Improving Deep NNs

Hyperparameters

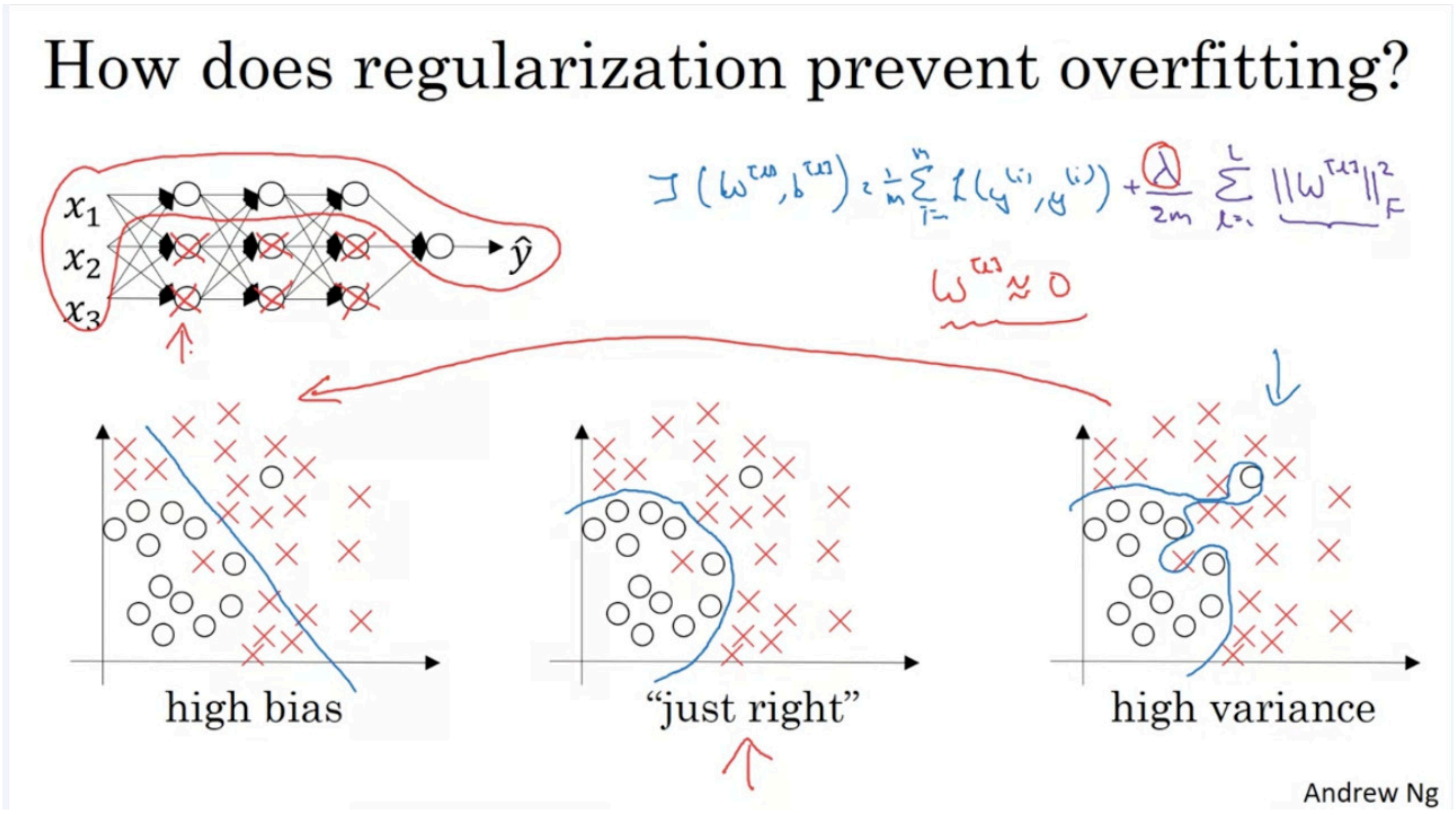
learning rate a
momentum b
how to pick each hyperparameter

train / dev(cross val) / test set
70-30,60-20-20
in the big data era, it is enough to leave it just big enough to see the results.
10m data, 10k is enough - %1

Bias and Variance
high bias -> underfitting
high variance -> overfitting

Similar things with the ML notes, may re-check
What to do, high bias/variance...

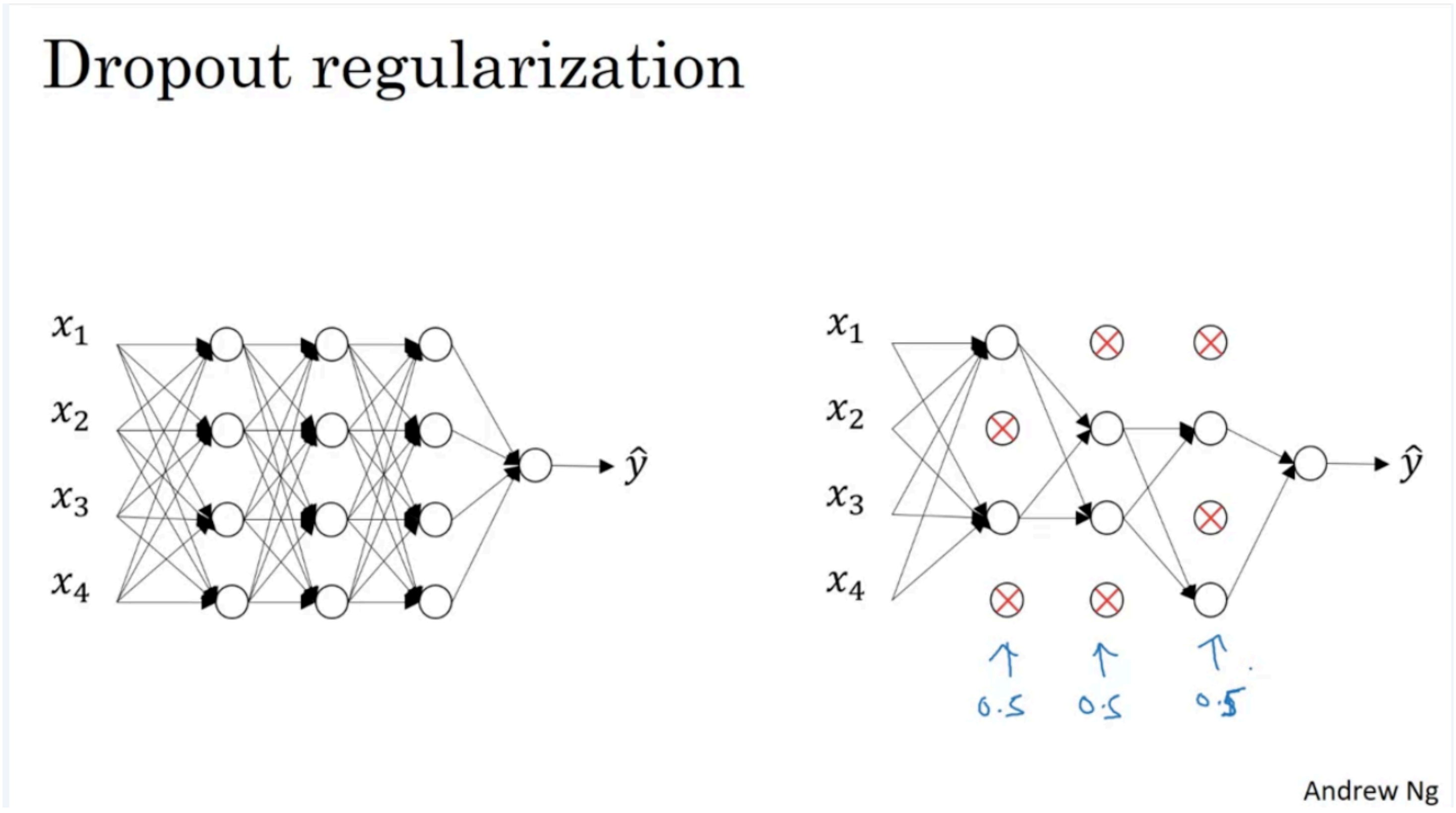
Regularization
prevent overfitting
Why does it work?



