Programming Assignment 1

CSE 251B: Deep Learning

Winter 2022

Instructions

Due on Wedensday, January 19th, 2022

1. Please submit your assignment on Gradescope. The instructions for this will be coming soon. There are two components to this assignment: mathematical solutions/proofs with English explanations (Part I: Problems 1 and 2). For the programming assignment portion of the homework (Part II: 1, 2, and 3), you will be writing a report in a conference paper format for this assignment, reporting your findings. All parts of the assignments must be typeset, including figures. We strongly recommend that you use some dialect of TEXor LATEX and use NeurIPS (one of the top ML/DL conferences) format, but this is not required. You may also use Word if you so choose, and figures may be generated with Excel or Python, so long as they are computer generated. **We will not be accepting any handwritten work - this includes the "written part."** NeurIPS templates in LATEX and Word are available from the 2015 NIPS format site. The page limits mentioned there don't apply.

2. For the group report, include an informative title (*not* **PAI**, but a real title that refers to the problem solved or whatever), author list, and an abstract. The abstract should summarize briefly what you did, and the best percent correct you got on each problem. The report should have clear sections for the three programming parts, **as well as a fourth section where each team member says what their contribution to the project was.** The report should be well-organized with an introduction, background (if you review previous work), methods, results, and discussion for each programming part. Figures should be near where they are referenced, there should be informative captions on figures, clearly specified axes and figure keys, etc.

3. You are expected to use Python (usually with NumPy). You also need to submit all of the source code files and a *readme.txt* file that includes detailed instructions on how to run your code.

   You should write clean code with consistent format, as well as explanatory comments, as this code may be reused in the future.

4. Using any off-the-shelf code is strictly prohibited.

5. ***If you end up dropping the class and your teammate does not, you are expected to help your teammate anyway!*** Please don't leave your teammate(s) without your assistance. Being on a team means just that: teamwork! When you join a team, you have made a commitment. Please honor it.

6. Any form of copying, plagiarizing, grabbing code from the web, having someone else write your code for you, etc., is cheating. We expect you all to do your own work, and when you are on a team, to pull your weight. Team members who do not contribute will not receive the same scores as those who do. Discussions of course materials and homework solutions are encouraged, but you should write the final solutions alone. Books, notes, and Internet resources can be consulted, but not copied from. Working together on homework must follow the spirit of the **Gilligan's Island Rule** (Dymond, 1986): No notes can be made (or recording of any kind) during a discussion, and you must watch one hour of Gilligan's Island or something equally insipid before writing anything down. Suspected cheating has been and will be reported to the UCSD Academic Integrity office.

Part I

Problems to be solved and turned in individually

For this part we will *not* be accepting handwritten reports. Please use latex or word for your report. MathType is a handy tool for equations in Word. The free version (MathType Lite) has everything you need.

1. **Problems from Bishop (20 points)**

   Work problems 1-4 (5 points each) on pages 28-30 of Bishop. *Note: In Equation 1.51, the argument of exp should be* $(-\epsilon^2/\sigma^2)$. *Correction in Q1.2, perpendicular to one of the* ***faces*** *not the edge.*

2. **Logistic Regression (5 points)**

   Logistic regression is a binary classification method. Intuitively, logistic regression can be conceptualized as a single neuron reading in a d-dimensional input vector $x \in \mathbb{R}^d$ and producing an output y between 0 and 1 that is the system's estimate of the conditional probability that the input is in some target category. The "neuron" is parameterized by a weight vector $w \in \mathbb{R}^{d+1}$, where w_0 represents the bias term (a weight from a unit that has a constant value of 1).

   Consider the following model parametrized by w:

$$y = P(\mathcal{C}_1|x) = \frac{1}{1+\exp(-w^\top x)} = g(w^\top x) \tag{1}$$

$$P(\mathcal{C}_0|x) = 1 - P(\mathcal{C}_1|x) = 1 - y, \tag{2}$$

   where we assume that x has been augmented by a leading 1 to represent the bias input. With the model so defined, we now define the Cross-Entropy cost function, equation 3, the quantity we want to minimize over our training examples:

$$E(w) = -\sum_{n=1}^{N} \left\{ t^n \ln(y^n) + (1-t^n)\ln(1-y^n) \right\}. \tag{3}$$

   Here, $t^n \in \{0,1\}$ is the label (or target or or teaching signal) for example n ($t^n = 1$ represents $x^n \in \mathcal{C}_1$). We minimize this cost function via gradient descent.

