Geospatial Deep Learning

CNNs

Image from NASA: https://commons.wikimedia.org/wiki/File:Earth_Western_Hemisphere_transparent_background.png#filelinks

Now that you have an understanding of fully connected ANNs and DL more broadly, we will move on to investigate a specific type of DL that has shown great promise in the object-detection, computer vision, and image labelling fields.

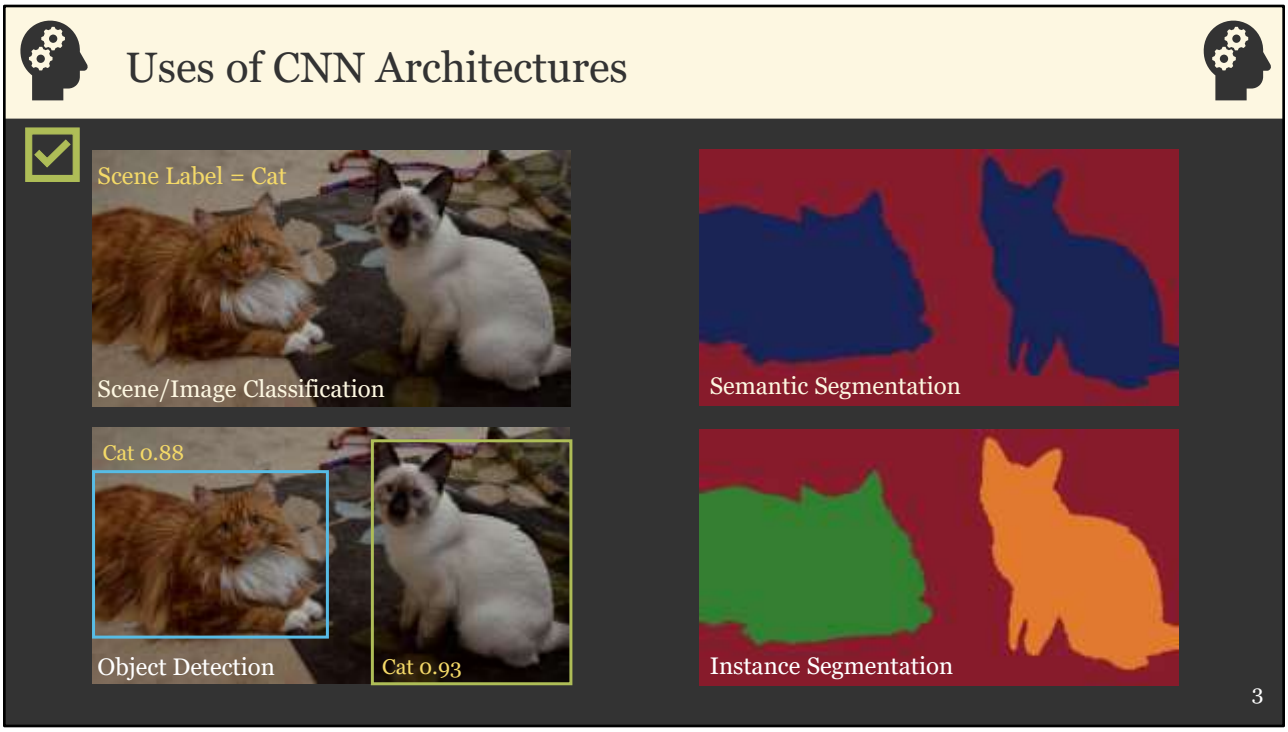
Convolutional neural networks (CNNs) allow for the analysis of image data by learning spatial patterns using local moving windows. In this section, we will specifically focus on problems that require an image to be labeled as a representation of a category. Such tasks are termed scene labeling or scene classification. We will discuss other applications of CNNs, pixel-level classification, object detection, and instance segmentation, in later modules.

Module Topics

1. Scene classification/labeling
2. Convolution operations
3. CNN building blocks: convolutional layers, pooling layers, activation functions, and fully connected layers
4. kernel size, stride, and padding
5. Max pooling
6. Feature maps
7. Batch normalization
8. ReLU activation function
9. CNN architectures
10. Combating overfitting
11. Preparing data
12. VGGNet
13. InceptionNet
14. ResNet
15. DenseNet
16. MobileNet

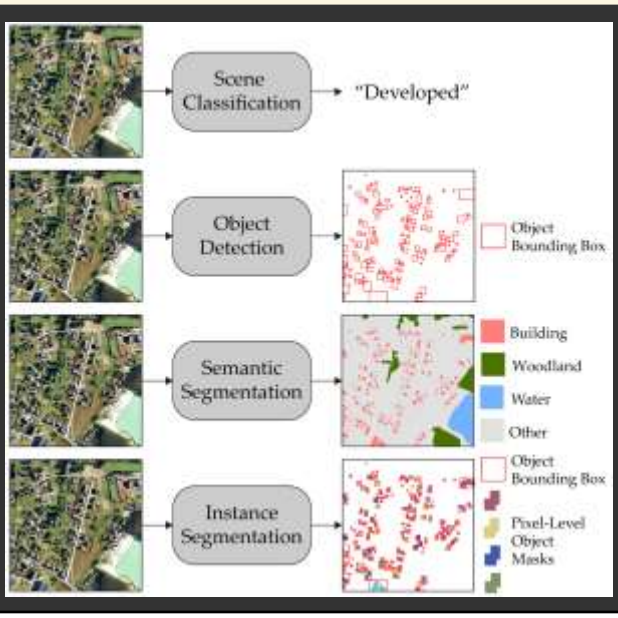
2

These are the topics covered in this module.



Scene labeling consists of predicting a label for an entire image. This is not a localized or pixel-level prediction; instead, one label is predicted for the entire image. For example, the image shown on this slide could be labeled to the “cat” class. The location of the cat or cats in the image are not predicted, and individual pixels are not assigned to categories. We will explore the other use cases conceptualized on this slide, semantic segmentation, object detection, and instance segmentation, in the following modules.

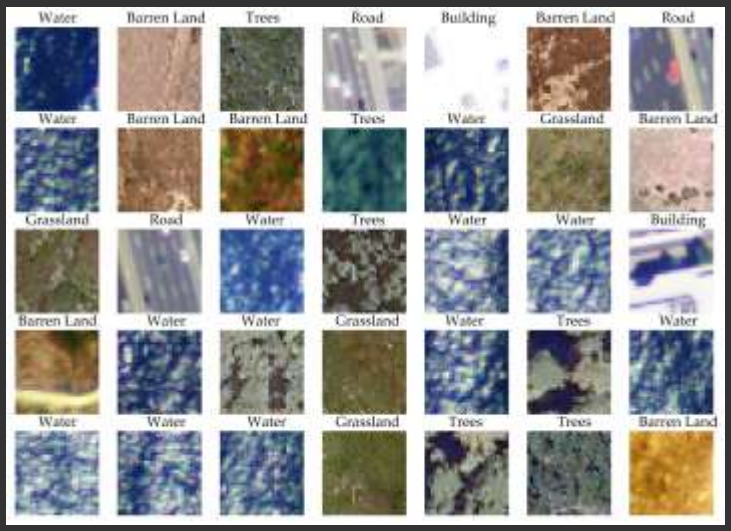
Uses of CNN Architectures



4

This figure on this slide further conceptualizes the difference between scene classification/labeling, object detection, semantic segmentation, and instance segmentation.

Scene Classification



https://csc.lsu.edu/~saikat/deepsat/

- Label image as representation of a category or multiple categories

5

This is an example of a scene classification problem where small aerial orthophotograph extents are labeled to one of a set of predefined land cover classes. These samples are from the SAT-6 dataset. These data are available at the link included on this slide.

Scene Classification

		Reference						
		Barren	Building	Grassland	Road	Tree	Water	**UA**
Classification	Barren	18,205	0	47	0	0	0	0.997
	Building	0	3,678	0	4	0	0	0.999
	Grassland	162	0	12,542	0	15	0	0.986
	Road	0	36	2	2066	0	0	0.982
	Tree	0	0	5	0	14,170	0	1.000
	Water	0	0	0	0	0	30,068	1.000
	PA	0.991	0.990	0.996	0.998	0.999	1.000	

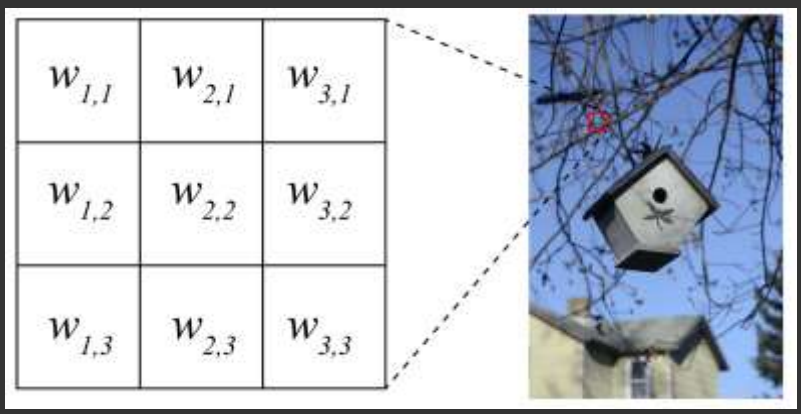
6

This confusion matrix represents an assessment of a scene classification problem, where each sample is an entire image as opposed to pixels or other sampling units. In contrast, each assessment unit for semantic segmentation would be individual pixels as opposed to entire scenes or images.

Introduction to Convolution

What is Convolution?

- CNNs or ConvNets
- Learn spatial patterns and context
- Based on moving windows
- Apply moving windows to create feature maps



8

Convolutional neural networks, ConvNets, or CNNs are a sub-type of DL ANNs that allow for the incorporation of spatial patterns into the learning process.

Instead of learning weights/parameters associated with links between neurons, as is the case for fully connected layers, convolutional layers learn weights to build a moving window or kernel that is passed over the image to manipulate the data and learn spatial patterns and abstractions.

Once the filter or moving window is learned, it is used to alter the values in the array and produce a feature map.

The basic idea behind CNNs is to learn spatial patterns at different scales.

This type of DL has led to many of the recent (i.e., over the last two decades) improvements in computer vision tasks.

Convolution in Geospatial Science

- Moving windows, kernels, neighborhood analysis
- Examples in GIS = Focal Statistics, Majority Filter, Pixel Aggregation
- Examples in Remote Sensing = Texture, Edge Detection, Sharpening, Blurring, Statistical Filters, Focal Analysis

-1	-2	-3	-2	-1
-2	-3	-4	-3	-2
0	0	0	0	0
2	3	4	3	2
1	2	3	2	1

9

As a geospatial scientist or professional, you are likely already familiar with the concept and use of moving windows, as they are commonly applied to raster data, including images, to manipulate the data in some way. Such techniques are termed moving window analysis, kernel-based analysis, neighborhood analysis, or convolution.

