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 $\vec{\nabla}_W \mathcal{L}$ 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:

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

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

Note that implementing $\vec{\nabla}_W \mathcal{L}$ 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 $\vec{\nabla}_W \mathcal{L}$ 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.)

4 Lecture 4: Backpropagation and Neural Networks

4.1 Computational graphs and backpropagation

A computational graph is a visual way to represent a function that makes it easy to find gradient values analytically. It essentially breaks the function up into a composition of simple components as nodes connected by edges, each of which is easy to differentiate. In such a graph, each node represents an operation, and each edge represents the output value of one node (i.e. operation) being routed as an input to another node. The utility of this representation is that it allows us to recursively use the chain rule to calculate the gradient of the output with respect to any of the input or output values in the graph. The usual process for doing so is known as backpropagation.

More formally, suppose we have a node in a computational graph with inputs $x_1, x_2, \ldots, x_n$ that performs the operation f and has its output routed as inputs $y_1, y_2, \ldots, y_m$ to later nodes. Then letting L be the final output of the computational graph, we have the following for each $x_i, 1 \leq i \leq n$:

$$\frac{\partial L}{\partial x_i} = \frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_i} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j}.$$

As an example, consider the function

$$L = f(w, x) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

(see Figure 1 for a visualization of the computational graph). Letting $y_i, 1 \leq i \leq 5$ represent the output of node f_i (in our case, each output is routed to at most one downstream node, so such a definition is

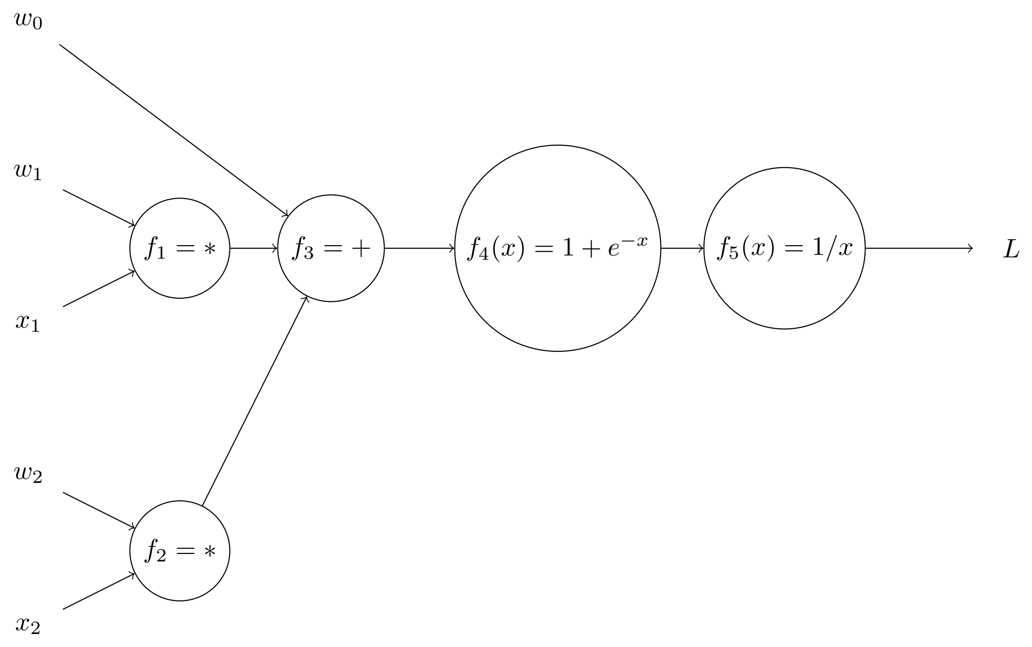

Figure 1: The function $\left(1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}\right)^{-1}$ depicted as a computational graph.

unambiguous), we have:

$$\begin{aligned}
\frac{\partial L}{\partial y_5} &= \frac{\partial L}{\partial L} = 1 \\
\frac{\partial L}{\partial y_4} &= \frac{\partial y_5}{\partial y_4}\frac{\partial L}{\partial y_5} = \ln y_4 \frac{\partial L}{\partial y_5} \\
\frac{\partial L}{\partial y_3} &= \frac{\partial y_4}{\partial y_3}\frac{\partial L}{\partial y_4} = \left(-e^{-y_3}\right)\frac{\partial L}{\partial y_4} \\
\frac{\partial L}{\partial y_2} &= \frac{\partial y_3}{\partial y_2}\frac{\partial L}{\partial y_3} = 1 \cdot \frac{\partial L}{\partial y_3} \\
\frac{\partial L}{\partial y_1} &= \frac{\partial y_3}{\partial y_1}\frac{\partial L}{\partial y_3} = 1 \cdot \frac{\partial L}{\partial y_3} \\
\frac{\partial L}{\partial w_0} &= \frac{\partial y_3}{\partial w_0}\frac{\partial L}{\partial y_3} = 1 \cdot \frac{\partial L}{\partial y_3} \\
\frac{\partial L}{\partial w_1} &= \frac{\partial y_1}{\partial w_1}\frac{\partial L}{\partial y_1} = x_1 \frac{\partial L}{\partial y_1} \\
\frac{\partial L}{\partial x_1} &= \frac{\partial y_1}{\partial x_1}\frac{\partial L}{\partial y_1} = w_1 \frac{\partial L}{\partial y_1} \\
\frac{\partial L}{\partial w_2} &= \frac{\partial y_2}{\partial w_2}\frac{\partial L}{\partial y_2} = x_2 \frac{\partial L}{\partial y_2} \\
\frac{\partial L}{\partial x_2} &= \frac{\partial y_2}{\partial x_2}\frac{\partial L}{\partial y_2} = w_2 \frac{\partial L}{\partial y_2}
\end{aligned}$$

Each y_i and each partial of L with respect to the y_i can be expanded back into an expression in terms of the inputs, but if our goal is to compute gradient values (rather than symbolic gradients), it's more straightforward to:

- Starting from the left side of the computational graph, compute the intermediate values, i.e. the y_i, by applying each node's operation to its inputs in sequence. This process is known as *forward propagation.*
- Then, starting from the right side of the computational graph, compute the values of partials with respect to the y_i. Note that in the above example (as in general), each gradient $\partial L/\partial y_i$ can be written in terms of y_i and the gradients with respect to the nodes after y_i in the graph. This process is known as *backpropagation.*

In this way, we can get the gradient values of L with respect to the inputs.

