## 2 - Building blocks of the model

In this part, you will build two important blocks of the overall model:

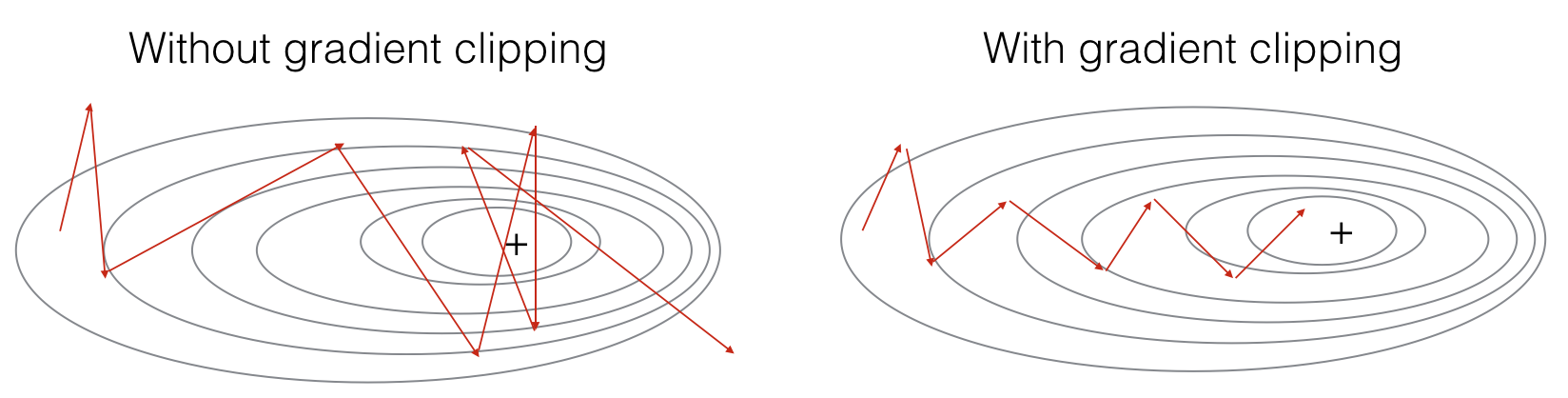
- Gradient clipping: to avoid exploding gradients

- Sampling: a technique used to generate characters

### **2.1 - Clipping the gradients in the optimization loop**

In this section you will implement the clip function that you will call inside of your optimization loop. Recall that your overall loop structure usually consists of a forward pass, a cost computation, a backward pass, and a parameter update. **Before updating the parameters, you will perform gradient clipping when needed to make sure that your gradients are not "exploding," meaning taking on overly large values.**

In the exercise below, you will implement a function clip that takes in a dictionary of gradients and returns a clipped version of gradients if needed. There are different ways to clip gradients; we will use a simple element-wise clipping procedure, in which every element of the gradient vector is clipped to lie between some range [-N, N]. More generally, you will provide a maxValue (say 10). In this example, if any component of the gradient vector is greater than 10, it would be set to 10; and if any component of the gradient vector is less than -10, it would be set to -10. If it is between -10 and 10, it is left alone.

