AI VIETNAM
All-in-One Course
(TA Session)

Seminar 3

Exploring Disease Classification Methods for Volumetric (3D) Medical Image (CTs/MRIs)

Presenter: Vũ Mai Anh

Upcoming PhD Student (Fall 2024)
University of Houston

AI VIETNAM
All-in-One Course
(TA Session)

Outline

Volumetric data

3D CNN

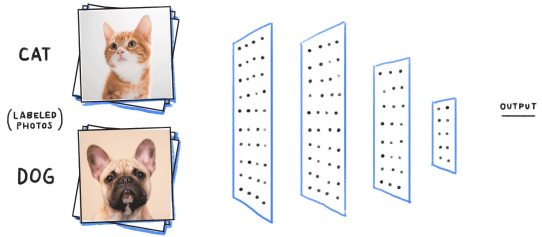
3D Unet

Exploring RSNA 2023 with 3D Unet and 2.5D CNN + RNN

Introduction

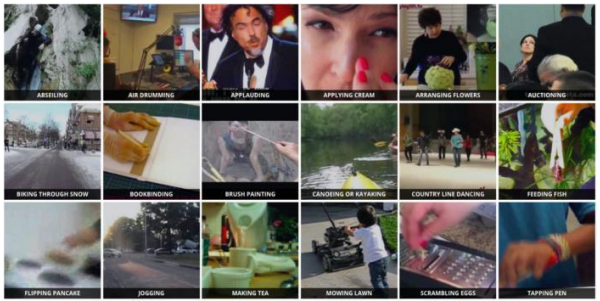
❑ Volumetric Data

Image

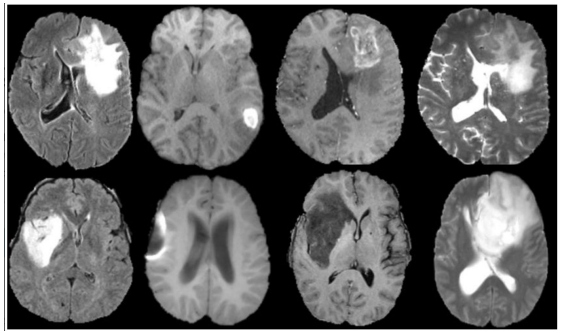


Sources [OleksandrKosovan]

Series of Images



Kinetics Human Action Video Dataset
(Kay et al.)



The Multimodal Brain Tumor Image Segmentation Benchmark
(BRATS 2017, Bjoern H. Menze et al.)

RSNA Abdominal Trauma Detection AI
Challenge (2023)

2D CNN to 3D CNN

❑ 2D CNN

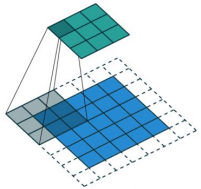
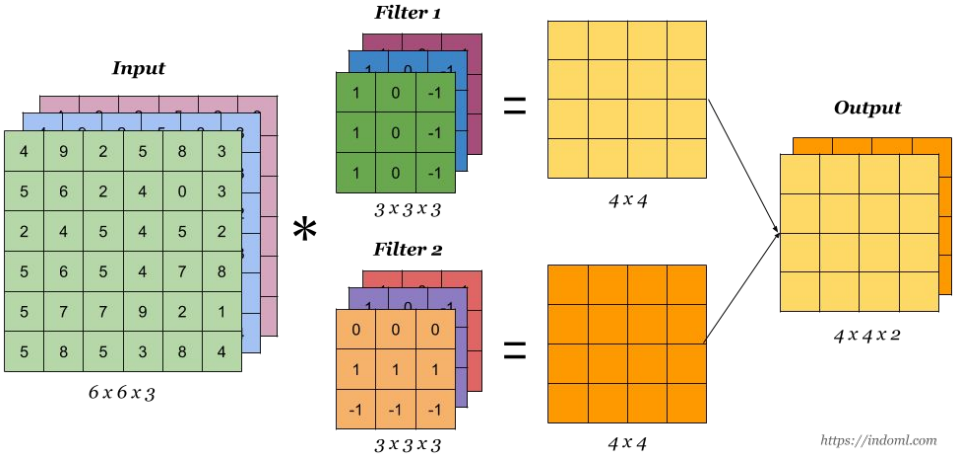
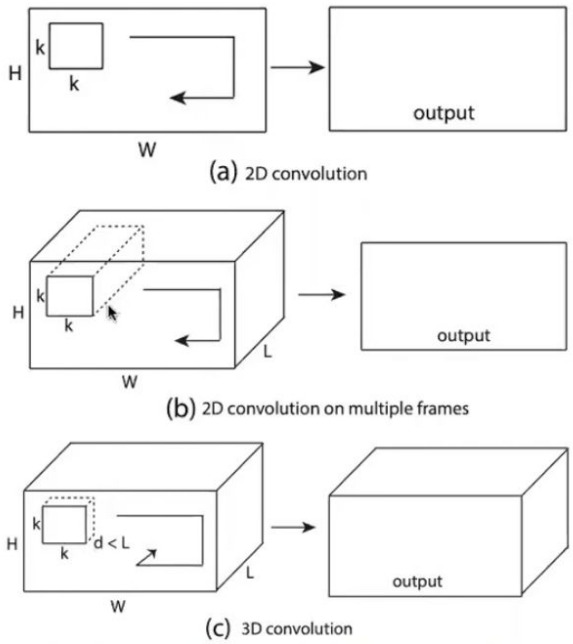


Image source: hannibunny.github.io



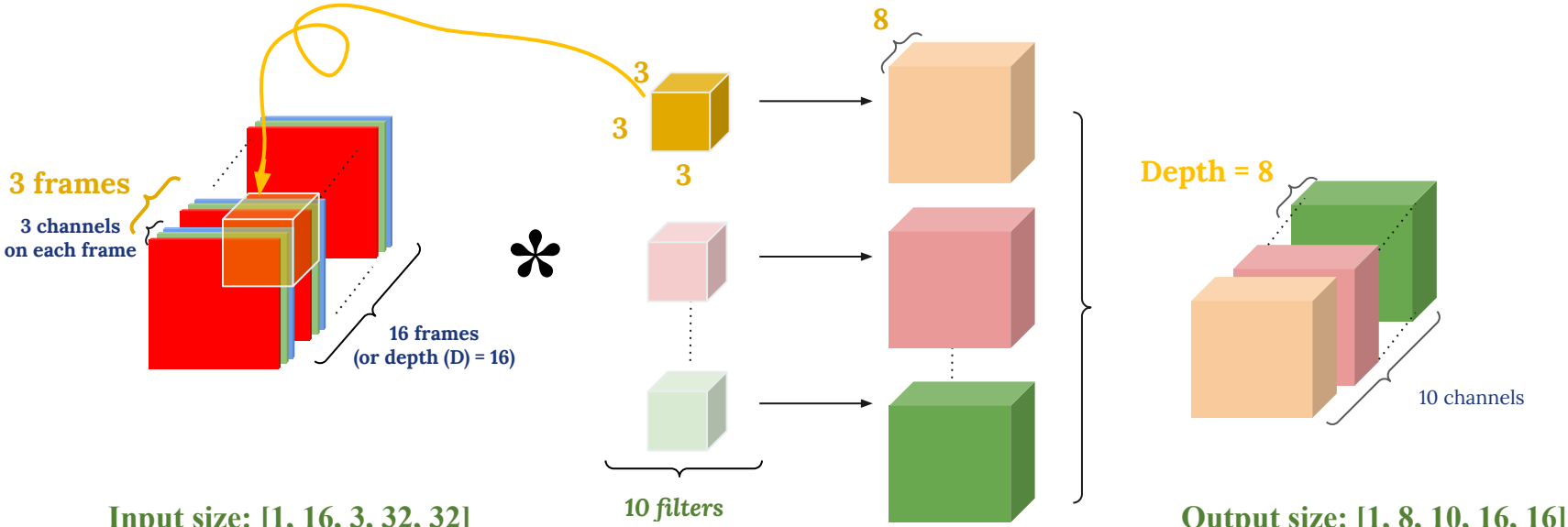
Source: https://www.coursera.org/learn/convolutional-neural-networks

