Stanford CS231n CNN for Visual Recognition

Ian Poon

November 2024

1st course to CNNs by Stanford! Finally some coding and hands on! Many good reviews on this one unlike ahem...CS229. Also remember to return to psets and that few theory sections you skipped in CS229. To be fair, my primary focus then was the math given my background(and probably went beyond what was necessary...I don't think Andrew expected one to go through the nonlinear/convex analysis proofs for obvious reasons). Hopefully after CS231 you will feel more motivated for the "computing" aspect of CS229. CS231 gives practical guided applications of a lot CS229 content,really helps a lot.

Contents

1 Appendix 0: Matrix Derivatives

Aka baby tensor calculus

1.1 simplify simplify simplify

Example 1

Suppose we have a column vector $\vec{y}$ of length C that is calculated by forming the product of a matrix W that is C rows by D columns with a column vector $\vec{x}$ of lenght D

$$\vec{y} = W\vec{x}$$

Calculate $\frac{d\vec{y}}{d\vec{x}}$

Solution. Expressing everything as 1D variables we have

$$\begin{aligned}\frac{\partial \vec{y}_i}{\partial \vec{x}_j} &= \frac{\partial}{\partial \vec{x}_j}[W_{i,1}\vec{x}_1 + W_{i,1}\vec{x}_2 + \ldots + W_{D,1}\vec{x}_D] \\ &= 0 + 0 + \ldots + \frac{\partial}{\partial \vec{x}_j}[W_{i,j}\vec{x}_7] + \ldots + 0 \\ &= W_{i,j}\end{aligned}$$

Therefore

$$\frac{d\vec{y}}{d\vec{x}} = W$$

where W is also known as the **jacobian matrix**

1.2 Row vectors instead of column vectors

Example 2

Suppose we have a *row* vector $\vec{y}$ of length C that is calculated by forming the product of a matrix W that is D rows by C columns with a *row* vector $\vec{x}$ of lenght D

$$\vec{y} = \vec{x}W$$

Calculate $\frac{d\vec{y}}{d\vec{x}}$

Solution. Similar to above we find

$$\overrightarrow{y}_i = \sum_{j=1}^{D} \overrightarrow{x}_j W_{j,i}$$

which is equivalent to $\overrightarrow{x}_j W_{j,i}$(with summation convention which we assume for the rest of the notes) so we have that

$$\frac{\partial \overrightarrow{y}_i}{\partial \overrightarrow{x}_j} = W_{j,i}$$

Note that the indexing of W is now opposite as compared to the previous example. But we still arrive at

$$\frac{d\overrightarrow{y}}{d\overrightarrow{x}} = W$$

1.3 Dealing with more than two dimensions

Example 3

Suppose we have the same situation as above($y = xW$) but now we want to find

$$\frac{d\overrightarrow{y}}{dW}$$

Solution. again expressing as 1D variables notice that

$$\overrightarrow{y}_i = \overrightarrow{x}_1 W_{1,i} + \overrightarrow{x}_2 W_{2,i} + \ldots + \overrightarrow{x}_D W_{D,i}$$

therfore we see that if

$$F_{i,j,k} = \frac{\partial \overrightarrow{y}_i}{\partial W_{j,k}}$$

then

$$F_{i,j,i} = \overrightarrow{x}_j \text{ and 0 otherwise}$$

So replacing i, j with : we have

$$\frac{d\overrightarrow{y}}{dW} = \overrightarrow{x}$$

Corollary 4

If $y = Wx$ then we have

$$F_{i,i,j} = \overrightarrow{x}_j \text{ and 0 otherwise}$$

where

$$F_{i,j,k} = \frac{\partial \overrightarrow{y}_i}{\partial W_{j,k}}$$

1.4 Multiple data points

Example 5

Suppose now X is a 2D array with N rows and D columns while W is a matrix with D rows and C columns. Let Y be

$$Y = XW$$

which is an N row by C column matrix.Find $\frac{dY}{dX}$

Solution. It is clear to see that

$$Y_{i,j} = X_{i,k}W_{k,j}$$

reminder: note the use of summation convention of k here

Now this implies

$$\frac{\partial Y_{i,j}}{\partial X_{i,k}} = W_{k,j}$$

Let $Y_{i,:}$ and $X_{i,:}$ refer their respetive ith rows. Then we see that

$$\frac{\partial Y_{i,:}}{\partial X_{i,:}} = W$$

1.5 The chain rule in combination with vectors and matrices

Example 6

Consider $\overrightarrow{y} = VW\overrightarrow{x}$. But from above we know that

$$\frac{d\overrightarrow{y}_i}{d\overrightarrow{x}_j} = [VM]_{i,j} = V_{i,k}W_{k,j}$$

reminder: note the use of summation convention over k here

Note that if you recall `tensor analysis` this is essentially *covariant* transformation.

2 Module 0: Preparation

2.1 NumPy

`NumPy` is basically the `Matlab` for `Python`. Basically a toolset of working with multidimensional arrays efficiently

2.1.1 init and datatypes

Example 7 (NumPy Array Initialization)

Consider how we initialize arrays

```
1  import numpy as np
2
3  a = np.array([1, 2, 3])   # Create a rank 1 array
4  print(type(a))            # Prints "<class 'numpy.ndarray'>"
5  print(a.shape)            # Prints "(3,)"
6  print(a[0], a[1], a[2])   # Prints "1 2 3"
7  a[0] = 5                  # Change an element of the array
8  print(a)                  # Prints "[5, 2, 3]"
9
10 b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array: notice the argument is a list of lists
11 print(b.shape)                     # Prints "(2, 3)": 2 lists of length 3
12 print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

as you can see essentially `a.shape()` returns the dimensions of the array.For example a shape of (x, y) corresponds the familiar matrix with x rows and y columns. And in line 12 the indexing $b[i, j]$ where i is list index j is the element index of list i

Example 8 (NumPy Array Creation)

Consider how we create arrays

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

Analagous to `np.shape(x,y)` the arguments (x, y) in `np.full(),np.zeroes()` represent the creation of a 2D array.But instead of manually specifying the shape we can do

Example 9

Consider

```
>>import numpy as np
>>x = np.arange(6)
>>x = x.reshape((2, 3))
>>x
array([[0, 1, 2],
       [3, 4, 5]])
>>np.zeros_like(x)
array([[0, 0, 0],
       [0, 0, 0]])
```

Where the same automatically copying the shape of the array passed into the argument applies to

`ones_like(x),full_like(x,n),eye_like(x)`

Example 10

Consider another useful way of creating numpy arrays.

```
import numpy as np
a=np.arange(10)
print(a) #prints "[0,1,2,3,4,5,6,7,8,9]"
```

This is very much analagous to `a=[range(x,y,z)]` for normal python lists which recall creates a list of integers from x to y(not including) with step/interval z.

Remark 11. *Note that the spelling. it is "arange" with 1 "r" not to "rr"'s. In a sense it is referring to "a-range" nothing got to do with "arranging"*

Example 12

Create an array of evenly spaced intervals between a start and end point

```
>>import numpy as np
>>np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. ,  2.2,  2.4,  2.6,  2.8])
>>np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

retstep i means return step which returns an tuple containing the array and the step size as shown. The rest are self explanatory

2.1.2 indexing

Example 13 (NumPy Array Indexing)

Instead of just accessing individual elements like $b[i, j]$ above, like in python we can also do index slicing. Consider

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

where like ordinary python $[a : b]$ which is start at a before b(i.e. not including b)

Example 14

Note that there are conseuqneces of mixing integers with integer slicing

```
1  import numpy as np
2
3  # Create the following rank 2 array with shape (3, 4)
4  # [[ 1  2  3  4]
5  #  [ 5  6  7  8]
6  #  [ 9 10 11 12]]
7  a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
8  # see that each [[x,y,z]...] where [x,y,z] represents a row
9  # Two ways of accessing the data in the middle row of the array.
10 # Mixing integer indexing with slices yields an array of lower rank,
11 # while using only slices yields an array of the same rank as the
12 # original array:
13 row_r1 = a[1, :]    # Rank 1 view of the second row of a
14 row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
15 print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
16 print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
17
18 # We can make the same distinction when accessing columns of an array:
19 col_r1 = a[:, 1]
20 col_r2 = a[:, 1:2]
21 print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
22 print(col_r2, col_r2.shape)  # Prints "[[ 2]
23                              #          [ 6]
24                              #          [10]] (3, 1)"
```

as you can see in lines (13,14) `1` and `1:2` both index to the 2nd row of a. But "slices" the 2nd row maintaining the original rank of the array.

Example 15

Now consider what we call **integer array indexing**

```
1  import numpy as np
2
3  a = np.array([[1,2], [3, 4], [5, 6]])
4
5  # An example of integer array indexing.
6  # The returned array will have shape (3,) and
7  print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
8
9  # The above example of integer array indexing is equivalent to this:
10 print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
11
12 # When using integer array indexing, you can reuse the same
13 # element from the source array:
14 print(a[[0, 0], [1, 1]])  # Prints "[2 2]"
15
16 # Equivalent to the previous integer array indexing example
17 print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

That is, instead of passing a[slice/integer index] we pass a[list of slices/integer indexes]. The important point is to pay attention to the equivalence of the integer array slice with the usual way of creating an array with elements of another array using $np.array()$ as portrayed in lines

- $(9, 10)$ where we have

$$a[\underbrace{[0,1,2]}_{\text{indexes for dim 0}}, [0,1,0]] \Leftrightarrow [a[0,0], a[1,1], a[2,0]]$$

- $(16, 17)$ same concept

Example 16

Another useful trick with integer array indexing is selecting or mutating one element from each row of a matrix.

```
1  import numpy as np
2
3  # Create a new array from which we will select elements
4  a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
5
6  print(a)  # prints "array([[ 1,  2,  3],
7            #                [ 4,  5,  6],
8            #                [ 7,  8,  9],
9            #                [10, 11, 12]])"
10
11 # Create an array of indices
12 b = np.array([0, 2, 0, 1])
13
14 # Select one element from each row of a using the indices in b
15 print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
16 #note that this is equivalent to: print(a[[0,0],[1,2],[2,0],[3,1]])
17 # Mutate one element from each row of a using the indices in b
18 a[np.arange(4), b] += 10
19
20 print(a)  # prints "array([[11,  2,  3],
21           #                [ 4,  5, 16],
22           #                [17,  8,  9],
23           #                [10, 21, 12]])
```

For line 15 recall 10 what `arange(4)` does. Basically returns the array `[0,1,2,3]`. Note that with this we are saying

$$a[[0,1,2,3],[0,2,0,1]] = a[np.arange(4), b]$$

This is the essence of integer array indexing. The 1st list are the indexes for the 1st dimension while the second are for the 2nd dimension and so on.

Example 17

Finally there is something called **boolean array indexing**

```
import numpy as np

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

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

basically we pass the condition into the indice.i.e *a*[condition]

Example 18

The essentials of `NumPy` datatypes

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

You can't see really the effects of the force dytype here after all by default `np([list of integers])` by default is an array of int64 dtype. But it is more apparent in next example(see 20) where we floats.

Example 19

Another interesting method is `np.compress`

```
import numpy as np
a = np.array([[1, 2], [3, 4], [5, 6]])
np.compress([0, 1], a, axis=0) #returns [[3,4]]
np.compress([False, True, True], a, axis=0) #returns [[3,4],[5,4]]
np.compress([False, True], a, axis=1) #returns [[2],[4],[5]]
```

recall axis 0 is the 1st dimension so in the case of 2D it is vertical(i.e columns) while axis 1 is the horizontal(i.e rows).$[0, 1] = [False, True]$. The conditions inside determine whether to include that row or column(which is determined by the axis you specify in the arguments) in the copy of the input array.

2.1.3 pointwise operations

Example 20

Now here's something interesting. You may do element wise math operations with `NumPy` arrays. Unlike normal python lists which just do basic scalar addition or concatenation.

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

note the elementwise square root.

Example 21

Oh and speaking of concatanation consider

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0) #gets [[1, 2],[3, 4],[5, 6]]
np.concatenate((a, b.T), axis=1) #gets [[1, 2, 5],[3, 4, 6]]
np.concatenate((a, b), axis=None) #gets [1, 2, 3, 4, 5, 6]
```

The 1st cases should be clear. As a reminder, axis 0 refers to the columns,axis 1 the rows The last case is analogous to 23, sum without specifying an axis. So it just concatenates everything into a single 1D array irregardless of dimension etc.

Example 22

However one should note that $*$ refers to to element wise multiplication but `dot` computes inner product.

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v)) # 1x9+2x10,3x9+4x10

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

One should take note of the 3 cases here:

1. vector x vector

$$\begin{bmatrix} 9 & 10 \end{bmatrix} \begin{bmatrix} 11 & 12 \end{bmatrix}^T = 219$$

2. vector x matrix: we basically did

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 9 & 10 \end{bmatrix}^T = \begin{bmatrix} 29 \\ 67 \end{bmatrix}$$

3. matrix x matrix: we basically did

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Note the last case in this example is simply the standard matrix product

Example 23 (np.sum)

Another useful operation: `sum`

```
import numpy as np

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

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
print(np.sum(x,axis=1,keepdims=True)) #Same as above but keep dim. "[[3][7]]"
```

Example 24

Matrix transpose

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

2.1.4 broadcasting

Example 25

Broadcasting motivation

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

notice that this makes sense `x[i,:]` selects row i then we do a term-wise addition via `+x`.

Example 26

But we can do the above in a more compact way via **broadcasting**

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

To understand this first consider

Fact 27

Broadcasting rules

1. given 2 shapes (a, b, c) and (d, e)
2. we compare from the rightmost of each to the left. That is, in this case we compare $(e, c) \to (d, b) \to (a, \_)$
3. during comparison there are 3 cases
   (a) $\_a$(one of them is empty). Then a is assigned to the empty slot in the broadcast
   (b) a, b where $a == b$, then broadcast at that position is a or b
   (c) $a \neq b$ and one of them is 1, then broadcast the $\max(a, b) \neq 1$
   (d) $a \neq b$ but neither of them is 1 - *NOT COMPATIBLE*

To visualize this chunk of arabic consider examples broadcasting: a+b

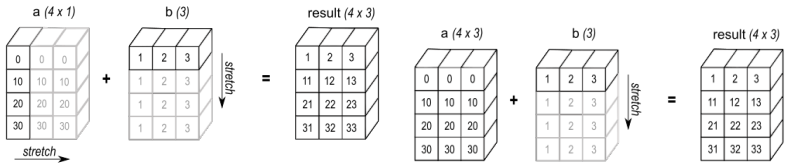
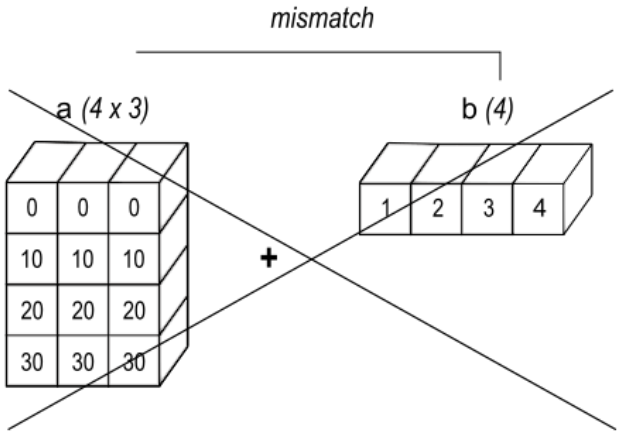


Figure 1: compatible as can get same dimensions by streching either axis on either block



Definition 28

To recap from tensor anaysis, recall an outer product is defined by

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{A} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \dots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \dots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \dots & u_m v_n \end{bmatrix}$$

Figure 2: Outer product

where $u \in \mathbb{R}^m$ and $v \in \mathbb{R}^n$. That is,

$$(u \otimes v)_{ij} = u_i v_j$$

Example 29

Compute outer product of vectors

```
import numpy as np
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5] comparison: a #    2
#  [ 8 10]             b #  3 1
#  [12 15]]              #  3 2
print(np.reshape(v, (3, 1)) * w)
```

To see how this made sense consider that 1st reshape to $(3, 1)$ did this

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} \rightarrow \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array}$$

then broadcasting did the following pointwise multiplication which yielded the result as stated above

$$\begin{array}{|c|c|} \hline 1 & 1 \\ \hline 2 & 2 \\ \hline 3 & 3 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 4 & 5 \\ \hline 4 & 5 \\ \hline 4 & 5 \\ \hline \end{array}$$

Example 30

However more directly an outer product can be calculated by

```
x = np.array(['a', 'b', 'c'], dtype=object)
np.outer(x, [1, 2, 3])
array([['a', 'aa', 'aaa'],
       ['b', 'bb', 'bbb'],
       ['c', 'cc', 'ccc']], dtype=object)
```

Example 31

Continuing from above, we now Add a vector to each row of a matrix

```
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6] comparision: a: 2 3
#  [5 7 9]]             b    3
print(x + v)               2 3
```

basically recall

$$v = \boxed{1}\boxed{2}\boxed{3}$$

then on broadcasting $x + v$ we have the poinwise addition

$$x + v = \begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5&6\\\hline\end{array} + \begin{array}{|c|c|c|}\hline 1&2&3\\\hline 1&2&3\\\hline\end{array}$$

I think you get the idea by now.

2.2 SciPy

Numpy as mentioned earlier does cool stuff with n-d arrays. Scipy builds on this and makes it cooler. We cover some specific applications to basic image file handling and operations.

Example 32 (Image tinting and resize)

Please first refer to the section on image classifiers on `Module 1` below to learn more about image files read by SciPy and RGB values before returning. The following will then make sense:

```
from scipy.misc import imread, imsave, imresize

# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9] #elementwise multipication after broadcasting

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

The key takeaways use is the general procesing of image file reading and basic editiong using SciPy. In particular pay attention to the use of `imread,imsave,imresize`

Figure 3: Original image left,new image right

2.3 MatPlotLib

2.3.1 Plotting

We'll cover the basics of plotting using `MatPlotLib` here.

