Regularization

L2 Regularization

For neural network:

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

$||w^{[l]}||_F^2$ is the Frobenius norm

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

$$\frac{\partial J}{\partial w^{[l]}} = dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m}w^{[l]}$$
$$w^{[l]} = w^{[l]} - \alpha \cdot dw^{[l]}$$

Also called "weight decay".

Why does regularization prevent overfitting?

1. the loss function penalizes for large value of w. Therefore, chances are some weights will get negligible values for min loss purpose, making the model simpler and less likely to overfit.
2. For instance, suppose you are using $tanh$ function as the activation function. $tanh$ function approximates linear when the absolute value of weights is small. Therefore, when weights decay, the model gets simpler and simpler to some extent.

Dropout Regularization

In each layer, you specify the drop out ratio and randomly select nodes to set to zero according to the ratio.

```
# eg
L = 3

keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
a3 /= keep_prob

"""
/= keep prob ensures that even if
you don't implement drop-out at test time
the expected value of 'a' is still the same
"""
```

Why does drop-out work?

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

It's possible to show that dropout regularization has similiar effect to L2 regularization. It's also feasible to vary 'keep_prob' for diff layers.

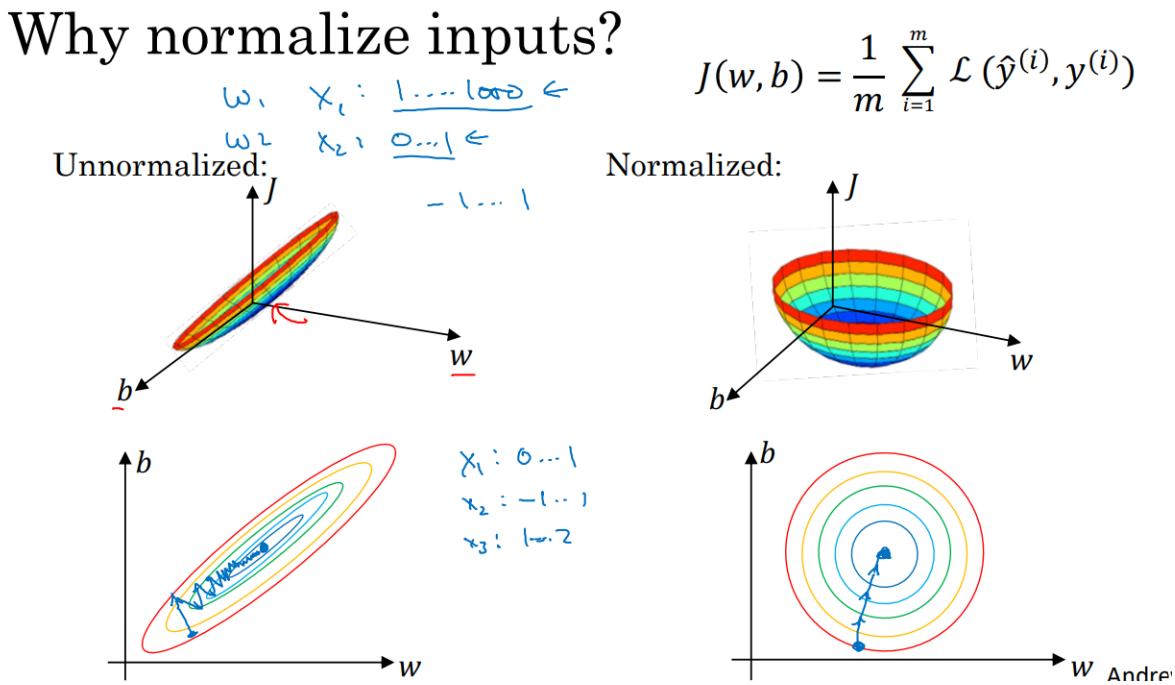
Drawback of dropout: J-cost function is no longer well-defined.

Other Regularization Method:

1. Data Augmentation
2. Early Stopping

Data Processing

Normalizing Training Sets



If variables are on same scale, don't need to 走之字形 when train. More likely can head towards the optimal point directly.

Vanishing/Exploding Gradients

General idea is, if weights are not properly initialized, at some iteration the gradients will fade to zero or blow up.

