Transpose Convolutions and the Encoder-Decoder Architectures for Semantic Segmentation of Images

Avinash Kak

Purdue University

Lecture Notes on Deep Learning
Avi Kak and Charles Bouman

Wednesday 12th March, 2025 23:40

Preamble

To TOC

At its simplest, the purpose of semantic segmentation is to assign correct labels to the different objects in a scene, while localizing them at the same time.

What I have said above does not sound too different from what is described in my Week 7 lecture on Multi-Instance Object Detection (MIOD). So what's new with semantic segmentation?

The difference is one of the approach used rather than the end goal.

The difference between MIOD and semantic segmentation is best explained with an analogy drawn from the more traditional computer vision:

- MIOD is akin to cross-correlating an image with a pattern you are looking for and you mark those locations in the image where the output of cross-correlation is high. In such cross-correlations you are allowed to use the pattern at different scales.

- On the other hand, semantic segmentation is based on associating a class label with **each pixel** in the image based on a general understanding of what the **object pixels** look like in relation to the **background pixels**.

Preamble (contd.)

To give meaning to the phrase "general understanding of what the object pixels look like" at the bottom of the previous slide, let's consider an ML (Maximum-Likelihood) classifier that has been trained to separate the foreground pixels from the background pixels based on their statistical properties. Just to make a point, such a classifier may assume that both the foreground and the background pixels are normally distributed, the foreground with mean m_f and covariance Σ_f, and the background with mean m_b and covariance Σ_b. **Training such a classifier would amount to learning (m_f, Σ_f) for the foreground and (m_b, Σ_b) for the background.** At inference time, for a given pixel x, the classifier would calculate the two discriminants $d_c = \ln|\Sigma_c| + (x - m_c)\Sigma_c^{-1}(x - m_c)$ for $c \in \{f, b\}$. A pixel would be given the foreground class if $d_f > d_b$.

The ML analogy presented above is highly relevant here since, when you train a neural-network classifier with cross-entropy loss, the class label assigned to an input image is the maximum-likelihood estimate for that label. While this statement is about ML estimation of the class label for an entire image, the same logic applies to constructing such an estimate for a single pixel.

Preamble (contd.)

Pixel-level classification needed for semantic segmentation means that you certainly cannot use the more widely known convolutional networks that are meant for whole-image classification. These better-known networks push the image data through progressively coarser abstractions until at the final output the number of nodes equals the number of classes to which an image can belong.

For semantic segmentation, a network must still become aware of the abstractions at a level higher than that of a pixel — otherwise how else would the network be able to acquire a *general* sense of what the meaningful objects look like in relation to the background — but, at the same time, the network must be able to map those abstractions back to the pixel level.

This calls for using some sort of an Encoder-Decoder architecture for a neural network. The Encoder's job would be to create the high-level abstractions in an image for acquiring a general sense of what the objects look like. And the Decoder's job would be to map that general sense back into the image.

Preamble (contd.)

That what's mentioned on the previous slide could be implemented in a fully convolutional network was first demonstrated by Ronneberger, Fischer and Brox in the paper "U-Net: Convolutional Networks for Biomedical Image Segmentation" that you can download from this link:

`https://arxiv.org/abs/1505.04597`

The two key ideas that the Unet network is based on are: **(1)** The concept of a "Transpose Convolution"; and **(2)** notion of skip connections. The transpose convolutions are used in the Decoder to up-sample the abstractions till we get back to the pixel-level resolution. And the skip connections are used as pathways between the Encoder and the Decoder **to help the Decoder recover the fine pixel-level detail that was lost during encoding where the focus is on helping the neural network understand the inter-pixel relationships at high levels of abstraction**.

It is important to appreciate the fact that the role of skip connections in the networks meant for semantic segmentation goes beyond what I discussed in my Week 6 lecture — now it is also on helping the decoder recover the fine image detail that was lost during encoding.

Preamble (contd.)

Using skip connections for the recovery of fine detail in the images as mentioned on the previous slide, think of that as yet another "architectural feature" in designing neural networks. For a deeper understanding of this feature, you'll need to dig deeper into Generation 5 networks for semantic segmentation — See Slides 24-27.

I'll be illustrating several of the ideas presented so far in this Preamble through DLStudio's implementation of Unet under the name `mUnet`, with the letter 'm' standing for "multi' for `mUnet`'s ability to segment out multiple objects of different types simultaneously from a given input image.

Additionally, `mUnet` uses skip connections not only between the Encoder part and the Decoder part, as in the original `Unet`, but also within the Encoder and within the Decoder. You will find the code for `mUnet` the inner class `SemanticSegmentation` of the DLStudio platform:

`https://pypi.org/project/DLStudio/`

Preamble (contd.)

That takes me to saying a couple of things about how this lecture is organized: Since the Decoder part of an Encoder-Decoder architecture uses the notion of transpose convolutions to enlarge the Encoder output progressively until its size is the same as that of the input image, my first task in this lecture is to explain what's meant by a transpose convolution.

However, to fully appreciate the notion of a transpose convolution, you must first understand how a 2D convolution can be expressed as a matrix-vector product. Therefore, that's the first topic I'll address in this lecture after a brief survey of some of the better-known approaches that have been developed over the years for semantic segmentation.

Outline

Outline

Semantic Segmentation and Scale Invariance

- State of the art semantic segmentation networks use atrous convolution for achieving scale invariance without increasing the number of learnable parameters.
- Scale invariance means that the ability of a neural network to segment out the pixel blobs for a given class label should not depend on the size of that blob.
- Ordinarily, the larger the same-class pixel blobs, the larger the number of layers in a neural network needed for discovering the togetherness of the pixels in each blob. Obviously, the larger the number of layers, the larger the number of learnable parameters involved. **Atrous convolutions are a way out of this dilemma.**
- In the rest of this section, I'll first explain the idea of atrous convolutions and then present a brief survey of the more significant milestones in the literature on semantic segmentation.

Explaining Atrous Convolution

- Before explaining the idea of atrous convolutions, note that it is also known as the dilated convolution.

- In order to understand atrous convolution, let's first take another look at regular convolution of a 16×16 input with a 3×3 kernel.

- Here is a 16×16 input for the convolution (actually a correlation):

```
[[4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]
 [4. 4. 4. 4. 4. 4. 4. 4. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

- And here is the 3×3 kernel:

```
[[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]
```

Explaining Atrous Convolution (contd.)

- Shown below is the result of convolving the 16×16 input with the 3×3 kernel. We obtain the result by sweeping the kernel over the input.
- Since no padding is being used, the output is smaller than the input, the size of the output being 14×14.
- As you can see, the feature extracted is the vertical edge in the middle of the image.

```
[[0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 9. 9. 0. 0. 0. 0. 0. 0.]]
```

Explaining Atrous Convolution (contd.)

- Let's consider Rate 2 atrous convolution. At this rate for the dilation of the kernel, we alternate the original kernel entries with rows and columns of zeros.

- Shown below is the new kernel:

```
[[-1.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
```

- The dilated kernel leads to the following output. Since we are still not using any padding, the enlarged kernel results in a smaller output. Its size is now 11×11.

