Exercise 3: Image Classification

Due: 20.12.2018

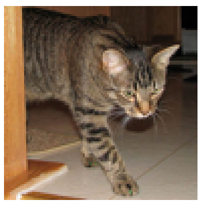
Please note that Exercise 3 is long and may require all five sessions to complete. Don't leave it all to the last day and don't be afraid to ask for help from the assistants!

1 Bag-of-Words Classification with Histograms of Oriented Gradients

In a previous exercise, you explored the detection of interest points in an image, e.g. corners or edges, which are relevant for the extraction of local features. In this exercise, you will build two complete Computer Vision pipelines which comprise both the initial low-level feature extraction and the subsequent high-level representation of images from low-level features, so that each image is classified according to its semantic content. The first part of the exercise focuses on a **bag-of-words (BoW) classifier** using hand-crafted features and a simple nonparametric nearest neighbor classification principle. On the other hand, the second part of the exercise examines a **convolutional neural network (CNN) classifier** with implicitly learned features in an end-to-end architecture.

1.1 STL-10 Dataset

For the purposes of both parts of this exercise, we will use the *STL-10* dataset: `https://cs.stanford.edu/~acoates/stl10`, and in particular the labeled part of it. *STL-10* contains 96×96 color images that belong to one of the following ten semantic classes: *airplane*, *bird*, *car*, *cat*, *deer*, *dog*, *horse*, *monkey*, *ship*, and *truck*. It contains a training set with 500 images from each class (5000 images in total) and a test set with 800 images from each class (8000 images in total). The "universal" supervised learning protocol, which also applies to classification, states that the training set is used to optimize the values of any parameters or hyperparameters of the classification pipeline for maximum accuracy, whereas the test set is used for the final evaluation of the classification pipeline using the aforementioned optimized values. For this part of the exercise, *STL-10* will be input in the form of distinct image files organized in directories according to their semantic class. Download the zip file `STL-10.zip` from the course webpage (see section Exercises–Extra material link) and unzip it. You should see the file structure `images_per_class/{train,test}/`, followed by class-specific directories. In the provided Python and Jupyter code, you should specify exactly the full path to directory `images_per_class` as the root directory of the dataset. Briefly inspect some images to get a visual overview of *STL-10*. Two sample images are shown in Figure 1.


(a) *cat*

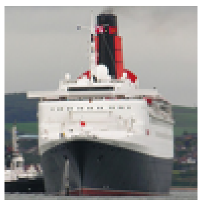

(b) *ship*

Figure 1: Sample images from the *STL-10* dataset

1.2 Bag-of-Words Classification Pipeline Overview

Before analyzing each individual component of the first classification pipeline, i.e. bag-of-words, we first provide you with a general overview of it, so that you can identify each code module that you need to implement or complete with the corresponding part of the pipeline. Figure 2 provides a visual presentation of the BoW pipeline using **Histograms of Oriented Gradients (HOG)** as low-level features and a simple nearest neighbor classification principle.

