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

Geospatial Deep Learning

Semantic Segmentation

WV VIEW

AmericaView
Empowering Earth Observation Education
AmericaView.org - Est. 2003

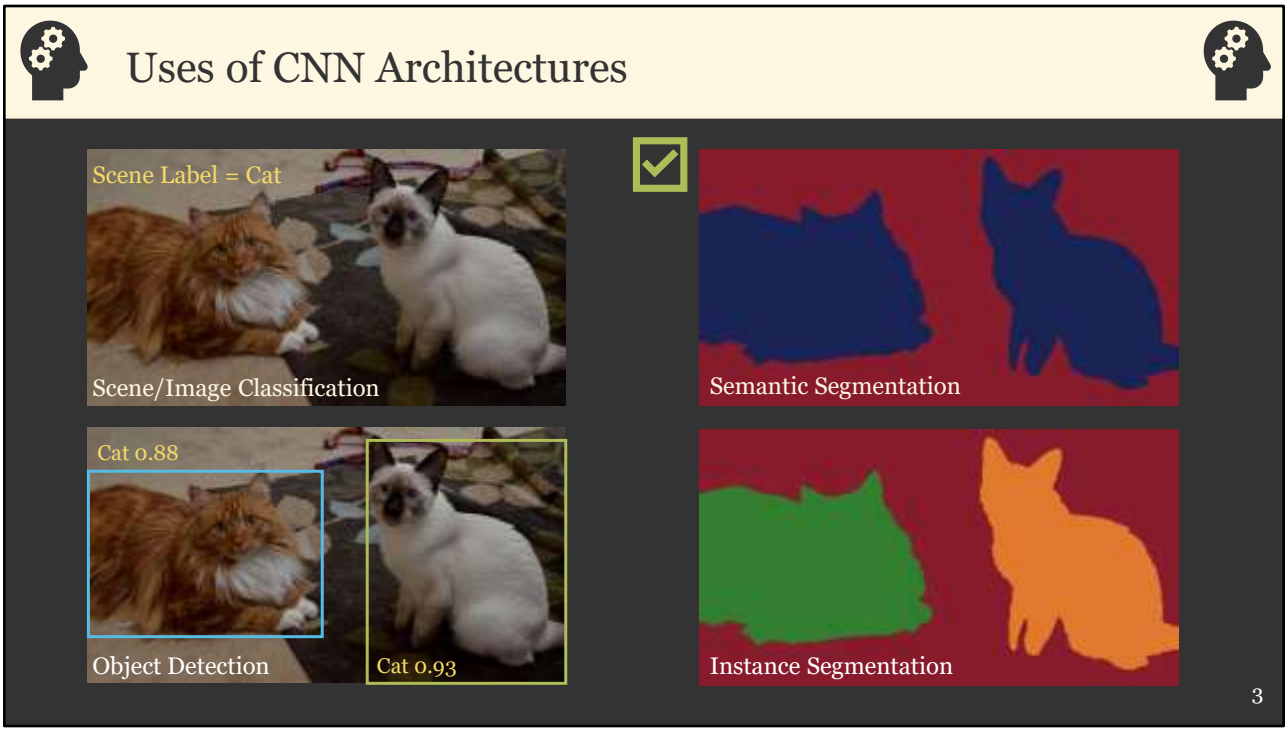
This module expands upon the last module by exploring CNN-based architectures that allow for pixel-level, as opposed to scene-level, classification. Within computer vision, pixel-level classification is commonly referred to as semantic segmentation.

Module Topics

1. Semantic segmentation/pixel-level classification
2. Fully convolutional neural networks
3. Upsampling and transpose convolution
4. UNet
5. UNet++
6. UNet3+
7. Atrous convolution, the ASPP module, and DeepLabv3+
8. HRNet
9. Encoder backbones
10. Data preparation and considerations
11. Combating overfitting

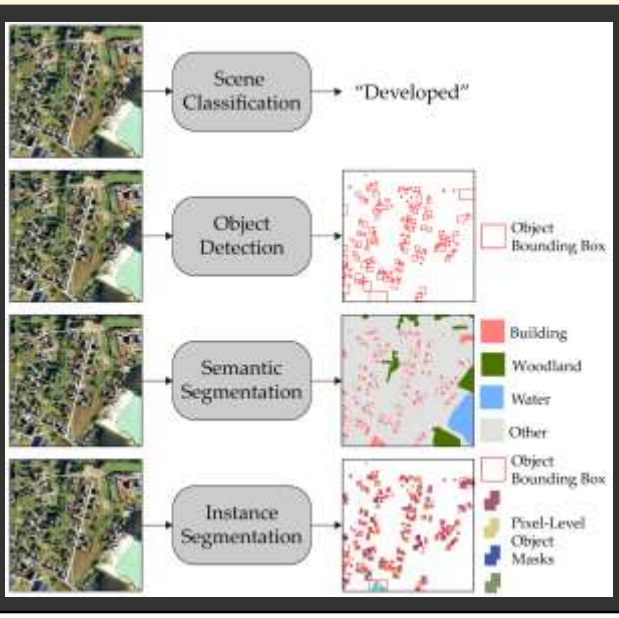
2

These are the topics covered in this module.



When conducing semantic segmentation, each pixel in an image is labeled to a category. In the example on this slide, the pixels are labelled to the “cat” or “not cat” class. Note that individual instances of a class are not differentiated. In other words, we are only predicting what pixels belong to the “cat” or “not cat” class. We are not differentiating pixels that belong to Freddy from pixels that belong to Peri.

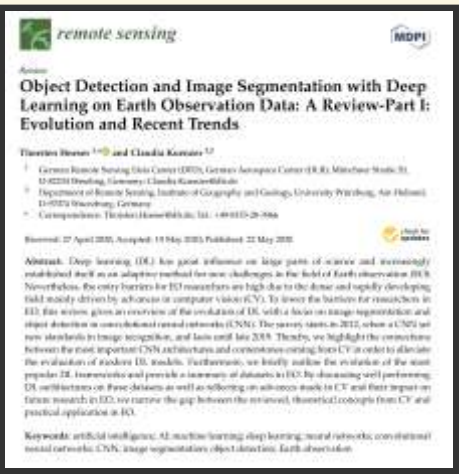
Uses of CNN Architectures



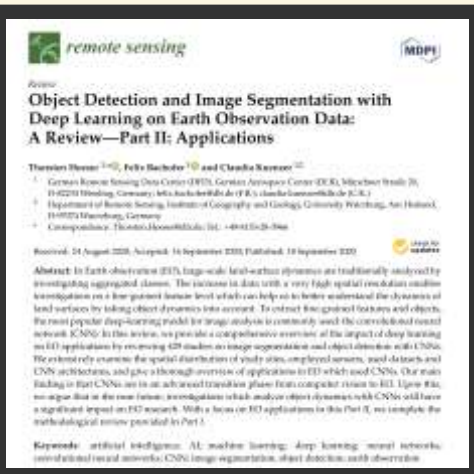
4

One common use of semantic segmentation in remote sensing is land cover classification where each pixel is assigned to one of a pre-defined set of categories. In this example, the following classes are differentiated: “building”, “woodland”, “water”, and “other”.

Hoeser et al.



Hoeser, T. and Kuenzer, C., 2020. Object detection and image segmentation with deep learning on Earth observation data: A review-part I: Evolution and recent trends. *Remote Sensing*, *12*(10), p.1667.



Hoeser, T., Bachofer, F. and Kuenzer, C., 2020. Object Detection and Image Segmentation with Deep Learning on Earth Observation Data: A Review—Part II: Applications. *Remote Sensing*, *12*(18), p.3053.

5

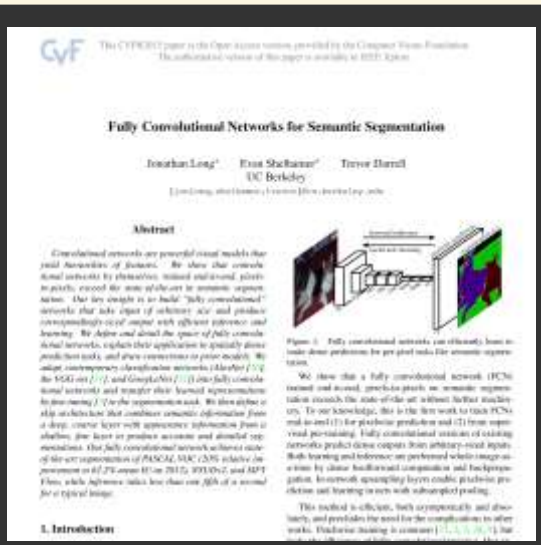
If you are interested in learning more about semantic segmentation, I recommend this series of articles by Hoeser et al. Both are open access and freely available.

Fully Convolutional Neural Networks

Fully Convolutional ANNs

- Series of convolutional layers, activation functions, and max pooling to learn features at different scales
- Some means of upsampling
- Sigmoid or softmax activation function to get probabilities
- No fully connected layers or flattening

Long, J., Shelhamer, E. and Darrell, T., 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431-3440).



Fully Convolutional Networks for Semantic Segmentation

Jonathan Long* Evan Shelhamer* Trevor Darrell
UC Berkeley

Abstract

1. Introduction

7

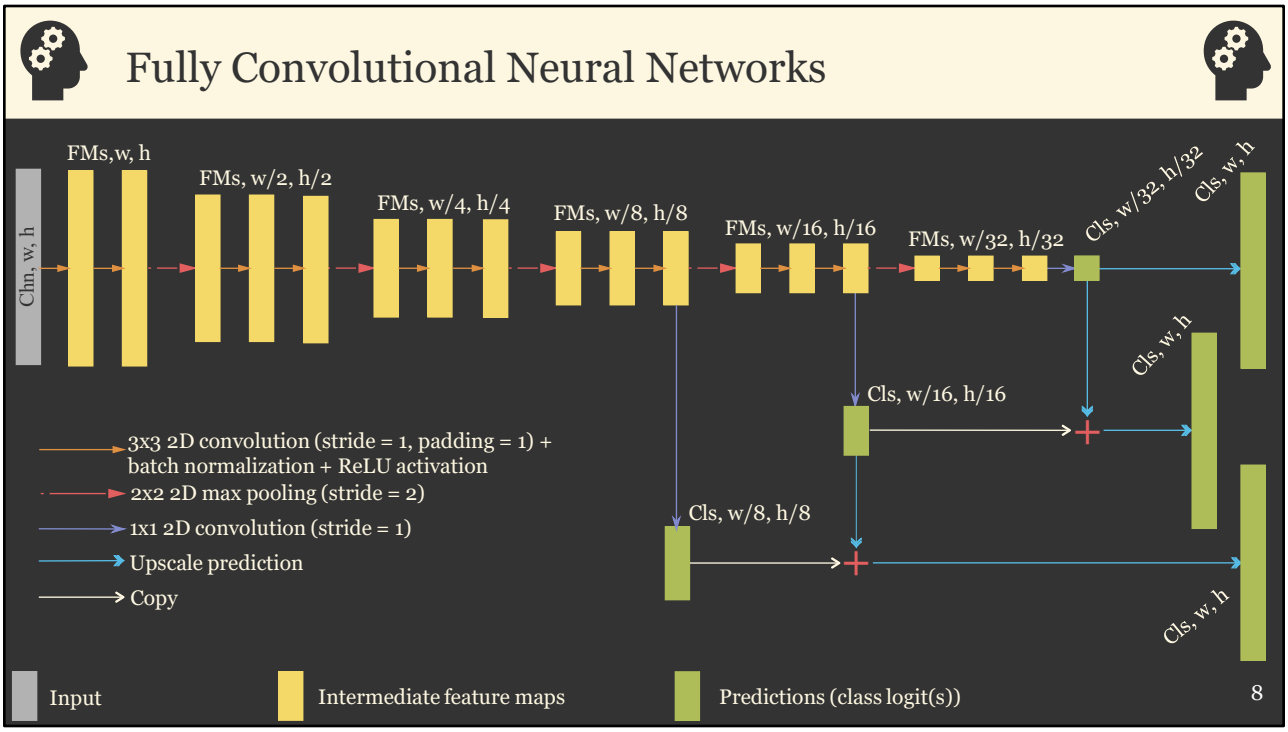
How can we obtain pixel-level classifications using convolutional neural networks? Generally, the first component of semantic segmentation architectures are very similar to the architectures of those designed for scene classification/labeling. Here, 2D convolution and max pooling are used to learn spatial filters at different scales. Also, the ReLU activation function is commonly used to add nonlinearity. It is also possible to incorporate batch normalization. In the case of scene classification, the final feature maps generated by the last convolutional layer are flattened to a 1D vector and fed to fully connected layers to predict the scene-level membership to a defined set of classes. However, this will not work for semantic segmentation.

