Setting Up Your ML Application

Train / Dev / Test Sets

Data	training set	dev	test

hold-out
cross-validation
- development set

Mismatched train/test distribution

make sure dev and test sets come from the same distribution.
devset: what we make decisions based on (which algorithm to choose)

a problem ⇒ training set: high quality cat pictures from web

test set: cat pictures from users using your app (low resolution images)

if you only have a train / test set, your train acts like a dev because you use that set to tune hyperparameters and algorithmic choices

Bias and Variance

bias ⇒ high bias: many mistakes on training data (underfitting)

variance ⇒ high variance: very sensitive to small fluctuations in training set (overfitting)

Train set error: 1%
Dev set error: 11%
} Overfitting: not generalizing well (high variance)

Train set error: 15%
Dev set error: 16%
} Underfitting: performs poorly but generalizes well (high bias)

Train set error: 15%
Dev set error: 30%
} high bias & high variance (you can get BOTH)

BUT, all of this analysis is based on the assumption that

HUMAN ERROR ≈ 0.0 % OR MORE GENERALLY Optimal (Bayes) ERROR: 0.0%

Basic Recipe for ML

High Bias? (diagnose by: looking at training data performance)

- → Bigger network (almost always helps) (doesn't hurt variance)
- → Train Longer
- → Try different NN architecture

High Variance? (diagnose by: dev set performance)

- → Get more data (doesn't hurt bias)
- → Regularization
- → Try finding better NN Architecture

Bias / Variance Tradeoff: In the modern DNN era, getting bigger network almost always reduces bias w/o hurting variance and getting more data almost always reduces variance w/o hurting bias.

Regularizing your NN

Regularization

Logistic Regression (Developing the Argument)

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

λ: regularization parameter

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

$\|w\|_2^2$: norm

L2 Regularization: (Euclidean Norm)

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

L1 Regularization:

$$\frac{\lambda}{m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}\|w\|_1$$

Sparse model, feature selection

L1 regularization produces a sparse model, a model that has most of its parameters equal to zero.

L1 makes feature selection by deciding which features are essential for prediction and which ones are not.

λ: Regularization Parameter

Another hyperparameter $\Rightarrow$ we can decide its value with validation set.

Neural Network

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

$$\|w^{[\ell]}\|^2 = \sum_{i=1}^{n^{[\ell]}}\|w_i^{[\ell]}\|^2 = \sum_{r=1}^{n^{[\ell]}}\sum_{j=1}^{n^{[\ell-1]}}(w_{ij}^{[\ell]})^2$$

"Frobenius Norm"

$\|\cdot\|_F^2$

sum of squared of elements of a matrix

$w_i^{[\ell]}$: ith row, row vector for the unit in ℓ-1 layer

recall

$$w^{[\ell]} \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}}$$

Change In Backprop

we add gradient of regularization term to gradients of weights

$$dW^{[\ell]} = \frac{1}{m} dW^{[\ell]} A^{[\ell-1]T} + \frac{\lambda}{m} W^{[\ell]}$$

$$\frac{\partial}{\partial w}\left(\frac{1}{2}\frac{\lambda}{m}w^2\right) = \frac{\lambda}{m}w$$

How Does This Affect Backprop?

new addition

$$dW^{[\ell]} = (\text{from backprop}) + \frac{\lambda}{m} W^{[\ell]}$$

$$W^{[\ell]} := W^{[\ell]} - \alpha\, dW^{[\ell]}$$

$$:= W^{[\ell]} - \frac{\alpha\lambda}{m} W^{[\ell]} - \alpha\,(\text{from backprop})$$

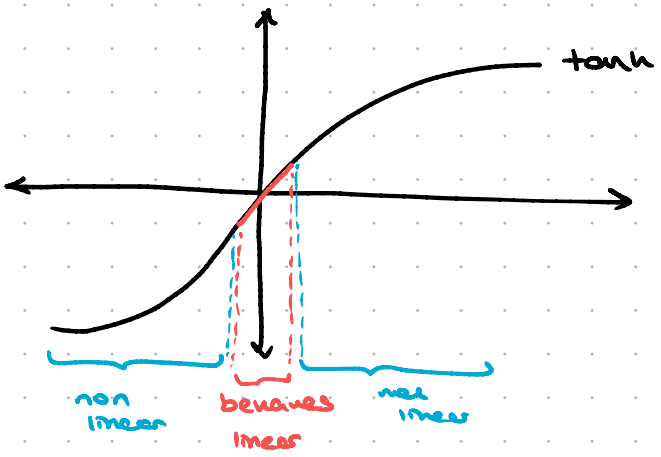
this is why L_2 regularization has been called "weight decay"

