:

Initialization: $v_{dw} = 0, s_{dw} = 0, v_{db} = 0, s_{db} = 0$

On batch iteration t:

Compute dW, db on the current mini-batch

$v_{dw} = \beta_1 v_{dw} + (1 - \beta_1)dw,$

$v_{db} = \beta_1 v_{db} + (1 - \beta_1)db$

$s_{dw} = \beta_2 s_{dw} + (1 - \beta_2)(dw * dw),$

$s_{db} = \beta_2 s_{db} + (1 - \beta_2)(db * db)$

$v_{dw} \coloneqq \frac{v_{dw}}{1 - \beta_1^t}$

$v_{db} \coloneqq \frac{v_{db}}{1 - \beta_1^t}$

$s_{dw} \coloneqq \frac{s_{dw}}{1 - \beta_2^t}$

$s_{db} \coloneqq \frac{s_{db}}{1 - \beta_2^t}$

$W \coloneqq W - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \varepsilon}, \; b \coloneqq b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \varepsilon}$

The following two functions are designed to perform the parameter update for Adam algorithm.

```
def initialize_adam(parameters) :

def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,

                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):
```

The algorithm performs the following computations for parameters of all layers:

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and v_corrected (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and s_corrected (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".
4. It returns parameters, v and s.

where:

- t counts the number of steps taken of Adam, i.e. how many times the parameters have been updated so far. It is not the index of epochs, nor the index of mini-batches. It is equal to the product of epoch number and mini-batch number.
- L is the number of layers
- β1 and β2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

9.4 Python Programming

9.4.1 Python files

This section attaches Python codes which demonstrate how the ideas are exactly implemented in the neural network. The first file is opt_utils.py for some basic functions. The second file ch8.py includes all other required functions, top-level neural network model, and applications to two classification tasks.

\machine learing\NN_m_class\opt_utils.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 11 15:57:44 2019
opt_utils.py
@author: weido
"""

import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy.io
import sklearn
import sklearn.datasets

def sigmoid(x):
    """
    Compute the sigmoid of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- sigmoid(x)
    """
    s = 1/(1+np.exp(-x))
    return s
def softmax(x):
    """
    Compute the sigmoid of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- softmax(x)
    """
    t=np.exp(x)
    s = t/np.sum(t, axis=0)

    return s
def relu(x):
    """
    Compute the relu of x
    Arguments:
    x -- A scalar or numpy array of any size.
    Return:
    s -- relu(x)
    """
    s = np.maximum(0,x)

```

```
    return s

def load_params_and_grads(seed=1):
    np.random.seed(seed)
    W1 = np.random.randn(2,3)
    b1 = np.random.randn(2,1)
    W2 = np.random.randn(3,3)
    b2 = np.random.randn(3,1)

    dW1 = np.random.randn(2,3)
    db1 = np.random.randn(2,1)
    dW2 = np.random.randn(3,3)
    db2 = np.random.randn(3,1)

    return W1, b1, W2, b2, dW1, db1, dW2, db2


def initialize_parameters(layer_dims):
    """
    Arguments:
    layer_dims -
- python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -
- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                    W1 -- weight matrix of shape (layer_dims[l], layer_dims[l-
1])
                    b1 -- bias vector of shape (layer_dims[l], 1)
                    Wl -- weight matrix of shape (layer_dims[l-
1], layer_dims[l])
                    bl -- bias vector of shape (1, layer_dims[l])

    Tips:
    - For example: the layer_dims for the "Planar Data classification model" woul
d have been [2,2,1].
    This means W1's shape was (2,2), b1 was (1,2), W2 was (2,1) and b2 was (1,1).
 Now you have to generalize it!
    - In the for loop, use parameters['W' + str(l)] to access Wl, where l is the
iterative integer.
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims) # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-
1])*  np.sqrt(2 / layer_dims[l-1])
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == layer_dims[l], layer_dims[l-
1])
        assert(parameters['W' + str(l)].shape == layer_dims[l], 1)

    return parameters


def compute_cost(a3, Y):

    """
```

```
    Implement the cost function

    Arguments:
    a3 -- post-activation, output of forward propagation
    Y -- "true" labels vector, same shape as a3

    Returns:
    cost - value of the cost function
    """
    m = Y.shape[1]
    k = Y.shape[0]
    if k==1: # sigmoid for 2 classes
        logprobs = np.multiply(-np.log(a3),Y) + np.multiply(-
np.log(1 - a3), 1 - Y)
        cost = 1./m * np.sum(logprobs)

    else:  #softmax for multiple classes
        logprobs = np.multiply(np.log(a3),Y)
        cost = -1/m*np.sum(logprobs)
        cost = np.squeeze(cost)
    return cost

def forward_propagation(X, parameters):
    """
    Implements the forward propagation (and computes the loss) presented in Figur
e 2.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    parameters -
- python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3"
:
                    W1 -- weight matrix of shape ()
                    b1 -- bias vector of shape ()
                    W2 -- weight matrix of shape ()
                    b2 -- bias vector of shape ()
                    W3 -- weight matrix of shape ()
                    b3 -- bias vector of shape ()

