**Notes of Study on Dec 28 (Wed), 2022 & Jan 5 (Thu), 2023**

Based on lecture notes of Stanford CS231n

Module 1: Neural Networks

[Putting it together: Minimal Neural Network Case Study](https://cs231n.github.io/neural-networks-case-study/)

minimal 2D toy data example

This section talks about implementation of a linear classifier for data in 2 dimensions, and its extension to be a 2-layer neural network.

Task: to classify spiral data points generated as follows.

テキスト

自動的に生成された説明

グラフ, 散布図

自動的に生成された説明

Note:

**X**: data matrix of size [**NxD**], where

**N**: # points (valued 300 here)

**D**: dimensionality (valued 2 here)

テキスト

自動的に生成された説明…

**y**: class labels for each data point, of size [**Nx1**]

図形, 矢印

自動的に生成された説明

K: # classes

**A Linear Softmax Classifier**

ダイアグラム

自動的に生成された説明

1. Initialize parameters

*# initialize parameters randomly*

W **=** 0.01 **\*** np.random.randn(D,K)

b **=** np.zeros((1,K))

グラフィカル ユーザー インターフェイス, テキスト, アプリケーション

自動的に生成された説明

1. Compute class scores

*# compute class scores for a linear classifier*

scores **=** np.dot(X, W) **+** b

テキスト

自動的に生成された説明… of size [300x3].

1. Compute the loss

For Softmax classifier, the cross-entropy loss is as:

Where

can therefore be considered as

A larger score at its true class (bigger ), a small data loss (smaller ).

To calculate the data loss:

num\_examples **=** X.shape[0] *# 300*

*# get unnormalized probabilities*

exp\_scores **=** np.exp(scores)

*# normalize them for each example*

probs **=** exp\_scores **/** np.sum(exp\_scores, axis**=**1, keepdims**=**True)

*# get Li for each data point, with a total size of [300x1]*

correct\_logprobs **=** **-**np.log(probs[range(num\_examples),y])

テキスト

自動的に生成された説明…

…

Note:

Remember axis=1 means for all dimensions?

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

Recall the full Softmax classifier loss is defined as

Where

N: total # data points

: regularization strength, usually varies from 1e-5 to 1e-3.

Thus, to update total loss:

*# compute the loss: average cross-entropy loss and regularization*

data\_loss **=** np.sum(correct\_logprobs)**/**num\_examples

reg\_loss **=** 0.5**\***reg**\***np.sum(W**\***W)

loss **=** data\_loss **+** reg\_loss

Note at the beginning (with random parameters) it might give us loss = 1.1, which is -np.log(1.0/3), since with small initial random weights all probabilities assigned to all classes are about one third.

1. Gradient descent and backpropagation

Let

Then

By the chain rule, the gradient of Li with respect to score becomes:

Explanation:

Suppose the probabilities we computed were p = [0.2, 0.3, 0.5], and that the correct class was the middle one (with probability 0.3). According to this derivation the gradient on the scores df (in other words, dscores) would be dscores = [0.2, -0.7, 0.5].

The gradient of -0.7 is telling us that increasing the correct class score would lead to a decrease of the loss Li.

To get gradients on the scores, which is also the direction to update parameters W and b:

*# dscores = array holding , the gradient of scores(f) at which the loss(L) gets smaller*

dscores **=** probs *# dscores = (in other words, )*

*# ∂Li/∂fk = pk − 𝟙(yi=k). -1 only at its true class*

*# loss decreases while scores increase*

dscores[range(num\_examples),y] **-=** 1

*# the following division by the number of data has two benefits:*

*# 1. Normalization the batch size*

*# 2. Let the gradient changes smaller (sort of regularization)*

*# https://stackoverflow.com/questions/65275522/why-is-softmax-classifier-gradient-divided-by-batch-size-cs231n*

dscores **/=** num\_examples

Lastly, get the change of W and b (i.e. dW and db):

*# Refer to* [*https://people.cs.umass.edu/~sheldon/teaching/cs335/lec/17-neural-net-demo.html*](https://people.cs.umass.edu/~sheldon/teaching/cs335/lec/17-neural-net-demo.html)*, at the part of ‘Train a 2-Layer Neural Network Using Backprop’.*

*# dW = array holding , the gradient of weights (W) at which the loss (L) gets smaller.*

*# By the chain rule: where was got in the previous step as dscores. And by f = XW + b, = X.*

*# Note in this case is of size (D, K), same as W. And is of size (N, K); X is of size (N, D). To meet the requirements, we must do some manipulations, including transposing, and switching positions during dot multiplication.*

