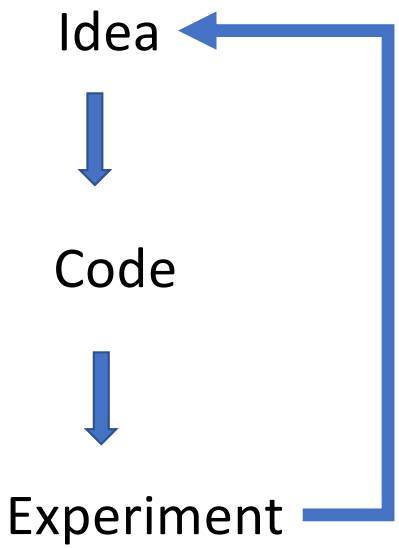
Improve NN model by Validation and regularization

Qingrun Zhang

ML is a highly iterative process

- # layers
- # hidden units
- Learning rates
- Activation functions
- Regularization
- ….



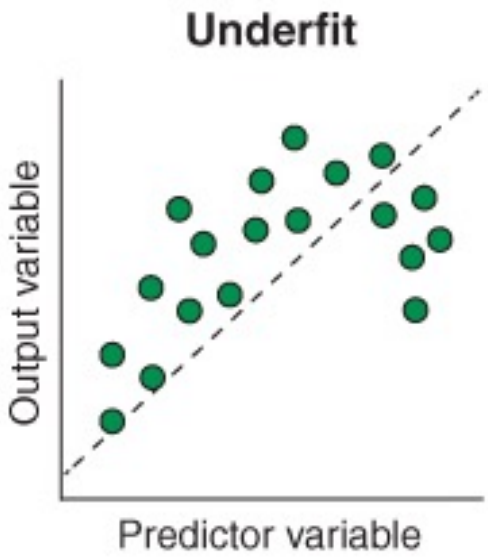
Train/Dev/Test datasets

Training set to train your models
Cross Validation/Development Set to evaluate which model works the best
Test set to estimate of how well the model is doing
Common practice 60/20/20 percentage for each set (when total data point less than 10K)
In big data era, for example 1M example, 10K for Dev and 10K for test is enough 98/1/1

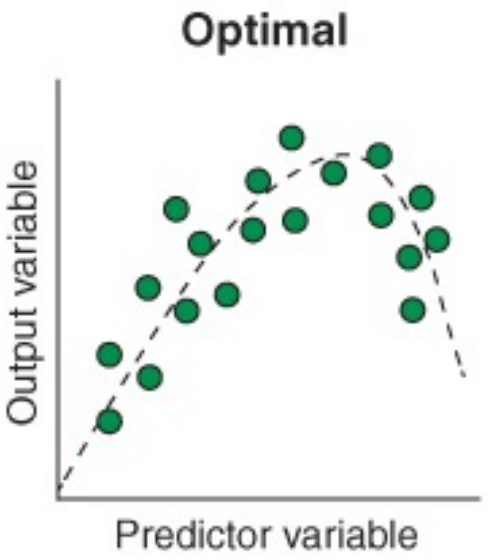
Mismatched train/test distribution

- When we have mismatched train and test set, for example, different image quality for train and test set, the thumb of rule is to make sure Dev and train set should come from the same distribution.
- Sometimes it is OK of not having test set, only Dev set is fine