How does regularization prevent overfitting?
x_1
x_2
x_3
$\hat{y}$
$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$
$w^{[l]} \approx 0$
high bias
"just right"
high variance
Andrew Ng

L2 Regularization

Dropout Regularization



multiple ways to implement

inverted dropout

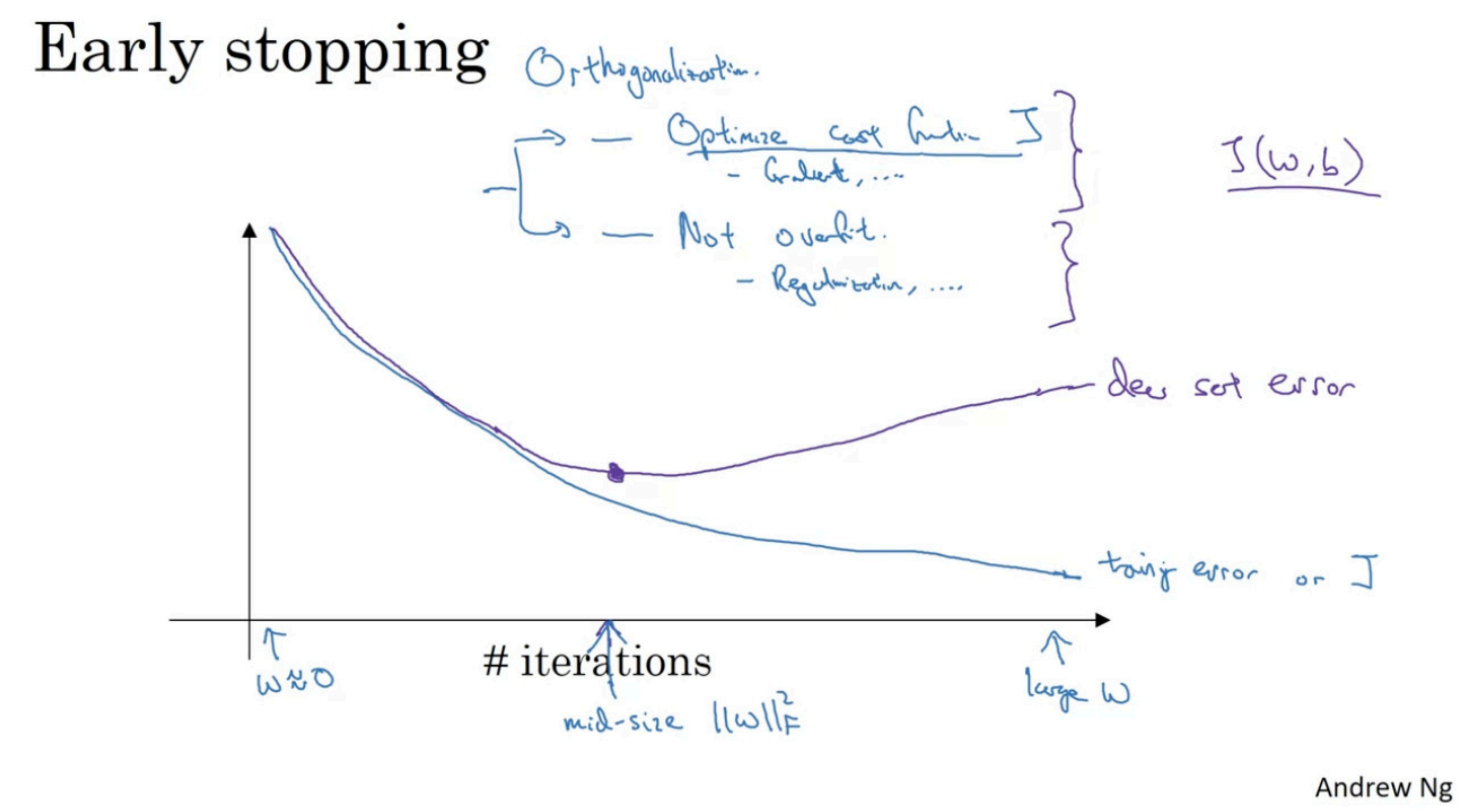
Why does it work?

intuition, cant rely on any one feature, spreads out weights

Other techniques

Getting more data -> expensive

Augmentation techniques

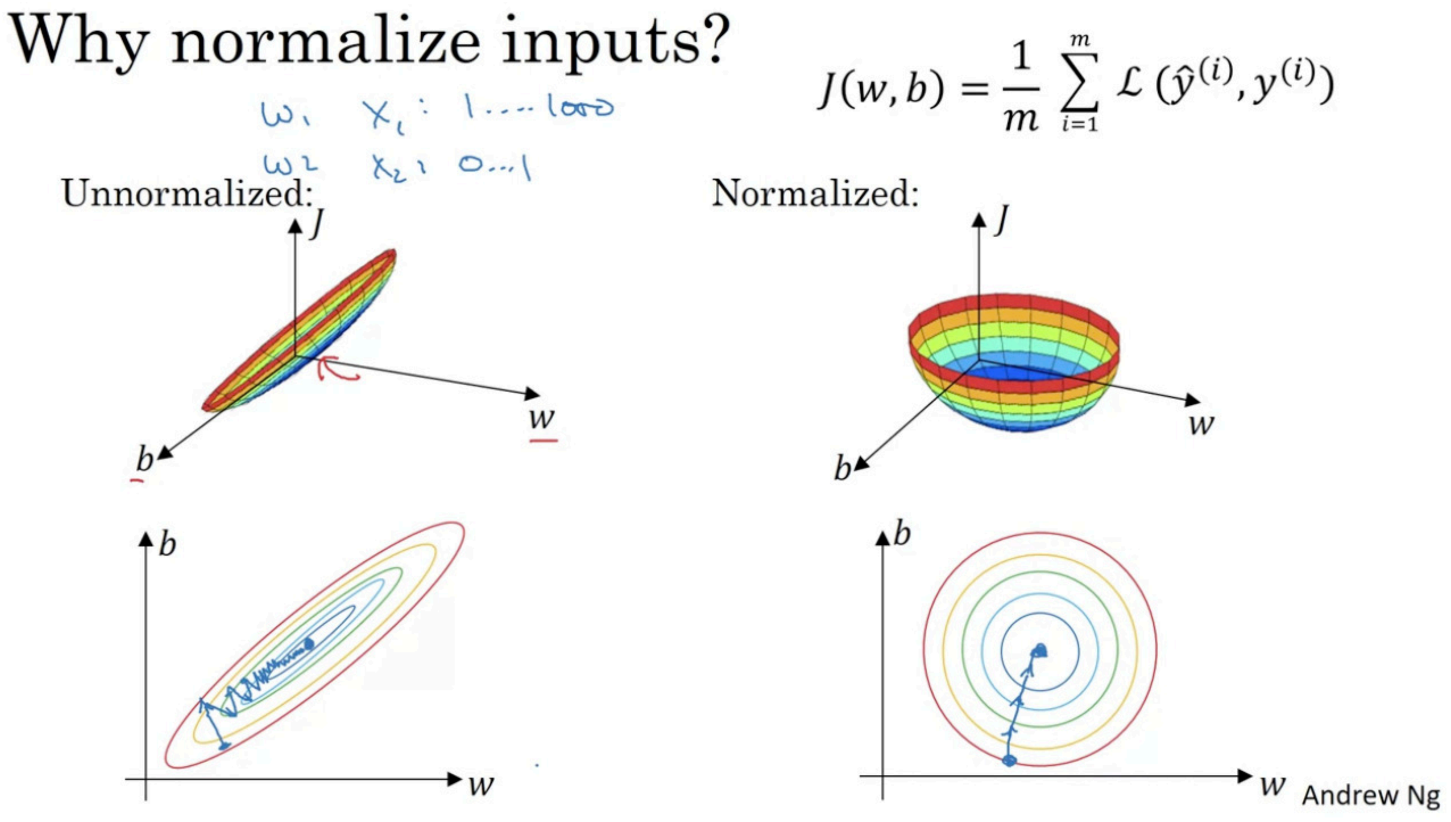


Early stopping

some downsides

Normalize Inputs

Logic behind:



Vanishing/Exploding Gradients

sometimes happens, especially for deep NNs

Vanishing and Exploding Gradients in Deep Neural Networks

• **Problem Overview**: In deep neural networks, gradients can either grow or shrink exponentially as the network depth increases. This leads to two problems:
• **Exploding Gradients**: When weight matrices have values slightly larger than 1, activations grow exponentially with depth, making the network unstable.
• **Vanishing Gradients**: When weight matrices have values slightly smaller than 1, activations shrink exponentially, causing gradients to diminish and slowing down learning.

