Week 2 - prog ex: Mini-batch, Adam & RMSprop & Momentum

笔记本: DL 2 - Deep NN Hyperparameter Tunning, Regularization & Optimization
创建时间: 2021/1/9 11:56 **更新时间:** 2021/1/9 12:33

Welcome to the optimization's programming assignment of the hyper-parameters tuning specialization. There are many different optimization algorithms you could be using to get you to the minimal cost. Similarly, there are many different paths down this hill to the lowest point.

By completing this assignment you will:

- Understand the intuition between Adam and RMS prop

- Recognize the importance of mini-batch gradient descent

- Learn the effects of momentum on the overall performance of your model

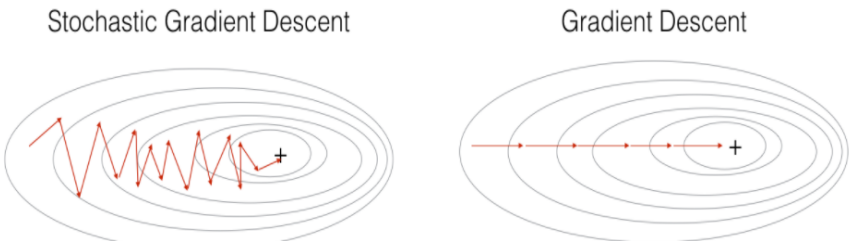
- **(Batch) Gradient Descent**:

```
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)
```

- **Stochastic Gradient Descent**:

```
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)
```

In Stochastic Gradient Descent, you use only 1 training example before updating the gradients. When the training set is large, SGD can be faster. But the parameters will "oscillate" toward the minimum rather than converge smoothly. Here is an illustration of this:



In practice, you'll often get faster results if you do not use neither the whole training set, nor only one training example, to perform each update. Mini-batch gradient descent uses an intermediate number of examples for each step. With mini-batch gradient descent, you loop over the mini-batches instead of looping over individual training examples.

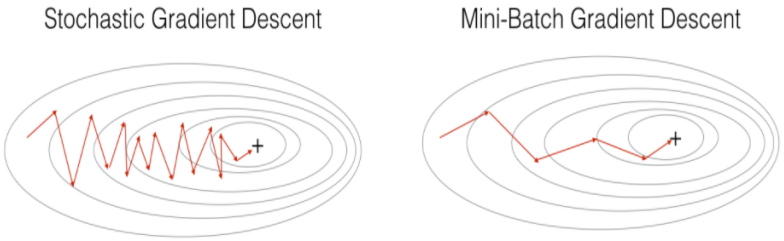


Figure 2 : **SGD vs Mini-Batch GD**
"+" denotes a minimum of the cost. Using mini-batches in your optimization algorithm often leads to faster optimization.

What you should remember:

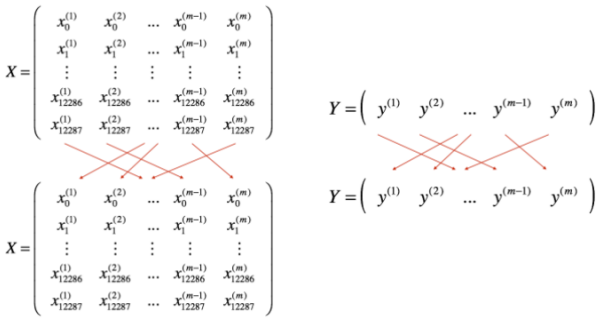
- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter α.
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

2 - Mini-Batch Gradient descent

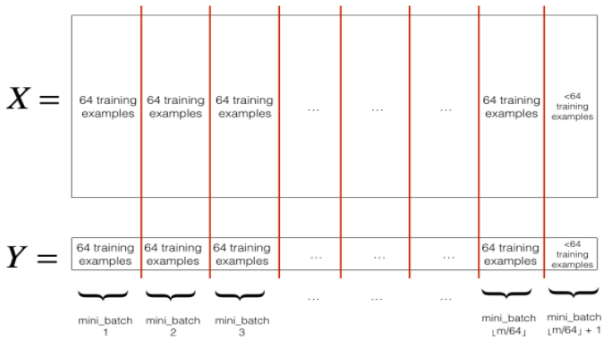
Let's learn how to build mini-batches from the training set (X, Y).

