Week 1 Practical Aspects of Deep Learning

Sunday, 15 July, 2018 22:56

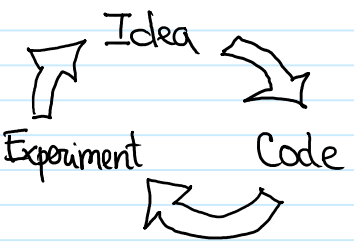
Setting up an ML App

Sunday, 15 July, 2018 23:02

Applied Machine Learning is a highly iterative process

We need to figure out the optimal :

- # of layers
- # of hidden units
- learning rate
- what activation functions to use for each layer



ML is applied in many different domains but in many cases the intuitions from one domain do not easily transfer to other domains.

Natural Language Processing, Vision, Speech,

Structured data { ads, Search, security, logistics

The best choices for the optimal parameters depend on:

- amount of data available
- input features available
- computer configuration (GPU vs CPU)
- etc.

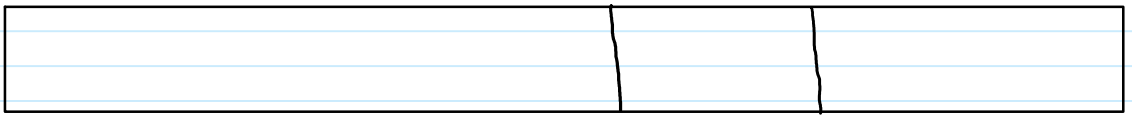
Even for the best experts it is impossible to correctly guess the correct hyperparameters.

→ use an iterative cycle to optimize.

→ how fast we can iterate determines how fast we can optimize.

TRAIN / DEV / TEST SETS SIZES

Data:



training set | hold-out cross-validation set (aka development set aka dev set) | test set

Algo:

train model on training set

every now & then check which model performs well on the dev set

take the best model & evaluate on test set

↳ this allows us to get an unbiased estimate on how well our algorithm is doing.

Historically it was common to take our data and split it:

70% : 30%
training test

or

60% : 20% : 20%
train dev test

this was used for datasets of up to 10,000 examples.

Currently (in the big data era) when we have 1,000,000 example data sets the trend is for the dev set & test set to be allocated a much

smaller percentage b/c the goal of the

dev set is to evaluate which of the 10 different models work better and we don't need 20% of the 1E6 data set for that. We are probably ok w/ 10,000 examples in our dev set.

test set similarly is used to evaluate how well we are doing given a final classifier.

→ we might determine that 10,000 examples are more than enough to determine how well we are doing.

So in this case we have:

98% : 1% : 1

train. dev test

MISMATCHED TRAIN / TEST DISTRIBUTIONS

Let's say we have a ML App that finds cats in pictures:

	Training Set	Test set
Cat pictures from	Webpages - nicely framed, high resolution - many millions available by crawling webpages	Users using the app - poorly framed, various resolutions - using phone camera, low lighting conditions - few compared to training set

these two distributions of data might be different

A good heuristic is to make sure the dev set and test set come from the same distribution.

→ b/c we are using the dev set is used to improve performance.

Sometimes the test set is not needed (only the dev set is used).

↳ test set is used to give us an unbiased estimate of the performance of our model but if we don't need the unbiased estimate we can use only the dev set

↳ in this case some will call the dev set the test set but they are using it as a hold out cross validation set.

REMEMBER: in this case we DO NOT get an unbiased estimate of the performance and we need to be wary of not overfitting to this "test set"

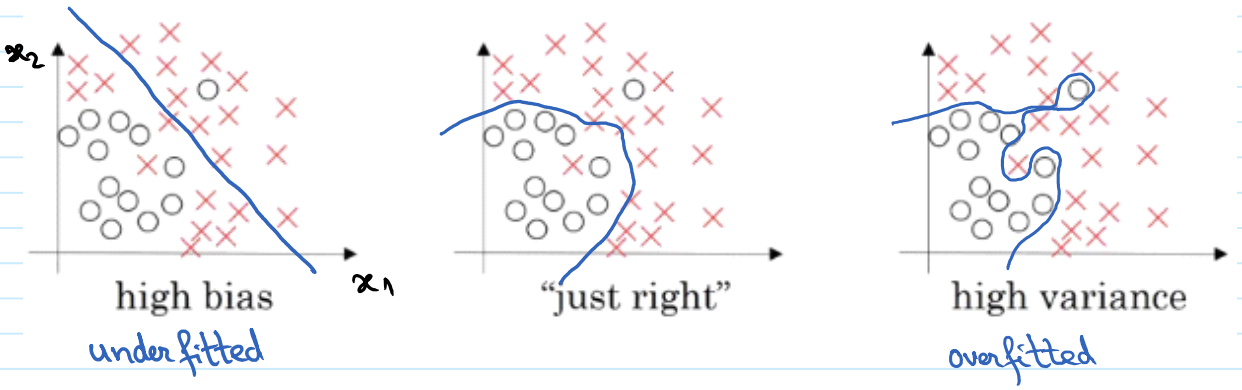
Bias / variance

Sunday, 15 July, 2018 23:50

Almost all good practioners of ML have a good intuition for Bias & Variance.

Also there is not enough discussion around bias & variance trade off.

BIAS & VARIANCE:



Screen clipping taken: 2018-07-15 23:56

In this case as we only have 2 features (x_1 & x_2) we can plot the data and visualize bias & variance but in a high dimensional models we can't plot the data so we can't visualize the decision boundary.

In the case of high dimensional problems we will use a couple of metrics to get a feel for bias & variance.

Training set error vs Dev Set error.

Let:	CASE A	CASE B	CASE C	CASE D
Traing set error	1% (we are really	15%	15%	0.5%

Traing set error	1% (we are really good)	15%	15%	0.5%
Dev set error	11% (relatively poorly)	16%	30%	1%
	this is a HIGH VARIANCE case that might indicate overfitting as the model doesn't generalize well to the dev set.	it looks like the model is not doing very well on the training set which is indicative of underfitting the data and so this algorithm has HIGH BIAS Note: this algo generalizes pretty well to the dev set so it is LOW VARIANCE	HIGH BIAS & HIGH VARIANCE	LOW BIAS LOW VAR

ABOVE DESCRIPTION ASSUMPTIONS:

- Optimal error (aka Bayes error) is $\approx 0\%$ (i.e assume humans can perform really well & achieve $\sim 0\%$ error)

↳ if the optimal error was much higher (say 15%.) then for case B we wouldn't say we have high bias though we have low variance.

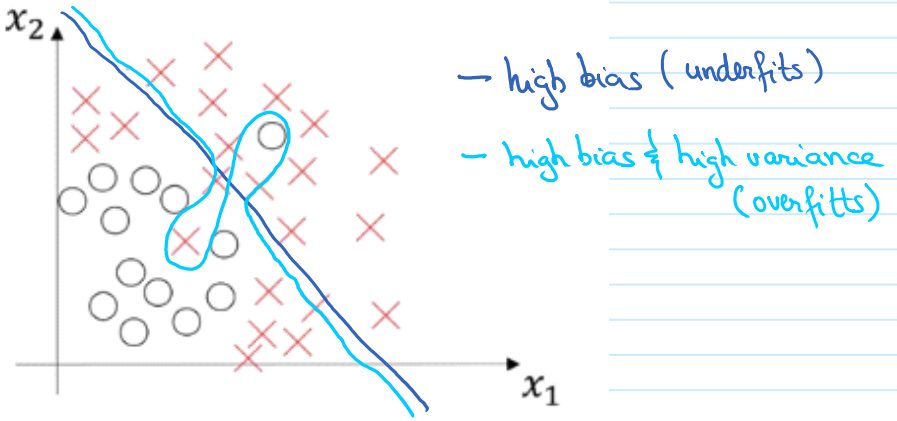
↳ what do we do when no classifier can do very well (i.e. we have blurry images) where no system can do very well than the Bayes error is much higher and the analysis changes.

- The train & dev set are drawn from same distribution.

↳ if this is not the case a more sophisticated analysis is used.

How does high bias & high variance look like

This is the worst of both worlds.



Screen clipping taken: 2018-07-16 00:31

In high dimensional data sets this case is more common.

Basic recipe for ML

Monday, 16 July, 2018 10:26

RECIPE:

1) Q: Does the model have high bias?

→ look @ the training data performance

A: Try bigger network (more hidden layers or hidden units)

Train it for more iterations

Train it w/ different optimization algorithms.

Try a different NN architecture (it might not work)

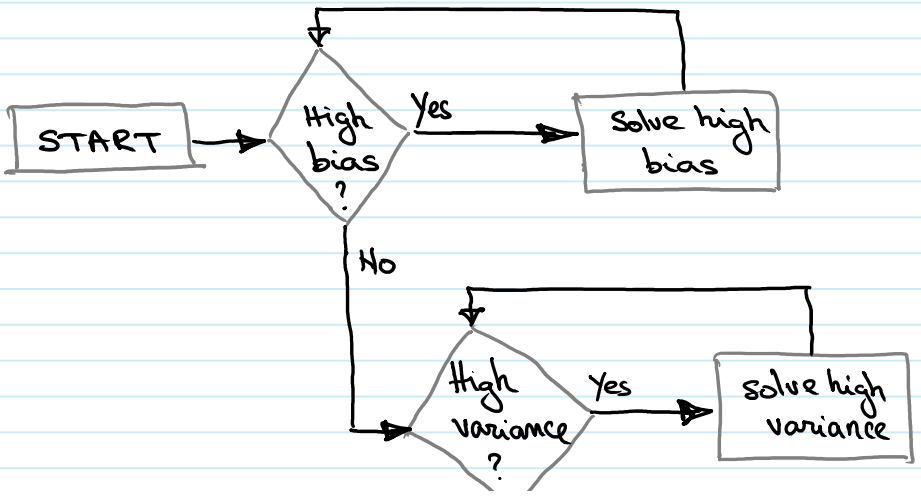
2) Q: Do you have a variance problem?

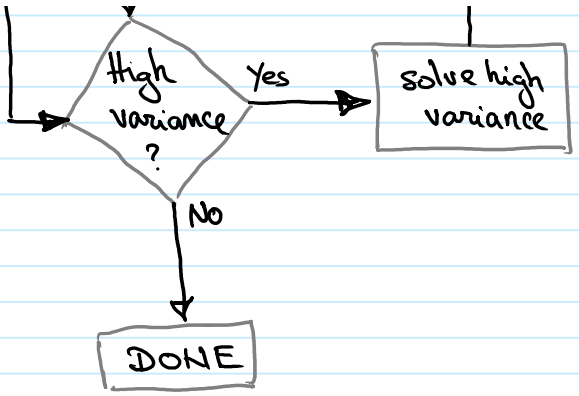
→ look @ the dev set and check if we are able to generalize the training set performance to the dev set performance.

A: Use more data.

Try regularization.

Try a different NN architecture.





The solutions to high bias and high variance don't overlap much.

BIAS VARIANCE TRADEOFF:

→ in the old days this used to be the case:

improving bias hurt variance & viceversa.

→ nowadays as long as we can

This can not always be acomplished
- train deeper networks (as long as we regularize appropriately)
- get more data

this tradeoff is less proeminent (we can drive down bias w/out hurting variance & vice versa).

The decrease impact of the bias-variance tradeoff is one of the big reasons that supervised learning has benefited so much from deep learning.

Regularization - L2 Norm

Monday, 16 July, 2018 10:48

When we suspect the NN is overfitting the data (i.e. high variance problem) we can try resolving it by

1) Regularization or 2) add more training data.

We'll develop the Regularization idea using Logistic Regression and then apply them to Neural Networks.

REGULARIZATION USING LOGISTIC REGRESSION:

In Logistic Regression we are trying to minimize the cost function $J(w,b)$

$$\min_{w,b} J(w,b) = \min_{w,b} \left[\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \right]$$

over all training examples m, where $m \in \mathbb{R}^{n_x}$ (a $[n_x, 1]$ vector), $b \in \mathbb{R}$ (a real number).

To regularize $J(w,b)$ we redefine it as:

$$J(w,b) = \underbrace{\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}_{\text{doing well on training set}} + \underbrace{\frac{\lambda}{2m} \|w\|_2^2}_{\text{regularization term}} + \cancel{\frac{\lambda}{2m} b^2}$$

in practice this is ommitted

Where $\|w\|_2^2$ is called the "norm of w squared" aka "L2 norm":

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

↳ Square Euclidian norm of the parameter vector w

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

L2 norm

→ Square Euclidian norm of the parameter vector w.

λ = regularization parameter

→ it is set using the development set and it is a trade off between doing well on the training set and setting the L2 norm parameter to be small which helps overfitting.

λ is another hyperparameter that we have to tune.

NOTE: "lambda" is a reserved keyword in Python so we will use "lambd" (missing "a") to prevent clashing.

Using the square Euclidean norm in the regularization is called an L2 REGULARIZATION (b/c the Euclidean norm is also called the L2 norm)

Regularizing b is ommitted b/c w is a high dimensional value, while b is a single dimensional value. So in practice most of the parameters are in w so "most of the variance lies with w" and b is somewhat inconsequential and regularizing it won't make much of a difference.

L1 REGULARIZATION (used less often):

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

Using L1 Regularization results in a SPARSE w (w will have a lot of 0s in it.) which some argue helps

compress the model b/c for the parameters that are 0 we need less memory to store the model.

REGULARIZATION USING NEURAL NETWORKS:

In a neural network we have a cost function J that depends on all our parameters:

$$J(w^{[1]}, b^{[1]} \dots w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

to regularize:

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

where the square norm used $\| w \|_F^2$ is known as the Frobenius norm:

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

this norm is known as the Frobenius norm

the indices of the sumation are determined by the dimensions of $w \Rightarrow w: (n^{[l]}, n^{[l-1]})$

# of hidden units in layers l & $l-1$

How do we implement gradient descent over the regularized form of the loss function J?

Previously we used back prop to compute

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

and then update w as:

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

After regularization:

$$dw^{[l]} = \frac{\partial J}{\partial w^{[l]}} + \frac{\lambda}{m} w^{[l]}$$

and we compute the update to $w^{[l]}$ same as before.

$$w^{[l]} = w^{[l]} - \alpha \left(\frac{\partial J}{\partial w^{[l]}} + \frac{\lambda}{m} w^{[l]} \right)$$

$$= \underbrace{w^{[l]} - \alpha \frac{\lambda}{m} w^{[l]}}_{w^{[l]}\left(1 - \alpha \frac{\lambda}{m}\right)} - \alpha \frac{\partial J}{\partial w^{[l]}}$$

also known as "weighted decay" b/c every update we decrease $w^{[l]}$ by $\left(1 - \alpha \frac{\lambda}{m}\right)$ regardless of $\frac{\partial J}{\partial w^{[l]}}$

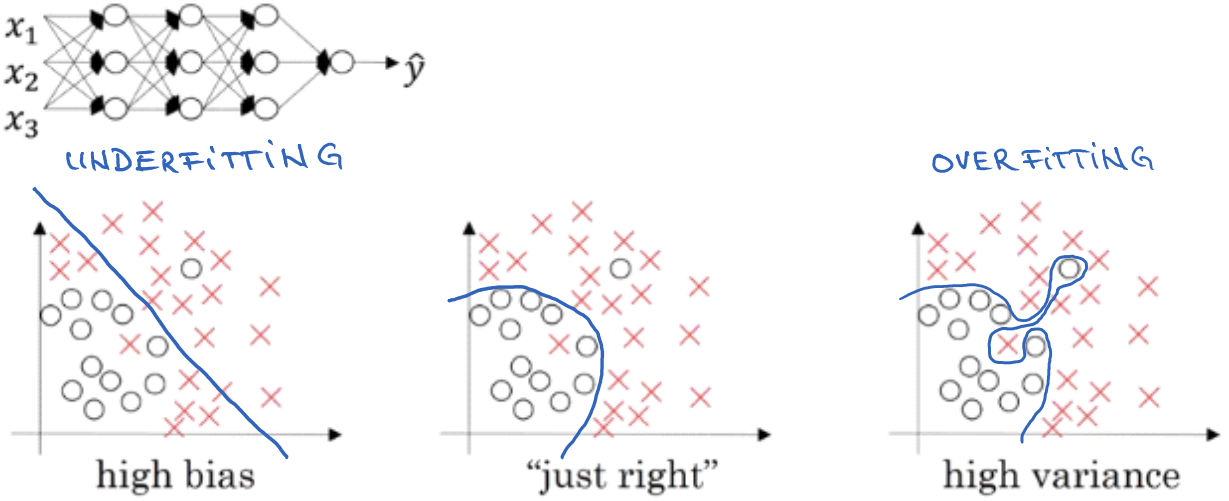
$\left(1 - \alpha \frac{\lambda}{m}\right) < 1$

So the alternative name for L2 Regularization is Weight decay.

