Semantic Segmentation of Images with Convolutional Networks

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Thursday 9th April, 2020 21:34

Preamble

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.

At a more sophisticated level, a system that carries out semantic segmentation should also output a symbolic expression that reflects an understanding of the scene that is based on the objects found in the image and their spatial relationships with one another. This more ambitious definition of semantic segmentation is still largely unfulfilled in computer vision, but continues to remain as a goal to aim for.

This lecture on semantic segmentation is based on the first definition — being able to segment out different objects based on their identity and being able to localize them at the same time.

Semantic segmentation obviously involves classification, but it is NOT classification at the level of the entire image.

Preamble (contd.)

Since the overarching goal is to segment out of an image the objects that belong to different categories, **the classification that is needed for semantic segmentation is at the pixel level.**

Pixel-level classification needed for semantic segmentation means that now you 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 aggregate the pixels together for any object — but, at the same time, the network must be able to map those abstractions back to the pixel level.

Preamble (contd.)

This calls for using some sort of an encoder-decoder architecture for a neural network. The encoder's job would be to create high-level abstractions in an image and the decoder's job to map those back to the image.

That what's mentioned above 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 (which is something that should be very familiar to you by this time). The transpose convolutions are used 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 part and the decoder part in order to harmonize them.

Preamble (contd.)

The main goal of this lecture is present my implementation of the `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. The original `Unet`, on the other hand, outputs a single output mask for the segmentation. **By contrast, `mUnet` produces multiple masks, one for each type of object.**

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` in Version 1.0.9 of `DLStudio` that you can download from:

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

Make sure your install of `DLStudio` is 1.0.9 or higher.

Preamble (contd.)

Regarding the organization of this lecture, since basic to understanding the notion of "transpose convolution" is the representation of a 2D convolution as a matrix-vector product, the section that follows is devoted to that.

Subsequently, I'll introduce you to my `mUnet` network for semantic segmentation and explain in what sense this network goes beyond the now classic `Unet`.

That will be followed by a presentation of the `PurdueShapes5MultiObject` dataset for experimenting with semantic segmentation.

Finally, I will show some semantic segmentation results on the `PurdueShapes5MultiObject` dataset as obtained with the `mUnet` network.

Outline

1. A Brief Survey of Networks for Semantic Segmentation
2. Transpose Convolutions
3. The Building Blocks for mUnet
4. The mUnet Network for Semantic Segmentation
5. The PurdueShapes5MultiObject Dataset for Semantic Segmentation
6. Training the mUnet
7. Testing mUnet on Unseen Data

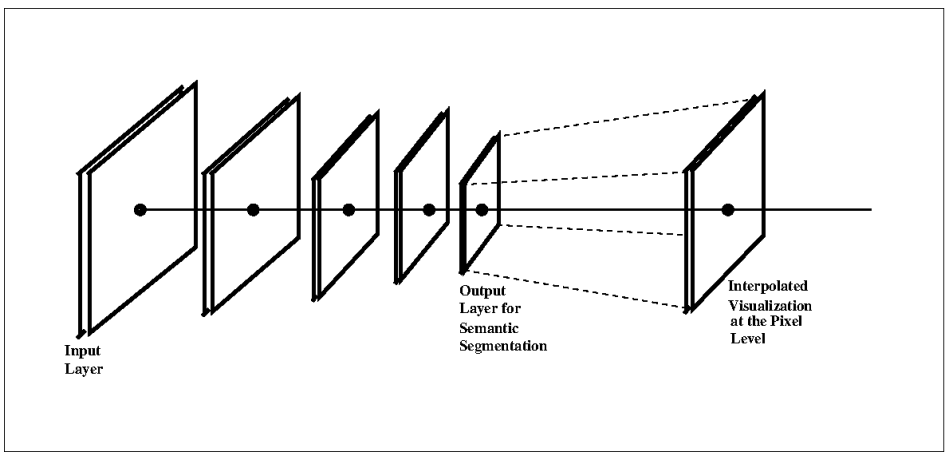
Outline

1. A Brief Survey of Networks for Semantic Segmentation
2. Transpose Convolutions
3. The Building Blocks for mUnet
4. The mUnet Network for Semantic Segmentation
5. The PurdueShapes5MultiObject Dataset for Semantic Segmentation
6. Training the mUnet
7. Testing mUnet on Unseen Data

Generation 1 Networks for Semantic Segmentation

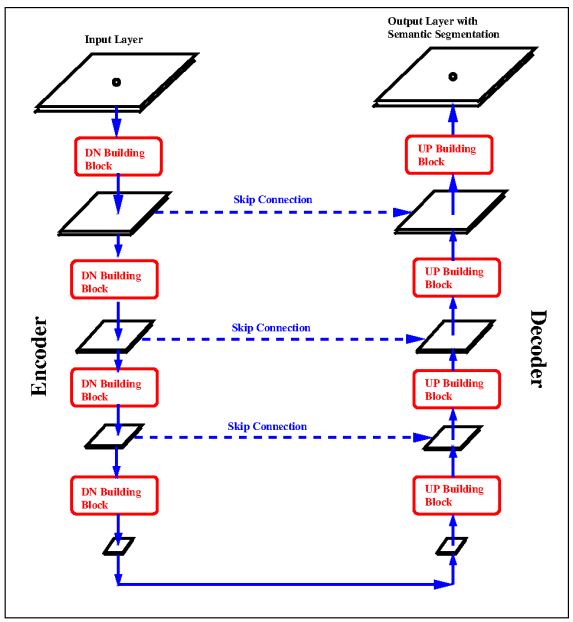
- These fully convolutional networks for semantic segmentation involve mostly the same steps as a CNN for image classification, the main difference being that whereas the latter uses a linear layer at the end of the network whose output yields the class label, the former ends in a reduced resolution representation of the input image.
- That such an approach for semantic segmentation could be made to work with a fully convolutional network was first demonstrated in 2015 by Shelhamer et al. in