1. **Random initilizaiton**

W = np.random.randn(shape[0], shape[1]) * scaler b = np.zeros((shape[0], 1))

2. **He initialization**

W = np.random.randn(shape[0], shape[1]) * np.sqrt(2 / n^[l-1])

Week 2: Better Learning/Training Algorithm

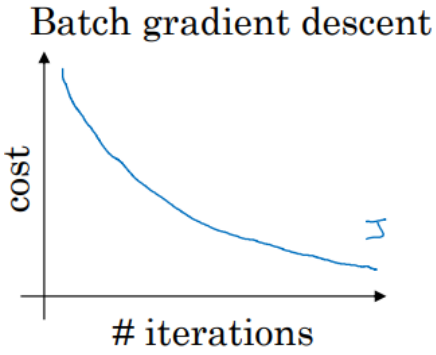
Mini-Batch Gradient Descent

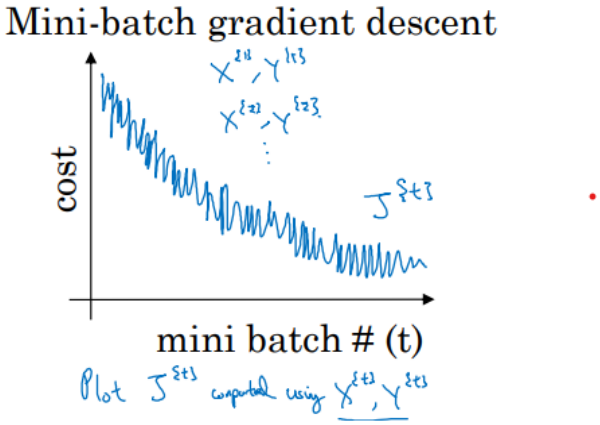
Vectorization saves time, but if we were to train our model on a large dateset, every iteration it has to process every example and this takes a great amount time.

We can use the mini-batch technique, where we put examples into minibatches and each time we only update parameters based on one batch.

Suppose we have m = 500, 000 examples and put them into 5000 batches. Each minibatch is $X^{\{t\}}$ and $Y^{\{t\}}$

```
for i in range(num_epoch):
    ### the following is called one epoch
    for t in range(5000):
        forward propagation on X^{t} (will be vectorized)
        compute cost J^{t} =  1 / 1000 * \sum(L(\hat{y}, y)) + regula
rization
        back propagation to compute gradient w.r.t J^{t}
        update parameters
    ### end of one epoch
```





Since each time the parameters are updated according to different minibatch, there are noises involved. Instead of a smooth downward-bending learning curve, the curve under minibatche is wiggly downward.

Choose minibatch size
If minibatch size = m: Blue
If minibatch size = 1: Purple
Stochastic Gradient Descent - Each time the model is trained on one example.

The green is somewhere between 1 and m. Note that the stochastic and mini-batch gradient descent never converge to the min objective. Once it approaches, it may jump around the objective.

Typical batch size: 2^n

Exponentially Weighted Average

Exponentially Weighted Average is a smoothing technique that gives closer values higher weights.

Let θ_t be the observed value at t

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

For instance, $\beta = 0.9$

$v_t = 0.1\theta_t + 0.1 \cdot 0.9\theta_{t-1} \dots$
When t is large enough, the weights sum to 1 $\Rightarrow \frac{0.1}{(1-0.9)}$ - Geometric Sequence

$\beta = 0.9$ is typically used in practice. $0.9^{10} \approx 0.35 \approx \frac{1}{e}$ which is smaller enough, meaning that the weights decay small enough 10 periods back in time.

