Train Fully Connected Network

# Affine Forward (layers.py)

Affine forward is simple using weight and bias to calculate output, don’t use any activation function



**Input:**

- x: A numpy array containing input data, of shape (N, d\_1, ..., d\_k). Actually(we also can reshape out side, input can be like x(N,D) )

- w: A numpy array of weights, of shape (D, M)

- b: A numpy array of biases, of shape (M,)

**Output:**

- out: output, of shape (N, M)

- cache: (x, w, b)

Note: x van la size N\*d\_1\*…\*d\_k

**Implement:**

Step1: Reshape X into format X(N,N) ()

Step2: out=Wx+b (use dot product)

# Affine Backward(layers.py)

**Input:**

- dout: Upstream derivative, of shape (N, M)

- cache: Tuple of:

* x: Input data, of shape (N, d\_1, ... d\_k)
* w: Weights, of shape (D, M)

**Output:**

- dx: Gradient with respect to x, of shape (N, d1, ..., d\_k), backward to previous layer (in case we have many layers)

- dw: Gradient with respect to w, of shape (D, M)

- db: Gradient with respect to b, of shape (M,)

**Implement:**



*Calculate dx*

Step1:

dx= (N,M) \*(M,D)= N\*D Note: (M,D)= W transpose

Step2:

reshape dx

*Calculate dw:*

Step1: we have dy(N,M)

reshape X into format N\*D

Step2:

dW=Xtranpose\*dy (N,D).Tranpose \*(N,M)=dW(D,M)

*Calculate db:*

Sum through N dimension dout=dy(N,M) Db(M)

Db =np.sum(dout,axis=0)

# ReLu-forward

**Input:**

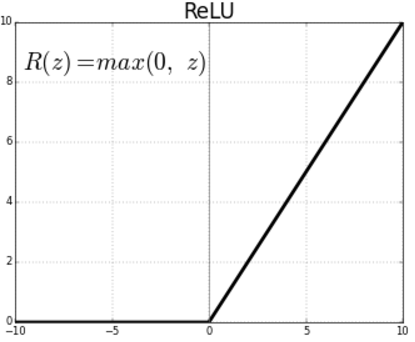
- x: Inputs, of any shape

**Output**:

- out: Output, of the same shape as x

- cache: x (remember to do backward)

**Implement:**

 Note: using np.maximum(0,x): is a max element-wise function

# ReLu-backward

**Input:**

- dout: Upstream derivatives, of any shape

- cache: Input x, of same shape as dout

**Output**:

- dx: Gradient with respect to x

**Implement:**

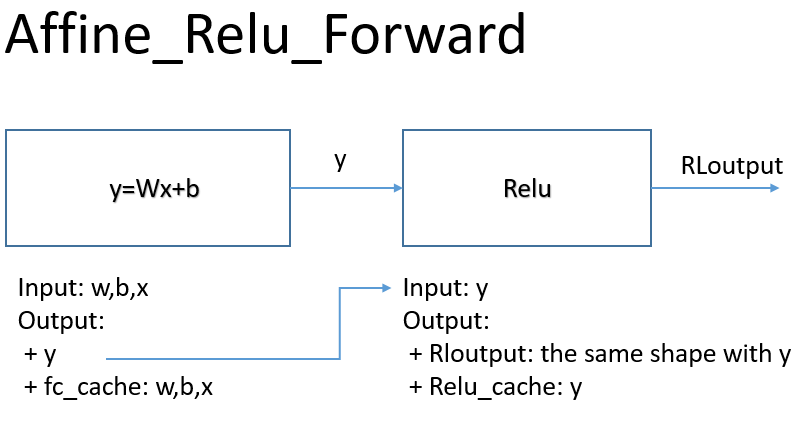
Only the value bigger than 0 have the gradient back ward flow.

dx=dout\*(x>=0)

