CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 5

1 Reading

Convolutional Neural Networks

SOURCE: CS 231N

CNN architectures make the assumption that the inputs are images, which allows us to make the forward function more efficient and vastly reduce the amount of parameters in the network.

Other than that, they're still just neural networks: they're composed of neurons with learnable weights and biases, where each neuron receives some inputs, performs a dot product, and optionally follows it with a nonlinearity. In the context of image classification, the network as a whole will express a single function from raw image pixels to class scores.

Regular neural networks receive an input as a single vector and transform it through a series of hidden layers. (Each neuron in a hidden layer will be fully connected to all neurons in the previous layer.) However, the fully connected structure does not scale well to larger images. For example, if the input were an image of size $200 \times 200 \times 3$, a single fully connected neuron in the first layer of a network would have $200 \cdot 200 \cdot 3 = 120,000$. And we would have several such neurons, so the number of parameters would add up extremely quickly! Such a large number of parameters is wasteful and prone to overfitting.

Accordingly, CNNs constrain the architecture in a more sensible way. The layers of a CNN have neurons arranged in three dimensions: width, height, and depth. For example, each CIFAR-10 image is an input volume of activations with dimensions $32 \times 32 \times 3$ (width, height, and depth respectively). The neurons in a CNN layer will only be connected to a small region of the previous layer, as opposed to *all* of the neurons in the previous layer. The final output layer for CIFAR-10 would have dimensions $1 \times 1 \times 10$, because by the end of the CNN architecture we would reduce the full image into a single vector of class scores, arranged along the depth dimension.

Each layer of a CNN transforms a 3D input volume into a 3D output volume of neuron activations, using some differentiable function that may or may not have parameters.

The three main types of layers in a CNN are the **convolutional layer**, the **pooling layer**, and the **fully connected layer** (exactly as seen in regular neural networks). For example, a simple CNN for CIFAR-10 might have the architecture

- **INPUT** the raw pixels of the image ($32 \times 32 \times 3$ if using the CIFAR-10 dataset).
- **CONV** a layer which computes the output of neurons that are connected to local regions in the input. Each neuron will compute a dot product between its weights and a small region it is connected to in the input volume. If we were to use 12 filters, the output volume might be $32 \times 32 \times 12$.
- **RELU** a layer which applies an element-wise activation function such as $\max(0, x)$.
- **POOL** a layer which performs a downsampling operation along the (width, height) dimensions.
- **FC** a layer in which each neuron is connected to all of the neurons in the previous volume.

Note that some layers (CONV, FC) involve parameters, while others (RELU, POOL) do not. The parameters in the CONV and FC layers will be trained with gradient descent so that the computed class scores agree with the labels for each image.

Convolutional Layer

The *convolutional* layer is the core building block of a *convolutional* neural network. Its parameters consist of a set of learnable filters – each is small spatially (along width and height), but extends through the full depth of the input volume.

For example, a filter on a first layer might have size $5 \times 5 \times 3$. During the forward pass, we would slide (more precisely convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the entries of the input. As we slide the filter, we will produce a 2D activation map that gives the responses of the filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature, like a blotch of color on the first layer or an entire honeycomb on a later layer.

In each CONV layer we will have an entire set of filters (e.g. 12 filters), and each of them will produce a separate 2D activation map. We will stack these activation maps along the depth dimension in order to produce the output volume.

Every entry in the 3D output volume can be interpreted as the output of a neuron that looks at a small region in the input and shares parameters with all neurons to the left and right spatially (because these numbers all result from applying the same filter).

The spatial extent of each neuron's connectivity (to a local region of the input volume) is called the **receptive field** or **filter size** of the neuron. Note that connections are local in space (along width and height), but are always full along the depth of the input volume.

For example, if the input volume were of size $32 \times 32 \times 3$ and the filter size were 5×5, *each neuron* in the layer would have weights to a $5 \times 5 \times 3$ region in the input volume (for a total of $5 \cdot 5 \cdot 3 = 75$ weights, plus one bias parameter). Note that the connectivity is local in space (5×5) but full along the depth (3).

Neurons still compute a dot product of their weights with the input, followed by a nonlinearity. The difference is that their connectivity is now restricted to be local spatially. Specifically, the output of a neuron is

$$f\left(\sum_i w_i x_i + b\right)$$

where f is a nonlinearity function and i ranges over the relevant local input region.

Three output parameters control the size of the output volume: **depth**, **stride**, and **zero-padding**.

- **Depth.** This corresponds to the number of filters we would like to use; each will [hopefully] learn to look for something different in the input. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column**.
- **Stride.** This refers to the amount by which we slide the filter. When the stride is 1, we move the filters one pixel at a time. When the stride is 2, the filters jump two pixels at a time as they slide around. A larger stride will produce smaller output volumes spatially.
- **Zero-padding.** Sometimes it will be convenient to pad the input volume with zeros around the border; this hyperparameter refers to the size of this padding. Zero padding allows us to control the spatial size of the output volumes. For instance, setting zero padding to be $P = (F - 1)/2$ when the stride is $S = 1$ ensures that the input and output volume will have the same size spatially.