❑ 2D CNN to 3D CNN



2D CNN to 3D CNN

3D CNN



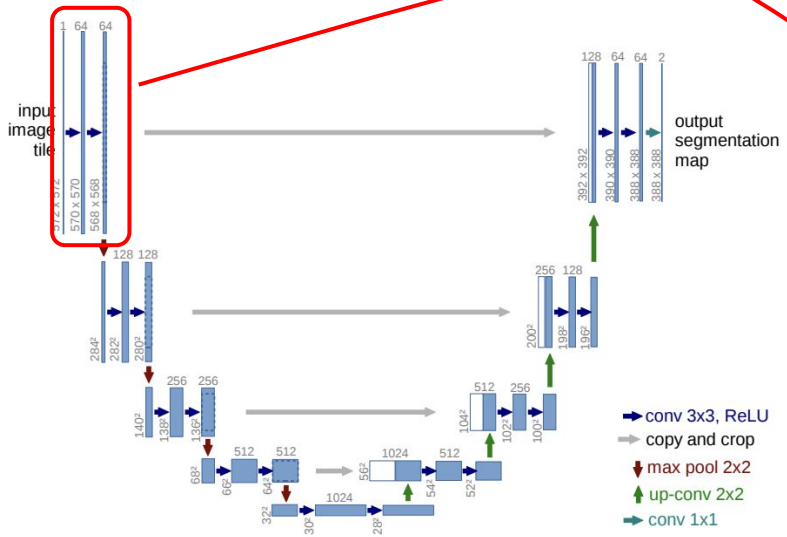
Input size: [1, 16, 3, 32, 32]
(batch size, D, C ,W, H)

10 filters
3 x 3 x 3 Conv
(padding = 1, stride =2)

Output size: [1, 8, 10, 16, 16]
(batch size, D, C ,W, H)

2D Unet to 3D Unet

Review 2D Unet




```python
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DoubleConv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),

            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        return self.conv(x)
```

```python
class UNET2D(nn.Module):
    def __init__(self, in_channels=3, out_channels=1, features = [64, 128, 256, 512]):
        super(UNET2D, self).__init__()
        self.downsamplings = nn.ModuleList()
        self.upsamplings = nn.ModuleList()
        self.pool = nn.MaxPool2d(2, 2)

        #down sampling
        for feature in features:
            self.downsamplings.append(DoubleConv(in_channels, feature))
            in_channels = feature

        #up sampling
        for feature in reversed(features):
            self.upsamplings.append(nn.Sequential(
                    nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True),
                    DoubleConv(feature * 2, feature)
                )
            )
            self.upsamplings.append(DoubleConv(feature*2, feature))

        self.bottleneck = DoubleConv(features[-1], features[-1]*2)
        self.final_conv = nn.Conv2d(features[0], out_channels, kernel_size=1)
```

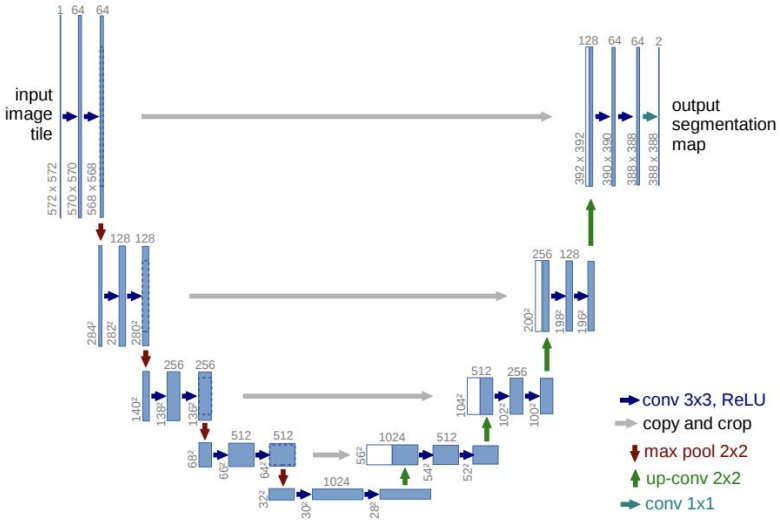
U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for Biological Signalling Studies,
University of Freiburg, Germany
ronneber@informatik.uni-freiburg.de,
WWW home page: http://lmb.informatik.uni-freiburg.de/

2D Unet to 3D Unet

Review 2D Unet



U-Net: Convolutional Networks for Biomedical Image Segmentation

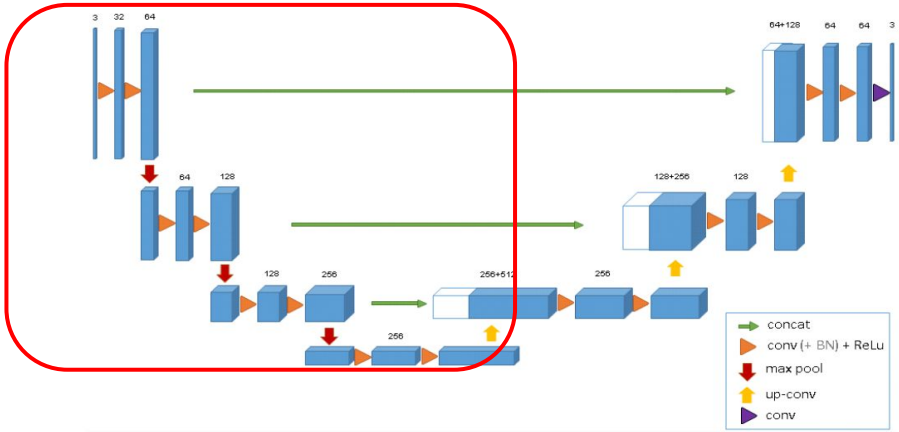
Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for Biological Signalling Studies, University of Freiburg, Germany
ronneber@informatik.uni-freiburg.de,
WWW home page: http://lmb.informatik.uni-freiburg.de/

```python
class UNET2D(nn.Module):
    def __init__(self, in_channels=3, out_channels=1, features = [64, 128, 256, 512]):
        super(UNET2D, self).__init__()
        self.downsamplings = nn.ModuleList()
        self.upsamplings = nn.ModuleList()
        self.pool = nn.MaxPool2d(2, 2)

        #down sampling
        for feature in features:
            self.downsamplings.append(DoubleConv(in_channels, feature))
            in_channels = feature

        #up sampling
        for feature in reversed(features):
            self.upsamplings.append(nn.Sequential(
                    nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True),
                    DoubleConv(feature * 2, feature)
                )
            )
            self.upsamplings.append(DoubleConv(feature*2, feature))

        self.bottleneck = DoubleConv(features[-1], features[-1]*2)
        self.final_conv = nn.Conv2d(features[0], out_channels, kernel_size=1)

    def forward(self, x):
        #skip connection
        skip_connection = []

        for down_sampling in self.downsamplings:
            x = down_sampling(x)
            skip_connection.append(x)
            x = self.pool(x)

        x = self.bottleneck(x)
        skip_connection = skip_connection[::-1] #reverse

        for idx  in range(0, len(self.upsamplings), 2):
            x = self.upsamplings[idx](x)
            skip_connection_feature = skip_connection[idx//2]
            concatenated_features = torch.cat([x, skip_connection_feature], dim=1)
            x = self.upsamplings[idx+1](concatenated_features)


        return self.final_conv(x)
```

2D Unet to 3D Unet

3D Unet




```python
class DownSampingDouble3DCNN(nn.Module):
    def __init__(self, in_channels=None, out_channels=None, bottleneck=False) -> None:
        super(DownSampingDouble3DCNN, self).__init__()
        self.conv1 = nn.Conv3d(
          in_channels=in_channels, out_channels=out_channels//2, kernel_size=(3,3,3), padding=1)
        self.bn1 = nn.BatchNorm3d(num_features=out_channels//2)
        self.conv2 = nn.Conv3d(
          in_channels=out_channels//2, out_channels=out_channels, kernel_size=(3,3,3), padding=1)
        self.bn2 = nn.BatchNorm3d(num_features=out_channels)
        self.relu = nn.ReLU()
        self.bottleneck = bottleneck
        if not bottleneck:
            self.pooling = nn.MaxPool3d(kernel_size=(2,2,2), stride=2)

    def forward(self, x):
        residual = self.relu(self.bn1(self.conv1(x)))
        residual = self.relu(self.bn2(self.conv2(residual)))
        out = residual
        if not self.bottleneck:
            out = self.pooling(residual)
        return out, residual
```

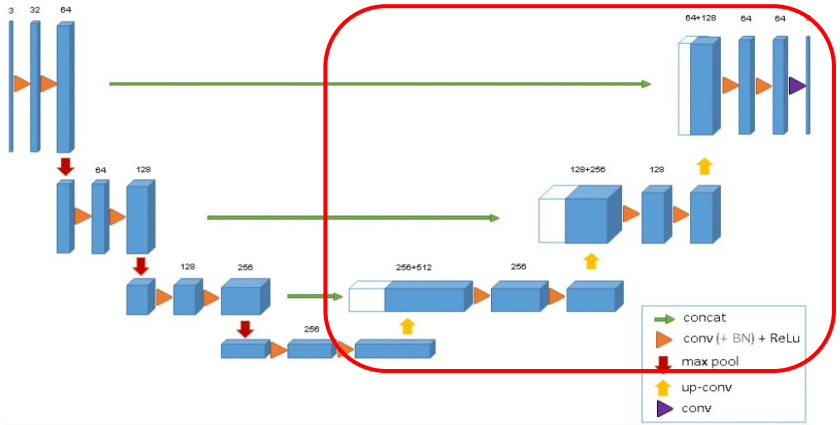
3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation

Özgün Çiçek[1,2], Ahmed Abdulkadir[1,4], Soeren S. Lienkamp[2,3], Thomas Brox[1,2], and Olaf Ronneberger[1,2,5]

[1] Computer Science Department, University of Freiburg, Germany
[2] BIOSS Centre for Biological Signalling Studies, Freiburg, Germany
[3] University Hospital Freiburg, Renal Division, Faculty of Medicine, University of Freiburg, Germany
[4] Department of Psychiatry and Psychotherapy, University Medical Center Freiburg, Germany
[5] Google DeepMind, London, UK
cicek@cs.uni-freiburg.de

2D Unet to 3D Unet

3D Unet



3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation

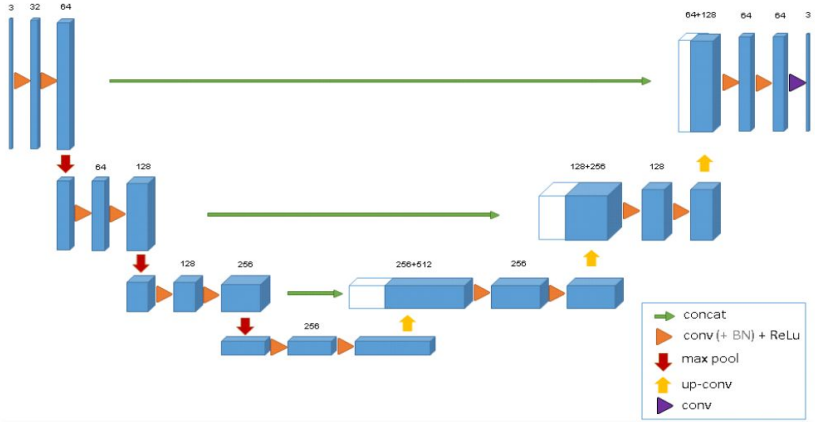
Özgün Çiçek[1,2], Ahmed Abdulkadir[1,4], Soeren S. Lienkamp[2,3], Thomas Brox[1,2], and Olaf Ronneberger[1,2,5]

[1] Computer Science Department, University of Freiburg, Germany
[2] BIOSS Centre for Biological Signalling Studies, Freiburg, Germany
[3] University Hospital Freiburg, Renal Division, Faculty of Medicine, University of Freiburg, Germany
[4] Department of Psychiatry and Psychotherapy, University Medical Center Freiburg, Germany
[5] Google DeepMind, London, UK
cicek@cs.uni-freiburg.de

```python
class UpsamplingDouble3DCNN(nn.Module):
    def __init__(self, in_channels=None, res_channels=0, out_channels=None, last_layer=False,
num_classes=None):
        super(UpsamplingDouble3DCNN, self).__init__()
        self.last_layer = last_layer
        self.num_classes = num_classes

        self.upconv1 = nn.ConvTranspose3d(
            in_channels=in_channels, out_channels=in_channels, kernel_size=(2, 2, 2), stride=2)
        self.relu = nn.ReLU()
        self.bn1 = nn.BatchNorm3d(num_features=in_channels)
        self.bn2 = nn.BatchNorm3d(num_features=in_channels // 2)
        self.conv1 = nn.Conv3d(
            in_channels=in_channels + res_channels,
            out_channels=in_channels,
            kernel_size=(3,3,3),
            padding=1)
        self.conv2 = nn.Conv3d(
            in_channels=in_channels, out_channels=in_channels // 2, kernel_size=(3,3,3), padding=1)
        if self.last_layer:
            self.conv3 = nn.Conv3d(
                in_channels=in_channels // 2,
                out_channels=num_classes,
                kernel_size=(1,1,1))

    def forward(self, x, residual=None):
        out = self.upconv1(x)

        if residual is not None:
            if out.shape[2:] != residual.shape[2:]:
                residual = F.interpolate(
                    residual,
                    size=out.shape[2:],
                    mode='trilinear',
                    align_corners=False)

            out = torch.cat([out, residual], dim=1)

        out = self.relu(self.bn1(self.conv1(out)))
        out = self.relu(self.bn2(self.conv2(out)))

        if self.last_layer:
            out = self.conv3(out)
        return out
```

2D Unet to 3D Unet

3D Unet




```python
class Unet3D(nn.Module):
    def __init__(self, in_channels=3, num_classes=3, num_features=[64, 128, 256, 512]):
        super(Unet3D, self).__init__()
        self.downsamplings = nn.ModuleList()
        self.upsamplings = nn.ModuleList()

        # Down sampling path
        for i in range(len(num_features) - 1):
            self.downsamplings.append(
              DownSampingDouble3DCNN(
                in_channels=in_channels if i == 0 else num_features[i-1],
                                                        out_channels=num_features[i]))

        # Bottleneck
        self.bottleneck = DownSampingDouble3DCNN(
          in_channels=num_features[-2], out_channels=num_features[-1], bottleneck=True)

        # Up sampling path
        for i in range(len(num_features) - 1, 0, -1):
            self.upsamplings.append(
                UpsamplingDouble3DCNN(in_channels=num_features[i], res_channels=num_features[i-1],
                                      out_channels=num_features[i-1])
            )

        self.last_layer = UpsamplingDouble3DCNN(in_channels=num_features[0], res_channels=0,
                                                last_layer=True, num_classes=num_classes)

    def forward(self, x):
        residuals = []

        # Down sampling path
        for downsampling_layer in self.downsamplings:
            x, residual = downsampling_layer(x)
            residuals.append(residual)

        x, _ = self.bottleneck(x)

        # Up sampling path
        for idx, upsampling_layer in enumerate(self.upsamplings):
            x = upsampling_layer(x, residual=residuals[-idx-1])

        x = self.last_layer(x)
        return x
```

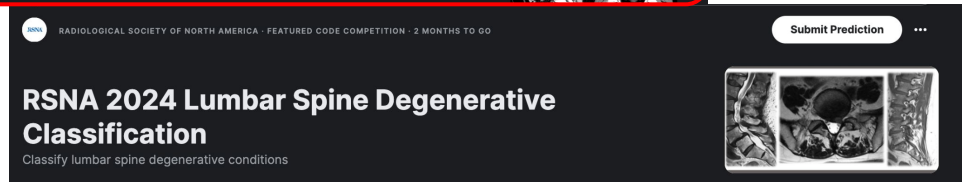
RSNA - RADIOLOGICAL SOCIETY OF NORTH AMERICA

2021

2022

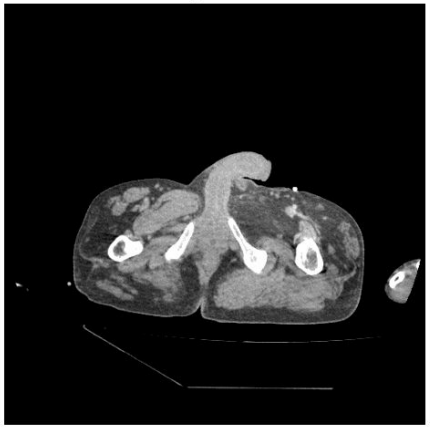
2023

2024

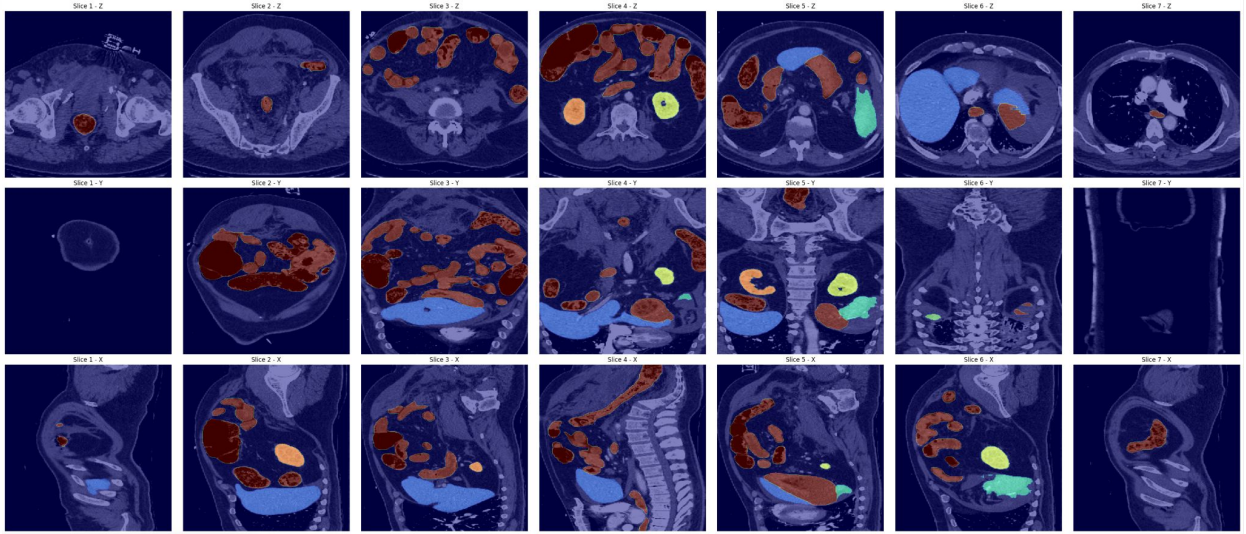


RNSA 2023

Volumetric data - Sample of 1 study (5 organs)