   To do so, we need to derive the gradient of the cost function with respect to the parameters w_j. Assuming we use the logistic activation function g as in equation 1, prove that this gradient is:

$$-\frac{\partial E(w)}{\partial w_j} = \sum_{n=1}^{N} (t^n - y^n) x_j^n \tag{4}$$

Part II
Programming Assignment

Requirements: Write your own Logistic Regression and Softmax Regression classifiers using Python, following the instructions below.
You **are allowed** to use the following Python libraries and packages: NumPy, PIL, matplotlib, os, and random.
You **are not allowed** to use any SciPy implementations or logistic or softmax regression or any other high-level machine learning packages (including, but not limited to TensorFlow, PyTorch, Keras, etc.).

Note: For this and future assignments it will be required for you to find a team to join. You may as well figure that out now!

For this assignment, a "team" is defined as two or three people.

1. **Load and preprocess the data (5 points).**
   Dataset: In this problem, we will use logistic regression to separate types of traffic signs (for example, discriminate yield signs from stop signs). We will use traffic signs derived from the German Traffic Sign Recognition Benchmark here. We are providing you with two sets of data from that dataset: **unaligned** and **aligned**. The **unaligned** set contains traffic signs that may not appear in the centers of images. This is provided to show you what happens with PCA when the data is not aligned. The **aligned** set has been preprocessed to align the traffic signs. The 43 traffic signs included in the portions of the dataset we are providing you are: Speed limit (20 km/h), No passing, Yield, *etc.* A zip version of these datasets has been uploaded to Piazza under "Resources", as well as a starter code and a basic dataloader: the output of the dataloader is stored as a Python tuple with NumPy arrays for the *images* and *labels* (ranging from 0 to 42) associated to the traffic sign. You don't have to use this function, you may write your own.

2. **Cross Validation Procedure (5 pts for a correct implementation).**
   We will use the method of *k-fold cross-validation* to estimate our model's performance on unseen data. For each problem below, you should divide the data into k mutually exclusive sets. Each set should contain a representative sample with respect to the problem. For example, we will have you recognize Speed limit (100km/h) and Speed limit (120km/h) signs. In this case, it would not be good to have one of the sets contain all 100km/h signs or all 120km/h signs. Basically, you want to have the proportion of each category in each of the k sets to be similar to the original training set. Don't worry if they don't divide up *perfectly* equally; but make sure they are mutually exclusive! This is a common error - some researchers have mistakenly included some of the training data in the test data! This is Very Bad. Don't do it!

   Repeat k times: Choose two of the sets to be holdout (or validation) and test sets. Then, train your model on the remaining $k-2$ of the sets. Each of the sets should be the test set once and the holdout set once. Psuedocode for this is in the Training algorithm below.

   We use the holdout set in a special way: It is used to check whether your model is overfitting on the training set. The error on the holdout set should go down as you train the model, even though it is not being used to change the weights. However, at some point, the holdout error should start to rise. For this problem, in our experiments, this never happens, so perhaps this problem is too easy to display overfitting. This means the model is overfitting and you should stop training. This is called *early stopping.* At that point, you run the trained model on the unseen test set, and record the performance.

   It is possible that the holdout set error never goes up over the M epochs. This will happen if the problem is too easy or if the holdout set is too easy. So you should also have some limit on the number of training steps, say 100 passes through the complete training data. One pass through all of the training data is called an *epoch.*

3. **Principal Components Analysis (5 pts for a correct implementation).** Below, we ask you to perform dimensionality reduction on the images first, by using a linear dimensionality reduction technique called Principal Components Analysis, or PCA to its friends. Note also, it is PrincipAL Components Analysis, *not* PrincipLE Components Analysis. I personally hate that typo! You should have realized that the role of the holdout set is to stand in for the unseen test set, to avoid overfitting. Hence, the PCA in each case *should only be performed on the training set.* In real life, we would not have access to the test set. Hence we would have to use the PCA we used for our data on that new data to reduce its dimensionality. So, we should also treat the holdout set the same way. We will apply the PCA computed on the training set to the holdout and test sets without change.

   N.B. After projecting the data onto the principal components, you should divide by the standard deviation of the projections, which is the square root of the eigenvalue. The resulting projections - the inputs to the network - should then be zero mean and unit standard deviation. This is easy to check, so please check it. Why do you think this is a good idea?