Why regularization reduces overfitting

Monday, 16 July, 2018 13:13

How come regularization doesn't help with reducing bias? How does it work?



Screen clipping taken: 2018-07-16 13:19

Given a regularized cost function:

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

So why is it that shrinking the parameters by the L2 norm or the Frobinius norm might cause less overfitting.

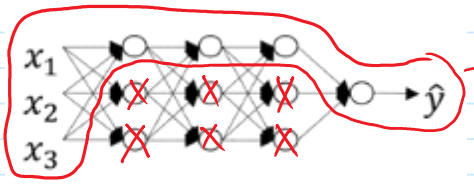
INTUITION A:

If we set λ to be really big then we will be incentivized to set the weight matrices $w^{[l]}$ to be close to $\emptyset$.

Remember λ participates in the "weight decay" $\left(1 - \alpha\frac{\lambda}{2m}\right)$ factor of the weights when we update:

$$w^{[l]} = w^{[l]}\left(1 - \alpha\frac{\lambda}{2m}\right) - \alpha\frac{\partial J}{\partial w^{[l]}}$$

so the larger λ the closer to $\emptyset$ $w^{[l]}$ gets.



this is a simplified NN where we "zero out" the impact of many of the hidden units.

Screen clipping taken: 2018-07-16 13:29

a simplified NN is a much smaller NN so it's going to move us from overfitting to underfitting.

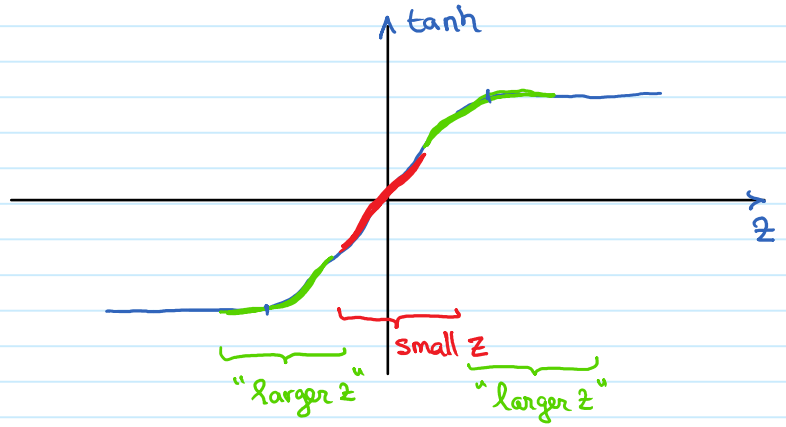
smaller NN can't capture the "nuances" of the data → they are more "linear"

Hopefully we can find the right λ that results in the "just right" fitting case in the middle.

The weight are not actually "zeroed out" as much as their effect on the network output is greately diminished. Functionally this behaves like a "smaller" network that is less prone to overfitting.

INTUITION B:

Assume we are using the activation function $g(z) = \tanh(z)$



When z is small (close to 0): we are using the linear regime of tanh

When z is large (farther from 0): we are using the non linear regime of tanh

(non-linear activation function)

$\lambda \uparrow \rightarrow W^{[l]} \downarrow \rightarrow Z^{[l]} \downarrow \rightarrow$ operate in the linear regime

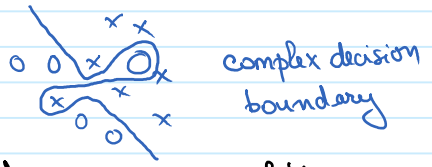
parameters are penalized for being large

b/c $z^{[l]} := w^{[l]} a^{[l-1]} + b^{[l]}$

(ignore effects of b)

We operate as if every layer is roughly linear → as if the network operates using linear regression.

↳ In this lecture we saw that if every layer is linear than the network (even if it is deep) behaves like a linear network and cannot fit complicated decision boundries.

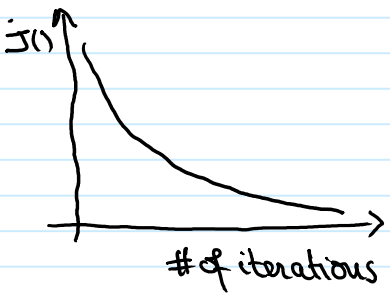


→ Fitting non-linear decision boundaries causes overfitting

→ If we can't fit complex decision boundaries then we can't overfit.

IMPLEMENTATION TIP:

When we debug gradient descent we plot the cost function J vs # of iterations and we want to observe a monotonic decrease after every iteration.



$$J(\ldots) = \ldots \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_l \|w^{[l]}\|_F^2$$

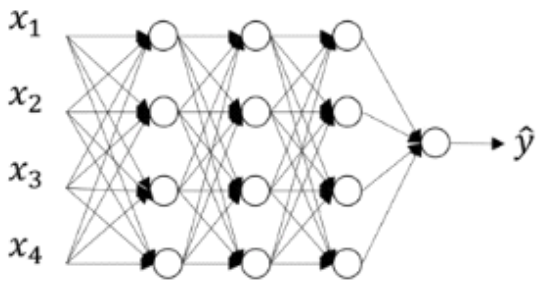
Make sure when we plot the cost function we used the regularized form or else the decrease won't be monotonic.

Dropout Regularization

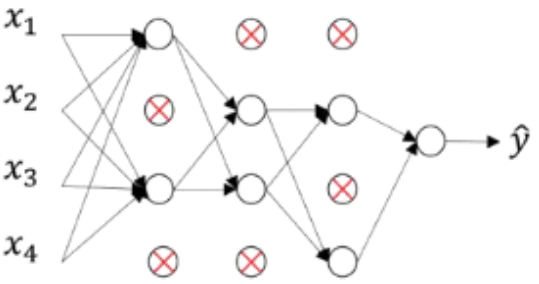
Monday, 16 July, 2018 17:08

Another way to prevent overfitting training data is to use dropout regularization.

→ this effectively decreases the number of nodes that are used @ any given time to learn the pattern in the training data.



Screen clipping taken: 2018-07-16 17:23



probability of "dropping" a node in a given layer

0.5 0.5 0.5

1. For each node set a probability for keeping or removing a node.
2. Remove "unlucky" nodes and we end up w/ a much smaller diminished network
3. Do backpropagation on this much diminished network training one example
4. Then repeat 1-3 for a different example.

So for each training example we are training the model using a "reduced" network.

IMPLEMENT "INVERTED DROPOUT"

Let's say we want to illustrate this for layer $l=3$.

FORWARD PROP.

```
keep_prob = 0.8 # probability of not tossing the given unit.
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
```

