Improving Deep Neural Networks

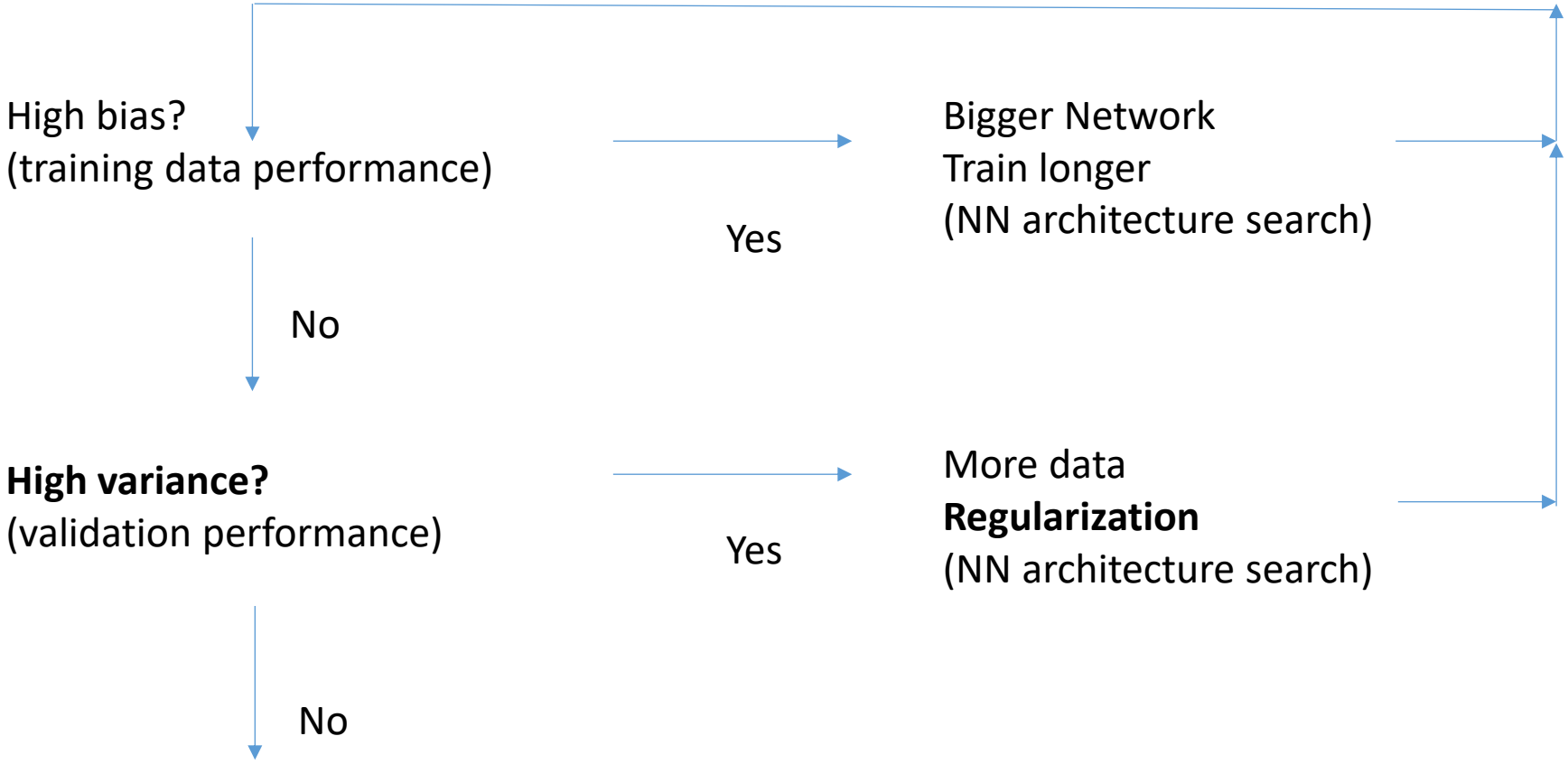
Text and Web Mining (IS6751)

School of Communication and Information

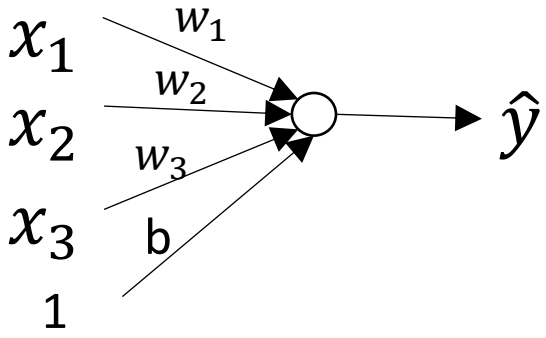
Overview

- Introduce various approaches for model optimization.
  - Regularization and Early stopping
  - Input normalization and Weights initialization
  - Mini-batch gradient descent
  - Gradient descent with momentum, RMSprop, Adam
  - Learning rate decay
  - Hypermeter turning process
  - Batch normalization

Basic recipe for machine learning



Regularization: Logistic regression



这个slide主要介绍 Logistic Regression 的正则化。
L2 正则化（岭回归） 通过惩罚权重的平方和来防止过拟合。
L1 正则化（Lasso 回归） 通过惩罚权重的绝对值和来实现特征选择，使模型更稀疏。
正则化参数 λ 控制正则化强度，影响模型的复杂度和预测能力

- $\min_{w,\,b} J(w, b)$
- Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, λ = regularization parameter (positive value)
- $J(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\|w\|_2^2 + \frac{\lambda}{2m}b^2$

第一项是 逻辑回归的损失函数，用于衡量预测值与真实值之间的误差。

omit

- L_2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$
- L_1 regularization: $\|w\|_1 = \sum_{j=1}^{n_x} |w_j|$ (note: w becomes sparse)

变得稀疏

$$\sum_{j=1}^{n_x} w_j^2 = w^T w = [w_1\ w_2\ w_3] \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

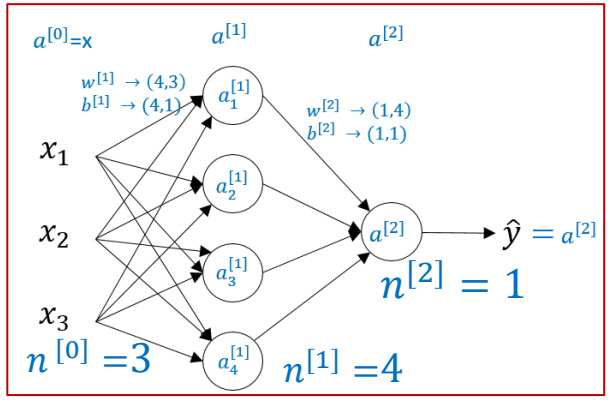
$$= w_1^2 + w_2^2 + w_3^2$$

$$\sum_{j=1}^{n_x} |w_j| = |w_1| + |w_2| + |w_3|$$

Rationale: When λ approaches infinity, shrink the estimates of w_i towards zero.

Regularization: Neural network

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



$$\|\boldsymbol{W}^{[l]}\|_2^2 = \sum_{i=1}^{n[l]}\sum_{j=1}^{n[l-1]}(w_{ij}^{[l]})^2 \quad where\ \boldsymbol{W}^{[l]}\ is\ (n^{[l]}, n^{[l-1]})$$

When $\boldsymbol{W}^{[l]} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$,

$\|\boldsymbol{W}^{[l]}\|_2^2 = {w_{11}}^2 + {w_{12}}^2 + {w_{21}}^2 + {w_{22}}^2$

$$\frac{\partial J}{\partial \boldsymbol{W}^{[l]}} = \boldsymbol{dW}^{[l]} = (from\ backprop) + \frac{\lambda}{m}\boldsymbol{W}^{[l]}$$

$$\boldsymbol{W}^{[l]} = \boldsymbol{W}^{[l]} - \alpha \boldsymbol{dW}^{[l]} = \boldsymbol{W}^{[l]} - \alpha[(from\ backprop) + \frac{\lambda}{m}\boldsymbol{W}^{[l]}]$$

$$= \boldsymbol{W}^{[l]} - \alpha\frac{\lambda}{m}\boldsymbol{W}^{[l]} - \alpha(from\ backprop)$$

$$= (1 - \alpha\frac{\lambda}{m})\boldsymbol{W}^{[l]} - \alpha(from\ backprop)$$

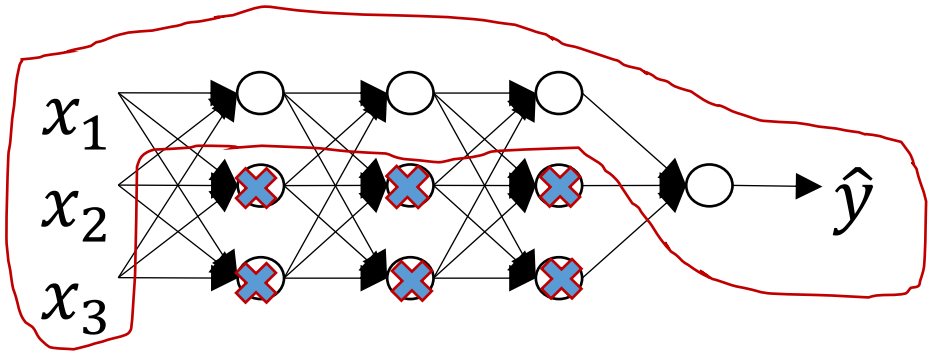
$0 \le \alpha\frac{\lambda}{m} < 1$, so called "**Weight decay**" 体重下降

Without Regularization:

$\boldsymbol{W}^{[l]} = \boldsymbol{W}^{[l]} - \alpha(from\ backprop)$

深度学习训练时，通常会使用 L2正则化（Weight Decay）来防止神经网络模型过拟合。
在 Adam, SGD, RMSProp 等优化算法中，通常可以设置 weight_decay 参数来控制 L2 正则化强度。

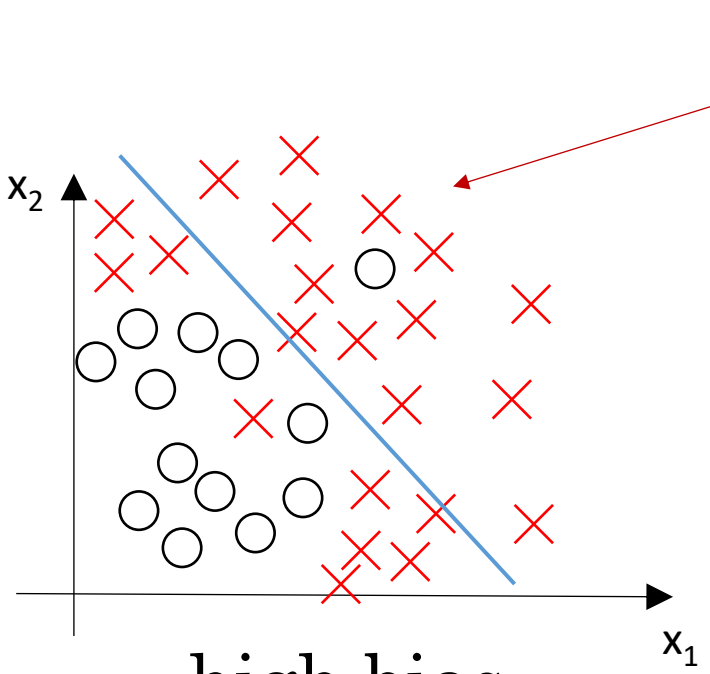
How does regularization prevent overfitting?