There are two steps:

- **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:



Exercise: Implement `random_mini_batches`. We coded the shuffling part for you. To help you with the partitioning step, we give you the following code that selects the indexes for the 1^{st} and 2^{nd} mini-batches:

```
first_mini_batch_X = shuffled_X[:, 0 : mini_batch_size]
second_mini_batch_X = shuffled_X[:, mini_batch_size : 2 * mini_batch_size]
```

Note that the last mini-batch might end up smaller than `mini_batch_size=64`. Let $\lfloor s \rfloor$ represents s rounded down to the nearest integer (this is `math.floor(s)` in Python). If the total number of examples is not a multiple of `mini_batch_size=64` then there will be $\lfloor \frac{m}{mini\_batch\_size} \rfloor$ mini-batches with a full 64 examples, and the number of examples in the final mini-batch will be $(m - mini\_batch\_size \times \lfloor \frac{m}{mini\_batch\_size} \rfloor)$.

What you should remember:

- Shuffling and Partitioning are the two steps required to build mini-batches
- Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

3 - 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 "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.

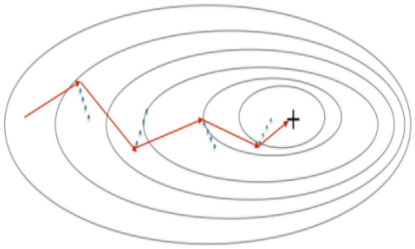


Figure 3: The red arrows shows the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step. Rather than just following the gradient, we let the gradient influence v and then take a step in the direction of v.

Exercise: Initialize the velocity. The velocity, v, is a python dictionary that needs to be initialized with arrays of zeros. Its keys are the same as those in the `grads` dictionary, that is: for $l = 1, \ldots, L$:

```
v["dW" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["W" + str(l+1)])
v["db" + str(l+1)] = ... #(numpy array of zeros with the same shape as parameters["b" + str(l+1)])
```

Note that the iterator l starts at 0 in the for loop while the first parameters are v["dW1"] and v["db1"] (that's a "one" on the superscript). This is why we are shifting l to l+1 in the `for` loop.

Exercise: Now, implement the parameters update with momentum. The momentum update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1 - \beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases} \tag{3}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1 - \beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases} \tag{4}$$

where L is the number of layers, β is the momentum and α is the learning rate. All parameters should be stored in the `parameters` dictionary. Note that the iterator `l` starts at 0 in the `for` loop while the first parameters are $W^{[1]}$ and $b^{[1]}$ (that's a "one" on the superscript). So you will need to shift `l` to `l+1` when coding.

