

week6: Training Neural Networks

✓ 예습	미완료
✓ 복습	미완료
복습과제 날짜	
예습과제 날짜	
내용	

Activation Function(활성함수)

Activaiton function의 종류

Sigmoid

Tanh

ReLU

Leaky ReLU

ELU(Exponential Linear Units)

Maxout “Neuron”

Data Preprocessing

Step1: Preprocess the data (데이터 전처리)

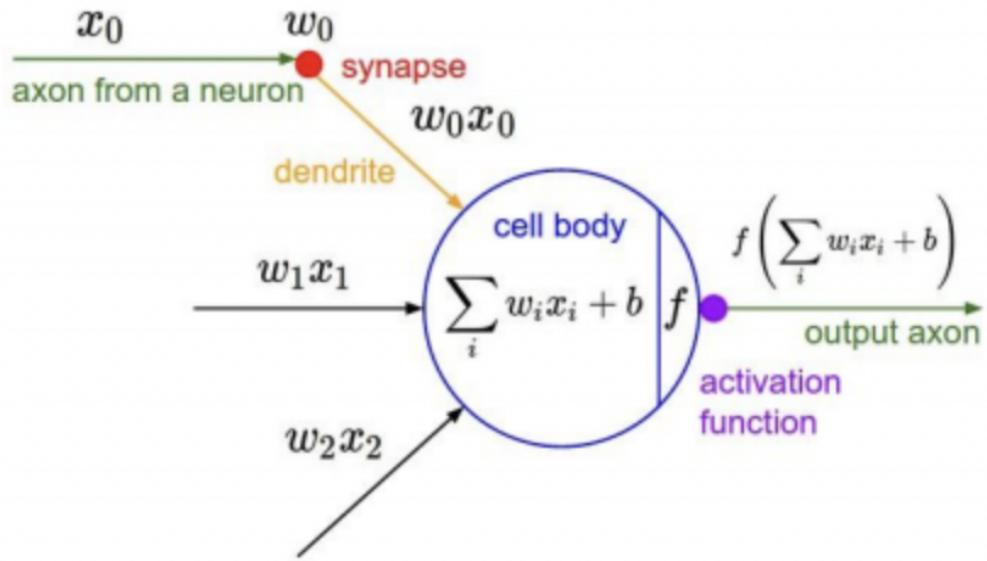
Weight Intialization

Batch normalization

Step2: Choose the architecture

Hyperparameter optimization

Activation Function(활성함수)



활성 함수는 $Wx+b$ 에 대해 다음 노드로 어떻게 보낼지 결정해주는 역할을 한다. 활성함수는 필수적인 요소이고, 비선형 형태여야한다.

선형함수이거나 활성함수가 없다면?

$$\begin{aligned}
 a &= wx + b \\
 a_2 &= w_2 a + b_2 \\
 &= w_2(wx+b) + b_2 \\
 &= (w_2w)x + (w_2b + b_2)
 \end{aligned}$$

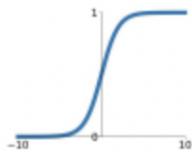
결국 $wx+b$ 의 형태와 같은 형태가 나오게 된다. 레이어를 여러번 깊게 쌓아도 의미가 없다.

Activaiton function의 종류

Activation Functions

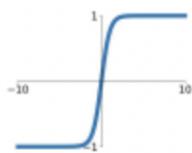
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



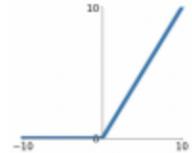
tanh

$$\tanh(x)$$



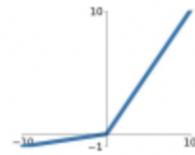
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

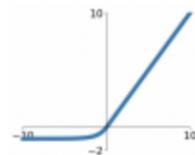


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

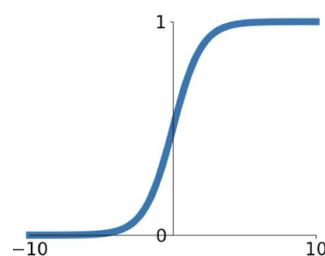
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid

Activation Functions



Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

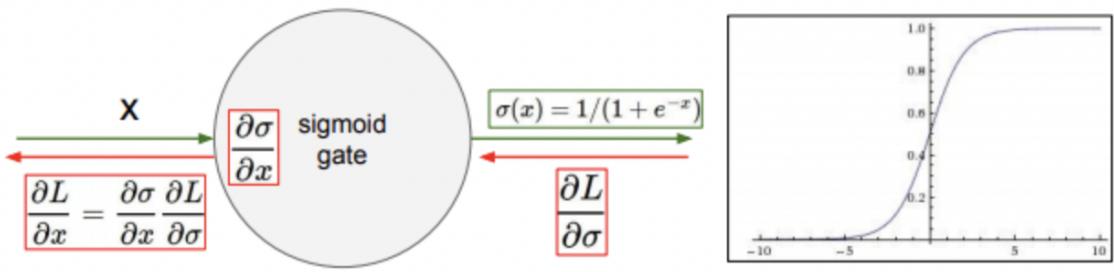
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. *exp()* is a bit compute expensive

3가지 문제점이 있다.

1. 기울기 소실 =kill the gradients



What happens when $x = -10$?

What happens when $x = 0$?

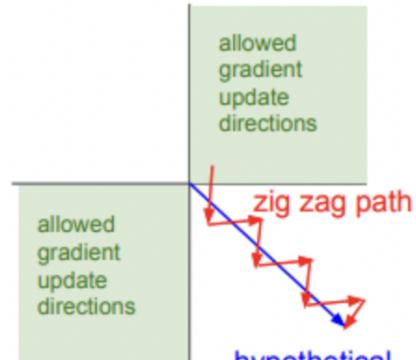
What happens when $x = 10$?

$x=+-10$ 인 구간에서 기울기가 0이 된다. 또한 기울기의 최대값이 0.5라서 backprop을 하면 모두 최대값이더라도 $0.5*0.5=0.25$ 로 gradient가 점점 작아진다. 레이어를 많이 쌓아둔 경우에는 기울기가 소실된다.

2. not 0 centered

Consider what happens when the input to a neuron is always positive...

$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on w ?

Always all positive or all negative :(

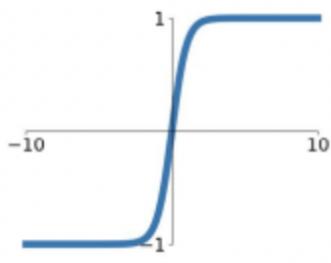
(this is also why you want zero-mean data!)

그래프가 0을 중심으로 되어있지 안한. 즉, 모든 input에 대해 output이 양수값으로 나온다.

→ dL/dw 에 의해 부호가 결정되고, 모두 양수이거나 음수인 상태이기 때문에 대각선으로 가지 못하고 지그재그를 그리면서 간다.

3. $\exp()$ is a bit compute expensive

Tanh



$\tanh(x)$

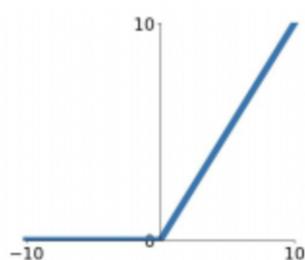
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

sigmoid의 단점인 not zero centered를 개선. 그러나, 나머지 문제는 해결되지 않았다.

ReLU

Activation Functions



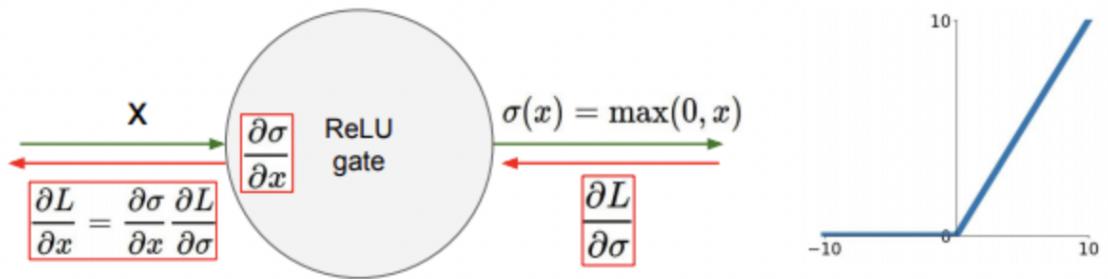
ReLU
(Rectified Linear Unit)

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid
- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?

가장 대중적으로 사용한다.

non zero-centered와 0이하의 값이 모두 버려지는 문제가 있다.



