**Notes of Study on Dec 15-16, 2022 (Thu - Fri)**

Based on lecture notes of Stanford CS231n

Module 1: Neural Networks

[Neural Networks Part 2: Setting up the Data and the Loss](https://cs231n.github.io/neural-networks-2/)

preprocessing, weight initialization, batch normalization, regularization (L2/dropout), loss functions

This section talks about setting up data and initial parameters for the model.

To start with, assuming the original data matrix **X** is of size (**N**, **D**), where

Take an example.

For a array A = np.array([[1, 2, 3], [4, 5, 6]]):

グラフィカル ユーザー インターフェイス, アプリケーション

自動的に生成された説明

Matrix size: [2x3] (A.shape = [2,3])

A single image size: [1x3] (.shape = [1,3])

Note:

**Data preprocessing for training data**

**(DO NOT modify test data, never!)**

1. **Substrate the mean of each dimension to make data0-centered.**

Note: must declaim **axis = 0**. By default, **axis = None** where the mean are based on all elements by each row and column .

1. **Normalize the data to be (approximately) equal-scaled.** 
   1. A common way is to divide each dimension by its standard deviation, i.e.
   2. For images, the relative scales of pixels are already approximately equal (and in range from 0 to 255), so it is not strictly necessary to perform this additional preprocessing step.

グラフ

自動的に生成された説明

* 1. Batch Normalization.

Refer to the paper [Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift](https://arxiv.org/pdf/1502.03167.pdf).

A technique developed in 2015 which properly initializes neural networks by explicitly forcing the activations. It takes on a unit gaussian distribution at the beginning of the training.

**Why do we do batch normalization?**

*‘’To understand the goal of batch normalization, it is important to first recognize that machine learning methods tend to perform better with input data consisting of uncorrelated features with zero mean and unit variance.*

*When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features. This will ensure that the first layer of the network sees data that follows a nice distribution.*

*However, even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance, since they are output from earlier layers in the network.*

*Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.’’*

**To overcome this problem:**

*‘’The authors propose to insert into the network layers that normalize batches. At training time, such a layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features. And the batch normalization layer includes learnable shift (called beta) and scale parameters (called gamma) for each feature dimension.’’*

In the implementation, applying this technique usually amounts to insert the BatchNorm layer immediately after fully connected layers (or convolutional layers), and before non-linearities.

For an easier understanding, refer to [Understanding the backward pass through Batch Normalization Layer](https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html), as well as python script layers.py of assignment 2, in functions batchnorm\_forward, batchnorm\_backward and batchnorm\_backward\_alt.

Batch normalization forward:

At training time, a batch norm layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

* Input of size (n, D), denoted as x in codes.
* Batch mean
* Batch variance
* Standardized batch

where (epsilon) is a constant for numeric stability.

* Normalized batch (scaled and shifted)

where (gamma) is the scaling factor and (beta) the shifting factor.

and are learnable parameters.

* Output of size (n, D)
* Implemented with numpy as:

if mode == "train":

mean = x.mean(axis=0)  *# (D, ) ; axis-0: vertically*

var = x.var(axis=0) *# (D, )*

std = np.sqrt(var + eps)  *# add eps for numeric stability; (D, )*

x\_std = (x - mean) / std *# standardized x; (N, D)*

out = x\_std \* gamma + beta *# scaled and shifted x\_std (N, D)*

cache = x, x\_std, mean, var, std, gamma *# save for backprop*

*# a running average of these means and standard deviations is kept during training, where momentum defines the percents of keeping the earlier value*

running\_mean = momentum \* running\_mean + (1 - momentum) \* mean

running\_var = momentum \* running\_var + (1 - momentum) \* var

elif mode == ‘test’:

*# at test time these running averages are used to center and normalize feature*

std = np.sqrt(running\_var + eps)  *# add eps for numeric stability; (D, )*

x\_std = (x - running\_mean) / std *# standardized x; (N, D)*

out = x\_std \* gamma + beta *# scaled and shifted x\_std (N, D)*

return out, cache

Batch normalization backward:

In this section, the goal is to find the gradients (derivatives) of 3 parameters: gamma, beta and batch data .

