Deep learning

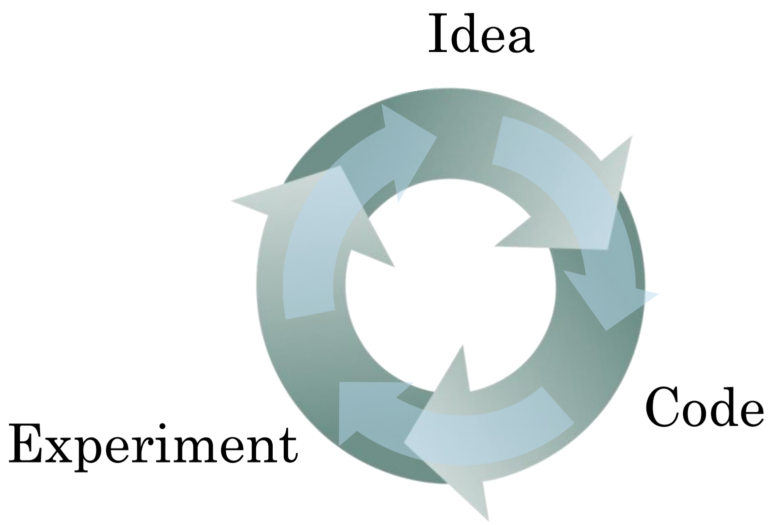
Dr. Aissa Boulmerka
a.boulmerka@centre-univ-mila.dz

2023-2024

CHAPTER 4
PRACTICAL ASPECTS OF DEEP LEARNING

Applied ML is a highly iterative process

# layers
# hidden units
learning rates
activation functions …



- Its impossible to get all your **hyperparameters** right on a new application from the **first time**.
- So the idea is you go through the loop: **Idea $\Longrightarrow$ Code $\Longrightarrow$ Experiment**.
- You have to go through the **loop many times** to figure out your hyperparameters.

Train/dev/test sets

Classical ML (100 – 10000 samples):

Data:	Training	Dev	Test
	60%	20%	20%

Deep learning (1M samples)

Data:	Training	Dev	Test
	98% 99.5%	1% 0.25%	1% 0.25%

- Your **data** will be split into three parts:
  - **Training set** (Has to be the largest set)
  - Hold-out cross validation set / Development or **"dev" set**.
  - **Testing set**.
- You will try to build a model upon **training set.**
- Then try to optimize hyperparameters on **dev set** as much as possible.
- Then after your model is ready you try and evaluate the **testing set**.
- The trend now gives the **training data** the biggest sets.

Mismatched train/test distribution

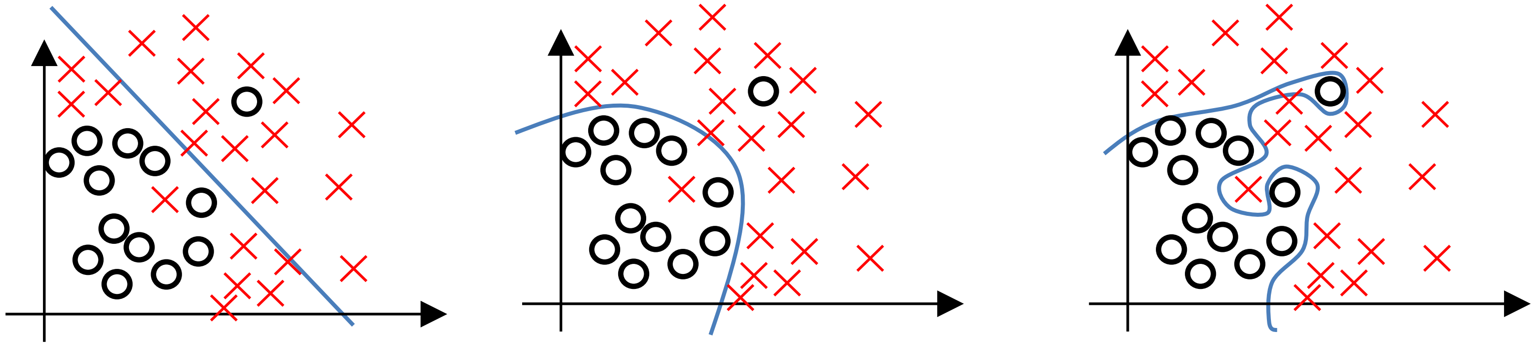
- Make sure dev set and test set come from the same distribution.
- For example if cat **train set** is from the web and the **dev/test images** are from users cell phone they will mismatch. It is better to make sure that dev and test set are from the same distribution.

Training set: Cat pictures from webpages	**Dev/test sets:** Cat pictures from users using your app

- The dev set rule is to try them on some of the **good models you've created**.

- Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a **dev set as its used in the development.**

Bias and Variance



high bias — **Underfitting**

"just right" — **Appropriate**

high variance — **Overfitting**

- Bias / Variance techniques are **easy to learn**, but **difficult to master**.
- So here the explanation of Bias / Variance:
  - If your model is **underfitting**, it has a "high bias"
  - If your model is **overfitting** then it has a "high variance"
  - Your model will be **alright** if you balance the Bias / Variance.

Bias and Variance

Cat classification

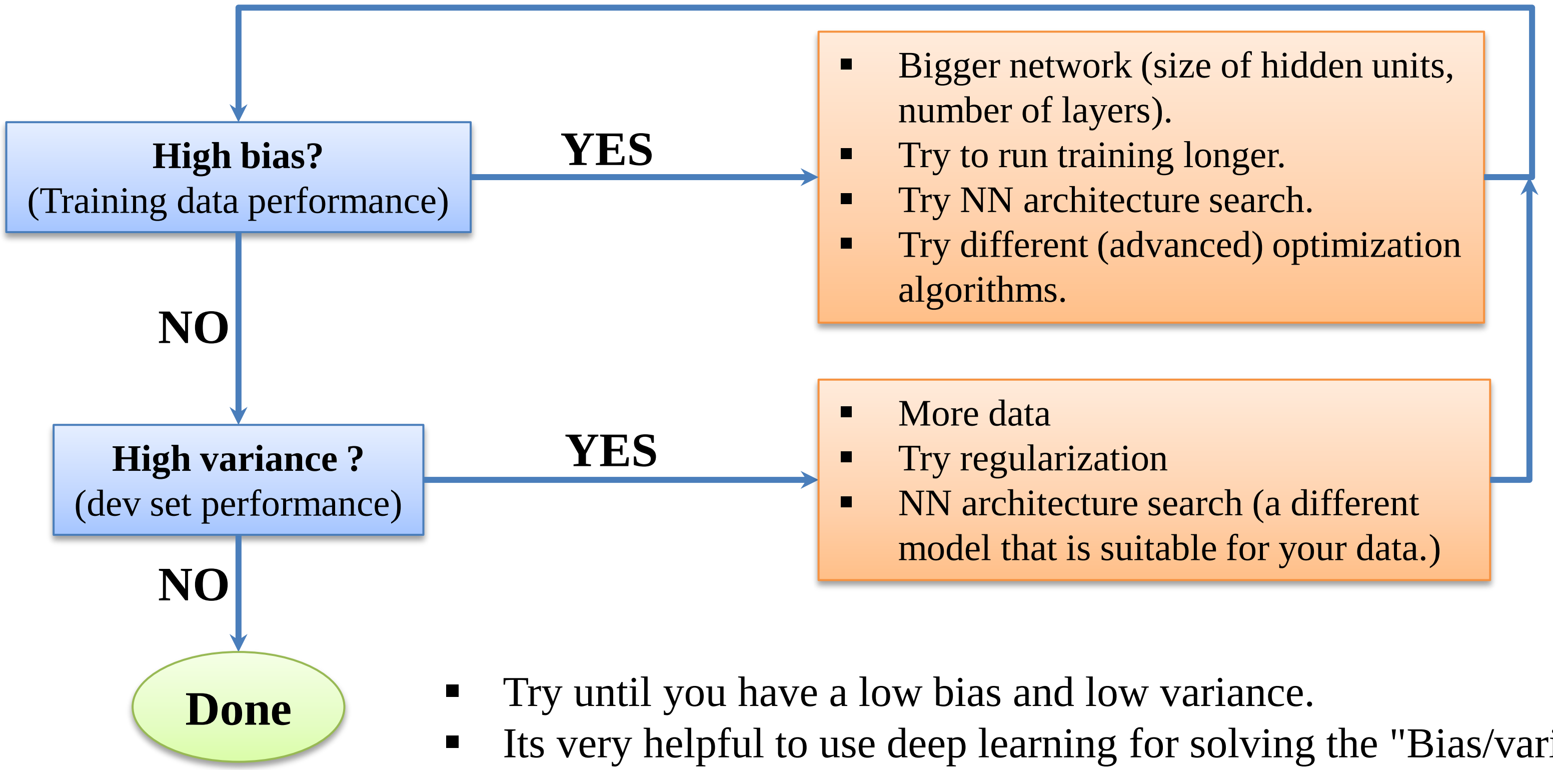
$y = 1$

$y = 0$

Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	High variance (Overfitting)	High bias (Underfitting)	High bias and high variance (Overfitting and underfitting	Low bias and low variance (Best)

Assuming humans get 0% error

Basic recipe for machine learning



- Bigger network (size of hidden units, number of layers).
- Try to run training longer.
- Try NN architecture search.
- Try different (advanced) optimization algorithms.

- More data
- Try regularization
- NN architecture search (a different model that is suitable for your data.)

- Try until you have a low bias and low variance.
- Its very helpful to use deep learning for solving the "Bias/variance tradeoff" problem because with deep learning you have more options/tools.
- Training a bigger neural network never hurts.