Oftentimes when working with vectorized code, our inputs and outputs of various nodes will be vectors rather than scalars. The logic is the same, but we would need to replace our scalar partial derivatives

$$\frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_i}$$

with Jacobians

$$\frac{\partial \left(f_1(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n), \ldots, f_m(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n)\right)}{\partial \vec{x}_i} = \begin{bmatrix} \frac{\partial f_1}{\partial \vec{x}_i^{(1)}} & \cdots & \frac{\partial f_1}{\partial \vec{x}_i^{(d_i)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial \vec{x}_i^{(1)}} & \cdots & \frac{\partial f_m}{\partial \vec{x}_i^{(d_i)}} \end{bmatrix},$$

where now each x_i has become a vector $\vec{x}_i$ with dimension d_i, and f is now a vector-valued function (with m-dimensional output) that takes vector inputs.

Note that the node boundaries in a computational graph are somewhat arbitrary; in Figure 1 we could just has well have combined f_1, f_2, f_3 into a single node $g(w_0, w_1, x_1, w_2, x_2) = w_0 + w_1x_1 + w_2x_2$. The choice of node boundary is just a tradeoff between concision of the graph and the ease of calculating analytic node-local partial derivative values.

4.2 Neural networks

Neural networks are a highly diverse class of computational structures, but at their core they are essentially just stacks of simple vectorized operations (often linear transforms) with nonlinear activations in between. Structured correctly, neural networks can be used as parametric models for machine learning (and that is their most common use today). A simple example of a neural network is one with parameters W_1, W_2 (matrices) and input x (vector) and performing the computation

$$f(x|W_1, W_2) = W_2 \max(0, W_1x).$$

We can say that this neural network has two linear layers (the multiplications by W_1 and W_2), and that the function $g(y) = \max(0, y)$ applied after the first layer W_1x is the *activation function* on that layer.

Note also that neural networks fit nicely into the computational graph framework - each layer and each activation function can be a node in a computational graph. This is where computational graphs really shine: they allow us to easily calculate the gradient values of the output (or any loss function we may want to apply on top of the output) with respect to the parameters (in the above example, W_1 and W_2), which enables us to efficiently perform gradient descent to optimize the parameters over a collection of training data.

A few vocabulary words when talking about neural networks:

- The *input layer* of a neural network consists simply of the input values that are fed into the network. Note that the input layer does not include the neural network's parameters. In our example above, this would simply be x.
- The *output layer* of a neural network refers to its final output. In our example this would be f, i.e. $W_2 \max(0, W_1 x)$.
- A *hidden layer* in a neural network refers to any layer that is not the input or output layer. The definition of a layer is somewhat arbitrary, but in most cases a layer refers to a vector that is the result of a matrix multiplication (or a group of parallel matrix multiplications) and an activation function on a previous layer. In our example above, $\max(0, W_1 x)$ would be considered to be a hidden layer.
- When people refer to the layers as being ordered (e.g. "the third hidden layer"), the ordering starts from the input layer and ends with the output layer. So the first hidden layer would be the one immediately after the input layer, and last hidden layer would be the one immediately before the output layer.
- A neural network with an input layer, an output layer, and $n-1$ hidden layers is generally called an n-layer neural network. (The input layer "doesn't count".) Sometimes it is also called an $(n-1)$-hidden-layer neural network.

4.2.1 The importance of nonlinearity

Here I will try to offer an informal but intuitive explanation of why the nonlinear activation functions are important (mostly derived from an explanation that was given in the lecture). I've never really been satisfied with the terse orthodox explanation, which is that a neural network without nonlinear activations would collapse into a single linear transform. While this proves that a neural network without nonlinearities would not represent any improvement over a simple linear model, it doesn't do a good job of providing intuition as to how or why the nonlinearities would actually be of any help, which is what I will try to do here.

Consider again the simple 2-layer neural network

$$f(x|W_1, W_2) = W_2 \max(0, W_1 x).$$

Recall the CIFAR-10 dataset (an image classification dataset consisting of 32×32 images that are classified into 10 categories) and the difficulties of training an effective linear classifier on this dataset (each row of W in a linear classifier $f(x|W) = Wx$ just ends up being an arithmetic mean of the images of the class corresponding to that row). For example, consider the class "horse". If we train a linear classifier $f(x|W) = Wx$, where x is a vector of dimension $32 \cdot 32 \cdot 3 = 3072$ (32×32 picture with 3 color channels) and W is a matrix of dimension 10×3072 (10 classes by 3072 elements in x), on CIFAR-10, we'd notice that a visualization of the row of W (i.e. taking the 3072-dimensional vector of that row's entries and turning it back into a 32×32 color image) corresponding to the "horse" class would look like an animal with a horse body and a horse head on each end of the body, because the model averaged right-facing horses and left-facing horses together (just to be clear, this is what you would actually observe).

However, suppose now that we train a 2-layer neural network $f(x|W_1, W_2) = W_2 \max(0, W_1 x)$, where x is the same 3072-dimensional vector, W_1 is a 20×3072 matrix, and W_2 is a 10×20 matrix. Now the model can (for sake of example, it may not actually do this when trained) separately detect left and right-facing horses in the first layer $W_1 x$ and then combine the two intermediate scores in the final result $W_2 \max(0, W_1 x)$.

To see why the nonlinear activation function (here, $g(x) = \max(0, x)$) is important, note that the score for left-facing horses in the first layer $W_1 x$ may be very negative when the score for right-facing horses is very positive, and vice versa (again, just for sake of example - maybe this isn't what actually happens when you train the model). This means that if we try to combine them linearly in the final layer, the model wouldn't really be able to effectively detect the presence of one or the other, as the scores would essentially cancel each other out. But by inserting our activation function g, the model can simply throw away the negative score (it would be replaced by 0), and the final layer would be able to detect the presence of either a left-facing horse *or* a right-facing horse without having to deal with the negative score from the opposite-facing horse.

