Setting up your machine learning application.

First we need to split the data into:

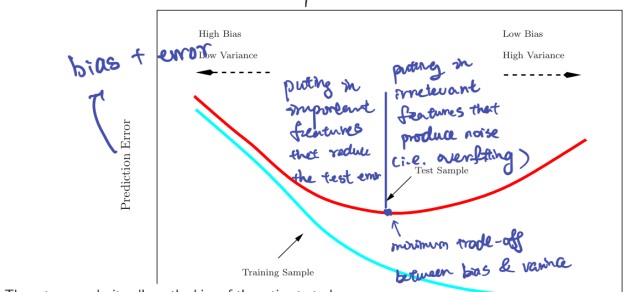
TRAINING SET	DEV/VALIDATION SET	TESTING SET
↓	↓	↓
model fitting	model selecting (e.g. select model from subsets select, or selecting λ for lasso)	model assessment. (i.e. prediction accuracy).

* usually split into 2:1:1 for small dataset (<10,000)
* usually split into 98:1:1 for large dataset (1,000,000)

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 perform on new data, i.e. the variance problem of the model, but the DEV error indicates how bad the variance problem is.



The extra complexity allows the bias of the estimate to be reduced, but at a cost of extra variance associated with estimating parameters.

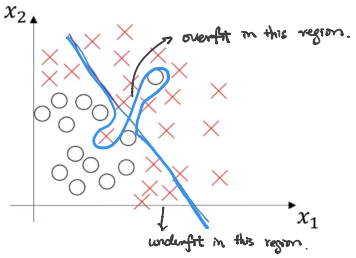
* model complexity ↑, Tr error ↓
  Te error ↑ (due to overfitting)

Model Complexity → e.g. # features from low to high, degree of polynomial from low to high.

3 / 44

(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 → bigger network
train longer
more advanced optimization algorithms
other neural network architecture

(2) High variance → more data
regularization
other neural network architecture.

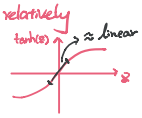
→ trying these methods to drive down bias/variance almost never hurt the other one, which is known as bias/variance tradeoff where you have to carefully balance these two, which is known as Orthorganization Principle, where we can try seperate tools to work on bias/variance independently.

Regularization.

Method 1:

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

- minimise cost (i.e. ensure fitness)
- a function of λ (regularization parameter) and other parameters (e.g. w, m)
  - by minimising it, λ penalise w for being too large, so effectively shrink them towards 0 $\Rightarrow$ z will be relatively small and $g(z)$ will be relatively linear: $\tanh(z) \approx$ linear
  - $\Rightarrow$ simpler model



where the regularization-term can be:

- L_2 regularization:

logistic regression: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 + \frac{\lambda}{2m}b^2$ (regularization parameter; use cross-validation to decide)

omit this because it's small comparing to high-dimensional w.

where $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ → l_2 norm

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

where $\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$ → l_2 norm or Frobenius norm or weight decay

Then $dw^{[l]} = \frac{1}{m} dz^{[l]} \cdot A^{[l-1]T} + \frac{\lambda}{m} w^{[l]}$

and $w^{[l]} = w^{[l]} - \alpha \cdot dw^{[l]} = w^{[l]} - \alpha(\cdots + \frac{\lambda}{m} w^{[l]}) = (1 - \frac{\alpha\lambda}{m}) w^{[l]} - \alpha(\cdots)$

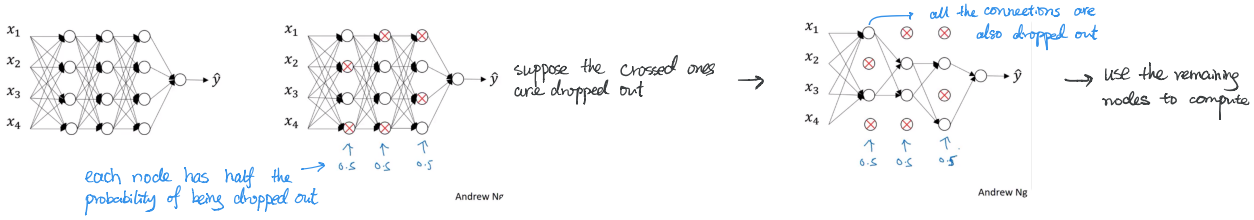
↳ each gradient descent shrinks the weight, thus $\|w^{[l]}\|_2^2$ is also known as "weight decay"

- L_1 regularization:

logistic regression: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_1$ (regularization parameter) → might produce a more sparse model (i.e. smaller model since many parameters will shrink to zero.)

where $\|w\|_1 = \sum_{j=1}^{n_x} |w_j|$ → l_1 norm

Method 2: Dropout regularization.