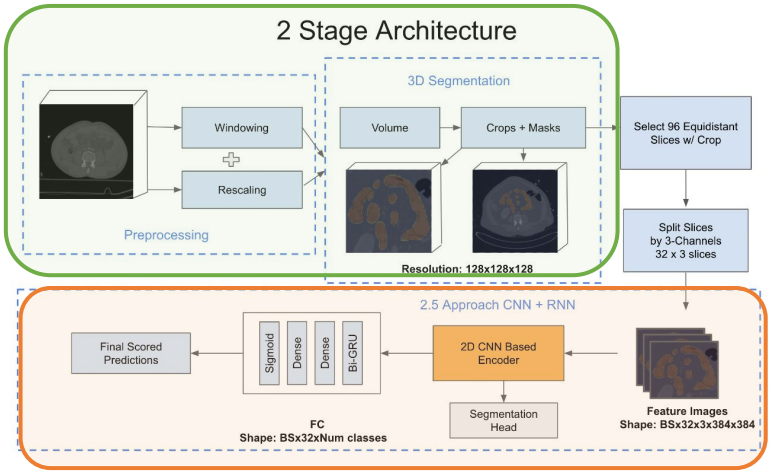


Frame 1



RNSA 2023

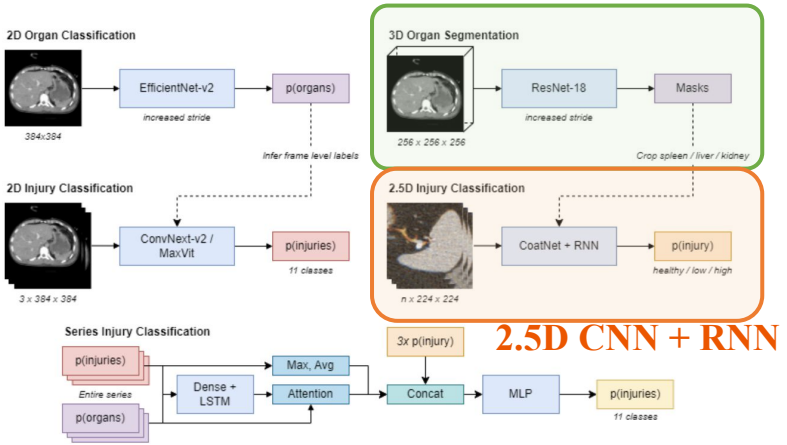
Stage 1 - 3D segmentation



Stage 2 - 2.5D CNN + RNN

Source: 1st place solution: https://github.com/Nischaydnk/RSNA-2023-1st-place-solution

3D segmentation



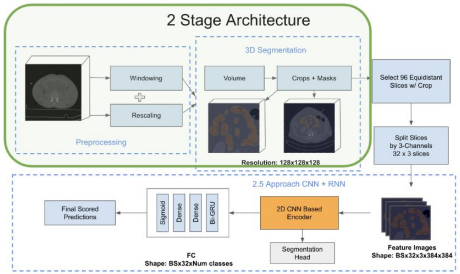
2.5D CNN + RNN

Source: 2nd place solution: https://github.com/TheoViel/kaggle_rsna_abdominal_trauma

RNSA 2023

Preprocessing

Stage 1 - 3D segmentation



Stage 2 - 2.5D CNN + RNN

Windowing

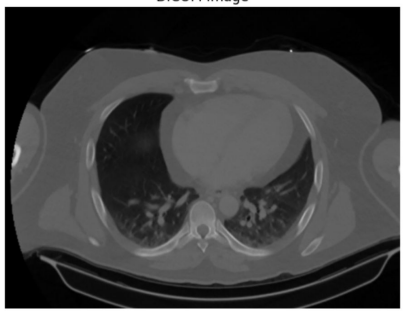
```python
def get_windowed_image(dcm, WL=50, WW=400):
    resI, resS = dcm.RescaleIntercept, dcm.RescaleSlope

    img = dcm.to_numpy_image()

    img = get_standardized_pixel_array(dcm)

    img = resS * img + resI

    upper, lower = WL+WW//2, WL-WW//2
    X = np.clip(img.copy(), lower, upper)
    X = X - np.min(X)
    X = X / np.max(X)
    X = (X*255.0).astype('uint8')

    return X
```

```python
[14]:
import pandas as pd

# Specify the file path
file_path = '/kaggle/input/rsna-2023-abdominal-trauma-detection/train_dicom_tags.parquet'

# Read the Parquet file
dicom_tags_df = pd.read_parquet(file_path)

# Display the first few rows of the DataFrame

print(dicom_tags_df.shape)
dicom_tags_df.head()
```

1500653, 35)

HighBit	ImageOrientationPatient	ImagePositionPatient	InstanceNumber	...	WindowCenter	WindowWidth	FileMetaInformationVersion	ImplementationClassUID	In
15	[1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000, ...	[-235.8665, -235.8665, 1459.20]	532	...	50.0	400.0		1.2.3.123456.4.5.1234.1.12.0	
15	[1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000, ...	[-235.8665, -235.8665, 1474.40]	513	...	50.0	400.0		1.2.3.123456.4.5.1234.1.12.0	
15	[1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000, ...	[-235.8665, -235.8665, 1310.40]	718	...	50.0	400.0		1.2.3.123456.4.5.1234.1.12.0	
15	[1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000, ...	[-235.8665, -235.8665, 1520.00]	456	...	50.0	400.0		1.2.3.123456.4.5.1234.1.12.0	
15	[1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000, ...	[-235.8665, -235.8665, 1756.00]	161	...	50.0	400.0		1.2.3.123456.4.5.1234.1.12.0	

DICOM Image

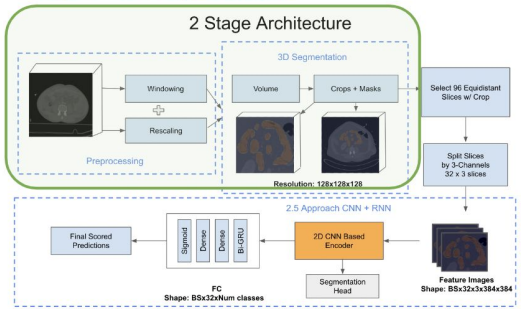


Image

RNSA 2023

Preprocessing

Stage 1 - 3D segmentation



Stage 2 - 2.5D CNN + RNN

❑ Rescaling

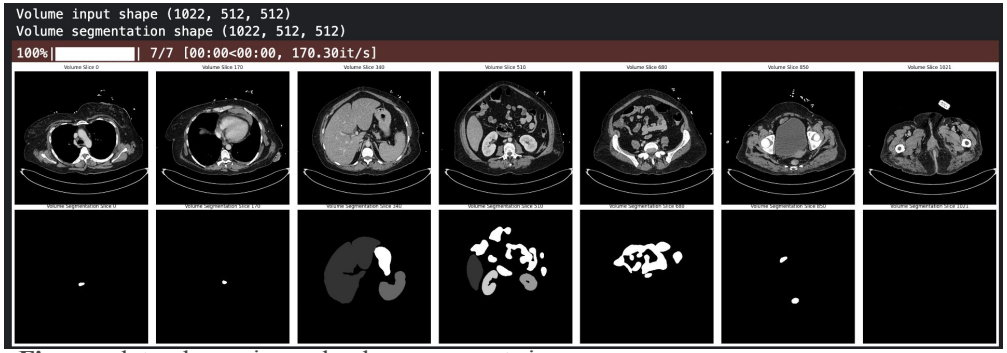
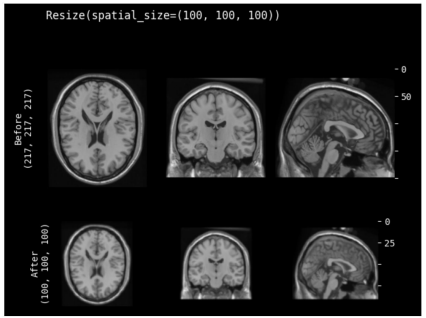