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



Generation 2 Networks

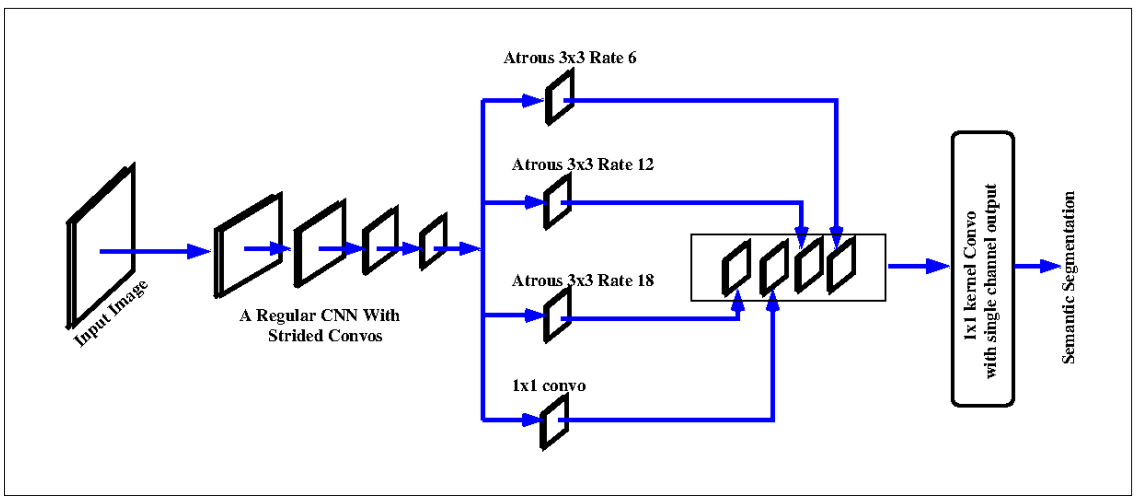
- The second generation networks are based on the encoder-decoder principle, with the job of the encoder being to create a low-res rep that captures the semantic components in the input image and the job of the decoder being to map those back to the pixel level. The accuracy of the detail in the back-to-pixel mapping in the docoder is aided by the skip links from the corresponding levels in the encoder.
- Here is the original paper on this encoder-decoder approach:
  `https://arxiv.org/abs/1505.04597`



Generation 3 Networks

- The third generation networks for semantic segmentation are based on the idea of atrous convolution. [See slide 14 for what is meant by atrous convolution.] The goal here is to detect semantically meaningful blobs in an input at different scales without increasing the number of learnable parameters.
- Here is the paper that got everybody in the DL community excited about the possibilities of atrous convolutions:

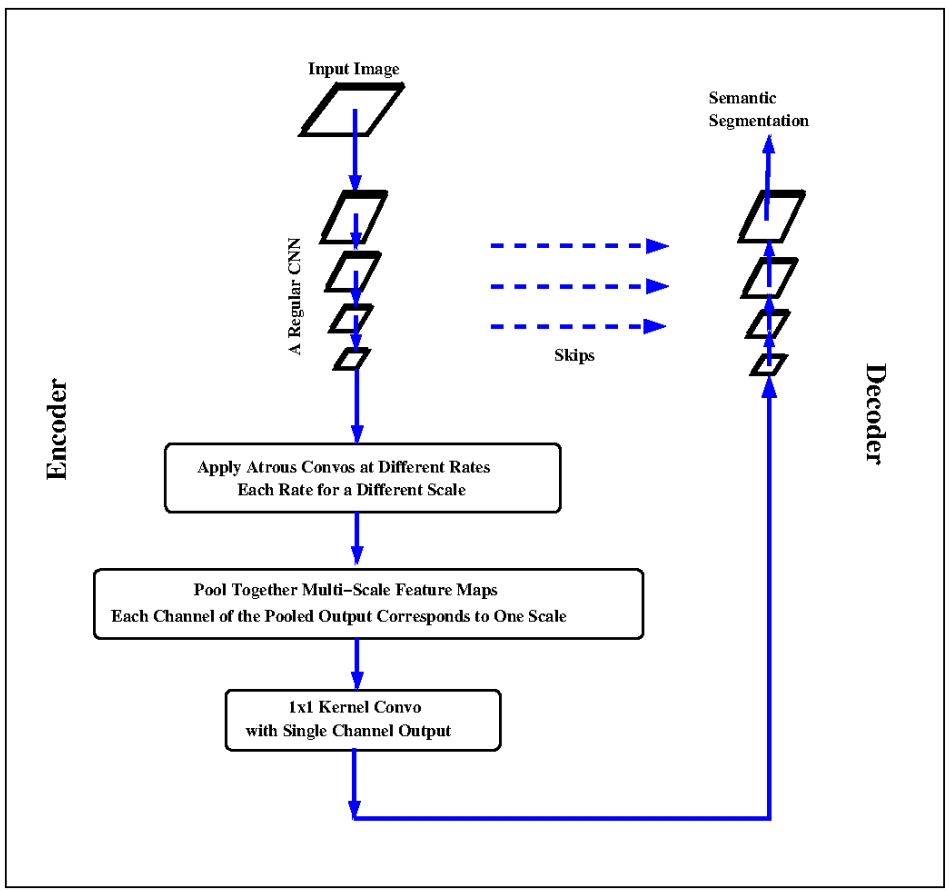
  `https://arxiv.org/abs/1606.00915`



Generation 4 Networks

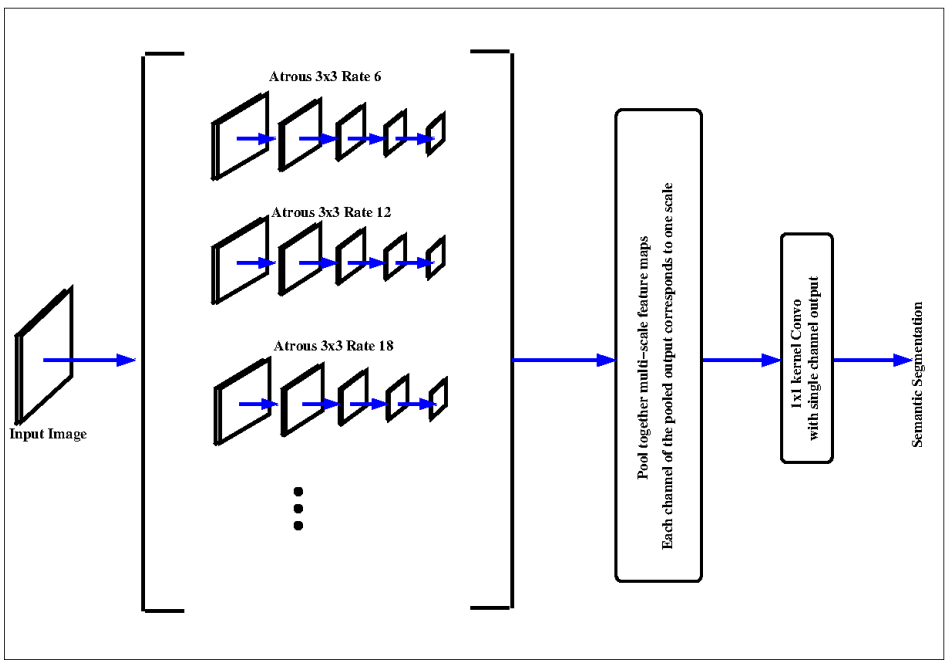
- These use atrous convolutions in an encode-decoder framework.
- Here is the paper that showed how the atrous-convo based learning could be carried in an encoder-decoder framework:

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



A Possible Generation 5 Network

- Here is my prediction for a network of the future based on atrous convolutions. I base this prediction on the notion that two successive applications of a rate M atrous convolution do not yield the same information a single application of an atrous convolution at rate $2M$.
- Therefore, there is learnable information in creating multiple pyramids based on successive applications of the same rate atrous convolution — but with a very elevated computational burden.



What is Atrous Convolution?

- Atrous convolution 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 the 16×16 input:

```
[[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.]]
```

What is 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.]]
```

What is 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.]]
```

What is Atrous Convolution (contd.)

- The basic idea of an atrous convolution is to detect an image feature at a specific scale. So 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.
- My example on the last couple of slides does not do full justice to the idea described above since the nature of grayscale variation in the input pattern I used should be detectable at all scales (albeit with different 'signatures').
- Nevertheless, you should be able extend that example in your own head and imagine variants of my input pattern that would lead to detection of the image feature in question at different scales.

Outline

1. A Brief Survey of Networks for Semantic Segmentation
2. **Transpose Convolutions**
3. The Building Blocks for mUnet
4. The mUnet Network for Semantic Segmentation
5. The PurdueShapes5MultiObject Dataset for Semantic Segmentation
6. Training the mUnet
7. Testing mUnet on Unseen Data

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), it is best to implement a convolution as a vector-matrix 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:

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

  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.

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 Figure 1 on the next slide.

Convolution as a Matrix-Vector Product (contd.)