5 Lecture 5: Convolutional Neural Networks

Convolutional neural networks (CNNs) are so named after their use of *convolutional layers*. Before digging into what those are, however, we will first introduce the other building blocks that typically occur in 2017-era CNNs. A typical image classification architecture consists of:

- A large number of repetitions of:
  - 1-5 repetitions of *convolutional layers* with ReLU activations (ReLU is short for Rectified Linear Unit, which is just the function $f(x) = \max(x, 0)$.)
  - Followed by a *pooling layer*
- Followed by a few (0-2) fully-connected layers with ReLU activations (A fully-connected layer is the "standard" kind of layer in a neural network, where one layer is transformed into the next via a simple matrix multiplication and activation function; the name comes from the fact that every neuron's value in the previous layer is an input, via the matrix multiplication, to the value of every neuron in the next layer)
- With softmax loss at the end.

A more concise way to write this is:

$$((CONV + RELU) \cdot N + POOL) \cdot M + (FC + RELU) \cdot K + SOFTMAX.$$

The new concepts here are the convolutional layer and the pooling layer. We will cover these in turn.

5.1 Convolutional layers

Convolutional layers are useful in computer vision for their spatial-structure-preserving properties.

When we run an image through a fully-connected layer, we lose all spatial information about the image. Each pixel is treated independently by the matrix multiplication, and there is nothing inherent in the computation that could encode, for example, the spatial proximity of two pixels in the input. Another way to think about this is that you could permute the pixels of the input images any way you want, and as long as you apply the same permutation to every image, a model using fully-connected layers would react no differently to any choice of permutation versus any other, implying that all that the model "sees" is a soup of pixels rather than a coherent image.

A convolutional layer solves this problem by convolving position-invariant and trainable 2-dimensional *filters* over a 2-dimensional image, where each filter operates over a small spatial cluster of pixels, and produces a 2-dimensional output which preserves the spatial structure of the image. Because a filter only "sees" groups of the pixels that are close to each other and because the output is kept spatially consistent with the input image, a convolutional layer is able to inherently encode some notion of what things are close to each other and where they are.

- Strictly speaking, "pixel" and "image" are not the right words here, as a convolutional layer could be operating over the output of an earlier convolutional layer rather than operating directly over the input image, but the idea is the same regardless.

We will now define more precisely what a filter is. A filter F has dimensions (height, width, depth) $h_f \times w_f \times d_f$, is *convolved* over an input M_1 having dimensions $H_1 \times W_1 \times D_1$ (the input is logically 2-dimensional, but e.g. in the case of a 2D RGB image we need a depth of 3 to account for the 3 color channels) with *padding* p and *stride* s, and produces an output M_2 having dimensions $H_2 \times W_2$ (the lack of a depth dimension here is intentional). Each of the entries in the filter is a trainable parameter of the model. The output M_2 is then passed elementwise through an activation function to produce an *activation map*, which serves as the input to the next layer in the CNN. Defining the terms used here:

- Convolving the filter means to apply the filter at each spatial position (h, w) in the input.
- Applying the filter at a position (h, w) of the input means to compute the value

  $$b + \sum_{i=1}^{h_f} \sum_{j=1}^{w_f} \sum_{k=1}^{d_f} F^{(i,j,k)} M_1^{(h+i-1, w+j-1, k)},$$

  where b is a trainable bias term of the filter. This value becomes one of the values in M_2. Although the exact indices in M_2 depend on the padding and stride (defined below), the spatial arrangements of the input and output are kept identical: adjacent positions at which the filter is applied in the input become adjacent values in the output, and their relative orientation (left/right/up/down) is preserved.
  - Note that this definition implies that $d_f = D_1$; this is intentional and in fact required.
  - In practice, we almost always see $w_f = h_f$, and the most common values are 1, 3, 5, and 7.
- Stride refers to how far we move (in terms of pixels/indices) when going from one application of the filter to the next. Stride 1 means we apply the filter at every index/pixel; stride 2 means we apply the filter at every other index/pixel, etc. Stride is typically either 1 or 2.
- Padding refers to adding a border of zeroes around the input (along the height and width dimensions and extending fully through the depth dimension). This is typically done to preserve the height and width of the activation maps from one layer to the next (if e.g. you convolve a $3 \times 3 \times d$ filter over a $32 \times 32 \times d$ input with stride 1, you will end up with a 30×30 output, but padding a single layer of zeroes, i.e. $p = 1$, around each side will give you a 32×32 output instead). Padding is usually just set to whatever value will exactly preserve (for stride 1) or halve (for stride 2) the dimensions of the input.
- Note that H_2 and W_2 can be determined exactly from H_1, W_1, h_f, d_f, p, s. The formulas are:

  $$H_2 = \frac{H_1 + 2p - h_f}{s} + 1$$
  $$W_2 = \frac{W_1 + 2p - w_f}{s} + 1.$$

  - Note that this formula implies that horizontal and vertical stride are equal to the same value s, and that the padding p is the same on all four sides of the input (the factor of 2 in $2p$ comes from the fact that the padding is on both the left and right sides, and on both the top and bottom of the input). While this is not strictly necessary, it is nearly always true in practice.
  - This formula also implies that s divides evenly into both $H_1 + 2p - h_f$ and $W_1 + 2p - w_f$. What divisibility by s means is that the filter should "fit evenly" when convolved over the input with stride s: if we start with the filter at the leftmost position and move s pixels to the right at a time, it should cover the rightmost pixel of the (padded) input once it reaches its own rightmost possible position, as opposed to ending up somewhere where there are additional pixels to the right that have not been operated over by the filter, but from where the filter cannot be moved over s more pixels without its own right edge overshooting the right edge of the (padded) input. (Same for the vertical direction.) Again, this is not strictly necessary, but is nearly always true in practice.