Figure: plot volume size and volume segment size

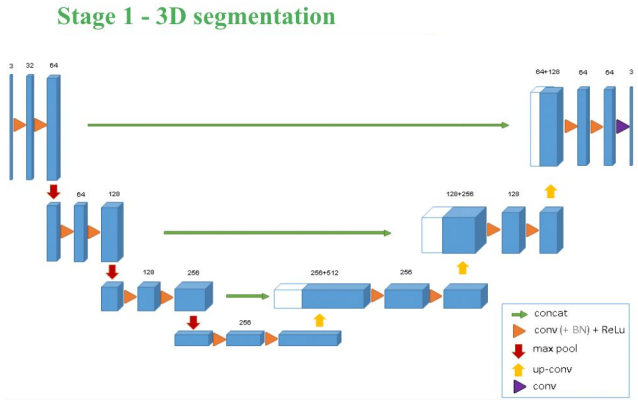



```
from monai.transforms import Resize
image_sizes = [128, 128, 128]
R = Resize(image_sizes)
```

Source: monai

RNSA 2023

3D segmentation




```python
class TimmSegModel(nn.Module):
    def __init__(self, backbone, segtype='unet', pretrained=False):
        super(TimmSegModel, self).__init__()

        self.encoder = timm.create_model(
            backbone,
            in_chans=3,
            features_only=True,
            drop_rate=drop_rate,
            drop_path_rate=drop_path_rate,
            pretrained=pretrained
        )
        g = self.encoder(torch.rand(1, 3, 64, 64))
        encoder_channels = [1] + [_.shape[1] for _ in g]
        decoder_channels = [256, 128, 64, 32, 16]
        if segtype == 'unet':
            self.decoder = smp.decoders.unet.decoder.UnetDecoder(
            #self.decoder = smp.unet.decoder.UnetDecoder(
                encoder_channels=encoder_channels[:n_blocks+1],
                decoder_channels=decoder_channels[:n_blocks],
                n_blocks=n_blocks,
            )

        self.segmentation_head = nn.Conv2d(
            decoder_channels[n_blocks-1],
            out_dim, kernel_size=(3, 3),
            stride=(1, 1), padding=(1, 1))

    def forward(self,x):
        global_features = [0] + self.encoder(x)[:n_blocks]
        seg_features = self.decoder(*global_features)
        seg_features = self.segmentation_head(seg_features)
        return seg_features
```

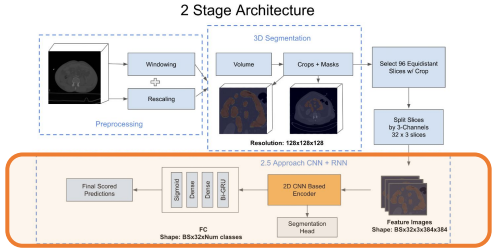
```python
backbone = 'resnet18d'
m = TimmSegModel(backbone)
m = convert_3d(m)
print(m(torch.rand(1, 3, 128,128,128)).shape)
```

```python
def convert_3d(module):

    module_output = module
    if isinstance(module, torch.nn.BatchNorm2d):
      ....

    elif isinstance(module, Conv2dSame):
        module_output = Conv3dSame(
            in_channels=module.in_channels,
            out_channels=module.out_channels,
            kernel_size=module.kernel_size[0],
            stride=module.stride[0],
            padding=module.padding[0],
            dilation=module.dilation[0],
            groups=module.groups,
            bias=module.bias is not None,
        )
        module_output.weight =
torch.nn.Parameter(module.weight.unsqueeze(-1).repeat(1,1,1,1,module.kernel_size[0]))
```

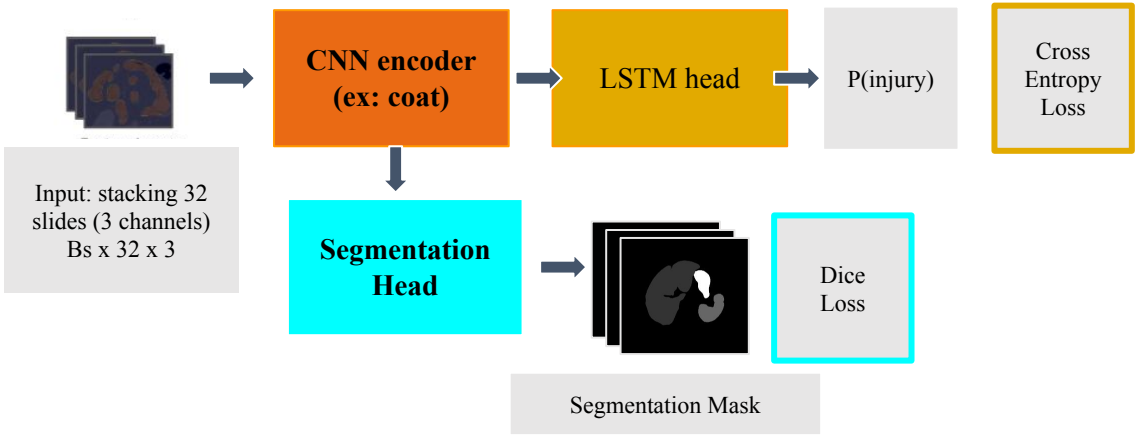
3D segmentation (note read the comment to adjust the code) :
https://www.kaggle.com/code/haqishen/rsna-2023-1st-place-solution-train-3d-seg/notebook

2.5D Model

❑ Feature extraction and LSTM head



Stage 2 - 2.5D CNN + RNN
Using 2D segmentation (auxiliary)

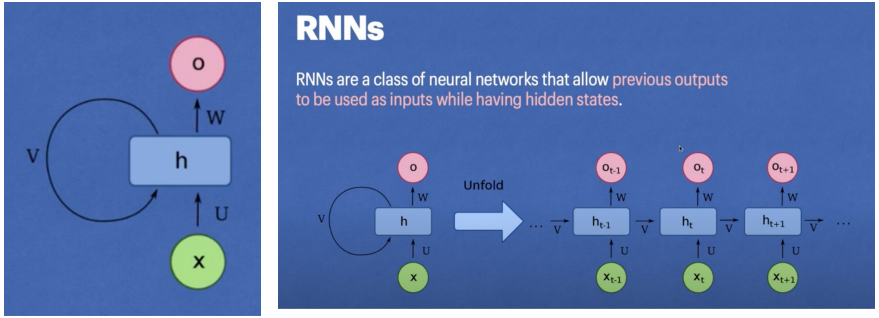


- **CNN model** can be replaced by ViT model (Swin, CoaT…)
- **RNN** can be replaced RNN series (RNN, LSTM) + Attention / Transformer Head

2.5D Model

❑ Review RNN/LSTM

RNN



Source: *PyTorch RNN Tutorial - Name Classification Using A Recurrent Neural Net*

Bidirectional RNN

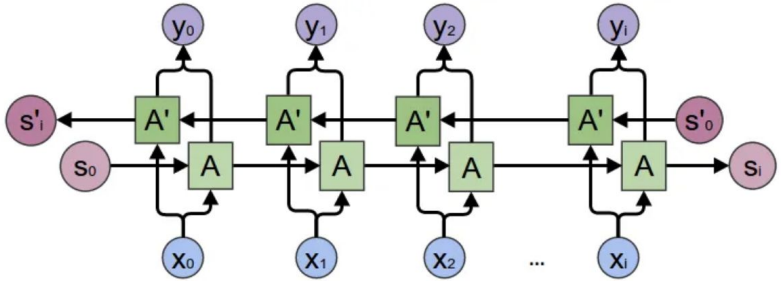
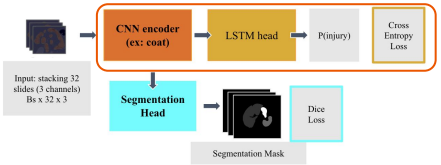


Fig 1: General Structure of Bidirectional Recurrent Neural Networks. Source: colah's blog