* Output out of size (n, D), whose derivative is denoted as dout()
* Scale parameter gamma of size (D, ), whose derivative is denoted as dgamma ():

By the chain rule,

While by , we have

Therefore,

=

Since is of size (D, ); of (n, D); and of (n, D),

then it’s implemented with numpy as:

dgamma = np.sum(dout \* x\_std, axis=0) *#axis=0: vertically*

*# Remember we store x\_std in cache?*

* Shift parameter beta of size (D, ), whose derivative is denoted as dbeta ():

By the chain rule,

While by , we have

Therefore,

=

Since is of size (D, ) and of (n, D),

then it’s implemented with numpy as:

(just do an adding-up vertically)

dbeta = np.sum(dout, axis=0) *#axis=0: vertically*

* Input of size (n, D), whose derivative is denoted as dX ():

Since

(1)

(2)

(3)

(4)

and by the chain rule:

(5)

where

(6)

is very hard to derive straightforwardly, so let’s do it step by step.

To be fixed: there are some errors in the derivation below.

??? means the derivation may be wrong (where I was not sure).

Hint: restart the derivation at the level of instead of . Get and then sum them up to get .

For the true derivation, refer to

[Deriving Batch-Norm Backprop Equations](https://chrisyeh96.github.io/2017/08/28/deriving-batchnorm-backprop.html)

and

[Understanding the backward pass through Batch Normalization Layer](https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html).

By the formular (2), we have

(7)

Recall for derivative functions U(x) and V(x), there is a rule:

(8)

So we let

(9)

By formula (7) (8) (9), we obtain

(10)

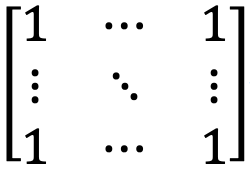
Thereby the task becomes to find and .

To find :

Since the mean of batch is defined as

(4)

then by the sum rule:

(11) 

The sum rule: the derivative of a sum of functions is the sum of their derivatives.

To find :

By formula (8) and (9),

(12)

let

(13)

go back to (12) with (13),

(14) =

where

(15)

let

(16)

go back to (15) with (16), we have

(17)

where

(18)

(neglect since it’s a constant)

)

where

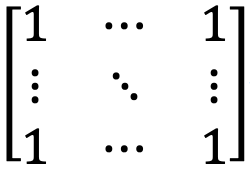
(19)

(20)

by the sum rule,

P.S. let

We thereby convert



???

(21)

Go back to (18) with (19), (20) and (21),

(21)

since = n

Thus, go back to (17) with (16) and (21):

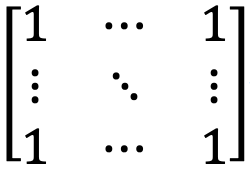
(22)

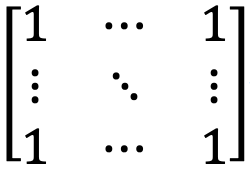
Go back to (14) with (13) and (22), we have

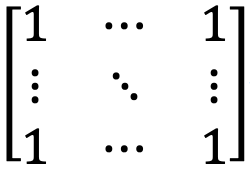
(23)

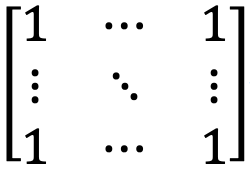
Finally we are moving back to (10) with (9), (11) and (23):

(24)







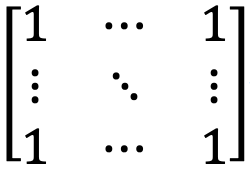
where 

(25) (NOT TRUE)

Combining formula (5), (6) and (24), finally we obtain

(26)

(NOT TRUE)

??? 

Which implemented with numpy as:

N, D = dout.shape

x, x\_std, mean, var, std, gamma = cache

one = np.ones((N, D))

dx = dout \* gamma \* (x\_std / x) \* (one - x \* x / 2 /N) (NOT TRUE)

dx = dout \* gamma \* (one \* (1 - 1/N) - x \* (x - mean) / (2\*N) ) / std

(NOT TRUE)

*# Refer to* [Understanding the backward pass through Batch Normalization Layer](https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html) *where the author used a computational graph.*

dx\_std = dout \* gamma *# (N,D)*

dstd = -np.sum(dx\_std \* (x - mean), axis=0) / (std\*\*2) *# (D, )*

dvar = 0.5 \* dstd / std *# (D, )*

dx1 = dx\_std / std + 2 \* (x-mean) \* dvar / N *# (N, D)*

dmean = -np.sum(dx1, axis=0)  *# (D, )*

dx2 = dmean / N *# (D, )*

dx = dx1 + dx2 *# (N, D)*

P.S. Difference between dot and \*

Fow A and B of type np.darray, np.dot(A, B) works as:

時計 が含まれている画像

自動的に生成された説明

arr0 **=** np.array([[1,2],[3,4]])

arr1 **=** np.array([[1,2],[3,4]])

**print(np.dot(arr0, arr1))**

*# [[ 7 10]*

*[15 22]]*

while A\*B works as:

**print(arr0 \* arr1)**

*# [[ 1 4]*

*[ 9 16]]*

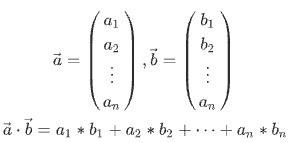
1. **PCA to minimize noises of 0-centered data.** 
   * 1. First get the covariance of data, by

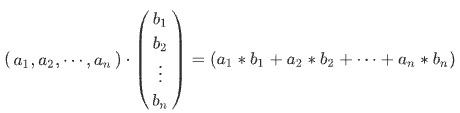
or

Note:

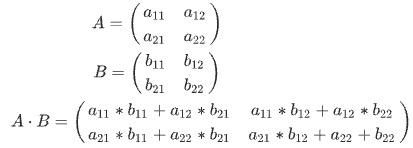
**.dot** and its calculation:

1. For vectors of same length





1. For matrix of same size:



1. For A of size [axb] and B of size [bxc],their inner vector is of [axc]. e.g.
   1. Then compute the SVD factorization of covariance matrix.

where

whose columns are the eigenvectors and orthogonal to each other with norm 1

whose columns are the singular values

…

For more information of SVD (singular value decomposition), refer to <https://qiita.com/kyoro1/items/4df11e933e737703d549> and <https://en.wikipedia.org/wiki/Singular_value_decomposition>**.**

* 1. Decorrelate the data by projection into eigenbasis
  2. Select the features with top Eigenvectors, e.g.

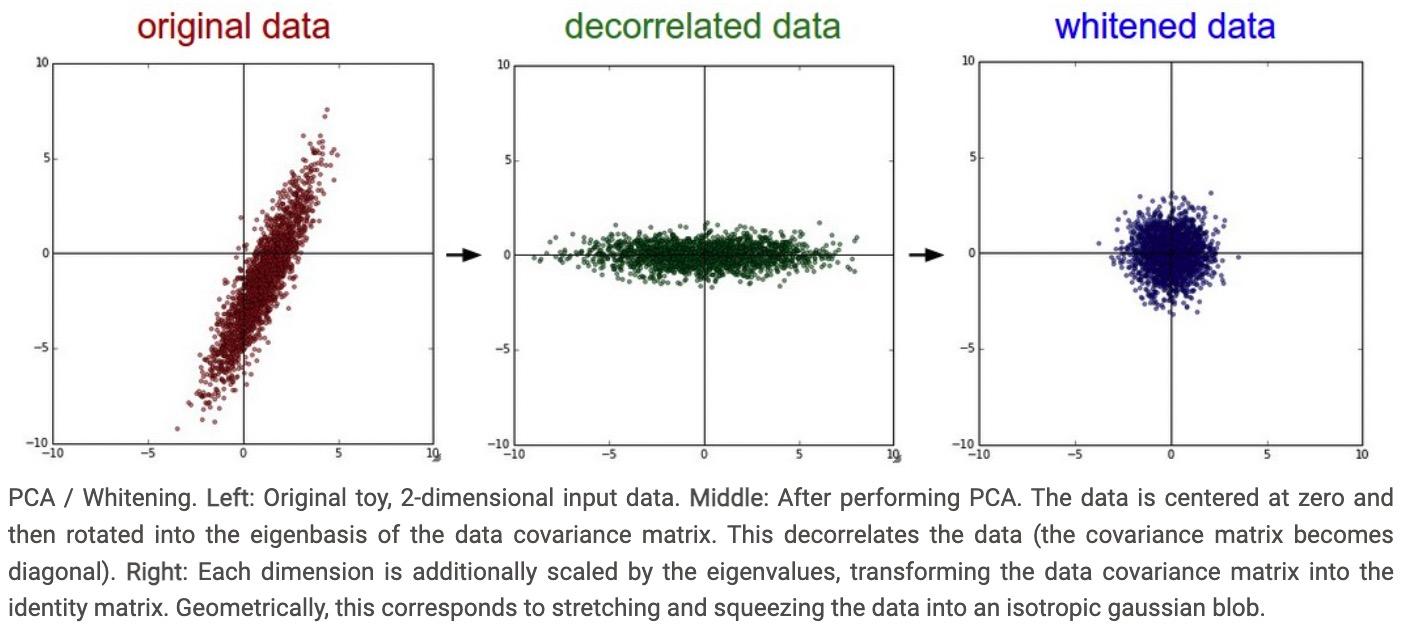
Which reduce the matrix to the size of [N x 100].