Example 33

Consider

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # Compute the x and y coordinates for points on a sine curve
5  x = np.arange(0, 3 * np.pi, 0.1)
6  y = np.sin(x)
7
8  # Plot the points using matplotlib
9  plt.plot(x, y)
10 plt.show()  # You must call plt.show() to make graphics appear.
```

Fpr line 5 again recall `arange` from 10. So essentially we are plotting x from 0 to 3π for every 0.1. For line 9 see that we are the arguments of plot are of the form $plot(x, f(x))$ which takes the points to plot and function of the points to plot in the 1st and 2nd argument repsectively

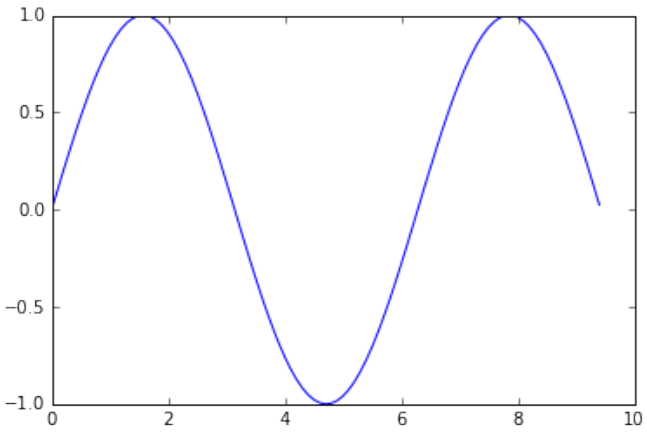


Figure 4: Plot for 33

Example 34

What if we want to plot multiple graphs on the same axis? What if we want to label these graphs as well as the axes? Then consider

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

See that `xlabel,ylabel` are properties of `plt` which specifically refer to the x and y axis labels respectively. title isn self explanatory. Legend takes in a list of plot labels and assigns them to the plot in the order in which they wre created. For instance in this case we defined the sin graph first so the `'Sine'` label is assigned to it. show is self explanatory.

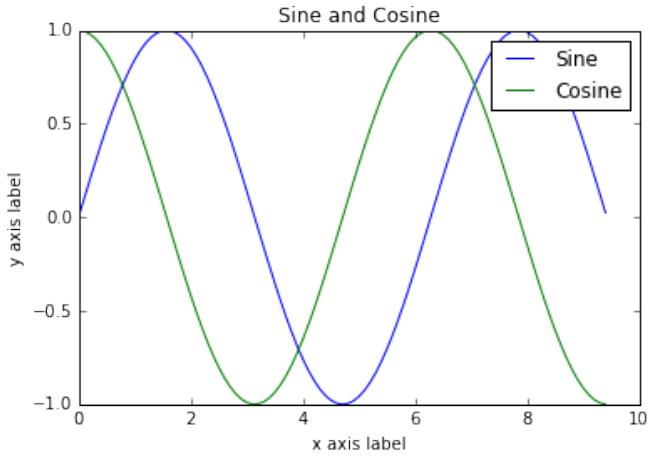


Figure 5: Plot for 34

Example 35

Finally note that if we do note specify the y points in the plot function by default they take $0, 1, 2, 3 \ldots$ that is

```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10, 5, 7])
plt.plot(ypoints)
plt.show()
```

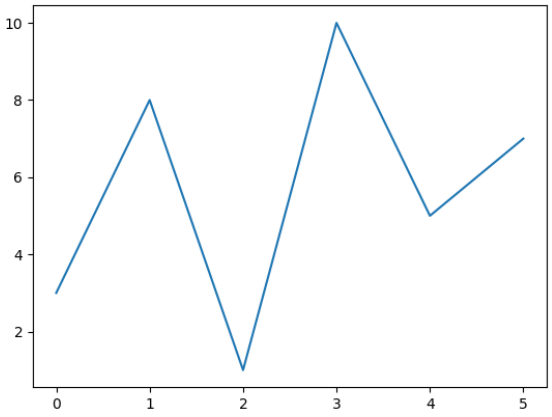


Figure 6: Ouput for the above

2.3.2 subplots

Basically if you want to have multiple plots in the same figure(not just one like we had above). That is plt refers to a single "figure". Then the "subplots" refer to the plots on this figure.

Example 36

For instance consider

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active via the last arguement "1"
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot via the last arguement "2"
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

to summarize for `plt.subplot(x,y,z)` where

- x is the height
- y is the width
- z can be seen as the position of the subplit

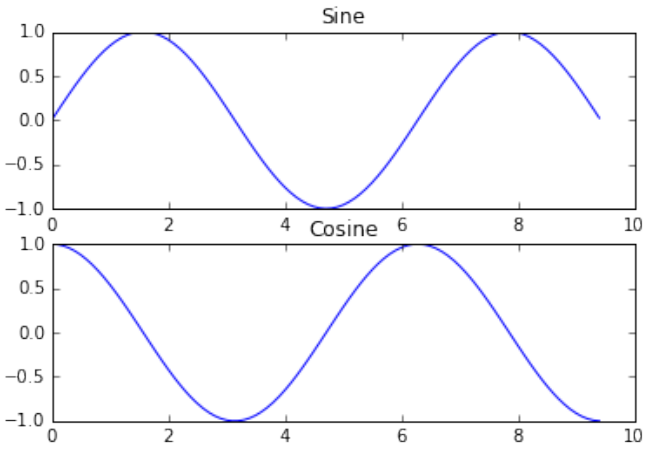


Figure 7: Plot for 36

2.3.3 Images

Example 37

What if we want to use plots with images? Recalling 43 Now consider

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt
#recall an our image(2D array of pixels) is height x
#width x 3 rgb values(describing the color of a single pixel)
img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]
# Show the original image
plt.subplot(1, 2, 1)
plt.imshow(img)

# Show the tinted image
plt.subplot(1, 2, 2)

# A slight gotcha with imshow is that it might give strange results
# if presented with data that is not uint8. To work around this, we
# explicitly cast the image to uint8 before displaying it.
plt.imshow(np.uint8(img_tinted))
plt.show()
```

The key idea is that now the images are displayed in subplots(i.e they have axes).Also note the use of the `plt.imshow` object method. It basically draws the image on the current figure "plt" in our case. Therefore we must call it *before* we call "plt.show" which shows the figure.Otherwise if you call "plt.show" first, if you haven't drawn anything what do you show?(so an error will result)

Now, in this example our output is

Figure 8: in our case the cat image is 400 pixels in height and 428 pxiels in width. the individual pixels$[i, j]$ on this 2D array represented on a 2 axes plot reflect the rgb value

2.3.4 Scatter plots

Example 38

Another useful plotting function in matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y, color = 'hotpink')

x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```

Very intuitive to understand no explanation required

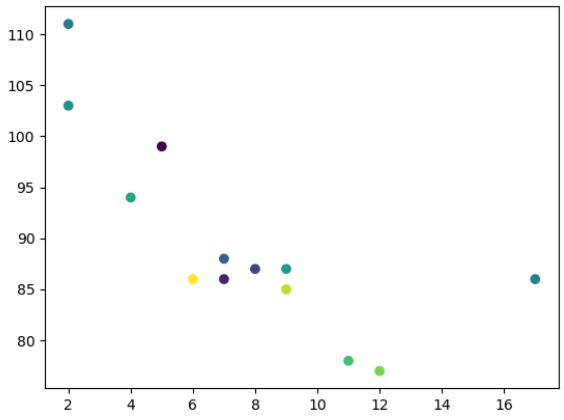


Figure 9: Output of the above example 38

and some basic data analysis stuff

Example 39

np mean

```
>>import numpy as np
>>a = np.array([[1, 2], [3, 4]])
>>np.mean(a)
2.5
>>np.mean(a, axis=0)
array([2., 3.])
>>np.mean(a, axis=1)
array([1.5, 3.5])
```

Example 40

Consider

```
import numpy as np
a = np.arange(6).reshape(2,3) + 10
a
array([[10, 11, 12],
       [13, 14, 15]])
np.argmax(a)
5
np.argmax(a, axis=0)
array([1, 1, 1])
np.argmax(a, axis=1)
array([2, 2])
```

basically returns the indices of the maximum element

the `np.std` can be used similarly.

3 Module 1: Neural Networks

3.1 Image Classifier

We briefly mentioned on we can use SciPy to tint images in 32. But first we discuss what exactly an image file is

Name	Color	Code	RGB	HSL
white		#ffffff or #fff	rgb(255,255,255)	hsl(0,0%,100%)
silver		#c0c0c0	rgb(192,192,192)	hsl(0,0%,75%)
gray		#808080	rgb(128,128,128)	hsl(0,0%,50%)
black		#000000 or #000	rgb(0,0,0)	hsl(0,0%,0%)
maroon		#800000	rgb(128,0,0)	hsl(0,100%,25%)
red		#ff0000 or #f00	rgb(255,0,0)	hsl(0,100%,50%)
orange		#ffa500	rgb(255,165,0)	hsl(38.8,100%,50%)
yellow		#ffff00 or #ff0	rgb(255,255,0)	hsl(60,100%,50%)
olive		#808000	rgb(128,128,0)	hsl(60,100%,25%)
lime		#00ff00 or #0f0	rgb(0,255,0)	hsl(120,100%,50%)
green		#008000	rgb(0,128,0)	hsl(120,100%,25%)
aqua		#00ffff or #0ff	rgb(0,255,255)	hsl(180,100%,50%)
blue		#0000ff or #00f	rgb(0,0,255)	hsl(240,100%,50%)
navy		#000080	rgb(0,0,128)	hsl(240,100%,25%)
teal		#008080	rgb(0,128,128)	hsl(180,100%,25%)
fuchsia		#ff00ff or #f0f	rgb(255,0,255)	hsl(300,100%,50%)
purple		#800080	rgb(128,0,128)	hsl(300,100%,25%)

Figure 10: Color codes

Now consider that chonky cat image we used in 32. In that case our image consists of $248 \times 400 \times 3$ numbers. This means the image is 248 pixels wide, 400 pixels long amd each pixel contains 3 digits each ranging from 0(black) – 255(white) corresponding to the RGB color codes

3.1.1 Nearest Neighbour Classifier

This is just a very basic 1st approach to image classification, nothing to do with CNNs but we use it as a way develop certain foundational ideas about the image classication problem. Implementation details on KNN/nearest neighbour are left to the assignment 1(Q1: KNN), here we outline some key concepts.

Problem 41

Say we are given a set of images. For example the `CIFAR-10` dataset which consists of 60000 tiny images that are 32 pixels high and wide. Each image is labelled with one of 10 classes(eg. "airplane,automobile...etc"). Now we

1. set aside 50000 images to be used as the training images
2. we wish to label the remaining 10000 images(which we call the training images) with their correct category $0 - 9$

So one simple way to do so is to compare 2 images is simply compare the images pixel by pixel and add up the differences. Then we simply assign the test image the label of the training image nearest in "pixel distance" to it. To

that end we first define

Definition 42

The **L1 distance**

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

Fact 43

Note that you could see an image as a 2d array with each pixel at row i,column j specified by RGB 3 values.

In our case, each of the 50000 images are 32(height) x 32(width) 2D arrays of pixel. And the color of each pixel is specified by the 3 number RGD color code.

Another distance measure we could use is

Definition 44

The **L2 distance** is defined by

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

so basically the L1 distance is the L1 norm while the L2 distance is the L2 norm as so on.

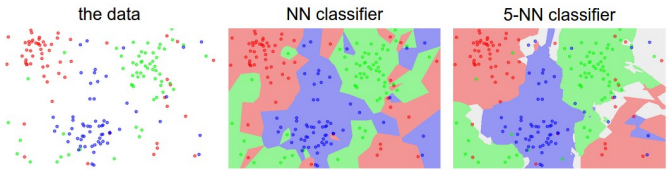
However if you ran code(see assignment 1(Q1: KNN)) using this as a means to classify the CIFAR-10 dataset, you will find that it is rather inaccurate. Specifically it has $35,38\%$ accuracy for $L1, L2$ distance repsectively which is far cry from $94,95\%$ by humans and CNNs respectively. Yes you heard me right.

CIFAR-10 classification: CNNs>humans in accuracy

Read on to know more.

3.1.2 K-Nearest Neighbor Classifier

Now lets try to improve on this. Instead of just taking the label of the closest neighbour we do take the top *k-nearest neighbours* and have them "vote" on the label of the test image. In particular the case for $k = 1$ is just the *nearest neighbour classifier* described above. To visualize this see



The voting process for your information involves taking the mean among the group of k classifiers. Again refer to assignment 1(Q1: KNN) for implementation details.

3.1.3 Validation sets for hyperparameter tuning

Now this begs the question:

Question 45. *What k works best? Should we use the L1 norm or the L2 norm?*

Now these choices are known as **hyperparameters**.

Fact 46

Note that it is best practice not to choose hyperparamters based on the **test set** since this might lead to **overfitting**(which basically means accurate for the current data but not accurate for future unknown data sets)

If that is the case, how then should we choose our hyperparamters.The answer is that we use what we call a *validation set* .

Definition 47

Basically it involves splitting our **training set** into two. The slightly smaller set will be the **validation set**.

Example 48

Using the above CIFAR-10 exampe, we could use the 49000 of the training images for training and leave 1000 aside for validation.

Another way to evaluate hyperparamters is the use of **cross validation**. We will explore the implementation details of the aforementioned evalution methods for hyperparamters in assignment 1(Q1: KNN) too.

3.2 Linear Classification

As hinted in the last example on image classification of the CIFAR-10, we now develop the concept of CNNs which apparently were superior to humans in that example. We do so by first introducing two key components

1. the **score function**
2. the **loss function**

3.2.1 Parametrized mapping from images to label scores

Starting with defining the score function we first we assume a training dataset of images $x_i \in R^D$ each associated with label y_i. Here $i = 1, \ldots, N$ and $y_i \in 1 \ldots K$(these were analagous to the categories $0 - 9$ in the previous example).

Definition 49

The score function is one that maps the pixel values of an image to "confidence scores fo each class". In particular it is map

$$f : R^D \to R^k$$

from *raw image pixels* to *class scores*

Definition 50

The **linear classifier** is defined by

$$f(x_i, W, b) = Wx_i + b$$

Definition 51

Where

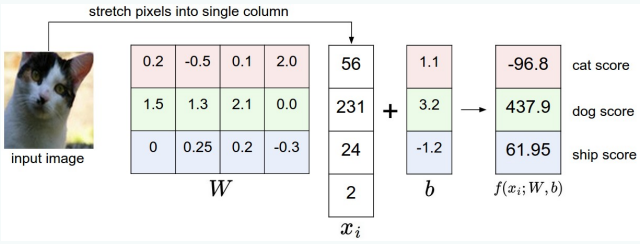
- The matrix W(of size $K \times D$)
  we often call W the weights
- the vector b(of size $K \times 1$)
  we often call b the bias vector

are the **paramters** of the function.

Let us try to interpret what this means.

Example 52

Consider the CIFAR-10 example we had from earlier



- the weights has the capacity to like or dislike certain colors at a certain position.
- For instance you can imagine the "ship class" might be more likely if there is a lot of blue on the sides of the image. Then you might expect the "ship classifier" to have lots of positive weights across its blue channel weights(increases ship score) and negative for the red/green channels(decreases ship score)

Before proceeding consider

Fact 53

The **Bias trick**. It is common simplifying trick to represent the two paramters W, b as one. To be specific recall we have the score function

$$f(\underbrace{x_i, W, b}_{\text{parameters}}) = Wx_i + b \in \text{class scores}$$

but what if we want to change this to

$$f(x_i, W) = Wx_i$$

How we do this is essentially via

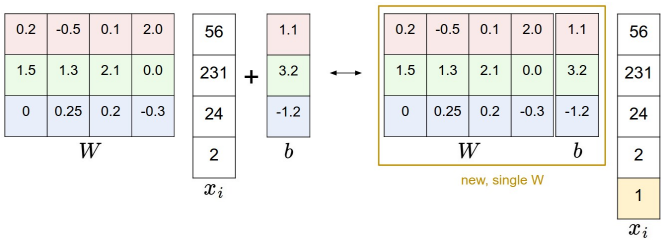


Figure 11: the matrix product/addition operation on the LHS and RHS are equivalent!where W is defined in 51

Also note that

Remark 54. *It is good practice to* **center your data**. *That is in the case our turn out pixel values from* $[0 - 255]$ *to* $[-1, 1]$. *The idea is so that we get zero mean centering. We will learn why later when you learn about* **gradient descent**

3.2.2 Loss Function

Definition 55

Essentially the **loss function**(also caled the **cost function** or the **objective function** (if you recall optimization and CS229) is an indicator of how good a job we are doing. The loss will be high if we are doing a poor job of classifying the linear data and low otherwise.

Like we did for the score function let us try to interpret what the loss function.

Example 56

Using the same example as in 52 see that the the particular set of weights performed rather poorly. Since we are given an image of a cat but the cat score came out low compared to the dog and ship score. So in this case the loss function ought to be a high value.

There are several ways to define the details of the loss function. The first example we will introduce is

Definition 57

Known as the **multiclass support vector machine**(SVM) loss. It is defined by

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

where $s_j = f(x_i, W)_j$ is the score for the for the j-th class of the ith element. Δ is some "fixed margin" in which the SVM wants the correct class for each image to score higher than incorrect margins. That is to say we want

So we see that if any score lies in the red region or higher, there will be an accumulated loss.
Again lets intepret this with an example.

Example 58

Suppose that we have three classes that receive the scores $s = [13, -7, 11]$ and that the first class(i.e $y_i = 0$) is the true class in reality. Assume $\Delta = 10$. Then the above expression sums over all incorrect classes($j \neq y_i$) so we get

$$\begin{aligned} L_i &= \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10) \\ &= 0 + 8 \end{aligned}$$

Now because we are working with linear score functions that is we may express $f(x_i, W) = Wx_i$ we may write L_i equivalently as

$$L_i = \sum_{j \neq y_i} \max(0. w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

where w_j is the j-th row of W(from 51) reshaped as a column

Definition 59

The $\max(0, -)$ function is known as the **hinge loss**

However there is one problem with the loss function we presented above. Suppose that we have a dataset and a set of parameters **W** that correctly classify every example(i.e all scores are so that all margins are met/$L_i = 0$ for all i). The issue is that this set of **W** is not necessarily unique, meaning there could be other **W** that can also correctly classify the examples.