# “Sandwich” layers

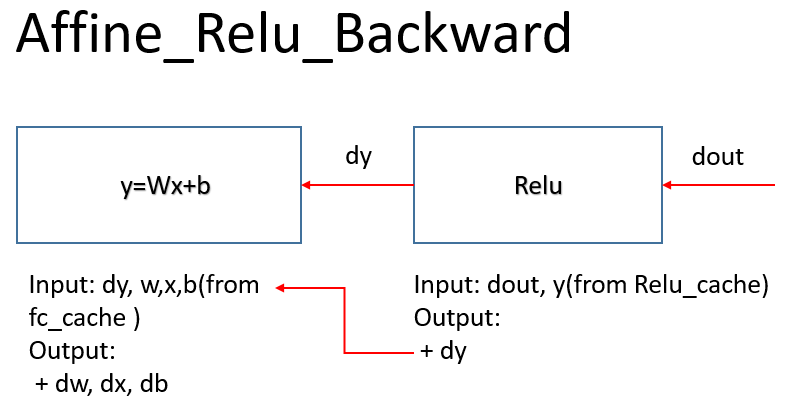
In this exercise, we combine the affine transform followed by Relu as an activation function (provided in layer\_utils)

**Affine\_relu\_forward**



**Affine\_relu\_backward:**

dout: normally we backward from the upper layer. In here, we random initialize it

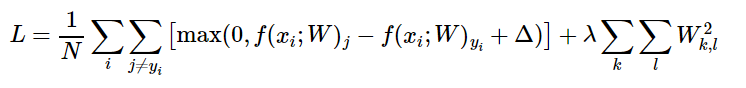


# Loss layers: Softmax and SVM

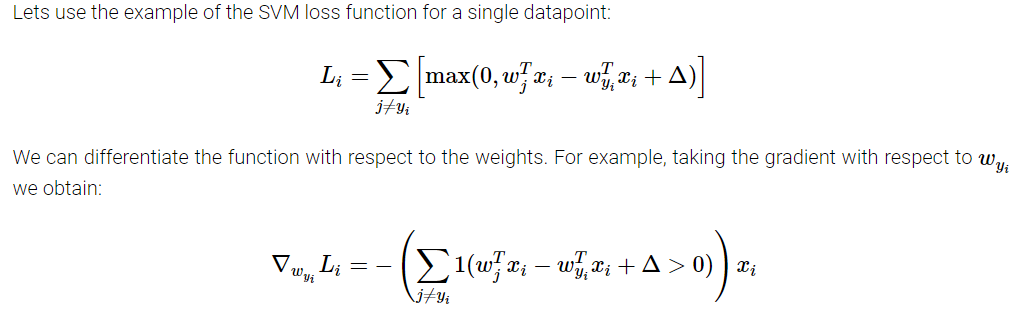
Reference link(Q2+Q3) and also can find in classnote 3 and 4: <https://bruceoutdoors.wordpress.com/cs231n-tutorials/>

## SVM

**SVM loss**: in this layers.py they didn’t add regularization term



**Gradient dw** (assigment1): but in this exercise, we need to calculate dx



**Implement:**

Inputs**:**

- x: Input data, of shape (N, C) where x[i, j] is the score for the jth class

for the ith input.

- y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and

0 <= y[i] < C

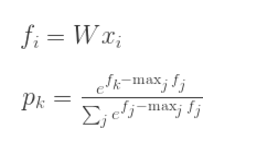
Output:

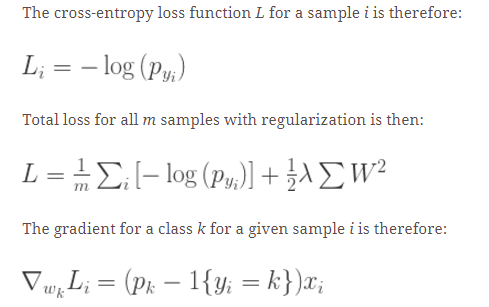
- loss: Scalar giving the loss

- dx: Gradient of the loss with respect to x

## Softmax:

**Loss and dw:** (but in here we need to calculate dx), in this layers.py they didn’t add regularization term





**Implement:**

Inputs:

- x: Input data, of shape (N, C) where x[i, j] is the score for the jth class (assignment 1: scores= W\*X with X(N,D) W(D,C) )

for the ith input.

- y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and

0 <= y[i] < C

Output:

- loss: Scalar giving the loss

- dx: Gradient of the loss with respect to x (N,C)

# Two-layer network

The architecure should be **affine - relu - affine - softmax.**

Note that this class does not implement gradient descent; instead, it

will interact with a separate Solver(solver.py) object that is responsible for running

optimization.

**Initialize network:**

Inputs:

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

- hidden\_dim: An integer giving the size of the hidden layer

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

- dropout: Scalar between 0 and 1 giving dropout strength.

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

initialization of the weights.