We can compute the spatial size of a CONV layer's output volume as a function of the input volume size W, the receptive field size F, the stride with which they are applied S, and the amount of zero padding P used on the border. The number of neurons that "fit" along one dimension (i.e. when strided) is

$$(W - F + 2P)/S + 1$$

Hence for a 7×7 input and a 3×3 filter with stride 1 and padding 0 we would get a 5×5 output. (Here $W = 7$, $F = 3$, $P = 0$, and $S = 1$ for each of the x- and y-dimensions, so we have $(7 - 3 + 0)/1 + 1 = 5$ neurons along both spatial dimensions.)

Each 2D activation volume is created by a single filter (one set of weights) sliding across the image.

Therefore it is possible to compute the forward pass of a CONV layer as a **convolution** of the neuron's weights with the input volume for each depth slice.

Also, since weights are shared within a depth slice, during backpropagation the gradients will be added up across all the neurons in a depth slice and we'll only update a single set of weights per slice.

The justification for sharing weights is that if it's useful to detect a feature at one location in the image, it's probably also useful to detect that feature at another location in the image.

To summarize, a convolutional layer

- accepts a volume of size $W_1 \times H_1 \times D_1$
- requires four hyperparameters: number of filters K, their spatial extent F, the stride S, and the amount of zero padding P
- produces a volume of size $W_2 \times H_2 \times D_2$, where $\{W, H\}_2 = (\{W, H\}_1 - F + 2P)/S + 1$ and $D_2 = K$
- introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- performs a valid convolution of the dth filter with the input volume, in the process of computing the dth depth slice (of size $W_2 \times H_2$) of the output volume

Note: FC and CONV layers consist of the same operations (only difference: CONV neurons share parameters and are connected to *local* input regions), so can convert between CONV and FC. To convert FC to CONV, reshape to $w \times h \times d$ (e.g. $1 \times 1 \times 4096$) and run a convolution with `numOutputs` filters, each covering the entire input volume ($w \times h$, e.g. 1×1). This works because each filter extends through the whole depth.

Pooling Layer

A pooling layer's function is to progressively reduce the spatial size of the representation in order to reduce the amount of parameters and computation in the network, and by extension to control overfitting. It operates independently on every depth slice of the input and resizes it spatially using a MAX operation.

The most common form of a pooling layer has filters of size 2×2 applied with stride 2, which downsamples every depth slice in the input by 2 along both width and height – discarding 75% of the activations. Every MAX operation in this case would thus be taking a max over 4 numbers.

To summarize, a pooling layer

- accepts a volume of size $W_1 \times H_1 \times D_1$
- requires two hyperparameters: the spatial extent F and the stride S
- produces a volume of size $W_2 \times H_2 \times D_2$, where $\{W, H\}_2 = (\{W, H\}_1 - F)/S + 1$ and $D_2 = D_1$
- introduces no parameters, as it only computes a fixed function of the input

In addition to max pooling, we can perform average pooling or L2-norm pooling. However, for its intended purpose max pooling seems to work the best.

The purpose of a pooling layer is to downsample the volume spatially.

Layer Patterns

The most common form of a CNN architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is also common to transition to fully-connected layers. The last FC layer will hold the output, e.g. class scores.

INPUT $\rightarrow$ [[CONV $\rightarrow$ RELU] $*$ N $\rightarrow$ POOL?] $*$ M $\rightarrow$ [FC $\rightarrow$ RELU] $*$ K $\rightarrow$ FC

In the above flowchart, $*$ indicates repetition, and POOL? indicates an optional pooling layer.

2 Lecture

Recap

In practice, because a lot of datasets are quite large, it's more effective to take many small steps (that are noisy) than a few large steps (that are more accurate). This is the basis of SGD.

Backpropagation

Backpropagation is really just an application of the chain rule in an efficient form.

Chain rule: the basic idea of calculus is that when functions are sufficiently smooth, everything is locally linear. So if we want to see how loss changes with respect to some weights for a single-valued function, it's enough to look at how loss changes with respect to the function and then how the function changes with respect to the weights.

$$\frac{dL}{dW} = \frac{dL}{df}\frac{df}{dW}$$

We then have

$$\nabla_W L = \frac{dL}{df}\nabla_W f$$

for W a vector of parameters. This is really just the first rule applied to all of the partial derivatives w.r.t. elements of W.

Generally, the functions in a neural network are all vector-valued. If f is vector-valued with an input vector W, and we have multiple variables depending on W, we have to consider all of the possible paths through which changes in W could affect L. So we have to sum over the intermediate coordinates of the function f in order to compute the complete derivative of $\partial L/\partial W$. In other words,

$$\frac{\partial L}{\partial W_j} = \sum_{i=1}^{k} \frac{\partial L}{\partial f_i}\frac{\partial f_i}{\partial W_j}$$

for $i = 1, ..., m$. This can be written as a matrix multiplication

$$J_L(W) = J_L(f)J_f(W)$$

where $J_f(W)$ is a Jacobian matrix (the derivative of a vector-valued function with a vector-valued input) such that

$$J_f(W)_{ij} = \frac{\partial f_i}{\partial W_j}$$

Just as df/dx is a function of x (it changes with x), the Jacobian $J_f(W)$ is a function of the input W.