2.5D CNN + Sequences Model

❑ Feature extraction and LSTM head | Simple Step by Step version




```python
import timm
import torch
import torch.nn as nn


# Generate a dummy input tensor
input_tensor = torch.randn(2, 11, 3, 224, 224)
print("- Original input size:", input_tensor.size())


# Define model name and create encoder
name = 'coatnet_1_rw_224'
encoder = timm.create_model(
    name,
    pretrained=None,
    num_classes=0,
    global_pool="avg" if "coat" in name else "",
)

# Reshape input tensor for CNN
bs, n_frames, c, h, w = input_tensor.size()
x = input_tensor.view(bs * n_frames, c, h, w)
print("- Reshaped input for CNN:", x.shape,
      f" \n batch size = {bs * n_frames}",
      f"num channels = {c}, height and width is {h}x{w}")

# Pass the reshaped input through the encoder
features = encoder(x)
num_classes = 11

print("- Output after encoder:", features.shape)
```

```
- Original input size: torch.Size([2, 11, 3, 224, 224])
- Reshaped input for CNN: torch.Size([22, 3, 224, 224])
 batch size = 22 num channels = 3, height and width is 224x224
- Output after encoder: torch.Size([22, 768])
```

```python
# Define LSTM layer

lstm_layer = nn.LSTM(
    input_size=encoder.num_features,
    hidden_size=encoder.num_features // 4,
    # we will take the output mean and max -> combine
    batch_first=True,
    bidirectional=True
)
# Reshape features for LSTM input
features = features.view(bs, n_frames, -1)
print("- Reshaped features for LSTM:", features.shape)
print("- After compute mean: ", features.shape)
lstm_output, (hn, cn) = lstm_layer(features)
print("\n LSTM result shape")
print("+ Output size after LSTM layer:", lstm_output.shape)
```

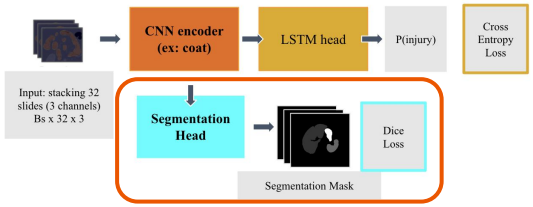
```
- Reshaped features for LSTM: torch.Size([2, 11, 768])
- After compute mean:  torch.Size([2, 11, 768])

 LSTM result shape
+ Output size after LSTM layer: torch.Size([2, 11, 384])
```

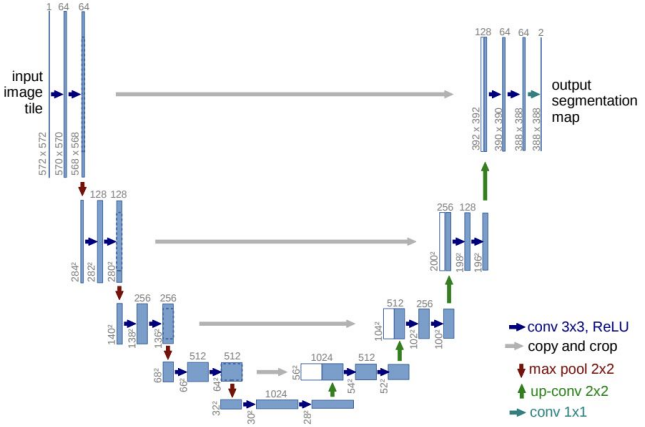
```python
mean_ = lstm_output.mean(1)
max_ = lstm_output.amax(1)
print("- Mean:", mean_.shape)
print("- Max:", max_.shape)
features_after_lstm = torch.cat([mean_, max_], -1)
print("- After LSTM mean max:", features_after_lstm.shape)

linear_layer = nn.Linear(encoder.num_features, 13)
cls_result = linear_layer(features_after_lstm)
print(" \n \n >> Final output logit", cls_result.shape)
```

```
- Mean: torch.Size([2, 384])
- Max: torch.Size([2, 384])
- After LSTM mean max: torch.Size([2, 768])


 >> Final output logit torch.Size([2, 13])
```

2.5D CNN + Sequences Model

Segmentation Head | Simple Step by Step version






```python
model_name = 'coatnet_1_rw_224'

model = timm.create_model(
    model_name,
    pretrained=True,
)


stem_input = model.stem(x)

layer1_fts = model.stages[0](stem_input)
layer2_fts = model.stages[1](layer1_fts)
layer3_fts = model.stages[2](layer2_fts)
layer4_fts = model.stages[3](layer3_fts)
print(features.shape)
print("stem:", stem_input.shape)
print("layer 1:", layer1_fts.shape)
print("layer 2:", layer2_fts.shape)
print("layer 3:", layer3_fts.shape)
print("layer 4:", layer4_fts.shape)
```

```
stem: torch.Size([22, 64, 112, 112])
layer 1: torch.Size([22, 96, 56, 56])
layer 2: torch.Size([22, 192, 28, 28])
layer 3: torch.Size([22, 384, 14, 14])
layer 4: torch.Size([22, 768, 7, 7])
```

```python
import torch
import torch.nn as nn

class UNetDecoderBlock(nn.Module):
    def __init__(self, in_channels, res_channels, out_channels):
        super(UNetDecoderBlock, self).__init__()
        self.upconv = nn.ConvTranspose2d(
            in_channels, out_channels, kernel_size=2, stride=2)
        self.conv = nn.Sequential(
            nn.Conv2d(
                out_channels + res_channels, out_channels, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(
                out_channels, out_channels, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip):
        x = self.upconv(x)
        x = torch.cat([x, skip], dim=1)
        x = self.conv(x)
        return x
```

```python
encoder_channels = [64, 96, 192, 384, 768]
decoder_channels = [384, 192, 96, 64]

unet_block_4 = UNetDecoderBlock(
    in_channels = 768, res_channels = 384, out_channels = 384)
unet_fts_4 = unet_block_4(layer4_fts, layer3_fts)
print("unet_fts_4: ", unet_fts_4.shape)

unet_block_3 = UNetDecoderBlock(
    in_channels = 384, res_channels = 192, out_channels = 192)
unet_fts_3 = unet_block_3(layer3_fts, layer2_fts)
print("unet_fts_3: ", unet_fts_3.shape)

unet_block_2 = UNetDecoderBlock(
    in_channels = 192, res_channels = 96 , out_channels = 96)
unet_fts_2 = unet_block_2(layer2_fts, layer1_fts)
print("unet_fts_2: ", unet_fts_2.shape)

unet_block_1 = UNetDecoderBlock(
    in_channels = 96, res_channels = 64 , out_channels =
unet_fts_1 = unet_block_1(layer1_fts, stem_input)
print("unet_fts_1: ", unet_fts_1.shape)

features_multiple_resolutions = [
    unet_fts_4, unet_fts_3, unet_fts_2, unet_fts_1]
```

```
torch.Size([22, 384, 14, 14])
unet_fts_4:  torch.Size([22, 384,
unet_fts_3:  torch.Size([22, 192,
unet_fts_2:  torch.Size([22, 96,
unet_fts_1:  torch.Size([22, 64,
```

2.5D CNN + Sequences Model

Segmentation Head - use FPN to combine features

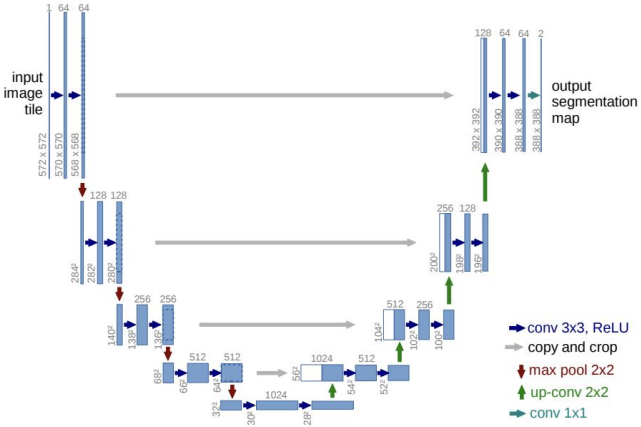
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class FPN(nn.Module):
    def __init__(self, input_channels: list, output_channels: list, mask_size = (112, 112)):
        super(FPN, self).__init__()
        self.mask_size = mask_size
        self.convs = nn.ModuleList(
            [nn.Sequential(
                nn.Conv2d(in_ch, out_ch * 2, kernel_size=3, padding=1), # 384, 64  (batch size =
2*11=22frames)
                nn.ReLU(inplace=True),
                nn.BatchNorm2d(out_ch * 2),  # Changed to BatchNorm2d
                nn.Conv2d(out_ch * 2, out_ch, kernel_size=3, padding=1) # 64, 32
            ) for in_ch, out_ch in zip(input_channels, output_channels)]
        )

    def forward(self, xs: list):
        # Scale feature maps to the same resolution
        hcs = [
            F.interpolate(c(x), size=self.mask_size, mode='bilinear', align_corners=False)
            for i, (c, x) in enumerate(zip(self.convs, xs))
        ]
        print(len(hcs))
        return torch.cat(hcs, dim=1)  # Concatenate along channel dimension
```

Segmentation Head - all together!

