CS231n

Thomas Lu

Note: statements that I am unsure about are enclosed in double square brackets, e.g. [[Every even integer greater than 4 can be expressed as the sum of two primes.]]

1 Lecture 1: Introduction

Computer vision is a hugely important and impactful field today. In recent years, the number of cameras in the world has exploded, and so correspondingly has the amount of visual data in existence, and, therefore, the available opportunity from understanding that data. Today (2017) visual data constitutes the vast majority of Internet traffic - for example, 5 hours of video gets uploaded to YouTube every second. There is also biological evidence for the imporance of visual understanding - a "evolutionary big bang" where the number of animal species rapidly shot up around 540M years ago coincided with what we believe to be the first occurrences of eyes in animals.

A brief history of computer vision:

- 1959: An early neurobiological study on vision in cats found that simple cells in the brain respond to edges of certain orientation, and that successive layers of more complex cells depended on the previous ones and responded to higher-level elements.
- 1963: A PhD thesis called Block World that many consider to be foundational to computer vision was introduced. The thesis described an attempt to make a computer visually understand a world made up of geometric shapes.
- 1970s: David Marr suggested that in going from a 2D picture to a 3D representation, human and primate brains first turn the 2D picture into a "primal sketch", then a "2.5D sketch", before finally arriving at a 3D representation of the world.
- $\tilde{1}$980: researchers dreamed up other ways to visually represent objects: Generalized Cylinder and Pictorial Structure were two of the more well-known ones.
- 1997: An image segmentation technique called Normalized Cut was introduced. It would cluster pixels in an image into semantically meaningful groups.
- 1999: SIFT was introduced. SIFT was a method for finding invariant "critical points" in an image that could be used to match similar objects to each other.
- 2001: A group of researchers figured out how to do face detection cheaply enough for the technology to be built into a handheld digital camera.
- 2006: Spatial Pyramid Matching, a technique for image classification, was introduced.
- 2005 (Histogram of Gradients) and 2009 (Deformable Part Model): Human detection techniques.
- 2005: The PASCAL Visual Object Challenge was introduced - an object classification benchmark dataset with around 20k images and 20 classes.

- 2006: Work began on creating ImageNet, a huge image classification dataset with (now, 2021) 14M images and 22k categories.
  - The motivation is that we need a big, complex model to effectively process images, and big models need to be trained on a lot of data.
  - There was also the question of whether it was possible to train a model to "recognize everything."
  - In 2010, the first ImageNet Large Scale Visual Recognition Challenge was held. In the beginning, top models averaged around 75% top-5 recall.
  - A step-function changed in 2012, when a strong CNN model (AlexNet) achieved 84% top-5 recall.
  - By 2015, top-5 recall surpassed human performance (over 95%). Since then, CNNs have gotten even deeper and even more powerful.
  - But CNNs weren't that new! The idea was actually introduced in 1998 by Yann LeCun. But the reason they took a long time show such good performance was because hardware and training data availability needed to catch up.

A lot of the latest advances in computer vision literature have been around image classification, but there are other important fundamental problems too, e.g.

- Semantic segmentation: given an image and a particular pixel, what object is the pixel a part of?
- 3D representation: given a 2D image, construct a 3D model of the scene.
- Activity recognition: given a video, figure out what the subject is doing.
- Image description: given an image, write a detailed natural-language description of the content.
- Humor understanding: given a humorous image, explain why it's funny.

And these problems have a ton of impactful applications: content moderation, medicine, autonomous driving, and more!

2 Lecture 2: Image Classification

Image classification is a foundational problem in computer vision. The problem statement is as follows: given an image I and a fixed set of classes C, select the category $c \in C$ (e.g. "cat", "airplane", "flower", etc.) to which I belongs. Although conceptually simple, there is in fact a very wide gap between the digital representation of an image (a grid of numbers, e.g. $w \times h \times 3$ integers between 0 and 256 for a normal RGB image) and any sort of human-interpretable semantic meaning. To illustrate the difficulty of crossing this gap, consider the following challenges:

- All the pixels in an image change if the camera viewpoint or the lighting changes.
- Objects do not necessarily assume the same (physical) shape in every picture - for example, not all pictures of cats will have the cats in the same pose.
- Objects can be occluded (partially hidden).
- The background could look very similar to the subject (e.g. a polar bear on a white background).
- Not all objects of a particular class look exactly alike - for example, a Honda Odyssey and a Lamborghini Aventador are both cars, but they are quite different in terms of appearance.