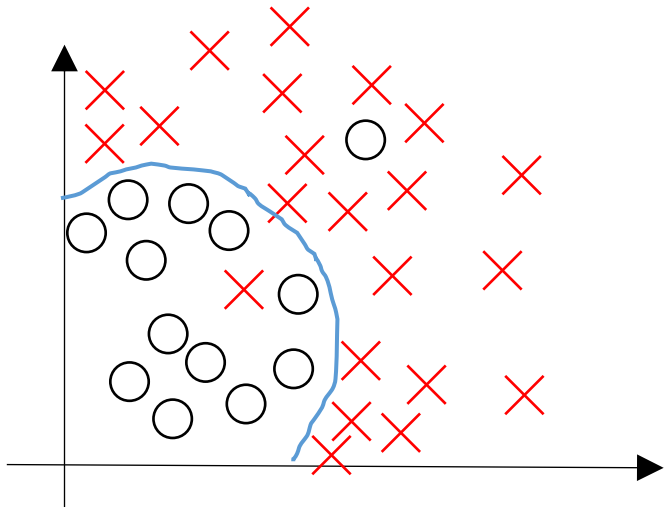
$$J(\boldsymbol{W}^{[L]}, \boldsymbol{b}^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{l}\|\boldsymbol{W}^{[l]}\|_2^2$$

$$\boldsymbol{W}^{[l]} \approx 0$$

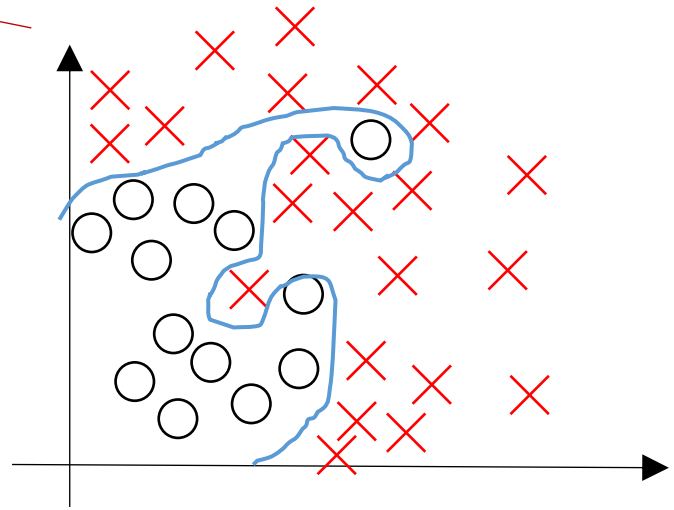
λ is a regularization parameter.



high bias



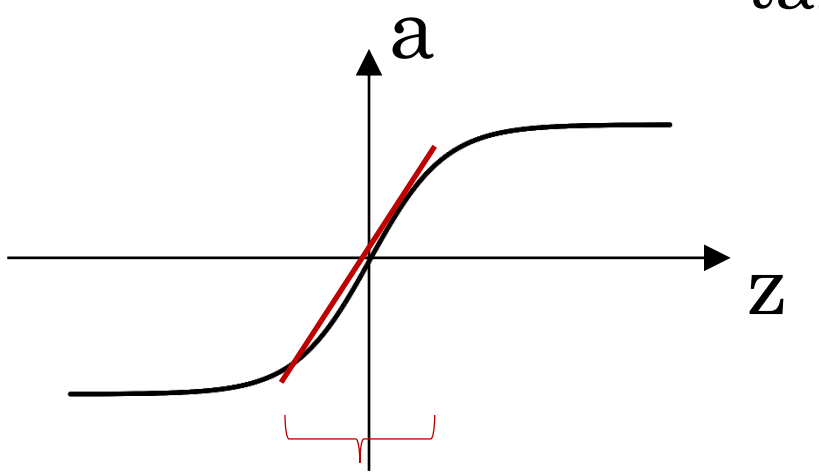
"just right"

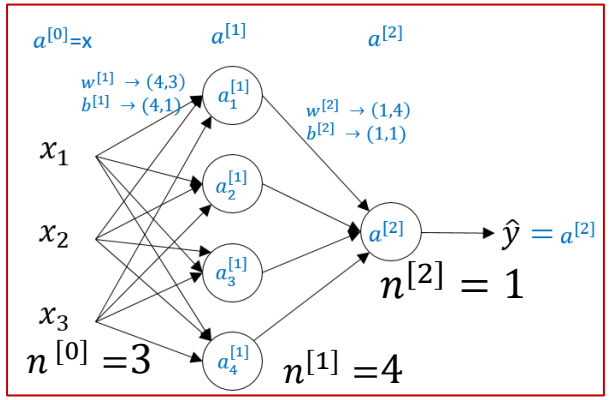


high variance

How does regularization prevent overfitting?

tanh: g(z) = tanh(z)





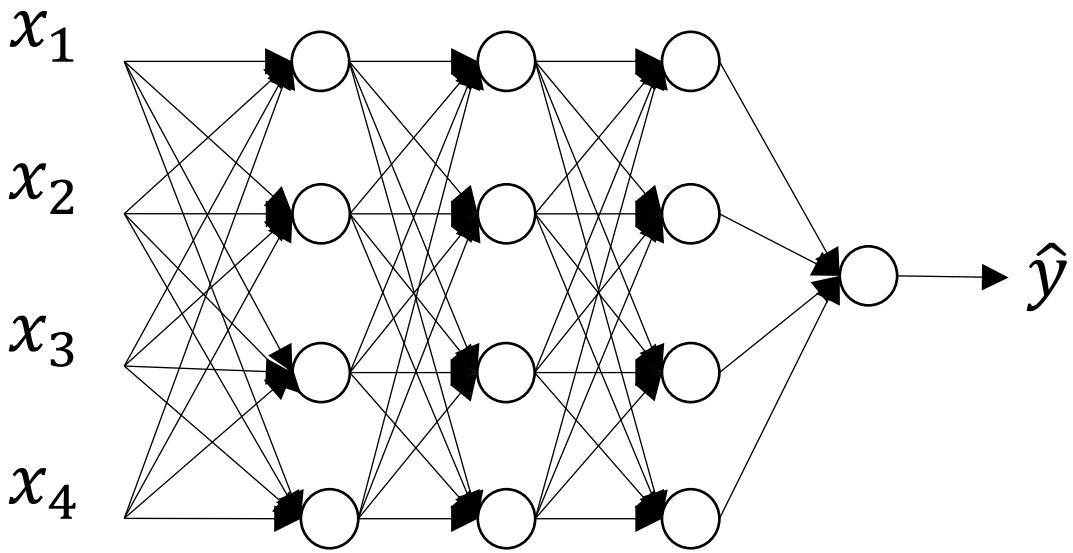
$$J(\boldsymbol{W}^{[L]}, \boldsymbol{b}^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{l=1}^{l}\|\boldsymbol{W}^{[l]}\|_2^2$$

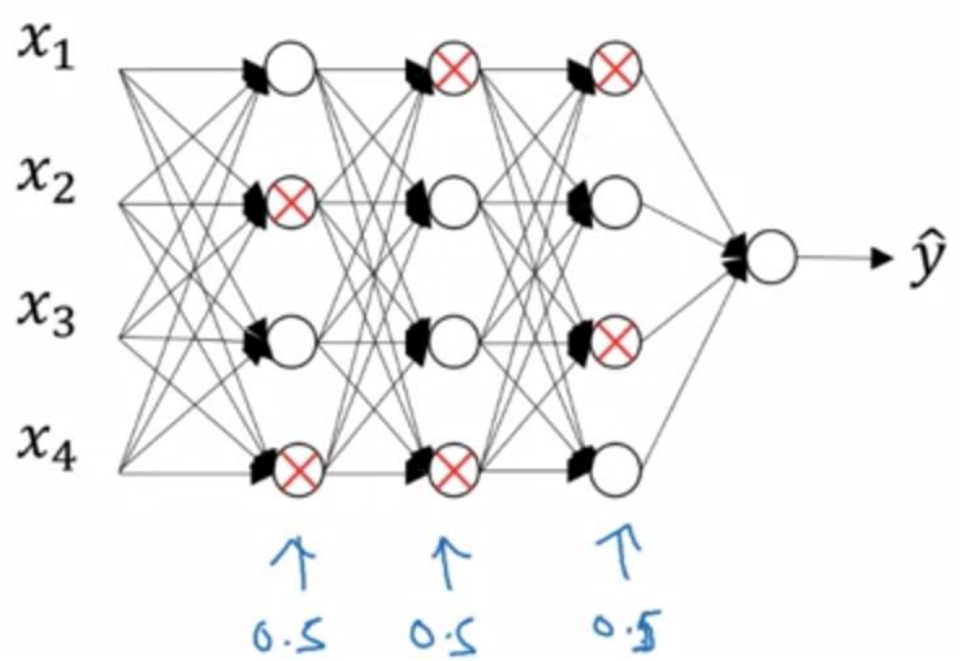
When λ increases, $\boldsymbol{W}^{[l]}$ decreases. So, $\boldsymbol{z}^{[l]}$ decreases. $\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}$

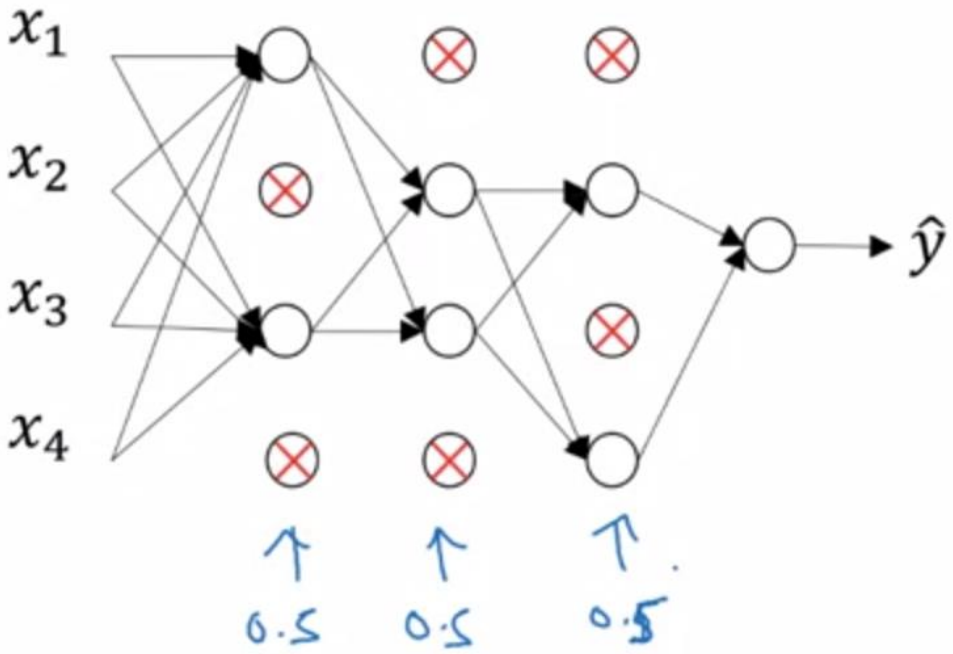
Every layer ≈ linear

Dropout regularization

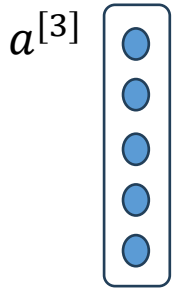
Dropout 是一种正则化方法，用于防止神经网络过拟合。
在训练时随机丢弃部分神经元，使得网络不会过分依赖某些特定神经元，提高模型的泛化能力。
在测试时不进行 Dropout，但要对神经元的激活值进行缩放，以匹配训练时的期望值。
Dropout rate（丢弃率）通常设为 0.2 - 0.5，根据不同的网络结构进行调整。







Implementing dropout



E.g., d3 = $\begin{bmatrix}[0.9]\\[0.6]\\[0.4]\\[0.2]\\[0.3]\end{bmatrix} < 0.8 = \begin{bmatrix}[0]\\[1]\\[1]\\[1]\\[1]\end{bmatrix}$

- Illustrate with layer **l** = 3 and **Keep-prob** = 0.8

  d3 = torch.rand(a3.shape[0],a3.shape[1]) < keep-prob

  a3 = torch.mul(a3,d3) # a3 *= d3

  a3 /= keep-prob

- When 5 units, 1 unit shuts off.

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

Reduced by 20%
So apply "/= 0.8"

E.g., $\underset{\text{a3}}{\begin{bmatrix}[0.5]\\[0.6]\\[0.9]\\[0.3]\\[0.5]\end{bmatrix}} * \underset{\text{d3}}{\begin{bmatrix}[1]\\[1]\\[0]\\[1]\\[1]\end{bmatrix}} = \underset{\text{a3}}{\begin{bmatrix}[0.5]\\[0.6]\\[0.0]\\[0.3]\\[0.5]\end{bmatrix}}$

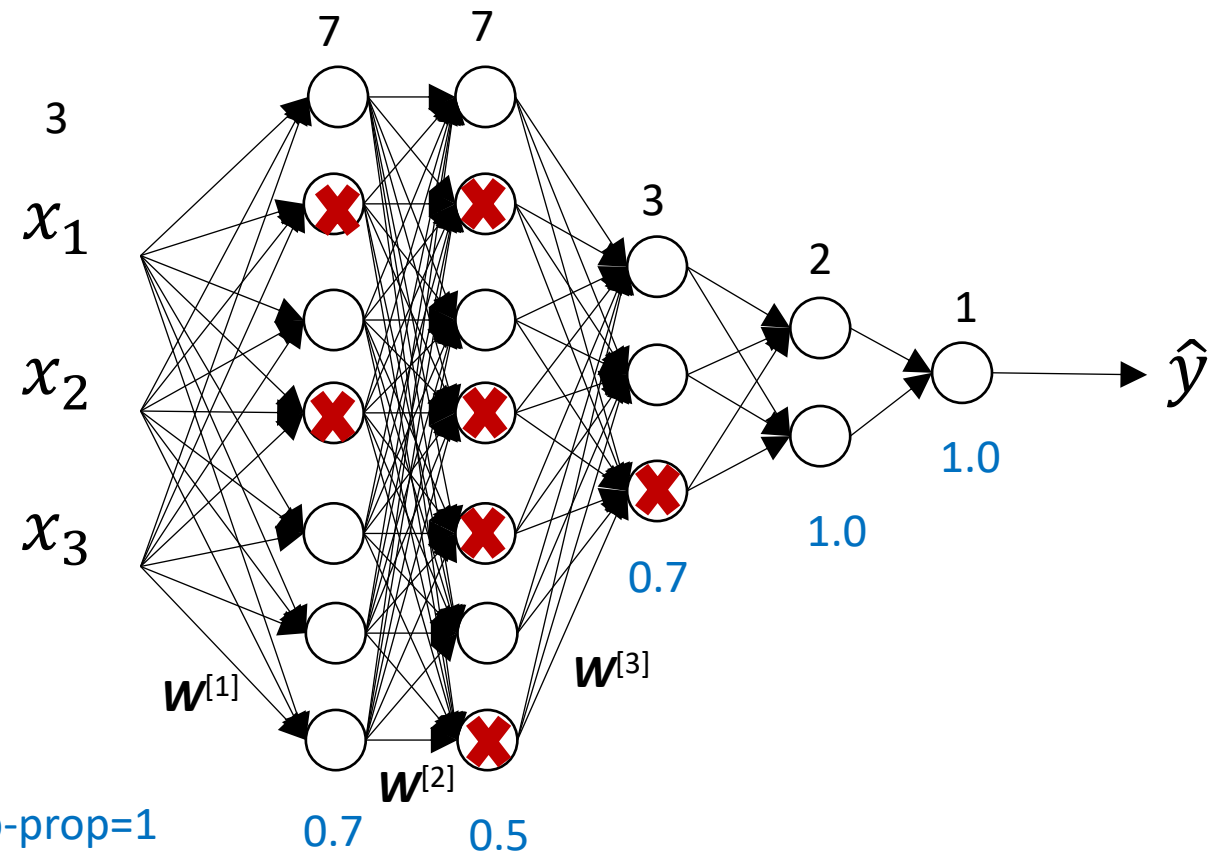
$\begin{bmatrix}[0.5]\\[0.6]\\[0.0]\\[0.3]\\[0.5]\end{bmatrix} / 0.8 = \begin{bmatrix}[0.625]\\[0.750]\\[0.000]\\[0.375]\\[0.625]\end{bmatrix}$

E.g., d3 = $\begin{bmatrix}[0.7]\\[0.5]\\[0.9]\\[0.5]\\[0.6]\end{bmatrix} < 0.8 = \begin{bmatrix}[1]\\[1]\\[0]\\[1]\\[1]\end{bmatrix}$

No drop out during test time

Why does drop-out work?