```python
class SegmentationHead(nn.Module):
    def __init__(self, encoder, mask_size=(112, 112)):
        super(SegmentationHead, self).__init__()
        self.encoder = encoder
        self.mask_size = mask_size

        if 'coat' in self.encoder.name:
            encoder_channels = [64, 96, 192, 384, 768]
            decoder_channels = [384, 192, 96, 64]
            output_channels = 32
            self.unet_blocks = nn.ModuleList([
                UNetDecoderBlock(encoder_channels[-(i + 1)], encoder_channels[-(i + 2)],
decoder_channels[i])
                for i in range(len(decoder_channels))
            ])
            self.fpn = FPN(decoder_channels, [output_channels * 4])

            current_channels = output_channels * 3
            num_layers = len(decoder_channels)  # Adjust as needed
            intermediate_channels = [current_channels // 2**i for i in range(num_layers)]
            layers = []
            for ic in intermediate_channels:
                layers.append(nn.Conv2d(current_channels, ic, kernel_size=3, stride=2, padding=1))
                layers.append(nn.BatchNorm2d(ic))
                layers.append(nn.ReLU(inplace=True))
                current_channels = ic
            layers.append(nn.Conv2d(current_channels, 1, kernel_size=1))
            self.seg_cnn_layers = nn.Sequential(*layers)

    def forward(self, intermediate_features):
        if 'coat' in self.encoder.name:
            layer4_encoder_feature, layer3_encoder_feature, layer2_encoder_feature,
layer1_encoder_feature, stem_input = intermediate_features

            unet_fts_4 = self.unet_blocks[3](layer4_encoder_feature, layer3_encoder_feature)
            unet_fts_3 = self.unet_blocks[2](layer3_encoder_feature, layer2_encoder_feature)
            unet_fts_2 = self.unet_blocks[1](layer2_encoder_feature, layer1_encoder_feature)
            unet_fts_1 = self.unet_blocks[0](layer1_encoder_feature, stem_input)

            segmentation_features = self.fpn([unet_fts_4, unet_fts_3, unet_fts_2, unet_fts_1])
            x = F.interpolate(
                segmentation_features, size=self.mask_size, mode='bilinear', align_corners=False)
            pred_masks = self.seg_cnn_layers(x)
            return pred_masks
        else:
            return None
```

2.5D CNN + Sequences Model

Segmentation Head - all together!




```python
class SegmentationHead(nn.Module):
    def __init__(self, encoder, mask_size=(112, 112)):
        super(SegmentationHead, self).__init__()
        self.encoder = encoder
        self.mask_size = mask_size

        if 'coat' in self.encoder.name:
            encoder_channels = [64, 96, 192, 384, 768]
            decoder_channels = [384, 192, 96, 64]
            output_channels = 32
            self.unet_blocks = nn.ModuleList([
                UNetDecoderBlock(encoder_channels[-(i + 1)], encoder_channels[-(i + 2)],
decoder_channels[i])
                for i in range(len(decoder_channels))
            ])
            self.fpn = FPN(decoder_channels, [output_channels * 4])

            current_channels = output_channels * 3
            num_layers = len(decoder_channels)  # Adjust as needed
            intermediate_channels = [current_channels // 2**i for i in range(num_layers)]
            layers = []
            for ic in intermediate_channels:
                layers.append(nn.Conv2d(current_channels, ic, kernel_size=3, stride=2, padding=1))
                layers.append(nn.BatchNorm2d(ic))
                layers.append(nn.ReLU(inplace=True))
                current_channels = ic
            layers.append(nn.Conv2d(current_channels, 1, kernel_size=1))
            self.seg_cnn_layers = nn.Sequential(*layers)

    def forward(self, intermediate_features):
        if 'coat' in self.encoder.name:
            layer4_encoder_feature, layer3_encoder_feature, layer2_encoder_feature,
layer1_encoder_feature, stem_input = intermediate_features

            unet_fts_4 = self.unet_blocks[3](layer4_encoder_feature, layer3_encoder_feature)
            unet_fts_3 = self.unet_blocks[2](layer3_encoder_feature, layer2_encoder_feature)
            unet_fts_2 = self.unet_blocks[1](layer2_encoder_feature, layer1_encoder_feature)
            unet_fts_1 = self.unet_blocks[0](layer1_encoder_feature, stem_input)

            segmentation_features = self.fpn([unet_fts_4, unet_fts_3, unet_fts_2, unet_fts_1])
            x = F.interpolate(
                segmentation_features, size=self.mask_size, mode='bilinear', align_corners=False)
            pred_masks = self.seg_cnn_layers(x)
            return pred_masks
        else:
            return None
```

2.5D CNN + Sequences Model

2.5D CNN + Sequence Model - all together!

```python
class _25DCnnRnnSegAux(nn.Module):
    '''
    2.5 D model : CNN model + RNN with Segmentation head
    '''
    def __init__(self,
                 encoder,
                 num_classes = 2,
                 num_classes_aux = 11,
                 n_channels = 3,
                 head_3d = "",
                 n_frames = 1,
                 dropout_rate=0,
                 segmentation_aux=True):
        super().__init__()
        #set up
        self.encoder = encoder
        self.num_features= encoder.num_features
        self.num_classes = num_classes
        self.num_classes_aux = num_classes_aux
        self.n_channels = n_channels
        self.head_3d = head_3d
        self.n_frames = n_frames
        self.dropout = nn.Dropout(p=dropout_rate) if dropout_rate > 0 else nn.Identity()
        # layers.
        if head_3d == 'lstm':
            self.lstm = nn.LSTM(input_size = self.num_features,
                                hidden_size = self.num_features // 4,
                                batch_first = True,
                                bidirectional=True)


        self.last_layer = nn.Linear(self.num_features, self.num_classes)

        if self.num_classes_aux > 0:
            if segmentation_aux:
                self.aux_layer = SegmentationHead(self.encoder)
            else:
                self.aux_layer = nn.Linear(self.num_features, self.num_classes_aux)
```

```python
    def extract_features(self, x):
        feature_output = self.encoder(x)
        intermediate_feature = None
        if self.segmentation_aux and 'coat' in self.encoder.name:
            layer0_encoder_feature = self.encoder.stem(x)
            layer1_encoder_feature = self.encoder.stages[0](layer0_encoder_feature)
            layer2_encoder_feature = self.encoder.stages[1](layer1_encoder_feature)
            layer3_encoder_feature = self.encoder.stages[2](layer2_encoder_feature)
            layer4_encoder_feature = self.encoder.stages[3](layer3_encoder_feature)

            intermediate_features = [
                layer4_encoder_feature,
                layer3_encoder_feature,
                layer2_encoder_feature,
                layer1_encoder_feature,
                layer0_encoder_feature
                ]

        fts = self.dropout(fts)
        return fts, intermediate_features

    def forward_head3d(self, x):
        if self.head_3d == 'lstm':
            x, _ = self.lstm(x)
            mean  = x.mean(1)
            max_ = x.amax(1)
            x = torch.cat([mean, max_], dim=1)
        return x
```

```python
    def forward(self, x):
        if self.head_3d:
            bs, n_frames, c, h , w  = x.size()
            x = x.view(bs*n_frames, c, h, w)

        fts, intermediate_features = self.extract_features(x)

        print("fts size: ", fts.shape)

        if self.head_3d != "":
            fts = fts.view(bs, n_frames, -1)
            fts = self.forward_head3d(fts)

        output = self.last_layer(fts)

        if self.num_classes_aux:
            segmentation_output = self.segmentation_head(intermediate_features)
        else:
            segmentation_output = None
        if self.num_classes_aux > 0:
            aux_output = self.aux_layer(fts)
        else:
            aux_output = torch.zeros((fts.size(0)))
        # processing volumn mask into slice -> input to the model

        return output, aux_output
```

```python
class CustomLoss(nn.Module):
    def __init__(self):
        super(CustomLoss, self).__init__()
        #self.bce = nn.BCEWithLogitsLoss(pos_weight=torch.as_tensor([2.318]).cuda())
        self.bce = nn.BCEWithLogitsLoss()
        self.dice = smp.losses.DiceLoss(smp.losses.MULTILABEL_MODE, from_logits=True)

    def forward(self, outputs, targets, masks_outputs, masks_targets):
        loss1 = self.bce(outputs, targets)

        masks_outputs = masks_outputs.float()
        # masks_outputs2 = masks_outputs2.float()