```
[[0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]
 [0. 0. 0. 9. 9. 9. 9. 0. 0. 0. 0.]]
```

Explaining Atrous Convolution (contd.)

- The basic idea of an atrous convolution is to detect an image feature at a specific scale — the scale being determined by the "rate" as set by the value of the `dilation` parameter in the call to the constructor for the `nn.Conv2d` class.
- Therefore, if you carry out such convolutions at different dilation rates, you will have a multi-scale representation of an image for a given feature type, the type of the feature being determined by the *base form* of the kernel (which is only a nominal notion since the neural network will decide on its own the values for the learnable parameters in the convolutions at each of the scales used.)
- Nonetheless, it remains that in a multi-scale representation created by atrous convolution, it is possible for the number of learnable parameters to remain the same at all scales.

Atrous Spatial Pyramid Pooling (ASPP)

- To elaborate what I mentioned on the previous slide, given a set of training images, let's say we have designed a semantic segmentation network that uses just 3×3 kernels but with different dilation rates for creating a multiscale pyramid representation for the input images.

  Each scale will only involve 9 learnable parameters since the rest would be clamped to 0.

- The neural network will figure out on its own what values to learn for the 9 parameters at each scale.

- Typically, the output at each scale serves as one channel of a multi-channel input to a 1×1 convo layer whose output is the segmentation map for the input. This is the basic idea behind what is known as Atrous Spatial Pyramid Pooling (ASPP). I'll explain this in greater detail in the slides to follow in the rest of this section of the lecture.

ASPP (contd.)

- For best results, ASPP is NOT applied directly to the training images. On the other hand, it is applied to the output of a convolutional network.
- Each channel of the output of a convolutional layer is typically referred to as a feature map. What the previous bullet says is that ASPP is likely applied to the feature maps at the output of a multi-layer purely convolutional network as opposed to the raw images.
- **The job of ASPP is to examine the feature maps at multiple scales and to put out a segmentation map for the input on that basis.**
- The above implies that a segmentation map produced by ASPP will, in general, not be of the same size as the original images.
- Therefore, when a semantic segmentation network is based on ASPP, you still have to solve the problem of mapping the segmentations at the output of ASPP to an output image of the same size as the input images.

Outline

Generation 1 Networks for Semantic Segmentation

- These fully convolutional networks for semantic segmentation involve mostly the same steps as a CNN for image classification, the main difference being that whereas the latter uses a linear layer at the end of the network whose output yields the class label, the former ends in a reduced resolution representation of the input image.
- The main idea here was to compare the reduced resolution predicted output image with the ground-truth segmentation masks at the same resolution and to then propagate out the assigned segmentation labels to the pixels in the input image taking into account pixel-value continuities vis-a-vis label-value continuities. See the pub:

`https://arxiv.org/pdf/1605.06211.pdf`

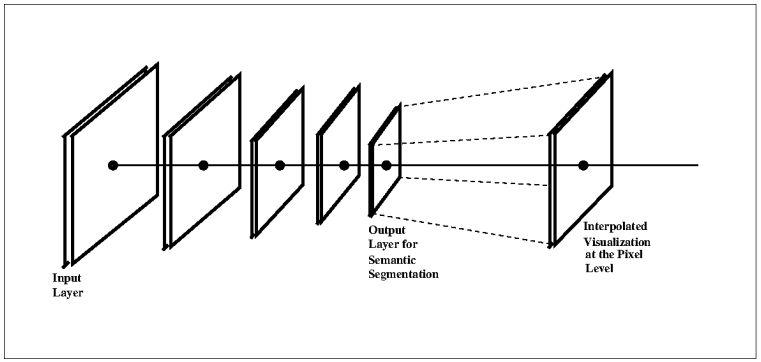


Figure: A First Generation Network for Semantic Segmentation

Generation 2 Networks

- The second generation networks are based on the Encoder-Decoder principle, in which the job of the Encoder is to learn which neighboring pixels belong together (because they possess the same semantic label) through progressively reduced-resolution (although with increased number of channels) representations of the input.
- Decoder's job is to map the inter-pixel properties learned back to the pixel level at the original resolution. The accuracy of the detail in the back-to-pixel mapping in the Decoder is aided by the skip connections from the Encoder as shown below: `https://arxiv.org/abs/1505.04597`

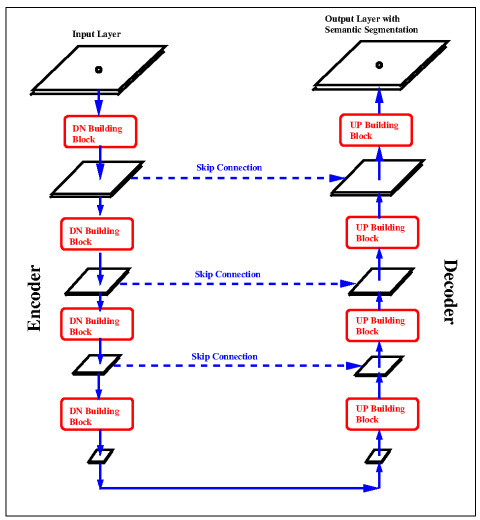


Figure: Second Generation Networks for Semantic Segmentation

Generation 3 Networks

- The third generation networks for semantic segmentation are based on the idea of ASPP — Atrous Spatial Pyramid Pooling — an idea that I first described on Slides 15 and 16.
- As shown below, along with a 1×1 convo, the three Atrous operators are applied to the output of an Encoder (see prev slide for Encoder).
- Each atrous operator produces a single channel output. You pool together all these one-channel outputs to form a multi-channel input to a 1×1-kernel convolutional layer whose output is again a one-channel segmentation map.

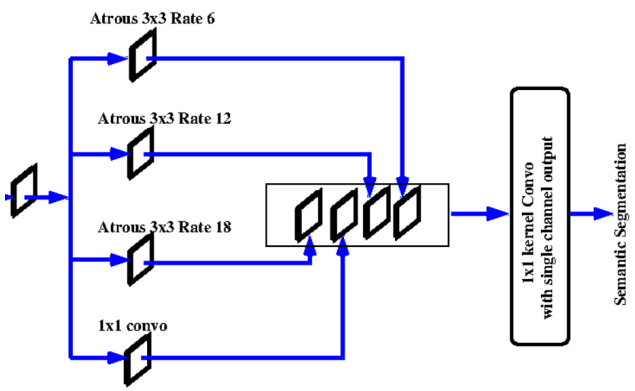


Figure: The basic idea of ASPP for Semantic Segmentation

Generation 3 Networks (contd.)

- Typically, ASPP as shown on the previous slide is applied to the output of a regular convolutional network (used an an Encoder) like ResNet-101 or VGG-15.
- If the output of the Encoder consists of N channels, each atrous operator will map N channels at its input into a single output channel.
- It is these single-channel outputs from the individual atrous operators that are pooled together as shown in the figure on the previous slide and also in the figure on the next slide.
- If you used M atrous operators, the 1×1 convo in the vertical box at right in the figure on the next slide will have $M+1$ channels at its input (which includes the output of the the 1×1 convo running in parallel with the atrous convos) and a single channel at its output. That output channel is the segmentation mask predicted by the network.

Generation 3 Networks (contd.)

- That creating atrous-based multi-scale representations for the output of an Encoder could lead to powerful segmenters was first shown in the following paper. In the authors' architecture shown below, the network you see on the left of ASPP can be either ResNet-101 or VGG-16.

`https://arxiv.org/abs/1606.00915`

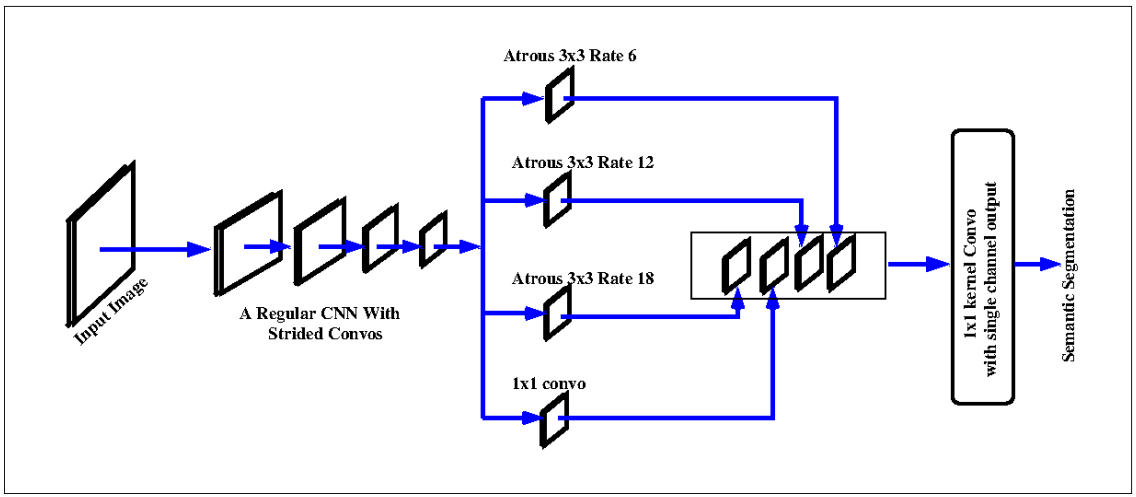


Figure: Third Generation Networks for Semantic Segmentation

Generation 4 Networks

- The spatial resolution in the segmentation masks produced by a third generation semantic-seg network will be the same as for the features maps produced by the Encoder. You would still be faced with the problem of extrapolating those pixel labels to what you'd need at the resolution of the input images.
- This problem is resolved in the fourth-generation networks by deploying a Decoder (along the lines of the 2nd generation Encoder-Decoder networks) that learns to extrapolate the labels to the full resolution of the input data.
- Here is the paper that showed how the atrous-convo based learning could be carried in an Encoder-Decoder framework to produce high-quality segmentation masks:

  `https://arxiv.org/abs/1706.05587`

- The next slide has additional information regarding fourth geneation networks.

Generation 4 Networks (contd.)

- Shown below is the semantic-seg network from the publication cited on the previous slide. Note that the Encoder side of the network along with its atrous head is exactly the same as the network depicted on Slide 22.

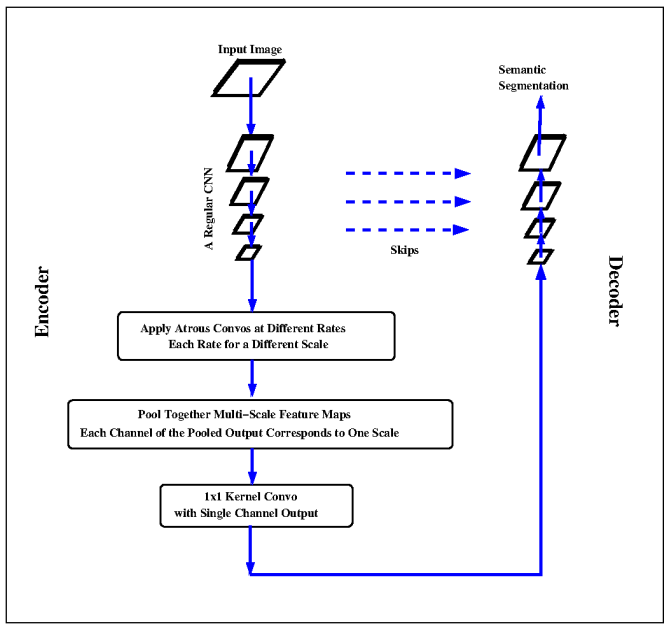


Figure: Fourth Generation Networks for Semantic Segmentation

- The job of the Decoder side shown above is to grow the size of what's produced by the atrous head of the Encoder back to the input image size for a pixel-level classification of the input

A Generation 5 Network

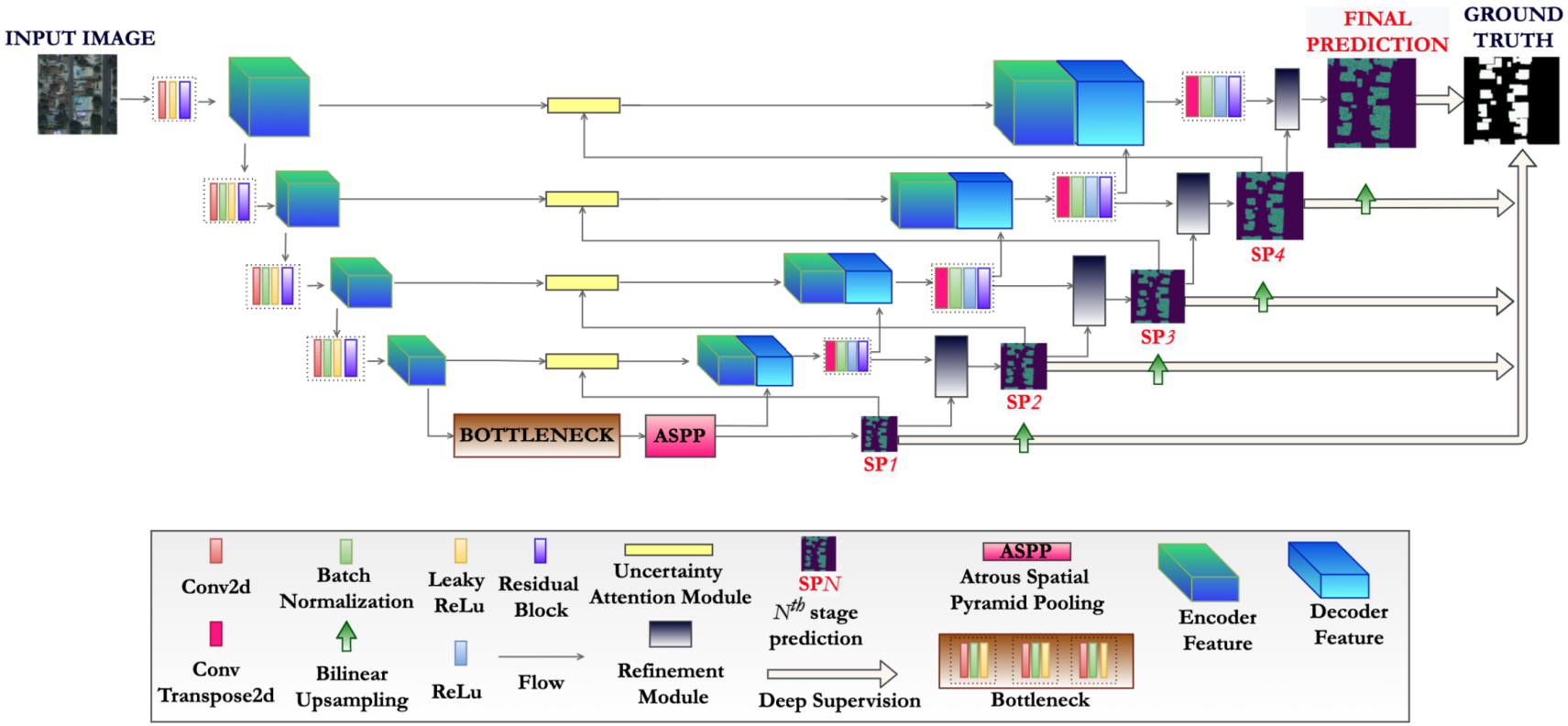
- Shown on the next slide is a 5^{th} generation semantic generation network for the segmentation of buildings in satellite and aerial images.
- This contribution, from Purdue, is ranked at number 2 in the "DeepGlobe Building Extraction Challenge" at the following website:

  `https://competitions.codalab.org/competitions/18544#results`

  Our entry is under the username 'chattops' with the upload date November 30, 2021.
- The details of this network are presented in the following paper that was published in IEEE-JSTAR, Vol, 15, pp. 3146-3167, 2022:

  `https://arxiv.org/pdf/2112.05335.pdf`

A Generation 5 Network (contd.)



A 5th generation image segmentation network from
https://arxiv.org/pdf/2112.05335.pdf
published in IEEE-JSTAR, Vol, 15, pp. 3146-3167, 2022

A Generation 5 Network (contd.)

- The encoder side of the network shown on the previous slide is not that different from a typical encoder in, say, the Unet. Note, however, that, in the network shown, in addition to the skip connections from the encoder side to the decoder side, we also have skip connections *along* the encoder side and *along* the decoder side. These are the "bottleneck residual blocks (a la ResNet)" in each arm of the U.

  [**Note the TWO different uses of the word "bottleneck": Following the ResNet terminology, we talk about the "bottleneck residual blocks", which are basically skip-blocks, in the Encoder and the Decoder. And we talk about the "BOTTLENECK layer at the bottom of the U". The bottleneck residual blocks are shown by dotted squares containing four verical bars, with each bar standing for an element like a conv2d or convTranspose2d layer, a BN layer, etc.**]

- An original aspect of the network architecture is that the skip connections that go across from the encoder side to the decoder side are processed by the Uncertainty Attention Module described in the paper.

- The feature maps that are produced by the BOTTLENECK layer at the bottom of the "U" are processed by an ASPP module for scale-invariant learning of segmentations. The next slide presents the parameters of the ASPP module at the bottom in the network.

A Generation 5 Network (contd.)

- The ASPP module (shown in red at the very bottom of the network) consists of a 1×1 Conv layer, three 3×3 Conv layers with dilation rates of 2, 4, and 6, and a global context layer incorporating average pooling and bilinear interpolation. The resulting feature maps from the five layers of ASPP are concatenated and passed through another 3×3 Conv layer, where they form the output of the ASPP module that is fed directly into the decoder.
- In addition to the above, the feature maps from the ASPP module pass through a 1×1 Conv layer to produce the top-most prediction map that is low in resolution but rich in semantic information. This is the same Conv layer you see in the vertically oriented box at the right end of what is shown in the figure on Slide 22.

Outline

Revisiting the Convolution in a Convo Layer

- As mentioned in the Preamble, semantic segmentation requires pixel-level classification in an image, but the logic of this classification must be driven by higher-level pixel groupings that are detected in an image.
- The Preamble also mentioned that this requires using an encoder-decoder framework, with the encoder devoted to the discovery of higher-level data abstractions and the decoder devoted to the mapping of these abstractions back to the pixel level with the help of **transpose convolutions**.
- **To understand what is meant by a transpose convolution, we must first express a regular 2D convolution as a matrix-vector product.** We start with that on the next slide.

Implementing a Convolution as a Matrix-Vector Product

- Consider convolving a 3×3 input pattern with a 2×2 kernel to produce a 2×2 output:

```
                              1  2  3
a  b                                          1xa+2xb+4xc+5xd   2xa+3xb+5xc+6xd
          convolved with      4  5  6     =
c  d                                          4xa+5xb+7xc+8xd   5xa+6xb+8xc+9xd
                              7  8  9
```

- In our own mind's eye we like to imagine 2D convolution as sweeping the kernel over the input pattern, multiplying the two element-wise at each position of the kernel, and summing the result.

- However, for efficient computations, especially when you want to use a GPU (**GPUs are designed to carry out thousands of matrix-vector products in parallel**), we must implement convolution as a matrix-vector product.

Convolution as a Matrix-Vector Product (contd.)

- We can create a matrix-vector product implementation of the convolution shown on the previous slide by either vectorizing the kernel or the input. If we vectorize the kernel, the vector-matrix product will look like:

$$\begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix}$$

  This would produce a 4-element output vector that can be reshaped into the desired 2×2 result.

- As to how the elements of the inputs are arranged in each column of the matrix shown above would depend on both the size of the kernel and the size of the input. [Regarding the matrix, along each column you will see contiguous elements except at those positions that correspond to the end of one horizontal sweep by the kernel. For example, 1 is followed by 2 in the first column, but then 2 jumps to 4 because that marks the beginning of the next sweep. The same thing happens in the second column. Also note that the start of the second column is a unit-shifted version of the start in the previous column. After the start in the second column, you again have the same contiguity property as in the first column. So the starting value of 2 must be followed by 3. And so on.]

Convolution as a Matrix-Vector Product (contd.)

- This slide shows the other approach to representing a convolution by a matrix-vector product:

$$\begin{bmatrix} a & b & 0 & c & d & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & c & d & 0 & 0 & 0 \\ 0 & 0 & 0 & a & b & 0 & c & d & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & c & d \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}$$

  This would again return an array of 4 elements that would need to be reshaped into a 2×2 output.

- I believe this approach to the representation of a 2D convolution by a matrix-vector product is more popular — probably because it is easier to write down the matrix with this approach.

Convolution as a Matrix-Vector Product (contd.)

- In the second approach shown on the previous slide, you lay down all the rows of the kernel in the top row of the matrix and then you shift it by one position to the right with the caveat that the shift needs to be more than one position when, in the 2D visualization, the kernel has reached the rightmost positions in each horizontal sweep over the input.
- Let $\mathbf{C}$ represent the matrix obtained in our matrix-vector product representation of a 2D convolution. If $\mathbf{x}$ represents the input pattern and $\mathbf{y}$ the output, we can write:

$$\mathbf{y} = \mathbf{C}\mathbf{x}$$

This relationship between the input/output for one convo layer is displayed at left in the figure on the next slide.

Convolution as a Matrix-Vector Product (contd.)

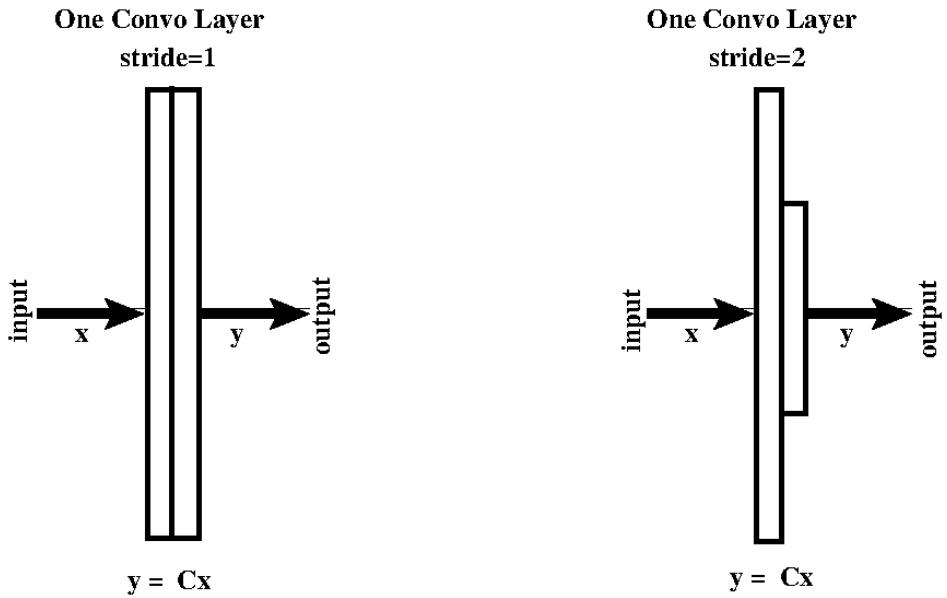


Figure: Forward data flow through one convo layer in a CNN. The depiction at left is for the case when the stride equals 1 and the one at right for the case when the stride equals 2.

Using the Matrix $\mathbf{C}^T$ to Backpropagate the Loss

- What's interesting is that the mathematics of the backpropagation of the loss dictates that if the data flows forward through a convo layer as $\mathbf{y} = \mathbf{C}\mathbf{x}$, then the loss backpropagates through the same layer as

$$Loss_{backprop} = \mathbf{C}^T Loss$$

  as shown at left in the figure on the next slide.

- In our discussion so far on representing a convolution by a matrix-vector product, we implicitly assumed that the stride was equal to one along both axes of the input. With this assumption, except for the boundary effects, the output of the convo layer will be of the same size as that of the input.

- Let's now consider what happens to the matrix $\mathbf{C}$ in our matrix-vector product representation of a convolution when the stride equals 2 along both axes of the input.

Using the Matrix C^T to Backpropagate the Loss (contd.)

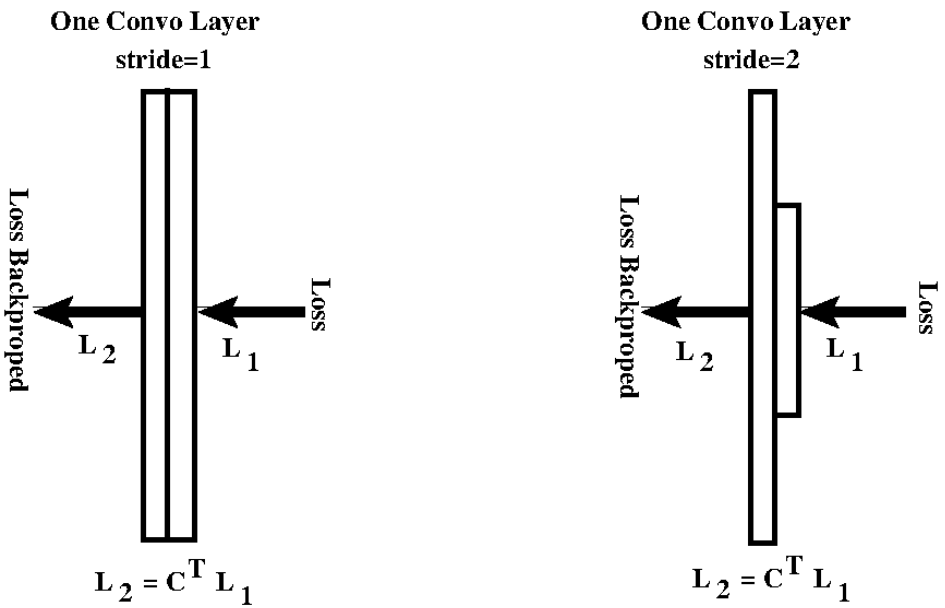


Figure: Backpropagating the loss through one convo layer in a CNN. The depiction at left is for the case when the stride equals 1 and the one at right for the case when the stride equals 2.

Convolution as a Matrix-Vector Product for a Larger Input

- To best visualize what happens to the matrix **C** when the stride is greater then 1, we need a slightly larger input. So let's consider a 2×2 kernel and a 4×4 input as shown below for a stride that equals 1 as before along both axes of the input:

```
a  b                          1  2  3  4
                convolved with
c  d            [stride=(1,1)]  5  6  7  8      =   1a+2b+5c+6d      2a+3b+6c+7d      3a+4b+7c+8d
                                9  10 11 12         5a+6b+9c+10d     6a+7b+10c+11d    7a+8b+11c+12d
                                13 14 15 16         9a+10b+13c+14d   10a+11b+15c+16d  11a+12b+15c+16d
```

- We can represent this 2D convolution with the matrix-vector product:

$$\begin{bmatrix} a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \\ 16 \end{bmatrix}$$

Transitioning from Stride $= (1, 1)$ to Stride $= (2, 2)$

- In the matrix shown on the previous slide, as before, each row of the matrix is a shifted version of the row above. While the shift is one element for the consecutive sweep positions of the kernel vis-a-vis the input image, the shift becomes 2 positions when the kernel has to move from one row of the input to the next.
- The matrix-vector product can again be presented by the notation $\mathbf{y} = \mathbf{C}\mathbf{x}$ where $\mathbf{C}$ is now the 9×16 matrix shown on the previous slide.
- **Let's now consider the same convolution but with a stride of $(2, 2)$.**
- In order to understand how to modify $\mathbf{C}$ when *stride* $= 2$, it is best to start with the $\mathbf{C}$ on the previous slide for the unit stride case and to mentally partition this matrix vertically into sections, with each section corresponding to one row of the output.

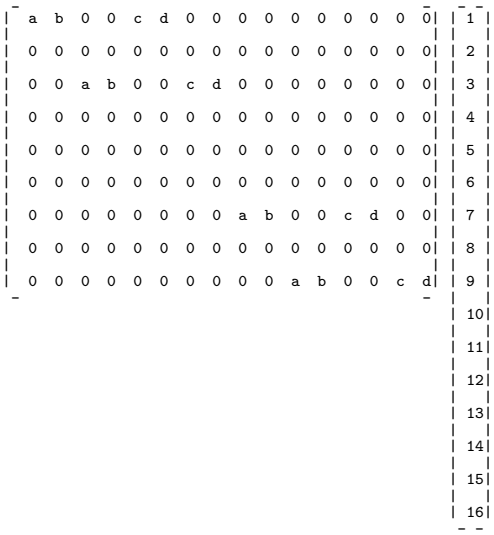
Convolution as a Matrix-Vector Product for Stride $= (2, 2)$

- For *stride* $= 1$, the first three rows of the matrix **C** on Slide 38 correspond to the first row of the output, the second three rows to the second row of the output. And the last three rows to the third row of the output.
- For the case of *stride* $= 2$, we zero out alternate rows **in** each above-mentioned vertical divisions of the matrix shown on Slide 38 since those would correspond to the horizontal jumps as dictated by the stride. This explains the all-zero rows for the 2nd and the 8th rows of the matrix shown on the next slide for the case of *stride* $= 2$.

  [**We want the second row of the matrix shown on Slide 38 to be all zeros since the second element of the 3×3 output will now be 0. By the same token, we want the 8^{th} row of the matrix to be all zeros since the 8^{th} element of the 3×3 output will be zero. Also note that the entire middle row of the 3×3 output will be zero.**]
- For the vertical jumps dictated by the striding action on the sweep motions of the kernel, we must also zero out entire vertical partitions of the matrix that was shown earlier for the *stride* $= 1$ case. This is the reason for why the 4th, the 5th and the 6th rows are all zeroed out in the matrix shown on the next slide for the case of *stride* $= 2$.

Convolution as a Matrix-Vector Product for Stride $= (2, 2)$ (contd.)

- Here is the matrix-vector product for a stride of 2 along both axes:



- If $\mathbf{C}$ again represents the 9×16 matrix that is shown above, the data flow in the forward direction and the loss backpropagation can again be expressed as before:

$$\begin{aligned} \mathbf{y} &= \mathbf{C}\mathbf{x} \\ Loss_{backprop} &= \mathbf{C}^T Loss \end{aligned}$$

provided you account for the intentionally zeroed out rows in the matrix $\mathbf{C}$. The figures on the right in Slides 35 and 37 are for the case when the stride equals 2.

Reformatting the Output Vector for Stride $= (2, 2)$

- As it is, the output vector **y** will have 9 elements in it (the same as was the case when the stride was equal to 1). However, since we intentionally deleted (by zeroing them out) several rows of the matrix **C**, we would need to drop those elements from **y** before we accept the output of the matrix-vector product.
- **It is this deletion of the elements of y that makes the output of the convo layer to have a size that, except for the boundary effects, would be half the size of the input.**
- Note the bottom line here: When the stride is 2, the matrix **C** maps the input data **x** to the output **y** that is roughly half the size of **x**. And, going in the opposite direction, the transpose matrix $\mathbf{C}^T$ maps the loss in the layer where **y** resides to the layer where **x** resides. In other words, **$\mathbf{C}^T$ maps the loss from a layer of some size to a layer that is twice as large.**
- It is this property of $\mathbf{C}^T$ that we will focus on in the next slide.

Convolutions with Dilations

- Now that you understand what is meant by a transpose convolution, let me revisit the topic of performaing convolutions with dilations — a topic I discussed earlier in the context of atrous convolutions.
- As I mentioned in our discussion on atrous convolutions, the simplest example of a dilation would be to insert alternating zeros in your kernel, which would double the footprint of the kernel in both the horizontal and the vertical directions as the kernel is sweeping over the image. This would correspond to setting the `dilation` parameter to 2 in a call to the constructor for `nn.Conv2d`. The default value for this parameter is 1, which amounts to no dilation.
- When you have a $k \times k$ kernel with a `dilation` of, say, 2, you are actually convolving the image with a $2k \times 2k$ operator — **but you are forcing the learning framework into making sure that the alternating elements of the operator remain zeros.** What that implies is that for all values of the `dilation` parameter, the number of learnable parameter remains the same.

Focusing on the Upsampling Ability of $\mathbf{C}^T$

- **With dilations, when you talk about a $k \times k$ kernel, that's only a statement about the number of the learnable parameters involved. And not about the size of the operator for the convolutions.**
- In general, based on the discussion so far, we can assign two different roles to the matrices $\mathbf{C}$ and $\mathbf{C}^T$:
  - We can use $\mathbf{C}$ in the encoder since (when the strides are greater than 1) it creates coarser abstractions from the data and that's what an encoder is supposed to do.
  - And we can use $\mathbf{C}^T$ in the decoder since (when the strides are greater than 1) it creates finer abstractions from coarser abstractions.
- **We can use matrices like C to create feature and object level abstractions from the pixels and use matrices like $\mathbf{C}^T$ to map those abstractions back to the pixel level.**
- Given the importance of $\mathbf{C}^T$ in encoder-decoder learning frameworks, it has name of its own: It is known as the "**Transpose Convolution**".

A Useful Reference on "Convolution Arithmetic"

- For deeper insights into the relationship between the convolutional layers and the transposed convolutional layers, see the following 2018 article by Dumoulin and Visin:

  `https://arxiv.org/abs/1603.07285`

- This reference is a great resource for understanding how the output shape depends on the input shape, the kernel shape, the zero padding used, and the strides for both convolutional layers and the transposed convolutional layers.

Outline

PyTorch's `nn.ConvTranspose2d` Class

- In PyTorch, the class that helps you carry out transpose convolutions for regular images is `nn.ConvTranspose2d`. This link will take you directly to its API:

  https://pytorch.org/docs/stable/nn.html#convtranspose2d

- If this is your first exposure to transpose convolutions, you are likely to be confused (nay frustrated) by the relationship between the kernel size, the value you choose for padding, and the output size in the API of the `nn.ConvTranspose2d`. The goal of this section is to clarify this relationship.

- Starting with the next slide, I'll illustrate the above mentioned relationship with an example that uses transpose convolution to dilate 4-channel 1×1 noise image into a 2-channel 4×4 image by using a 4×4 kernel. The code for this example is on the next slide.

Dilating a One Pixel Image into a 4×4 Image with ConvTranspose2d