Example 60

An easy example will be all the other scalar multiples of W

$$\lambda W$$

where $\lambda > 1$. This is because such a transformation uniformly stretches all score magnitudes and hence also their absolute differences. So ifthe difference between the correct and wrong class is 15, multiplying **W** by 2 will make the difference 30 instead.

We would like to encode some preference for a certain set of weights **W** over others to remove this ambiguity. To that end we extend the loss function with a **regularization penalty** $R(W)$

Example 61

A common example of a regularization penalty is the square L2 norm that discourages large weights through an elementwise quadratic penalty over all parameters

$$R(W) = \sum_k \sum_\ell W_{k,\ell}^2$$

Note that here we are basically summing up all the squared elements of W

With this our Multiclass SVM becomes

$$L = \frac{1}{N} \sum_i L_i + \lambda R(W)$$

where the 1st term corresponds to the data loss and the second the regularization loss. Expanding this out in full form we obtain

$$L = \frac{1}{N}\sum_i \sum_{j\neq y_i}[\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + \Delta] + \lambda \sum_k \sum_\ell W_{k,\ell}^2$$

3.2.3 Softmax classifier

In turns out that Multiclass SVM is one of two commonly seen classifiers(among the many other options beyond these 2). The other popular choice is the **softmax classifier**

Fact 62 (Similarities with Multiclass SVM)
Note

- In the softmax classifier the function mapping $f(x_i, W) = Wx_i$(i.e it still linear as the name of this section suggests).
- The full loss of the dataset is the mean of L_i over all training examples together with the regularization term $R(W)$

Definition 63
However we replace the hinge loss from above with something called a **cross-entropy loss** that has the form

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \quad \text{or equivalently} \quad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

where we are using the notation f_j to mean the j-th element of the vector of class scores f

Definition 64
The function

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{e^{z_k}}}$$

is called the **softmax** function. Recall CS229 for derivation

Now there are various ways to make sense of this

information theory view

Definition 65
The **cross entropy** between a "true" distribution p and an estimated distribution q is defined as

$$H(p, q) = -\sum_x p(x)\log q(x)$$

The softmax classifier is hence minimizing the cross-entropy between the estimated class probabilities($q = e^{f_{y_i}}/\sum_j e^{f_j}$) and the "true distribution which in this intepretation is the distribution where all probability mass is on the correct class(i.e $p = [0, \ldots, 1, \ldots, 0]$ contains a single 1 at the y_i-th position).

probabilistic interpretation

Fact 66

Upon further inspection of the softmax function we notice that may express

$$P(y_i|x_i;W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

which can be interpretated as the (normalized) probability assigned to the correct label y_i given the image x_i and paramterized by W.Recall CS229 for derivation

To see this, remember that the Softmax classifier inteprets the scores inside the output vector f as the unnormalized log probabilities.

Fact 67 (practical issues)

Note that intermediat terms like $e^{f_{y_i}}$ and $\sum_j e^{f_j}$ may be very large due to the exponentials. Because in computers, dividing large numbers may be numerically unstable it is often important to use a normalization trick as folows

Multiply top and bottom of fraction by a constant C of our choice and see that

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j} + \log C}$$

where log refers to ln here. Recall that $e^{\ln C} = C$. A common choice for C is that

$$\log C = -\max_j f_j$$

Problem 68

Derive the expression for the gradient(with respect to weights) of the cross entropy loss in 63 which we denoted to be

$$L_i = -f_{y_i} + \log \sum_j e^{f_j}$$

Essentially we have to find $\frac{\partial L_i}{\partial W}$. As we did in 3 expressing in terms of single variables and using chain rule we have

$$\frac{\partial L_i}{\partial W_{a,b}} = \frac{\partial L_i}{\partial [f_k]_c} \cdot \frac{\partial [f_k]_c}{\partial W_{a,b}}$$

Reminder: note the use of summation convention over c here. But we also know that $f_k = X[i]W$. Hence again from 3, replacing a, b with : we have that

$$\frac{\partial L_i}{\partial W} = \frac{\partial L_i}{\partial f_k} \cdot X[i]$$

where we know that $L_i \in \mathbb{R}$ and $f_i \in \mathbb{R}^C$. But by observation we immediately see that

$$\begin{aligned}
\frac{\partial L_i}{\partial f_k} &= \frac{\partial(-f_u)}{\partial f_k} + \frac{\partial \ln(\sum_j e^{f_j})}{\partial f_k} \\
&= -\delta_{k,y_i} + \frac{1}{\sum_j e^{f_j}} \cdot \frac{\partial \sum_j e^{f_j}}{\partial f_k} \\
&= -\delta_{k,y_i} + \frac{e^{f_k}}{\sum_j e^{f_j}}
\end{aligned}$$

where δ_{k,y_i} is the kronecker delta function with is 1 when $k = y_i$ and 0 otherwise. The 2nd term is the familiar *softmax function* we defined earlier! Now notice that $\frac{\partial L_i}{\partial f_k}$ has shape (C) while $\frac{\partial f_k}{\partial W}$ has shape (D). So how do we calculate this? Normal matrix multiplication rules don't make sense? This is because as you know from `tensor analysis` this a limited way of thinking about matrix multiplication. We should be thinking in terms of tensors and that the matrix transformation in our case corresponds to a *covariant transformation*. With this perspective consider that

$$\frac{\partial L_i}{\partial W_{a,b}} = \frac{\partial L_i}{\partial [f_k]_c} \cdot \frac{\partial [f_k]_c}{\partial W_{a,c}} = \frac{\partial L_i}{\partial [f_k]_c} \cdot [X[i]]_a$$

so the cth element of a lenght C vector $\frac{\partial L_i}{\partial f_k}$ multiplies the ath element of lenght D vector $\frac{\partial f_k}{\partial W}$. And since c, a are arbitrary this clearly corresponds to the outer product as defined in 28. Note that if you recall `tensor analysis` a, c are what we call "free indices". Or explicitly

$$\left[\frac{\partial L_i}{\partial W}\right]_{ab} = \left[\frac{\partial L_i}{\partial f_k}\right]_c [X[i]]_a$$

3.3 Optimization

Recall that previously we have introduced two key components in the context of the image classification task

1. a(parameterized) **score function** from raw image pixels to class scores(eg. a linear function)
2. a **loss function** that measures the quality particular set of parameters with regards to the performance of the classifier when this set of parameters are used. (eg. Softmax/SVM)

3.3.1 Computing the gradient

There are two ways to compute the gradient.The first is a slow,approximate but easy way(**numerical gradient** and a fast,exact but more error prone way that requires calculus(**analytic gradient**)

Computing the gradient numerically with finite differences

Consider a naive implementation of numerical gradient of f at x where

- f should be a function that takes a single argument
- x is the point (numpy array) to evaluate the gradient at

```
def eval_numerical_gradient(f, x):
  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished: #it.finished returns true when the pointer
  #has reached the end of the interable array
```

Remark 69. *note that* `nditer` *numpy method that allows for efficient interation and operations on a numpy array*

Example 70

the arguement `flags=['multi_index']` basically allows for tracking/flagging of each iterated element by its multi index. More precisely consider

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
# Without multi_index
it1 = np.nditer(x)
for elem in it1:
    print(elem)  # Outputs 1, 2, 3, 4 (in a flat iteration)
# With multi_index
it2 = np.nditer(x, flags=['multi_index'])
for elem in it2:
    print(it2.multi_index, elem)  # Outputs (0, 0) 1, (0, 1) 2, (1, 0) 3, (1, 1) 4
```

Example 71

The argument `op_flags=['readwrite']` means that operation flags are set to read and write. This means when you iterate via the pointer say `it1,it2`, then we may directly modify the array w iterate

```
x = np.array([1, 2, 3])
# Read-only iteration (default)
it1 = np.nditer(x)
for elem in it1:
    print(elem)  # This works, but you can't modify the elements.
# Read-write iteration
it2 = np.nditer(x, op_flags=['readwrite'])
for elem in it2:
    elem[...] += 10  # Modifies the element directly
print(x)  # Output: [11, 12, 13]
```

```
  # evaluate function at x+h
  ix = it.multi_index #in our case this is simply the 1D index
  #of the element interated in the array
  old_value = x[ix]
  x[ix] = old_value + h # increment by h
  #essentially the ith coordinate of x is now x_i+h so
  fxh = f(x) # evalute f(x + h)
  x[ix] = old_value # restore to previous value (very important!)

  # compute the partial derivative
  grad[ix] = (fxh - fx) / h # the slope
  it.iternext() # step to next dimension
```

```
  return grad
```

Effectively what is going on is

$$\text{grad}[i] = \frac{\partial f(\mathbf{x})}{\partial x_i} = \left(f([x_1 \ldots, x_i + h, \ldots, x_n]) - f(x_1 \ldots, x_i, \ldots, x_n)\right)/h$$

where h is small

Computing the gradient analytically with Calculus

Example 72

Recall 3.2.2 SVM loss function for a linear classifier we have

$$L_i = \sum_{j \neq y_i} [\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)]$$

then differenting this with respect to the weights w_{y_i} we have

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)\right) x_i$$

where recall $1(\ldots)$ is the indicator function. Well this is because all summands in the sum have w_{y_i} in it and that in each summand the other terms are just constants. If they we have a non-zero gradient it must be $-x_i$ clearly. The condition $w_j^T x_i - w_{y_i}^T x_i + \Delta > 0$ must obviously be satisfied for non zero because the original function is hinge loss as mentioned in 59. If its negative then will be $L_i = 0$

And similarly $j \neq y_j$ we have

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

Where this time only 1 summand in sum has w_j in it and that is when j in $\sum_j$ corresponds to it. Now obviously x_i will be the gradient is its non-zero.

That is to say the (i,j) element in the gradient matrix dW corresponds to the derivative of L_i(loss function of item i) with respect to w_j(weights for class j)

3.3.2 Gradient Descent

Now that we can copute the gradient of the loss function the now consider the procedure of repeatedly evaluating the gradient and then performing a parameter update known as **gradient descent**. The "vanilla" version is as follows

```
while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

3.4 Backprogation

3.4.1 compound expressions, chain rule, backprogation

Example 73

Consider

$$f(x, y, z) = (x + y)z$$

The green numbers in the diagram below should be pretty obvious. See that $(-2 + 5) * (-4) = -12$

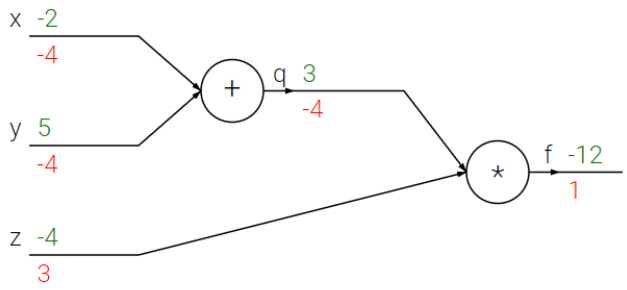


Figure 12: red numbers represent the partial derivative of f with respect to the node variable where the number is directly behind

For example for the "*" node we have $\frac{\partial f}{\partial q} = z = -4$. To see this consider the code

```
# set some inputs
x = -2; y = 5; z = -4
# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12
# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
dqdx = 1.0
dqdy = 1.0
# now backprop through q = x + y
dfdx = dfdq * dqdx  # The multiplication here is the chain rule!
dfdy = dfdq * dqdy
```

3.4.2 Modularity: Sigmoid example

Example 74

Consider

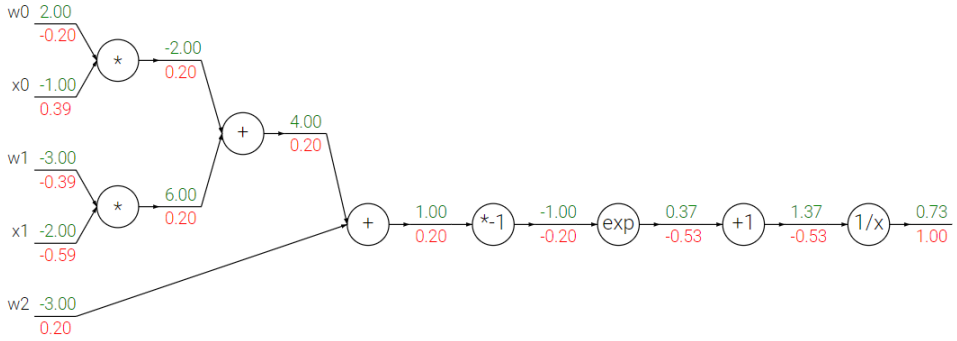
$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

Observe

$$f(x) = \frac{1}{x}\frac{df}{dx} = -1/x^2$$
$$f_c(x) = c + x\frac{df}{dx} = 1$$
$$f(x) = e^x\frac{df}{dx} = e^x$$
$$f_a(x) = ax\frac{df}{dx} = a$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$
$$\rightarrow \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

Which is represented here



For example for the $1/X$ node we know that $d/dX(1/X) = -X^{-2}$. In our case we have $0.73 = \sigma = 1/X$ where $X = 1+e^{-x}$. So solving for X then subbing into $-X^{-2}$ gets -0.53. Do similar to get the rest of the red numbers(this process is called backpropagation). In code we have

```
w = [2,-3,-3] # assume some random weights and data
x = [-1, -2]
# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function
# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

3.4.3 Backprop in practice: stages computation

Example 75

Consider

$$f(x,y) = \frac{x+\sigma(y)}{\sigma(x)+(x+y)^2}$$

Then the forward pass(which the green numbers of the diagrams above) is calculated in code by

```
x = 3 # example values
y = -4
# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                 #(8)
```

Notice that we have structured code in such a way that it contains multiple intermediate variables namely

$$\texttt{sigy}, \texttt{num}, \texttt{sigx}, \texttt{xpy}, \texttt{xpysqr}, \texttt{den}, \texttt{invden}$$

each of which are only simple expressions for which we already know the local gradients.Note for the following any additional "d" in front of the variables above indicate holding the gradient of the *output*(i.e f) of the circuit with respect to that variable.For example

$$dnum = \frac{\partial f}{\partial num} \quad dx = \frac{\partial f}{\partial x} \quad \text{and so on} \ldots$$

With this in mind, consider that for the back-propagation we have:

```
# backprop f = num * invden
dnum = invden # gradient on numerator                          #(8)
dinvden = num                                                  #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                             #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                             #(6)
dxpysqr = (1) * dden                                           #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                     #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                #(4)
dy = (1) * dxpy                                                #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below #(3)
# backprop num = x + sigy
dx += (1) * dnum                                               #(2)
dsigy = (1) * dnum                                             #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                              #(1)
# done! phew
```

where we have labeled `#(n)` as the part of the forward propagation it corresponds to. Also notice for every backprogation step except that corresponding to `#(8)`, chain rule applied meant that it is always multipied by the previous derivative(in red below). For example

$$dden = (-1.0/(den**2)) * dinvden$$

That is to say at `#(1)` is a string 8 partial derivatives here.

Fact 76

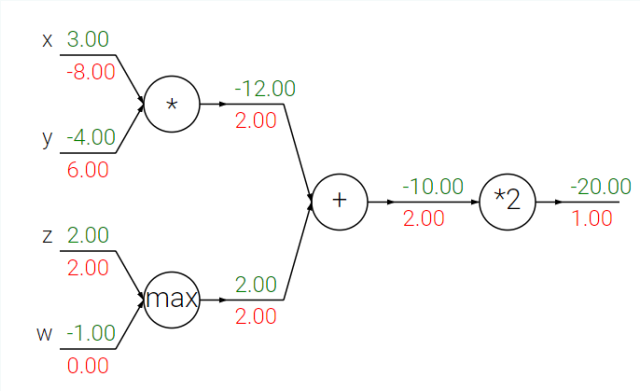
Notice that

- **Cache forward pass variables**: to compute backward pas it is very helpful to have some variables used in the forward pass to avoid recomputation
- **Gradients add up at forks**: the forward expression involves the variables x, y multiple times so when we perform back progration we must be careful to use $+=$ instead of $=$ to accumulate the gradient on these variables

3.4.4 Patterns in backward flow

Example 77

Consider



Looking at the above diagram we see that

- The **add gate** always takes the gradient on its output and distributes it equally to all of its inputs regardless of what their values were during the forward pass for example the "+" gate routed the gradient 2.00(in red) to both of its inputs equally and unchanged. The simple reason for this is that the local gradient for the add operation is simply $+1.0$

$$f = x + y \quad \Rightarrow \quad \frac{\partial f}{\partial x} = 1 \quad \text{and} \quad \frac{\partial f}{\partial y} = 1$$

- The **max gate** routes the gradient. The max gate distributes the gradient(unchanged) to exactly one of its inputs(which has the highest value during the forward pass) since

$$f = \max(x, y) \quad \Rightarrow \quad \frac{\partial f}{\partial x} = \mathbf{1}\{x > y\} \quad \text{and} \quad \frac{\partial f}{\partial y} = \mathbf{1}\{y > x\}$$

- For the **multiply gate**, the local gradient is simply the the local gradients of the input values(except switched)

multiplied by the gradient on its output during the chain rule. For example the gradient on "x" above is $-8.00 = -4.00 \times 2.00$. THis is because

$$f = xy \quad \Rightarrow \quad \frac{\partial f}{\partial x} = y \quad \text{and} \quad \frac{\partial f}{\partial y} = x$$

So far we have only considered the case of single variables. Look at 136 in assigment 1 to see how we can apply our discussion to matrix multiplication operations too.

3.5 Neural Networks 1: setting up architecture

3.5.1 quick intro

Example 78

A single neural network could look like

$$s = Wx$$

A two later neural network could look like

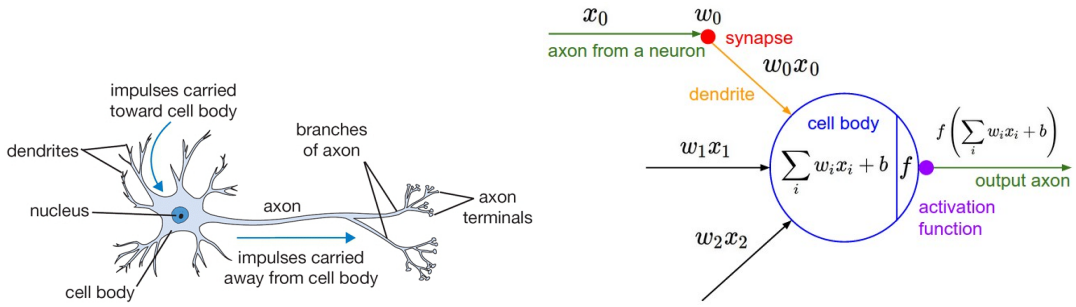
$$s = W_2 \max(0, W_1 x)$$

A 3 layer neural nettwork could look like

$$s = W_3 \max(0, W_2 \max(0, W_1 x))$$

3.5.2 Modelling one neuron

Nueral Networks are primarily inspired by the goal of modelling biological neural systems.



Fact 79 (Sigmoid)

Historically a common choice of activation fuction is the **sigmoid function** σ since it takes a real-valued input(the signal strenght after the sum) and squases it to range between 0 and 1. The reason is that this reflects the neuron's capacity to "like"(activation near one) or "dislike"(activation near zero) certain linear regions of its input space

And example code for this is