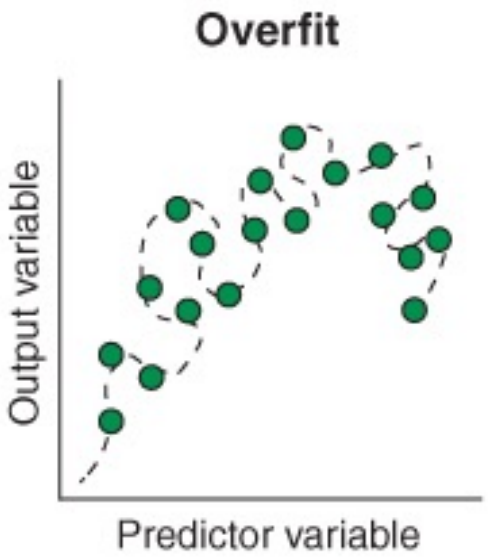
Bias and Variance



High Bias



Just right



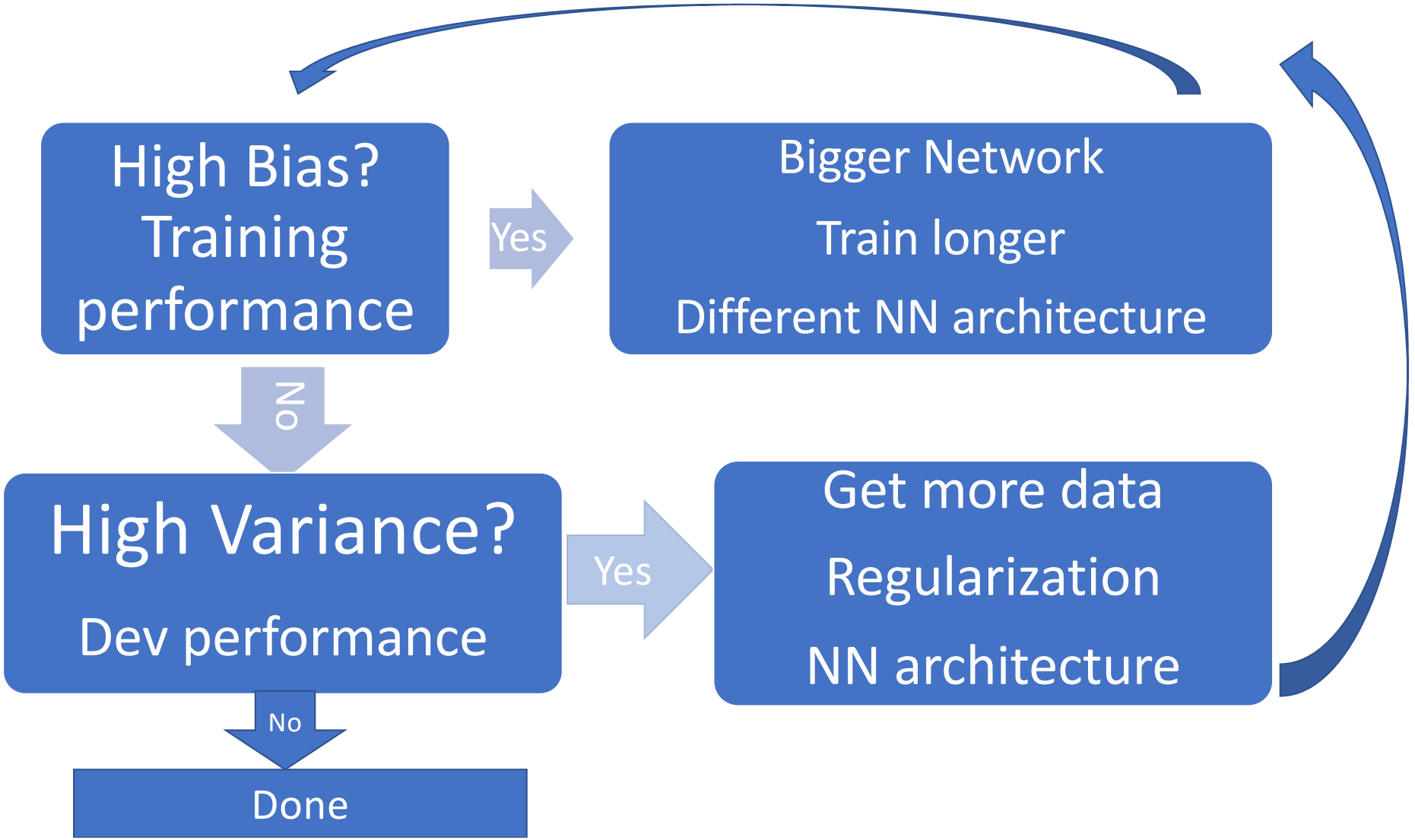
High Variance

https://livebook.manning.com/book/machine-learning-for-mortals-mere-and-otherwise/chapter-11/21

Bias and Variance

- Picture classification: Cat or Dog. Human optimal error is close to 0%

Training set error	1%	20%	0.5%	10%
Dev set error	10%	21%	1%	20%
	High variance	High bias	Low bias Low variance	High bias High variance