• **Activation Derivation**:
• Assume a linear activation function , and biases .
• The output is the product of all weight matrices:
• If is proportional to the identity matrix:
• For , activations grow as , leading to exploding gradients.
• For , activations shrink as , leading to vanishing gradients.
• **Impact on Training**:
• Exploding gradients make parameter updates erratic.
• Vanishing gradients result in slow learning due to tiny steps in gradient descent.
• Both issues are exacerbated in very deep networks ().
• **Partial Solution**: Proper weight initialization can mitigate these problems, making training more stable, though it doesn’t fully solve them.

Weight Initialization to Mitigate Vanishing and Exploding Gradients

• **Problem**: Deep networks can suffer from exploding or vanishing gradients due to inappropriate weight initialization.

• **Solution**:
• For **linear activation**, set the variance of weights to (where is the number of inputs to a neuron).
• For **ReLU activation**, use **He initialization**: .
• For **TanH activation**, use **Xavier initialization**: .

• **Implementation**:
• Initialize weights as , where is determined by the activation function and input size.
• **Impact**:
• Proper initialization keeps activations and gradients at reasonable scales.
• Helps stabilize training for deep networks but does not fully eliminate gradient problems.
• **Note**: Weight variance can also be tuned as a hyperparameter, though it is usually less critical than other tuning parameters.

Gradient Approximation and Checking

• **Purpose**: Gradient checking ensures the correctness of your backpropagation implementation by comparing analytical gradients with numerically approximated gradients.

• **Numerical Gradient Approximation**:
• For a function , approximate the gradient using:
• This is a **two-sided difference**, which is more accurate than a one-sided difference ().
• **Example**: If , for , :
Analytical gradient: (approximation error = 0.0001).

• **Advantages of Two-Sided Difference**:
• More accurate: Error is , compared to for one-sided difference.
• Worth the extra computation cost (twice as slow as one-sided).

• **Key Takeaways**:
• Use two-sided differences for gradient checking.
• This helps verify that your backpropagation implementation computes correct gradients by comparing numerical and analytical results.
• A critical step for debugging deep learning models.

Gradient Checking for Debugging Backpropagation

• **Purpose**: Gradient checking verifies the correctness of your backpropagation by comparing analytical gradients with numerically approximated gradients.

- **Steps**:

  1. **Reshape Parameters**:
     - Flatten all parameters () into a single vector .
     - Similarly, flatten all gradients () into a vector with the same dimensions as .
  2. **Numerical Gradient Approximation**:
     - For each parameter :
     - : Unit vector with 1 at position .
     - : Small constant (e.g.,).
  3. **Compare Gradients**:
     - Compute the relative difference between and :
     - If the difference:
     - : Likely correct.
     - : Double-check components.
     - : Possible bug; inspect specific components.

- **Debugging Workflow**:
- Implement forward and backward propagation.
- Run gradient checking.
- If the difference is large, identify problematic components and debug.

- **Key Takeaways**:
- Gradient checking is a valuable debugging tool.
- Use it to ensure gradients are computed correctly before relying on backpropagation.

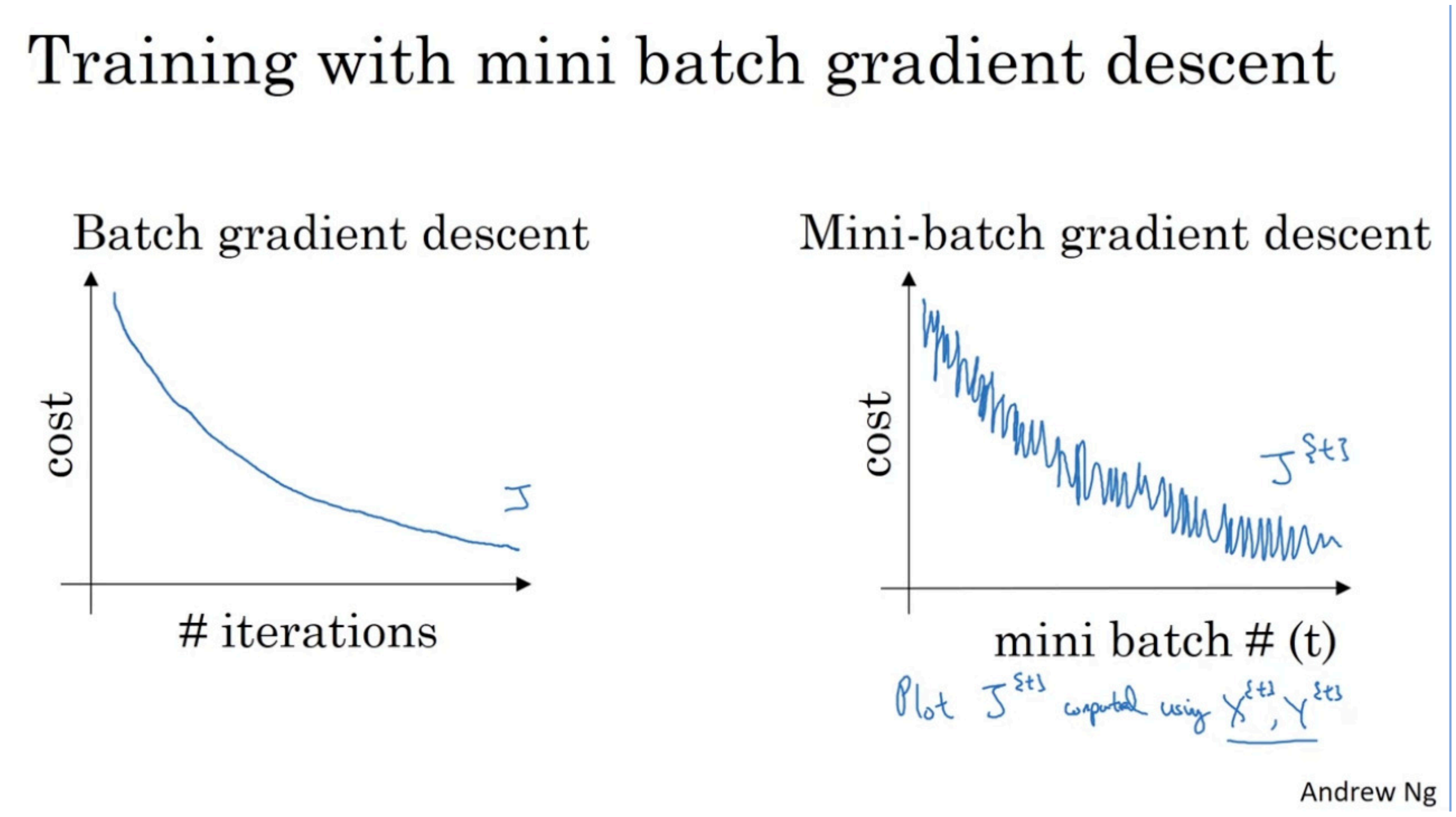
Practical Tips for Gradient Checking

1. **Use for Debugging Only**:
   - Gradient checking is computationally expensive. Use it only for debugging, not during training.
2. **Identify Bugs by Components**:
   - If gradient checking fails, examine individual components (,) to pinpoint the layer or parameter causing discrepancies.
3. **Account for Regularization**:
   - Include regularization terms in the cost function and gradients when performing gradient checking.
4. **Handle Dropout Carefully**:
   - Gradient checking does not work with dropout due to randomness in node elimination.
   - Disable dropout (set keep_prob = 1.0) during gradient checking, then re-enable it for training.
5. **Check Beyond Initialization**:
   - Gradient descent may behave differently as weights and biases grow. Perform gradient checking both at initialization and after training for a few iterations.
6. **Summary**:
   - Gradient checking helps debug backpropagation by ensuring computed gradients match numerical approximations.
   - It's a powerful tool to verify correctness before fine-tuning or using advanced techniques like dropout.