- reg: Scalar giving L2 regularization strength.



Implement:

-Initial weights according to size of layers and weight\_scale

-Initial bias = 0, size= size of layers(number of units)

**Loss:**

Inputs:

- X: Array of input data of shape (N, d\_1, ..., d\_k)

- y: Array of labels, of shape (N,). y[i] gives the label for X[i].

Output:

If y is None, then run a test-time forward pass of the model and return:

- scores: Array of shape (N, C) giving classification scores, where

scores[i, c] is the classification score for X[i] and class c.

If y is not None, then run a training-time forward and backward pass and

return a tuple of:

- loss: Scalar value giving the loss

- grads: Dictionary with the same keys as self.params, mapping parameter

names to gradients of the loss with respect to those parameters.

Implement:

* Step1: Implement affine\_forward through layers
* Step2: Check training/testing?
  + Testing: return output
  + Training:
    - Step1: apply SVM loss 🡪 output: loss and dscore
    - Step2: treat dscore for backward pass
      * Affine\_backward output-> hidden2 🡪 output: dw2, db2, dl1(input to hidden1)
      * Affine\_backward hidden2-> hidden1🡪 output: dw1,dw2,db1,db2

# Solver (Training model solver.py)

Train model, store loss function, choose the best parameters which worked on validation set

They use the implemented backward, forward, gradient and loss. To train through many interactions and epoch, also decide the way to update weights (adam, sgd momentum..)

"""

A Solver encapsulates all the logic necessary for training classification

models. The Solver performs stochastic gradient descent using different

update rules defined in optim.py.

The solver accepts both training and validataion data and labels so it can

periodically check classification accuracy on both training and validation

data to watch out for overfitting.

**To train a model**, you will first construct a Solver instance, passing the

model, dataset, and various optoins (learning rate, batch size, etc) to the

constructor. You will then call the train() method to run the optimization

procedure and train the model.

After the train() method returns, model.params will contain the parameters

that performed best on the validation set over the course of training.

In addition, the instance variable solver.loss\_history will contain a list

of all losses encountered during training and the instance variables

solver.train\_acc\_history and solver.val\_acc\_history will be lists containing

the accuracies of the model on the training and validation set at each epoch.

Example usage might look something like this:

data = {

'X\_train': # training data

'y\_train': # training labels

'X\_val': # validation data

'X\_train': # validation labels

}

model = MyAwesomeModel(hidden\_size=100, reg=10)

solver = Solver(model, data,

update\_rule='sgd',

optim\_config={

'learning\_rate': 1e-3,

},

lr\_decay=0.95,

num\_epochs=10, batch\_size=100,

print\_every=100)

solver.train()

A Solver works on a model object that must conform to the following API:

- model.params must be a dictionary mapping string parameter names to numpy

arrays containing parameter values.

- model.loss(X, y) must be a function that computes training-time loss and

gradients, and test-time classification scores, with the following inputs

and outputs:

Inputs:

- X: Array giving a minibatch of input data of shape (N, d\_1, ..., d\_k)

- y: Array of labels, of shape (N,) giving labels for X where y[i] is the

label for X[i].

Returns:

If y is None, run a test-time forward pass and return:

- scores: Array of shape (N, C) giving classification scores for X where

scores[i, c] gives the score of class c for X[i].

If y is not None, run a training time forward and backward pass and return

a tuple of:

- loss: Scalar giving the loss

- grads: Dictionary with the same keys as self.params mapping parameter

names to gradients of the loss with respect to those parameters.

"""

**Training:**

* Decide how many iteration for 1 epoch (depend on batch size)
* Decay leaning rate every epoch
* Store the best parameters (check > min loss of validation set => parameters of the best => best params). # Check train and val accuracy on the first iteration, the last iteration, and at the end of each epoch.d

# Multilayer Network

**Initial loss and gradient check:**

* Implement forward and backward for FC\_Network in file fc\_net.
  + Choose to using batch\_norm or not (in file layers.py)

explained post: <https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>

*Note: mean and variance at test time = average of running time . (a=monentun\*a+(1-momentum)\*mu)*

* + Choose to use dropout or not (in file layers.py)

Follow the post: <http://cs231n.github.io/neural-networks-2/#reg>

Implement drop\_out và và batch\_norm for the last part, design a best network. In file Batchnormalize.ipynb, Dropout.ipynd

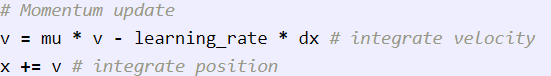
**Sanity check:**

After creating network, one of the sanity check is that we need to make sure that our network can overfit a small dataset.