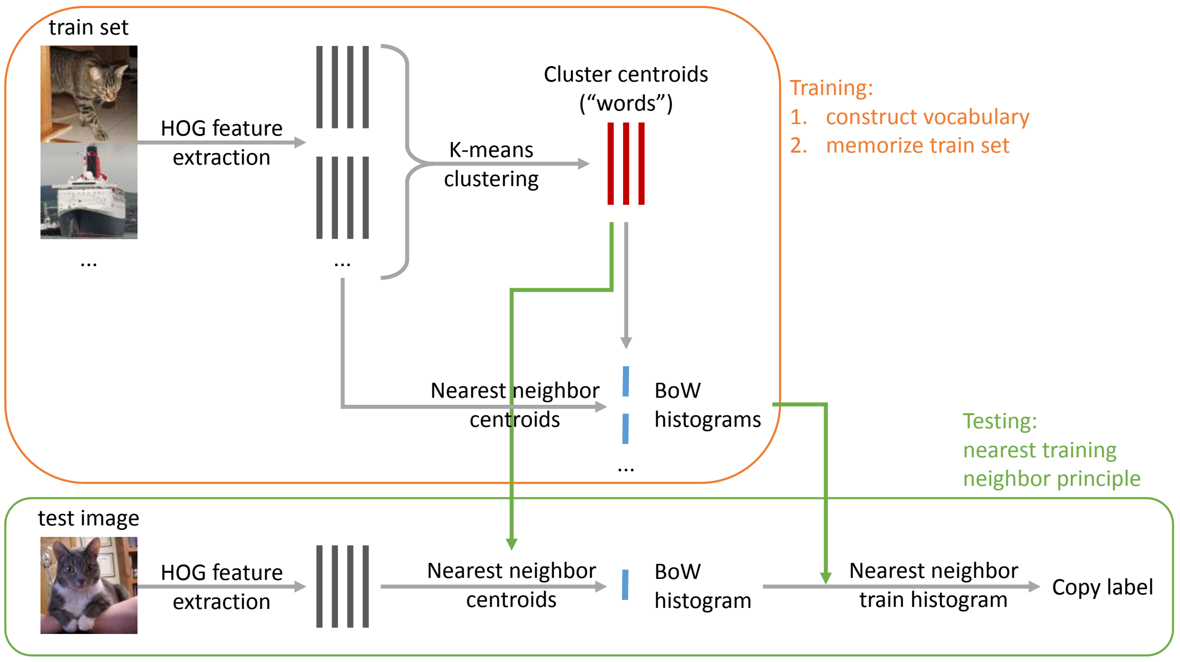


Figure 2: Overview of BoW pipeline with HOG features and nearest neighbor classifier

At training, HOG features are extracted densely from each image. K-means clustering is applied to the set of extracted features from the entire training set, resulting in K centroids which serve as the **visual words**. Each training image is finally represented as a bag of such words in the form of a BoW histogram vector, based on the proximity of its HOG feature vectors with the words.

At testing, the aforementioned steps for computing a BoW histogram are applied to each test image. The final classifier is nonparametric and it is based on the **nearest neighbor** principle: the label of the nearest neighbor of the test image in the training set (based on BoW histogram representation of images) is predicted to be the label of the test image. More formally, denote the training set as a set of pairs of BoW histograms and class labels

$$\mathcal{T} = \{(\mathbf{h}_i, c_i) : i = 1, \ldots, T, \; c_i \in \{1, \ldots, C\}\}, \tag{1}$$

where C is the total number of semantic classes we consider. For a test image that is represented as a BoW histogram $\mathbf{g}$, the nearest neighbor classifier predicts the label

$$\hat{c} = c_{\hat{k}}, \; \text{where } \hat{k} = \arg\min_{k \in \{1, \ldots, T\}} \{d(\mathbf{g}, \mathbf{h}_k)\}\,. \tag{2}$$

In (2), $d(\cdot, \cdot)$ denotes some distance metric between two BoW histograms.

This decision rule has certain implications for the efficiency of each individual prediction of the classifier. In particular, suppose that we do not use any approximation technique (e.g. K-D tree) to determine the nearest neighbor in the training set, but compute all required distances exactly. What is the space and time complexity of predicting the label of a single test image with respect to the size of the training set?
Both the space and time complexity of a single prediction is linear in the size of the training set, i.e. $O(T)$.

1.3 Feature Description with Histograms of Oriented Gradients (HOG)

Your **first task** is to implement the feature extraction step for an individual image at a dense grid of points using histograms of oriented gradients as feature descriptors for each point of the grid. This operation is performed via the function `feature_extraction`, which is provided to you and outlines the two main parts of feature extraction.

First, a dense regular grid of `n_points_x`×`n_points_y` points is defined on the image via the function `grid_of_feature_points`, which you are asked to implement. The grid should not extend to the edges of the image; rather, it should leave a margin of `margin_x` pixels from the left and right edge and `margin_y` pixels from the top and bottom edge. We need this margin because the HOG descriptor for each grid point is computed on a square patch centered at the point. Function `grid_of_feature_points` should output the column (x) indices and the corresponding row (y) indices of the grid points as two separate 1D NumPy arrays of length `n_points_x`×`n_points_y`. Useful functions: `numpy.linspace`, `numpy.meshgrid`.

