**Introduction to Convolutional Neural Networks (CNNs) in PyTorch**

**Representing images digitally:**

* While convolutional neural networks (CNNs) see a wide variety of uses, they were originally designed for images, and CNNs are still most commonly used for vision-related tasks. For today, we’ll primarily be focusing on CNNs for images. Before we dive into convolutions and neural networks, it’s worth prefacing with how images are represented by a computer, as this understanding will inform some of our design choices.
* Previously, we saw an example of a digitized MNIST handwritten digit. Specifically, we represent it as an ***H \* W*** table, with the value of each element storing the intensity of the corresponding pixel
* With a 2D representation as above, we for the most part can only efficiently represent grayscale images. What if we want color? There are many schemes for storing color, but one of the most common ones is the RGB color model. In such a system, we store 3 tables of pixel intensities (each called a channel), one each for the colors red, green, and blue (RGB), resulting in an ***H \* W \* 3*** tensor. Pixel values for a particular channel indicate how much of the corresponding color the image has at a particular location.
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer program

  Description automatically generated
* While we have so far considered only 3 channel RGB images, there are many settings in which we may consider a different number of channels. For example “hyperspectral imaging” uses a wide range of the electromagnetic spectrum to characterize a scene. Such modalities may have hundreds of channels or more. Additionally, we’ll soon see that certain intermediate representations in a CNN can be considered images with many channels.

**Convolutions**

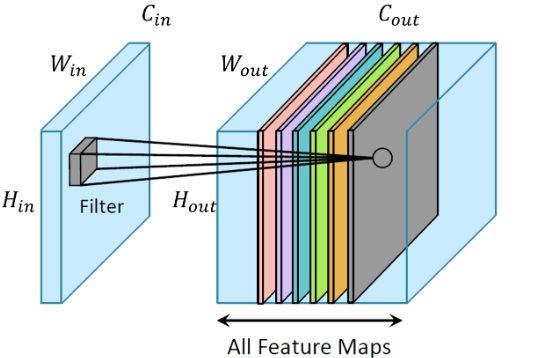
* Convolutional neural networks (CNNs) are a class of neural networks that have convolutional layers. CNNs are particularly effective for data that have spatial structures and correlations (e.g. images). We’ll focus on CNNs applied to images in this tutorial. Recall that a multilayer perceptron (MLP) is entirely composed of fully connected layers, which are each a matrix multiply operation (and addition of a bias) followed by a non-linearity (e.g. sigmoid, ReLU). A convolutional layer is similar, except the matrix multiply operation is replaced with a convolution operation (in practice a cross-correlation). Note that a CNN need not be entirely composed of convolutional layers; in fact, many popular CNN architectures end in fully connected layers.
* A close-up of a white background

  Description automatically generated
* A screenshot of a math problem

  Description automatically generated
* A screenshot of a computer program

  Description automatically generated

**Convolutional layer:**

* In a convolutional layer, we convolve the input x with a convolutional kernel (aka filter), which we also call ***W***, producing output “y”.
* In the context of CNNs, the output y is often referred to as feature maps. As with a fully connected layer, the goal is to learn “W” and “b” for our model.
* Unlike the input of a fully connected layer, which is 𝑥∈ℝ^(𝑀×𝐶𝑖𝑛) the dimensionality of an image input is , where “M” is still the batch size “C” is the number of channels of the input (e.g. 3 for RGB), and “H” and “W” are the height and width of the image.
* The weight parameter “W” is also different in a convolutional layer. Unlike the 2-D weight matrix for fully connected layers, the kernel is 4-D with dimensions , where “H” and “W” are the kernel height and weight, respectively. A common choice for “H” and “W” is “H = W = 3 or 5”, but this tends to vary depending on the architecture. Convolving the input with the kernel and adding a bias then gives an output . If we use “same” padding and a stride of 1 in our convolution, our output will have the same spatial dimensions as the input: “Hout = Hin” and “Wout = Win”
* If you’re having trouble visualizing this operation in 4D, it’s easier to think about for a single member of the minibatch, one convolutional kernel at a time. Consider a stack of “Cout” number of kernels, each of which are 3D ( Cin x H x W ). This 3D volume is then slid across the input ( which is also 3D: Cin x Hin x Win ) in two spatial dimensions (along Hin and Win). The outputs of the multiplication of the kernel and input at every location creates a single feature map that is Hout x Wout. Stacking the feature maps generated by each kernel gives the 3D output: ( Cout x Hout x Wout ). Repeat the process for all “M” inputs in the minibatch and we get a 4D output ( M x Cout x Hout x Wout )
* ****
* A few more things to note:
  + Notice the ordering of the dimensions of the input (batch, channels in, height, width). This is commonly referred to as ***“ N C H W “*** ordering. Many other languages and libraries (e.g. MATLAB, TensorFlow) instead default to the slightly different ***“ N H W C “*** ordering. PyTorch defaults to ***“ N C H W “*** , as it is more efficient computationally, especially with CUDA.
  + An additional argument for the convolution is the “stride”, which controls the how far we slide the convolutional filter as we move it along the input image. The convolutional operator, from its signal processing roots, by default considers a stride length of 1 in all dimensions, but in some situations we would like to consider strides more than 1 (or even less than 1)
  + In the context of signal processing, convolutions usually result in outputs that are larger than the input size, which results from when the kernel “hangs off the edge” of the input on both sides. This might not always be desirable. We can control this by controlling the padding of the input. Typically, we use pad the input to ensure the output has the same spatial dimensions as the input (assuming stride of 1); this makes it easier for us to keep track of what the size of our model is.
  + Below is the convolution operator in code. There is a convolution implementation in “torch.nn.functional”.
* **A screenshot of a computer code

