Setting up your Machine Learning process

Train, Dev and Test

Applied machine learning is a highly iterative process: You start with an idea, you experiment with it, get back results and based on the outcome you refine your ideas, change your choices and repeat the process.

To train a neural network, you have several decisions to make:

- Number of laters
- Number of hidden units
- Learning rate
- Activation functions

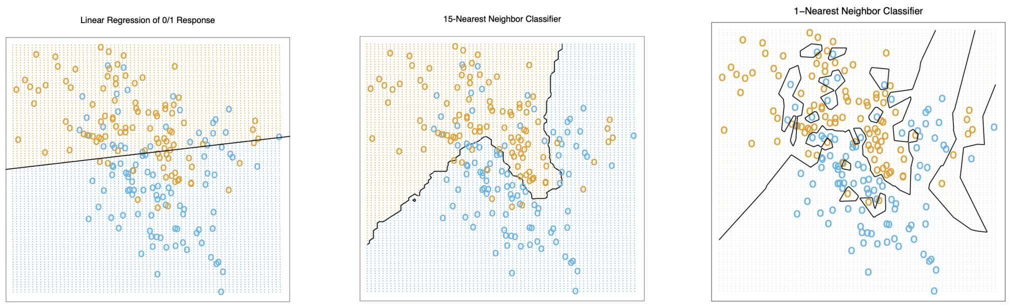
One of the things that determine how quickly you can make progress is how efficiently you can go around this cycle. Making good choices in how you set up your training, development, and test sets can make a huge difference in helping you quickly find a good high performance neural network.

Usually, the rule of thumb in machine learning is a 60% train, 20% dev and 20% test partition of the dataset. It's important to consider the size of your dataset, since the dev and test sets only are to evaluate the performance of a model, it might be sufficient to have 10,000 observations in each set, for a dataset of a million observations that split would be 98% train, 1% dev and 1% test.

Another rule of thumb is to make sure that the dev and test sets come from the same distribution.

It might be okay not to have a test set. The goal of a test set is to give an unbiased estimate of the error of your algorithm, if you don't need that estimate it's okay not to have a test set.

Bias and Variance



(a) High bias (b) Just right (c) High variance

Figure 1: Bias and variance

Since we have several dimensions it's difficult to plot the decision boundary. We can compare the train and dev errors in order to obtain how is our bias and variance:

- If the difference between train and dev errors is small we have low variance and if it's big we have high variance.
- If the train error is near to the optimal (bayes) error we have low bias and in the other case we have high bias.

The following curve ilustrates this tradeoff:

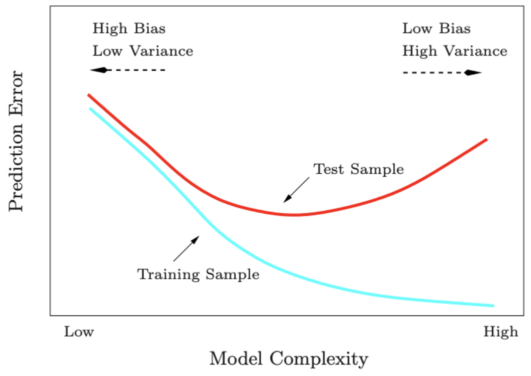


Figure 2: Test and training error as a function of model complexity

Basic Recipe for Machine Learning

If the model has a high bias (train set performance) you can:

- Train a bigger network
- Train longer
- Try a different Neural Network architecture

If the model has a high variance (test set performance) you can:

- Get more data (if possible)
- Regularization
- Try a different Neural Network architecture

In "traditional" machine learning we speak of the bias-variance tradeoff because we could only increase one sacrificing the other. In deep learning is a little bit difference training a bigger network almost always reduce bias and doesn't increase variance (with proper regularization) and getting more data almos always reduce variance without hurting the bias. This is why deep learning has been so successfull with supervised learning.

Regularizing your netural network

Regularization

If you have a high variance problem (your neural network is overfitting the data) you should try regularization.

Let's begin explaining how regularization works for the logistic regression, we add the following term to the loss:

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

Where λ is called the *regularization parameter* and $\|\dot{}\|_p$ is the p norm, the must common regularizations are:

- $p = 2$ (L2 regularization)

- $p = 1$ (L1 regularization)

L2 regularization is the most common and L1 regularization is used because it makes the solution sparse.

For a neural network, the regularization looks as follows:

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

Where $\|W^{[l]}\|$ is the *Frobenius* norm of W.