Second, HOG feature descriptors are computed for all grid points from the previous step via the function `compute_HOG_descriptors`, which you are asked to implement. For each point, the HOG feature description operates on a square image patch centered at the point and partitioned into a 4×4 set of cells. Each cell has a size of `cell_height`×`cell_width` pixels. In our case, `cell_height`=`cell_width`= 4, so the entire square patch has a size of 16×16 pixels. Compute the approximate image gradient using the Sobel operator and get the direction of the gradient as a angle in the interval $[-\pi, \pi]$. For each grid point, compute the 8-bin histogram of gradient directions at pixels that belong to each separate cell of the corresponding patch and concatenate the histograms from all the cells into the final 128-D HOG descriptor for the point. Function `compute_HOG_descriptors` should output a 2D NumPy array of size `n_points`×128, where `n_points` is the total number of grid points defined on the image. Useful functions: `scipy.ndimage.sobel`, `numpy.histogram`.

1.4 Visual Vocabulary Construction with K-means Clustering

Your **second task** is to implement the construction of a vocabulary (or codebook) of K visual words by applying **K-means clustering** to the complete set of HOG descriptors from all images in the training set. The centroids which are computed with K-means serve as the codebook words. In particular, you are asked to implement the function `kmeans`, which executes the K-means algorithm. Perform random initialization of the cluster centroids with data points and implement the main K-means loop. At each iteration of the loop, you need to implement the assignment step and the update step. First, assign each data point to that cluster whose centroid is nearest to the point with respect to the Euclidean distance. Implement the function `find_nearest_neighbor_L2` to perform this assignment, i.e. compute the index of the nearest centroid for each data point. Second, update each cluster centroid to the mean of all data points that are currently assigned to the respective cluster. For the update step, you also need to consider the case of an empty cluster, i.e. a cluster left without any assigned points at some iteration, and randomly re-initialize its centroid with a data point. Function `kmeans` should output a 2D NumPy array of size K×128, containing the final cluster centroids as its rows. Useful functions: `numpy.random.permutation`, `scipy.spatial.distance.cdist`.

1.5 BoW Histogram Representation

The codebook is used to compute the BoW histogram for each image, which serves as a high-level representation of the image that can be directly used as input to a classifier. This operation is performed by the function `bow_histograms_and_labels` for all images belonging to a set (training or test), along with creation of an array containing the ground truth labels of these images. We provide you with the skeleton of this function and your **third task** is to fill in its inner `for` loop over all images that belong to a certain class, so that at each iteration the BoW histogram of the corresponding image is computed. The BoW histogram is a K-bin histogram, where the k-th bin counts how many HOG descriptors of the input image are closest to the k-th codebook word (centroid). For this task, you may reuse the function `find_nearest_neighbor_L2` that you implemented for the previous task. Useful functions: `numpy.bincount`.

1.6 Nearest Neighbor Classifier

BoW histograms are computed for all images both in the training and the test set. Classification of test images is performed by using the nearest training neighbor principle that we reviewed previously in (2). Your **fourth task** is to implement this classifier via the function `nearest_neighbor_classifier`, using the Euclidean (L_2) norm to measure distances between BoW histograms. You may reuse the function `find_nearest_neighbor_L2` that you implemented previously. Function `nearest_neighbor_classifier` should output a 1D NumPy array containing the predicted labels for the corresponding test BoW histograms, copying the respective labels of their nearest training neighbors.

1.6.1 Theoretical Question: Norm Selection for Histograms

Consider the space that BoW histograms which are defined as we described above belong to. In particular, express all constraints on the elements g_i of a BoW histogram $\mathbf{g}$, $i \in \{1, \ldots, K\}$. Based on the space that is implied from these constraints, rethink the distance metric that is more appropriate to measure distances between BoW histograms. More specifically, find a minimal example with three such histograms $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$ in which $d(\mathbf{g}_1, \mathbf{g}_2)$ should be equal to $d(\mathbf{g}_1, \mathbf{g}_3)$, but the usage of the L_2 norm results in $d(\mathbf{g}_1, \mathbf{g}_2) \neq d(\mathbf{g}_1, \mathbf{g}_3)$. Which norm would assign equal distance to the two pairs?
Each BoW histogram $\mathbf{g}$ is a K-dimensional vector which expresses how many instances of each visual word i, $i \in \{1, \ldots, K\}$ are contained in the respective image. If the employed dense grid for feature description comprises W points, then the elements of the histogram must add up to W. More formally,

