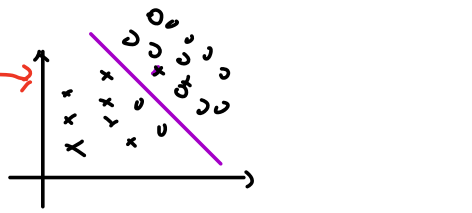
Week 1. Practical Neural Network

layers
hidden units
learing rates
activation function

- train/dev/test sets for big data
- make sure dev and test come from the same distribution

- bias and variance

Train set error : 1%	15%	15%	0.5%
Dev set error : 11%	16%	30%	1%
↓	↓	↓	
high variance	high bias	high bias & high variance	



Optimal (Bayes) error

In high dimension, less contrived

- basic recipe for NN ☆☆

high bias? —Y→ Bigger Network / Train longer ... / new network architecture

(training data performance)

↓ N

high variance? —Y→ more data / Regularization / new network architecture

(dev set performance)

↓ N

Done

In deep learning

Bias Variance

You can change one and doesn't hurt the other. (no trade-off)

Regularization Frobenius Norm: $\|A\|_F^2 = \sum_{i,j} a_{ij}^2 = \text{trace}(A^T A)$

For logistics Regression

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

L_2 regulation: $\|w\|_2^2 = \sum_{j=1}^{n} w_j^2 = w^T w$

L_1 regulation: $\|w\|_1$ → w will be sparse

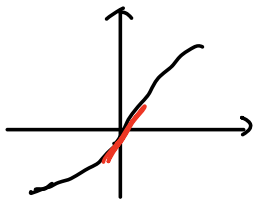
For neural Network

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

Frobenius Norm. add up every element in matrix.

Q: Why L_2-Regularization can prevent overfitting?

A:



tanh. $\lambda \uparrow \quad w^{[l]} \downarrow$

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

$\Rightarrow \downarrow$ become linear

Tips of implement:

J has new defination when you add the penalty. So please plot the figure using the new J.