        masks_targets = masks_targets.float().flatten(0, 1)

        loss2 = self.dice(masks_outputs, masks_targets) #+ self.dice(masks_outputs2,


        loss = loss1 + (loss2 * CFG.segw)

        return loss
```

Upsampling layer (optional)

- **Upsampling layer: ConvTranspose2d, Interpolation or Pixel Shuffle**

ConvTranspose2d

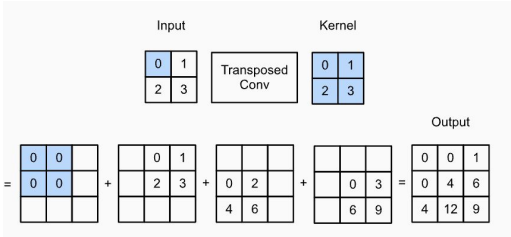


Figure: ConvTranspose2d 2x2
Src: D2l.ai

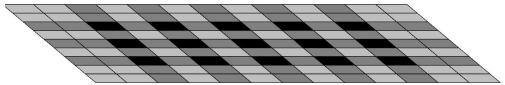
Using deconvolution.
Heavy checkerboard artifacts.

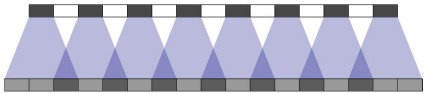
Artifact

Using resize-convolution.
No checkerboard artifacts.

Less Artifact

The overlap pattern also forms in two dimensions. The uneven overlaps on the two axes multiply together, creating a characteristic checkerboard-like pattern of varying magnitudes.

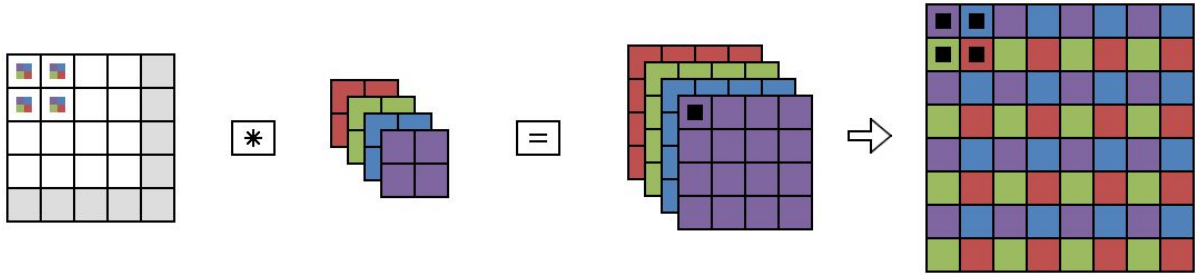




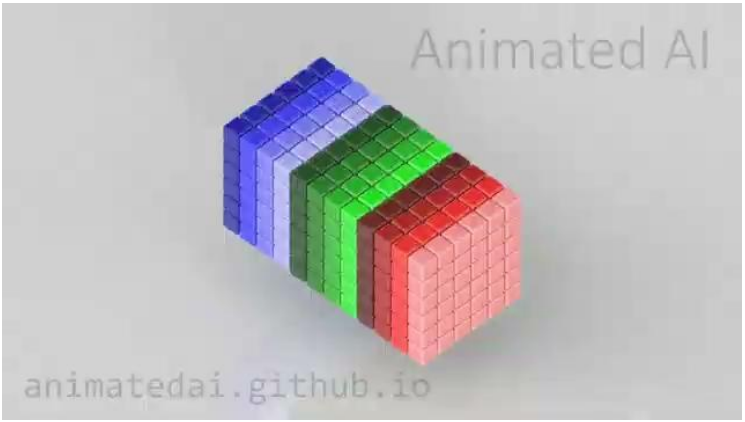
Source: https://distill.pub/2016/deconv-checkerboard/

Upsampling layer (optional)

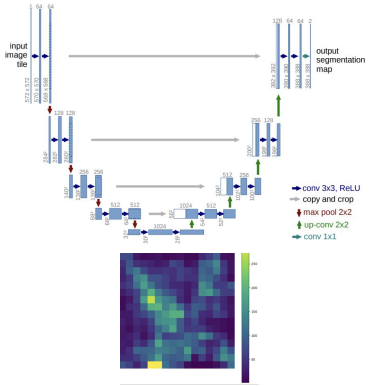
Pixel Shuffle



Source: https://stackoverflow.com/questions/72069125/is-convolution-pixelshuffle-the-same-as-subpixel-convolution



Upsampling layer (optional)



?

Upsampled Image (ConvTranspose2d)

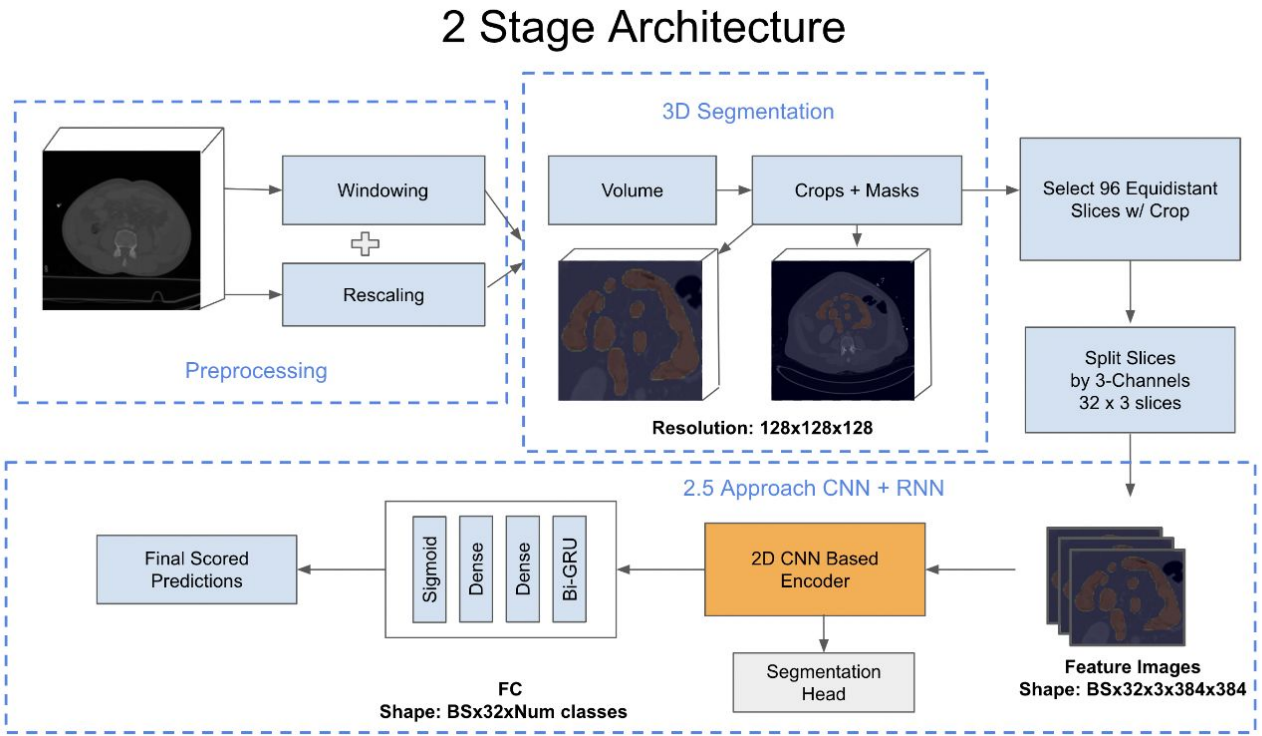
Upsampled Image (Interpolate)

Upsampled Image (Pixel Shuffle)

Upsampled Image (PixelShuffle with ICNR initialization)

Notebook Link: [vizualize] pixel-shuffle vs conv-transpose

Review pipeline



AI VIETNAM
All-in-One Course
(TA Session)

Summary

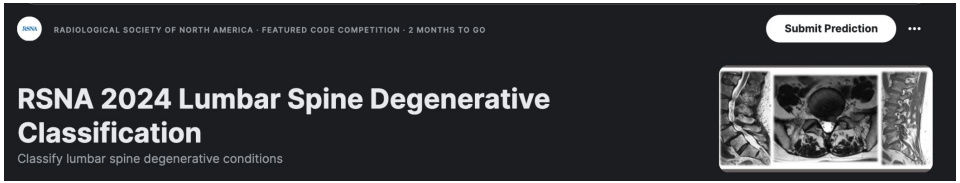
Volumetric data

3D CNN

3D Unet

Exploring RSNA with 3D Unet and 2.5D CNN + RNN

?



Thank you for watching !

Let's learn more together!