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)   # multiply will interpret False as 0 and True as 1.
a3 /= keep_prob            # prevent the expected value of a3 from being reduced.
```

inverted dropout technique

Let's say we have 50 units in layer 3.

@ 0.8 keep_prob we expect 20% → 10 units to be "dropped out" (shut off)

$$z^{[4]} = W^{[4]} \cdot a^{[3]} + b^{[4]}$$

↑ the power of $a^{[3]}$ will be reduced by 20% (i.e. 20 of its elements are zeroed out) so we need to divide by 0.8 in order to not change the expected value of $a^{[3]}$

The vector d3 is used to zero out different units.

A similar algorithm is used for BACK PROP.

The inverted dropout technique makes testing easier b/c we don't have scaling problems

On multiple passes through the training set we zero out different units.

MAKING PREDICTIONS AT TEST TIME:

$a^{[0]} = X$

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

No dropout is used explicitly → if we were we would only add noise to our output

The effect of the inverted dropout scaling

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = \ldots$$

$$\ldots$$

$$a^{[L]} = g^{[L]}(z^{[L]}) = \hat{y}$$

The effect of the inverted dropout scaling during training ensures no scaling is needed during testing.

Understanding Dropout

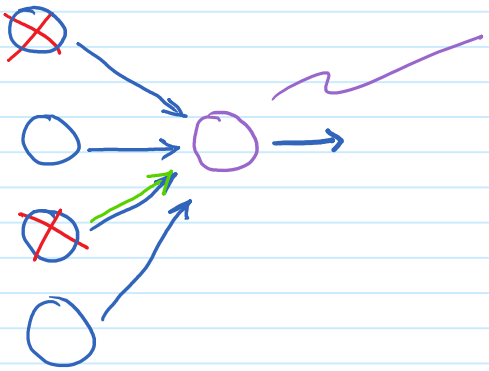
Monday, 16 July, 2018 17:48

INTUITION A:

By randomly knocking out hidden units from the network its as if we are using a smaller network. Using a smaller NN has a regularizing effect.

INTUITION B:

Let's look @ training from the perspective of a hidden unit.



This unit can't rely on any one feature as this feature could go away @ random during a given iteration.

So it would be reluctant to put all its weight on only a single feature and would be "motivated" to spread out its weights across all units.

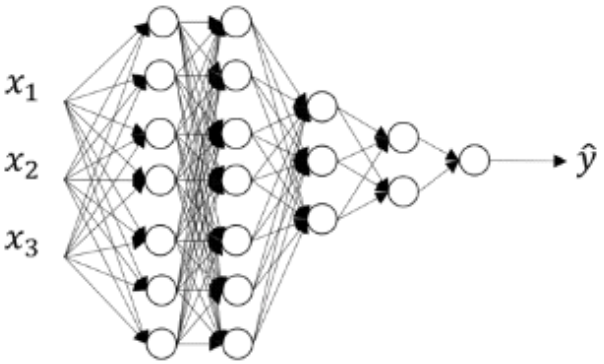
Spreading out the weights has the same effect as shrinking the squared norm of the weights (similar to L2 regularization) and helps to prevent overfitting.

Dropout can be shown to be an adaptive form of L2 regularization but the L2 penalty of each weight is different depending on the size of the activation multiplied into that weight.

IMPLEMENTATION HINT FOR DROPOUT

Screen clipping taken: 2018-07-16 18:05

Layer size 3 7 7 3 2 1



keep-prob (0.9) 0.7 0.5 0.7 1.0 1.0

$W^{[1]}$ $W^{[2]}$ $W^{[3]}$ $W^{[4]}$ $W^{[5]}$

We can vary keep-prob by layer.

$W^{[1]}$: (7,3)
$W^{[2]}$: (7,7)
$W^{[3]}$: (3,7)
$W^{[4]}$: (2,3)
$W^{[5]}$: (1,2)

As $W^{[2]}$ is the biggest weight matrix (7,7) so to reduce overfitting we can crank-up the regularization for layer 2 by setting a low keep-prob (0.5).

For layers where we worry less about overfitting we have a higher keep-prob. When we don't worry at all we set keep-prob = 1.0 (we are keeping all units in that layer i.e. we are not using dropout in that layer).

Technically we could also assign a keep-prob ≠1 for the input layer but this is rare.

NOTE RE DROPOUT:

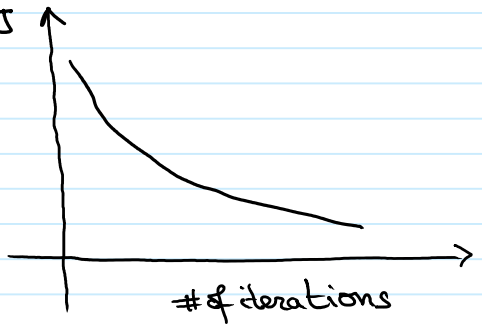
- First successful implementations of dropout where in computer vision.

B/C in computer vision the input size is so large we almost never have enough data (the ratio of $\frac{\#\text{ of features}}{\#\text{ of examples}}$ is large so the training results in overfitting if no regularization is applied)

→ but this intuition doesn't always generalize.

DOWNSIDE OF DROPOUT:

The cost function J is no longer well defined b/c on any given iteration some units are zeroed out



So checking that J is going down with every iteration is impossible. We lose this debugging tool when we use dropout.

WORKAROUND: Turn-off dropout by setting all keep_prob = 1
Check J is decreasing monotonously.
Turn-on dropout and hope no bugs were introduced.

Other Regularization Techniques

Monday, 16 July, 2018 18:29

DATA AUGMENTATION

When we are overfitting the best solution is to add more data, but getting more data can be expensive.

For image recognition flipping the input data can double the data set.

We didn't flip vertically b/c we don't want to detect upside down cats.

Screen clipping taken: 2018-07-16 18:33

We can also take random transformations of the initial data set (like rotation, zoom in, etc)

This tells the algorithm that flipping horizontally an image of a cat (or rotating or zooming etc) results in an image of a cat.

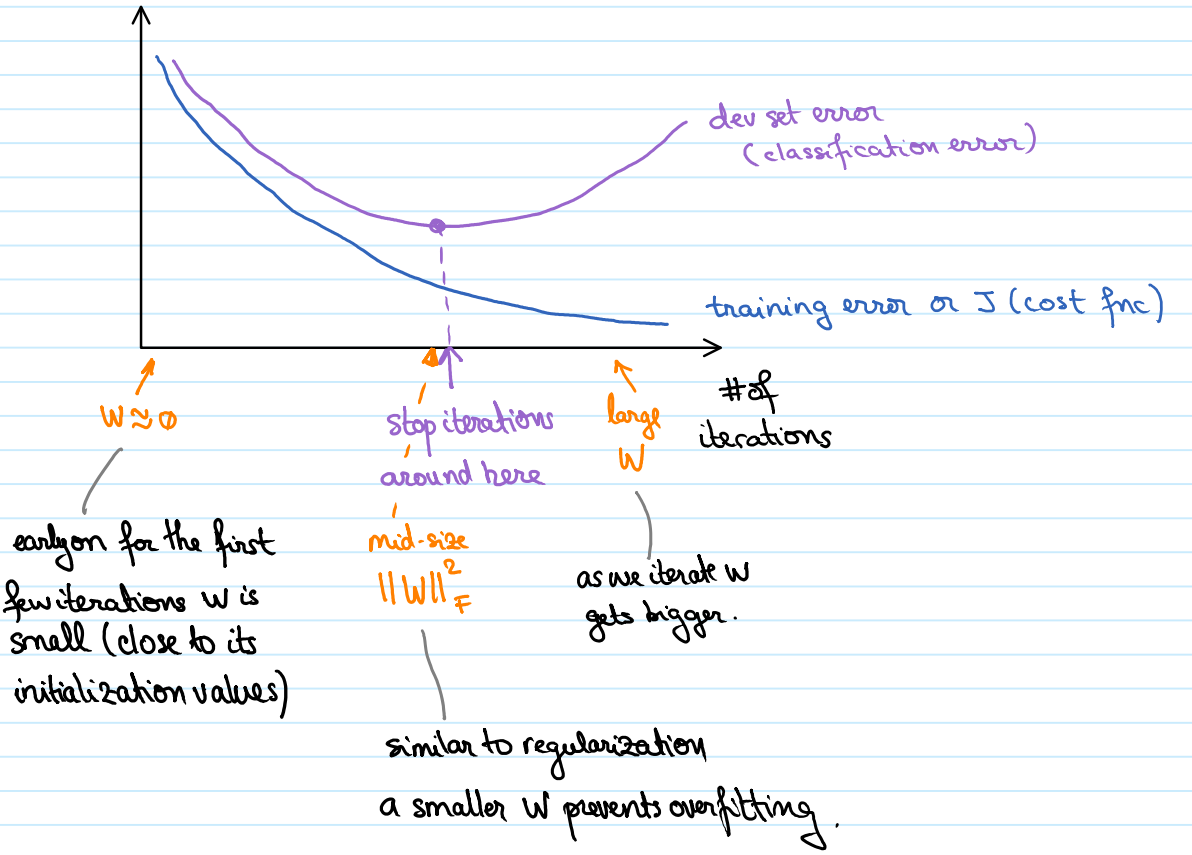
Screen clipping taken: 2018-07-16 18:35

Though these are cheap to augment the training data set they don't add as much information as getting a brand new independent image.

We can also add various distortions to augment the data.

Screen clipping taken: 2018-07-16 18:41

EARLY STOPPING:



DOWNSIDE OF EARLY STOPPING:

The machine learning process is made of a few steps.

1) Optimize cost function J
   - Gradient Descent, Momentum, RMS prob, etc.
2) Not Overfit
   - Regularization

We want to work on 1) independently of 2). This is called Orthogonalization.

↳ it keeps the exploration of hyperparameters well behaved.

don't optimize cost function and regularize @ the same time.

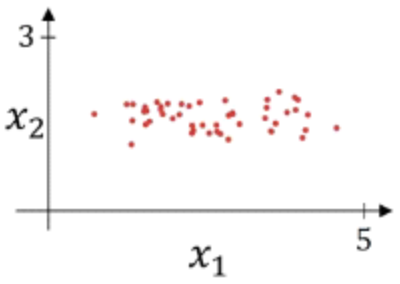
If we can afford the computation cost we can instead use L2 regularization using different values of λ. Though early stopping allows you a similar output w/out needing to try different values of λ.

Normalizing inputs

Monday, 16 July, 2018 22:41

Normalizing inputs speeds up training.

Given a two dimensional feature set $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

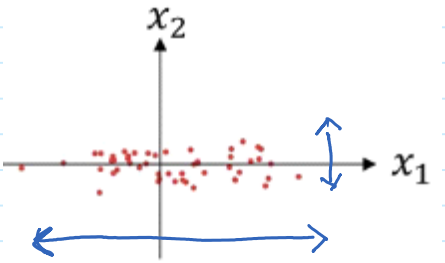


Screen clipping taken: 2018-07-16 22:44

Subtract mean (zero mean)

$\mu := \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$

$x := x - \mu$



Feature x_2 has a smaller variance than feature x_1

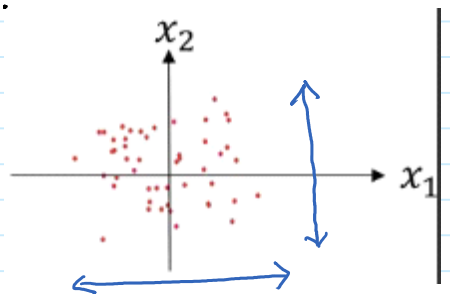
Screen clipping taken: 2018-07-16 22:48

normalize variances

$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} **2$

element wise Squaring.

$x /= \sigma^2$



Feature variance is now 1 for each feature

Screen clipping taken: 2018-07-16 22:54

Implementation Tip:

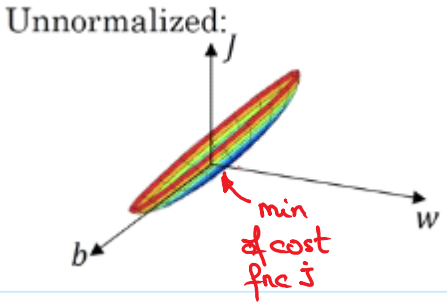
Use the same μ and σ^2 from training set to normalize the test set also.

↳ we want both train & test sets to undergo same transformation.

Why Normalize Inputs?

Cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

If we use unnormalized inputs the cost function ends up looking distorted (i.e. like an elongated bowl)



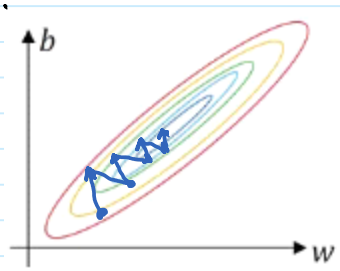
Screen clipping taken: 2018-07-16 23:01

If features are on very different scales. Say

$x_1 = 1 \ldots 1,000$

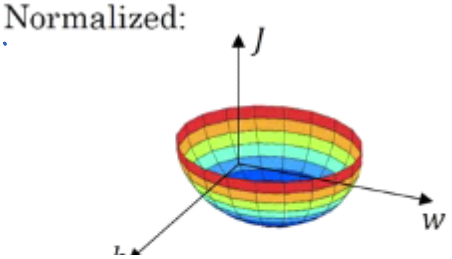
$x_2 = 0 \ldots 1$

This will cause w_1 & w_2 to take very different values.



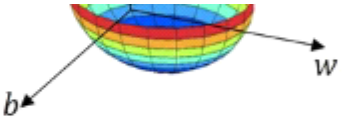
Screen clipping taken: 2018-07-16 23:05

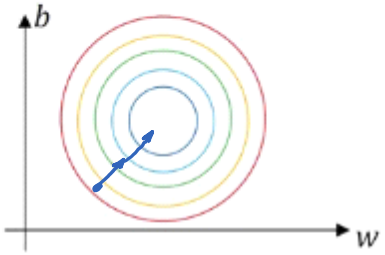
Because the features are not similar in scale we are forced to use a very small (slow) learning rate to accomodate the smallest scale and oscilate towards the minimum.



The cost function is easier to optimize when all features are similar scale.

$x_1 : 0 \ldots 1$ } these are similar scaled





x_1 : 0 ... 1
x_2 : -1 ... 1
x_3 : 1 ... 2

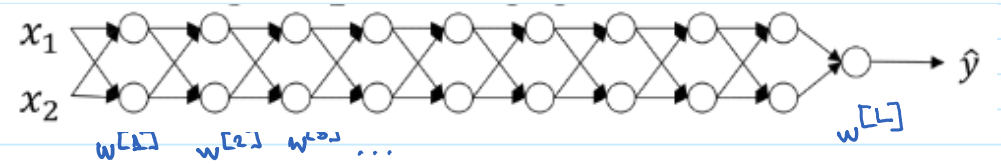
} these are similar scaled features → still it doesn't hurt to normalize them to $\mu = 0$ & $\sigma^2 = 1$

Vanishing/Exploding gradients

Monday, 16 July, 2018 23:13

Vanishing / Exploding gradients mean the derivatives can get either very small or very large.

→ this throws off the behaviour of the learning rate, making training difficult..



Screen clipping taken: 2018-07-16 23:16

Let's say for sake of simplicity we use a linear activation function like:

$g(z) = z$ and we set $b^{[l]} = 0$ (that is we ignore b)

In this case $\hat{y} = W^{[L]} W^{[L-1]} \dots W^{[2]} W^{[1]} X$

$z^{[1]} = W^{[1]} X$

$a^{[1]} = g(z^{[1]}) = z^{[1]}$

$a^{[2]} = W^{[2]} a^{[1]}$

$= W^{[2]} W^{[1]} X$

$\vdots$

$$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

technically the last weight matrix has a different dimension it is a (1,2) ... we'll ignore it for now

$$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$$

this is the last weight matrix

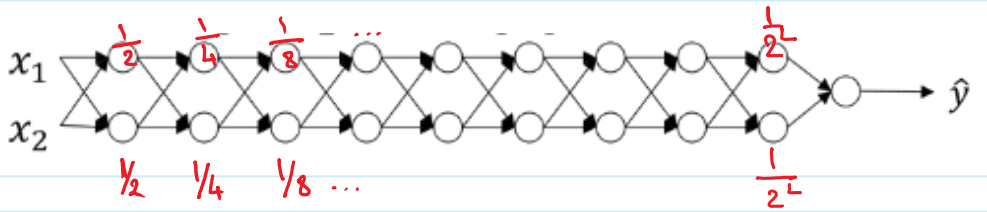
it's dimension is (1,2)

$$\hat{y} = 1.5^{L-1} X$$

If L is large (i.e. we are dealing with a very deep neural network)

$\hat{y}$ is very large b/c 1.5^{L-1} explodes (increases exponentially)

Conversely if $w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ then $\hat{y} = 0.5^{L-1}$ and the activation values vanish as they decrease exponentially as a function of the depth of the layer.



Similarly to the weights ($w^{[L]}$) the gradients ($dw^{[L]}$) either increase or decrease exponentially in a deep neural net as a function of layer depth.

Microsoft just showed they used a deep neural net with L = 152 and got great results

→ but w/ such a deep neural net training is very difficult as activation & gradients get very large & very small

↳ small gradients result in very slow learning.

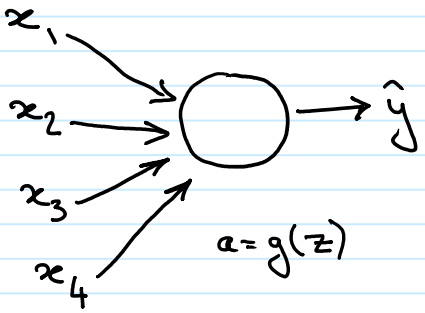
Weight initialization for Deep NN

Tuesday, 17 July, 2018 11:46

A partial solution to vanishing/exploding gradients is careful initialization of the weights.

Start w/ showing this for a single neuron & then generalize to an entire network.

$a^{[\emptyset]}$ Let $b = \emptyset$ for now



$$z = w_1 x_1 + w_2 x_2 + \dots w_n x_n$$

Notice that in order to prevent z from exploding the larger n is the smaller we want w_i to be b/c we want each term in the addition to be small.

Approach A (used for ReLU activation functions)

set the variance of $w_i = \frac{1}{n}$

i.e. $w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

# of input features into layer l.

If we use a ReLU activation function setting

$Var(w_i) = \frac{2}{n^{[l-1]}}$ works slightly better.

STASTICS NOTE:

multiplying a Gaussian variable by sqrt(x) sets the variance of the variable to x.

If the input features are roughly $\mu = 0$ & $\sigma^2 = 1$ this causes z to take a similar scale this reduces the impact of the explode/vanishing gradient problem as it sets each of the W matrices so they are around 1 and they don't explode or vanish too quickly

Other variances:

For tanh activation functions use $\sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier initialization)

Some use $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$ (He 2015)

Used for ReLU activation function.

In general σ^2 can be used as a hyperparameter also. Though it wouldn't be the first one to be tuned.

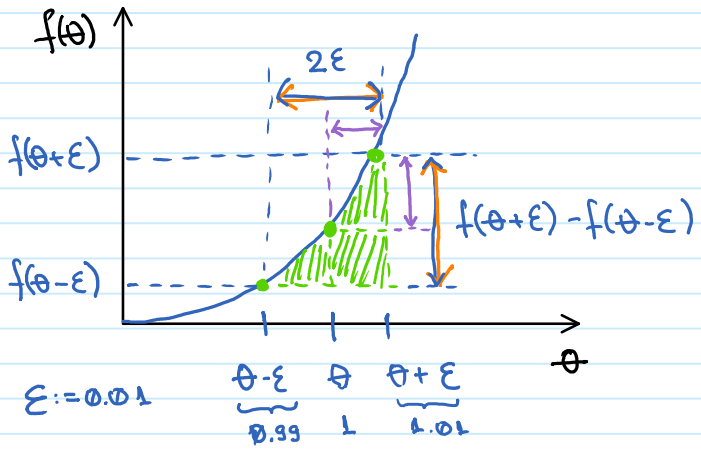
Numerical approximation of gradients

Tuesday, 17 July, 2018 13:01

To check that back propagation has been implemented correctly we can use a test called GRADIENT CHECKING to make sure we didn't introduce any bugs.

To understand how Gradient Checking works we'll first understand how to numerically approximate gradients.

Let's take $f(\theta) = \theta^3$



$$\frac{\text{height}}{\text{width}} = \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} = \frac{(1.01)^3 - 1^3}{0.01}$$

$$= 3.0301$$

NUMERICAL APPROXIMATION	ANALYTICAL GRADIENT
$\frac{\text{height}}{\text{width}} = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$	$g(\theta) = \frac{d}{d\theta} f(\theta) = \frac{d}{d\theta} \theta^3$
$= \frac{(1.01)^3 - (0.99)^3}{2(0.01)}$	$= 3\theta^2$
	$g(1) = 3 \times 1^2$
	$= 3$
$= 3.0001$	

approx error = 0.0001

approx error = 0.0301

So if we use the $\theta+\varepsilon$ & $\theta-\varepsilon$ to approximate the derivative (i.e. use the big triangle) we get a better approximation of the gradient

↳ even though this runs twice as slow as the single sided approximation it is worth it b/c it has much higher accuracy

CALCULUS NOTE:

If $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$

order of ε^2

We can show that for non-zero values of ε the error of $f'(\theta)$ is $O(\varepsilon^2)$

If $\varepsilon = 0.01 \Rightarrow$ Error $= 0.0001$

If $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$ than the error is $O(\varepsilon)$

So if $\varepsilon = 0.01 \Rightarrow$ Error $= 0.01$.

better to use $f'(\theta) = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ to minimize error.

Gradient checking

Tuesday, 17 July, 2018 13:33

This technique will save us a lot of time to check that our implementation of back prop is correct.

Take $W^{[1]}, b^{[1]}, \dots W^{[L]}, b^{[L]}$ and reshape and concatenate all values into a $(n, 1)$ vector θ

