

Deep Learning Specialization

by DeepLearning.AI

Course #2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization



[Course Site](#)

Made By: [Matias Borghi](#)

Table of Contents

Summary	3
Week 1 Practical aspects of Deep Learning	4
Setting up your Machine Learning Application	4
Train/dev/tests sets	4
Mismatched train/test distribution	4
Bias/Variance	5
Regularización de una red neuronal	7
Regularización	7
Aplicado a Regresión Logística	7
¿Qué forma toma la regularización para una red neuronal?	8
¿Por qué la regularización reduce overfitting?	8
Dropout regularización	9
Otros métodos de regularización	10
Data Augmentation	10
Configurando la optimización de problemas	11
Aproximación numérica de gradientes	12
Gradient Checking: técnica para verificar la implementación de backpropagation	13
Notas en gradient checking	14
Week 2: Algoritmos de optimización	15
Optimization algorithms	15
Mini-batch gradiente descendiente	15
Función de coste en función del número de iteraciones	16
¿Qué tamaño se elige entonces?	18
Exponentially weighted averages	18
Implementación computacional	19
Corrección del sesgo (bias)	19
Gradiente descendente con momentum	20
Otro algoritmo: RMSProp (RootMeanSquareProp)	21
Algoritmo de optimización de Adam (Adaptive moment estimation)	21
Decaimiento del ritmo de aprendizaje (α)	22
Definición de epoch y decaimiento del ritmo de aprendizaje	23
Óptimos locales en redes neuronales	24
Week 3: Hyperparameter tuning, Batch Normalization and Programming Frameworks	26
Hyperparameter tuning	26
Tuning process	26
Normalización Batch	28
Normalizando activaciones en una red	28
Implementar Batch Norm	28
Agregando Batch Norm a una red neuronal	29

Trabajando con mini-batch	30
Implementando gradiente descendiente	30
Resumiendo Batch norm	32
Multi-class classification	32
Regresión Softmax	32
Entrenar un clasificador Softmax	34
Introduction to programming frameworks	34
Deep Learning Frameworks	34
Tensor Flow	34

Summary

Week 1 Practical aspects of Deep Learning

Learning Objectives

- Give examples of how different types of initializations can lead to different results
- Examine the importance of initialization in complex neural networks
- Explain the difference between train/dev/test sets
- Diagnose the bias and variance issues in your model
- Assess the right time and place for using regularization methods such as dropout or L2 regularization
- Explain Vanishing and Exploding gradients and how to deal with them
- Use gradient checking to verify the accuracy of your backpropagation implementation

Week 2 Optimization algorithms

Learning Objectives

- Apply optimization methods such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Use random minibatches to accelerate convergence and improve optimization
- Describe the benefits of learning rate decay and apply it to your optimization

Week 3 Hyperparameter tuning, Batch Normalization and Programming Frameworks

Learning Objectives

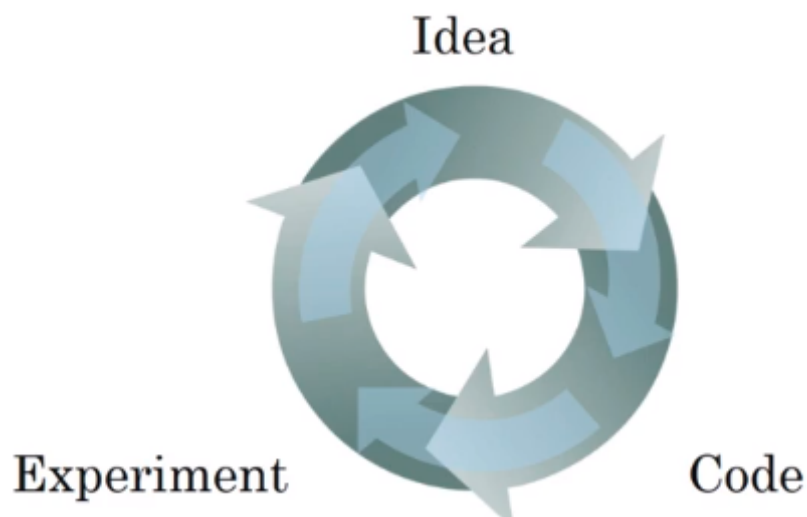
- Master the process of hyperparameter tuning

Week 1 Practical aspects of Deep Learning

Setting up your Machine Learning Application

Train/dev/tests sets

Generalmente estaba aceptado que los conjuntos de datos se separen de la siguiente forma 70/30% o 60/20/20. Aunque con el Big Data, del orden de 1M de ejemplos, se suelen modificar hasta del orden de 95.5/0.25/0.25 por ejemplo.



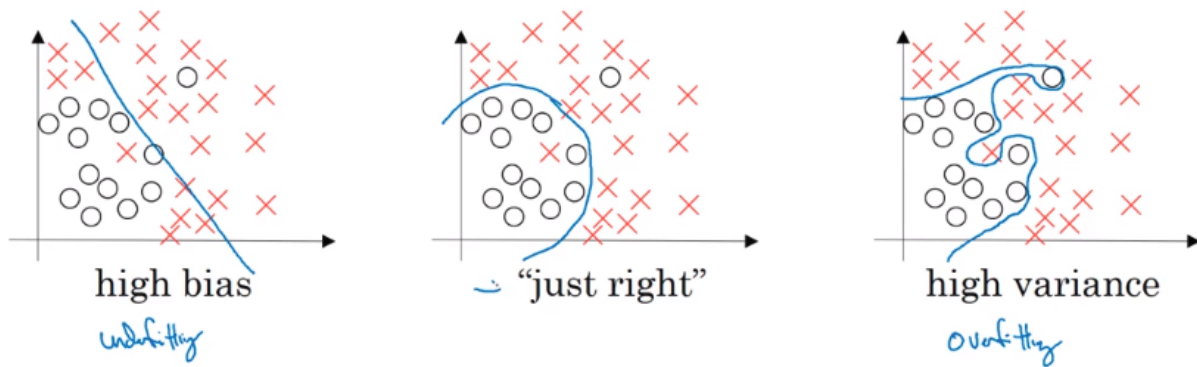
Mismatched train/test distribution

Generalmente se requiere que los conjuntos de datos tengan la misma distribución de probabilidad.



Bias/Variance

Ejemplo donde se tienen dos features.



En el caso de haber más de una hay diferentes métricas que se pueden utilizar.

Como ejemplo de clasificación de gatitos, una manera de determinar si se comete algún sesgo o varianza es mirando los errores en el train y dev set.

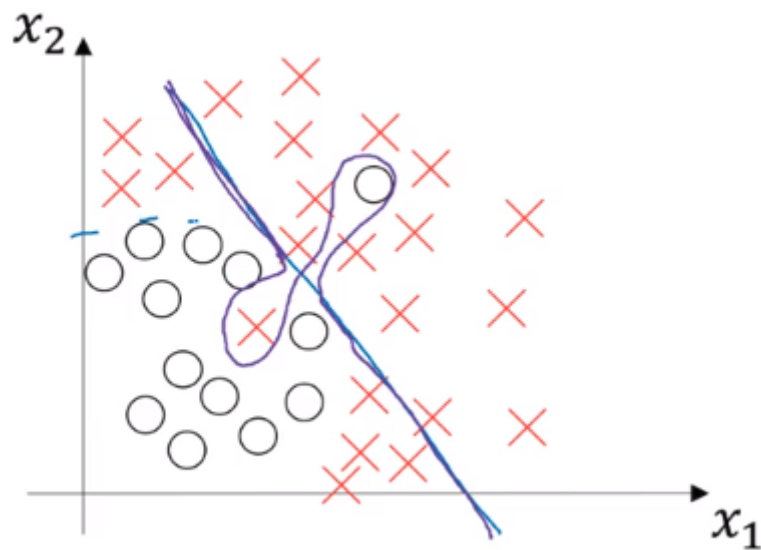
Cat classification



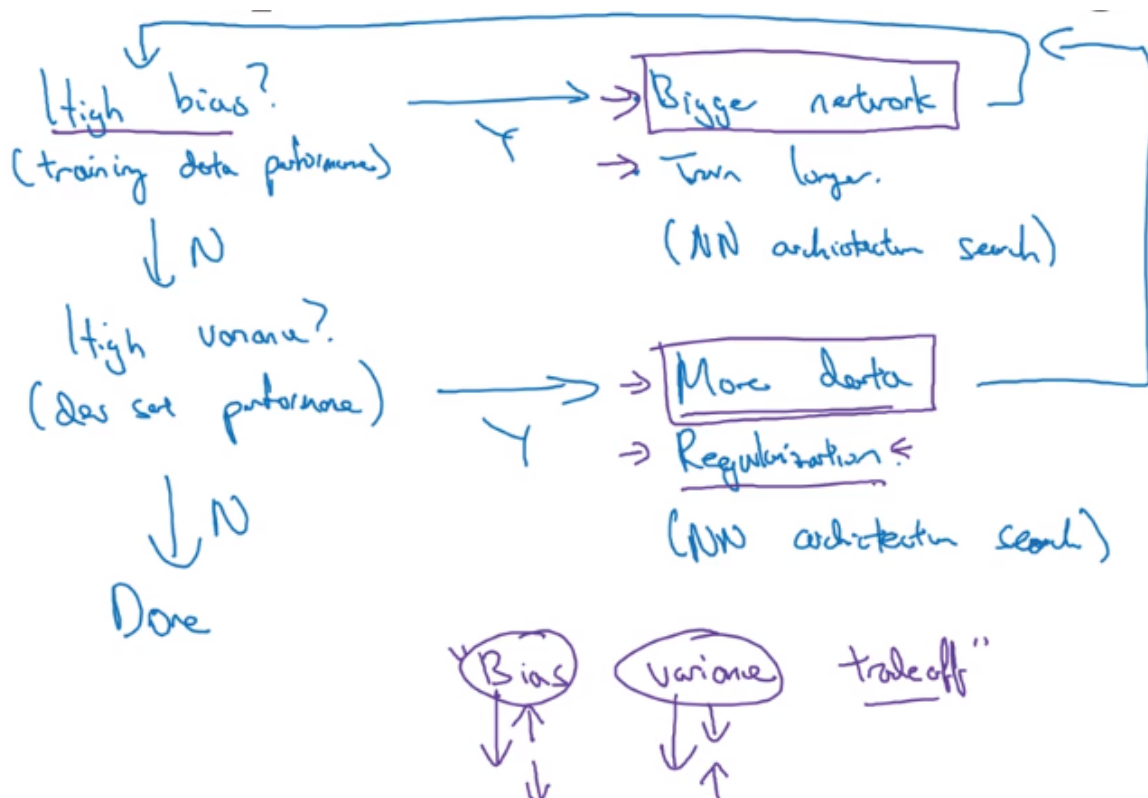
Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance
Human: ~0%				
Optimal (Bayes) error: ~0%				

Estas medidas suponen tanto que, como se dijo antes, las distribuciones de los datos provienen de la misma distribución de probabilidad, como que las imágenes no son borrosas para que se considere el error humano como 0%. La falla del primer caso se verá más adelante. En el caso de tener imágenes borrosas por ejemplo, puede provocar que por ejemplo el segundo caso de errores de 15% y 16% en el train y dev set error respectivamente, sean considerados relativamente aceptables.