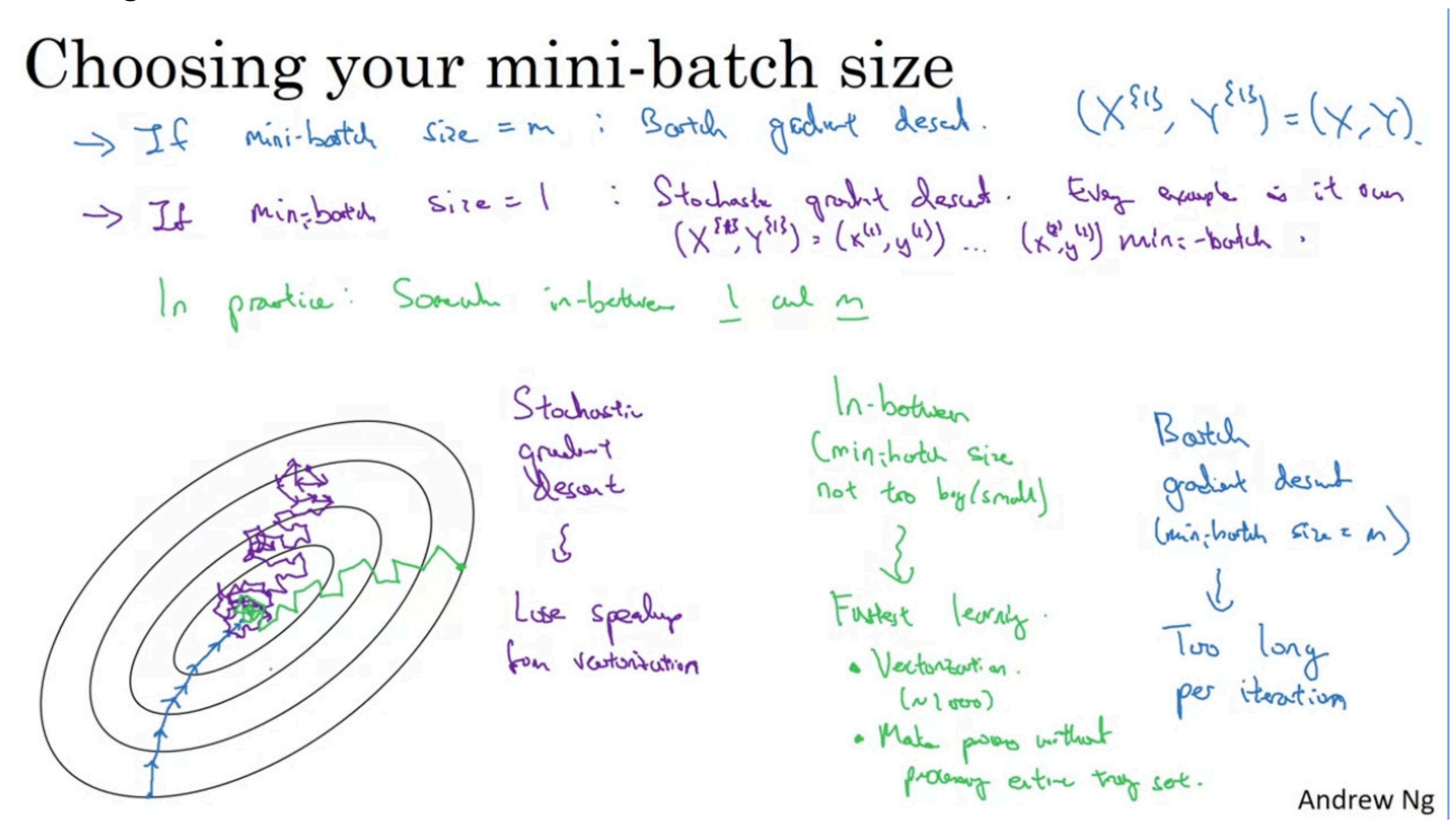
Optimization Algorithms

Batch vs mini-batch Gradient Descent
NN -> Big data (slow)
therefore, need optimization algorithms. (*vectorization*)
train examples: 5m
mini batches: 1k x 5k
mini batch gradient descent: the gd you calculate for a single mini-batch

batch gradient descent -> cost decreases (unless there is a problem - like too big learning rate)
mini-batch gradient descent -> does not always decrease, but the trend is downwards
reason: Xi,Yi is easy mini batch but Xy,Yy is a hard mini batch



Choosing mini-batch size



Choosing your mini-batch size

→ If mini-batch size = m : Batch gradient descent. $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is it own mini-batch. $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)})$ … $(x^{(2)}, y^{(2)})$ mini-batch.

In practice: Somewhere in-between $\underline{1}$ and $\underline{m}$

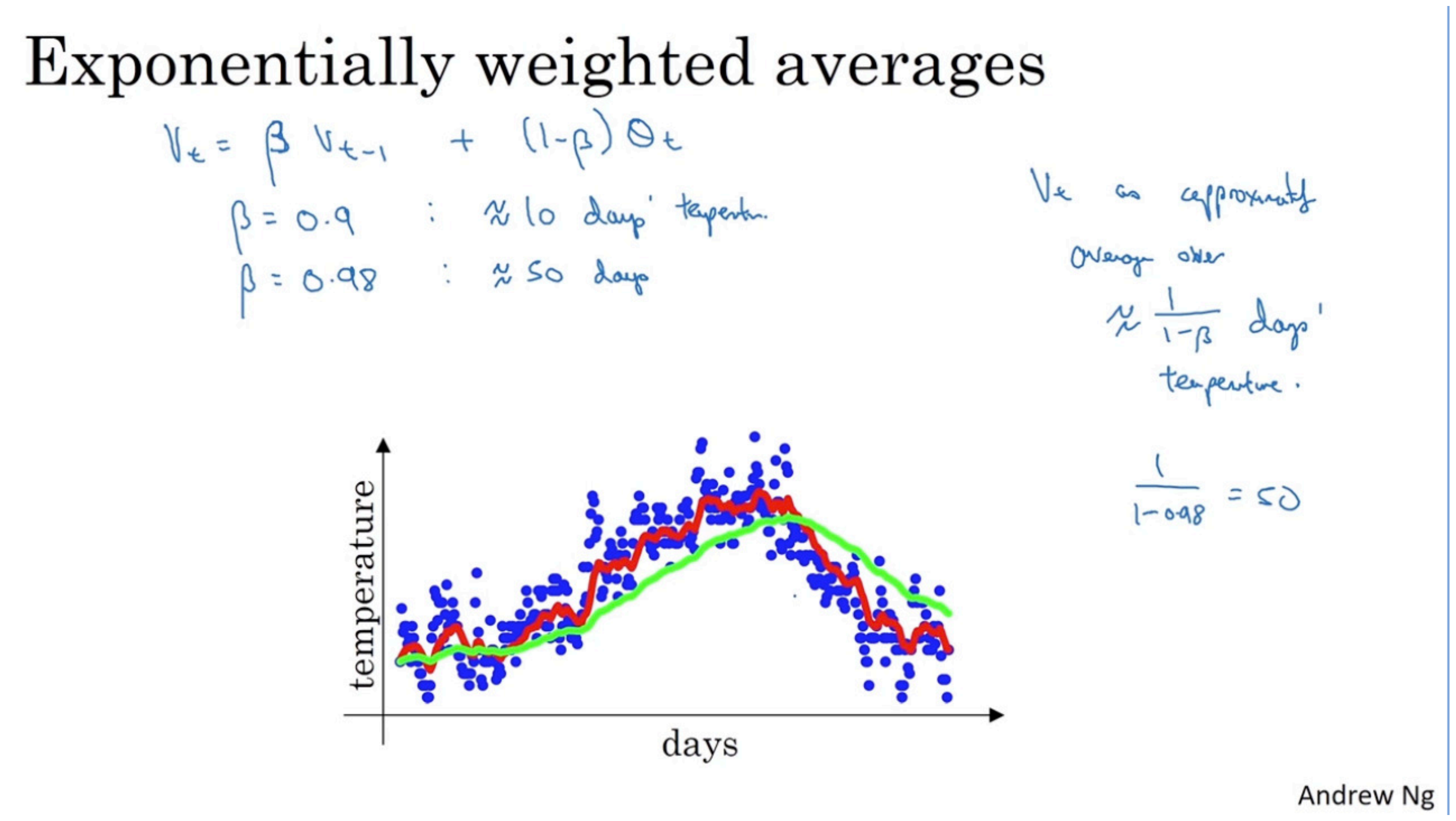
Stochastic gradient descent
⇓
Lose speedup from vectorization

In-between (mini-batch size not too big/small)
⇓
Fastest learning.
- Vectorization. (~1000)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m)
⇓
Too long per iteration

Andrew Ng

Exponentially Weighted (Moving) Averages



Exponentially weighted averages

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

$\beta = 0.9$: ≈ 10 days' temperature.

