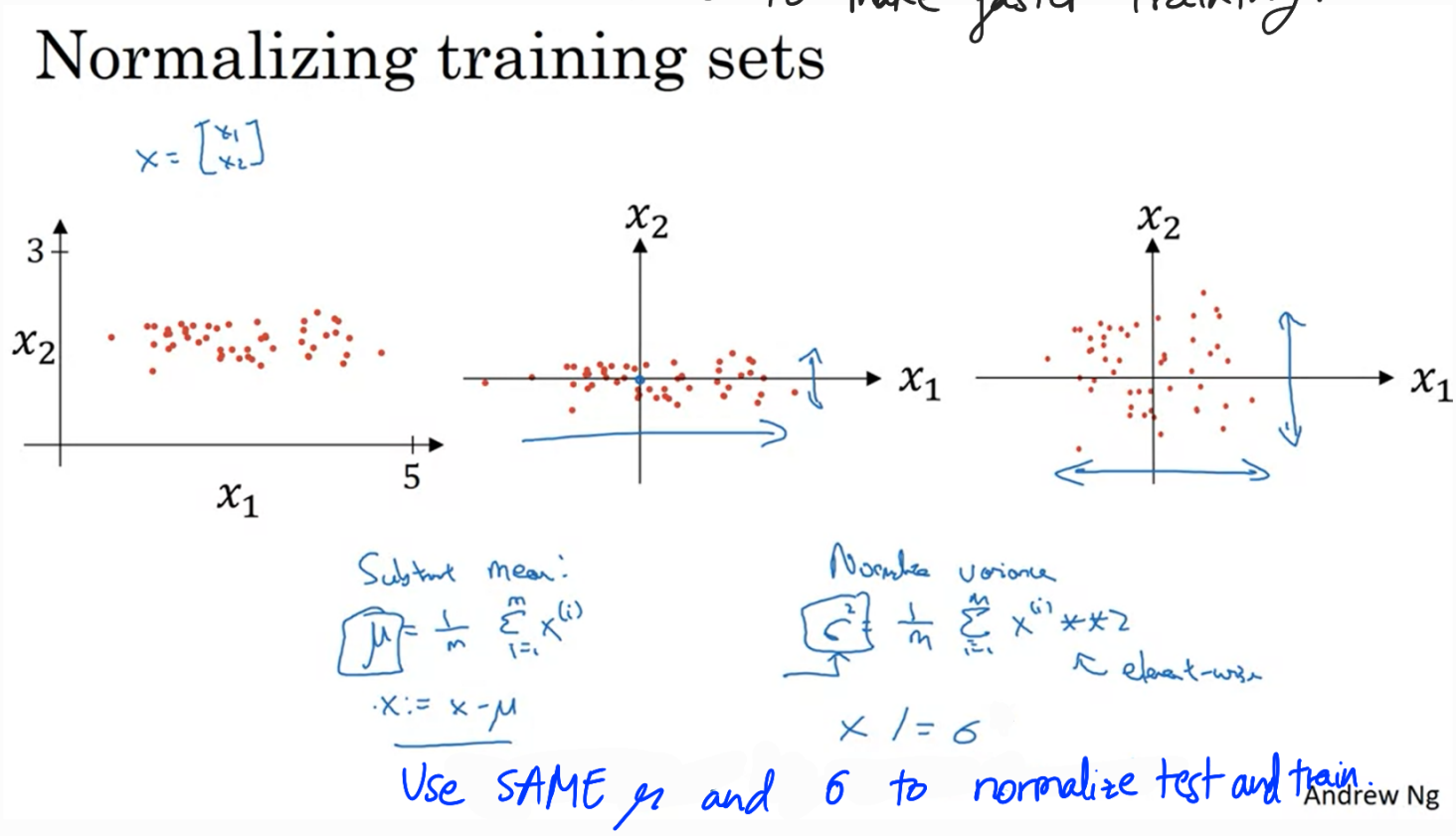
NORMALIZE INPUT Make variances of all features = 1 and mean = 0 to make faster training.



With this process, we put all features in same scales.

VANISHING / EXPLODING → Activations increase exponentially

↓ Activations decrease exponentially

It appears in VERY deep neural networks.

$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$ ~~+b~~

large n → smaller w_i

$Var(w_i) = \frac{1}{n}$

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

We want z to be not very big nor very small, so we must reduce the variance.

MINI-BATCH G.D.

Batch ⇒ set of training examples

Epochs ⇒ number of iterations.

In common gradient descent, we have to pass through all training samples for each epoch.

In mini-batch, we'll divide the training set into multiple SMALL batches.