For semantic segmentation when using fully convolutional neural networks, after spatial patterns are learned at multiple scales using the 2D convolution and max pooling operations, the resulting feature maps are not flattened to a 1D vector and no fully connected layers are used. Instead, the data are upsampled to generate pixel-level predictions. The first, downsampling component of the model that uses 2D convolution to learn patterns is often termed the encoder or backbone while the upsampling component is termed the decoder.

Fully convolutional neural networks commonly make use of either upsampling or transpose 2D convolution to perform the upsampling or decoding. We will

discuss these methods in later slides. Next, logits or estimated class probabilities are generated at each pixel using a sigmoid function, in the case of binary classification, or softmax, in the case of multiclass classification.

In order to make use of feature maps generated at different scales and from prior convolutional layers, not just the last convolutional layer, skip connections can be included. This allows for feature maps to be fed to a later process or layer as opposed to the next set of convolution and max pooling operations. For multiple feature maps from different layers to be combined or concatenated, they must have the same size in the spatial dimensions. This is why some means of upsampling is required.



Fully convolutional neural networks were first introduced in 2015 by Long et al. and are conceptualized on this slide. Arrows represent operations being performed while blocks represent input or output tensors. The gray block represents the input data, yellow blocks represent intermediate tensors of feature maps, and green blocks represent output predictions, or class logits at each pixel location, at varying spatial resolutions. Note that this is a generalization of the architecture proposed by Long et al.

Fully convolutional neural networks do not include any fully connected or densely connect layers. Instead, they make use of convolutional layers, pooling operations, activation functions, and upsampling or transpose convolution. Batch normalization can also be implemented throughout the architecture.

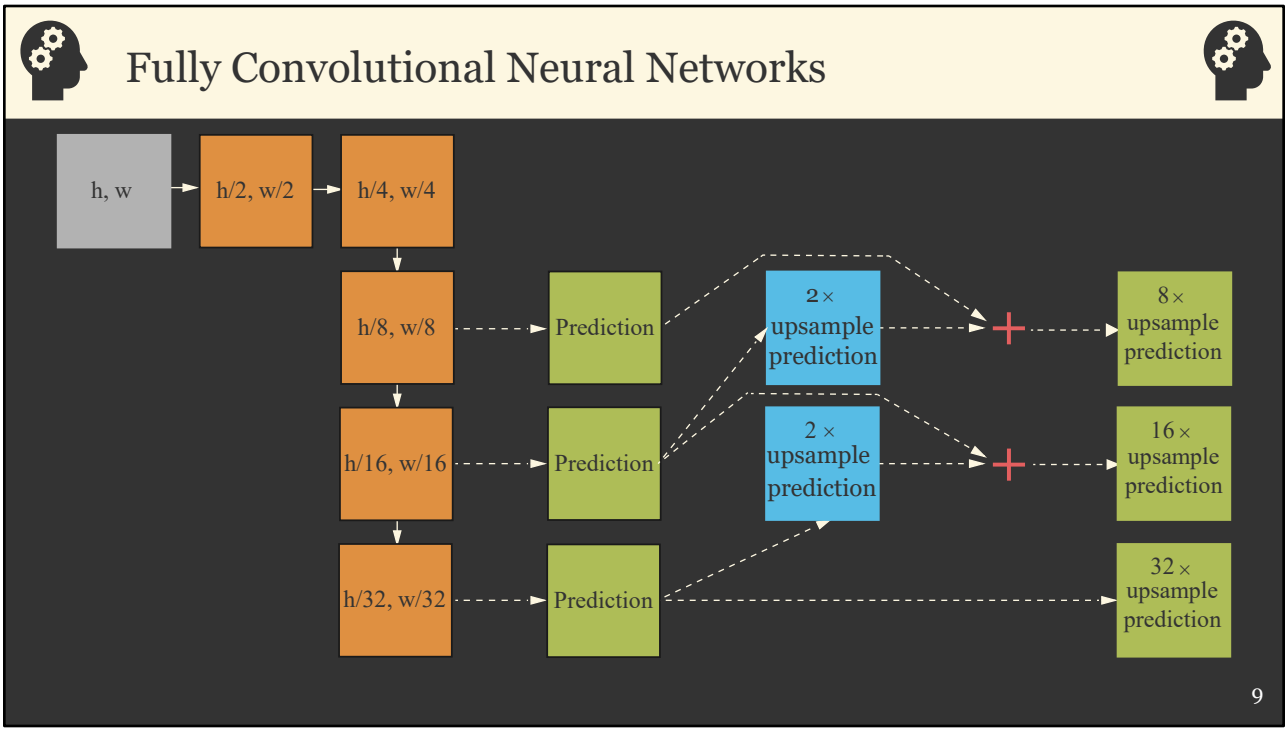
In the example architecture, two 3×3 2D convolution operations, conceptualized as orange arrows, are applied before the first max pooling operation, which is shown as a dashed red arrow. Next, a series of five sets of three 3×3 2D convolutional layers are applied with max pooling performed after all blocks except the final block. Using multiple convolutional layers between max pooling operations allow for a higher

degree of data abstraction than using only a single layer. This series of 3×3 2D convolutional layers, with associated batch normalization and ReLU activation functions, and max pooling operations is generally termed the encoder since the goal is to capture or encode useful information for the task at hand as feature maps at varying spatial scales.

Due to the application of max pooling operations in the architecture, the array size decreases in the spatial dimensions as the data pass through the encoder. In the example and after the last set of 2D convolution operations are applied, the original tensor spatial dimensions has decreased by a factor of 32 since five max pooling operations have been performed.

Instead of making only a single prediction at the end of the architecture, predictions at a reduced spatial resolution relative to the input data are made at multiple stages in the architecture using 1×1 2D convolutional layers with a stride and padding of 1. These operations are represented on the slide using purple arrows. 1×1 2D convolution can be used to change the number of feature maps as opposed to being used to learn spatial filters. For multiclass classification tasks and in order to make predictions at each pixel, the feature maps that are provided to the 1×1 2D convolution layer are reduced to a tensor of shape [number of classes, width, height].

In the pictured architecture, predictions are made at three reduced resolutions: the original resolution divided by 8, 16, and 32. In order to restore the spatial resolution of the arrays to that of the input data, the predictions must be upsampled. One issue with upsampling the data is that decreasing the size of the array in the spatial dimensions then subsequently upsampling the array results in fuzzy class boundaries or reduced semantic detail. To combat this issue, Long et al. proposed a mechanism to combine predictions made at multiple scales, as conceptualized on the slide. These predictions can then be merged to take into account the multiscale information. This process is discussed on the next slide.



This slide further elaborates on the process of combining predictions at multiple scales.

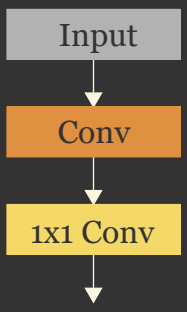
First, predictions are made using 1×1 2D convolution and the feature maps generated by the encoder at reduced spatial resolutions of 8, 16, and 32 relative to the spatial resolution of the input data. Upsampling methods are then used to increase the size of these arrays in the spatial dimensions by a factor of 8, 16, or 32 to obtain predictions at the original spatial resolution. In order to incorporate the semantic information from the later stages of the encoder for the 8- and 16-level predictions, predictions from the subsequent stage are upsampled by a factor of 2 to match the resolution of the stage being processed. The predictions are then added and collectively upsampled by the factor required to obtain predictions at the original spatial dimensions or resolution. For the 8-level predictions, this consists of upscaling the predictions made at the 16-level by a factor of 2, adding these to the 8-level predictions, then upscaling the result by a factor of 8. For the 16-level predictions, this consists of upscaling the predictions made at the 32-level by a factor of 2, adding these to the 16-level predictions, then upsampling the result by a factor of 16. Again, the goal is to make use of semantic information characterized at varying scales to improve the final prediction and to combat the loss of spatial detail introduced by the max pooling operations in the encoder.

In the final predictions, the class dimension has a length equal to the number of

classes being differentiated, and each band or channel holds the logit at each pixel location for a specific class. A softmax function can be applied to rescale the logits at each pixel location and occurring in separate bands to a scale of 0 to 1 with all estimated class predictions at the pixel location summing to 1. For a binary classification problem, a single logit can be returned for the positive class. Alternatively, both the background and positive class logits can be returned as separate channels. In such cases, the problem is treated as a multiclass problem. When a continuous variable is being predicted, only a single output channel is required.

1x1 2D Convolution

- Final Feature Maps → Class Logits
- Does not impact spatial dimensions of the array
- Model can accept images of varying sizes



10

Why use 1x1 2D convolution to obtain the final class logits? This is simply a means to change the number of channels without changing the spatial dimensions of the data or augmenting the pixel values using values in adjacent cells. In other words, a set of feature map values at a pixel location can been reduced to a set of class logits. These logits can then be passed through a sigmoid or softmax activation to obtain the estimated positive class probability or all estimated class probabilities, respectively.

Using 1x1 2D convolution as opposed fully connected layers has an added benefit: this allows the architecture to accept images of varying sizes in the spatial dimensions. Since no flattening is applied, there is no issues with varying length vectors being provided to fully connected layers.

So, fully convolutional architectures can be trained using and make inferences to images of varying sizes.

In short, 1x1 2D convolution is not really a convolution operation. Instead, it is a means to augment the number of input values at a pixel location to a different set and number of values. In the case of classification, the last set of activations at a pixel location, or the feature map values at that pixel location, are transformed to class logits.

1x1 2D convolution is effectively a fully connected layer applied to each pixel.

Upsampling

- Used for “deconvolution” in decoder
- Upsample feature maps
- Similar to resampling in GIS/remote sensing
- Different methods are available:
  - Nearest
  - Linear
  - Bilinear
  - Bicubic
  - Trilinear

11

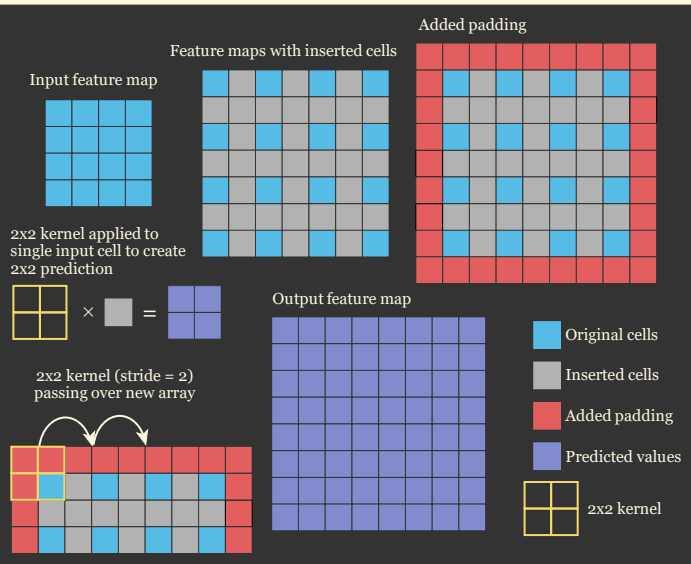
A variety of techniques are available to upsample images or feature maps. For example, resampling methods, such as nearest neighbor, bilinear interpolation, or cubic convolution, can be used. These are the same techniques commonly used in remote sensing and GIS operations to change the cell size of an image or raster grid using near cell values in the original array.

The earliest fully convolutional architectures used upsampling methods. Upsampling methods are not strictly “deconvolution” since they do not reverse the convolution process and do not use kernels with trainable weights. Upsampling methods, such as nearest neighbor, do not have trainable weights/parameters. They are simply mathematical interpolation/estimation operations used to estimate cell values from near cell values in the original array.

Transpose Convolution

- Used for “deconvolution” in decoder
- Upsample feature maps
- Have trainable weights/parameters, similar to traditional 2D convolutional layers

https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11



12

Another means to increase the size of tensors in the spatial dimensions is transpose convolution.

In contrast to upsampling methods, transpose convolution allows for the learning of weights/parameters to upsample the feature maps. Note that this is not strictly a “deconvolution” operation; in other works, it does not simply reverse the convolution process to obtain back the original data with the original spatial resolution, or number of rows and columns.