Regularization (Logistic regression)

Euclidian norm

$$\min_{w,b} J(w,b), \quad w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$$

$$\boldsymbol{L_2}\ \textbf{regularization :}\ J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|w\|_2^2, \left(\|w\|_2^2 = \sum_{j=1}^{n_x} w_j{}^2 = w^T w\right)$$

$$\boldsymbol{L_1}\ \textbf{regularization :}\ J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{m}\|w\|_1, \left(\|w\|_1 = \sum_{j=1}^{n_x} |w_j|\right)$$

- Adding regularization to NN will help it **reduce variance** (overfitting)
- **L1 regularization** version makes a lot of w values become zeros, which makes the model size smaller.
- **L2 regularization** is being used much more often.
- λ (lambda) is the **regularization parameter** (hyperparameter).

Regularization (Neural network)

$$\min_{w,b} J\left(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}\right),$$

$$J\left(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\left\|w^{[l]}\right\|_F^2$$

$\|.\|_F^2$: Forbenius norm

$$\left\|w^{[l]}\right\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}\left(w_{ij}^{[l]}\right)^2 \qquad w^{[l]}:\left(n^{[l]}, n^{[l-1]}\right)$$

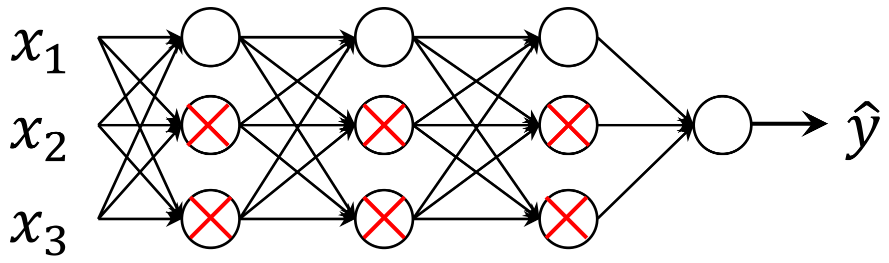
$$dw^{[L]} = (from\ backprop) + \frac{\lambda}{m}w^{[L]}$$
$$w^{[L]} = w^{[L]} - \alpha dw^{[L]}$$

Weight decay:

$$w^{[L]} = w^{[L]} - \alpha\left((from\ backprop) + \frac{\lambda}{m}w^{[L]}\right)$$
$$w^{[L]} = w^{[L]} - \frac{\alpha\lambda}{m}w^{[L]} - \alpha(from\ backprop)$$
$$w^{[L]} = \left(1 - \frac{\alpha\lambda}{m}\right)w^{[L]} - \alpha(from\ backprop)$$

- In practice this penalizes large weights and effectively limits the freedom in your model.
- The new term $\left(1 - \frac{\alpha\lambda}{m}\right)w^{[L]}$ causes the weight to decay in proportion to its size.

How does regularization prevent overfitting?



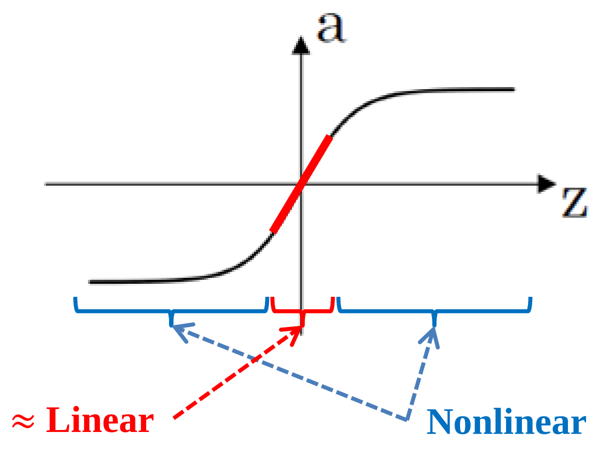
$$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \sum_{l=1}^{L}\frac{\lambda}{2m}\|w^{[l]}\|_F^2$$

- If λ is very big $\Rightarrow w^{[l]} \approx 0 \Rightarrow$ more simple neural network.

Here are some intuitions:

- **Intuition 1:**
  - **If λ is too large :** a lot of w's will be close to zeros which will make the NN simpler (you can think of it as it would behave closer to logistic regression).
  - **If λ is good enough:** it will just reduce some weights that makes the neural network overfit.