- In addition, for an application of a filter at position (h, w) in the input, we can compute the position (h', w') of the resulting value in the output given the padding p and stride s:

$$h' = \frac{h+p-1}{s} + 1$$
$$w' = \frac{w+p-1}{s} + 1.$$

  Again, the implications about divisibility by s are intentional. (In these equations, the indices (h, w) are considered to start at 1 at the boundaries of the original input; h and w can potentially take values less than 1 on the boundary of the padding.)

- A convolutional layer will typically include multiple filters, all with the same dimensions, padding, and stride. Each filter produces an activation map of dimension $H_2 \times W_2$, and so a convolutional layer with D_2 filters produces an output of dimension $W_2 \times H_2 \times D_2$. The number of filters is usually a power of 2, and typical values start at 32 and go as high as your hardware can handle/as long as you are willing to wait for your model to finish training.

With these definitions settled, it may be helpful to re-read the beginning of this section, which laid out the big picture without defining anything precisely.

One thing to note is that the convolution takes place only over the height and width dimensions, and not the depth dimension, of the input. This is because the height and width indices encode useful spatial information, while the depth does not - in the initial input layer, each index in the depth dimension simply represents a different color channel, while in subsequent layers, each index in the depth dimension is simply an activation map produced by a different filter. The ordering of these indices is meaningless (e.g. it doesn't make any logical difference whether red or green comes first in the input), and so each index is treated identically and independently by the model, i.e. the model is indifferent to their permutations - the only thing that matters is that the permutations are kept consistent. For the same reason, we never pad the depth dimension - only the height and width dimensions.

A note on intuition: recall that a CNN has multiple convolutional layers, with each one's output providing the input for the next one. If we visualize the filters in the CNN (i.e. visualize the patterns in the input image that would maximize the output values of each filter), we would see that filters in earlier (closer to the input) layers tend to "detect" (i.e. produce the highest values when they encounter) simple things like edges in particular directions and patches of certain colors, and that filters in later layers tend to detect higher-level visual elements, like the presence of an eye or a door.

Lastly, an additional benefit of convolutional layers, totally unrelated to their spatial awareness, is that they have far fewer trainable parameters than fully-connected layers, which dramatically improves training efficiency. For example, if our CNN has a 128×128 RGB input, a 5×5 filter would contain only $3 \times 25 + 1 = 76$ parameters (3 colors by 25 pixels plus a bias term; note in fact that the number of parameters in the filter is invariant of the size of the input), while a fully-connected layer would require $128 \times 128 \times 3 = 49152$ parameters for every value in the output layer (as many parameters as 646 5×5 filters!).

5.2 Pooling layers

Compared to convolutional layers, pooling layers are conceptually far simpler. A pooling layer has a height h, width w, and stride s, and essentially downsamples the entries of a $H \times W \times D$ input M_1 to produce an $H' \times W' \times D$ output M_2 with $H' < H$ and $W' < W$. It does this by applying a pooling function f (which is typically defined to output the max, but is sometimes defined to output the average, of its inputs) to each $h \times w$ area of each activation map in the input, spaced s pixels apart. More formally,

$$M_2^{(i',j',k)} = f\left(M_1^{(i+i_h,j+j_w,k)}, \quad 0 \leq i_h \leq h-1, 0 \leq j_w \leq w-1\right),$$

where

$$i' = \frac{i-1}{s} + 1$$
$$j' = \frac{j-1}{s} + 1.$$

Note that pooling layers do not use padding. Typically $h = w = 2$ or 3, and $s = 2$.

The intuition for pooling layers is that activation maps output by convolutional layers conceptually represent the presence of certain visual elements at particular locations in the input image, and that a 2×2 group of adjacent values in an activation map are just reporting the presence or absence of a visual element in almost the same place in the input image, i.e. they are likely "seeing" the same thing. This means that we can combine these four values into one without losing much information, which is exactly what pooling does. Now a 2×2 pooling layer is still different from just using a stride of 2 in the previous convolutional layer - the pooling layer aggregates the information of all four outputs in its vicinity, while using a stride of 2 in the previous convolutional layer would just take one of every 4 outputs, chosen somewhat arbitrarily.

Pooling layers, just like convolutional layers, operate only over the height and width dimensions of their inputs. The reasoning as to why is the same as for convolutional layers - the ordering of the depth indices carries no logical information, so it wouldn't make any sense to pool two different activation maps that happen to be adjacent to each other in the depth dimension.

5.3 The fully-connected layers at the end

Given all the advantages of convolutional layers, one might wonder why we need the fully-connected layers at the end. While they are not strictly necessary (some architectures feed the ouptut of the last convolutional or pooling layer, fully connected, straight into the softmax classification layer with no additional fully-connected layers in between), they can be helpful for combining all the visual information extracted by the previous convolutional layers. The last convolutional layer might tell us that there is a fin here or a branch there, but to most effectively classify whether the input image is of a fish, a tree, or something else, we need to combine all the information in some way, and often this is done through a number of fully-connected layers before the final classification.

6 Lecture 6: Training Neural Networks, Part I

This lecture covers several practical tips for training neural networks.

6.1 Activation functions

There are many different activation functions that can be used as the nonlinearity in a neural network. The following are several commonly used ones.

6.1.1 Sigmoid

In general, the sigmoid does not actually work very well, and is in fact quite bad for image classification (e.g. some 2015-era ImageNet-SOTA CNN architectures will totally fail to train if the sigmoid activation is used). However, it's one of the earliest activation functions that ML practitioners originally used. The function is

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

As an activation function, the sigmoid has 3 major shortcomings:

- If $|x|$ exceeds 10 or so, the derivative of this function becomes vanishingly small (around 4.5×10^{-5} for $x = \pm 10$). This is problematic because we will be multiplying by this derivative many times when computing gradients in our neural net - e.g. if we have a neuron computing $a = \sigma(w^T x + b)$ in a model with final loss L, we will have

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w_i} = \left(\frac{\partial L}{\partial a}\right) x_i \sigma'(w^T x + b),$$

  which will be very small if $\sigma'(w^T x + b)$ is very small. This is not ideal because it will cause our gradient updates to be small, and our model to train slowly.