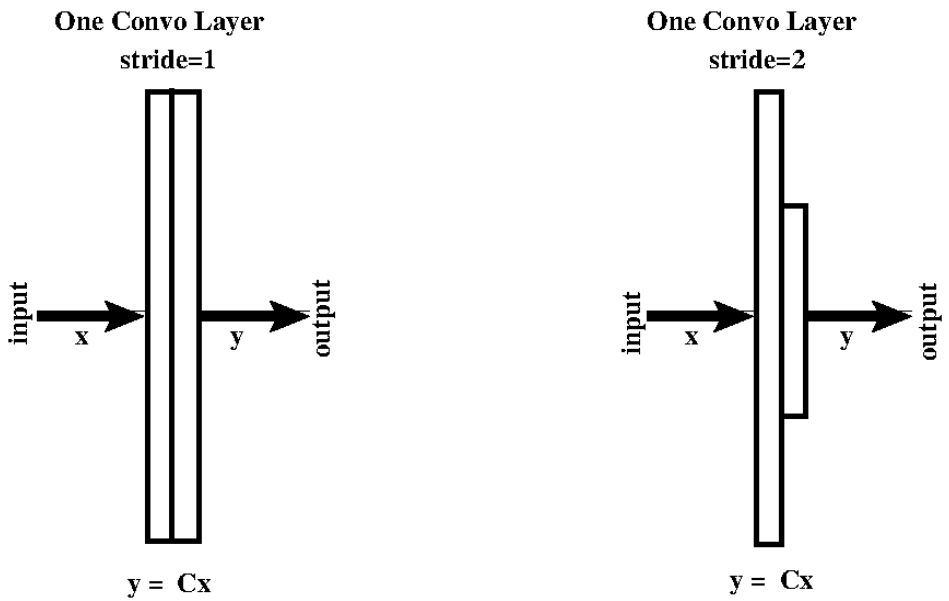


Figure: Forward data flow through one convo layer in a CNN

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 Figure 2 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.)

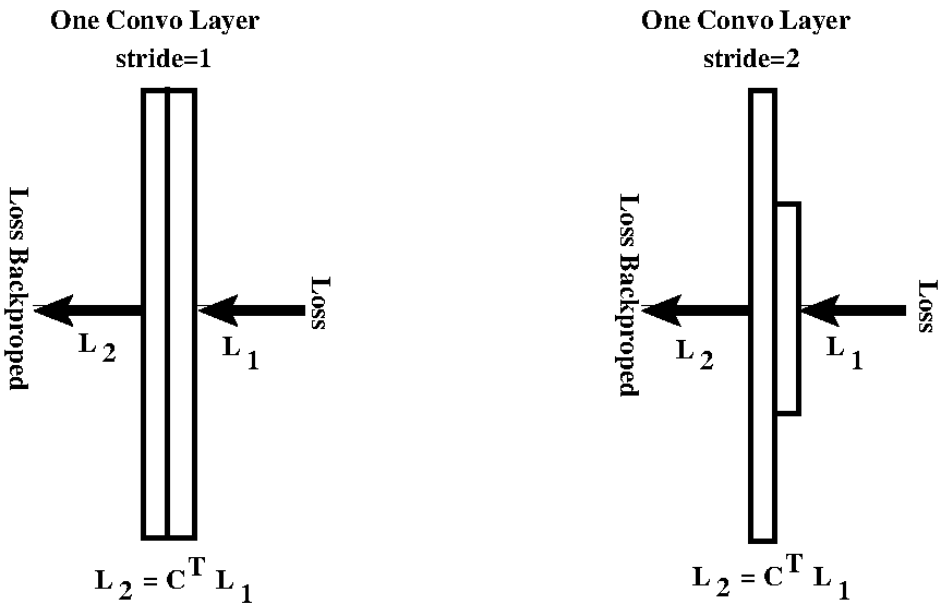


Figure: Forward data flow through one convo layer in a CNN

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      5  6  7  8          =   1a+2b+5c+6d     2a+3b+6c+7d     3a+4b+7c+8d
c  d                [stride=(1,1)]  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:

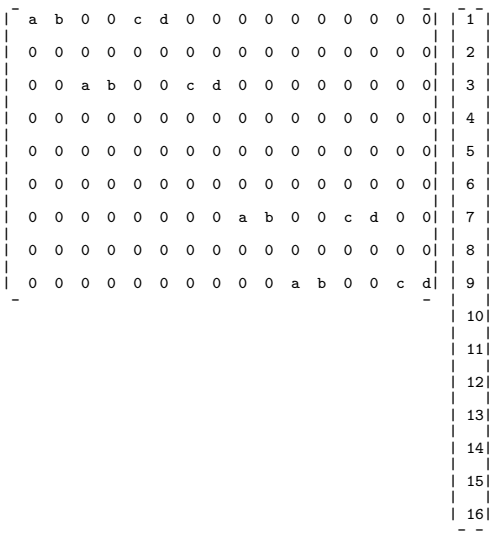
$$\begin{bmatrix} a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & b & 0 & 0 & c & d & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & a &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

Convolution as a Matrix-Vector Product for 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, 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 $\mathbf{y} = \mathbf{Cx}$ where $\mathbf{C}$ is the 9×16 matrix shown above.
- Let's now consider the same convolution but with a stride of $(2, 2)$. As shown on the next slide, all we need do in the matrix-vector representation for this new case is to zero-out the alternative rows of the matrix in those vertical portions of the matrix that correspond to the kernel moving horizontally across the input. We must also zero out entirely the alternate vertical partitions of the matrix for the individual horizontal sweeps.

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 $\mathbf{C}$.

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

- 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 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.,

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

- 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 an 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.

