deeplearning.ai

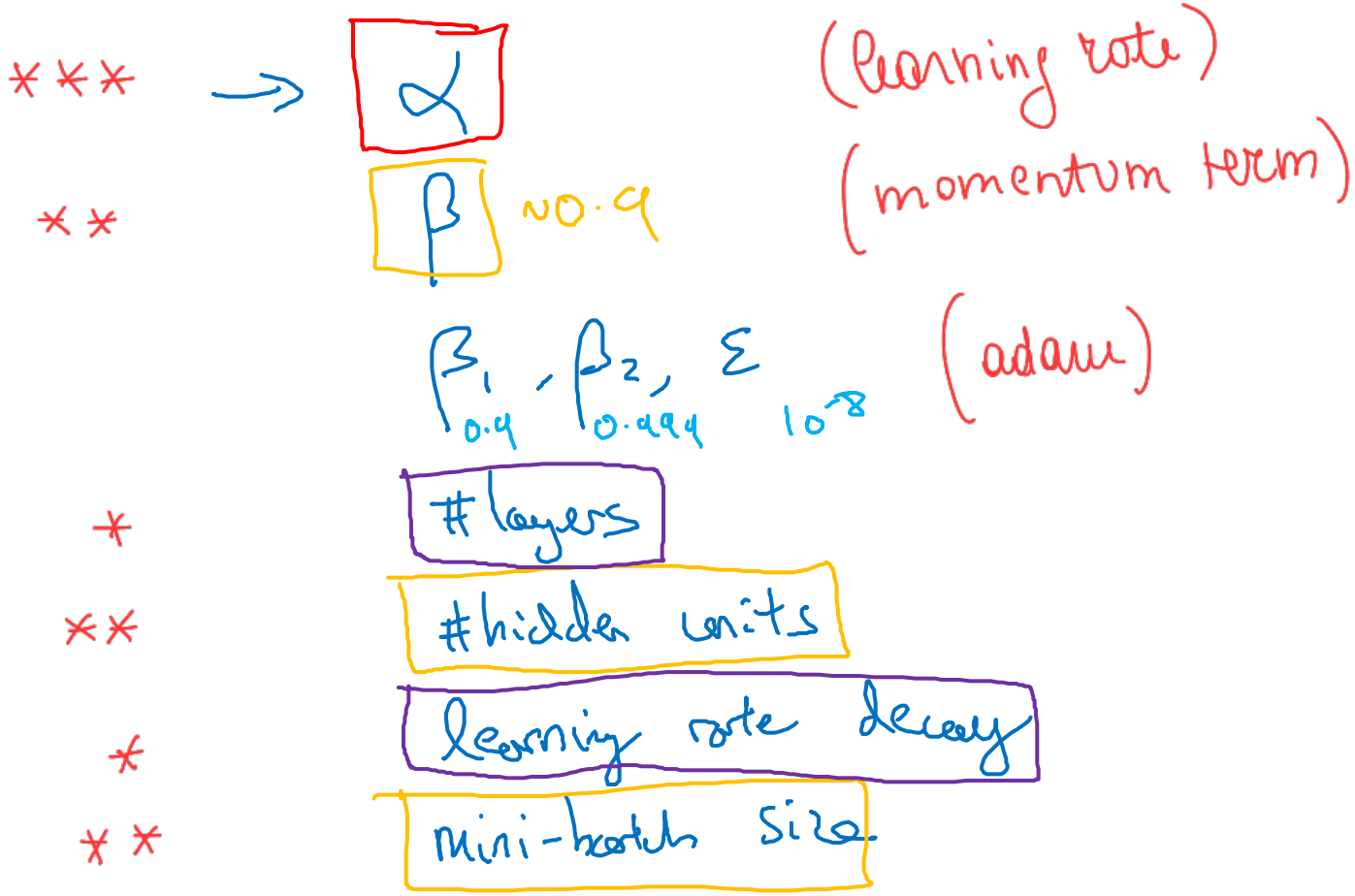
Hyperparameter tuning

Tuning process

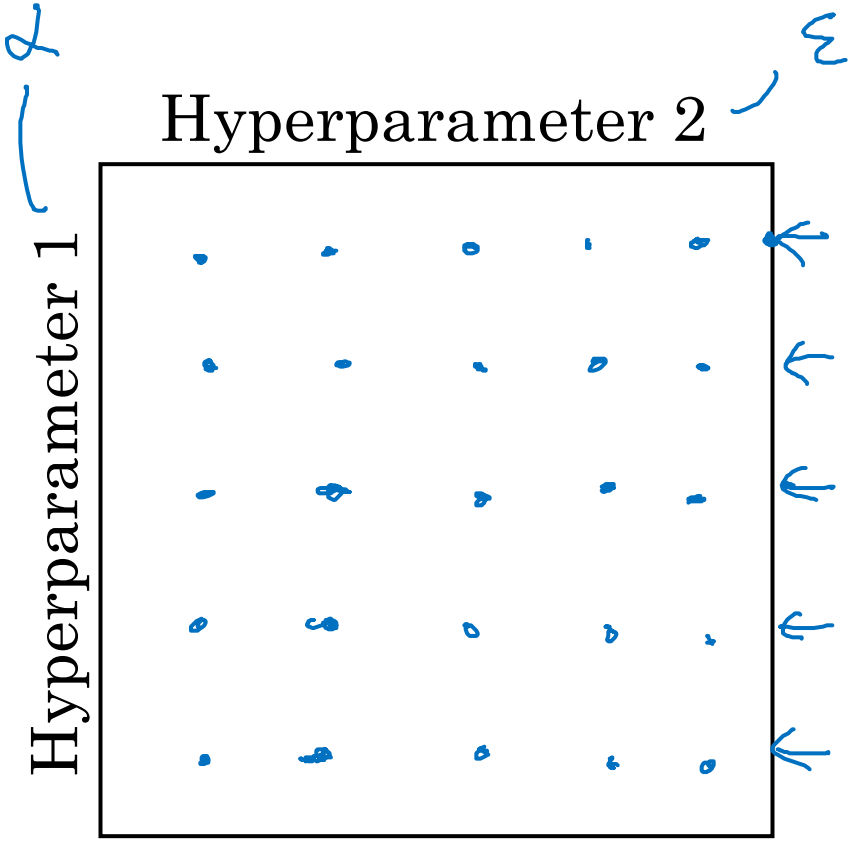
Hyperparameters

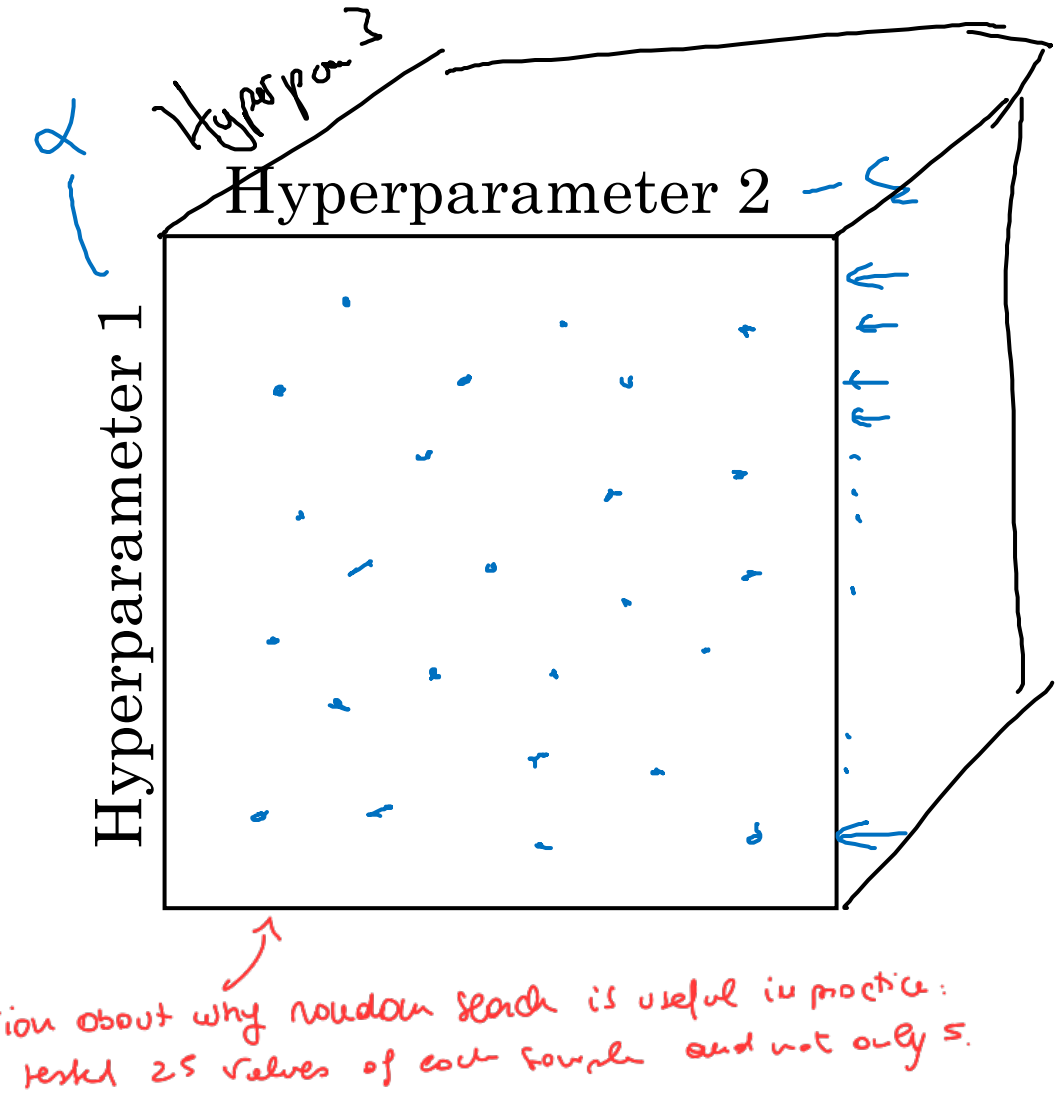
*** most important to tune

* least important to tune



Try random values: Don't use a grid

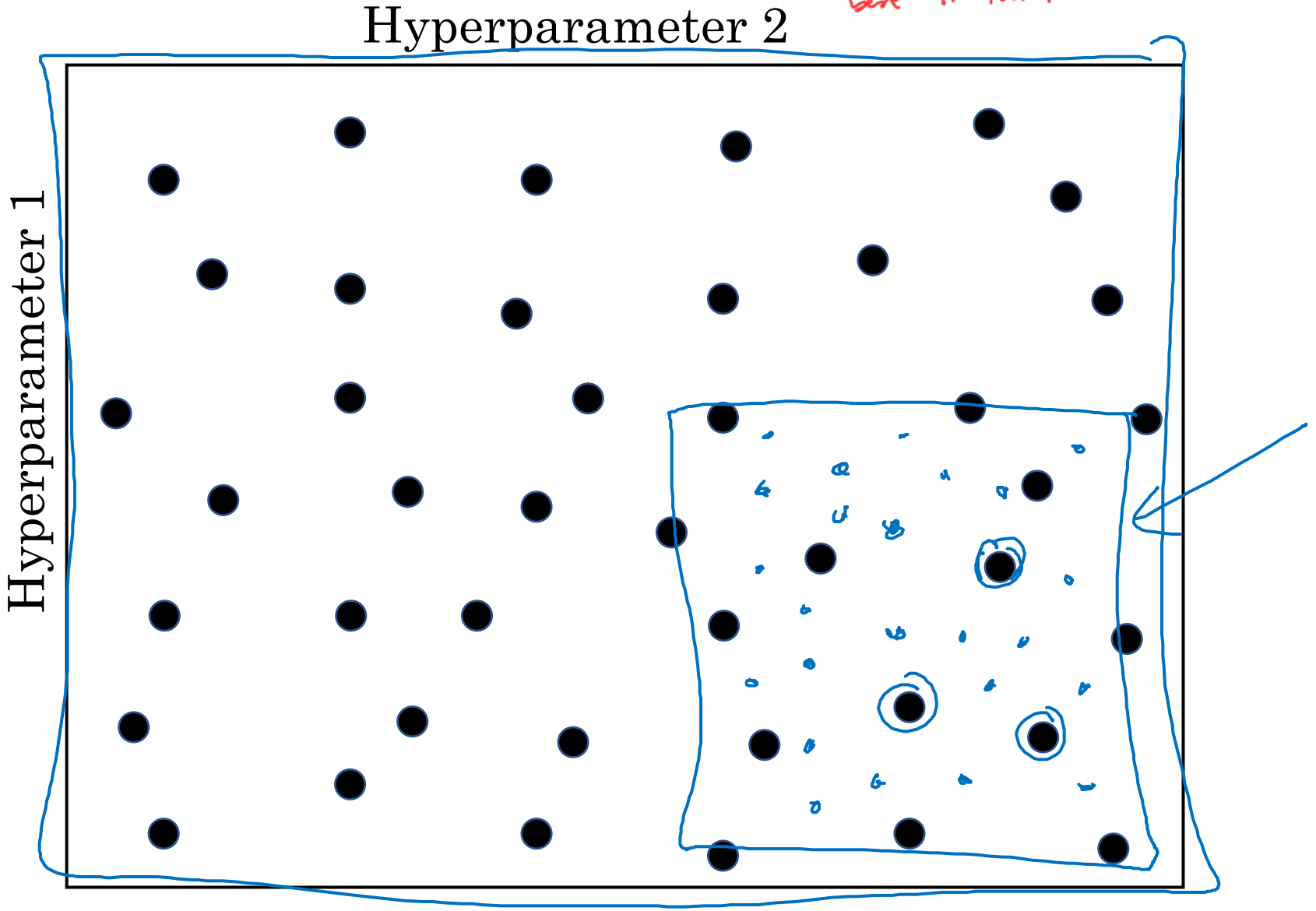




intuition about why random search is useful in practice: You tested 25 values of each hyperparameter and not only 5.

Coarse to fine

→ special search scheme where you progressively repeat new search sampling more densely in the areas of the search space that performed best in the previous search.



Andrew Ng