What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

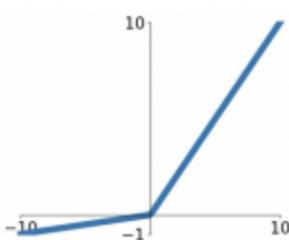
만일 입력 값이 -10과 0이 들어오면 gradient와 output 모두 0이된다. (=Dead ReLU)

이 상태를 해결하기 위해 0.01의 bias를 주어 0을 없애는 방법이 있지만, 효능은 좋지 않다.

Leaky ReLU

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

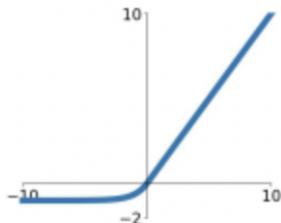
ReLU에 대한 보완. 0이하의 값을 0으로 두지 않고, 작은 값을 주는 것이다.

ELU(Exponential Linear Units)

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- Computation requires $\exp()$

또 다른 ReLU의 변형.

ReLU의 장점과 zero mean에 가까운 결과가 나오는 장점이 있지만, \exp 연산이 있는 단점이 있다.

Maxout “Neuron”

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ReLU와 LeakyReLU를 일반화하였다.

2개의 파라미터를 넘겨주고 max를 이용하여 더 좋은 것을 선택하는 방식이지만 연산량이 2배가 되어 잘 사용하지 않는다.

일반적으로 ReLU와 Leaky ReLU를 많이 사용한다.

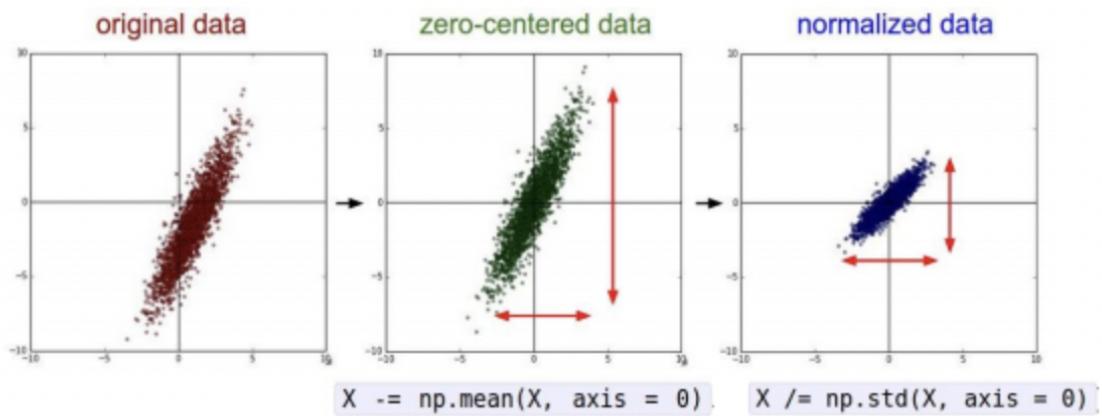
Maxout/ELU > tanh

Tanh는 RNN LSTM에서 주로 사용, CNN에서는 사용하지 않음. Sigmoid는 사용하지 않음.

Data Preprocessing

Step1: Preprocess the data (데이터 전처리)

주로 zero-centered, normalized를 많이 사용.



(Assume X [NxD] is data matrix,
each example in a row)

zero-centered는 행렬에 양, 음수를 모두 가지고 있어 효율적으로 이동가능. normalized는 표준편차로 나눠 데이터의 범위를 줄여 학습을 가속화, local optimum에 빠지는 가능성을 줄여줌.

이미지는 이미 [0,255]의 특정 범위를 가지고 있어 normalized보다 zero-centered를 많이 사용.

외에 분산에 따라 차원을 감소하는 PCA, whitened data등이 존대하는데, 이미지에서 잘 사용하지 않음.

Weight Initialization

weight가 어떻게 초기화 되었는지에 따라 학습의 결과에 영향을 준다.