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

# 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 ###
```

Note that:

- 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 $\beta = 0$, then this just becomes standard gradient descent without momentum.

How do you choose β?

- 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, $\beta = 0.9$ 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.

What you should remember:

- 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 have to tune a momentum hyperparameter β and a learning rate α.

4 - Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) and Momentum.

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 $l = 1, \ldots, L$:

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

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

Exercise: Initialize the Adam variables v, s which keep track of the past information.

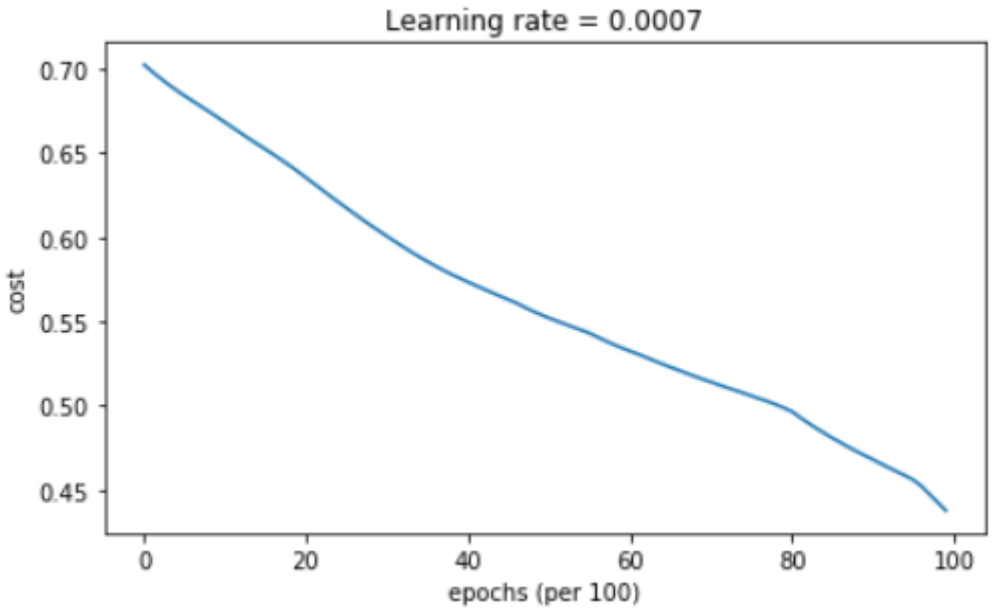
```
        # Perform Adam update on all parameters
        for l in range(L):
            # Moving average of the gradients. Inputs: "v, grads, beta1". Output: "v".
            ### START CODE HERE ### (approx. 2 lines)
            v["dW" + str(l+1)] = beta1 * v["dW" + str(l+1)] + (1 - beta1) * grads["dW" + 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 - beta1**(l+1))
            v_corrected["db" + str(l+1)] = v["db" + str(l+1)] / (1 - beta1**(l+1))
            ### 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.square(grads["dW" + str(l+1)])
            s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1 - beta2) * np.square(grads["db" + str(l+1)])
            ### END CODE HERE ###

            # Compute bias-corrected second raw moment estimate. Inputs: "s, beta2, t". Output: "s_corrected".
            ### START CODE HERE ### (approx. 2 lines)
            s_corrected["dW" + str(l+1)] = s["dW" + str(l+1)] / (1 - beta2**(l+1))
            s_corrected["db" + str(l+1)] = s["db" + str(l+1)] / (1 - beta2**(l+1))
            ### 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["dW" + str(l+1)])
            parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * v_corrected["db" + str(l+1)] / np.sqrt(s["db" + str(l+1)])
            ### END CODE HERE ###