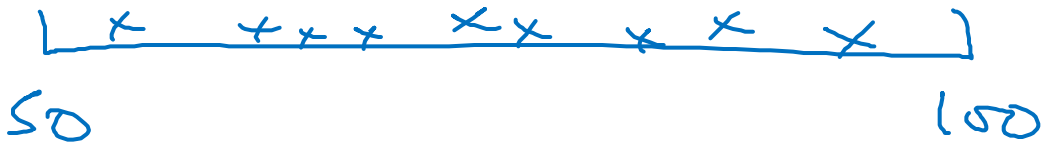
deeplearning.ai

Hyperparameter tuning

Using an appropriate scale to pick hyperparameters

Picking hyperparameters at random

→ $n^{[l]} = 50, \ldots, 100$



→ #layers L: 2 — 4

2, 3, 4

- Random search for hyp-opt should be performed at specific scales depending on the nature of the hyperparameter.
- discrete/integer hyperparameters should be sampled from a uniform distribution on a linear scale

$\theta \sim U([\theta_{min}, \theta_{max}])$

- Continuous hyperparameters should be sampled from a uniform distribution on a logarithmic scale.

$\theta = 10^{x \sim U([a,b])}$

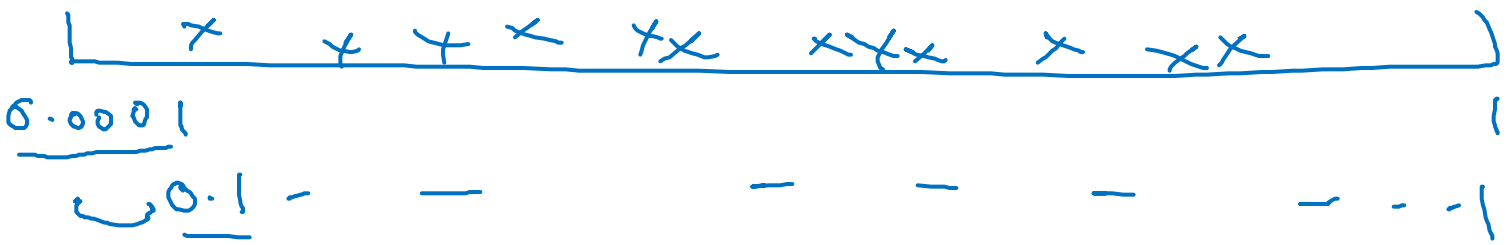
this can be done by:

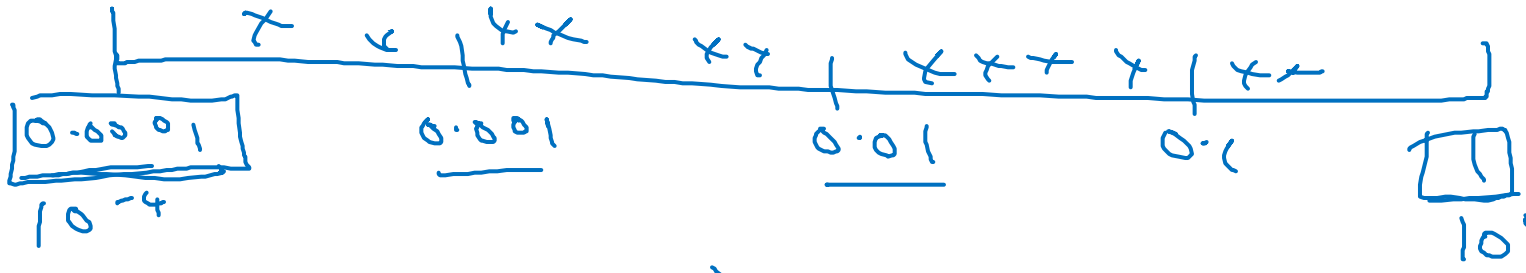
- seeing θ as 10^x and sampling the exponent x uniformly.
  θ in $[10^a, 10^b]$ => sample $\theta = 10^{x \sim U([a,b])}$
- sampling $\theta \sim \log U([a,b])$

Andrew Ng

Appropriate scale for hyperparameters

$\alpha = 0.0001, \ldots, 1$





10^{a} 10^{b}

$a = \log_{10} 0.0001 = -4$ $b = \log_{10} 1 = 0$

$r = -4 * np.random.rand()$ $\leftarrow$ $r \in [-4, 0]$

$\alpha = 10^{r}$ $\leftarrow$ $10^{-4} \ldots 10^{0}$

$10^{a} \ldots 10^{b}$ $r \in [a, b]$ $[-4, 0]$ $\alpha = 10^{r}$

Andrew Ng

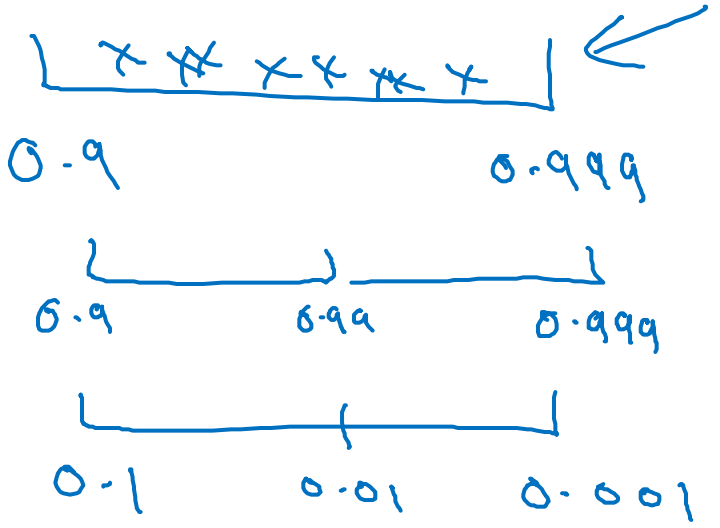
Hyperparameters for exponentially weighted averages

② Example: if params in $[0.9, 0.99]$, write it as $1-[10^{-2}, 10^{-1}]$ and then sample exponent in $[-2, -1]$.

$\beta = 0.9 \quad \ldots \quad 0.999$

$\downarrow \qquad \downarrow$

$10 \qquad 1000$



$1-\beta = 0.1 \quad \ldots \quad 0.001$

$10^{-1} \qquad 10^{-3}$

$\beta: 0.9000 \rightarrow 0.9005 \} \sim 10$

$\beta: 0.999 \rightarrow 0.9995$

$\sim 1000 \qquad \sim 2000$

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

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

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

$\frac{1}{1-\beta}$

⇒ loguniform sampling is crucial

① - intuition: an example in which loguniform sampling is very relevant is when sampling the momentum parameter β. As β increases toward 1, smaller intervals become much denser of relevant points. Varying β in $[0.9, 0.9995]$ is much more sensitive than varying it in $[0.9, 0.9005]$.

average over ~1000 samples — average over ~2000 samples.

Andrew Ng

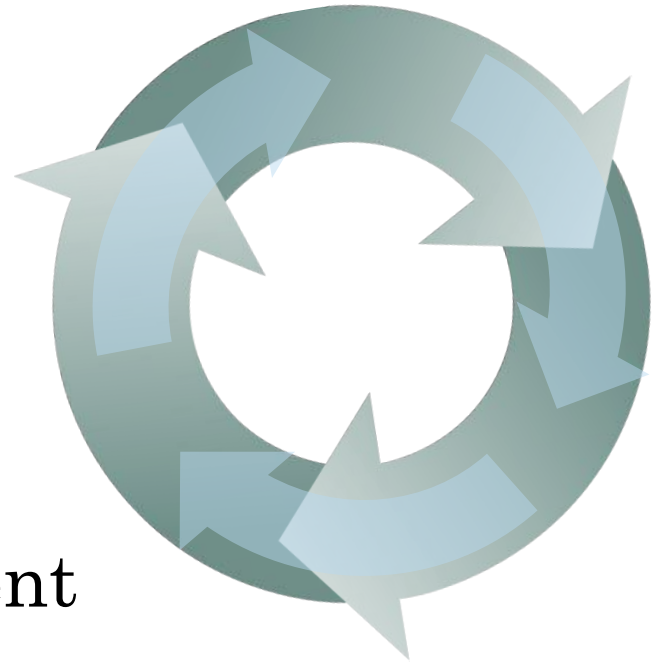
deeplearning.ai

Hyperparameters tuning

Hyperparameters tuning in practice: Pandas vs. Caviar

Re-test hyperparameters occasionally

Idea



Experiment

Code

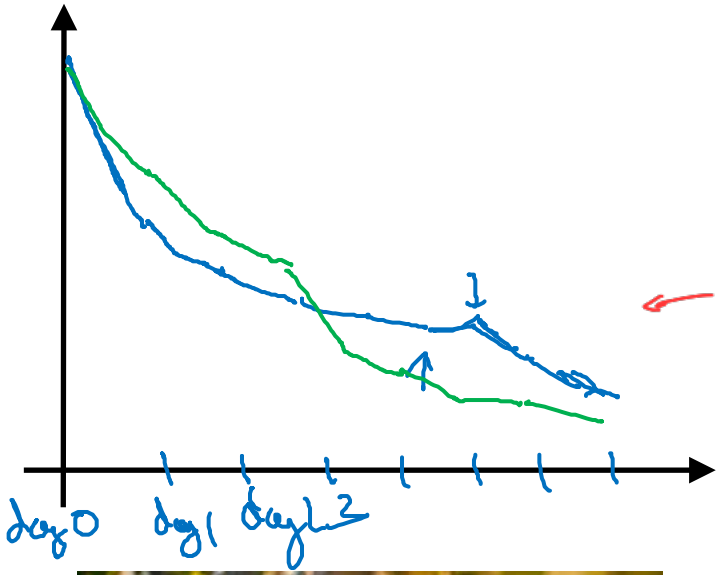
- NLP, Vision, Speech, Ads, logistics, ...

- Intuitions do get stale. Re-evaluate occasionally.

Real world data (in real applications) may be non-stationary, therefore it may be useful to periodically re-check the model's hyperparams.

Andrew Ng

Babysitting one model



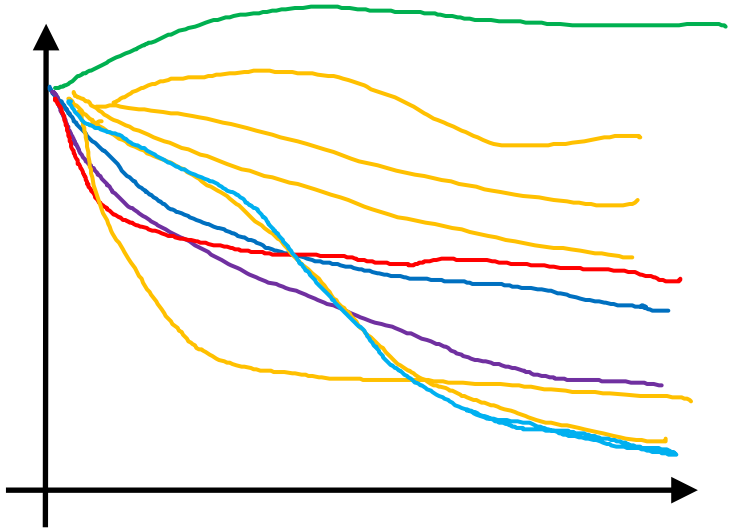
How to test for different hyperparameters depends on the computational resources.

limited resources: Test a set of hyperparams (a model), as it trains (maybe over days of training). Keep adjusting the hyperparams based on the convergence trend.