- The sigmoid function always outputs in the range $(0, 1)$ - in particular, its outputs are all positive. This is also not ideal - consider again our example neuron that calculates $a = \sigma(w^T x + b)$. Our gradients with respect to the weights w_i are (again)

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w_i} = \left(\frac{\partial L}{\partial a}\right) x_i \sigma'(w^T x + b).$$

  Now note that if the x_i are outputs of a sigmoid activation in a previous layer, they will all be positive, which would mean that all the gradients $\partial L/\partial w_i$ would be the same sign (the other quantities in the above expression are invariant with respect to i). This means that our gradient updates will always move all the w_i in the same direction (all moving positively or all moving negatively), which is unlikely to be optimal; there is no reason why the most direct path to the nearest minimum would be a direction along which the w_i are all changing positively or all changing negatively. Gradient descent would still be able to eventually get us to the right spot, but the path will likely look jagged and inefficient rather than smooth and direct.

- Lastly, the exponential function in the sigmoid is somewhat expensive to compute.

6.1.2 tanh

The tanh function is very similar to the sigmoid function:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} = 2\sigma(2x) - 1.$$

It solves the second problem of the sigmoid - the outputs of tanh are in the range $(-1, 1)$ rather than the rante $(0, 1)$ - but shares the other two issues of vanishing gradients and high computational demands.

6.1.3 ReLU

The ReLU, or Rectified Linear Unit, is a very popular activation that works quite well in practice. It has a very simple formulation: $f(x) = \max(0, x)$. It has several advantages:

- There is no issue with vanishing gradients; the derivative of ReLU is either 1 (for $x > 0$) or 0 (for $x < 0$). (The 0 gradient for negative input sometimes causes issues though; we will return to this in a bit.)
- Being very logically simple, it's fast to compute.
- In practice, models using ReLU activation converge faster than models using sigmoid or tanh activation.

However, it has two drawbacks as well:

- As with sigmoid, its outputs are all nonnegative.
- The derivative of ReLU is 0 for negative inputs. This means that if a neuron computing $a = f(w^T x + b)$ (in a network with final loss L) ends up in a place where $w^T x + b < 0$ for every example x in the training set, its parameters w will never update, as the gradients

$$\frac{\partial L}{\partial w_i} = \left(\frac{\partial L}{\partial a}\right) x_i f'(w^T x + b), \quad \frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial a}\right) f'(w^T x + b)$$

  will all be zero if $w^T x + b < 0$. (Note: if this neuron is in an intermediate layer, it is possible that an update in an earlier layer shifts the distribution of the inputs to this neuron, so that we do get $w^T x + b > 0$ for some training example at a later time. But this isn't something we can count on in general.) This can happen due to an unlucky initialization, or it can also happen during training if a gradient update knocks our parameters w into a bad place that causes $w^T x + b < 0$ for all later training examples x. This phenomenon is commonly called the "dead ReLU" phenomenon.

  – People sometimes initialize bias parameters to small positive values to try to avoid this issue, but in practice this doesn't seem to yield much benefit generally. (In spite of the dead ReLU issue, ReLU works pretty well.)

6.1.4 ReLU generalizations

People have also created several generalizations/modifications of ReLU to try to solve the issues of "standard" ReLU (all outputs are the same sign, dead ReLU). Several of these other activation functions are:

- Leaky ReLU: $f(x) = max(\alpha x, x)$. α is a hyperparameter and is usually set to some small positive value, like 0.01, which solves both the issue of uniform-sign outputs and zero gradients.
- Parametric ReLU: same as Leaky ReLU, but α is a trainable parameter instead of a hyperparameter.
- Exponential linear unit (ELU):

$$f(x) = \begin{cases} x, & \text{if} x > 0, \\ \alpha(e^x - 1), & \text{if} x \leq 0. \end{cases}$$

  ELU also has the problem of (nearly) zero gradients for negative values of x (although only if $|x|$ is large), but has been shown to be more robust to noise than Leaky ReLU and parametric ReLU in some cases.
- Maxout: $f(x) = \max(w_1^T x + b_1, w_2^T x + b_2, \ldots, w_n^T x + b_n)$. This further generalizes Parametric ReLU, but multiplies the number of parameters in the model, which can make training much more computationally intensive.

In practice, ReLU with an appropriate learning rate is usually good enough, and these other fancy activation functions are probably not going to be necessary.

6.2 Data preprocessing

Typically, when training neural networks, we want to normalize our inputs so that each one has zero mean and unit variance. Sometimes we can go further and decorrelate the inputs as well, so that every pair of inputs has zero covariance. This smooths out the loss landscape and helps the model converge faster.

However, when training an image processing model, we typically don't do anything to normalize the variances or covariances. This is because the input values are already typically constrained to the range $[0, 255]$ (or some other fixed range) and because diagonalizing the covariance matrix could significantly distort the visual elements in the image (which the filters in CNNs are actually quite good at learning to recognize). Normalizing to zero mean is still important, however - as discussed previously, we don't want our inputs to all be the same sign (recall that this forces all the gradients of the parameters operating on those inputs to be the same sign). Additionally, there are two different methods that are commonly used to calculate the means for normalization:

- Subtracting the mean image from each image. This basically means taking separate R, G, and B averages at each pixel, combining all of these to form a RGB "mean image", and subtracting that image, per-pixel and per-channel, from each input image.
- Subtracting channel averages from each pixel in each image. This means to take global R, G, and B means over all pixels in all input images (so we end up with 3 scalar values at the end), and subtracting the R mean from every R value in every pixel on every image (and analogously for G and B).

6.3 Initialization

A naive way to initialize a neural network would be to set all the weights to zero at the beginning. However, the issue with this is that every neuron in a layer would then be getting the same values during forward propagation and the same gradients during backpropagation, as they would all be parameterized exactly the same. In fact, the value 0 is not special here - in general we would run into the same issue if we initialized all the weights to the same value, regardless of what that value was.