- Intuition: Can't rely on any one feature, so must spread out weights.
  - shrink weights like L_2 regularization



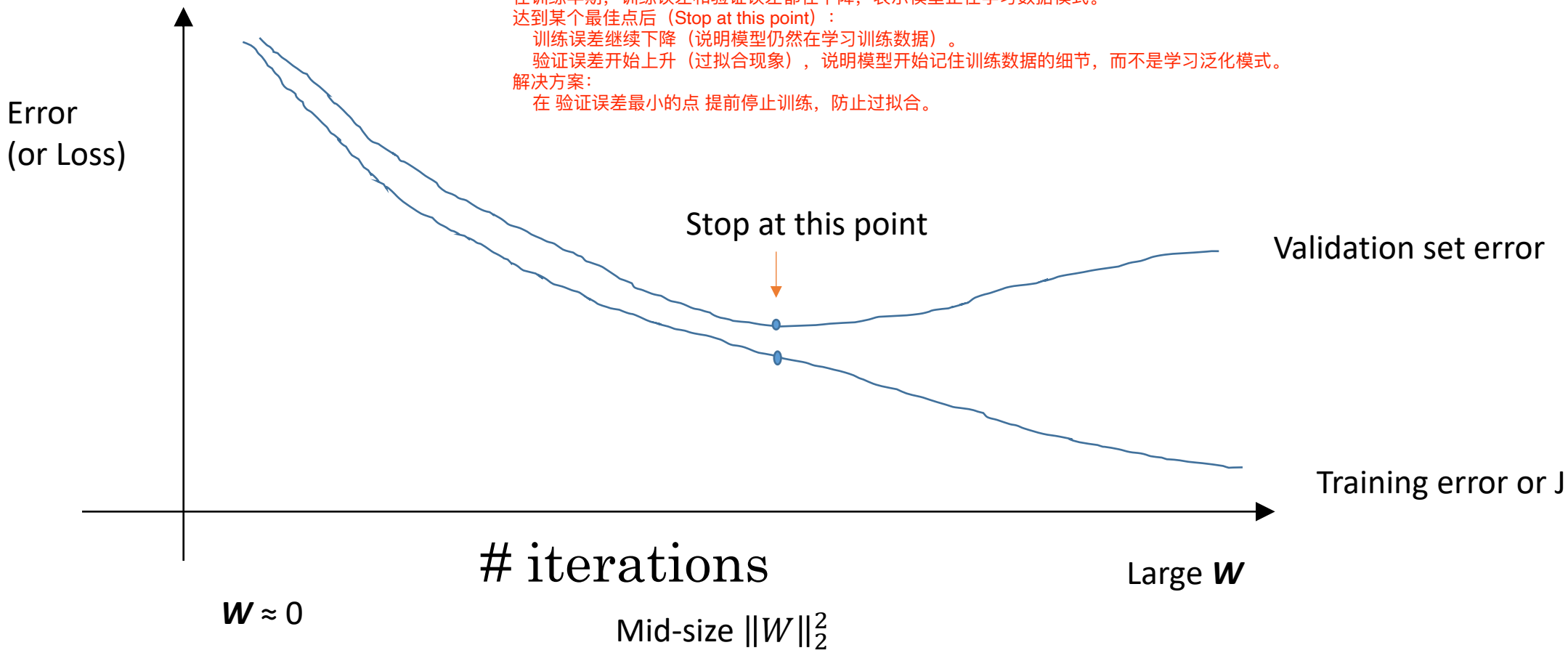
[since $W^{[2]}$ has more parameters, a small keep-prop value (0.5) is used]

Early stopping

图表解析

横轴（# iterations）：表示训练迭代次数（或模型复杂度），从 $w \approx 0$（初始权重接近 0）到较大的 W（训练后权重）。
纵轴（Error or Loss）：表示损失函数值，包括：
Training error（训练误差）：随着训练的进行，训练误差不断下降。
Validation set error（验证集误差）：起初下降，但在某个点后开始上升。
Early Stopping 关键点

在训练早期，训练误差和验证误差都在下降，表示模型正在学习数据模式。
达到某个最佳点后（Stop at this point）：
训练误差继续下降（说明模型仍然在学习训练数据）。
验证误差开始上升（过拟合现象），说明模型开始记住训练数据的细节，而不是学习泛化模式。
解决方案：
在 验证误差最小的点 提前停止训练，防止过拟合。



Normalizing inputs: Normalizing training sets

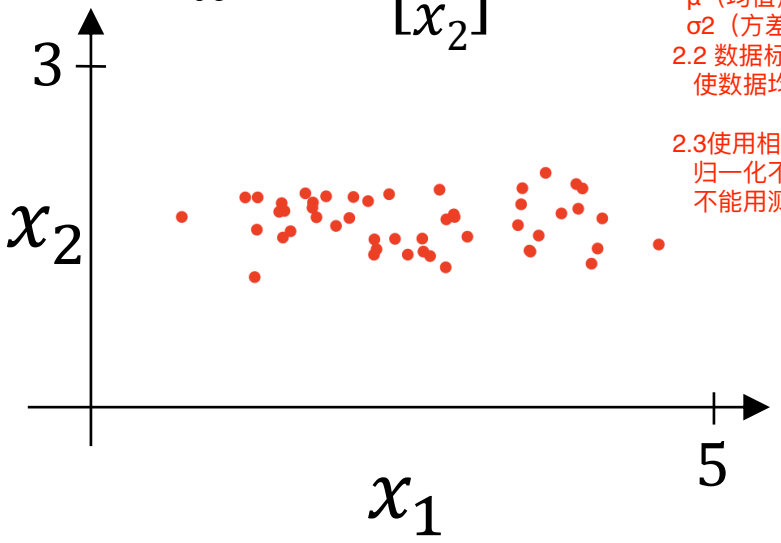
1. 什么是输入归一化？
输入归一化（Normalization）是指对数据进行标准化处理，使其具有：

均值 0：所有特征值围绕 0 分布
标准差 1：数据的尺度被统一，避免某些特征值过大而主导模型训练

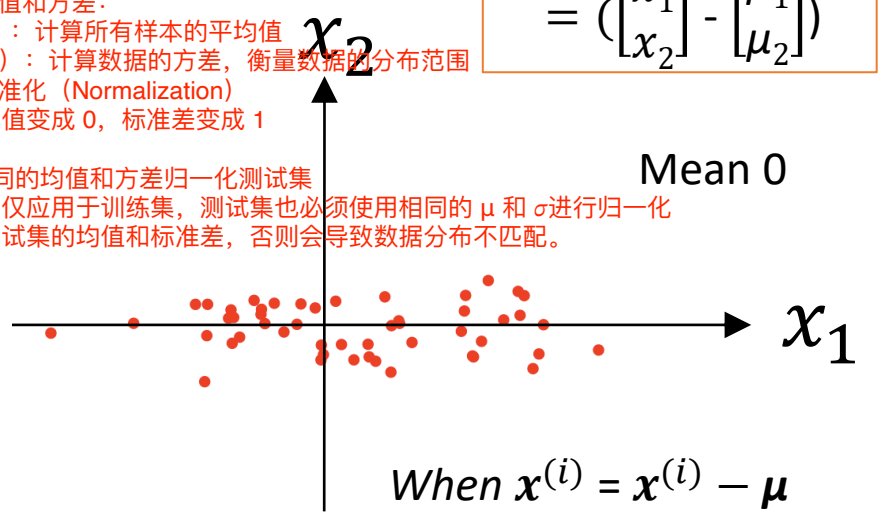
2. 归一化的步骤
2.1 计算均值和方差：
μ（均值）：计算所有样本的平均值
σ2（方差）：计算数据的方差，衡量数据的分布范围
2.2 数据标准化（Normalization）
使数据均值变成 0，标准差变成 1

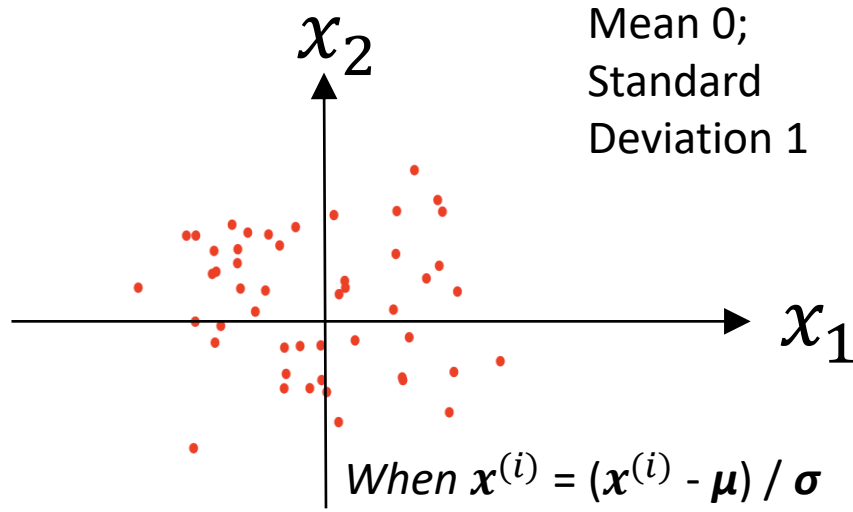
2.3使用相同的均值和方差归一化测试集
归一化不仅应用于训练集，测试集也必须使用相同的 μ 和 σ进行归一化
不能用测试集的均值和标准差，否则会导致数据分布不匹配。

$$x^{(i)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



$$x^{(i)} = (x^{(i)} - \mu) = (\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix})$$





E.g., use m number of $x^{(1)}$, $x^{(2)}$, …, $x^{(m)}$ to calculate μ_1 and μ_2

$$\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \leftarrow \mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$\begin{bmatrix} \sigma_1^2 \\ \sigma_2^2 \end{bmatrix} \leftarrow \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$$

Calculate mean and variance:

Normalize inputs:

$$x^{(i)} = (x^{(i)} - \mu) / \sigma$$

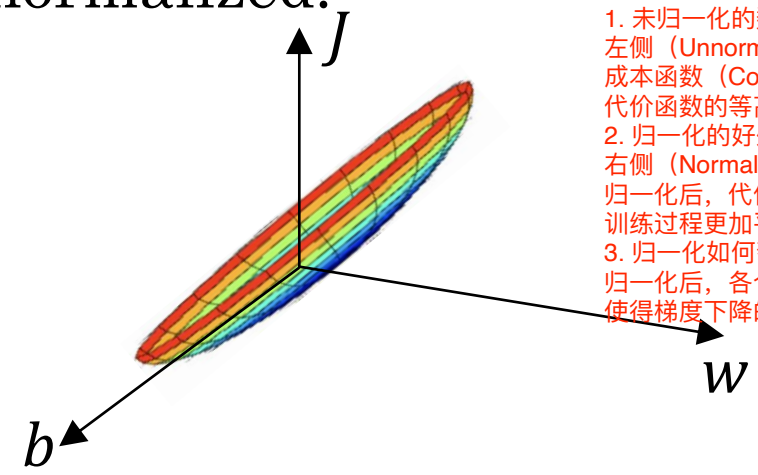
$$x^{(i)} = (x^{(i)} - \mu) / \sigma = (\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}) / \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$$

- Use the same μ and σ to normalize test set: $x^{(i)} = (x^{(i)} - \mu) / \sigma$

Why normalize inputs?

