CS 182/282A: Designing, Visualizing and Understanding Deep Neural Networks **Spring 2019**

Section 4: CNNs, Batch Normalization, and Dropout

Notes by: Daniel Seita

4.1 Course Logistics

- Unfortunately we are still not sure if we will be able to change the discussion sections.
- We are currently grading the first assignment and checking the project proposals. We will get back to you as soon as possible.
- Alert: we have a midterm coming up. There is a practice midterm on bCourses, so feel free to look at that.

4.2 Convolutional Neural Networks

Convolutional neural networks[1] are a type of neural network architecture that have become the key ingredient for state of the art modern computer vision performance. Convolutional neural networks are like the "ordinary" feed-forward neural networks that we've discussed, except they make the explicit assumption that the inputs are images. A key caveat: as is often the case with various concepts, there are exceptions. Convolutional neural networks can also be applied to non-image data that have certain structure in the input, such as time series data, but for this class, we mostly focus on when images are input.

4.2.1 The Convolution Operator: Dimensions

We will go through a series of convolution-related operations to test understanding of how dimensions are affected. Note that in all cases, we implicitly assume a minibatch size of one; this is repeated for all elements in a minibatch. [*Note to GSIs*: go through the computations slowly, drawing on the board.]

1. Assume a 32x32x3 image with a 5x5x3 filter with a stride of 1 and padding of 0. What's the output dimension? See Figure 4.1.
2. Now assume the same, except we have 10 of these 5x5x3 filters, and we also use stride 1 (same as before) but with pad of 2 for the input. What size is the output and how many parameters in this layer? See Figure 4.2.
3. Now let's assume we do a 1x1 convolution instead. What is the output dimension after that? How do we implement in matrix multiplication? See Figure 4.3.
4. Finally, consider the pooling operator, a operation with no trainable parameters which helps to reduce the size of the images (increasing the "effective receptive field" of subsequent convolution operators, so they act on a larger image). See what happens in Figure 4.4. Though we note that many modern architectures don't use pooling.

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

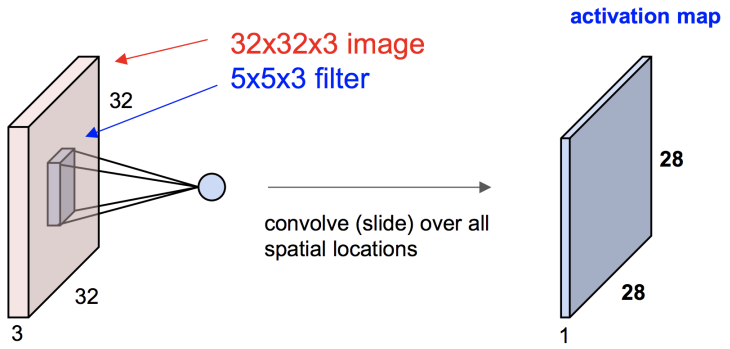

Figure 4.1: An example filter applied to an input image. (From Lecture 5 slides)

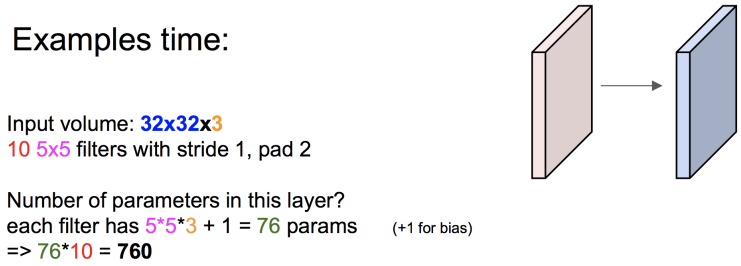

Figure 4.2: An example for determining the number of parameters in the layer (don't forget the bias!). Note that the output volume is 32x32x10, since we pad with 2 (so the first two dimensions are the same) and there are 10 filters (explaining the third dimension). (From Lecture 5 slides)

4.2.2 The Convolution Operator: Matrix Multiplication

The convolution operator can be implemented as matrix multiplication. The advantage of doing so is that there are highly optimized libraries for matrix multiplication operations, which justify the cost of reshaping inputs into matrices.