Similar to 2D convolution used in the encoder, transpose convolution allows for the learning of weights/parameters associated with kernels of a defined size. As a result, these layers are trainable. In order to upsample the array in the spatial dimensions, the array size is increased by adding zeros in between values or existing pixels and applying padding.

Kernels and associated weights/parameters are then learned to upsample the feature maps.

Please see the blog post linked on this slide for a detailed explanation of transpose convolution.

To restore the array size in the spatial dimensions following 2×2 2D max pooling

with a stride of 2, transpose convolution can be applied with a 2×2 kernel size and a stride of 2.

UNet and Derivatives

UNet


U-Net: Convolutional Networks for Biomedical Image Segmentation

Ronneberger, O., Fischer, P. and Brox, T., 2015, October. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention* (pp. 234-241). Springer, Cham.

https://arxiv.org/abs/1505.04597

The UNet method was first introduced in 2015 and builds upon fully convolutional architectures. This slide provides a reference to the paper that introduced UNets. Although originally developed for medical image segmentation, it has been shown to have many applications.

UNet Components

Encoder or Downsampling or Contraction

- Repeating blocks
- 2D Convolution, 3×3, stride of 1
- Activation (ReLU)
- Batch Normalization
- Max Pooling, 2×2, stride of 2

Decoder or Upsampling or Expansion

- Repeating blocks
- Upsample with transpose convolution, 2×2, stride of 2
- Skip connections between encoder and decoder blocks
- 2D Convolution, 3×3, stride of 1
- Classification Head = 2D convolution, 1×1, stride of 1

15

This slide outlines the key components of UNets. There are different flavors of UNets, and the example provided here does not follow the method outlined in the original paper perfectly.

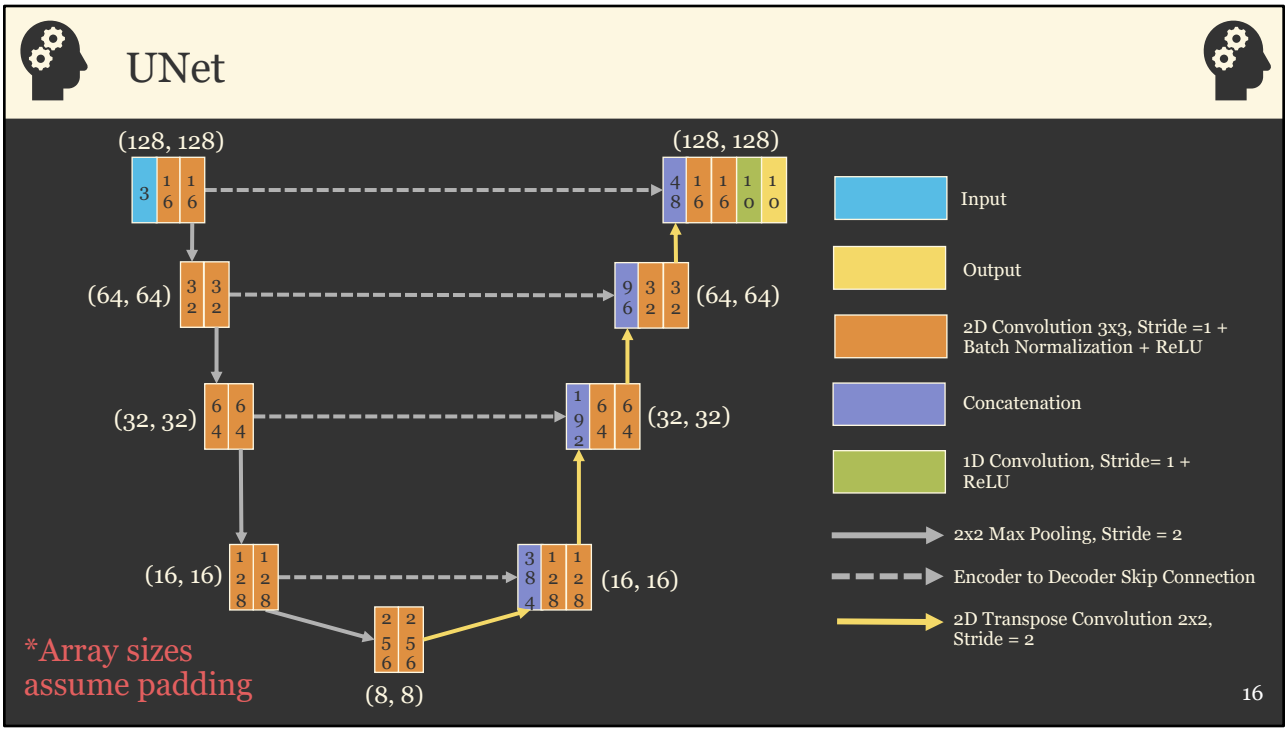
The encoder, downsampling, or contracting component is a traditional CNN architecture, similar to those we discussed in the last module. 3x3 2D convolutional layers with a stride of 1 and 2x2 max pooling with a stride of 2 are used to produce feature maps at different scales. It is common to use ReLU as the activation function for the convolutional layers; however, other activation functions can also be used, such as leaky ReLU. Batch normalization can also be incorporated. Padding is used to maintain the array size and edge pixels. It is also possible to include dropouts, but that is not always applied.

The decoder, upsampling, or expansion component works like a traditional CNN, but in reverse. 2x2 transpose convolution with a stride of 2, or sometimes upsampling methods, are used to increase the size of the array in the spatial dimensions. For a decoder block, skip connections are used to concatenate the feature maps from the encoder component with the same spatial resolution to the outputs produced by the prior decoder block following the application of transpose convolution. This stack of feature maps is then provided to the current decoder block. This allows for incorporating the information learned in the encoder into the later stages of the learning process

occurring in the decoder. In other words, semantic information is shared between the encoder and decoder blocks having the same spatial resolutions. Additional 3x3 2D convolution operations with a stride of 1 are used to learn new filters in the decoder blocks. The 2D convolution is generally followed by an activation function, such as ReLU, and sometimes batch normalization.

Finally, 2D convolution with a 1x1 filter and a stride of 1 is used to perform the classification at the pixel- or cell-level to obtain logits. To obtain the estimated positive class probability for a binary classification where only the positive case is predicted, the sigmoid function is used. For a multiclass classification, or a binary classification where logits for both the positive and background classes are predicted, a softmax function is used.

UNet has a symmetrical architecture: the number of encoder blocks is equal to the number of decoder blocks. This allows for skip connections were feature maps from the associated encoder block are concatenated with the outputs from the prior decoder block, following the application of transpose convolution, prior to being provided to the current decoder block to maintain more spatial detail and localization and share semantic information between the encoder and decoder. Since the feature maps being concatenated via the skip connections have the same size in the spatial dimensions as the tensors they are being concatenated with, there is no need to incorporate upsampling techniques into the skip connections. This is in contrast to the fully convolutional architecture already discussed.



This slide further describes the structure of a UNet.

The encoder consists of a series of blocks. Each block includes a pair of 2D convolution layers, each using a kernel size of 3x3 and a stride of 1. In the example, each 2D convolution operation is accompanied by batch normalization and a ReLU activation function. In the encoder, the size of the array in the spatial dimensions is reduced between each block using max pooling with a kernel size of 2x2 and a stride of 2. There are a total of 4 decoder blocks in the example.

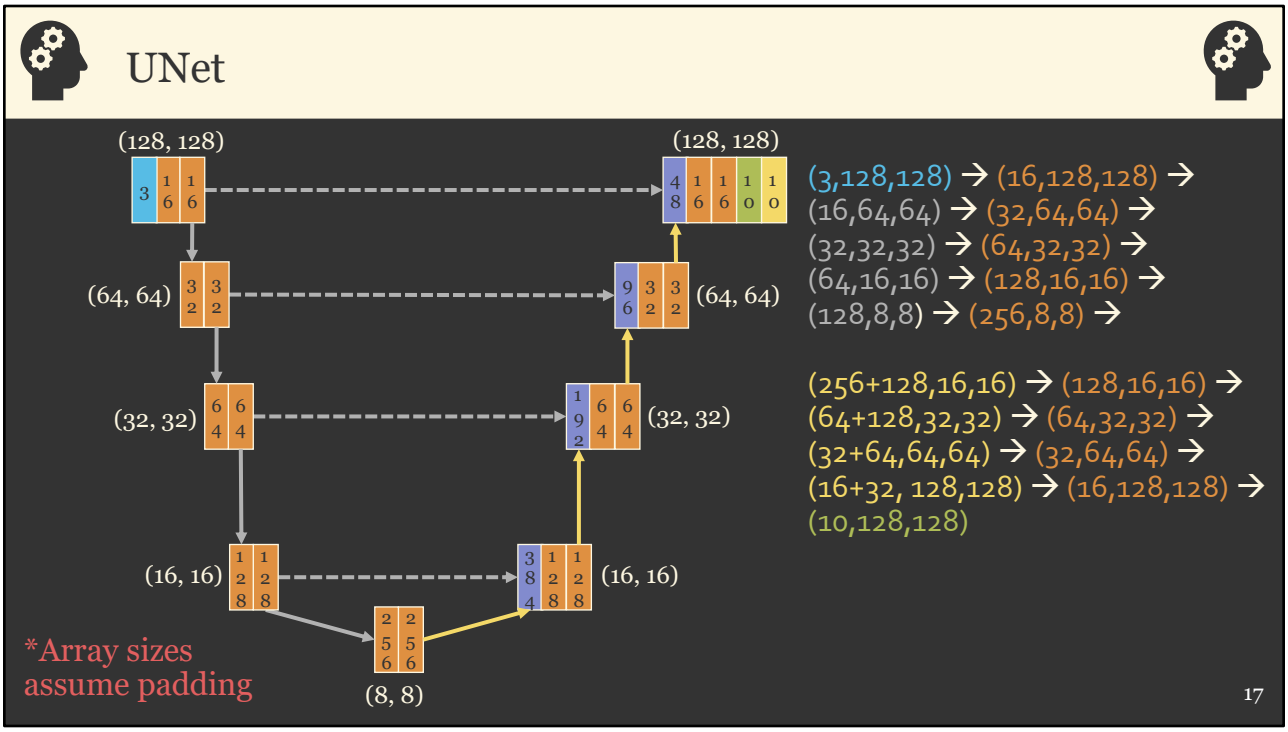
The block between the encoder and decoder is generally termed the bottleneck. It also consists of double-2D convolution operations. The array will be at its smallest size in the spatial dimensions at the bottleneck stage.

To undo the reduction in array size in the spatial dimensions resulting from the max pooling operations in the encoder, the decoder uses 2D transpose convolution with a kernel size of 2x2 and a stride of 2. This results in each encoder block and associated decoder block accepting tensors with the same sizes in the spatial dimensions.

In order to make full use of the spatial context information learned in the encoder blocks, the learned feature maps from the encoder are concatenated

with the upsampled feature maps form the prior decoder block using skip connections. Since the arrays already have the same sizes in the spatial dimensions, there is no need to apply an upsampling method in the skip connections.