$\beta = 0.98$: ≈ 50 days

V_t as approximately average over $\approx \frac{1}{1-\beta}$ days' temperature.

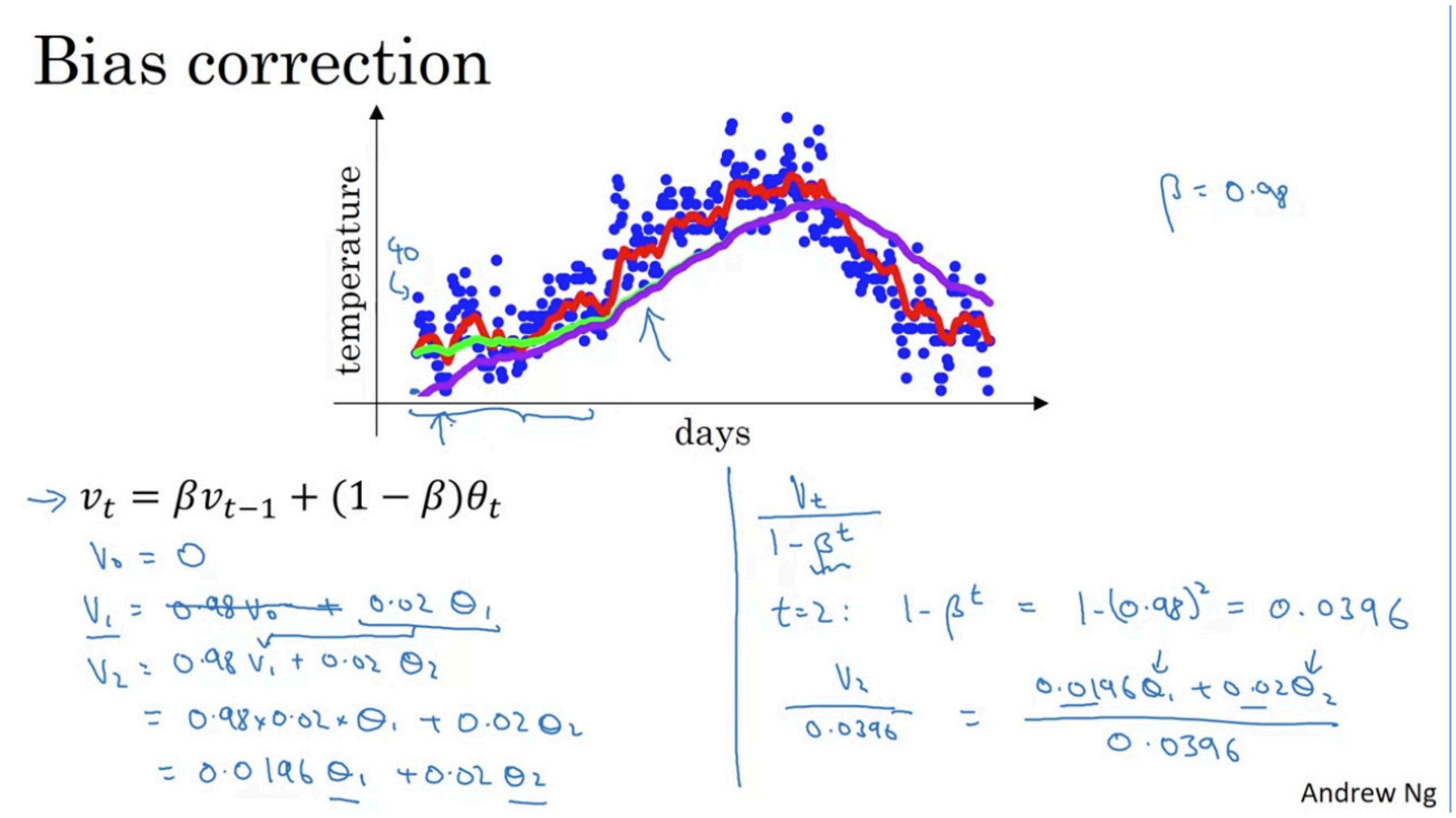
$\frac{1}{1-0.98} = 50$

Andrew Ng

Bias Correction in Exponentially Weighted Averages

in fact does not get green line with above formula, because V0 is zero, thus affecting many of the first elements, curve starts from near 0 value.

thus, use the formula on the right:



Bias correction

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

$V_0 = 0$

$V_1 = \cancel{0.98 V_0 +} \; 0.02\,\theta_1$

$V_2 = 0.98\, V_1 + 0.02\,\theta_2$

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

$= 0.0196\,\underline{\theta_1} + 0.02\,\underline{\theta_2}$

$\frac{V_t}{1-\beta^t}$

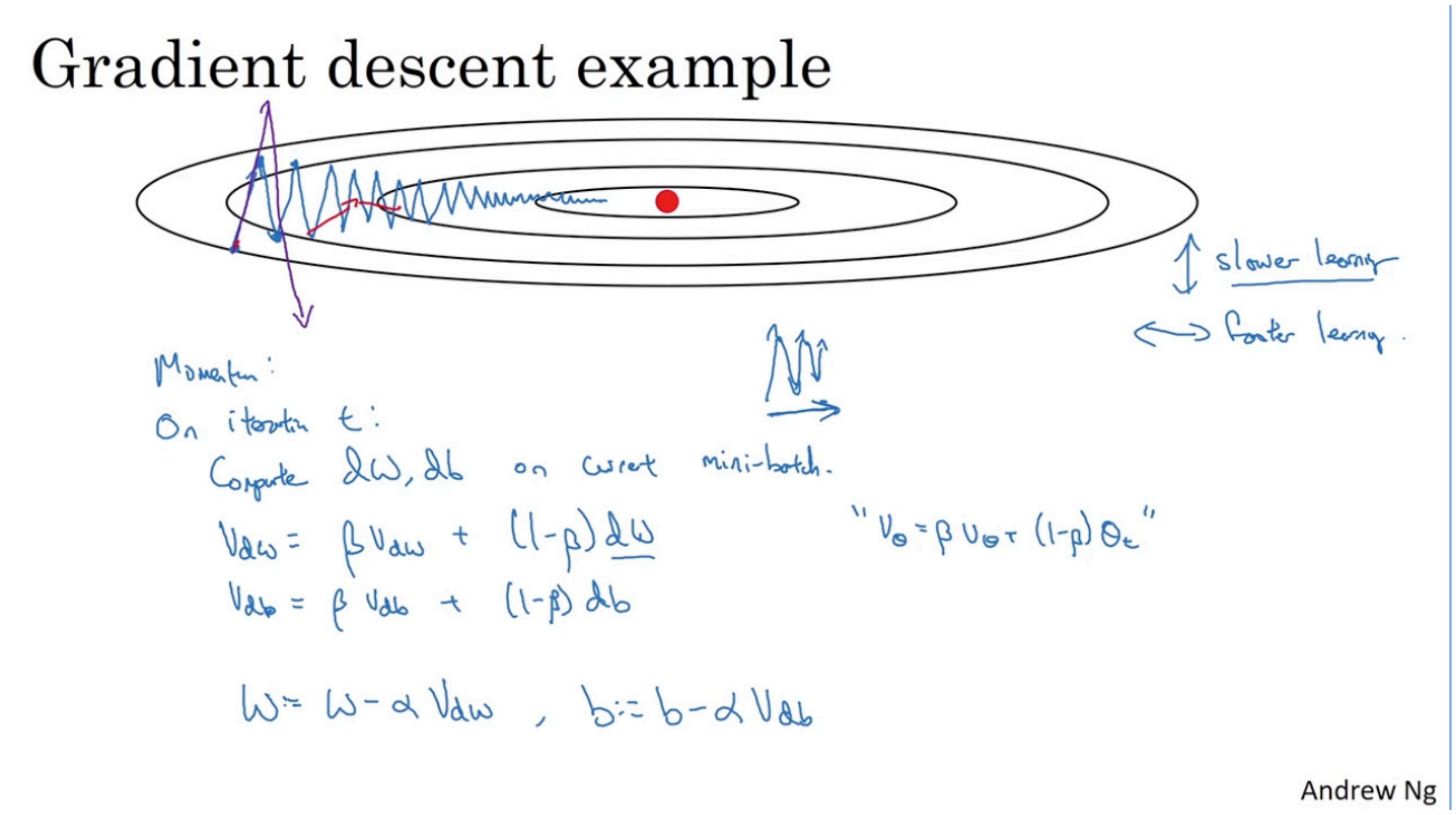
$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