```
batch_size = 1                                                                                                  ## (A)
input = torch.randn((batch_size, 4)).view(-1, 4, 1, 1)          ## A 4-channel image of size 1x1                ## (B)
print(input)                                                                                                    ## (C)
#                             tensor([[[[ 1.5410]],
#
#                                     [[-0.2934]],
#
#                                     [[-2.1788]],
#
#                                     [[ 0.5684]]]])
print(input.shape)       # (1, 4, 1, 1)                                                                       ## (D)
#
#                           args:   ch_in, ch_out, ker_size, stride, padding                                                    <<< PAY ATTENTION TO ARGS FOR nn.ConvTranspose2d
trans_conop = nn.ConvTranspose2d(4,     2,      4,        1,      0)                                            ## (E)
print(trans_conop.weight)                                                                                       ## (F)
#                             tensor([[[[-0.0157,  0.0468, -0.0534, -0.0347],
#                                       [-0.1689, -0.1171, -0.0729,  0.0065],
#                                       [ 0.0699,  0.1061, -0.1198, -0.0770],
#                                       [ 0.0642,  0.1468, -0.0364,  0.1323]],
#
#                                      [[-0.0285,  0.0187,  0.1601, -0.1640],
#                                       [-0.1113, -0.0448, -0.0689,  0.1527],          As shown in the output in line
#                                       [-0.1146, -0.0814, -0.1235, -0.1656],          (G), the shape of the kernel
#                                       [-0.1032,  0.1520,  0.0789,  0.0857]]],        weights is (4,2,4,4).  We want
#                                                                                      the 4 input channels to go into
#                                                                                      2 channels in the output and this
#                                     [[[ 0.0093, -0.0906,  0.0299, -0.1651],          must be accomplished with 4x4
#                                       [-0.1277, -0.0911,  0.1115,  0.1036],          kernel.   The first two 4x4
#                                       [-0.0784, -0.0064,  0.1131,  0.1757],          matrices shown are for taking the
#                                       [ 0.0702,  0.0239,  0.1185, -0.1041]],         first channel of the input to
#                                                                                      each of the two channels of the
#                                      [[ 0.0329, -0.1371, -0.1225, -0.0913],          output.  The same applies to the
#                                       [ 0.0800,  0.0711, -0.1047,  0.0534],          second and the third 4x4 matrics:
#                                       [ 0.0970, -0.0223,  0.0067,  0.0410],          the first of these will be applied
#                                       [ 0.1097,  0.1697, -0.1362, -0.0648]]],        to the second input channel and
#                                                                                      the result sent to the first
#                                                                                      output channel; and the second
#                                     [[[ 0.0695,  0.1465,  0.1538,  0.1560],          applied to the same second input
#                                       [ 0.0352, -0.1537,  0.0163, -0.1106],          channel but its output sent to the
#                                       [-0.1647,  0.1571,  0.1344, -0.1763],          second output channel. And so on.
#                                       [ 0.0331, -0.0298, -0.0291, -0.0809]],
#
#                                      [[ 0.0680, -0.1047,  0.0648,  0.0894],
#                                       [ 0.1265,  0.0661, -0.1750, -0.1147]
#                                       [ 0.0883,  0.0370, -0.1379, -0.1018],
#                                       [ 0.1663,  0.1191, -0.0771, -0.0445]]],
#
#
#                                     [[[-0.1684, -0.0032, -0.1331, -0.1364],
#                                       [-0.0097,  0.0265, -0.0724,  0.1049],
#                                       [-0.1076,  0.1604,  0.1211, -0.1491],
#                                       [-0.0440,  0.0080,  0.0258,  0.0419]],
#
#                                      [[ 0.0694,  0.0106, -0.0863,  0.0836],
#                                       [-0.1696, -0.1048, -0.0443, -0.0861],
#                                       [-0.0618, -0.1449, -0.0376,  0.0378],
#                                       [-0.1152, -0.0091,  0.1265, -0.0182]]]], requires_grad=True)
print(trans_conop.weight.shape)     #  (4, 2, 4, 4)                                                            ## (G)
x = trans_conop(input)                                                                                          ## (H)
print(x)                                                                                                        ## (I)
#                             tensor([[[[-0.2691, -0.2173, -0.4970, -0.4176],
#                                       [-0.3000,  0.2013, -0.2167,  0.2852],
#                                       [ 0.4334, -0.0808, -0.4369,  0.1342],
#                                       [-0.0138,  0.2935, -0.0079,  0.4394]],         Shown here is the 2-channel
#                                                                                      4x4 output image
#                                      [[-0.1775,  0.2879,  0.0771, -0.3884],
#                                       [-0.5823, -0.3086,  0.2653,  0.4053],
#                                       [-0.4478, -0.2971,  0.0715, -0.0391],
#                                       [-0.6342, -0.0956,  0.3861,  0.2224]]]],
#                                    grad_fn=<ConvTranspose2DBackward>)
print(x.shape)              # (1, 2, 4, 4)                                                                      ## (J)
```

Explaining the Example

- In line (B) of the code shown on the previous slide, the call

```
input = torch.randn((batch_size, 4)).view(-1, 4, 1, 1)
```

  generates from a 4-dimensional noise vector a 1×1 image with 4 channels. To explain: The call `torch.randn(batch_size, 4)` generates a 1×4 array of noise samples that are drawn from a zero-mean and unit-variance Gaussian distribution. As shown in line (A) on the previous slide, the variable `batch_size` is set to 1 for this demonstration. Subsequently, in accordance with the explanation in the last section of my Week 5 slides, the invocation `view(-1, 4, 1, 1)` reshapes the 1×4 noise array into a tensor of shape $(1, 4, 1, 1)$, which can be interpreted as a 4-channel one-pixel image. The first value in the shape $(1, 4, 1, 1)$ is for the batch-size of 1. The output of the code in line (B) is shown in the commented-out block just after line (C).

- The statement in line (E) and what comes after that are explained on the next slide.

Explaining the Example (contd.)

- In line (E) of the example code in Slide 48, we called the constructor of `nn.ConvTranspose2d` class as follows:

```
trans_conop = nn.ConvTranspose2d(4, 2, 4, 1, 0)
```

where the arguments supplied carry the following meanings in the left-to-right order:

```
input channels    =  4
output channels   =  2
kernel size       =  (4,4)
stride            =  (1,1)
padding           =  (0,0)
```

For the last three arguments, if the two values in a tuple are identical, we can supply the argument as a scalar.

- The constructor call shown above means that a $(4, 4)$ kernel will be used for the transpose convolution with our 4-channel 1×1 input image.

Explaining the Example (contd.)

- **VERY IMPORTANT:** As the PyTorch documentation page for `nn.ConvTranspose2d` says, before actually carrying out the transpose convolution operation, the input image is padded with N zeros on "both" sides where N is given by

```
N   =   dilation * (kernel_size - 1) -  padding
```

  where `dilation` is a constructor parameter for the `ConvTranspose2d` class whose default value is 1.

- Shown below are the values for N, which is the number of padding zeros for "both" sides of the input image:

```
        kernel_size  =   (4,4)
           dilation  =   1 [default]
-------------------------------------
    padding     |       N
-------------------------------------
       0        |       3
       1        |       2
       2        |       1
-------------------------------------
```

Explaining the Example (contd.)

- As to what is meant by "both sides" with regard to the padding of the input image with N zeros, shown below are the padded versions of our one-pixel input image for different values for the `padding` constructor parameter:

```
padding = 0:
N = 3
padded version of the 1x1 input_image:
                                        0 0 0 0 0 0 0
                                        0 0 0 0 0 0 0
                                        0 0 0 0 0 0 0
                                        0 0 0 1 0 0 0
                                        0 0 0 0 0 0 0
                                        0 0 0 0 0 0 0
                                        0 0 0 0 0 0 0
padded input_image size:  7x7
output_image size:        4x4

-------------------------------------------------------------

padding = 1
N = 2
padded version of the 1x1 input_image:
                                          0 0 0 0 0
                                          0 0 0 0 0
                                          0 0 1 0 0
                                          0 0 0 0 0
                                          0 0 0 0 0
padded input_image size:  5x5
output_image size:        2x2

-------------------------------------------------------------

padding = 2
N = 1
padded version of the 1x1 input_image:
                                            0 0 0
                                            0 1 0
                                            0 0 0
padded input_image size: 3x3
output_image size:       ERROR
```

Calculating the Size of the Output Image

- Based on what is shown on the previous slide, the reader should now be able to make sense of the output shown in the commented out portion after line (I) on Slide 48. That shows a 2-channel 4×4 output image obtained with the transpose convolution operator invoked with the statement in line (H) on the same slide.

- The reason why the output image is of size 4×4 when `padding=0` is because the padded input-image as shown on the previous slide for this case is of size 7×7. So a 4×4 kernel can be fit at four different locations width-wise and height-wise as the kernel sweeps over the padded input image. And that gives us a 4×4 output.

- The discussion so far has been on explaining the code as shown on Slide 48. As the reader knows, we used `padding=0` in that example. In the next slide, let's see what happens when we use `padding=1` and `padding=2`.

Calculating the Size of the Output Image (Contd.)

- As you know, using `padding` in a convo layer increases the size of the output. So you would expect that using `padding` would have the *opposite* effect for transpose convo. That's exactly what happens.
- Let's say we want to use `padding=1` in the previous example. Now the constructor call for the `nn.ConvTranspose2d` class would need to be:

```
trans_conop = nn.ConvTranspose2d(4, 2, 4, 1, 1)
```

- In this case, as shown in the middle in Slide 52, the padded-input image will be of size 5×5. That would allow for only two different shifts for the 4×4 kernel for the production of the output. Therefore, in this case the size of the output image would of size 2×2.
- On the other hand, if we had set the `padding` parameter to 2, the padded input image would like what is shown at the bottom on the previous slide. This is too small for a 4×4 kernel to fit in. So in this case, we get an error message from PyTorch.

Outline

The `stride` Parameter for Upsampling

- The discussion in the previous section was all based on `stride=1`. However, for most applications of the `nn.ConvTranspose2d` class, you are likely to use the `stride` parameter for a bulk of the upsampling effect. Whereas the kernel size has mostly an additive effect on the size of the output, the stride parameter has a multiplicative effect.
- Assume that the stride along both image axes is the same and set to s. Assume the same for the kernel and let its size be denoted k.
- If H_{in} and W_{in} denote the height and the width of the image at the input to a transpose convolution operator and H_{out} and W_{out} the same at the output, the relationship between the sizes at the input and the output is given by

$$\begin{aligned} H_{out} &= (H_{in} - 1) * s + k \\ W_{out} &= (W_{in} - 1) * s + k \end{aligned}$$

where I have also assumed that you are using the default choices for padding ($= 0$), for output_padding ($= 0$), and for dilation ($= 1$).

An Upsampling Example with Stride

- The next slide shows an example of upsampling achieved with a stride of size 2×2 and a kernel of size 3×3 on an input image of size 4×4.
- In line (A), we set the batch size to 1. With its first argument set to 1, the call in line (B) generates a random-number vector of size 32 that is then shaped into a 2-channel 4×4 image. We want to feed this image into a `nn.ConvTranspose2d` operator and check the output image for its size.
- The `nn.ConvTranspose2d` operator in line (E) has its stride set to 2×2 and its kernel set to be of size 3×3. The formulas shown on the previous slide say that this should result in the output image produced by the operator to be of size 9×9. Note that the operator in line (E) also sets the number of output channels to 4. This is borne out by the output displayed for the call to the operator in line (F) and for the shape of the output displayed in line (G).

Example of Upsampling with Stride

```
import torch
import torch.nn as nn
torch.manual_seed(0)

batch_size = 1                                                                                                    ## (A)
input = torch.randn((batch_size, 32)).view(-1, 2, 4, 4)                                                           ## (B)
print("\n\n\nThe 32-dimensional noise vector shaped as an 2-channel 4x4 image:\n\n")
print(input)                                                                                                      ## (C)
#    tensor([[[[ 0.7298, -1.8453, -0.0250,  1.3694],
#              [ 2.6570,  0.9851, -0.2596,  0.1183],
#              [-1.1428,  0.0376,  2.6963,  1.2358],
#              [ 0.5428,  0.5255,  0.1922, -0.7722]],
#
#             [[ 1.6268,  0.1723, -1.6115, -0.4794],
#              [-0.1434, -0.3173,  0.9671, -0.9911],
#              [ 0.5436,  0.0788,  0.8629, -0.0195],
#              [ 0.9910, -0.7777,  0.3140,  0.2133]]]])
#
print(input.shape)                            ##  (1, 2, 4, 4)                                                    ## (D)

#                               ch_in  ch_out  kernel  stride  padding
trans_conop = nn.ConvTranspose2d(  2,     4,      3,      2,      0    )                                          ## (E)

x = trans_conop(input)                                                                                            ## (F)

print(x)                                                                                                          ## (G)

#    tensor([[[[-4.4152e-02,  1.2544e-01, -1.2656e-01,  9.6014e-02, -2.2811e-01, -2.7850e-01,  7.4894e-02, -2.1415e-01,  1.2354e-01],
#              [ 2.4731e-02, -1.6544e-01, -1.3804e-01,  2.2369e-01, -2.8944e-01, -4.7890e-02,  6.9339e-02, -2.5903e-01,  1.0654e-01],
#              [ 3.6436e-01, -3.7065e-01,  6.3544e-01, -3.4491e-02, -2.9987e-01,  1.6315e-01,  4.4707e-03, -2.8466e-01,  8.7493e-02],
#              [ 1.0370e-01, -4.5312e-01,  2.8402e-01, -2.0143e-01,  6.5421e-02, -1.5800e-02, -1.2667e-01, -7.0222e-02, -8.5209e-03],
#              [-3.0782e-01, -9.1773e-02,  1.3661e-01, -1.0635e-01,  4.5713e-01, -1.3314e-01,  5.2770e-01, -1.1149e-01,  1.4809e-01],
#              [-1.1116e-01,  1.1756e-01, -1.8814e-01, -5.9404e-02,  1.2867e-01, -4.6027e-01,  2.7642e-01, -2.3950e-01,  7.7300e-02],
#              [-1.1877e-04,  1.1310e-01,  5.9378e-02, -2.0451e-01, -4.0972e-02, -2.5601e-01,  2.0241e-01, -6.2370e-02,  7.7144e-03],
#              [ 2.7385e-05, -1.3653e-01, -1.3757e-02, -1.3173e-01,  4.5617e-02, -8.2951e-02, -8.5837e-02,  6.2223e-02, -1.4210e-01],
#              [-4.3369e-02, -1.3176e-01, -8.8378e-03, -6.3154e-02,  2.0169e-02, -7.9904e-02, -9.0935e-03, -4.3222e-03, -1.5681e-01]],
#
#             [[-6.3428e-02, -1.9877e-01,  8.4883e-02, -5.5194e-02,  1.8710e-02,  2.0991e-01, -5.2714e-02,  8.7261e-02, -5.6352e-03],
#              [ 3.0607e-01, -1.8582e-01, -1.5720e-01, -1.0889e-01, -3.6943e-01,  2.1707e-01,  1.1601e-01,  1.2883e-01,  1.1567e-01],
#              [-4.9457e-02,  1.3517e-01, -3.1747e-01,  1.4358e-01, -1.5253e-01, -2.2648e-01, -1.1735e-01,  4.7839e-02,  3.2453e-02],
#              [ 2.9721e-01,  1.4327e-01,  2.8931e-01,  8.9015e-02,  1.8221e-01, -1.4263e-01, -1.5140e-01,  1.3982e-01,  2.1605e-02],
#              [ 2.9850e-01, -2.0616e-01, -9.8867e-02, -6.7920e-02, -2.4457e-01,  4.6328e-03, -4.7882e-01, -3.9504e-02, -4.9977e-02],
#              [-6.2585e-02, -1.2643e-01, -8.2678e-02, -8.6435e-03,  4.3937e-01,  9.0001e-03,  3.5033e-01,  6.0396e-02,  9.9465e-02],
#              [-2.9405e-02, -4.0695e-02, -1.3474e-01,  1.1429e-01,  3.3068e-01, -9.3352e-02,  5.3747e-02, -9.2056e-02, -3.4831e-02],
#              [ 1.9825e-01, -1.0853e-01, -1.0243e-02,  1.2991e-01,  1.1680e-01, -3.3269e-02, -5.1290e-02, -6.4512e-02, -6.4124e-02],
#              [ 1.9561e-01,  3.8178e-02, -1.1138e-01, -6.7620e-02,  4.1267e-02,  1.1471e-02, -1.5264e-02,  4.4058e-02,  2.8338e-02]],
#
#             [[-2.4940e-01,  2.8603e-01,  2.1782e-01, -6.6619e-02,  5.7356e-02, -4.9506e-02,  1.5497e-01,  1.6988e-01,  3.2393e-01],
#              [ 9.2061e-02, -9.0965e-02, -1.6751e-01,  2.9311e-01,  1.4122e-01,  1.7886e-01,  3.0424e-01, -4.8374e-02,  1.0782e-01],
#              [-3.4120e-01,  4.7060e-01,  5.4504e-01,  3.4156e-01,  3.3260e-01, -7.7028e-02, -1.2933e-01, -1.7688e-01,  7.6152e-02],
#              [ 3.9559e-01, -2.1855e-01,  2.3644e-01, -1.2267e-02,  2.5593e-02,  6.2246e-02,  1.2039e-01,  1.2731e-01,  1.1114e-01],
#              [-1.4819e-01, -2.3600e-01, -5.2384e-01, -3.8295e-02, -4.8354e-01,  5.5731e-01,  2.0288e-01,  4.8694e-02,  2.1953e-01],
#              [-7.0234e-02,  1.8975e-01,  6.4450e-02,  7.6979e-02,  3.5079e-01, -2.7943e-01,  2.2640e-01, -5.8386e-02,  9.5471e-02],
#              [ 3.3609e-03,  3.9622e-01,  1.9816e-01,  7.6436e-02, -1.4130e-01,  3.2905e-04, -2.3993e-01, -8.2371e-02, -1.6212e-01],
#              [ 1.0082e-01, -3.3432e-02,  1.6210e-01,  6.7511e-02,  1.1597e-01,  4.5647e-02, -1.8666e-02,  1.6465e-01,  7.4799e-02],
#              [ 2.3834e-02,  1.7224e-01, -9.4561e-03, -6.9326e-02, -3.4250e-03,  1.1134e-01,  1.6478e-01,  1.8597e-01,  1.6523e-01]],
#
#             [[ 1.2324e-01, -6.7804e-02, -4.1464e-01,  3.9657e-02, -4.1462e-01, -3.2299e-01,  1.5135e-02, -3.4607e-01, -1.1564e-01],
#              [-3.4457e-02, -2.7444e-01, -3.6810e-01, -3.7973e-01, -6.1502e-02,  2.5991e-02, -7.0435e-02,  4.2427e-02, -2.9984e-01],
#              [-3.4998e-02, -4.2079e-01,  3.6114e-01, -1.0331e-01, -2.7160e-01, -1.2306e-01, -7.8796e-01, -4.3289e-01,  4.2614e-02],
#              [-1.5327e-01,  1.4117e-01, -5.0262e-01, -1.7961e-02, -1.9241e-01, -3.0302e-01, -2.7004e-01, -3.1703e-02, -1.0608e-01],
#              [-4.5616e-01, -2.5729e-01, -3.6639e-02, -2.6968e-01,  2.6512e-01, -2.6265e-01, -1.6591e-01, -3.5190e-01, -2.7374e-01],
#              [-1.2856e-01, -3.4800e-01, -4.9452e-02, -1.6643e-01, -8.8080e-02,  2.6840e-02, -5.5462e-01, -2.6119e-02, -3.1550e-01],
#              [ 1.7508e-01,  2.6454e-02, -4.1953e-01, -2.9197e-01, -2.7528e-01, -3.6171e-01,  7.2971e-02, -1.8341e-01, -3.9131e-02],
#              [-8.3295e-02, -2.1962e-01, -3.5403e-01, -1.3091e-02, -1.4763e-01, -1.7753e-01, -1.9705e-01, -2.6927e-01, -7.8970e-02],
#              [-1.7471e-01, -1.5918e-01, -7.2195e-02, -2.5484e-01, -2.0372e-01, -1.6250e-01, -6.8606e-04, -7.4715e-02, -2.2458e-01]]]],
#          grad_fn=<ConvTranspose2DBackward>)
#
print(x.shape)                                ##   (1, 4, 9, 9)                                                 ## (H)
```

Outline

Revisiting `padding` and `stride` Parameters

- As mentioned at the beginning of the previous section, the `stride` parameter has a multiplicative effect on the size of the output image in relation to that of the input image. To fine-tune the output-size enlargement achieved with `stride`, you can use the `padding` parameter when constructing an instance of the `nn.ConvTranspose2d` operator.

- Recall that whereas the `stride` parameter in the `nn.Conv2d` constructor causes an input image to be downsampled while the image is subject to a convolutional operation, the effect of the same parameter in the `nn.ConvTranspose2d` constructor is exactly the opposite, in the sense that it causes the input image to be upsampled.

- The same thing is true for the `padding` parameter. For the `nn.Conv2d` case, it causes the input image to be enlarged by the extent of padding on both sides. However, for the `nn.ConvTranspose2d` case, it shrinks the input image correspondingly.

Using `stride` and `padding` at The Same Time

- As in the previous section, assume that the stride along both image axes is the same and set to s and that the same is true of the kernel size that we represent by k. Continuing with this assumption, let the padding along both axes also be the same and let's represent it by p.
- If H_{in} and W_{in} denote the height and the width of the image at the input to a transpose convolution operator and H_{out} and W_{out} the same at the output, the relationship between the sizes at the input and the output is given by

$$\begin{aligned} H_{out} &= (H_{in} - 1) * s + k - 2 * p \\ W_{out} &= (W_{in} - 1) * s + k - 2 * p \end{aligned}$$

  where I have also assumed that you are using the default choices for output_padding $(= 0)$, and for dilation $(= 1)$.
- Shown on the next slide is an example similar to the one on Slide 58, except that now we also include padding $(= 1)$ in line (E), with causes the size of the output image to change from 9×9 to 7×7 as shown in line (H).

Example of Upsampling with Stride and Padding