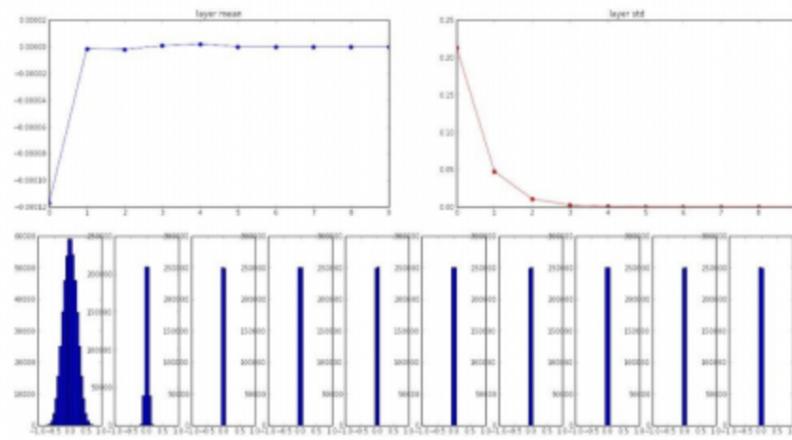
1st idea : small random numbers

```
w = 0.01*np.random.randn(D,H)
```

작은 랜덤 값을 weight에 주어 사용하였다. 작은 네트워크에서는 잘 작동하지만 깊은 네트워크에서는 잘 작동하지 않았다.

만일 10개의 레이어에 각각 500개의 유런, tanh non-linearities 사용, weight init을 랜덤 small number이라고 하면,

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!

Q: think about the backward pass.
What do the gradients look like?

Hint: think about backward pass for a W^*X gate.

모든 가운데의 값을 제외한 activation이 0이 된것을 확인 할 수 있다. tanh는 0에 가까운 부분만 기울기를 제공하고 0에서 멀어질 수록 기울기가 0에 가깝게 되어 결국 없는 값이 된다.

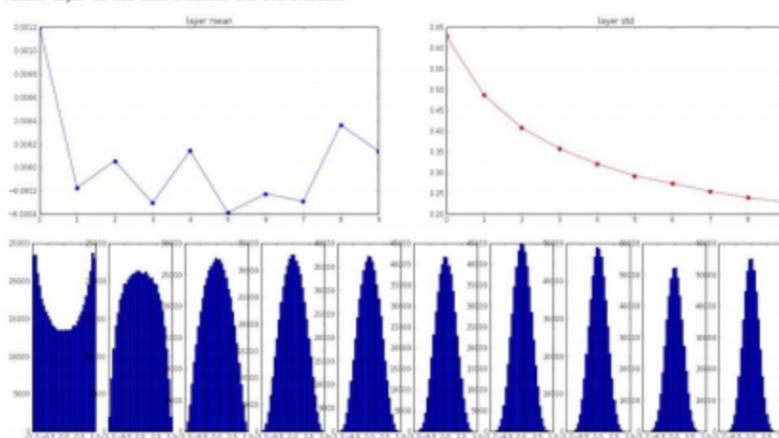
그렇다면 W값을 크도록 초기화한다면?

tanh가 1,-1에 포화되어 오버슈팅이 일어나게된다.

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486651
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean 0.000386 and std 0.357188
hidden layer 5 had mean 0.000142 and std 0.328918
hidden layer 6 had mean -0.000389 and std 0.292216
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

`W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization`

"Xavier initialization"
[Glorot et al., 2010]



Reasonable initialization.
(Mathematical derivation assumes linear activations)

적절한 weight 초기값을 주기 위해 Xavier Initialization을 사용한다. 노드의 수를 normalize하는 방법으로, input의 개수가 커지만 나누어 값을 줄이고, input의 개수가 적으면 weight가 커지는 방식으로 weight를 초기화한다.

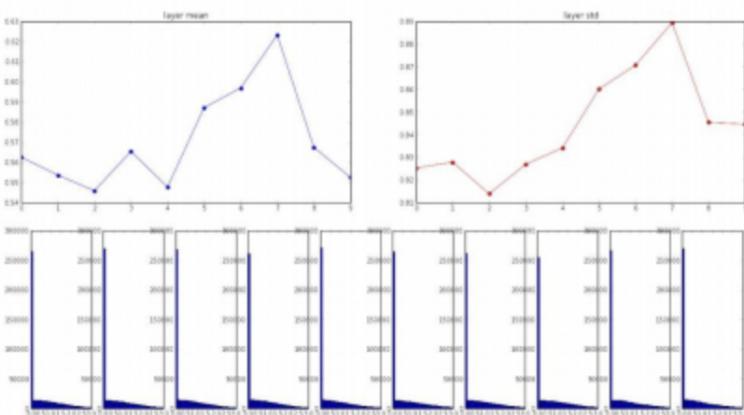
즉, input과 output의 분산을 동일하게 만든다.

gradient vanishing 현상을 완화하기 위해선 출력값들이 정규분포를 갖게 하는 것이 중요하다.