Why Does Regularization Help with Overfitting?

regularization incentivizes the model to set W close to zero, i.e. reducing the impact of some hidden layers.

↓

creates a simpler + smaller network



$\lambda \uparrow \quad W^{[\ell]} \downarrow$

$z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}$

every layer $\approx$ linear → model is closer to linear regression (simpler decision boundaries)

Dropout Regularization

1) go through each layer & drop a unit based on a probability.

2) if dropped, remove in-going and outgoing connections of the unit

3) train the smaller network

At each step of training, dropout applies one step of backprop learning to a new version of the network. This new version is created by deactivating a randomly chosen subset of the units.

This is a low cost way to train a large ensemble of different networks.

Implementing Dropout ("Inverted Dropout")

we'll illustrate with layer $\ell = 3$ — keep_prob = 0.8

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
```

create a 0/1 matrix

we make different units for different examples

```
a3 = np.multiply(a3, d3)
```

eliminate some hidden units

```
a3 /= keep_prob
```

on expectation, 20% of elements of a3 will be zeroed out. → so, to keep expected value at $z^{[4]}$ constant, divide it by keep_prop

expected value of z remains constant

AT test time:

→ don't use dropout.

→ because we divided by keep_prob at train time, we ensure that the scaling ⟶ no need for funny scaling

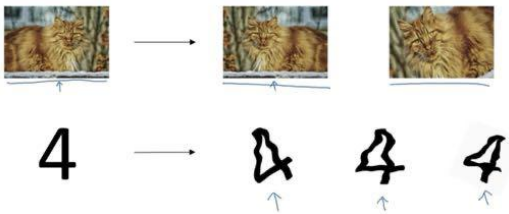
The intuition is because we are turning off units, no one unit can rely on another unit. So it has to spread the weight around, which reduces L_2 norm of weights.

Other Regularization Methods

augmenting training set

⇒ flip your images + add them to train set

⇒ random distortions of image



note: these methods don't add a lot of value but a cheap way to get more data

Early Stopping

Orthogonalization:

- optimize cost function J
  - gradient descent
- not overfit
  - regularization

Early stopping works on both tasks at once. Violates orthogonalization.

Setting Up Your Optimization Problem

Normalizing Inputs

Subtract mean:

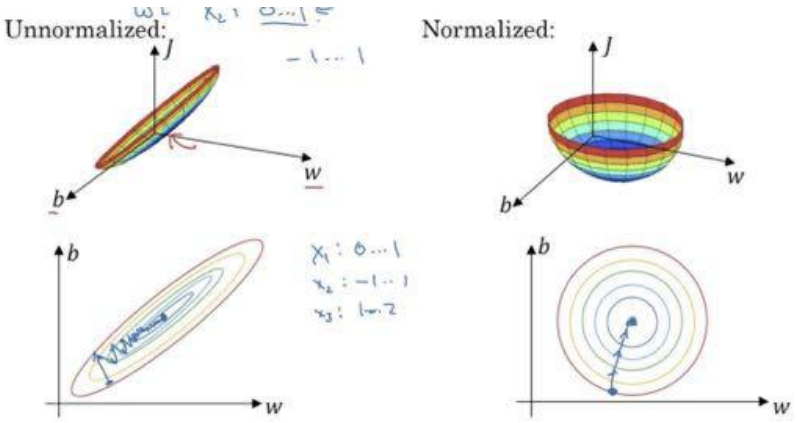
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

Normalize variance (element-wise):

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

$$x \mathrel{/}= \sigma$$

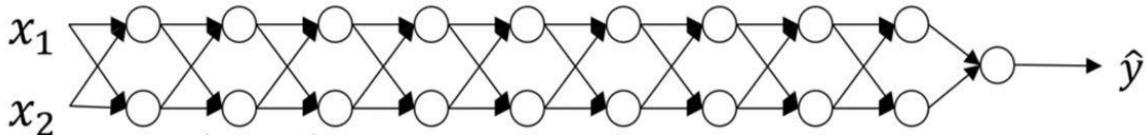


If features are at different scales, it might slow down gradient descent.

Hence, we should always normalize

Vanishing / Exploding Gradients

When we train very-very deep networks, our gradients may be really small or really large.



Let's run a quick example.

- use linear function $g(z) = z$
- not use bias $b^{[\ell]} = 0$

Then, $\hat{y} = W^{[L]} W^{[L-1]} \dots W^{[2]} W^{[1]} X$

If we had

$$W^{[\ell]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

Then, activations would get exponentially larger and larger

If we used

$$W^{[\ell]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

They would get smaller and smaller.

