

Setting up your machine learning application.

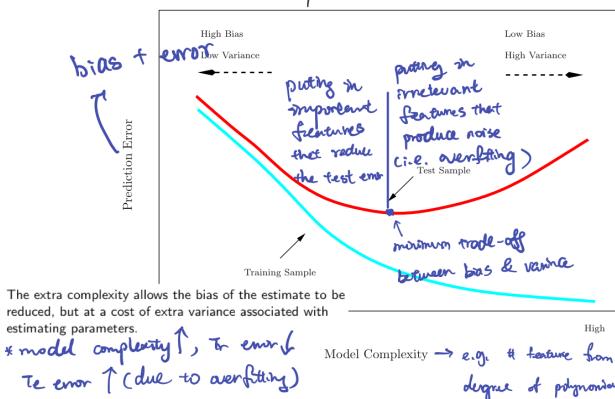
First we need to split the data into :

TRAINING SET	DEV/VALIDATION SET	TESTING SET
model fitting	model selecting (e.g. select model from subsets select, or selecting λ for lasso)	model assessment (i.e. prediction accuracy).

Why do we split data to assess model performance :

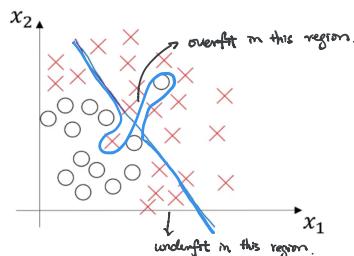
(1) Bias / Variance Trade-off

- DEV error is the average error that results from using a model to predict the response on DEV dataset
- TRAN error is the residuals that result from a model fitting on observations
- If we use TRAN error, we tend to choose the model with the smallest TRAN error and thus over-fit the data (low bias). Then the model is too complex and might include unnecessary terms, i.e. high variance. This is also known as the bias/variance tradeoff.
- So TRAN error doesn't tell us how well the model performs on new data, i.e. the variance problem of the model, but the DEV error indicates how bad the variance problem is.



(2) High bias and high variance

- There is also model that produce high variance and high bias because it underfits part of the data as well as overfits other part of the data. e.g.



Andrew Ng

Then we can try these to improve our model:

(1) High bias \rightarrow bigger network

- train longer
- more advanced optimization algorithms
- other neural network architecture

(2) High variance \rightarrow more data

- regularization
- other neural network architecture.

trying these methods to drive down bias/variance almost never hurt the other one, which is known as bias/variance tradeoff where you have to carefully balance these two, which is known as Orthogonality Principle, where we can try separate tools to work on bias/variance independently.

Regularization.

Method 1:

$$w, b = \min_{w, b} J(w, b) = \min \left\{ \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \text{regularization-term} \right\}$$

↓ a function of regularization parameter and other parameters (e.g. w, m)
minimise cost (i.e. ensure fitness)

by minimising it, λ penalise w for being too large,
so effectively shrink them towards 0 $\Rightarrow \hat{z}$ will be relatively
small and g(z) will be relatively linear:
 \Rightarrow simpler model



where the regularization-term can be:

- L_2 regularization:

logistic regression: $J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2 + \frac{\lambda}{2m} b^2$ → omit this because it's small comparing to high-dimensional w.

where $\|w\|_2^2 = \sum_{j=1}^n w_j^2 = w^T w \rightarrow L_2$ norm

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

where $\|W^{[L]}\|_F^2 = \sum_{i=1}^n \sum_{j=1}^{n^{[L]}} (w_{ij}^{[L]})^2 \rightarrow L_2$ norm or Frobenius norm or weight decay

Then $dW^{[L]} = \frac{1}{m} dL^{[L]} \cdot A^{[L-1]T} + \frac{\lambda}{m} W^{[L]}$

and $W^{[L]} = W^{[L]} - dW^{[L]} = W^{[L]} - d(\dots + \frac{\lambda}{m} W^{[L]}) = (1 - \frac{\lambda}{m}) W^{[L]} - d(\dots)$

↳ each gradient descent shrinks the weight,
thus $\|W^{[L]}\|_F^2$ is also known as "weight decay"

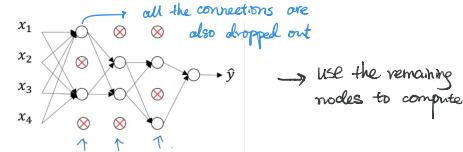
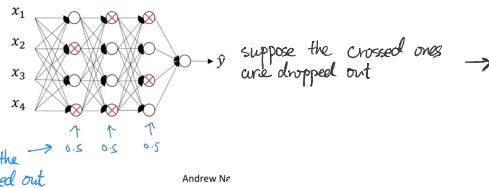
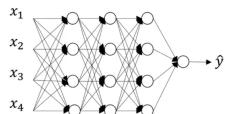
- L_1 regularization:

logistic regression: $J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_1$, → might produce a more sparse model

where $\|w\|_1 = \sum_{j=1}^n |w_j| \rightarrow L_1$ norm

(i.e. smaller model since many parameters will shrink to zero.)