The Jacobian generalizes the gradient of a scalar-valued function f to a k-valued function. If we think of f as a layer with m inputs and k outputs, the Jacobian will have dimensions $k \times m$, i.e. $n_{outputs} \times n_{inputs}$.

$$J_f(W) = \begin{bmatrix} \frac{\partial f_1}{\partial W_1} & \frac{\partial f_1}{\partial W_2} & \cdots & \frac{\partial f_1}{\partial W_m} \\ \frac{\partial f_2}{\partial W_1} & \frac{\partial f_2}{\partial W_2} & \cdots & \frac{\partial f_2}{\partial W_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial W_1} & \frac{\partial f_k}{\partial W_2} & \cdots & \frac{\partial f_k}{\partial W_m} \end{bmatrix}$$

If we have several vector-valued functions A, B, C, ... composed in a chain (e.g. a deep network),

$$W \to A \to B \to C \to \ldots \to L$$

i.e. (W is an argument to A, whose output is an argument to B, and so on)

$$L(W) = L(K \cdots C(B(A(W))) \cdots)$$

we can multiply Jacobians in order to get the gradient:

$$J_L(W) = J_L(K) \cdots J_C(B) \cdot J_B(A) \cdot J_A(W)$$

and $J_L(W) = (\nabla_W L)^T$, which is the gradient we need in order to minimize loss over W!

Note that Jacobians are matrices, and matrix multiplication is associative. We can actually evaluate the product of Jacobians in any order. Backpropagation will choose to evaluate the Jacobian product (loss gradient w.r.t. params) left-to-right, i.e. from the output of the neural network toward its input. This way, we can perform matrix-vector multiplications instead of matrix-matrix multiplications.

In summary:

- We first make a forward pass, computing activations from the first layer to the last layer.
- Then backprop computes derivatives of the loss w.r.t. other layers, from the last layer to the first.
- Backpropagation only requires matrix-vector multiplications, and paths from the loss layer to inner layers are reused.

Convolutional Neural Networks

With images, inputs are very big. Hence the naive approach (fully connected layers) becomes intractable. Plus, spatial correlation is local! It's a waste of resources to have each neuron connect to every single pixel in its input image. *The local characteristics of an image provide cues to higher-level features. We don't need to look at the entire image all at once. We can look at little patches of the image and build up progressively more complex feature maps.*

In other words, we can take shift-invariant filters (meaning we apply the same operator at each position) and shift them across the image (applying the filter weights to the underling pixels) to produce a feature map. Algebraically, this operation is known as a **convolution**:

$$g(x, y) = (h * f)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(i, j) f(x - i, y - j)$$

There are two functions of x and y, h and f, which are convolved to produce another function g of x and y. We enumerate over the coordinates of h, and take f at the output coordinates (x, y) and subtract the coordinates used to compute f.

Technically, the true convolution operation also flips the filter in x and y before applying it. There are a few reasons for this, which aren't very interesting.

Note: the convolution of h with f is the same as the convolution of f with h. In other words, convolution is commutative:

$$h * f = f * h$$

Also, we obviously don't want to sum over the infinite plane. So typically we consider a square region (it doesn't have to be square – it could be width by height – but let's call it square) and center it around the output pixel. Then one set of coefficients (the filter h) only has to be defined over a relatively small range $[-w/2, w/2]^2$. Now, if we call f the input image we only need to look at a small collection of its pixels:

$$(h * f)(x, y) = \sum_{i=-w/2}^{w/2} \sum_{j=-w/2}^{w/2} h(i, j) f(x - i, y - j)$$

To convolve a filter with an image, we slide it over the image spatially (i.e. in the x and y but not depth dimensions) and compute dot products. Note that a filter must have the same number of depth dimensions

as the input image. Each dot product yields one number, so a $32 \times 32 \times 3$ image convolved with a $5 \times 5 \times 3$ filter would be a $28 \times 28 \times 1$ activation map. We will have many such activation maps; this gives depth to intermediate results. (If we had six filters in our previous example, we could stack them up to get a "new image" of size $28 \times 28 \times 6$.)

We *stride* (by more than 1) because otherwise adjacent filters have very similar views of the image and become redundant. By stepping further – and potentially adding more filters – we get more independent/diverse views of the image.

When we zero pad the border, we should first subtract out the image mean so that we get a symmetric mean zero image coming in (and the zeros we add are therefore expected values).

By using odd-size filters, we can symmetrically pad the image and preserve its shape. In general, it's common to use stride 1, filters of size $F \times F$, and zero-padding of $(F-1)/2$ pixels. This will spatially preserve the size of the image.

Pooling makes the network a little less sensitive to the specific location of things. Formally: "by pooling filter responses at different locations we gain robustness to the exact spatial location of features." Commonly, it's used with a stride to further reduce the size of the input. It also operates over each activation map independently.

References

[1] CS 231n convnet notes,
`http://cs231n.github.io/convolutional-networks/`