```
import torch
import torch.nn as nn
torch.manual_seed(0)

batch_size = 1                                                                                                    ## (A)
input = torch.randn((batch_size, 32)).view(-1, 2, 4, 4)                                                           ## (B)
print("\n\n\nThe 32-dimensional noise vector shaped as an 2-channel 4x4 image:\n\n")
print(input)                                                                                                      ## (C)
#    tensor([[[[ 0.5769, -0.1692,  1.1887, -0.1575],
#              [-0.0455,  0.6485,  0.5239,  0.2180],
#              [ 1.9029,  0.3904,  0.0331, -1.0234],
#              [ 0.7335,  1.1177,  2.1494, -0.9088]],
#
#             [[-0.1434, -0.1947,  1.4903, -0.7005],
#              [ 0.1806,  1.3615,  2.0372,  0.6430],
#              [ 1.6953,  2.0655,  0.2578, -0.5650],
#              [ 0.9278,  0.4826, -0.8298,  1.2678]]]])

print(input.shape)                            ##  (1, 2, 4, 4)                                                    ## (D)

#                               ch_in  ch_out  kernel  stride  padding
trans_conop = nn.ConvTranspose2d(  2,     4,      3,      2,      1   )                                           ## (E)

x = trans_conop(input)                                                                                            ## (F)

print(x)                                                                                                          ## (G)

#    tensor([[[[-1.4602e-01,  3.4987e-02, -4.9057e-02, -2.1430e-01, -1.1586e-01,  1.3547e-01, -8.5486e-02],
#              [ 3.6219e-02, -2.1292e-02,  9.6213e-02,  3.3451e-02,  3.8808e-01, -4.7209e-01, -1.2999e-02],
#              [-3.9848e-02, -1.3260e-01, -5.1958e-02, -9.8983e-03,  1.1389e-02,  1.4386e-02, -4.3486e-02],
#              [ 2.1806e-01, -5.7146e-02,  3.0747e-01, -3.9247e-01,  8.6604e-02, -3.7835e-01, -1.2648e-01],
#              [-1.9795e-01,  1.7612e-01,  3.1326e-02,  1.7133e-02, -4.5113e-02,  6.3181e-02,  4.0513e-02],
#              [ 3.7797e-01, -4.1503e-01,  1.4547e-01, -4.7569e-01, -3.7961e-02,  5.1364e-02, -1.1437e-01],
#              [-9.3304e-02, -7.3418e-02, -1.7575e-01, -1.4605e-01, -4.0518e-01,  3.2027e-01,  1.5138e-01]],
#
#             [[ 1.5954e-01,  1.4756e-01,  1.1634e-01,  1.3148e-01,  2.3051e-01,  2.5622e-01,  1.0570e-01],
#              [ 1.5098e-01,  1.8500e-01,  2.9588e-01,  3.0268e-01,  9.4988e-02,  1.7306e-01,  2.9879e-01],
#              [ 1.3169e-01,  2.0943e-01,  1.9719e-01,  4.1675e-01,  2.0525e-01,  3.5726e-01,  1.5688e-01],
#              [ 2.3013e-01,  2.8313e-01,  1.4138e-01,  1.3809e-02, -1.4228e-01, -2.8762e-01,  7.0353e-03],
#              [ 2.7535e-01,  5.2515e-01,  1.9836e-01,  3.3347e-01,  1.3785e-01,  2.0058e-01,  5.9918e-02],
#              [-9.6119e-02, -2.8195e-01, -1.5659e-01, -6.8087e-02, -8.4310e-02, -2.9354e-02,  4.2759e-01],
#              [ 1.9229e-01,  1.7491e-01,  2.0401e-01, -7.2777e-02,  2.3285e-01,  3.9534e-01,  1.0732e-01]],
#
#             [[-9.4420e-02, -3.9821e-02, -4.0897e-02, -3.6939e-02,  6.1995e-02,  2.1703e-01, -1.0549e-01],
#              [-1.5969e-03,  2.5784e-01,  1.9578e-03,  3.7270e-01,  3.5862e-01, -2.0336e-01, -8.5515e-02],
#              [-3.6023e-03,  5.1168e-02,  8.9228e-02,  3.1974e-01,  1.8428e-01,  3.5481e-01,  3.3411e-02],
#              [-6.5154e-02,  4.2694e-01,  3.4081e-01,  2.6115e-02,  2.7636e-01, -2.5520e-01,  1.2862e-01],
#              [ 3.0369e-02,  4.2223e-01,  1.9858e-01,  3.4078e-01, -1.9767e-04,  3.4468e-02, -1.8808e-02],
#              [ 2.6914e-01,  3.8698e-01,  1.6409e-01,  6.9061e-02, -3.4989e-01,  3.1591e-01,  1.0464e-01],
#              [ 2.7803e-02,  1.1538e-01, -5.9130e-02, -9.0690e-02, -3.0726e-01,  2.9721e-02,  2.0266e-01]],
#
#             [[ 3.1302e-02, -1.7105e-02,  3.9255e-02,  1.2681e-03,  2.6324e-01,  8.2351e-02, -3.6774e-02],
#              [ 1.3670e-01, -1.3665e-01,  2.7974e-01,  1.5993e-01,  2.7132e-01,  1.6712e-01,  2.2299e-01],
#              [ 9.2896e-02, -8.1275e-03,  2.5529e-01, -8.3837e-02,  3.5914e-01,  1.0724e-01,  1.5666e-01],
#              [ 3.2818e-01,  3.0762e-01,  2.5367e-01,  4.6363e-01, -8.1398e-02,  3.6939e-01, -9.5386e-02],
#              [ 2.7898e-01, -2.4000e-01,  3.6618e-01,  1.5419e-01,  1.0282e-01,  7.0281e-02,  1.6960e-03],
#              [ 1.4526e-01,  1.8005e-01, -3.1231e-02,  6.9455e-02, -1.4039e-02,  1.6355e-01,  2.1294e-01],
#              [ 1.8852e-01,  6.4715e-02,  1.1374e-01,  1.7605e-01, -1.0454e-01, -4.8272e-01,  2.7392e-01]]]],
#          grad_fn=<ConvTranspose2DBackward>)
#
print(x.shape)                                ##   (1, 4, 7, 7)                                                 ## (H)
```

Outline

The Two Building Blocks for the mUnet Network

- That brings us to DLStudio's inner class SemanticSegmentation. Its workhorse for semantic segmentation is called mUnet that I will be presenting in the next section. In this section, I'm going to focus on the two building-block classes used in mUnet.
- The two building-blocks are the network classes SkipBlockDN and SkipBlockUP, the former for the Encoder in which the size of the image becomes progressively smaller as the data abstraction level becomes higher and the latter for the Decoder whose job is to map the abstractions produced by the Encoder back to the pixel level.
- The network SkipBlockDN used in the Encoder is based on regular convolutional layers that when used with strides will give us a progression of reduced resolution representations for an image.
- And the network SkipBlockUP for the Decoder is based on transpose convolutions that will gradually increase the resolution in the output of the Encoder until it is at the same level as the original image.

The Two Building Blocks for mUnet (contd.)

- In the code that I'll show for `SkipBlockDN` and `SkipBlockUP` in the next two slides and later for the class `mUnet` in Slides 70 and 71, do not be confused by two different types of usages of the skip connections. The skip connections that you have already learned in my Week 6 lecture are used within each of the network classes `SkipBlockDN` and `SkipBlockUP`.
- In addition, I will also use skip connections between these two network classes, as you will see later when I show you the code for `mUnet`.
- That is, as you will see on Slides 70 and 71, `mUnet` also includes skip connections as shortcuts between the corresponding levels of abstractions in the encoder and the decoder. Those skip connections are critical to the operation of the encoder-decoder combo if the goal is to see pixel-level semantic segmentation results.

SkipBlockDN — The Definition

```
class SkipBlockDN(nn.Module):
    def __init__(self, in_ch, out_ch, downsample=False, skip_connections=True):
        super(DLStudio.SemanticSegmentation.SkipBlockDN, self).__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, stride=2)
    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = out + torch.cat((identity, identity), dim=1)
        return out
```

SkipBlockUP — The Definition

```
class SkipBlockUP(nn.Module):
    def __init__(self, in_ch, out_ch, upsample=False, skip_connections=True):
        super(DLStudio.SemanticSegmentation.SkipBlockUP, self).__init__()
        super(SkipBlockUP, self).__init__()
        self.upsample = upsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convoT1 = nn.ConvTranspose2d(in_ch, out_ch, 3, padding=1)
        self.convoT2 = nn.ConvTranspose2d(in_ch, out_ch, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if upsample:
            self.upsampler = nn.ConvTranspose2d(in_ch, out_ch, 1, stride=2, dilation=2, output_padding=1, paddi
    def forward(self, x):
        identity = x
        out = self.convoT1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        out  =  nn.ReLU(inplace=False)(out)
        if self.in_ch == self.out_ch:
            out = self.convoT2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.upsample:
            out = self.upsampler(out)
            identity = self.upsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
                out = out + identity
            else:
                out = out + identity[:,self.out_ch:,:,:]
        return out
```

Outline

The Architecture of mUnet

- This network is called `mUnet` because it is intended for segmenting out multiple objects simultaneously form an image.
- Shown on the next slide is the implementation code for `mUnet` that is in the inner class `SemanticSegmentation` of the DLStudio platform.
- As is the case with all such classes, the constructor of the class `mUnet` defines the components elements that are subsequently used in the definition of `forward()` to create a network.
- Note how I save in lines labeled (A), (B), (C) a portion of the data that is passing through the different levels of the encoder. Also note how I subsequently mix in this saved data at the corresponding levels in the decoder in lines (D), (E), and (F) of the code.

mUnet – The Definition

```
class mUnet(nn.Module):
    """
    Class Path:   DLStudio  ->  SemanticSegmentation  ->  mUnet
    """
    def __init__(self, skip_connections=True, depth=16):
        super(DLStudio.SemanticSegmentation.mUnet, self).__init__()
        self.depth = depth // 2
        self.conv_in = nn.Conv2d(3, 64, 3, padding=1)
        ##  For the DN arm of the U:
        self.bn1DN  = nn.BatchNorm2d(64)
        self.bn2DN  = nn.BatchNorm2d(128)
        self.skip64DN_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(64, 64, skip_connections=skip_connections))
        self.skip64dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 64,   downsample=True, skip_connections=skip_connections)
        self.skip64to128DN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 128, skip_connections=skip_connections )
        self.skip128DN_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(128, 128, skip_connections=skip_connections))
        self.skip128dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(128,128, downsample=True, skip_connections=skip_connections)
        ##  For the UP arm of the U:
        self.bn1UP  = nn.BatchNorm2d(128)
        self.bn2UP  = nn.BatchNorm2d(64)
        self.skip64UP_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(64, 64, skip_connections=skip_connections))
        self.skip64usUP = DLStudio.SemanticSegmentation.SkipBlockUP(64, 64, upsample=True, skip_connections=skip_connections)
        self.skip128to64UP = DLStudio.SemanticSegmentation.SkipBlockUP(128, 64, skip_connections=skip_connections )
        self.skip128UP_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(128, 128, skip_connections=skip_connections))
        self.skip128usUP = DLStudio.SemanticSegmentation.SkipBlockUP(128,128, upsample=True, skip_connections=skip_connections)
        self.conv_out = nn.ConvTranspose2d(64, 5, 3, stride=2,dilation=2,output_padding=1,padding=2)

    def forward(self, x):
        ##  Going down to the bottom of the U:
        x = nn.MaxPool2d(2,2)(nn.functional.relu(self.conv_in(x)))
        for i,skip64 in enumerate(self.skip64DN_arr[:self.depth//4]):
            x = skip64(x)
        num_channels_to_save1 = x.shape[1] // 2
        save_for_upside_1 = x[:,:num_channels_to_save1,:,:].clone()                                    ## (A)
        x = self.skip64dsDN(x)
        for i,skip64 in enumerate(self.skip64DN_arr[self.depth//4:]):
            x = skip64(x)
        x = self.bn1DN(x)
        num_channels_to_save2 = x.shape[1] // 2
        save_for_upside_2 = x[:,:num_channels_to_save2,:,:].clone()                                    ## (B)
        x = self.skip64to128DN(x)
        for i,skip128 in enumerate(self.skip128DN_arr[:self.depth//4]):
            x = skip128(x)
```

(Continued on the next slide .....)

mUnet – The Definition (contd.)

(..... continued from the previous slide)

