§ Week 2 Assignment — Optimization Methods

1. Gradient Descent

- **(Batch) Gradient Descent:**

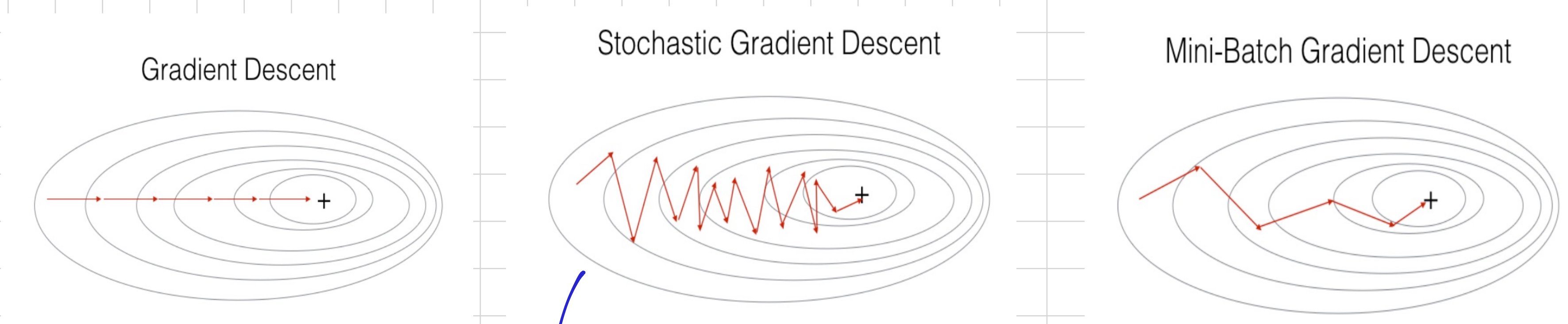
```python
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost = compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

→ 1 batch, mini-batch = m

- **Stochastic Gradient Descent:**

```python
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost = compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

→ m batch, each mini-batch has 1 example



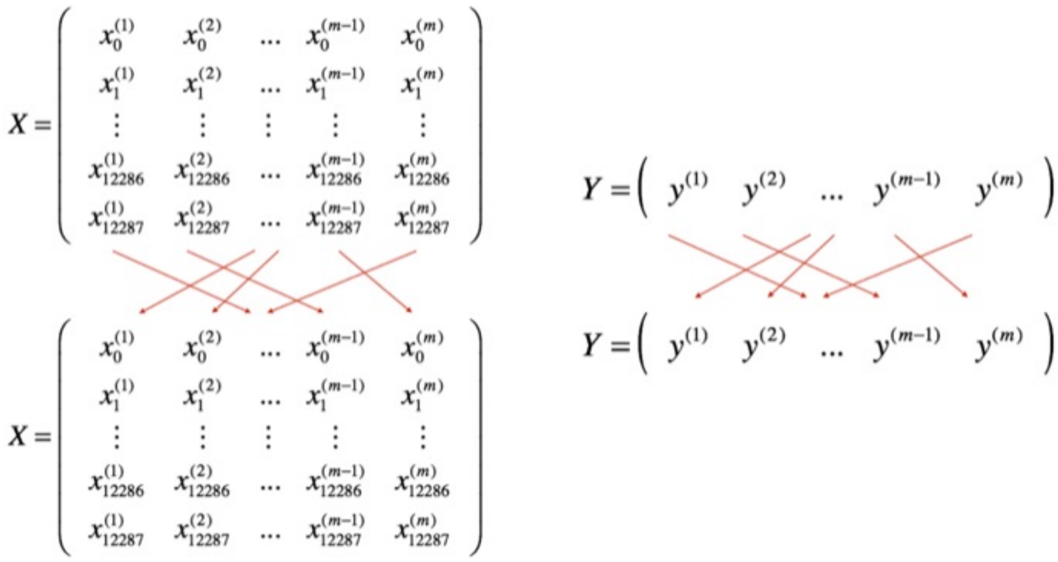
SGD leads to many oscillations to reach convergence.
but each step is a lot faster for SGD than for GD