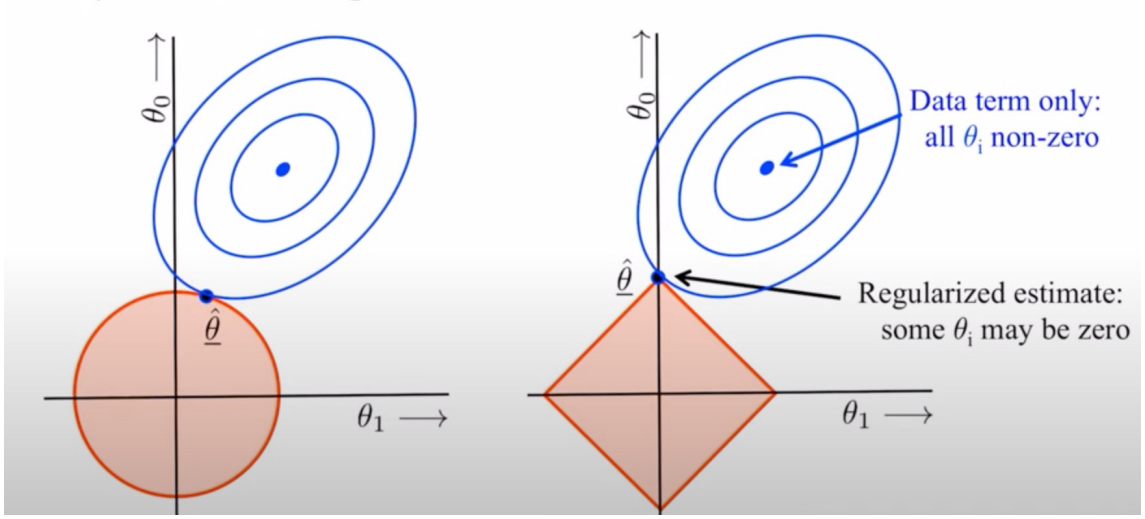

High Bias?
Training performance
Yes
Bigger Network
Train longer
Different NN architecture
No
High Variance?
Dev performance
Yes
Get more data
Regularization
NN architecture
No
Done

Regularization

λ *is the regularization parameter*

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

- L1 regularization: $\frac{\lambda}{2m}\sum_{j=1}^{n_x}\|w\|$
- L1 regularization w will be sparse
- L2 regularization:
- $\frac{\lambda}{2m}\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$



Regularization in Neural Network

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

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

- Frobenius Norm

$$\|A\|_F \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$$

- $dw^{[l]} = \frac{\partial J}{\partial w^{[l]}}$
- $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$

- With regularization term
- $dw^{[l]} = \frac{\partial J}{\partial w^{[l]}} + \frac{\lambda}{2m} w^{[l]}$
- $w^{[l]} = w^{[l]} - \alpha(\frac{\partial J}{\partial w^{[l]}} + \frac{\lambda}{2m} w^{[l]})$
- $w^{[l]} = w^{[l]} - \alpha \frac{\lambda}{2m} w^{[l]} - \alpha \frac{\partial J}{\partial w^{[l]}}$
- Weight decay

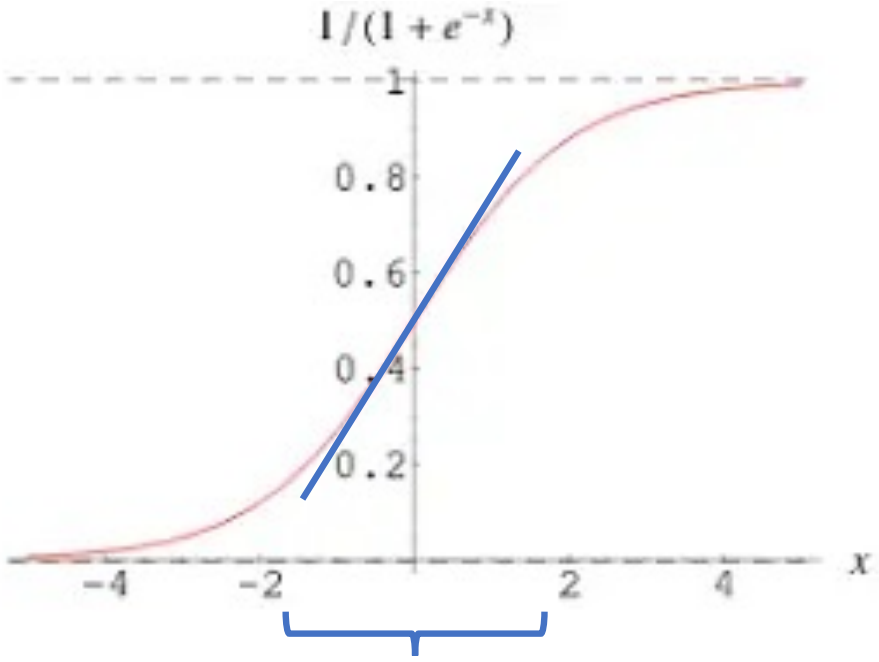
Why regularization prevent overfitting?

- $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|w^{[l]}\right\|_2^2$
- When λ is big, which will force $w^{[l]} \approx 0,$ when $w^{[l]} \approx 0$ which means the complicated model will get simplified, make it closer to linear model
- $Z^{[l]} = W^{[l]}A^{[l-1} + b^{[l]}$
- When $w^{[l]} \approx 0$, $Z^{[l]}$ is more likely