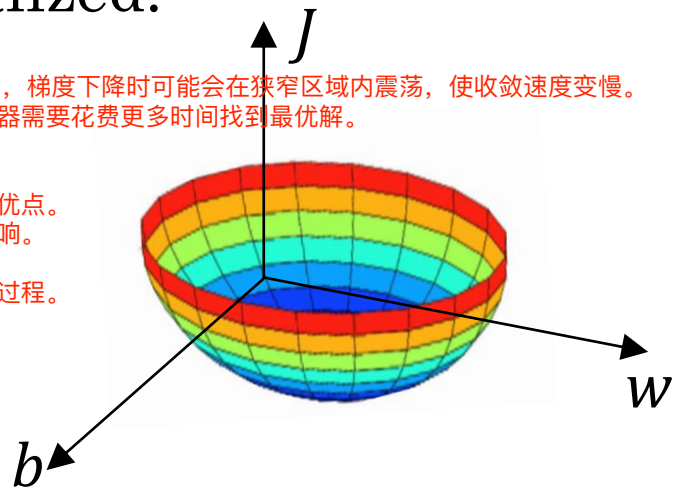
$$J(\boldsymbol{w}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

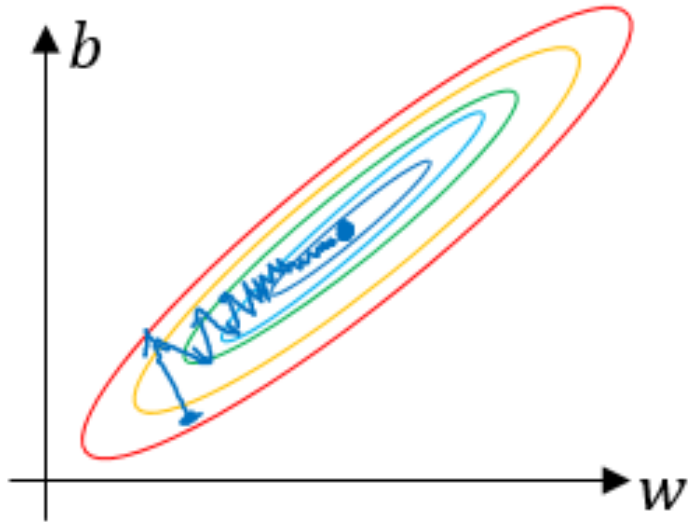
Unnormalized:



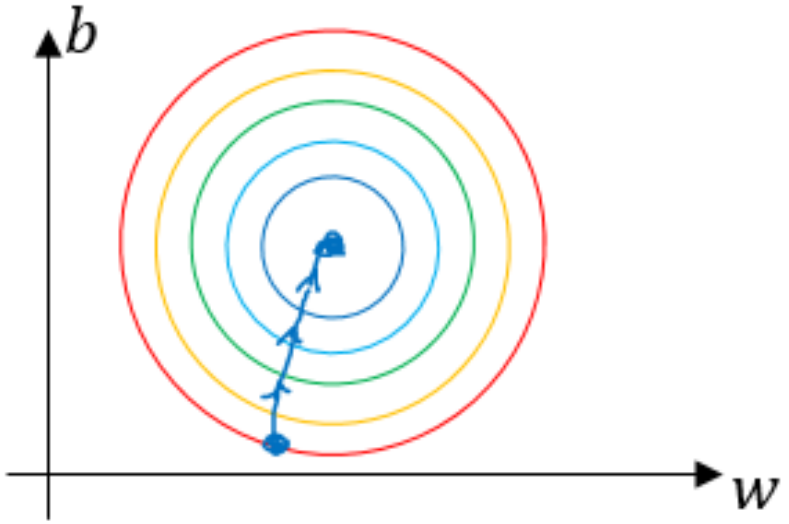
1. 未归一化的数据如何影响优化?
左侧（Unnormalized）：
成本函数（Cost function）变成拉长的椭圆形（elongated shape），梯度下降时可能会在狭窄区域内震荡，使收敛速度变慢。
代价函数的等高线呈现出长条形，导致梯度下降路径很曲折，优化器需要花费更多时间找到最优解。
2. 归一化的好处
右侧（Normalized）：
归一化后，代价函数变成对称的圆形，梯度下降能够快速收敛到最优点。
训练过程更加平稳，参数更新不会受到某些特征值过大或过小的影响。
3. 归一化如何帮助神经网络?
归一化后，各个特征的值处于相同的尺度，防止某些特征主导训练过程。
使得梯度下降的路径更加直接，提高模型收敛速度。

Normalized:

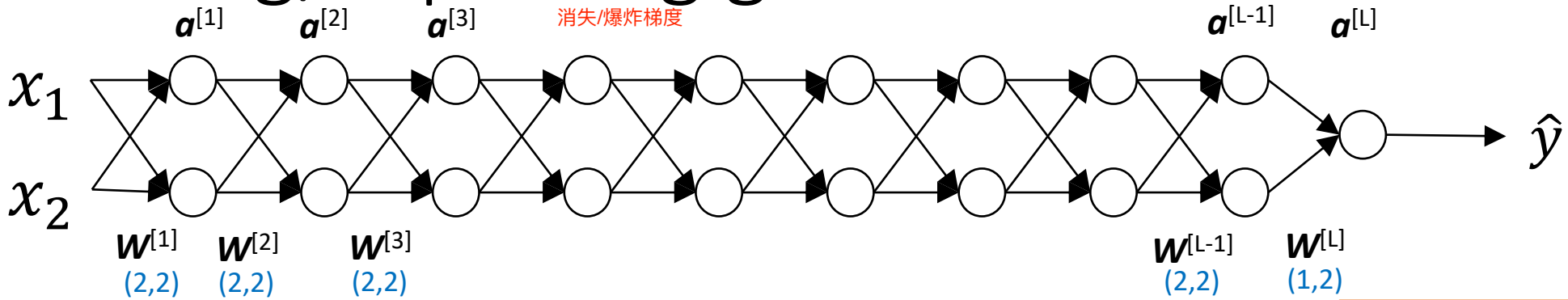




- Cost function becomes an **elongated shape** (long and thin)
- Take longer to learn weights or reach to an optimal point.



Vanishing/exploding gradients



Computing gradients:

$dz^{[2]} = a^{[2]} - y$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$

$dW^{[1]} = dz^{[1]} x^T$

$db^{[1]} = dz^{[1]}$

Through back propagation, ***W*** values are multiplied. So, derivatives can increase or decrease exponentially.

When $g(z) = z$, $b^{[l]} = 0$

$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \dots W^{[3]} W^{[2]} W^{[1]} x$

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

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

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

$a^{[2]} = g(z^{[2]}) = z^{[2]}$

1. 什么是梯度消失和梯度爆炸?
梯度消失（Vanishing Gradient）：
发生在深层网络的前层权重很小（<1）时，使得梯度在反向传播过程中指数级缩小，导致前层权重更新几乎为零，网络无法学习。
梯度爆炸（Exploding Gradient）：
发生在前层权重很大（>1）时，使得梯度在反向传播过程中指数级增长，导致梯度更新过大，模型发散（不收敛）。

ReLU / Leaky ReLU / GELU 代替 Sigmoid/Tanh 作为激活函数。
使用 He/Xavier 初始化 进行权重初始化，防止初始梯度过大或过小。
Batch Normalization 归一化数据，使梯度更稳定。
梯度裁剪（Gradient Clipping） 控制梯度大小，避免梯度爆炸。
ResNet 的残差连接 让梯度可以更容易传播，防止梯度消失。

$$\begin{vmatrix} 1.5 & 0 \\ 0 & 1.5 \end{vmatrix} \begin{vmatrix} 1.5 & 0 \\ 0 & 1.5 \end{vmatrix} = \begin{vmatrix} 1.5^2 & 0 \\ 0 & 1.5^2 \end{vmatrix}$$

$$\begin{vmatrix} 1.5 & 0 \\ 0 & 1.5^{l-1} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = 1.5^{l-1} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} = 1.5^{l-1} x$$

When $W^{[l]} = \begin{vmatrix} 1.5 & 0 \\ 0 & 1.5 \end{vmatrix}$ (i.e., $W^{[l]} > I$), $\hat{y} = W^{[L]} \begin{vmatrix} 1.5 & 0 \\ 0 & 1.5 \end{vmatrix}^{L-1} x = W^{[L]} 1.5^{L-1} x$ ($\hat{y}$ **increases exponentially**)

When $W^{[l]} = \begin{vmatrix} 0.5 & 0 \\ 0 & 0.5 \end{vmatrix}$ (i.e., $W^{[l]} < I$), $\hat{y} = W^{[L]} 0.5^{L-1} x$ ($\hat{y}$ **decreases exponentially**)

Weights Initialization

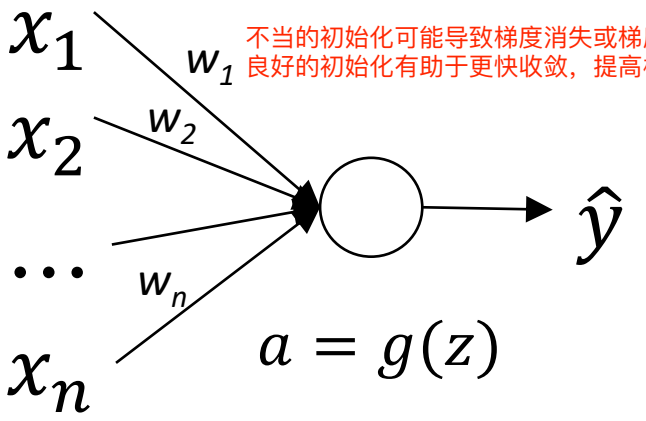
$n^{[l-1]}$ = n $n^{[l]}$ =1

在神经网络中，权重 w的初始化对训练效果有很大的影响：

不当的初始化可能导致梯度消失或梯度爆炸，使得网络难以收敛

良好的初始化有助于更快收敛，提高模型稳定性。

Single Neuron Example



When $n^{[l-1]}$ = 5, $n^{[l]}$ = 3

```
w1=np.random.randn(3,5) * np.sqrt (2/5)
w1
```

```
array([[-0.22929147,  0.6993235 , -0.10430622, -0.2149747 ,  1.06357577],
       [ 0.33875831,  0.62626452, -0.94417563, -0.72660319, -0.79194391],
       [ 0.15885362,  0.46539225, -0.76860632,  0.08878817,  0.2415148 ]])
```

```
np.var(w1)
```

2/5 = 0.4

```
0.3454423104406922
```

np.sqrt(2/5) ≈ 0.63

When $n^{[l-1]}$ = 100 , $n^{[l]}$ = 3

```
w2=np.random.randn(3,100) * np.sqrt (2/100)
np.var(w2)
```

```
0.018709112797490716
```

2/100 = 0.02

np.sqrt(2/100) ≈ 0.14

$z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$ (ignore b)

When $n^{[l-1]}$ is large, w_i becomes smaller.

Variance(w_i) = $\frac{2}{n^{[l-1]}}$

shape of ***W***: $(n^{[l]}, n^{[l-1]})$

✅好的权重初始化方法可以：

防止梯度消失或爆炸，提高训练稳定性。
加速网络收敛，减少训练时间。
适配不同激活函数，确保信息能够有效传递。
🚀在深度学习中，合理选择 权重初始化方式 是训练成功的重要一步！

$\boldsymbol{W}^{[l]}$ = np.random.randn(shape of ***W***) * np.sqrt ($\frac{2}{n^{[l-1]}}$) (with Relu)

$\boldsymbol{W}^{[l]}$ = np.random.randn(shape of ***W***) * np.sqrt ($\frac{1}{n^{[l-1]}}$) (with tanh, called Xavier initialization)

$\boldsymbol{W}^{[l]}$ = np.random.randn(shape of ***W***) * np.sqrt ($\frac{2}{n^{[l-1]}+n^{[l]}}$) (another approach)

Mini-batch gradient descent:
Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

What if m = 5,000,000 ?