*# Therefore:*

dW **=** np.dot(X.T, dscores) *# dW =*

# Similarly, where = 1. Thus,

# Note: of size (1, K), or say, (K, ); and of size (N, K).

# Thus, we must merge data of into one row. i.e. np.sum(axis = 0).

*# Therefore:*

db **=** np.sum(dscores, axis**=**0, keepdims**=**True) *# db =*

*# And don’t forget the regularization loss!*

*# P.S.*  where by  *f = XW + b.*

*# to meet matrix manipulation format requirement,*

*# dX = np.dot(dscores, W.T)*

dW **+=** reg**\***W *# remember the regularization loss and its derivative?*

*# P.S. How about dX?*

*# dX = array holding , the gradient of data (X) at which the loss (L) gets smaller.*

*# Similarly, where = W. Thus, .*

*# Note dX is of size (N, D); df (dscores) of size (N, K); W of size (D, K).*

*# Therefore, to meet the size requirement:*

*# dX = np.dot(dscores, W.T) # dX =*

1. Update parameters W and b

*# perform a parameter update*

W **+=** **-**step\_size **\*** dW

b **+=** **-**step\_size **\*** db

Pulling it together: (code left and print right)

グラフィカル ユーザー インターフェイス, テキスト, アプリケーション

自動的に生成された説明**テキスト

自動的に生成された説明**

1. Calculate prediction accuracy

*# evaluate training set accuracy*

scores **=** np.dot(X, W) **+** b

predicted\_class **=** np.argmax(scores, axis**=**1)

**print** ('training accuracy: %.2f' **%** (np.mean(predicted\_class **==** y)))

*# print ~ 0.5.*

**An Extended 2-layer Neural Network**

ダイアグラム, 概略図

自動的に生成された説明

1. The forward computation of scores becomes:

*# evaluate class scores of size [Nxk] with a 2-layer Neural Network*

hidden\_layer **=** np.maximum(0, np.dot(X, W) **+** b) *# note, ReLU activation*

scores **=** np.dot(hidden\_layer, W2) **+** b2

1. While dscores remains the same form:

*# compute the class probabilities*

exp\_scores = np.exp(scores)

probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) # [N x K]

*# compute the gradient on scores*

*# dscores = array holding , the gradient of scores(f) at which the loss(L) gets smaller*

dscores = probs *# dscores = (in other words, )*

dscores[range(num\_examples),y] -= 1

dscores /= num\_examples

1. Backpropagation
   1. Backprop from scores to W2 and b2

dW2 = np.dot(hidden\_layer.T, dscores)

db2 = np.sum(dscores, axis=0, keepdims=True)

* 1. Backprop from scores to hidden layer

Since

Then

*# backprop from dscores to hidden layer*

dhidden = np.dot(dscores, W2.T)

*# backprop the ReLU non-linearity*

dhidden = dhidden \* (hidden\_layer > 0)

*# # which is equivalent to*

*# dhidden[hidden\_layer <= 0] = 0*

* 1. Backprop from hidden layer to W and b

dW **=** np.dot(X.T, dhidden)

db **=** np.sum(dhidden, axis**=**0, keepdims**=**True)

1. Regularization contribution

dW2 += reg \* W2

dW += reg \* W

1. Update weights and bias

W += -step\_size \* dW

b += -step\_size \* db

W2 += -step\_size \* dW2

b2 += -step\_size \* db2

1. Setting hyperparameters!

*# input X of size [300x2], where 300: # data, 2: D (dimensionality)*

*# i.e. X: [NxD] (note: let N = # data here)*

h **=** 100 *# parameter of hidden layer*

W **=** 0.01 **\*** np.random.randn(D,h)

b **=** np.zeros((1,h))

*# hidden layer of size [300x100]*

*# i.e. hidden layer output: [Nxh]*

W2 **=** 0.01 **\*** np.random.randn(h,K)

b2 **=** np.zeros((1,K))

*# final scores of size [300x3], where 3: K (# classes)*

*# i.e. scores: [NxK]*

Pulling all together:

*# initialize parameters randomly*

h **=** 100 *# size of hidden layer*

W **=** 0.01 **\*** np.random.randn(D,h)

b **=** np.zeros((1,h))

W2 **=** 0.01 **\*** np.random.randn(h,K)

b2 **=** np.zeros((1,K))

*# some hyperparameters*

step\_size **=** 1e-0

reg **=** 1e-3 *# regularization strength*

*# gradient descent loop*

num\_examples **=** X.shape[0]