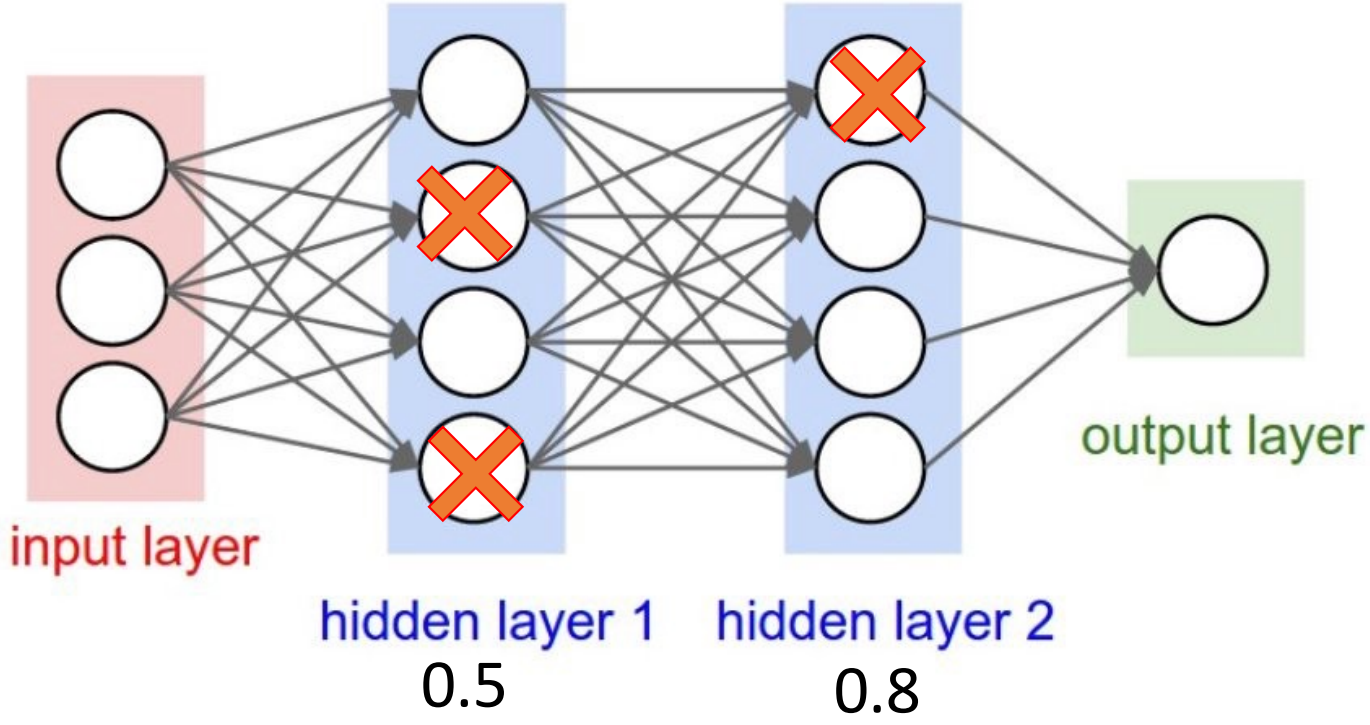
Fall in the range where the sigmoid

or tanh function at the linear

part



Dropout regularization

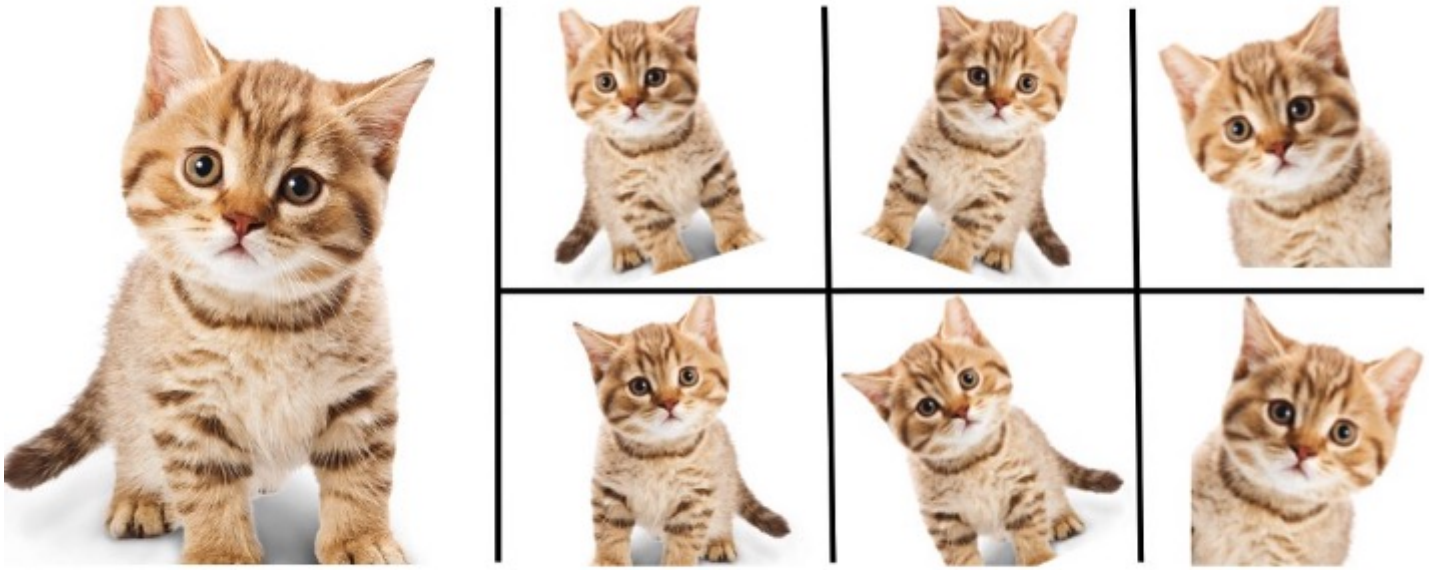


At test time, we don’t use dropout.