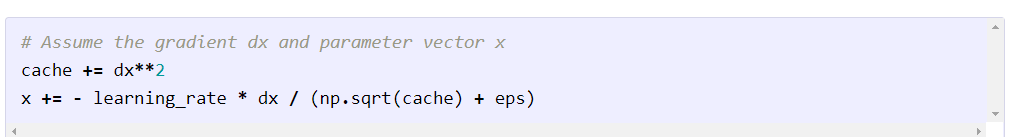
# Update rules

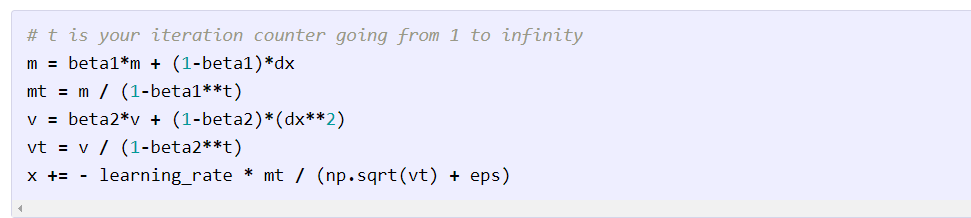
Reference: <http://cs231n.github.io/neural-networks-3/#sgd>

**Stochastic gradient descent with momentum** is a widely used update rule that tends to make deep networks converge faster than vanilla stochstic gradient descent.



**RMSProp and Adam:**





# Train a good model

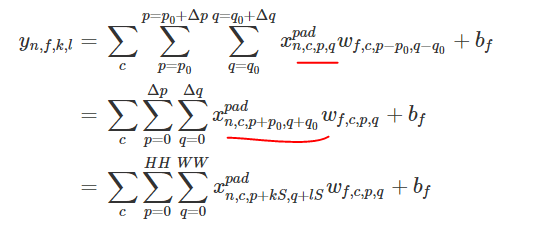
Using the batchnorm and drop\_out to desgin a good model

Train Convolutional Neural Network

Ref: <http://cthorey.github.io./backprop_conv/>

# Convolution: Naïve Forward part:

We can think that like, we use the filter to stride in both horizontal and vertical direction. Note: We need to move the pixel in the image base on the size and the stride of filter



"""

A naive implementation of the forward pass for a convolutional layer.

The input consists of N data points, each with C channels, height H and width

W. We convolve each input with F different filters, where each filter spans

all C channels and has height HH and width HH.

Input:

- x: Input data of shape (N, C, H, W)

- w: Filter weights of shape (F, C, HH, WW)

- b: Biases, of shape (F,)

- conv\_param: A dictionary with the following keys:

- 'stride': The number of pixels between adjacent receptive fields in the

horizontal and vertical directions.

- 'pad': The number of pixels that will be used to zero-pad the input.

Returns a tuple of:

- out: Output data, of shape (N, F, H', W') where H' and W' are given by

H' = 1 + (H + 2 \* pad - HH) / stride

W' = 1 + (W + 2 \* pad - WW) / stride

- cache: (x, w, b, conv\_param)

"""

# Aside: Image processing via convolutions (as sanity check)

Step1: read 2 image then resize image to 200x200

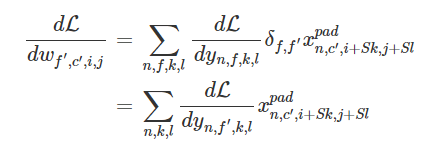
Step2: Design 2 filter to detect edge and transform to grey image.

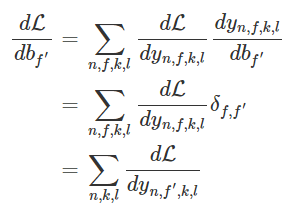
Detect edge is similar to sobel filter.