For example, consider a 1D convolution where we have input vector $\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^T$ and three weight filters w_1, w_2, and w_3. With a stride of 1 and a padding of 1 on the input, we can implement the convolution operator using the following matrix multiplication:

$$\underbrace{\begin{bmatrix} w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \end{bmatrix}}_{W \in \mathbb{R}^{4\times 6}} \underbrace{\begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}}_{\mathbf{x} \in \mathbb{R}^6} = \underbrace{\begin{bmatrix} x_1 w_2 + x_2 w_3 \\ x_1 w_1 + x_2 w_2 + x_3 w_3 \\ x_2 w_1 + x_3 w_2 + x_4 w_3 \\ x_3 w_1 + x_4 w_2 \end{bmatrix}}_{\mathbf{o} \in \mathbb{R}^4} \tag{4.1}$$

or more concisely, $W\mathbf{x} = \mathbf{o}$.

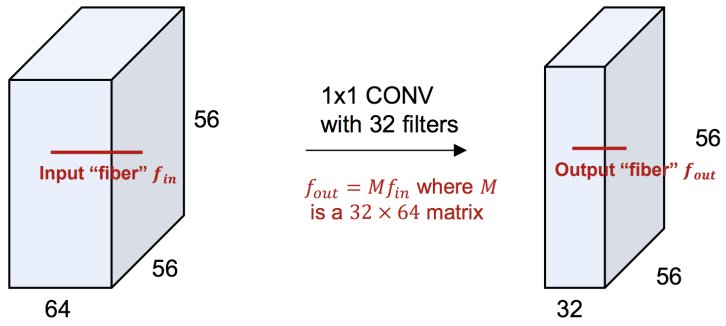

Figure 4.3: An example of a 1x1 convolution. (From Lecture 5 slides)

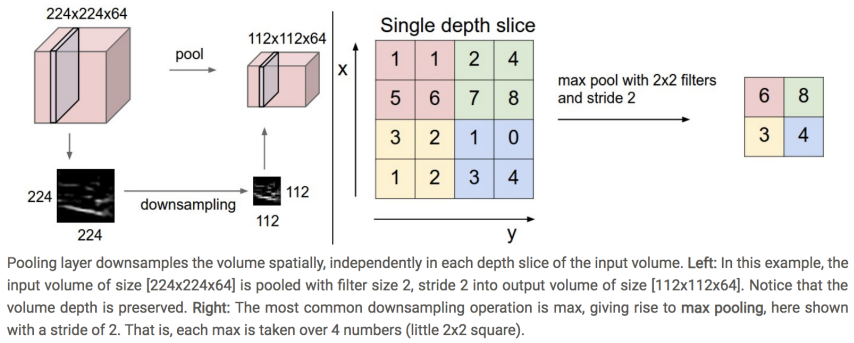

Figure 4.4: An example of a 2x2 max pooling operation. (From CS 231n notes)

4.2.3 Different Architectures

There is a long history of neural network architectures that use convolutions of some form. In the following list, we try and summarize the most important architectures from the literature:

1. **LeNet**. Among the earlier convolutional neural network architectures, LeNet [5] is the most widely known, and was used for early handwritten digit recognition. The general architecture idea of using a series of convolutions and pooling layers, followed by several fully connected layers, is still used in practice today.

2. **AlexNet**. The AlexNet architecture [4] popularized convolutional neural networks in computer vision by winning the ImageNet ILSVRC challenge in 2012. It has a similar architecture to LeNet except it is bigger and deeper, and presented the benefits of ReLUs, GPUs, dropout, and so on. It has about 60 million parameters. A key quote at the end of the first section proved prescient:

   > In the end, the network's size is limited mainly by the amount of memory available on current GPUs and by the amount of training time that we are willing to tolerate. Our network takes between five and six days to train on two GTX 580 3GB GPUs. All of our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available.