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

Let `dl` denotes the dropout vector for layer l, then

```
dl = np.random.rand(Al.shape[0], Al.shape[1]) < keep-prob
Al = np.multiply(Al, dl)
Al /= 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.
- since False times any number is 0, the corresponding nodes will be dropped out
- inverted dropout technique ensures the expected value of Al will remain the same after dropping (1-keep_prob)% also eliminate the problem of scaling at test time.

Note that at test time, there's no need to implement drop-out, for one reason it's inefficient and wouldn't make much difference. Also 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?:

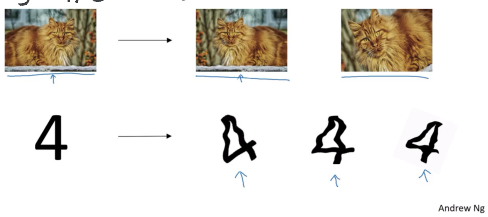
By removing some features randomly, it prevents the NN from assigning too many weights on particular features because they could go away. So it has to spread out weights. This would have the effect of shrinking weights similar to l_2 regularization.

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_probs, i.e. more hyperparameters to use cross-validation for.

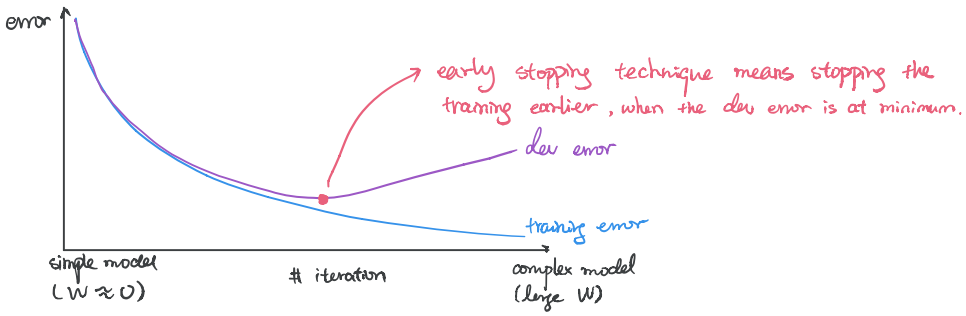
Other Methods:

- 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.



- Early stopping

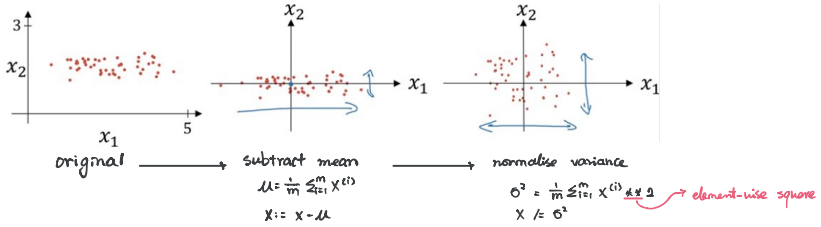


The downside of early stopping is that there's a tradeoff between model complexity and fitness. By stopping early we can't minimise the cost function even though it produces a simpler model, which is not in the favor of orthorganization principle.

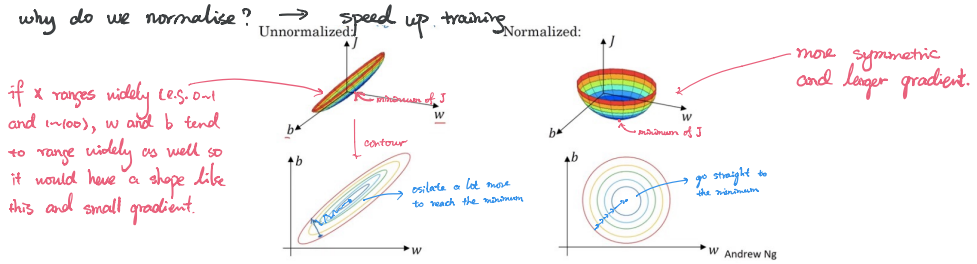
Setting up optimization problem

- normalising inputs

  let $X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



why do we normalise? → speed up training



- 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, $\hat{y} = W^{[L]}\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x$ suppose weight matrixes 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:

  $$Var(W^{[l]}) = \frac{1}{n^{[l-1]}} \Rightarrow W^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$$

if $g^{[l]}$ is Relu, we want $Var(w^{[l]}) = \frac{2}{n^{[l-1]}}$ $\Rightarrow$ $w^{[l]}$ = np.random.randn(shape) * np.sqrt($\frac{2}{n^{[l-1]}}$)
if $g^{[l]}$ is tanh, we want $Var(w^{[l]}) = \sqrt{\frac{1}{n^{[l-1]}}}$ $\Rightarrow$ Xavier initialization
other initialization such as Yoshua Bengio's, we want $Var(w^{[l]}) = \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
we can also try $Var(w^{[l]}) = \frac{p}{n^{[l-1]}}$ 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 $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$, we can approximate using $\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ with very small ε (error = $O(\varepsilon^2)$)

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

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

$\approx$ $d\theta[i] = \frac{\partial J}{\partial \theta_i}$

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

$$\text{check } \frac{\| d\theta_{approx} - d\theta \|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \begin{cases} \approx 10^{-7} \rightarrow \text{great.} \\ \approx 10^{-5} \rightarrow \text{okay but need double check} \\ \approx 10^{-3} \rightarrow \text{concerning.} \end{cases}$$

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 let keep_prob = 1 and grad check whether the algorithm is correct without dropout)
- run at random initialization, grad check again after some training