CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 10

1 Lecture

Recap

RNNs are specialized for processing sequences of data and are capable of producing sequences of outputs. From one iteration to the next, there's some output that's held and delayed by one time step so it can become the input at the next step.

If we're operating over a long period of time, we'd ideally like to remember some undistorted input (similar to a ResNet where we shortcut across two layers and have the identity map between them). An LSTM RNN does this, and therefore has two recurrent inputs (cell input and "hidden" input). The *cell input* is not transformed, merely elementwise weighted and remembered from one time step to the next (we can also add additional inputs to it). This becomes the memory for the LSTM – although an LSTM can simulate a general RNN through its other recurrent input, it will typically want to use the cell state for longer-term memory (because it's undistorted).

LSTMs are capable of remembering high-level structure and state over many time steps. They are both more powerful and easier to train.

Visualizing Representations

t-SNE is a general-purpose clustering method that allows us to visualize very high-dimensional datasets (or activations) in two or three dimensions. For example, we might want to pick a certain layer of a CNN, take its 4096-dimensional code for many images, and cluster those vectors using t-SNE.

t-SNE is "stochastic network embedding," and involves embedding high-dimensional points in a way that preserves the distance measure between points in the high-dimensional space. Things that are close in the high-dimensional space should appear close in the plane (if we're mapping to a plane). Things that are far apart can end up wherever.

Visualizing CNNs

With CNNs, it's good to visualize the filter weights – however, these are only easily representable for the first layer. (After the first layer, the filters aren't reacting to the image directly.)

We can also visualize via **ablation** (eliminating or cutting out some kind of input). One way to understand the relevance of some patch in an image is to remove it, then measure the change in output. Note: we should replace the ablated portion of the image with the mean value.

Network-Centric Approaches

One of the main lines of research in understanding networks has been network-centric visualizations. In network-centric visualization, we try to understand what's going on with the pixels without directly using the input images to help understand the network. (We look at the network itself without having to know that it was trained with a particular set of images.) Our goal is to produce a visual representation of how a pixel in the input image affects a class score on the output.

For example, we might want to find an image (starting from 0) that maximizes an output score. We can do this via gradients. Note: in order to produce a sensible image, we'll also want to use L2 regularization; we can maximize the score while simultaneously minimizing the L2 magnitude of the image. (This is also good because the problem is massively underconstrained – we're going from a single output score to a $\sim 40,000$ pixel image.)

$$\arg\max_I S_c(I) - \lambda \|I\|_2^2$$

This method is nice because it generalizes the idea of just showing the weights for the first layer of filters to other layers of filters. It involves an activation that grows with the sensitivity to the image, but is also balanced by the L2 norm. In the general case, it has the property that the activation at a certain pixel is equal to the gradient of the score at that pixel.

We'll be optimizing the image with respect to the output scores vector.

Image-Centric Approaches

The goal of network-centric approaches is to intrinsically represent the sensitivity of neurons in different layers of a network, by producing an image to capture the sensitivity. *Image*-centric approaches use actual images, with the aim of producing more intuitive and/or interpretable visualizations.

For example, instead of generating an image from scratch, we might want to start with a seed image that's in the class that we're interested in. We can activate the network with a particular image and then look at the gradients of that network in that activation state.

For a particular image, we can then view the pixels that are most sensitive in terms of conditioning the output (in terms of the gradients of the output score).

We can also feed an image into a network, then propagate activations backward using a deconvolutional network ("deconvnet"). A deconvnet is a CNN model that uses the same components but in reverse – so instead of mapping pixels to features, it would do the opposite.

Guided backpropagation is an improved deconv method which will give us even better (e.g. more interpretable) saliency maps.

We can attempt to take the entire set of activations from the (intermediate) output of some network and see if we can invert the network. This will tell us how much information the network is really capturing about an image in its output layers. To do this, we define an optimization problem of finding an image that most closely produces the observed vector output, and also has low loss according to some regularizers.

Another way of performing code inversion is to directly train a network to map features to image.

Note: we can use gradients to produce highly activating images, but many of these images don't actually look like anything... therefore CNNs can be "fooled!"