El tercer caso de alto sesgo y varianza parece ser contradictorio. Veamos un ejemplo:



Receta básica para eliminar sesgo y varianza en Machine Learning



Primero podríamos ver si el algoritmo tiene un sesgo alto sobre el training set. En caso afirmativo se podría agrandar la red neuronal con más unidades ocultas, entrenarlo más tiempo, ya sea corriendo gradiente descendente un mayor tiempo o (a veces funciona a

veces no) tratar de encontrar una mejor arquitectura para la red neuronal. Luego de probar esto se podría probar nuevamente si el problema de alta sesgo fue solucionado.

Si el problema de alto sesgo fue solucionado se podría probar si hay alta varianza sobre el dev set, probando con más datos o aplicando regularización y también, como antes, probar una nueva arquitectura para la red neuronal.

Cuando ambas son negativas se podría afirmar que se terminó.

Regularización de una red neuronal

Regularización

Este método ayuda a prevenir el overfitting y errores de la red neuronal.

Aplicado a Regresión Logística

$$\begin{aligned} & \min_{w,b} J(w,b) \quad \underline{w \in \mathbb{R}^{n_x}}, b \in \mathbb{R} \\ J(w,b) &= \frac{1}{m} \sum_{i=1}^m \ell(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2 + \underbrace{\frac{\lambda}{2m} b^2}_{\text{omit}} \\ \text{L}_2 \text{ regularization} \quad \underline{\|w\|_2^2} &= \sum_{j=1}^{n_x} w_j^2 = w^T w \end{aligned}$$

Esto se llama regularización L_2 , donde lambda es el parámetro de regularización. El último término se omite dado que w suele poseer muchos más parámetros que b ($n_x \gg 1$).

A su vez, existe lo que se llama regularización L_1 , pero no es utilizado tan comúnmente.

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

¿Qué forma toma la regularización para una red neuronal?

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \underbrace{\frac{1}{m} \sum_{i=1}^m \ell(y^{(i)}, y^{(i)})}_{\text{loss}} + \underbrace{\frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|_F^2}_{\text{regularization}}$$

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

"Frobenius norm"

$\| \cdot \|_2^2$ $\| \cdot \|_F^2$

$w: \begin{pmatrix} n^{[l-1]} & n^{[l-1]} \end{pmatrix}$

Podemos ver que la expresión es similar pero ahora se toma una norma de Frobenius sobre los parámetros w .

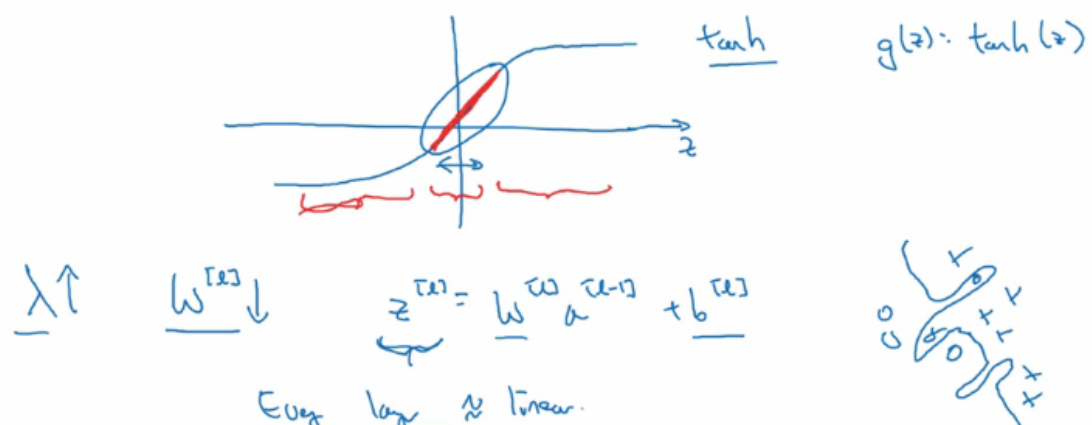
Por otra parte, la backpropagation es de la forma

$$dw^{[l]} = \frac{\partial J}{\partial w^{[l]}} = \text{(from backprop)} + \frac{\lambda}{m} w^{[l]}$$

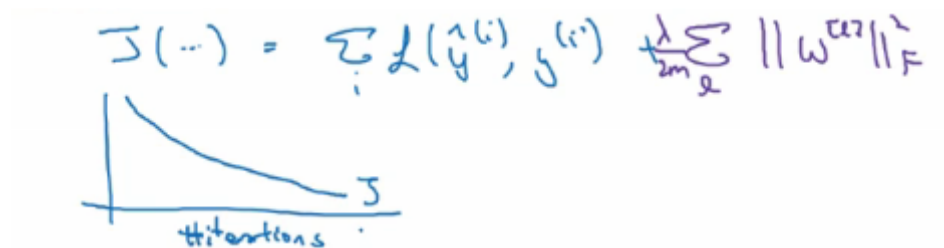
$$\rightarrow w^{[l]} := w^{[l]} - \alpha dw^{[l]}$$

¿Por qué la regularización reduce overfitting?

Tomando como ejemplo la función de activación $\tanh(z)$, para un valor grande de λ , w es pequeño y por consiguiente z también. De aquí que \tanh se valúe prácticamente en la región lineal y se prevenga el overfitting. La red neuronal será prácticamente una red lineal.

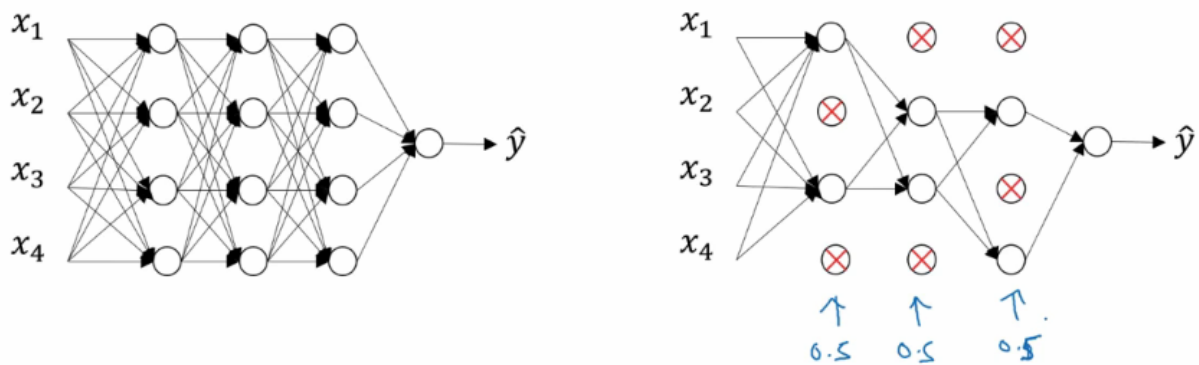


Por otra parte, para probar el correcto funcionamiento de la implementación se puede probar graficando la función de coste en función del número de iteraciones y ver que esta es una función monotónicamente decreciente.



Dropout regularización

Esta es una nueva técnica de regularización. La misma consiste en eliminar nodos con una dada probabilidad. El nuevo cálculo se realiza sin los nodos eliminados y se previene la regularización.



Pero, ¿Cómo se implementa esta regularización? Una técnica se conoce como inverted dropout.

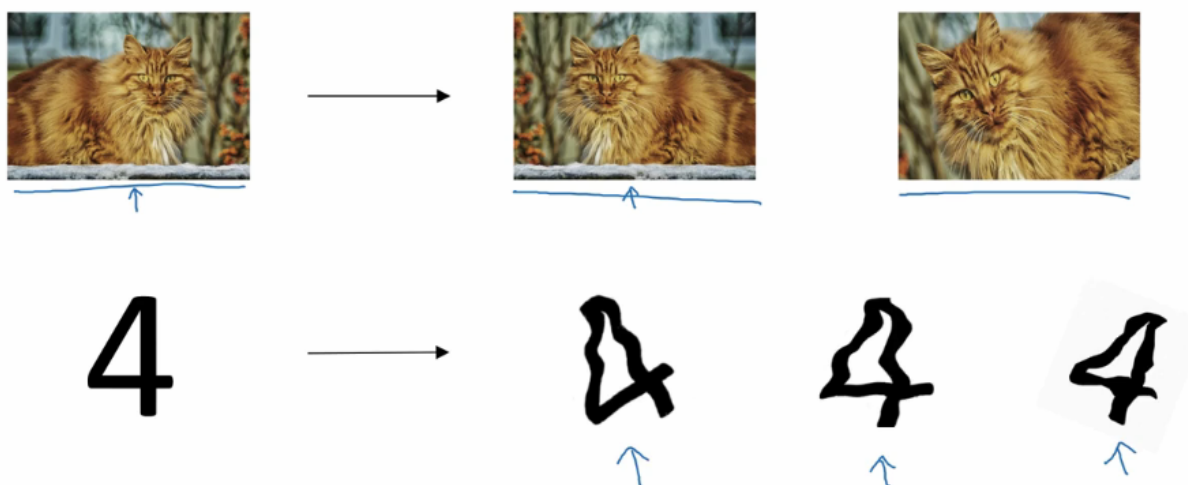
Illustrate with layer $l=3$. $\text{keep-prob} = \frac{0.8}{x}$ 0.2
 $\Rightarrow \underline{d3} = \text{np.random.rand}(a3.\text{shape}[0], a3.\text{shape}[1]) < \text{keep-prob}$
 $\underline{a3} = \text{np.multiply}(a3, d3)$ # $a3 \neq d3$.
 $\Rightarrow \boxed{a3 /= \text{keep-prob}}$ \leftarrow
 50 units. \leadsto 10 units shut off

$$z^{[4]} = w^{[4]} \cdot \underbrace{a^{[3]}}_{\substack{\uparrow \\ \text{reduced by } 20\% \\ /= 0.8}} + b^{[4]}$$
Test

