Creating and Training New Networks

Transfer Learning Limitations

Transfer learning allows you to leverage the work of deep learning researchers who have access to large amounts of data and computational resources. This gives you a significant head start on training because the network parameters have already been tuned to extract features for a wide variety of images.

However, transfer learning works only if you don't need to modify the early layers of the network. Because information flows forward from layer to layer, if you change the input layer of your network, the downstream architecture and parameters are no longer valid. In this situation, you can't use transfer learning.

Different Types of Images

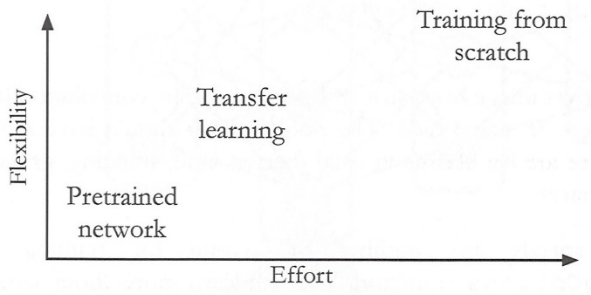
Most pretrained networks expect inputs to be color images of a specific size. If you have grayscale images, or color images of a slightly different size, you can always resize your images to match the expected input size.

But what if your images are a significantly different size? Or your images have more than three channels, like infrared, sonar, or seismic readings? You won't be able to force your input into a size these pretrained networks expect without losing potentially important information.

Training from Scratch

If you have enough training data, the best solution might be to train a network from scratch. The primary difference from transfer learning is that you need to create the array of layers, rather than importing a network as a starting point.

How do you know what layers to use, and in what order? Rather than trying to design your own deep network, you can take advantage of common architectures developed by experts over many iterations of trial and error. This approach starts with an existing architecture, but trains the weights and biases from scratch.



Creating a Network from Layers

Your new network is going to be much shallower than AlexNet, but the structure applies the same ideas. The layers in this architecture are common to most pretrained networks.

You can create a column vector of seven layers with the functions listed below.

```
layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(filterSize, numFilters)
    reluLayer()
    maxPooling2dLayer(poolSize)
    fullyConnectedLayer(numClasses)
    softmaxLayer()
    classificationLayer()
    ]
```

Each land cover image has a size 28-by-28-by-4. The convolution layer should have 20 filters of size 3-by-3. The pooling layer should have a pool size of 3-by-3. There are six classes in total (barren land, building, grassland, road, trees, and water).

You can specify any number of setting for training using the `trainingOptions` function. You will learn more setting training options in a later chapter.

Once you have created a network architecture, you can confirm the network is valid using `analyzeNetwork`. Use your created layers as input to the function to visualize the architecture and detect errors.

```
analyzeNetwork(layers)
```

Neurons in a Network Layer

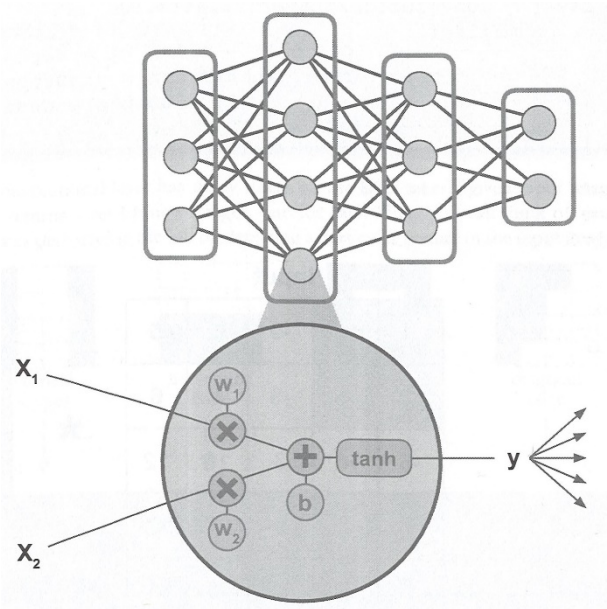
Many layers in deep networks are specific to deep learning applications, but these layers are derived from classical neural networks.

A classical neural network uses fully connected layers. Every neuron in a layer is connected to every neuron in the next layer. The information passed along each connection is a single number.

Inside a neuron, the input values are multiplied by parameters known as weights. The results are added, along with another parameter known as the bias. The sum is passed to a function called the transfer function to get the output value.