Batch size
- All samples ⇒ Batch GD (m)
- 1 sample ⇒ Stochastic GD
- t samples ($1 < t < m$) ⇒ Mini batch

} $2^6, 2^7, 2^8, 2^9$

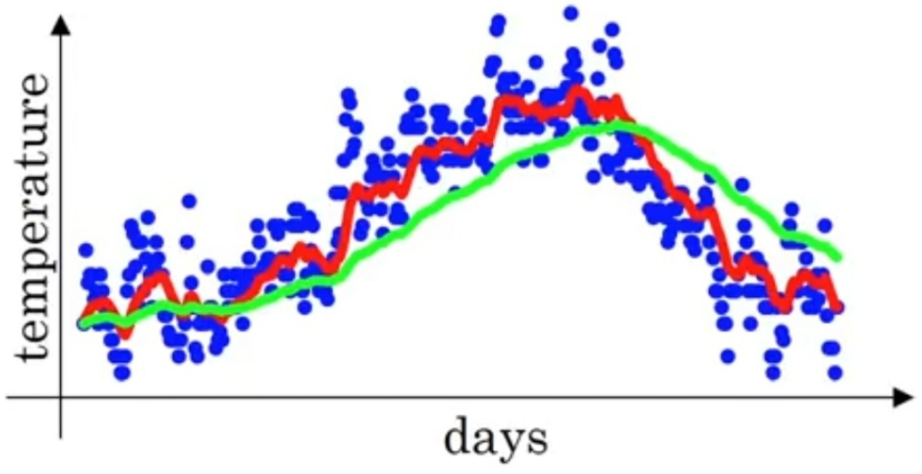
Make sure that the minibatch size fits in CPU/GPU memory.

MOVING AVERAGES *

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9$: ≈ 10 days' temperature

$\beta = 0.98$: ≈ 50 days



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's what that looks like:

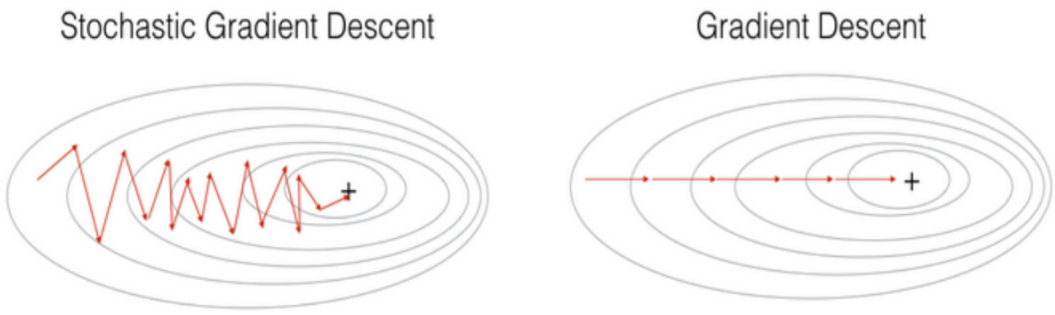


Figure 1 : **SGD vs GD**
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence, but each step is a lot faster to compute for SGD than it is for GD, as it uses only one training example (vs. the whole batch for GD).

Note also that implementing SGD requires 3 for-loops in total:

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 $(W^{[L]}, b^{[L]})$)

In practice, you'll often get faster results if you don't use the entire training set, or just 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.

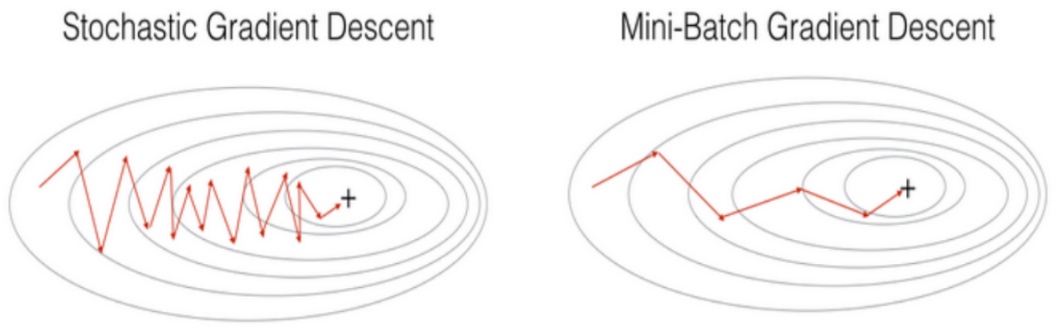


Figure 2 : **SGD vs Mini-Batch GD**
"+" denotes a minimum of the cost. Using mini-batches in your optimization algorithm often leads to faster optimization.

The following figure describes the forward and backward propagation of your fraud detection model.