Given these challenges, it becomes clear that there's no good way to do this algorithmically in the same way one might sort a list of numbers or find the shortest distance between two points on a graph. One might

think that we can try to detect edges and corners and somehow write rules for how we should expect these edges and corners to be arranged in a picture of e.g. a cat, but even if this could be done, such an approach has several limitations:

- The resulting algorithm would likely be quite fragile, and would likely need new adjustments if a new breed of cat were introduced (or a new, particularly feline-looking, breed of dog).
- We'd have to write a new algorithm for every single class of object that we want to detect - our knowledge about what cats look like wouldn't transfer over to airplanes or flowers.

The crux of machine learning is that instead of taking an algorithmic approach, we can take a data-driven approach. Given a large number of images labeled with their correct classes, we can:

1. Train: using the labeled examples (the "training set"), we can train a model.
2. Predict: using the trained model, we can predict the labels of new examples.

2.1 Nearest neighbor

One very simple model is the nearest neighbor model. The train phase of the nearest neighbor model consists simply of memorizing the training set. The predict phase involves comparing the given example to every training example, and selecting the label of the nearest training example.

The notion of "nearest" requires us to define a distance metric. One possible metric is the L1 distance, or the sum of the absolute differences in the values of each pixel. More formally, supposing pictures $x^{(1)}$ and $x^{(2)}$ are both $w \times h$ and have red, green, and blue values at i, j of $r_{ij}^{(1)}, r_{ij}^{(2)}, g_{ij}^{(1)}, g_{ij}^{(2)}, b_{ij}^{(1)}, b_{ij}^{(2)}$, respectively, then the L1 distance between $x^{(1)}$ and $x^{(2)}$ would be

$$\sum_{i=1}^{w}\sum_{j=1}^{w}\left|r_{ij}^{(1)} - r_{ij}^{(2)}\right| + \left|g_{ij}^{(1)} - g_{ij}^{(2)}\right| + \left|b_{ij}^{(1)} - b_{ij}^{(2)}\right|.$$

Another possibility is the L2 distance:

$$\sum_{i=1}^{w}\sum_{j=1}^{w}\left(r_{ij}^{(1)} - r_{ij}^{(2)}\right)^2 + \left(g_{ij}^{(1)} - g_{ij}^{(2)}\right)^2 + \left(b_{ij}^{(1)} - b_{ij}^{(2)}\right)^2.$$

One issue with nearest neighbor, however, is that it tends to overfit: while it may perform very well if asked to predict on examples it has seen in the training set, its performance is much worse on new examples that it hasn't seen before. To address this, we can instead look at the K nearest neighbors from the training set (for some fixed value of K) and take the label that occurs most frequently among those K training examples. If we think of nearest neighbor as defining decision boundaries which, when crossed, cause the model to make a different prediction, increasing K (with the special case of vanilla nearest neighbor being equivalent to $K = 1$) tends to make the decision boundaries smoother.

Nearest neighbor and its generalization K-nearest neighbors, however, don't perform very well on image classification tasks in practice. There are several reasons for this:

- Predicting is very slow, taking $O(N)$ time where N is the number of examples in the training set (often very large for image classification).
- Distances between individual pixels is not very semantically meaningful - for example, if I take a black-and-white image of a checkerboard and its negative (flip the value of every pixel), it is essentially still the same image, but the two images would have the highest possible distance! As a less extreme example, I could make a copy of an image very "far" from the original image just by shifting it a few pixels.

- The background of an image might take up more pixels than the subject, but nearest neighbor has no concept of background vs. subject and will weigh the unimportant background pixels equally with the subject pixels.
- For nearest neighbor to work well, the training examples need to cover the input space reasonably densely. However, with data as incredibly high-dimensional as images, there is just no way to cover the input space densely.

2.2 Hyperparameters

One might notice that in K-nearest neighbors, the value of K needs to be chosen. Such hand-picked parameters (that are not learned automatically via the training process) are called *hyperparameters*. The typical way to select the best hyperparameters is just to try a bunch of settings and compare them, but the method of comparison is important.