For example, raster-based focal statistics techniques implemented in GIS allow for the calculation of a statistical measure within local windows while majority filtering is used to generalize data by returning the most commonly occurring value in a local neighborhood. Pixel aggregation is used to decrease spatial resolution by merging cells.

In remote sensing, moving windows are used to calculate textural measures, such as the measures after Haralick, detect edges, sharpen or blur images, and calculate local statistics.

Moving windows are a key component of working with raster data and images in general. For example, many digital photo effects make use of moving windows.

The example on the slide represents a Sobel filter that is used to highlight edges in a particular direction.

Convolution in Geospatial Science

-1	-2	-3	-2	-1
-2	-3	-4	-3	-2
0	0	0	0	0
2	3	4	3	2
1	2	3	2	1

10

Here are some examples of window-based tool in remote sensing.

Note here that users can define their own filters by changing the values in the filter array or the shape or size of the array.

The key difference between these uses of filters and ConvNets is that the CNN learns these values or weights as part of the training process as opposed to the user defining them manually.

Convolution Types

1D Convolution

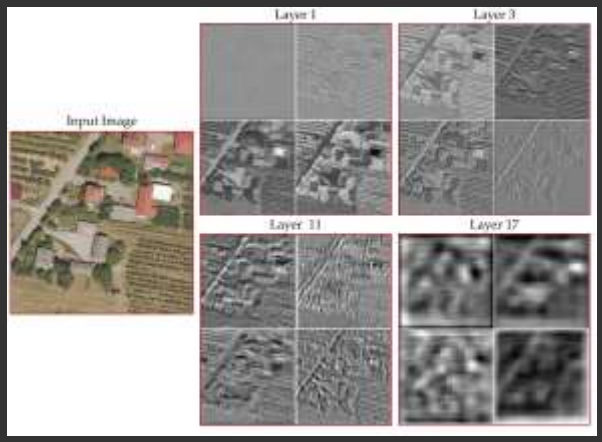
- Time, spectral reflectance, audio signals, elevation

2D Convolution

- Images

3D Convolution

- Point clouds, voxels



11

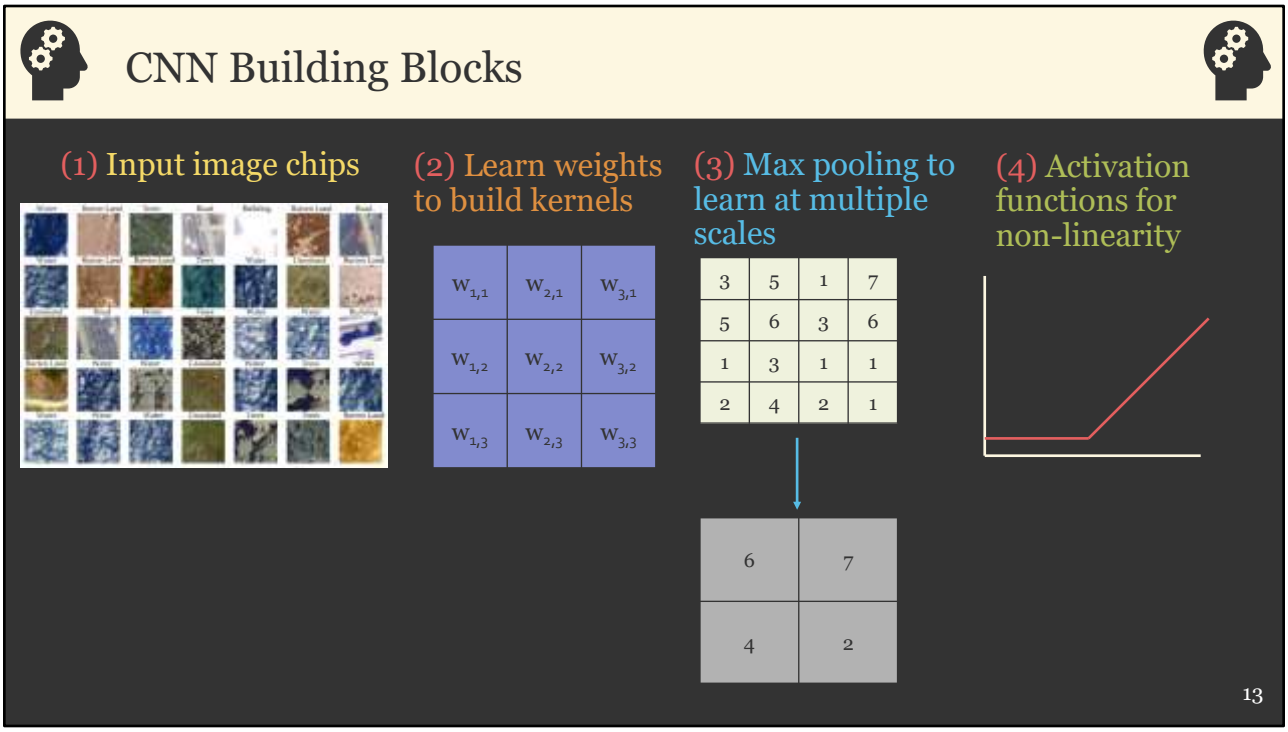
There are different types of convolution operations that can be performed on tensor data.

1D convolution can be used to learn patterns in a single dimension, such as patterns over time, in spectral reflectance curves, in audio signals, or in elevation data. In GIS and remote sensing 1D convolution is sometimes used for time series analysis and spectral analysis of hyperspectral data.

3D convolution allows for convolution to be applied in 3 dimensions. These types of operations are useful for analyzing 3D point clouds, such as LiDAR, and voxels, which are raster grids extended into 3 dimensions as cubes.

In this course, we will focus on 2D convolution, which is applied in two dimensions. This type of convolution can be used to explore patterns in raster-based data, such as images, occurring in the spatial dimensions: width and height.

CNN Building Blocks



CNNs are generally built using a fairly simple set of operations or components including 2D convolutional, max pooling, and activation layers. In the following slides, we will explore these different components of CNN architectures.

2D Convolutional Layers

- Apply convolution to input layer/learn kernel in window
- Kernel Size
  - 3 x 3
  - 5 x 5
  - 7 x 7
- Filters
  - Number of filters to learn to generate feature maps
- Stride
- Padding
- Input Shape = (samples, in channels/feature maps, rows, columns)
- Output Shape = (samples, new feature maps, new rows, new columns)

14

One of the key layer types in 2D convolution is the 2D convolutional layer. The learned filters are commonly a 3x3 or 5x5 array of weight values that are trainable. Other sizes are possible; however, these sizes are most common. Learnable parameters include the kernel weights and a bias term for each kernel.

When setting up a 2D convolution layer, you will need to define the kernel size, the number of filters learned, the stride, and whether to apply padding.

The kernel size specifies the size of the window while the number of filters represents the number of filters that are learned. For example, an input image with three bands could be processed using 64 learned filters to generate 64 feature maps or data abstractions.

The stride is how much the filter moves as it passes over the image. If the stride is set to 1, then no pixels will be skipped and the output array size in the spatial dimensions will be maintained. Using a stride greater than 1 results in decreasing the size of the array. However, it is more common to apply a pooling operation as opposed to increasing the stride as a means to reduce the size of the array in the spatial dimensions.

Cells at the margin of the image do not have a full set of neighbors to fill the

extent of the kernel. So, when only cells that have a full set of neighbors are processed, the array size will be decreased in the spatial dimensions. To maintain the array size, padding can be added to the margins of the image/input data or feature maps being processed.

Trainable Parameter Counts

Weights = # of Input feature maps × # of Output Feature Maps × Kernel Height × Kernel Width

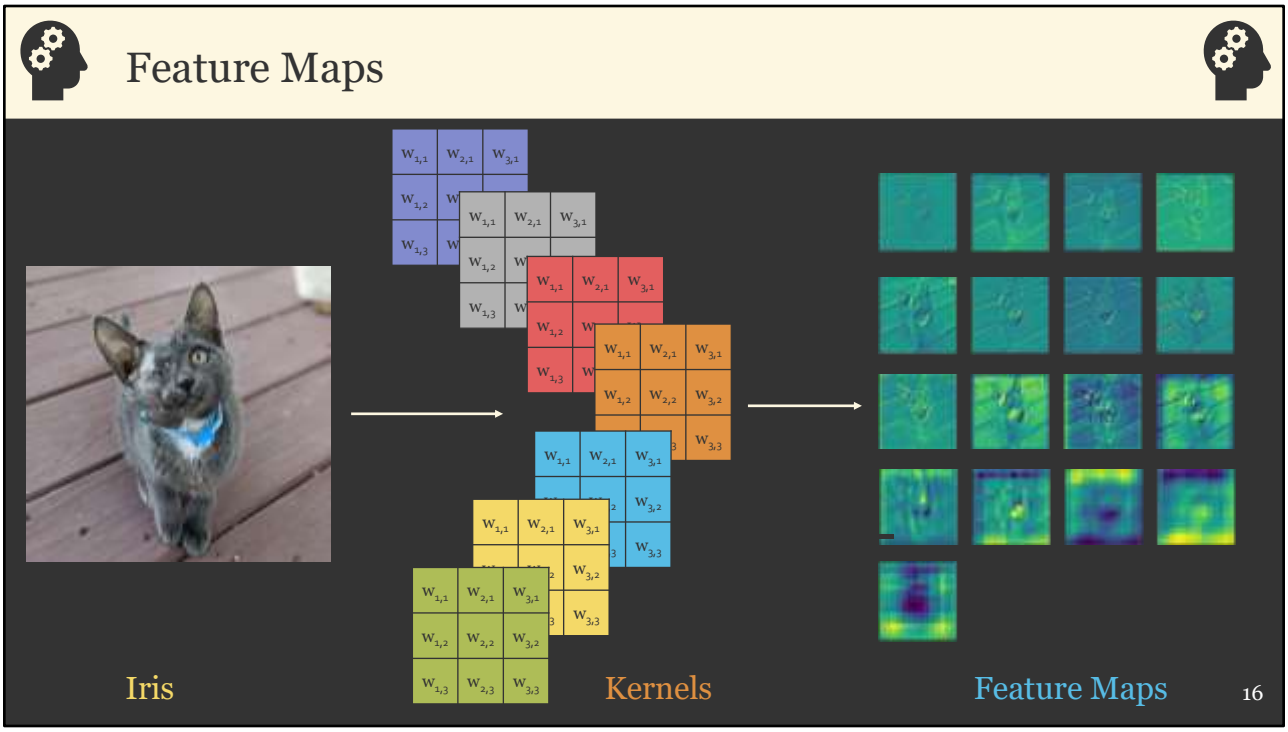
Bias = # Output Feature Maps

15

The number of trainable weights in a kernel is equal to the number of input feature maps or channels multiplied by the number of output feature maps or channels then multiplied by the kernel size (for example, 3x3 or 9).