Why does dropout work?

- Random knockout units in the network, the network is getting smaller.
- With dropout, the inputs get randomly eliminated, it can't rely on any one feature, so have to spread out weights, which will shrink the squared norm of the weight, similar to L_2 regularization
- Different layer can have different keep-prob.
- Computer vision input size is very big that you almost never have enough data, so dropout is widely used by computer vision. But not neccessary other type of data.

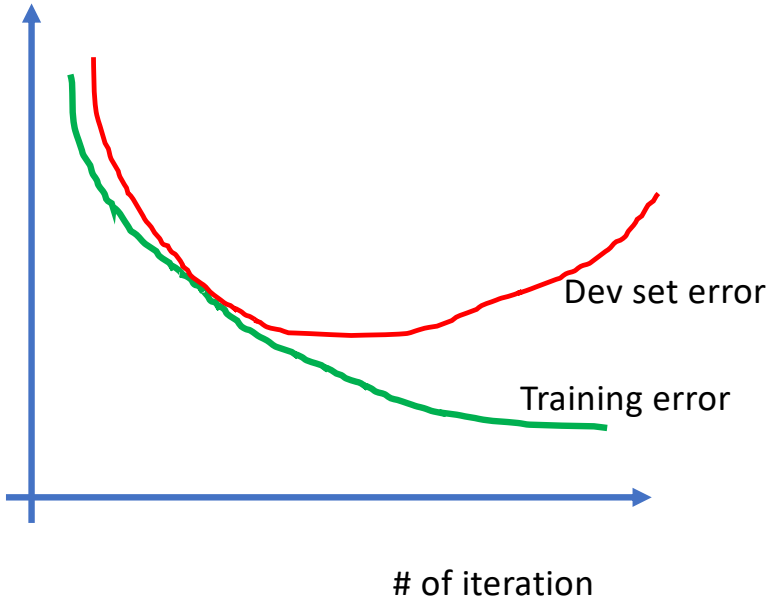
Data augmentation



https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Another approach: early stopping

- Philosophy: **Stop** if we may on a wrong way!
- Assumption: we have an **independent** dataset