SGD for-loops:

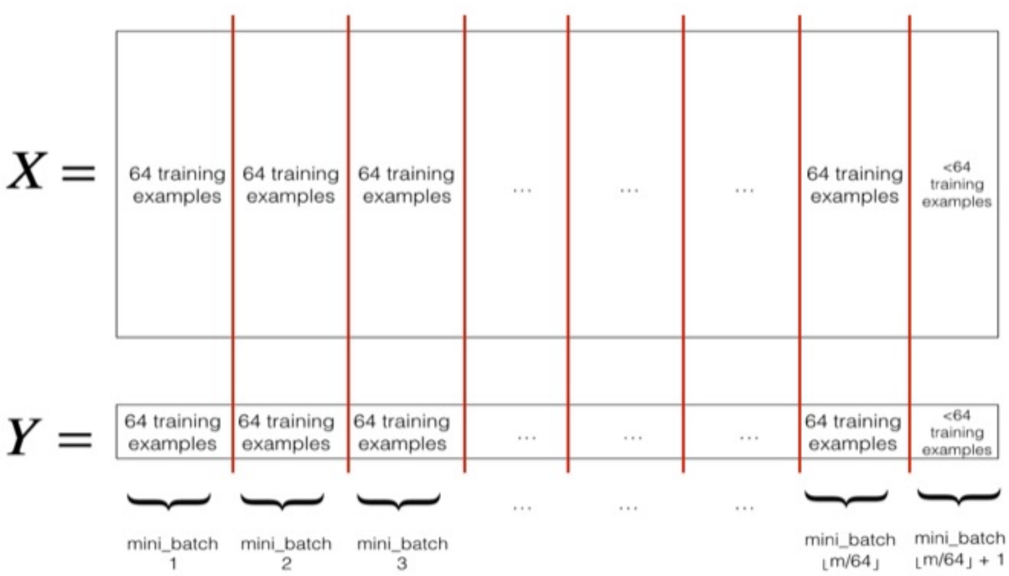
① Over the number of iterations
② Over the m training examples
③ Over the layers:
To update all parameters, from $(W^{[1]}, b^{[1]})$ to $(W^{[L]}, b^{[L]})$

2. Mini-Batch Gradient Descent

- **Shuffle**: Create a shuffled version of the training set (X, Y) as shown below. 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 i^{th} column of X is the example corresponding to the i^{th} label in Y. The shuffling step ensures that examples will be split randomly into different mini-batches.



- **Partition**: Partition the shuffled (X, Y) into mini-batches of size `mini_batch_size` (here 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:




```python
# GRADED FUNCTION: random_mini_batches

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 examples)
    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((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):
        ### 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
```

mini-batch $\left\lfloor \frac{m}{\text{mini-batch size}} \right\rfloor$

1st 0~ mini-batch size

2nd mini-batch size ~ 2*mini-batch size

...

ith (i-1)*mini-batch size ~ i*mini-batch size

...

last (kth) k*mini-batch size ~

usually choose 2^n as mini-batch size,

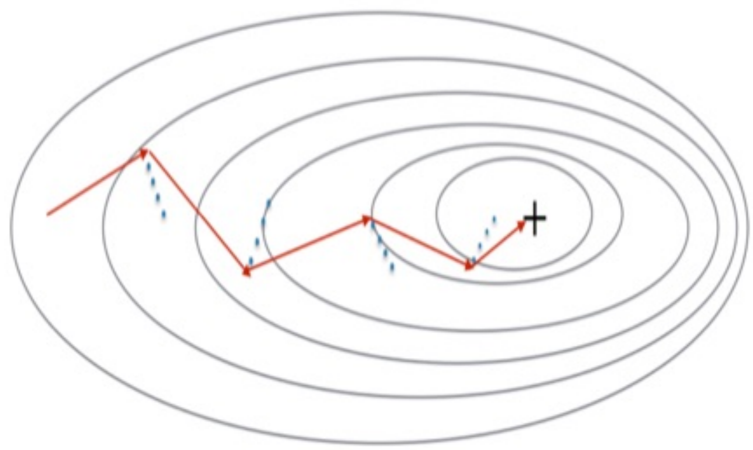
eg. 2, 4, 8, 16, 32, ...

