**Assignment 1**

* Np.sum() sums all of the elements in a matrix across a specific axis which you specify and returns a matrix with a lower dimension (e.g. if it was a 2 by 2, would return a 2 by 1 or 1 by 2)
* If you don’t specify an axis, it will add all of the individual elements in a matrix
* Axis 0 is the equivalent of summing down rows (adding elements downward)
* Axis 1 is the equivalent of summing along columns (adding elements across)
* Np.argsort() takes an array and returns the indices corresponding to the sorted array
* E.g. if an\_array = [0, 2,1,5]
* Out\_array = np.argsort(an\_array) => [0, 2, 1, 3]
* Print(an\_array[Out\_array]) would print the sorted array [0, 1, 2, 5]
* Np.bincount(some\_array) will return the count of each element in some\_array
* For example if some\_array = [1, 1, 0, 2]
* Np.bincount(some\_array) => [1, 2, 1]
  + You could then use np.argmax(result above) to get the most frequent element in an array
* **Broadcasting**
  + Working with one vector to compute an arithmetic multiple times on a larger matrix
  + Let’s say you have one vector i and you want to subtract x from a matrix j made of up of (n\_samples x same dimension of i)
    - i – j[:,] will accomplish this
  + Could also do this by stacking i to create a matrix of the same size as j (can do np.tile(i, (num\_rows to copy, 1)
  + The principle of broadcasting is if we have an m x n matrix and a 1 x n matrix and we tried to apply an arithmetic operation between them
    - The smaller matrix will be copied to an m x n matrix and then the operation will be carried out
* Can splice an array in numpy as you would a normal array – only difference is spice as [condition for row, condition for column]
* E.g. array[:, 1:3] would get the element from the 1st and 2nd across every row
* [:,] is the same as [:,:]
* X\*y is elementwise multiplication (multiply each element of X by the corresponding one in y)
* X\*\*2 is squaring every element in a matrix
* Np.nexaxis will increase the dimensionality of a matrix e.g. used like a = a[:, np.nexaxis]
* Useful tool for broadcasting and adding two 1-dimensional arrays of different sizes
* For example, imagine if you had a bunch of row vectors in a matrix and you want to subtract the maximum value for each row from every element in the matrix
* To do that
* Maxes = np.max(x, axis = 1)
* Maxes = Maxes[:, np.newaxis] #increase the dimensionality to allow numpy to do some magic
* X = x – maxes
* When you use array[:,], remember that it means it does some operation or assigns some variable to EVERY ROW AND COLUMN
* Thus if you are trying to code a vectorized loss function – doing np.exp(array[:.]) / np.sum(np.exp(array[:,])) would NOT PERFORM IN THE EXPECTED WAY
  + Should make temp variables for numerator and denominator then divide
* Make sure you have reshaped any column/row vectors (N,) in numpy before broadcasting otherwise it won’t work correctly

**Vectorized Version of Calculating L2 Distances**

* Fully vectorized version of calculating l2 distances involves expanding (x-y)^2 = x^2 + y^2 -2xy
* Can obtain the desired size by doing X\*\*2 and summing across the columns then doing Y\*\*2 and summing across the columns -2x.self.X\_train.T
* This gives will give X\*\*2 the right number of rows and Y\*\*2 the right number of columns and -2xy is already the right size meaning two broadcasting operations are then done to generate the correct l2distances

**Splitting Arrays**

* Np.array\_split(Matrix to split, subfolds) will split an array into folds e.g. if you have 500 samples calling np.array\_slit(matrix, 5) will return a list of 5 arrays and each array will contain total\_samples/5

**Accuracy**

* Given an array of predictions and an array of labels, can easily get the accuracy by doing np.sum(predictions == labels)
* If you’re getting 100% accuracy you’re most definitely testing on training data you’ve seen

**Cross-Validation**

* Used for hyperparameter tuning
* Involves splitting the training data into folds
* Typically it’s called K-cross validation because for each value of k (in a k-nearest neighbor program) we train each value of k num\_fold times
  + For each time, we train on all of the folds but 1 and validate on the last fold
  + **IMPORTANT POINT – each time you run it**, have **TO PICK A DIFFERENT VALIDATION FOLD**