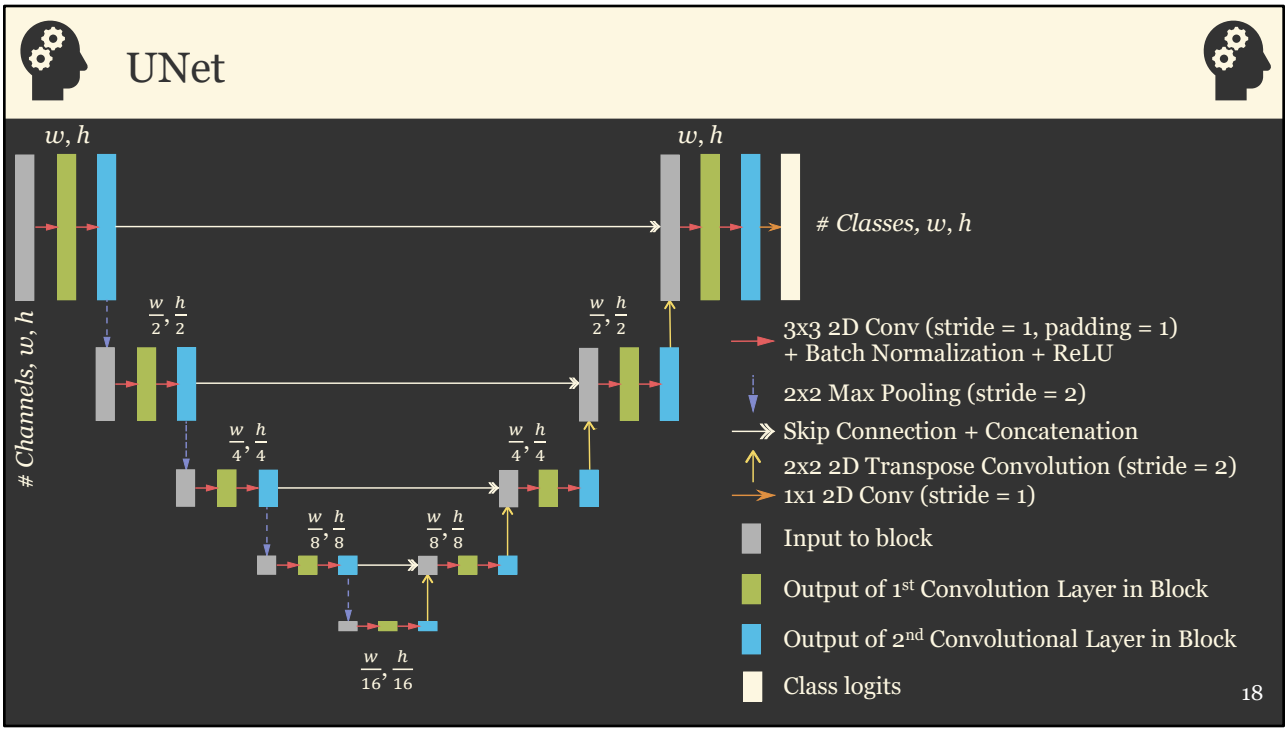
Similar to the fully convolutional architecture already discussed, the final logits are obtained using 1x1 2D convolution with a stride of 1.



This slide highlights the sizes of the tensors at each operation or block in the UNet architecture. Blue indicates the input image shape while green represents the output shape. In this case a 3-band input is provided, and 10 class logits are returned.

Orange indicates the shapes associated with the 2D convolution operations in the encoder and decoder blocks while gray arrows map to max pooling operations and yellow arrows maps to 2D transpose convolution.

Make sure you understand why these array sizes are obtained throughout the architecture.



Here is a final representation of a UNet architecture. Again, note that the entire architecture is constructed from a small set of layers or operations: 3x3 2D convolution with a stride of 1, batch normalization, ReLU activation functions, 2x2 2D max pooling with a stride of 2, 2x2 2D transpose convolution with a stride of 2, and 1x1 convolution with a stride of 2.

Between each stage of the encoder, the spatial resolution of the tensor is decreased by a factor of 2 using max pooling. For example, if the input data had spatial dimensions of 128x128 cells the array sizes would change as follows through the encoder: 128x128 → 64x64→ 32x32 → 16x16 → 8x8. The resolutions would then be increased through the decoder: 8x8 → 16x16→ 32x32 → 64x64 → 128x128.

The final classification is performed using 1x1 2D convolution with a stride of 1. This results in the final set of feature map values at a cell location being transformed into predicted class logit(s).

UNet Trainable Parameters Example

Block	Layer	Output shape	CNN	BN
Encoder Block 1	3×3 2D Conv + BN + ReLU	[B, 16, 256, 256]	448	32
	3×3 2D Conv + BN + ReLU	[B, 16, 256, 256]	2,320	32
	2×2 2D Max Pooling	[B, 16, 128, 128]	0	0
Encoder Block 2	3×3 2D Conv + BN + ReLU	[B, 32, 128, 128]	4,640	64
	3×3 2D Conv + BN + ReLU	[B, 32, 128, 128]	9,248	64
	2×2 2D Max Pooling	[B, 32, 64, 64]	0	0
Encoder Block 3	3×3 2D Conv + BN + ReLU	[B, 64, 64, 64]	18,496	128
	3×3 2D Conv + BN + ReLU	[B, 64, 64, 64]	36,928	128
	2×2 2D Max Pooling	[B, 64, 32, 32]	0	0
Encoder Block 4	3×3 2D Conv + BN + ReLU	[B, 128, 32, 32]	73,856	256
	3×3 2D Conv + BN + ReLU	[B, 128, 32, 32]	147,584	0
Bottleneck	2×2 2D Max Pooling	[B, 128, 16, 16]	0	0
	3×3 2D Conv + BN + ReLU	[B, 256, 16, 16]	295,168	512
	3×3 2D Conv + BN + ReLU	[B, 256, 16, 16]	590,080	512
Decoder Block 1	2×2 2D Transpose + BN + ReLU	[B, 256, 32, 32]	262,400	512
	3×3 2D Conv + BN + ReLU	[B, 128, 32, 32]	442,496	256
	3×3 2D Conv + BN + ReLU	[B, 128, 32, 32]	147,584	256
Decoder Block 2	2×2 2D Transpose + BN + ReLU	[B, 128, 64, 64]	65,664	256
	3×3 2D Conv + BN + ReLU	[B, 64, 64, 64]	110,656	128
	3×3 2D Conv + BN + ReLU	[B, 64, 64, 64]	36,928	128
Decoder Block 3	2×2 2D Transpose + BN + ReLU	[B, 64, 128, 128]	16,448	128
	3×3 2D Conv + BN + ReLU	[B, 32, 128, 128]	27,680	64
	3×3 2D Conv + BN + ReLU	[B, 32, 128, 128]	9,248	64
Decoder Block 4	2×2 2D Transpose + BN + ReLU	[B, 32, 256, 256]	4,128	64
	3×3 2D Conv + BN + ReLU	[B, 16, 256, 256]	6,928	32
	3×3 2D Conv + BN + ReLU	[B, 16, 256, 256]	2,320	32
Classification Head	1×1 2D Convolution	[B, 10, 256, 256]	170	0
		Totals	**2,315,066**	**3,648**
			Overall Total	**2,318,714**

19

The table on this slide outlines the number of trainable parameters within a UNet architecture. Remember that the ReLU activation function and max pooling operations do not have trainable parameters. The 3x3 2D convolution, batch normalization, 2x2 2D transpose convolution, and 1x1 convolution do have trainable parameters.

For a 3x3 2D convolutional layer, the number of learned weights is equal to the number of input channels or feature maps × the number of output feature maps × the number of cells in the kernel (in this case 9). There is also a trainable bias term for each learned kernel. For the first 3x3 2D convolutional layer, this results in a total of 448 trainable parameters: (3 x 16 x 9) + 16 = 448.

The number of trainable parameters for batch normalization is 2x the number of input feature maps since gain and shift parameters are learned for each feature map.

For 2x2 2D transpose convolution, the number of trainable parameters is equal to the number of input feature maps × the number of output feature maps × the number of cells in the kernel (in this case 4). There is also a bias term for each learned kernel. For the 2x2 2D transpose convolutional layer in this first decoder block, there would be a total of 262,400 trainable parameters: (256 x 256 x 4) + 256 = 262,400.

The number of trainable parameters in the classification head, or final 1x1 2D convolutional layer, is the number of input feature maps × the number of output logits plus a bias term for each output. In the example, there are 170 trainable parameters for this layer: (16 x 10) + 10 = 170.

In this architecture, there are more than 2 million trainable parameters. Note that not including fully connected layers, which can have a large number of trainable parameters due to the large number of interconnected neurons, has the added benefit of reducing the number of parameters that must be learned.

UNet Implementations

- Segmentation Models
  - https://github.com/qubvel/segmentation_models
  - https://github.com/qubvel/segmentation_models.pytorch
- ArcGIS Pro
  - https://developers.arcgis.com/python/api-reference/arcgis.learn.toc.html

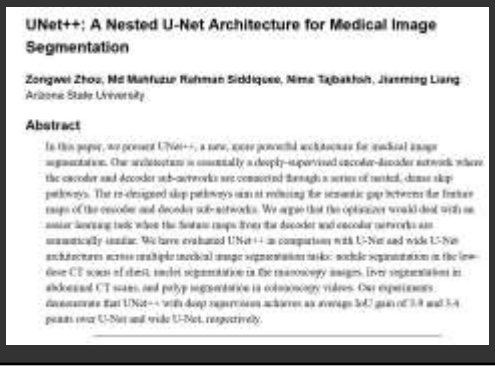
20

Currently, there are implementations of UNet available using PyTorch, Tensorflow/Keras, and ArcGIS Pro. The PyTorch modules will show how to build a UNet architecture from scratch by subclassing nn.Module. We will also explore the Segmentation Models library, which simplifies the implementation of semantic segmentation methods using PyTorch.

Note also that UNet is a flexible framework, so a variety of different architectures, augmentations, and/or backbones can be used. We will use UNet-like architectures in some of the examples in the improving models module.

UNet++

Zhou, Z., Siddiquee, M.M.R., Tajbakhsh, N. and Liang, J., 2018. Unet++: A nested u-net architecture for medical image segmentation. In *Deep learning in medical image analysis and multimodal learning for clinical decision support* (pp. 3-11). Springer, Cham.



UNet++: A Nested U-Net Architecture for Medical Image Segmentation

Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, Jianming Liang
Arizona State University

Abstract

In this paper, we present UNet++, a new, more powerful architecture for medical image segmentation. Our architecture is essentially a deeply-supervised encoder-decoder network where the encoder and decoder sub-networks are connected through a series of nested, dense skip pathways. The re-designed skip pathways aim to reducing the semantic gap between the feature maps of the encoder and decoder sub-networks. We argue that the optimizer would deal with an easier learning task when the feature maps from the decoder and encoder networks are semantically similar. We have evaluated UNet++ in comparison with U-Net and wide U-Net architectures across multiple medical image segmentation tasks: nodule segmentation in the low-dose CT scans of chest, nuclei segmentation in the microscopy images, liver segmentation in abdominal CT scans, and polyp segmentation in colonoscopy videos. Our experiments demonstrate that UNet++ with deep supervision achieves an average IoU gain of 3.9 and 3.4 points over U-Net and wide U-Net, respectively.

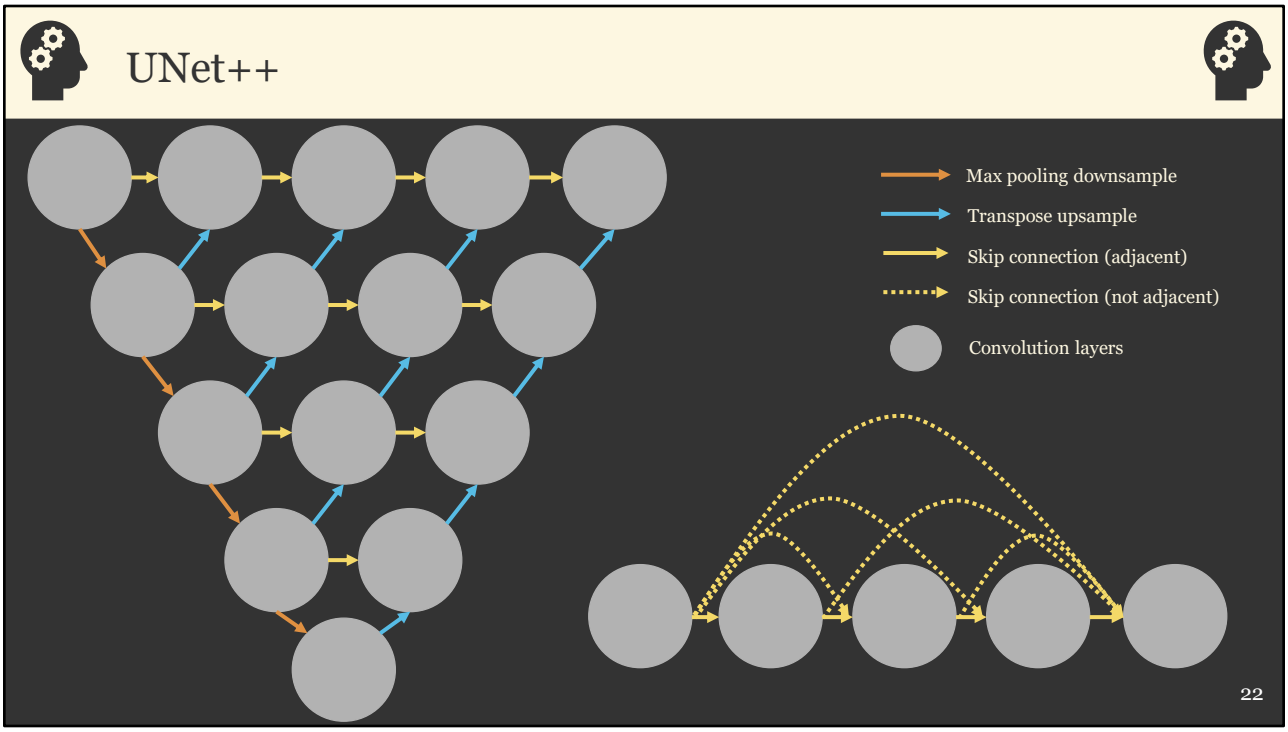
- Convolution layers between encoder and decoder
- Redesigned skip paths
- Dense skip connections
- Deep supervision

21

Since the introduction of UNet, many researchers/developers have proposed alterations and expansions of the originally proposed architecture. For example, UNet++ was proposed in 2018.