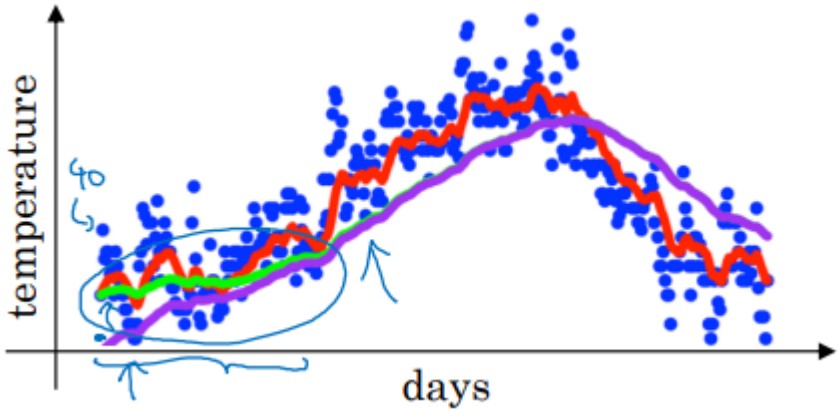
$(1-\epsilon)^{1/\epsilon} \approx \frac{1}{e}$. So if $\beta = 0.98$, need 1/0.02 = 50 periods back in time to decay small enough.

Implementation

Normally don't have to tune beta. $\beta = 0.9$ is good.

```
Initially set v_0 = 0

for t in range(T):
     v_theta = \beta * v_\theta +(1-\beta) * \theta_t
```



The result of exponential weigted average is the green curve, it is shifted towards right compared to the real trend because the importance of the concurrent value is lessened by adding the previous terms.

However, if we initialize $v_0 = 0$, the actual curve we'd get is the purple one since it takes several iteration for the model to warm up.

bias correction

$v_t \Rightarrow \frac{v_t}{1-\beta^t}$

How it works:
When t = 3,

$$v_2 = \theta_1 * \beta^2 * (1-\beta) + \theta_2 * \beta * (1-\beta) + \theta_3 * (1-\beta)$$

$$\beta^2 * (1-\beta) + \beta * (1-\beta) + (1-\beta) = (1-\beta) * \frac{1-\beta^3}{1-\beta} = 1-\beta^3$$

We see that the sum of all coefficients is exactly $1-\beta^3$. So to descale the weight sum in order to correct bias, divide v_t by $1-\beta^t$. And when t goes large, the effect of dividing $1-\beta^3$ diminishes.

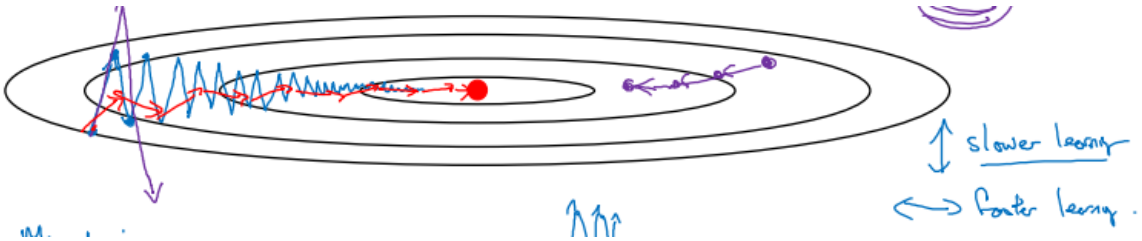
Faster Learning Algorithm

Gradient Descent with Momentum

```
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 -= alpha * v_dW
    b -= alpha * v_db
```

People normally don't do exponential weighting on v_dW and v_db since it the result of not doing it is just to make the model warm up slower.

Intuition:



During learning the trajectory might be oscillating and make it slower to reach the objective. Adding the effect of previous gradient help to average out the vertical oscillation and and make the model learn fast in the horizontal direction.

RMSprop

```
On iteration t:
    Compute dW, db on the current mini-batch
    s_dW = beta * s_dW + (1-beta) * dW**2
    s_db = beta * s_db + (1-beta) * db**2
    W -= alpha * dW / sqrt(s_dW+epsilon)
    b -= alpha * db / sqrt(s_db+epsilon)
```

s_dW and s_db can be thought of as the exponentially weighted volatility, descaling the gradient by volatility can suppress the oscillation. Epsilon is to avoid division by zero.