$$\sum_{i=1}^{K} g_i = W,$$
$$g_i \geq 0, \forall i \in \{1, \ldots, K\}.$$

These constraints imply that the space which encompasses the BoW histograms is a $(K-1)$-simplex. Considering that each individual change of assignment of a feature vector to a visual word should be penalized equally with the selected norm, a more appropriate choice than L_2 is the L_1 norm. An indicative example is taking $K = 4$, $W = 3$, $\mathbf{g}_1 = (2, 0, 1, 0)$, $\mathbf{g}_2 = (1, 1, 0, 1)$, $\mathbf{g}_3 = (0, 2, 1, 0)$. Going from $\mathbf{g}_1$ to either $\mathbf{g}_2$ or $\mathbf{g}_3$ requires at least two assignment changes. This is reflected by using the L_1 norm, as $\|\mathbf{g}_1 - \mathbf{g}_2\|_1 = \|\mathbf{g}_1 - \mathbf{g}_3\|_1 = 4$. On the contrary, using the L_2 norm does not treat the two pairs equally, as $\|\mathbf{g}_1 - \mathbf{g}_2\|_2^2 = 4 \neq \|\mathbf{g}_1 - \mathbf{g}_3\|_2^2 = 8$.

1.7 Experiments

- The provided functions `confusion_matrix` and `accuracy_from_confusion_matrix` measure the performance of the constructed image classification pipeline based on the test set predictions. Use the default values provided in the template implementation (in particular, set the number of K-means clusters K to 100, the number of K-means iterations to 10 and the number of rounds of training with K-means and subsequent evaluation to 10) and firstly run the pipeline for only two classes, namely *cat* and *ship*. Measure the reported accuracy on the test set. Why are multiple rounds of training and evaluation necessary when reporting accuracy?

  Since K-means initialization is random, the resulting cluster centroids are also random and the same holds for the obtained BoW histograms. Thus, we must perform multiple rounds of training and evaluation and report the mean accuracy (and standard deviation) over all rounds, since it has a reduced variance compared to the result from a single round.

- Secondly, run the pipeline on the full *STL-10* dataset, i.e. for all ten classes, with the same default values as before. Measure the reported accuracy on the test set and compare it to the previous case with only two classes. Is the figure for ten classes expected to be higher or lower, and why? Relate to the performance of a random classifier in the two cases.

The accuracy of the pipeline for ten classes is expected to drop. A random classifier has an expected accuracy of 50% for two classes, but only 10% for ten classes.

- Thirdly, implement an alternative function to `find_nearest_neighbor_L2` to determine nearest neighbors via the norm of your answer in Section 1.6.1. Use this function alternatively to `find_nearest_neighbor_L2` inside the function `nearest_neighbor_classifier` and repeat the classification experiments for two and ten classes using the new classifier. Compare the performance to the L_2 norm baseline.

  Using the L_1 norm for the nearest neighbor classifier gives a small consistent improvement over the L_2 norm. For $K = 100$, the improvement in the mean accuracy in our experiments was ca. 1% both for two and ten classes.

- Finally, vary the number K of K-means clusters that are used to determine the visual words and assess the impact on the performance. More specifically, repeat the initial experiment with two classes and L_2 norm for nearest neighbors, but evaluate the accuracy of the classification pipeline for $K \in \{50, 100, 200, 400\}$.

  The impact of varying K on the accuracy of the system is small in the examined setting. Performance deteriorates when choosing too low or too high K.

2 Image Classification with Convolutional Neural Networks

Before starting to work on this part of the exercise, please activate the conda environment named 'ex3' by running in the terminal →
source activate ex3
This will setup the libraries required for running the code. Also, you will have to unpack the STL-10 dataset again by running →
tar zxf `/home/cvcourse/public_html/pics/stl10_binary.tar.gz`
in the same directory as the code files of the exercise.