Method 2: Dropout regularization.



Repeat this for many times and we will have different NNs from different iterations. The following will show how we can implement "Inverted Dropout", the most common technique, in python.

Let D_l denotes the dropout vector for layer l , then

$$D_l = np.random.rand(A_l.shape[0], A_l.shape[1]) < keep_prob$$

$$A_l = np.multiply(A_l, D_l) \rightarrow \text{since False times any number is 0, the corresponding nodes will be dropped out}$$

$$A_l / keep_prob \rightarrow \text{inverted dropout technique ensures the expected value of } A_l \text{ will remain the same after dropping } (1 - keep_prob)\% \text{ also eliminate the problem of scaling at test time.}$$

$$dA_l = np.multiply(dA_l, D_l) / keep_prob$$

if the generated random number is less than keep_prob,
then it will return True, otherwise False. So approximately
there will be keep_prob% nodes will be kept in layer l.

Note that at test time, there's no need to implement drop-out, because we don't want our output to be random, and we want cost function J to be well-defined.

How does dropout work so well with regularizer?:

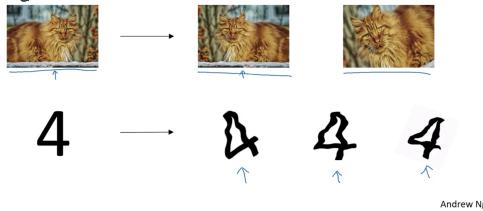
By removing some features randomly, it prevents the NN from assigning too many weights on particular features because they could go away. So it has to spread out weights. This would have the effect of shrinking weights similar to L_2 regularization.

Note that if we want to avoid over-fitting of some layers, we can apply dropout to some layers and don't apply dropout to other layers, but the downside is that we need to decide on which layers and keep_prob, i.e. more hyperparameters to use cross-validation for.

Other Methods:

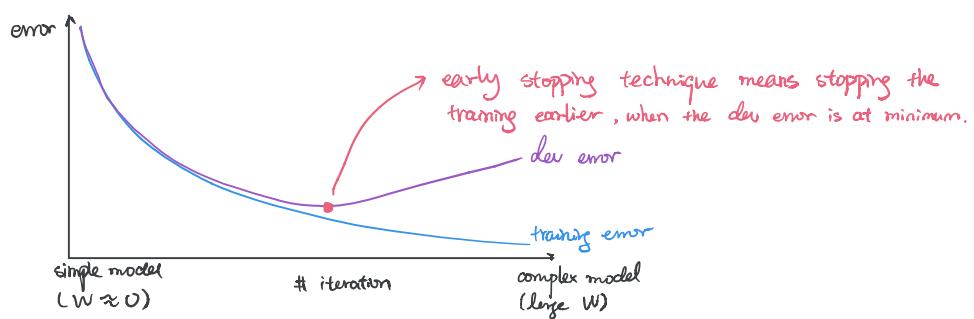
- Data augmentation

Instead of using brand-new examples, we can augment the existing data and treat those as new data. For example, flipping the images horizontally, randomly cropping the images etc.



Andrew Ng

- Early stopping

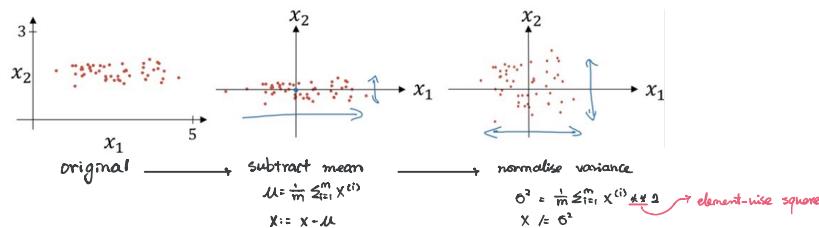


The downside of early stopping is that there's a tradeoff between model complexity and fitness. By stopping early we can't minimize the cost function even though it produces a simpler model, which is not in the favor of orthogonality principle.