So instead of $J(W^{[1]}, b^{[1]} \dots W^{[L]}, b^{[L]})$ we use $J(\theta)$

Do the same for $dW^{[1]}, db^{[1]} \dots dW^{[L]}, db^{[L]}$ into $d\theta$

QUESTION: Is $d\theta$ the gradient of $J(\theta)$?

GRADIENT CHECKING IMPLEMENTATION (GRAD CHECK):

$J(\theta) = J(\theta_1, \theta_2, \theta_3 \dots)$

for each i in size θ:

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

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

So we want to check if:

$d\theta_{approx} \overset{?}{\approx} d\theta$

Calculate the Euclidean distance $\sqrt{\sum_{i=1} (d\theta_{approx\,i} - d\theta_i)^2}$

$$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$$

so we scale the output just in case any of the vectors are really small or large.

In practice use $\varepsilon = 10^{-7}$

$\approx 10^{-7}$ great! probably no bugs

10^{-5} (maybe check implementation for bugs)

10^{-3}

10^{-7} - there's a bug

If we think there is a bug check for which i the value of $d\theta_{approx}$ is much more different than $d\theta$ and hopefully it will help track down where the derivative computations are incorrect.

Gradient checking implementation notes

Tuesday, 17 July, 2018 13:51

Practical tips:

1) Don't use in training → only use to debug (and then turn off)

2) if implementation fails Grad Check look @ the individual components to find a bug.

→ compare $d\theta_{approx}[i]$ with $d\theta[i]$

if we observe that $db^{[l]}$ are far apart but $dw^{[l]}$ are quite close

we know the bug is probably in how we compute db.

(and vice versa)

3) Remember the regularization term.

$$\text{If } J(\theta) = \frac{1}{m}\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \underbrace{\frac{\lambda}{2m}\|w^{[l]}\|_F^2}_{\text{regularization term}}$$

$d\theta$ = gradient of J wrt θ including regularization term.

4) Grad Check DOES NOT work with Dropout regularization.

B/c dropout randomly eliminates units there is no easy cost function J that dropout can do gradient descent on.

↳ there is a very difficult J that we can compute but it is not feasible to use.

Workaround: Implement Grad Check w/ no Dropout (i.e. set all

keep-prob = 1)

Test algorithm

Disable Grad Check

Activate Dropout by resetting keep-prob.

5) Run Grad Check at random initialization and also after some training.

→ it might be that when w & b are initialized (and are small) Grad Check is ok. But as w & b get larger (as network is trained) Grad Check indicates problems.

Week 2 Optimization Algorithms

Tuesday, 17 July, 2018 22:22

The algorithms proposed in this week are meant to speed up the learning rate of our models. They decrease the # of iterations required to reach the global minimum.

Mini-batch gradient descent

Tuesday, 17 July, 2018 22:24

1) Training Deep Neural Nets is a highly iterative proccess.

2) ———— " ———— is done best with large amounts of data (this takes a long time)

Because of 1 & 2 having fast optimization algorithms can speed up the iteration cycle.

BATCH VS MINI-BATCH GRADIENT DESCENT:

Vectorization is one of the ways we can speed up training on m examples.

↳ process the whole training set w/out using for loops.

Vectorization might not be fast enough for very large data sets.

i.e. $m = 5{,}000{,}000$.

Vectorization has to traverse all 5EG examples before outputting a set of parameters w & b.

What if we were to get intermediate values for w & b after processing only 1,000 or 10,000 examples. (baby training sets aka mini-batches)

For mini-batches of 1,000 examples each we have 5,000 minibatches in a 5,000,000 examples data set

Let us denote each minibatch using curly braces as follows

$X^{\{t\}}, Y^{\{t\}}$

$(n_x, 1000)$ $(1, 1000)$ ...

where $t \in \{1, 2 \ldots 5000\}$

$$X = \left[\; x^{(1)}, x^{(2)}, \ldots x^{(1000)} \,\middle|\, x^{(1001)} \ldots x^{(2000)} \,\middle|\, \ldots \quad \middle| \ldots \; x^{(m)} \right]$$

$X_{(n_x, m)} = \left[x^{(1)}, x^{(2)}, \ldots x^{(1000)} \mid x^{(1001)} \ldots x^{(2000)} \mid \ldots \mid \ldots x^{(m)} \right]$

$X^{\{1\}}$ $X^{\{2\}}$ $X^{\{5000\}}$

$Y_{(1,m)} = \left[y^{(1)}, y^{(2)}, y^{(3)} \ldots y^{(1000)} \mid y^{(1001)} \ldots y^{(2000)} \mid \ldots \mid \ldots y^{(m)} \right]$

$Y^{\{1\}}$ $Y^{\{2\}}$ $Y^{\{5,000\}}$

RECAP

round $X^{(i)}$: i^{th} example (1D vector of feature values)

square $Z^{[l]}$: l^{th} layer parameters (1D vector of linear parameter values)

curly $X^{\{t\}}, Y^{\{t\}}$: t^{th} mini-batch of training examples.

MINI-BATCH GRADIENT DESCENT

repeat until J levels off:

{

for $t = 1 \ldots 5{,}000$:

{

Forward prop on $X^{\{t\}}$

$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$\vdots$

$A^{[L]} = g^{[L]}(Z^{[L]})$

Compute cost $J = \frac{1}{1000} \sum_{i=1}^{1{,}000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \| w^{[l]} \|_F^2$

(example index in batch $X^{\{t\}}, Y^{\{t\}}$)

Implement 1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$ (a training set of $m = 1{,}000$ examples)

}

}

5,000 X

||

"1 epoch" of

Backprop to compute gradients wrt $J^{\{t\}}$
(using $(X^{\{t\}}, Y^{\{t\}})$)

Update weights:

$$W^{[\ell]} := W^{[\ell]} - \alpha dW^{[\ell]}$$

$$b^{[\ell]} := b^{[\ell]} - \alpha db^{[\ell]}$$

}

}

"1 epoch" of Gradient Descent.

||

a single pass through the entire training set.

||

5,000 updates of the parameters

||

5,000 learning steps

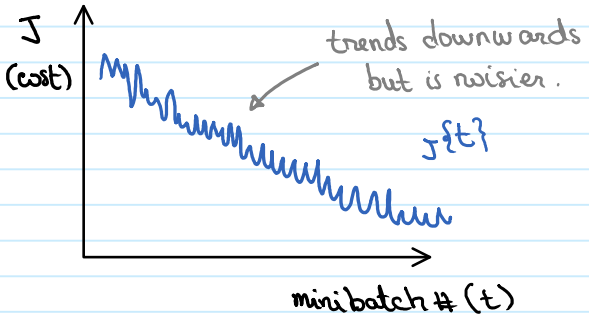
At the extreme when mini-batch size is 1 example we devolved to the non-vectorized gradient descent.

Understanding mini-batch gradient descent

Wednesday, 18 July, 2018 10:59

Batch Gradient Descent

Mini-batch gradient descent



Plots $J^{\{t\}}$ computed using $X^{\{t\}}, Y^{\{t\}}$

Cost is noisier because maybe $X^{\{1\}}, Y^{\{1\}}$ is an easy mini-batch (all examples are well behaved) but $X^{\{2\}}, Y^{\{2\}}$ is a harder mini-batch (some examples are mislabeled)

CHOOSING MINI-BATCH SIZE

If mini-batch size = m we end up w/ standard Batch Gradient Descent

$$(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$$

PROBLEM: Takes too long per iteration

If mini-batch size = 1 we end up w/ Stochastic Gradient Descent. Every example is its own mini-batch

$$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \dots$$

PROBLEM: Loose all the speed up from vectorization. The noise problem can be resolved by minimizing the learning rate.

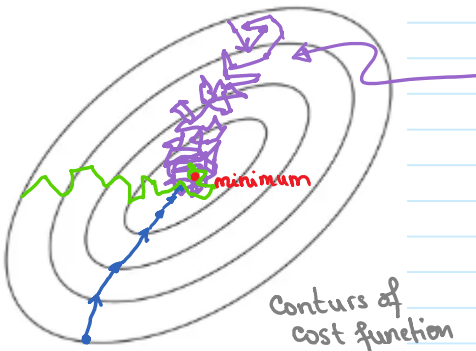
In practice in between 1 and m Not too large & not too small which results in fastest learning

Advantages: Speed up from vectorization (mini-batch size = 1,000)

Make progress w/out needing to wait to process entire training set

Each epoch allows us to take 5,000 steps.

Screen clipping taken: 2018-07-18 11:14



most of the time we point towards the global minimum but sometimes we head in the wrong direction (if we have a bad example) → very noisy. Stochastic Gradient Descent never converges it abits noisily the global minimum.

relatively large low noise learning steps for Batch Gradient Descent

GUIDELINES:

If small training size: Use Batch Gradient Descent.
($m \leq 2,000$)

If large training set size:

- Mini Batch size: 64, 128, 256

  B/C of the way computer memory is accessed powers of 2 mini-batch sizes are accessed faster.

- Make sure the Mini-Batch fits in CPU/GPU memory
  both $X^{\{t\}}$, $Y^{\{t\}}$ fit in memory.

Exponentially weighted averages

Wednesday, 18 July, 2018 11:36

To learn about alternatives to Gradient Descent optimization learning algorithms we need to understand EXPONENTIALLY WEIGHTED MOVING AVERAGES (this is a key component of several optimization algorithms)

DAILY TEMPERATURE IN LONDON OVER 1 YEAR

Jan 1st $\theta_1 = 4°C$

$\theta_2 = 9°C$

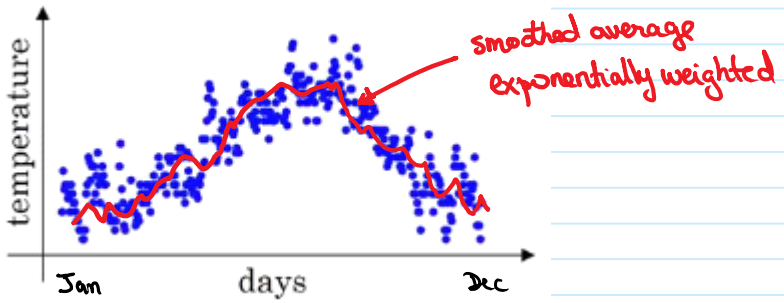
$\theta_3 = 7°C$

$\vdots$

Jan $\theta_{180} = 15°C$

$\theta_{181} = \vdots$

$\vdots$



Screen clipping taken: 2018-07-18 11:43

The data is noisy. If we want to smoothen it we can run the following algorithm:

$V_0 := 0$

$V_1 := 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$V_3 = 0.9 V_2 + 0.1 \theta_3$

$\vdots$

$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

- V_t: today smoothed temperature
- V_{t-1}: yesterday smoothed temperature
- θ_t: today raw temperature

In general:

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

When β is large previous values are given more weight (β) than the current value ($1-\beta$)

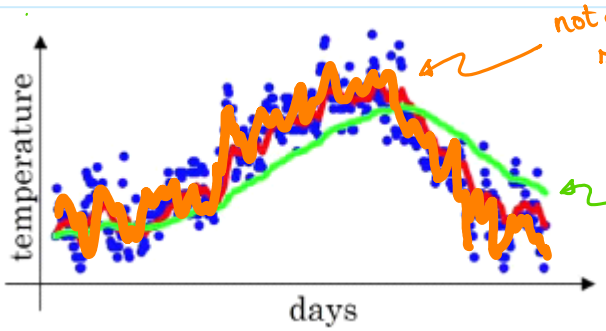
We can show that V_t averages over approximately

$$\approx \frac{1}{1-\beta} \text{ days of temperatures}$$

In the case above $\beta = 0.9 \Rightarrow \frac{1}{1-0.9} = 10$ days previous temperatures

If $\beta = 0.98 \Rightarrow \frac{1}{1-0.98} = 50$ days previous temperatures.

If $\beta = 0.5 \Rightarrow \frac{1}{1-0.5} = 2$ days previous temperatures (more supcestible to outliers)



not as smooth but responds rapidly to changes in temperature.

smoother curve but shifted to the right (i.e. lagging current day temperature)

Screen clipping taken: 2018-07-18 11:54

β is another hyperparameter that we will need to tune to find which one works best.

Understanding Exponentially Weighted Averages

Wednesday, 18 July, 2018 12:35

Remember : $V_t = \beta V_{t-1} + (1-\beta)\theta_t$

For $\beta = 0.9$

$v_{100} = 0.9\, v_{99} + 0.1\,\theta_{100}$

$v_{99} = 0.9\, v_{98} + 0.1\,\theta_{99}$

$v_{98} = 0.9\, v_{97} + 0.1\,\theta_{98}$

...

$v_{100} = 0.1\theta_{100} + 0.9\, v_{99}$

$= 0.1\theta_{100} + 0.9\left(0.1\theta_{99} + 0.9\, v_{98}\right)$

$= 0.1\theta_{100} + 0.9\left(0.1\theta_{99} + 0.9\left(0.1\theta_{98} + 0.9\, v_{97}\right)\right)$

...

$= 0.1\theta_{100} +$

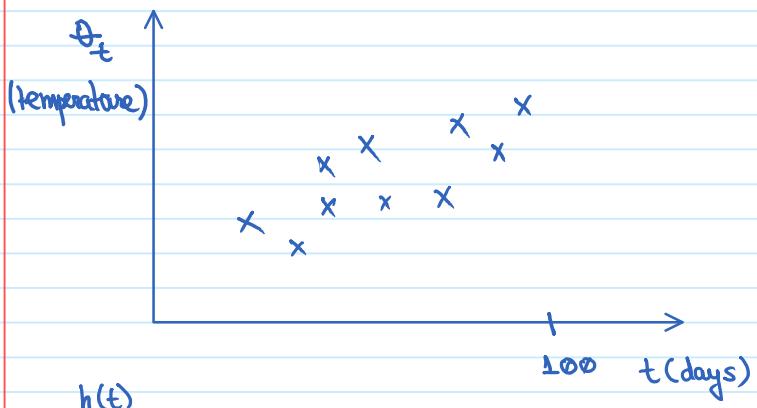
$0.1 \times 0.9 \times \theta_{99} +$ } "v_{99}"

$0.1 \times 0.9 \times 0.9 \times \theta_{98} +$ } "v_{98}"

$0.1 \times 0.9^3 \times \theta_{97} +$

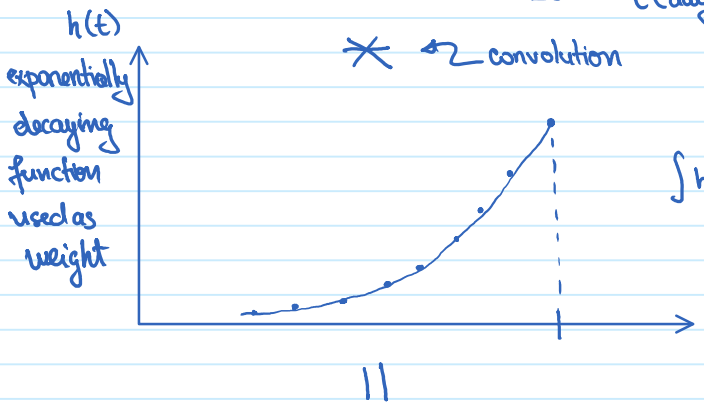
$0.1 \times 0.9^4 \times \theta_{96} +$

...



All the coefficients add up to ≈ 1 (up to a detail called bias correction)

→ b/c of this it is an exponentially weighted average.



$\int h(t) \simeq 1$

$$\text{Smoothed}(t) = \sum_{i=1}^{t} \theta_i \cdot h(i)$$

How many days are averaging over?

For $\beta = 0.9$ we notice that: $0.9^{10} \simeq 0.35 \simeq \frac{1}{e}$

in general $\underbrace{(1-\varepsilon)}_{0.9}^{1/\varepsilon} = \frac{1}{e}$

for $\varepsilon = 0.1$

so it takes about 10 days ($\alpha\ 1/\varepsilon$) for the weight to decay to about $1/3$ ($\frac{1}{e} \simeq 0.35$) of the peak.

For $\beta = 0.98$ it takes about $\frac{1}{0.02} = 50$ days $\quad (0.98^{50} \simeq \frac{1}{e} \simeq 0.35)$

for the weights values to drop to $\simeq 0.35$

IMPLEMENTATION:

$V = 0$

repeat {

get next θ_t

$V_\theta := \beta V_\theta + (1-\beta)\theta_t$

}

ADVANTAGES:
- this takes very little memory.
- very few computations.

Bias correction in exponentially weighted averages

Wednesday, 18 July, 2018 13:16

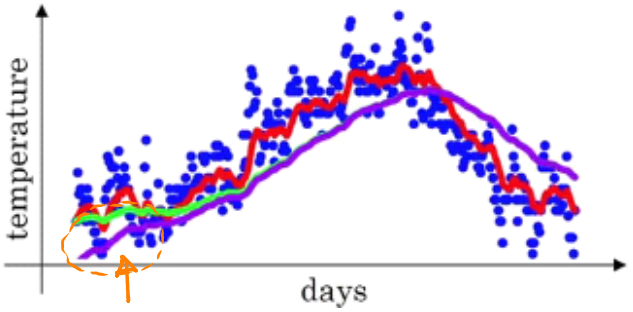
If we implement EWA as in the previous algorithm (using $\beta = 0.98$)

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

we expect to obtain the green curve.

But it turns out we obtain the purple curve which starts @ a lower value.

Bias correction fixes this sort inconsistency.



Screen clipping taken: 2018-07-18 13:20

The problem arises from initializing v_0 to 0 which causes

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

$$v_0 = 0$$

$$v_1 = 0.98 \overset{0}{v_0} + (1-0.98)\theta_1$$

$$= 0 + 0.02\theta_1$$

$$= 0.02\theta_1$$ (which is 2% of θ_1 → a very poor estimate of the 1st day temperature)

$$v_2 = 0.98 v_1 + 0.02\theta_2$$

$$= 0.98 \times 0.02\theta_1 + 0.02\theta_2$$

$$= 0.0196\theta_1 + 0.02\theta_2$$ (again v_2 much less than either θ_1 or θ_2 and it is not a good estimate for the 1st two days of the year)

A better approach is to use

$$v_t = \frac{v_t}{1-\beta^t}$$ (especially for small values of t)

↳ when EWA is still "warming up")

So for $t=2$ $\quad 1-\beta^t = 1-0.98^2 = 0.0396$

$$v_2 = \frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

(0.0396 = $+$)

as t increases $1-\beta^t \to 1$ which means the bias correction effect is no longer making a difference on the EWA output.

NOTE: Not everyone uses the bias correction but it is useful for tidying up the data.

Gradient Descent with Momentum

Wednesday, 18 July, 2018 13:35

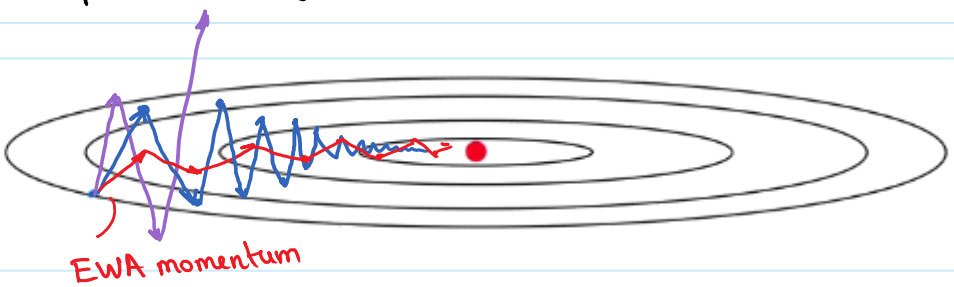
Also known as Momentum almost always works faster than the basic Gradient Descent.

BASIC IDEA:

Compute an Exponentially Weighted Average of the Gradients and use this averaged gradients to update the weights.