Dropout Regulation ⭐

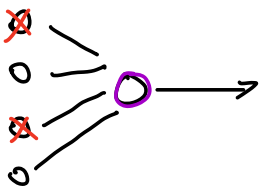
- Inverted dropout

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
a3 /= keep_prob   ← Scaling
```

Q: Why does "dropout" work?

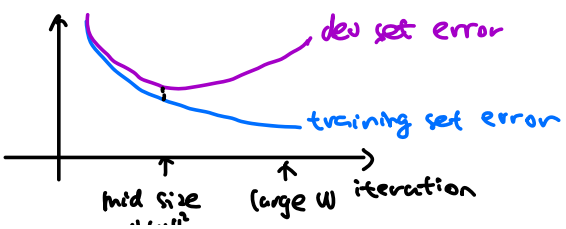
A:

Intuition: Can't rely on any one feature, so have to spread out weights → shrink weights



Other regularization methods

- Data augmentation : Rotation / flipping / random cropping
- Early stopping



$\|W\|_F$

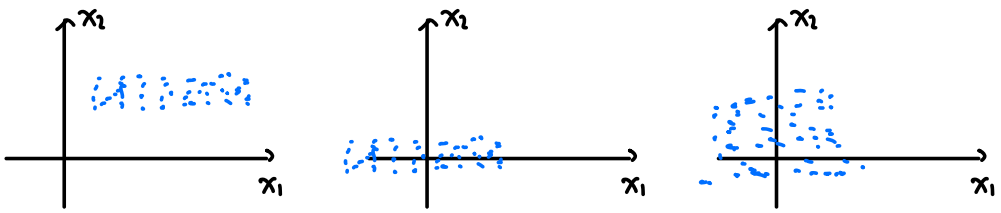
- Process of ML
  - Optimize cost function J
    - Gradient descent, ...
  - Not overfitting
    - Regularization, ...

Downside of early stopping: Combined two tasks, not orthogonalization

Optimization

Normalizing training sets

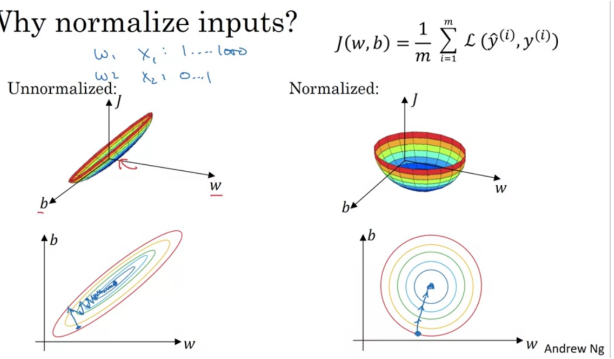
$$x = \frac{x - \mu}{\sigma}$$



- use same μ and σ to normalize training data and testing data!

Q: Why normalize input?

A:



It makes the cost function faster to optimize

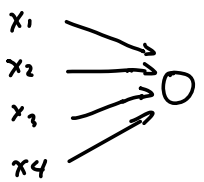
Vanishing / exploding gradients

- $W^l > I$ → explode
- $W^l < I$ → vanish

layer↑, $W^{\sum}$ ↑

→ partial solution: carefully chose initial weight

Weight Initialization for deep network



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

$\hat{y}$ To keep z not too large, we hope that n↑ w_i↓

RELU as activation function:

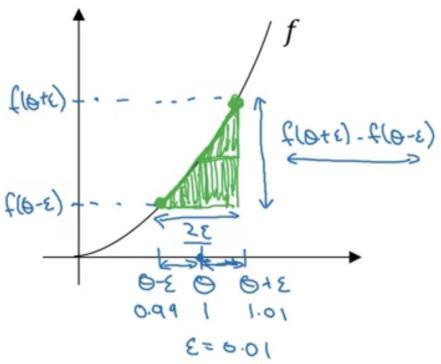
$W^{[l]}$ = np.random.randn(shape w) * np.sqrt($\frac{2}{n^{[l-1]}}$) ← Var(w_i)

tanh:
np.sqrt($\frac{1}{n^{[l-1]}}$)
or np.sqrt($\frac{2}{n^{[l-1]}+n^{[l]}}$)
As a hyperparameter
(but not priority one)

Numerical Approximation of Gradients

Aim: Prerequesite for Gradient checking (*)

$f(\theta) = \theta^3$



$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$

$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$

$g(\theta) = 3\theta^2 = 3$

approx error: 0.0001
(previous slide: 3.0301. error: 0.03)

$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ → error: $O(\varepsilon^2)$ | $\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$ → error: $O(\varepsilon)$

Conclusion: two-sided difference formula is much more accurate.

Gradient Checking

Aim: Debug, see if the back prop algorithm has been performed correctly.

STEP: ① Take $W^{[1]}, b^{[1]}, \dots, W^{[n]}, b^{[n]}$ and reshape into a big vector θ

(concatenate to a giant vector θ)

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

② Take $dW^{[1]}, db^{[1]}, \dots, dW^{[n]}, db^{[n]}$ and reshape into a big vector $d\theta$

Gradient checking (Grad check) $J(\theta) = J(\theta_1, \theta_2, \dots)$

for each i:

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

$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$ | $d\theta_{approx} \overset{?}{\approx} d\theta$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ ≈ 10^{-7} — great! ← / 10^{-5} → 10^{-3} — worry. ←

$\varepsilon = 10^{-7}$

Gradient checking implementation Notes

- Don't use in training — only to debug
- If algorithm fails grad check, look at components to try to identify bug. different i
- Remember regularization!
- Doesn't work with dropout. → turn off dropout, use grad check, then turn on it.
- Run at random initialization; perhaps again after some training.

Some takeaways

① Initialization

- Different initializations lead to very different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Resist initializing to values that are too large