If the weights are little bit greater than identity / in a very deep network, activations (and gradients) will grow exponentially.

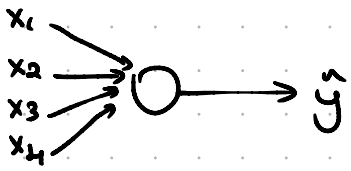
→ makes training difficult

→ Weight initialization Helps

Weight Initialization

Why: careful choice for how weights are initialized can help with gradient vanishing / exploding problem.

Single Neuron Example



$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + x_4 x_4$$

NOTE THAT

to prevent blowup, we want w_i's to be small

large n → smaller w_i

this is a good goal → $\text{Var}(w_i) = \frac{1}{n}$

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

$n^{[\ell-1]}$: number of features in layer ℓ

IDEA: can write a Medium article on which runs better

if we especially use ReLU function,

$$\text{Var}(w_i) = 2/n \quad \text{works better}$$

$$W^{[\ell]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{2}{n^{[\ell-1]}}\right)$$

if we use tanH function

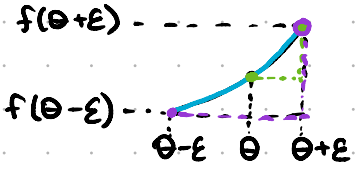
$$\text{Var}(w_i) = 1/n^{[\ell-1]}$$

Some other authors use

$$\text{Var}(w_i) = 2/(n^{[\ell]} + n^{[\ell-1]})$$

Numerical Gradient Approximation

$f(\theta) = \theta^3$



best way to approximate is take two triangles

1 Triangle

$$\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$

Approximation Err: 0.03

Error $\in O(\varepsilon)$

2 Triangles

$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$$

Approx. error 0.0001

Error $\in O(\varepsilon^2)$

(at $\theta = 1$)

in two triangle method V.S. in one triangle method

Error $\in O(\varepsilon^2)$ Error $\in O(\varepsilon)$

Gradient Checking (super useful for checking backprop implementation)

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

then $\Rightarrow J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = J(\theta)$

2) Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$

Remember, J is now a function of θ.

Grad Check Implementation:

for each i:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \varepsilon, \dots, \theta_n) - J(\theta_1, \theta_2, \dots, \theta_i - \varepsilon, \dots, \theta_n)}{2\varepsilon}$$

$d\theta_{approx}$ will be same dimensions as $d\theta$. We want to check if these two vectors are approximately equal

Check

EUCLIDIAN DIST

$$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$$

notice that there is no square on top, so this is sum of squared of elements, then square root of the sum

$\varepsilon = 10^{-7}$

What Does Values Mean?

$\approx 10^{-7}$ great!

10^{-5}

10^{-3} worry...

HOW to FIX

Check the vectors to see if there is a specific value of i such that

$d\theta_{approx} \neq d\theta$

then debug + run test again

Gradient Checking Implementation Notes

- grad check is costly. don't use in training. only use to debug
- check which value of $d\theta$ is different
- remember regularization (check that dJ includes it)
- doesn't work with dropout $\rightarrow$ J is different
  - ↳ workaround:
    - implement grad check before dropout
    - implement dropout after checking backprop correct.