- **In other words, 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

1. A Brief Survey of Networks for Semantic Segmentation
2. Transpose Convolutions
3. **The Building Blocks for mUnet**
4. The mUnet Network for Semantic Segmentation
5. The PurdueShapes5MultiObject Dataset for Semantic Segmentation
6. Training the mUnet
7. Testing mUnet on Unseen Data

The Two Building Blocks for the mUnet Network

- The building blocks of `mUnet` are the two 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.
- We need two different types of building blocks on account of the different needs of the encoder and the decoder. The network `SkipBlockDN` 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`, do not be confused by the significance of the skip connections in these two classes. These are not as essential as the skip connections you will see in `mUnet`.
- I'm using the skip connections in `SkipBlockDN` and `SkipBlockUP` to make the learning in these building blocks a bit more efficient. As it turns out, there is a well-known semantic segmentation network out there called Tiramisu that also has skip connections in the encoder and the decoder.
- On the other hand, the skip connections that you will see in `mUnet` are 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 = torch.nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = torch.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 += identity
            else:
                out[:,:self.in_ch,:,:] += identity
                out[:,self.in_ch:,:,:] += identity
        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__()
        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, padding=0)
    def forward(self, x):
        identity = x
        out = self.convoT1(x)
        out = self.bn1(out)
        out = torch.nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convoT2(out)
            out = self.bn2(out)
            out = torch.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 += identity
            else:
                out += identity[:,self.out_ch:,:,:]
        return out
```

Outline

1 A Brief Survey of Networks for Semantic Segmentation

2 Transpose Convolutions

3 The Building Blocks for mUnet

4 The mUnet Network for Semantic Segmentation

5 The PurdueShapes5MultiObject Dataset for Semantic Segmentation

6 Training the mUnet

7 Testing mUnet on Unseen Data

The Architecture of mUnet

- This network is called `mUnet` because it is intended for segmenting out multiple objects simultaneously form an image. [A weaker reason for "Multi" in the name of the class is that it uses skip connections not only across the two arms of the "U", but also also along the arms.]
- Shown on the next slide is the implementation code for `mUnet` that is in the inner class `SemanticSegmentation` of the `DLStudio` module.
- 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):
    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)
        self.pool = nn.MaxPool2d(2, 2)
        ##  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_co
        self.skip64dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 64, downsample=True, skip_connections=s
        self.skip64to128DN = DLStudio.SemanticSegmentation.SkipBlockDN(64, 128, skip_connections=skip_connectio
        self.skip128DN_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128DN_arr.append(DLStudio.SemanticSegmentation.SkipBlockDN(128, 128, skip_connections=skip
        self.skip128dsDN = DLStudio.SemanticSegmentation.SkipBlockDN(128,128, downsample=True, 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_co
        self.skip64usUP = DLStudio.SemanticSegmentation.SkipBlockUP(64, 64, upsample=True, skip_connections=ski
        self.skip128to64UP = DLStudio.SemanticSegmentation.SkipBlockUP(128, 64, skip_connections=skip_connectio
```

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

mUnet – The Definition (contd.)

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

```
        self.skip128UP_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128UP_arr.append(DLStudio.SemanticSegmentation.SkipBlockUP(128, 128, skip_connections=skip
        self.skip128usUP = DLStudio.SemanticSegmentation.SkipBlockUP(128,128, upsample=True, skip_connections=s
        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 = self.pool(torch.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)
        x = self.bn2DN(x)
        num_channels_to_save3 = x.shape[1] // 2
```

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

mUnet – The Definition (contd.)

