CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 3

This discussion will cover some practice applying backpropagation, and introduce the convolution operator.

1 Mechanical Backpropagation

In this section, we will work through some calculations used during backpropagation.

Recall the softmax function $\mathbf{p} : \mathbb{R}^k \to \mathbb{R}^k$, with entries given by

$$p_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}.$$

Each entry p_i corresponds to the probability assigned to the label i. We derived in Discussion 1 that the partial derivatives of $p_i(\mathbf{z})$ for each entry of $\mathbf{z}$ is given by,

$$\begin{aligned}\frac{\partial p_i(\mathbf{z})}{\partial z_j} &= \begin{cases} p_i(\mathbf{z})(1 - p_j(\mathbf{z})) & \text{if } i = j \\ -p_i(\mathbf{z})p_j(\mathbf{z}) & \text{if } i \neq j \end{cases} \\ &= p_i(\mathbf{z})(\delta_{ij} - p_j(\mathbf{z})).\end{aligned}$$

We can then concicsely write the full gradient with respect to $\mathbf{z}$ as

$$\nabla p_i(\mathbf{z}) = p_i(\mathbf{z})(\mathbf{e}_i - \mathbf{p}(\mathbf{z})),$$

where $\mathbf{e}_i$ is the unit vector with 1 at index i and 0 elsewhere.

In this example, we will maximize the log-likelihood of the given labels in our dataset, which motivates the following loss for a multiclass logistic regression model.

$$L(\mathbf{x}, y, W, \mathbf{b}) = -\log p_y(W\mathbf{x} + \mathbf{b}).$$

Problem 1: Gradient with respect to linear layer parameters

Utilize the chain rule to compute the gradient of $L(\mathbf{x}, y, W, \mathbf{b})$ with respect to W and $\mathbf{b}$.

Solution 1: Gradient with respect to linear layer parameters

Let $\mathbf{z} = W\mathbf{x} + \mathbf{b}$. Given we already know $\boldsymbol{\nabla} p_i(\mathbf{z})$, we first compute $\boldsymbol{\nabla} \log p_i(\mathbf{z})$ as

$$\begin{aligned}\boldsymbol{\nabla} \log p_i(\mathbf{z}) &= \frac{\boldsymbol{\nabla} p_i(\mathbf{z})}{p_i(\mathbf{z})} \\ &= \mathbf{e}_i - \mathbf{p}(\mathbf{z}).\end{aligned}$$

We note that since our loss is the *negative* log likelihood, we will need to flip the sign of our gradient. We first consider the gradient of the loss with respect to the bias $\mathbf{b}$. Notice that,

$$\frac{\partial z_i}{\partial b_j} = \delta_{ij},$$

and so the Jacobian $\frac{\partial \mathbf{z}}{\partial \mathbf{b}}$ is simply the identity matrix. Utilizing the chain rule to compute the gradient with respect to the bias, we have

$$\begin{aligned}\boldsymbol{\nabla}_{\mathbf{b}} L &= -\frac{\partial \mathbf{z}}{\partial \mathbf{b}} \boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \\ &= -I \boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \\ &= -\boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \\ &= \mathbf{p}(\mathbf{z}) - \mathbf{e}_y.\end{aligned}$$

Now, we consider the partial derivatives of $\mathbf{z}$ with respect to W.

$$\frac{\partial z_k}{\partial W_{ij}} = \delta_{ik} x_j.$$

Noting that the derivative with respect to W_{ij} depends only on the i'th entry of $\mathbf{z}$, we compute

$$\begin{aligned}\frac{\partial L}{\partial W_{ij}} &= \frac{\partial L}{\partial z_i} x_j \\ \frac{\partial L}{\partial W} &= -\boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \mathbf{x}^T \\ &= (\mathbf{p}(\mathbf{z}) - \mathbf{e}_y)\mathbf{x}^T.\end{aligned}$$

Suppose now that we had a multilayer neural network and $W, \mathbf{b}$ were the the parameters of the last layer of the network. To compute gradients of the earlier parameters of the network with backpropagation, we also need to compute the gradient of the loss with respect to $\mathbf{x}$ and pass it backwards.