**for** i **in** range(2000):

*# evaluate class scores, [N x K]*

hidden\_layer **=** np.maximum(0, np.dot(X, W) **+** b) *# note, ReLU activation*

scores **=** np.dot(hidden\_layer, W2) **+** b2

*# compute the class probabilities*

exp\_scores **=** np.exp(scores)

probs **=** exp\_scores **/** np.sum(exp\_scores, axis**=**1, keepdims**=**True) *# [N x K]*

*# compute the loss: average cross-entropy loss and regularization*

correct\_logprobs **=** **-**np.log(probs[range(num\_examples),y])

data\_loss **=** np.sum(correct\_logprobs)**/**num\_examples

reg\_loss **=** 0.5**\***reg**\***np.sum(W**\***W) **+** 0.5**\***reg**\***np.sum(W2**\***W2)

loss **=** data\_loss **+** reg\_loss

**if** i **%** 100 **==** 0:

**print** ("iteration %d: loss %f" **%** (i, loss))

*# compute the gradient on scores*

*# dscores = array holding , the gradient of scores(f) at which the loss(L) gets smaller*

dscores **=** probs *# dscores = (in other words, )*

dscores[range(num\_examples),y] **-=** 1

dscores **/=** num\_examples *# don’t forget it!*

*# backpropate the gradient to the parameters*

*# first backprop into parameters W2 and b2*

dW2 **=** np.dot(hidden\_layer.T, dscores)

db2 **=** np.sum(dscores, axis**=**0, keepdims**=**True)

*# next backprop into hidden layer*

dhidden **=** np.dot(dscores, W2.T)

*# backprop the ReLU non-linearity*

dhidden = dhidden \* (hidden\_layer > 0)

*# # which is equivalent to*

*# dhidden[hidden\_layer <= 0] = 0*

*# finally into W,b*

dW **=** np.dot(X.T, dhidden)

db **=** np.sum(dhidden, axis**=**0, keepdims**=**True)

*# add regularization gradient contribution*

dW2 **+=** reg **\*** W2

dW **+=** reg **\*** W

*# perform a parameter update*

W **+=** **-**step\_size **\*** dW

b **+=** **-**step\_size **\*** db

W2 **+=** **-**step\_size **\*** dW2

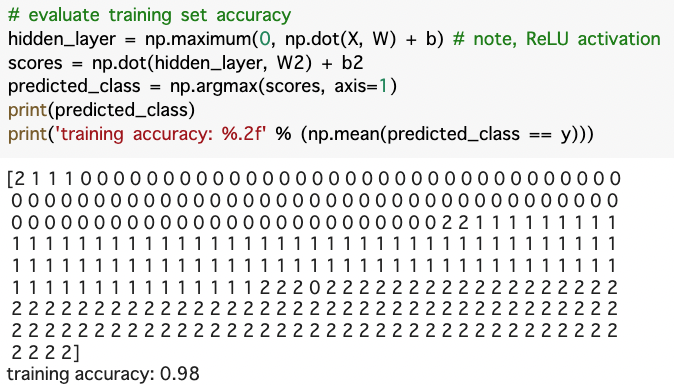
b2 **+=** **-**step\_size **\*** db2

which prints:

テキスト

自動的に生成された説明

1. Calculate prediction accuracy



**Note**

If implementing the algorithm in SVM hinge loss, especially when it comes to loss and gradient:

def svm\_loss\_vectorized(W, X, y, reg):

"""

Structured SVM loss function, naive implementation (with loops).

Inputs have dimension D, there are C classes, and we operate on minibatches

of N examples.

Inputs:

- W: A numpy array of shape (D, C) containing weights.

- X: A numpy array of shape (N, D) containing a minibatch of data.

- y: A numpy array of shape (N,) containing training labels; y[i] = c means

that X[i] has label c, where 0 <= c < C.

- reg: (float) regularization strength

Returns a tuple of:

- loss as single float

- gradient with respect to weights W; an array of same shape as W

"""

loss = 0.0

dW = np.zeros(W.shape) *# initialize the gradient as zero*

C = W.shape[1] *# W: [D, C]*

N = X.shape[0] *# X: [N, D]*

delta = 1

scores = np.dot(X, W) *# (N, C)*

*# to get the correct scores for each test point, in size of (N, ):*

correct\_scores = scores[np.arange(N), y] *# (N, ).*

margins = np.maximum(scores - correct\_scores.reshape(N, 1) + delta, 0) *# (N, C)*

*# reshape is essential because size (N, ) and (N, 1) is totally different in numpy*

margins[np.arange(N), y] = 0

*# don't forget to do the division by # training data (N)*

loss += np.sum(margins) / N

*# don't forget the regularization loss!*

loss += 0.5 \* reg \* np.sum(W \* W)