3. **GoogLeNet**. The winner of ILSVRC 2014, this network introduced the inception architecture [9] for reducing the number of parameters. The network has only 4 million parameters.

4. **VGG**. This network [7] was the runner-up in ILSVRC 2014 to GoogLeNet, and showed the benefit of (a) increasing the number of layers, and (b) using only convolutional operators stacked on each other. A downside is that this network has roughly 138 million parameters, so in general, consider using Residual Nets (see next item).

5. **Residual Networks**. These networks use skip connections to allow inputs and gradients to propagate faster throughout the network (either forward or backwards). Residual networks [1] were state of the art for image recognition results as of mid-2016, and the general backbone is commonly used as of today (early 2019) for research. They have substantially fewer parameters than VGG. (The exact number depends on what type of "ResNet-X" is used, where "X" represents layers; PyTorch offers pretrained models for 18, 34, 50, 101, and 152 — ResNet-152 should have about 60 million parameters.)

6. **Dense Networks**. Another popular architecture class is the Dense Network (DenseNets) [2], which take the idea of skip connections and applies it further to connect all layers with each other. It was reported that they outperform ResNets, but in practice, DenseNets can be difficult to train due to memory requirements.[2]

To reiterate the last point from the Residual Networks item above, you should not in general be training large, deep convolutional neural networks from scratch. It is better to take a pre-trained model as the "backbone" and then fine-tune the last few layers (perhaps after adjust them to match your desired output dimensions). For example, PyTorch has a suite of pre-trained models here.[3] Also, be aware that for the Inception, ResNet, and DenseNet models, there are multiple variations of them (number of layers, width, etc.) so be careful about which model you're using.

4.3 Localization, Detection, and Segmentation

For most of the class thus far, when we've been talking about applying neural networks in practice, we have assumed an image classification task: given an image, we'd like the network to output the probabilities of the true label belonging to a variety of classes. There are many other tasks in computer vision, though. We introduced a few popular ones last week:

1. **Image Localization** Determine a *bounding box* for an image showing the object in it that determines the class. One object is involved, which is why it's not "object detection" but "image localization" or "classification plus localization" — we know ahead of time that there is only one object class of interest in the image. Often the bounding box objective may be simultaneously trained with the classification objective, resulting in a loss objective that's the sum of the two loss terms, the L_2 and cross entropy losses, respectively.

2. **Object Detection**. Determine multiple objects in an image and their bounding boxes, with performance measured by *mean average precision (MAP)*. Here, there may be many objects in an image, and there may be several instances of the same object class (e.g., several dogs), which means the network has to predict a varying number of bounding boxes. An example approach to object detection would be Faster R-CNN (discussed later).

[2] We didn't review DenseNets in lecture in part because they were introduced after the Winter 2016 version of the Stanford CS 231n course, but it's an important architecture class to know. In addition, note that the memory requirements are because convolutional layers require lots of memory; efficient implementations involve re-expressing the operator via matrix multiplication, where the matrices end up being large.