Otros métodos de regularización

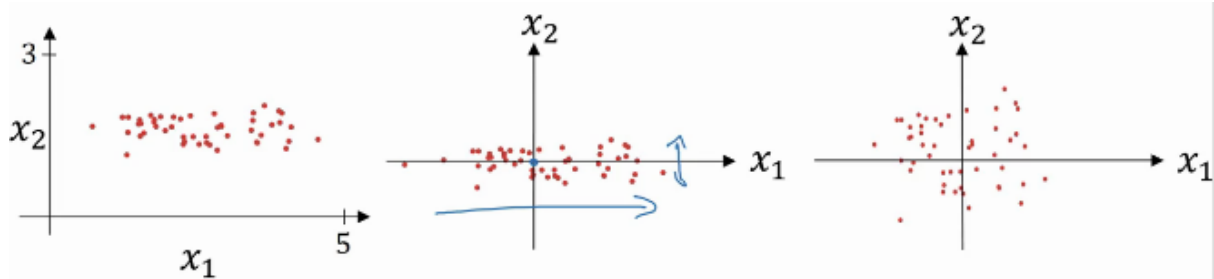
Data Augmentation

En el caso de estar overfitteando. Se pueden agregar más ejemplos. Pero puede ser muy caro. Para ello se pueden espejar horizontalmente algunas fotos o modificarlas sutilmente, como se muestra en la figura inferior.

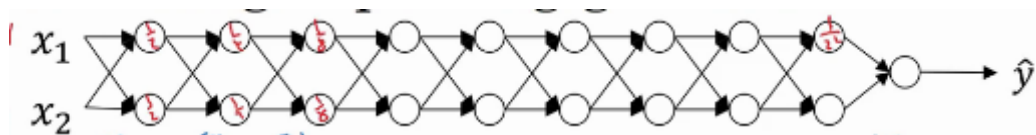


Configurando la optimización de problemas

- Normalización de resultados: evitar w grandes o pequeños. Escalar el training y test set mediante de la misma manera.



- Vanishing/ exploding gradients: cuando se entrenan redes neuronales muy profundas (muchos hidden layers), las activaciones pueden ser, o muy chicas, o muy grandes.



Handwritten notes illustrating the vanishing/exploding gradient problem:

Assuming $b=0$ and a linear activation function $g(z) = z$:

$$\hat{y} = W^{[L]} \cdot \dots \cdot W^{[2]} \cdot x$$

Where $W^{[L]} > I$ and $W^{[2]} < I$ (e.g., $\begin{bmatrix} 0.9 & 0.9 \end{bmatrix}$).

For a specific example, $W^{[2]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 0.5 \end{bmatrix}$.

The activation at layer l is $a^{[l]} = g(z^{[l]}) = g(W^{[l]} a^{[l-1]})$.

For $W^{[2]} > I$, the activation grows exponentially: $1.5^{L-1} \times 0.5^{L-1} \times$.

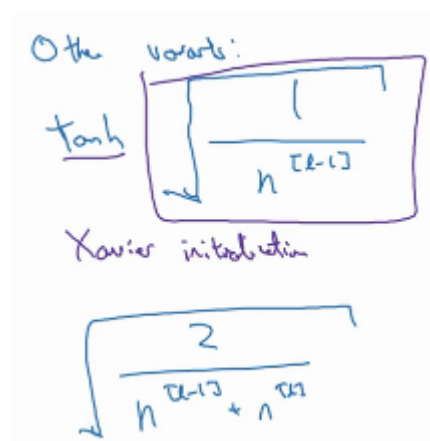
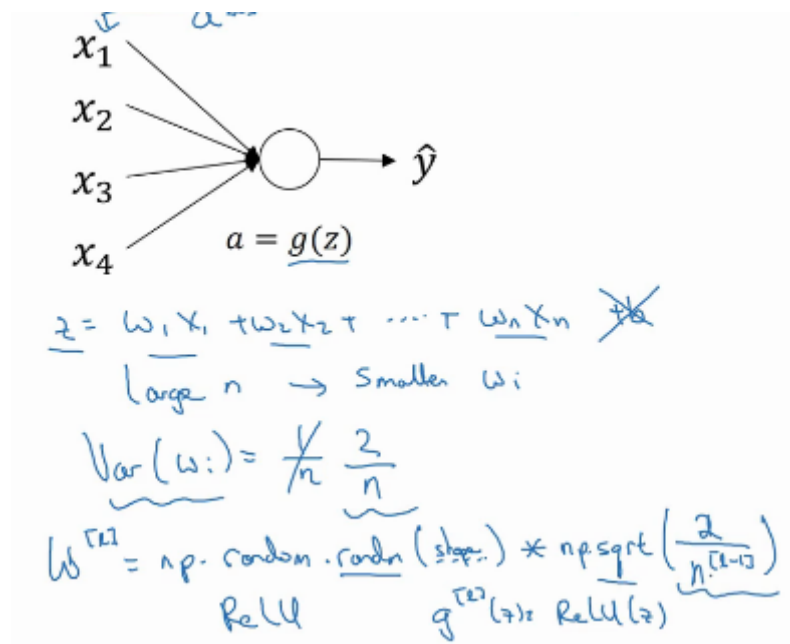
For $W^{[2]} < I$, the activation shrinks exponentially: $0.5^{L-1} \times$.

Suponiendo que $b=0$ y que la función de activación es lineal, se puede demostrar que w puede ser o muy grande o muy pequeño. Esto se puede resolver inicializando los pesos.

Comencemos con un ejemplo para una única neurona, luego se verá el caso de una red neuronal más compleja.

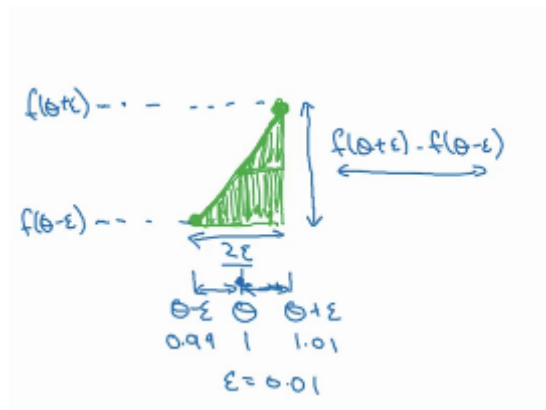
Dado que z es la suma de términos de la forma $w \cdot x$, si n aumenta se quiere que w disminuya. ¿De qué manera? Una buena opción es que la varianza de los w vaya como $1/n$. Para el caso de las funciones RELU, $2/n$ funciona mejor. De esta manera se llega a una expresión para w que no es muy grande o muy pequeña si los features aumentan considerablemente.

Otras variantes para distintas funciones de activación como tanh: Inicialización de Xavier.



Aproximación numérica de gradientes

Calculando derivadas a ambos lados.



A diferencia de la derivada a un lado, esta posee un error más pequeño, pero se tarda más en calcular su valor.

$$f'(\theta) = \lim_{\epsilon \rightarrow 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon} \quad \begin{array}{l} O(\epsilon^2) \\ 0.01 \\ 0.0001 \end{array} \quad \left| \quad \frac{f(\theta+\epsilon) - f(\theta)}{\epsilon} \quad \begin{array}{l} \text{error: } O(\epsilon) \\ 0.01 \end{array}$$

Gradient Checking: técnica para verificar la implementación de backpropagation

Primero conviene realizar dos pasos previos y quedarnos con un vector $d\theta$. Y preguntarse, ¿es este vector el gradiente de $J(\theta)$?

Take $\boxed{W^{[1]}}$, $\boxed{b^{[1]}}$, ..., $\boxed{W^{[L]}}$, $\boxed{b^{[L]}}$ and reshape into a big vector $\underline{\theta}$.

concatenate

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

Take $\boxed{dW^{[1]}}$, $\boxed{db^{[1]}}$, ..., $\boxed{dW^{[L]}}$, $\boxed{db^{[L]}}$ and reshape into a big vector $\underline{d\theta}$.

concatenate

Is $\underline{d\theta}$ the gradient of $J(\theta)$?

for each i :

$$\rightarrow \underline{d\theta}_{\text{approx}}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i^{\downarrow} + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i^{\downarrow} - \epsilon, \dots)}{2\epsilon}$$

$$\approx \underline{d\theta}[i] = \frac{\partial J}{\partial \theta_i} \quad \left| \quad d\theta_{\text{approx}} \stackrel{?}{\approx} d\theta$$

¿Cómo hacemos para chequear que estos dos vectores sean los mismos? Se puede calcular la distancia euclídea entre ellos.

Handwritten notes showing a formula for checking the L2 distance between two vectors and a comparison with epsilon.

$$\text{Check } \frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2} \approx \begin{matrix} 10^{-7} & - \text{great!} \\ 10^{-5} & \\ 10^{-3} & - \text{worry.} \end{matrix}$$

$\epsilon = 10^{-7}$

Si ϵ es del orden de 10^{-7} y la distancia euclídea la implementación probablemente es la correcta. Pero si la distancia es más grande en 2 órdenes de magnitud por lo menos, entonces conviene mirar si se cometió algún error (idealmente en las componentes i individuales, ej. si el error esta en dw o en db).

Notas en gradient checking

- No usar en training - solo para debug.
- si el algoritmo falla, mirar las componentes para identificar el bug.
- recordar regularización.
- no funciona con dropout: para implementar ambas se podría “apagar” dropout ($\text{keep_prob}=1.0$) chequear gradient checking y luego prenderlo.
- Correr bajo inicialización aleatoria; luego tal vez después del entrenamiento: suele suceder que no hay problema con $w, b \sim 0$, pero falla para errores más grandes.