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

```
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

1. A Brief Survey of Networks for Semantic Segmentation
2. Transpose Convolutions
3. The Building Blocks for mUnet
4. The mUnet Network for Semantic Segmentation
5. **The PurdueShapes5MultiObject Dataset for Semantic Segmentation**
6. Training the mUnet
7. Testing mUnet on Unseen Data

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 input, the mask array associated with the input is "aware" of the full extent of each object in the input image. It is the mask array that is applied in the different object-type specific channels in the output of the network that generates the loss vis-a-vis the prediction for the mask in that channel.

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.)

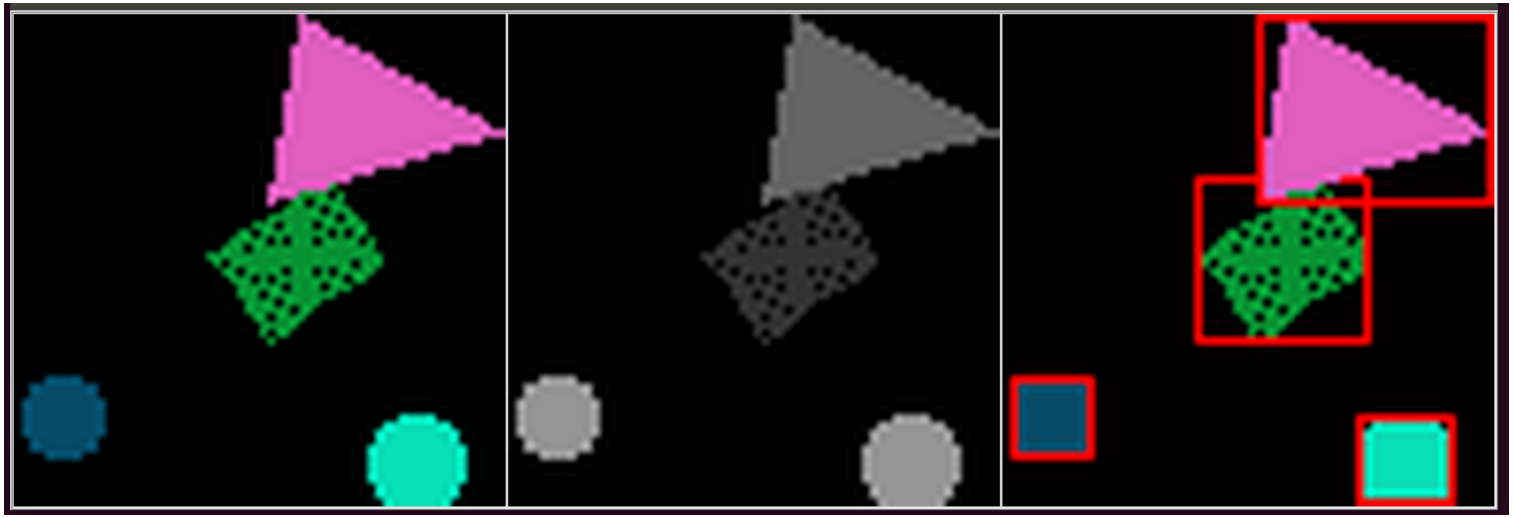


Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