How does regularization prevent overfitting?



tanh:

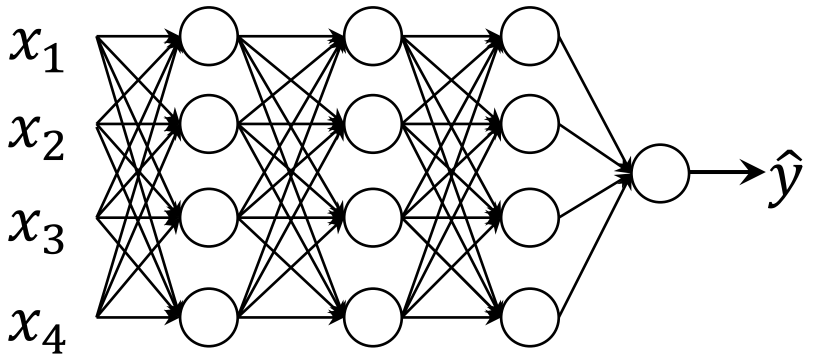
$$a = \frac{z^{z} - e^{-z}}{z^{z} + e^{-z}}$$

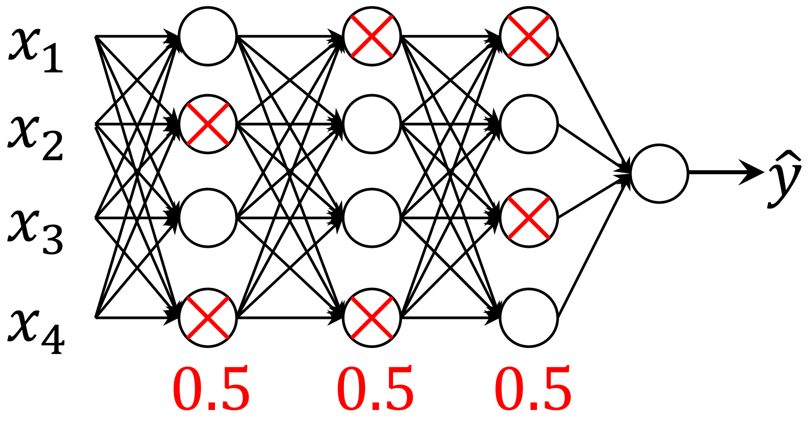
$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

If λ is very big $\Rightarrow w^{[l]} \approx 0 \Rightarrow z^{[l]}$ will be relatively small, then every layer will be ≈ **Linear**

- **Intuition 2 (with tanh activation function):**
  - **If lambda is too large:** w's will be small (close to zero) - will use the linear part of the tanh activation function, so we will go from non linear activation to roughly linear which would make the NN a roughly linear classifier.
  - **If lambda good enough:** it will just make some of tanh activations roughly linear which will prevent overfitting.

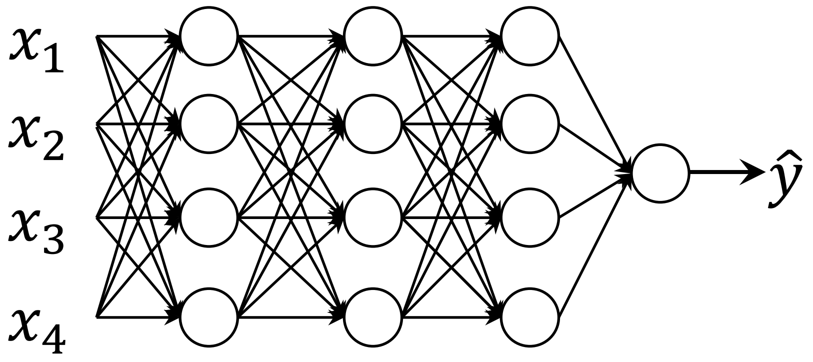
Dropout regularization

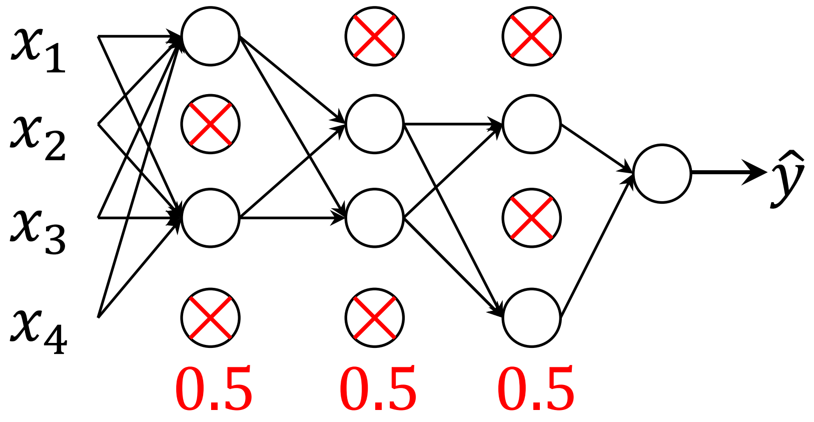




- Go through each of the layers of the network and **set some probability** of **eliminating a node** in neural network.
- For each of these layers, we're going to, for each node, toss a coin and have a 0.5 chance of keeping each node and 0.5 chance of removing each node.
- Then **remove all the outgoing** things from that node as well.

Dropout regularization





- We end up with a **much smaller network**.
- And then do back **propagation training**.