The number of bias terms is equal to the number of output feature maps. In other words, there is a trainable bias for each output feature map. Note that the convolution operation can be configured so that the bias term is not trainable or applied.

It is important to note here that each input feature map contributes to each output feature map and that different weights can be applied to each input when producing a feature map. This allows for the convolution operations to be very expressive and powerful in terms of their ability to abstract input data.

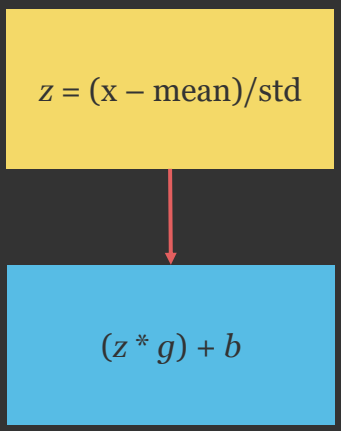


This slide visualizes some feature maps derived by multiplying the input image or prior feature maps produced by prior layers in the model by the trainable kernel weights and adding a trainable bias term. Again, the goal here is to model useful spatial information that can aid in the classification.

This is the central idea behind CNNs. Learning weights associated with kernels allows for creating spatial abstractions of the data that can be useful for differentiating classes.

Batch Normalization

- Reduce network instability and avoid gradient explosion
- Normalize output to minimize the impact of very large weights
- Maintain balance in weights
- Keep data normalized throughout the network
- Can train the batch normalization parameters (g and b)
- Increase training speed
- Reduces overfitting
- Can use a larger learning rate
- Occurs on a mini-batch-by-mini-batch basis



17

Batch normalization is generally used to minimize overfitting in convolutional neural networks. Batch normalization is used to normalize the learned weights and prevent the activation or signal from becoming extremely large or small, a problem known as gradient explosion. This maintains balance in the weights, keeps the data normalized, and can increase training speed and convergence time. The normalized weights can then be augmented using a coefficient (g) and an offset (b), which are both trainable.

In CNNs specifically, it is common to apply 2D batch normalization after every 2D convolution layer in the network. You can also apply 1D batch normalization within the fully connected component of the network.

Layer Activation

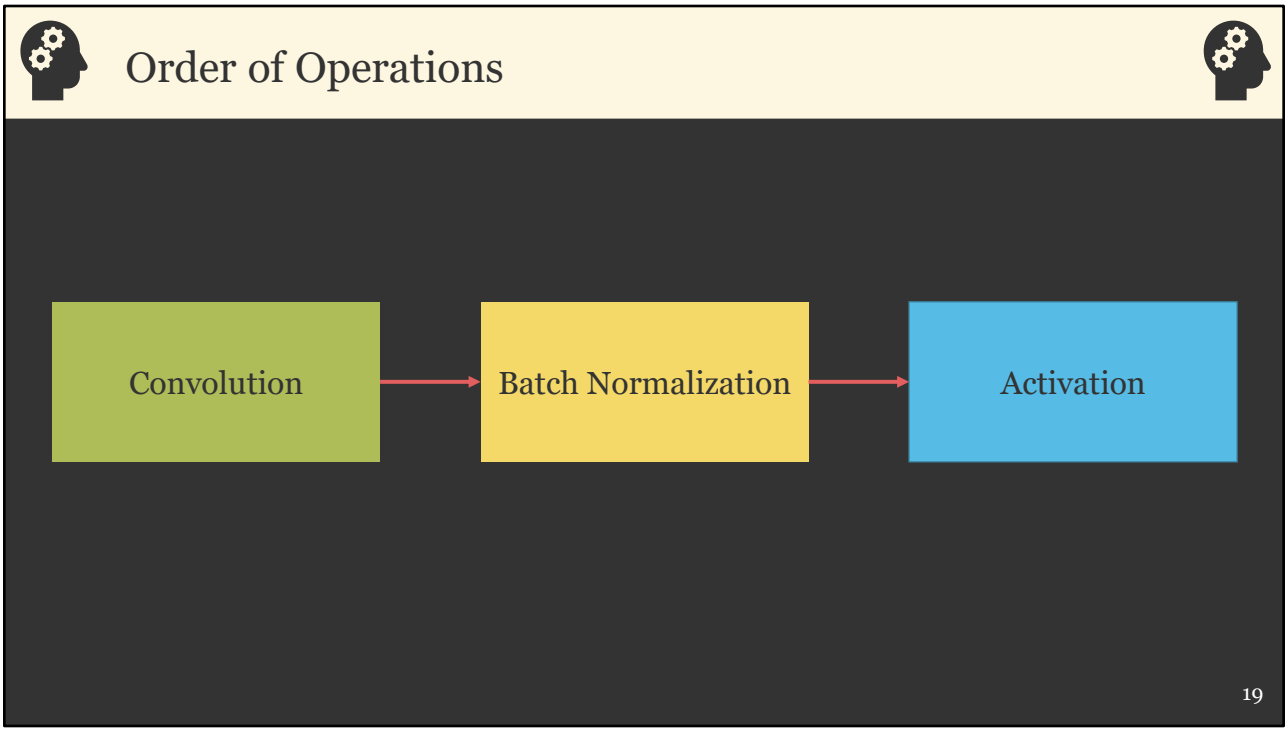
- Apply activation function to model nonlinear relationships
- ReLU is a common choice

18

Different activation functions can be applied following the 2D convolution layers and 2D batch normalizations. It is common to use ReLU or a similar method, such as Leaky ReLU or Parameterized ReLU.

Remember that ReLU converts all negative activations to 0 and maintains all positive activations. It is a simple means to add non-linearity to the process.

As mentioned in prior modules, if "dying ReLU" is an issue in your network, it would be good to experiment with Leaky ReLU or Parameterized ReLU.



Within the sequence of operations, it is commonly preferred to apply batch normalization before applying the activation function. You will see this put into practice in the PyTorch modules.

Max Pooling

- Reduces size of tensor in row and column dimensions by taking the max value in a windows
- 2 x 2 window
- Allow for learning patterns at different spatial scales for context
- Common to use stride of (2,2)

3	5	1	7
5	6	3	6
1	3	1	1
2	4	2	1

6	7
4	2

20

Once the filter weights and bias are learned, they are applied to the image as a moving window to generate feature maps. This is accomplished using a dot product: the cell values are multiplied by their associated weights, the products within the window are added together, and the bias term is added. The kernel moves over the input data to process the entire array.

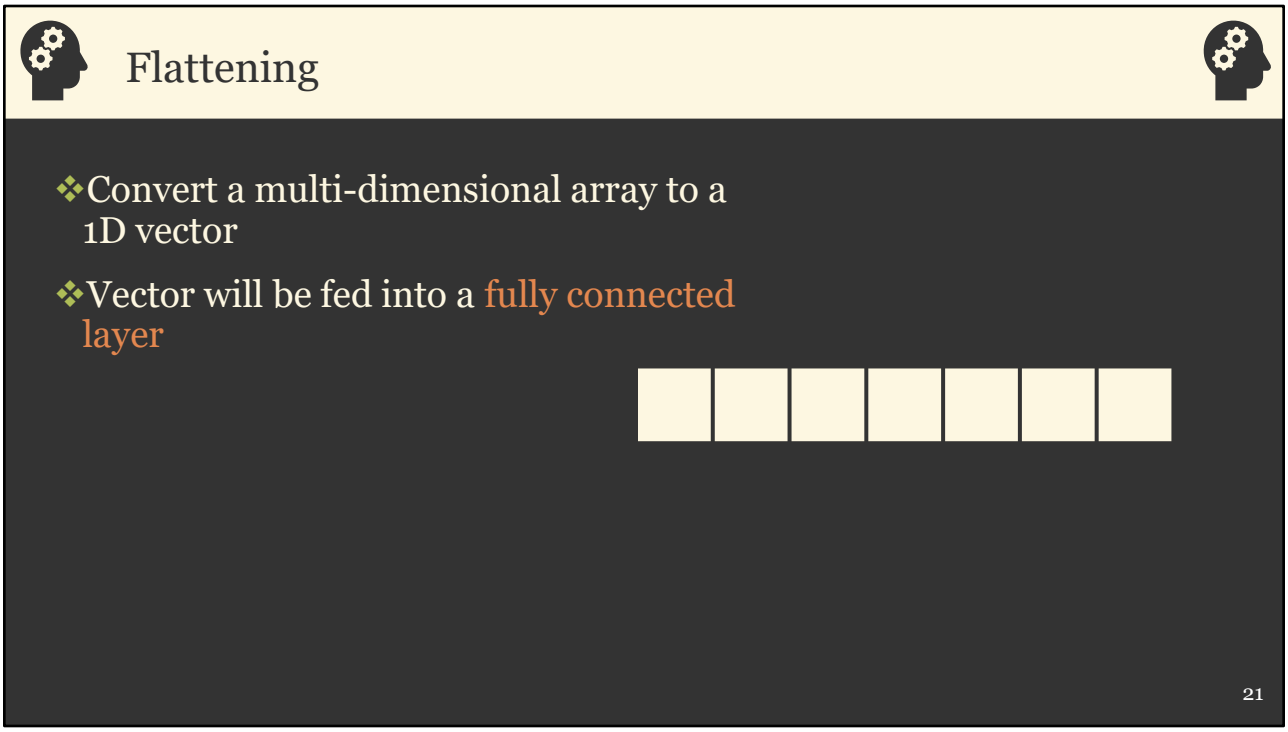
In order to learn patterns in the spatial dimensions at different scales, the size of the array in the spatial dimensions needs to be reduced before it is used as input to the next convolutional layer. Without this reduction in size, it would not be possible for the CNN to learn patterns at different scales.

This is commonly accomplished using max pooling layers. Max pooling consists of applying a 2x2 moving window to the data and returning the maximum value from the 4 cells. An example of such an operation is shown on the slide. It is also common to use a stride of (2,2), which will result in decreasing the size of the array in the spatial dimensions by a factor of two.

Other operations can be used to reduce the size of the array, such as mean pooling. However, max pooling has been shown to be useful and is the most common method used.

As mentioned above, increasing the stride of the 2D convolution operations

will also decrease the size of the array. However, pooling operations are generally used instead. However, some architectures, such as ResNet, will use a stride of 2 in some of the 2D convolution layers as opposed to pooling to decrease the size of the array in the spatial dimensions.



Once a series of convolution, batch normalization, activation function, and max pooling operations have been applied, the resulting data are flattened to a 1D tensor (or vector) as input to fully connect layers. This 1D tensor becomes the input features, which have been learned at multiple spatial scales, that act as input to the final classification performed by the network using the fully connected layers.

The size of the 1D tensor will depend on (1) the number of feature maps generated by the final 2D convolution layer and (2) the size of the final array in the spatial dimensions following the series of max pooling operations. Since the size of the array prior to flattening will vary based on the number of max pooling operations performed and the input width and height of the image, the architecture requires the input images to have the same size. In short, the requirement for consistent input heights and widths is a product of flattening the array prior to the fully connected component of the architecture.