Similarly to dynamically adjusting the learning rate to decrease the noise of the gradient of each activation.

As an example let's say we want to optimize the cost function below:



Screen clipping taken: 2018-07-18 13:40

The oscillations prevent us from using a larger learning rate (faster learning)

↳ if we do use a large learning rate we might end up with a diverging behavior.

Another way to interpret our optimization problem is that on

- the vertical axis we want a small learning rate while on
- the horizontal axis we want a faster learning rate.

MOMENTUM IMPLEMENTATION:

On iteration t:

Compute dW, db on current mini-batch

$V_{dW} = \beta V_{dW} + (1-\beta) dW$ β is used to emphasize how much the

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

$$W := W - \alpha V_{dW}$$

$$b := b - \alpha V_{db}$$

"historical" (V_{dW}) gradient is to be used as the current gradient (dW)

higher β => less noise.

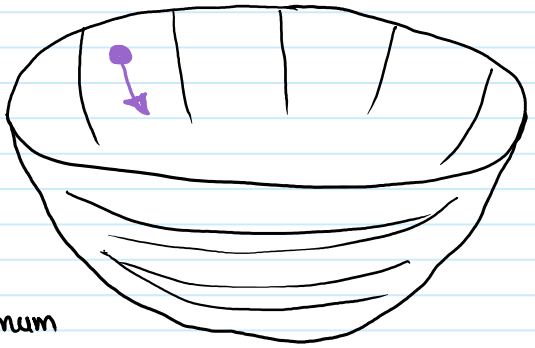
What this does is to smooth out the steps of gradient descent.

INTUITION (PHYSICS)

Given a bowl shape function that we want to minimize.

a ball is in the bowl and is subjected to
- acceleration
- velocity
- friction

on its path to the minimum



$$V_{dW} := \beta V_{dW} + (1-\beta)\, dW$$

$$V_{db} := \beta V_{db} + (1-\beta)\, db$$

β: friction; V_{db}: velocity; db: acceleration

IMPLEMENTATION DETAILS:

$V_{dW} = [0], db = [0]$ ← same dimension as dW & db

On iteration t:

$v_{dW} = [0, \ldots, 0]$, $db = [0, \ldots]$ ← same dimension as dW, b

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

Screen clipping taken: 2018-07-18 14:03

Hyperparameters: α & β The most common value for $\beta = 0.9$.

- Do we want to apply bias correction $\left(\frac{VdW}{1-\beta^t}\right)$?

  In practice not many people apply bias correction as after 10 iterations the moving average is already "warmed-up" and is no longer biased.

- in the literature of Gradient Descent w/ Momentum often the term $(1-\beta)$ is ommitted such that we get:

  $$VdW = \beta VdW + dW$$

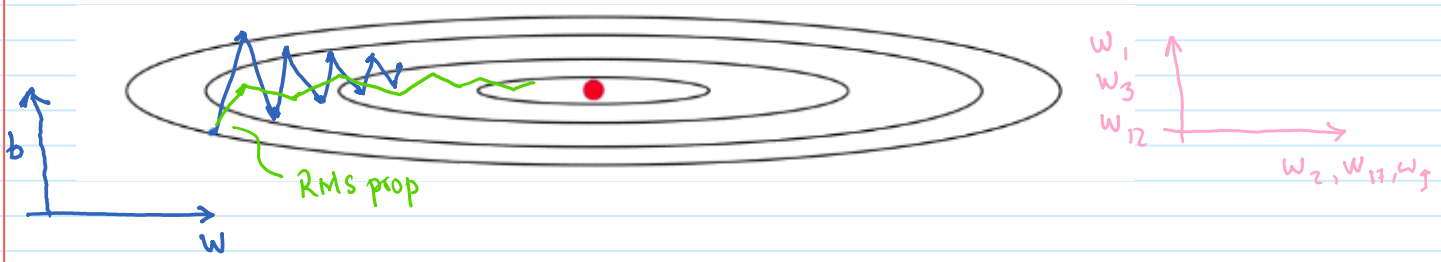
  The net effect of this is that VdW gets scaled by a factor of $\frac{1}{1-\beta}$ which actually is going to cause α (the learning rate) to be scaled by $\frac{1}{1-\beta}$.

  This formulation is more complex as it ties in the optimization of hyperparameter α and β together so is not as easy to use.

RMS prop

Wednesday, 18 July, 2018 14:14

Root Mean Square prop is another algorithm that can speed up Gradient Descent.



Let w & b be the parameters associated w/ the horizontal & vertical dimensions.
We want to slow down the learning rate on b while affecting w as little as possible.

IMPLEMENTATION

On iteration t:

Compute dW, db on current mini-batch.

$$sdW = \beta_2 sdW + (1-\beta_2)(dW)^2$$ ← element wise Squaring. the sign of the derivative is eliminated.

$$sdb = \beta_2 sdb + (1-\beta_2)(db)^2$$

$$W := W - \alpha \frac{dW}{\sqrt{sdW} + \varepsilon}$$

$$b := b - \alpha \frac{db}{\sqrt{sdb} + \varepsilon}$$

To prevent clashes b/w Momentum & RMS prop we use β_2 as the parameter for RMS prop

ε a small non-zero value (i.e. 10^{-8})

ε ensures numerical stability i.e. the division doesn't blow up when $\sqrt{\ } \to 0$

INTUITION:

Recall that on the horizontal direction (W) we want learning to go fast

vertical direction (b) we want learning to go slow

(damp it out)

So we are hoping that

sdW to be small so that in $W := W - \alpha \frac{dW}{\sqrt{sdW}}$ we are dividing by a small value.

sdb to be relatively large so that in $b := b - \alpha \frac{db}{\sqrt{sdb}}$ we are dividing by a large number. to slow down the updates

→ indeed the vertical derivatives are much larger than the horizontal derivatives

In high dimensional space (where there are many features) RMS prop will

- dampen the learning rate for the parameters whose gradients are oscillating wildly
- Keep the learning rate for the parameters whose gradients are not oscilating much at maximum rate

FUN FACT: RMS Prop was first proposed in a Coursera course Jeff Hinton taught.

Adam optimization algorithm

Wednesday, 18 July, 2018 17:23

Most optimization algorithms that have been proposed don't generalise well to the wide range of deep neural networks that we want to use.

RMS prop & Adam optimization are the exception. These two generalize well.

ADAM OPTIMIZATION = MOMENTUM + RMS PROP

ADAM IMPLEMENTATION:

$V_{dW} = 0 \qquad V_{db} = 0$

$S_{dW} = 0 \qquad S_{db} = 0$

On iteration t of all mini-batch iterations available:

Compute dw, db using current mini-batch.

Momentum:

$$\begin{cases} V_{dW} = \beta_1 V_{dW} + (1-\beta_1)\, dW \\ V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db \end{cases}$$

RMS Prop:

$$\begin{cases} S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, dW^2 \\ S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \end{cases}$$

In the original implementation of Adam we are doing bias correction:

$$\begin{cases} V_{dW}^{corrected} = \dfrac{V_{dW}}{1-\beta_1^t} \; ; \; V_{db}^{corrected} = \dfrac{V_{db}}{1-\beta_1^t} \\ S_{dW}^{corrected} = \dfrac{S_{dW}}{1-\beta_2 t} \; ; \; S_{db}^{corrected} = \dfrac{S_{db}}{1-\beta_2 t} \end{cases}$$

Update parameters using both Momentum:

$$W := W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \varepsilon} \; ; \; b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

parameters using both Momentum & RMS Prop

$\sqrt{S_{dw}^{corrected}} + \varepsilon$ $\sqrt{S_{db}^{corrected}} + \varepsilon$

ADAM HYPERPARAMETERS:

α : needs to be tuned

β_1 : 0.9 (dw Momentum) aka 1st moment

β_2 : 0.999 (dw^2 RMS Prop) aka 2nd moment

ε : 10^{-8}

} as recomended by the authors of the Adam paper

ADAM: Adaptive Moment estimation

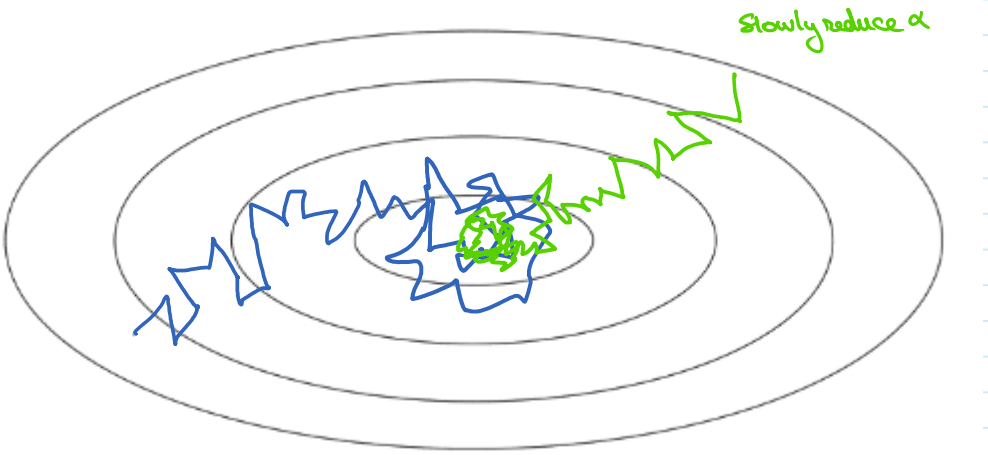
Not related to Adam Coates.

Learning Rate Decay

Wednesday, 18 July, 2018 20:24

Another way to speed up our learning algorithm is to slow down the learning rate over time. (as we expect we are getting closer to the global minimum).

→ this is refered to as Learning Rate Decay



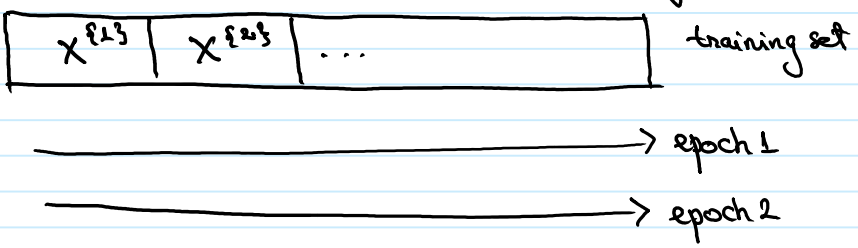
Screen clipping taken: 2018-07-18 20:29

The cost for mini-batch will tend towards the minimum but it won't converge.
It will oscilate (orbit) around the minimum

As we start α (learning rate) is large so we are learning fast. As we aproach the global minimum the learning rate gets smaller which results in a closer "orbit" around the minimum.

LEARNING RATE DECAY IMPLEMENTATION

Note: 1 epoch = 1 pass through entire training set



NOTE: decay_rate becomes another hyperparameter that needs tunning.

Idea A: $\alpha = \frac{1}{1 + \text{decay\_rate} \times \text{epoch\_num}} \alpha_0$ ← initial learning rate

Example:

$\alpha_0 = 0.2$

decay_rate = 1

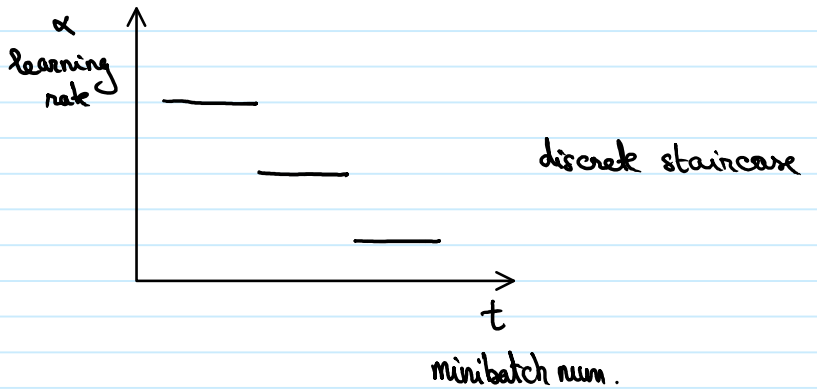
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	⋮

Idea B: $\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0$ (aka exponential decay)

Idea C: $\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$ ← minibatch #

Idea D:

Learning rate decreases after a certain # of steps.



Idea E: Manual decay.

If we are training a large model over many days, monitor the cost function and decrease the learning rate when we observe the learning has slowed down.

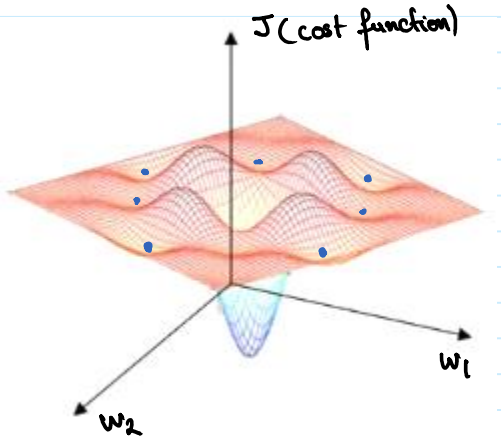
This works only if we are training a small # of models.

Learning Rate Decay is not at the top of the list of things to try to optimize speed.

Local optima problem

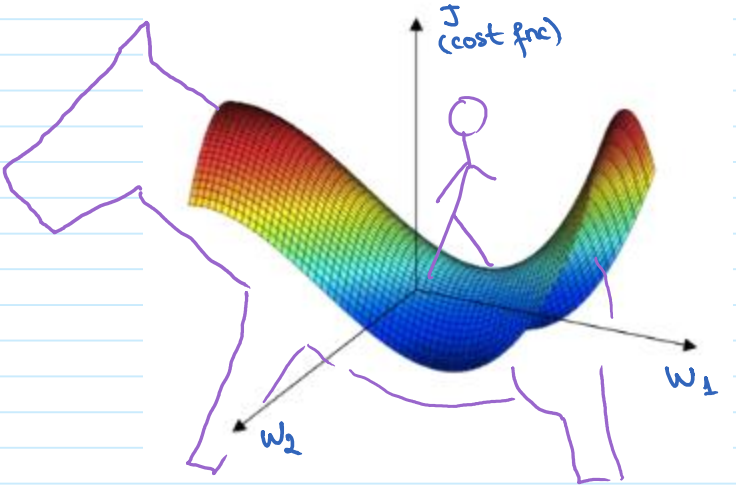
Wednesday, 18 July, 2018 21:02

People used to worry about learning algorithms getting stuck in local optima. Our understanding of this has changed as we learned more about NN.



Screen clipping taken: 2018-07-18 21:04

This is how we used to think about local optima but this is misleading as it turns out most points of local optima are like this. (saddle points) in high dimensional space.



Screen clipping taken: 2018-07-18 21:10

If the gradient is 0 the gradient in each dimension is either

concave  or convex 

If we are in a 20,000 dimensional space then for this point of gradient 0 to be a local optima then all 20,000 directions need to be convex ✓.

→ the chances of this happening are very small maybe $2^{-20,000}$.

It is much more likely that the gradients bend both up and down (i.e. saddle points)

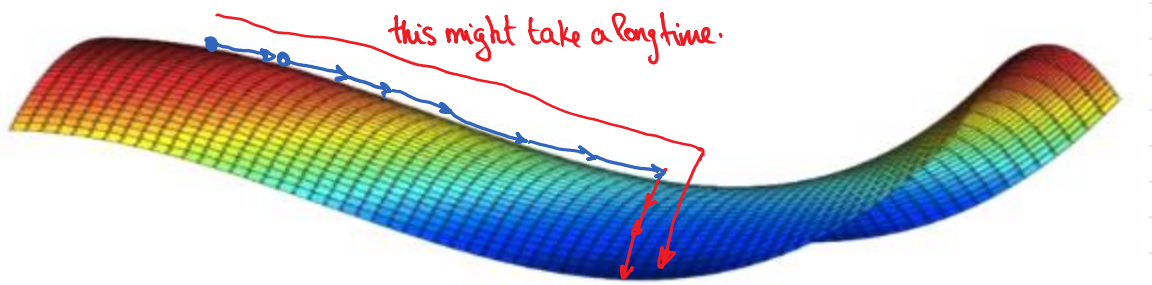
! As long as we are training a relatively high dimensional network (w/ lots of parameters) it is highly unlikely to get stuck in a local optima.

If local optima are not a problem then what is?

LOCAL PLATEAUS.

Plateaus are local maxima w/ gradients close to 0.

It takes a long time for the learning to get off the plateau into the saddle.



Screen clipping taken: 2018-07-18 21:19

Momentum & RMS Prop & ADAM help with moving fast off local plateaus.

Week 3 Hyperparameter tuning

Thursday, 19 July, 2018 17:39

Tuning process

Thursday, 19 July, 2018 22:16

GUIDELINES ON HOW TO ORGANIZE HYPERPARAMETER TUNING:

1. α : learning rate
3. β : momentum term ~ 0.9 a good start
7. $\beta_1, \beta_2, \varepsilon$: ADAM parameters (VERY RARELY TUNED)
   0.9 0.999 10^{-8}
5. # layers
4. # of hidden units
6. learning rate decay
2. mini-batch size.

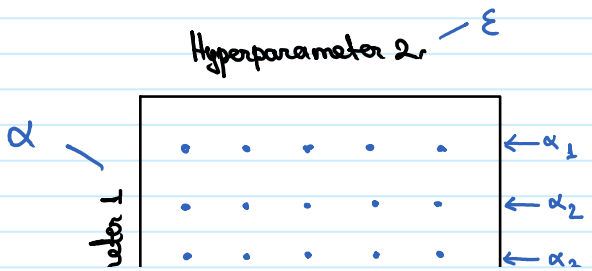
- tuned most often
- tuned often
- tuned less often
- rarely tuned

HOW DO WE SELECT WHAT SET OF VALUES TO EXPLORE?

CLASSICAL APPROACH

(sample in a grid)

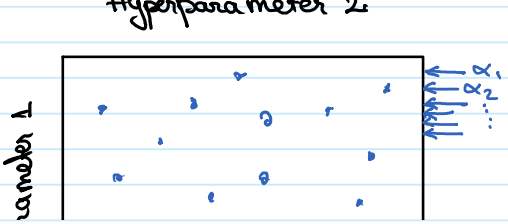
Used to choose parameter space and then sample methodically fa best hyperparameter combo.

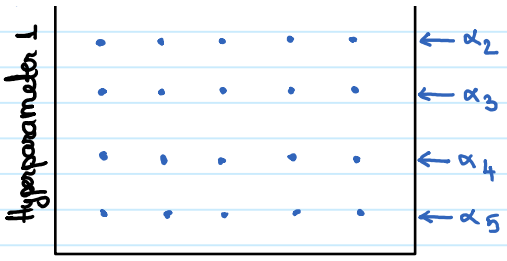


MODERN APPROACH (DEEP LEARNING)

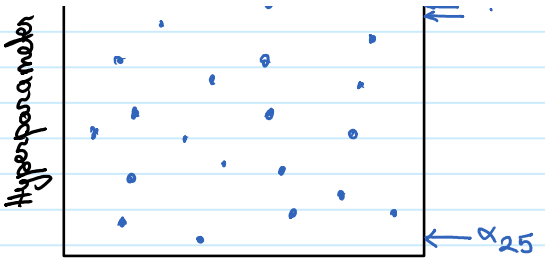
(sample @ random)

Choose parameter space and then sample @ random from this set of points (same # of points as the classical approach).





This practice works OK when the # of hyperparameters was relatively small.



This is better b/c it is difficult to know in advance which hyperparameter are going to be the most important.

Ex: We want to look at hyperparameters α & ε (from ADAM optimization)

↳ we kind of know that ε doesn't have much effect on learning rate.

For this we train 25 models.

- CLASSICAL APPROACH (Grid Sampling)

  Using the "CLASSICAL" approach we sample 5 values of α & 5 of ε.

  α adds a lot of information – we observe trends in the learning rate dependent on values of α

  BUT We only look at 5 values of α in our 25 training models

  ε adds very little information – the learning rate is constant regardless of ε's values.

- MODERN APPROACH (Random Sampling)

  Same observation as in classical approach:

  α adds a lot of information

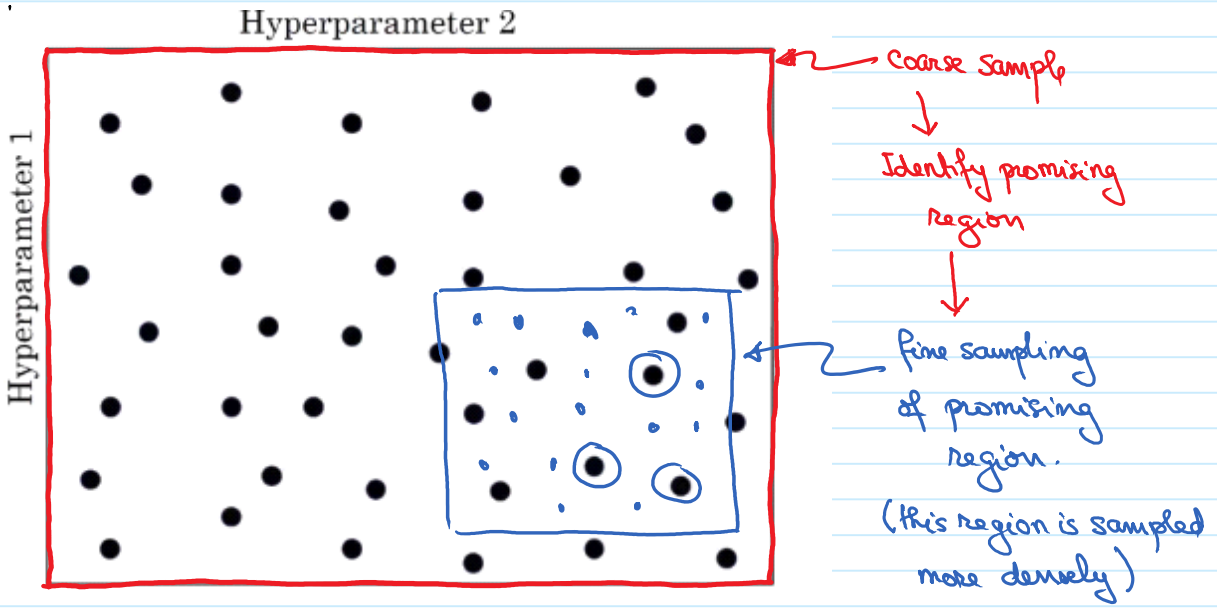
  ε adds little information

  BUT this time we sampled 25 different values of α in our 25 training models.

This random sampling (especially in high dimensional hyperparameter

This random sampling (especially in high dimensional hyperparameter spaces where we do not know in advance which ones are the important hyperparameters) ensures we are more richly exploring the set of possible values for potentially important hyperparameters.

COARSE TO FINE SAMPLING SCHEME:



Screen clipping taken: 2018-07-19 18:34

Both the coarse and fine sampling are done @ random.

The goal of the hyperparameter optimization can be to

- do better on training set objective
- do better on development set
- ⋮

Appropriate hyperparameter scales

Thursday, 19 July, 2018 22:19

Sampling at random doesn't mean sampling uniformly at random over all range of parameters. We need to pick an appropriate scale to sample (which region of hyperparameters do we want to explore?

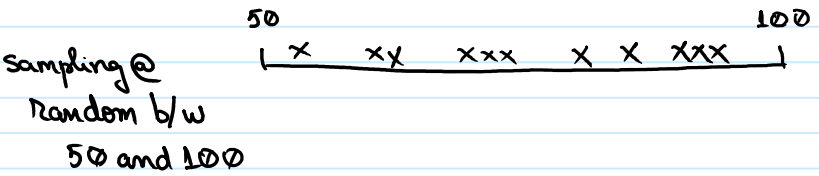
PICKING HYPERPARAMETERS AT RANDOM:

- Option A - # of hidden units

  Let's say we are looking at optimizing the # of hidden units for a given layer l.

  Let's also say we think a good range of values is 50 to 100.

  $n^{[l]} = 50 \ldots 100$

- Option B - # of layers.

  Let's say we want to evaluate how many layers our network should use

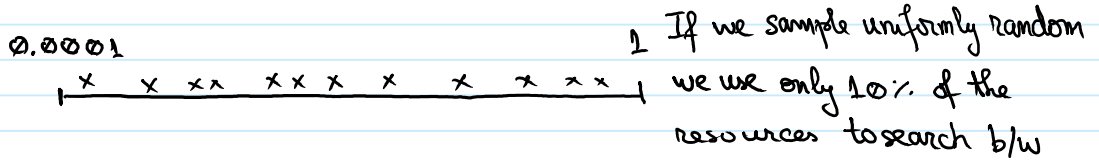
  i.e. determine optimal L

  Let's say we determine the right # should be b/w 2 & 4.

  In this case a grid search is viable (uniformly random is also a good alternative)

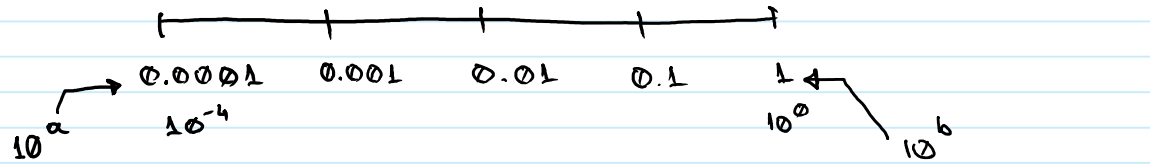
- Option C - learning rate

  Let's say we want to optimize α and a good range is: 0.0001 to 1

0.0001 & 0.001

It is better to search using a logarithmic scale.



$a = \log_{10}(0.0001) \Rightarrow a = -4$

$b = \log_{10}(1) \Rightarrow b = 0$

Python implementation:

```
r = -4 * np.random.rand()   <- r ∈ [-4 ; 0]
α = 10^r                    <- α 10^-4 ... 10^0
```

In general random sampling on log scale:

$10^a \dots 10^b$ $r \in [a, b]$ and then $\alpha = 10^r$

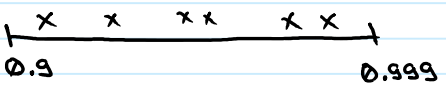
- Option 3 – β (exponential weighted averages)

Let's say we want to optimize β and determined a good range is

$\beta = 0.9 \dots 0.999$

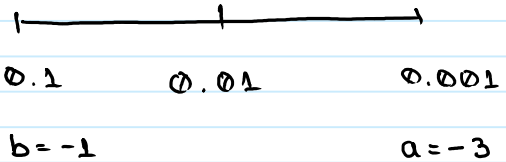
averaging over the last: 10 values ... 1000 values.

Similar to learning rate it doesn't make sense to sample randomly on the linear scale.



We actually can explore the range of values for $(1-\beta) = 0.1 \dots 0.001$

↑ the weight assigned to the current value which is complementary to the weight assigned to the historical values (β)



Implementation:

$r \in [-3; -1]$

$(1 - \beta) = 10^{-r} \Rightarrow \beta = 1 - 10^{-r}$

$r = -2 * np.random.rand() - 1$

$\beta = 1 - 10^{r}$

This will result in uniformly sampling across a logarithmic scale.

INTUITION FOR WHY NOT TO SAMPLE ON LINEAR SCALE

When β is close to 1 the sensitivity of the results changes even w/ small changes to β ($\frac{1}{1-\beta}$ is very sensitive to small changes the closer β is to 1)

β : 0.9000 changes to 0.9005 → no big deal

average over ~1000 values — meh... — ~1000 values — learning speed not affected much.

β : 0.9990 changes to 0.9995 → this can have a huge impact on

average over 1000 values — wow! — 2000 values. learning speed.

LAST BUT NOT LEAST: For all the scale explorations above

REMEMBER TO ALSO USE COARSE TO FINE SAMPLING !

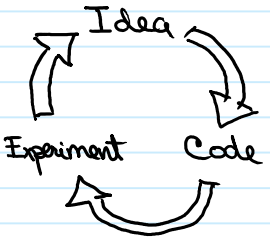
Hyperparameters tuning: Pandas vs Caviar

Friday, 20 July, 2018 16:20

Hyperparameters intuitions from one domain to another domain do not always transfer well. (NLP, Vision, Speech, Advertisments, Logistics ...)

- Intuitions about which hyperparameter values are optimal for a given system get stale over time (as data sets grow, algorithms improve, hardware changes ...)

  → Reevaluate hyperparameter choices occasionally (at least once every 6 months)

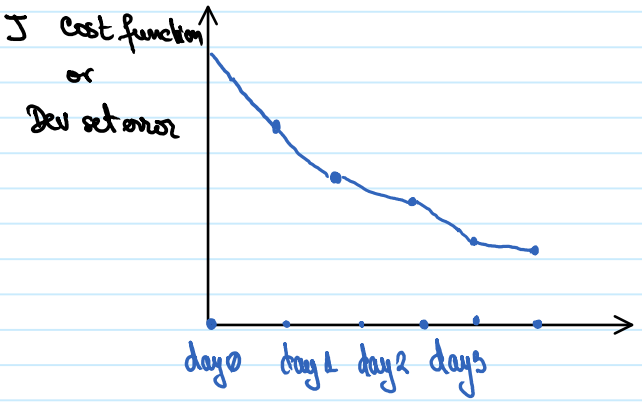


HOW TO SEARCH FOR HYPERPARAMETERS?

- Baby sitting one model (PANDA → put a lot of effort into each model)

  → if we have a huge data set but not a lot of compute available (CPUs / GPUs)

  (i.e. we can only afford to train one model at a time so we will keep watch over the model as it is training.



initialise parameters & start

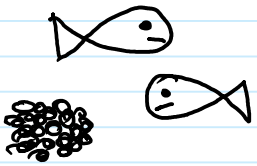
increase learning rate

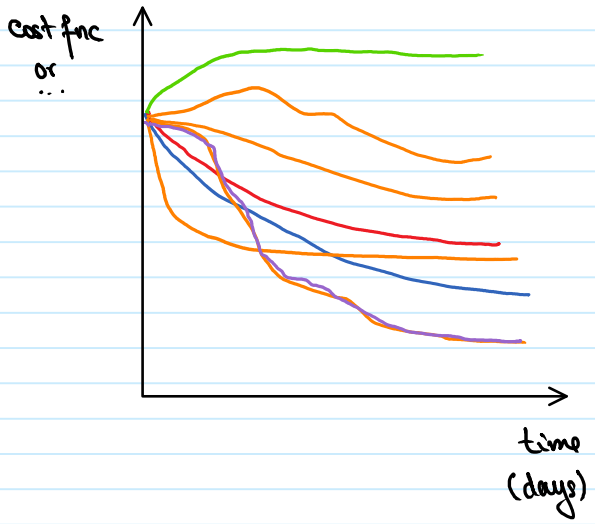
up the momentum rate

decrease learning rate

training

- Training many models in parallel (CAVIAR → lay many eggs and hope some do well)
  - → when there is a lot of compute available.
  - → use different hyperparameter settings for each model and see which one performs the best.
  - → @ the end pick the one that worked best.





Normalizing Activations

Saturday, 21 July, 2018 11:14

BATCH NORMLIZATION ALGORITHM:

→ Makes hyperparameter search easier

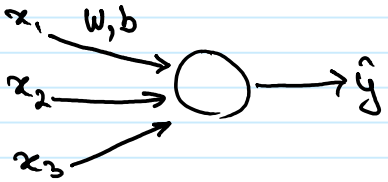
→ Makes network more robust to the choice of hyperparameters
(i.e. NN works well with a larger range of hyperparameters)

→ Allows training of deeper networks.

! This is done by normalizing the μ & σ^2 for the "inputs" in each hidden layer.

NORMALIZING INPUTS SPEEDS UP LEARNING:

- We previously showed that normalizing the input features can speed up learning

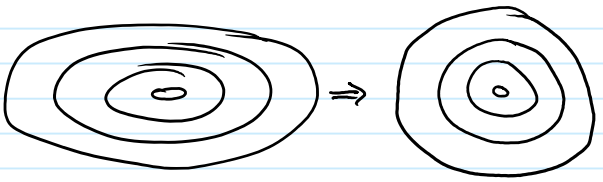


$$\mu = \frac{1}{m}\sum_i x^{(i)}$$
$$X = X - \mu$$

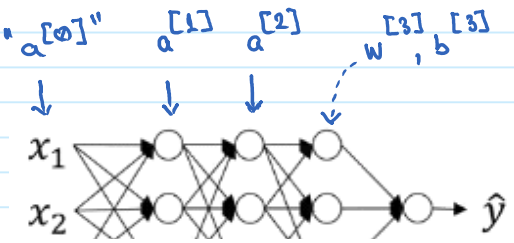
} Normalize the mean

$$\sigma^2 = \frac{1}{m}\sum_i \left(x^{(i)}\right)^2 \quad \leftarrow \text{element wise squaring}$$
$$X = X / \sigma^2$$

} Normalize the variance

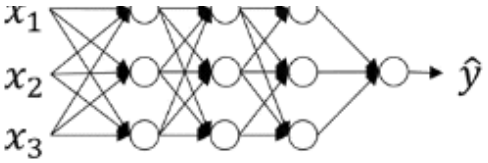


- How about normalizing "the input features" for each layer (i.e. the activations for each layer)?



To optimize $W^{[3]}, b^{[3]}$ faster it helps if $a^{[2]}$ ("input features" for $l=3$) are normalized.

Actually we are going to normalize $Z^{[2]}$ (i.e. before we apply the activation func.)



Screen clipping taken: 2018-07-21 11:24

Actually we are going to normalize $z^{[2]}$ (i.e. before we apply the activation func.)

NOTE: there is some debate on normalizing before or after applying the activation function.

→ in practice z is normalized more.

IMPLEMENTING BATCH NORM:

Given some intermediate values in your NN: $\underbrace{z^{(1)} \dots z^{(m)}}_{z^{[l](i)}}$ (all z values of layer l)

$$\mu = \frac{1}{m} \sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

ε: used for numerical stability in case $\sigma^2 = 0$

At this point all hidden inputs have mean 0 & variance 1.

But we don't want the hidden units to always have $\mu = 0$ & $\sigma^2 = 1$

maybe we want them to have a different distribution.

So we define:

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

γ, β: learnable parameters of the model. similar to w & b.

The effect of γ & β allows us to set $\tilde{z}$ mean to be whatever we want it to be.

i.e. if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$ then $\tilde{z} = \gamma z^{(i)}_{norm} + \beta$

Undoes the initial normalization and we get $\tilde{z}^{(i)} = z^{(i)}$

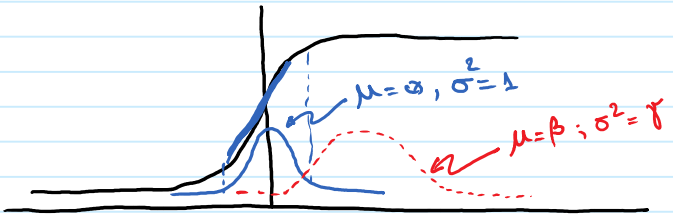
So after we run the normalization we would use:

$\tilde{Z}^{[l](i)}$ in the NN instead of $Z^{[l](i)}$

WHY DO WE NOT ALWAYS WANT $\mu = 0$ & $\sigma^2 = 1$

Even though we want X_{norm} input features to have $\mu = 0$ & $\sigma^2 = 1$ we don't want the same for $Z^{(i)}_{norm}$.

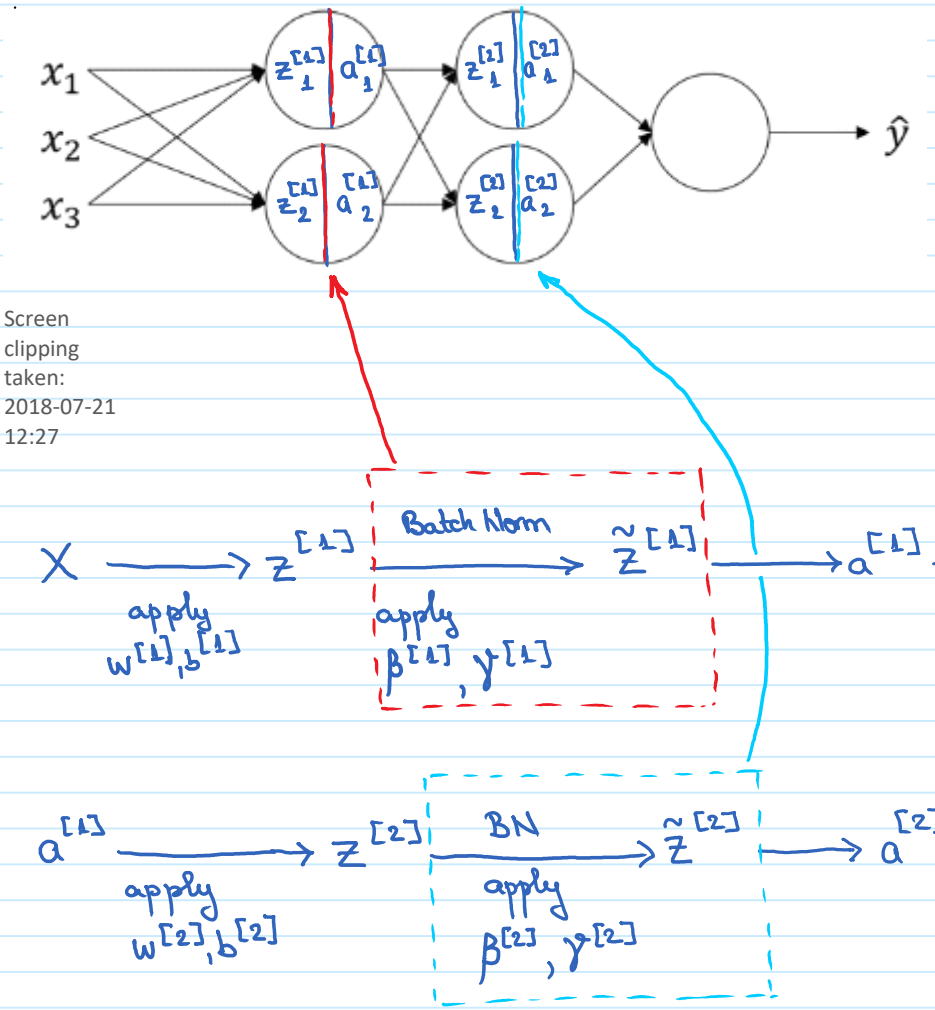
For example given a sigmoid activation function:



we don't want all our values to be clustered around $\mu = 0, \sigma^2 = 1$ where the sigmoid is in its linear regime. We probably want to take advantage of the non-linear regimes of the sigmoid activation function which means $\mu \neq 0$. So w/ γ & β we are controlling the distribution of activation to best optimize learning speed.

Adding Batch Norm to a deep NN

Saturday, 21 July, 2018 12:25



...

The batch norm is inserted b/w computing z & computing a.

So the parameters of the network are

$w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \dots w^{[L]}, b^{[L]}$

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]} \dots \beta^{[L]}, \gamma^{[L]}$

} We then use whichever algorithm we want to update the parameters i.e. Gradient descent :

$\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$

...

Note: the β in Batch Norm has nothing to do with β in ADAM nor with β in Momentum

WORKING WITH MINI-BATCHES

We talked about Batch Norm as if we are training the entire data set at a time (i.e. we are applying the standard vectorized Batch Gradient Descent) in practice we use it with mini-batch Gradient Descent.

$$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow g^{[1]}(\tilde{z}^{[1]}) = a^{[1]} \longrightarrow z^{[2]} \dots$$

$$X^{\{2\}} \longrightarrow z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow \dots$$

BN here uses only the data set examples in mini-batch $\{2\}$

$$X^{\{3\}} \longrightarrow \dots$$

THE DOWNFALL OF B WHEN USING BATCH NORM:

Previously we stated that the parameters are

$$w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$$

Notice that $z^{[l]} = w^{[l]} \cdot a^{[l-1]} + b^{[l]}$

but what Batch Norm does is to

1) normalise $z^{[l]}$ to have $\mu = 0$ & $\sigma^2 = 1$.
2) rescale by β & γ

and this means that the value of $b^{[l]}$ gets subtracted out during the μ subtraction step. (i.e. the zeroing out of the mean)

So when using Batch Normalization we don't use $b^{[l]}$ (i.e. $b^{[l]}$ is set to 0)

Parameters: $w^{[l]}, \beta^{[l]}, \gamma^{[l]}$

$$z^{[l]} = w^{[l]} a^{[l-1]}$$

$$z^{[l]}_{norm}$$

$$\tilde{z}^{[l]}_{norm} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$$

$\beta^{[l]}$ becomes the "new $b^{[l]}$" (it determines the μ of the distribution of activations that get passed on to the next layer)

When running a single example dimensions of

$$z^{[l]} : (n^{[l]}, 1)$$

↑ # of hidden units in layer l.

$$\beta^{[l]}, \gamma^{[l]} : (n^{[l]}, 1)$$

β & γ are used to scale the μ and σ^2 of each hidden unit of the l hidden layer of the NN

$$: (n^{[l]}, m)$$

↑ # of examples

Implementing Gradient Descent Using Batch Norm

Assume we are using mini batch gradient descent:

for $t = 1 \dots$ numMiniBatches

Compute forward prop on $X^{\{t\}}$

In each hidden layer use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

Use back prop to compute for all values of l

$dw^{[l]}, ~~db^{[l]}~~, d\beta^{[l]}, d\gamma^{[l]}$

Update parameters.

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

$$\beta^{[l]} := \beta^{[l]} - \alpha \, d\beta^{[l]}$$

} also works w/ Momentum, RMS Prop

$\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$
$\gamma^{[l]} \ldots$

} RMS Prop
Adam.

In most deep learning frameworks batch normalization does not need to be implemented by hand. For example in Tensor Flow one can call:

tf.nn.batch-normalization

Why does Batch Norm work?

Saturday, 21 July, 2018 23:18

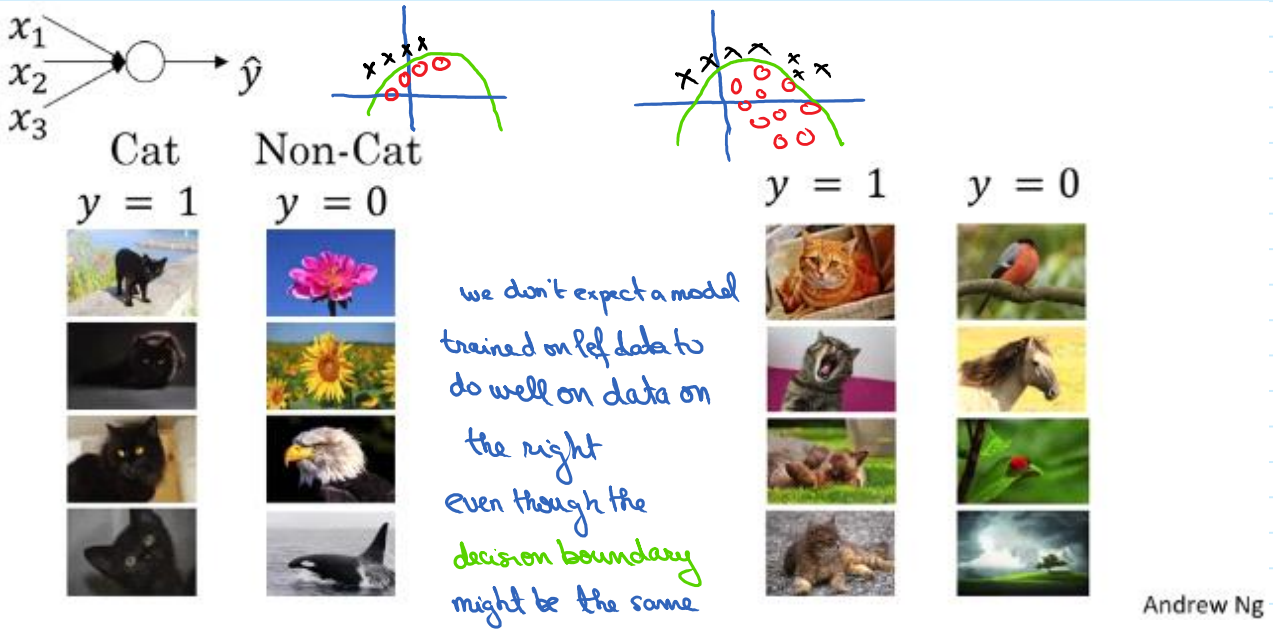
Similar to how normalizing input features works to speed up learning, normalizing the inputs for all hidden layers speeds up learning. There are actually some more intuitions as to why Batch Norm works.

- DECOUPLES WEIGHTS IN DEEPER LAYERS FROM EARLIER LAYERS:

Makes weights in deeper layers more robust to changes in earlier layers.

Say we trained a neural net to detect cats but we only used black cats in our training set.

x - negative example
O - positive example
— decision boundary



Screen clipping taken: 2018-07-22 11:22

The idea of the data distribution changing is known as the COVARIANT SHIFT

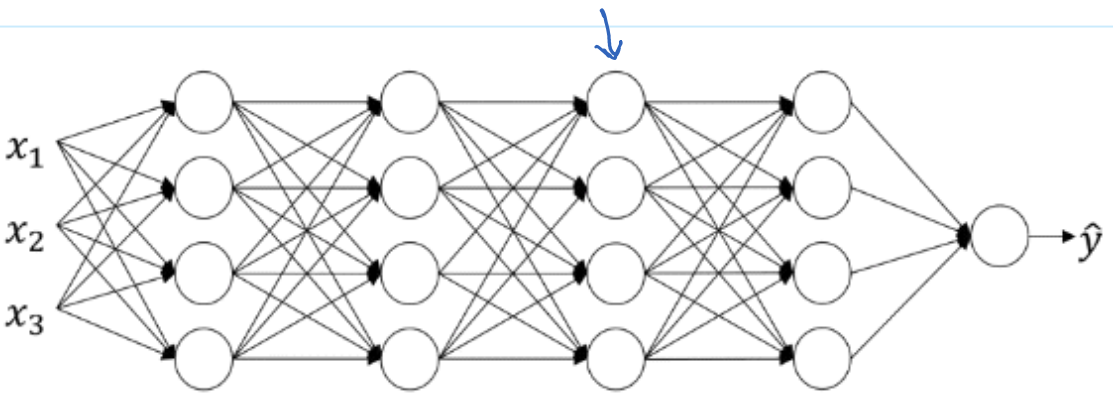
→ If we learn on a distribution a x to y mapping & the distribution changes then we might need to retain our learning algorithm.

→ This is true even if the ground truth function mapping X to Y remains unchanged.

→ The problem becomes worse if both the training distribution & ground truth mapping function change.

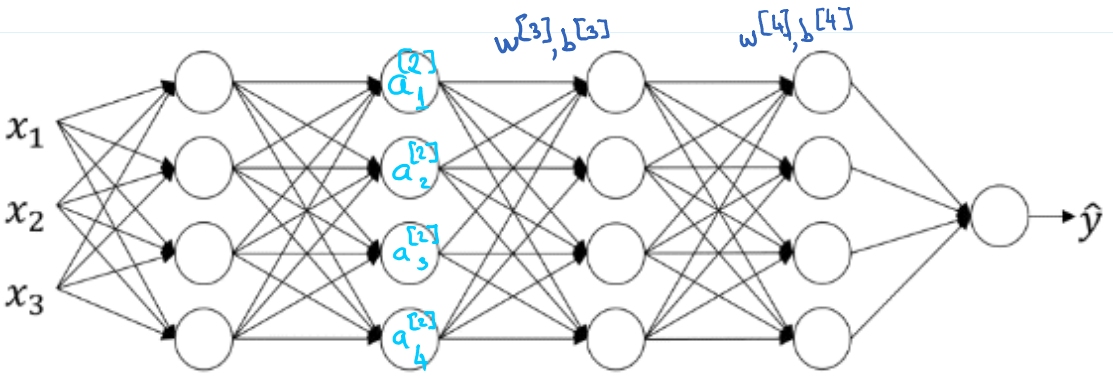
How does covariant shift apply to a deep network?

Consider a NN like below and let's look @ the learning process from the perspective of the 3rd hidden layer.



Screen clipping taken: 2018-07-22 11:34

It needs to learn $w^{[3]}, b^{[3]}$ from getting some training data from earlier layers and minimizing the distance b/w output $\hat{y}$ and ground truth y.

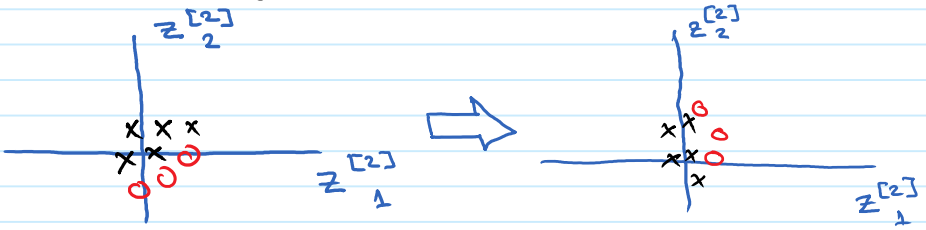


However as layers upstream of layer 3 are learning the "input features"

for layer 3 are changing constantly → they are suffering from the problem of COVARIANT SHIFT.