How do you implement gradient descent with this?

$$\begin{aligned} dW^{[l]} &= \text{(from backprop)} + \frac{\lambda}{m}W^{[l]} \\ W^{[l]} &:= W^{[l]} - \alpha dW^{[l]} \\ &= W^{[l]}\left(1 - \frac{\alpha\lambda}{m}\right) - \alpha\text{(from backprop)} \end{aligned}$$

That's why this kind of regularization is also known as *weight decay* because it's just like the original backpropagation but you are multiplying the weght matrix $W^{[l]}$ by $\left(1 - \frac{\alpha\lambda}{m}\right)$ a number strictly smaller than 1.

Why regularization reduces overfitting? As lambda gets bigger the penalization for W being a large matrix get's bigger and so the error is bigger. Intuitively as we make λ bigger we start making the impact of more and more hidden units negligible and therefore obtaining a more simple network that is therefore less prone to overfitting.

Dropout regularization

With dropout we go through each layer of the network and set some probability p of eliminating a node in the neural network so you end up with a much smaller, diminished network.

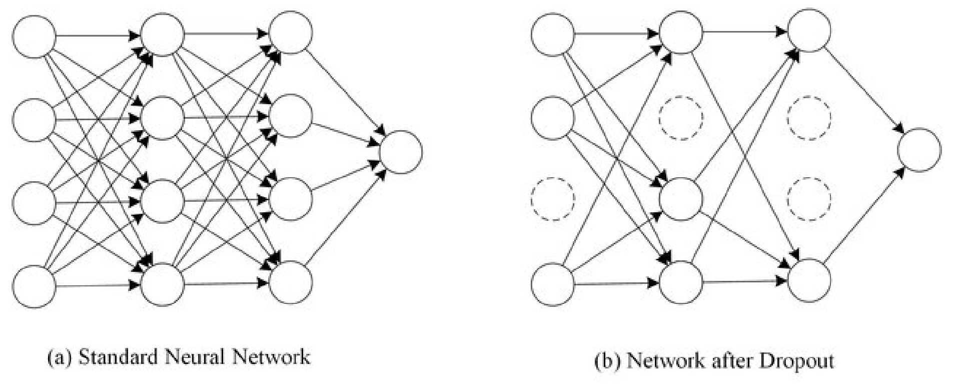


Figure 3: Ilustration of regularization

Inverted dropout After removing nodes from the layer l with probability p we have to divide a by that probability so that the expected value remains the same.

Notice that each layer can have different probability p, for more dense layers you might use smaller p (probability to keep a node) and in thinner layers that p can be bigger or even 1 (no dropout).

Also notice that at test time we shouldn't do any dropout (you would just add noise to your predictions)

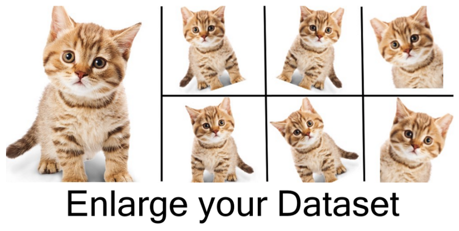
Why does dropout work? The intuition is that a single unit can't rely solely in any feature because any one feature could be made zero (because of droput) so it has to spread and give a little bit of weight to each incoming feature. By spreading weights it has an effect of shrinking the squared norm of the weights, similar to L2 regularization.

Other regularization methods

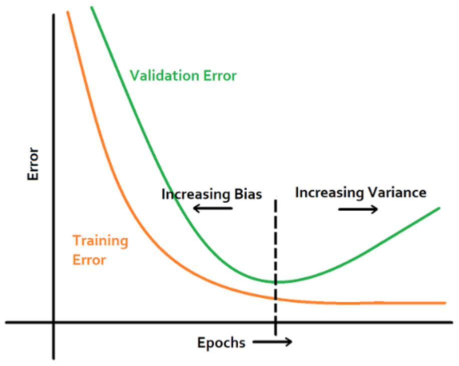
Data augmentation: Take the examples of an neural network which purpose is image classification. Then you can flip, zoom, rotate and perform an slight distortion to each image. The new dataset is bigger and might help reducing to reduce the variance.

Early stopping: By stopping early you have a mid-size W that hasn't still learned some really specific configurations of the train dataset and therefore overfitting.

Early stopping has one downside, in general you would like to follow the principle of orthogonalization, that is, solving different tasks at different times. When training neural networks we have to worry about optimizing the cost function J and after that to not overfit. In general we can tackle this two problem in a totally independent way but early stopping couples this two tasks.