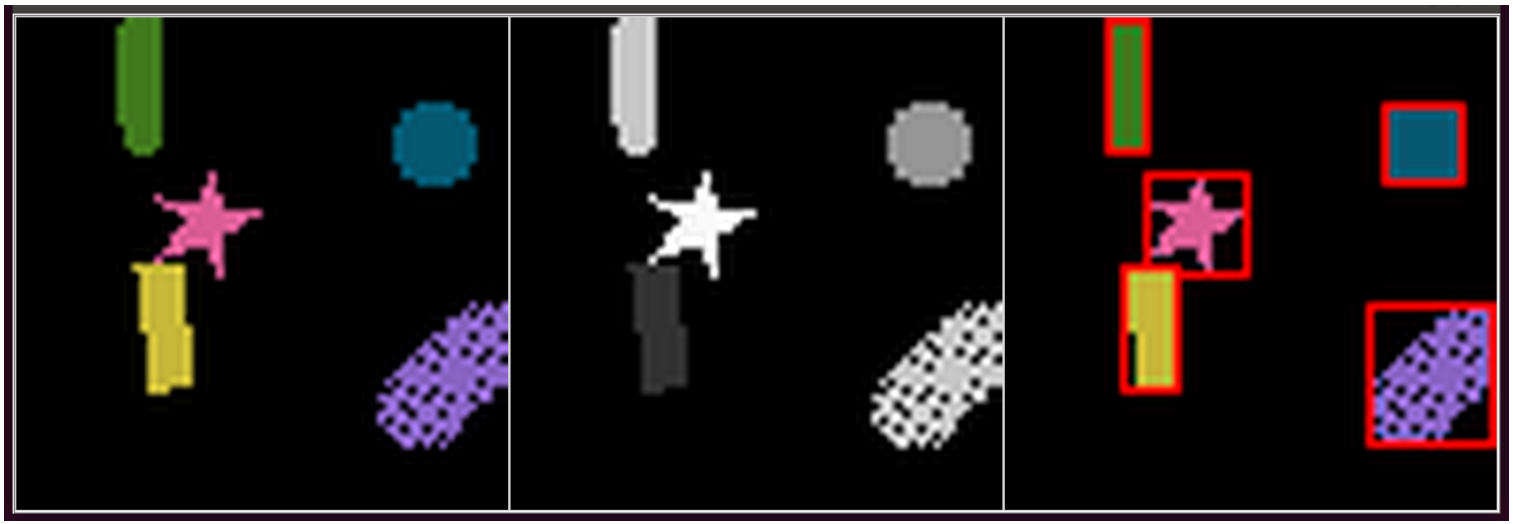


Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

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.)

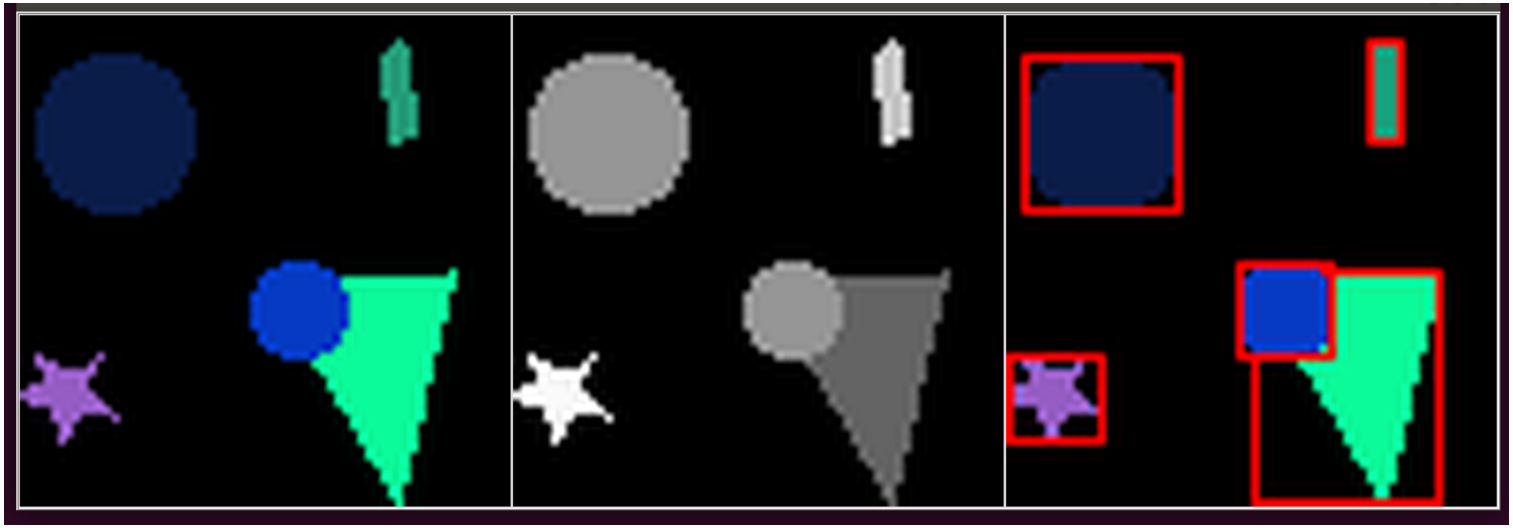


Figure: Left: A dataset image; Middle: Its multi-valued mask; Right: BBoxes for the objects

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

Downloading the PurdueShapes5MultiObject Dataset

- This dataset is available in the following files in the "data' subdirectory of the "Examples" directory of the DLStudio distribution (version 1.0.9). You will see the following 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

1. A Brief Survey of Networks for Semantic Segmentation
2. Transpose Convolutions
3. The Building Blocks for mUnet
4. The mUnet Network for Semantic Segmentation
5. The PurdueShapes5MultiObject Dataset for Semantic Segmentation
6. **Training the mUnet**
7. Testing mUnet on Unseen Data