Xavier을 ReLU와 함께 사용한다면? ReLU는 출력값이 0으로 수렴하고, 평균과 표준오차도 0으로 수렴하기 때문에 사용하지 않는다. 대신 fan_in size를 2로 나누어 사용한다.

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015
(note additional /2)



Batch normalization

batch norm은 training 과정에서 gradient vanishing 문제를 일어나지 않도록 하기 때문에 weight init을 하지 않아도 괜찮다.

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

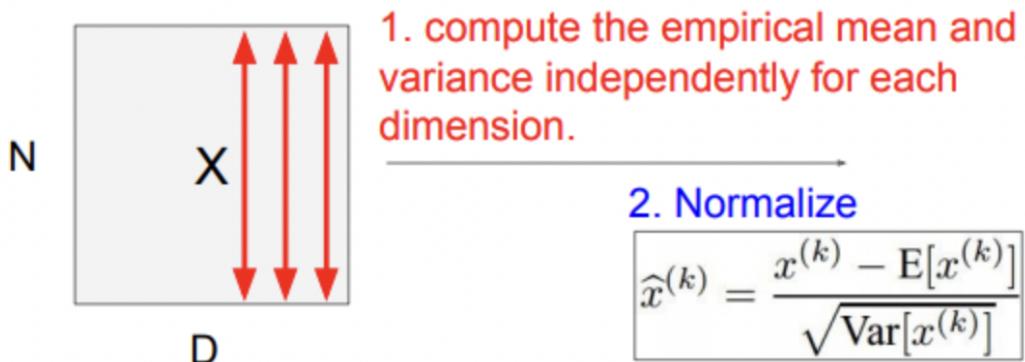
this is a vanilla
differentiable function...

기본적으로 batch norm은 각 층의 input distribution을 평균 0, 표준편차 1로 만드는 것이다. 네트워크 각 층의 activation마다 input distribution이 달라지는 ‘internal covariance shift가 변경되는 문제’를 해결하기 위해 사용.

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations?
just make them so."

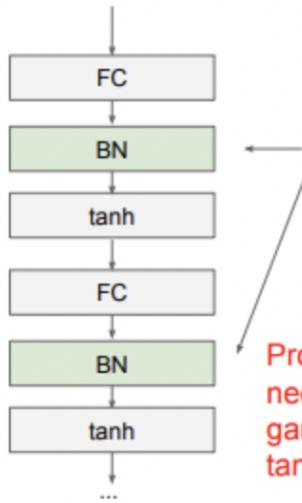


batch별로 train시키는데, NxD의 batch input이 들어오면 이것을 normalize한다.

평균값을 빼주어 평균을 0으로, 분산으로 나누어 분산값을 1로 변경

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

activation layer 전에 사용 → 분포가 균일하게 되게 하여 activation의 원활한 진행

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

batch norm을 사용할 때 unit gaussian이 적합한지 판단해야 한다.

batch norm이 적절한지 판단은 학습에 의해 조절할 수 있다. 처음 normalize를 진행하고 이후 감마값과 같은 하이터 파라미터 값들을 조정하여 batch norm의 사용 여부를 선택할 수 있다.

감마값 = normalizing scale

베타값 = shift 조절

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$$

$$\beta^{(k)} = E[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

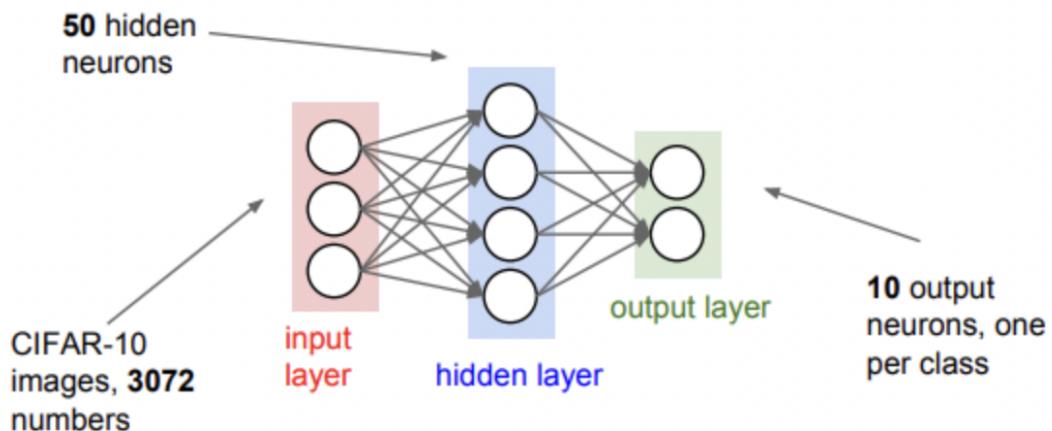
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Step2: Choose the architecture