A screenshot of a cell phone

Description automatically generated

**To get the training data in this case, use np.concatenate()**

* If you used np.array\_split() as above, you will need to concatenate them but it won’t let you because often you will be concatenating two things of different dimension (remember in array\_split returns smt like [array[….], array[…]
* Thus you need to add the two different dimension of training data e.g. in the picture round 2 above set\_1 (1 dimension) + set\_2 (8 dimension) then call np.concatenate on the result
  + Very important that you add the two lists of arrays with +
  + If you try to create a new array and append each of the sets, you add a dimension
    - Becomes like [[array[…], array[…]] and numpy won’t be able to parse it

**Linear Classification**

* Do the same as with a neural network except have one score function Y=W.x +b and you optimize this by minimizing a lost function
* Once trick is to add an extra dimension in the weight matrix and inputs as the *bias dimension* so that everything can be computed with one matrix calculation
* A picture containing clock

  Description automatically generated
* It’s very important to always normalize data in machine learning
* In image classification, this corresponds to computing the mean image across all of the training data and subtracting it from every image

**Multiclass SVM Loss**

* Hinge Loss
* A screenshot of a cell phone

  Description automatically generated
* To compute the loss for any individual example, we sum over all of the incorrect classes (i.e. if this was a cat from CIFAR 10, sum over all of the 9 other classes) except for the true class
* For each class, if difference between correct category (denote by yi) is greater than the incorrect score by a safety margin (we set to 1), then it means the true score is much larger than the incorrect score, thus we set the loss to 0
* Note it may be more helpful to think of the second hand as Sj + 1 -Syi (thus if the score of Syi is much larger than Sj + 1, this will sum to something -ve and max(0, -ve) will return 0
* The 1 in this case is a safety margin, but this is a hyperparameter we can tweak

A picture containing clock

Description automatically generated

* **At the start of training, weights initialized will be small and random thus all of the scores will be around 0. This means the expected loss should be the number of categories – 1 (since the loss will be 1 for each category and we loop through this for all of the incorrect categories thus num categories – 1)**
* Could also change the loss function so that it’s max(0,….)^2
  + The type of loss function you choose is very important because it’s how you quantify what types of mistakes your algorithm is making and how it weighs up A screenshot of a cell phone

    Description automatically generated
  + Snippet of vectorized code – trick is in margins[y] = 0 (this is so that we ignore the score produced when the correct category is compared with itself – since with SVM, we only look at all the incorrect categories and compare them with the correct category)
* **Occam’s Razor** 
  + Idea that among competing hypothesis, simplest one is the best (most likely to be able to generalize)
* We usually, end up a regularization parameter to the loss
* Type of regularization we use depends on our problem – how do we think complexity should be measured?
  + Each L1 looks at how many 0s there are (=simple model) whereas L2 looks at the one with smaller norm

**Softmax Classifier**

* More commonly used loss function A screenshot of a cell phone

  Description automatically generated
* **Note Syi represent the score of the correct class and the denominator is the sum of all of the scores across all classes**
* At the start, when W is small and initialized randomly, expect loss to be -log(1/c) which log(c )
* While SVM’s loss tries to only get the correct category score larger than the incorrect scores above by a certain margin (and then will stop), SVM will continually try to pile more and more probability mass onto the correct class to drive it to infinity and incorrect class to -infinity
* Softmax is the activation function and the loss (Li) is known as cross-entropy
* Derivative of the derivative of cross-entropy loss can be found here: <https://deepnotes.io/softmax-crossentropy>
* **MOST CLEAR DERIVATION OF DERIVATIVE OF SOFTMAX AND CROSS-ENTROPY:** <https://mattpetersen.github.io/softmax-with-cross-entropy>

**Some Important Derivative Notes**

<http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf>

<https://explained.ai/matrix-calculus/index.html#sec4.2>

* When there are a lot of subscripts, it can get confused what we’re differentiating with respect to, especially if there are sums involved. Some helpful notes:
  + If you have subscripts or superscripts, remember these denote values for different categories or often scores for different classes

A screenshot of a cell phone

Description automatically generated

* + - For example in the expression above, ai represents the score for one class
    - In order to express the idea that we are applying the same operation to the score of every category, we introduce a different subscript k (in the denominator) and sum it from k=1 and N
    - Note that within this range, i would be included

A screenshot of a cell phone

Description automatically generated

* Now when you see this derivative – remember, in order to do back-prop, we need to compute the derivative with respect every score and thus weight
  + This is why we introduce a new subscript j in aj since we need to calculate this derivative for every single score
  + Need to use the quotient rule here
  + There are two cases that occur that result in different derivatives, when i = j and when i != j
  + Note that the denominator sums all of the scores thus aj will always be included exactly once meaning the derivative of the denominator is just e^aj
  + Once you’ve found the derivative of the numerator for each case, remember the derivative of a function is the sum of its derivatives
  + Thus to compute the derivative above, we need to add the results from i=j and i != j
* Once you’ve worked through all of the math, the derivative of the loss function with respect to the weights becomes
* A close up of a logo