```
x = self.bn2DN(x)
num_channels_to_save3 = x.shape[1] // 2
save_for_upside_3 = x[:,:num_channels_to_save3,:,:].clone()                    ## (C)
for i,skip128 in enumerate(self.skip128DN_arr[self.depth//4:]):
    x = skip128(x)
x = self.skip128dsDN(x)
## Coming up from the bottom of U on the other side:
x = self.skip128usUP(x)
for i,skip128 in enumerate(self.skip128UP_arr[:self.depth//4]):
    x = skip128(x)
x[:,:num_channels_to_save3,:,:] =  save_for_upside_3                           ## (D)
x = self.bn1UP(x)
for i,skip128 in enumerate(self.skip128UP_arr[:self.depth//4]):
    x = skip128(x)
x = self.skip128to64UP(x)
for i,skip64 in enumerate(self.skip64UP_arr[self.depth//4:]):
    x = skip64(x)
x[:,:num_channels_to_save2,:,:] =  save_for_upside_2                           ## (E)
x = self.bn2UP(x)
x = self.skip64usUP(x)
for i,skip64 in enumerate(self.skip64UP_arr[:self.depth//4]):
    x = skip64(x)
x[:,:num_channels_to_save1,:,:] =  save_for_upside_1                           ## (F)
x = self.conv_out(x)
return x
```

Outline

The PurdueShapes5MultiObject Dataset

- I have created an annotated dataset, `PurdueShapes5MultiObject`, for the training and testing of networks for semantic segmentation — **especially if the goal is to segment out multiple objects simultaneously**. **Each image in the dataset is of size** 64×64**.**

- The program that generates the `PurdueShapes5MultiObject` dataset is an extension of the program that generated the `PurdueShapes5` dataset you saw earlier in the context of object detection and localization. Whereas each image in the `PurdueShapes5` dataset contained a single randomly scaled and randomly oriented object at some random location, each image in the `PurdueShapes5MultiObject` can contain a random number of up to five objects.

- The images in the `PurdueShapes5MultiObject` dataset come with two annotations, the masks for each of the objects and the bounding boxes. Even when the objects are overlapping in the images, the masks remain distinct in a mask array whose shape is $(5, 64, 64)$.

Some Example Images from the PurdueShapes5MultiObject Dataset

- The next four slides show some example images from the `PurdueShapes5MultiObject` dataset.
- Note in particular that each image comes with a multi-valued mask for the object it contains. The gray values assigned to the objects in the mask are according to the following rule:

```
rectangle  :   50          oval      :  200
triangle   :  100          star      :  250
disk       :  150
```

- A most important aspect of `mUnet` learning is that even when the objects are overlapping in the input RGB image, the 5-layer deep mask array associated with the input image is "aware" of the full extent of each object in the input image. The mask array in each object-type-specific channel at the input is used as the target for the prediction in the corresponding channel of the output for calculating the loss.

Example Images from the PurdueShapes5MultiObject Dataset (contd.)

Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Example Images from the PurdueShapes5MultiObject Dataset (contd.)

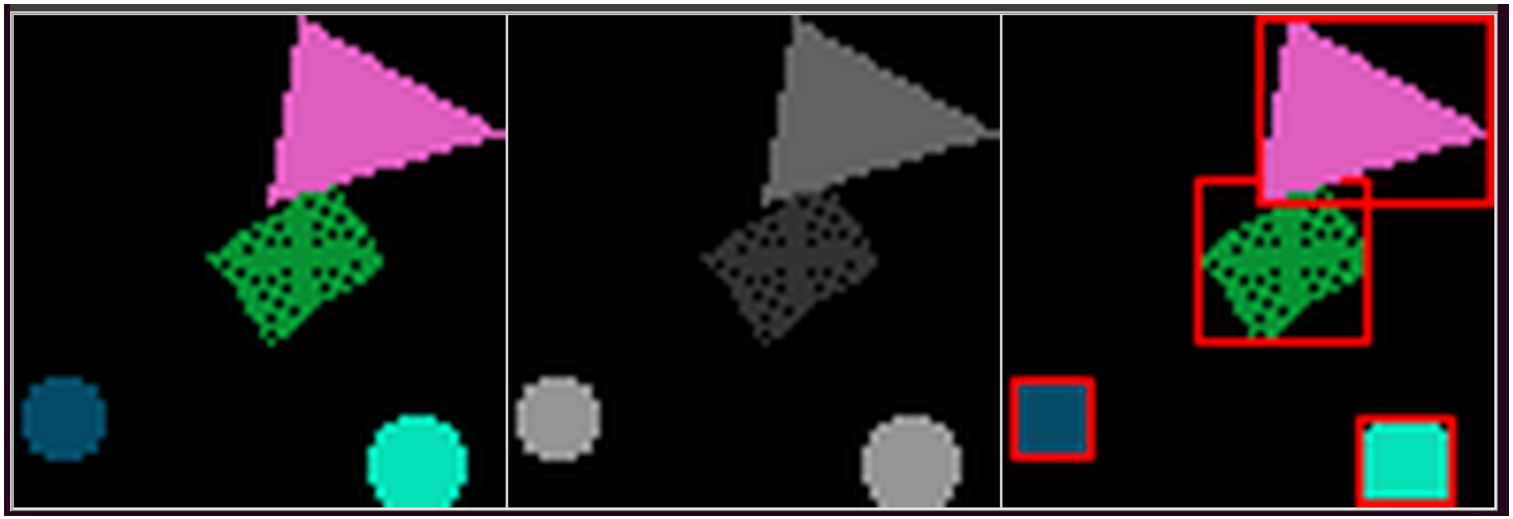


Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

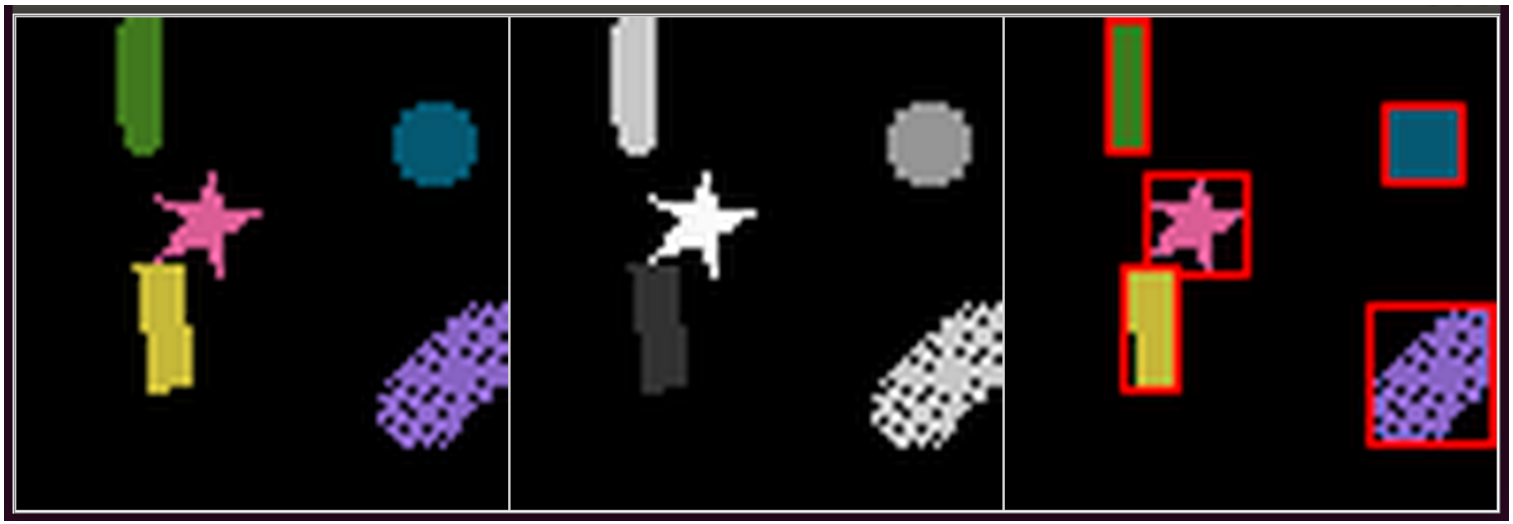


Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Example Images from the PurdueShapes5MultiObject Dataset (contd.)

Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Example Images from the PurdueShapes5MultiObject Dataset (contd.)

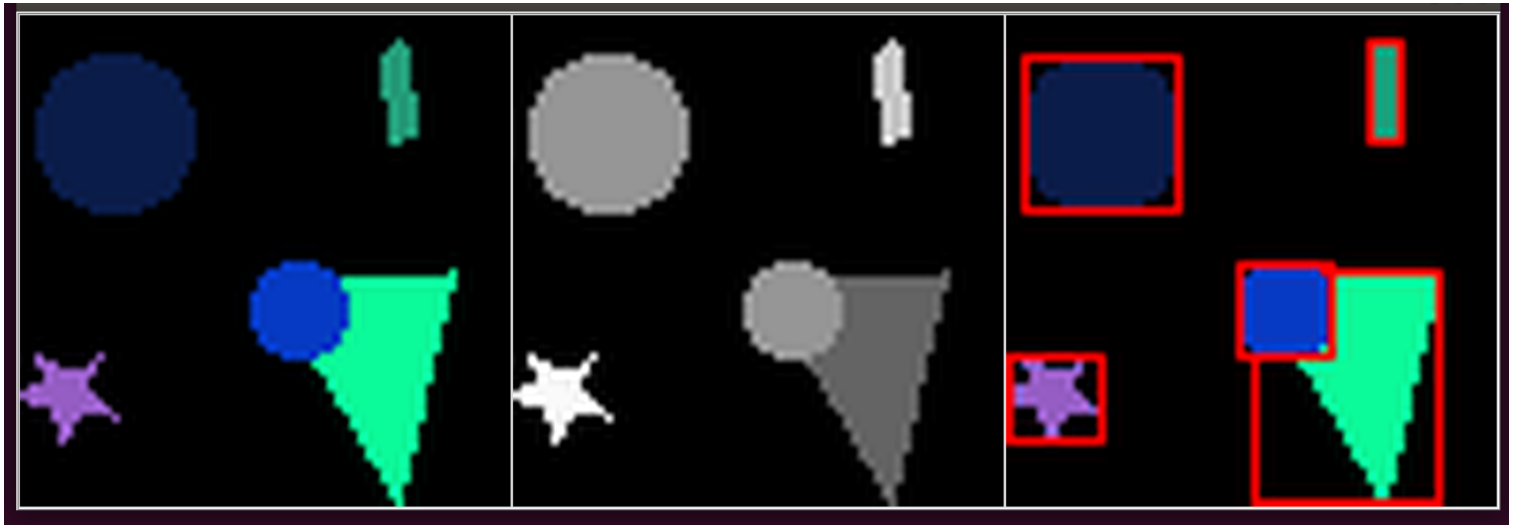


Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

Downloading the PurdueShapes5MultiObject Dataset

- This dataset is available through the link "Download the image datasets for the main DLStudio module" at the main website for DLStudio. After you have unpacked the top-level archive, you will see the following dataset archive files there:
  - PurdueShapes5MultiObject-10000-train.gz
  - PurdueShapes5MultiObject-1000-test.gz
  - PurdueShapes5MultiObject-20-train.gz
  - PurdueShapes5MultiObject-20-test.gz
- You will find the two smaller datasets, with just 20 images each, useful for debugging your code. You would want your own training and the evaluation scripts to run without problems on these two datasets before you let them loose on the larger datasets.

Data Format Used for the PurdueShapes5MultiObject Dataset

- Each 64×64 image in the dataset is stored using the following format:

```
Image stored as the list:
                  [R, G, B, mask_array, mask_val_to_bbox_map]
where

R     :   is a 4096 element list of int values for the red component
          of the color at all the pixels
B     :   the same as above but for the blue component of the color
G     :   the same as above but for the green component of the color

mask_array : is an array whose shape is (5,64,64). Each 64x64 "plane"
             in the array is a mask corresponding to the first index
             for that plane.  The mask assignments are as follows:

                mask_array[0]   :  rectangle      [val stored:  50]
                mask_array[1]   :  triangle       [val stored: 100]
                mask_array[2]   :  disk           [val stored: 150]
                mask_array[3]   :  oval           [val stored: 200]
                mask_array[4]   :  star           [val stored: 250]

             The values stored in the masks are different for the
             different shapes as shown in the third column above.

mask_val_to_bbox_map :  is a dictionary that tells us what bounding-box
             rectangle to associate with each shape in the image.  To
             illustrate what this dictionary looks like, assume that an image
             contains only one rectangle and only one disk, the dictionary
             in this case will look like:

             mask values to bbox mappings:  {200: [],
                                             250: [],
                                             100: [],
                                              50: [[56, 20, 63, 25]],
                                             150: [[37, 41, 55, 59]]}

             Should there happen to be two rectangles in the same image,
             the dictionary would then be like:

             mask values to bbox mappings:  {200: [],
                                             250: [],
                                             100: [],
                                              50: [[56, 20, 63, 25], [18, 16, 32, 36]],
                                             150: [[37, 41, 55, 59]]}
```

Outline

Training mUnet with MSE Loss

- Shown below are the training loss values calculated in the script `semantic_segmentation.py` in the `Examples` directory of the distro. I executed the script with the batch size set to 4, number of epochs to 6, the learning rate to 10^{-4}, and the momentum to 0.9.