Adam

Adam is the combination of RMSprop and Momentum.

```
v_dw = 0, s_dw = 0, v_db = 0, s_db = 0
On iteration t:
    Compute dW, db on the current mini-batch
    v_dW = beta1 * v_dW + (1-beta1) * dW
    v_db = beta1 * v_db + (1-beta1) * db
    s_dW = beta2 * s_dW + (1-beta2) * dW**2
    s_db = beta2 * s_db + (1-beta2) * db**2
    v_dW^corrected = v_dW / (1-beta1^t)
    v_db^corrected = v_db / (1-beta1^t)
    s_dW^corrected = s_dW / (1-beta2^t)
    s_db^corrected = s_db / (1-beta2^t)
    W -= alpha * v_dW^corrected / sqrt(s_dW^corrected + eps)
    b -= alpha * v_db^corrected / sqrt(s_db^corrected + eps)
```

Hyperparameter
alpha: need to be tune
beta1: 0.9 (dw)
beta2: 0.999 (dw^2)
eps: 10e-08

Learning Rate Decay

When mini-batch gradient descent approaches objective, it swirls around and refuses to converge. So as it gets closer to the objective, we want the step it takes to be smaller -- by decreasing learning rate.

some decay methods

1. $\alpha = 1/(1 + decay\_rate * epoch\_num)$
2. $\alpha = 0.95^{epoch\_num} * \alpha_0$
3. $\alpha = k/\sqrt{epoch\_num} * \alpha_0$
4. step-wise, mod interval == 0

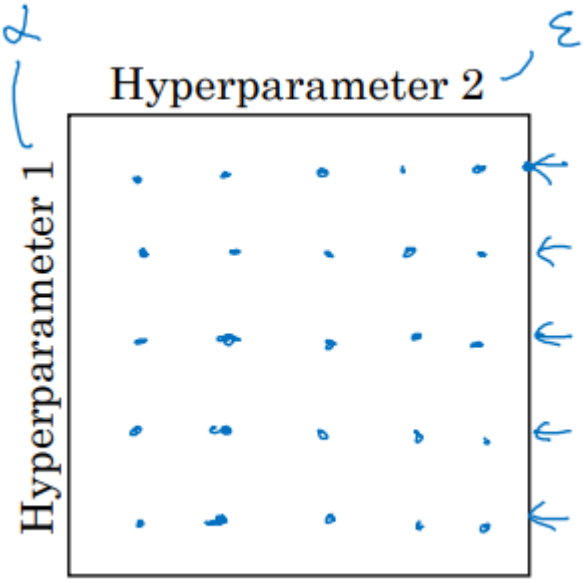
Week 3: Hyperparameter Tuning, Batch Normalization and Softmax

Hyperparameter Tuning

Hyperparameter List

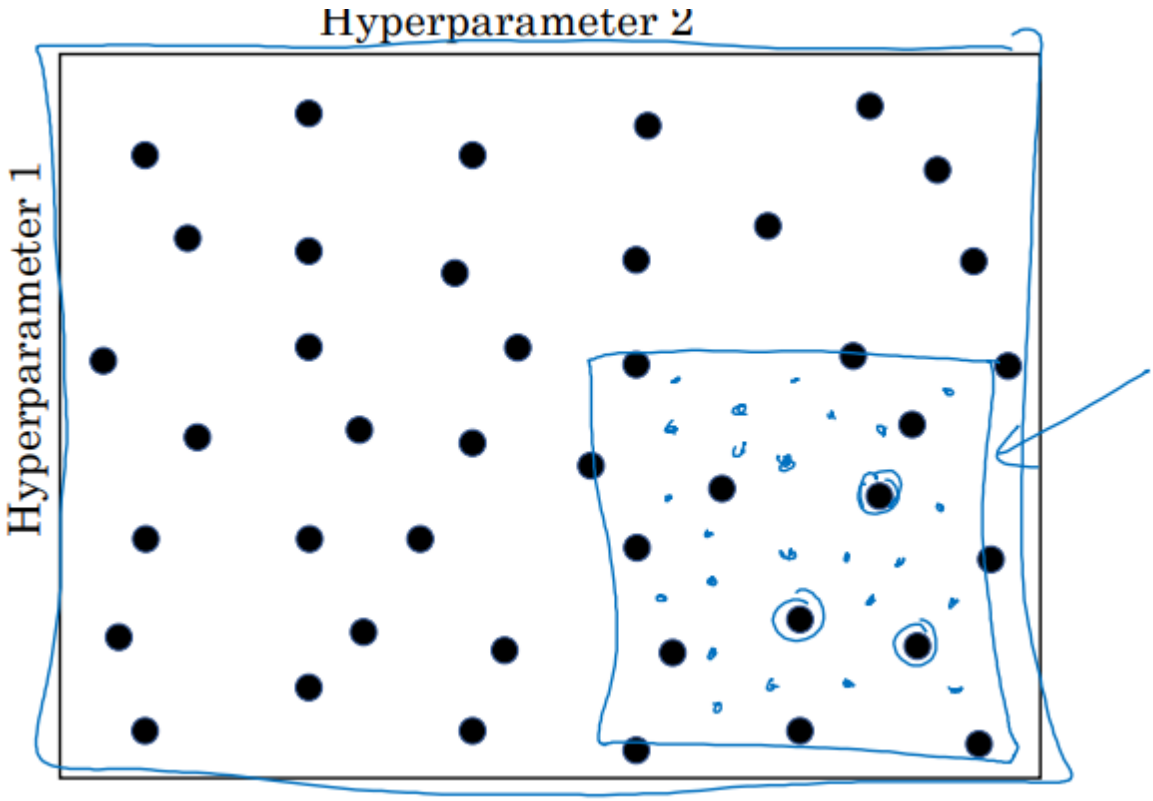
1. Most Important:
   α: learning rate
2. Second Important:
   number of layers and batch size
3. Not Imperative:
   β (exponential weighted avg): ~0.9
   decay rate
4. Generally Don't Tune:
   β_1, β_2, eps (Adam): can be tuned, but generally don't\