사용할 neural architecture을 지정.

say we start with one hidden layer of 50 neurons:



위는 예시

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0) disable regularization
print loss
```

2.30261216167 ← loss ~2.3.
"correct" for 10 classes

returns the loss and the gradient for all parameters

sanity 체크를 통해 loss가 적절한지 확인

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3) crank up regularization
print loss
```

3.06859716482 ← loss went up, good. (sanity check)

regularization을 올렸을 때 loss 증가, network가 잘 동작하는지 확인.

그리고, 먼저 작은 데이터 셋을 이용해 훈련한다.

Lets try to train now...

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
model, two_layer_net,
num_epochs=200, reg=0.0,
update='sgd', learning_rate_decay=1,
sample_batches = False,
learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'

작은 데이터를 이용하고 있기 때문에 overfitting problem이 발생하고 train acc가 100%가 나온다.

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches=True,
                                    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

이후 적절한 hyper parameter을 설정한다. lr=1e-6일 때 loss가 조금씩 바뀌는 것을 확인할 수 있다. train acc가 조금씩 높아지는 것을 보면 train이 되는 것을 확인할 수 있다.

lr=1e6으로 설정했을 때, cost = NaN임을 확인할 수 있다. 너무 커서 그렇다.

lr=1e-5~1e-3의 값이 가장 적절한 값.

Hyperparameter optimization

범위를 크게 잡아 좁혀주는 방법을 사용한다.

First stage: only a few epochs to get rough idea of what params work

Second stage: longer running time, finer search

... (repeat as necessary)

Tip for detecting explosions in the solver:

If the cost is ever > 3 * original cost, break out early

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6) ←
    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                             model, two_layer_net,
                                             num_epochs=5, reg=reg,
                                             update='momentum', learning_rate_decay=0.9,
                                             sample_batches = True, batch_size = 100,
                                             learning_rate=lr, verbose=False)
```

note it's best to optimize
in log space!

```
val acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

주어진 범위에서 train을 시켜 val_acc가 낮은 learning rate를 확인한다. 여기서 찾은 best case와 비슷하게 영역을 설정해서 확인한다. 정확한 best case를 찾기 위해 좁은 범위에서 실험하는 것이 좋다.

이러한 hyperparameter를 찾기 위한 방법으로 grid search와 random search가 있다.

Random Search vs. Grid Search

Random Search for
Hyper-Parameter Optimization
Bergstra and Bengio, 2012

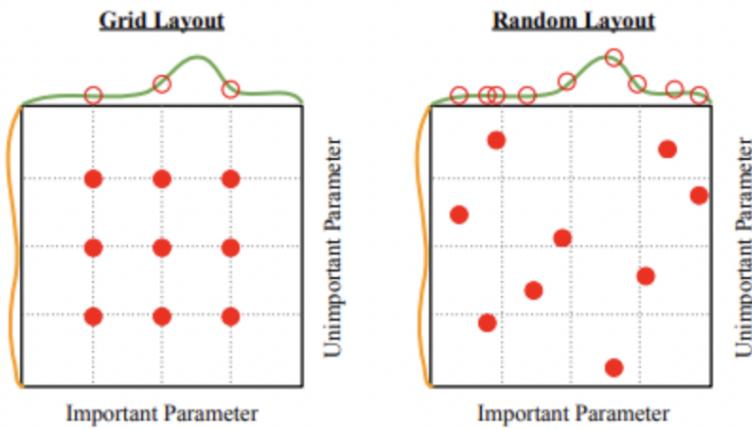


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

주로 random search를 사용한다.