If you change the parameters – the weights and biases – then the output from the neuron will be different. These parameters govern the behavior of the neurons and, collectively, the whole network. The training process involves adjusting these parameters so that the network produces the desired output for known samples.

The Weights property of a layer contains the weights of the layers. Weights are learned during the training process. Pretrained network have already learned weights and biases for image classification.



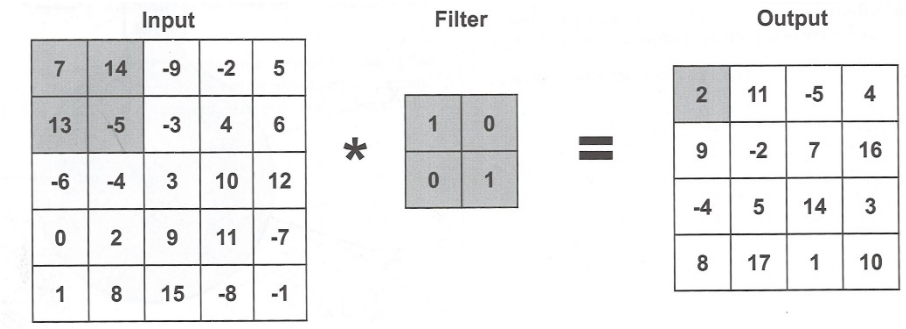
Convolutional Layer

In contrast to fully connected layers, a convolution layer applies sliding filters to the input image. More details on the convolution operation are on the next page.

A convolution will usually apply several filters to the input image. The number of filters is specified with the numFilters input.

Each filter must have the same size, specified by filterSize. Ideally, each filter will learn to detect different features.

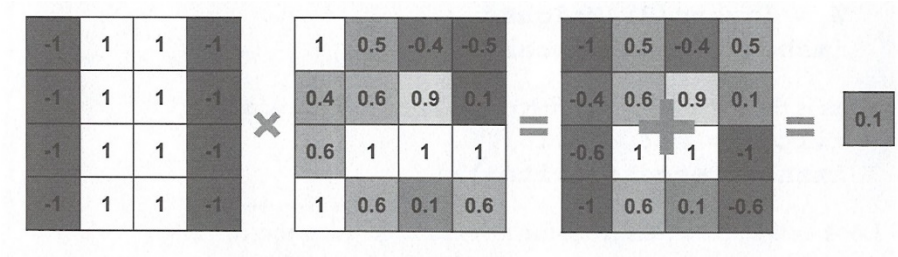
```
layer = convolution2dLayer(filterSize, numFilters)
```



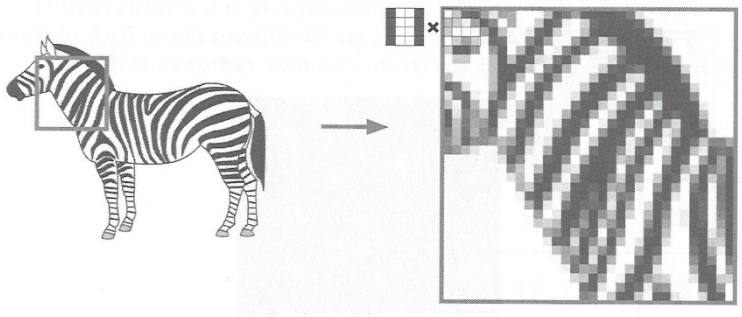
Performing Convolutions

Convolution highlights parts of images that match a filter. The filters are represented in a convolution layer's weights.

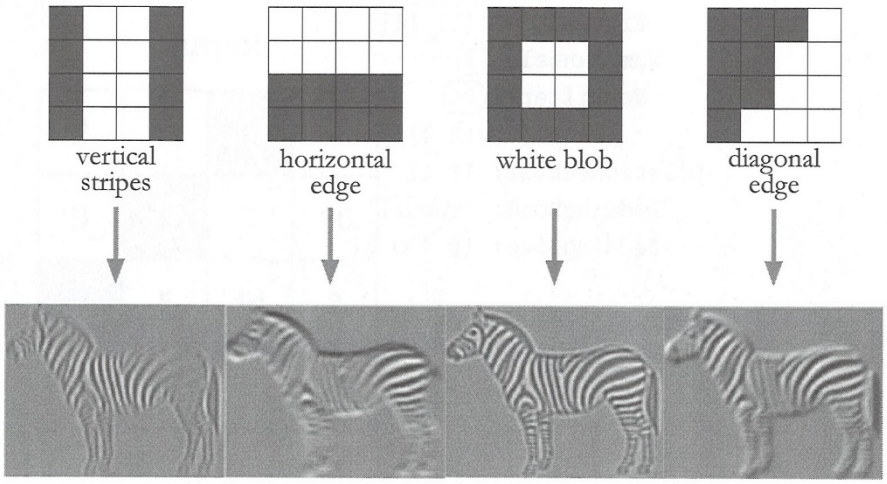
A small piece of the image is multiplied with the filter, resulting in a 2-D matrix. All the values of this 2-D matrix are added to get a scalar value. This value can be interpreted as how closely that small piece of the image matches the pattern of the filter.



