λ = regularization parameter

Cost function:

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}_{\text{mean of training losses}} + \frac{\lambda}{2m}\sum_{\ell=1}^{L} \|w^{[\ell]}\|^2$$

Where $\|w\|^2 = \sum_{i=1}^{n^{[\ell-1]}} \sum_{j=1}^{n^{[\ell]}} (w_{ij}^{[\ell]})^2 \quad \forall\, w \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}}$

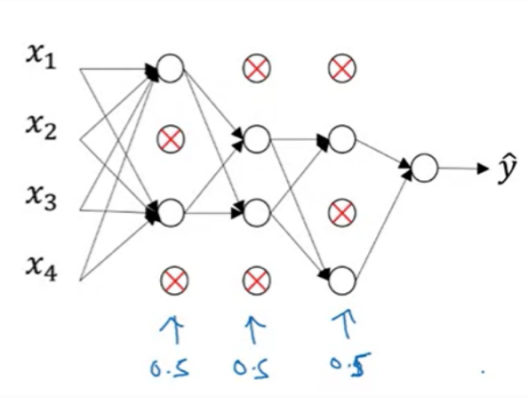
"Frobenius norm"

[Update parameters $\Rightarrow W^{[\ell]} := W^{[\ell]} - \alpha\, dw^{[\ell]} = W^{[\ell]} - \frac{\alpha}{m}\left(dz^{[\ell]} A^{[\ell-1]T} + \frac{\lambda}{m} w^{[\ell]}\right)$]

Why regularization reduces overfitting?

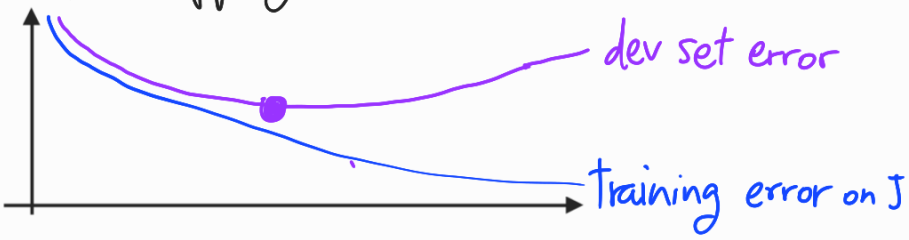
Because it limits the values of z to a small range.

Dropout regularization: go eliminating nodes of network randomly.



Data augmentation: add data from existing (flip, zoom image...)

Early stopping:



Stop when dev set error is minimum.

Divides → Optimize J
→ Not overfit

Observations:

- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

What is L2-regularization actually doing?:

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember: the implications of L2-regularization on:

- The cost computation:
  - A regularization term is added to the cost.
- The backpropagation function:
  - There are extra terms in the gradients with respect to weight matrices.
- Weights end up smaller ("weight decay"):
  - Weights are pushed to smaller values.

Note:

- A **common mistake** when using dropout is to use it both in training and testing. You should use dropout (randomly eliminate nodes) only in training.
- Deep learning frameworks like tensorflow, PaddlePaddle, keras or caffe come with a dropout layer implementation. Don't stress - you will soon learn some of these frameworks.

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.