Lots of resources: test many hyperparams in parallel.

Panda

Training many models in parallel



Caviar

Andrew Ng

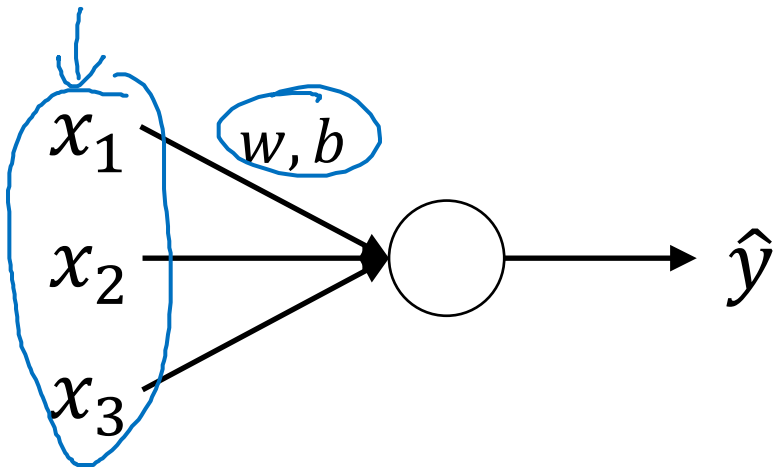
deeplearning.ai

Batch Normalization

Normalizing activations in a network

Batch normalization simplifies the hyperparameter search and speeds up convergence of the network.

Normalizing inputs to speed up learning

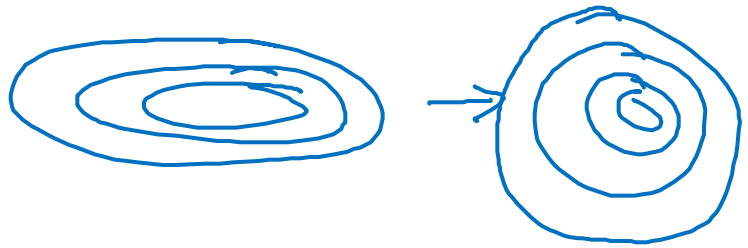


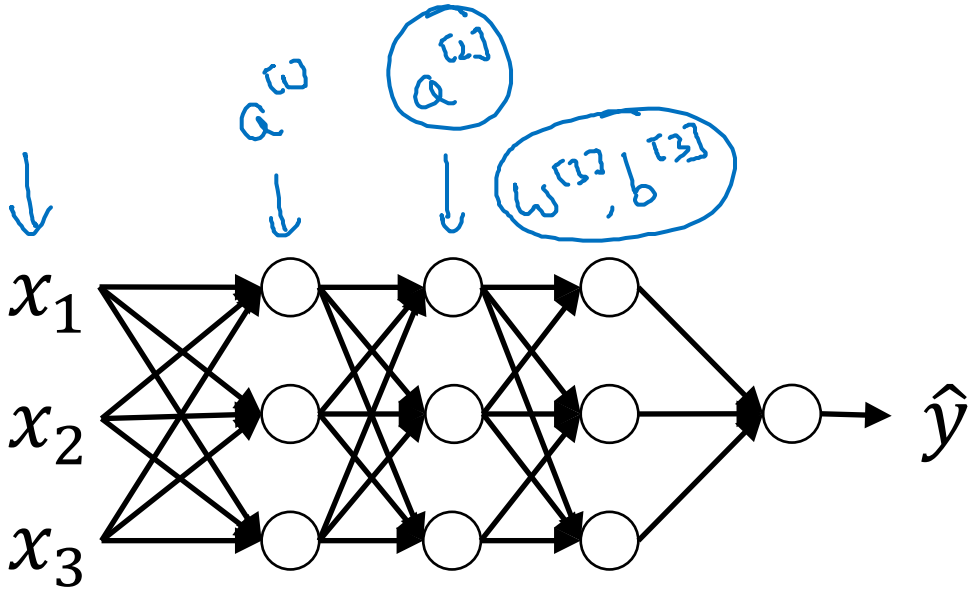
$$\mu = \frac{1}{m}\sum_i x^{(i)}$$

$$X = X - \mu \quad \text{(element-wise)}$$

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

$$X = X / \sigma^2$$





Can we normalize $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster?

Normalize $z^{[2]}$ so to train params of layer $n+1$ faster.

($a^{[n]}$ → actually $z^{[n]}$ is most often normalized.)

Andrew Ng

Implementing Batch Norm

IDEA of BATCHNORM: normalizing the output of layer n ($a^{[n]}$, or more commonly $z^{[n]}$) so to train the parameters of layer n+1 faster.

Given some intermediate values in NN $z^{(1)}, \dots, z^{(m)}$ of a certain layer l: $z^{[l](i)}$

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

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

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

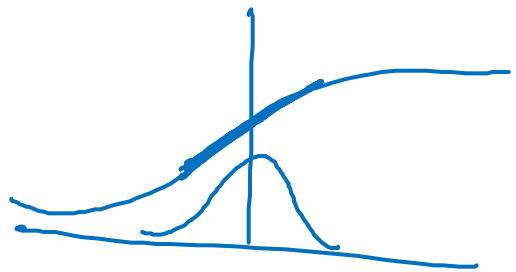
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

(γ: modifies variance; β: modifies mean)

If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, then $\tilde{z}^{(i)} = z^{(i)}$

$X \leftarrow$

$z^{(i)} \leftarrow$



learnable parameters of model.

Use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$.

NOTES:
- γ, β are params and not hyperparams (=> they are learned)
- $\gamma^{[l](i)}, \beta^{[l](i)}$ defined for each layer l and hidden unit i.
- The param $\beta^{[l]}$ has nothing to do with the momentum hyperparameter

- For layer l and all the samples in the mini-batch, compute the mean and variance of the activations $z^{[l](i)}$ of the units in that layer. Then normalize the activation $z^{[l](i)}$ of each unit i using μ and σ.
- Differently from the normalization of the inputs (input layer) we may want $z^{[l](i)}$ to have μ and σ different from 0 and 1 respectively. For this reason the params γ, β are introduced and tuned. $\tilde{z}^{[l](i)} = \gamma z^{[l](i)}_{norm} + \beta$. γ modifies the variance of z and β the mean.