This structure is designed to improve the localization of UNet for mapping fine features. Instead of using simple skip connections between the encoder and decoder paths, additional convolutional operations are performed.

This is just one example of an augmentation or expansion of the original UNet architecture.

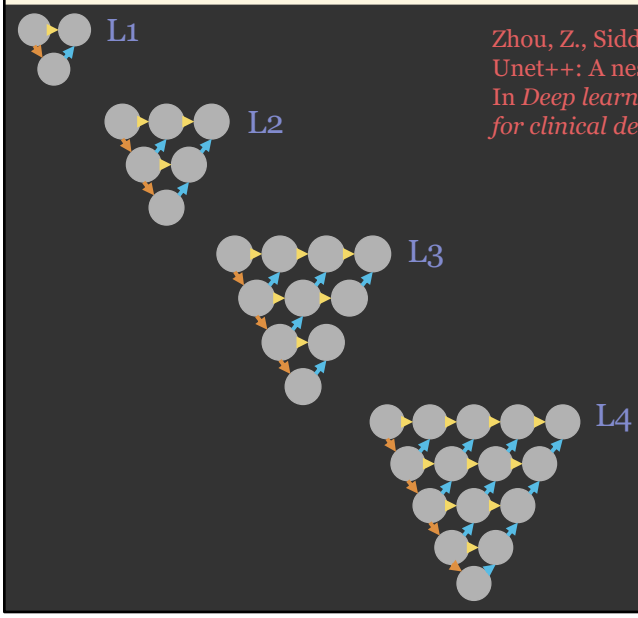


The graphics presented here attempt to characterize the key components of UNet++. The gray circles represent 2D convolutional operations or blocks. The outer operations represent the base UNet architecture with a series of convolution blocks in the encoder and decoder. The array sizes in the encoder are decreased using max pooling with a kernel size of 2x2 and a stride of 2 while array sizes in the decoder are increased using transpose convolution with a kernel size of 2x2 and a stride of 2.

UNet++ augments the base architecture of UNet by incorporating additional 2D convolution blocks along each skip connection path. Also, additional 2D transpose convolution operations are performed between these paths. Along each path, skip connections are included to bypass intermediate 2D convolution blocks. Specifically, the feature maps from the first 2D convolution block are concatenated before each subsequent 2D convolution block along the path, and all results from each prior 2D convolution block are bypassed to the final 2D convolution block.

The idea here is to reduce the semantic gap between the encoder and decoder with additional 2D convolution blocks between the encoder and decoder and with the skip connections and a densely connected structure.

Deep Supervision in UNet++



Zhou, Z., Siddiquee, M.M.R., Tajbakhsh, N. and Liang, J., 2018. Unet++: A nested u-net architecture for medical image segmentation. In *Deep learning in medical image analysis and multimodal learning for clinical decision support* (pp. 3-11). Springer, Cham.

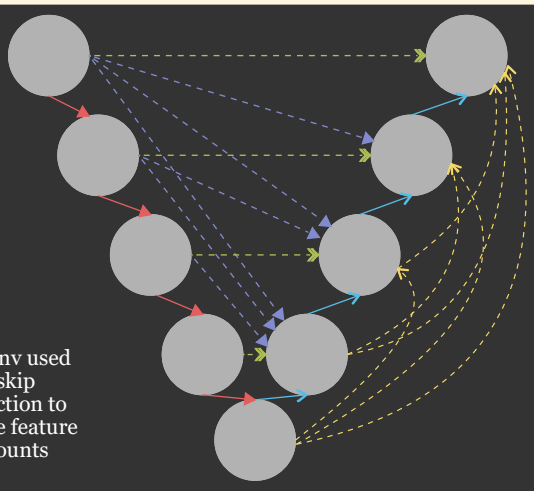
- Accurate = Average across all branches
- Fast = Select one model from all branches
- Allows for model pruning

23

Another component of UNet++ is the ability to implement deep supervision. The UNet++ architecture can be thought of as a nested structure of UNets of varying sizes or number of encoder and decoder blocks. Each of these nested UNets can be combined with a final 1x1 2D convolution layer to obtain separate predictions. The predicted logits can be averaged across these nested levels to potentially improve the performance. Alternatively, one of the nested UNets, as opposed to the largest UNet, could be used to make the final prediction, which could increase the speed of the computation and inference. This is essentially a means to prune the structure.

Using multiple predictions within this nested structure allows for deep supervision. However, it is not necessary to implement deep supervision when using UNet++. We will discuss deep supervision in more detail and more generally in the improving models module.

UNet3+



1x1 Conv used along skip connection to change feature map counts

Convolution block

Downsample/max pooling

Upsample

Skip connection

Skip connection with downsampling/max pooling

Skip connection with upsampling

Huang, H., Lin, L., Tong, R., Hu, H., Zhang, Q., Iwamoto, Y., Han, X., Chen, Y.W. and Wu, J., 2020. UNet 3+: A Full-Scale Connected UNet for Medical Image Segmentation. ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Barcelona, 4-8 May 2020, 1055-1059.

24

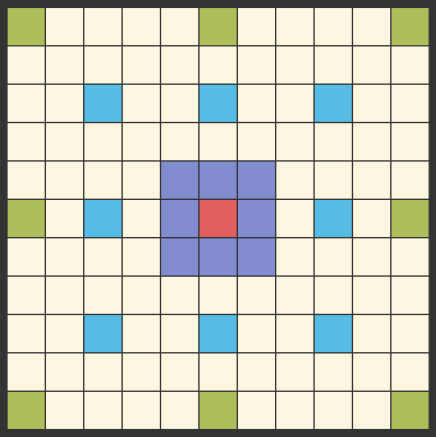
UNet3+ expands UNet by incorporating a large number of skip connections between different blocks of the architecture, further decreasing the semantic gap and providing skip connections to enhance gradient flow.

Due to the varying sizes of the tensors at different stages of the architecture different mechanisms are used to upsample or downsample tensors as necessary so that arrays can be concatenated.

DeepLabv3+

Convolution with Dilation

- Atrous convolution
- Add zeros in kernel to expand kernel size
- Increases the size of the receptive field
- Learn spatial patterns at varying scales
- Dilation Rate = 1
- Dilation Rate = 3
- Dilation Rate = 5



26

In order to increase the effective field of view or receptive field of convolutional operations and in order to learn spatial context information over larger extents, some semantic segmentation architectures make use of dilation or atrous convolution. The idea is to insert zeros into the kernel filters to increase the distance between cells included in the operation. This is generally controlled using a dilation rate.

In the example image, the red cell represents the cell that is being processed, which is included in all the kernel windows. The purple cells are included when using a dilation rate of 1. This is equivalent to a normal 3x3 kernel. The blue cells are included when the dilation rate is 3, and the green cells are included when the dilation rate is 5. In all cases, the number of cells with trainable weights is 9. The difference is in the spacing between the included cells, which again is controlled by the dilation rate. The dilation rate determines the number of inserted zeros.

It is also possible to use dilated convolution with more then 9 trainable weights in the kernel. In such configurations, the dilation rate within the kernel controls how many zeros are inserted between cells that have trainable parameters.

In, summary, the key idea behind atrous convolution is to change the effective

field of view, or receptive field, by incorporating offsets in the moving window, which is controlled by a dilation rate. So, windows can be made up of pixels that are not adjacent, allowing for a larger effective field of view.

DeepLabv3+

1. Use backbone network with dilation to generate feature maps
2. Atrous Spatial Pyramid Pooling to obtain multi-scale spatial context information. Consists of:
   - 1x1 convolution
   - 3x3 convolution with different dilation/atrous rates
   - Image pooling for global context
3. 1x1 convolution and 3x3 convolution to learn additional filters
4. Upsampling to resize feature maps
5. Concatenation to stack feature maps


Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation

Chen, L.C., Zhu, Y., Papandreou, G., Schroff, F. and Adam, H., 2018. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 801-818).

https://towardsdatascience.com/review-deeplabv3-atrous-convolution-semantic-segmentation-6d818bfd1d74

https://developers.arcgis.com/python/guide/how-deeplabv3-works/

https://github.com/tensorflow/models/tree/master/research/deeplab

27

As an example of an architecture that incorporates atrous convolution, we will explore the DeepLabv3+ architecture, which originated at Google and was subsequently open-sourced. This algorithm is one of a series of architectures in the DeepLab family.

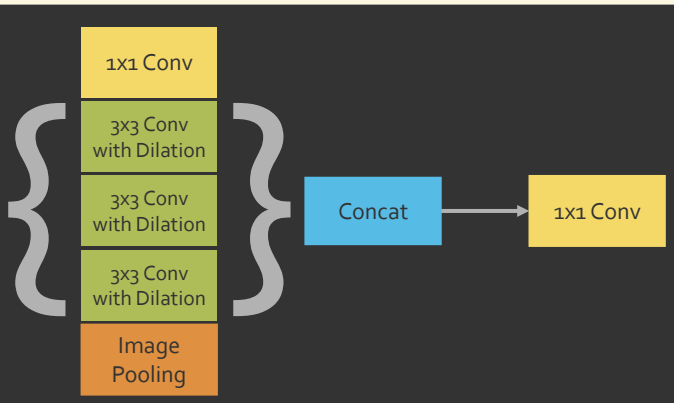
Similar to UNet, DeepLabv3+ relies on an encoder/decoder architecture. However, it incorporates atrous convolution and spatial pyramid pooling. The key components of the model are described on this slide.

First, an encoder or backbone is used to extract feature maps at vary spatial resolutions. Some of the kernels in this architecture use atrous convolution. There is also a second atrous spatial pyramid pooling (ASPP) component that is designed to further capture spatial context information at multiple scales. We will discuss this component on the next slide. 1x1 and 3x3 convolution are used to learn additional filters, and upsampling and concatenation are used to standardize the sizes of feature maps in the spatial dimensions and merge them into a single tensor, respectively.

Please have a look at the links provided for additional details. The paper that introduced the method is also referenced here.

Atrous Spatial Pyramid Pooling

- ASPP
- Allows for multi-scale context information to be combined/learned without significantly increasing the architecture size and number of parameters



Chen, L.C., Zhu, Y., Papandreou, G., Schroff, F. and Adam, H., 2018. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 801-818).