   Hence, you will be doing PCA many times, so we should do it as efficiently as possible! It is ok to use an eigenvalue/eigenvector routine such as `numpy.linalg.eigh`. Try keeping three different numbers of components (e.g., when training on all traffic sign types, try a few hundred or larger; when training on only two traffic sign types, try a much smaller number), and report your results. Figure 1 shows the top four principal components as images for a subset that contains 50km/h and 60km/h signs.

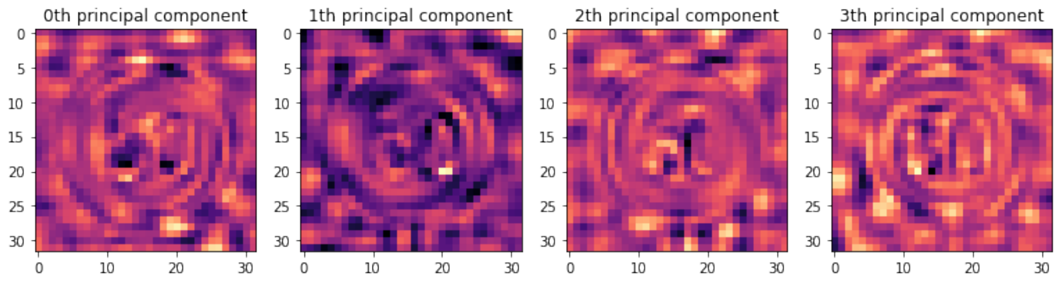

Figure 1: An example of top 4 principal components for binary classification of Speed limit (50km/h) vs Speed limit (60km/h) (aligned dataset).

4. **What to report for each experiment (except where noted) Loss and Performance Plots (10 pts)**

   As described above, you need to divide the dataset into a training set, a validation (or hold-out) set, and a test set, k times. (When you are just developing your model to see if it works, though, you can just use one training/holdout/test split until you are sure your code is working.)

   For each question in this assignment, we ask you to repeat your experiment with $k = 10$, so we will be using 10-fold cross validation. During your training process, you want to update your model parameters with the training set, and save the parameters of the model with the lowest validation loss as your best model, which you then use to measure performance on the test set. This is described in the "Training Procedure" below.

   In order to give you insight into the process (and check that your code is working), we want you to plot the average training and holdout error, and its standard deviation, at each epoch. The average is over the 10 runs of cross-validation. Since the holdout error should be lowest at different points during training with different folds of the data, pick some maximum number of epochs M for all of the runs and run all models for that maximum number. Here we suggest M=300.

   We also ask you to report your final performance in the text of your report. This should not be based on your best performance on the test set! That's called "Data Snooping", and is also Very Bad, don't do it! What we mean by final performance is the average over the 10 runs, but where in each of the 10 runs, you record the error on the test set *when the error on the holdout set is lowest.* It is this test set error at that point in each run that should be averaged when reporting your final performance. Here by "error", we mean the loss, i.e., the cross-entropy. In order to normalize this over datasets with different numbers of examples, and different numbers of outputs, you should report the average loss over the number of examples, N, and the number of outputs, c.

The error isn't a particularly intuitive measure of performance, except we do want it to be low. Hence, we *also* want a plots of percent correct on each set. The percent correct is determined by recording how often the model would choose the correct answer for an image.

To be clear, then, each experiment with one setting of the hyperparameters (learning rate and number of principal components) will result in *two* plots: a plot of the average loss and a plot of the performance over the 300 training epochs for the training set and the holdout set. These training and holdout curves should be easily distinguishable in your plots by using different colors or solid and dashed lines. We also want you to plot the standard deviation of your result as error bars for every 50 epochs (i.e., at 50, 100, ... epochs). See this page for an example of a plot with error bars.

```
1: procedure TRAINING PROCEDURE
2:     folds = $k$ mutex split of training data;
3:     for fold = 1 to $k$ do
4:         val, test ← folds[fold], folds[(fold+1) mod k];
5:         train ← remaining folds;
6:         Perform PCA on train;
7:         Project all data onto top $p$ train PC's, scaled as in text.
8:         for epoch = 1 to $M$ do
9:             train the model with train, and test the performance on val every epoch
10:            record test,val loss and accuracy from each epoch (for plotting)
11:            save the best model based on val performance
12:        use the best model to record loss and accuracy on test
13:    average test loss and accuracy from all $k$ trials;          ▷ Put in a table in your report
14:    plot the average and standard deviation of training and validation curves
```

5. **Logistic Regression (25 points)**
Here, we build and evaluate classifiers on the data. We will experiment with discerning between two classes of data. For this problem, train for $M = 50$ epochs as opposed to 300 epochs.