  Description automatically generated**
* Just like in a MLP, we can stack multiple of these convolutional layers. In the Representing Images Digitally section, we briefly mentioned considering images with channels more than 3. Observe that the input to the second layer (i.e. the output of the first layer) can be viewed as an “image” with Cout channels. Instead of each channel representing a color content though, each channel effectively represents how much the original input image activated a particular convolutional kernel. Given Cout kernels that are each ( Cin x H x W ), this results in Cout channels for the output of the convolution.
* Note that we need to change the dimensions of the convolutional kernel such that its input channels matches the number of output channels of the previous layer
* **A screenshot of a computer program

  Description automatically generated**
* In fact, we typically perform these convolution operations many times. Popular CNN architectures for image analysis today can be 100+ layers

**Reshaping:**

* You’ll commonly find yourself needing to reshape tensors while building CNNs. The PyTorch function for doing so is “view()”. Anyone familiar with NumPy will find it very similar to np.reshape(). Importantly, the new dimensions must be chosen so that it is possible to rearrange the input into the shape of the output (i.e. the total number of elements must be the same). As with NumPy, you can optimally replace one of the dimensions with a -1, which tells “torch” to infer the missing dimension.
* **A screenshot of a computer

  Description automatically generated**
* CNN architectures also commonly contain fully connected layers or a softmax, as we’re often interested in classification. Both of these 2D inputs with dimensions [batch, dim], so you have to “flatten” a CNN’s 4D output to 2D. For example, to flatten the convolutional feature maps we created earlier:
* **A screenshot of a computer code

  Description automatically generated**

**Pooling and Striding**

* Almost all CNN architectures incorporate either pooling or striding. This is done for a number of reasons, including:
  + Dimensionality reduction: pooling and striding operations reduces computational complexity by shrinking the number of values passed to the next layer. For example, a 2x2 maxpool reduces the size of the feature maps by a factor of 4.
  + Translational invariance: Oftentimes in computer vision, we’d prefer that shifting the input by a few pixels doesn’t change the output. Pooling and striding reduce sensitivity to exact pixel locations.
  + Increasing receptive field: by summarizing a window with a single value, subsequent convolutional kernels are seeing a wider swath of the original input image. For example, a max pool on some input followed by a 3x3 convolution results in a kernel “seeing” a 6x6 region instead of 3x3.
* **Pooling:**
  + The two most common forms of pooling are max pooling and average pooling. Both reduce values within a window to a single value, on a per-feature-map basis. Max pooling takes the maximum value of the window as the output value; average pooling takes the mean.
  + **A diagram of a number of squares

    Description automatically generated with medium confidence**
* **A screenshot of a computer program

  Description automatically generated**
* **Striding:**
  + One might expect that pixels in an image have high correlation with neighboring pixels, so we can save computation by skipping positions while sliding the convolutional kernel. By default, a CNN slides across the input one pixel at a time, which we call a stride of 1. By instead striding by 2, we skip calculating 75% of the values of the output feature map, which yields a feature map that’s half the size in each spatial direction. Note, while pooling is an operation done after the convolution, striding is part of the convolution operation itself.
  + **A screenshot of a computer code

    Description automatically generated**

**Building a custom CNN**

* Let’s revisit MNIST digit classification, but this time, we’ll use the following CNN as our classifier: 5 x 5 convolution -> 2 x 2 max pool -> 5 x 5 convolution -> 2 x 2 max pool -> fully connected to Real Num^256 -> fully connected to Real Num^10 (prediction). ReLU activation functions will be used to impose non-linearities. Remember, convolutions produce 4-D outputs, and fully connected layers expect 2-D inputs, so tensors must be reshaped when transitioning from one to the other
* We can build this CNN with the components introduced before, but as with the logistic regression example, it may prove helpful to instead organize our model with a “nn.module”
* **A screenshot of a computer program