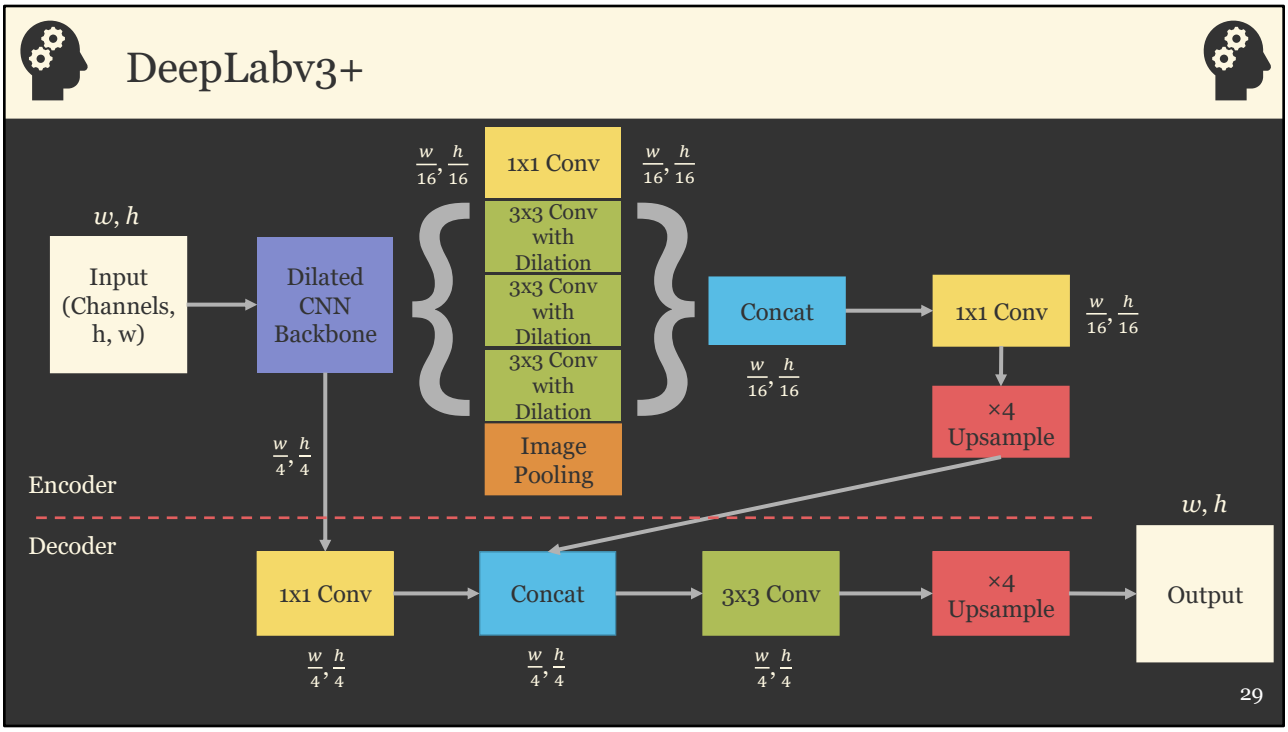
28

The atrous spatial pyramid pooling (ASPP) module allows for the aggregation of spatial context information across scales and also the incorporation of global information from the entire input tensor.

Feature maps from prior layers are passed through a series of operations including 1x1 2D convolution, multiple 3x3 2D convolutions with different dilation rates, and image pooling to incorporate global context information.

The learned feature maps are then concatenated and passed to a 1x1 2D convolution layer to further abstract the data and augment the number of feature maps.

Again, the goal here is to learn spatial context information at different scales and increase the size of the receptive field.



This slide conceptualizes the components of DeepLabv3+.

The input data are passed through a backbone encoder network. The feature maps learned in the backbone are then passed to the ASPP module. The learned feature maps from the ASPP module are then concatenated and passed through a 1x1 2D convolution layer to reduce the number of feature maps. Lastly, upsampling by a factor of 4 is applied so that the feature maps have the same size in the spatial dimensions as the feature maps to which they will be concatenated in the decoder component of the model.

In the decoder component of the model, the feature maps from the encoder backbone are passed through a 1x1 2D convolution layer. The feature maps from the ASPP module that have been upsampled are then merged with these feature maps via concatenation. The layers then pass through a 3x3 2D convolutional layer and are then upsampled by a factor of 4 to obtain the pixel-level prediction at the spatial resolution of the input data.

This architecture has the same purpose as the UNet architecture: to make predictions at the pixel-level. However, the architecture is different. As described here, one of the key differences is the use of atrous convolution and the ASPP module.

It should be noted that atrous convolution can be integrated into other architectures, even UNets. This is not a technique specific to the DeepLab family. However, it is a key component of the DeepLab algorithms.

Take some time to consider that tensor sizes outlined on this slide. Make sure you understand how the array sizes are obtained.

Note that some implementations of DeepLabv3+ may use slightly different designs and/or feature map sizes in the spatial dimensions throughout the architecture.

HRNet

HRNet

Wang, J., Sun, K., Cheng, T., Jiang, B., Deng, C., Zhao, Y., Liu, D., Mu, Y., Tan, M., Wang, X. and Liu, W., 2020. Deep high-resolution representation learning for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, *43*(10), pp.3349-3364.

IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, MARCH 2020 1

Deep High-Resolution Representation Learning for Visual Recognition

Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao

Abstract—High-resolution representations are essential for position-sensitive vision problems, such as human pose estimation, semantic segmentation, and object detection. Existing state-of-the-art frameworks first encode the input image as a low-resolution representation through a subnetwork that is formed by connecting high-to-low resolution convolutions in series (e.g., ResNet, VGGNet), and then recover the high-resolution representation from the encoded low-resolution representation. Instead, our proposed network, named as High-Resolution Network (HRNet), maintains high-resolution representations through the whole process. There are two key characteristics: (i) Connect the high-to-low resolution convolution streams *in parallel*; (ii) Repeatedly exchange the information across resolutions. The benefit is that the resulting representation is semantically richer and spatially more precise. We show the superiority of the proposed HRNet in a wide range of applications, including human pose estimation, semantic segmentation, and object detection, suggesting that the HRNet is a stronger backbone for computer vision problems. All the codes are available at https://github.com/HRNet.

Index Terms—HRNet, high-resolution representations, low-resolution representations, human pose estimation, semantic segmentation, object detection.

31

We will now discuss one final semantic segmentation architecture: HRNet. The paper that introduced this architecture is provided on this page.

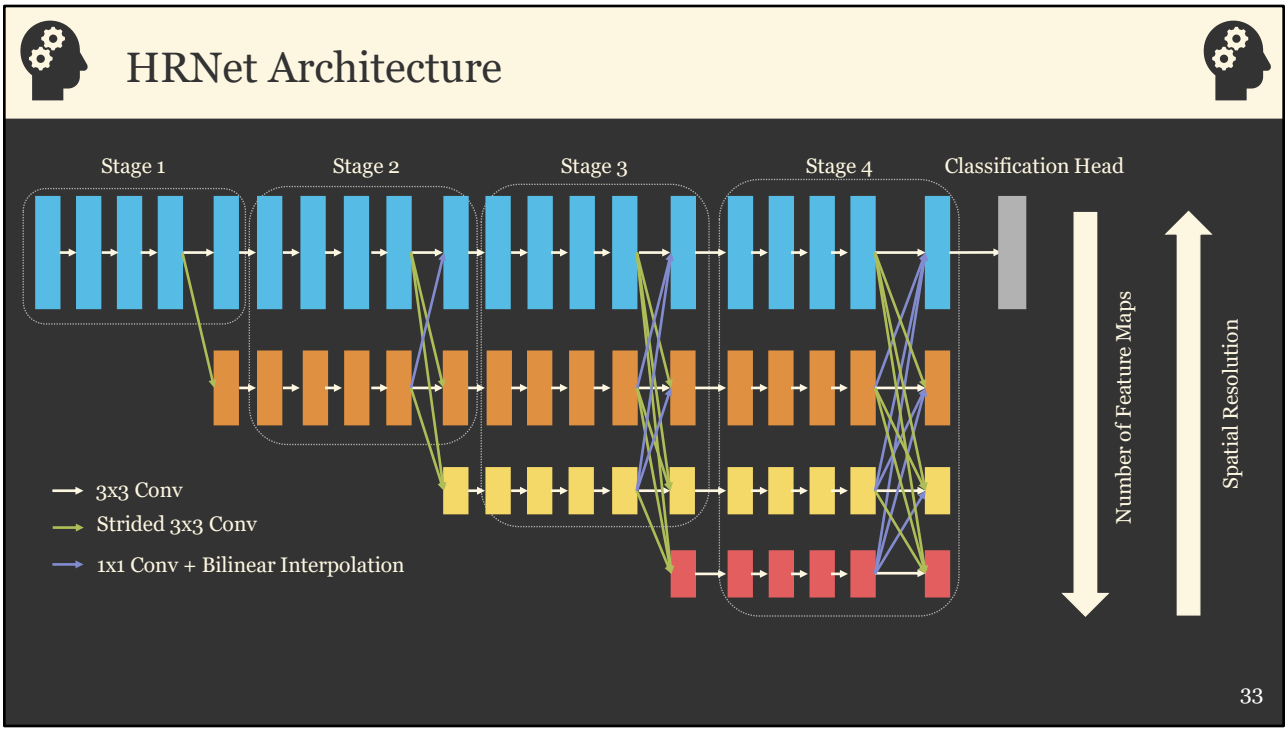
Key Characteristics

- Maintain high spatial resolution information throughout entire architecture
- Maintains parallel branches at different spatial resolutions
- Information shared between scales
- Separated into stages
- Add new, lower resolution path at each stage
- High memory usage

Wang, J., Sun, K., Cheng, T., Jiang, B., Deng, C., Zhao, Y., Liu, D., Mu, Y., Tan, M., Wang, X. and Liu, W., 2020. Deep high-resolution representation learning for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, *43*(10), pp.3349-3364.

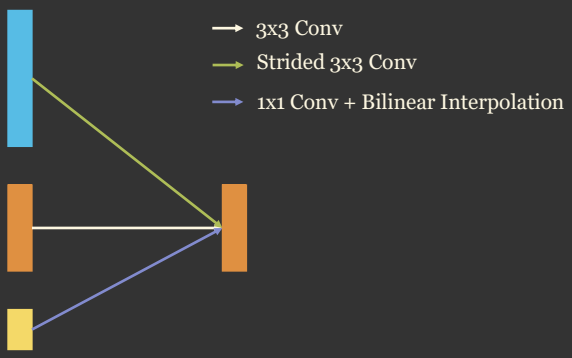
32

The key characteristics of HRNet are listed on this page. In comparison to other semantic segmentation methods, HRNet maintains high spatial resolution representations or feature maps throughout the entire architecture. Following the first stage of the architecture, a lower spatial resolution branch is added at each subsequent, and information is shared between these different spatial scales.



This slide conceptualizes a generic HRNet architecture containing 4 stages. A series of convolutional operations are implemented at each stage. After the first stage, each stage adds one more processing pipeline at a reduced spatial resolution. The second stage has 2 scales, the third has 3, and the fourth has 4. Information between these stages are shared at the end of each stage, as is described on the next slide.

HRNet Architecture



- Feature maps are added not concatenated
- Each resolution must have the same number of feature maps
- Change feature map counts = convolution
- Downsample = strided convolution decreases spatial resolution
- Upsample = bilinear interpolation

34

To share information between stages, HRNet uses addition as opposed to concatenation, which reduces the computational load of the model. This requires that all resolutions produce the same number of feature maps.

Downsampling is commonly performed using strided convolution where the stride used depends on the resolution reduction required.

Upsampling generally uses a combination of 1x1 convolution and bilinear interpolation.

Backbones

Backbones

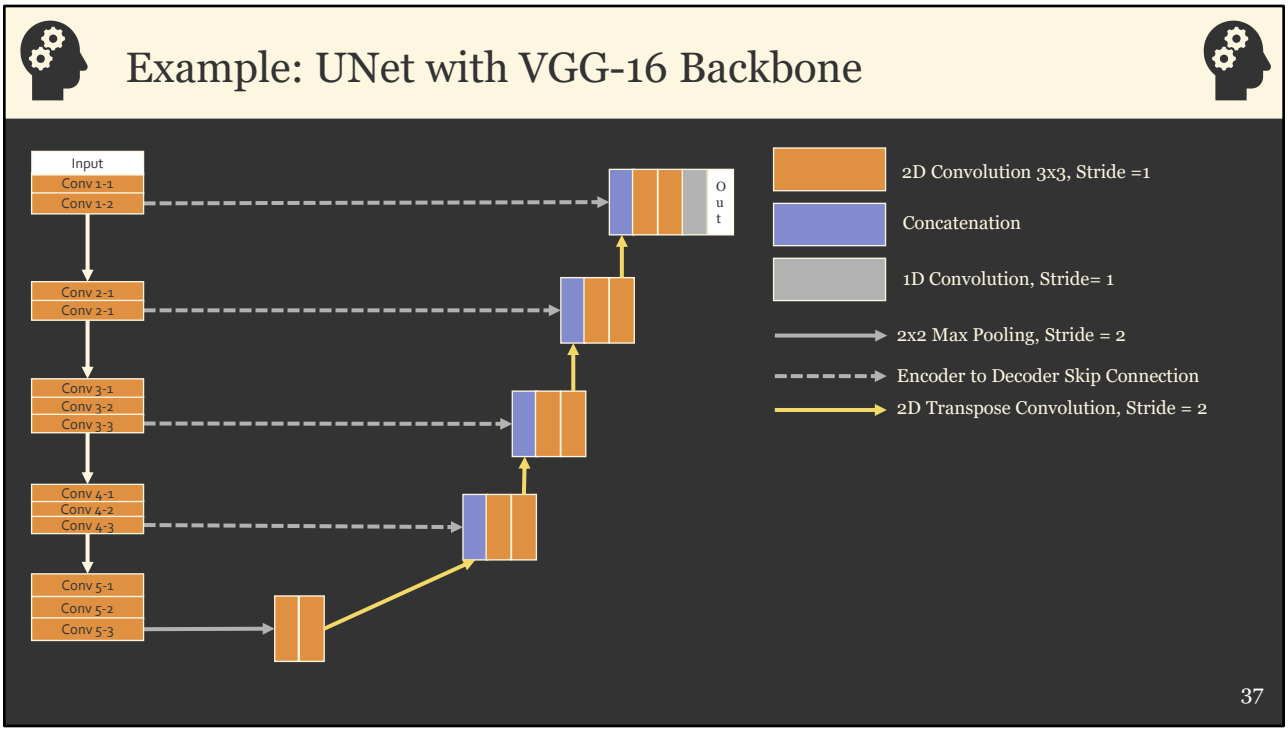
- Can use CNN backbones as encoder component
- Allows for using pretrained weights
- Encoder weights can be:
  - Frozen
  - Trained
  - Trained using variable learning rates
  - Trained during a subset of epochs
- Examples = ResNet, DenseNet, InceptionNet, EfficientNet, MobileNet, VGG

36

The backbone or encoder component of semantic segmentation models is equivalent to the convolutional layers of a CNN for scene labeling tasks. As a result, different CNN architectures can be used as the encoder component of semantic segmentation models.