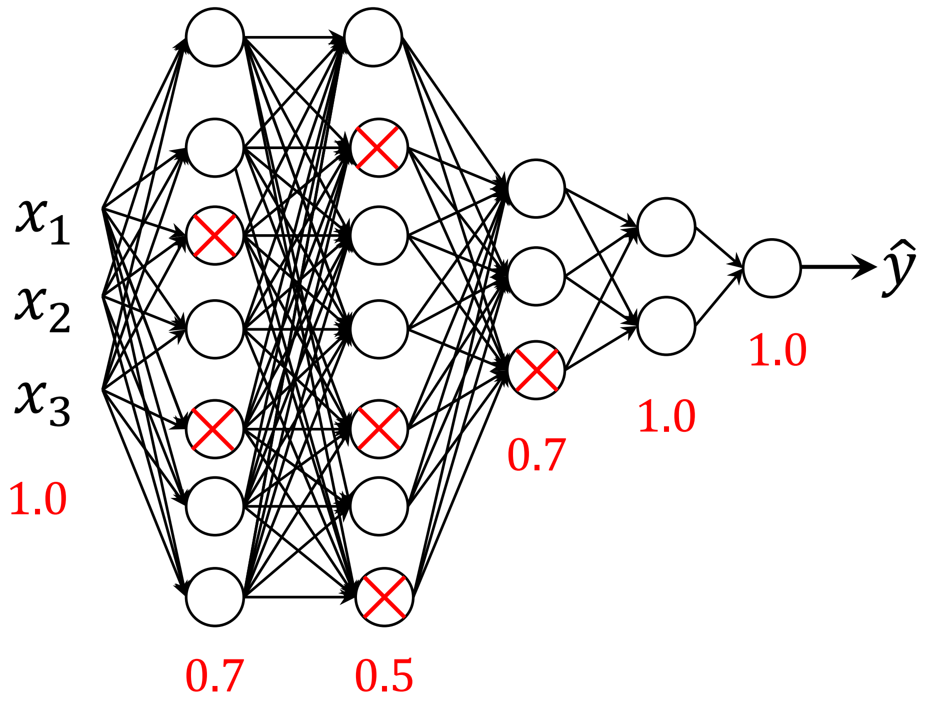
Implementing dropout (“Inverted dropout”)

```
keep_prob = 0.8 # 0 <= keep_prob <= 1
l = 3 # this code is only for layer 3
# the generated number that are less than 0.8 will be dropped.
80% stay, 20% dropped
d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob
a3 = np.multiply(a3,d3) # keep only the values in d3
# increase a3 to not reduce the expected value of output
# (ensures that the expected value of a3 remains the same) - to
solve the scaling problem
a3 = a3 / keep_prob
```

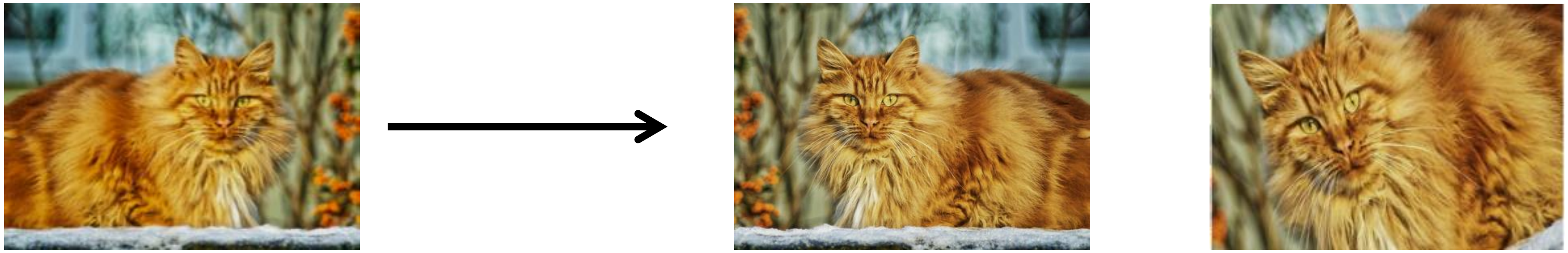
- Vector d[l] is used for **forward and back propagation** and is the same for them, but it is different for each iteration (pass) or training example.
- At test time **we don't use dropout**. If you implement dropout at test time - it would **add noise to predictions**.

Why does drop-out work?

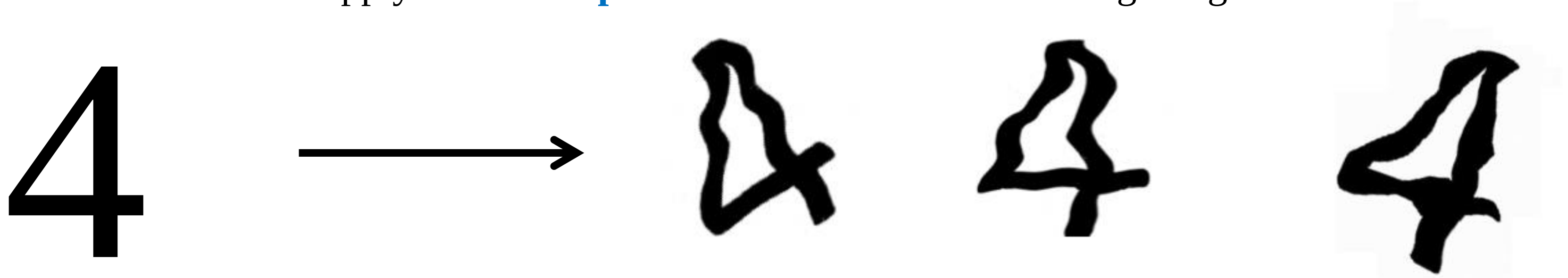
- Intuition: Can’t rely on any one feature, so have to spread out weights.