$\frac{V_2}{0.0396} = \frac{0.0196\,\theta_1 + 0.02\,\theta_2}{0.0396}$

Andrew Ng

Gradient Descent with Momentum

almost always works faster then the normal gradient descent



Gradient descent example

Momentum:

On iteration t:

Compute dW, db on current mini-batch.

$V_{dW} = \beta V_{dW} + (1-\beta)\,\underline{dW}$ "$V_\theta = \beta V_\theta + (1-\beta)\,\theta_t$"

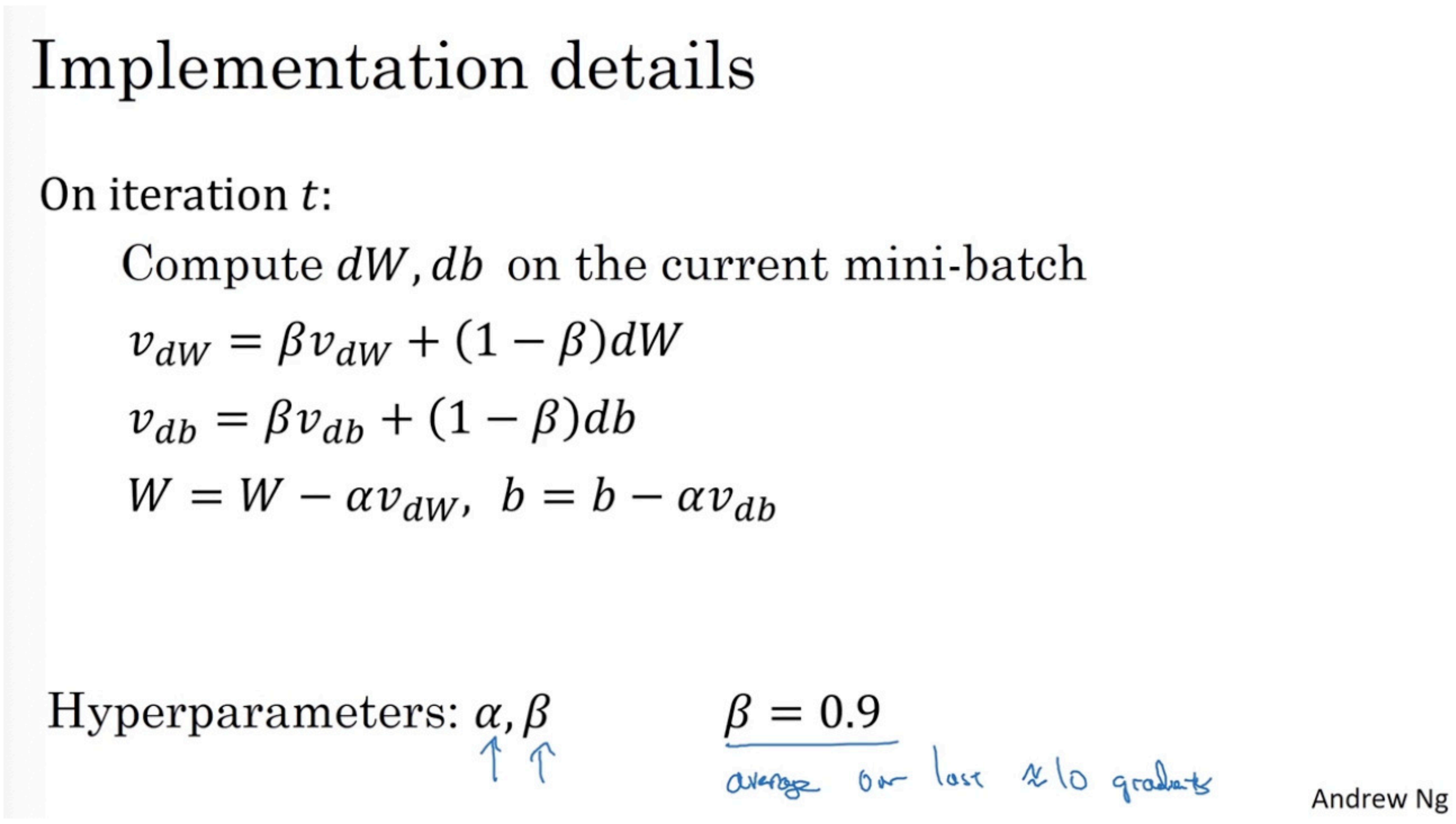
$V_{db} = \beta V_{db} + (1-\beta)\,db$

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

Andrew Ng

common logic

unnecessary details:



Implementation details

On iteration t:

Compute dW, db on the current mini-batch

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

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

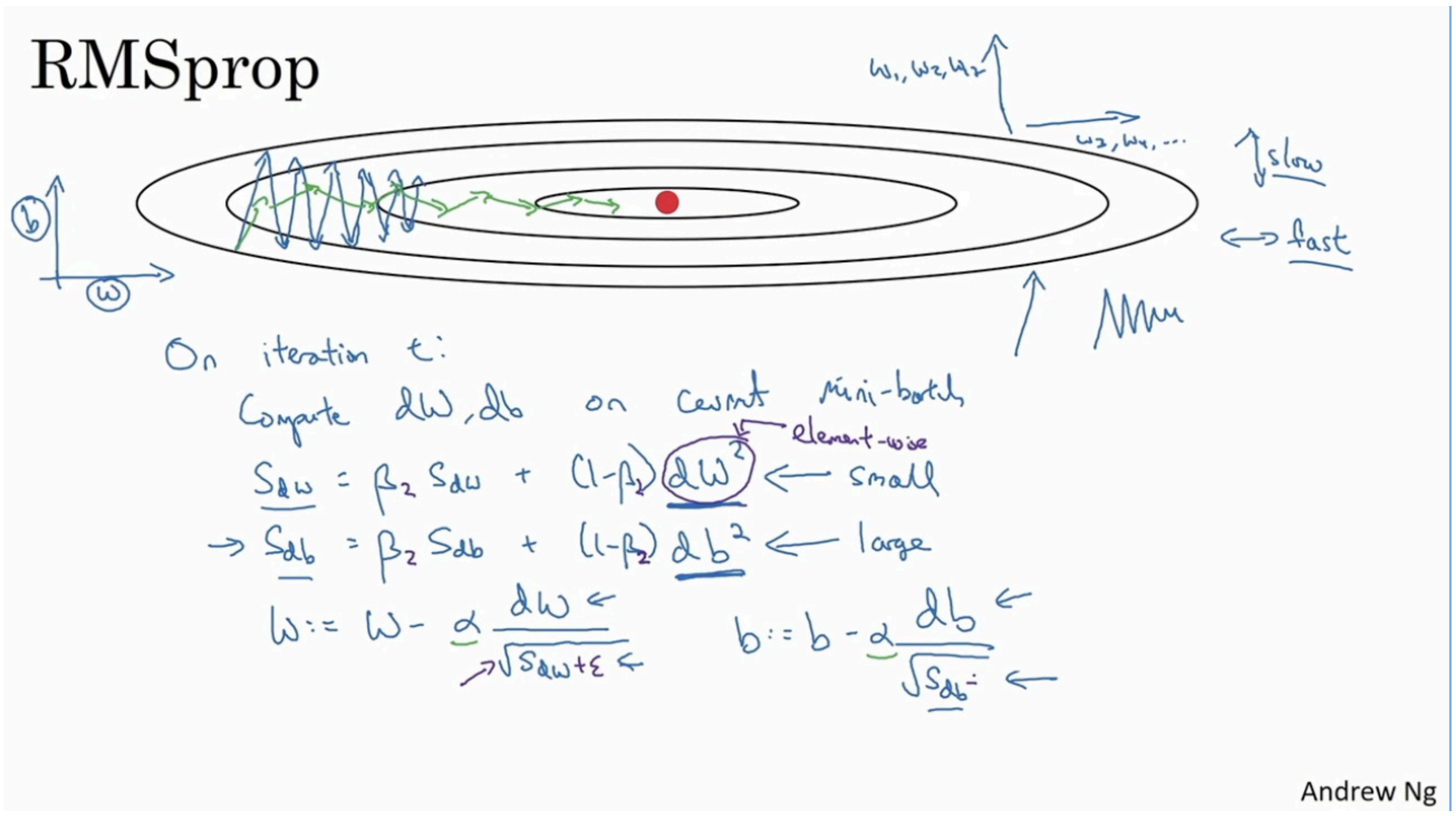
$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: $\underset{\uparrow}{\alpha}, \underset{\uparrow}{\beta}$ $\qquad \underline{\beta = 0.9}$