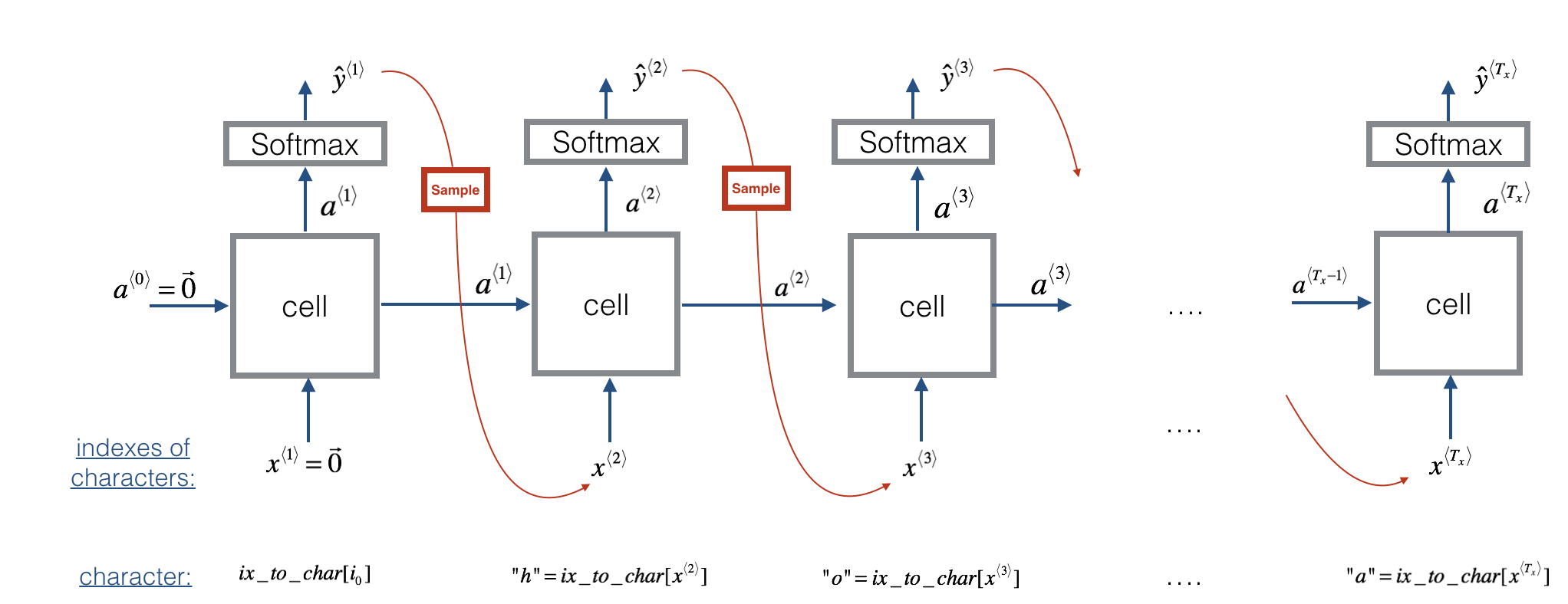


**Figure 2**: Visualization of gradient descent with and without gradient clipping, in a case where the network is running into slight "exploding gradient" problems

for gradient in [dWax, dWaa, dWya, db, dby]:

np.clip(gradient, -maxValue, maxValue, out=gradient)

Now assume that your model is trained. You would like to generate new text (characters). The process of generation is explained in the picture below:



**Figure 3**: In this picture, we assume the model is already trained. We pass in x⟨1⟩=0⃗ x⟨1⟩=0→ at the first time step, and have the network then sample one character at a time.

**Exercise**: Implement the sample function below to sample characters. You need to carry out 4 steps:

* **Step 1**: Pass the network the first "dummy" input x⟨1⟩=0⃗ x⟨1⟩=0→ (the vector of zeros). This is the default input before we've generated any characters. We also set a⟨0⟩=0⃗ a⟨0⟩=0→
* **Step 2**: Run one step of forward propagation to get a⟨1⟩a⟨1⟩ and ŷ ⟨1⟩y^⟨1⟩. Here are the equations:

a⟨t+1⟩=tanh(Waxx⟨t⟩+Waaa⟨t⟩+b)(1)(1)a⟨t+1⟩=tanh⁡(Waxx⟨t⟩+Waaa⟨t⟩+b)

z⟨t+1⟩=Wyaa⟨t+1⟩+by(2)(2)z⟨t+1⟩=Wyaa⟨t+1⟩+by

ŷ ⟨t+1⟩=softmax(z⟨t+1⟩)(3)(3)y^⟨t+1⟩=softmax(z⟨t+1⟩)

Note that ŷ ⟨t+1⟩y^⟨t+1⟩ is a (softmax) probability vector (its entries are between 0 and 1 and sum to 1). ŷ ⟨t+1⟩iy^i⟨t+1⟩ represents the probability that the character indexed by "i" is the next character. We have provided a softmax() function that you can use.

* **Step 3**: Carry out sampling: Pick the next character's index according to the probability distribution specified by ŷ ⟨t+1⟩y^⟨t+1⟩. This means that if ŷ ⟨t+1⟩i=0.16y^i⟨t+1⟩=0.16, you will pick the index "i" with 16% probability. To implement it, you can use [np.random.choice](https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.random.choice.html).

Here is an example of how to use np.random.choice():

np.random.seed(0)  
p = np.array([0.1, 0.0, 0.7, 0.2])  
index = np.random.choice([0, 1, 2, 3], p = p.ravel())

This means that you will pick the index according to the distribution: P(index=0)=0.1,P(index=1)=0.0,P(index=2)=0.7,P(index=3)=0.2P(index=0)=0.1,P(index=1)=0.0,P(index=2)=0.7,P(index=3)=0.2.

* **Step 4**: The last step to implement in sample() is to overwrite the variable x, which currently stores x⟨t⟩x⟨t⟩, with the value of x⟨t+1⟩x⟨t+1⟩. You will represent x⟨t+1⟩x⟨t+1⟩by creating a one-hot vector corresponding to the character you've chosen as your prediction. You will then forward propagate x⟨t+1⟩x⟨t+1⟩ in Step 1 and keep repeating the process until you get a "\n" character, indicating you've reached the end of the dinosaur name.