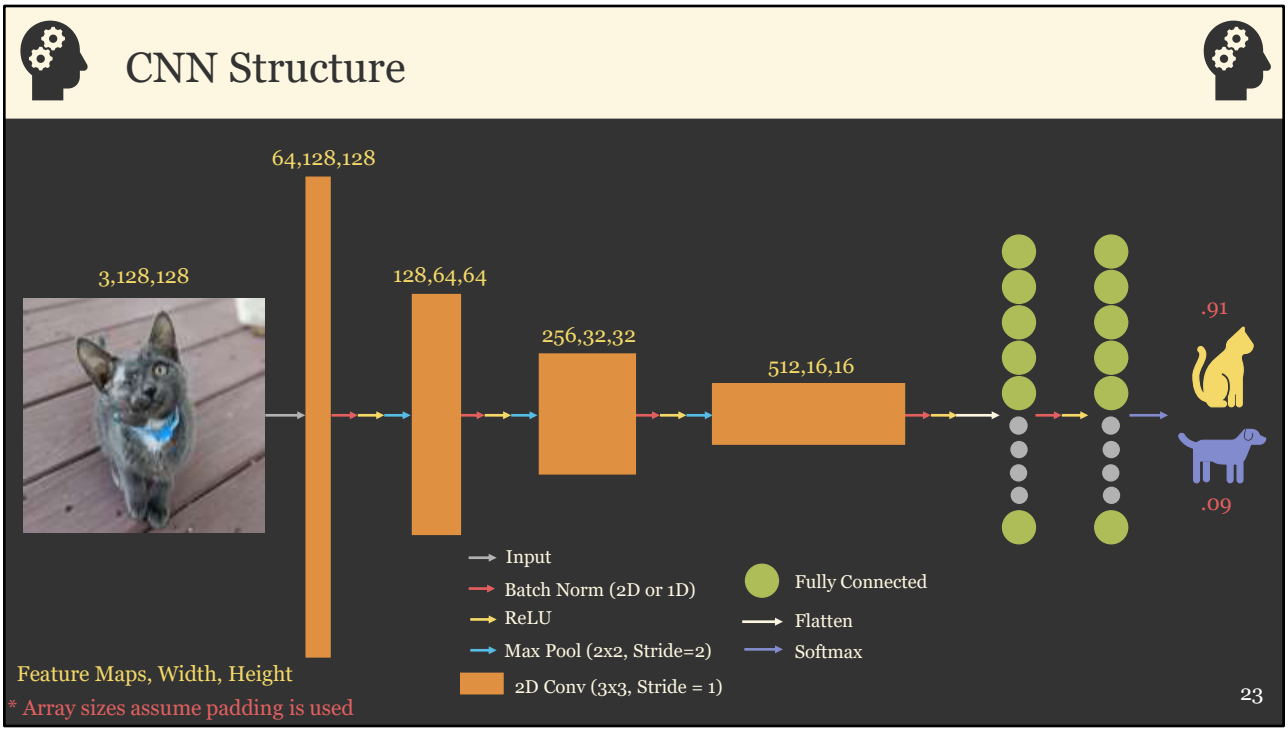
Fully Connected Layers

- Referred to as: fully connected or densely connected layers
- Expect a vector (1D tensor)
- Occur after convolutional layers and flattening
- Can incorporate 1D batch normalization and an activation function (ReLU)
- Can incorporate dropouts
- Final dense layer can use a sigmoid activation for binary classification or a softmax activation for multiclass classification

22

A CNN's ultimate goal is to label the entire image or input to one category from a set of predefined classes. As a result, the end of the process involves using fully connected or dense layers that accepted a 1D vector or tensor and are used to perform classification.

Depending on the loss metric being used, the final fully connected layer can return logits or the logits can be passed through an activation function to generate a rescaled logit or estimated class probability. A sigmoid activation function is used when a single logit is returned for a binary classification problem, resulting in an estimated probability of the sample being an example of the positive case. For a multiclass classification, a softmax activation can be used to obtain probabilities for each class. These probabilities will sum to 1.



Let's now bring all of these concepts together to explore the complete architecture of a simple CNN.

This CNN accepts images with 3 channels and a width and height of 128 pixels. The image is then passed to a 2D convolutional layer that learns 64 filters that are applied to the input image to generate 64 feature maps. The 2D convolutional layer uses a kernel size of 3x3, a stride of 1, and a padding of 1 so that the size of the array in the spatial dimensions is not changed by the operations. The resulting feature maps then pass through a 2D batch normalization layer and a ReLU activation function to add non-linearity. Max pooling is used to reduce the size of the array in the spatial dimensions by half using a kernel size of 2x2 and a stride of 2. After these steps, the array now has a shape of (64,64,64).

The data then pass through the following layers:

2D Convolution (kernel = 3x3, stride=1, padding=1) → 2D Batch Normalization → ReLU → Max Pooling (kernel = 2x2, stride=2) → 2D Convolution (kernel = 3x3, stride=1, padding=1) → 2D Batch Normalization → ReLU → Max Pooling (kernel = 2x2, stride=2) → 2D Convolution (kernel = 3x3, stride=1, padding=1) → 2D Batch Normalization → ReLU

After this sequence of operations are applied, the array will have a size of (512,16,16). This array will then be flattened to a 1D vector with a length of 131,072 (512×16×16). The flattened data are then passed through the following sequence:

Fully Connected → 1D Batch Normalization → ReLU → Fully Connected

The first fully connected layer will have an input size equal to 131,072. Its output size will be the input size for the last fully connected layer. The last fully connected layer will output two logits, one for each class. Note that this could also be designed to output one logit in the case of a binary classification. If 1 logit is predicted, then it can be passed through a sigmoid activation to obtain a probability for the positive case. If two or more logits are predicted, then they can be passed through a softmax activation to obtain probabilities that sum to 1.

In this example, the image of my cat Peri is predicted to be a picture of a cat with a 0.91 probability as opposed to a picture of a dog.

CNN Architectures

(Bands, Height, Width) → (Feature Maps 1, Height/2, Width/2) → (Feature Maps 2, Height/4, Width/4) → (Feature Maps 3, Height/8, Width/8) → (Feature Maps 4, Height/16, Weight/16) → Flatten → (Feature Maps 4 * Height/16 * Weight/16, Fully Connected 1) → (Fully Connected 2, Number of Classes)

(3, 128, 128) → (8, 64, 64) → (16, 32, 32) → (32, 16, 16) → (64, 8, 8) → Flatten → (4096, 1024) → (1024, 10)

Blue = Input
Green = 2D Convolution + ReLU + Batch Norm (Size=3x3, Stride=1, Padding=1 or "same")
Red = Max Pooling (Size = 2x2, Stride=2)
Purple = Flatten Tensor to 1D
Orange = Fully Connected Layers + ReLU + Batch Norm
Yellow = Fully Connected

* Array sizes assume padding is used

24

Here is a series of steps that outlines another simple CNN architecture.

The 2D convolution layers are used to learn filters that are then applied to the input image or prior feature maps to create feature maps. All kernels have a size of 3x3, a stride of 1, and padding to maintain the size of the array in the spatial dimensions. After each 2D convolutional layer except the last layer, 2D batch normalization and a ReLU activation is applied.

The max pooling operations, with a window size of 2x2 and a stride of 2, decrease the size of the array in the spatial dimensions by a factor of two.

After the convolutional component of the model, the array is flattened to a 1D vector. The data are passed through a fully connected layer followed by 1D batch normalization and a ReLU activation function. The last fully connected layer has an output size of 10, representing 10 differentiated classes.

The resulting 10 logits can be converted to probabilities that sum to 1 using a softmax function.

CNN Architecture Trainable Parameters

Layer	Out Shape	Number of Trainable Parameters
2D convolution 1	[B, 16,64,64]	(3 × 16 × 9) + 16 = 448
2D batch normalization 1	[B, 16,64,64]	16 × 2 = 32
ReLU activation 1	[B, 16,64,64]	0
2D max pooling 1	[B, 16,32,32]	0
2D convolution 2	[B, 32,32,32]	(16 × 32 × 9) + 32 = 4,640
2D batch normalization 2	[B, 32,32,32]	32 × 2 = 64
ReLU activation 2	[B, 32,32,32]	0
2D max pooling 2	[B, 32,16,16]	0
2D convolution 3	[B, 64,16,16]	(32 × 64 × 9) + 64 = 18,528
2D batch normalization 3	[B, 64,16,16]	64 × 2 = 128
ReLU activation 3	[B, 64,16,16]	0
2D max pooling 3	[B, 64,8,8]	0
2D convolution 4	[B, 128,8,8]	(64 × 128 × 9) + 128 = 73,856
2D batch normalization 4	[B, 128,8,8]	128 × 2 = 256
ReLU activation 4	[B, 128,8,8]	0
Flatten		
Fully connected 1	[B, 512]	(8,192 × 512) + 512 = 4,194,816
1D batch normalization 1	[B, 512]	512 × 2 = 1,024
ReLU activation 1	[B, 512]	0
Fully connected 2	[B, 256]	(512 × 256) + 256 = 131,328
1D batch normalization 2	[B, 256]	256×2 = 512
ReLU activation 2	[B, 256]	0
Fully connected 3	[B, 5]	(256 × 5) + 5 = 1,285
	Total	4,426,917

25

This table summarizes the number of trainable parameters in a CNN architecture. The max pooling and ReLU activation functions do not include trainable parameters while the 2D convolutional, batch normalization, and fully connected layers do have trainable parameters.

The number of trainable parameters for a 2D convolutional layer is equal to the number of input channels or feature map × the number of learned kernels × the kernel size. For the first layer, which uses a kernel size of 3×3, accepts 3 input channels, and produces 16 feature maps, this yields 432 trainable parameters. There is also a trainable bias term for each learned kernel, adding an additional 16 parameters for a total of 448 parameters.

The number of trainable parameters for a batch normalization layer is equal to the number of input feature maps × 2 since each feature map has an associated gain and shift parameter.

The number of trainable parameters in a fully connected layer is equal to the number of input features × the number of output features + the number of bias terms, which is equal to the number of output features since each neuron has an associated bias.

In this architecture, there are a total of 4,426,917 trainable parameters. The

majority of the trainable parameters are associated with the fully connected layers due to the large number of neurons contained in the layers.

Settings and Considerations

Let's talk about some specifics associated with using CNNs and 2D convolutional layers.

First, you can change the size of the moving window or kernel. It is common to use a 3x3 or a 5x5 window. Within these windows, each cell represents a weight that can be learned. There are separate learnable weights for each input channel for each generated feature map.

It is also possible to insert zeros into the kernel to allow for learning patterns between cells that are not directly adjacent. We will discuss this later in the context of semantic segmentation.