Data augmentation

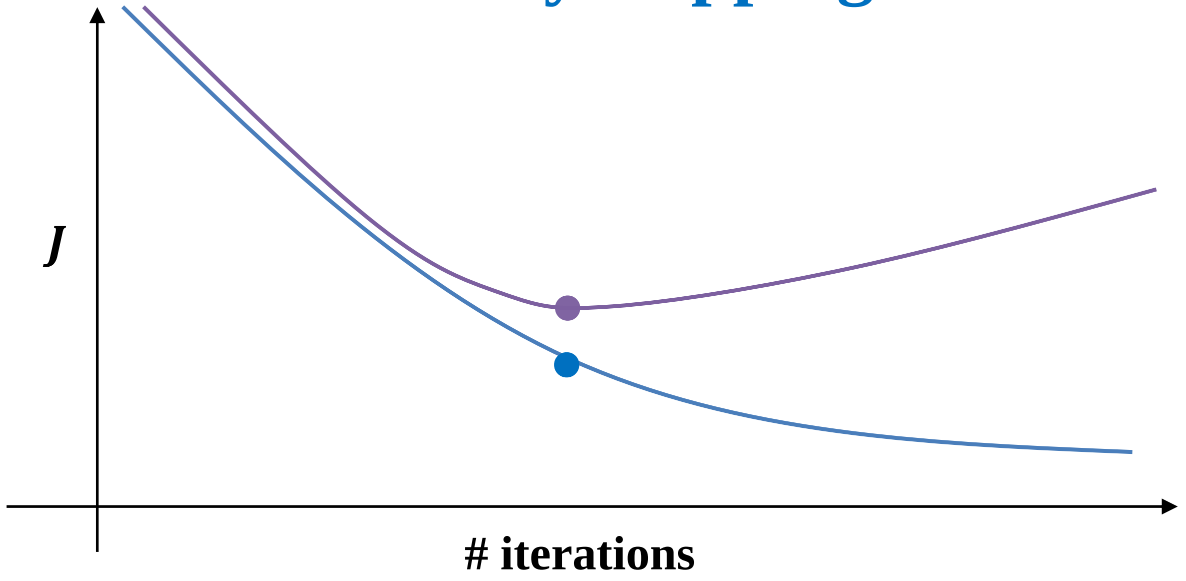


- In a **computer vision** data:
  - You can **flip all your pictures horizontally** this will give you m more data instances.
  - You could also apply a **random position and rotation** to an image to get more data.



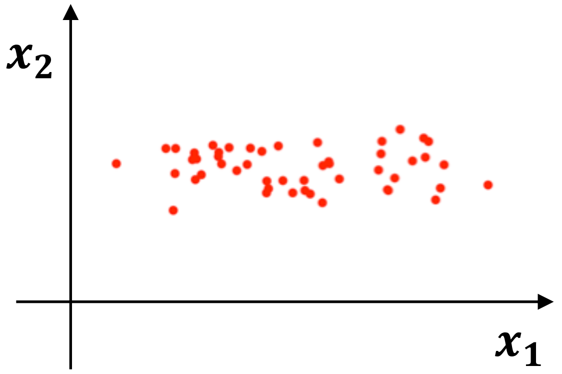
- in **OCR**, you can impose **random rotations and distortions** to digits/letters.
  - New data obtained using this technique isn't as good as the real independent data, but **still can be used as a regularization technique**.