  Description automatically generated
* The derivative of X.W with respect to W would just be the vector X

**Backpropagation**

* In order to compute gradients for backprop, it’s helpful to focus in one example e.g. one element in the input and weight matrix and see what the gradient would be for that. This allows you to identify patterns, which give us implicit expressions for computing the Jacobian without actually doing so (since calculating it for large datasets would be computationally inefficient)
* The derivative ∂L /∂Y if L is a scalar and Y is a matrix will **be a matrix of dimension Y**
* If L was a matrix and Y was a scalar, then the derivative would **be a matrix of dimension L**
* The best way to make sure you’re gradients are correct are to
  + Draw the computation and graph and derive everything from scratch to make sure you’re not skipping steps/forgetting intermediate gradients
  + Check the dimensions of each variable you are differentiating with respect to (since loss is a scalar
* With certain activation functions e.g. RELU, it’s applied elementwise instead of a dot product
  + Since the derivative of RELU is a step function, once you have a matrix of the same size with just 1s and 0s, \* with dLast layer
* If z2 = x.W2 + b2
  + The derivative of the loss with respect to db2 will be the same as dz2
  + The only difference, is that to get the correct dimension – need to do np.sum(dz2, axis=0) – summing down the rows since bias is essentially a vector
* Learning rate decays make a MASSIVE difference to training (make sure to multiply the learning rate by the decay after every iteration

**Practical Tips**

**Data Preprocessing**

3 common techniques

1. Mean subtraction – **VERY IMPORTANT**
   1. Subtract mean from every single feature of the data
   2. Geometrically equivalent to centering data around origin
   3. Implemented as X -= np.mean(X, axis = 0)
      1. Can be common to subtract a single value from all of the pixels, no axis=0
      2. Can also do this across the different rgb channels
2. Normalization
   1. Ensures data dimensions are all the same – all data has the same scale
   2. Can be done in one of two ways
      1. Dividing by std as X /= np.std(X, axis = 0)
      2. Or normalize so that min = -1 and max = 1 (or 0 and 1)
         1. **Why would you do this?**
         2. If you think reason to believe that inputs have different sizes/dimensions but should be of equal importance
         3. Don’t need to do this with images since dimensions of (0, 255) are all consistent
3. PCA (Principal Component Analysis) and Whitening
   1. Data is first centered and normalized above

**Principal Component Analysis**

* 1. The compute the covariance matrix to tell us about correlation and structure
  2. Cov = **np.dot(X.T, X) / X.shape[0]**
     1. The (i, j) element contains the covariance between the ith and jth dimension of the data
  3. Using this can compute the SVD factorization of the data covariance matrix – **U, S, V = np.linalg.svd(cov)**
     1. U is columns of eigenvectors and S is a 1-D array of the singular values
  4. You then decorrelate data by projecting original data onto the eigenbasis
     1. **Xrot = np.dot(X, U)**
  5. Some Important Lin Algebra Notes
     1. Because columns of U are orthonormal, can be regarded as basis vectors (all linearly independent and span a subspace) s
     2. This projection then corresponds to projecting X onto axis where axis are eigenvectors
     3. Because U’s columns are sorted by their eigenvalues, we can select top few eigenvectors and discard the dimensions with little various = **PRINCIPLE COMPONENT ANALYSIS or** Dimensionality reduction
     4. **Xrot\_reduced = np.dot(X, U[:,:100])**
        1. Xrot\_reduced now instead of an N x D matrix of input data (X) is now N x 100 – have kept the 100 dimensions of data with the most variance

**Whitening![A close up of a map

Description automatically generated]()**

* 1. The middle is after performing PCA
  2. Whitening takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize it
  3. **Xwhite = Xrot / np.sqrt(S + 1e-5)**
  4. In practice, not used with CNNs

**Important Pitfall for Data Preprocessing**

* Any data preprocessing statistic should only be calculated on the training data then applied separately onto validation/testing data
* Meaning subtracting mean image from all the data then dividing into train/val/testing **IS WRONG**
* Instead, compute mean from only training data, the subtracting it from val and testing data