This filter is moved over the image until it is scanned completely. The mathematical operation is called a *convolution*. A value is calculated for each position, and the result is a new image – the activation – that highlights the presence of a pattern in the input image.



A convolutional layer has many filters, so the layer takes a given input image and returns a set of new images, one for each filter. You can think of each filter as performing the job of detecting a particular feature in the input image.



Viewing Learned Filters

Consider the first convolution layer in AlexNet and its properties shown below.

Hyperparameters are parameters that are set when a layer is created. Recall that the first two inputs to convolution2dLayer is the filter size and the number of filters. ‘NumChannels’ is the number of channels of the input to this convolution layer. In ‘conv1’, the input the image sized 227-by-227-by-3, so the number of channels is three.

The *learnable* parameters are updated during training, but the size of these arrays is calculated from the filter size, number of input channels, and the number of filters.

convolution2dLayer with properties:

Name: ‘conv1’

Hyperparameters

FilterSize: [11 11]

NumChannels: 3

NumFilters: 96

Stride: [4 4]

DilationFactor: [1 1]

PaddingMode: ‘Manual’

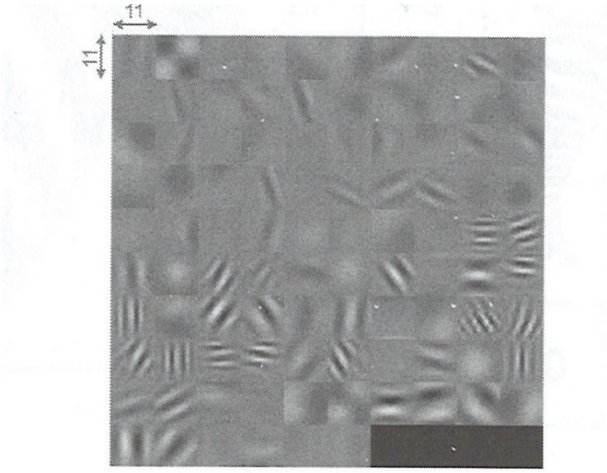
PaddingSize: [0 0 0 0]

Learnable Parameters

Weights: [11x11x3x96, single]

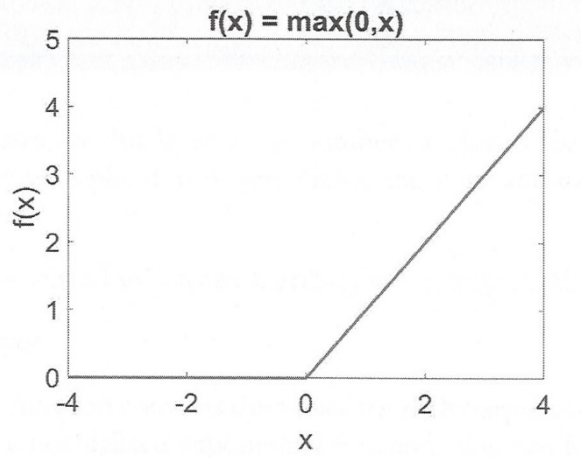
Bias: [1x1x96, single]

In AlexNet, the weights of the first convolution layer is a matrix sized 11-by-11-by-3-by-96. This means that there are 96 different filters. Each of those filters is size 11-by-11-by-3, so you can view them each as an RGB image.