    Returns:
    loss -- the loss function (vanilla logistic loss)
    """

    # retrieve parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    z1 = np.dot(W1, X) + b1
    a1 = relu(z1)
    z2 = np.dot(W2, a1) + b2
    a2 = relu(z2)
    z3 = np.dot(W3, a2) + b3
    a3 = softmax(z3)

    cache = (z1, a1, W1, b1, z2, a2, W2, b2, z3, a3, W3, b3)

```

```
    return a3, cache

def backward_propagation(X, Y, cache):
    """
    Implement the backward propagation presented in figure 2.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat)
    cache -- cache output from forward_propagation()

    Returns:
    gradients -
- A dictionary with the gradients with respect to each parameter, activation and
pre-activation variables
    """
    m = X.shape[1]
    (z1, a1, W1, b1, z2, a2, W2, b2, z3, a3, W3, b3) = cache

    dz3 = 1./m * (a3 - Y)
    dW3 = np.dot(dz3, a2.T)
    db3 = np.sum(dz3, axis=1, keepdims = True)

    da2 = np.dot(W3.T, dz3)
    dz2 = np.multiply(da2, np.int64(a2 > 0))
    dW2 = np.dot(dz2, a1.T)
    db2 = np.sum(dz2, axis=1, keepdims = True)

    da1 = np.dot(W2.T, dz2)
    dz1 = np.multiply(da1, np.int64(a1 > 0))
    dW1 = np.dot(dz1, X.T)
    db1 = np.sum(dz1, axis=1, keepdims = True)

    gradients = {"dz3": dz3, "dW3": dW3, "db3": db3,
                 "da2": da2, "dz2": dz2, "dW2": dW2, "db2": db2,
                 "da1": da1, "dz1": dz1, "dW1": dW1, "db1": db1}

    return gradients

def predict(X, parameters):
    """
    This function is used to predict the results of a  n-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    y_esti -- predictions (labels:0,1,2,3,4,,5) for the given dataset X
    """

    m = X.shape[1]
    y_esti = np.zeros((1,m), dtype = np.int)
    # Forward propagation
    a3, caches = forward_propagation(X, parameters)


    for i in range(0, a3.shape[1]):
        y_esti[0,i]=np.argmax(a3[:,i])

    return y_esti
```

```
215.
216. def load_2D_dataset():
217.     data = scipy.io.loadmat('datasets/data.mat')
218.     train_X = data['X'].T
219.     train_Y = data['y'].T
220.     test_X = data['Xval'].T
221.     test_Y = data['yval'].T
222.
223.     plt.scatter(train_X[0, :], train_X[1, :], c=train_Y.ravel(), s=40, cmap=plt.c
    m.Spectral);
224.
225.     return train_X, train_Y, test_X, test_Y
226.
227. def plot_decision_boundary(model, X, y):
228.     # Set min and max values and give it some padding
229.     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
230.     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
231.     h = 0.01
232.     # Generate a grid of points with distance h between them
233.     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

234.     # Predict the function value for the whole grid
235.     Z = model(np.c_[xx.ravel(), yy.ravel()])
236.     Z = Z.reshape(xx.shape)
237.     # Plot the contour and training examples
238.     plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
239.     plt.ylabel('x2')
240.     plt.xlabel('x1')
241.     plt.scatter(X[0, :], X[1, :], c=y.ravel(), cmap=plt.cm.Spectral)
242.     plt.show()
243.
244. def predict_dec(parameters, X):
245.     """
246.     Used for plotting decision boundary.
247.
248.     Arguments:
249.     parameters -- python dictionary containing your parameters
250.     X -- input data of size (m, K)
251.
252.     Returns
253.     predictions -- vector of predictions of our model (red: 0 / blue: 1)
254.     """
255.
256.     # Predict using forward propagation and a classification threshold of 0.5
257.     a3, cache = forward_propagation(X, parameters)
258.     predictions = (a3 > 0.5)
259.     return predictions
260.
261. def load_dataset():
262.     np.random.seed(3)
263.     train_X, train_Y = sklearn.datasets.make_moons(n_samples=300, noise=.2) #300
    #0.2
264.     # Visualize the data
265.     plt.scatter(train_X[:, 0], train_X[:, 1], c=train_Y.ravel(), s=40, cmap=plt.c
    m.Spectral);
266.     train_X = train_X.T
267.     train_Y = train_Y.reshape((1, train_Y.shape[0]))
268.
269.     return train_X, train_Y
270.
271. #### for multiple class classification
```