Setting up optimization problem

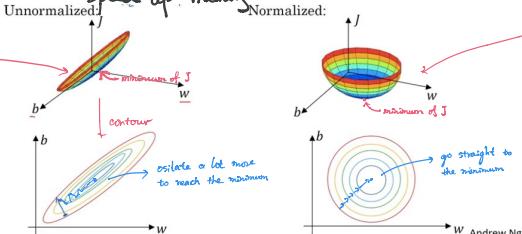
- normalising inputs

$$\text{let } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



why do we normalize? → speed up training

If \mathbf{x} ranges widely (e.g. 0-1 and 100-100), w and b tend to range widely as well so it would have a shape like this and small gradient.



more symmetric and larger gradient.

- Initializing weights for deep NN

Sometimes when the NN are very deep, it's likely that the gradients will decrease/increase exponentially (vanishing/exploding gradients). For example, $\hat{y} = w^{(L)} \begin{bmatrix} 1 & 0.5 \end{bmatrix}^{T, L} x$ suppose weight matrices are the same (<1 or >1) and activation is identity function for the sake of simplicity. Then \hat{y} could be exponentially small/large, which would make gradient exponentially small/large as well. This would make training difficult, especially if gradients are small then the gradient descent will take tiny little step in every iteration. Then it will take a long time to train the NN. A partial solution to this is properly initializing the weights by trying the followings:

$$\text{Var}(w^{(L)}) = \frac{1}{n^2 k^2} \Rightarrow w^{(L)} = np.random.rand(\text{shape}) * np.sqrt(\frac{1}{n^2 k^2})$$

if $g^{(l)}$ is ReLU, we want $\text{Var}(W^{(l)}) = \frac{2}{n^{(l+1)}}$ $\Rightarrow W^{(l)} = \text{np.random.randn(shape)} * \text{np.sqrt}(\frac{2}{n^{(l+1)}})$

if $g^{(l)}$ is tanh, we want $\text{Var}(W^{(l)}) = \sqrt{\frac{1}{n^{(l+1)}}}$ \Rightarrow Xavier initialization

other initialization such as Yoshua Bengio's, we want $\text{Var}(W^{(l)}) = \sqrt{\frac{2}{n^{(l+1)} + n^{(l)}}}$

we can also try $\text{Var}(W^{(l)}) = \frac{p}{n^{(l+1)}}$ where p is another parameter we can tune but often it has very modest effect.

• Gradient checking

Sometimes when we're not perfectly sure whether our back propagation implementation is correct, we can check by numerical approximation of gradients:

Since $f'(\theta) = \lim_{\epsilon \rightarrow 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$, we can approximate using $\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$ with very small ϵ (error = $O(\epsilon^3)$)

Now take $w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}$ and reshape into a big vector θ ; take $dw^{(1)}, db^{(1)}, \dots, dw^{(L)}, db^{(L)}$ and reshape into a big vector $d\theta$.

$(w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}) \Rightarrow (w_{1,1}^{(1)}, w_{1,2}^{(1)}, \dots, w_{n^{(1)},1}^{(1)}, b_{1,1}^{(1)}, \dots, b_{n^{(1)},1}^{(1)}, \dots, w_{1,2}^{(L)}, w_{1,3}^{(L)}, \dots, w_{n^{(L)},2}^{(L)}, b_{1,1}^{(L)}, \dots, b_{n^{(L)},1}^{(L)})$

similar for dw and db

Then $J(\theta) = J(\theta_1, \theta_2, \dots)$, for each i :

$$d\theta_{\text{approx}, i:i} = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

$$\approx d\theta_{i:i} = \frac{\partial J}{\partial \theta_i}$$

we can then check whether the approximate value is close enough to the real value by:

check	$\frac{\ d\theta_{\text{approx}} - d\theta\ _2}{\ d\theta_{\text{approx}}\ _2 + \ d\theta\ _2}$	$\begin{cases} \approx 10^{-7} & \rightarrow \text{great.} \\ \approx 10^{-5} & \rightarrow \text{okay but need double check} \\ \approx 10^{-3} & \rightarrow \text{concerning.} \end{cases}$	$\ x\ _2 = \sqrt{\ x\ ^2}$ $= \text{np.linalg.norm}(x)$
-------	---	--	--

note: - don't use when training, only use when debugging

- if algorithm fails grad check, look at components (e.g. w or b) to try to identify bug

- don't forget the regularization term

- grad check doesn't work with dropout as J is not well-defined (but we can set $\text{keep_prob}=1$ and grad check whether the algorithm is correct without dropout)

- run at random initialization, grad check again after some training