Rectified Linear Unit Layer

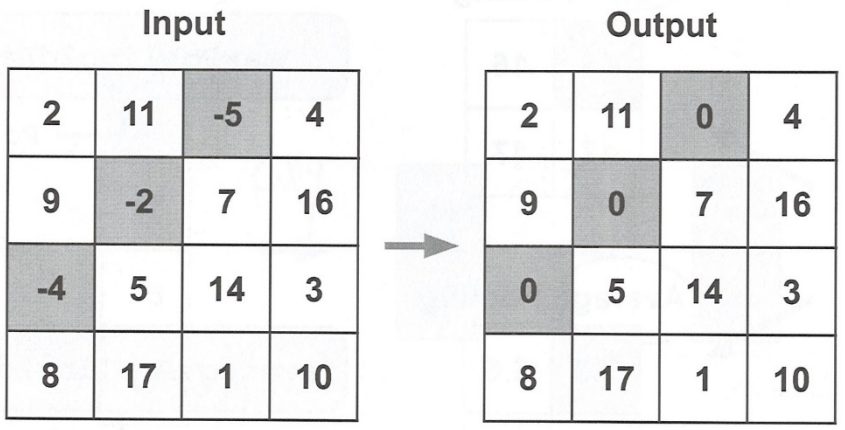
Convolutional layers are usually followed by a nonlinear activation layer such as a rectified linear unit(ReLU). A ReLU layer performs a threshold operation on each element of the input. Any value less than zero is set to zero.



The ReLU layer does not change the size of its input.

ReLU layers are commonly used in deep networks because they are fast to compute.

```
layer = relulayer ( )
```



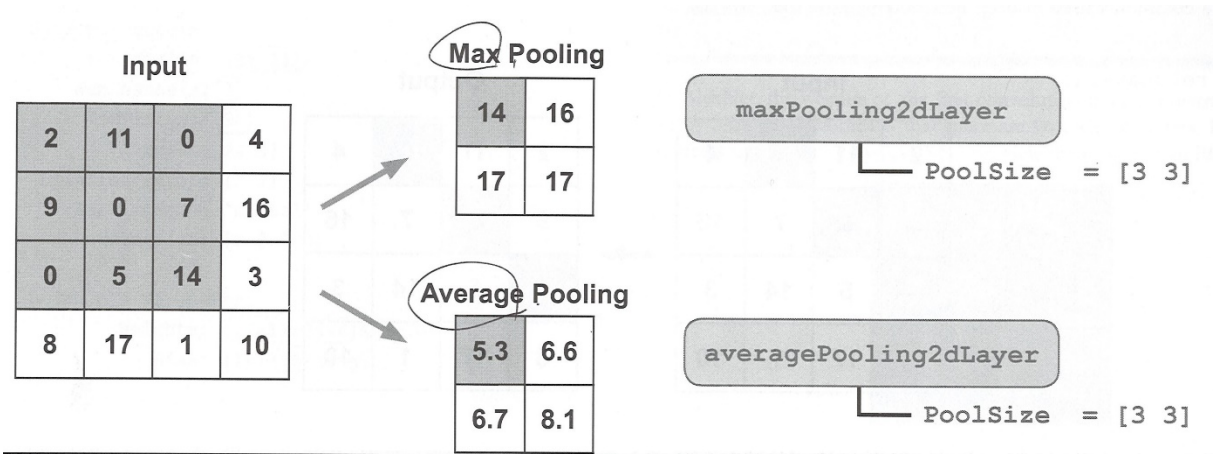
Pooling Layer

A pooling layer performs downsampling by dividing the input into rectangular pooling regions and computing the maximum (max pooling) or average (average pooling) of each region.

Pooling reduces the network complexity and create a more general network.

layer = maxPooling2dLayer (poolSize)

layer = averagePooling2dLayer (poolSize)



Output Layers

Fully Connected Layer

Features passing through the network are stored in a collection of matrices until they reach the fully connected layer. At this layer, the input is “flattened” so that it can be mapped to the output classes. This layer is a classical neural network.

The output size for this layer is the number of lasses for your classification problem. For example, if you were classifying dogs and cats, the output size would be two.

```
layer = fullyConnectedLayer (outputSize)
```

Softmax Layer

The softmax function converts the values for each output class into normalized scores using a normalized exponential function. You can interpret each value as the probability that the input image belongs to each class.

```
layer = softmaxLayer ()
```

Output Layer

The classification output layer returns the name of the most likely class.

```
layer = classificationLayer ( )
```

Others Layers

More information on the layers can be found in the documentation:

doc(‘Deep Learning Training From Scratch’) **> Layers**