A better way to do things would be to initialize weights to random numbers via a zero-mean distribution. However, the variance chosen is important here: too high of a variance and your neuron activations will explode (if using e.g. ReLU) or saturate (if using e.g. sigmoid or tanh; recall that $f'(x) \approx 0$ for $|x| > 10$ when f is the sigmoid or tanh function); too low of a variance and your neuron activations will approach uniformity (all zero for ReLU and tanh activations; all 0.5 for sigmoid since $\sigma(0) = 0.5$). The correct variance is not actually obvious from first principles, and depends heavily on the activation function. The paper "On weight initialization in deep neural networks" (Kumar 2017) derives the ideal variances for sigmoid, tanh, and ReLU activations; they are:

$$\frac{1}{\sqrt{N}} \qquad \text{(tanh, \textit{Xavier initialization})}$$

$$\frac{\sqrt{12.8}}{\sqrt{N}} \qquad \text{(sigmoid)}$$

$$\frac{2}{\sqrt{N}} \qquad \text{(ReLU, \textit{He initialization})}$$

where N denotes the number of inputs into the neuron. (These are the ideal variances for the initializations of the weights in the transform from the previous layer to the current neuron, i.e. the neuron with N inputs.)

6.4 Batch normalization

Batch normalization (Ioffe & Szegedy, 2015) is a technique that actually reduces the need to get initialization just right. It operates on a per-neuron basis. Specifically, suppose there is a neuron in the network that outputs a value y, i.e. $y = f(x|w)$ where x and w represent the inputs and parameters, respectively, of the neuron. Batch normalization can be applied to the output of neuron, and in a mini-batch during training, it computes the value

$$\hat{y} = \gamma \frac{y - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta,$$

where μ_B and σ_B^2 are the mean and variance, respectively, of y over the mini-batch, γ and β are (per-neuron) learned parameters of the model, and ϵ is a positive-valued hyperparameter (it apparently improves stability, but its exact value doesn't seem to be too important; it doesn't get much screen time in the paper). Later neurons which would have taken y as input are given $\hat{y}$ as input instead.

One important point about applying "batch norm" to a CNN is that each filter is treated as a single neuron in a CNN, so that the batch-wise means and variances for a filter's output are calculated not only over every image in a batch, but also over every position in each image at which the filter is applied (so if there are m images per mini-batch and a filter is applied at k positions in each image, batch norm would calculate the mean and variance of the filter's output over a total of $m \cdot k$ values).

During inference, the values μ_B and σ_B^2 are replaced with global means μ_y and σ_y^2 (as, obviously, during inference, there is no mini-batch).

The benefits of batch norm include:

- It reduces the need to initialize the parameters of the model with the correct variances, as vanishing/exploding values are fixed by normalization. This is a bit more obvious if we remove γ and β in the equation for $\hat{y}$ - it then becomes just a fairly standard zero-mean and unit-variance normalization. (The reason that γ and β are included is because normalizing everything to zero mean and unit variance may not always be ideal, since some activation functions, like sigmoid and tanh, are largely linear in those input domains.)
  - For the same reason, batch norm allows us to use higher learning rates - the normalization reduces the risk of gradient explosions that would normally come with higher learning rates.
- It speeds up model convergence. The authors explain that this is because it reduces *covariate shift* in the inputs to intermediate layers in the model (covariate shift is when the inputs to a model undergo a change in their distribution). The intermediate layers of the model will normally see covariate shift because shifts in the weights of one layer (due to gradient updates) will cause the distributions of inputs to all subsequent layers to shift as well. However, batch norm solves this problem by normalizing the means and variances of all the outputs.
- Because normalization statistics are computed per mini-batch during training, the model will not always see identical representations of the same input each time it appears, as that representation will depend on the other examples in the mini-batch. This has a regularizing effect on the model, and the authors have observed in some cases that batch norm makes Dropout unnecessary.

The details behind why batch normalization works and the motivations behind the various decisions made in its formulation can be found in the original paper.

6.5 Babysitting the learning process

When training a model for the first time, there are a few sanity checks you can do to make sure that your implementation doesn't have any obvious issues:

- Set your regularization hyperparameters to zero, initialize your model randomly, and make sure that your loss matches what it should be for a totally random predictor. For example, if you are using softmax loss to classify images into 10 categories, your loss (per training example) should be roughly equal to $-\ln(0.1) \approx 2.30$ after randomly initializing the model and before doing any training, assuming you have zeroed out any regularization terms in the loss function.
- Then, adding the regularization term back to the loss, do the same thing, and verify that your loss is higher than before.

- Now, with regularization off again, initialize your model and train it on a very small training set (say, 10-100 examples or so). The model should be able to fit the training set perfectly; if not, there is probably a bug preventing it from updating properly.

Once the sanity checks are complete, you can start to train your model for real. The first step typically involves tuning your learning rate.

- If you see that your loss is not changing, your learning rate is likely too small.
- If you see that your loss is increasing and/or going to NaN, your learning rate is probably too big.
- In practice, a learning rate between 10^{-5} and 10^{-3} usually works well.

These issues could also be caused by bad initialization, but if you're using batch norm and Xavier initialization for tanh activation/He initialization for ReLU activation, your issues are unlikely to be coming from initialization.

6.6 Hyperparameter tuning

A few assorted tips for hyperparameter tuning:

- It's generally good to tune hyperparameters in two stages: a "coarse" stage where we find what hyperparameter values allow the model to train normally (i.e. the loss goes down at a reasonable pace and doesn't explode), followed by a "fine" stage where we find the exact hyperparameter values within the "working" range that give the best possible model performance.
  - During the coarse stage, we can improve efficiency by stopping model training early if we observe that the loss explodes (exceeds, say, 3x the original loss). During the fine stage it's better to train models to the end to get an accurate observation of the overall effects.
  - If, after the fine stage, we see that the best hyperparameter settings are clustered around the edge of the value range of one of our hyperparameters, it can be good to try some more values of that hyperparameter past the range.
- Selecting hyperparameter values randomly within our "working range" is likely better than searching over a strict grid, as a grid search limits the number of distinct values of each hyperparameter in our search. For example, if we have two hyperparameters and have the compute budget to search over 9 different settings, it's more efficient to search somewhat randomly and allow each hyperparameter to take 9 distinct values than to select 3 distinct values for each hyperparameter and only test the $3 \times 3 = 9$ combinations afforded by those values.
- If training accuracy is much better than cross-validation accuracy, the model is probably overfitting and likely needs stronger regularization. If training accuracy and cross-validation accuracy are the same (or cross-validation accuracy is somehow better), we can probably either reduce the strength of the regularization or increase the capacity of our model (i.e. the size, as in the width (the size of each layer) or the depth (the number of layers)).
- If the loss is largely flat at the start of training, then suddenly starts to drop quickly after training for a while, the model was probably initialized poorly (maybe bad luck, maybe bad initialization parameters).
- As a rule of thumb, each update to a weight should be changing its value by around $\pm 0.1\%$. Much more or less than that may indicate that the learning rate is bad, and that the model will either be slow to converge or unstable.