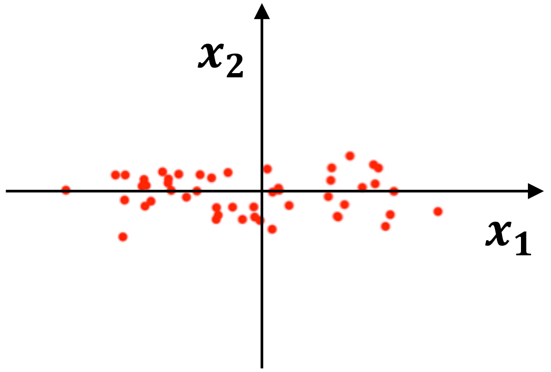
Early stopping

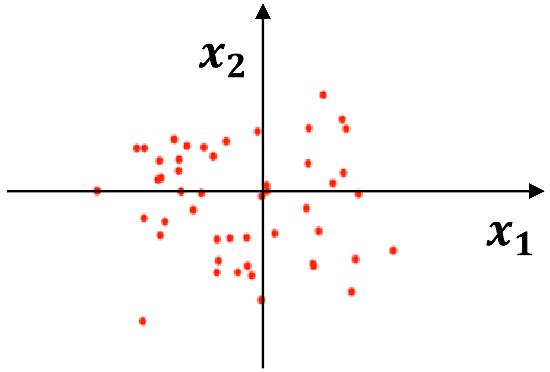


- In this technique we plot the **training set** and the **dev set cost** together for each iteration. At some iteration the dev set cost will stop decreasing and will start increasing.
- We will **pick the point** at which the training set error and dev set error are best (lowest training cost with lowest dev cost).
- We will take these parameters as the **best parameters**.
- The advantage of this method is that you **don't need to search a hyperparameter** like in other regularization approaches (like lambda in L2 regularization).

Normalizing training sets







Subtract mean:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} X^{(i)}$$

$$X := X - \mu$$

Normalize variance:

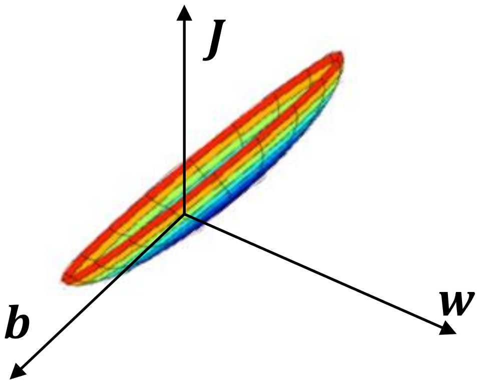
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} {X^{(i)}}^2$$

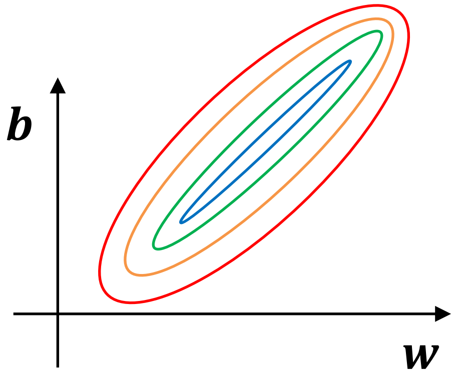
$$X := X/\sigma^2$$

Use the same parameters μ and σ^2 to normalize the test set.

Why normalize inputs?

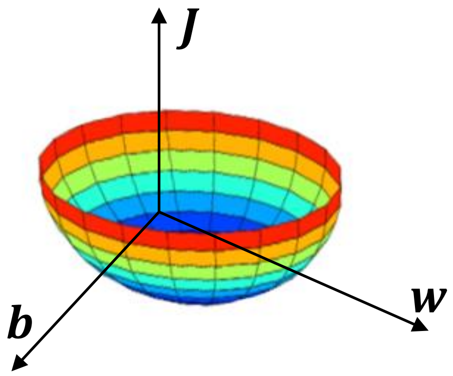
Unnormalized :

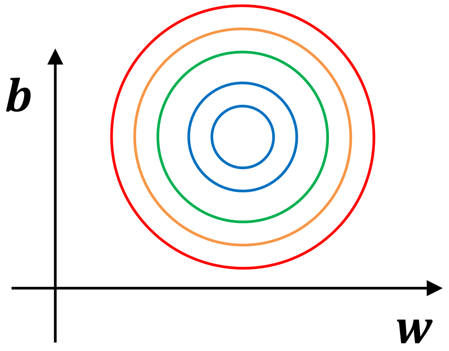




$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

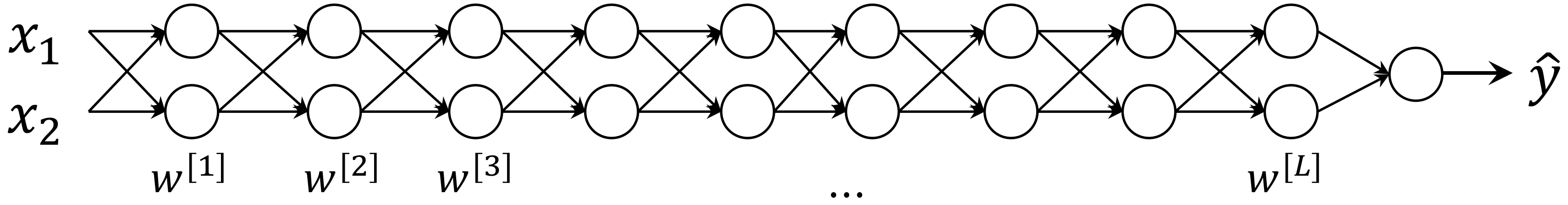
Normalized :





If we **normalize**, we can use a much larger learning rate $\alpha \Longrightarrow$ **speed up the training process**

