parameters VS Hyperparameters

parameters :

$w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}$

Hyperparameters

Optimization algorithm :

- learning rate α / learning rate decay
- # iterations
- mini batch size
- momentum
- regularization parameter λ / Keep prob (dropout)

network :

- # hidden layers L
- # hidden units in each layer
- # choice of activation functions
- * the variance of $w^{[l]}$ when initializing them

Regularization techniques

- normal regularization
- dropout
- data augmentation
- early stopping
- * batch normalization
- * bagging

~~Hyperparameter~~

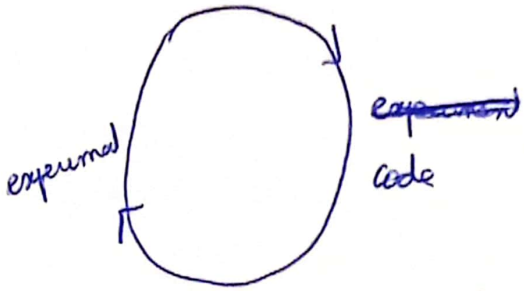
We plot cost function of # iterations

changing a value of a hyperparameter

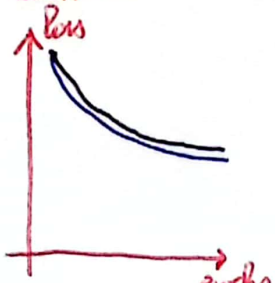
⇒ applied ML is a highly iterative process

(

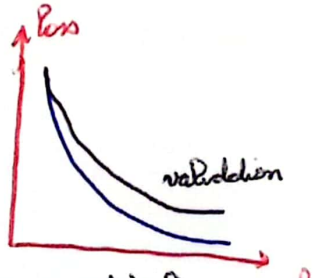
Idea



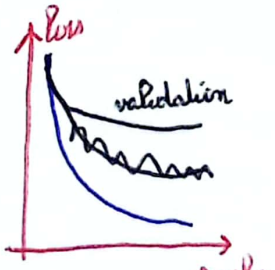
overfitting / underfitting avec les courbes d'apprentissage



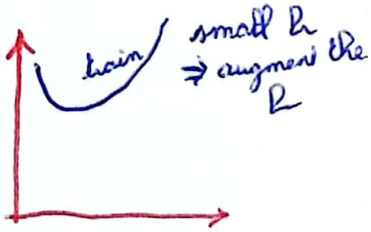
underfitting



optimal

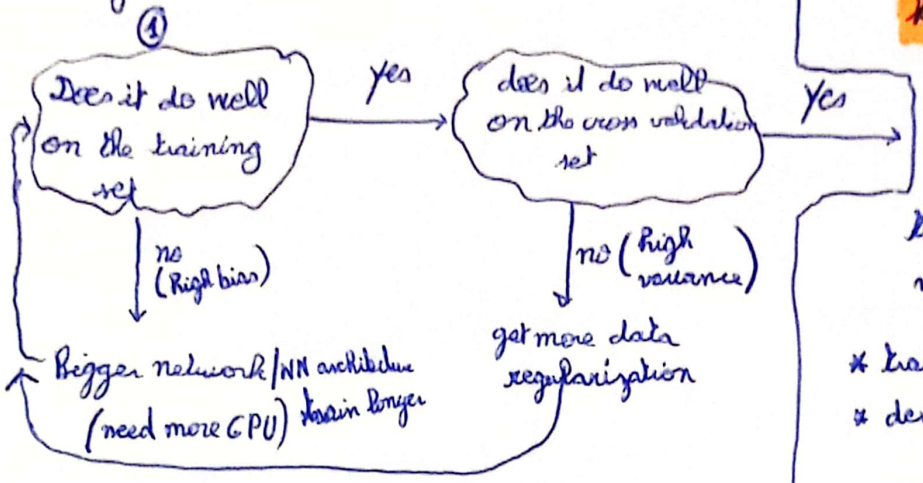


overfitting



bias / variance and neural networks

* Large neural networks are low bias machines



* a large neural network will usually do as well or better than a small one so long as regularization is chosen correctly

"Bias Variance tradeoff"

we don't have as many tools to reduce bias without affecting variance (and vice versa) in ML algorithms

but in modern big data era :

1/ training bigger network almost reduces bias without hurting your variance as you regularize well

2/ getting more data always reduces variance without hurting bias

⇒ we don't have necessarily to give attention to the tradeoff

Regularized MNIST model

```
l1 = Dense(25, activation = 'relu',
           kernel_regularizer = L2(0.01))
```

---

to have an idea about bias and variance of our model

* train set error
* dev set error

example 1

train set error : 1%
test set error : 11%

⇒ High variance

example 2

train set error : 15%
test set error : 16%

we suppose human error in this field ≃ 1%

⇒ train error ≫ human error
⇒ underfitting (High bias)

example 3

train set error : 15%
test set error : 30%
human error : 0%

⇒ High bias
high variance (test error ≫ train e

example 4

train error : 0.5%
test error : 1%

Regularization

applied

Logistic Regression

$$\min_{w,b} J(w,b)$$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|w\|_2^2$$

$$\|w\|_2^2 = \sum_{j=1}^{n} w_j^2 = w^T w$$

↓ L2 regularization

λ → regularization parameter

L1: regularization : $\frac{\lambda}{m}\|w\|_1 = \frac{\lambda}{2m}\sum_{i=1}^{n}|w|$

↓

⇒ w will be sparse (a lot of zeros)

⚠ L2 : regularization is much more used

↓

it is a weight decay

NN

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

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

↓ norme de Frobenius

$w^{[l]} : \left(n^{[l]}, n^{[l-1]}\right)$

why regularization prevents overfitting

telling $\|w^{[l]}\| \simeq 0$ et donc $w^{[l]} \simeq 0$

⇒ ~~reducing~~ reducing the impact of some Neurons

⇒ simpler network

if λ is very big ⇒ very very simple NN

⇒ (underfitting)

if λ too small ⇒ overfitting

⇒ we need to change λ to get the best parameter

dropout regularization

with dropout

during training : a percentage of the features were set to 0

during testing : all neurons are used

⇒ the model during testing is more robust and can lead to higher testing accuracies.