- "He initialization" works well for networks with ReLU activations

$$W_L = (\text{Layer\_dims}[L], \text{layer\_dims}[L-1]) \times \sqrt{\frac{2}{\text{dimension of the previous layer}}}$$

② Regularization

L_2 - regularization

- The cost computation: A regularization term is added to the cost
- The backprop function: There are extra terms in the gradients with respect to weight matrices.
- Weights end up smaller ("weight decay")

Dropout

- Only use dropout during training
- Apply dropout both during forward & backward prop.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations.
- Regularization will drive your weights to lower values.

③ Gradient Checking

- Gradient Checking doesn't work with drop out. Work before dropout, make sure the gradient has been calculated correctly! Then turn on the gradient descent.

Optimization Algorithms

Mini-batch gradient descent

- Vectorization allows you to efficiently compute on m examples.

Assume $m = 5{,}000{,}000$, then the mini-batch is

$$X_{n \times m} = [\underbrace{x^{(1)}, x^{(2)}, \dots x^{(1000)}}_{X^{\{1\}}\ (n \times 1000)}, \underbrace{x^{(1001)}, \dots x^{(2000)}}_{X^{\{2\}}}, \dots, \underbrace{\dots x^{(m)}}_{X^{\{5000\}}}]$$

$$Y_{1 \times m} = [\underbrace{y^{(1)}, y^{(2)}, \dots y^{(1000)}}_{y^{\{1\}}}, \underbrace{y^{(1001)}, \dots, y^{(2000)}}_{y^{\{2\}}}, \dots, \underbrace{\dots y^{(m)}}_{y^{\{5000\}}}]$$

- Algorithm

for $t = 1, 2, \dots, 5000$

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

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$Z^{[l]} = W^{[l]} X^{\{t\}} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

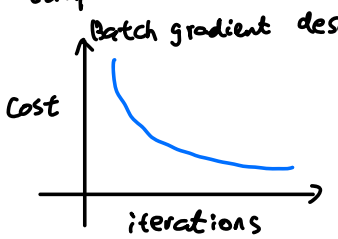
} Vectorized implement

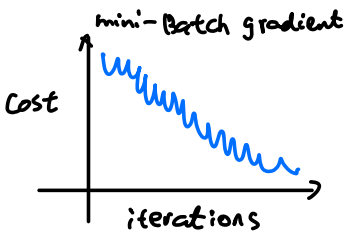
Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$ (i in $X^{\{t\}}$)

Backprop to compute gradient $J^{\{t\}}$

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

- Compare with Batch



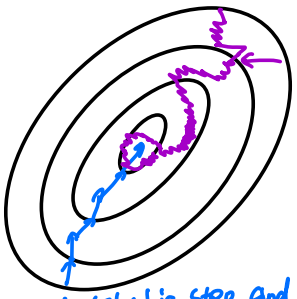


]

- How to choose mini-batch size ?

If mini-batch size = m ⇒ Batch gradient descent

= 1 ⇒ Stochastic gradient descent



SGD: small step and no converge

Drawback: Lose speedup from vectorization

batch: big step and converge fast

Drawback: too long per iteration

mini-batch $\in (1, m)$

Advantage: ① Vectorization

② make process without using entire training set

Guideline:

If small training set: Use batch gradient descent

Otherwise, mini-batch size: 64. 128. 256. 512 ... (2^n)

Exponentially weighted average

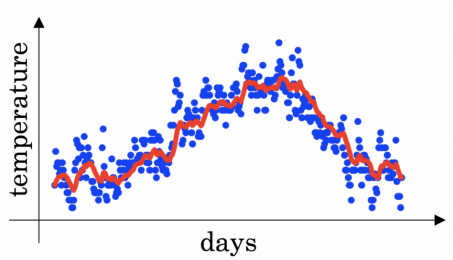
formula:

$V_0 = 0$

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

V_t is approximately $\frac{1}{1-\beta}$ days

- Increase β will shift the red line slightly to the right
- Decrease β will create more oscillation within the red line



Algorithm:

$V_\theta = 0$

Repeat {

get next θ_t

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

}

Bias correction Exponentially weighted Averages

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

Gradient descent with momentum

Algorithm:

On Iteration t:

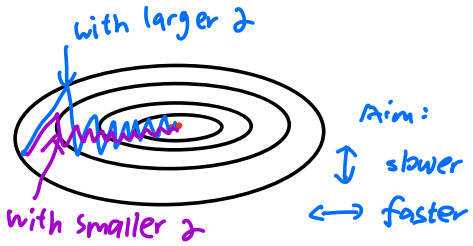
Compute dw, db on current mini-batch

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

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

$W = W - \alpha V_{dw}$

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



Hyperparameters: α, β (0.9 is robust)