```
class Neuron(object):
  # ...
```

```
def forward(self, inputs):
  """ assume inputs and weights are 1-D numpy arrays and bias is a number """
  cell_body_sum = np.sum(inputs * self.weights) + self.bias
  firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
  return firing_rate
```

Definition 80 (ReLU)
Another popular activation function is the **rectified linear unit**(ReLU) which computes the function

$$f(x) = \max(0, x)$$

In other words the activation threshold is simply zero.

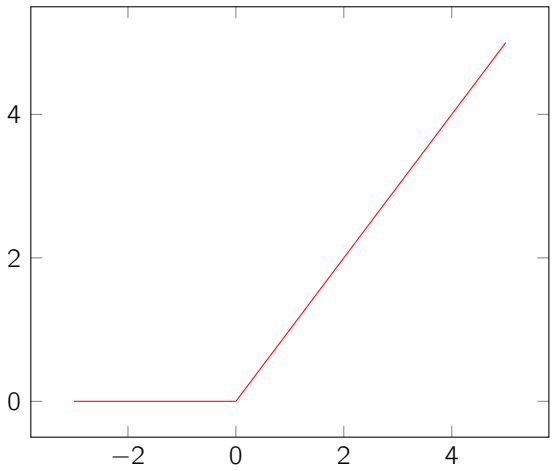
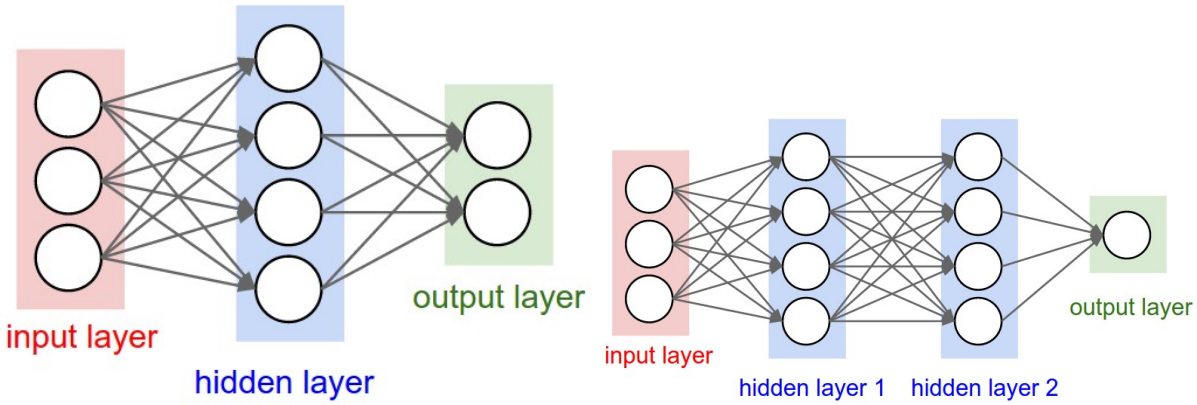


Figure 13: ReLU activation graph

Compare this graph with the sigmoid function!

3.5.3 Neural Network Architectures



Example 81

Considering the examples as depicted above we have

- sizing
  - the first network(left) has $4+2=6$ neurons(not counting the inputs essentially), $[3\times 4]+[4\times 2]=20$ weights and $4+2=6$ biases for a total of 26 learnable paramters
  - the second network(right) has $4+4+1=9$ neurons,$[3\times 4]+[4\times 4]+[4\times 1]=32$ weights and $4+4+1=9$ biases

Remark 82. *For context modern CNNs contain orders of 100 million paramters and made up of approximately 10-20 layers(hence the name "deep learning"). We will explore this more in later lectures*

Example 83

Let us now consider an example feed-forward computation as in 78

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

where `W1,W2,W3,b1,b2,b3` are the learnable parameters for the above code.

3.6 Neural Networks 2: setting up data and loss

3.6.1 data preprocessing

There are 3 common forms of data processing a data matrix X where we will asume that X is of size $[N\times D]$ where N is the number of data and D is their dimensionality

1. **Mean subtraction**
   It involves subtracting the mean across every indiviudal feature in the data and has geometric interpretation of centering the cloud of data around the origin along every dimension. In numpy this is implemented by

   ```
   X-=np.mean(X,axis=0)
   ```

2. **Normalization**
   There are two ways to do this
   (a) one is to divide each dimension by its standard deviation
   (b) another is to normalize each dimesion so that the min and max along the dimension is -1 and 1 respecitvely.

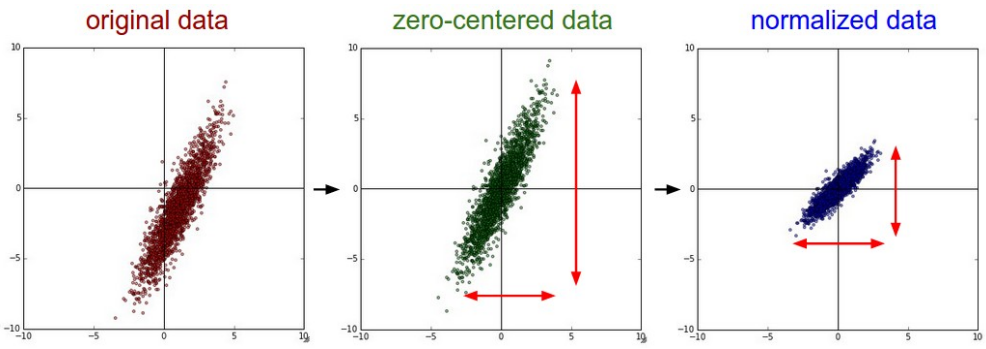


Figure 14: **Left**: 2 dimensional input data. **Middle**:The data is zero centered by substracting the mean in each dimension. **Right**: Each dimension is additionally scaled by its standard deviation

3. **PCA and Whitening**

   This can implemented by

   ```
   # Assume input data matrix X of size [N x D]
   X -= np.mean(X, axis = 0) # zero-center the data (important)
   cov = np.dot(X.T, X) / X.shape[0] # get the data covariance matrix
   ```

   To see why this is true recall that for random variable X,

   $$E[X] = \mu = \sum x_i/n$$

   (for the discrete case in our context). Also recall that $E[X^2] = \sum_i x_i^2/n$. Now recall the formula for covariance

   $$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])]$$

   and in our context(recall multivariate gaussian from CS229) alterantively simply see it as the same as the above(also have N total cases) but each event is in $\mathbb{R}^D$ instead of just $\mathbb{R}$.

   $$E(X - \mu)^T(X - \mu)] = E[X^T X] = \frac{X^T X}{N}$$

   because for the same dsicrete random variable X, and its posible values, $P(X^2 = y_i^2) = P(X = y_i) = \frac{1}{N}$ and $\mu = 0$(zero centered)

...to be continued

3.6.2 weight initialization

Fact 84

Apparently we are not allowed to set all initial weights to zero because then every neuron in the network

3.6.3 batch normalization

3.6.4 regularization

3.7 Neural Networks 3: learning and evaluation

3.7.1 gradient checks

Fact 85

instead of

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

use

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

because it is more precise. To see this, use taylor expansion of $f(x+h)$ and $f(x-h)$ where you will find the error term for the former is $O(h)$ while the latter is $O(h^2)$

To see this recall a taylor expansion given a small h and sufficiently differentiable function f we have

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)$$

similar expansion can be found for $f(x-h)$ where we sub $-h$ above. Notice that for the centered difference, the $hf'(x)$ term cancels so the highest lowest power of h becomes 2.(since small number the higher the power the smaller the number). Therefore the error term is $O(h^2)$ which is smaller than $O(h)$ for the non centered difference.

Remark 86. *although the centered formula is more precise, it is more expensive as we need to evaluate the loss function wice to check every single dimension of the gradient*

3.7.2 sanity checks

3.7.3 babysitting the learning process

loss function

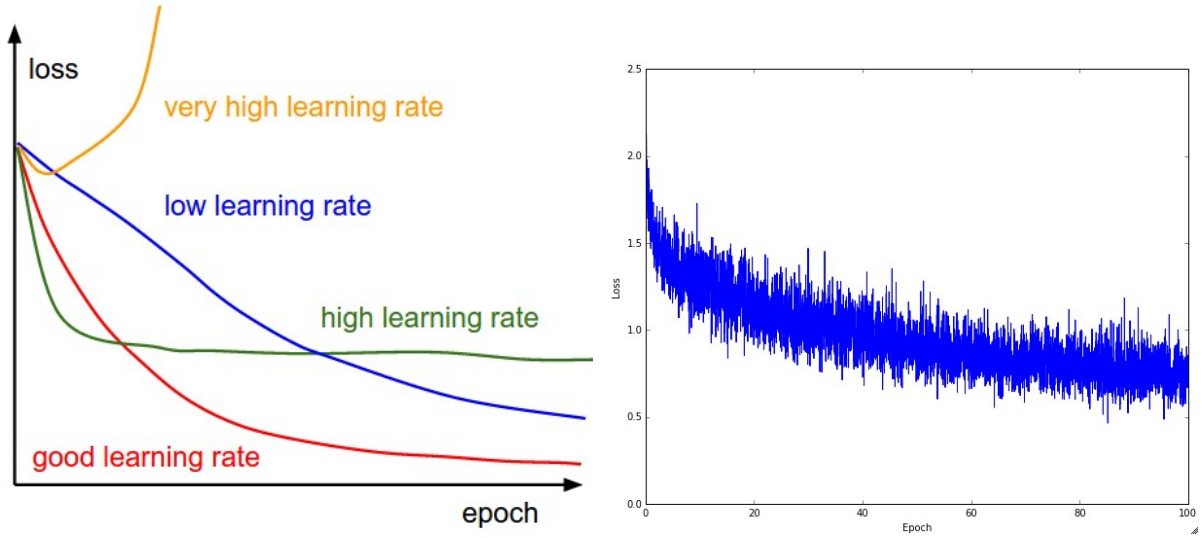


Figure 15: **Left**: A cartoon depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape. **Right**: An example of a typical loss function over time, while training a small network on CIFAR-10 dataset. This loss function looks reasonable (it might indicate a slightly too small learning rate based on its speed of decay, but it's hard to say), and also indicates that the batch size might be a little too low (since the cost is a little too noisy).

Train/Val accuracy

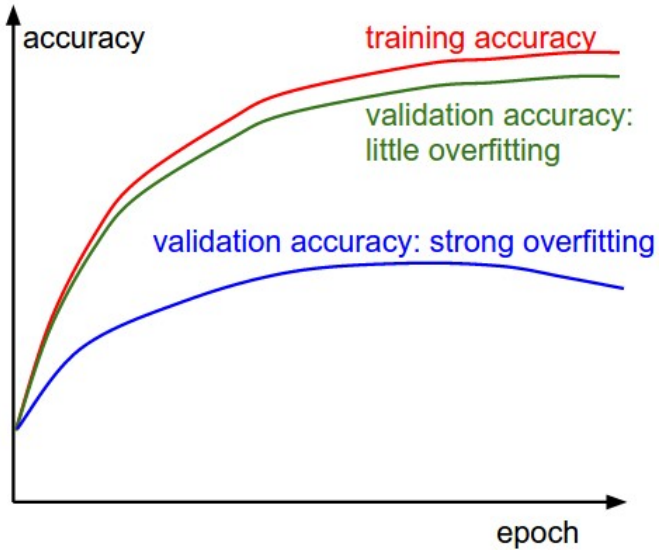


Figure 16: The gap between training and validation accuracy indicates the amount of overfitting

ratio of weights: updates

Example 87

Consider

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

where `arr.ravel()` is the numpy method that is equivalent of `arr.reshape(-1)`

Ideally we roughly want this value to be $\sim e^3$.

- If it's lower that this it is likely the learning rate is too low
- and conversely if higher, the learning rate is too high

3.7.4 paramter updates

Example 88 (vanilla update)

code

```
# Vanilla update
x += - learning_rate * dx
```

Example 89 (momentum update)

code

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

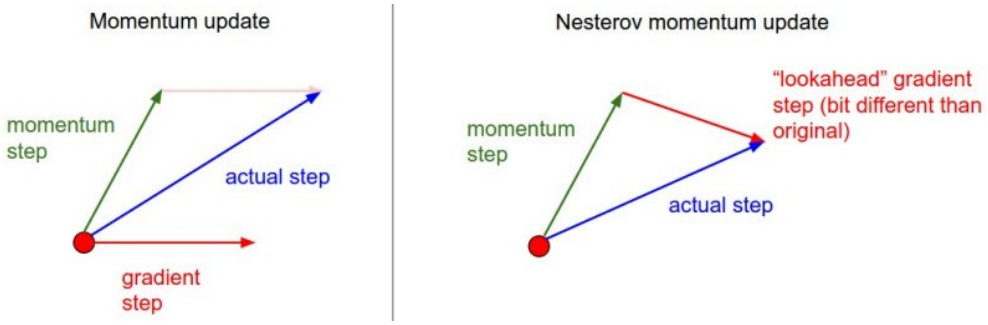
Example 90 (nesterov momentum)

code

```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

which is equivalent to

```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```



3.7.5 hyperparamter optimization

3.7.6 evaluation

4 Assignment 1

4.1 Q1: KNN

Question 91. *What is the most efficient way to calculate the frobrenius L2 norm between matrices `A-B`?(without using the function `np.linalg.norm`)?*

We will compare three approaches to calculate this same value

1. see problem 95
2. see problem 97
3. see problem 100

The point is for you notice the power of broadcasting and elementwise computations which reduces the need to use loops and hence reducing time complexity. Please look at the other questions contained in the code too for a complete understanding

4.2 Q2: SVM

The important questions are

1. Find gradient of (linear)SVM cost function :see problem 107
2. Vectorized version of above: see problem 112
3. Stochastic Gradient Descent(SGD): see problem 120
4. SGD evaluation: see problem 123

as usual please look at the other questions contained in the code too for a complete understanding

4.3 Q3: Softmax

The important questions are

1. Find gradient of (linear)softmax cost function :see problem 126

2. Vectorized version of above: see problem 129

as usual please look at the other questions contained in the code too for a complete understanding

4.4 Q4: Two layer NN

The important questions are

1. affine forward pass :see problem 132
2. affine backward: see problem 134
3. ReLU forward pass :see problem 139
4. ReLU backward: see problem 141
5. Multiclass SVM 144
6. Multiclass softmax 146
7. 2 layer forward pass 151
8. 2 layer backward pass 152

as usual please look at the other questions contained in the code too for a complete understanding. Additionally please read and understand

- solver 157
- optim 153

4.5 Q5: Features

4.6 Appendix 1

K_nearest_neighbours.py

```
from builtins import range
from builtins import object
import numpy as np


class KNearestNeighbor(object):
    """ a kNN classifier with L2 distance """

    def __init__(self):
        pass

    def train(self, X, y):
```

Fact 92 (Train the classifier.)

For k-nearest neighbors this is just memorizing the training data.

Inputs:

- X: A numpy array of shape (num_train, D) containing the training data consisting of num_train samples each of dimension D.
- y: A numpy array of shape (N,) containing the labels, where y[i] is the label for X[i].

```
        self.X_train = X
        self.y_train = y

    def predict(self, X, k=1, num_loops=0):
```

Fact 93

Predict labels for test data using this classifier.

Inputs:

- X: A numpy array of shape (num_test, D) containing test data consisting of num_test samples each of dimension D.
- k: The number of nearest neighbors that vote for the predicted labels.
- num_loops: Determines which implementation to use to compute distances between training points and testing points.

Returns:

- y: A numpy array of shape (num_test,) containing predicted labels for the test data, where y[i] is the predicted label for the test point X[i].

```
        if num_loops == 0:
            dists = self.compute_distances_no_loops(X)
        elif num_loops == 1:
            dists = self.compute_distances_one_loop(X)
        elif num_loops == 2:
            dists = self.compute_distances_two_loops(X)
        else:
            raise ValueError("Invalid value %d for num_loops" % num_loops)

        return self.predict_labels(dists, k=k)

    def compute_distances_two_loops(self, X):
```

Fact 94 (compute_distances_two_loops)

Compute the distance between each test point in X and each training point in $self.X\_train$ using a nested loop over both the training data and the test data.

- Inputs:X, A numpy array of shape (num_test, D) containing test data.
- Returns: dists, A numpy array of shape (num_test, num_train) where dists $[i, j]$ is the Euclidean distance between the ith test point and the jth training point.

```
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in range(num_test):
            for j in range(num_train):
```

Problem 95

Compute the l2 distance between the ith test point and the jth training point, and store the result in dists[i, j]. You should not use a loop over dimension, nor use `np.linalg.norm()`.

```
                # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

                dists[i, j] = np.sqrt(np.sum(np.power(self.X_train[j] - X[i], 2)))

                # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return dists

    def compute_distances_one_loop(self, X):
```

Fact 96 (compute_distances_one_loop)

Compute the distance between each test point in X and each training point in `self.X_train` using a single loop over the test data.

Input / Output: Same as `compute_distances_two_loops`

```
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in range(num_test):
```

Problem 97

Compute the l2 distance between the ith test point and all training points, and store the result in `dists[i, :]`.Do not use `np.linalg.norm()`.