```
272.
273. def load_dataset_signs():
274.     train_dataset = h5py.File('train_signs.h5', "r")
275.     train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set
      features
276.     train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set
      labels
277.
278.     test_dataset = h5py.File('test_signs.h5', "r")
279.     test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set fea
    tures
280.     test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set lab
    els
281.
282.     classes = np.array(test_dataset["list_classes"][:]) # the list of classes
283.
284.     train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
285.     test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))
286.
287.     return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig,
    classes
288. def convert_to_one_hot(Y, C):
289.     Y = np.eye(C)[Y.reshape(-1)].T
290.     return Y
```

\machine_learning\NN_m_class\ch9.py

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  Created on Tue Jun 11 16:00:23 2019
4.  dnn_opt.py changed to ch9.py
5.  @author: weido
6.  """
7.
8.  import numpy as np
9.  import matplotlib.pyplot as plt
10. import scipy.io
11. import math
12. import sklearn
13. import sklearn.datasets
14.
15. from opt_utils import load_params_and_grads, initialize_parameters, forward_propa
    gation, backward_propagation
16. from opt_utils import compute_cost, predict, predict_dec, plot_decision_boundary,
     load_dataset, load_dataset_signs
17. from opt_utils import convert_to_one_hot
18.
19.
20. #%matplotlib inline
21. plt.rcParams['figure.figsize'] = (7.0, 4.0) # set default size of plots
22. plt.rcParams['image.interpolation'] = 'nearest'
23. plt.rcParams['image.cmap'] = 'gray'
24.
25. def update_parameters_with_gd(parameters, grads, learning_rate):
26.     """
27.     Update parameters using one step of gradient descent
28.
29.     Arguments:
```

```
    parameters -- python dictionary containing your parameters to be updated:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -
- python dictionary containing your gradients to update each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    learning_rate -- the learning rate, scalar.

    Returns:
    parameters -- python dictionary containing your updated parameters
    """

    L = len(parameters) // 2 # number of layers in the neural networks

    # Update rule for each parameter
    for l in range(L):
        ### START CODE HERE ### (approx. 2 lines)
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate*g
rads['dW' + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate*g
rads['db' + str(l+1)]
        ### END CODE HERE ###

    return parameters


def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    """
    Creates a list of random minibatches from (X, Y)

    Arguments:
    X -- input data, of shape (input size, number of examples)
    Y -
- true "label" vector (1 for blue dot / 0 for red dot), of shape (1, number of ex
amples)
    mini_batch_size -- size of the mini-batches, integer

    Returns:
    mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
    """

    np.random.seed(seed)            # To make your "random" minibatches the same
as ours
    m = X.shape[1]                  # number of training examples
    mini_batches = []

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((Y.shape[0],m))

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini bat
ches of size mini_batch_size in your partitionning
    for k in range(0, num_complete_minibatches):
        ### START CODE HERE ### (approx. 2 lines)
        mini_batch_X = shuffled_X[:, k*mini_batch_size : (k+1)*mini_batch_size]
        mini_batch_Y = shuffled_Y[:, k*mini_batch_size : (k+1)*mini_batch_size]
        ### END CODE HERE ###
```

```
            mini_batch = (mini_batch_X, mini_batch_Y)
            mini_batches.append(mini_batch)

               # Handling the end case (last mini-batch < mini_batch_size)
        if m % mini_batch_size != 0:
            ### START CODE HERE ### (approx. 2 lines)
            mini_batch_X = shuffled_X[:, num_complete_minibatches*mini_batch_size:]
            mini_batch_Y = shuffled_Y[:, num_complete_minibatches*mini_batch_size:]
            ### END CODE HERE ###
            mini_batch = (mini_batch_X, mini_batch_Y)
            mini_batches.append(mini_batch)

        return mini_batches


    def initialize_velocity(parameters):
        """
        Initializes the velocity as a python dictionary with:
                    - keys: "dW1", "db1", ..., "dWL", "dbL"
                    - values: numpy arrays of zeros of the same shape as the correspo
    nding gradients/parameters.
        Arguments:
        parameters -- python dictionary containing your parameters.
                        parameters['W' + str(l)] = Wl
                        parameters['b' + str(l)] = bl

        Returns:
        v -- python dictionary containing the current velocity.
                        v['dW' + str(l)] = velocity of dWl
                        v['db' + str(l)] = velocity of dbl
        """

        L = len(parameters) // 2 # number of layers in the neural networks
        v = {}

        # Initialize velocity
        for l in range(L):
            ### START CODE HERE ### (approx. 2 lines)
            v["dW" + str(l+1)] = np.zeros(parameters['W'+str(l+1)].shape)
            v["db" + str(l+1)] = np.zeros(parameters['b'+str(l+1)].shape)
            ### END CODE HERE ###

        return v


    def update_parameters_with_momentum(parameters, grads, v, beta, learning_rate):
        """
        Update parameters using Momentum

        Arguments:
        parameters -- python dictionary containing your parameters:
                        parameters['W' + str(l)] = Wl
                        parameters['b' + str(l)] = bl
        grads -- python dictionary containing your gradients for each parameters:
                        grads['dW' + str(l)] = dWl
                        grads['db' + str(l)] = dbl
        v -- python dictionary containing the current velocity:
                        v['dW' + str(l)] = ...
                        v['db' + str(l)] = ...
        beta -- the momentum hyperparameter, scalar
        learning_rate -- the learning rate, scalar