-> 5,000 mini-batches of 1,000 each

-> Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$

📌 Mini-batch Gradient Descent 是深度学习中最常用的优化方法：

数据集分成多个 mini-batch，每个 batch 计算一次梯度更新，比 Batch 计算更快，比 SGD 更稳定。
利用向量化计算提升 GPU 计算效率。
适用于大规模数据集训练（如 500 万样本），可以有效加速收敛。

$$X = [\underbrace{x^{(1)}, x^{(2)}, \ldots, x^{(1000)}}_{X^{\{1\}}\ (n_x,1000)} \mid \underbrace{x^{(1001)}, \ldots, x^{(2000)}}_{X^{\{2\}}\ (n_x,1000)} \mid \ldots \mid \underbrace{\ldots x^{(m)}}_{X^{\{5000\}}\ (n_x,1000)}]$$

(n_x,m)

$$Y = [\underbrace{y^{(1)}, y^{(2)}, \ldots, y^{(1000)}}_{Y^{\{1\}}\ (1,1000)} \mid \underbrace{y^{(1001)}, \ldots, y^{(2000)}}_{Y^{\{2\}}\ (1,1000)} \mid \ldots \mid \underbrace{\ldots y^{(m)}}_{Y^{\{5000\}}\ (1,1000)}]$$

$(1,m)$

Mini-batch gradient descent

Epoch: pass through training set

Repeat {

for t =1, …, 5000 {

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

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

$$\boldsymbol{A}^{[1]} = g^{[1]}\left(\boldsymbol{Z}^{[1]}\right)$$

……

$$\boldsymbol{A}^{[L]} = g^{[L]}\left(\boldsymbol{Z}^{[L]}\right)$$

Vectorised implementation (1000 example)

Compute cost $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{1000}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2*1000}\|\boldsymbol{w}\|_2^2$

Backprop to compute gradients with regard to $J^{\{t\}}$

$\boldsymbol{W}^{[l]} = \boldsymbol{W}^{[l]} - \alpha\boldsymbol{dW}^{[l]}$, $\boldsymbol{b}^{[l]} = \boldsymbol{b}^{[l]} - \alpha\boldsymbol{db}^{[l]}$

}

}

✅比 Batch Gradient Descent 计算更高效

Batch Gradient Descent 需要用整个数据集计算梯度更新，计算量大。
Mini-batch Gradient Descent 每次只计算 1000 个样本，计算更快。
✅比 Stochastic Gradient Descent 更稳定

SGD（随机梯度下降） 由于每次只用 1 个样本计算梯度，噪声较大。
Mini-batch 计算 1000 个样本的均值梯度，梯度更新更平滑，收敛更稳定。
✅利用向量化计算，提高 GPU 计算效率

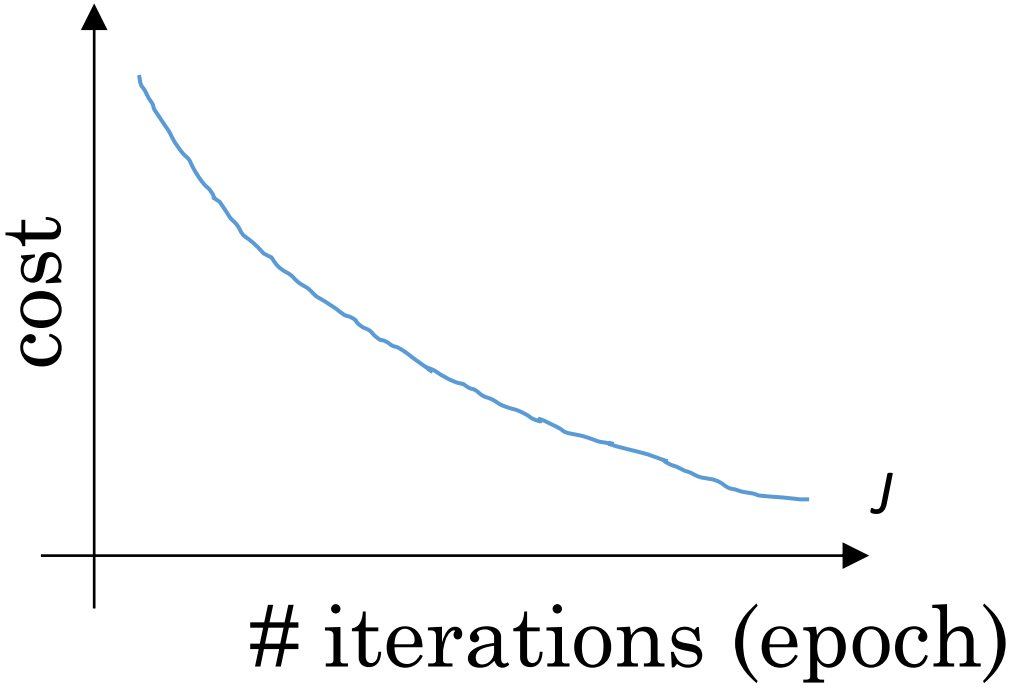
批量处理 1000 个样本，可以并行计算，大幅度提高训练速度。

1 step of gradient descent using $\boldsymbol{X}^{\{t\}}, \boldsymbol{Y}^{\{t\}}$

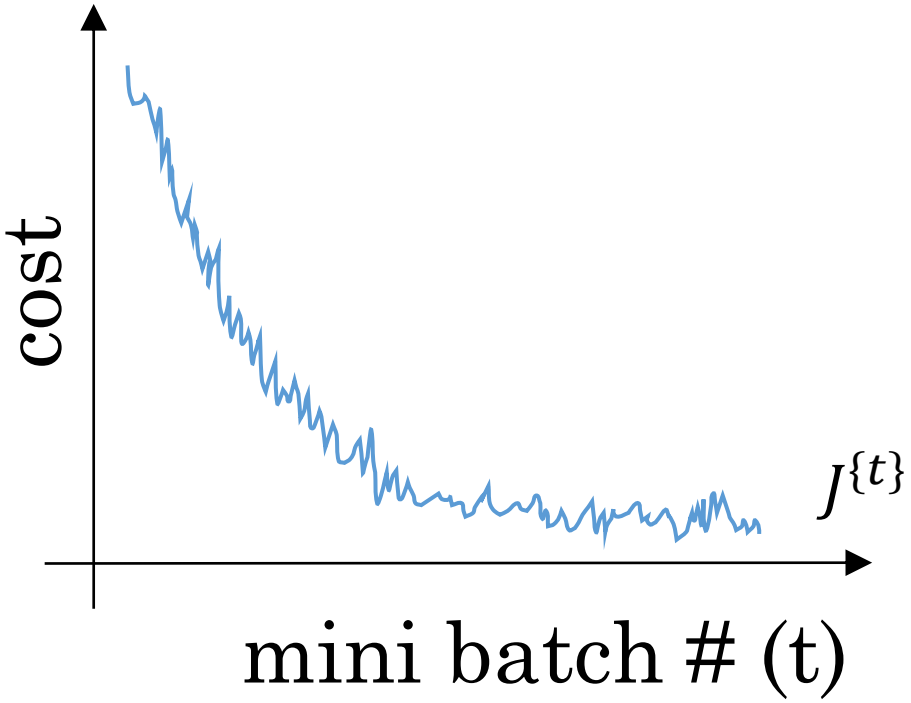
Understanding mini-batch gradient descent:

Training with mini batch gradient descent

Batch gradient descent



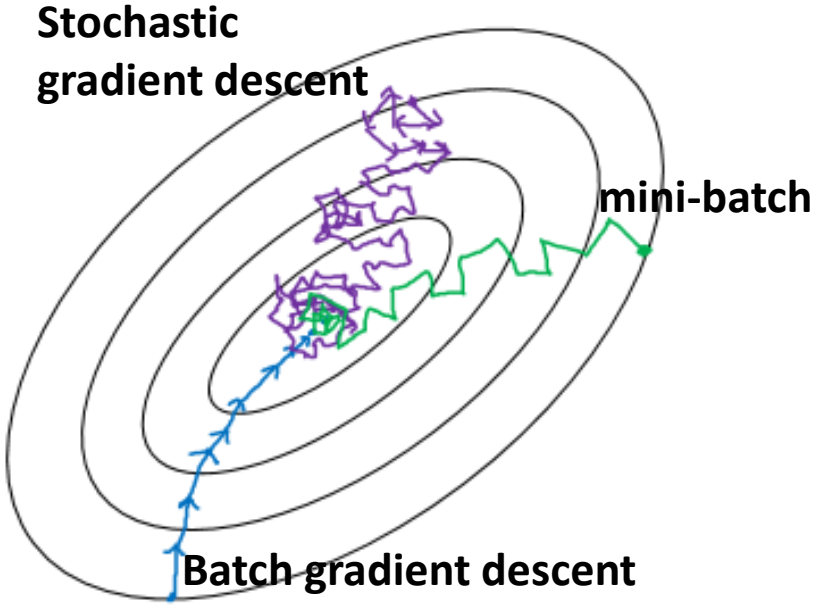
Mini-batch gradient descent



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

Choosing your mini-batch size

- If mini-batch size = *m*, **batch** gradient descent $(\boldsymbol{X}^{\{t\}}, \boldsymbol{Y}^{\{t\}}) = (\boldsymbol{X}, \boldsymbol{Y})$
- If mini-batch size = 1, **stochastic** gradient descent $(\boldsymbol{X}^{\{t\}}, \boldsymbol{Y}^{\{t\}}) = (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$
- In practice, somewhere in-between 1 and *m*



Stochastic gradient descent

Lose speedup from vectorization

In-between (mini-batch size not too big/small)

Fastest learning

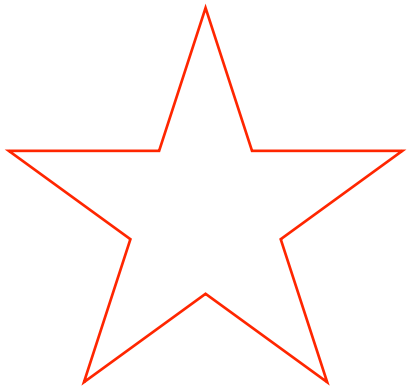
- vectorization (e.g., 1000)
- Make process without processing entire training set

Batch gradient descent

Too long per iteration

Choosing your mini-batch size

- If small training set, use **batch** gradient descent (m <= 2000)
- Typical mini-batch sizes:
  - 64 (2^6), 128 (2^7), 256 (2^8), and 512 (2^9)
- Make sure mini-batch ($\boldsymbol{X}^{\{t\}}$, $\boldsymbol{Y}^{\{t\}}$) fit in CPU/GPU memory.



Exponentially weighted (running) averages:

Temperature in a city

$\theta_1 = 40°\text{F}$

$\theta_2 = 49°\text{F}$

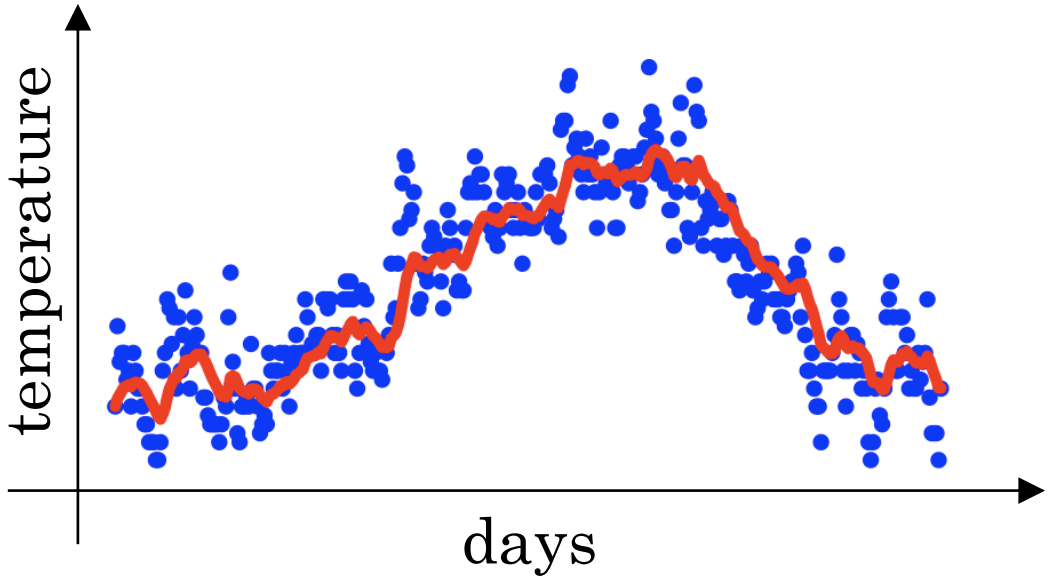
$\theta_3 = 45°\text{F}$

$\vdots$

$\theta_{180} = 60°\text{F}$

$\theta_{181} = 56°\text{F}$

$\vdots$



$V_0 = 0$

$V_1 = 0.9\, V_0 + 0.1\, \theta_1$

$V_2 = 0.9\, V_1 + 0.1\, \theta_2$

...