  Description automatically generated**
* Notice how our “nn.Module” contains several operation chained together. The code for submodule initialization, which creates all the stateful parameters associated with each operation, is placed in the \_\_init\_\_() function, where it is run once during object instantiation. Meanwhile, the code describing the forward pass, which is used every time the model is run, is placed in the “forward()” method. Printing an instantiated model shows the model summary
* **A screenshot of a computer code

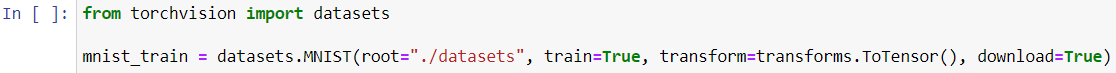
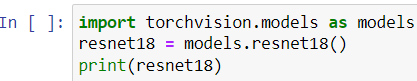
  Description automatically generated**
* We can drop this model into our logistic regression training code, with few modifications beyond changing the model itself. A few other changes:
  + CNNs expect a 4-D input, so we no longer have to reshape the images before feeding them to our neural network.
  + Since CNNs are a little more complex than models we’ve worked with before, we’re going to increase the number of epochs (complete passes through the training data) during training.
  + We switch from a vanilla stochastic gradient descent optimizer to the Adam optimizer, which tends to do well for neural networks

**A screenshot of a computer

Description automatically generated**

* **Keep in mind that CPU computation takes a minute, while GPU is seconds**
* Test accuracy: 0.9905999898910522

**Torch vision**

* **Datasets and transforms**
  + As any experienced ML practitioner will say, data wrangling is often half (sometimes even 90%) of the battle when building a model. Often, we have to write significant code to handle downloading, organizing, formatting, shuffling, pre-processing, augmenting, and batching examples. For popular datasets, we’d like to standardize data handling so that the comparisons we make are specific to the models themselves.
  + Enter “TorchVision”: TorchVision includes easy-to-use APIs for downloading and loading many popular vision datasets. We’ve previously seen this in action for downloading the MNIST dataset
* ****
  + Of course, there’s many more. Currently, datasets for image classification (e.g. MNIST, CIFAR, ImageNet), object detection (VOC, COCO, Cityscapes), and video action recognition (UCF101, Kinetics) are included.
  + For formatting, pre-processing, and augmenting, transforms can come in handy. Again, we’ve seen this before, when we used a transform to convert the MNIST data from PIL images to PyTorch tensors. However, transforms can be used for much more. Preprocessing steps like data whitening are common before feeding the data into the model. Also, in many cases, we use data augmentations to artificially inflate our dataset and learn invariances. Transforms are a versatile tool for all of these.
* **Leveraging popular convolutional neural networks**
  + While you certainly can build your own custom CNNs like we did above, more often than not, it’s better to use one of the popular existing architectures. The Torch vision documentation has a list of supported CNNS, as well as some performance characteristics. There’s a number of reasons for using one of these CNNs instead of designing your own
  + First, for image datasets larger and more complex than MNIST (which is basically all of them), a fair amount network depth and width is often necessary. For example, some of the popular CNNS can be over 100 layers deep, with several tricks and details beyond what we’ve covered in this notebook. Coding all of this yourself has a high potential for error, especially when you’re first getting started. Instead, you can create the CNN architecture using Torchvision, using a couple lines:
* ****
  + Loading a working CNN architecture in a couple lines can save a significant amount of time both implementing and debugging
  + The second, perhaps even more important, reason to use one of these existing architectures is the ability to use pre-trained weights. Early on in the recent resurgence of deep learning, people discovered that the weights of a CNN trained for ImageNet classification were highly transferable. For example, it is common to use the weights of an ImageNet-trained CNN as a weight initialization for other vision task, or even to freeze the bulk of the weights and only re-train the final classification layer(s) on a new task. This is significant, as in most settings, we rarely have enough labeled data to train a powerful CNN from scratch without overfitting. Loading pre-trained CNN is also pretty simple, involving an additional argument to the previous cell block:
  + 
* **Other computer vision tasks**
  + The base CNN architectures were often designed for image classification, but the same CNNS are often used as the backbone of most modern computer vision models. These other models often take this base CNN and include additional networks or make other architecture changes to adapt them to other tasks, such as object detection. Torchvision contains a few models (and pre-trained weights) for object detection, segmentation, and video action recognition. For example, to load a “Faster R-CNN” with a “ResNet50” convolutional feature extractor with “Feature Pyramid Networks” pre-trained on “MS COCO”:
* 
  + Torchvision’s selection of non-classification models is relatively light, and not particularly flexible. A number of other libraries are available, depending on the task. For example, for object detection and segmentation, Facebook AI Research’s “Detectron2” is highly recommended.