```
[epoch:1,batch: 1000]  MSE loss: 453.626
[epoch:1,batch: 2000]  MSE loss: 445.210
[epoch:2,batch: 1000]  MSE loss: 426.408
[epoch:2,batch: 2000]  MSE loss: 414.780
[epoch:3,batch: 1000]  MSE loss: 413.198
[epoch:3,batch: 2000]  MSE loss: 398.390
[epoch:4,batch: 1000]  MSE loss: 396.653
[epoch:4,batch: 2000]  MSE loss: 389.081
[epoch:5,batch: 1000]  MSE loss: 388.792
[epoch:5,batch: 2000]  MSE loss: 387.379
[epoch:6,batch: 1000]  MSE loss: 386.174
[epoch:6,batch: 2000]  MSE loss: 381.055
```

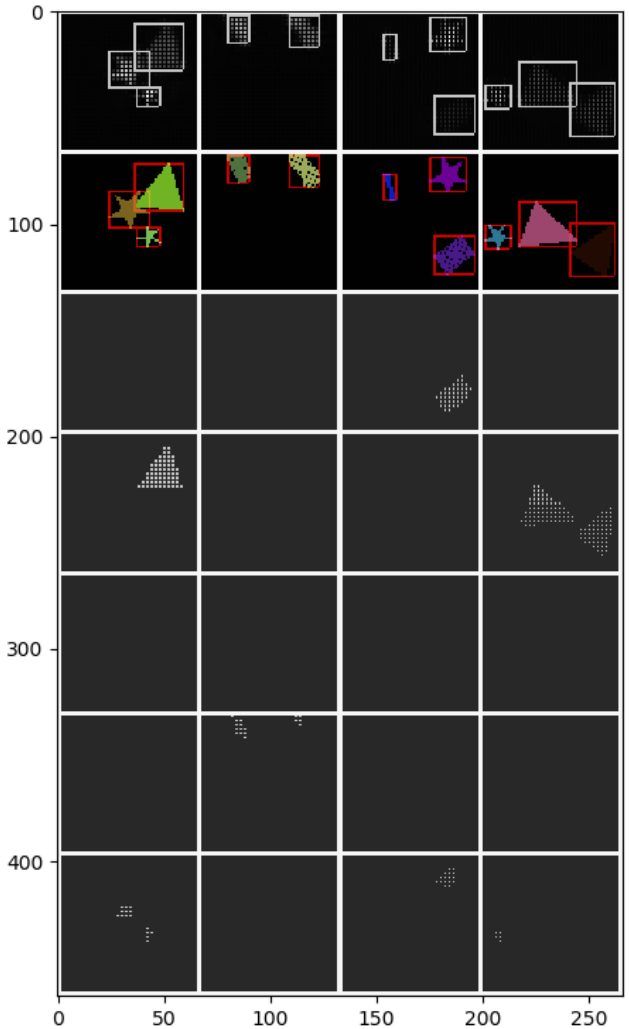
- What's interesting is that if you print out the MSE loss for several iterations when the training has just started, you will see loss numbers much smaller than what are shown above. However, it is easy to verify that those represent states of the network in which it has overfitted to the training data. **A classic example of overfitting is low training error but large error on unseen data.**

Outline

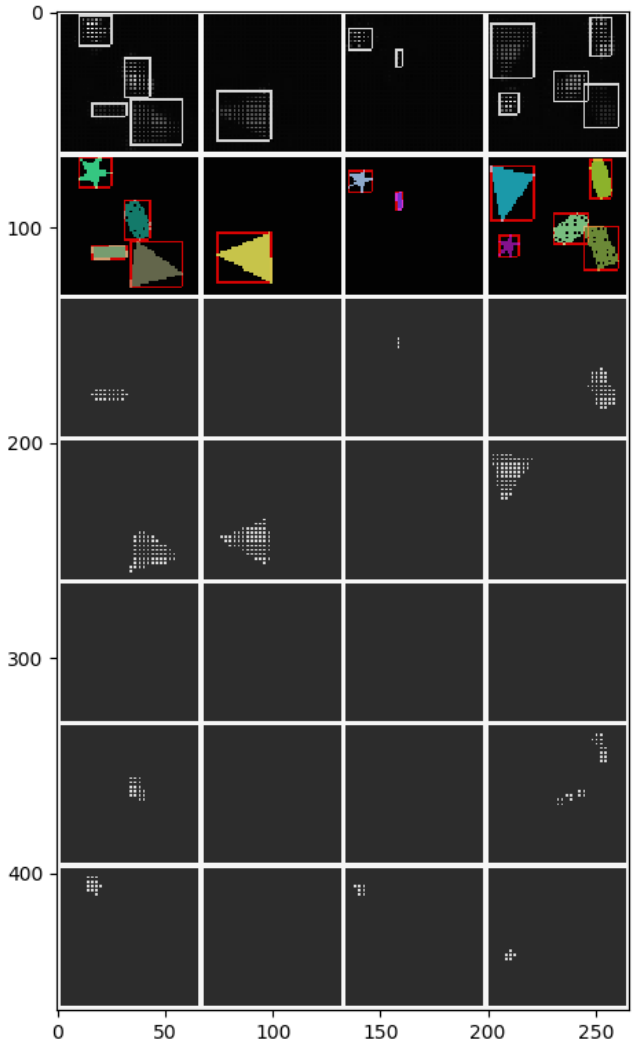
Results on Unseen Test Data

- As you can infer from the archive names shown on Slide 79, the `PurdueShapes5MultiObject` dataset comes with a test dataset that contain 1000 instances of multi-object images produced by the same randomizing program that generates the training dataset.
- Shown in the next three slides are the some typical results on this unseen test dataset. You can see these results for yourself by executing the script `semantic_segmentation.py` in the `Examples` directory of the distribution.
- The top row in each display is the combined result from all the five output channels. The bounding boxes from the dataset are simply overlaid into this composite output. The second row shows the input to the network. The semantic segmentations are shown in the lower five rows.

Results on Unseen Test Data (contd.)

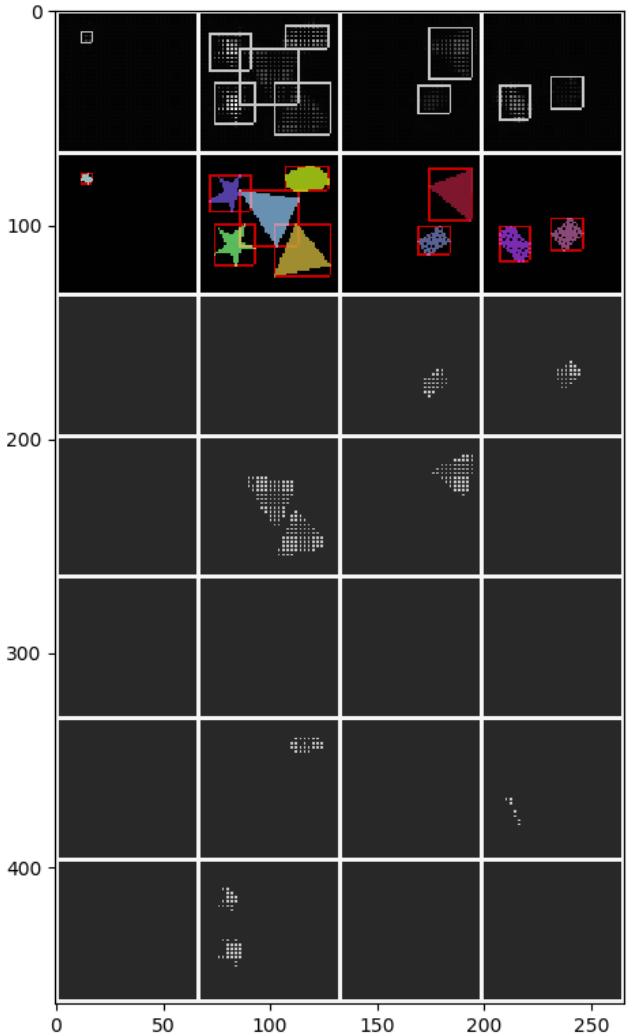


(a) Lower five rows show semantic segmentation. Row 3: rect; Row 4: triangle; Row 5: disk; Row 6: oval; Row 7: star

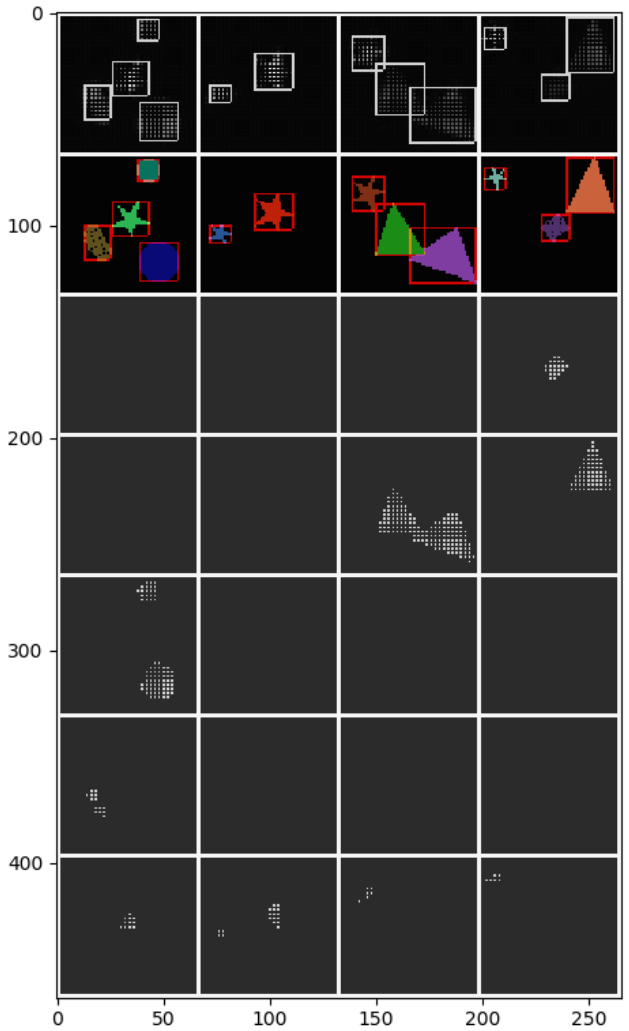


(b) Lower five rows show semantic segmentation. Row 3: rect; Row 4: triangle; Row 5: disk; Row 6: oval; Row 7: star

Results on Unseen Test Data (contd.)



(a) Lower five rows show semantic segmentation. Row 3: rect; Row 4: triangle; Row 5: disk; Row 6: oval; Row 7: star



(b) Lower five rows show semantic segmentation. Row 3: rect; Row 4: triangle; Row 5: disk; Row 6: oval; Row 7: star

Results on Unseen Test Data (contd.)