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

Exponentially weighted averages

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

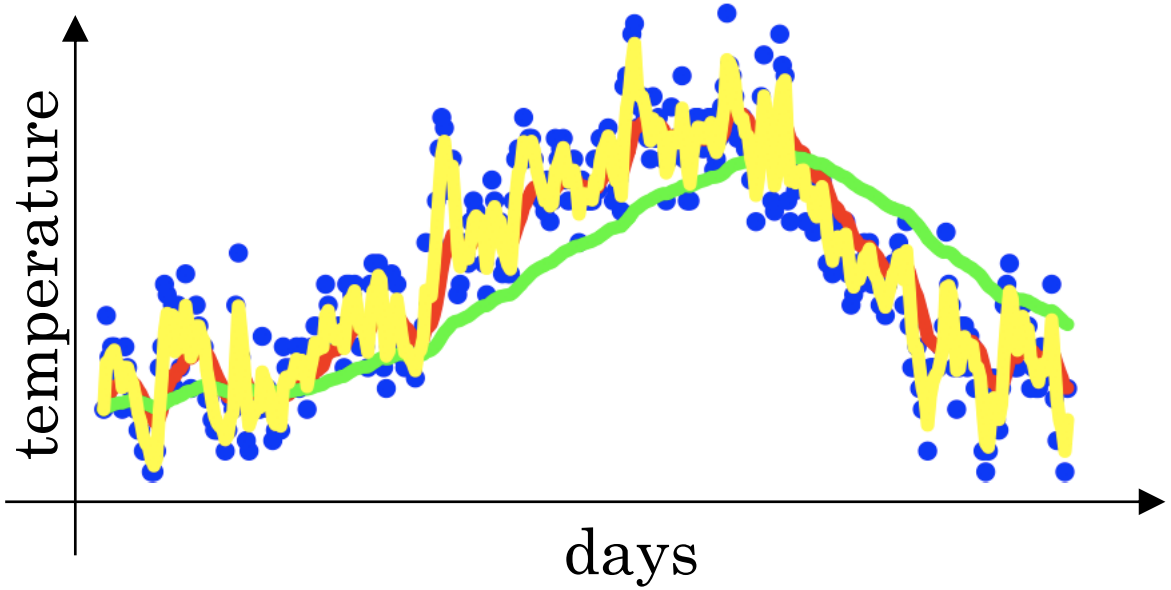
V_t is approximately averaged over

$\approx 1/(1-\beta)$ days' temperature

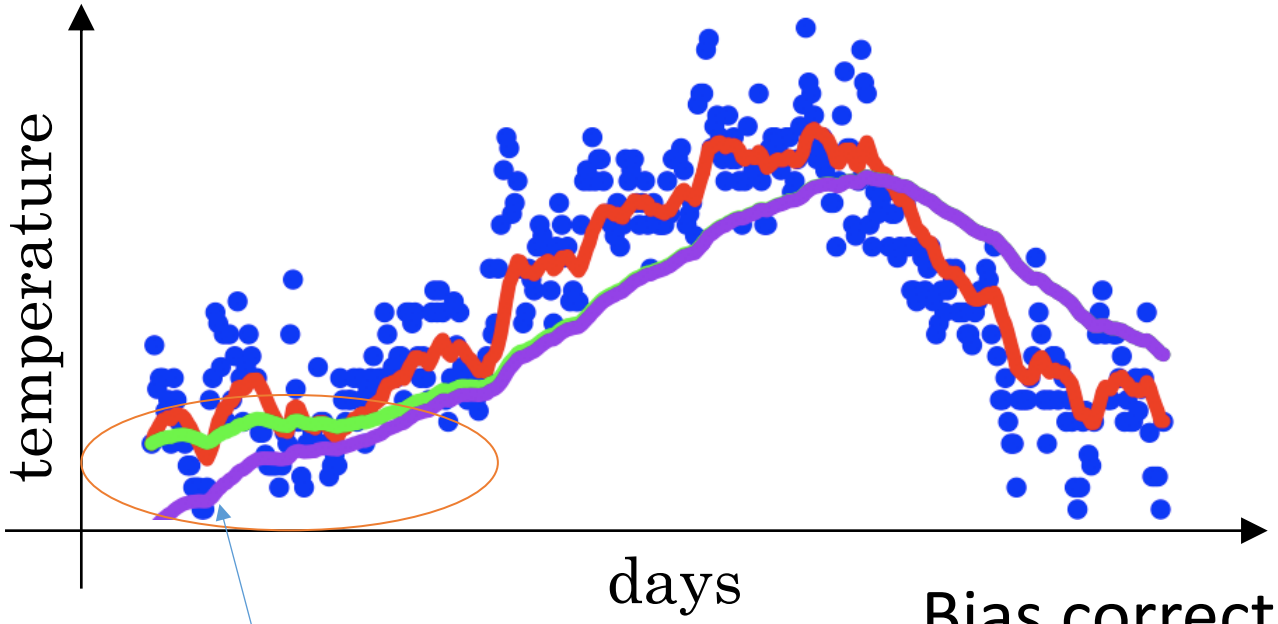
When $\beta = 0.9$, $\approx$ 10 days temperature (red)

$\beta = 0.98$, $\approx$ 50 days temperature (green)

$\beta = 0.5$, $\approx$ 2 days temperature (yellow)



Bias correction in exponentially weighted average



when β = 0.98

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

$$v_0 = 0$$
$$v_1 = 0.98v_0 + 0.02\theta_1 = 0.02\theta_1$$
$$v_2 = 0.98v_1 + 0.02\theta_2$$
$$= 0.0196\theta_1 + 0.02\theta_2$$

(**lower values in early stage**, purple line)

Bias correction: $v_t = \frac{v_t}{1-\beta^t}$

When t=1, $1-\beta^1$ =0.02

$$corrected\ v_1 = \frac{v_1}{1-\beta^1} = \frac{0.02\theta_1}{0.02} = \theta_1$$

When t=2, $1-\beta^2$ =0.0396

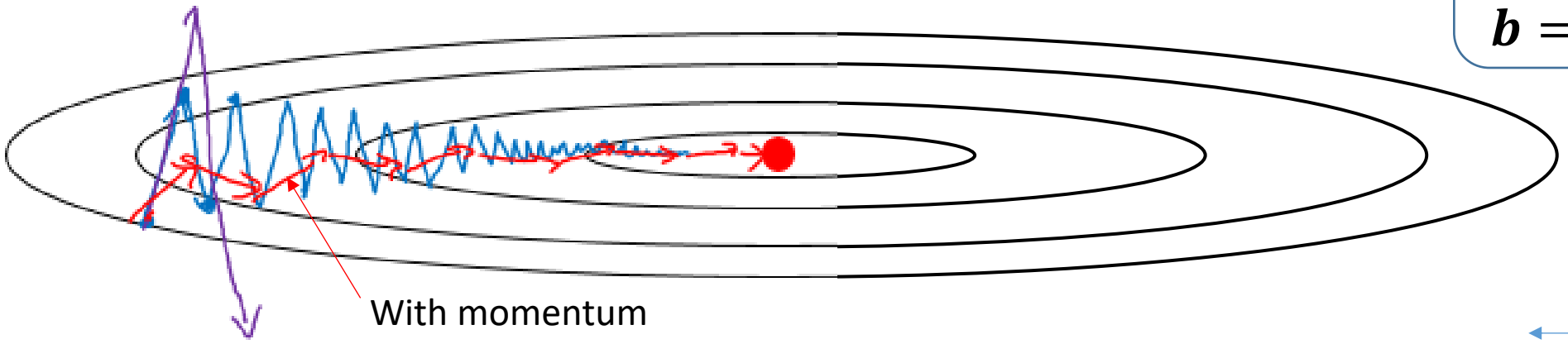
$$corrected\ v_2 = \frac{v_2}{1-\beta^2} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

Gradient descent with momentum uses exponentially weighted average

Basic approach:

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{dW}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha \boldsymbol{db}$$



$\boldsymbol{V_{dW}} = 0, \boldsymbol{v_{db}} = 0$ # initialize running averages

On iteration t:

Compute $\boldsymbol{dW}, \boldsymbol{db}$ on the current mini-batch

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

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

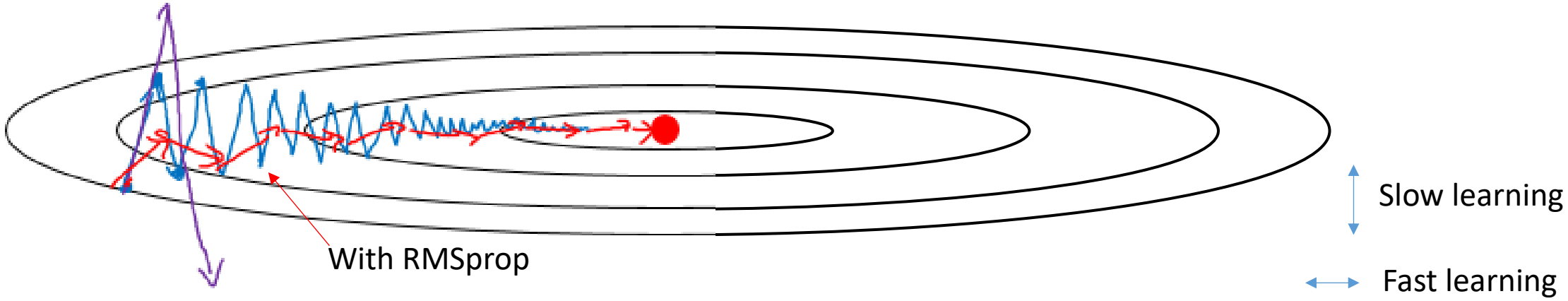
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{V_{dW}}, \ \boldsymbol{b} = \boldsymbol{b} - \alpha \boldsymbol{v_{db}}$$

Hyperparameters: α, β

$\beta = 0.9$

Average over last ≈10 gradients

RMSprop (Root Mean Square Propagation)



$\boldsymbol{S_{dw}} = 0, \boldsymbol{s_{db}} = 0$ # initialize running averages

On iteration t:

Compute $\boldsymbol{dW}, \boldsymbol{db}$ on the current mini-batch

$$\boldsymbol{S_{dW}} = \beta \boldsymbol{S_{dW}} + (1-\beta)\boldsymbol{dW}^2$$ ← Element-wise square of ***dW*** (or ***db***)

← small

$$\boldsymbol{s_{db}} = \beta \boldsymbol{s_{db}} + (1-\beta)\boldsymbol{db}^2$$ ← large

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{\boldsymbol{dW}}{\sqrt{\boldsymbol{S_{dW}} + \varepsilon}}, \quad \boldsymbol{b} = \boldsymbol{b} - \alpha \frac{\boldsymbol{db}}{\sqrt{\boldsymbol{s_{db}} + \varepsilon}}, \quad \varepsilon = 10^{-8}$$

Adam (*Adaptive moment estimation*) optimization algorithm

$\boldsymbol{V_{dW}} = 0, \boldsymbol{S_{dw}} = 0, \boldsymbol{v_{db}} = 0, \boldsymbol{s_{db}} = 0$

On iteration t:

Compute $\boldsymbol{dW}, \boldsymbol{db}$ on the current mini-batch

$\boldsymbol{V_{dW}} = \beta_1 \boldsymbol{V_{dW}} + (1-\beta_1)\boldsymbol{dW}, \ \boldsymbol{v_{db}} = \beta_1 \boldsymbol{v_{db}} + (1-\beta_1)\boldsymbol{db}$ (from momentum β_1)

$\boldsymbol{S_{dW}} = \beta_2 \boldsymbol{S_{dW}} + (1-\beta_2)\boldsymbol{dW}^2, \ \boldsymbol{s_{db}} = \beta_2 \boldsymbol{s_{db}} + (1-\beta_2)\boldsymbol{db}^2$ (from RMSprop β_2)

$$\boldsymbol{V_{dw}^{corrected}} = \frac{\boldsymbol{V_{dW}}}{1-\beta_1^t}, \ \boldsymbol{v_{db}^{corrected}} = \frac{\boldsymbol{v_{db}}}{1-\beta_1^t}$$

$$\boldsymbol{S_{dw}^{corrected}} = \frac{\boldsymbol{S_{dW}}}{1-\beta_2^t}, \ \boldsymbol{s_{db}^{corrected}} = \frac{\boldsymbol{s_{db}}}{1-\beta_2^t}$$

Hyperparameters choice:

α : needs to be tune

β_1: 0.9

β_2: 0.999

ε : 10^{-8}

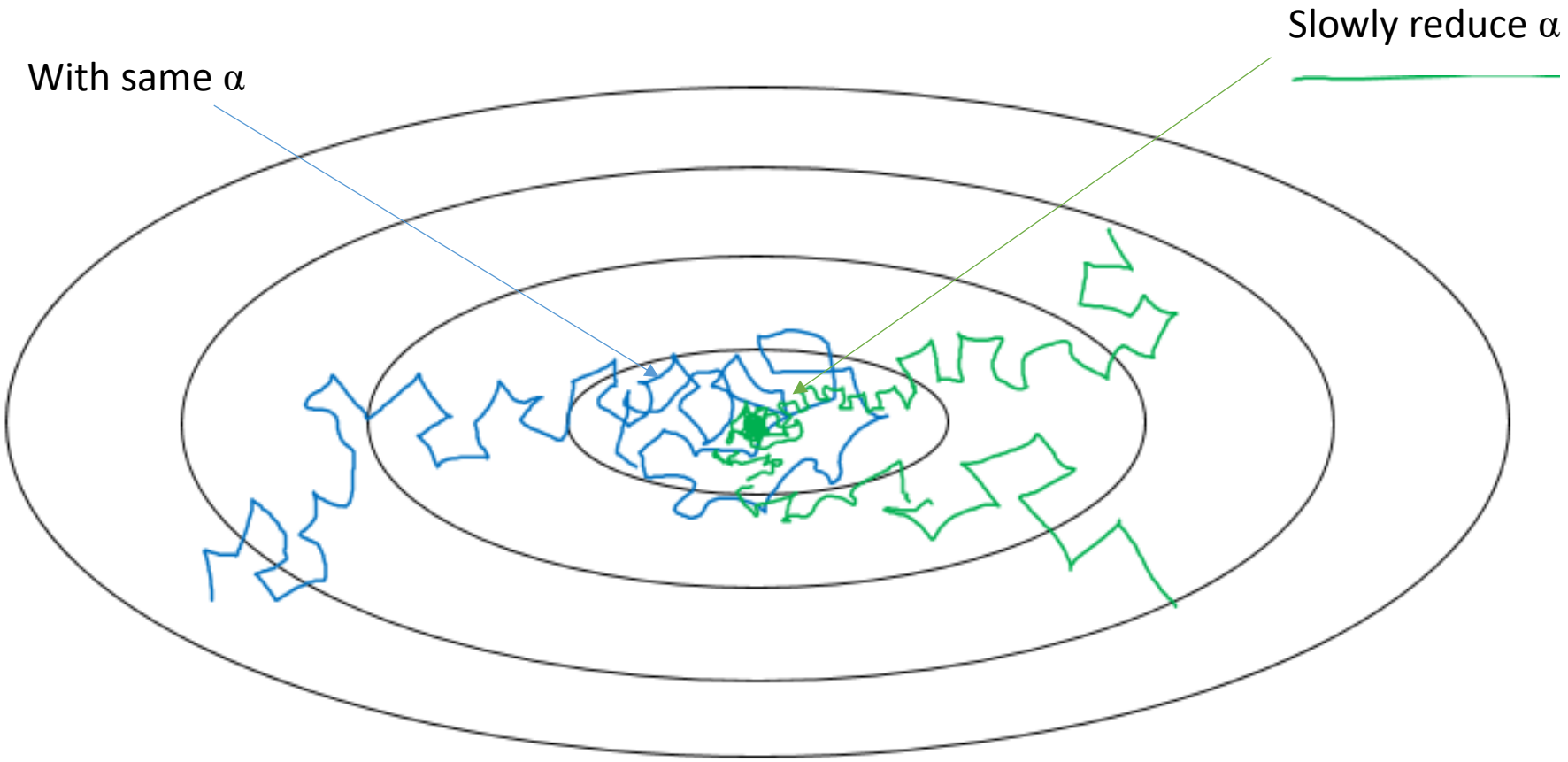
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{\boldsymbol{V_{dw}^{corrected}}}{\sqrt{\boldsymbol{S_{dw}^{corrected}} + \varepsilon}}, \qquad \boldsymbol{b} = \boldsymbol{b} - \alpha \frac{\boldsymbol{v_{db}^{corrected}}}{\sqrt{\boldsymbol{s_{db}^{corrected}} + \varepsilon}}$$

Learning rate decay

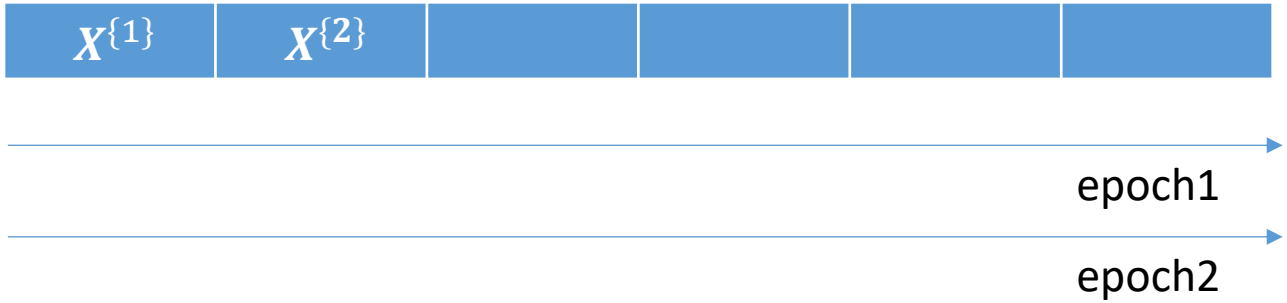
Basic approach:

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{dW}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha \boldsymbol{db}$$



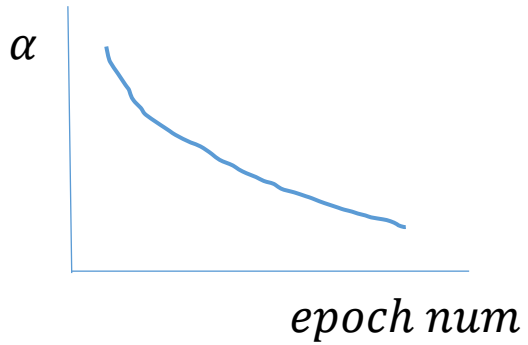
Learning rate decay



1 epoch = 1 pass through data

$$\alpha = \frac{1}{1 + decay{-}rate * epoch{-}num}\,\alpha_0$$, where α_0=0.2 ; *decay-rate*=1

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



Basic approach:

$$\boldsymbol{W = W - \alpha dW}$$

$$\boldsymbol{b = b - \alpha db}$$

Tuning Process: Hyperparameters

α (the most important)

β (the second most important)

# of hidden units (the second most important)

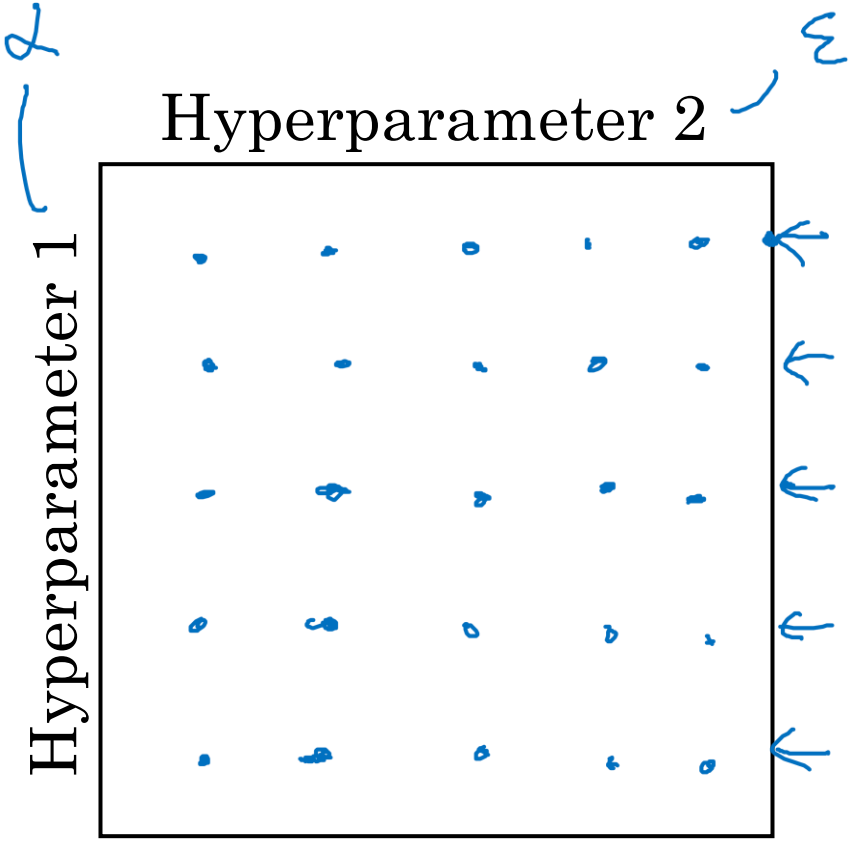
Mini-batch size (the second most important)

# of layers (the third most important)

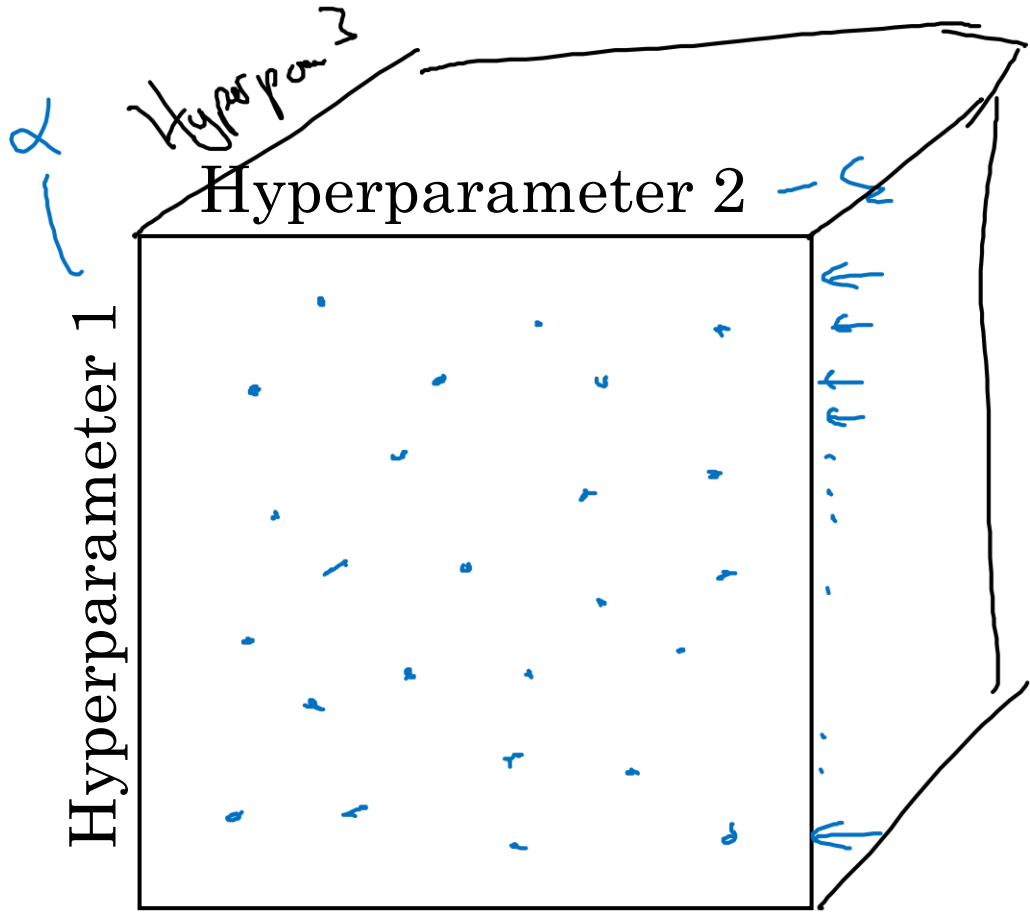
Learning rate decay (the third most important)

β_1, β_2, ε in adam

Try random values: Don't use a grid

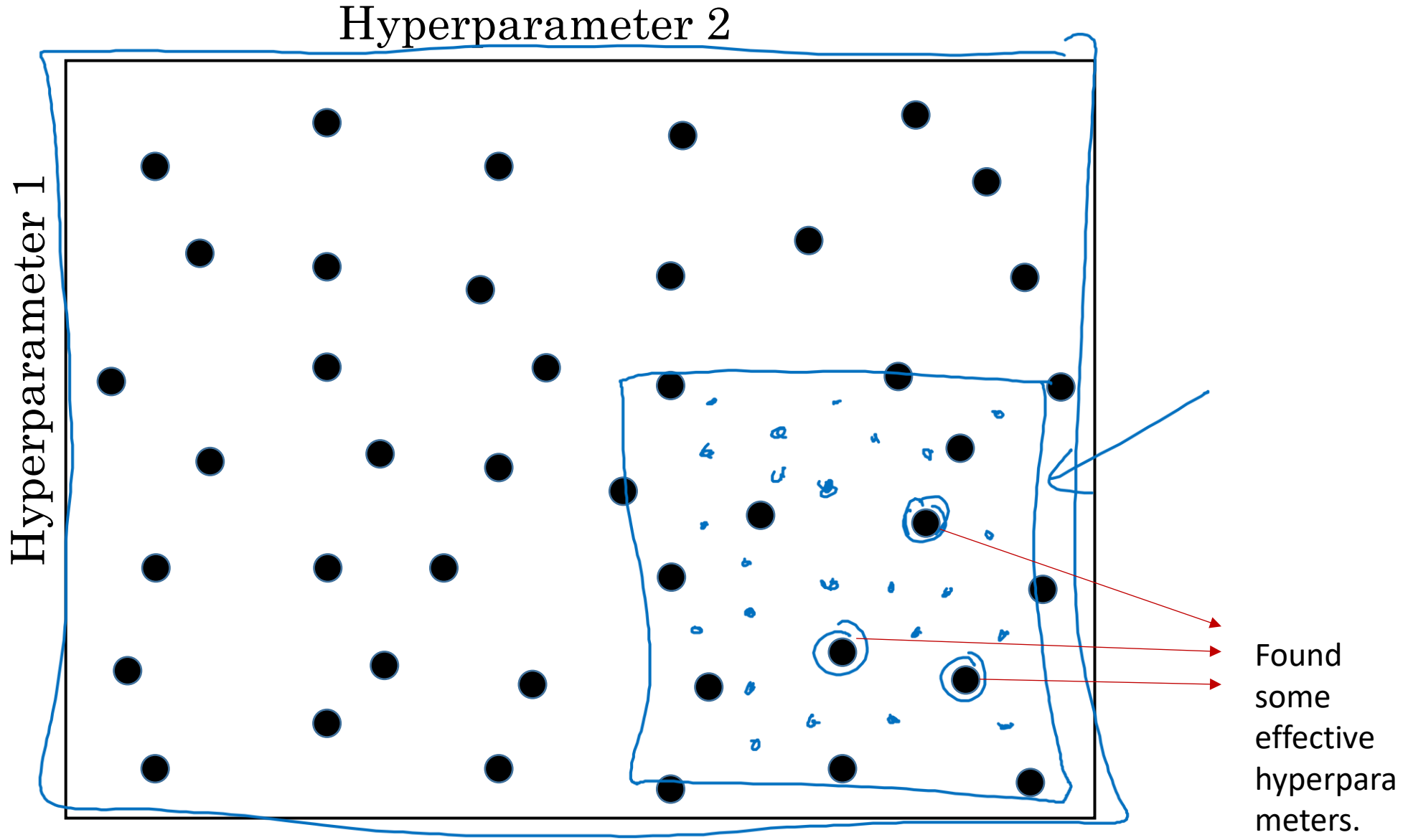


Consider only 5 different α values



Consider 25 different α values
(α is more important than ε)