average over last ≈10 gradients

Andrew Ng

RMSprop



RMSprop

On iteration t:

Compute dW, db on current mini-batch

$$\underline{S_{dW}} = \beta_2 S_{dW} + (1-\beta_2)\,\underline{dW^2} \leftarrow \text{small}$$

element-wise

$$\rightarrow \underline{S_{db}} = \beta_2 S_{db} + (1-\beta_2)\,\underline{db^2} \leftarrow \text{large}$$

$$W := W - \underset{\smile}{\alpha}\frac{dW}{\sqrt{S_{dW}+\varepsilon}} \qquad b := b - \underset{\smile}{\alpha}\frac{db}{\sqrt{S_{db}}}$$

Andrew Ng

Adam Optimization Algorithm

Momentum + RMSprop

...a lot of calculations and derivatives

Learning Rate Decay

may never really converge to the most optimal point - steps become too big as you continue to get closer to the minima

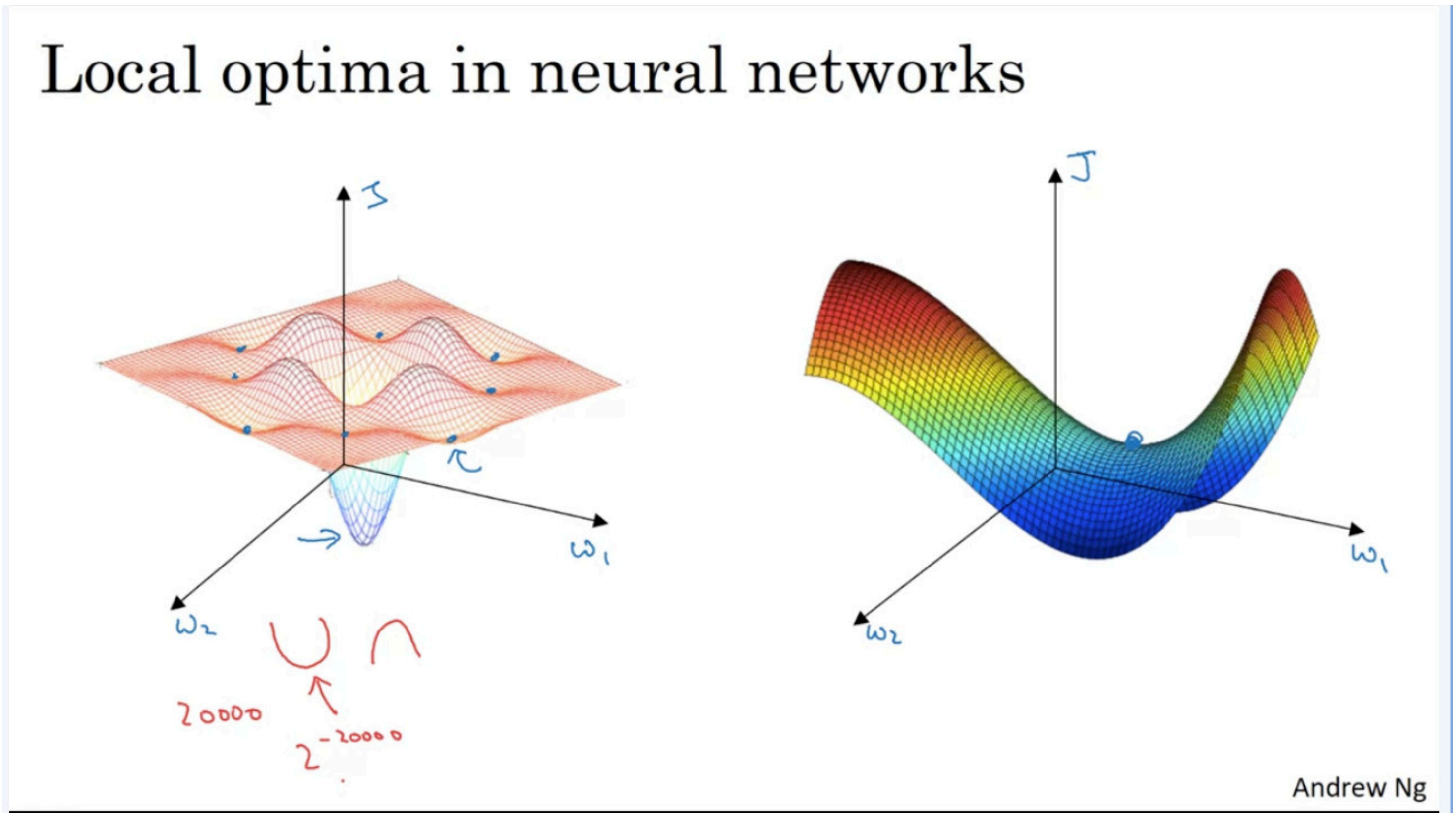
logic: the outer layers of the gradient is a bigger area, does not need that much precision, unlike the centre parts