```

```
144.
145.     Returns:
146.     parameters -- python dictionary containing your updated parameters
147.     v -- python dictionary containing your updated velocities
148.     """
149.
150.     L = len(parameters) // 2 # number of layers in the neural networks
151.
152.     # Momentum update for each parameter
153.     for l in range(L):
154.
155.         ### START CODE HERE ### (approx. 4 lines)
156.         # compute velocities
157.         v["dW" + str(l+1)] = beta * v['dW' + str(l+1)] + (1-
    beta)*(grads['dW' + str(l+1)])
158.         v["db" + str(l+1)] = beta * v['db' + str(l+1)] + (1-
    beta)*(grads['db' + str(l+1)])
159.         # update parameters
160.         parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - learning_rate *
     v['dW' + str(l+1)]
161.         parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - learning_rate *
     v['db' + str(l+1)]
162.         ### END CODE HERE ###
163.
164.     return parameters, v
165.
166.
167. def initialize_adam(parameters) :
168.     """
169.     Initializes v and s as two python dictionaries with:
170.                 - keys: "dW1", "db1", ..., "dWL", "dbL"
171.                 - values: numpy arrays of zeros of the same shape as the correspo
    nding gradients/parameters.
172.
173.     Arguments:
174.     parameters -- python dictionary containing your parameters.
175.                     parameters["W" + str(l)] = Wl
176.                     parameters["b" + str(l)] = bl
177.
178.     Returns:
179.     v -
    - python dictionary that will contain the exponentially weighted average of the g
    radient.
180.                     v["dW" + str(l)] = ...
181.                     v["db" + str(l)] = ...
182.     s -
    - python dictionary that will contain the exponentially weighted average of the s
    quared gradient.
183.                     s["dW" + str(l)] = ...
184.                     s["db" + str(l)] = ...
185.
186.     """
187.
188.     L = len(parameters) // 2 # number of layers in the neural networks
189.     v = {}
190.     s = {}
191.
192.     # Initialize v, s. Input: "parameters". Outputs: "v, s".
193.     for l in range(L):
194.     ### START CODE HERE ### (approx. 4 lines)
195.         v["dW" + str(l+1)] = np.zeros(parameters['W' + str(l+1)].shape)
```

```
        v["db" + str(l+1)] = np.zeros(parameters['b' + str(l+1)].shape)
        s["dW" + str(l+1)] = np.zeros(parameters['W' + str(l+1)].shape)
        s["db" + str(l+1)] = np.zeros(parameters['b' + str(l+1)].shape)
    ### END CODE HERE ###

    return v, s

def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate = 0.01,

                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8):
    """
    Update parameters using Adam

    Arguments:
    parameters -- python dictionary containing your parameters:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients for each parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    v -- Adam variable, moving average of the first gradient, python dictionary
    s -
- Adam variable, moving average of the squared gradient, python dictionary
    learning_rate -- the learning rate, scalar.
    beta1 -- Exponential decay hyperparameter for the first moment estimates
    beta2 -- Exponential decay hyperparameter for the second moment estimates
    epsilon -- hyperparameter preventing division by zero in Adam updates

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- Adam variable, moving average of the first gradient, python dictionary
    s -
- Adam variable, moving average of the squared gradient, python dictionary
    """

    L = len(parameters) // 2                 # number of layers in the neural net
works
    v_corrected = {}                         # Initializing first moment estimate
, python dictionary
    s_corrected = {}                         # Initializing second moment estimat
e, python dictionary

    # Perform Adam update on all parameters
    for l in range(L):
        # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v"
.
        ### START CODE HERE ### (approx. 2 lines)
        v["dW" + str(l+1)] = beta1 * v['dW' + str(l+1)] + (1-
beta1) * grads['dW' + str(l+1)]
        v["db" + str(l+1)] = beta1 * v['db' + str(l+1)] + (1-
beta1) * grads['db' + str(l+1)]
        ### END CODE HERE ###

        # Compute bias-
corrected first moment estimate. Inputs: "v, beta1, t". Output: "v_corrected".
        ### START CODE HERE ### (approx. 2 lines)
        v_corrected["dW" + str(l+1)] = v['dW' + str(l+1)] / (1 - np.power(beta1,
t))
        v_corrected["db" + str(l+1)] = v['db' + str(l+1)] / (1 - np.power(beta1,
t))
        ### END CODE HERE ###
```

```
245.
246.         # Moving average of the squared gradients. Inputs: "s, grads, beta2". Out
     put: "s".
247.         ### START CODE HERE ### (approx. 2 lines)
248.         s["dW" + str(l+1)] = beta2 * s['dW' + str(l+1)] + (1-
     beta2) * np.power(grads['dW' + str(l+1)], 2)
249.         s["db" + str(l+1)] = beta2 * s['db' + str(l+1)] + (1-
     beta2) * np.power(grads['db' + str(l+1)], 2)
250.         ### END CODE HERE ###
251.
252.         # Compute bias-
     corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_corrected
     ".
253.         ### START CODE HERE ### (approx. 2 lines)
254.         s_corrected["dW" + str(l+1)] = s['dW' + str(l+1)] / (1 - np.power(beta2,
     t))
255.         s_corrected["db" + str(l+1)] = s['db' + str(l+1)] / (1 - np.power(beta2,
     t))
256.         ### END CODE HERE ###
257.
258.         # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_c
     orrected, epsilon". Output: "parameters".
259.         ### START CODE HERE ### (approx. 2 lines)
260.         parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - learning_rate *
      v_corrected['dW' + str(l+1)] / np.sqrt(s_corrected['dW' + str(l+1)] + epsilon)
261.         parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - learning_rate *
      v_corrected['db' + str(l+1)] / np.sqrt(s_corrected['db' + str(l+1)] + epsilon)
262.         ### END CODE HERE ###
263.
264.     return parameters, v, s
265.
266.
267. def model(X, Y, layers_dims, optimizer, learning_rate = 0.0007, mini_batch_size =
      64, beta = 0.9,
268.           beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-