① set a certain probability of eliminating a node in a neural network

⇓

we get a smaller NN

but we need to fix this probability

How to implement dropout with "inverted dropout"

illustrate with layer 3 ; keep.prob = 0,8

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep.prob
a3 = np.multiply(a3, d3)
a3 /= keep.prob
```

if we have a total of 50 units

10 were shut off

$$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$$

↑ 20% of these elements were = 0

↓

/ 0.8

if we have alot of neurons in a layer

⇒ more risk of overfitting

⇒ we chose a keep.prob (0.5)

else

we chose a high keep.prob (0.8, 0.9, 1.0)

if we have only one neuron in a layer : chose 1.0

other regularization methods

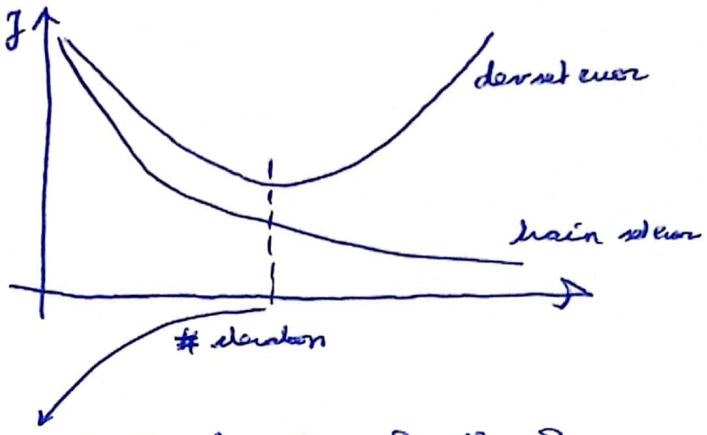
1/ data augmentation

add extra fake training examples

applying rotations / zooming

adding distortions

2/ early stopping



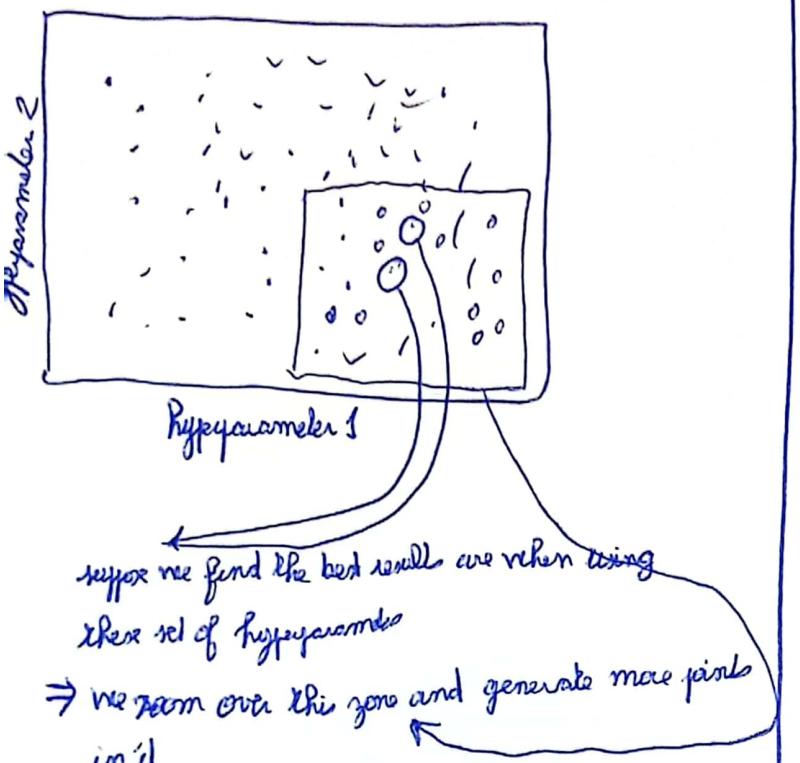
we stop the optimization algorithm here

Hyperparameter tuning

1/ use a grid for the hyperparameters

2/ but in deep learning try random values instead of using a grid.

3/ coarse to fine



suppose we find the best results are when using these set of hyperparameters

⇒ we zoom over this zone and generate more points in it

appropriate scale for hyperparameters

α:

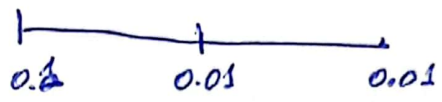
$r = -4 * np.random.rand() \rightarrow [-4, 0]$

$\alpha = 10^r$

β: exponentially weighted averages hyperparameter

$[0.9, \ldots, 0.999]$

$1-\beta = 0.1 \ldots\ldots 0.001$



$r = -$