Week 2: Algoritmos de optimización

Optimization algorithms

Dado que Deep Learning funciona mejor con Big Data, esto ralentiza el cómputo de problemas. En esta sección se estudiarán diferentes técnicas para acelerar este proceso mediante algoritmos de optimización.

Mini-batch gradiente descendiente

Supongamos que se tiene la siguiente configuración. Para configurar gradiente descendiente se debería procesar todos los features y realizar un paso del gradiente. Luego procesar nuevamente para otro paso y así sucesivamente. En el caso de que $m = 5M$, esto se vuelve muy lento.

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & \dots & x^{(m)} \end{bmatrix}$$

(n_x, m)

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$

Una solución es usar mini-batch, el cual consiste en separar el training set en por ejemplo dos trainings set.

Para el ejemplo anterior, se pueden separar conjuntos de sets de 1000 elementos. En total se tendrán 5000 subconjuntos, introducidos con la notación entre corchetes.

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(1000)} & | & x^{(1001)} & \dots & x^{(2000)} & | & \dots & | & \dots & x^{(m)} \end{bmatrix}$$

(n_x, m) $X^{f13} (n_x, 1000)$ $X^{f23} (n_x, 1000)$ $X^{f5,0003} (n_x, 1000)$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & | & y^{(1001)} & \dots & y^{(2000)} & | & \dots & | & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$ $Y^{f13} (1, 1000)$ $Y^{f23} (1, 1000)$ $Y^{f5,0003} (1, 1000)$

What if $m = 5,000,000$?
 5,000 mini-batches of 1,000 each
 Mini-batch t : $\underline{X^{t+1}, Y^{t+1}}$.

De esta manera la implementación vectorial del algoritmo mini-batch es la siguiente

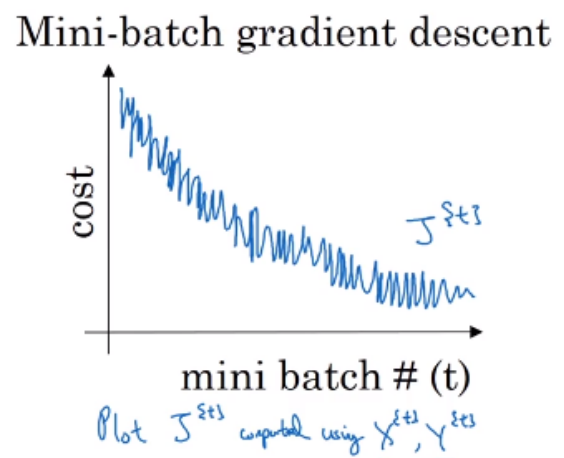
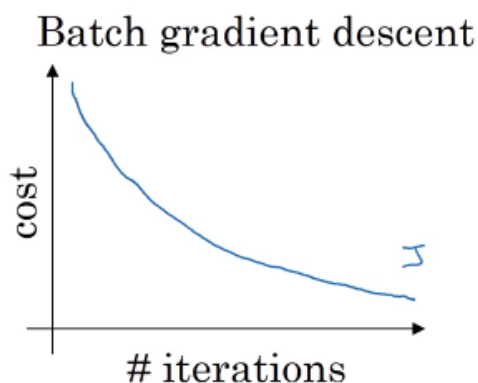
```

for  $t = 1, \dots, 5000$  {
  Forward prop on  $X^{t+1}$ .
   $Z^{(t)} = W^{(t)} X^{t+1} + b^{(t)}$ 
   $A^{(t)} = g^{(t)}(Z^{(t)})$ 
  ...
   $A^{(t)} = g^{(t)}(Z^{(t)})$ 
  Compute cost  $J^{t+1} = \frac{1}{1000} \sum_{i=1}^{\frac{1}{2}} \ell(y^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{i=1}^{\frac{1}{2}} \|W^{(t)}\|_F^2$ .
  Backprop to compute gradients w.r.t  $J^{t+1}$  (using  $X^{t+1}, Y^{t+1}$ )
   $W^{(t+1)} = W^{(t)} - \alpha \Delta W^{(t)}, b^{(t+1)} = b^{(t)} - \alpha \Delta b^{(t)}$ 
}
  
```

} Vectorial implementation (1000 examples)
 ↓ for X^{t+1}, Y^{t+1} .

A el loop exterior habría que agregarle otro loop del número de iteraciones.

Función de coste en función del número de iteraciones



Uno podría preguntarse, ¿qué tamaño del mini-batch elegir y cómo?

- Si el tamaño del mini-batch = m : batch es el de gradiente descendiente.
- si tamaño mini-batch = 1: gradiente descendiente estocástico. Cada ejemplo es un mini-batch.
- En la practica el tamaño del mini-batch va a estar entre 1 y m .

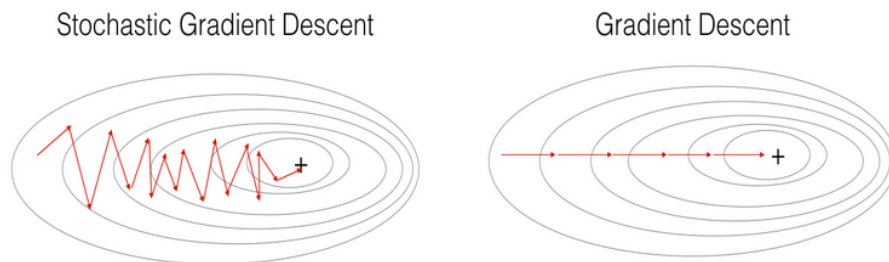
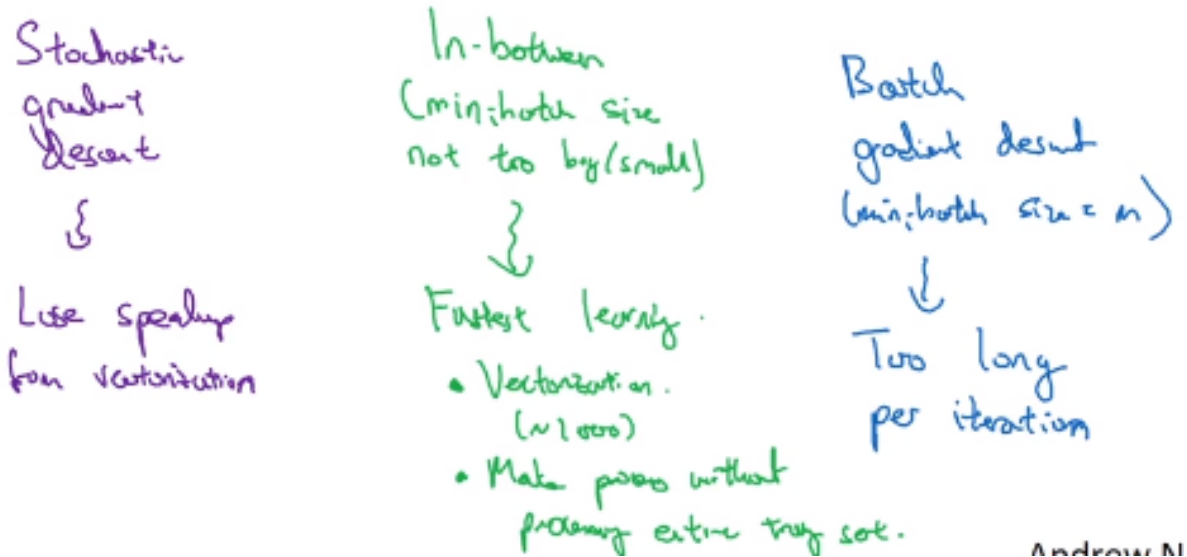


Figure 1: SGD vs GD

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

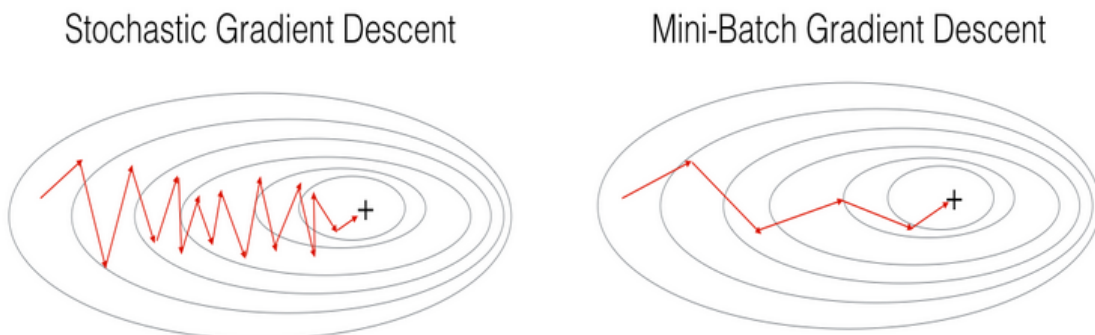


Figure 2: SGD vs Mini-Batch GD

"+" denotes a minimum of the cost. Using mini-batches in your optimization algorithm often leads to faster optimization.

¿Qué tamaño se elige entonces?

En la práctica este es otro hiper parámetro el cual se debe encontrar.

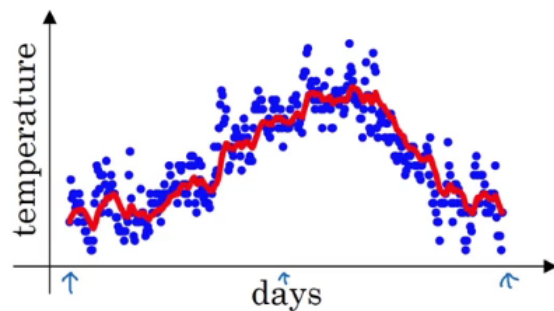
Si el training set es pequeño ($m \leq 2000$) conviene utilizar batch gradiente descendiente. Tamaños usuales para el mini-batch suelen ser potencias de 2: $64(2^6)$, $128(2^7)$, $256(2^8)$, $512(2^9)$. Conviene chequear que el tamaño del mini-batch puede almacenarse en la memoria del CPU/GPU.

Exponentially weighted averages

Para introducir algoritmos de optimización más avanzados, primero hay que explicar el concepto de promedios pesados exponenciales.