(a) Data augmentation



(b) Early stopping

Figure 4: Other regularization methods

A better alternative to early stopping is just using L2 regularization and training for as long as possible. The downside of this alternative is that you add another hyperparameter (λ).

Setting the optimization problem

Normalizing inputs

In general, the cost function of the optimization problem is easier to optimize when the training dataset is normalized. To normalize it you have to:

- Obtain the dataset mean $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ and substract it to every observation $x^{(i)} = x^{(i)} - \mu$
- Obtain the dataset variance $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$ and divide each observation $x^{(i)} = \frac{x^{(i)}}{\sigma^2}$

Notice that you should store this values of μ and σ^2 so you can normalize your test set with this same μ and σ^2 values.

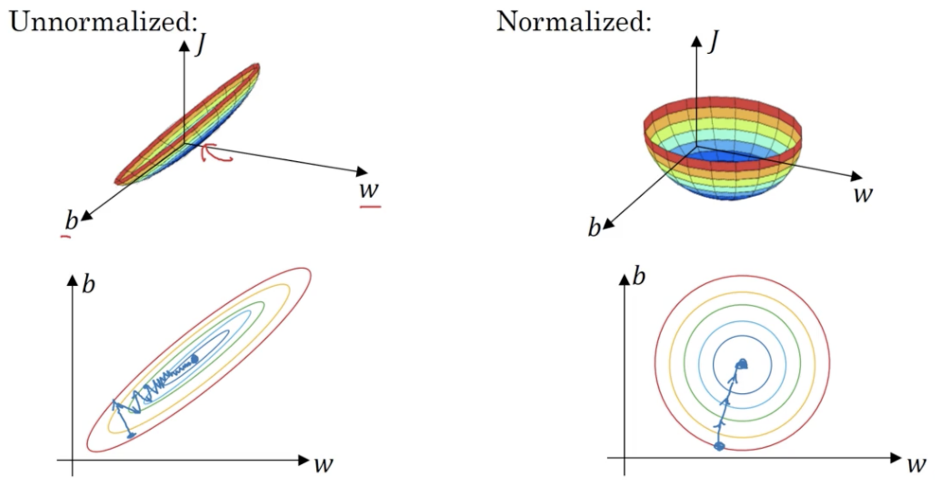


Figure 5: Comparison of countours of the normalized an unnormalized cases

Vanishing / Exploding gradients

One of the problems of training neural network, especially very deep neural networks, is vanishing and exploding gradients, that is, the derivatives can get either very big or very small; this makes the training very difficult.

Weight Initialization for Deep Networks: Usually a better or more careful choice of the random initialization for the neural network can address the problem of vanishing and exploding gradients.

The recomenended initialization of the weights matrix at the l layer is:

$$W^{[l]} \sim N\left(0, \frac{2}{n^{[l-1]}}\right) \qquad \text{for relu activations}$$

$$W^{[l]} \sim N\left(0, \sqrt{\frac{1}{n^{[l-1]}}}\right) \qquad \text{for tanh activations}$$

In this way each W matrix has a norm near to 1 for each layer l trying to prevent the gradients to vanish or explode to quickly.

Gradient checking

When implementing backpropagation there's a test called *gradient checking* that can help you to make sure that your implementation of backpropagation is correct. In order to aproach this test we need to introduce first a way tu approach numerically the gradients.

Numerical approximation of gradients: We can use the forward difference where, for $\varepsilon > 0$:

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

A better expression is the central (two sided) difference where:

$$\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} = f'(\theta) + O(\varepsilon^2)$$

Notice that now the error is $O(\epsilon^2)$ instead of $O(\epsilon)$. There are also Higher-order differences that might give a better approximation but are also more complicated to calculate.

Gradient checking procedure:

1. Reshape $W^{[1]}, b^{[1]}, ..., W^{[L]}, b^{[L]}$ into a single vector θ
2. Reshape $dW^{[1]}, db^{[1]}, ..., dW^{[L]}, db^{[L]}$ into a single vector $d\theta$
3. Compute $\hat{d\theta_i} = \dfrac{J(\theta_1, ..., \theta_i + \varepsilon + ...) - J(\theta_1, ..., \theta_i - \varepsilon + ...)}{2\varepsilon}$
4. Compute $\delta = \dfrac{\|\hat{d\theta_i} - d\theta_i\|}{\|\hat{d\theta_i}\| + \|d\theta_i\|}$.
5. If δ is smaller than ε there's no reason to woerry, if it's around epsilon its okay but double check your backpropagation algorithm and if it's much bigger than epsilon then it might be a bug.

It's important not to use it in training (it's really slow). It should be used just to debug the implementation.