What Batch Norm does is to minimize the amount the distribution of "input features" for layer 3 shift around.

Below we plot only 2 of the 4 "input features" distributions for layer 3



even though the distribution have changed Batch Norm ensures that the μ and σ^2 stay the same.

i.e $\mu = 0$, $\sigma^2 = 1$ (in general $\mu = \beta^{[2]}$, $\sigma^2 = \gamma^{[2]}$)

And this limits the amount by which updating the parameters in earlier layers can affect the distribution of values layer 3 uses to train on.

So Batch Norm reduces the problem of input values changing. It stabilizes the input data for each layer reducing the amount of change each layer needs to adapt to.

(i.e. it weakens the coupling between layers which speeds up learning as each layer "dithers" less on its learning path).

- BATCH NORM AS REGULARIZATION METHOD:

When using mini-batch along batch normalization, batch normalization has a regularization effect (similar to dropout)

has a regularization effect (similar to dropout)

INTUITION:

- Each mini-batch is scaled by the mean/variance computed only on the training data in the mini-batch $X^{\{t\}}$.
- These μ & σ^2 are intrinsically more noisy than μ & σ^2 of the entire data set. (we are using less data)
- This adds noise to the values of $z^{[l]}$ for that mini-batch, which is similar to how dropout adds noise to the activations on each layer.
- This causes a slight regularization effect.

NOTE: Dropout adds multiplicative noise (i.e. each activation is multiplied by 0 or 1.

Batch Norm adds multiplicative noise (i.e. we are scaling by σ^2)
additive noise (subtracting the μ)

- By adding noise to the hidden units in a given layer Batch Norm is forcing the downstream units not to rely to heavily on a few units.

Because we don't add a lot of noise this doesn't have a huge regularization effect and it should not be relied on as a regularization technique.

→ OBSERVATION: The larger the mini-batch size the less noise in μ and σ^2 and the smaller the regularization effect.

Batch Norm at test time

Sunday, 22 July, 2018 12:15

At training time Batch Norm processes data one mini-batch at a time. However at test time we most often won't be using mini-batches (we'll probably test on single examples at a time). We need to approach testing slightly differently to ensure we can interpret the results properly.

Recall that during training we use the following equations:

m denotes # of examples in one mini-batch not in the entire training set.

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$ (scale by mean & standard variance w/ ε added for numerical stability)

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$ (rescale by γ & β)

Screen clipping taken: 2018-07-22 12:20

Notice that μ & σ^2 are calculated over the entire mini-batch during training.

At test time if we have only one example taking the mean & standard variance of one example doesn't make sense.

SOLUTION: We need to come up with a different μ & σ^2 for testing.

The new μ & σ^2 used for testing are exponentially weighted average estimates across all mini-batches.

Let's say we are on layer l and we are going through mini-batches

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}} \ldots$ (and associated Y values)

$\mu^{\{1\}[l]}$ & $\sigma^{\{1\}[l]}$ $\quad$ $\mu^{\{2\}[l]}$ & $\sigma^{\{2\}[l]}$ $\quad$ $\mu^{\{3\}[l]}$ & $\sigma^{\{3\}[l]}$

Use exponential weighted average accross $\{t\}$ to get the estimated μ for the z for layer l to be used at test time

Similarly w/ σ^2.

So as we train the NN accross all minibatches we are keeping a running average of μ & σ^2 (μ_{EWA} & σ^2_{EWA})

At test time z_{norm} becomes:

$$z_{norm} = \frac{z - \mu_{EWA}}{\sqrt{\sigma^2_{EWA} + \varepsilon}} \Rightarrow \tilde{z} = \gamma z_{norm} + \beta$$

We could instead estimate the μ & σ^2 by running the entire training data but EWA is fast and not memory intensive and in practice it works well enough.

Multi-class classification

Sunday, 22 July, 2018 13:05

Until now we looked only at binary classification NN (is it cat or not cat).

SOFTMAX Regression is a generalization of logistic regression that allows us to make predictions when trying to classify one of multiple classes

RECOGNIZING CATS, DOGS AND BABY CHICKS:

3 1 2 0 3 2 0 1

Screen clipping taken: 2018-07-22 13:11

Class 1 - cat 3 - baby chick

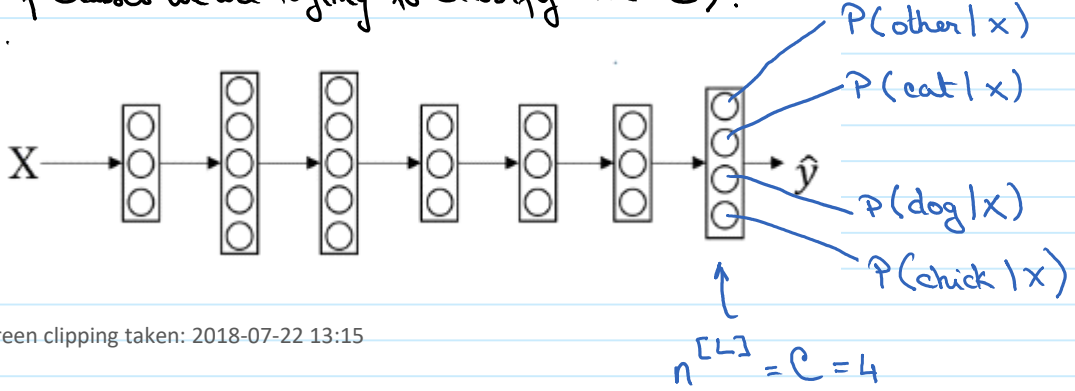
2 - dog 0 - other (none-of-the-above)

Notation: C : # of classes (including other) that we are trying to classify.

C = 4

the indexes for classes are : 0, 1, ... C-1

We are going to build a NN where the output layer has 4 output units. (the # of classes we are trying to classify i.e. C).



Screen clipping taken: 2018-07-22 13:15

$n^{[L]} = C = 4$

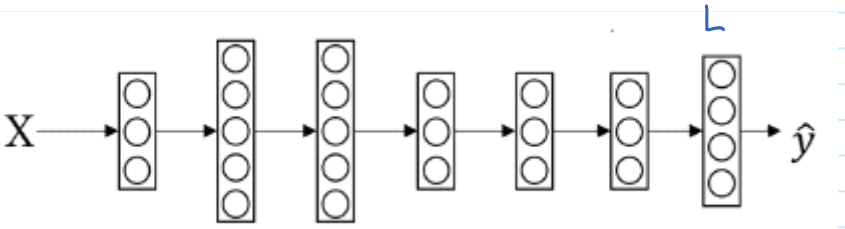
so $\hat{y}$ is a $(4, 1)$ vector.

Because all probabilities are given X they should sum up to 1.

$$P(\text{other}|x) + P(\text{cat}|x) + P(\text{dog}|x) + P(\text{chick}|x) = 1$$

The standard way to get a NN to output $\hat{y}$ is to use a SOFTMAX Layer in the output layer to generate these outputs.

SOFTMAX LAYER:



$$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

now we need to apply the SoftMax activation function.

$t = e^{z^{[L]}}$ (applied element wise: i.e. z is a $(4, 1)$ dimensional vector so t is also a $(4, 1)$ vector).

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{C=4} t_i}$$

($a^{[L]}$ is $e^{z^{[L]}}$ normalised so it sums up to 1)

$a^{[L]} = \hat{y}$, $(4, 1)$

$a^{[L]}$ is also $a(4, 1)$ vector

$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

$a^{[L]} = g^{[L]}(z^{[L]})$ ← this activation function is uncommon as it takes $a(4, 1)$ dimension vector as

input and outputs a(4,1) vector.

previously activation functions took as inputs single values (one real #) as inputs and output single values.

EXAMPLE:

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \Rightarrow t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \Rightarrow \sum_{j=1}^{4} t_i = 176.3$$

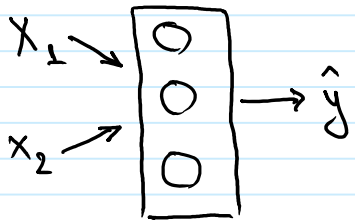
$$a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} \frac{e^5}{176.3} \\ \frac{e^2}{176.3} \\ \frac{e^{-1}}{176.3} \\ \frac{e^3}{176.3} \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

← given x we have a 0.842 chance to classify it as class 0.

Softmax examples

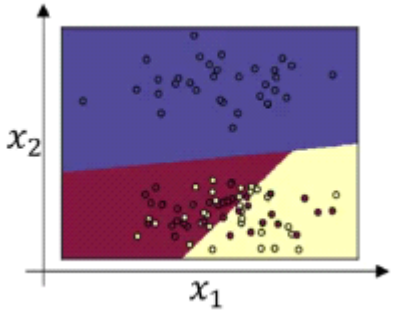
Sunday, 22 July, 2018 13:36

An example of a NN with a single softmax layer (and no other hidden layers)



$$z^{[1]} = w^{[1]} x + b^{[1]}$$

$$a^{[1]} = \hat{y} = g(z^{[1]})$$

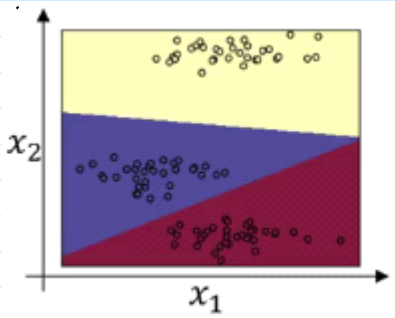


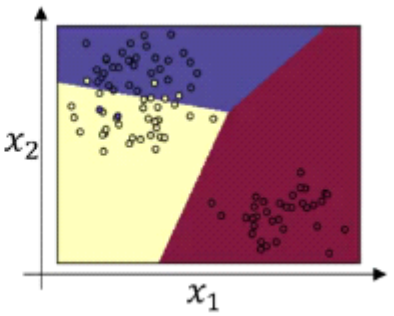
A softmax NN with $C = 3$ (3 output classes)

This is a generalization of logistic regression w/ linear decision boundaries and more than 2 classes.

Screen clipping taken: 2018-07-22 13:40

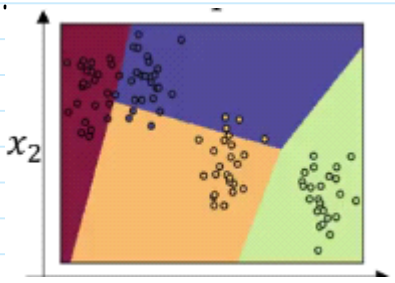
Other examples:





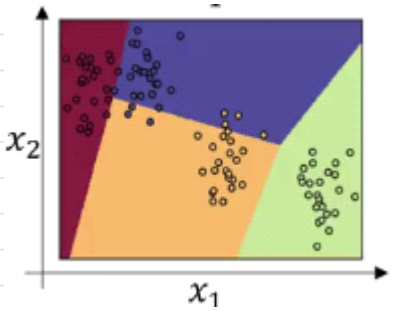
$C = 3$

Screen clipping taken: 2018-07-22 13:44

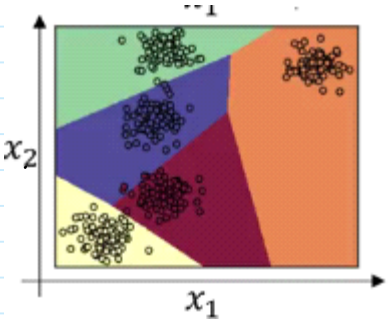


$C = 4$

c = 4

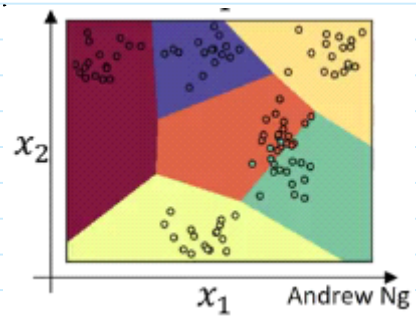


Screen clipping taken: 2018-07-22 13:45



c = 5

Screen clipping taken: 2018-07-22 13:46



c = 6

Screen clipping taken: 2018-07-22 13:47

Training Softmax classifier

Sunday, 22 July, 2018 13:55

Remember the previous section example:

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

"soft max"

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

where $C = 4 \Rightarrow z^{[L]} = (4,1)$ vector

$g^{[L]}$ = Softmax activation func.

"hard max"

Screen clipping taken: 2018-07-22 13:58

"hard max" takes vector z and maps it on $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ — largest element gets an output of 1 and all other elements get an output of 0

Softmax regression generalizes logistic regression to C classes.

If $C = 2$, softmax reduces to logistic regression.

SKETCH PROOF:

$a^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$ → b/c these two numbers have to sum to 1 we don't need to compute both of them so this single # is $\hat{y}$ for logistic regression.

LOSS FUNCTION DEFINITION:

$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ — this is an image of a cat (i.e. class 2 100%)

ground truth

$y_1 = y_3 = y_4 = 0$

$a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ ←

The NN is not doing very well as it assigned this image a 20% ...

as it assigned this image a 20% probability of being a cat.

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{c=4} y_j \log \hat{y}_j$$

make this small

$$= -y_2 \log \hat{y}_2 = -\log \hat{y}_2$$

$y_2 = 1$

makes this small $\Rightarrow$ make $\hat{y}_2$ as big as possible

For more details see maximum likelihood estimation statistics.

This is the loss for a single example. How about the cost J over all training examples?

$$J(w^{[1]}, b^{[1]} \dots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y)$$

minimize cost function J

IMPLEMENTATION DETAIL:

When $c = 4$ (4 classes) y is $(4,1)$ & $\hat{y}$ is $(4,1)$

$$Y = [y^{(1)} \; y^{(2)} \dots y^{(m)}]$$

$$= \begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$$

cat dog other

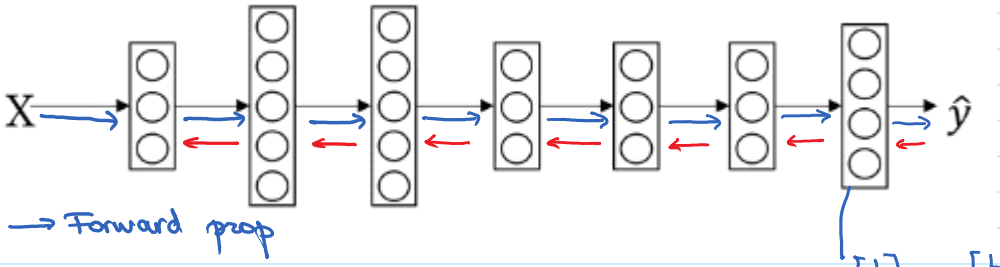
so Y is a $(4, m)$ dimensional matrix.

Similarly.

$$\hat{Y} = [\hat{y}^{(1)} \; \hat{y}^{(2)} \dots \hat{y}^{(m)}]$$

$$= \begin{bmatrix} 0.3 & \dots \\ 0.2 & \\ 0.1 & \\ 0.4 & \end{bmatrix}$$ and $\hat{y}$ is also a $(4, m)$ dimensional matrix.

GRADIENT DESCENT w/ SOFTMAX OUTPUT LAYER:



Screen clipping taken: 2018-07-22 14:22

$z^{[L]} \rightarrow a^{[L]} = \hat{y} \rightarrow \mathcal{L}(\hat{y}, y)$

$(4,1)$

Back prop: $dz^{[L]} = \hat{y} - y$

$(4,1) \quad (4,1) \quad (4,1)$

$dz^{[L]} = \frac{\partial J}{\partial z^{[L]}}$

In most programming frameworks as long as we specify the forward prop correctly the framework will determine how to do the back prop.

Deep Learning frameworks

Saturday, 28 July, 2018 11:42

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

These frameworks are evolving rapidly so it's more important to know how to sort them than choosing one and sticking with it

Screen clipping taken: 2018-07-28 11:44

Choosing deep learning frameworks

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)

open source the software and then as the framework is adopted it is slowly being closed off.

Screen clipping taken: 2018-07-28 11:47

TensorFlow

Saturday, 28 July, 2018 11:49

GOAL: Learn the basic structure of a TensorFlow program.

MOTIVATING PROBLEM

We have a cost function given by $J(w) = w^2 - 10w + 25$

same as $(w-5)^2$

which means $w = 5$ is the value that minimizes J

Let's see how we can implement something in Tensorflow that minimizes this.

→ this would be similar with being able to minimize $J(w, b)$