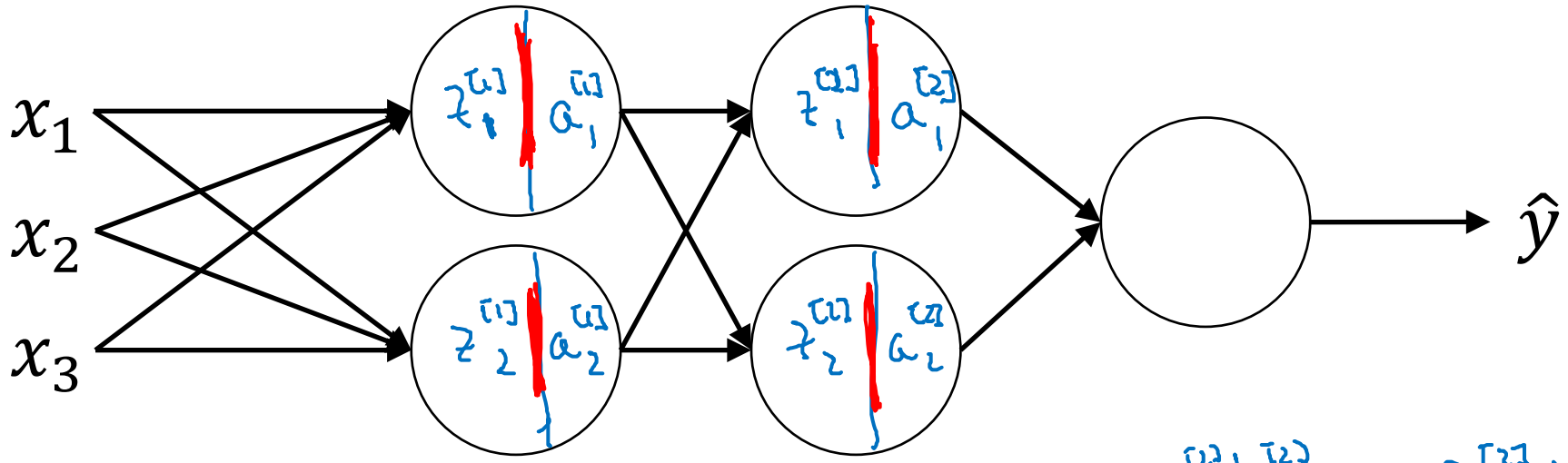
Andrew Ng

deeplearning.ai

Batch Normalization

Fitting Batch Norm into a neural network

Adding Batch Norm to a network



$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$

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

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

$d\beta^{[l]}$

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

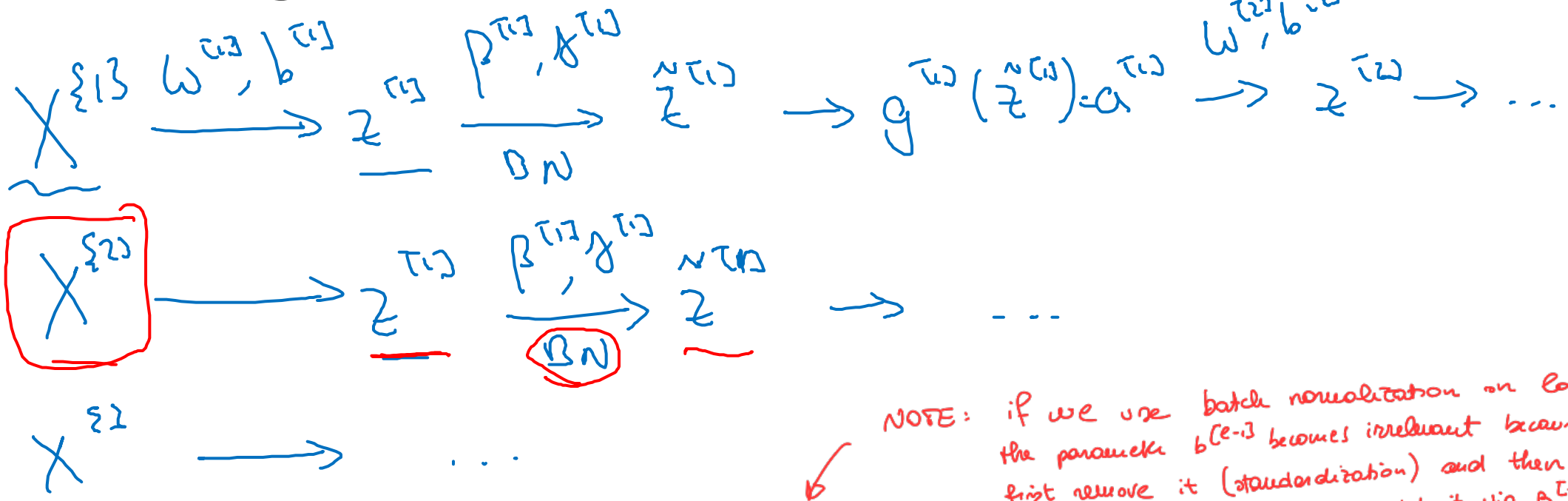
tf.nn.batch_normalization ←

- β and γ are learned with backpropagation (like W and b).

Andrew Ng

Working with mini-batches

- For minibatch GD, the batch norm is repeated for each minibatch (i.e. using only the data from that minibatch).



NOTE: if we use batch normalization on layer l, then the parameter $b^{[l-1]}$ becomes irrelevant because the BN[l] will first remove it (standardization) and then potentially override it via $\beta^{[l]}$.

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

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

* One β, γ for each hidden unit of each layer.

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

$$Z^{[l]} = W^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$$

$$Z^{[l]} = W^{[l]} a^{[l-1]}$$

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

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

Andrew Ng

Implementing gradient descent with Batch Norm

for $t = 1 \ldots$ numMiniBatches

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

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

Use backprop to compute $dW^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$

Update parameters $W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$

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

$\gamma^{[l]} := \ldots$

Works w/ momentum, RMSprop, Adam.

Same as without BN. Works also with other optimization algorithms such as Adam.

Motivation 1: (As said before) because it standardizes the inputs of intermediate layers therefore making GD faster at learning those layers' params (similar to what happens for the input normalization).

M2: because it makes the weights of later layers more robust to changes to weights in earlier layers. It does so by mitigating the distribution shift of the inputs $a^{[l-1]}$ to the layer l due to changes in the weights of earlier layers $(1,\dots,l-1)$. Reducing the distribution shift "decouples" earlier and later layers, that is, makes later layers a bit more independent from changes of the previous ones therefore simplifying/speeding up learning.

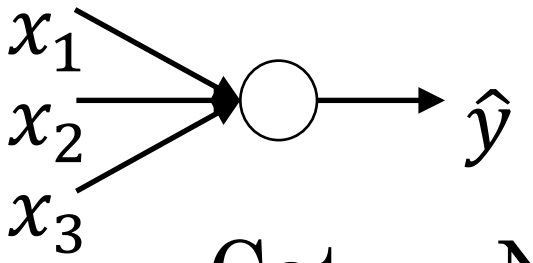
deeplearning.ai

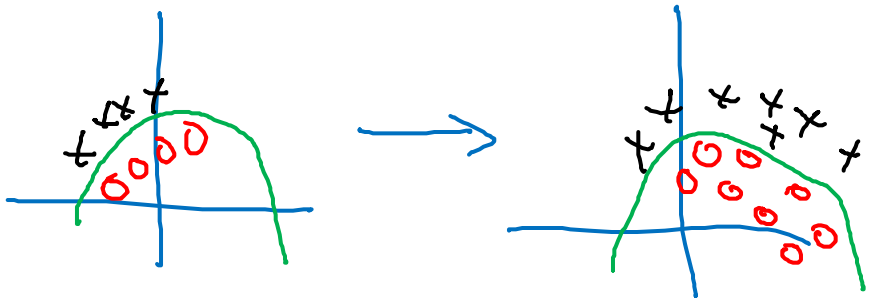
Batch Normalization

Why does Batch Norm work?

NOTE: Batch normalization has a slight regularization side-effect.