```
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            dists[i] = np.sqrt(np.sum(np.power(self.X_train - X[i], 2), axis=1))
            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Remark 98. *Essentially `self.X_train` is 2D array. So is broadcasts with the 1D array `X[i]` to do a pointwise subtraction.*

```
    #selfXtrain N D
    #X[i]         D
    #           N D
```

Then after that as you know `sum` takes sum of the list of results.

```
        return dists

    def compute_distances_no_loops(self, X):
```

Fact 99 (compute_distances_no_loops)
Compute the distance between each test point in X and each training point in self.X_train using no explicit loops.
Input / Output: Same as compute_distances_two_loops

Problem 100
Compute the l2 distance between all test points and all training points without using any explicit loops, and store the result in dists. You should implement this function using only basic array operations; in particular you should not use functions from scipy,nor use np.linalg.norm().

HINT: Try to formulate the l2 distance using matrix multiplication and two broadcast sums.

```
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # Note: (a - b)^2 = -2ab + a^2 + b^2
        dists = np.sqrt(
          -2 * (X @ self.X_train.T) +
          np.power(X, 2).sum(axis=1, keepdims=True) +
          np.power(self.X_train, 2).sum(axis=1, keepdims=True).T
        )
```

Remark 101. *Note that `@` refers to matrix multiplication in numpy. Recall what `keepdims` does from 23*

Example 102

Consider

```
A = [[1, 2],    B = [[11, 12],
     [3, 4]]         [13, 14]]
A * B = [[1 * 11,   2 * 12],
         [3 * 13,   4 * 14]]
A @ B  =  [[1 * 11 + 2 * 13,   1 * 12 + 2 * 14],
           [3 * 11 + 4 * 13,   3 * 12 + 4 * 14]]
```

```
        # dists = np.sqrt(np.sum(np.power(np.expand_dims(X, axis=1) - self.X_train, 2), axis=2))

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return dists
    def predict_labels(self, dists, k=1):
```

Fact 103

Given a matrix of distances between test points and training points, predict a label for each test point.

Inputs: dists: A numpy array of shape (num_test, num_train) where dists[i, j] gives the distance betwen the ith test point and the jth training point.

Returns: y: A numpy array of shape (num_test,) containing predicted labels for the test data, where y[i] is the predicted label for the test point X[i].

Problem 104

Use the distance matrix to find the k nearest neighbors of the ith testing point, and use self.y_train to find the labels of these neighbors. Store these labels in closest_y.

Hint: Look up the function numpy.argsort.

```
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in range(num_test):
            # A list of length k storing the labels of the k nearest neighbors to
            # the ith test point.
            closest_y = []
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            closest_y = self.y_train[dists[i].argsort()[:k]]

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Problem 105

Now that you have found the labels of the k nearest neighbors, you need to find the most common label in the list `closest_y` of labels. Store this label in `y_pred[i]`. Break ties by choosing the smaller label.

```
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            y_pred[i] = np.argmax(np.bincount(closest_y))

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        return y_pred
```

linear_svm.py

```
from builtins import range
import numpy as np
from random import shuffle


def svm_loss_naive(W, X, y, reg):
```

Fact 106 (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. Note that is is the transpose of 51 so do not them mix up.
- 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

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

    # compute the loss and the gradient
    num_classes = W.shape[1]
    num_train = X.shape[0]
    loss = 0.0
    for i in range(num_train):
        scores = X[i].dot(W)
        correct_class_score = scores[y[i]]
        for j in range(num_classes):
            if j == y[i]:
```

```
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            loss += margin #recall we are using hinge loss so if <=0 the max function returns 0
```

Problem 107

Implement the gradient for the SVM cost function

```
            ############################################################
            #                     START OF CHANGE                      #
            ############################################################
            dW[:, j] += X[i]    # update gradient for incorrect label
            dW[:, y[i]] -= X[i] # update gradient for correct label
            ############################################################
            #                      END OF CHANGE                       #
            ############################################################
```

Remark 108. *Note that $X[i]$ is a column vector in $\mathbb{R}^D$.$dW[:,j]$(corresponds to column j. So such a pointwise addition makes sense(follows broadcasting rules as we are doing)*

X[i] D
dW[:;j] D

This gradient should make sense! Recall 72

```
    # Right now the loss is a sum over all training examples, but we want it
    # to be an average instead so we divide by num_train.
    loss /= num_train

    # Add regularization to the loss.
    loss += reg * np.sum(W * W)
```

Problem 109

Compute the gradient of the loss function and store it dW. Rather that first computing the loss and then computing the derivative, it may be simpler to compute the derivative at the same time that the loss is being computed. As a result you may need to modify some of the code above to compute the gradient.

```
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    dW /= num_train   # scale gradient ovr the number of samples
    dW += 2 * reg * W # append partial derivative of regularization term

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Remark 110. *All these should make sense too! Refer to 3.2.2.*

```
    return loss, dW


def svm_loss_vectorized(W, X, y, reg):
```

Fact 111 (Structured SVM loss function, vectorized implementation)
Inputs and outputs are the same as `svm_loss_naive`

Problem 112
Implement a vectorized version of the structured SVM loss, storing the result in loss.

```
    loss = 0.0
    dW = np.zeros(W.shape)  # initialize the gradient as zero
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N = len(y)      # number of samples
    Y_hat = X @ W   # raw scores matrix

    y_hat_true = Y_hat[range(N), y][:, np.newaxis]     # scores for true labels
```

Remark 113. *If you cant make sense of this recall from 57 what exactly the raw score matrix represents. The answer is that $Y\_hat = f(W, X)$ where $= f(W, x_i)_j$ corresponds to the score for the ith element for the jth class. In hat case you will know that*

$$Y\_hat[range(N), y][:, np.newaxis]$$

Recall that X and W is an $N \times D$ and $D \times C$ array respectively.So if we want Wx_i where x_i is a column vector in $\mathbb{R}^D$ we must have $X@W$ which is a $N \times C$ array. This is consistent with the fact that $[X@W]_{ij} = f(W, x_i)_j$.Now we know that the black segments correspond to an extension of Y_hat by one new dimension(115). So our y_hat_true has the shape $(N, 1)$ □

```
    margins = np.maximum(0, Y_hat - y_hat_true + 1)   # margin for each score
    #Y_hat-    : N C
    #y_hat_true: N 1 = N C
    #+1        :       1 = N C
    margins[range(N),y]=0 #don't consider the correct class in loss
    loss = margins.sum() / N  + reg * np.sum(W**2) # regularized loss
    #W**2 is pointwise power 2
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Remark 114. *Friendly reminder that `range(N)` refers $[0, \ldots, N-1]$. Recall the common for loop python expression `for i in range(N)`. Clearly range returns a list.Also another friendly reminder that `y` is the array of correct class numbers y_i for each x_i. I think by now you should have made sense of the integer array indexing of Y_hat. As for np.maximum refer to 116*

Example 115

`np.newaxis` is a way to increase the dimension an array specifically you define some array x. Then

$$x[np.newaxis, \ldots, :, np.newaxis, \ldots]$$

whereever you put your *np.axis* it will add 1 to that new dimension of your array. For example

```
In [7]: arr = np.arange(4)
In [8]: arr.shape
Out[8]: (4,)

# make it as row vector by inserting an axis along first dimension
In [9]: row_vec = arr[np.newaxis, :]     # arr[None, :]
In [10]: row_vec.shape
Out[10]: (1, 4) #corresponding to [[0,1,2,3]]

# make it as column vector by inserting an axis along second dimension
In [11]: col_vec = arr[:, np.newaxis]     #arr[:, None]
In [12]: col_vec.shape
Out[12]: (4, 1) #corresponding to [[0],[1],[2],[3]]
```

Example 116

np.maximum is basically pointwise max function with proper broadcasting

```
import numpy as np
np.maximum([2, 3, 4], [1, 5, 2]) #returns np.array([2,5,4])
```

Problem 117

Implement a vectorized version of the gradient for the structured SVM loss, storing the result in dW.

Hint: Instead of computing the gradient from scratch, it may be easier to reuse some of the intermediate values that you used to compute the loss.

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

dW = (margins > 0).astype(int)    # initial gradient with respect to Y_hat
#see that this updates all the gradients for non correct labels
dW[range(N), y] -= dW.sum(axis=1) # update gradient to include correct labels
#(which recall (neg) sum of wrong label gradients)
#reminder: axis 1 refers to the classes
dW = X.T @ dW / N + 2 * reg * W   # gradient with respect to W

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Remark 118. *Note that* `margins>0` *essentially creates a boolean matrix with same shape as margins which is true if the element in margin is* > 0 *and false otherwise, Si astype correctly turns them into integers* $1, 0$ *instead.This is essentially the indicator function we covered in 72 Then see that* $X.T$ *has shape* (D, N) *while dW has shape* (N, C) *so the resultant of their matrix product is* (D, C) *as desired*

```
    return loss, dW
```

linearclassifier.py

```
from __future__ import print_function

from builtins import range
from builtins import object
import numpy as np
from ..classifiers.linear_svm import *
from ..classifiers.softmax import *


class LinearClassifier(object):
    def __init__(self):
        self.W = None

    def train(self,X,y,learning_rate=1e-3,reg=1e-5,num_iters=100,batch_size=200,verbose=False):
```

Fact 119 (Train this linear classifier using stochastic gradient descent)

Inputs:

- X: A numpy array of shape (N, D) containing training data; there are N training samples each of dimension D.
- y: A numpy array of shape (N,) containing training labels; y[i]= c means that X[i] has label 0 <= c < C for C classes.
- learning_rate: (float) learning rate for optimization.
- reg: (float) regularization strength.
- num_iters: (integer) number of steps to take when optimizing
- batch_size: (integer) number of training examples to use at each step.
- verbose: (boolean) If true, print progress during optimization.

Outputs:

- A list containing the value of the loss function at each training iteration.

```
        num_train, dim = X.shape #num_train=N,dim=D
        num_classes = (np.max(y) + 1)  # assume y takes values 0...K-1 where K is number of classes
        if self.W is None:
```

```
        # lazily initialize W
        self.W = 0.001 * np.random.randn(dim, num_classes)
        #(D,C)
    # Run stochastic gradient descent to optimize W
    loss_history = []
    for it in range(num_iters):
        X_batch = None
        y_batch = None
```

Problem 120

Sample batch_size elements from the training data and their corresponding labels to use in this round of gradient descent.Store the data in X_batch and their corresponding labels in y_batch; after sampling X_batch should have shape (batch_size, dim) and y_batch should have shape (batch_size,)

Hint: Use np.random.choice to generate indices. Sampling with replacement is faster than sampling without replacement.

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        #samples batchsize elements from np.arrange(num_train)
        indices = np.random.choice(num_train, batch_size)
        X_batch = X[indices]
        y_batch = y[indices]

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # evaluate loss and gradient
        loss, grad = self.loss(X_batch, y_batch, reg)
        loss_history.append(loss)

        # perform parameter update
```

Problem 121

Update the weights using the gradient and the learning rate.

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.W -= learning_rate * grad

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        if verbose and it % 100 == 0:
            print("iteration %d / %d: loss %f" % (it, num_iters, loss))
```

```
        return loss_history

    def predict(self, X):
```

Fact 122

Use the trained weights of this linear classifier to predict labels for data points.

- Inputs: X: A numpy array of shape (N, D) containing training data; there are N training samples each of dimension D.
- Returns: y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional array of length N, and each element is an integer giving the predicted class.

Problem 123

Implement this method. Store the predicted labels in y_pred.

```
        y_pred = np.zeros(X.shape[0])
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        y_pred = np.argmax(X @ self.W, axis=1) #indices in this case the
        #class with greatest score in the score function matrix

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return y_pred

    def loss(self, X_batch, y_batch, reg): pass
```

Fact 124 (Compute the loss function and its derivative)

Subclasses will override this.

Inputs

- X_batch: A numpy array of shape (N, D) containing a minibatch of N data points; each point has dimension D.
- y_batch: A numpy array of shape (N,) containing labels for the minibatch.
- reg: (float) regularization strength.

Returns: A tuple containing:

- loss as a single float
- gradientt with respect to self.W; an array of the same shape as W

```
class LinearSVM(LinearClassifier):
    """ A subclass that uses the Multiclass SVM loss function """
```

```
    def loss(self, X_batch, y_batch, reg):
        return svm_loss_vectorized(self.W, X_batch, y_batch, reg)


class Softmax(LinearClassifier):
    """ A subclass that uses the Softmax + Cross-entropy loss function """

    def loss(self, X_batch, y_batch, reg):
        return softmax_loss_vectorized(self.W, X_batch, y_batch, reg)
```

softmax.py

```
from builtins import range
import numpy as np
from random import shuffle


def softmax_loss_naive(W, X, y, reg):
```

Fact 125 (Softmax 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

Problem 126

Compute the softmax loss and its gradient using explicit loops. Store the loss in loss and the gradient in dW. If you are not careful here, it is easy to run into numeric instability. Don't forget the regularization!

```
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    N = X.shape[0] # num samples

    for i in range(N):
        y_hat = X[i] @ W                    # raw scores vector for x_i
        #Shape of y_hat: (D)@(D,C)=(C)
        #note that y_hat[j] gives the score for category j
        y_exp = np.exp(y_hat - y_hat.max()) # numerically stable exponent vector
        #np.exp is a pointwise exponential. So it's still shape (C)
```

Remark 127. *recall the use of* `y_hat.max()` *from 67. Also the use of outer product should make sense if you recall how we found the gradient for the cross entropy loss in 68*

```
        softmax = y_exp / y_exp.sum()       # pure softmax for each score
        loss -= np.log(softmax[y[i]])       # append cross-entropy
        softmax[y[i]] -= 1                  # update for gradient
        dW += np.outer(X[i], softmax)       # gradient

    loss = loss / N + reg * np.sum(W**2)    # average loss and regularize
    dW = dW / N + 2 * reg * W               # finish calculating gradient

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss, dW


def softmax_loss_vectorized(W, X, y, reg):
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)
```

Fact 128 (Softmax loss function, vectorized version)
Inputs and outputs are the same as softmax_loss_naive.

Problem 129
Compute the softmax loss and its gradient using explicit loops. Store the loss in loss and the gradient in dW. If you are not careful here, it is easy to run into numeric instability. Don't forget the regularization!

```
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N = X.shape[0] # number of samples
    Y_hat = X @ W  # raw scores matrix

    P = np.exp(Y_hat - Y_hat.max())      # numerically stable exponents
```

```
    P /= P.sum(axis=1, keepdims=True)     # row-wise probabilities (softmax)

    loss = -np.log(P[range(N), y]).sum() # sum cross entropies as loss
    loss = loss / N + reg * np.sum(W**2) # average loss and regularize

    P[range(N), y] -= 1                  # update P for gradient
    dW = X.T @ P / N + 2 * reg * W       # calculate gradient

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss, dW
```

Remark 130. *This should make sense recall 68 that the gradient for non correct classs is simply the softmax and for correct it is softmax-1.*

layers.py

```
from builtins import range
import numpy as np



def affine_forward(x, w, b):
```

Fact 131 (Computes the forward pass for an affine (fully-connected) layer)

The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N examples, where each example x[i] has shape (d_1, ..., d_k). We will reshape each input into a vector of dimension D = d_1 * ... * d_k, and then transform it to an output vector of dimension M.

Inputs:

- x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
- w: A numpy array of weights, of shape (D, M)
- b: A numpy array of biases, of shape (M,)

Returns a tuple of:

- out: output, of shape (N, M)
- cache: (x, w, b)

Problem 132

Implement the affine forward pass. Store the result in out. You will need to reshape the input into rows.

```
    out = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    x_reshaped = x.reshape(x.shape[0], -1)
    out = x_reshaped @ w + b

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                             END OF YOUR CODE                               #
    ##############################################################################
    cache = (x, w, b)
    return out, cache


def affine_backward(dout, cache):
```

Fact 133 (Computes the backward pass for an affine layer)

Inputs:

- 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)
- b: Biases, of shape (M,)

Returns a tuple of:

- dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
- dw: Gradient with respect to w, of shape (D, M)
- db: Gradient with respect to b, of shape (M,)

Problem 134

Implement the affine backward pass

```
    x, w, b = cache
    dx, dw, db = None, None, None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x_reshaped = x.reshape(x.shape[0], -1)
    dx = (dout @ w.T).reshape(x.shape[0], *x.shape[1:])
    dw = x_reshaped.T @ dout
    db = dout.sum(axis=0)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
```

```
#                             END OF YOUR CODE                             #
###############################################################################
return dx, dw, db
```

Remark 135. *Note that* $dx = (dout@w.T).reshape(x.shape[0], *x.shape[1:])$ *reshape was required because in the forward pass we rehsaped from* $(N, d\_1, ..., d\_k) \quad \to \quad (N, M)$ *so we have to restore it back to* $(N, d\_1, ..., d\_k)$

Question 136 (Matrix-matrix multiply gradient)**.** *How did we derive* `dx,dw,db` *above?*

Hint: you might want to recap the appendix above on matrix derivatives if you have no idea where to start.

Solution. consider that from the forward pass we have

$$\text{out}_{ij} = x_{\text{reshaped}\,ik} w_{kj} + b_j = [x_{\text{reshaped}} @ W]_{ij} + b_j$$

reminder: note the use of summation convention over k here.

- for dx:

  Then by chain rule we know that

$$\begin{aligned}\frac{\partial L}{\partial x_{\text{reshaped}\,ik}} &= \frac{\partial L}{\partial \text{out}_{ij}} \cdot \frac{\partial \text{out}_{ij}}{\partial x_{\text{reshaped}\,ik}} \\ &= \text{dout}_{ij} w_{kj} \\ &= \text{dout} @ w^T\end{aligned}$$

- for dw:

  Then by chain rule we know that

$$\begin{aligned}\frac{\partial L}{\partial w_{kj}} &= \frac{\partial L}{\partial \text{out}_{ij}} \cdot \frac{\partial \text{out}_{ij}}{\partial w_{kj}} \\ &= \text{dout}_{ij} x_{\text{reshaped}\,ik} \\ &= x^T_{\text{reshaped}} @ \text{dout}\end{aligned}$$

- for dx:

  Then by chain rule we know that

$$\begin{aligned}\frac{\partial L}{\partial b_j} &= \frac{\partial L}{\partial \text{out}_{ij}} \cdot \frac{\partial \text{out}_{ij}}{\partial b_j} \\ &= 1 \cdot \sum_i \text{dout}_{ij} \quad \text{(removing summation convention here)}\end{aligned}$$

□

Remark 137. *Note that understanding this is quite important as it generalizes finding the back propagation for matrix multiplication operations*

Fact 138 (Computes the forward pass for a layer of rectified linear units (ReLUs))

Input:

- x: Inputs, of any shape

Returns a tuple of:

- out: Output, of the same shape as x
- cache: x

Problem 139

Implement the ReLU forward pass

```
def relu_forward(x):
    out = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    out = np.maximum(0, x)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache


def relu_backward(dout, cache):
```

Fact 140

Computes the backward pass for a layer of rectified linear units (ReLUs).