     8, num_epochs = 10000, print_cost = True):
269.     """
270.     3-layer neural network model which can be run in different optimizer modes.
271.
272.     Arguments:
273.     X -- input data, of shape (2, number of examples)
274.     Y -
     - true "label" vector (1 for blue dot / 0 for red dot), of shape (1, number of ex
     amples)
275.     layers_dims -- python list, containing the size of each layer
276.     learning_rate -- the learning rate, scalar.
277.     mini_batch_size -- the size of a mini batch
278.     beta -- Momentum hyperparameter
279.     beta1 -- Exponential decay hyperparameter for the past gradients estimates
280.     beta2 -
     - Exponential decay hyperparameter for the past squared gradients estimates
281.     epsilon -- hyperparameter preventing division by zero in Adam updates
282.     num_epochs -- number of epochs
283.     print_cost -- True to print the cost every 1000 epochs
284.
285.     Returns:
286.     parameters -- python dictionary containing your updated parameters
287.     """
288.
289.     L = len(layers_dims)             # number of layers in the neural networks
290.     costs = []                       # to keep track of the cost
```

```
    t = 0                            # initializing the counter required for Adam update
    seed = 10                        # For grading purposes, so that your "random" minibatches are the same as ours

    # Initialize parameters
    parameters = initialize_parameters(layers_dims)
     # Initialize the optimizer
    if optimizer == "gd":
        pass # no initialization required for gradient descent
    elif optimizer == "momentum":
        v = initialize_velocity(parameters)
    elif optimizer == "adam":
        v, s = initialize_adam(parameters)

    # Optimization loop
    for i in range(num_epochs):

        # Define the random minibatches. We increment the seed to reshuffle differently the dataset after each epoch
        seed = seed + 1
        minibatches = random_mini_batches(X, Y, mini_batch_size, seed)

        for minibatch in minibatches:

            # Select a minibatch
            (minibatch_X, minibatch_Y) = minibatch

            # Forward propagation
            a3, caches = forward_propagation(minibatch_X, parameters)

            # Compute cost
            cost = compute_cost(a3, minibatch_Y)
 # Backward propagation
            grads = backward_propagation(minibatch_X, minibatch_Y, caches)

            # Update parameters
            if optimizer == "gd":
                parameters = update_parameters_with_gd(parameters, grads, learning_rate)
            elif optimizer == "momentum":
                parameters, v = update_parameters_with_momentum(parameters, grads, v, beta, learning_rate)
            elif optimizer == "adam":
                t = t + 1 # Adam counter
                parameters, v, s = update_parameters_with_adam(parameters, grads, v, s,
                                                               t, learning_rate, beta1, beta2,  epsilon)

        # Print the cost every 1000 epoch
        if print_cost and i % 1000 == 0:
            #print(t)
            print ("Cost after epoch %i: %f" %(i, cost))
#       if print_cost and i % 10 == 0:
        if print_cost and i % 10 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(costs)
    plt.ylabel('cost')
```

```
    plt.xlabel('epochs (per 10)')
    plt.title("Learning rate = " + str(learning_rate))
    plt.show()

    return parameters, costs


# train 3-layer model
train_X, train_Y = load_dataset()

plt.scatter(train_X[0, :], train_X[1, :],c=train_Y[0,:], s=40, cmap=plt.cm.Spectral)
plt.xlabel("X[0,:]")
plt.ylabel("X[1,:]")
plt.title("data visualization, red: y=0, blue: y=1")
plt.show()

layers_dims = [train_X.shape[0], 5, 2, 2]
train_Y_hot = convert_to_one_hot(train_Y, 2)
#     model(X, Y, layers_dims, optimizer, learning_rate = 0.0007, mini_batch_size = 64, beta = 0.9,
#           beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 10000, print_cost = True)
#

parameters, costs=model(train_X, train_Y_hot, layers_dims, optimizer="momentum", learning_rate = 0.1, mini_batch_size = 64, beta = 0.9,
          beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 1000, print_cost = True)
# Predict
predictions = predict(train_X, parameters)
accuracy = np.mean(predictions == train_Y)

print("accuracy of train set is "+ str(accuracy))
# Predict
#predictions = predict(train_X, train_Y, parameters)
 # Plot decision boundary
plt.title("Model with momentum optimization")
axes = plt.gca()
axes.set_xlim([-1.5,2.5])
axes.set_ylim([-1,1.5])
plot_decision_boundary(lambda x: predict(x.T, parameters), train_X, train_Y)


# multiple class classification

np.random.seed(1)
# Loading the dataset
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset_signs()
# Example of a picture
index = 0
plt.imshow(X_train_orig[index])
plt.show()
print ("y = " + str(np.squeeze(Y_train_orig[:, index])))
# Flatten the training and test images
X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T
X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1).T
# Normalize image vectors
X_train = X_train_flatten/255.
```

```
X_test = X_test_flatten/255.
# Convert training and test labels to one hot matrices
Y_train = convert_to_one_hot(Y_train_orig, 6)
Y_test = convert_to_one_hot(Y_test_orig, 6)

print ("number of training examples = " + str(X_train.shape[1]))
print ("number of test examples = " + str(X_test.shape[1]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))

# train 3-layer model
layers_dims = [X_train.shape[0], 25, 12, 6]
#parameters, costs = model(X_train, Y_train, layers_dims, optimizer="adam", learning_rate = 0.0001, mini_batch_size = 32, num_epochs = 1500)
parameters, costs=model(X_train, Y_train, layers_dims, optimizer="adam", learning_rate = 0.0001, mini_batch_size = 32, beta = 0.9,
          beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 1500, print_cost = True)
# Predict
predictions = predict(X_train, parameters)
accuracy = np.mean(predictions == Y_train_orig)

print("accuracy of train set is "+ str(accuracy))

predictions = predict(X_test, parameters)
accuracy = np.mean(predictions == Y_test_orig)

print("accuracy of test set is "+ str(accuracy))
```