```
In [1]: import numpy as np
        import tensorflow as tf
```

```
In [5]: w = tf.Variable(0,dtype=tf.float32)
        #cost = tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
        cost = w**2 - 10*w + 25
        train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

        init = tf.global_variables_initializer()
        session = tf.Session()
        session.run(init)
        print(session.run(w))
```

```
0.0
```

```
In [6]: session.run(train)
        print(session.run(w))
```

```
0.1
```

```
In [7]: for i in range(1000):
            session.run(train)
        print(session.run(w))
```

```
4.99999
```

Screen clipping taken: 2018-07-28 12:43

Code example

```
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))

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

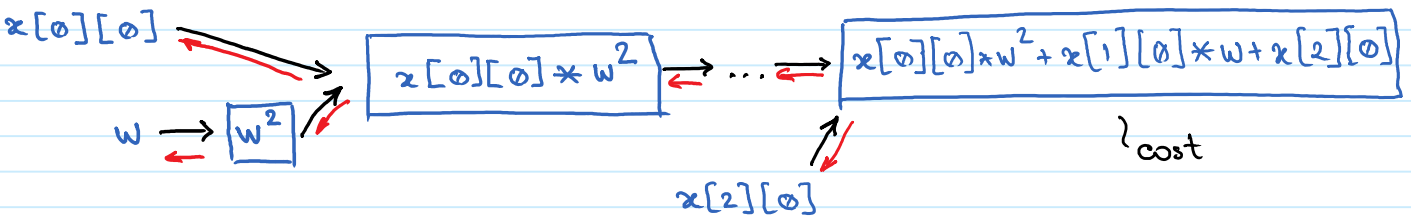
Screen clipping taken: 2018-07-28 12:47

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

Screen clipping taken: 2018-07-28 12:48

At the core of a tensor flow program is the definition of a cost & then Tensor Flow figures out the derivatives (gradients) needed to minimize that cost.

Tensor flow builds a [Computation Graph]



So the user defines the FORWARD PROPAGATION and TensorFlow builds the BACK PROPAGATION.

NOTE: Tensor Flow documentation uses a slightly different Computation Graph

model