*# starting from socres = np.dot(X, W), propagate to dW = np.dot(X.T, dscores)*

*# answer to this TODO part is still kinda confusing... have to figure it out anyway*

dscores = np.zeros\_like(scores) *# (N, C)*

dscores[margins > 0] = 1

dscores[np.arange(N), y] -= np.sum(dscores, axis=1) *# (N, 1) = (N, 1)*

*# The following division by the number of data has two benefits:*

*# 1. Normalization the batch size*

*# 2. Let the gradient changes smaller (sort of regularization)*

*# https://stackoverflow.com/questions/65275522/why-is-softmax-classifier-gradient-divided-by-batch-size-cs231n*

dscores /= N

*# backprop ate the gradient to the parameters*

dW += np.dot(X.T, dscores)

*# And don't forget the regularization gradient! it changes weight too.*

dW += reg \* W

return loss, dW

P.S.

For an easier understanding, using loops to get loss and dW:

def svm\_loss\_naive(W, X, y, reg):

dW = np.zeros(W.shape)

loss = 0.0

num\_classes = W.shape[1]

num\_train = X.shape[0]

for i in range(num\_train):

scores = X[i].dot(W)

correct\_class\_score = scores[y[i]]

for j in range(num\_classes):

"""

Simply count the number of classes that didn’t meet the desired margin, i.e. margin > 0, and then the data vector xi scaled by this number is the gradient.

"""

if j == y[i]:

continue

margin = scores[j] - correct\_class\_score + 1 *# note delta = 1*

if margin > 0:

loss += margin

"""

when the desired margin is met:

dW[Not True Class] += X

dW[True Class] -= X

"""

dW[:,j] += X[i].T

dW[:,y[i]] -= X[i].T

loss /= num\_train

dW /= num\_train *# Remember it!*

*# Add regularization to the loss.*

loss += 0.5 \* reg \* np.sum(W \* W)

dW += reg \* W # add it

return loss, dW

**Fully Connected Multi-Layer Classifier**

1. Initialize parameters

Given hidden\_dims as a list of integers giving the size of each hidden layer,

thus

def \_\_init\_\_(

self,

hidden\_dims,

input\_dim=3 \* 32 \* 32,

num\_classes=10,

dropout\_keep\_ratio=1,

normalization=None,

reg=0.0,

weight\_scale=1e-2,

dtype=np.float32,

seed=None,

):

"""Initialize a new FullyConnectedNet.

Inputs:

- hidden\_dims: A list of integers giving the size of each hidden layer.

- input\_dim: An integer giving the size of the input.

- num\_classes: An integer giving the number of classes to classify.

- dropout\_keep\_ratio: Scalar between 0 and 1 giving dropout strength.

If dropout\_keep\_ratio=1 then the network should not use dropout at all.

- normalization: What type of normalization the network should use. Valid values

are "batchnorm", "layernorm", or None for no normalization (the default).

- reg: Scalar giving L2 regularization strength.

- weight\_scale: Scalar giving the standard deviation for random

initialization of the weights.

- dtype: A numpy datatype object; all computations will be performed using

this datatype. float32 is faster but less accurate, so you should use

float64 for numeric gradient checking.

- seed: If not None, then pass this random seed to the dropout layers.

This will make the dropout layers deteriminstic so we can gradient check the model.

"""

self.normalization = normalization

self.use\_dropout = dropout\_keep\_ratio != 1

self.reg = reg

self.num\_layers = 1 + len(hidden\_dims)

self.params = {}

"""

Initialize the parameters of the network, storing all values in the self.params dictionary.

Store weights and biases for the first layer in W1 and b1; for the second layer use W2 and b2, etc.

Weights should be initialized from a normal distribution centered at 0 with standard deviation equal to weight\_scale.

Biases should be initialized to zero.

When using batch normalization:

Store scale and shift parameters for the first layer in gamma1 and beta1; for the second layer use gamma2 and beta2, etc.

Scale parameters should be initialized to ones and shift parameters should be initialized to zeros.

"""

for index, (i, j) in enumerate(zip([input\_dim] + hidden\_dims, hidden\_dims + [num\_classes])):

self.params['W' + str(index+1)] = np.random.normal(0, weight\_scale, (i, j))

self.params['b' + str(index+1)] = np.zeros(j)

if normalization and index < self.num\_layers-1:

self.params['gamma' + str(index+1)] = np.ones(j)

self.params['beta' + str(index+1)] = np.zeros(j)

An illustration of parameters and their size (shape):

テーブル

自動的に生成された説明