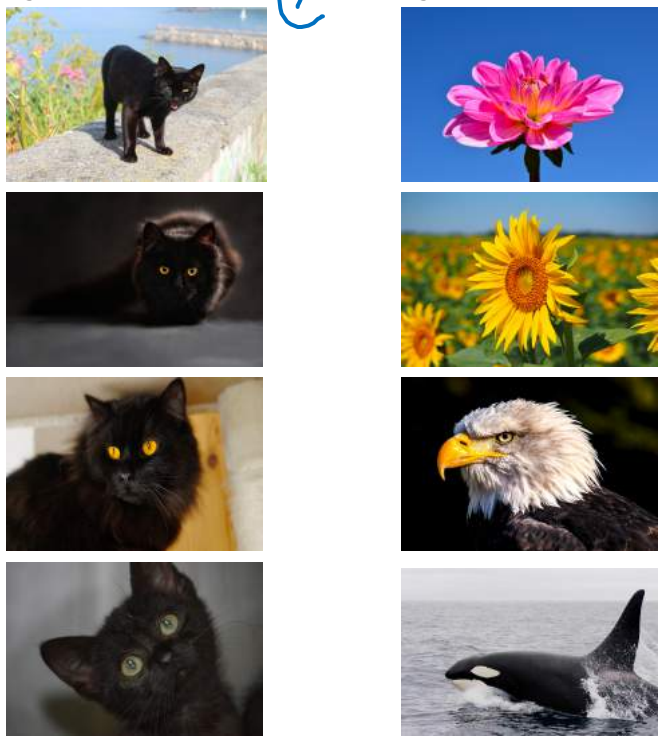
Learning on shifting input distribution





Cat $y = 1$

Non-Cat $y = 0$



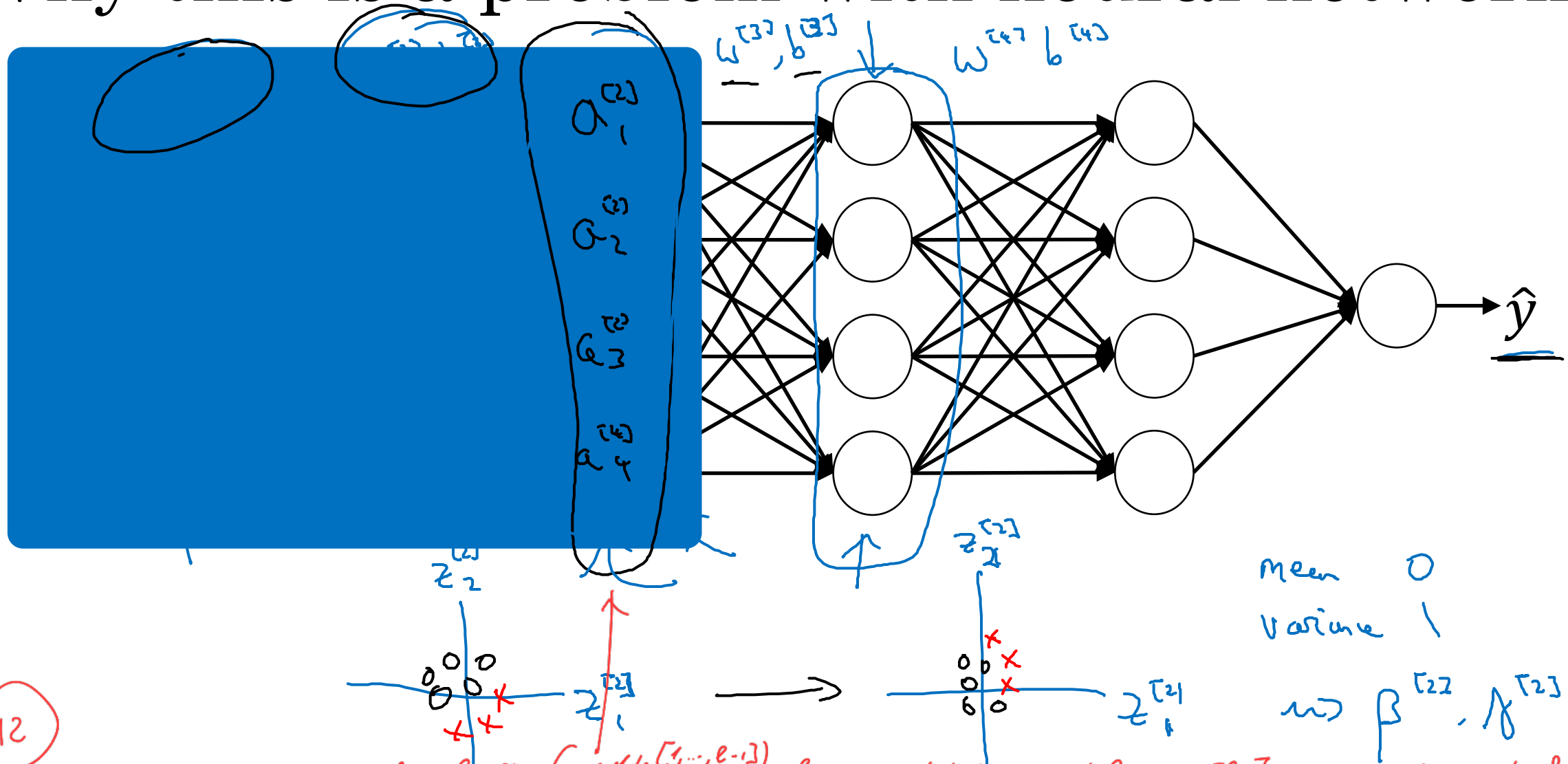
"Covariate shift"

$X \rightarrow y$

$y = 1$ $y = 0$

Andrew Ng

Why this is a problem with neural networks?



M2

- changes in earlier layers (weights $[1,\dots,\ell-1]$) cause a distribution shift in $a^{[\ell-1]}$, which is the input of layer ℓ.
- BN applied to layer N reduces the distribution shift by ensuring a constant mean & variance of the inputs.
- This makes learning a bit easier and also decouples the NN's layers (forces the network to learn layers independently).

Batch Norm as regularization

Secondary / side-effect of batch norm.

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

$\tilde{z}^{[l]}$ μ, σ^2 $X^{\{t\}}$ $Z^{[l]}$ 64, 128

mini-batch: 64 ⟶ 512

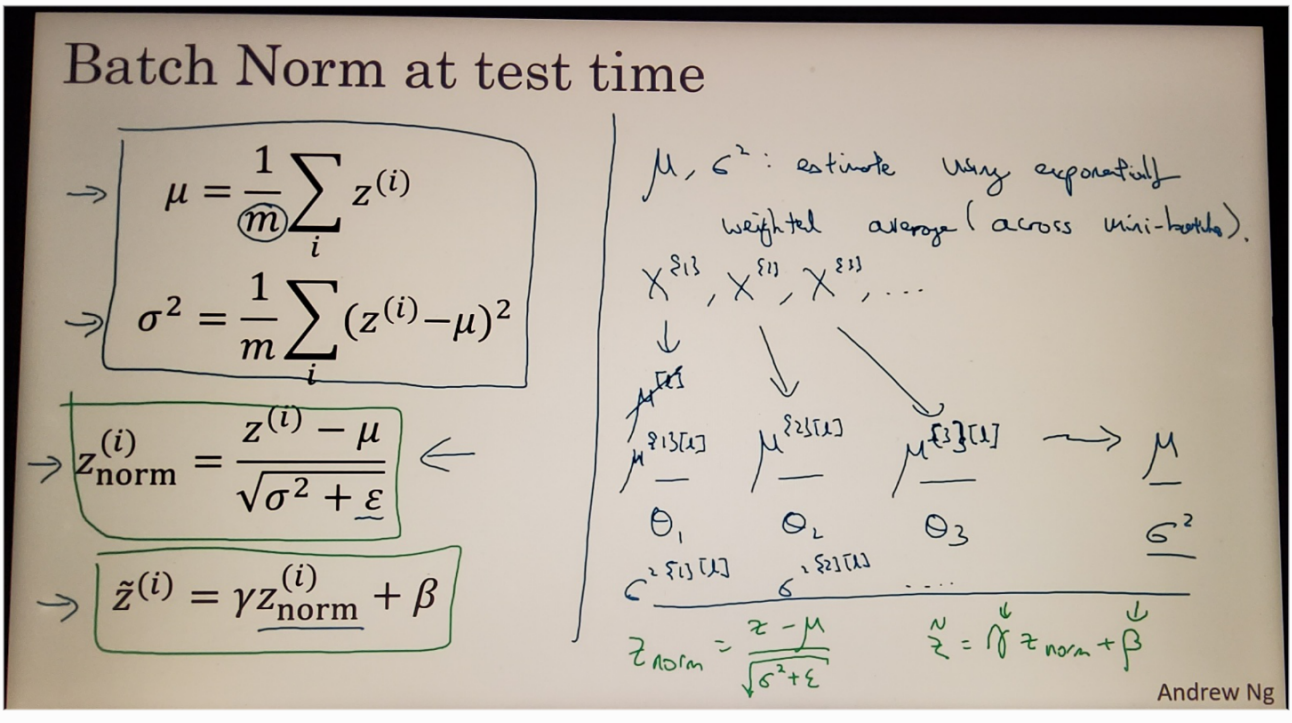
- BN in combination with mini batch learning has a slight regularization effect because layers activations are scaled by quantities computed on a portion of the data and are, therefore, not exactly right. This has the effect of adding some noise to each layer, therefore forcing the NN to "spread" the weights. Similar to what dropout does.
  ↳ => Regularization effect. => SMALLER MINIBATCH = more noise = more regularization

BATCH NORM AT TEST TIME (missing slide)

Problem: at test time, examples are potentially fed into the NN one at a time and is therefore impossible to estimate $\mu^{[\ell]}, \sigma^{[\ell]}$ to standardize the activation of the layer ℓ.

=> At training time: compute an exponentially weighted average of $\mu^{[\ell]}, \sigma^{[\ell]}$ over consecutive mini-batches (and keep learning β, γ).