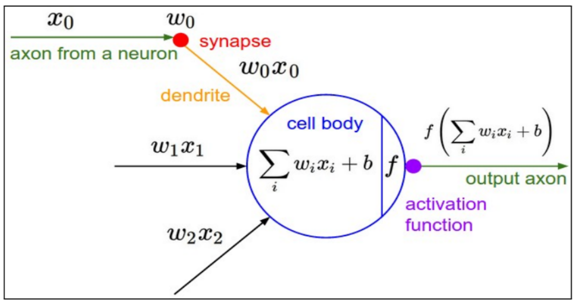
We now consider another approach for image classification: convolutional neural networks (CNNs). Most of the code has been implemented already. You are asked to fill out the missing parts of the implementation (described in Sec. 2.2) and train the model on the STL-10 dataset. The idea of this part of the exercise is for you to get familiar with techniques that are currently in vogue in computer vision.

2.1 CNN Basics

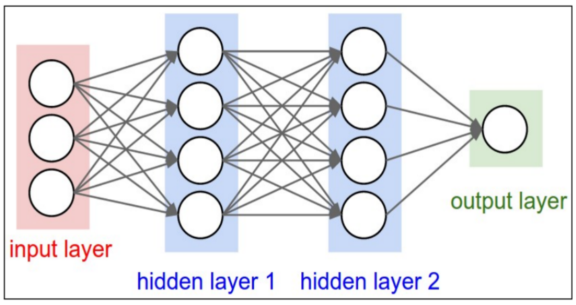
In this sub-section, we will provide a very brief overview of some of the important components of the learning pipeline involved while working with CNNs. The description of some of the components is left incomplete for you to fill out.

2.1.1 Neural Network Basics

- A **Perceptron** (Fig. 3a) is a simple computational model of a biological neuron. It computes a linear combination of its inputs, with learnable parameters w_i and b. This is then passed through a non-linearity (also known as **an activation function**) to form the perceptron's output.
- **Multi-Layered Perceptrons (MLPs)** (also known as Feed-Forward Neural Networks) are shown in Fig. 3b. Each layer in a MLP is typically **fully-connected** i.e. every unit of a given layer is connected to every unit of the next layer. In each layer, units compute a linear combination of their inputs, followed by a non-linear **activation function**. Typical choices for the activation function are the sigmoid function, the hyperbolic tangent function and the rectified linear unit (ReLU) (shown in Fig. 4 b,c,d respectively). The layers, apart from the input and the output layer, are called hidden layers.



(a)



(b)

Figure 3: (a) Perceptron - mathematical model of a neuron, the basic computational unit in the biological brain. (b) A Multi-layered perceptron with two hidden layers. Courtesy of [CS231n Convolutional Neural Networks for Visual Recognition - http://cs231n.github.io/neural-networks-1/].

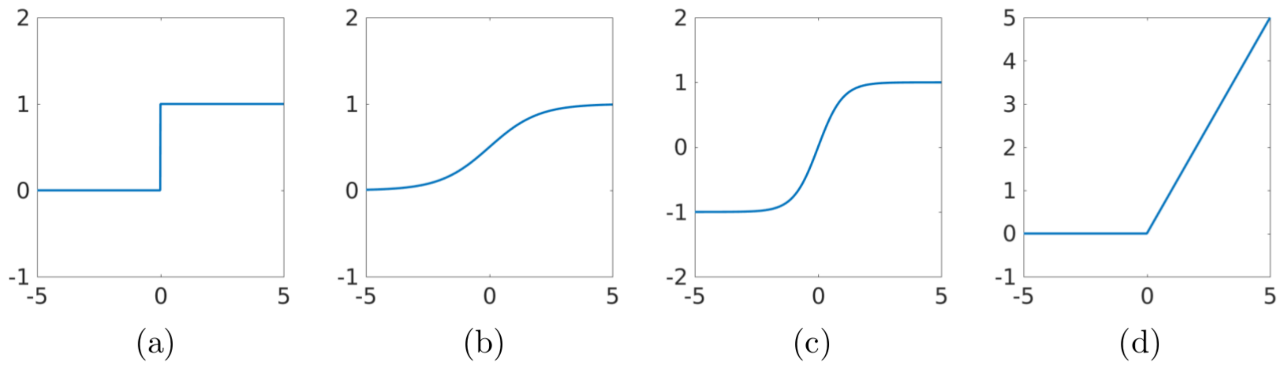


Figure 4: Commonly used activation functions. (a) Heavy-side / step function, (b) sigmoid ($f(x) = 1/(1+\exp(-x))$), (c) tanh ($f(x) = (1-\exp(-2x))/(1+\exp(-2x))$), (d) ReLU ($f(x) = max(0, x)$).