7 Lecture 7: Training Neural Networks, Part II

A quick note was made at the beginning of lecture on centering and normalizing input data: it's important to do it because it makes classification accuracy less sensitive to small perturbations in the parameters and decision boundary. As an example, suppose that we had a binary classification dataset with two input features x_1 and x_2 that could be perfectly separated by the line $x_1 = x_2$, and that the input space is roughly uniformly distributed across the unit disk. If we stretch everything out with the transformation $x_1' = 100x_1$, then the decision boundary becomes the line $x_1' = 100x_2$. But more importantly, a small rotation of the decision boundary could now affect overall accuracy by a ton, whereas it may not have affected accuracy much pre-transformation. When a model's performance (and, thus, likely loss as well) changes drastically in response to small changes in its parameters, the training process will have a much harder time converging.

7.1 Issues with stochastic gradient descent

Stochastic gradient descent usually works acceptably well on simple problems, but it suffers from a few common issues:

- When the partial derivatives of the loss are very large in one direction but very small in another (i.e. the loss changes quickly in one direction but slowly in another), SGD tends to zigzag across the "steep" dimension and move very slowly across the "shallow" dimension.
  - More formally, one would say that the loss function has a high "condition number" at the current point in the parameter space. This occurs when, among the singular values of the Hessian matrix of the loss function with respect to the parameters, the ratio between the largest and smallest is large.
  - This is not unlikely in high-dimensional settings with 10s or 100s of millions of parameters!
- SGD can easily get stuck in local minima (very rare) or saddle points, where the partial derivatives are all zero.
  - See the paper "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization" (Dauphin et al. 2014) for the reasons why saddle points are much more common than local minima.
- SGD is quite noisy and can take a while to converge since the gradient on each small minibatch often does not correlate that strongly with each other. Oftentimes the parameters take a jittery and meandering path toward the convergence point.

7.2 Momentum

One method that can deal with all of these problems at once is called momentum. The most basic technique applying this concept is simply called "SGD with momentum". In normal SGD, we perform the below update at each training step, with W representing the parameters, X representing a batch of training data, L representing the loss, and α representing the learning rate:

$$W \leftarrow W - \alpha \vec{\nabla} L(W, X).$$

In SGD with momentum, we additionally maintain a velocity term v (initialized to 0) and use it to update our parameters:

$$\begin{aligned} v &\leftarrow \rho v + \overrightarrow{\nabla} L(W, X) \\ W &\leftarrow W - \alpha v \end{aligned}$$

Here ρ is a hyperparameter; usually it is set to 0.9 or 0.99. Intuitively, what we are doing here is moving in the direction of a weighted average (with weights that exponentially decay over time) of all past minibatch gradients. This makes zigzags cancel out with each other, amplifies consistent small gradients in one direction, keeps us going if we hit a saddle point (or any other type of flat region), and makes the parameter update trajectory relatively smooth.

There is another variant of momentum called Nesterov momentum. To understand its difference from the "vanilla" momentum presented above, note that the above method first calculates a velocity by combining the previous velocity with the current gradient, and then performs a single update in the resulting direction. Nesterov momentum differs in that, conceptually, it first performs an update in the direction of the previous velocity, then recomputes the gradient (wih the updated params) and performs another update in the direction of the new gradient. Intuitively, this gives the gradient an opportunity to "correct" the momentum term somewhat if it causes the parameters to move too far in an unhelpful direction or overshoot the convergence point, which can allow models to converge faster. Formally, the update operation proceeds as follows:

$$\begin{aligned} v &\leftarrow \rho v - \alpha \overrightarrow{\nabla} L(W + \rho v, X) \\ W &\leftarrow W + v \end{aligned}$$

This formulation is somewhat inconvenient because it requires calculating the gradient and performing the update at different values of W. However, by reparameterizing $\tilde{W} = W + \rho v$, the equations then become

$$\begin{aligned} v &\leftarrow \rho v - \alpha \overrightarrow{\nabla} L(\tilde{W}, X) \\ \tilde{W} &\leftarrow \tilde{W} + v + \rho(v - v_0), \end{aligned}$$

where v_0 refers to the "old" value of v (i.e. the v on the right side of the first assignment equation).

7.3 Adagrad and RMSProp

Adagrad ("Adaptive Gradient") is different method for dealing with these issues. Instead of pushing movement in the direction of a relatively stable momentum vector, Adagrad attempts to normalize the elements of the gradient so that each parameter update makes roughly equal "progress" on every parameter. Formally, the update operation proceeds as follows (copying the same variable definitions from earlier):

$$\begin{aligned} \nu &\leftarrow \nu + (\overrightarrow{\nabla} L(W, X))^2 \\ W &\leftarrow W - \frac{\alpha}{\sqrt{\nu} + \epsilon} \overrightarrow{\nabla} L(W, X). \end{aligned}$$

As with v before, here ν is initialized to 0. Here vector multiplications are elementwise, and operations between vectors and scalars are broadcasted elementwise. Also, ϵ is a hyperparameter, usually set to 10^{-7}, whose purpose is to prevent division by zero; its exact value doesn't usually impact things much. The impact of the division by $\sqrt{\nu}$ is that step sizes in directions with large gradients are reduced, and step sizes in directions with small gradients are amplified, which is how Adagrad achieves similar rates of progress on each parameter during training.