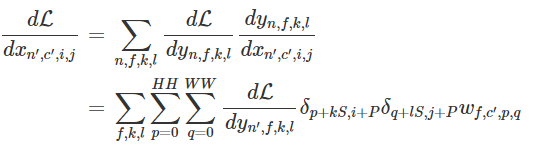
Transform to grey image: we need to add bias to filter

# Convolution: Naïve backward part:

We can find the explanation of backpropagation from this awesome post (many thanks for the author): <http://cthorey.github.io./backprop_conv/>



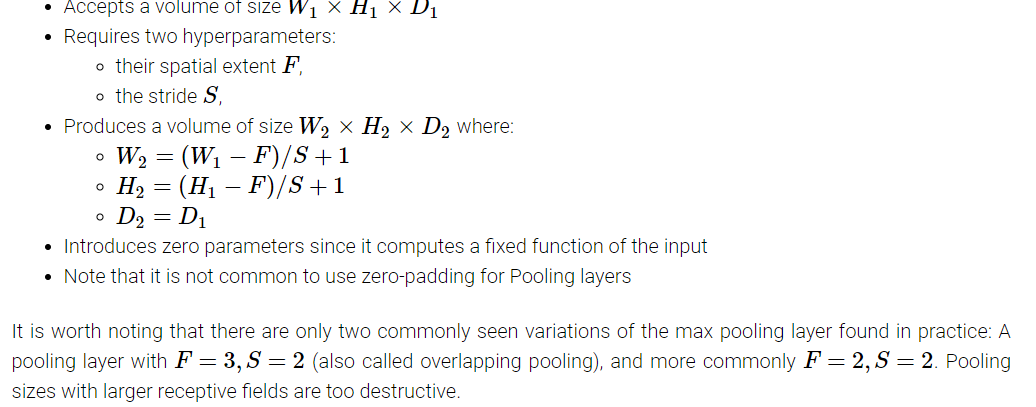




So many loops

# Max pooling: Naïve forward:

Ref: <http://cs231n.github.io/convolutional-networks/#pool>



*Loop through number of examples and channels.(2loops)*

*At each channel, calculate max at each spatial where the filter was applied (2 more loops)*

"""

A naive implementation of the forward pass for a max pooling layer.

Inputs:

- x: Input data, of shape (N, C, H, W)

- pool\_param: dictionary with the following keys:

- 'pool\_height': The height of each pooling region

- 'pool\_width': The width of each pooling region

- 'stride': The distance between adjacent pooling regions

Returns a tuple of:

- out: Output data

- cache: (x, pool\_param)

"""

# Fast Layers

Install the cython (use C to faster implement) to use the fast implement

Then test the time implement with the sandwich layers

# Three-layer ConvNet

"""

A three-layer convolutional network with the following architecture:

conv - relu - 2x2 max pool - affine - relu - affine - softmax

The network operates on minibatches of data that have shape (N, C, H, W)

consisting of N images, each with height H and width W and with C input

channels.

"""

def \_\_init\_\_(self, input\_dim=(3, 32, 32), num\_filters=32, filter\_size=7,

hidden\_dim=100, num\_classes=10, weight\_scale=1e-3, reg=0.0,

dtype=np.float32):

"""

A three-layer convolutional network with the following architecture:

conv - relu - 2x2 max pool - affine - relu - affine - softmax

The network operates on minibatches of data that have shape (N, C, H, W)

consisting of N images, each with height H and width W and with C input

channels.

"""

**Initial the weight and bias for conv layers and pooling layers.**

Conv layers: calculate the size of filter -> size of output (treat as input to calculate the next affine layers) -> the size of the weight (based on filter, and size of pooling filter) which will be random initialized (random with the weight-scale input)

**Calculate loss:**

*Forward pass:*

Step1: conv\_relu\_forward (store cache and output to calculate at hidden layer and backpropagation)

Step 2: affine\_relu\_forward (store cache,output)

Step3: affine\_relu\_forward (store cache, output)

*Calculate loss:*

Step1: softmax\_loss 🡪 return the loss and the dscores at the output to implement backprobagation

*Backward pass:*

Step1: affine\_backward 🡪 return dx3, dw3, db3.

Add regularization term to update dw3

Step2: affine\_relu\_backward🡪 return dx2,dw2,db2

Add regularization term to update dw2

Step3: conv\_relu\_pool\_backward🡪 return dx,dw1,db1

Add regularization term to update dw1

# Sanity check (Overfit small data set)

Use **Solver** to train the network

The network should be have the very high accuracy in training set, but very low accuracy in validation set (🡪 over fitting)🡪 learned model have a good capacity to train a big dataset

# Visualize Filters (visualize the weight of the conv filters)