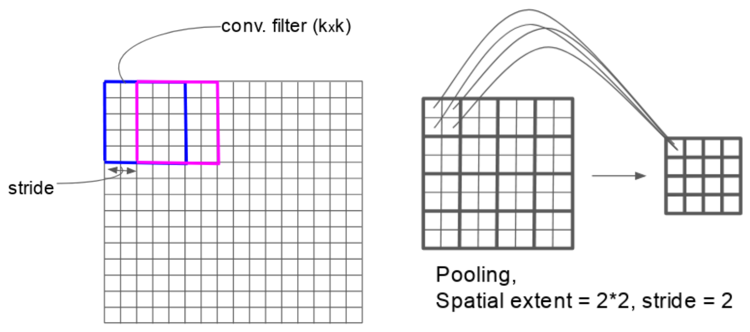
2.1.2 Typically used layers in a CNN

- **Convolutional Layers**: Owing to the fully-connected layers in MLPs, the number of network parameters increases greatly with both data dimensionality and network depth. In contrast, in convolutional layers, connections between layers are restricted to spatially local regions. Further, these local parameters or convolutional kernels are shared among neurons at different spatial locations in the same layer. This connectivity strategy leads to a much lower number of network parameters as compared to having dense connections in all layers. Also, with this formulation, CNNs readily incorporate the property of translation equivariance, that is known to hold for commonly encountered natural images.

  Some important parameters of a convolutional layer are its **kernel size** and its **stride**. Can you draw a rough schematic depicting these parameters?

- **Activation Layers**: These layers employ, element-wise, a non-linearity on their inputs. Typically employed non-linearities have already been discussed in the previous sub-section. In recent times, the ReLU non-linearity has gained increasing precedence over the sigmoid and tanh functions. This is mainly due to its ability to avoid the **vanishing gradients** problem. Can you describe this problem briefly?

  Parameters of NNs are determined via gradient-based optimization. The sigmoid and the tanh activation functions limit the range of their outputs between [0,1] and [-1,1] respectively. When the input values to these functions become higher or lower than a certain threshold, the corresponding outputs become *saturated* - relatively large changes in the input may lead to only a small change in the output (gradients become very small). This issue is further worsened for deep networks, where the gradients are computed using the chain rule - leading to multiplication of several very small gradients, until they become *vanishingly small*.



(a) Conv. parameters (b) Pooling parameters

- **Pooling Layers**: These layers reduce the spatial extent of their inputs, by *pooling* information over a specified spatial extent. Typically used pooling layers are **max-pooling** and **average-pooling**. Important parameters of a pooling layer are **spatial extent of the pooling** and **stride**. Can you draw a rough schematic depicting these parameters?
  The choice of the parameters of the pooling layer affects the **receptive field** of the CNN. The receptive field grows linearly (along the respective dimensions) with the stride value at a single pooling layer.

Typical sequence of layers: In a CNN architecture for image classification, the sequence of layers is typically like this:
input image
→ convolutional layer → activation layer → pooling layer
→ convolutional layer → activation layer → pooling layer
→ ...
→ convolutional layer → activation layer → pooling layer
→ vectorize
→ fully-connected layer → activation layer
→ fully-connected layer → activation layer
→ ...
→ fully-connected layer → output.

Number of parameters: In this part, you are required to compute the number of parameters for two network architectures. The size of the input images in the STL-10 dataset is 96*96*3. Consider the 10-class classification problem. Also, consider the case where no biases are added following either the fully-connected or convolutional layers.

- **MLP**:
  vectorize the input image
  → fully-connected layer with 256 output units → activation layer
  → fully-connected layer with 64 output units → activation layer
  → fully-connected layer with 10 output units.