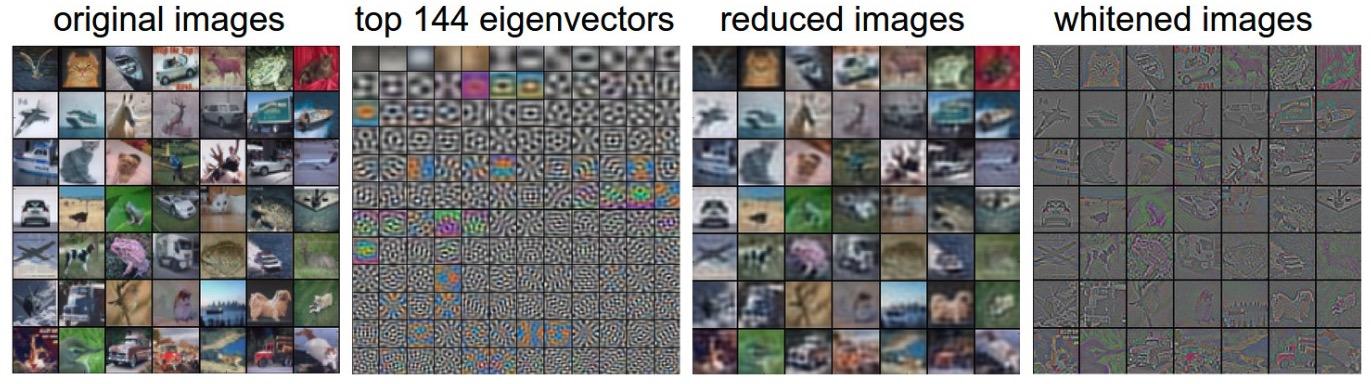
1. Whitening, another way to normalize the data:

Dividing **Xrot** by the eigenvalues (square roots of the singular values)

where 1e-5 is to prevent division by zero.

1. PCA and Whitening comparison.





**Initialize weights**

* 1. **Never starts with all zero weights!**

If every neuron in the network computes the same output, then

1. They will also all compute the same gradients during backpropagation, and
2. Undergo the exact same parameter updates.
   1. **Starts with relatively small random values**, like

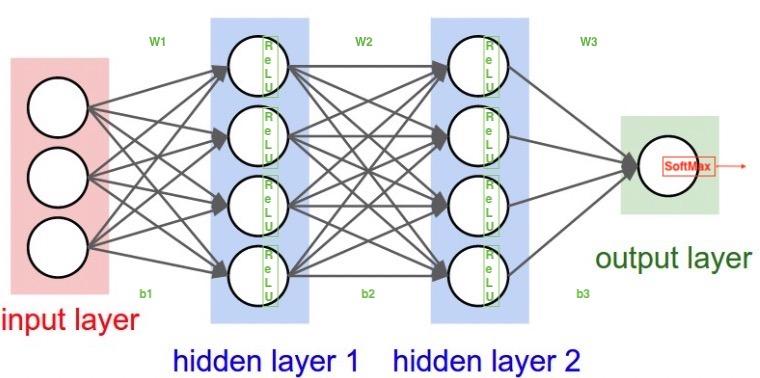
Where

And

np.random.randn: samples from a zero mean, unit standard deviation gaussian, and generate matrix of size [**HxD**].

For example, in the 3-layer neural network below, where the scores = np.dot(W,X) + b).

The sizes of parameters are:



1. Input data: [3x1] (3 rows, 1 dimension)
2. W1: [4x3] (4 neurons, 3 previous dimensions)
3. b1: [4x1]
4. W2: [4x4] (4 neurons , 4 previous dimensions)
5. b2: [4x1]
6. W3: [1x4] (1 neuron, 4 previous dimensions)
7. b3: [1x1]
8. Output data: [1x1]

Note:

In some cases, the scores are np.dot(X,W) +b). Then the matrix sizes can be different.