RMSprop

Algorithm:

On Iteration t:

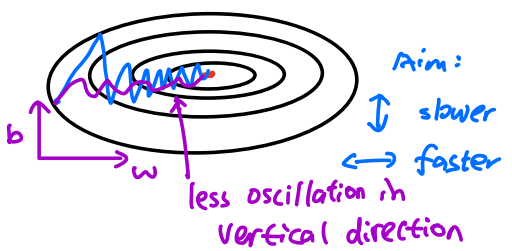
Compute dw, db on current mini-batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$ ← element-wise ← small

$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$ ← large

$W = W - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$

$b = b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$ ← small



Then, we can choose a larger learning rate α to speedup the converage

Adam optimization algorithm (combined momentum and RMSprop)

$V_{dw} = 0$, $V_{dw^2} = 0$ $V_{db} = 0$ $V_{db^2} = 0$

On Iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$

$V_{dw}^{correct} = \frac{V_{dw}}{1-\beta_1^t}$ $V_{db}^{correct} = \frac{V_{db}}{1-\beta_1^t}$ (t counts the number of steps taken of Adam)

$S_{dw}^{correct} = \frac{S_{dw}}{1-\beta_2^t}$ $S_{db}^{correct} = \frac{S_{db}}{1-\beta_2^t}$

$W = W - \alpha \frac{V_{dw}^{correct}}{\sqrt{S_{dw}^{correct}} + \varepsilon}$ $b = b - \alpha \frac{V_{db}^{correct}}{\sqrt{S_{db}^{correct}} + \varepsilon}$

Hyperparameter choices:

α: needs to be tuned

β_1: 0.9 (dw)

β_2: 0.999 (dw^2)

ε: 10^{-8}

Learning rate decay

1 epoch = 1 pass through data

$$\alpha = \frac{1}{1 + \text{decay\_rate} \times \text{epoch\_num}} \cdot \alpha_0$$

Fixed Interval Scheduling

$$\alpha = \frac{1}{1 + \text{decay\_rate} \times \left\lfloor \frac{\text{epochNum}}{\text{time Interval}} \right\rfloor} \alpha_0$$

e.g.

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

$\alpha_0 = 0.2$

decay_rate = 1

hyperparameters

The problem of Local Optima

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

Hints on practical

- Mini-batch
  - Shuffling and Partitioning are two steps required to build mini-batches (know how to handle the last mini-batch % batch_size != 0)
  - Powers of two are often chosen to be the mini-batch size, e.g. 16, 32, 64, 128...
- Momentum
  - Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch descent, Stochastic gradient descent
  - You have to tune a momentum hyperparameter β and a learning rate α

- Adam
  - Relatively low memory requirements
  - Usually works well even with little tuning of hyperparameters (except α)

Week 3 Hyperparameter Tuning. Batch Normalization and Programming Frameworks

. Hyperparameter tuning

• Tuning process

α ★★★

β ★★

$\beta_1, \beta_2, \varepsilon = 0.9, 0.999, 10^{-8}$

layers ★

hidden units ★★

learning rate decay ★

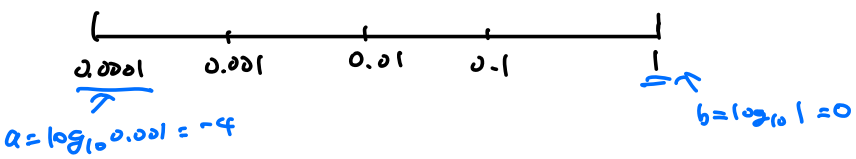
mini-batch size ★★

△ Try random values : Don't use a grid !

△ Coarse to fine

• Using an appropriate scale to pick hyperparameters

log_scale( )



$a = \log_{10} 0.001 = -4$

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

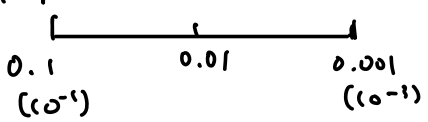
r = -4 * np.random.rand( ) ← $r \in [-4, 0]$

α = 10^r ← $10^{-4} \cdots 10^0$

try to find β

X

→ 1 - β



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

$1 - \beta = 10^r$

$\beta = 1 - 10^r$

• Pandas model & Covier model

• Batch Normalization

$x := \frac{x-\mu}{\sigma}$

· Normalizing Activations in a network

Implement Batch Norm

Give some intermediate values in Neural Network $z^{(1)}, \dots z^{(m)}$ ← $z^{[l](i)}$

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

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

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