Problem 2: Gradient with respect to input

Utilize the chain rule to compute the gradient of $L(\mathbf{x}, y, W, \mathbf{b})$ with respect to $\mathbf{x}$.

Solution 2: Gradient with respect to input

We can again compute

$$\begin{aligned}\frac{\partial L}{\partial x_j} &= \sum_{i=1}^{k} \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial x_j} \\ &= \sum_{i=1}^{k} \frac{\partial L}{\partial z_i} W_{ij}.\end{aligned}$$

Vectorizing, we obtain

$$\frac{\partial L}{\partial \mathbf{x}} = -W^T \boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z})$$

Having computed these, one then simply needs to also compute the backwards pass through the chosen activation function to able backpropagate through fully-connected feedforward networks!

We'll now move on to a slightly more complicated example of backpropagation involving a *skip connection*, which you'll see again when we cover ResNets.

Problem 3: Gradient in a nonlinear computation graph

Suppose we have $\mathbf{y} = W_2\sigma(W_1\mathbf{x}) + \mathbf{x}$, where σ is the ReLU activation. Letting $\delta_{\mathbf{y}}$ denote the gradient of the loss with respect to $\mathbf{y}$, compute the gradient of the loss with respect to $\mathbf{x}$.

Solution 3: Gradient in a nonlinear computation graph

y here is that $\mathbf{x}$ now contributes to $\mathbf{y}$ both through the $W_2\sigma(W_1\mathbf{x})$ (the usual feedforward pass) and the $\mathbf{x}$ term (the skip connection).
We'll first go tthrough the contribution to the gradient from the feedforward pass. Let $\mathbf{z} = W_1\mathbf{x}$ and $\mathbf{a} = \sigma(W_1\mathbf{x})$. From our earlier calculations, we see that

$$\frac{\partial L}{\partial \mathbf{a}} = W_2^T \delta_{\mathbf{y}}.$$

We now need to backpropagate through the ReLU activation. Let $R(\mathbf{z})$ denote the diagonal matrix such that $\mathbb{R}(\mathbf{z})_{ii} = 1$ if $\mathbf{z}_i > 0$ and 0 otherwise. We see that the backward pass through the ReLU activation simply multiplies the post-activation gradient by $R(\mathbf{z})$, and so we have

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}} &= R(\mathbf{z})\frac{\partial L}{\partial \mathbf{a}} \\ &= R(\mathbf{z})W_2^T\delta_{\mathbf{y}}.\end{aligned}$$

Finally, we can compute the feedforward pass's contribution to $\frac{\partial L}{\partial \mathbf{x}}$ as

$$W_1^T\frac{\partial L}{\partial \mathbf{z}} = W_1^T R(W_1\mathbf{x})W_2^T\delta_{\mathbf{y}}.$$

To handle the two different paths through to the output, we simply need to sum each gradient contribution, so our final gradient is

$$\frac{\partial L}{\partial \mathbf{x}} = W_1^T R(W_1\mathbf{x})W_2^T\delta_{\mathbf{y}} + \delta_{\mathbf{y}}.$$

2 Convolutional Neural Networks

Convolutional neural networks[1] (CNN) are a type of neural network architecture that have become the key ingredient for state of the art modern computer vision performance.

They perform operations similar to feed-forward neural networks that we have discussed, but explicitly account for spatial structure in the data, and so are very common for computer vision tasks where inputs are images. That said, CNNs can also be applied to non-image data with similar structure in the input, such as time series or text data (in which case they're takign advantage of temporal structure).

2.1 Convolution (Cross-Correlation) Operator

At the heart of CNNs is the convolution operator. In this discussion, what we refer to as a convolution is actually the **cross-correlation** operator here instead, which is the exact same but with the indexing of the weights in $\mathbf{w}$ inverted. For example, "convolutional" layers in the deep learning library Pytorch are also actually cross-correlations instead, and homework 1 will also similarly have you implement cross-correlation instead of the actual convolution.