        return parameters, v, s
```

We have already implemented a 3-layer neural network. You will train it with:

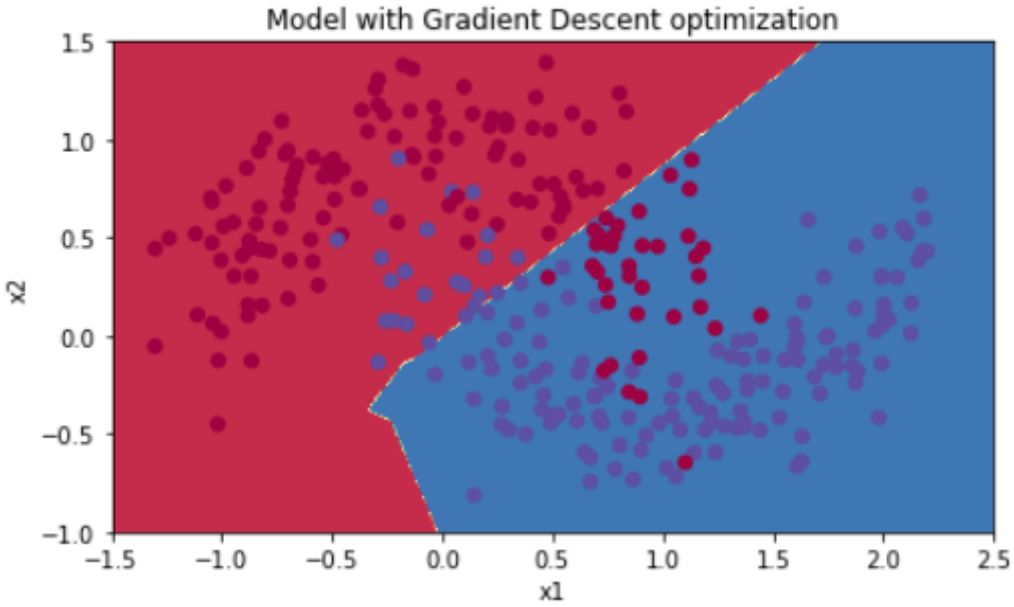
- Mini-batch **Gradient Descent**: it will call your function:
  - `update_parameters_with_gd()`
- Mini-batch **Momentum**: it will call your functions:
  - `initialize_velocity()` and `update_parameters_with_momentum()`
- Mini-batch **Adam**: it will call your functions:
  - `initialize_adam()` and `update_parameters_with_adam()`

5.1 - Mini-batch Gradient descent

Run the following code to see how the model does with mini-batch gradient descent.

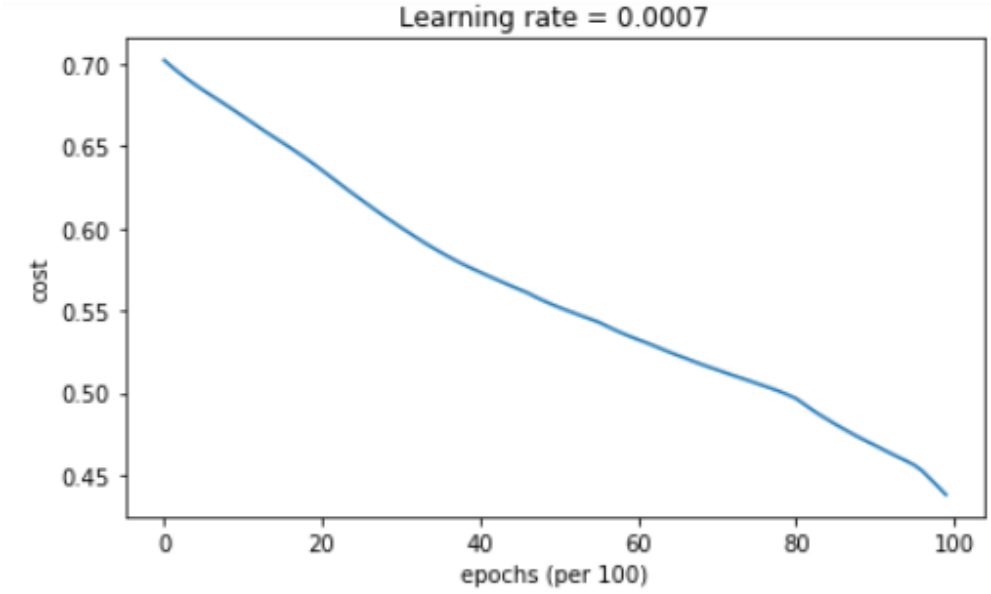


```
Accuracy: 0.796666666667
```

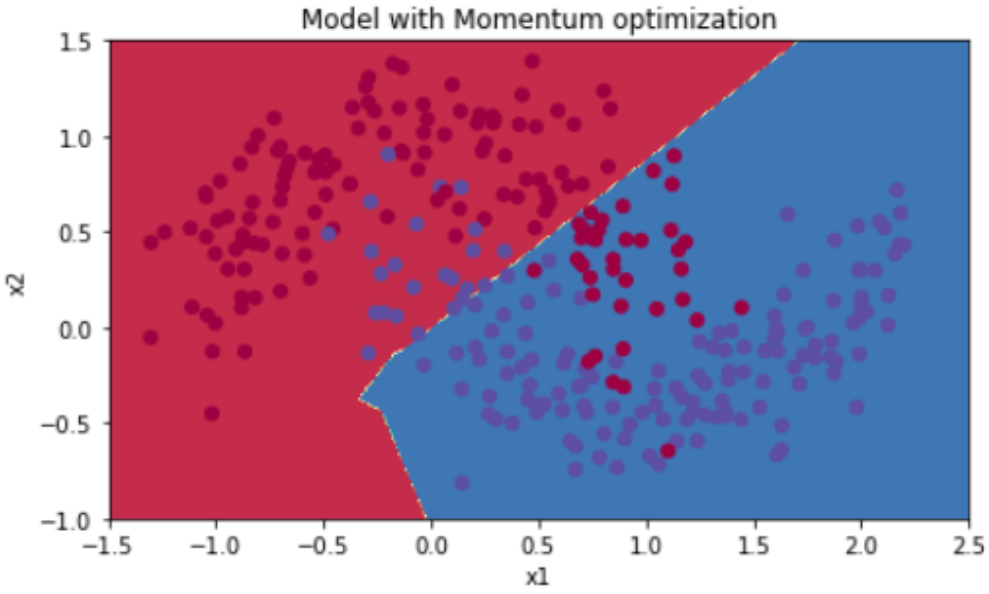


5.2 - Mini-batch gradient descent with momentum ¶

Run the following code to see how the model does with momentum. Because more complex problems you might see bigger gains.