Input:

- dout: Upstream derivatives, of any shape
- cache: Input x, of same shape as dout

Returns:

- dx: Gradient with respect to x

Problem 141

Implement the ReLU backward pass.

```
    dx, x = None, cache
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    dx = dout * (x > 0)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx
```

Remark 142. *This solution should be fairly simple. Note that x>0 returns 1(true) and 0(false) depending on whether* $x > 0$. *This should make sense if you recall 77*

Fact 143 (Computes the loss and gradient using for multiclass SVM classification)

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

Returns a tuple of:

- loss: Scalar giving the loss
- dx: Gradient of the loss with respect to x

Problem 144

Implement loss and gradient for multiclass SVM classification. This will be similar to the svm loss vectorized implementation in cs231n/classifiers/linear_svm.py(see 4.6).

```
def svm_loss(x, y):
    loss, dx = None, None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N = len(y)                                  # number of samples
    x_true = x[range(N), y][:, None]            # scores for true labels
    margins = np.maximum(0, x - x_true + 1)     # margin for each score
    loss = margins.sum() / N - 1
    dx = (margins > 0).astype(float) / N
    dx[range(N), y] -= dx.sum(axis=1)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return loss, dx
```

```
def softmax_loss(x, y):
```

Fact 145 (Computes the loss and gradient for softmax classification)

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

Returns a tuple of:

- loss: Scalar giving the loss
- dx: Gradient of the loss with respect to x

Problem 146

Implement the loss and gradient for multiclass softmax classification. This will be similar to the softmax loss vectorized implementation in cs231n/classifiers/softmax.py.

```
    loss, dx = None, None

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N = len(y) # number of samples

    P = np.exp(x - x.max(axis=1, keepdims=True)) # numerically stable exponents
    P /= P.sum(axis=1, keepdims=True)            # row-wise probabilities (softmax)

    loss = -np.log(P[range(N), y]).sum() / N # sum cross entropies as loss

    P[range(N), y] -= 1
    dx = P / N

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                             END OF YOUR CODE                               #
    ##############################################################################
    return loss, dx
```

fc_net.py

```
from builtins import range
from builtins import object
import numpy as np
```

```
from ..layers import *
from ..layer_utils import *


class TwoLayerNet(object):
```

Fact 147

A two-layer fully-connected neural network with ReLU nonlinearity and softmax loss that uses a modular layer design. We assume an input dimension of D, a hidden dimension of H, and perform classification over C classes.

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 object that is responsible for running optimization.

The learnable parameters of the model are stored in the dictionary self.params that maps parameter names to numpy arrays.

```
    def __init__(
        self,
        input_dim=3 * 32 * 32,
        hidden_dim=100,
        num_classes=10,
        weight_scale=1e-3,
        reg=0.0,
    ):
```

Fact 148

Initialize a new 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
- weight_scale: Scalar giving the standard deviation for random
- initialization of the weights.
- reg: Scalar giving L2 regularization strength.

Problem 149

Initialize the weights and biases of the two-layer net. Weights should be initialized from a Gaussian centered at 0.0 with standard deviation equal to weight_scale, and biases should be initialized to zero. All weights and biases should be stored in the dictionary self.params, with first layer weights and biases using the keys 'W1' and 'b1' and second layer weights and biases using the keys 'W2' and 'b2'.

```
        self.params = {}
        self.reg = reg
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.params = {
          'W1': np.random.randn(input_dim, hidden_dim) * weight_scale,
          'b1': np.zeros(hidden_dim),
          'W2': np.random.randn(hidden_dim, num_classes) * weight_scale,
          'b2': np.zeros(num_classes)
        }

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

    def loss(self, X, y=None):
```

Fact 150 (Compute loss and gradient for a minibatch of data)

.

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].

Returns: 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.

Problem 151

Implement the forward pass for the two-layer net, computing the class scores for X and storing them in the scores variable.

```
scores = None
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

W1, b1, W2, b2 = self.params.values()

out1, cache1 = affine_forward(X, W1, b1)
out2, cache2 = relu_forward(out1)
scores, cache3 = affine_forward(out2, W2, b2)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
############################################################################
#                             END OF YOUR CODE                             #
############################################################################

# If y is None then we are in test mode so just return scores
if y is None:
    return scores

loss, grads = 0, {}
```

Problem 152

Implement the backward pass for the two-layer net. Store the loss in the loss variable and gradients in the grads dictionary. Compute data loss using softmax, and make sure that grads[k] holds the gradients for self.params[k]. Don't forget to add L2 regularization!

NOTE: To ensure that your implementation matches ours and you pass the automated tests, make sure that your L2 regularization includes a factor of 0.5 to simplify the expression for the gradient.

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

loss, dloss = softmax_loss(scores, y)
loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2))

dout3, dW2, db2 = affine_backward(dloss, cache3)
dout2 = relu_backward(dout3, cache2)
dout1, dW1, db1 = affine_backward(dout2, cache1)

dW1 += self.reg * W1
```

```
        dW2 += self.reg * W2

        grads = {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2}

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        return loss, grads
```

optim.py

Fact 153 (optim)

This file implements various first-order update rules that are commonly used for training neural networks. Each update rule accepts current weights and the gradient of the loss with respect to those weights and produces the next set of weights. Each update rule has the same interface:

def update(w, dw, config=None):

Inputs:

- w: A numpy array giving the current weights.
- dw: A numpy array of the same shape as w giving the gradient of the loss with respect to w.
- config: A dictionary containing hyperparameter values such as learning rate, momentum, etc. If the update rule requires caching values over many iterations, then config will also hold these cached values.

Returns:

- next_w: The next point after the update.
- config: The config dictionary to be passed to the next iteration of the update rule.

NOTE: For most update rules, the default learning rate will probably not perform well; however the default values of the other hyperparameters should work well for a variety of different problems.

For efficiency, update rules may perform in-place updates, mutating w and setting next_w equal to w.

```
import numpy as np

def sgd(w, dw, config=None):
```

Fact 154

Performs vanilla stochastic gradient descent. config format - learning_rate: Scalar learning rate.

Problem 155

Implement the vanilla stochastic gradient descent update formula.

```
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    w -= config['learning_rate'] * dw

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ################################################################################
    #                             END OF YOUR CODE                                 #
    ################################################################################
    return w, config
```

Remark 156 (dict.setefault(a,b)). *the python dictionary method* `dict.setdefault(a,b)` *returns the value of* `dict[a]` *if they key* `a` *exists and if not adds the key value pair a:b into the dictionary. This is analagous to* `dict.pop(a,b)` *(see 162) which is the deleting version while this is the inserting version*

solver.py

```
from __future__ import print_function, division

from builtins import range
from builtins import object
import os
import pickle as pickle

import numpy as np

from cs231n import optim


class Solver(object):
```

Fact 157 (solver)

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

Example 158

Example usage might look something like this:

```
data = {
  'X_train': # training data
  'y_train': # training labels
  'X_val': # validation data
  'y_val': # 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()
```

Fact 159

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.

Fact 160 (Construct a new Solver instance)

Required arguments:

- model: A model object conforming to the API described above
- data: A dictionary of training and validation data containing: 'X_train': Array, shape (N_train, d_1, ..., d_k) of training images 'X_val': Array, shape (N_val, d_1, ..., d_k) of validation images 'y_train': Array, shape (N_train,) of labels for training images 'y_val': Array, shape (N_val,) of labels for validation images

Optional arguments:

- - update_rule: A string giving the name of an update rule in optim.py. Default is 'sgd'.
- optim_config: A dictionary containing hyperparameters that will be passed to the chosen update rule. Each update rule requires different hyperparameters (see optim.py) but all update rules require a 'learning_rate' parameter so that should always be present.
- lr_decay: A scalar for learning rate decay; after each epoch the learning rate is multiplied by this value.
- batch_size: Size of minibatches used to compute loss and gradient during training.
- num_epochs: The number of epochs to run for during training.
- print_every: Integer; training losses will be printed every print_every iterations.
- verbose: Boolean; if set to false then no output will be printed during training.
- num_train_samples: Number of training samples used to check training accuracy; default is 1000; set to None to use entire training set.
- num_val_samples: Number of validation samples to use to check val accuracy; default is None, which uses the entire validation set.
- checkpoint_name: If not None, then save model checkpoints here every epoch.

Example 161

Note that in python there are special arguments `*args,**kwargs` which allows you to pass in a variable length list and dictionary respectively as your arguments

```
def myFun(*args, **kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)
# Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('geeks', 'for', 'geeks', first="Geeks", mid="for", last="Geeks")
```

This gives the output

```
args: ('geeks', 'for', 'geeks')
kwargs: {'first': 'Geeks', 'mid': 'for', 'last': 'Geeks'}
```

We will use these below

Remark 162. *Also recall that* `dict.pop(a,b)` *where b is the value returned if key a does not exist. if not this returns the key value, dict[a] corresponding to key value pair (a,dict[b]) that has been removed from dict by the pop function*

```
    def __init__(self, model, data, **kwargs):
        self.model = model
        self.X_train = data["X_train"]
        self.y_train = data["y_train"]
        self.X_val = data["X_val"]
        self.y_val = data["y_val"]

        # Unpack keyword arguments
        self.update_rule = kwargs.pop("update_rule", "sgd")
        self.optim_config = kwargs.pop("optim_config", {})
        self.lr_decay = kwargs.pop("lr_decay", 1.0)
        self.batch_size = kwargs.pop("batch_size", 100)
        self.num_epochs = kwargs.pop("num_epochs", 10)
        self.num_train_samples = kwargs.pop("num_train_samples", 1000)
        self.num_val_samples = kwargs.pop("num_val_samples", None)

        self.checkpoint_name = kwargs.pop("checkpoint_name", None)
        self.print_every = kwargs.pop("print_every", 10)
        self.verbose = kwargs.pop("verbose", True)

        # Throw an error if there are extra keyword arguments
        if len(kwargs) > 0:
            extra = ", ".join('"%s"' % k for k in list(kwargs.keys()))
            raise ValueError("Unrecognized arguments %s" % extra)

        # Make sure the update rule exists, then replace the string
        # name with the actual function
        if not hasattr(optim, self.update_rule):
            raise ValueError('Invalid update_rule "%s"' % self.update_rule)
        self.update_rule = getattr(optim, self.update_rule)

        self._reset()

    def _reset(self):
        """
        Set up some book-keeping variables for optimization. Don't call this
        manually.
        """
        # Set up some variables for book-keeping
        self.epoch = 0
        self.best_val_acc = 0
```

```
        self.best_params = {}
        self.loss_history = []
        self.train_acc_history = []
        self.val_acc_history = []

        # Make a deep copy of the optim_config for each parameter
        self.optim_configs = {}
        for p in self.model.params:
            d = {k: v for k, v in self.optim_config.items()}
            self.optim_configs[p] = d
```

Remark 163. *that is `optim_configs` is a dictionary of dictionaries whose key values are the parameters Wi, Bi*

```
    def _step(self):
        """
        Make a single gradient update. This is called by train() and should not
        be called manually.
        """
        # Make a minibatch of training data
        num_train = self.X_train.shape[0]
        batch_mask = np.random.choice(num_train, self.batch_size)
        X_batch = self.X_train[batch_mask]
        y_batch = self.y_train[batch_mask]

        # Compute loss and gradient
        loss, grads = self.model.loss(X_batch, y_batch)
        self.loss_history.append(loss)

        # Perform a parameter update
        for p, w in self.model.params.items():
            dw = grads[p]
            config = self.optim_configs[p]
            next_w, next_config = self.update_rule(w, dw, config)
            self.model.params[p] = next_w
            self.optim_configs[p] = next_config

    def _save_checkpoint(self):
        if self.checkpoint_name is None:
            return
        checkpoint = {
            "model": self.model,
            "update_rule": self.update_rule,
            "lr_decay": self.lr_decay,
            "optim_config": self.optim_config,
            "batch_size": self.batch_size,
```

```
            "num_train_samples": self.num_train_samples,
            "num_val_samples": self.num_val_samples,
            "epoch": self.epoch,
            "loss_history": self.loss_history,
            "train_acc_history": self.train_acc_history,
            "val_acc_history": self.val_acc_history,
        }
        filename = "%s_epoch_%d.pkl" % (self.checkpoint_name, self.epoch)
        if self.verbose:
            print('Saving checkpoint to "%s"' % filename)
        with open(filename, "wb") as f:
            pickle.dump(checkpoint, f)

    def check_accuracy(self, X, y, num_samples=None, batch_size=100):
```

Fact 164 (Check accuracy of the model on the provided data)

Inputs:

- X: Array of data, of shape (N, d_1, ..., d_k)
- y: Array of labels, of shape (N,)
- num_samples: If not None, subsample the data and only test the model on num_samples datapoints.
- batch_size: Split X and y into batches of this size to avoid using too much memory.

Returns:

- acc: Scalar giving the fraction of instances that were correctly classified by the model.

```
        # Maybe subsample the data
        N = X.shape[0]
        if num_samples is not None and N > num_samples:
            mask = np.random.choice(N, num_samples)
            N = num_samples
            X = X[mask]
            y = y[mask]

        # Compute predictions in batches
        num_batches = N // batch_size
        if N % batch_size != 0:
            num_batches += 1
        y_pred = []
        for i in range(num_batches):
            start = i * batch_size
            end = (i + 1) * batch_size
```

```
            scores = self.model.loss(X[start:end])
            y_pred.append(np.argmax(scores, axis=1))
            #i.e highest category score index for each Xi
        y_pred = np.hstack(y_pred)
        acc = np.mean(y_pred == y)
        #mean of a binary numpy array will be 0<=acc<=1
        return acc
```

Remark 165. *for "`scores = self.model.loss(X[start:end])`" recall 150 that loss will return loss,grad via backprop if y is specified. Else it will just give the scores(i.e stopping at forward pass)*

```
    def train(self):
        """
        Run optimization to train the model.
        """
        num_train = self.X_train.shape[0]
        iterations_per_epoch = max(num_train // self.batch_size, 1)
        num_iterations = self.num_epochs * iterations_per_epoch

        for t in range(num_iterations):
            self._step()

            # Maybe print training loss
            if self.verbose and t % self.print_every == 0:
                print(
                    "(Iteration %d / %d) loss: %f"
                    % (t + 1, num_iterations, self.loss_history[-1])
                )

            # At the end of every epoch, increment the epoch counter and decay
            # the learning rate.
            epoch_end = (t + 1) % iterations_per_epoch == 0
            if epoch_end:
                self.epoch += 1
                for k in self.optim_configs:
                    self.optim_configs[k]["learning_rate"] *= self.lr_decay

            # Check train and val accuracy on the first iteration, the last
            # iteration, and at the end of each epoch.
            first_it = t == 0
            last_it = t == num_iterations - 1
            if first_it or last_it or epoch_end:
                train_acc = self.check_accuracy(
                    self.X_train, self.y_train, num_samples=self.num_train_samples
                )
```

```
            val_acc = self.check_accuracy(
                self.X_val, self.y_val, num_samples=self.num_val_samples
            )
            self.train_acc_history.append(train_acc)
            self.val_acc_history.append(val_acc)
            self._save_checkpoint()

            if self.verbose:
                print(
                    "(Epoch %d / %d) train acc: %f; val_acc: %f"
                    % (self.epoch, self.num_epochs, train_acc, val_acc)
                )

            # Keep track of the best model
            if val_acc > self.best_val_acc:
                self.best_val_acc = val_acc
                self.best_params = {}
                for k, v in self.model.params.items():
                    self.best_params[k] = v.copy()

    # At the end of training swap the best params into the model
    self.model.params = self.best_params
```

5 Assigment 2

5.1 Q1: Multi Layer Connected NN

5.2 Q2: Batch Normalization

5.3 Q3: Dropout

5.4 Q4: CNNs

5.5 Q5: PyTorch

5.6 Appendix 2

fc_net(2)

```
from builtins import range
from builtins import object
import numpy as np

from ..layers import *
```

```
from ..layer_utils import *


class FullyConnectedNet(object):
```

Fact 166 (Class for a multi-layer fully connected neural network)

Network contains an arbitrary number of hidden layers, ReLU nonlinearities, and a softmax loss function. This will also implement dropout and batch/layer normalization as options. For a network with L layers, the architecture will be

affine - [batch/layer norm] - relu - [dropout] x (L - 1) - affine - softmax

where batch/layer normalization and dropout are optional and the ... block is repeated L - 1 times.

Learnable parameters are stored in the self.params dictionary and will be learned using the Solver class.

```
    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,):
```

Fact 167 (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.

Problem 168

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.