3**. Calibrating the variances with 1/sqrt(n)**.

1. One problem follows random values: the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.
2. Solution:

Let

where **n** is the number of its inputs.

For derivation, please check <https://cs231n.github.io/neural-networks-2/> at the section of ‘Calibrating the variances with 1/sqrt(n)’

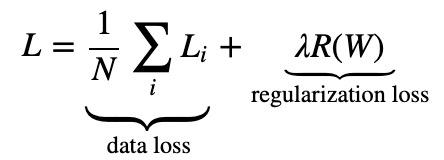
1. Specifically, let

**for ReLu neurons.**

1. Initializing biases with all zero is okay.

**Regularization**

* 1. Recalling total loss function, where **λ**(lambda) is the regularization strength.



* 1. L2 regularization: the most common form.

whose **gradient** is simply **λw**.

**W += -λ \* W** towards zero during gradient descent parameter update.

* 1. L1 regularization: less common one.

L1 regularization leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero) -- neurons end up using only a sparse subset of their most important inputs and become nearly invariant to the “noisy” inputs.

* 1. Dropout for regularization:

While training, keep a neuron active with some probability p, or setting it to zero otherwise.

We use dropout to prevent over-fitting.

For example, dropout for a 3-layer neural network

"""

Inverted Dropout: Recommended implementation example.

We drop and scale at train time and don't do anything at test time.

"""

p **=** 0.5 *# probability of keeping a unit active. higher = less dropout*

**def** **train\_step**(X):

*# forward pass for example 3-layer neural network*

H1 **=** np.maximum(0, np.dot(W1, X) **+** b1)

U1 **=** (np.random.rand(**\***H1.shape) **<** p) **/** p *# first dropout mask. Notice /p!*

H1 **\*=** U1 *# drop*

H2 **=** np.maximum(0, np.dot(W2, H1) **+** b2)

U2 **=** (np.random.rand(**\***H2.shape) **<** p) **/** p *# second dropout mask. Notice /p!*

H2 **\*=** U2 *# drop*

out **=** np.dot(W3, H2) **+** b3

*# backward pass: compute gradients... (not shown)*

*# perform parameter update... (not shown)*

**def** **predict**(X):

*# ensembled forward pass*

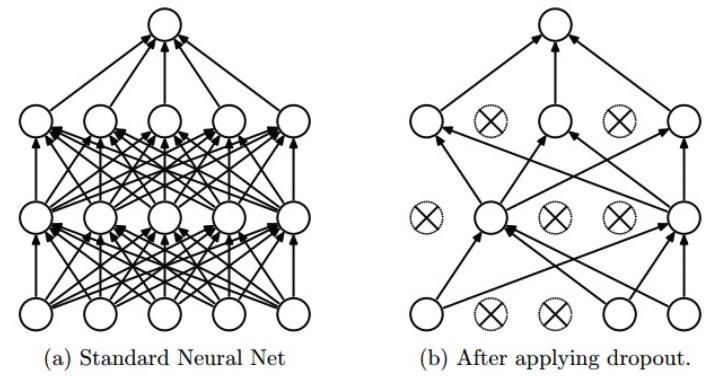
H1 **=** np.maximum(0, np.dot(W1, X) **+** b1) *# no scaling necessary*

H2 **=** np.maximum(0, np.dot(W2, H1) **+** b2)

out **=** np.dot(W3, H2) **+** b3

P.S. */p* ensure that the scale of outputs at test time is identical to the scale of the expected outputs at training time

General form of dropout:



Forward

p = 0.5 *# or 0.25, whatever between 1 and 0.*

if mode == "train":

*# only in training mode, randomly selected p portions of data to dropout*

mask = (np.random.randn(\*x.shape) < p) / p

out = x \* mask

cache = p, mask, mode

elif mode == ‘test’:

out = x

return out, cache

Backward:

if mode == "train":

*#*

dx = dout \* mask

elif mode == ‘test’:

dx = dout

return dx

Next study:

[Neural Networks Part 3: Learning and Evaluation](https://cs231n.github.io/neural-networks-3/)

gradient checks, sanity checks, babysitting the learning process, momentum (+nesterov), second-order methods, Adagrad/RMSprop, hyperparameter optimization, model ensembles