Early stopping: an example

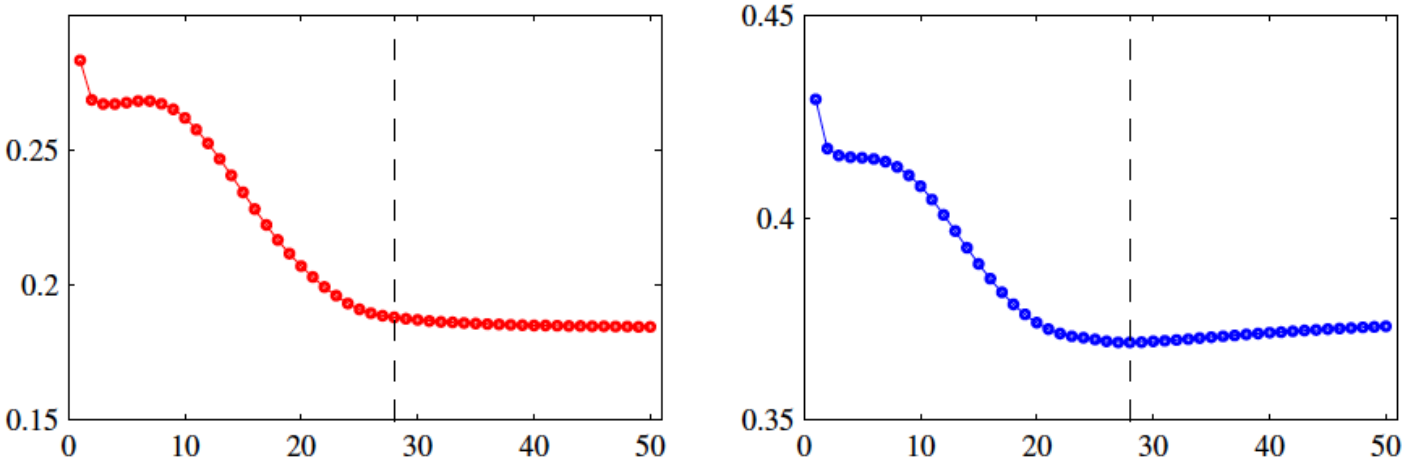


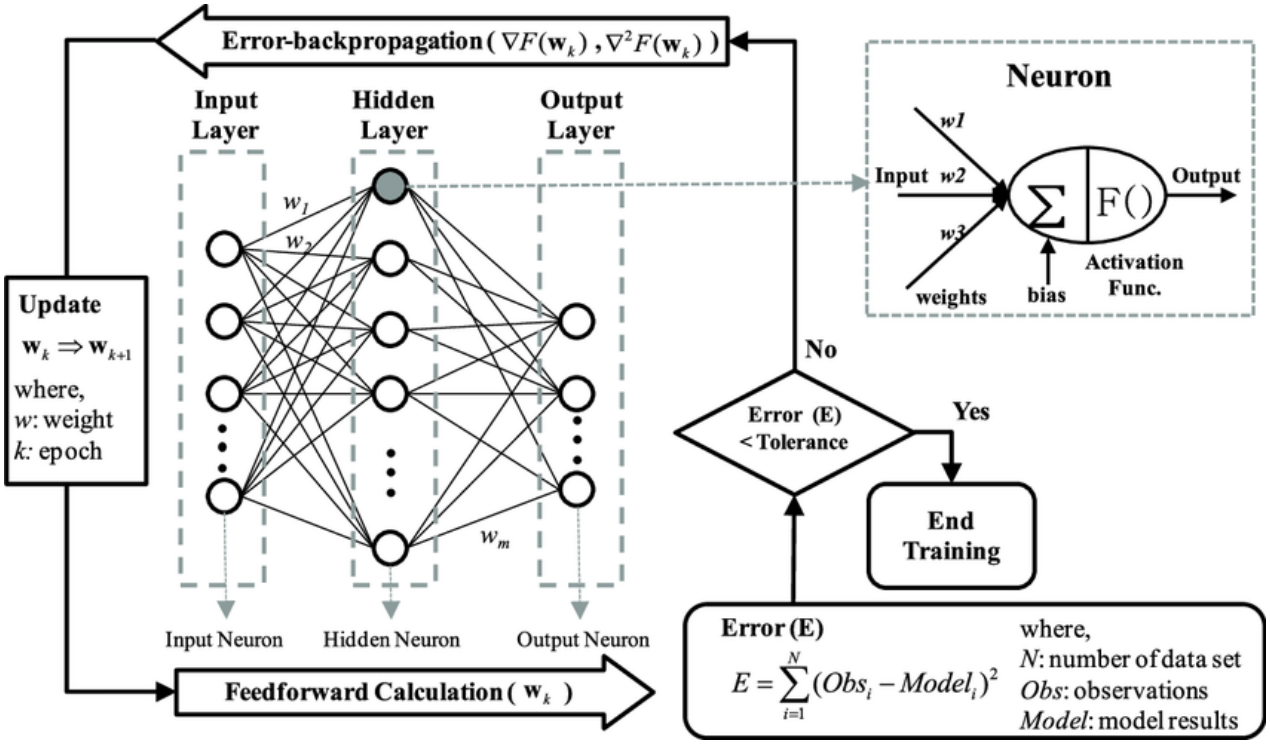
Figure 5.12 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

Code examples:

- Validation, weight decay and dropout
- https://colab.research.google.com/drive/1pikHsxLuZ7TZOrSe7iNt356mDZXra07I#scrollTo=nVfVcJXG9JC0
- https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/overfit_and_underfit.ipynb

- CallBack
- https://colab.research.google.com/github/tensorflow/docs/blob/snapshot-keras/site/en/guide/keras/custom_callback.ipynb