3. momentum

take past gradients into account to smooth out the steps of gradient descent

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 "oscillate" toward convergence. Using momentum can reduce these oscillations.

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. You can also think of v as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.



(1) initialize velocity

```
# 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 ###
```

(2) update parameters

```
# Momentum update for each parameter
for l in range(L):

    ### START CODE HERE ### (approx. 4 lines)
    # 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
)]
    ### END CODE HERE ###
```

$$V_{dw}^{[l]} = \beta V_{dw}^{[l]} + (1-\beta)\, dw^{[l]}$$
$$w^{[l]} = w^{[l]} - \alpha V_{dw}^{[l]}$$
$$V_{db}^{[l]} = \beta V_{db}^{[l]} + (1-\beta)\, db^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha V_{db}^{[l]}$$

① if $\beta = 0$, this just becomes standard gradient descent without momentum

② the larger the momentum β is, the smoother the update because the more we take the last gradient into account.

③ Common values for β range from 0.8 to 0.999

4. Adam

How does Adam work?

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".

The update rule is, for $l = 1, ..., L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1)\frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s^{corrected}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1 - (\beta_1)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}} + \varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 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

As usual, we will store all parameters in the `parameters` dictionary

(1) initialize

```
# Initialize v, s. Input: "parameters". Outputs: "v, s".
for l in range(L):
### START CODE HERE ### (approx. 4 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)
    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 ###
```

$$V_{dw^{[l]}} = \beta_1 V_{dw^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial w^{[l]}}$$

$$V_{db^{[l]}} = \beta_1 V_{db^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial b^{[l]}}$$

(2) update

```
# 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_correcte
d".
    ### 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 ###

    # Moving average of the squared gradients. Inputs: "s, grads, beta2". Output: "s".
    ### START CODE HERE ### (approx. 2 lines)
    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)
    ### END CODE HERE ###

    # Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_cor
rected".
    ### START CODE HERE ### (approx. 2 lines)
    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))
    ### END CODE HERE ###

    # Update parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon
". Output: "parameters".
    ### START CODE HERE ### (approx. 2 lines)
    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)
    ### END CODE HERE ###
```

$$V^{corrected}_{dw^{[l]}} = \frac{V_{dw^{[l]}}}{1-(\beta_1)^t}$$

$$V^{corrected}_{db^{[l]}} = \frac{V_{db^{[l]}}}{(1-\beta_1)^t}$$

$$S_{dw^{[l]}} = \beta_2 S_{dw^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial w^{[l]}})^2$$

$$S_{db^{[l]}} = \beta_2 S_{db^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial b^{[l]}})^2$$

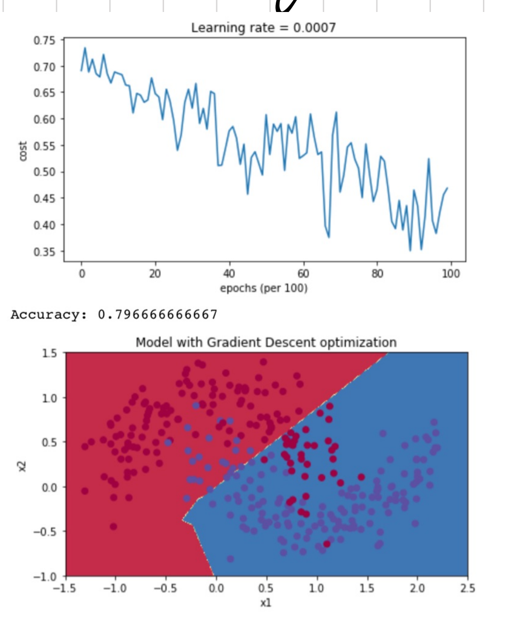
$$S^{corrected}_{dw^{[l]}} = \frac{S_{dw^{[l]}}}{1-(\beta_2)^t}$$