Para ello comencemos con un ejemplo de las temperaturas anuales de Londres.

$$\begin{aligned}\theta_1 &= 40^\circ\text{F} \quad 4^\circ\text{C} \leftarrow \\ \theta_2 &= 49^\circ\text{F} \quad 9^\circ\text{C} \\ \theta_3 &= 45^\circ\text{F} \quad \vdots \\ &\vdots \\ \theta_{180} &= 60^\circ\text{F} \quad 15^\circ\text{C} \\ \theta_{181} &= 56^\circ\text{F} \quad \vdots \\ &\vdots\end{aligned}$$



$$\begin{aligned}V_0 &= 0 \\ V_1 &= 0.9 V_0 + 0.1 \theta_1 \\ V_2 &= 0.9 V_1 + 0.1 \theta_2 \\ V_3 &= 0.9 V_2 + 0.1 \theta_3 \\ &\vdots \\ V_t &= 0.9 V_{t-1} + 0.1 \theta_t\end{aligned}$$

El promedio ponderado se calcula para el primer día $V_0 = 0$. Para el segundo, se supone que hay un peso del 90% respecto del valor del día anterior V_0 mas una proporción restante del 10% a determinar por θ_1 .

$$V_t = \beta V_{t-1} + (1-\beta) \Theta_t$$

$\beta = 0.9$: ≈ 10 days' temper.

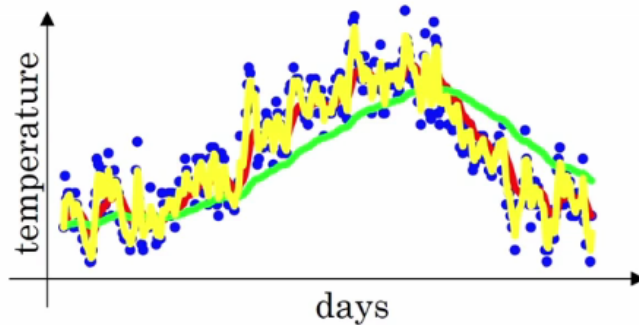
$\beta = 0.98$: ≈ 50 days

$\beta = 0.5$: ≈ 2 days

V_t is approximately
average over

$\rightarrow \approx \frac{1}{1-\beta}$ days' temperature.

$$\frac{1}{1-0.98} = 50$$



De manera general la ecuación que relaciona la temperatura para un día particular se expresa en la figura superior. Donde los parámetros a determinar son β y θ . Se puede notar que para β más grandes el valor de la temperatura del día anterior es muy importante y los cambios por nuevos puntos no son tan bruscos.

La línea roja equivale a $\beta = 0.9$. La verde a $\beta = 0.89$ y $\beta = 0.5$ a la línea amarilla.

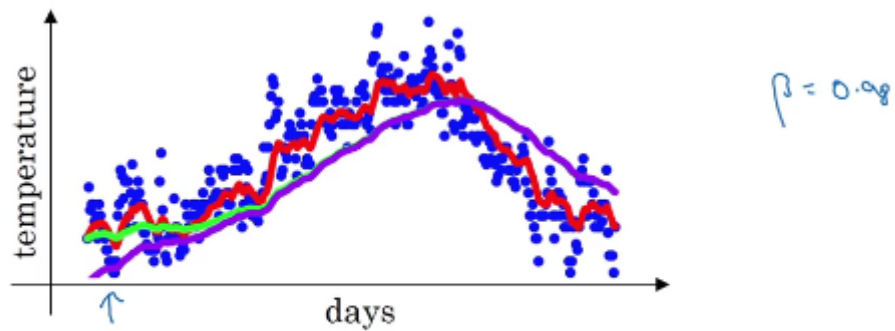
Implementación computacional

```

→  $V_0 = 0$ 
Repeat {
  Get next  $\Theta_t$ 
   $V_t := \beta V_0 + (1-\beta) \Theta_t$  ←
}

```

Corrección del sesgo (bias)



$$\rightarrow v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

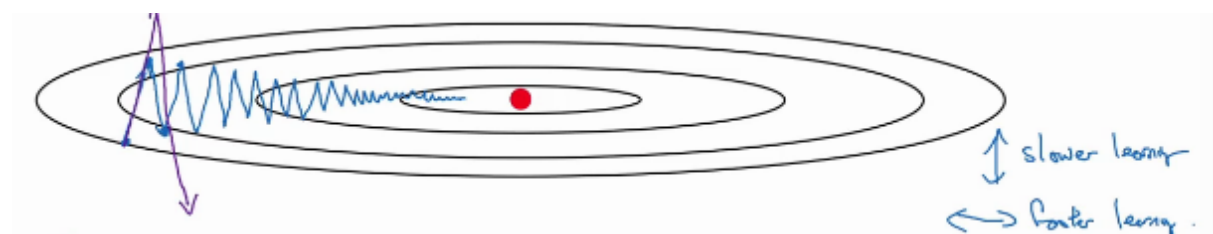
En el caso de $\beta = 0.98$, se debería obtener la línea verde, pero usando la ecuación superior se obtiene la púrpura. La corrección del sesgo permite calcular el promedio de manera más precisa. Simplemente consiste en ver que a tiempos pequeños hay un problema debido a que $V_0 = 0$, generalmente no es una buena implementación. Mientras que a tiempos

mayores esto se soluciona. De esta manera se aplica una corrección $\frac{v_t}{1 - \beta^t}$. La misma actúa cuando los tiempos son pequeños (ej., $1 - (0.98)^2 = 0.0396$), mientras que es prácticamente nula cuando t es mayor (ej., $1 - (0.98)^{100} = 1 - 0 = 1$).

Utilicemos ahora lo que se vio previamente como corrección del sesgo para generar mejores algoritmos de optimización.

Gradiente descendente con momentum

Generalmente funciona más rápido que el algoritmo de gradiente descendente estándar. En una oración, la idea básica es calcular el promedio pesado exponencial de los gradientes para luego usar los gradientes para actualizar los pesos.



On iteration t :

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1 - \beta) dW$$

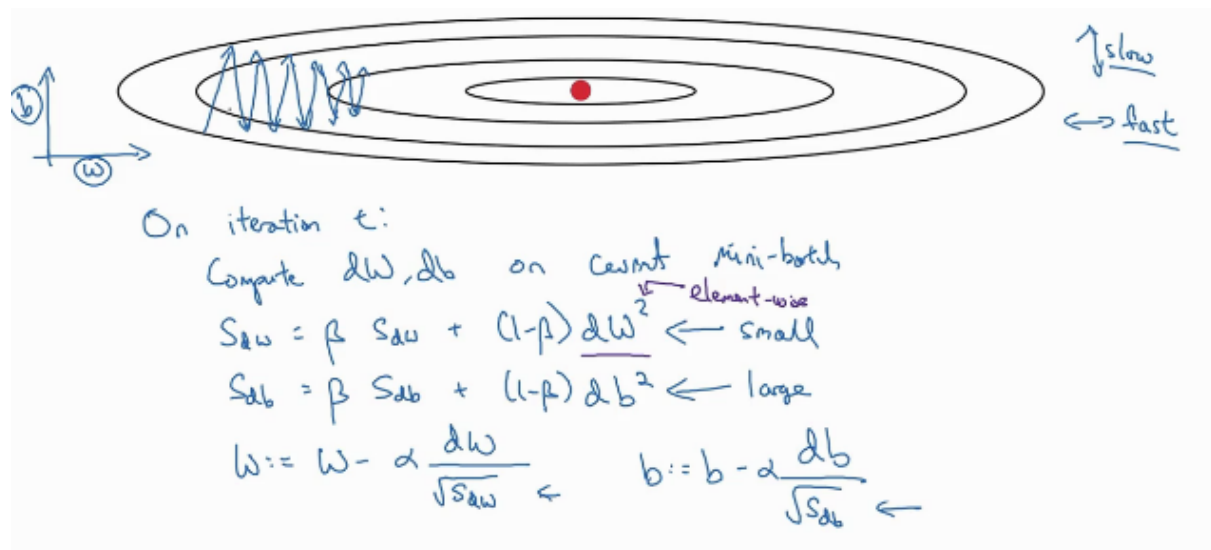
$$v_{db} = \beta v_{db} + (1 - \beta) db$$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β $\beta = 0.9$

Básicamente este algoritmo lo que hace es suavizar el movimiento lateral que es el que se quiere que converja más rápidamente. El algoritmo toma un camino más directo.

Otro algoritmo: RMSProp (RootMeanSquareProp)



Ahora combinaremos los algoritmos de RMSProp con el Gradiente descendente momentum.

Algoritmo de optimización de Adam (Adaptive moment estimation)

Este algoritmo a diferencia de los anteriores, funciona correctamente en un amplio rango de arquitecturas.

$$V_{dw}=0, S_{dw}=0. \quad V_{db}=0, S_{db}=0$$

On iteration t :

Compute dw, db using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db \quad \leftarrow \text{"moment"} \quad \beta_1$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2 \quad \leftarrow \text{"RMSprop"} \quad \beta_2$$

$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t), \quad V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$$

$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t), \quad S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}, \quad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$$

Elección de hiper parámetros: generalmente con la optimización de Adam se definen valores para β y ϵ y se trata de estimar el mejor valor de α entre un rango de valores.