  Number of parameters:

  (96*96*3)*(256) +
  (256)*(64) +
  (64)*(10)
  = 7094912.

- **CNN**:
  Input image
  → 3x3 convolutional layer with stride 1 and 32 output feature maps
  → activation layer → max-pooling layer with 2x2 spatial extent and stride 2
  → 3x3 convolutional layer with stride 1 and 32 output feature maps
  → activation layer → max-pooling layer with 2x2 spatial extent and stride 2
  → 3x3 convolutional layer with stride 1 and 32 output feature maps
  → activation layer → max-pooling layer with 2x2 spatial extent and stride 2
  → vectorize the feature maps
  → fully-connected layer with 256 output units → activation layer
  → fully-connected layer with 10 output units.

  Number of parameters:

(3)*(3*3)*(32) +
(32)*(3*3)*(32) +
(32)*(3*3)*(32) +
(12*12*32)*(256) +
(256)*(10)
= 1201504.

2.1.3 Optimization

The parameters of a CNN are typically learned using some form of **gradient descent optimization**, driven by a **loss function**. The gradients with respect to the network parameters are computed using a technique called back-propagation.

- **Loss function**: This measures the discrepancy between the network's outputs and the ground truth targets. A commonly used loss function for classification problems is the **cross-entropy loss**. Consider a case where instead of a classification task, you were solving a regression task. For instance, if the problem was: given an input image of a human face, predict the age of the person. What loss function could be used in such a case?

  Mean squared error or mean absolute error.

  Back to classification now, consider a case where the training dataset that you have for a 2-class classification problem consists of 1000 images of class 0 and only 10 images of class 1. Such a dataset is known as a **class imbalanced** dataset. For instance, in medical image classification of some rare disease, there could be a dataset consisting of many more healthy images than diseased images. Sometimes, in such cases, the network gets trained in such a way that after training, it simply predicts class 0 for all images. Can you suggest a modification in the cross-entropy loss function that could be helpful to prevent such a behaviour?

  The cross entropy loss for a binary classification is computed as:
  *target * -log(prediction) + (1 - target) * -log(1 - prediction)*
  where *prediction* is a soft-prediction between 0 and 1.
  So, if *target* for a particular sample is 0, the first term of the loss will vanish and the second term will be minimized when the *prediction* is also 0. Similarly, if *target* for a particular sample is 1, the second term of the loss will vanish and the first term will be minimized when the *prediction* is also 1.
  For class-imbalanced problems, we may wish to weigh the loss term corresponding to the class with fewer training samples higher than the other loss term. That is, if the training data contains fewer samples of class 1, then the first term of the loss can be multiplied with a positive weight and conversely if the training data contains fewer samples of class 0, then the second term of the loss can be multiplied with a positive weight.

- **Optimizer**: Once the loss is computed, we need to **assign the blame** of the loss to each parameter of the network and appropriately change their values so as to reduce the loss. As mentioned before, this is typically done by gradient descent - computing the gradient of the loss with respect to the parameters and then taking a **step** in the parameter space in the direction of the negative gradient. The **step size** (also known as the **learning rate**) is an important hyper-parameter for the optimization. If it is too low the progress of the optimization may be very slow and it may take a very long time to reach a solution. On the other hand, what could be a problem with a very high learning rate?

  Note that at any given iteration, the computed gradients are only locally correct. If the learning rate is too high, the local gradients may not be sufficiently accurate for the entire length of the step. This may be lead to instability in training - the value of the loss function may oscillate instead of decreasing as the optimization proceeds.

Usually, CNNs are trained on very large datasets. In such cases, it becomes computationally expensive to compute the loss for all the members of the dataset at each training iteration. Therefore, in each iteration, the computation is done only on a **batch** of the dataset. The **batch size** is another important hyper-parameter for the optimization. Can you briefly discuss the considerations that could be important while choosing the batch size? In particular, what could be pros and cons of a too small and a too large batch size?