Algorithm 1 Two Approaches to Gradient Descent

```
1: procedure BATCH GRADIENT DESCENT
2:     $w \leftarrow 0$
3:     for t = 1 to $M$ do                                  ▷ Here, t is one epoch.
4:         $w_{t+1} = w_t - \alpha \sum_{n=1}^{N} \nabla E^n(w)$
5:     return $w$
1: procedure STOCHASTIC GRADIENT DESCENT
2:     $w \leftarrow 0$
3:     for t = 1 to $M$ do                                  ▷ Here, t is one epoch.
4:         randomize the order of the indices into the training set
5:         for n=1, . . . , N do
6:             $w_{t+1} = w_t - \alpha \nabla E^n(w)$
7:     return $w$
```

(a) **Implement Logistic Regression via Gradient Descent. (5 points)**
Now, without using any high-level machine learning libraries, implement logistic regression. Here, you'll be using *batch gradient descent*, and will only need one logistic output unit. (Think about why we only need one if we're classifying two classes?) The most points will be given for clean, well-documented code.

(b) **Evaluate the model on Speed limit 100km/h (class 7) vs Speed limit 120km/h (class 8) using the unaligned dataset.** *(5 points)*
Here we ask you to perform logistic regression on the 100km/h (class 7) and 120km/h (class 8) signs from the unaligned dataset.

For this experiment only, you don't need to do cross-validation! Just do one run with a 80/10/10 percent split of the data. Then plot the usual curves and report the percent correct on your test set. Since there is only one run, you don't need to plot standard deviation, as there won't be any! Your results should be poor. We achieved about 90% accuracy. Why is that? To understand why, present a Figure showing the first 4 principal components plotted as images.

(c) **Evaluate on Speed limit 100km/h (class 7) vs Speed limit 120km/h (class 8) on the aligned dataset.** *(10 points)*

Here, repeat the above experiment, *using 10-fold cross-validation*, on the *aligned* dataset.

i. As described above, plot the average loss curves over the 10 runs for the training and holdout sets, including standard deviation error bars. If the error blows up, your learning rate is too high.
Use early stopping based on the holdout set error. I.e., as described above, save the weights any time the holdout set error is less than it has been before. If it goes back up again, you still have the best weights. However, we have found that it doesn't go up again for this data.
For *one* of the principal components calculation, present a figure with the first four principal components presented as images. This should look different than the ones from the unaligned case!
Make sure that your graph is well-labeled (i.e., x and y axes labeled with what they are) with a key, and with a figure caption that clearly states what is being shown in the graph. This should be the case for any graph you plot.

ii. Report the test accuracy after training. Here, by "accuracy", we mean the percent correct when choosing the category according to the rule above (> 0.5 = "100km/h"). Again, this should be an average over the 10 runs with the standard deviation in parentheses (e.g., like this: 94.4% (1.2). Our test accuracy was about 98%.

iii. Repeat the above for 3 different learning rates (i.e., 3 total). Try and find one that's too high, one that's too small, and one that's just right. Plot the training set error (cross-entropy loss) for the three different learning rates on one plot, including the standard deviation as above. For your report above, use the best learning rate you found.

(d) **Evaluate on Dangerous curve to the left (class 19) vs Dangerous curve to the right (class 20) on the aligned dataset.** *(5 points)*

i. Plot the cross-entropy loss over training epochs averaged over the ten runs. Use the best learning rate you found above.

ii. Report the test accuracy averaged over the 10 runs.

iii. Does this differ from what we observed above? If so, why do you think that is?

6. **Implement Softmax Regression via Gradient Descent.** *(25 points + 5 points)*
In this part, we will experiment with multi-class classification.

Softmax regression is the generalization of logistic regression for multiple (c) classes. Now given an input x^n, softmax regression will output a vector y^n, where each element, y_k^n represents the probability that x^n is in class k.

$$y_k^n = \frac{exp(a_k^n)}{\sum_{k'} exp(a_{k'}^n)} \tag{5}$$

$$a_k^n = w_k^T x^n \tag{6}$$

Here, a_k^n is called the *net input* to output unit y_k. Equation 5 is called the *softmax activation function*, and it is a generalization of the logistic activation function. For softmax regression, we use a *one hot encoding* of the targets. That is, the targets are a c-dimensional vector, where the k^{th} element for example n (written t_k^n) is 1 if the input is from category k, and 0 otherwise. Note each output has its own weight vector w_k. With our model defined, we now define the *cross-entropy* cost function for multiple categories in Equation 7:

$$E = -\sum_n \sum_{k=1}^{c} t_k^n \ln y_k^n \tag{7}$$

Again, taking the average of this over the number of training examples normalizes this error over different training set sizes. Also averaging over the number of categories c makes it independent of the number of categories. Please take the average over both when reporting results. Surprisingly, it turns out that the learning rule for softmax regression is basically the same as the one for logistic regression! The gradient is:

$$-\frac{\partial E^n(w)}{\partial w_{jk}} = (t_k^n - y_k^n)x_j^n \tag{8}$$

where w_{jk} is the weight from the j^{th} input to the k^{th} output.