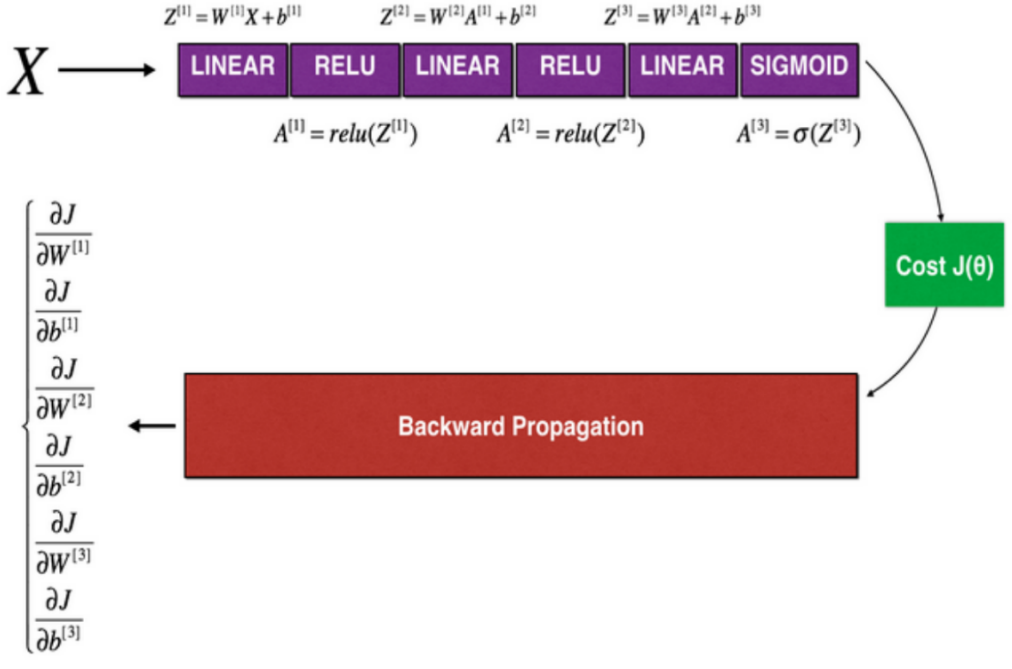


Figure 2: Deep neural network. LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID

Let's look at your implementations for forward propagation and backward propagation.

How does gradient checking work?.

As in Section 3 and 4, you want to compare "gradapprox" to the gradient computed by backpropagation. The formula is still:

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \tag{1}$$

However, θ is not a scalar anymore. It is a dictionary called "parameters". The function "`dictionary_to_vector()`" has been implemented for you. It converts the "parameters" dictionary into a vector called "values", obtained by reshaping all parameters (W1, b1, W2, b2, W3, b3) into vectors and concatenating them.

The inverse function is "`vector_to_dictionary`" which outputs back the "parameters" dictionary.

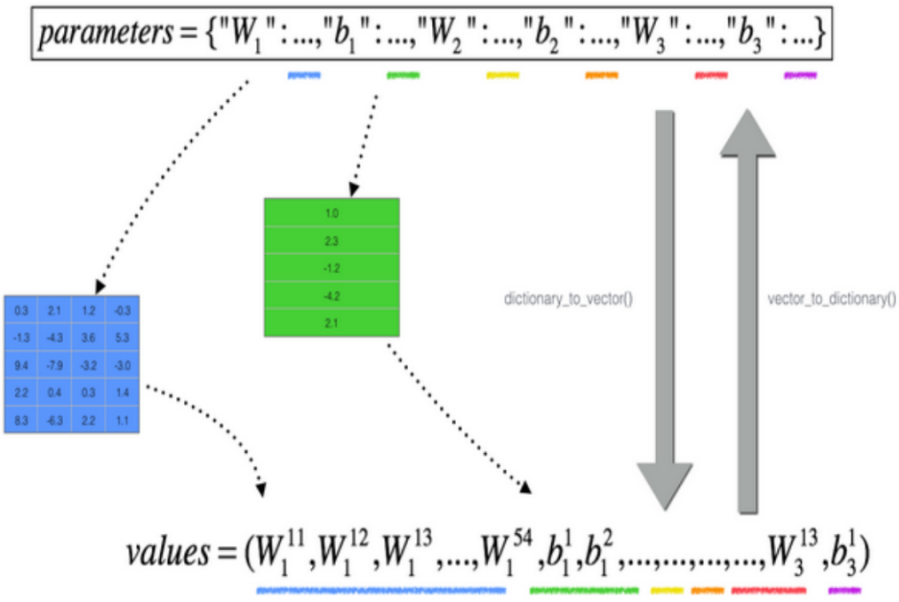


Figure 2: dictionary_to_vector() and vector_to_dictionary(). You will need these functions in gradient_check_n()

The "gradients" dictionary has also been converted into a vector "grad" using gradients_to_vector(), so you don't need to worry about that.

Now, for every single parameter in your vector, you will apply the same procedure as for the gradient_check exercise. You will store each gradient approximation in a vector `gradapprox`. If the check goes as expected, each value in this approximation must match the real gradient values stored in the `grad` vector.

Notes

- Gradient Checking is slow! Approximating the gradient with $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ is computationally costly. For this reason, we don't run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.
- Gradient Checking, at least as we've presented it, doesn't work with dropout. You would usually run the gradient check algorithm without dropout to make sure your backprop is correct, then add dropout.

Congrats! Now you can be confident that your deep learning model for fraud detection is working correctly! You can even use this to convince your CEO. :)

What you should remember from this notebook:

- Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
- Gradient checking is slow, so you don't want to run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.