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

1 A Brief Survey of Networks for Semantic Segmentation

2 Transpose Convolutions

3 The Building Blocks for mUnet

4 The mUnet Network for Semantic Segmentation

5 The PurdueShapes5MultiObject Dataset for Semantic Segmentation

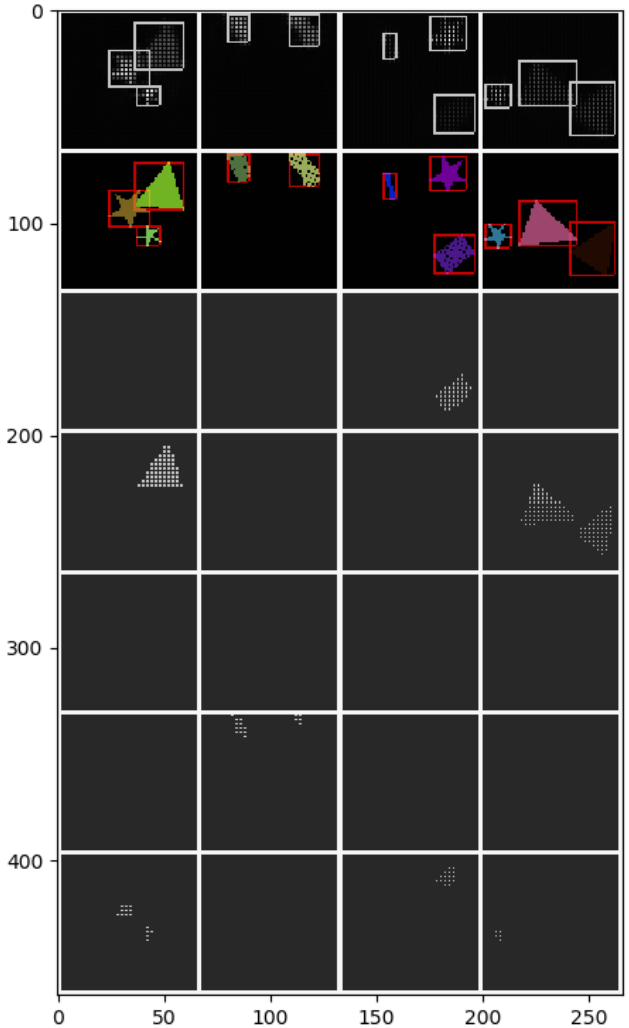
6 Training the mUnet

7 Testing mUnet on Unseen Data

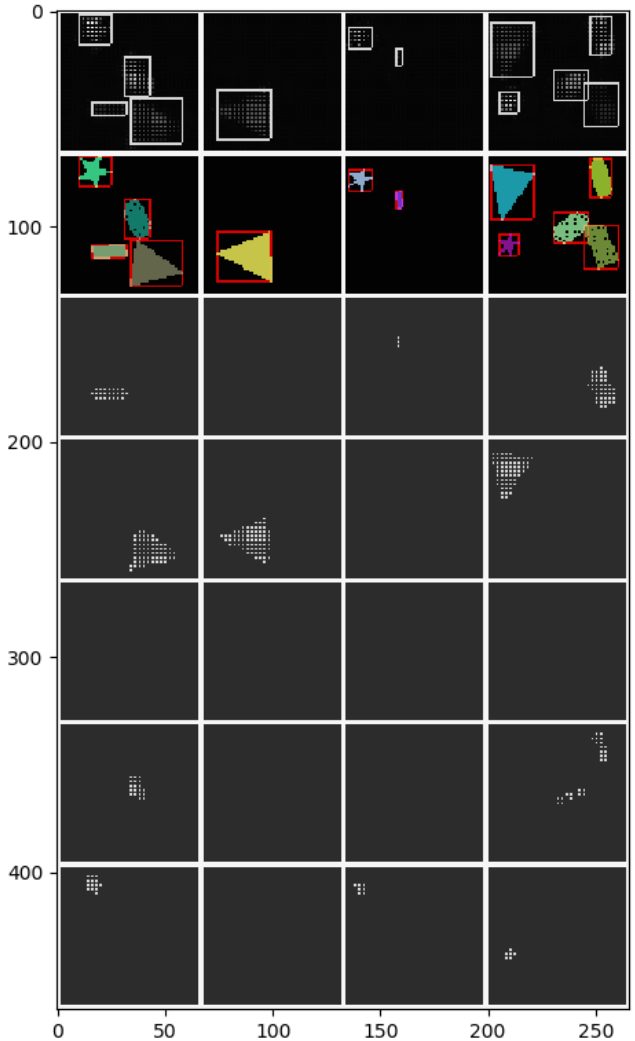
Results on Unseen Test Data

- As mentioned previously on Slide 35, 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.)

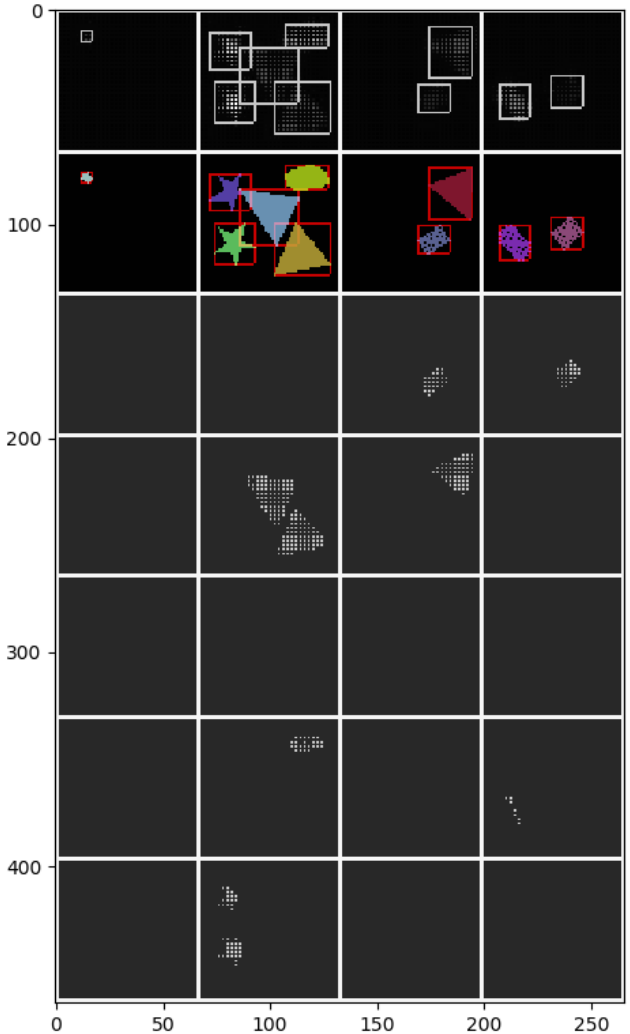


(a) Lower five rows show semantic segmentation. Row 3: rect; Row 4: triangle; Row 5: disk; Row 6: oval; Row 7: star

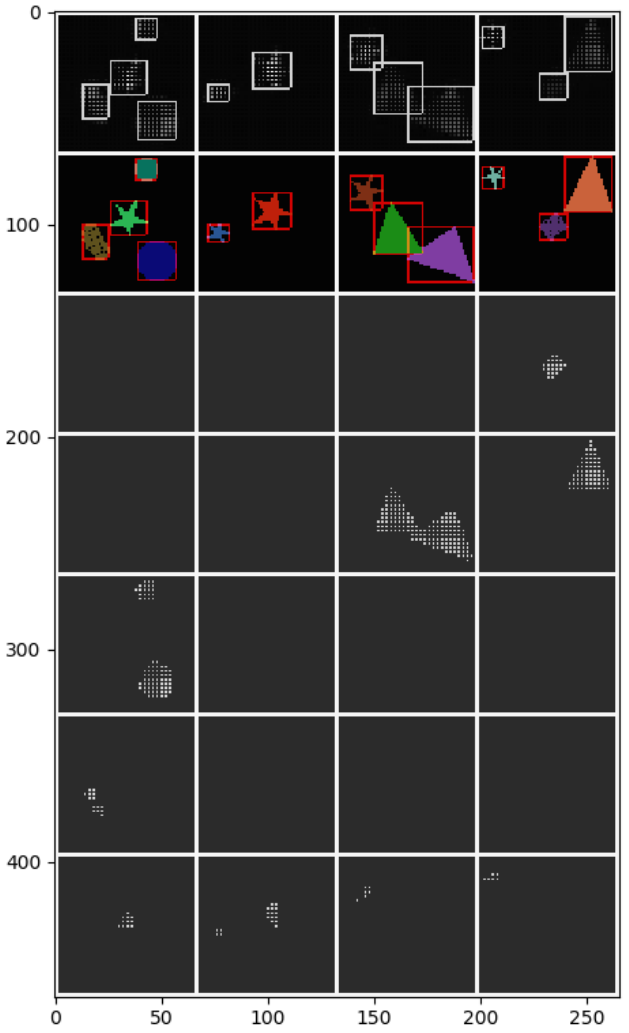


(b) Lower five rows show semantic segmentation. Row 3: rect; Row 4: triangle; Row 5: disk; Row 6: oval; Row 7: star

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.)