Now, we'll modify our network to classify all 43 traffic sign categories. To achieve multi-class classification, we'll need more output units, and use the gradient derived for Softmax Regression. As a sanity check, make sure your outputs are all positive and sum to 1. You will be using one-hot encoding of the targets here. When choosing the category for evaluating percent correct, you just choose the maximum output (as in Lecture 1).

(a) **Evaluate your network on all 43 traffic signs (aligned dataset).** *(15 points)*
Follow the same instructions as for Speed limit 100km/h vs Speed limit 120km/h - repeat this ten times using a different subject for the test set each time, use early stopping with a hold-out set, etc., plotting the standard deviation at 50, 100, 150, 200, 250, and 300 epochs.

  i. Plot the loss on the training set as well as the loss on the holdout set vs. number of epochs of gradient descent. In a case like this, we naturally find the maximum output, say y_i^n for input pattern x^n and choose the i^{th} category as the system's choice. Again, plot the average over 10 runs. We achieved a test accuracy of 90%.

  ii. Now, repeat part i, **without** PCA on the **aligned** dataset. Next, repeat part i, **with** PCA on the **unaligned** dataset. Compare these results against part i and comment on the importance of data preprocessing (i.e., PCA and aligning).

  iii. Create a table that is a 43×43 confusion matrix for this data based on the test set results. The class label should be listed along the top and left side of the matrix. The confusion matrix will have 43^2 entries, C_{ij}, where i is the correct traffic sign and j is the traffic sign chosen by the network, and C_{ij} is the percent time that j was chosen for i. Hence the rows should add to 100%. A perfect system will have 100% down the diagonal and 0's everywhere else. Your system will most likely not be perfect!

(b) **Batch versus stochastic gradient descent.** *(5 points)*

  i. Using your softmax network, implement the stochastic gradient descent algorithm above. Note you now have another `for` loop inside the "epoch" `for` loop. To randomize the patterns, use an indirect indexing method, i.e., have an integer array of N indices, and permute the order of that array. Let's call this array P. So initially, $P[i] = i$. Then on each epoch, you simply permute the integers in P. Let's call the $N \times d$ matrix of input patterns $Input$. For each epoch, as i goes from 1 to N on the innermost loop, we train on the input $Input[P[i]]$,instead of $Input[i]$.

  ii. Plot the training set loss over training epochs, with one curve for the batch mode above, and one for stochastic gradient descent. Is stochastic gradient descent faster, in terms of minimizing the error in fewer epochs? Explain your result.

(c) **Visualize the weights (5 points).**
Since there are as many weights as there are pixels, we can visualize the weights. Using one of your trained networks that was trained **without PCA**, transform the learned weights for each emotion into the range 0-256 (ignore the bias weight). That is, simply linearly scale them so the minimum weight for each traffic sign is 0 and the maximum is 256. (This will be different for each traffic sign). Then, visualize these weights as an image. What do you see? Explain why you end up seeing this. Randomly choose **four** categories and include the **four** images in your report as a well-labeled figure.

(d) **Extra Credit (5 points).**
The dataset we used for model training is imbalanced, i.e., each category does not contain the same number of instances. See Figure 2 for illustration. Can you improve the accuracy of a softmax regression using techniques that handle the imbalanced nature of the dataset?

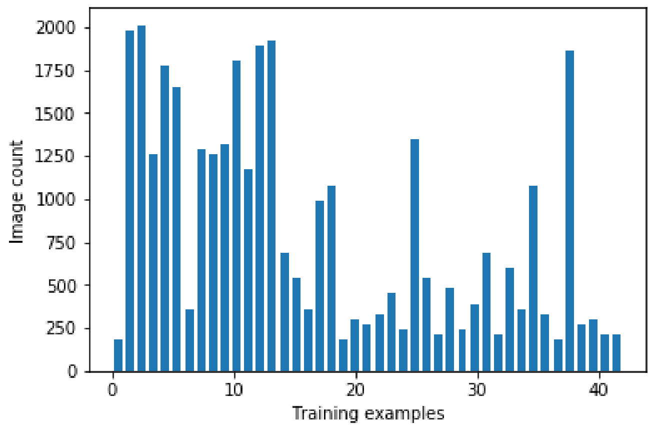

Figure 2: Histogram of image count for categories in the training set.