**Batch Normalization**

* Practically involves inserting a batch norm right after the fully connected layers before non-linearities
* <https://arxiv.org/abs/1502.03167>
* Allows using higher learning rates and be less careful with network initialization
* Intuitively, equivalent to doing preprocessing at every layer of the network but integrated so that it’s differentiable

**Weight Initialization**

* Should not all be 0s – WRONG
  + All neurons would have same output, thus same gradient, meaning would be updated by same parameter in backprop – NO ASYMMETRY
* Should be small and random
  + **W = 0.01 \* np.random.randn(D, H)**
  + Note though small = not necessarily good (may lead to diminishing gradients)
* Problem with doing it this:
  + Variance grows as number of neurons increase
  + Instead want to normalize variance of entire distribution to 1
  + Initialize weights as:
  + **W = np.random.randn(n) / sqrt(n)** or **W = np.random.randn(n) \*sqrt(2.0/n)**
* Common to set all biases to 0 since asymmetry is provided by initializing weights above as such

**Regularization – to prevent the dreaded overfitting**

* L2 regularization
  + Implemented by penalizing squared magnitude of parameters
  + Add term ½ \* regularization \* weight^2 (or np.dot(W.W))
  + Regularization = hyperparameter
  + Weights decay linearly to 0 – prefers diffuse weight vectors (i.e. [0.5, 0.5] instead of [1, 0]
* Max Norm Constraints
  + Involves bounding all weights below a certain constaint e.g. 3 or 4
  + Done by updating weights regularly then doing |w|^2 < c
  + Prevents network from exploding if learning rate is too high
* Dropout
  + Extremely effective and simple way to prevent overfitting
  + During training, involves randomly dropping (or setting weights to 0) of different neurons with a probability p (hyperparameter)
  + A screenshot of a social media post

    Description automatically generatedImplemented like:
* Note the scaling activation in the predict
* Important so that the expected output is the same as it was in training stage
* Max Norm Constraints
  + Involves bounding all weights below a certain constaint e.g. 3 or 4
  + Done by updating weights regularly then doing |w|^2 < c

Prevents network from exploding if learning rate is

**Assignment 2**

**Convolutional Neural Networks**

**Convolution Layers**

* Output size: ((N – F) / stride) + 1 (where N is the height/width of the image, and F is the width/height of the filter) and stride
* Common to zero-pad the border to create same size of activation as input (also allows centering on the corners)
* Sizes of activation maps end up being the output size (e.g. 3 by 3) by the number of filters so 3x3x3 (if using 3 filters)
* Note that filters multiply through the entire depth of the image so one filter dots the the weights across all channels (e.g. RGB) to produce one single value e.g. 1 filter for a 32x32x3 would produce for example a 32 x 32 x 1

For filter sizes, what padding to use?

A picture containing bottle, red

Description automatically generated

* **What’s the point of stride?**
  + Downsizes an image, acts kind of like a pooling player
  + Creates smaller outputs
* Each neuron in the activation map looks at a specific localized input
* Neurons in the same depth of the activation map can be interpreted as looking for different things in the same part of the image

**Summary**. To summarize, the Conv Layer:

* Accepts a volume of size W x H x D
* Requires four hyperparameters:
  + Number of filters K,
  + their spatial extent F,
  + the stride S,
  + the amount of zero padding P.
* Produces a volume of size W2 x H2 x D2
  + W2=(W1−F+2P)/S+1
  + H2=(H1−F+2P)/S+1 (i.e. width and height are computed equally by symmetry)
  + D2 = K
* With parameter sharing, it introduces F⋅F⋅D1F⋅F⋅D1 weights per filter, for a total of (F⋅F⋅D1)⋅K(F⋅F⋅D1)⋅K weights and KK biases.
* In the output volume, the dd-th depth slice (of size W2×H2W2×H2) is the result of performing a valid convolution of the dd-th filter over the input volume with a stride of SS, and then offset by dd-th bias.

**Pool Layers**

* Make representations smaller and more manageable
* Pools spatially to downsize it e.g. 224 x 224 x 12 to 112 x 112 x 12 – does NOT do anything to depth
* Operates over each activation map independently
* One way to do this is **Max Pooling**
  + Applies same filter, but instead of computing the dot product, takes max value from that filter
  + Often set the stride so that there is no overlap since we were down sampling
  + **Intuition:** Idea is that each parameter tells us how much the neuron fired in the image and when doing image recognition, useful to know if there is some part of the image that caused neurons to fire with a large value

A screenshot of a cell phone

Description automatically generated

* No need to use zero padding for pooling since we are just using it to down-sample images