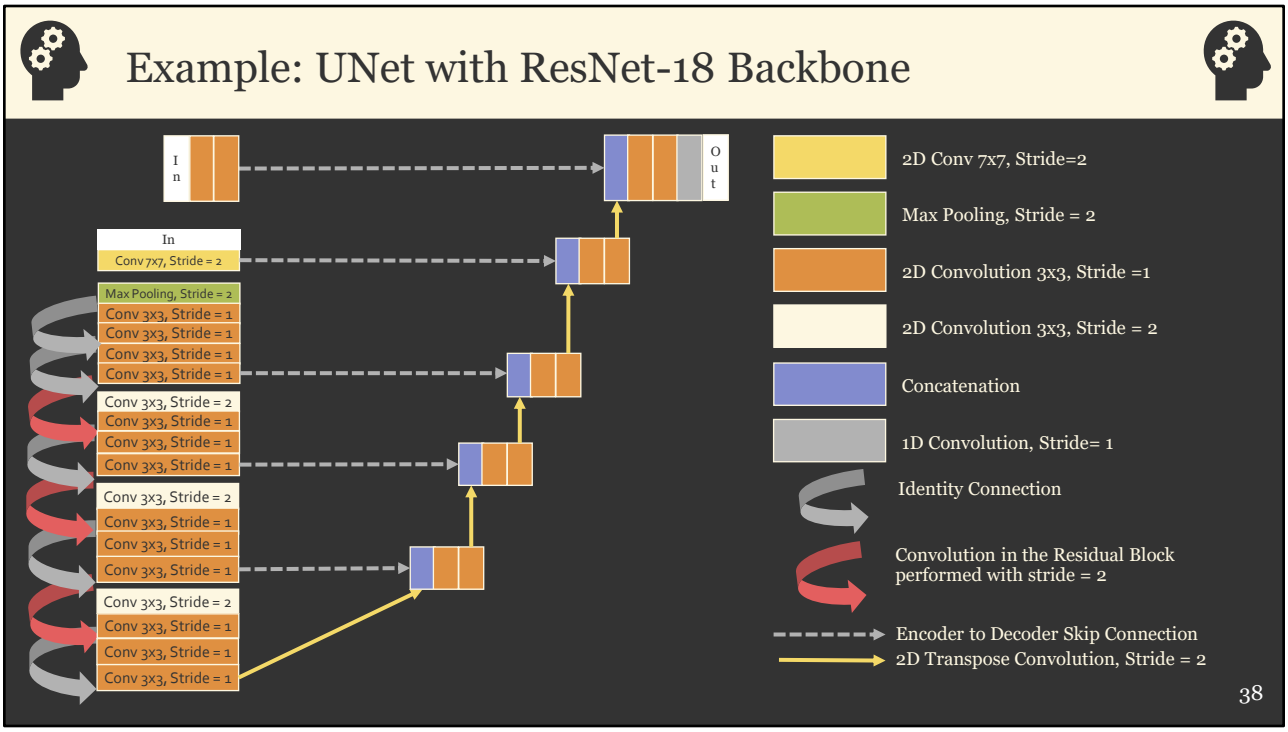
Since famous or commonly used architectures, such as those listed on this slide, have been trained using large datasets (e.g., ImageNet or COCO), this allows for pre-trained weights to be used in the backbone component of semantic segmentation architectures. Such transfer learning techniques may allow for improved model performance and/or faster training time. It is possible to freeze the weights/parameters in the backbone, initialize them with the pre-trained weights/parameters but still update them during the training process, or update them during only a subset of the training epochs.

The purpose of the encoder is to capture key, useful, or predictive spatial context information at varying spatial scales. When using transfer learning and a pre-trained backbone, the assumption is that the feature maps learned for the original task will be useful for the current task. If the encoder is frozen, it is assumed that the feature maps learned for the prior tasks will serve as an appropriate or predictive inputs to the decoder, which will be trained for the new task.



This slide conceptualizes using a VGGNet-16 architecture as the backbone or encoder for a UNet. Only the convolutional components are used; the fully connected layers are not included. Since skip connections are used, it is necessary to match the component of the VGGNet-16 architecture with the correct size in the spatial dimensions with the associated decoder block. This will be demonstrated in the PyTorch examples.

Once this architecture is defined, it is possible to initialize the model using pre-trained weights/parameters for the encoder and random weights/parameters for the decoder.



This slide conceptualizes using a ResNet-18 architecture as the encoder for UNet. Again, only the convolutional components are used, not the fully connected layers. The stages of the ResNet must be matched with the correct decoder block so that the tensor sizes in the spatial dimensions match. Also, the first set of operations in the ResNet architecture decrease the size of the input tensor in the spatial dimensions. As a result, an initial set of 2D convolution layers are used in the first encoder block while components of the ResNet architectures are used in the subsequent encoder blocks. Note that there are other ways to configure the first encoder block.

I will demonstrate implementing a ResNet architecture as the encoder for UNet in the PyTorch examples.

Practical Considerations

Segmentation Models

https://github.com/qubvel/segmentation_models.pytorch

https://github.com/qubvel/segmentation_models

40

In the PyTorch modules, I will demonstrate how to build a UNet from scratch by subclassing the nn.Module class defined by PyTorch. I will also demonstrate how to used pre-trained backbones or encoders. However, it can be difficult to build all semantic segmentation architectures from scratch, apply CNN-backbones, and load pre-trained weights/parameters. This is especially true as architectures become more complicated.

If you want to be able to use a variety of semantic segmentation architectures, define architecture components (such as the number of encoder and decoder blocks), use common backbones as the encoder component (such as VGGNet, ResNet, or MobileNet), and use pre-trained weights/parameters, I highly recommend checking out the Segmentation Models package. This package is available for both PyTorch and Keras/Tensorflow. I will demonstrate it in the PyTorch modules.

Backbones and Weights

- Use some common backbones as encoder
- Allow for application of pre-trained weights
- Decrease overfitting and training time
- May require certain number of input channels
- Freeze and unfreeze weights
- Train subset of layer weights

▼ ResNet

Encoder	Weights	Params, M
resnet18	imagenet / ssl / swsl	11M
resnet34	imagenet	21M
resnet50	imagenet / ssl / swsl	23M
resnet101	imagenet	42M
resnet152	imagenet	58M

▼ Inception

Encoder	Weights	Params, M
inceptionresnetv2	imagenet / imagenet+background	54M
inceptionv4	imagenet / imagenet+background	41M
xception	imagenet	22M

41

The Segmentation Models package allows for easily using different backbones and pre-trained weights/parameters. Again, this will be demonstrated in a PyTorch module.

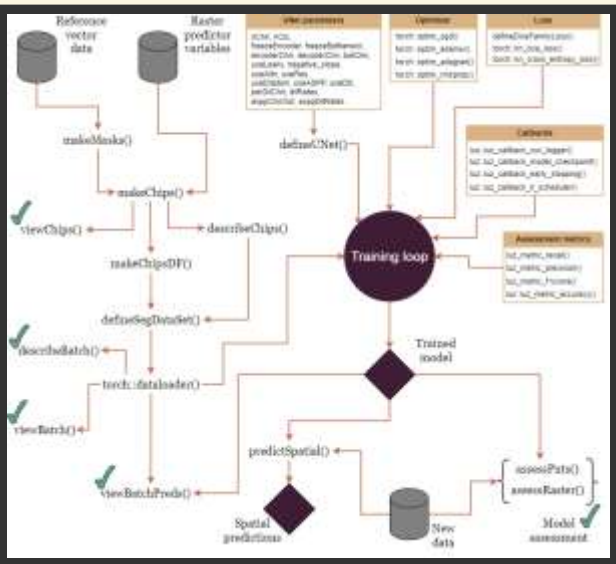
MMSegmentation

https://mmsegmentation.readthedocs.io/en/latest/
https://github.com/open-mmlab/mmsegmentation

42

Another great Python library that interfaces with and expands PyTorch is MMSegmentation.

Geospatial Deep Learning in R



https://github.com/maxwell-geospatial/geodl

https://mlverse.github.io/luz/

https://torch.mlverse.org/packages/

43

The author of this course has been working on developing an R package for geospatial semantic segmentation using the torch, luz, and terra packages. This package is called geodl.

This package is covered in our *Geospatial Supervised Learning* course.

Data Requirements

- Image chips of defined shape: (3, 512, 512), (3, 256, 256), (3, 128, 128), etc.
- Label/classified image with 1 numeric code per class and same shape as image chips
- Common formats: JPEG, PNG, TIFF

44

In order to perform semantic segmentation, you will need to generate image chips and associated masks where each code in the mask represents a category. For example, a binary classification would require a mask where each cell is coded as 0 or 1. The background class would be coded to 0 while the positive class would be coded to 1.

These chips are commonly saved in JPEG, PNG, or TIFF format in a folder structure that clearly links the image and its associated mask. For example, the images may be saved in an “image” folder while the masks are saved to a “label” or “mask” folder, and each image and its associated mask would have the same file name.

Some commonly used shapes are listed on this slide. Note that larger tensors will take more time and computational power to process. They will also need to be fed through the network during training using a smaller mini-batch size. The largest mini-batch size possible will depend on the size of each chip, your computer’s hardware (such as GPU VRAM), the complexity of the semantic segmentation architecture being trained, and the configuration of the training loop.

Using ArcGIS Pro

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

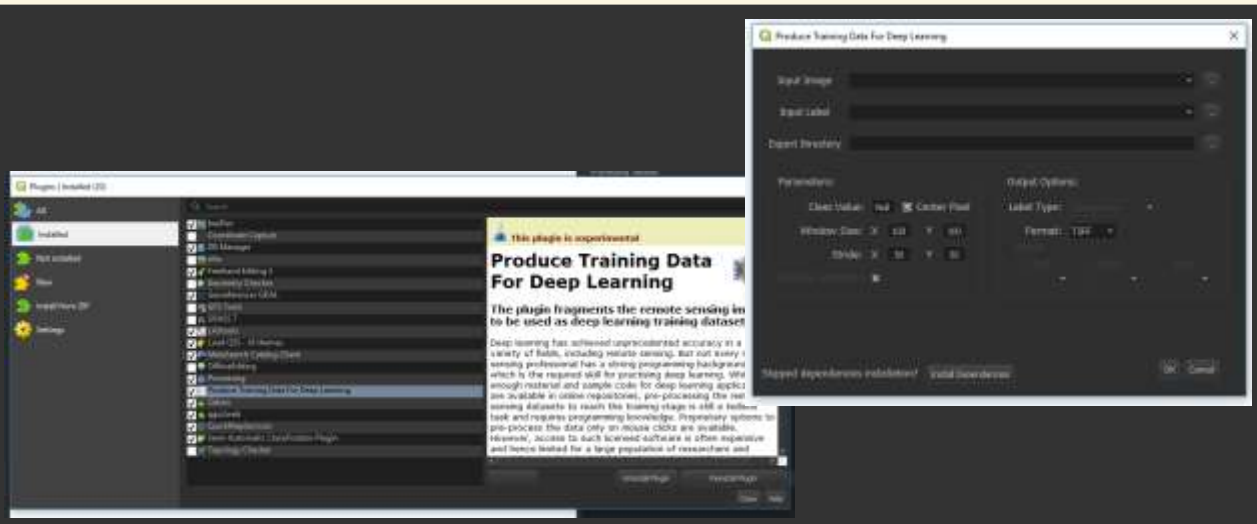
https://developers.arcgis.com/python/api-reference/arcgis.learn.html

45

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 is available with a valid Spatial Analyst or Image Analyst extension license.