```
Accuracy: 0.796666666667
```



DO NOT forget about the episilon for Adam(division)

```
pdate parameters. Inputs: "parameters, learning_rate, v_corrected, s_corrected, epsilon". Output: "parameters".
 START CODE HERE ### (approx. 2 lines)
ameters["W" + str(1+1)] = parameters["W" + str(1+1)] - learning_rate * v_corrected["dW" + str(1+1)] / (np.sqrt(s["dW" + str(1+1)]) + epsilon)
ameters["b" + str(1+1)] = parameters["b" + str(1+1)] - learning_rate * v_corrected["db" + str(1+1)] / (np.sqrt(s["db" + str(1+1)]) + epsilon)
 END CODE HERE ###
```

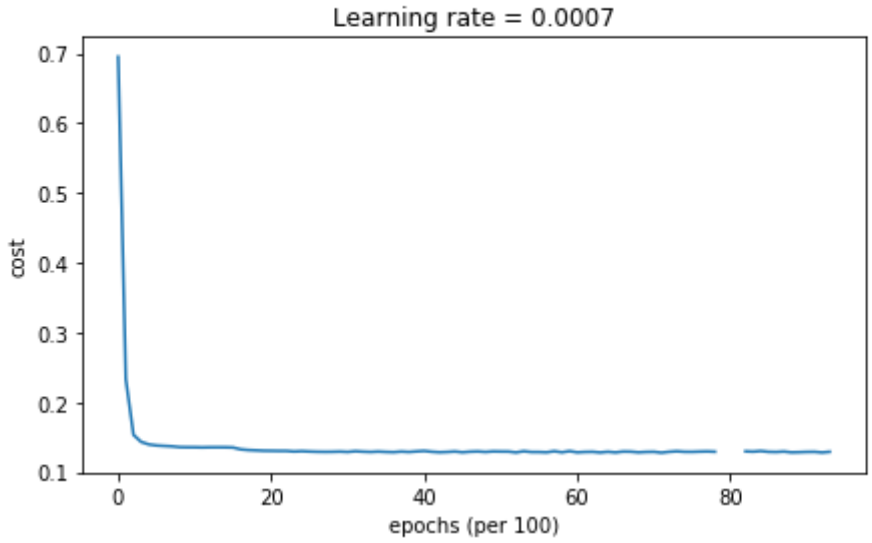
5.3 - Mini-batch with Adam mode

Run the following code to see how the model does with Adam.

In [50]:
```
# train 3-layer model
layers_dims = [train_X.shape[0], 5, 2, 1]
parameters = model(train_X, train_Y, layers_dims, optimizer = "adam")

# Predict
predictions = predict(train_X, train_Y, parameters)

# Plot decision boundary
plt.title("Model with Adam optimization")
axes = plt.gca()
axes.set_xlim([-1.5,2.5])
axes.set_ylim([-1,1.5])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```

```
Cost after epoch 0: 0.695414
Cost after epoch 1000: 0.135551
Cost after epoch 2000: 0.130452
Cost after epoch 3000: 0.128858
Cost after epoch 4000: 0.130299
Cost after epoch 5000: 0.129453
Cost after epoch 6000: 0.128204
Cost after epoch 7000: 0.129057
```

```
Cost after epoch 8000: nan
Cost after epoch 9000: 0.128871
```




```
Accuracy: 0.93
```

5.4 - Summary

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)

Adam paper

https://arxiv.org/pdf/1412.6980.pdf