α : needs to be tune

β_1 : 0.9 (dw)

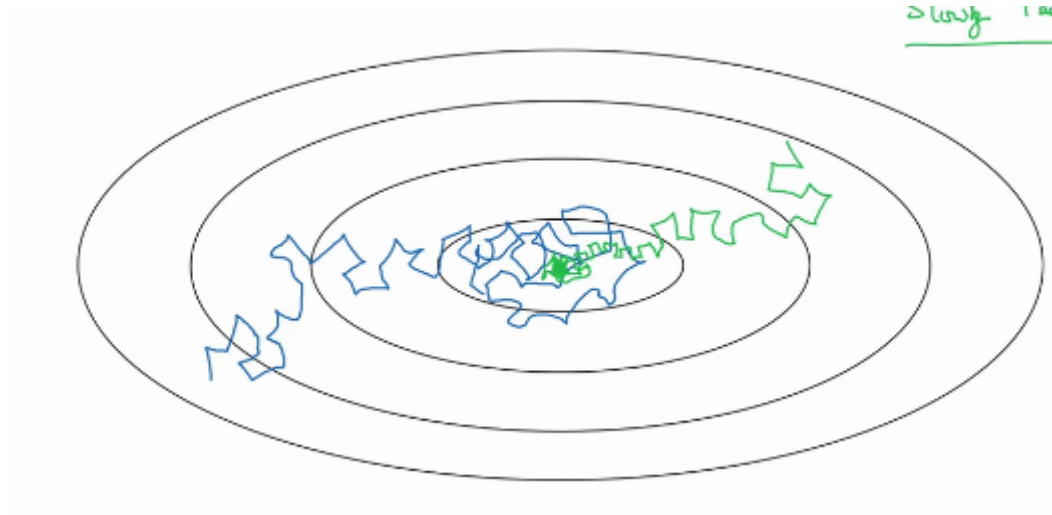
β_2 : 0.999 (dw^2)

ϵ : 10^{-8}

Decaimiento del ritmo de aprendizaje (α)

La idea detrás de este método es que, dado un valor de α fijo el algoritmo podría quedarse dando saltos alrededor del punto de equilibrio no tan cerca de él (línea azul).

El decaimiento propuesto es del tipo exponencial, inicialmente siendo grande dado que se está lejos de la solución óptima, y a medida que se va acercando α va siendo cada vez más pequeño (línea roja).



Definición de epoch y decaimiento del ritmo de aprendizaje

Learning rate decay

1 epoch = 1 pass through data.

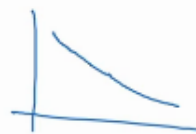
$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
.	.

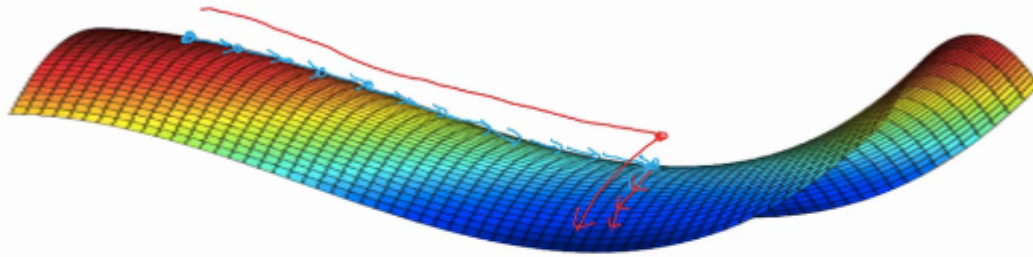


$$\alpha_0 = 0.2$$

$$\text{decay-rate} = 1$$



La definición del decaimiento es arbitraria, se podría haber elegido alguna otra como el decaimiento tipo escalera discreto.



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Week 3: Hyperparameter tuning, Batch Normalization and Programming Frameworks

Hyperparameter tuning

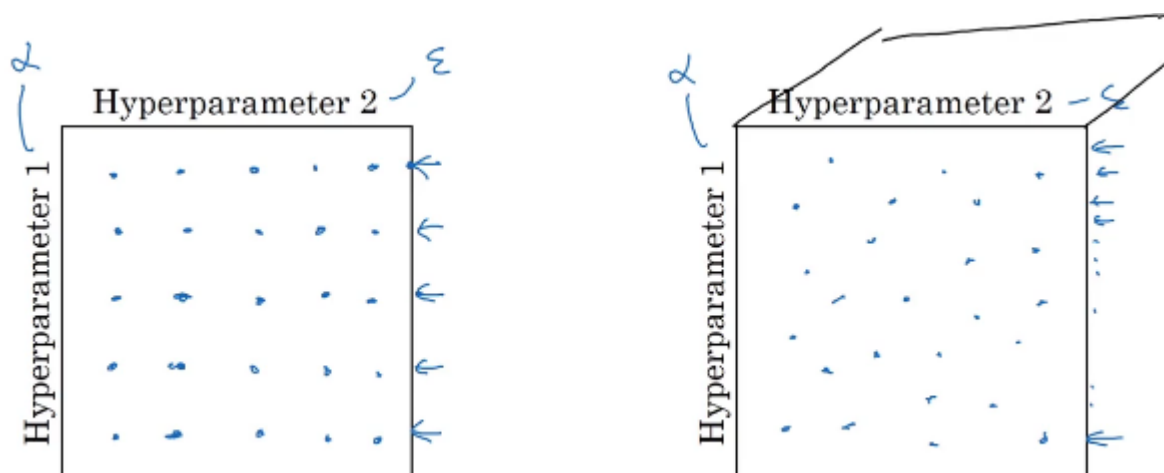
Tuning process

¿Cómo elegimos adecuadamente al conjunto de hiper parámetros: α , β , (β_1 , β_2 , ϵ) para Adam, número de capas, número de unidades ocultas, decaimiento de la tasa de aprendizaje, tamaño del mini-batch?

No todos son igual de importantes. α siendo el más importante. β , el número de unidades ocultas o el tamaño del mini-batch las siguientes:

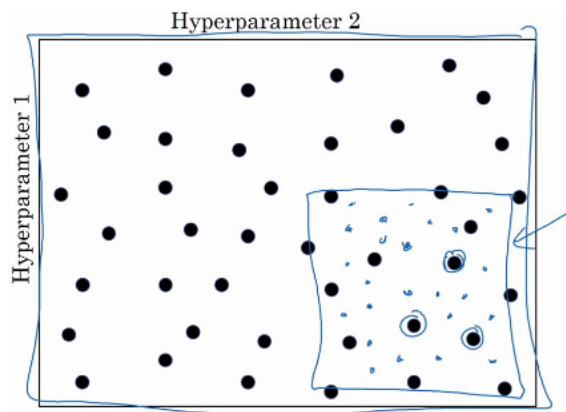
1. Elegir valores aleatorios de hiper parámetros: no usar una grilla!!

Cuando el número de hiper parámetros es pequeño la grilla puede funcionar bien, sino no.



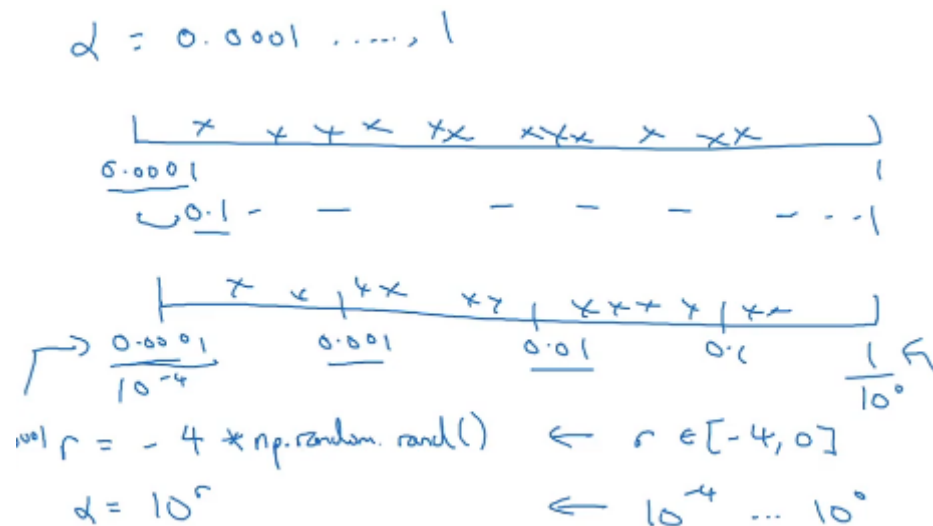
- 2) Coarse to fine

Enfocarse más en una región particular del espacio de parámetros.



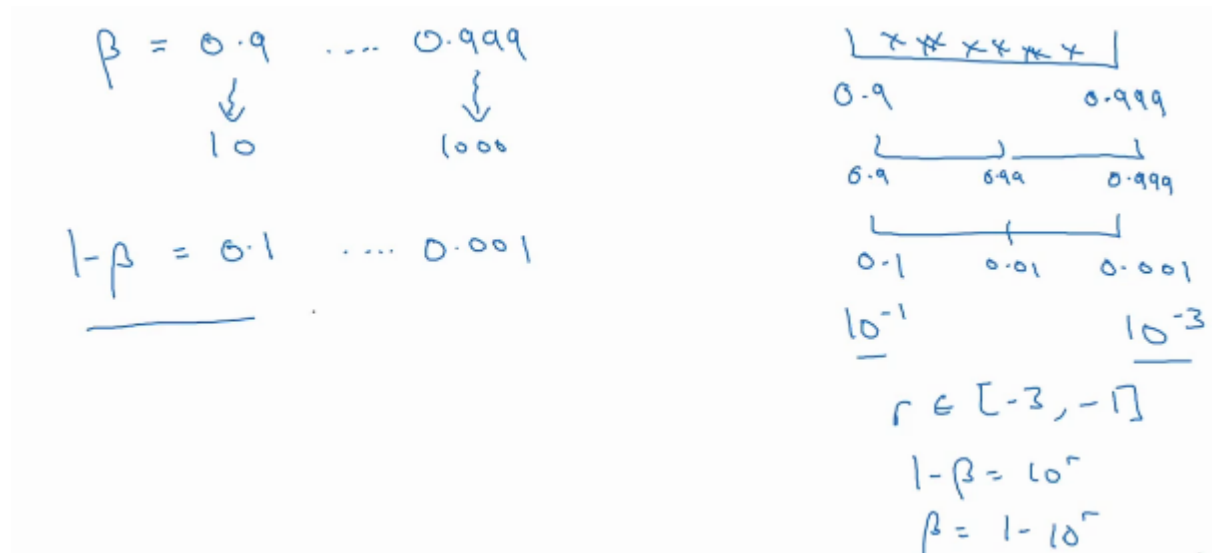
Usar la escala adecuada para elegir hiper parámetros

Por ejemplo, para alfa entre $10e^{-4}$ y 1 conviene elegir en escala logarítmica.



Hiper parámetros para promedios pesados exponencialmente

Por ejemplo, β yendo desde 0.9 a 0.999. Conviene hacer algo similar a lo visto anteriormente.