Number of Filters

- Number of filters, kernels, or weight matrices learned to apply to input to obtain new feature maps
- Generally, increases as you move through network

(3, 128, 128) → (64, 128, 128,) → (128, 64, 64) →
(256, 32, 32) → (512, 16, 16)

* Array sizes assume padding is used

28

You can also change the number of filters learned. It is common to increase the number of filters as you move through the network and the arrays become smaller in the width and height dimensions.

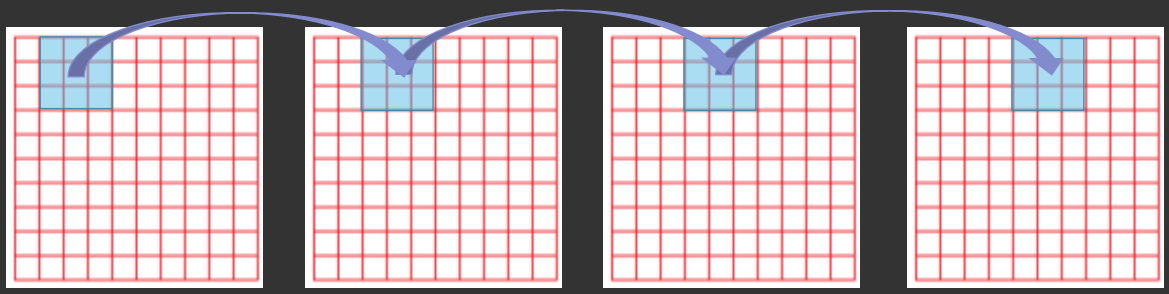
The input of a 2D convolutional layer will be the feature maps produced from the prior layer that were subsequently aggregated using max pooling.

As the size of the CNN architecture increases, the number of kernels and associated weights will increase, resulting in more trainable parameters. It is not uncommon for CNNs to contain millions of trainable parameters.

This can cause an issue with overfitting if the training dataset is small and/or no methods are used to minimize overfitting.

Stride

- How far widow shifts as it moves over the image
- Generally, use a stride of 1 so that each pixel is the center of a window
- Strides greater than 1 will downsample image
- Tend to use max pooling with stride of 2 to downsample



29

Stride relates to how far the moving window shifts in the x and y directions as it moves over the image. If a stride of (1,1) is used, then each cell or pixel in the array will be the center of the moving window.

If a stride larger than (1,1) is used, then the size of the array will decrease in the spatial dimensions. It is generally preferable to use max pooling as opposed to a larger stride to reduce the number of rows and columns in the data array, as discussed above.

Generally, we will use a stride of (1,1) for 2D convolutional layers since we do not want to change the size of the array in the spatial dimensions. In contrast, a stride of (2,2) is often used for max pooling because we want to reduce the size of the array in the spatial dimensions by a factor of two.

Padding

- Edge pixels do not have full set of neighbors
- Results in reduction of number of rows and columns in resulting feature map
- Can use padding to obtain same number of rows and columns
- Common to pad with zeros

0	0	0	0	0	0
0	3	5	1	7	0
0	5	6	3	6	0
0	1	3	1	1	0
0	2	4	2	1	0
0	0	0	0	0	0

30

Edge cells or pixels will not have a full set of neighbors. If only full windows are used, then the output size following the application of a convolution operation will be smaller than the input in the width and height dimensions.

To alleviate this issue, arrays can be padded with zeros in order to maintain the image size following the convolution operations. This is demonstrated for the corner pixel in the pictured array.

CNN Architecture without Padding

- Assuming 3x3 window and input spatial dimensions of 512x512
- (512, 512) → (254, 254) → (127, 127)

31

Again, if padding is not applied, then only cells that have a full set of neighbors will be maintained after the learned kernels are applied to create feature maps. This slide provides an example of spatial dimensions when a series of 3 2D convolutions and two max pooling layers are applied to an image with original spatial dimensions of 512x512, a kernel size of 3x3, and no padding.

Overfitting

What causes overfitting?

- Limited number of training samples
- Large ANN
- Many trainable parameters
- Poor ANN design
- Training for too many epochs

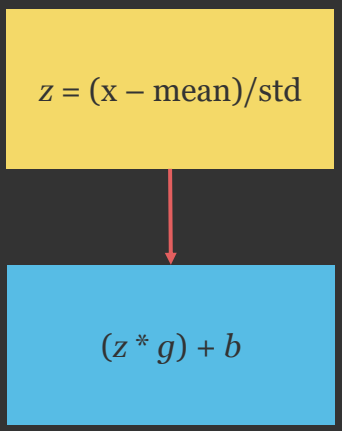
33

As with all deep learning methods, overfitting can be an issue for CNNs.

This is especially true if the number of training samples is limited, the CNN is large and there are many trainable parameters, the architecture is poorly designed, and/or the model is trained for a large number of epochs or iterations through the data.

Batch Normalization

- Reduce network instability and avoid gradient explosion
- Normalize output to minimize the impact of very large weights
- Maintain balance in weights
- Keep data normalized throughout the network
- Can train the batch normalization parameters (g and b)
- Increase training speed
- Reduces overfitting
- Can use a larger learning rate
- Occurs on a mini-batch-by-mini-batch basis



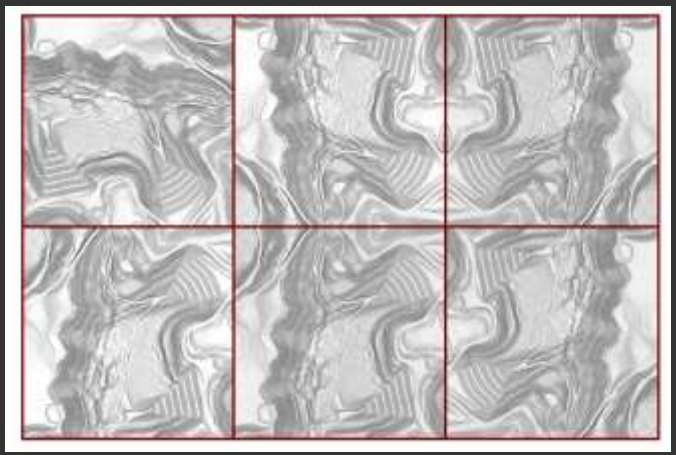
34

Batch normalization is generally used to minimize overfitting in convolutional neural networks. It is used to normalize the learned weights and prevent the activation or signal from becoming extremely large or small, a problem known as gradient explosion. This maintains balance in the weights, keeps the data normalized, and can increase training speed and convergence time. It is also commonly used instead of dropouts.

In CNNs specifically, it is common to apply 2D batch normalization after every 2D convolution layer in the network and 1D batch normalization after all fully connected layers other than the final.

Data Augmentation

- Add random noise to data
- Generate more variety and more samples
- Common when working with images
- Augmentations:
  - Rotate/Flip
  - Blurring/Sharpening
  - Contrast
  - Crop/Pad
  - Stretch/Distort
  - Brightness/Saturation/Hue
  - Shift/Zoom



Note: Some augmentations only work for RGB data (saturation and hue adjustments).

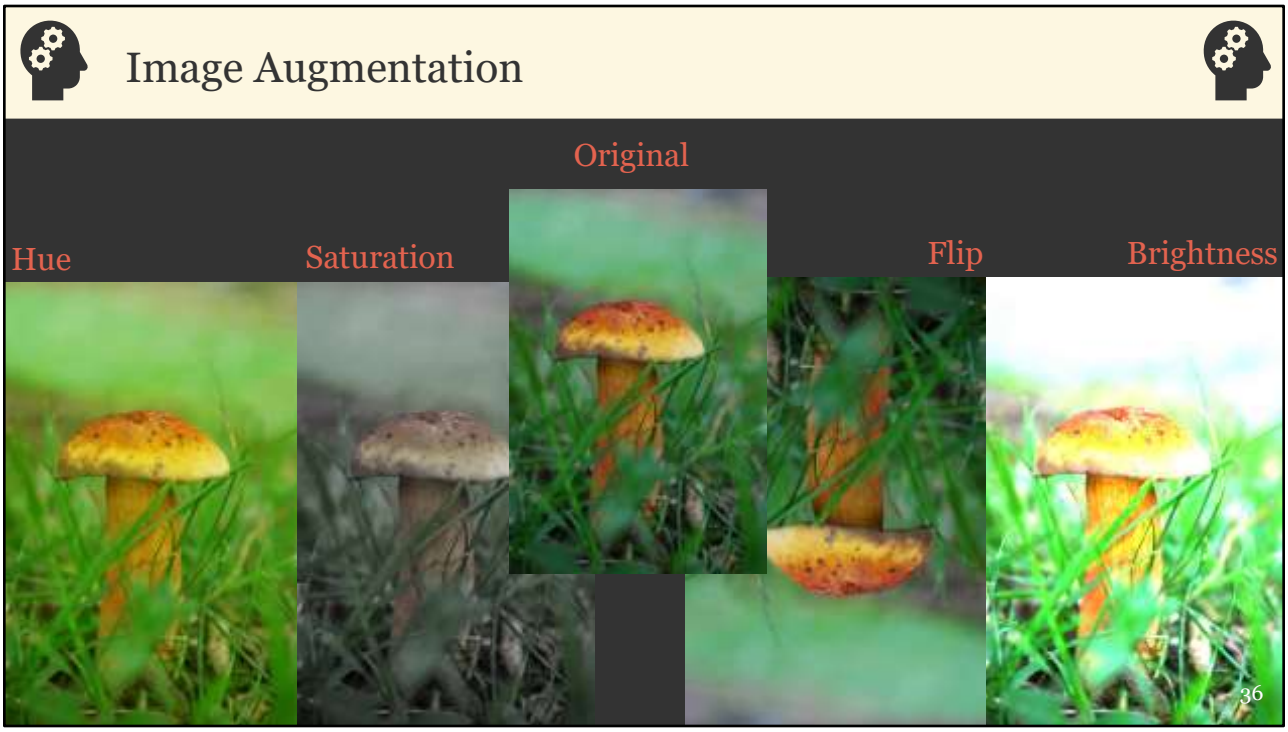
35

One common means to combat overfitting in CNNs is to augment the image data.

A wide variety of data augmentations can be applied including rotation/flips; blurring and sharpening; contrast alterations; cropping and padding; stretching and distortion; changing the brightness, saturation, and/or hue; and shifting or zooming within the array.

Generally speaking, data augmentation is a good idea when training CNN-based architectures.