```
        self.normalization = normalization
        self.use_dropout = dropout_keep_ratio != 1
        self.reg = reg
        self.num_layers = 1 + len(hidden_dims)
        self.dtype = dtype
        self.params = {}


        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        for l, (i, j) in enumerate(zip([input_dim, *hidden_dims], [*hidden_dims, num_classes])):
            self.params[f'W{l+1}'] = np.random.randn(i, j) * weight_scale
            self.params[f'b{l+1}'] = np.zeros(j)

            if self.normalization and l < self.num_layers-1:
                self.params[f'gamma{l+1}'] = np.ones(j)
                self.params[f'beta{l+1}'] = np.zeros(j)

        # del self.params[f'gamma{l+1}'], self.params[f'beta{l+1}'] # no batchnorm after last FC

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################
```

Remark 169. *the lines* `f'Wl+1'` *are known as* **f-strings** *and are used to format strings. For example in our case if* $l = 1$ *then* `f'Wl+1'=2` *and if* $l = 2$ *then* `f'Wl+1'=3` *and so on. Essentially we naming the paramters* $b1, W1$ *corresponding to layer 1,* $b2, W2$ *to layer 2 and so on.*

Remark 170. *to understand the line with* `enumerate,zip,*` *in it consider the examples below. Then you will see that essentially supposing we have 3 hidden layers then we are doing:*

l	i	j
0	$a \rightarrow$	h_1
1	$h_1 \rightarrow$	h_2
2	$h_2 \rightarrow$	h_3
3	$h_3 \rightarrow$	b

where the leftmost column represents l, the middle the i's and the right the j's. Also these numbers represent

$$a \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow b$$

the size of each layer and the traversal order like so. Therefore say for example

`self.params[f'Wl+1'] = np.random.randn(i, j) * weight_scale`

element in row i_n *column* j_n *of the 2D matrix of size* $i \times j$ `self.params[f'Wl+1']` *simply represents simply a particular*

pairwise weight between layer l and the adjacent layer

Example 171

enumerate does

```
my_list = ['a', 'b', 'c']
for index, value in enumerate(my_list):
    print(index, value)
```

where the output is

```
0 a
1 b
2 c
```

Example 172

zip pairs up items in the list to their lowest common length(that is to the minimum of the length of the 2 lists)

```
list1 = [1, 2, 3, 4, 5, 6]
list2 = ['a', 'b', 'c']
for pair in zip(list1, list2):
    print(pair)
```

where the output is

```
(1, 'a')
(2, 'b')
(3, 'c')
```

Example 173

The * operator unpacks a list into their elements and is often used in the following

```
hidden_dims = [64, 128, 256]
print([input_dim, *hidden_dims])
```

where the output is

```
[input_dim, 64, 128, 256]
```

```
        # When using dropout we need to pass a dropout_param dictionary to each
        # dropout layer so that the layer knows the dropout probability and the mode
        # (train / test). You can pass the same dropout_param to each dropout layer.
        self.dropout_param = {}
        if self.use_dropout:
            self.dropout_param = {"mode": "train", "p": dropout_keep_ratio}
            if seed is not None:
```

```
            self.dropout_param["seed"] = seed

        # With batch normalization we need to keep track of running means and
        # variances, so we need to pass a special bn_param object to each batch
        # normalization layer. You should pass self.bn_params[0] to the forward pass
        # of the first batch normalization layer, self.bn_params[1] to the forward
        # pass of the second batch normalization layer, etc.
        self.bn_params = []
        if self.normalization == "batchnorm":
            self.bn_params = [{"mode": "train"} for i in range(self.num_layers - 1)]
            #this creates a list of dictionaries  of length num_layers-1
            #eg.[{"mode": "train"},{"mode": "train"},...]
        if self.normalization == "layernorm":
            self.bn_params = [{} for i in range(self.num_layers - 1)]

        # Cast all parameters to the correct datatype.
        for k, v in self.params.items():
            self.params[k] = v.astype(dtype)

    def loss(self, X, y=None):
```

Fact 174 (Compute loss and gradient for the fully connected net)

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].

Returns: 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.

Problem 175

Implement the forward pass for the fully connected net, computing the class scores for X and storing them in the scores variable. When using dropout, you'll need to pass self.dropout_param to each dropout forward pass. When using batch normalization, you'll need to pass self.bn_params[0] to the forward pass for the first batch normalization layer, pass self.bn_params[1] to the forward pass for the second batch normalization layer, etc

```
        X = X.astype(self.dtype)
        mode = "test" if y is None else "train"
```

```
        # Set train/test mode for batchnorm params and dropout param since they
        # behave differently during training and testing.
        if self.use_dropout:
            self.dropout_param["mode"] = mode
        if self.normalization == "batchnorm":
            for bn_param in self.bn_params:
                bn_param["mode"] = mode
        scores = None
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        cache = {}

        for l in range(self.num_layers):
            keys = [f'W{l+1}', f'b{l+1}', f'gamma{l+1}', f'beta{l+1}']   # list of params
            w, b, gamma, beta = (self.params.get(k, None) for k in keys) # get param vals

            bn = self.bn_params[l] if gamma is not None else None  # bn params if exist
            do = self.dropout_param if self.use_dropout else None  # do params if exist

            X, cache[l] = generic_forward(X, w, b, gamma, beta, bn, do, l==self.num_layers-1)
            # generic forward pass
        scores = X

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        # If test mode return early.
        if mode == "test":
            return scores

        loss, grads = 0.0, {}
```

Problem 176

Implement the backward pass for the fully connected net. Store the loss in the loss variable and gradients in the grads dictionary. Compute data loss using softmax, and make sure that grads[k] holds the gradients for self.params[k]. Don't forget to add L2 regularization! When using batch/layer normalization, you don't need to regularize the scale and shift parameters.

NOTE: To ensure that your implementation matches ours and you pass the automated tests, make sure that your L2 regularization includes a factor of 0.5 to simplify the expression for the gradient.

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        loss, dout = softmax_loss(scores, y)
        loss += 0.5 * self.reg * np.sum([np.sum(W**2) for k, W in self.params.items() if 'W' in k])

        for l in reversed(range(self.num_layers)):
            dout, dW, db, dgamma, dbeta = generic_backward(dout, cache[l])

            grads[f'W{l+1}'] = dW + self.reg * self.params[f'W{l+1}']
            grads[f'b{l+1}'] = db

            if dgamma is not None and l < self.num_layers-1:
                grads[f'gamma{l+1}'] = dgamma
                grads[f'beta{l+1}'] = dbeta

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        return loss, grads
```

optim.py(2)

```
def sgd_momentum(w, dw, config=None):
```

Fact 177 (Performs stochastic gradient descent with momentum)

config format:

- learning_rate: Scalar learning rate.
- momentum: Scalar between 0 and 1 giving the momentum value. Setting momentum = 0 reduces to sgd.
- velocity: A numpy array of the same shape as w and dw used to store a moving average of the gradients.

Problem 178

Implement the momentum update formula. Store the updated value in the next_w variable. You should also use and update the velocity v.

```
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("momentum", 0.9)
    v = config.get("velocity", np.zeros_like(w))
```

```
    next_w = None


    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    v = config['momentum'] * v - config['learning_rate'] * dw # update velocity
    next_w = w + v                                            # update position

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ################################################################################
    #                               END OF YOUR CODE                               #
    ################################################################################
    config["velocity"] = v

    return next_w, config


def rmsprop(w, dw, config=None):
```

Fact 179

Uses the RMSProp update rule, which uses a moving average of squared gradient values to set adaptive per-parameter learning rates.

config format:

- learning_rate: Scalar learning rate.
- decay_rate: Scalar between 0 and 1 giving the decay rate for the squared gradient cache.
- epsilon: Small scalar used for smoothing to avoid dividing by zero.
- cache: Moving average of second moments of gradients.

Problem 180

Implement the RMSprop update formula, storing the next value of w in the next_w variable. Don't forget to update cache value stored inconfig['cache'].

```
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("decay_rate", 0.99)
    config.setdefault("epsilon", 1e-8)
    config.setdefault("cache", np.zeros_like(w))

    next_w = None
```

```
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    keys = ['learning_rate','decay_rate','epsilon','cache'] # keys in this order
    lr, dr, eps, cache = (config.get(key) for key in keys)  # vals in this order

    config['cache'] = dr * cache + (1 - dr) * dw**2         # update cache
    next_w = w - lr * dw / (np.sqrt(config['cache']) + eps) # update w

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

    return next_w, config


def adam(w, dw, config=None):
```

Fact 181

Uses the Adam update rule, which incorporates moving averages of both the gradient and its square and a bias correction term.

config format:

- learning_rate: Scalar learning rate.
- beta1: Decay rate for moving average of first moment of gradient.
- beta2: Decay rate for moving average of second moment of gradient.
- epsilon: Small scalar used for smoothing to avoid dividing by zero.
- m: Moving average of gradient.
- v: Moving average of squared gradient.
- t: Iteration number.

Problem 182

Implement the Adam update formula, storing the next value of w in the next_w variable. Don't forget to update the m, v, and t variables stored in config.

NOTE: In order to match the reference output, please modify t _before_ using it in any calculations.

```
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-3)
```

```
    config.setdefault("beta1", 0.9)
    config.setdefault("beta2", 0.999)
    config.setdefault("epsilon", 1e-8)
    config.setdefault("m", np.zeros_like(w))
    config.setdefault("v", np.zeros_like(w))
    config.setdefault("t", 0)

    next_w = None

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    keys = ['learning_rate','beta1','beta2','epsilon','m','v','t'] # keys in this order
    lr, beta1, beta2, eps, m, v, t = (config.get(k) for k in keys) # vals in this order

    config['t'] = t = t + 1                            # iteration counter
    config['m'] = m = beta1 * m + (1 - beta1) * dw     # gradient smoothing (Momentum)
    mt = m / (1 - beta1**t)                            # bias correction
    config['v'] = v = beta2 * v + (1 - beta2) * (dw**2) # gradient smoothing (RMSprop)
    vt = v / (1 - beta2**t)                            # bias correction
    next_w = w - lr * mt / (np.sqrt(vt) + eps)         # weight update

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################

    return next_w, config
```

layers.py(2)

Fact 183 (Forward pass for batch normalization)

During training the sample mean and (uncorrected) sample variance are computed from minibatch statistics and used to normalize the incoming data. During training we also keep an exponentially decaying running mean of the mean and variance of each feature, and these averages are used to normalize data at test-time.

At each timestep we update the running averages for mean and variance using an exponential decay based on the momentum parameter:

- running_mean = momentum * running_mean + (1 - momentum) * sample_mean
- running_var = momentum * running_var + (1 - momentum) * sample_var

Note that the batch normalization paper suggests a different test-time behavior: they compute sample mean and variance for each feature using a large number of training images rather than using a running average. For this implementation we have chosen to use running averages instead since they do not require an additional estimation step; the torch7 implementation of batch normalization also uses running averages.

Input:

- x: Data of shape (N, D)
- gamma: Scale parameter of shape (D,)
- beta: Shift paremeter of shape (D,)
- bn_param: Dictionary with the following keys:
- mode: 'train' or 'test'; required
- eps: Constant for numeric stability
- momentum: Constant for running mean / variance.
- running_mean: Array of shape (D,) giving running mean of features
- running_var Array of shape (D,) giving running variance of features

Returns a tuple of:

- out: of shape (N, D)
- cache: A tuple of values needed in the backward pass

```
def batchnorm_forward(x, gamma, beta, bn_param):
    mode = bn_param["mode"]
    eps = bn_param.get("eps", 1e-5)
    momentum = bn_param.get("momentum", 0.9)

    N, D = x.shape
    running_mean = bn_param.get("running_mean", np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get("running_var", np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == "train":
```

Problem 184

Implement the training-time forward pass for batch norm. Use minibatch statistics to compute the mean and variance, use these statistics to normalize the incoming data, and scale and shift the normalized data using gamma and beta. You should store the output in the variable out. Any intermediates that you need for the backward pass should be stored in the cache variable. You should also use your computed sample mean and variance together with the momentum variable to update the running mean and running variance, storing your result in the running_mean and running_var variables. Note that though you should be keeping track of the running variance, you should normalize the data based on the standard deviation (square root of variance) instead! Referencing the original paper (https://arxiv.org/abs/1502.03167) might prove to be helpful.

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        mu = x.mean(axis=0)         # batch mean for each feature
        var = x.var(axis=0)         # batch variance for each feature
        std = np.sqrt(var + eps)    # batch standard deviation for each feature
        x_hat = (x - mu) / std      # standartized x
        out = gamma * x_hat + beta # scaled and shifted x_hat

        shape = bn_param.get('shape', (N, D))              # reshape used in backprop
        axis = bn_param.get('axis', 0)                     # axis to sum used in backprop
        cache = x, mu, var, std, gamma, x_hat, shape, axis # save for backprop

        if axis == 0:                                                  # if not batchnorm
            running_mean = momentum * running_mean + (1 - momentum) * mu # update overall mean
            running_var = momentum * running_var + (1 - momentum) * var  # update overall variance

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ############################################################################
        #                           END OF YOUR CODE                               #
        ############################################################################
    elif mode == "test":
```

Problem 185

Implement the test-time forward pass for batch normalization. Use the running mean and variance to normalize the incoming data, then scale and shift the normalized data using gamma and beta.Store the result in the out variable.

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        x_hat = (x - running_mean) / np.sqrt(running_var + eps)
        out = gamma * x_hat + beta
```

```
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ##############################################################################
        #                           END OF YOUR CODE                                 #
        ##############################################################################
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param["running_mean"] = running_mean
    bn_param["running_var"] = running_var

    return out, cache


def batchnorm_backward(dout, cache):
```

Fact 186 (Backward pass for batch normalization)

For this implementation, you should write out a computation graph for batch normalization on paper and propagate gradients backward through intermediate nodes.

Inputs:

- dout: Upstream derivatives, of shape (N, D)
- cache: Variable of intermediates from batchnorm_forward.

Returns a tuple of:

- dx: Gradient with respect to inputs x, of shape (N, D)
- dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
- dbeta: Gradient with respect to shift parameter beta, of shape (D,)

Problem 187

Implement the backward pass for batch normalization. Store the results in the dx, dgamma, and dbeta variables. Referencing the original paper (https://arxiv.org/abs/1502.03167) might prove to be helpful.

```
    dx, dgamma, dbeta = None, None, None

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalizatio

    x, mu, var, std, gamma, x_hat, shape, axis = cache          # expand cache
```

```
    dbeta = dout.reshape(shape, order='F').sum(axis)             # derivative w.r.t. beta
    dgamma = (dout * x_hat).reshape(shape, order='F').sum(axis) # derivative w.r.t. gamma

    dx_hat = dout * gamma                                        # derivative w.t.r. x_hat
    dstd = -np.sum(dx_hat * (x-mu), axis=0) / (std**2)           # derivative w.t.r. std
    dvar = 0.5 * dstd / std                                      # derivative w.t.r. var
    dx1 = dx_hat / std + 2 * (x-mu) * dvar / len(dout)           # partial derivative w.t.r. dx
    dmu = -np.sum(dx1, axis=0)                                   # derivative w.t.r. mu
    dx2 = dmu / len(dout)                                        # partial derivative w.t.r. dx
    dx = dx1 + dx2                                               # full derivative w.t.r. x

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################

    return dx, dgamma, dbeta


def batchnorm_backward_alt(dout, cache):
```

Fact 188 (Alternative backward pass for batch normalization)

For this implementation you should work out the derivatives for the batch normalizaton backward pass on paper and simplify as much as possible. You should be able to derive a simple expression for the backward pass. See the jupyter notebook for more hints.

Note: This implementation should expect to receive the same cache variable as batchnorm_backward, but might not use all of the values in the cache.

Inputs / outputs: Same as batchnorm_backward """