This tool can generate image chips for a wide variety of tasks including scene labeling, semantic segmentation, object detection, and instance segmentation.

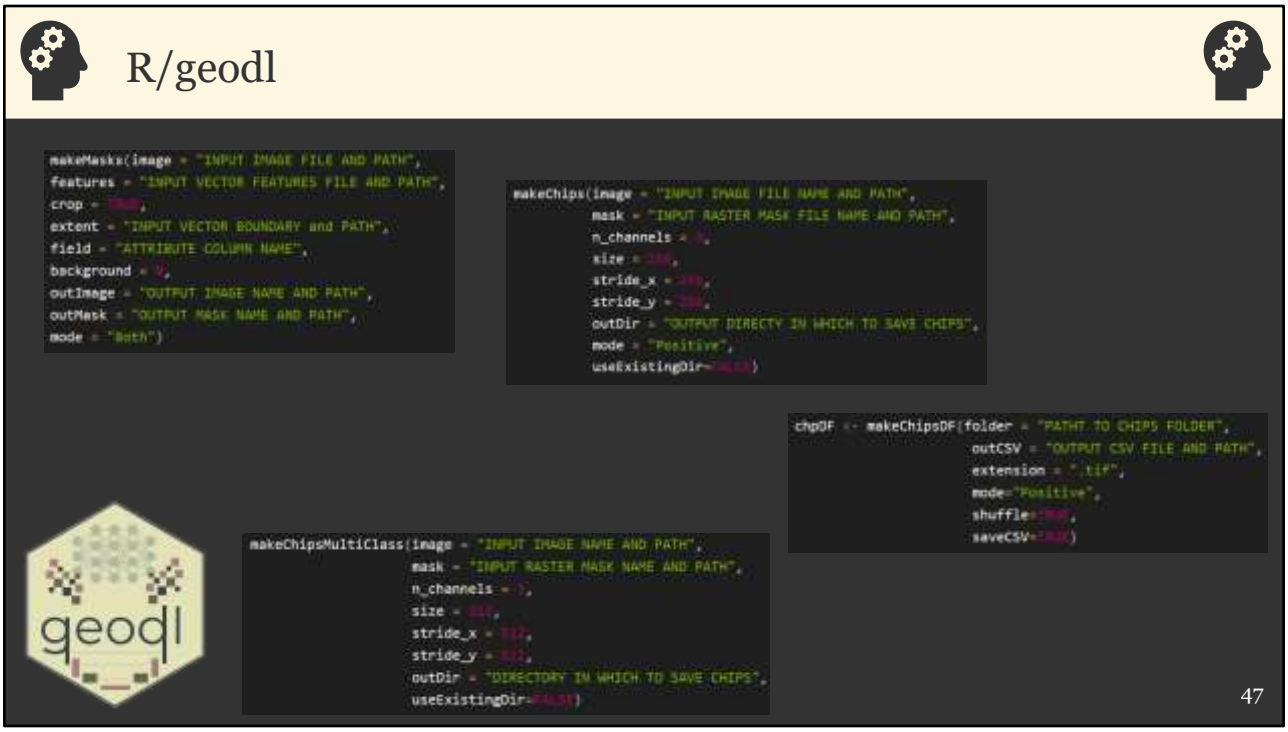
Using QGIS



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

46

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



The geodl R package provides functions for generating raster-based masks/labels, creating image chips and associated masks from larger spatial extents, and listing chips into a data frame, which can be saved to a CSV file.

Chips created using geodl can also be used in Python/PyTorch workflows.

Dynamic Chips

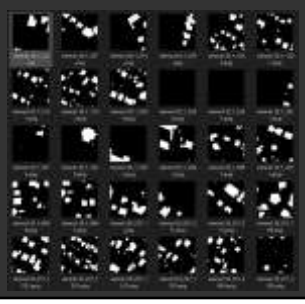
- TorchGeo
- geodl

48

Some tools, including TorchGeo in Python and geodl in R, allow for dynamically generating chips and associated masks of a defined size from larger extents or datasets as opposed to generating the chips beforehand and saving them to disk.

Dataset Prep

- Generate chips
- Mask with unique numeric code for each class
- Select chip size
- Augmentations applied to image and mask (as needed)
- Subclass PyTorch Dataset class



```
class MultiClassSegDataset(Dataset):

    def __init__(self, df, classes=None, transform=None,):
        self.df = df
        self.classes = classes
        self.transform = transform

    def __getitem__(self, idx):

        image_name = self.df.iloc[idx, 2]
        mask_name = self.df.iloc[idx, 3]
        image = cv2.imread(image_name)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        mask = cv2.imread(mask_name, cv2.IMREAD_UNCHANGED)
        image = image.astype('uint8')
        mask = mask[:,:,0]
        if(self.transform is not None):
            transformed = self.transform(image=image, mask=mask)
            image = transformed["image"]
            mask = transformed["mask"]
            image = torch.from_numpy(image)
            mask = torch.from_numpy(mask)
            image = image.permute(2, 0, 1)
            image = image.float()/255
            mask = mask.long().unsqueeze(0)
        else:
            image = torch.from_numpy(image)
            mask = torch.from_numpy(mask)
            image = image.permute(2, 0, 1)
            image = image.float()/255
            mask = mask.long().unsqueeze(0)
        return image, mask

    def __len__(self):
        return len(self.df)
```

49

In the provided PyTorch examples and in order to read in image chips and associated masks, I will subclass the PyTorch DataSet class to load in custom data. Some common processing tasks include reading in files, converting to numpy arrays, converting to torch tensors, moving data between the CPU and GPU, changing the order of the dimensions, and rescaling from 8-bit to float (0 to 1).

Again, we have and will continue to discuss DataSets and DataLoaders in detail in the PyTorch examples.

Binary vs. Multiclass

- Sigmoid = Binary
- Softmax = Multiclass
- Can apply class weights
- Can ignore class (incomplete training samples)

50

When performing binary classification, it is common to use a sigmoid function in the final layer to obtain class probabilities. For multiclass problems, softmax is used. Alternatively, if no function is applied raw logits are returned.

For binary classification and when using a sigmoid activation, only the logit for the positive class is predicted. When using softmax for a binary classification, logits for both the positive and negative classes are predicted. For multiclass problems, logits for all classes are predicted.

When determining whether you need to convert logits to probabilities with a sigmoid or softmax function, the key consideration is what is expected by the loss metric implementation. Some loss metric implementations will expect raw logits while other will expect estimated probabilities. We have and will continue to discuss this in detail in the PyTorch modules.

It is also possible to change the weight of each class when training the model and defining the loss metric. You can even ignore a class so that it does not impact the parameter updates, which can be useful when training data are incomplete or not wall-to-wall. We discussed class weighting methods in the Metrics and Losses module and will discuss them again in the Improving Models module.

Class Imbalance


remote sensing

MDPI

Article

Selecting and Interpreting Multiclass Loss and Accuracy Assessment Metrics for Classifications with Class Imbalance: Guidance and Best Practices

Sarah Farhadpour, Timothy A. Warner and Aaron E. Maxwell

- Use Dice loss (or Focal Dice, Tversky, Focal Tversky)
- Use Combined loss
- Use Weighted Cross-Entropy
- Augment class proportions in training set

Farhadpour, S., T.A. Warner, and A.E. Maxwell, 2024. Selecting and interpreting Multiclass loss and accuracy assessment metrics for classifications with class imbalance: guidance and best practices, *Remote Sensing*, 16(3): 533. https://doi.org/10.3390/rs16030533.

51

When classes are imbalanced, you can implement a loss function that is more robust to this issue, such as Dice loss, Tversky loss, or weighted CE loss. You can also use a combination of losses, such as Dice + CE or Tversky + CE.

You can also generate augmentations of the training data to increase the number of samples or the number of samples for specific classes. For example, you may choose to oversample the minority class and/or use a subsample of the more dominant class.

Learning from Incomplete Data

- Create a background class
- Background class has no weight in learning process or loss calculation
- An example: https://medium.com/geoai/high-resolution-land-cover-mapping-using-deep-learning-7126fee571dd
- Can be implemented using ArcGIS Pro

52

Adequate training data are not always available to generate masks with a classification or reference at each cell or location. This is a common issue; for example, you may only have a set of training polygons as opposed to a wall-to-wall reference set.

Such data can still be used as input to semantic segmentation methods. However, this will require that all cells that do not have an associated classification available be coded to an unlabeled class. During the process of training the algorithm, you will then need to assign a weight of zero to this unlabeled class so that it does not impact the model and loss calculations.

The link on this slide provides an example where this process was applied for land cover mapping. ArcGIS Pro/arcgis.learn can also generate and use incomplete training data for semantic segmentation.

Data Augmentations

```
test_transform = A.Compose(
    [A.PadIfNeeded(min_height=512, min_width=512, border_mode=4), A.Resize(512, 512),]
)

train_transform = A.Compose(
    [
        A.PadIfNeeded(min_height=512, min_width=512, border_mode=4),
        A.Resize(512, 512),
        A.RandomBrightnessContrast(brightness_limit=0.3, contrast_limit=0.3, p=0.5),
        A.HorizontalFlip(p=0.5),
        A.VerticalFlip(p=0.5),
        A.MedianBlur(blur_limit=3, always_apply=False, p=0.1),
    ]
)
```

https://github.com/albumentations-team/albumentations

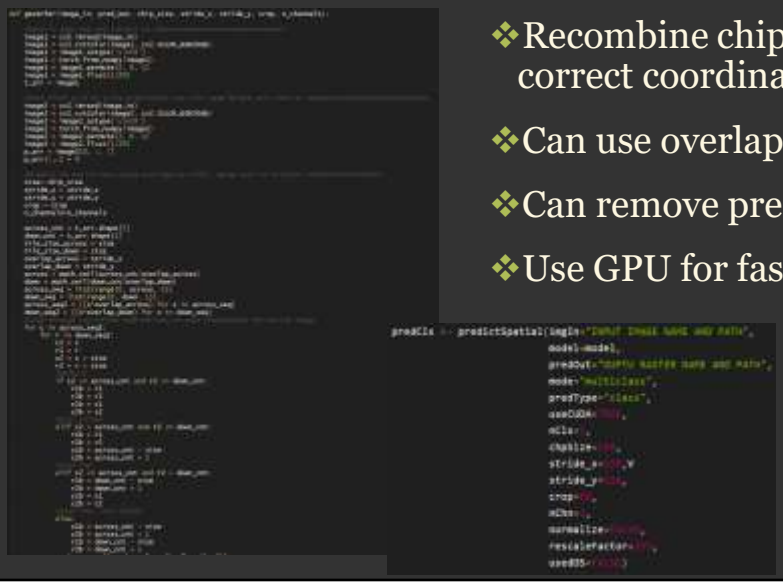
- Increase number of training samples
- Reduce overfitting
- Examples: flip, rotate, blur, sharpen, brightness, contrast, saturation, swap bands, and gamma
- Make sure to augment image and mask together!!!!
- Albumentations package

53

Again, data augmentation is useful for reducing overfitting, especially when the size of the training set is small. I recommend experimenting with the Albumentations package, which offers a wide variety of data augmentations and is easy to use. Torchvision also offers tools for data augmentation.

Since pixel-level alignment between images and masks must be maintained, it is important to apply the same set of transforms to image and mask pairs if the position of pixels in the chip are altered. This is important when applying rotations and flips, for example. This issue is generally handled well and automatically by the Albumentations package.

Merging Chips



- Recombine chips to single layer with correct coordinate reference information
- Can use overlap
- Can remove predictions at chip margins
- Use GPU for faster inference

54

For geospatial predictions, we commonly want to merge individual predictions made for individual image chips back to single, continuous maps covering the spatial extent of interest. When doing so, it is common to use overlapping chips and use only the predictions from the center of chips, which are generally more accurate than predictions near the margins.

In the examples, I will demonstrate methods for merging chips and assigning spatial reference information to generate map output.

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.