Coarse to fine



Using an appropriate scale to pick hyperparameters:
Picking hyperparameters at random

# of hidden units: $N^{[l]}$ = 50, ……., 100

# of layers: L = 2 – 4

2, 3, 4

Appropriate scale for hyperparameters

Learning Rate: $\alpha = 0.0001, \ldots\ldots, 1$

10^a

$a = \log_{10} 0.0001 = -4$

10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0}

10^b

$b = \log_{10} 1 = 0$

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

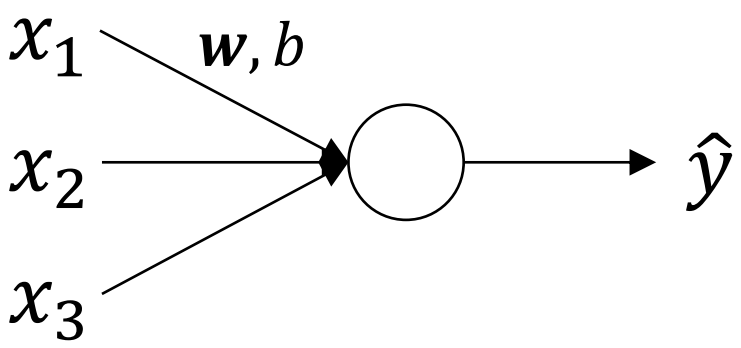
$\alpha = 10^r$ ← 10^{-4}.....10^0

10^a.....10^b ; $r \in [a,b]$; $\alpha = 10^r$ where r = numpy.random.uniform(*low=a*, *high=b*)

Batch Normalization: Normalizing activations in a network

$$x^{(i)} = (x^{(i)} - \mu) / \sigma = (\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}) / \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \end{bmatrix}$$

Normalizing inputs to speed up learning



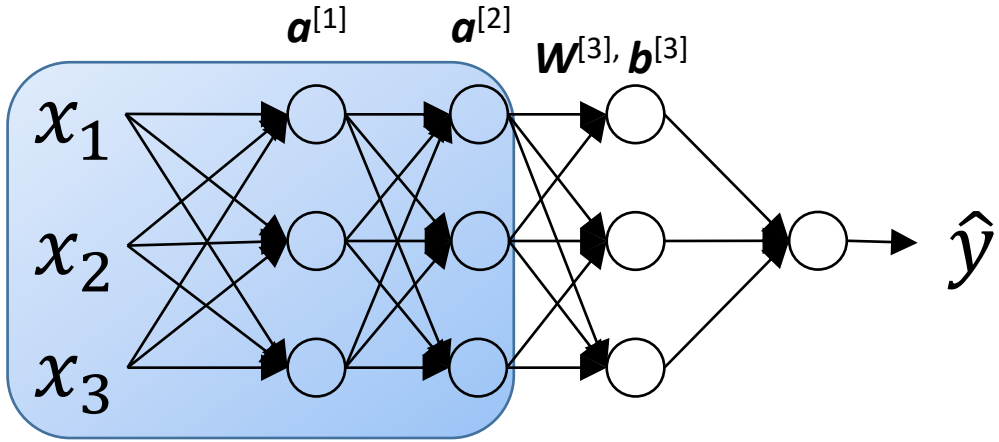
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

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

$$x^{(i)} = (x^{(i)} - \mu) / \sigma$$

E.g., use *m* number of x_1, x_2, x_3 to calculate μ_1, μ_2, μ_3

Element-wise



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

-> need to normalize $z^{[2]}$

Implementing Batch Norm

Given some intermediate values in NN, $z^{(1)}, z^{(2)}, \ldots\ldots, z^{(m)}$, in a layer L.

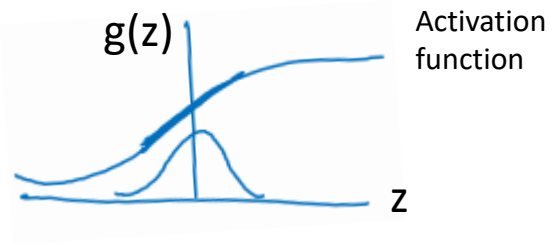
$$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{(i)}$$

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

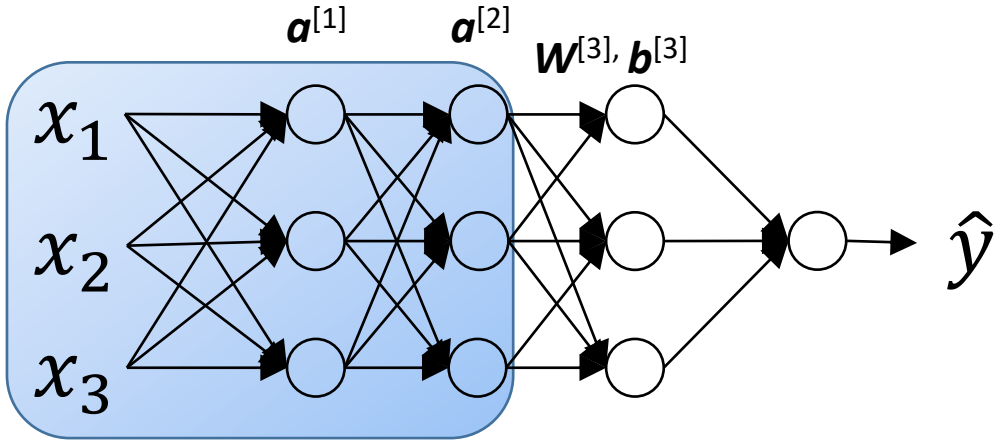
$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$ (mean 0 and standard deviation 1)

$$Z = \begin{bmatrix} | & | & & | \\ z^{(1)} & z^{(2)} & \ldots\ldots & z^{(m)} \\ | & | & & | \end{bmatrix}$$

$$\tilde{z}^{(i)} = \gamma\, z^{(i)}_{norm} + \beta$$ where γ and β are learnable parameters (adjust mean and standard deviation).

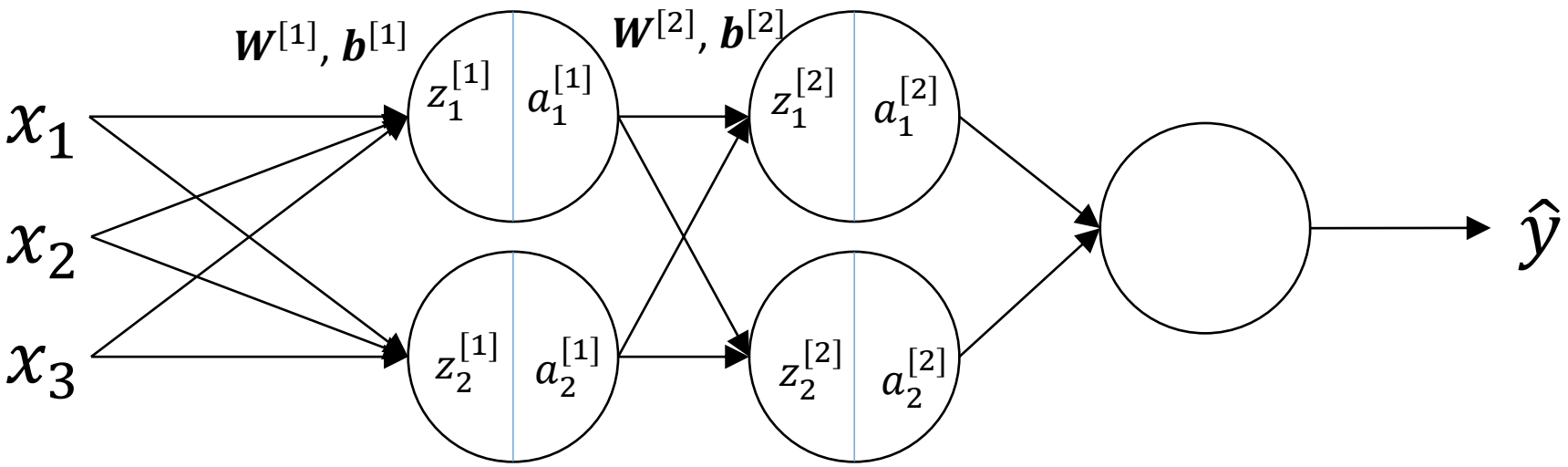


If $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, then $\tilde{z}^{(i)} = z^{(i)}$



Fitting Batch Norm into a neural network:

Adding Batch Norm to a network



$$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{(1)} \rightarrow A^{[1]} = g^{[1]}(\tilde{z}^{(1)}) \xrightarrow{W^{[2]}, b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{(2)} \rightarrow A^{[2]} = g^{[2]}(\tilde{z}^{(2)})$$

Batch Norm (BN)

BN

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots\ldots, W^{[L]}, b^{[L]}$

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

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

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

Working with mini-batches

m is the size of a mini batch

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

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

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

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

$$X^{\{1\}} \rightarrow Z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{W^{[1]}, b^{[1]} \quad \beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{(1)} \rightarrow A^{[1]} = g^{[1]}(\tilde{Z}^{(1)}) \rightarrow Z^{[2]} \xrightarrow[\text{BN}]{W^{[2]}, b^{[2]} \quad \beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{(2)} \rightarrow A^{[2]}$$

$$X^{\{2\}} \rightarrow Z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{W^{[1]}, b^{[1]} \quad \beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{(1)} \rightarrow A^{[1]} = g^{[1]}(\tilde{Z}^{(1)}) \rightarrow Z^{[2]} \xrightarrow[\text{BN}]{W^{[2]}, b^{[2]} \quad \beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{(2)} \rightarrow A^{[2]}$$

.....

$$X^{\{n\}} \rightarrow Z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{W^{[1]}, b^{[1]} \quad \beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{(1)} \rightarrow A^{[1]} = g^{[1]}(\tilde{Z}^{(1)}) \rightarrow Z^{[2]} \xrightarrow[\text{BN}]{W^{[2]}, b^{[2]} \quad \beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{(2)} \rightarrow A^{[2]}$$

Parameters: $W^{[L]}, b^{[L]}, \ldots\ldots, \beta^{[L]}, \gamma^{[L]}$

$$Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$$

$(n^{[L]},1)$ $(n^{[L]},1)$ $(n^{[L]},1)$

Implementing gradient descent

For t=1 …… number of Mini Batches

compute forward prop on $\boldsymbol{X}^{\{t\}}$

in each hidden layer, use BN to replace $\boldsymbol{Z}^{[L]}$ with $\tilde{\boldsymbol{z}}^{(L)}$

Use backprop to compute $\boldsymbol{dW}^{[L]}$, $\boldsymbol{db}^{[L]}$, ……, $\boldsymbol{d\beta}^{[L]}$, $\boldsymbol{d\gamma}^{[L]}$

Update parameter

$$\boldsymbol{W}^{[l]} = \boldsymbol{W}^{[l]} - \alpha \boldsymbol{dW}^{[l]}$$

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

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

$$\boldsymbol{b}^{[l]} = \boldsymbol{b}^{[l]} - \alpha \boldsymbol{db}^{[l]}$$

Also works with momentum, RMSprop, and Adam

Batch Norm at test time: one test data

In each layer L at training time:

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

m is the size of a mini batch

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

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

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

μ, σ^2 : estimate using exponentially weighted average (across mini-batch) during training time

$\mu^{[l]} = \beta\, \mu^{[l]} + (1-\beta)\, \mu^{\{\text{cur-batch}\}[l]}$

$X^{\{1\}}$, $X^{\{2\}}$, $X^{\{3\}}$, ... mini batch

$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}, \ldots \rightarrow \mu^{[l]}$ in each layer L

$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]}, \ldots \rightarrow \sigma^{2[l]}$ in each layer L

Test time for each test data: (in each layer L)

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu^{[l]}}{\sqrt{\sigma^{2[l]} + \varepsilon}} \qquad \tilde{z}^{(i)} = \gamma z^{(i)}_{\text{norm}} + \beta$$

Summary of today's lecture

- Learned various approaches for model optimization.
  - Regularization and Early stopping
  - Input normalization and Weights initialization
  - Mini-batch gradient descent
  - Gradient descent with momentum, RMSprop, Adam
  - Learning rate decay
  - Hypermeter turning process
  - Batch normalization

Reference

- Neural Network and Deep Learning, Andrew Ng, https://www.coursera.org/learn/neural-networks-deep-learning