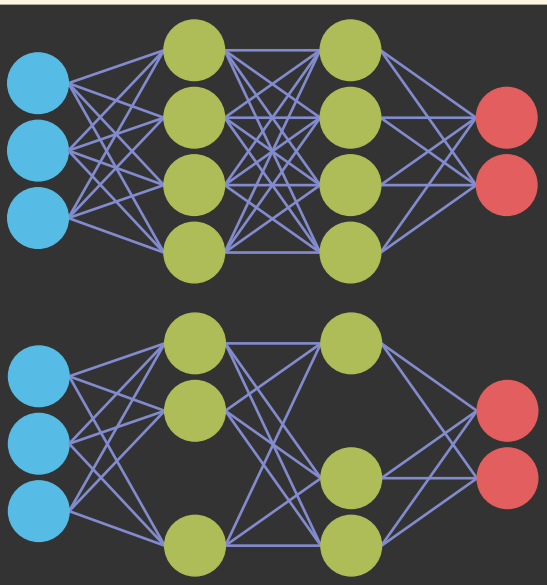
The example on the slide represents random rotations and flips of some digital terrain data.



This slide provides some example alterations of an image.

Dropouts

- Minimize overfitting by "dropping out" or ignoring neurons in the network
- Often between 0.5 and 0.8
- Can take longer for model to stabilize or converge



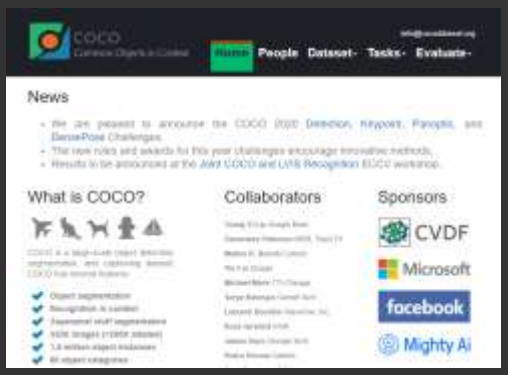
37

Dropouts can also be used to potentially reduce overfitting. Dropouts can be applied for both convolutional and fully connected layers.

Transfer Learning

- Start with weights learned from a larger training dataset
- Refine weights using new data

http://www.image-net.org/



https://cocodataset.org/#home

It is also possible to initialize your model with weights learned from another, larger dataset, such as ImageNet or COCO.

The idea here is that images contain some common, basic features and spatial patterns, so starting from these pre-trained parameters will generally be better than starting from random parameters even if your classification problem is different from the original problem. This can even be true if the defined classes are different.

Others

- Early stopping
- Learning rate schedulers
- Changing loss functions
- Custom training loop augmentations

39

There are some other methods that can be used to potentially improve model performance. These include stopping the training process early to prevent overfitting, adjusting the learning rate during the learning process, using a different loss function (e.g., Dice, Tversky, or focal Tversky), and applying other customizations to the training loop.

Creating Image Chips

40

Image Chips

- Define shape: (3, 512, 512), (3, 256, 256), (3, 128, 128), etc.
- Common formats: JPEG, PNG, TIFF
- Labels: class label for each image (use numeric codes to represent classes) (0 to n-1 or 1 to n)

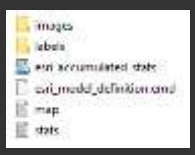
41

Before training a CNN model, you will need to develop training, validation, and testing data of a defined size. Some common sizes are listed on this slide. Note that larger tensors will require more memory and computational resources to process. You will also have to feed in smaller mini-batches. The maximum mini-batch size will depend on the size of the chips, architecture of the CNN, and GPU memory available. Using multiple GPUs or having more VRAM available generally allows for larger mini-batch sizes.

In GIS and remote sensing, it is common to work with large raster grids. These large grids will need to be divided into small image chips to feed into the DL model.

Common raster formats for generating image chips include JPEG, PNG, and TIFF. Since the goal is to predict labels or classes for each example, you will also need to provide the labels associated with each image for training, validation, and testing.

Using ArcGIS Pro



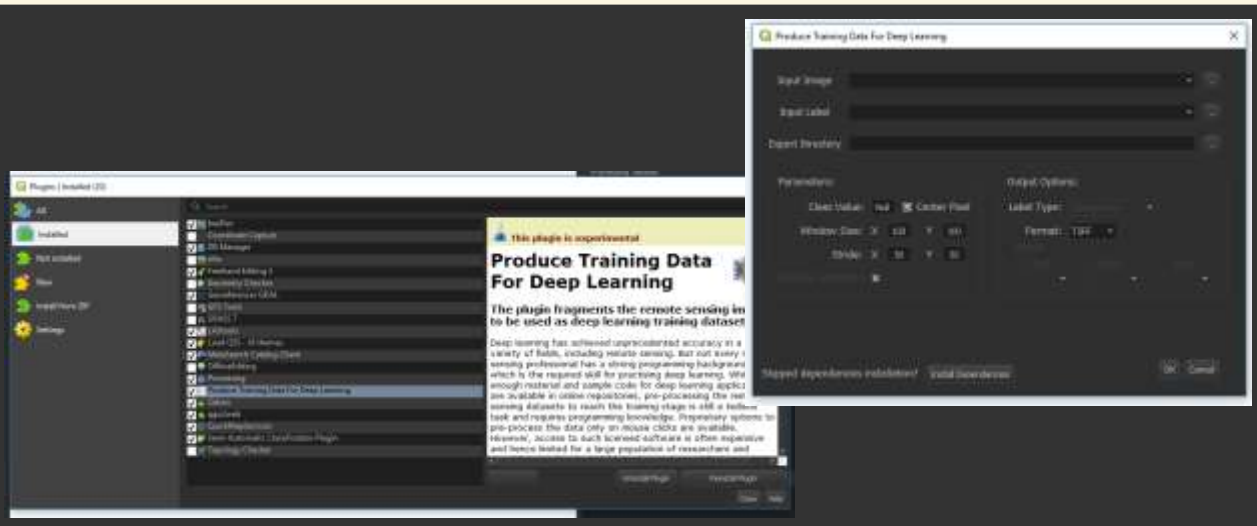
https://pro.arcgis.com/en/pro-app/tool-reference/image-analyst/export-training-data-for-deep-learning.htm

https://developers.arcgis.com/python/api-reference/arcgis.learn.html

42

ArcGIS Pro provides the Export Training Data for Deep Learning Tool that can be used to create image chips for training, testing, and validation. This tool can also be accessed via ArcPy and the arcgis.learn module. It requires a valid Spatial Analyst or Image Analyst extension license.

Using QGIS



https://github.com/PratyushTripathy/QGIS-Plugin-Produce-Training-Samples-For-Deep-Learning

Plugins have also been developed for QGIS to generate image chips. This provides an open-source and free alternative to ArcGIS Pro.

Architectures

Architectures

Architecture	Year of Publication	Publication
LeNet	1998	(Lecun et al., 1998)
AlexNet	2012	(Krizhevsky et al., 2012)
NiN	2013	(Lin et al., 2013)
VGGNet	2014	(Simonyan and Zisserman, 2014)
GoogleNet/Inception-v1	2015	(Szegedy et al., 2015a)
Inception-v2/v3	2015	(Szegedy et al., 2015b)
DenseNet	2016	(Huang et al., 2017)
ResNet	2016	(He et al., 2015)
Inception-v4	2017	(Szegedy et al., 2017)
Inception-ResNet	2017	(Szegedy et al., 2017)
Xception	2017	(Chollet, 2017)
ResNeXt	2017	(Xie et al., 2017)
MobileNet	2017	(Howard et al., 2017)
MobileNet-v2	2018	(Sandler et al., 2018)
MobileNet-v3	2019	(Howard et al., 2019)
EfficientNet	2019	(Tan and Le, 2019)

45

In this final section of the module, we will discuss different CNN architectures with a primary focus on their key contributions. Although CNN architectures were introduced in the late 1990s (i.e., LeNet), advancement and adoption of these architectures accelerated after 2012 when AlexNet won the 2012 ImageNet Large Scale Visual Recognition Challenge.

The rapid development of CNN architectures over the last decade is at least partially attributed to the availability of large training datasets and GPU-based computation.

Architectures

- Pre-defined architectures
- Examples = ResNet, DenseNet, InceptionNet, EfficientNet, MobileNet, DPN, VGG
- Different version of underlying architecture with varying numbers of layers or operations: ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-152
- Allow for application of pre-trained weights
- Can reduce training time and overfitting
- Consider when using small training set

46

Since the development of CNNs, a variety of architectures have been proposed with varying components and designs. Some examples are listed on this slide.

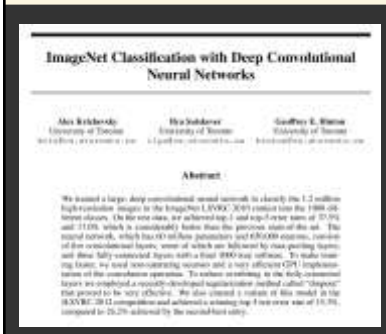
For common architectures, such as those listed here, it is common for trained models to already be available which were trained using large databases, such as ImageNet or COCO. When a limited set of training data are available, it might be useful to initiate a model using these pre-trained weights/parameters. This can be true even if the images being analyzed and the classes being differentiated are different between the original and current use case. For example, a model for classifying aerial images into different land cover types could be initiated using ImageNet weights, even though this dataset consists of photographs. The weights/parameters can then be adjusted by learning from the new data, which can be more efficient than training from random weights. This can also help reduce overfitting.

If you design a new architecture for a problem, you could generate weights from ImageNet, COCO, or some other dataset. However, you would need to undertake this training process yourself, which can be very memory intensive since the datasets are so large. This highlights the value of using existing architectures with pre-trained weights.

Note that the use of existing architectures and pre-trained weights/parameters

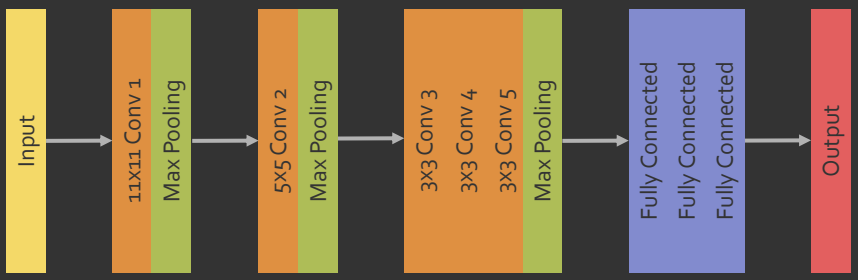
can be limiting. For example, the model may only accept 3-band images since it was originally trained on 3-band data. However, there are methods available to edit or generalize models. Some of these method will be demonstrated in the PyTorch modules.