- One very wrong way to select hyperparameters is to select the setting that gives the best performance on the training set. Earlier we mentioned a phenomenon called overfitting, and that is exactly what will happen if we select hyperparameters in this way. Note that in the example of K-nearest neighbors, $K = 1$ will always give the best performance (perfect performance, in fact) on the training set.
- A better way is to split one's data into separate training and test sets, and select the setting that yields the best performance on the test set after being trained on the training set. This will give us a model that generalizes better, but doesn't allow us to gauge how well our model would perform on unseen data - even though the test set wasn't used in training the model, it was used to optimize the hyperparameters, and we might not see identical performance on a new test set for which the hyperparmeters were not explicitly optimized.
- The standard way is the split our training data into three sets: train, validation, and test. We train the model on the training set, use the validation set to select hyperparameters, and report our results on the test set.
- One other approach is cross-validation: we split the training + validation data into n folds, and we retrain the model n times with each hyperparameter setting, once with each fold serving as the validation set, and average the results. This can help us select the best hyperparameters more "fairly", but is not so practical if we are training a large model.

2.3 Linear classifiers

A linear classifier is a type of *parametric model*, meaning that the training process learns a set of parameters W which are then used in a pre-defined function f to make predictions $\hat{y}$ for an input x according to $f(x, W) = y$. Conceptually, rather than memorizing the training set as in nearest neighbor, a parametric model attempts to summarize its knowledge of the training set in the parameters W. For a linear model, x would be a vector, while W would consist of a matrix θ and a *bias vector* b so that $f(x, W) = Wx + b$.

As a concrete example, we can consider the problem of classifying images in the image classification dataset CIFAR-10. This dataset consists of 32×32 color images that need to be classified into one of 10 categories. This means that x would be of dimension $32 \cdot 32 \cdot 3 \times 1 = 3072 \times 1$ (3 for the red, green, and blue channels) and y would be of dimension 10×1 (1 score per class, pick the class with the highest score). b would then need to be 10×1 as well, and W would need to be 10×3072.

However, it turns out that even linear classifiers are not very effective at image classification. We can visualize W by projecting each row of it back into a 32×32 image, and if we do this with a linear classifier trained on CIFAR-10, we see that each row is essentially an arithmetic mean of all the images in the corresponding class

(there are 10 rows, one per class). Doing this would show, for example, that all that the model learned about the appearance of airplanes is that pictures of airplanes tend to have blue backgrounds! In addition, linear classifiers can only learn linearly separable decision boundaries - what this means is that if we plotted each image as a point in 3072-dimensional hyperspace, the model would only be able to learn to classify images based on which side of a straight 3071-dimensional hyperplane they fall on. While it would be able to find the best hyperplane pretty effectively, it's not hard to think of many situations where a simple hyperplane would not be sufficient to distinguish examples of one class from those of another class.

3 Lecture 3: Loss Functions and Optimization

3.1 Loss functions

A *loss function* is some function that quantifies how bad a model's prediction is. The utility of such a function is that it allows us to unambiguously say that one prediction is better than another, or (say, averaging over many predictions) that one model is better than another. Given this notion, we can then try to find the best model (or the best parameterization of a model), as we can now make objective comparisons. The process of doing this is known as *optimization*.

More formally, given a model f parameterized by W and a dataset of examples $\{(x_i, y_i)\}_{i=1}^N$ where the x_i are the model inputs and the y_i are the labels, we can define some loss function $L(f(x, W), y)$ and calculate the overall loss $\mathcal{L}$ of the model over the entire dataset as

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} L\left(f\left(x_i, W\right), y_i\right).$$

Optimization then refers to any method of attempting to find the w that minimizes $\mathcal{L}$.

An example of a common loss function for multiclass classification (the task associated with CIFAR-10) is multiclass SVM loss. In this case, letting x_i represent inputs, y_i represent labels, and $s_i = f(x_i, W)$ represent the vector of scores (one per class) produced by the model, our loss function is then

$$L\left(f\left(x_i, W\right), y_i\right) = \sum_{j \neq y_i} \max\left(0, s_{ij} - s_{iy_i} + 1\right),$$

where j iterates over all the different classes and s_{ij} denotes the j-th element of s_i. In English, what this says is that:

- We want s_{iy_i} (the score for the correct class) to exceed all s_{ij} with $j \neq y_i$ (the scores for the incorrect classes) by at least 1. As long as this holds, we don't care.
- If s_{iy_i} does not exceed some s_{ij} with $j \neq y_i$ by at least 1, we want to penalize the model by an amount in proportion to the extent to which this does not hold.