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$ → learnable parameters of model

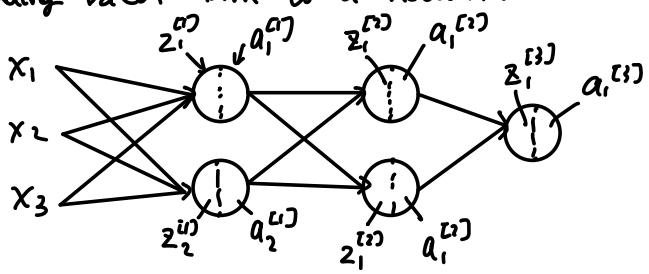
If $\gamma = \sqrt{\sigma^2+\varepsilon}$, $\beta = \mu$

then $\tilde{z}^{(i)} = z^{(i)}$

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

· Fitting Batch Norm into a Neural Network

Adding Batch norm to a network



$$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch-Norm}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = \varphi(\tilde{z}^{[1]}) \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow a^{[2]} = \varphi(z^{[2]}) \dots$$

Parameter : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \dots, \dots, w^{[l]}, b^{[l]}$

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

* $z^{[l]} = w^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$ ←

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

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

Algorithm

for $t=1, \ldots,$ numMiniBatch:

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

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

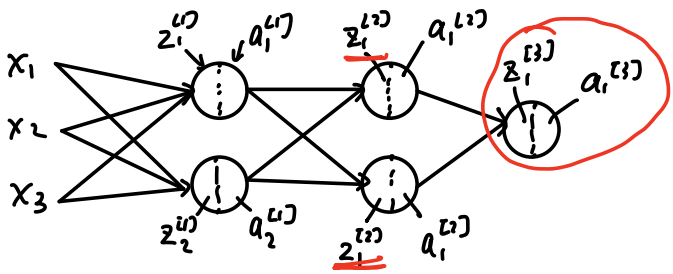
Use backprop to compute $dW^{[l]}, db^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$

Update parameters
$$\begin{cases} W = W - \alpha dW^{[l]} \\ b = b - \alpha db^{[l]} \\ \beta = \beta - \alpha d\beta^{[l]} \\ \gamma = \gamma - \alpha d\gamma^{[l]} \end{cases}$$
← can use momentum, RMSprop, Adam to replace GD

Why does Batch Norm work?

Covariate shift $X \longrightarrow Y$

If X distribution change, then we should re-train the network



$z^{[3]}$ depends on $z^{[2]}$ and $z^{[2]}$ would keep changing. However, after apply batch normalization, though the change of $z^{[2]}$ still happens, the means and variance of $z^{[2]}$ remain unchanged, its just affects by hyperparameter γ and β

↓

reduce the dependence among layers

↓

speed up the learning in the whole network

Batch normalization as regularization (weak, just a side effect)

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

- Use larger mini-batch size, the effect of regularization will be reduced.

Multi-class Classification

- Softmax Regression

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

Activation function:

$$t = e^{(z^{[l]})}$$

$$a^{[l]} = \frac{e^{(z^{[l]})}}{\sum_{i=1}^{4} t_i}, \quad a_i^{[l]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

($a^{[l]}$: 4×1)

e.g.

$$z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

$$\sum_{i=1}^{4} t_i = 176.3$$

$$a^{[l]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

($a^{[l]}$: 4×1)

In summary, softmax: $a^{[l]} = g^{[l]}(z^{[l]})$ (4×1, 4×1)

- Training a softmax classifier

Softmax regression generalizes logistic regression to C classes

Loss function

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad a^{[l]} = y^{(i)} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

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

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

Backprop:

$$dz^{[l]} = \hat{y} - y$$

$$\hookrightarrow \frac{\partial J}{\partial z^{[l]}}$$

Deep Learning Framework

Tensorflow

Code example


```
import numpy as np
import tensorflow as tf

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)
```

tf.data.Dataset.from_tensor_slices( )

x_train.element_spec

next(iter(x_train))

transform → use map

x_train.map(normalize)

x = tf.constant(np.random.rand(3,1), name="x")