AlexNet


ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

Abstract



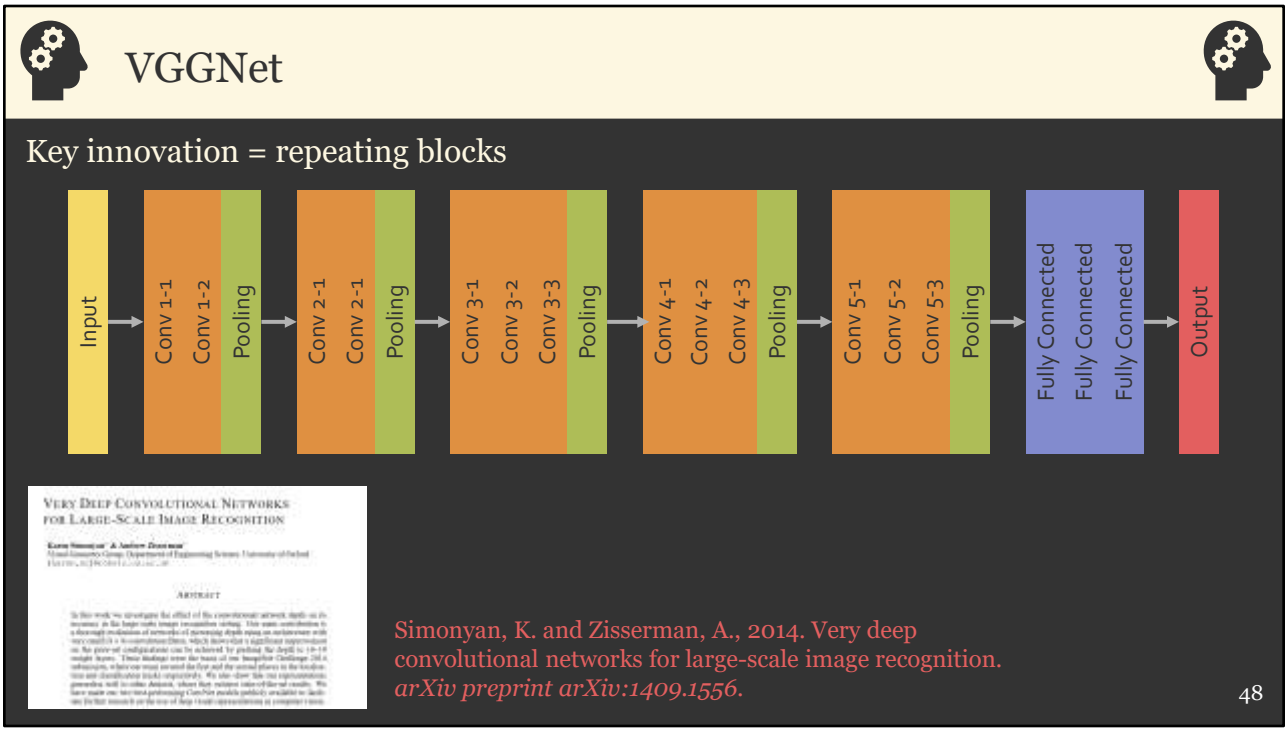
Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2017. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), pp.84-90.

47

We will now explore a few famous architectures as examples. We will begin with AlexNet, which was released in 2012 and helped to spur the interest in CNNs for image labeling tasks.

This model consists of the following steps:

2D Conv → Max Pooling → 2D Conv → Max Pooling → 2D Conv → 2D Conv → 2D Conv → Max Pooling → Flatten → Fully Connected → Fully Connected → Fully Connected

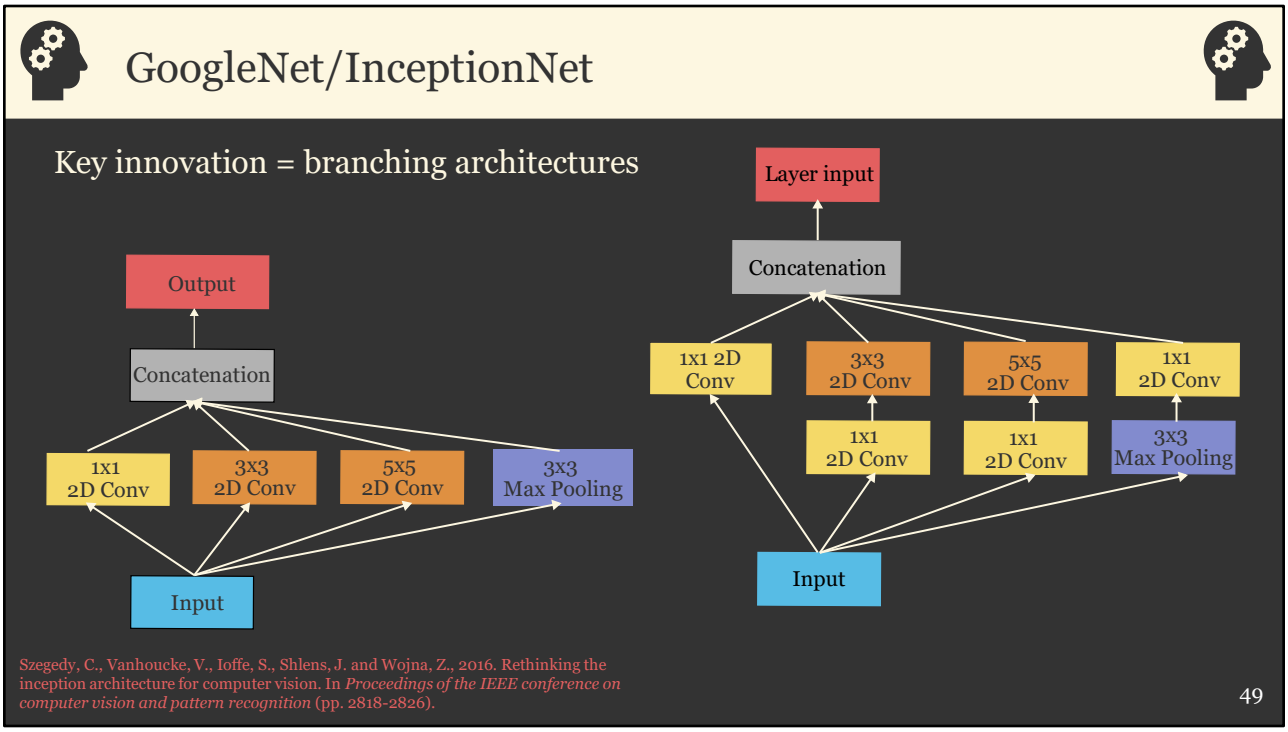


VGGNet-16 has the architecture diagrammed above. Similar to other CNNs, it consists of a few fairly simple building blocks: 2D convolution, max pooling, activation functions, and fully connected layers. Even though the original VGGNet-16 did not include batch normalization layers, later modifications added these.

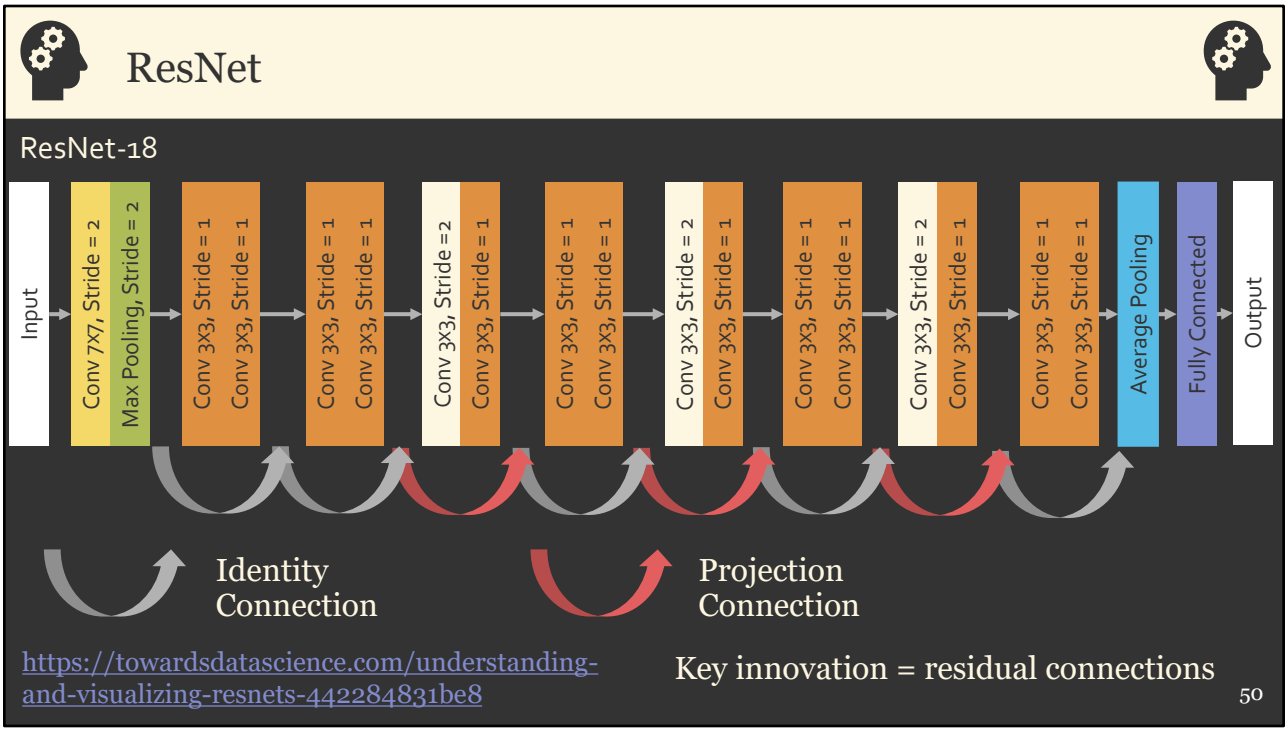
VGGNet-16 has a large number of trainable parameters: more than 100 million!

Its key contribution is introducing the use of repeating sequences of blocks of similar operations. This design characteristic is reflected in many later architectures.

Note that there are different versions of VGGNet that have different numbers of operations or blocks.



The GoogleNet/InceptionNet family introduced many innovations and design considerations. One of their key innovations was using branching structures with varying convolutional window sizes. This allows for capturing spatial patterns at different scales then concatenating the results before passing them on to the next operation in the sequence.



It would make sense that increasing the size, complexity, and/or number of layers in a CNN would result in improved model performance due to the ability to model more complex relationships and abstract the data to a greater degree. However, it has been found that this is not the case. Larger networks can lead to overfitting, especially when the training dataset is not large. Perhaps more problematic, when networks get very large, early layers are very separated from later layers, which can result in very small or infinitely small gradients. This is known as the vanishing gradient problem. This can result in the model being very difficult to train effectively. Thus, researchers hit a wall as continuing to increase the depth of the CNN architecture proved to be problematic.

One solution to this problem was introduced by the ResNet architecture. The idea here is to use shortcut connections in which some layers skip past components of the model and are merged back in at a later point by simple addition. This allows for less separation between earlier layers and later layers and reduces the impact of the vanishing gradient problem. In other words, it provides a shorter path over which the gradients can flow.