Vanishing/exploding gradients



- The Vanishing / Exploding gradients occurs when your **derivatives become very small or very big**.
- To understand the problem, suppose that we have a deep neural network with number of layers L, and all the activation functions are linear and each b = 0

$$g(z) = z \quad , b = 0$$
$$\hat{y} = w^{[L]} w^{[L-1]} \ldots w^{[2]} w^{[1]} x$$

Example: Deep neural network (L layers)

If $w = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \Longrightarrow 0.5^{L-1} \Rightarrow$ Vanishing

If $w = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \Longrightarrow 1.5^{L-1} \Rightarrow$ Exploding

In both cases gradient descent takes a very long time

Vanishing/exploding gradients

- A **partial solution** to the Vanishing / Exploding gradients in NN is better or more careful choice of the random initialization of weights.
- He/Xavier initialization:
  - **For ReLU:** $W^{[l]} = rand * \sqrt{\frac{2}{n^{[l-1]}}}$
  - **For tanh:** $W^{[l]} = rand * \sqrt{\frac{1}{n^{[l-1]}}}$
  - **For tanh (Bengio et al.):** $W^{[l]} = rand * \sqrt{\frac{2}{n^{[l]}+n^{[l-1]}}}$
- Number 1 or 2 in the nominator can also be a **hyperparameter** to tune **(but not the first to start with).**
- This is one of the best way of **partially solution to Vanishing / Exploding gradients** (ReLU + Weight Initialization with variance) which will help gradients not to vanish/explode too quickly.
- This initialization is called **"He Initialization / Xavier Initialization"** and has been published in 2015 paper.

Gradient Checking

- If your cost does **not decrease on each iteration** you may have a **back-propagation bug**.
- Gradient checking approximates the gradients and is very helpful for finding the **errors** in your **backpropagation implementation** but it's slower than gradient descent (so use only for debugging).
- Implementation of this is **very simple**.

Gradient Checking

- Take $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ.

The cost function will be J$(\boldsymbol{\theta})$

- Take $dW^{[1]}, db^{[1]}, \cdots, dW^{[L]}, db^{[L]}$ and reshape into a big vector dθ.

Is d$\boldsymbol{\theta}$ the gradient of J$(\boldsymbol{\theta})$?

Gradient Checking

- **Algorithm:**

```
eps = $10^{-7}$ # small number
for i in len($\theta$):
    $d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2 \ldots, \theta_i + eps, \ldots) - J(\theta_1, \theta_2 \ldots, \theta_i - eps, \ldots)}{2 * eps}$
```

- Finally we evaluate this formula $\frac{\|d\theta_{approx} - d\theta\|}{\|d\theta_{approx}\| + \|d\theta\|}$ and check (with $eps = 10^{-7}$) (*):
    - **if it is $< 10^{-7}$:** great, very likely the backpropagation implementation is correct.
    - **if around 10^{-5}:** can be OK, but need to inspect if there are no particularly big values in $d\theta_{approx} - d\theta$.
    - **if it is $\geq 10^{-3}$:** bad, probably there is a bug in backpropagation implementation.

(*) || ||: Euclidean vector norm.

Gradient checking implementation notes

- Don't use the **gradient checking** algorithm at training time because it's very slow.
- Use gradient checking only for **debugging**.
- If the algorithm **fails grad check**, look at components to **try to identify the bug**.
- Don't forget to add $\frac{\lambda}{m}\|w\|_1$ to J if you are using **L1 or L2 regularization**.
- **Gradient checking** doesn't work with dropout because J is not consistent.
  - You can first **turn off dropout** (set keep_prob = 1.0), **run gradient checking** and then **turn on dropout** again.
- Run gradient checking at **random initialization** and train the network for a while maybe there's a bug which can be seen when weights w and bias b **become larger** (further from 0) and can't be seen on the first iteration (when weights w and bias b are **very small**).

Initialization summary

- The **weights *W*** should be initialized **randomly** to **break symmetry.**
- However, you can initialize the biases b to **zeros**. Symmetry is still broken so long as ***W* is initialized randomly**.
- Different **initializations** lead to different **results**.
- **Random initialization** is used to break symmetry and make sure different hidden units can learn different things.
- Don't intialize to values that are **too large.**
- **He initialization** works well for networks with **ReLU activations**.

L2 Regularization summary

- **Observations:**
  - 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: Implications of L2-regularization on:**
  - **cost computation:** A regularization term is added to the cost
  - **backpropagation function:** There are extra terms in the gradients with respect to weight matrices
  - **weights:** weights end up smaller ("weight decay") - are pushed to smaller values.

Dropout summary

What you should remember about dropout:

- Dropout is a **regularization** technique.
- 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**.

References

- Andrew Ng. Deep learning. Coursera.
- Geoffrey Hinton. Neural Networks for Machine Learning.
- Kevin P. Murphy. Probabilistic Machine Learning An Introduction. MIT Press, 2022.
- MIT Deep Learning 6.S191 (http://introtodeeplearning.com/)