To motivate the use of convolutions, we will work through an example of a 1-D convolution calculation to illustrate how convolutions work over a single spatial dimension. Suppose we have an input $\mathbf{x} \in \mathbb{R}^n$, and filter $\mathbf{w} \in \mathbb{R}^k$. We can compute the convolution of $\mathbf{x} \star \mathbf{w}$ as follows:

1. Take your convolutional filter $\mathbf{w}$ and align it with the beginning of $\mathbf{x}$. Take the dot product of $\mathbf{w}$ and the $\mathbf{x}[0 : k-1]$ (using Python-style zero-indexing here) and assign that as the first entry of the output.
2. Suppose we have stride s. Shift the filter down by s indices, and now take the dot product of $\mathbf{w}$ and $\mathbf{x}[s : k-1+s]$ and assign to the next entry of your output.
3. Repeat until we run out of entries in $\mathbf{x}$.

Below, we illustrate a 1D convolution with stride 1.

$$\overbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{bmatrix}}^{\text{Input vector } \mathbf{x} \in \mathbb{R}^n} \star \overbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix}}^{\text{Convolutional filter } \mathbf{w} \in \mathbb{R}^k} = \overbrace{\begin{bmatrix} \sum_{i=1}^k w_i x_i \\ \sum_{i=1}^k w_i x_{i+1} \\ \vdots \\ \sum_{i=1}^k w_i x_{i+n-k} \end{bmatrix}}^{\text{Output vector } \mathbf{y} \in \mathbb{R}^{n-k+1}}$$

We see that the output vector is smaller than the input vector ($\mathbb{R}^{n-k-1}$ compared to $\mathbb{R}^n$). A common way to address this is **zero-padding**, in which we append zeros on both ends of the input vector before applying the convolution (note that there are other conventions for zero-padding as well).

Often, we'll be dealing with multiple spatial dimensions (2 spatial dimensions in the case of images). In this case, we would need to slide our filter along all spatial dimensions to construct the output.

[1] Recommended reading: http://cs231n.github.io/convolutional-networks/

Problem 4: Test your know knowledge of convolution dimensions

In this problem, we will run a series of convolution-related operations to better understand how dimensions are affected by convolutions.

1. Suppose you have a $32 \times 32 \times 3$ image (a 32×32 image with 3 input channels). What are the resulting dimensions when you convolve with a $5 \times 5 \times 3$ filter with stride 1 and 0 padding?
2. What if we zero-pad the input by 2?
3. Suppose we now stack 10 of these $5 \times 5 \times 3$ filters and continue to zero pad the input by 2. What is the new shape of the output, and how many parameters are in our filters (not including any bias parameters)?
4. What would be the spatial dimensions after applying a 1×1 convolution? Think about what this does.

Solution 4: Test your know knowledge of convolution dimensions

1. The resulting spatial dimensions are 28×28 (with one output channel).
2. The resulting spatial dimensions are 32×32, so we have preserved the same size as the input image.
3. The resulting outputs are $32 \times 32 \times 10$, with 10 output channels. There are $5 \cdot 5 \cdot 3 = 75$ parameters per filter, so with 10 filters, we have 750 parameters in this layer. Note that, if we did choose to include a bias parameter, then there would be 76 parameters per filter, and so 760 in total.
4. A 1×1 convolution does not change the spatial dimensions. For every spatial location, it performs a linear map of the the input channels pointwise over space. In practice, this is useful for changing the number of channels.

2.2 Convolutions as Matrix Multiplication

We note that convolutions are a linear operation. Recalling linear algebra, any linear map (between finite-dimensional spaces) can be expressed as a matrix, so we will see in this section how to write a convolution as a matrix multiplication.