- then, at test time, use that $\mu^{[\ell]}, \sigma^{[\ell]}$ to perform layer standardization.



deeplearning.ai

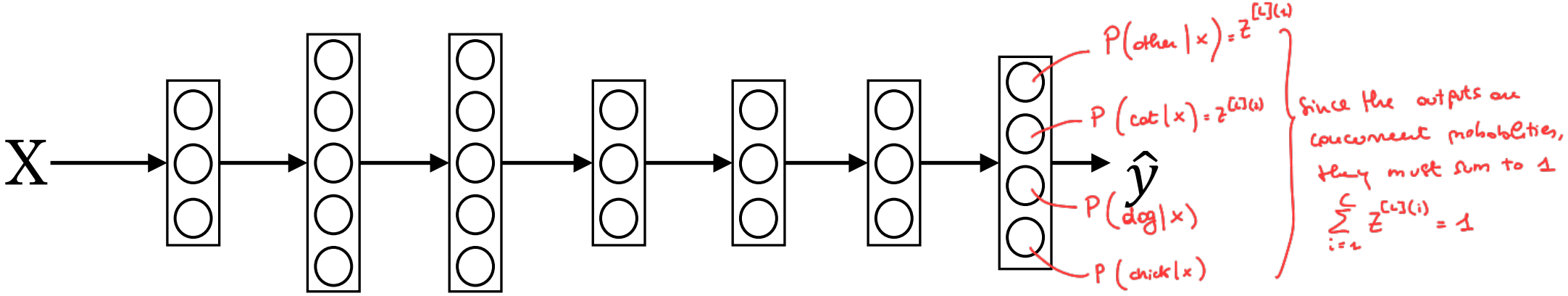
Multi-class classification

Softmax regression

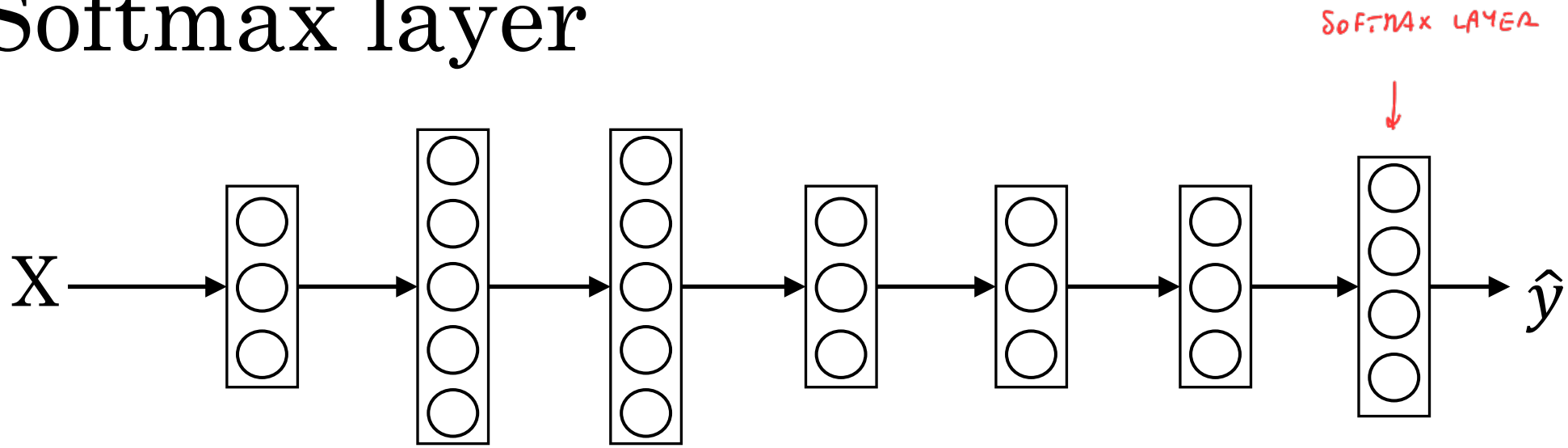
Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1

C = # classes (in this case C=4) → the n_units in the output layer must be = C



Softmax layer



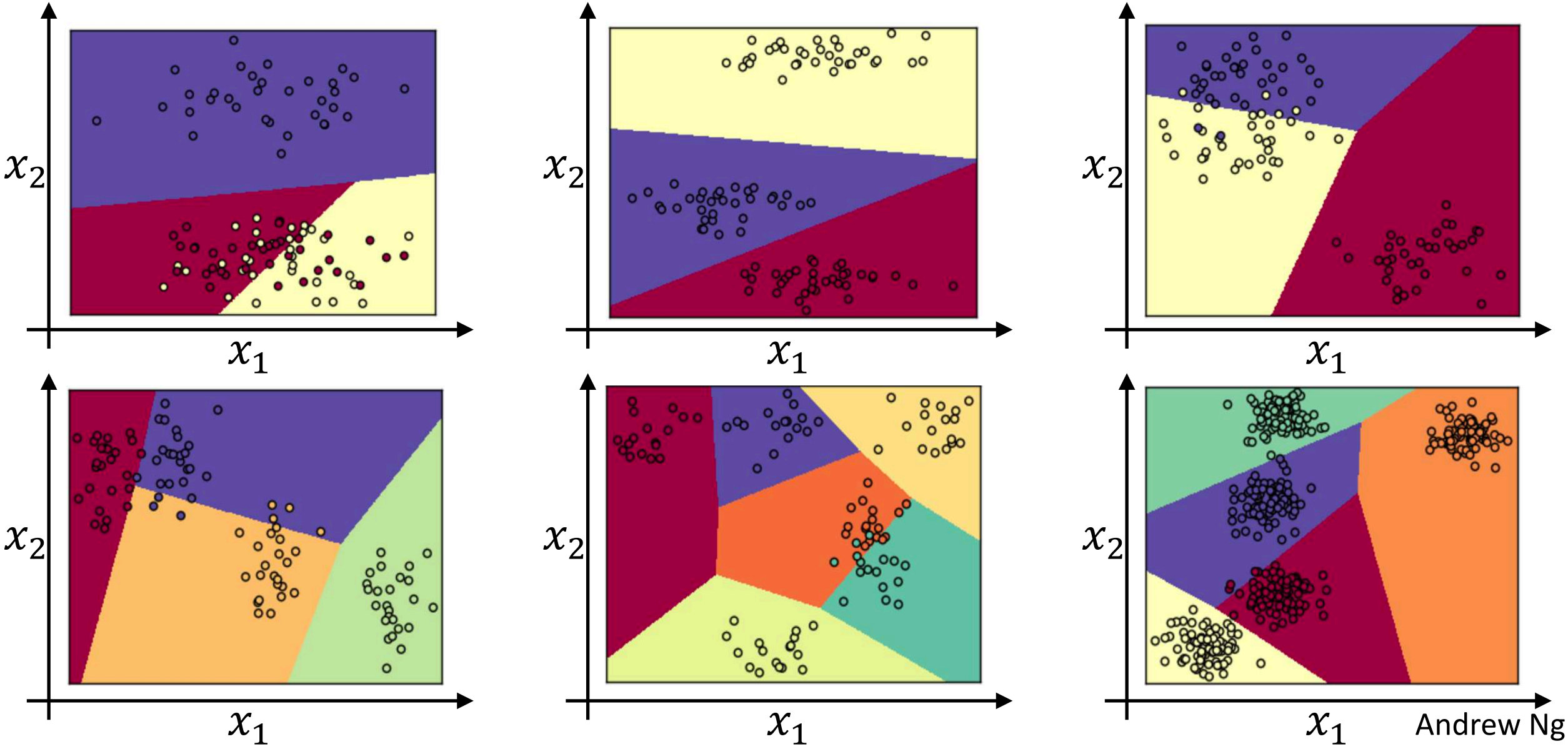
- to make the outputs of the last layer be probabilities that sum up to one, the last layer is made implement a softmax function. The activation of each hidden unit becomes

$$a^{[\ell](i)} = \frac{e^{z^{[\ell](i)}}}{\sum_{j=1}^{n_\ell} e^{z^{[\ell](i)}}}$$

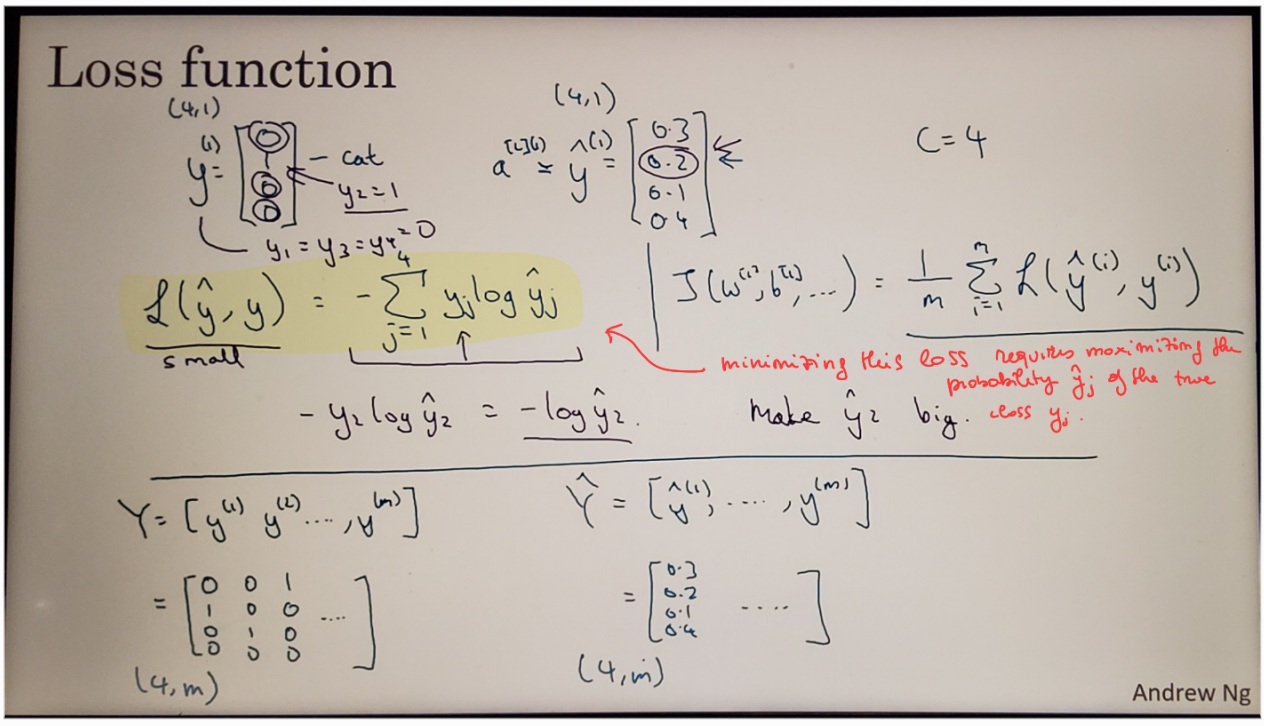
- Softmax regression generalizes logistic regression to $C > 2$ classes. For $C = 2$ the two activations converge.