Don't use grid search, use random



In the above fig, eps is not an important param to tune, so we don't want to waste a bunch of search on eps w/ a fixed alpha.

Coarse to fine



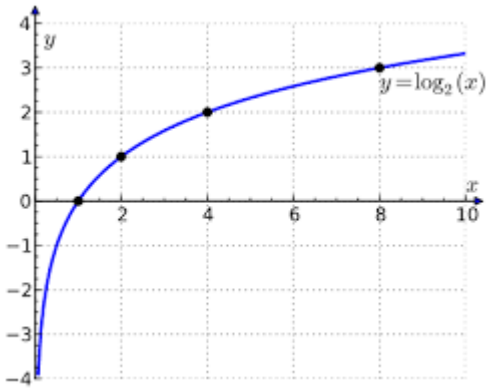
If you think params in a particular region would do better, you can zoom in to that specific region.

Scale of Hyperparameters

Alpha
for instance if you want to search in range [0.0001, 1] for learning rate alpha, you should do uniform random search on log scale.

```
r = -4 * np.random.rand()
alpha = 10 ** r
```

$r \in [-4, 0]$ and $\alpha \in [0.0001, 1]$



Uniform random on log scale is to do uniform random search along y axis. Log function has more dense x on short end, which means [0.0001, 0.001] is equally likely to be searched as [0.1, 1] and more likely to be searched than [0.9999, 1], which is what we want for alpha, putting more searching power on short end.

Beta (exponential weigted average)
beta 0.9 to 0.9005 still ~10, only need 10 periods for past value to decay
beta 0.999 to 0.9995 from 1000 periods decay to 2000 periods
Therefore, conversely, we want the search on beta to be dense on the long end.

For instance, if we want to search in the range [0.9, 0.999]
[1-0.9, 1-0.999] -> [0.1, 0.001] -> [10e-01, 10e-03]

```
r = np.random.uniform(-3, -1)
1-beta = 10**r
beta = 1-10**r
```

Batch Normalization

For each layer, normalize $z^{[l]}$ to get $z^{[l]}_{norm}$.
Then rescale and shift $z^{[l]}_{norm}$ to $\tilde{z}^{[l]} = \delta^{[l]} z^{[l]}_{norm} + \beta^{[l]}$, where δ and β are trainable params.

```
for t in range(num_minibatches):
    compute forward propagation on X^{t}:
        In each hidden layer, use \tilde(z) instead of z

    use back propagation to compute dW, dBeta, dDelta (omit db, the o
ffset since already have dBeta)
    update params
```

Batch normalization works with minibatch, Adam...

Batch normalization has slight regularization effect since each batch add some noise, similar to drop-out.

Why does batch normalization work?
Each layer, after normalization, has similar mean and variance, so that later layers are more robust.

Softmax

$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

$$L = -\sum_C y_c log(\hat{y}_c)$$

min L is equiv to MLE.