The exact implications of the effect of batch size on training are not well-understood. Nevertheless, it is helpful to keep some factors in mind while choosing this hyper-parameter. The gradients computed over a batch serve as an approximation to the gradients with respect to the entire dataset. A too-small batch size might imply that although the approximation is relatively accurate on average, the variance in the gradients might become large. On the other hand, the variance introduced due to the batch-wise training is also hypothesized to aid the optimization in escaping local minima in the non-convex optimization.

2.2 Implementation

The code provided to you consists of two training routines: first, for a two-class classification problem and secondly, for a ten-class classification problem. The two-class classification problem is setup by simply selecting two classes from the STL-10 dataset (ship and cat). The second ten-class classifier works with the entire dataset. This separation into two problems has been done as you will be running your networks on CPUs. The running times on CPUs are much longer as compared to GPUs, which are usually used for training such networks. Nevertheless, as you will see, several interesting insights can already be obtained by running experiments on smaller datasets.

2.2.1 Resolve bug in training

Let's start with the two-class classification problem first. CNNs are a powerful learning technique and one would imagine that they would be able to solve a two-class classification task relatively easily. Alas, when you run the script as it is, you will probably observe rather poor performance. This is due to a bug in the 'train' function. Resolve this bug and watch the CNN do wonders!

2.2.2 Network Architecture

The network architecture is to be implemented inside the 'CNN' class. The current implementation is that of a perceptron - a single layer fully-connected neural network. A perceptron has no place inside a 'CNN' class! Modify the implementation of the 'forward' function suitably, perhaps using the already defined attributes of the class. Create a new instance of the network and train it again. Do you observe a difference in performance for the two-class problem?

For the two-class problem, even the MLP seems to work reasonable well.

How about for the ten-class problem?

For the ten-class problem, the CNN works better than the MLP.

Play around with the number of layers in the networks and the number of filters in the different layers. Do you observe any trends in performance as the network architecture is varied?

Increasing the number of layers beyond 2 does not further increase the performance.

How would you select the optimal network architecture?

Usually, the network architecture is treated as another hyper-parameter (along with others such as the learning rate, batch size, etc.). One of the ways to select such hyper-parameters is by setting aside a validation dataset and evaluating the performance of networks trained with different hyper-parameter settings on the validation dataset.

How does the training time depend on the network architecture?

As the number of parameters of the network increases, there are more computations both in the forward pass (computing the network's outputs for a batch) and the backward pass (computing the gradients with respect to the batch and updating the network's weights according to these gradients). Thus, networks with more parameters take longer to train.

What is the best test accuracy that you could obtain for the 2-class and the 10-class classification problems?

Around 95 percent for the 2-class problem and around 60 percent for the 10-class problem.

2.2.3 Discrepancy between training and test error

Is there a large gap between the accuracies on the training and the test set in the 10-class problem? If so, what could be the reasons for this behaviour?

One reason for this behaviour might be that the training set contains fewer images that the test set for this dataset. Thus, the test set may be exhibiting some variations among the classes that are not present in the training data.

What is over-fitting? Do you know any measures to prevent over-fitting?

Over-fitting refers to a situation when the performance on the test set is poorer than that on the training set. A common measure to reduce over-fitting is to augment the training dataset with transformations such as rotations, translations, scaling, etc. that are small enough such that the labels of the training images are preserved. This also incorporates the knowledge in the dataset that the classification should be invariant to such transformations.

What is regularization? Do you know any regularization techniques?

Regularization may be thought of as any constraint that is imposed during the training of neural networks, that does not directly encourage the minimization of the task-specific loss on the training data. Such constraints instead impose other desirable features on the learned mappings such as smoothness or sparsity. Common regularization techniques include weight decay, dropout, etc.

2.2.4 Feature Visualization

Recall that in the first part of this exercise, you used the histogram of oriented gradients (HOG) as features of the images and subsequently, built your classifier based on these features. An oft-cited benefit of neural networks is that they relieve us of feature engineering - instead learning suitable features themselves, solely driven by the loss function and the optimization procedure. But what kind of features do they end up discovering? For a couple of test images, visualize the learned features at different layers of a trained network and comment on them.

CNNs trained for image classification typically learn edge filters in different orientations at the initial layer. Deeper layers learn composition of basic shapes into increasingly complex / abstract patterns. Well-trained networks usually have smooth filters rather than noisy patterns.