Problem 189

Implement the backward pass for batch normalization. Store the results in the dx, dgamma, and dbeta variables. After computing the gradient with respect to the centered inputs, you should be able to compute gradients with respect to the inputs in a single statement; our implementation fits on a single 80-character line.

```
    dx, dgamma, dbeta = None, None, None

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    _, _, _, std, gamma, x_hat, shape, axis = cache # expand cache
    S = lambda x: x.sum(axis=0)                     # helper function
```

```
    dbeta = dout.reshape(shape, order='F').sum(axis)            # derivative w.r.t. beta
    dgamma = (dout * x_hat).reshape(shape, order='F').sum(axis) # derivative w.r.t. gamma

    dx = dout * gamma / (len(dout) * std)          # temporarily initialize scale value
    dx = len(dout)*dx  - S(dx*x_hat)*x_hat - S(dx) # derivative w.r.t. unnormalized x

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##########################################################################
    #                             END OF YOUR CODE                           #
    ##########################################################################

    return dx, dgamma, dbeta


def layernorm_forward(x, gamma, beta, ln_param):
    """Forward pass for layer normalization.

    During both training and test-time, the incoming data is normalized per data-point,
    before being scaled by gamma and beta parameters identical to that of batch normalization.

    Note that in contrast to batch normalization, the behavior during train and test-time for
    layer normalization are identical, and we do not need to keep track of running averages
    of any sort.

    Input:
    - x: Data of shape (N, D)
    - gamma: Scale parameter of shape (D,)
    - beta: Shift paremeter of shape (D,)
    - ln_param: Dictionary with the following keys:
        - eps: Constant for numeric stability

    Returns a tuple of:
    - out: of shape (N, D)
    - cache: A tuple of values needed in the backward pass
    """
    out, cache = None, None
    eps = ln_param.get("eps", 1e-5)
    ##########################################################################
    # TODO: Implement the training-time forward pass for layer norm.         #
    # Normalize the incoming data, and scale and  shift the normalized data  #
    #  using gamma and beta.                                                 #
    # HINT: this can be done by slightly modifying your training-time        #
    # implementation of  batch normalization, and inserting a line or two of #
```

```
    # well-placed code. In particular, can you think of any matrix            #
    # transformations you could perform, that would enable you to copy over   #
    # the batch norm code and leave it almost unchanged?                      #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    bn_param = {"mode": "train", "axis": 1, **ln_param} # same as batchnorm in train mode + over which axi
    [gamma, beta] = np.atleast_2d(gamma, beta)          # assure 2D to perform transpose

    out, cache = batchnorm_forward(x.T, gamma.T, beta.T, bn_param) # same as batchnorm
    out = out.T                                                    # transpose back

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return out, cache


def layernorm_backward(dout, cache):
    """Backward pass for layer normalization.

    For this implementation, you can heavily rely on the work you've done already
    for batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, D)
    - cache: Variable of intermediates from layernorm_forward.

    Returns a tuple of:
    - dx: Gradient with respect to inputs x, of shape (N, D)
    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    dx, dgamma, dbeta = None, None, None
    ###########################################################################
    # TODO: Implement the backward pass for layer norm.                       #
    #                                                                         #
    # HINT: this can be done by slightly modifying your training-time         #
    # implementation of batch normalization. The hints to the forward pass    #
    # still apply!                                                            #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    dx, dgamma, dbeta = batchnorm_backward_alt(dout.T, cache) # same as batchnorm backprop
    dx = dx.T                                                 # transpose back dx

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dgamma, dbeta


def dropout_forward(x, dropout_param):
    """Forward pass for inverted dropout.

    Note that this is different from the vanilla version of dropout.
    Here, p is the probability of keeping a neuron output, as opposed to
    the probability of dropping a neuron output.
    See http://cs231n.github.io/neural-networks-2/#reg for more details.

    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
      - p: Dropout parameter. We keep each neuron output with probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes this
        function deterministic, which is needed for gradient checking but not
        in real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: tuple (dropout_param, mask). In training mode, mask is the dropout
      mask that was used to multiply the input; in test mode, mask is None.
    """
    p, mode = dropout_param["p"], dropout_param["mode"]
    if "seed" in dropout_param:
        np.random.seed(dropout_param["seed"])

    mask = None
    out = None

    if mode == "train":
        ###########################################################################
```

```
        # TODO: Implement training phase forward pass for inverted dropout.   #
        # Store the dropout mask in the mask variable.                        #
        ###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        mask = (np.random.rand(*x.shape) < p) / p
        out = x * mask

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ###########################################################################
        #                            END OF YOUR CODE                             #
        ###########################################################################
    elif mode == "test":
        ###########################################################################
        # TODO: Implement the test phase forward pass for inverted dropout.   #
        ###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        out = x

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ###########################################################################
        #                             END OF YOUR CODE                            #
        ###########################################################################

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache


def dropout_backward(dout, cache):
    """Backward pass for inverted dropout.

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (dropout_param, mask) from dropout_forward.
    """
    dropout_param, mask = cache
    mode = dropout_param["mode"]

    dx = None
    if mode == "train":
```

```
        ###########################################################################
        # TODO: Implement training phase backward pass for inverted dropout   #
        ###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        dx = dout * mask

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ###########################################################################
        #                            END OF YOUR CODE                             #
        ###########################################################################
    elif mode == "test":
        dx = dout
    return dx


def conv_forward_naive(x, w, b, conv_param):
    """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 WW.

    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.

    During padding, 'pad' zeros should be placed symmetrically (i.e equally on both sides)
    along the height and width axes of the input. Be careful not to modfiy the original
    input x directly.

    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)
    """
    out = None
```

```
    ###########################################################################
    # TODO: Implement the convolutional forward pass.                         #
    # Hint: you can use the function np.pad for padding.                      #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    P1 = P2 = P3 = P4 = conv_param['pad'] # padding: up = right = down = left
    S1 = S2 = conv_param['stride']        # stride:  up = down
    N, C, HI, WI = x.shape                # input dims
    F, _, HF, WF = w.shape                # filter dims
    HO = 1 + (HI + P1 + P3 - HF) // S1    # output height
    WO = 1 + (WI + P2 + P4 - WF) // S2    # output width

    # Helper function (warning: numpy version 1.20 or above is required for usage)
    to_fields = lambda x: np.lib.stride_tricks.sliding_window_view(x, (WF,HF,C,N))

    w_row = w.reshape(F, -1)                                          # weights as rows
    x_pad = np.pad(x, ((0,0), (0,0), (P1, P3), (P2, P4)), 'constant')   # padded inputs
    x_col = to_fields(x_pad.T).T[...,::S1,::S2].reshape(N, C*HF*WF, -1) # inputs as cols

    out = (w_row @ x_col).reshape(N, F, HO, WO) + np.expand_dims(b, axis=(2,1))

    x = x_pad # we will use padded version as well during backpropagation

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b, conv_param)
    return out, cache


def conv_backward_naive(dout, cache):
    """A naive implementation of the backward pass for a convolutional layer.

    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
```

```
    """
    dx, dw, db = None, None, None
    ###########################################################################
    # TODO: Implement the convolutional backward pass.                        #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # Helper function (warning: numpy 1.20+ is required)
    to_fields = np.lib.stride_tricks.sliding_window_view

    x_pad, w, b, conv_param = cache       # extract parameters from cache
    S1 = S2 = conv_param['stride']        # stride:  up = down
    P1 = P2 = P3 = P4 = conv_param['pad'] # padding: up = right = down = left
    F, C, HF, WF = w.shape                # filter dims
    N, _, HO, WO = dout.shape             # output dims

    dout = np.insert(dout, [*range(1, HO)] * (S1-1), 0, axis=2)         # "missing" rows
    dout = np.insert(dout, [*range(1, WO)] * (S2-1), 0, axis=3)         # "missing" columns
    dout_pad = np.pad(dout, ((0,), (0,), (HF-1,), (WF-1,)), 'constant') # for full convolution

    x_fields = to_fields(x_pad, (N, C, dout.shape[2], dout.shape[3]))   # input local regions w.r.t. dout
    dout_fields = to_fields(dout_pad, (N, F, HF, WF))                   # dout local regions w.r.t. filter
    w_rot = np.rot90(w, 2, axes=(2, 3))                                 # rotated kernel (for convolution)

    db = np.einsum('ijkl->j', dout)                                                 # sum over
    dw = np.einsum('ijkl,mnopiqkl->jqop', dout, x_fields)                           # correlate
    dx = np.einsum('ijkl,mnopqikl->qjop', w_rot, dout_fields)[..., P1:-P3, P2:-P4] # convolve

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dw, db


def max_pool_forward_naive(x, pool_param):
    """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

    No padding is necessary here, eg you can assume:
      - (H - pool_height) % stride == 0
      - (W - pool_width) % stride == 0

    Returns a tuple of:
    - out: Output data, of shape (N, C, H', W') where H' and W' are given by
      H' = 1 + (H - pool_height) / stride
      W' = 1 + (W - pool_width) / stride
    - cache: (x, pool_param)
    """
    out = None
    ###########################################################################
    # TODO: Implement the max-pooling forward pass                            #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    S1 = S2 = pool_param['stride'] # stride: up = down
    HP = pool_param['pool_height'] # pool height
    WP = pool_param['pool_width']  # pool width
    N, C, HI, WI = x.shape         # input dims
    HO = 1 + (HI - HP) // S1       # output height
    WO = 1 + (WI - WP) // S2       # output width

    # Helper function (warning: numpy version 1.20 or above is required for usage)
    to_fields = lambda x: np.lib.stride_tricks.sliding_window_view(x, (WP,HP,C,N))

    x_fields = to_fields(x.T).T[...,::S1,::S2].reshape(N, C, HP*WP, -1) # input local regions
    out = x_fields.max(axis=2).reshape(N, C, HO, WO)                    # pooled output

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, pool_param)
    return out, cache


def max_pool_backward_naive(dout, cache):
    """A naive implementation of the backward pass for a max-pooling layer.

    Inputs:
```

```
    - dout: Upstream derivatives
    - cache: A tuple of (x, pool_param) as in the forward pass.

    Returns:
    - dx: Gradient with respect to x
    """
    dx = None
    ###########################################################################
    # TODO: Implement the max-pooling backward pass                           #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x, pool_param = cache     # expand cache
    N, C, HO, WO = dout.shape # get shape values
    dx = np.zeros_like(x)     # init derivative

    S1 = S2 = pool_param['stride'] # stride: up = down
    HP = pool_param['pool_height'] # pool height
    WP = pool_param['pool_width']  # pool width

    for i in range(HO):
        for j in range(WO):
            [ns, cs], h, w = np.indices((N, C)), i*S1, j*S2    # compact indexing
            f = x[:, :, h:(h+HP), w:(w+WP)].reshape(N, C, -1)  # input local fields
            k, l = np.unravel_index(np.argmax(f, 2), (HP, WP)) # offsets for max vals
            dx[ns, cs, h+k, w+l] += dout[ns, cs, i, j]         # select areas to update

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx


def spatial_batchnorm_forward(x, gamma, beta, bn_param):
    """Computes the forward pass for spatial batch normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (C,)
    - beta: Shift parameter, of shape (C,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
```

```
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance. momentum=0 means that
      old information is discarded completely at every time step, while
      momentum=1 means that new information is never incorporated. The
      default of momentum=0.9 should work well in most situations.
    - running_mean: Array of shape (D,) giving running mean of features
    - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None

    ###########################################################################
    # TODO: Implement the forward pass for spatial batch normalization.       #
    #                                                                         #
    # HINT: You can implement spatial batch normalization by calling the      #
    # vanilla version of batch normalization you implemented above.           #
    # Your implementation should be very short; ours is less than five lines. #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N, C, H, W = x.shape                                    # input dims
    x = np.moveaxis(x, 1, -1).reshape(-1, C)                # swap axes to use vanilla batchnorm
    out, cache = batchnorm_forward(x, gamma, beta, bn_param) # perform vanilla batchnorm
    out = np.moveaxis(out.reshape(N, H, W, C), -1, 1)       # swap back axes for the output

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################

    return out, cache


def spatial_batchnorm_backward(dout, cache):
    """Computes the backward pass for spatial batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass
```

```
    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None

    ###########################################################################
    # TODO: Implement the backward pass for spatial batch normalization.      #
    #                                                                         #
    # HINT: You can implement spatial batch normalization by calling the      #
    # vanilla version of batch normalization you implemented above.           #
    # Your implementation should be very short; ours is less than five lines. #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N, C, H, W = dout.shape                             # upstream dims
    dout = np.moveaxis(dout, 1, -1).reshape(-1, C)      # swap axes to use vanilla batchnorm backprop
    dx, dgamma, dbeta = batchnorm_backward(dout, cache) # perform vanilla batchnorm backprop
    dx = np.moveaxis(dx.reshape(N, H, W, C), -1, 1)     # swap back axes for the gradient of dx

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################

    return dx, dgamma, dbeta


def spatial_groupnorm_forward(x, gamma, beta, G, gn_param):
    """Computes the forward pass for spatial group normalization.

    In contrast to layer normalization, group normalization splits each entry in the data into G
    contiguous pieces, which it then normalizes independently. Per-feature shifting and scaling
    are then applied to the data, in a manner identical to that of batch normalization and layer
    normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (1, C, 1, 1)
    - beta: Shift parameter, of shape (1, C, 1, 1)
    - G: Integer number of groups to split into, should be a divisor of C
    - gn_param: Dictionary with the following keys:
```

```
    - eps: Constant for numeric stability

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None
    eps = gn_param.get("eps", 1e-5)
    ##############################################################################
    # TODO: Implement the forward pass for spatial group normalization.          #
    # This will be extremely similar to the layer norm implementation.           #
    # In particular, think about how you could transform the matrix so that      #
    # the bulk of the code is similar to both train-time batch normalization     #
    # and layer normalization!                                                   #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    N, C, H, W = x.shape                                          # input dims
    ln_param = {"shape":(W, H, C, N), "axis":(0, 1, 3), **gn_param} # params to reuse batchnorm method

    x = x.reshape(N*G, -1)                                        # reshape x to use vanilla layernorm
    gamma = np.tile(gamma, (N, 1, H, W)).reshape(N*G, -1)         # reshape gamma to use vanilla layernorm
    beta = np.tile(beta, (N, 1, H, W)).reshape(N*G, -1)           # reshape beta to use vanilla layernorm

    out, cache = layernorm_forward(x, gamma, beta, ln_param)      # perform vanilla layernorm
    out = out.reshape(N, C, H, W)                                 # reshape back the output
    cache = (G, cache)                                            # cache involves G

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################
    return out, cache


def spatial_groupnorm_backward(dout, cache):
    """Computes the backward pass for spatial group normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
```

```
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (1, C, 1, 1)
    - dbeta: Gradient with respect to shift parameter, of shape (1, C, 1, 1)
    """
    dx, dgamma, dbeta = None, None, None

    ###########################################################################
    # TODO: Implement the backward pass for spatial group normalization.      #
    # This will be extremely similar to the layer norm implementation.        #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    G, cache = cache                                    # expand cache
    N, C, H, W = dout.shape                             # upstream dims
    dout = dout.reshape(N*G, -1)                        # reshape to use vanilla layernorm backprop

    dx, dgamma, dbeta = layernorm_backward(dout, cache) # perform vanilla layernorm backprop
    dx = dx.reshape(N, C, H, W)                         # reshape back dx
    dbeta = dbeta[None, :, None, None]                  # resape back dbeta
    dgamma = dgamma[None, :, None, None]                # reshape back dgamma

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dgamma, dbeta
```

layer_utils.py

```
from .layers import *
from .fast_layers import *
def affine_relu_forward(x, w, b):
```

Fact 190 (Convenience layer that performs an affine transform followed by a ReLU)

Inputs:

- x: Input to the affine layer
- w, b: Weights for the affine layer

Returns a tuple of:

- out: Output from the ReLU
- cache: Object to give to the backward pass

```
    a, fc_cache = affine_forward(x, w, b)
```

```
    out, relu_cache = relu_forward(a)
    cache = (fc_cache, relu_cache)
    return out, cache

def affine_relu_backward(dout, cache):
```

Fact 191

Backward pass for the affine-relu convenience layer.

```
    fc_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

def generic_forward(x, w, b, gamma=None, beta=None, bn_param=None, dropout_param=None, last=False):
```

Fact 192

Convenience layer that performs an affine transform, a batch/layer normalization if needed, a ReLU, and dropout if needed.

Inputs:

- x: Input to the affine layer
- w, b: Weights for the affine layer
- gamma, beta: Scale and shift params for the batch normalization
- bn_param: Dictionary of required BN parameters
- dropout_param: Dictionary of required Dropout parameters
- last: Indicates wether to perform just affine forward

Returns a tuple of:

- out: Output from the ReLU or Dropout
- cache: Object to give to the backward pass

```
    # Initialize optional caches to None
    bn_cache, ln_cache, relu_cache, dropout_cache = None, None, None, None

    # Affine forward is a must
    out, fc_cache = affine_forward(x, w, b)
```

```
    # If the the layer is not last
    if not last:
        # If it has normalization layer we normalize outputs: if it bn_param
        # has mode (train | test), it's batchnorm, otherwise, it's layernorm
        if bn_param is not None:
            if 'mode' in bn_param:
                out, bn_cache = batchnorm_forward(out, gamma, beta, bn_param)
            else:
                out, ln_cache = layernorm_forward(out, gamma, beta, bn_param)

        # Pass the outputs through activation
        out, relu_cache = relu_forward(out) # perform relu

        # Use dropout if we are given its parameters
        if dropout_param is not None:
            out, dropout_cache = dropout_forward(out, dropout_param)

    # Prepare cache for backward pass
    cache = fc_cache, bn_cache, ln_cache, relu_cache, dropout_cache

    return out, cache

def generic_backward(dout, cache):
```

Fact 193

Backward pass for the affine-bn/ln?-relu-dropout? convenience layer.

```
    # Init norm params to None
    dgamma, dbeta = None, None

    # Get the prapared caches from the forward pass
    fc_cache, bn_cache, ln_cache, relu_cache, dropout_cache = cache

    # If dropout was performed
    if dropout_cache is not None:
        dout = dropout_backward(dout, dropout_cache)

    # If relu was performed
    if relu_cache is not None:
        dout = relu_backward(dout, relu_cache)

    # If norm was performed
```

```
    if bn_cache is not None:
        dout, dgamma, dbeta = batchnorm_backward_alt(dout, bn_cache)
    elif ln_cache is not None:
        dout, dgamma, dbeta = layernorm_backward(dout, ln_cache)

    # Affine backward is a must
    dx, dw, db = affine_backward(dout, fc_cache)

    return dx, dw, db, dgamma, dbeta

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

def conv_relu_forward(x, w, b, conv_param):
    """A convenience layer that performs a convolution followed by a ReLU.

    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer

    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    a, conv_cache = conv_forward_fast(x, w, b, conv_param)
    out, relu_cache = relu_forward(a)
    cache = (conv_cache, relu_cache)
    return out, cache


def conv_relu_backward(dout, cache):
    """Backward pass for the conv-relu convenience layer.
    """
    conv_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db


def conv_bn_relu_forward(x, w, b, gamma, beta, conv_param, bn_param):
    """Convenience layer that performs a convolution, a batch normalization, and a ReLU.

    Inputs:
    - x: Input to the convolutional layer
```

```
    - w, b, conv_param: Weights and parameters for the convolutional layer
    - pool_param: Parameters for the pooling layer
    - gamma, beta: Arrays of shape (D2,) and (D2,) giving scale and shift
      parameters for batch normalization.
    - bn_param: Dictionary of parameters for batch normalization.

    Returns a tuple of:
    - out: Output from the pooling layer
    - cache: Object to give to the backward pass
    """
    a, conv_cache = conv_forward_fast(x, w, b, conv_param)
    an, bn_cache = spatial_batchnorm_forward(a, gamma, beta, bn_param)
    out, relu_cache = relu_forward(an)
    cache = (conv_cache, bn_cache, relu_cache)
    return out, cache


def conv_bn_relu_backward(dout, cache):
    """Backward pass for the conv-bn-relu convenience layer.
    """
    conv_cache, bn_cache, relu_cache = cache
    dan = relu_backward(dout, relu_cache)
    da, dgamma, dbeta = spatial_batchnorm_backward(dan, bn_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db, dgamma, dbeta


def conv_relu_pool_forward(x, w, b, conv_param, pool_param):
    """Convenience layer that performs a convolution, a ReLU, and a pool.

    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer
    - pool_param: Parameters for the pooling layer

    Returns a tuple of:
    - out: Output from the pooling layer
    - cache: Object to give to the backward pass
    """
    a, conv_cache = conv_forward_fast(x, w, b, conv_param)
    s, relu_cache = relu_forward(a)
    out, pool_cache = max_pool_forward_fast(s, pool_param)
    cache = (conv_cache, relu_cache, pool_cache)
```

```
    return out, cache


def conv_relu_pool_backward(dout, cache):
    """Backward pass for the conv-relu-pool convenience layer.
    """
    conv_cache, relu_cache, pool_cache = cache
    ds = max_pool_backward_fast(dout, pool_cache)
    da = relu_backward(ds, relu_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db
```

6 Assigment 3

6.1 Q1: Image Captioning with Vanilla RNNs

6.2 Q2: Image Captioning with Transformers

6.3 Q3: GANs

6.4 Q4: Self supervised learning for image classification

6.5 Q5: Image Captioning with LTSMs