Normalización Batch

Este algoritmo hace la búsqueda de hiper parámetros mucho más simple, haciendo la red neuronal más robusta. La elección de hiper parámetros es hecha en un rango mucho más amplio provocando que se pueda entrenar más fácilmente redes neuronales más profundas.

Normalizando activaciones en una red

Ya se ha visto previamente como realizar una normalización de los features de entrada y el efecto que provoca un cambio en la regresión. Pero, ¿cómo varía un modelo más profundo?

¿Se podrán normalizar las capas ocultas de manera que w y b se calculen más rápidamente?

Implementar Batch Norm

Básicamente lo que hace este algoritmo es normalizar los z en capas ocultas a un valor tal que z_i moño sea igual a z_i .

Given some intermediate values in NN $z^{(1)}, \dots, z^{(n)}$

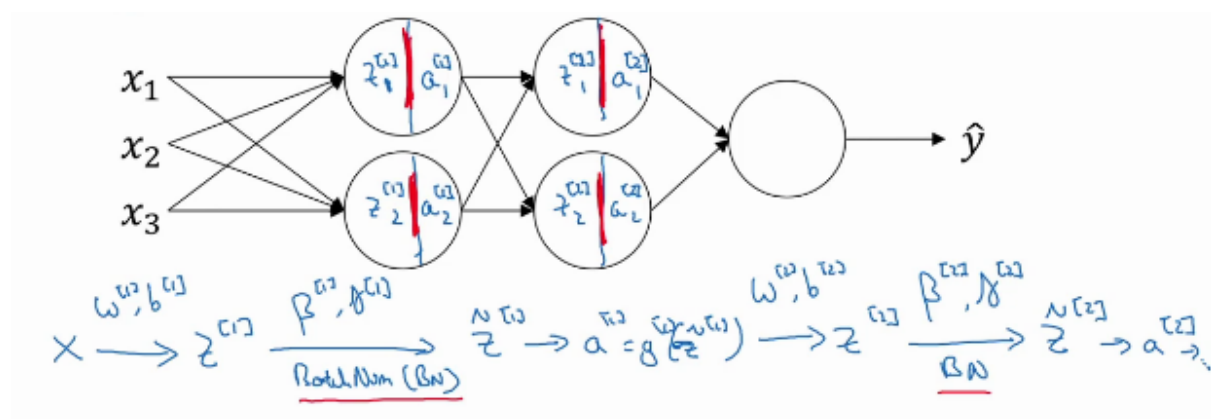
$$\left[\begin{array}{l} \mu = \frac{1}{n} \sum_i z^{(i)} \\ \sigma^2 = \frac{1}{n} \sum_i (z^{(i)} - \mu)^2 \\ z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta \end{array} \right.$$

If $\gamma = \sqrt{\sigma^2 + \epsilon}$ $\leftarrow z^{(i)}$
 $\beta = \mu$ \leftarrow
 then $\sum_i \tilde{z}^{(i)} = z^{(i)}$

learnable parameters of model.

Agregando Batch Norm a una red neuronal

Primer se comienza calculando $z^{[1]}$ dadas $w^{[1]}$ y $b^{[1]}$, luego se normaliza usando Batch Norm (BN) dados los parámetros $\beta^{[1]}$ y $\gamma^{[1]}$. Obtenida $\tilde{z}^{[1]}$ se calcula $a^{[1]}$ evaluando \tilde{z} con la función de activación. Esto se realiza para todas las capas ocultas.



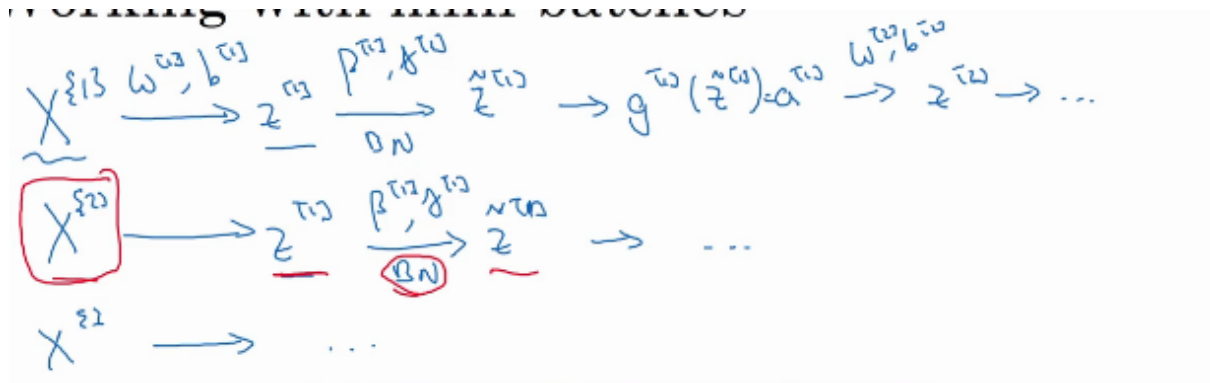
Los parámetros a ajustar serán el doble.

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$
 $\rightarrow \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$

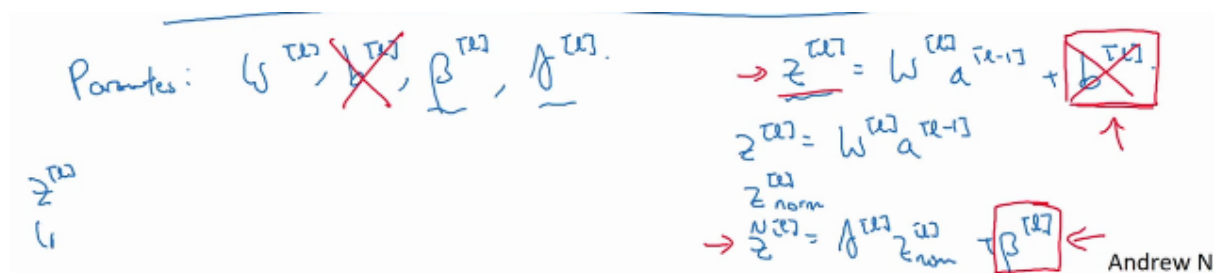
En TensorFlow esto se implementaría con `tf.nn.batch_normalization`.

Trabajando con mini-batch

El proceso es similar al anterior pero trabajando por separado con cada mini-batch.

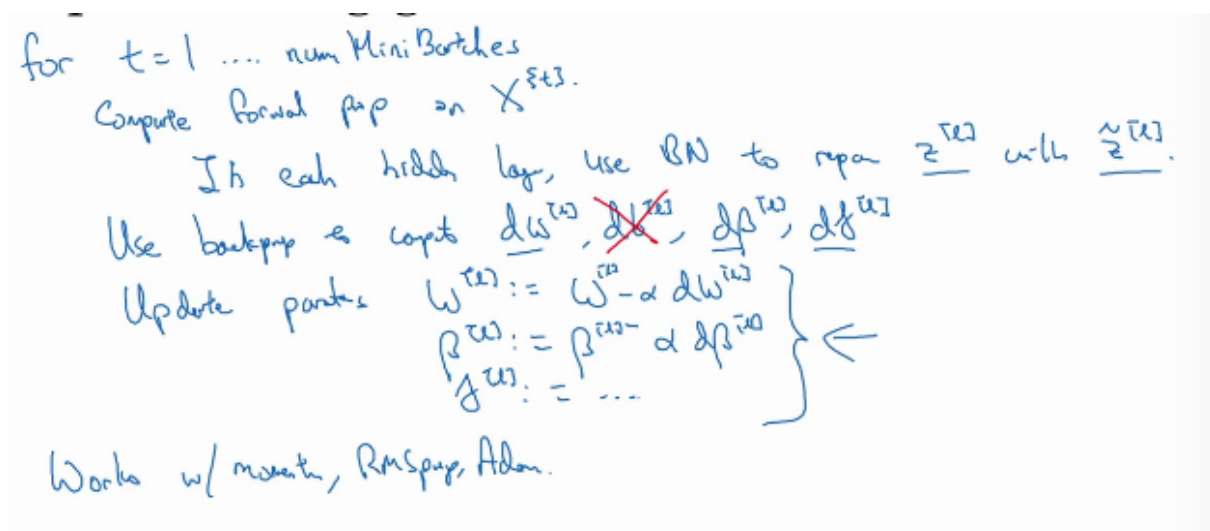


Por otra parte, se puede decir que en realidad el parámetro b es innecesario calcularlo.



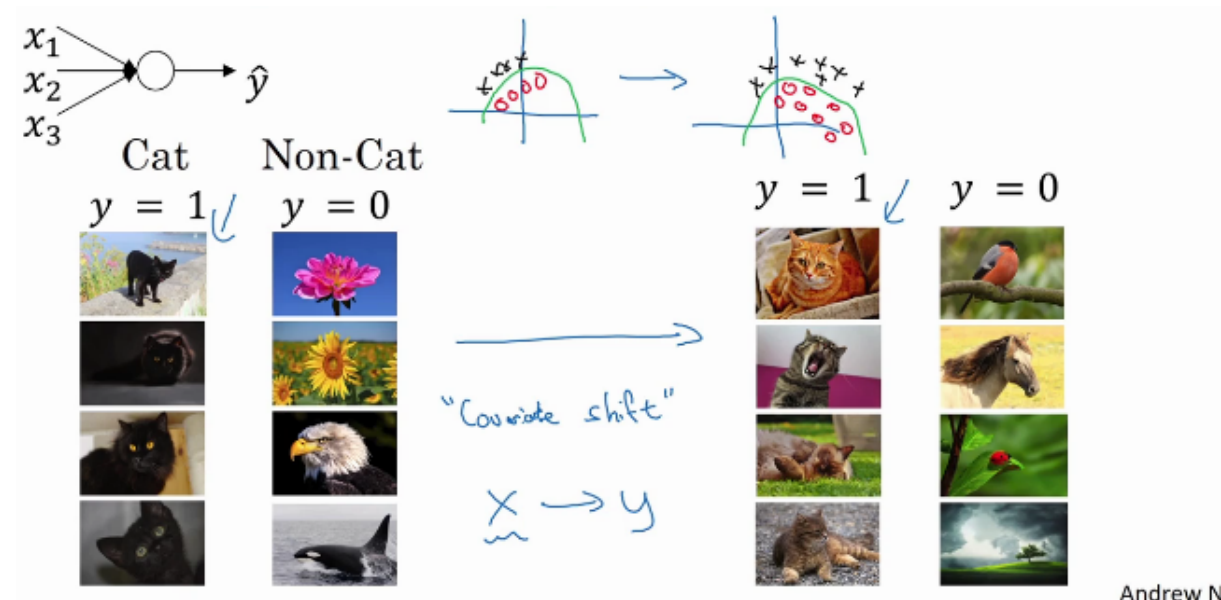
Implementando gradiente descendiente

Para el backpropagation podría utilizarse con momentum, RMSProp o Adam.