[3] `https://pytorch.org/docs/stable/torchvision/models.html`

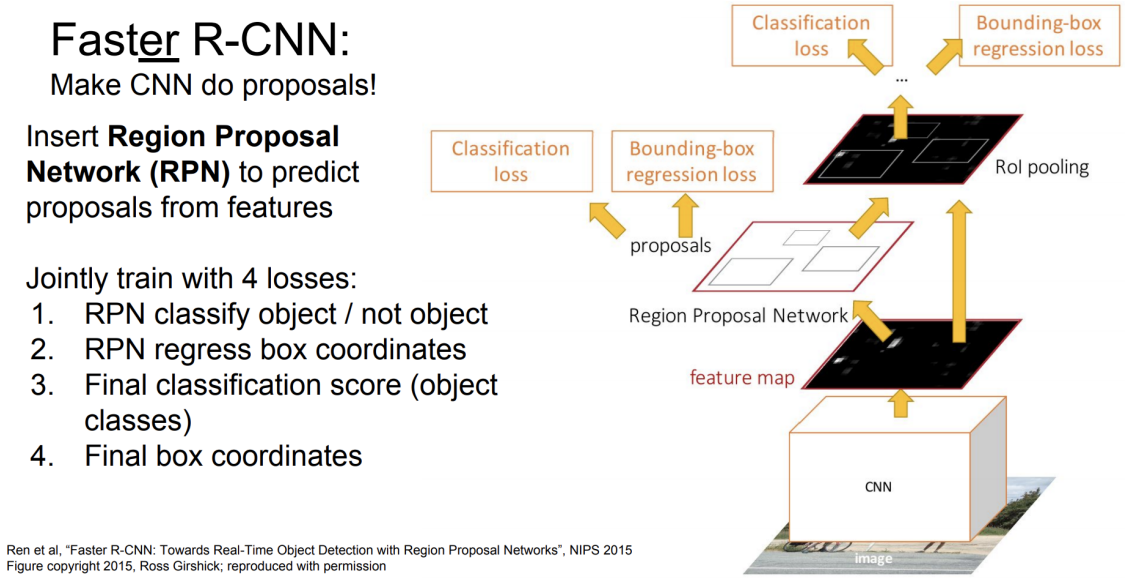

Figure 4.5: The Faster R-CNN architecture. (From CS 231n at Stanford, which used Girshick's image.)

3. **Segmentation**. Label every pixel in the image. Naive solution: run a CNN classifier for each pixel. Better solution exist in the literature. In lecture, we briefly reviewed "UNet," or a fully connected CNN combined with residuals. *Semantic* segmentation means we don't care about distinguishing between different instances of a class, in contrast to the aptly-named *instance* segmentation problem.

4.3.1 Faster R-CNN

Faster R-CNN [6] is a popular technique for object detection problems. The "R" stands for "region," as Faster R-CNN uses these regions as areas in the image that are likely to contain objects. More precisely, a *Region Proposal Network* predicts proposals from CNN features. The CNN features were obtained from passing the original input image through several convolutional layers.

The network is trained jointly using four losses, which normally means adding up the objectives (possibly with different weights).

4.3.2 Transposed Convolution

We briefly comment on an operator called the *transpose convolution*, because it's often used for *upsampling* a convolutional neural network during segmentation tasks. This operator increases the resolution of the intermediate tensors, which we often want if we want the output of our network to be an image (e.g., of the same size as the input images). Note that early convolutional and pooling layers tend to *downsample* or reduce the size of tensors. It is sometimes called a *deconvolution* operator, but this is not the preferred wording because it is an overloaded term with other definitions commonly used.[4]

If you're curious, the transpose convolution can be thought of as flipping the forward and backward

[4]For example, see some discussion here `https://github.com/tensorflow/tensorflow/issues/256`

passes of the convolution step.[5] In addition, the naming comes from how it can be implemented in a similar manner as in Equation 4.1 but with the weight matrix transposed (along with different padding).

[5]For additional details, we recommend checking out the visuals here: `https://github.com/vdumoulin/conv_arithmetic` along with the tech report at `https://arxiv.org/abs/1603.07285` .

A Other Stuff

In this appendix, we briefly review batch normalization and dropout. [*Note to GSIs*: It is unlikely that we will have time to fully discuss these in the section; use this as extra material if time permits.]