9.4.2 Binary classification for two-feature inputs

We apply the model to classify the data examples shown in Fig.6. To do this, we choose to assign layers_dims = [train_X.shape[0], 5, 2, 2], and then instantiate the model by

```
parameters, costs=model(train_X, train_Y_hot, layers_dims, optimizer="gd", learning_rate = 0.1, mini_batch_size = 64, beta = 0.9, beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 1000, print_cost = True)
```

Of course, we can tune the hyperparameters by changing their settings.

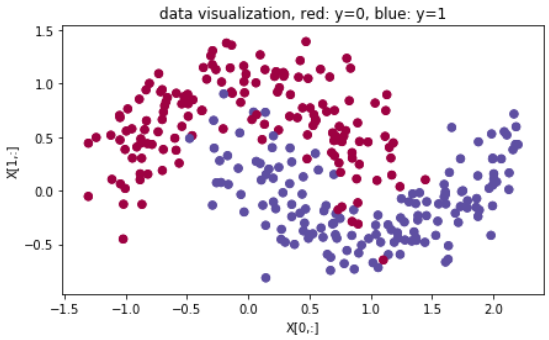
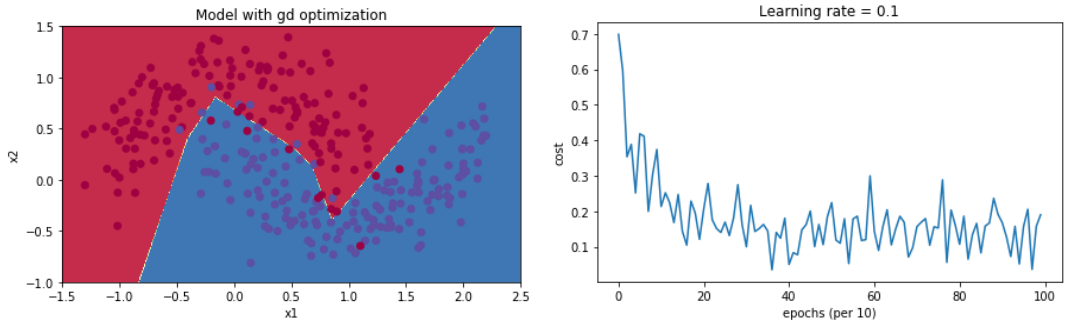
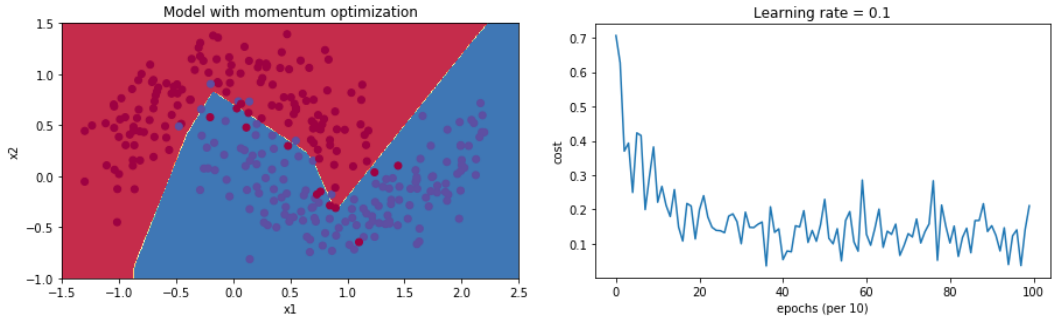