Problem 5: Expressing convolutions as matrix multiplication

We shall again consider a 1D convolution. Consider an input $\mathbf{x} \in \mathbb{R}^4$ and filter $\mathbf{w} \in \mathbb{R}^3$. Letting $\bar{\mathbf{x}}$ denote the result of zero-padding the input by 1 on each end, what is the matrix W such that

$$\overbrace{W}^{\mathbb{R}^{4\times 6}} \overbrace{\begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}}^{\text{Zero padded input } \bar{\mathbf{x}} \in \mathbb{R}^6} = \bar{\mathbf{x}} * \mathbf{w}?$$

Solution 5: Expressing convolutions as matrix multiplication

Computing the convolution, we see that

$$\bar{\mathbf{x}} * \mathbf{w} = \begin{bmatrix} x_1 w_2 + x_2 w_3 \\ x_1 w_1 + x_2 w_2 + x_3 w_3 \\ x_2 w_1 + x_3 w_2 + x_4 w_3 \\ x_3 w_1 + x_4 w_2 \end{bmatrix}.$$

Writing this out as a matrix multiplication, we obtain

$$W = \begin{bmatrix} w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \end{bmatrix}.$$

We can observe now that the resulting matrix will be very sparse (most entries are 0) if the filter size is much smaller than the input size, corresponding to the fact that such convolutions exploit spatial locality. We also observe that there is a lot of parameter reuse, as the convolutional filter weights are repeated many times throughout the explicit matrix.

This has several implications. First of all, this implies that convolutional layers are less expressive than fully-connected layers (as fully connected layers are represented by arbitrary matrices).

Another important implication stems from the fact that we have very optimized tools for computing matrix multiplications. While a naive implementation of a convolution will require looping over all the spatial dimensions, it will turn out that reformulating the convolution as a matrix multiplication will often be much faster due to these optimizations (for example, the Cythonized im2col function in part 4 of homework 1 essentially does this).

2.3 Backwards Pass for a Convolution

We'll consider the same 1D convolution as before, but without zero-padding for simplicity.

Problem 6: Backwards pass for convolutions

Let $\mathbf{y} = \mathbf{x} * \mathbf{w} \in \mathbb{R}^2$, where $\mathbf{w} \in \mathbb{R}^3$, $\mathbf{x} \in \mathbb{R}^4$. Let $\nabla_{\mathbf{y}} L$ denote the gradient of the loss with respect to the output of the convolution. Compute the gradients of L with respect to $\mathbf{x}$ and $\mathbf{y}$. Can you express the gradients as convolutions themselves?

Solution 6: Backwards pass for convolutions

Let $\delta_i = \frac{\partial L}{\partial y_i}$. We can explicitly write out the partial derivatives with respect to each entry of $\mathbf{x}$.

$$\begin{aligned}
\frac{\partial L}{\partial x_1} &= w_1\delta_1 \\
\frac{\partial L}{\partial x_2} &= w_2\delta_1 + w_1\delta_2 \\
\frac{\partial L}{\partial x_3} &= w_3\delta_1 + w_2\delta_2 \\
\frac{\partial L}{\partial x_4} &= w_3\delta_2
\end{aligned}$$

We recognize this as convolution where we zero pad δ by 2 on each end, and convolve with the filter $\tilde{\mathbf{w}}$, where $\tilde{\mathbf{w}}$ reverses the entries of the filter $\mathbf{w}$. (Draw this out for students, explain why sliding the filter along means that we should convolve the output derivative with $\tilde{\mathbf{w}}$ instead of $\mathbf{w}$).
Now, we can similarly compute the partial derivatives for $\mathbf{w}$

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \delta_1 x_1 + \delta_2 x_2 \\
\frac{\partial L}{\partial w_2} &= \delta_1 x_2 + \delta_2 x_3 \\
\frac{\partial L}{\partial w_3} &= \delta_1 x_3 + \delta_2 x_4
\end{aligned}$$

We see that $\frac{\partial L}{\partial \mathbf{w}} = \mathbf{x} \star \boldsymbol{\nabla}_{\mathbf{y}} L$ with no zero-padding.