Vanishing and exploding gradients



https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

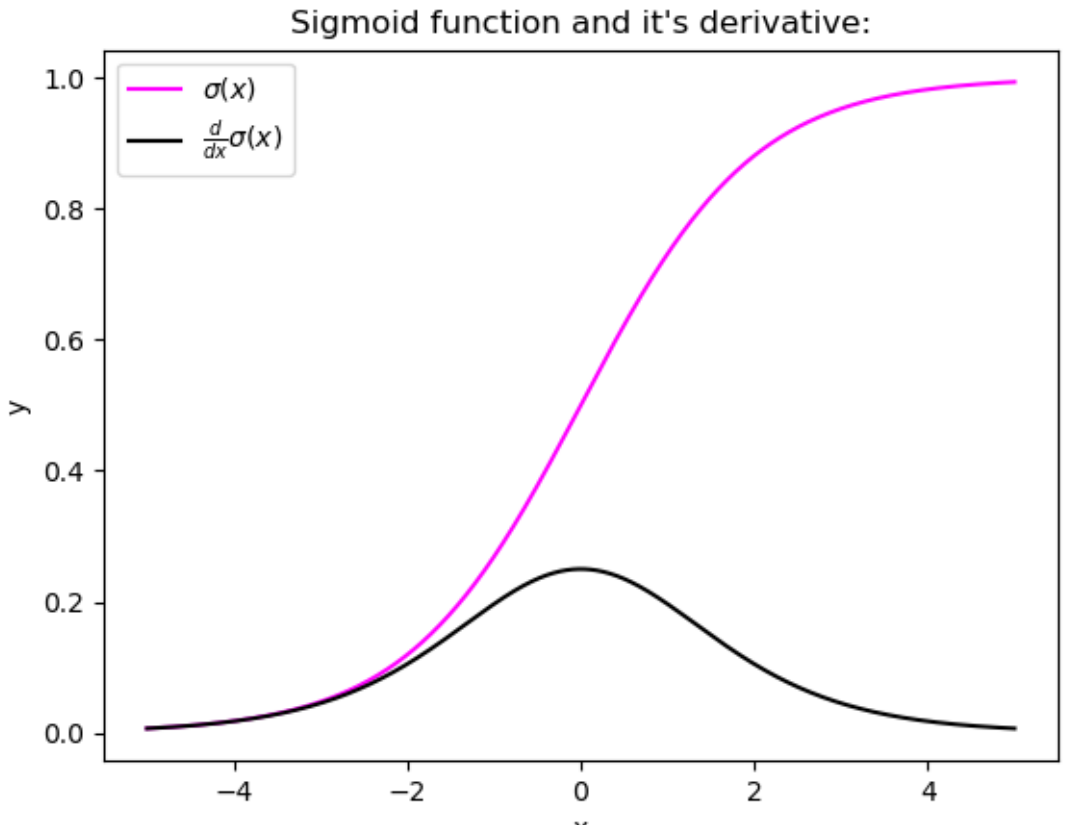
Vanishing and exploding gradients

- $a^{[1]} = g^{[1]}\left(w^{[1]}a^{[0]} + b\right), a^{[2]} = g^{[2]}\left(w^{[2]}a^{[1]} + b\right) \ldots.$
- $a^{[n]} = g^{[n]}(g^{[n-1]} \ldots . . g^{[1]}\left(w^{[1]}a^{[0]} + b\right) \ldots)$
- For simplicity: the activation function for each layer are the same linear function, ignore the bias terms:
- $w^{[1]}a^{[0]} \rightarrow w^{[2]}w^{[1]} \rightarrow \cdots \rightarrow w^{[n]} \ldots w^{[1]}a^{[0]}$
- Further simplify: $w^{n}a^{[0]}$
- With very deep network (large n), when $w > 1$ (e.g. $1.5^{16} = 656.84$) we get exploding weight. When $w > 1$ (e.g. $0.5^{15} = 3.05e - 05$) , we get vanishing weight

Vanishing and exploding gradients- backward

- $z^{[1]} = w^{[1]}x \rightarrow a^{[1]} = \sigma\left(z^{[1]}\right) \rightarrow z^{[2]} = w^{[2]}a^{[1]} \rightarrow \hat{y} = \sigma\left(z^{[2]}\right) \rightarrow J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$
- $J = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y) \rightarrow \frac{\partial J}{\partial w^{[2]}} \rightarrow \frac{\partial J}{\partial w^{[1]}}$
- $\frac{\partial J}{\partial w^{[2]}} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z^2}\frac{\partial z^2}{\partial w^2} = \left(\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\left[\sigma\left(z^{[2]}\right)\left(1 - \sigma\left(z^{[2]}\right)\right)\right]a^{[1]}$
- $\frac{\partial \hat{y}}{\partial z^2} = \sigma\left(z^{[2]}\right)\left(1 - \sigma\left(z^{[2]}\right)\right) = \frac{1}{1+e^{z^2}}\left(1 - \frac{1}{1+e^{z^2}}\right), when\ z^2$=5, $\frac{\partial \hat{y}}{\partial z^2} \approx 0.0066$
- $\frac{\partial J}{\partial w^{[1]}} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial w^1}$

Why do the gradients vanish/explode?



Larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero.

https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

When is the model suffering from the gradient exploding/vanishing?

Exploding	Vanishing
There is an exponential growth in the model parameters	The parameters of the higher layers change significantly while the parameters of lower layers change very little
The model weights may become NaN during training	The model weights may become 0 during training
The model experiences avalanche learning	The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations

https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

Solutions for gradient exploding/vanishing

- Proper Weight Initialization

Deep Sparse Rectifier Neural Networks

Xavier Glorot, Antoine Bordes, Yoshua Bengio *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, PMLR 15:315-323, 2011.

  - The variance of outputs of each layer should be equal to the variance of its inputs.
  - The gradients should have equal variance before and after flowing through a layer in the reverse direction.