Another example (also for multiclass classification) is softmax loss. Defining all our variables identically as before, the softmax loss is defined as

$$L\left(f\left(x_i, W\right), y_i\right) = -\log\left(\frac{e^{s_{iy_i}}}{\sum_j e^{s_{ij}}}\right)$$

The intuition behind this one is that:

- The term inside the log represents the model's predicted probability that the label y_i is correct given the input x_i. Replacing y_i with any class k in that term actually gives the model's predicted probability that the label k is correct given the input x_i. Note that the formulation guarantees that the predicted probability will be between 0 and 1, and that the probabilities across all the classes sum to 1.

- We want to maximize the predicted probability of the correct class, which is equivalent to minimizing its negative log.

3.2 Regularization

Note, however, that the above formulations of loss functions only incentivize the model to fit the given dataset well. If we optimize our models to minimize the value of a loss function formulated in such a way, one problem we will often run into is *overfitting*, where our model ends up fitting the training dataset extremely well by twisting itself in very complicated ways, but makes very inaccurate predictions on new, unseen data. We can combat this issue using *regularization*.

Regularization essentially means anything that encourages model simplicity, and one common form of regularization is to add a term $\lambda R(W)$ to the loss $\mathcal{L}$ that penalizes the model based on how "complex" it is (defined in terms of the function R and the parameters W). (λ here is a tunable hyperparameter that trades off between how much we value model simplicity vs. the ability to better fit the training data.) A few common regularization methods are:

- L2 loss: $R(W) = \sum_{w \in W} w^2$
- L1 loss: $R(W) = \sum_{w \in W} |w|$
- Elastic net (L1 + L2): $R(W) = \sum_{w \in W} \beta w^2 + |w|$ (here β is another hyperparameter)
- More advanced techniques (some of these will be covered in later lectures): dropout, max norm regularization, batch normalization, stochastic depth

3.3 Gradient descent

Earlier we mentioned that optimization refers to the process of finding a model that minimizes our chosen loss function. One common and reasonably effective optimization method is gradient descent. Note that for most reasonable loss functions, we can calculate the gradient $\overrightarrow{\nabla}_W \mathcal{L}$ of the loss $\mathcal{L}$ with respect to the model parameters W. Then we can simply "step downwards" to a (likely) lower value of $\mathcal{L}$ by moving W a little bit in the direction opposite the gradient:

$$W \leftarrow W - \alpha \overrightarrow{\nabla}_W \mathcal{L},$$

where α, called the *learning rate*, is another tunable hyperparameter.

Note that implementing $\overrightarrow{\nabla}_W \mathcal{L}$ can be fairly error-prone, so one common way of testing the implementations is to do a *gradient check*: to approximate the gradient numerically via finite differences, and checking that the approximation is close to the analytic value. (The reason, however, that we typically use an analytic gradient in actual gradient descent is that the numeric approximation can be very slow to calculate, especially when we have a lot of parameters W and/or when the loss function is complicated.)

Note also that gradient descent as formulated above requires calculating the gradient $\overrightarrow{\nabla}_W \mathcal{L}$ over the entire dataset, as $\mathcal{L}$ is defined over the entire dataset. This can be quite slow in practice when we have a large training set, so we typically use *stochastic gradient descent* instead, where we approximate $\mathcal{L}$ by the average loss over a minibatch of e.g. 1024 training examples. This allows us to calculate our gradients and update our model much more quickly.

3.4 Linear classifiers and image classification

In lecture 2, we mentioned that linear classifiers don't perform so well at image classification. But this raises the question of how people approached the problem of image classification before CNNs became viable. The answer is that instead of feeding raw pixel data into a linear classifier, which doesn't work so well, people instead first calculated intermediate featurized representations of images and then fed those features to a linear model that would make classification predictions. Some examples of these features are:

- Color histogram (relative frequences of different colors in the image)
- Histogram of oriented gradients. We calculate this by dividing hte image into small segments (say, 8×8), computing the dominant "edge directions" in each segment, creating a histogram of directions for each small segment, and then rolling all of this data up into a single vector.
- Bag of words. We can build some sort of visual "vocabulary" by taking small crops of many images and clustering the crops, and then look at how often each "word" appears in an input image. (In this case, we represent words as approximations rather than exact values, as an exact value match would probably not find anything in most cases.)