Fig.6 training examples generated by load_dataset()

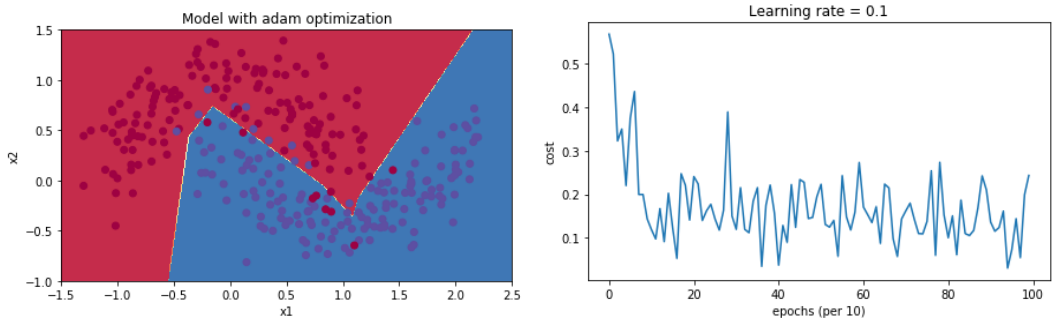
Fig. 7 shows the results using three different optimization methods, based on the above hyperparameter setting. With the current setting, three methods have similar performance. However, a meaningful comparsion requires various combinations of hyperparameter values.



a) Gradient descent: accuracy=0.947



b) Momentum: accuracy=0.94



c) Adam: accuracy =0.937

Fig.7 Results of different optimization methods

9.4.3 6-class classification for 12288-feature inputs: finger-signs

Suppose that we use finger-signs to send 6 digits for a communication purpose. Some example signs are shown in Fig.8. Each sign is represented by (64x64)-pixel color picture with a certain combination of finger positions. We will build a neural network to translate a sign to a digit. Thus, this model classifies six classes. The dataset files are train_signs.h5 (1080 images for training) and test_signs.h5 (120 images for testing).

To do this, we instantiate the model by

```
layers_dims = [X_train.shape[0], 25, 12, 6]

parameters,   costs=model(X_train,   Y_train,   layers_dims,   optimizer="adam",
learning_rate = 0.0001, mini_batch_size = 32, beta = 0.9, beta1 = 0.9, beta2 =
0.999,  epsilon = 1e-8, num_epochs = 1500, print_cost = True)
```

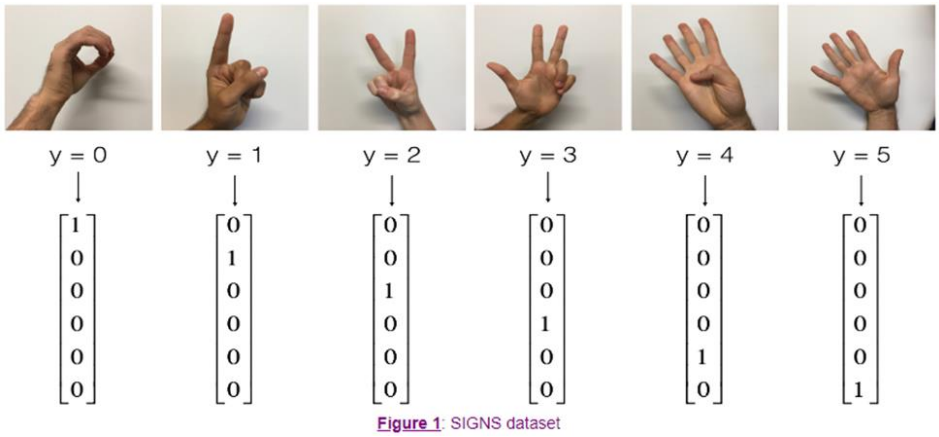


Fig.8 sign-digit mapping

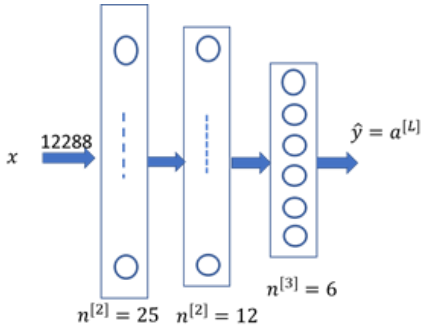
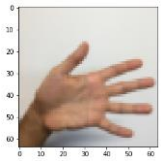


Fig.9 Architecture of neural network for finger sign recognition

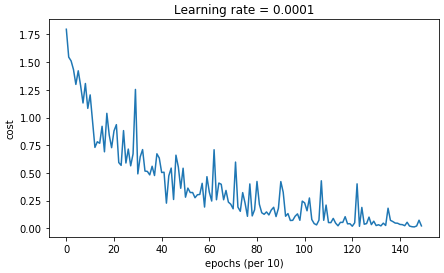
By running ch9.py, we got the following results:



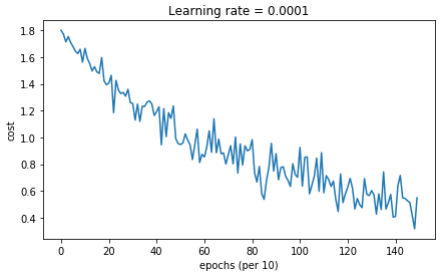
```
y = 5

number of training examples = 1080
number of test examples = 120
X_train shape: (12288, 1080)
Y_train shape: (6, 1080)
X_test shape: (12288, 120)
Y_test shape: (6, 120)
```

Here we change optimizer and α. The results for some combinations are shown below Fig.10:

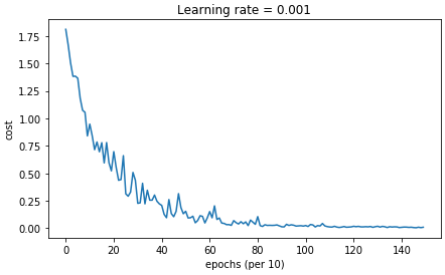
Optimizer=”adam”, α=0.0001, mini-batch=32
accuracy of train set is 0.9953703703703703
accuracy of test set is 0.7833333333333333



Optimizer=”gd”, α=0.0001, mini-batch=32
accuracy of train set is 0.887962962962963
accuracy of test set is 0.8083333333333333

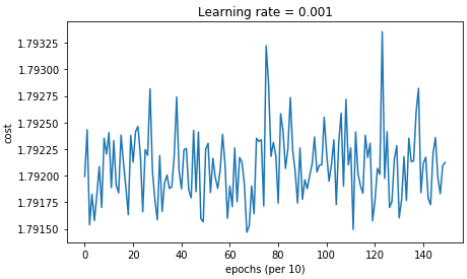


Optimizer=”gd”, α=0.001, mini-batch=32
accuracy of train set is 1.0
accuracy of test set is 0.8916666666666667



Optimizer=”adam”, α=0.001, mini-batch=32
accuracy of train set is 0.16666666666666666
accuracy of test set is 0.16666666666666666

Note: This setting does not work!



Optimizer =”adam”,α=0.001,mini-batch=1080 (batch)
accuracy of train set is 0.9861111111111112
accuracy of test set is 0.8416666666666667

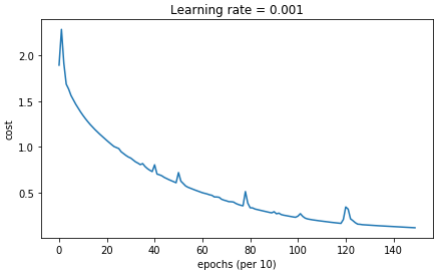


Fig.10 Results for some different settings with fixed arguments: `beta = 0.9, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, num_epochs = 1500.`

Based on the above trials, we realize that Adam may not work if other parameters are not correctly set. Therefor, it is essential to monitor the cost function to make sure the optimization method behaves correctly. In general, the effectiveness of an optimization method may be very sensitive to some hyperparameters. Thus, it is usually necessary to explore the design space of hyperparameters to search for a reasonably satisfied performance.

Summary

To digest the previous chapter (chapter 8), this chapter gives a comprehensive tutorial on mini-batch and Adam optimization through a project. The results of the project show that it is important to plot cost function curve to monitor the behavior of optimizer, and that selection of hyperparameters is essential.

Files:

\machine_learning\NN_m_class\opt_utils.py
\machine_learning\NN_m_class\ch9.py
\machine_learning\NN_m_class\train_signs.h5
\machine_learning\NN_m_class\test_signs.h5

References

[1] "Neural networks and deep learning" online course at www.coursera.org
[2]

Exercises

1. Based on the results shown in Fig.10, the best result has been achieved when `Optimizer="gd", α=0.001, mini-batch=32, (fixed: beta = 0.9, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, num_epochs = 1500).` The resulting training accuracy is 1 and testing accuracy is 0.89. Try different augment assignments to see whether you can find a case works better than this one.

2. Based on the results shown in Fig.10, for most working cases, the training accuracy is higher than testing accuracy by 10%. For example, the case with: `Optimizer="gd", α=0.001, mini-batch=32, (fixed: beta = 0.9, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, num_epochs = 1500),` achieves a training accuracy of 1, and testing accuracy of 0.89. An overfitting probably exists. Please modify opt_utils.py and/or ch9.py to introduce weight regularization to see if you can improve the testing accuracy.

3. Try different neural networks with different sizes by change layers_dims value. Note that the neural network should be still 3-layer.

   layers_dims=[12288, n_1, n_2, 6]

   The results in Fig.10 are based on *layers_dims=[12288, 25, 12, 6]*. Now you change numbers "25" and "12" to create a different neural network, apply the new neural network to the finger sign dataset with similar setting in Fig.10. Compare your results with Fig.10.