- Xavier initialization or Glorot initialization $fan_{avg} = (fan_{in} + fan_{out}) / 2$
  - Normal distribution with mean *0* and variance $\sigma^2 = 1/ fan_{avg}$
  - Or a uniform distribution between $-r$ and $+r$, with $r = sqrt(3 / fan_{avg})$

Solutions for gradient exploding/vanishing

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Source: Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow

Keras uses Xavier's initialization strategy with uniform distribution by default, you can choose your own by

keras.layer.Dense(25, activation = "relu", **kernel_initializer="he_normal"**) **or**
keras.layer.Dense(25, activation = "relu", **kernel_initializer="he_uniform"**)

If we wish to use use the initialization based on fan_{avg} rather than fan_{in}, we can use the VarianceScaling initializer like this :

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg', distribution='uniform')
keras.layers.Dense(20, activation="sigmoid", kernel_initializer=he_avg_init)
```

https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

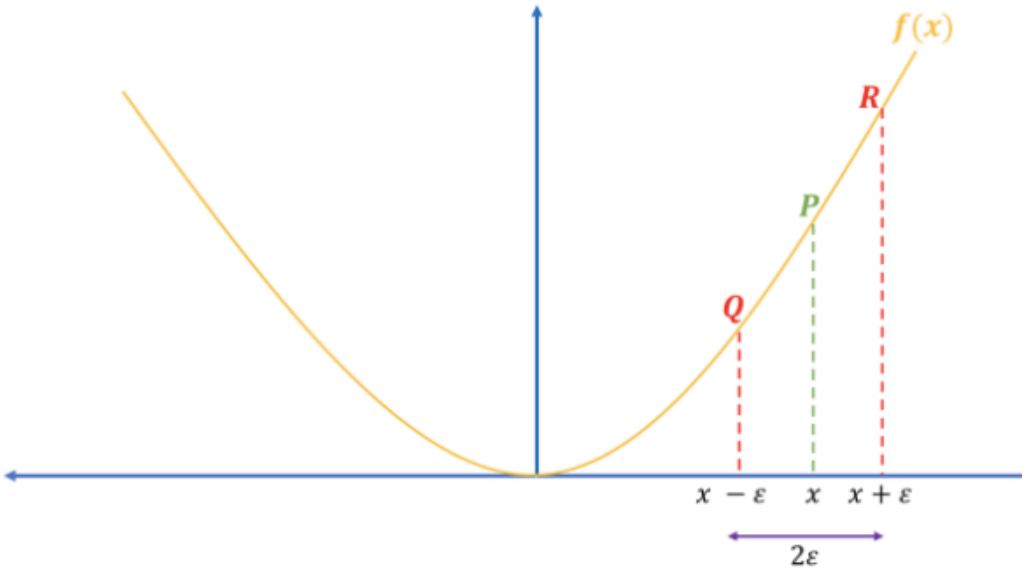
Solutions for gradient exploding/vanishing

- **Using Non-saturating Activation Functions**
  - **ReLU (Rectified Linear Unit)**
  - **LReLU (Leaky ReLU)**
- **Batch Normalization proposed by** Sergey Ioffe and Christian Szegedy in 2015
  - Added batch normalization after each layer
  - This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting.
  - keras.layers.BatchNormalization(),
- **Gradient Clipping:** clip the gradients during backpropagation so that they never exceed some threshold.
  - optimizer = keras.optimizers.SGD(clipvalue = 1.0) # clip gradient to value from -1.0 to 1.0
  - optimizer = keras.optimizers.SGD(clipnorm = 1.0) # clip gradient and preserve its orientation
- Source: https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

Gradient checking for neuro network

- Compare numerical gradients with gradient from backpropagation (analytical)

$$\frac{J(\theta+\epsilon)-J(\theta)}{\epsilon} \qquad \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$$



Analytical derivative

$$\nabla_x f(x) = 2x \Rightarrow \nabla_x f(3) = 6$$

Two-sided numerical derivative $O(\varepsilon^2)$

$$\frac{(3+1e-2)^2-(3-1e-2)^2}{2*1e-2} = 5.999999999999872$$

Right-hand numerical derivative $O(\varepsilon)$

$$\frac{(3+1e-2)^2-3^2}{1e-2} = 6.009999999999849$$

https://medium.com/towards-data-science/coding-neural-network-gradient-checking-5222544ccc64

How to do gradient checking?

- Reshape $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L[}$ into a big vector θ
  - $J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]} = J(\theta)$
- Reshape $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L[}$ into a big vector of dθ
- For each i:
  - $d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots \theta_i - \varepsilon, \dots)}{2\varepsilon}$
- Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$, set $\varepsilon = 10^{-7}$

Gradient checking implementation notes

- Don’t use in training, only to debug.
- If the gradient check fails, check each components to identify bug.
- If you use regularization term, don’t forget it when do gradient checking.
  - $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|w^{[l]}\right\|_2^2$
- Gradient checking doesn’t work with dropout.
  - In the case of dropout, first use dropout=1.0 (turn off dropout), using gradient checking to check backpropagation is correct, then turn on dropout