Project:

The project was to use different optimization methods (to update the parameters and minimize the cost) and compare their accuracy rate. Accuracy with Adam optimization method was highest.

Optimization methods used:

- Gradient descent
- Mini batch gradient descent
- Gradient descent with momentum
- Adam

Gradient descent

```
def update_parameters_with_gd(parameters, grads, learning_rate):
    """
    Update parameters using one step of gradient descent

    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):
        parameters["W" + str(l+1)] =
parameters["W"+str(l+1)]=parameters["W"+str(l+1)]-learning_rate*grads['dW'+str(l+1)]
        parameters["b" + str(l+1)] =
parameters["b"+str(l+1)]=parameters["b"+str(l+1)]-learning_rate*grads['db'+str(l+1)]
        return parameters
```

Mini batch gradient descent

```
def random_mini_batches(X, Y, mini_batch_size = 64):
    """
    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 examples)
    mini_batch_size -- size of the mini-batches, integer

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

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size) # number of mini batches of size mini_batch_size in your partitionning
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[:, k : mini_batch_size]
        mini_batch_Y = shuffled_Y[:,k:mini_batch_size]
        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:
        mini_batch_X = shuffled_X[:, num_complete_minibatches : mini_batch_size]
        mini_batch_Y = shuffled_Y[:, num_complete_minibatches : mini_batch_size]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)
    return mini_batches
```

Gradient descent with momentum

```
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 corresponding
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):
        v["dW" + str(l+1)] = np.zeros_like(parameters['W'+str(l+1)])
        v["db" + str(l+1)] = np.zeros_like(parameters['b'+str(l+1)])
    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

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- python dictionary containing your updated velocities
    """
    L = len(parameters) // 2 # number of layers in the neural networks
    # Momentum update for each parameter
    for l in range(L):
        # compute velocities
        v["dW" + str(l+1)] = beta*v["dW"+str(l+1)]+(1-beta)*grads['dW'+str(l+1)]
        v["db" + str(l+1)] = beta*v["db"+str(l+1)]+(1-beta)*grads['db'+str(l+1)]
        # update parameters
        parameters["W" + str(l+1)] = parameters["W"+str(l+1)]-learning_rate*v["dW"+str(l+1)]
        parameters["b" + str(l+1)] = parameters["b"+str(l+1)]-learning_rate*v["db"+str(l+1)]
```

return parameters, v

--

Adam

def initialize_adam(parameters) :

"""

Initializes v and s as two python dictionaries with:

- keys: "dW1", "db1", ..., "dWL", "dbL"

- values: numpy arrays of zeros of the same shape as the corresponding gradients/parameters.

Arguments:

parameters -- python dictionary containing your parameters.

parameters["W" + str(l)] = Wl

parameters["b" + str(l)] = bl

Returns:

v -- python dictionary that will contain the exponentially weighted average of the gradient.

v["dW" + str(l)] = ...

v["db" + str(l)] = ...

s -- python dictionary that will contain the exponentially weighted average of the squared gradient.

s["dW" + str(l)] = ...

s["db" + str(l)] = ...

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

    # Initialize v, s. Input: "parameters". Outputs: "v, s".
    for l in range(L):
        v["dW" + str(l+1)] = np.zeros_like(parameters["W"+str(l+1)])
        v["db" + str(l+1)] = np.zeros_like(parameters["b"+str(l+1)])
        s["dW" + str(l+1)] = np.zeros_like(parameters["W"+str(l+1)])
        s["db" + str(l+1)] = np.zeros_like(parameters["b"+str(l+1)])
    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 networks
    v_corrected = {}                        # Initializing first moment estimate, python
dictionary
    s_corrected = {}                        # Initializing second moment estimate, python
dictionary
```

```
    # Perform Adam update on all parameters
    for l in range(L):
        # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v".
        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)]
        # Compute bias-corrected first moment estimate. Inputs: "v, beta1, t". Output: "v_corrected".
        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))
        # Moving average of the squared gradients. Inputs: "s, grads, beta2". Output: "s".
        s["dW" + str(l+1)] = beta2*s["dW"+str(l+1)]+(1-beta2)*np.power((grads['dW'+str(l+1)]),2)
        s["db" + str(l+1)] = beta2*s["db"+str(l+1)]+(1-beta2)*np.power(grads['db'+str(l+1)],2)
        # Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_corrected".
        s_corrected["dW" + str(l+1)] = s["dW"+str(l+1)]/(1-np.power(beta2,t))
        s_corrected["db" + str(l+1)] = s["db"+str(l+1)]/(1-np.power(beta2,t))
        # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon". Output: "parameters".
```

```
        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)

        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)

    return parameters, v, s
```