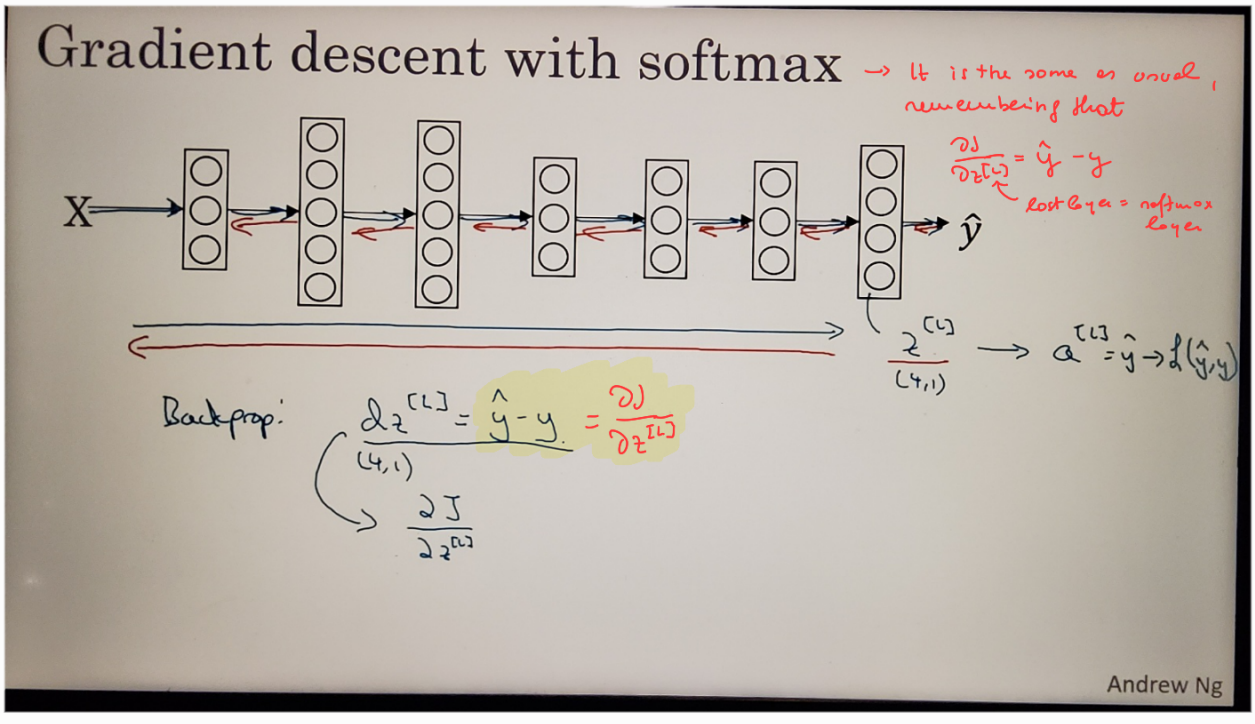
Andrew Ng

Softmax examples



Loss function for softmax regression





deeplearning.ai

Programming Frameworks

Deep Learning frameworks

Deep learning frameworks

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

Choosing deep learning frameworks

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

↳ how likely is it that the company owning the software keeps it open over the years.

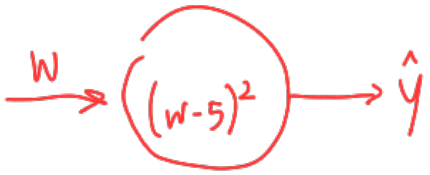
deeplearning.ai

Programming Frameworks

TensorFlow

- Cool feature: you only have to implement the forward step and it will compute the backprop automatically.

Motivating problem



$J(w) = w^2 - 10w + 25$

(cost)

$(w-5)^2$

$w = 5$

The loss only wants to minimize the parabola. No error wrt inputs is considered. We know that the optimum will be $w=5$. Let's find it using Tensorflow.

$J(w,b)$

solution:

File Edit View Insert Cell Kernel Widgets Help — Trusted — Python 3

▶ Run — Code

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

In [2]:
```
w = tf.Variable(0, dtype=tf.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def train_step():
    with tf.GradientTape() as tape:
        cost = w ** 2 - 10 * w + 25
    trainable_variables = [w]
    grads = tape.gradient(cost, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))

print(w)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0>
```

In [3]:
```
train_step()
print(w)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.09999997>
```

In [4]:
```
for i in range(1000):
    train_step()
print(w)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000001>
```

In [ ]:

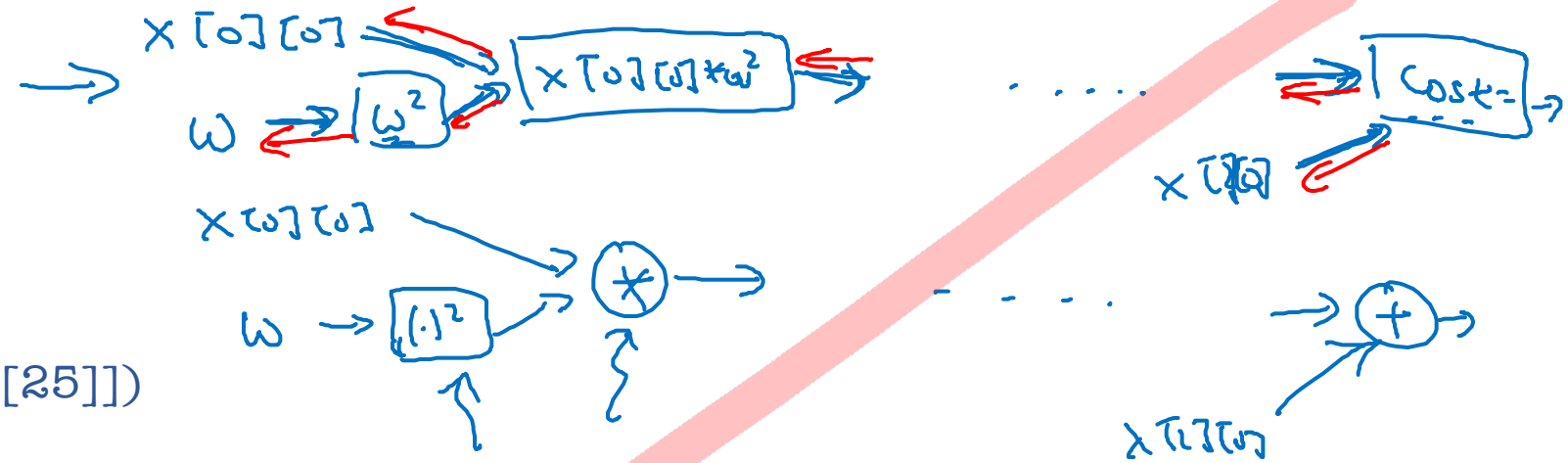
Other problem: the cost function includes the training data

$$J(w,b) = w_1^2 x_1 + w_2 x_2 + x_3$$

In [7]:
```
w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def training(x, w, optimizer):
    def cost_fn():
        return x[0] * w ** 2 + x[1] * w + x[2]
    for i in range(1000):
        optimizer.minimize(cost_fn, [w])

    return w

w = training(x, w, optimizer)
print(w)
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000001>
```

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))
```

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

Andrew Ng