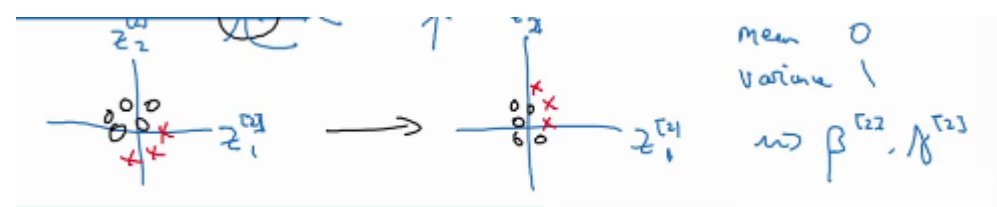


Veamos porque es que realmente Batch Normalization funciona y hace que el aprendizaje sea más rápido.

Para ello consideremos que sucede en el caso del cambio de input features. Por ejemplo, gatos negros y luego gatos que no sean negros. Esto puede provocar que la red no funcione correctamente.



Básicamente Batch Norm lo que hace es tratar que los parámetros no varíen demasiado ante el cambio de valores de entrada. Los parámetros β y γ deberían fluctuar muy poco.



Batch norm aplicado a mini-batch actúa como regularización:

- cada mini batch es escalado por el valor medio/varianza en ese mini batch.
- esto agrega algo de ruido a los valores z de ese mini batch. Similarmente a dropout agrega ruido a cada activación de cada capa oculta.
- Esto tiene un efecto de regularización.

Dado que es pequeño no es un efecto notorio de regularización, para ello conviene implementar dropout.

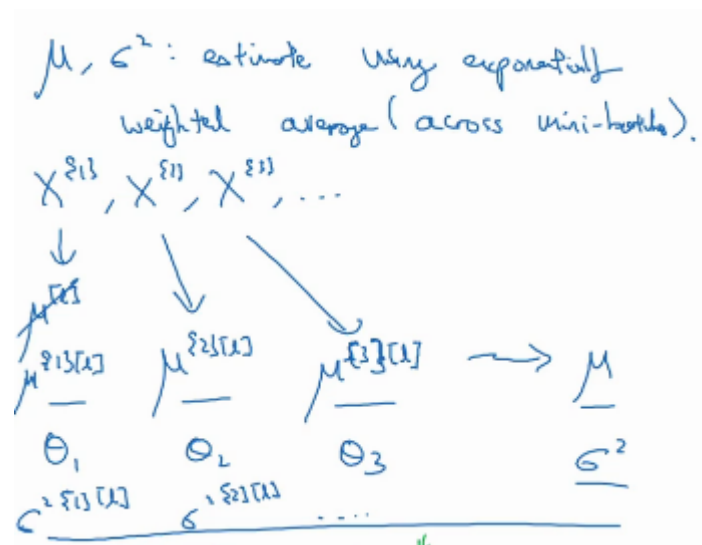
No es recomendable utilizarlo ya que no es la idea del mismo.

Resumiendo Batch norm

Dado que para calcular \tilde{z} se necesita μ y σ y, para que estos sean calculados con precisión deben ser calculados con un número considerable de ejemplos. Para hacer esto hace lo siguiente: se calcula μ para el conjunto de mini-batches del training set y de ese conjunto se calcula un estimador. Esto mismo se hace para σ .

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$
$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$
$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
$$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta$$

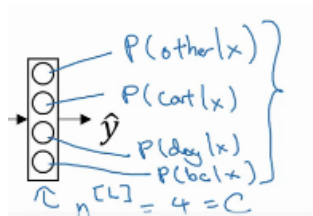
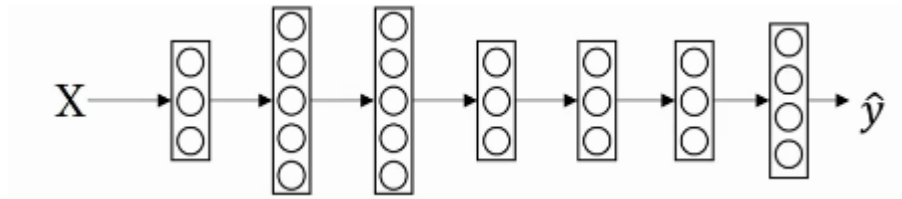
Luego se implementa el promedio pesado exponencial para utilizar en el test set, para hacer la normalización batch.



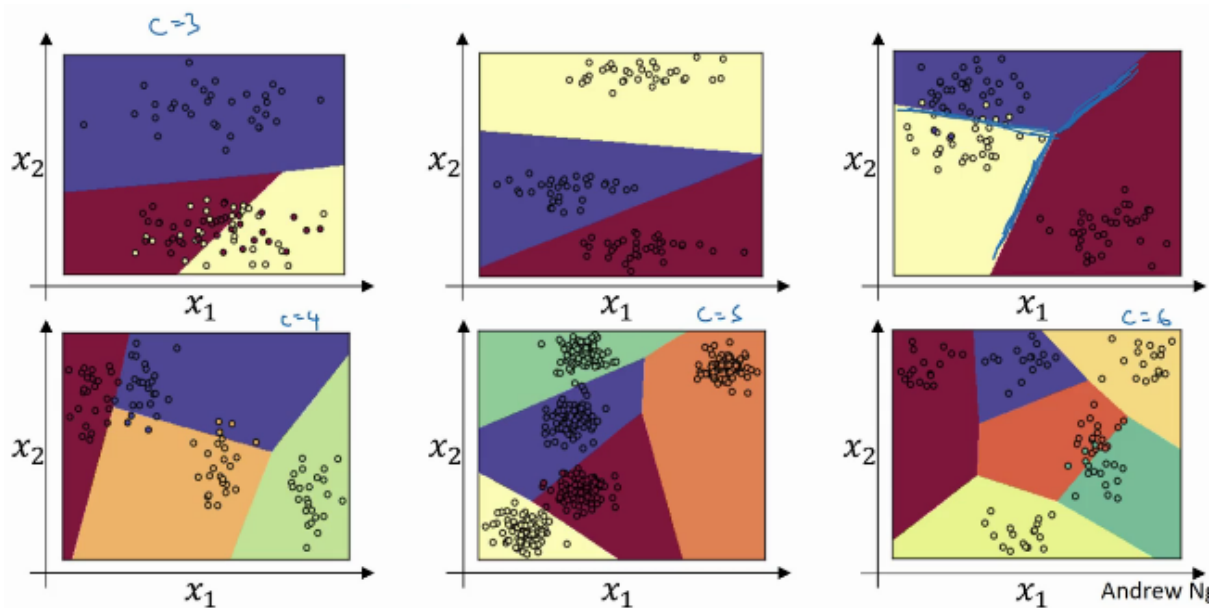
Multi-class classification

Regresión Softmax

Si en lugar de requerir una clasificación binaria se requiere una clasificación multi conviene usar softmax. Por ejemplo, si se quiere clasificar perros, gatos, pollitos y los que no lo son.



Veamos algunos ejemplos sin capas ocultas, veamos que los contornos son rectos.



Para tener contornos más complejos se deberían agregar capas ocultas.

Entrenar un clasificador Softmax

Si el número de salidas en softmax = 2, se obtiene una regresión logística.

Introduction to programming frameworks

Deep Learning Frameworks

Ejemplos de estos frameworks:

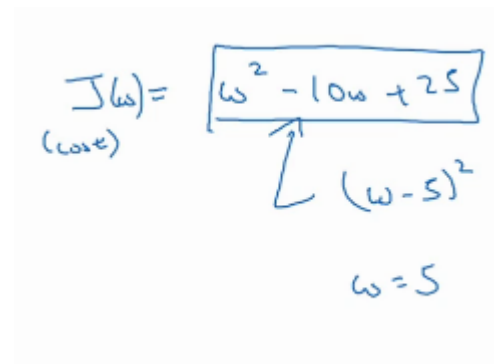
Caffe/Caffe2, CNTK, DL4J, Keras, Lasagne, mxnet, PaddlePaddle, TensorFlow, Theano, Torch.

Elección:

- Facilidad de programación (desarrollo e implementación).
- Velocidad para correr.
- Que sea verdaderamente open source (con una gran comunidad que lo mejore).

Tensor Flow

Ejemplo de motivación: minimizar la función de coste. De manera más general, se tendrá una función $J(w,b)$.


$$J(w) = \boxed{w^2 - 10w + 25}$$

(cost)

\swarrow $(w-5)^2$

$w=5$

```

1 import numpy as np
2 import tensorflow as tf
3
4 w = tf.Variable(0, dtype=tf.float32) # Initialize to zero
5 cost = tf.add(tf.add(w**2, tf.multiply(-10., w)), 25) # Define cost
6 # cost = w**2 - 10*w + 25 also works. Some basic ops are overloaded.
7 train = tf.train.GradientDescentOptimizer(0.01).minimize(cost) # Define training method
8
9 init = tf.global_variables_initializer()
10 session = tf.Session()
11 session.run(init)
12 print( sess.run(w) ) -> 0.0
13
14 # Como no se corrió nada todavía se obtiene cero.
15
16 session.run(train) # correr un paso de gradient descent -> 0.1
17
18 for i in range(1000):
19     session.run(train)
20 print(Session.run(w)) # se obtiene el valor deseado 4.999 cercano a 5.

```

```

import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0], dtype=tf.float32)
x = tf.placeholder(tf.float32, [3, 1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0] # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()

session = tf.Session()
session.run(init)
print(session.run(w))
}
with tf.Session() as session:
    session.run(init)
    print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x: coefficients})
print(session.run(w))

```