$$S^{corrected}_{db^{[l]}} = \frac{S_{db^{[l]}}}{1-(\beta_2)^t}$$

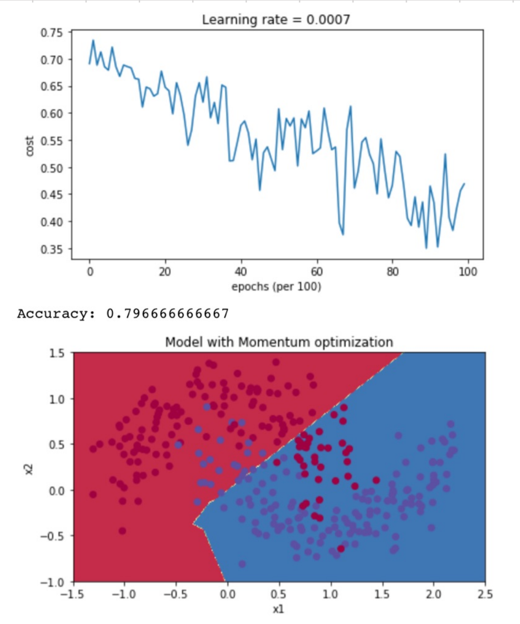
$$W^{[l]} = W^{[l]} - \alpha \frac{V^{corrected}_{dw^{[l]}}}{\sqrt{S^{corrected}_{dw^{[l]}}} + \varepsilon}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{V^{corrected}_{db^{[l]}}}{\sqrt{S^{corrected}_{db^{[l]}}} + \varepsilon}$$

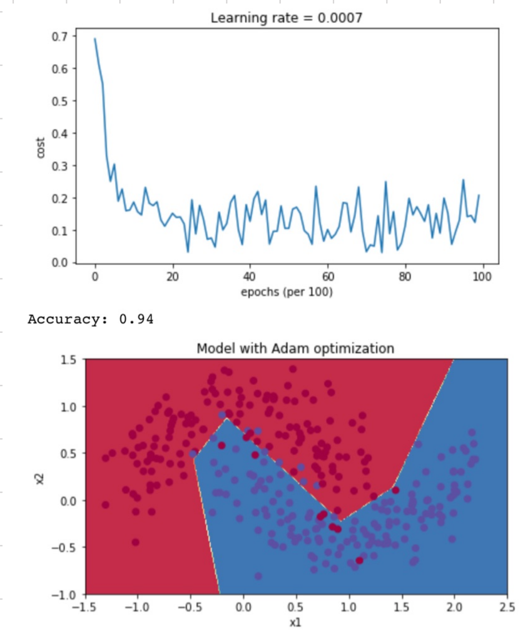
5. compare different models



mini-batch GD



mini-batch GD with momentum



mini-batch GD with Adam

optimization method	**accuracy**	**cost shape**
Gradient descent	79.7%	oscillations
Momentum	79.7%	oscillations
Adam	94%	smoother

Momentum usually helps, but given the small learning rate and the simplistic dataset, its impact is almost negligeable. Also, the huge oscillations you see in the cost come from the fact that some minibatches are more difficult thans others for the optimization algorithm.

Adam on the other hand, clearly outperforms mini-batch gradient descent and Momentum. If you run the model for more epochs on this simple dataset, all three methods will lead to very good results. However, you've seen that Adam converges a lot faster.

Some advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters (except α)