A.1 Batch Normalization

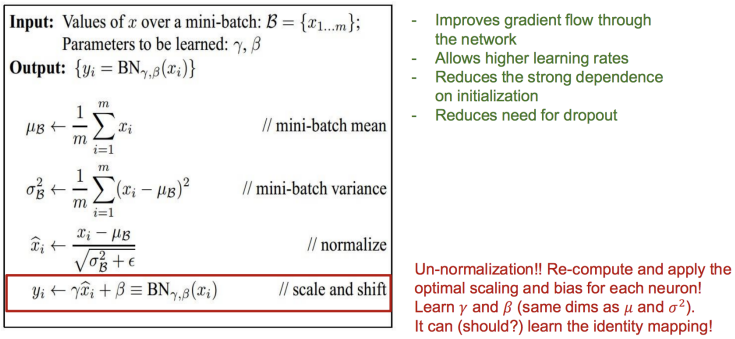

Figure 4.6: Batch normalization. (From Lecture 7)

Batch normalization [3] is a popular technique to help train deep neural networks, based on the intuition that it is better to have unit Gaussian inputs to layers at initialization. (Though, we caution, the reason for why batch normalization works in practice is not entirely understood, but probably not actually due to "internal covariate shift.")

The computational details are shown in Figure 4.6. [*Note to GSIs*: draw out the computational graph of batch normalization on the board, using $\hat{x}_i$, $\mu_\mathcal{B}$, and $\sigma^2_\mathcal{B}$ as intermediate nodes in the network. The input nodes are γ (shape $(D,)$), β (also shape $(D,)$), and $x_{1,\ldots,M}$ representing the minibatch, of shape (M, D). The output node is the minibatch $y_{1,\ldots,M}$.]

The choice of whether to apply batch normalization before or after an activation function (typically ReLU) appears to be a source of disagreement among many practitioners. Currently, it is likely best to try both orderings and using what works best on held-out validation data.

A.2 Dropout and Ensembles

Dropout [8] (see Figure 4.7) is a popular technique for regularizing neural networks by randomly removing nodes during the training process. To implement this, use "inverted dropout" (explained in the CS 231n notes). Benefits:

- Forces network to learn redundant expressions.
- Dropout can be thought of as representing an ensemble of neural networks, because each forward pass effectively is a different neural network since random nodes are removed.

For ensembles, there are two ways one can think of doing these for neural networks [*Note to GSIs*: draw these on the board]:

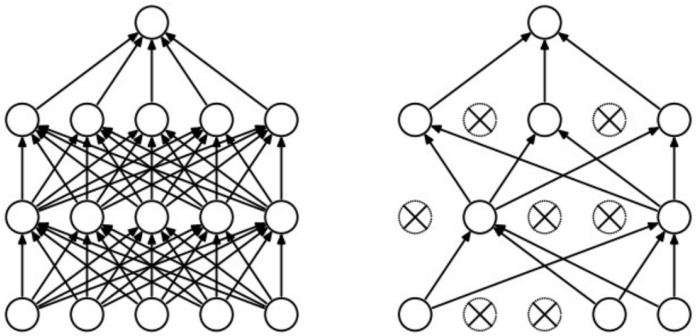

Figure 4.7: Applying dropout.

- **Prediction averaging**. Train N neural networks independently. Average their predictions, or their final outputs. This works.
- **Parameter averaging**. Train N neural networks independently. Average their parameters $\theta = \frac{1}{N}\sum_{j=1}^{N}\theta^{(j)}$. This doesn't work.

Parameter averaging *can* work ... if the parameters are relatively constrained. For example, consider training a neural network while keeping snapshots of the parameters at different times, $\{\theta_1, \theta_2, \ldots, \theta_T\}$. Keeping a moving average of these parameters and using that for predictions can work (and has the side benefit of being memory-efficient).

Residual Networks were the winner of ILSVRC 2015. In 2016 (the second to last year), researchers used ensembles to get the highest performance.

References

[1] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[2] Gao Huang et al. “Densely Connected Convolutional Networks”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[3] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *International Conference on Machine Learning (ICML)*. 2015.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems (NIPS)*. 2012.

[5] Yann Lecun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[6] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *Neural Information Processing Systems (NIPS)*. 2015.

[7] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *International Conference on Learning Representations (ICLR)*. 2015.

[8] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research (JMLR)* (2014).

[9] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.