One problem with Adagrad, however, is that ν can only grow over time, and step sizes are (roughly) proportional to $1/\nu$. While this might not be such a bad thing - we should probably be reducing our step sizes as we train for more steps and get closer to the convergence point - in practice this causes Adagrad to often get stuck around saddle points. A similar algorithm called RMSProp ("Root Mean Squared Propagation") gets around this issue by introducing an exponential decay into ν:

$$\begin{aligned}\nu &\leftarrow \rho\nu + (1-\rho)(\overrightarrow{\nabla} L(W,X))^2 \\ W &\leftarrow W - \frac{\alpha}{\sqrt{\nu}+\epsilon}\overrightarrow{\nabla} L(W,X).\end{aligned}$$

In practice, Adagrad is almost never used, and RMSProp is favored instead.

7.4 Adam

Adam is an optimization algorithm that essentially combines the principles of both momentum and Adagrad/RMSProp. The update step proceeds as follows (elementwise vector arithmetic, broadcasted scalar-vector arithmetic):

$$\begin{aligned}m_1 &\leftarrow \beta_1 m_1 + (1-\beta_1)\overrightarrow{\nabla} L(W,X) \\ m_2 &\leftarrow \beta_2 m_2 + (1-\beta_2)(\overrightarrow{\nabla} L(W,X))^2 \\ m_1' &\leftarrow m_1/(1-\beta_1^t) \\ m_2' &\leftarrow m_2/(1-\beta_2^t) \\ W &\leftarrow W - \frac{\alpha m_1'}{\sqrt{m_2'}+\epsilon}\end{aligned}$$

Here β_1 and β_2 are hyperparameters analogous to ρ from momentum and RMSProp (respectively) and ϵ is analogous to ϵ from RMSProp. One notable difference about Adam is the construction and usage of m_1' and m_2'; the theoretical reasoning can be found in the paper (Kingma & Ba 2015), but the practical outcome of this is that the magnitudes of the initial update steps are kept reasonable - since β_2 is usually set very close to 1, m_2 will be very small initially, which would make a division of something by $\sqrt{m_2}$ potentially very large.

In practice, using Adam with $\beta_1 = 0.9, \beta_2 = 0.999, \alpha \in \{5 \times 10^{-4}, 1 \times 10^{-3}\}$ tends to work well for most problems.

7.5 Other optimization techniques

One interesting optimization algorithm relies on the idea of quadratic approximation. All the algorithms discussed thus far rely on linear approximation - we create a linear approximation of the loss function (using the gradient) and move a little bit in a direction that minimizes it. With a quadratic approximation, we can move directly to the minimum of the approximation!

The naive application of this idea is called Newton's method, but it doesn't work very well in practice since it requires calculating the Hessian matrix (matrix of second derivatives) of the loss function, which has size $N \times N$ (N being the number of trainable parameters of the model), and then inverting it (an $O(N^3)$ operation). When N is much more than 10^6, this is clearly impractical. BFGS and L-BFGS are popular approximations of Newton's method that are far more reasonable in terms of their resource consumption, but it turns out they just don't work that well in practice when faced with stochastic/non-convex settings (like training neural networks on complex datasets), so people don't tend to use them much.

Another popular optimization technique orthogonal to the optimization algorithms discussed thus far is learning rate decay - basically, adjusting the learning rate (usually decreasing it) over time during training. This makes intuitive sense - in general, learning rate tends to be a highly impactful hyperparameter, and there's no reason why the best value of α should be constant throughout the entire training process. There isn't a single best decay schedule that works in all settings; people tend to just try a bunch of things (periodic step function decay, logistic/exponential decay, cosine "decay", etc.) and see what works. As a brief note: tuning the decay schedule is like tuning a hyperparameter, but it's typically done after tuning the learning rate rather than done concurrently with it.

7.6 How do we actually get good test results?

All of the previous techniques discussed in this lecture focus on minimizing our training loss, but recall that our true goal is to optimize our results at test time on unseen data. One of the simplest and most reliable ways to improve test results is ensembling - essentially, training multiple models and combining (e.g. by averaging or majority vote) their results. An interesting way to do this is to simply ensemble multiple snapshots of a single model (from when it is being trained) rather than actually retraining the model several times, and research has been done into how best to actually do this (e.g. Huang et al. 2017).

To improve single-model test perormance, however, we usually turn to regularization. A common pattern of regularization is that we introduce some amount of randomness during training time (to prevent overfitting and force generalization) and average out the randomness at test time. Popular regularization techniques include:

- Dropout (Baldi 2013 is a good explainer). This technique involves randomly setting neurons' outputs to zero ("drop them out") during training with some predetermined probability p. During test time, outputs are not dropped, but are scaled by a factor of p so that the sum of the inputs to each neuron remains similar in magnitude.
- Data augmentation is especially common for image processing models - basically, during training time, instead of just training on the original images, apply random crops, scales, flips, rotations, and (sometimes) color jitter to them.
- Batch norm (Ioffe & Szegedy 2015, also see Lecture 6 notes)

More exotic techniques include:

- DropConnect (Wan et al. 2013) is similar to Dropout, but instead of randomly zeroing neuron activations, it randomly zeroes model weights.
- Fractional max pooling (Graham 2014)
- Stochastic depth (Huang et al 2016)

7.7 Transfer learning

Another problem we must commonly deal with is a lack of training data - while images are plentiful, well-labeled images are not, and training a high-quality CNN from scratch requires millions of well-labeled images. However, we can get around this problem with transfer learning: we can take a model that has been trained on ImageNet, replace its last layer (which predicts the most likely ImageNet class) with something that predicts the result that we are interested in, and then train only the last layer of the model on our data while keeping the rest of the model frozen. This actually tends to work quite well for many applications where training data is limited (e.g. X-ray image reading). If you have enough data, you can also unfreeze more layers of the model and finetune those as well (quick note: typically it's good to use a lower learning rate during fine-tuning, since the pre-trained model will usually have reasonable parameter values already).