Adding shortcuts in the architectures allows for shorter paths over which the gradient can be calculated since they bypass components of the model architecture. Practically, learning the original signal plus an added signal is easier than learning just the new signal generated by the block. Learning a residual, or the difference between the input

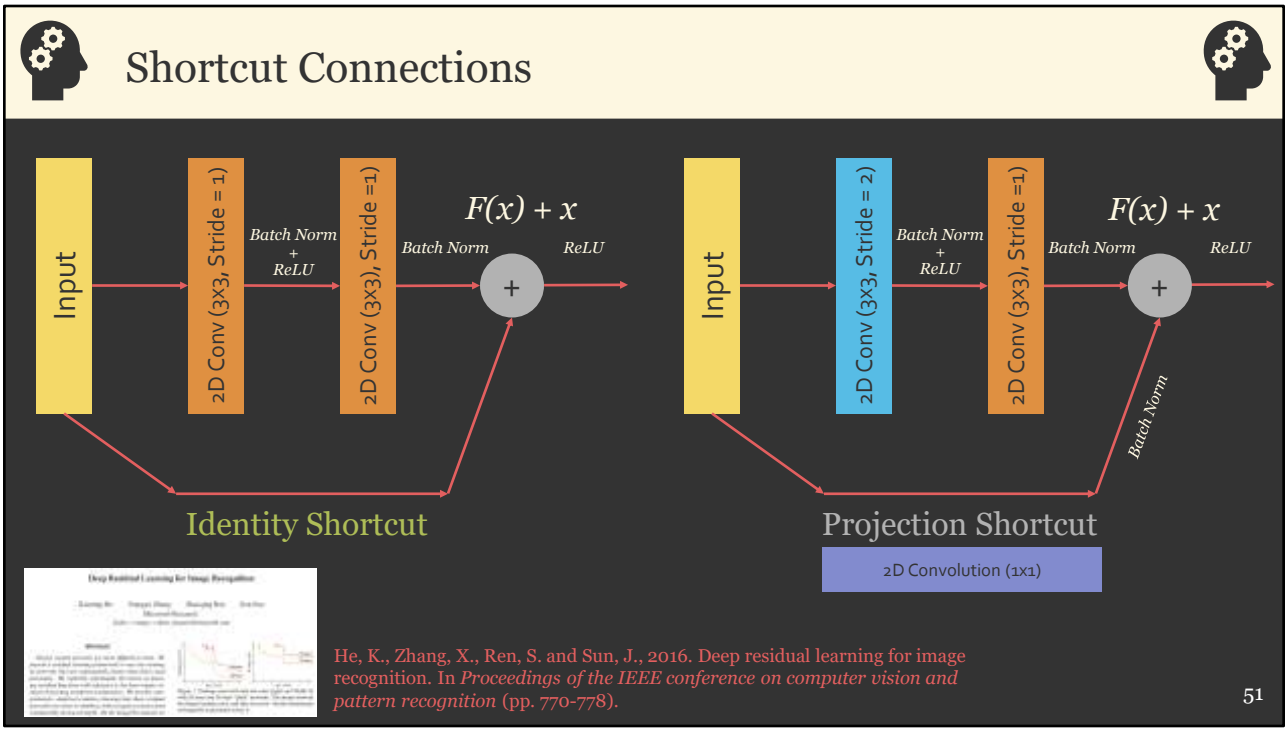
and output of a residual block, is easier than learning the original feature maps.

The diagram on this page conceptualizes a ResNet-18 architecture. The first component is 2D convolution with a kernel size of 7x7 and a stride of 2. This if followed by a max pooling operation with a stride of 2. Both operations will decrease the size of the array in the spatial dimensions. This is then followed by a series of 2D convolution operations, some of which have a stride of 2 while others have a stride of 1. When a stride of two is used, the size of the array is decreased in the spatial dimensions. This serves a similar purpose as max pooling in other CNN architectures explored here.

Within the convolutional component of the model, the gray and red arrows represent the shortcut connections. A gray arrow indicates an identity shortcut connection in which the output is simply added to the arrays at the new location in the architecture. A red arrow indicates a projection shortcut connection in which a technique is used to make sure the array sizes and number of channels match between the input and output of the block. This is required so that the data that is skipping over some of the steps can be added to the output of these steps.

The last phase of the model consists of average pooling. Note that this can be replaced with adaptive average pooling, which allows the model to accept images with different sizes in the spatial dimension. A final fully connected layer is then used to perform the classification.

There are actually a variety of ResNet architecture with varying number of layers: ResNet-18, -34, -50, -101, and -152.

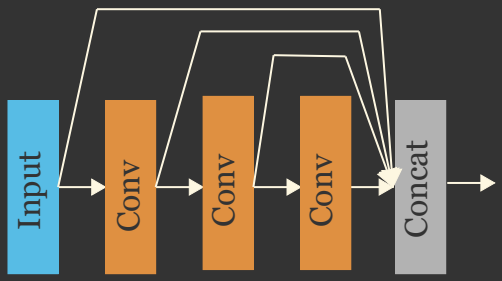


This slide conceptualizes the two types of shortcut connections used within the ResNet architecture. When the size of the spatial dimensions and number of channels of the output data from the block and the data that are being fed through the shortcut are the same, there is no need to alter the shape of the array, and the data can simply be added together. This is known as an identity shortcut connection.

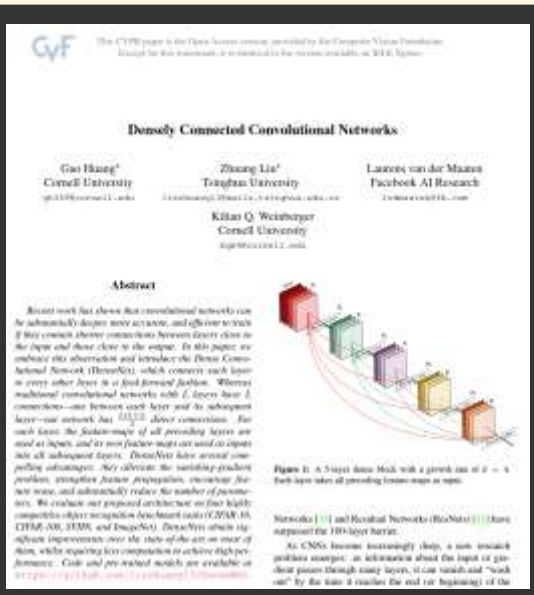
When the arrays that are to be merged have different sizes in the spatial dimensions and/or different numbers of channels, the data being fed along the shortcut path must be altered to match the output data from the block. This type of shortcut is called a projection shortcut connection. There are several ways to accomplish this resizing. One option is to use a 2D convolutional layer with a kernel size of 1x1 and a stride of 2 to decrease the array size by half. If the number of channels needs altered to match, 1x1 2D convolution can also be used. A stride of 1 will not decrease the array size but can be used to alter the number of channels or feature maps.

DenseNet

Key innovation = many residual connections/connections between non-adjacent layers



Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K.Q., 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700-4708).



Densely Connected Convolutional Networks

Gao Huang* Cornell University

Zhuang Liu* Tsinghua University

Laurens van der Maaten Facebook AI Research

Kilian Q. Weinberger Cornell University

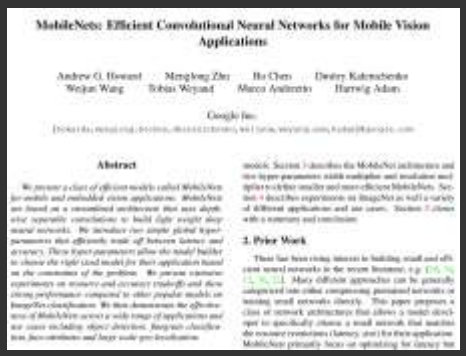
Abstract

52

DenseNets extend the basic idea of ResNet. A large set of skip connections are used to allow for connections between both adjacent and non-adjacent blocks.

MobileNet

- Efficient architecture for use on mobile devices
- Depth-wise separable convolution



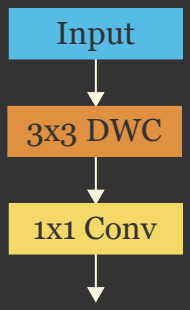
MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Andrew G. Howard Menglong Zhu Bo Chen Dmitry Kalenichenko
Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam

Google Inc.

Abstract

1. Prior Work



Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H., 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

53

MobileNets focus on creating efficient architectures that can be used on mobile and edge devices. One key innovation is the use of depth-wise separable convolution.

Depth-wise separable convolution consists of two components: depth-wise and point-wise convolution. The depth-wise component consists of performing separate convolution operations on each input channel or feature map to generated the same number of output feature maps (focus on spatial information).

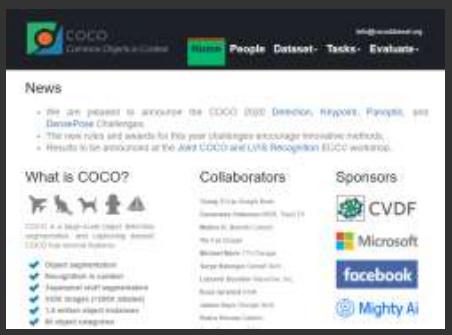
The point-wise component uses 1x1 2D convolution to change the number of feature maps and share information between the learned representations.

Depth-wise separable convolution is computationally lighter than traditional convolution (less matrix multiplication required).

Pre-trained Weights

- Pre-trained weights from large datasets (ImageNet and COCO)
- Initialize model (common backbones) on pre-trained weights

http://www.image-net.org/



https://cocodataset.org/#home

54

Again, one of the benefits of using a common architecture, such as AlexNet, VGGNet-16, or a ResNet, is that large datasets, such as ImageNet and COCO, have been used to train these models. The associated weight/parameter sets can then be used to initialize the model as opposed to initializing them randomly. This can allow for the model to be applied to new data with potentially less overfitting.

In the PyTorch modules, I will demonstrate how to implement transfer learning using pre-trained models and common architectures.

torchvision

- Model architectures
  - Examples: AlexNet, DenseNet, EfficientNetV2, MobileNetV2, MobileNetV3, ResNet, ResNeXt, VGG, etc.
- Pre-trained weights

https://pytorch.org/vision/stable/index.html
https://pytorch.org/vision/stable/models.html

55

The torchvision package provides access to several common architectures and pre-trained weights/parameters for these architectures. We will investigate this package in the PyTorch modules.

Number of Bands

- Using pre-trained weights may require a certain number of bands
- Can be an issue for multispectral data
- Possible to alter model architecture

56

Again, using pre-trained weights and existing architectures may require you to conform to the existing structure. For example, you may need to provide 3-band data since the network was originally trained on RGB data.

There are ways to overcome this issue, such as replacing the first convolution layer with a new one that accepts the correct number of inputs. It is also common to have to replace or alter the fully connected layer(s) when the number of differentiated classes is different from the original implementation.

This is the end of this lecture module.

Please return to the West Virginia View Webpage for additional content.

Thanks! Hope you found this useful.