There are different methods

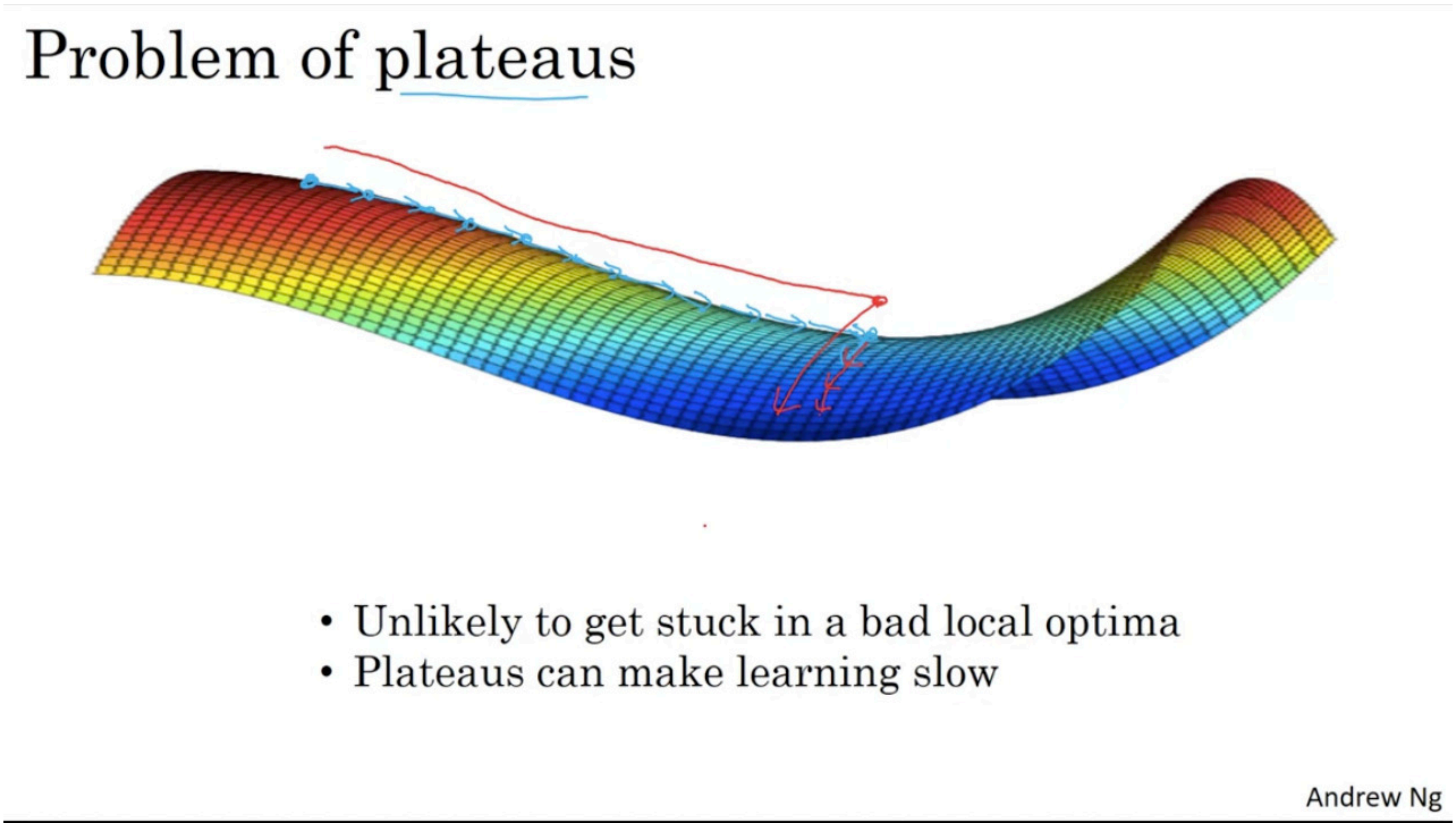
The Problem of Local Optima

how we now think about l.o.?

most 0 point in gradient **are not local optimas** but rather **saddle points** for high dimensional spaces:



Local optima in neural networks



Problem of plateaus

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

What you should remember:

- Shuffling and Partitioning are the two steps required to build mini-batches
- Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You have to tune a momentum hyperparameter β and a learning rate α.

Tuning Process

Hyperparameters

learning rate (alpha), momentum (beta), # of layers, # of hidden layers, learning rate decay, mini-batch size...

Importance order

Try random values: Don't use a grid
more richly exploring

Coarse to fine sampling scheme
find few set of parameters that perform the best, narrow down the region that you pick the hyperparameters based off on them

How to pick appropriate scale to pick the hyperparameters from

- for # of layers, logical to uniformly try new random values: 2,3,4...
- learning rate: 0.0001 to 1: not feasable to apply same logic. but better to pick in a logarithmic scale
- for exponentially weighted averages: using beta = 0.9 roughly means averaging last 10 values. 0.9 to 0.999, again better to use logarithmic scale.

How to organize hyperparameter tuning process

Babysitting one model

Train many models in parallel

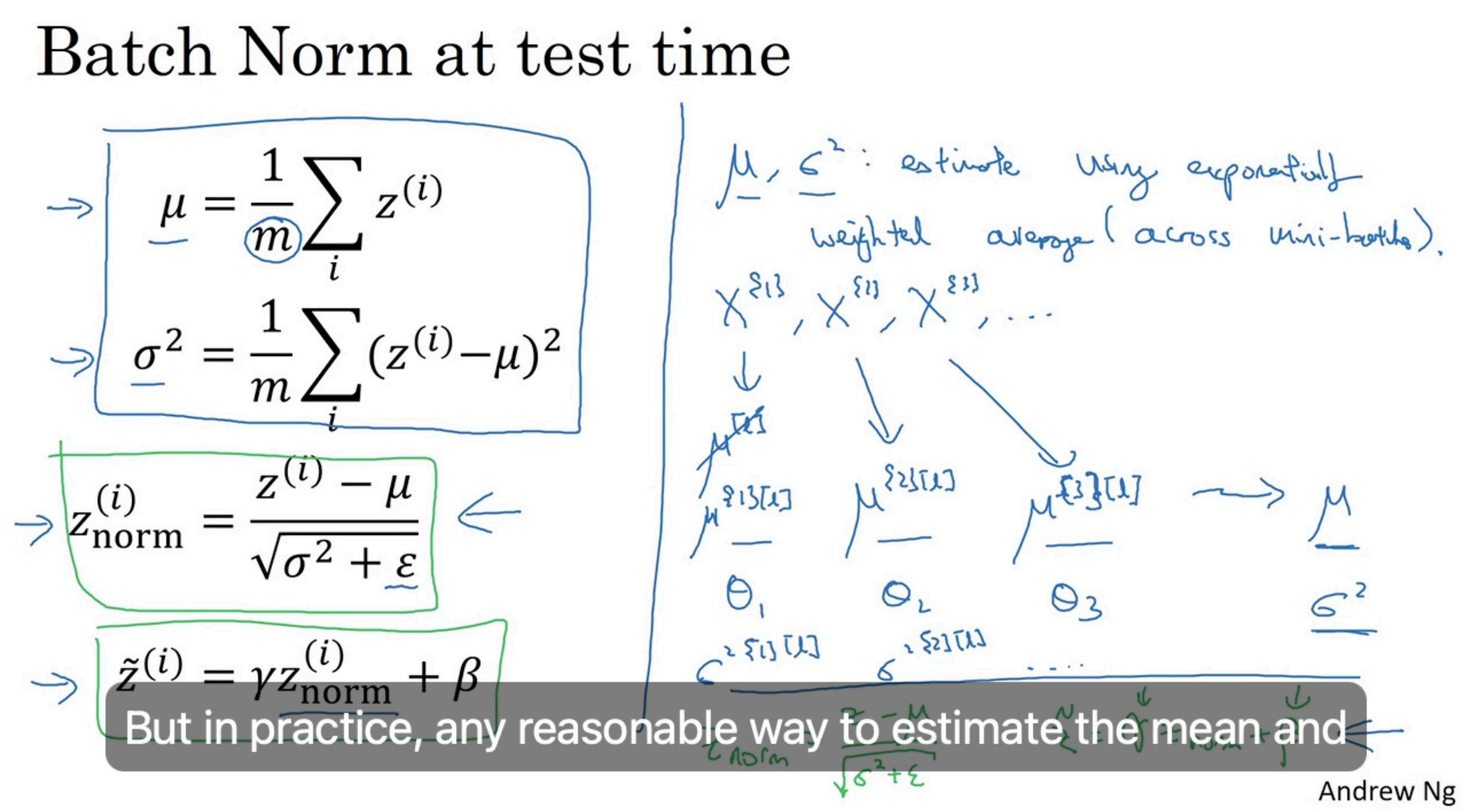
Normalizing Activations in a Network

Batch Normalization
normalizing the value of z^x so that z^(x+1) runs faster

adding batch norm to a network

Why does Batch Norm work?
doing the similar thing, not for only input but also for hidden layers

bath norm at test time



Batch Norm at test time

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

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

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

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

$\underline{\mu}$, $\underline{\sigma^2}$: estimate using exponentially weighted average (across mini-batch).

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \dots$

$\mu^{\{1\}[l]}$, $\mu^{\{2\}[l]}$, $\mu^{\{3\}[l]}$ $\longrightarrow$ $\underline{\mu}$

θ_1 θ_2 θ_3 $\underline{\sigma^2}$

$\sigma^{2\{1\}[l]}$ $\sigma^{2\{2\}[l]}$ ....

$z_{\text{norm}} = \frac{z-\mu}{\sqrt{\sigma^2+\varepsilon}}$ $\tilde{z} = \gamma z_{\text{norm}} + \beta$

But in practice, any reasonable way to estimate the mean and

Andrew Ng

Softmax Regression

multi-class classification

cat/no cat - cat/dog/human/...

for ex. output layer has 4 units, each are P(x | y), sum should be equal to 1. (confidence)

softmax activation function

may need to check the formula

Training a softmax classifier

cost, gradient descent, may need to check formulas & eqns.

Deep Learning Frameworks

there are many, each with relative pros and cons

TensorFlow

check code examples and practice