7. **Individual Contributions to the Project**
   Don't forget to include a paragraph at the end of your report, one per team member, describing what that team member's contribution to the project was.

Supplementary Section: Principal Component Analysis
In this programming assignment, we are asking you to perform principal component analysis (PCA) as a dimension reduction technique on your training images. Here are a few more hints on how to implement your PCA. Have the Turk & Pentland article (posted under resources) in hand, as I am going to refer to it a lot in what follows.
On page 74 (page 4 of the pdf), lower left side, they start to explain PCA. They explain it the "normal way" first (a method that results in a huge matrix - the way *not* to do it), then they explain what we used to call "the Turk Pentland trick", although it's been known to mathematicians for about 70 years. I didn't explain this before because most PCA routines automatically choose the smaller dimension (using the notation in the article), either M, the number of images, or d, the dimension of the images (number of pixels). In the example in the paper, they are assuming the images are $N \times N$, so $d = N^2$

1. The first step in either method is to subtract the mean image from every image. This is described in the first four lines of the paragraph starting with "Let the training set of faces images be..." The Φ_i's are the centered data.

2. Equation 3 shows constructing the covariance matrix. In this equation, the Φ_n's are column vectors, so this is the outer product of the two vectors, resulting in an $N^2 \times N^2$ matrix. They then rewrite this as AA^T, where A is a matrix composed of the normalized images as column vectors (although this is missing a factor of $1/M$, you will get the same eigenvectors, just longer). It is the eigenvectors of this matrix that are the principal component vectors.

3. Now, since there can't be more than M-1 eigenvectors, as one point in this high dimensional space (i.e., one image), can't have a principal component as it is just a point, but 2 points can, etc. The "Turk and Pentland trick" is shown in equation 4, where instead of AA^T, which is $N^2 \times N^2$, they form the matrix $A^T A$, which is $M \times M$, a much smaller matrix. [A is $d \times M$, so A^T is $M \times d$ (where d is the number of pixels in the images), so $A^T A$ is a $M \times d$ matrix times an $d \times M$ matrix, so $A^T A$ is $M \times M$.] So if you have 10 images, this matrix will be 10×10. They find the eigenvectors of that matrix (v_i), i.e., vectors such that $A^T A v_i = \lambda_i v_i$, where λ_i is the eigenvalue.

4. Equation 5 shows that the Av_i's are actually the eigenvectors of the original huge matrix C. So, to find the eigenvectors of that huge matrix C, you never actually form that matrix. Instead, you form the matrix $A^T A$, find the eigenvectors of that, and then the Av_i's are the eigenvectors of C. Now, A is $d \times M$, v_i is $M \times 1$,

so Av_i is $d \times 1$, i.e., it is a column vector of the same dimension as the images when they are treated as a long column vector. Note: You need to sort the eigenvectors by their eigenvalues: the eigenvector with the largest eigenvalue is the first principal component, the one with next largest eigenvalue is the second principal component, etc.

Sanity Check:

1. Project each (centered) image onto the first principal component, i.e., you compute this:

   $\Phi_i^T Av_1/||Av_1||$, which is a scalar, for each image Φ_i. (the "1" refers to the fact that this is the eigenvector with the biggest eigenvalue, i.e., the first principal component).

2. If you do this for all M images used to compute the principal components, you will get a bunch of scalars whose average is 0, and whose standard deviation is $\lambda_1^{0.5}$. If you don't get this, something is wrong.

3. Now, you divide all of these numbers by $\lambda_1^{0.5}$, and you will have a set of numbers whose mean is 0 and whose standard deviation is 1. Obviously, if you first compute $v_1^* = Av_1/(||Av_1||\lambda_1^{0.5})$, you just have to form the inner products of the centered images with v_1^*. This is the second sanity check.

4. Repeat the above for the first k principal components, and you will have a vector of k small numbers representing each image, and this is what you use to train the logistic and softmax regression models.

5. For the holdout and test sets, you should first subtract the mean of the *training set images*, and then project them onto the v_i^* vectors in the same manner. Now the mean of the projections (the scalars you get out) over the test and hold out set will no longer be 0, but they should be close.