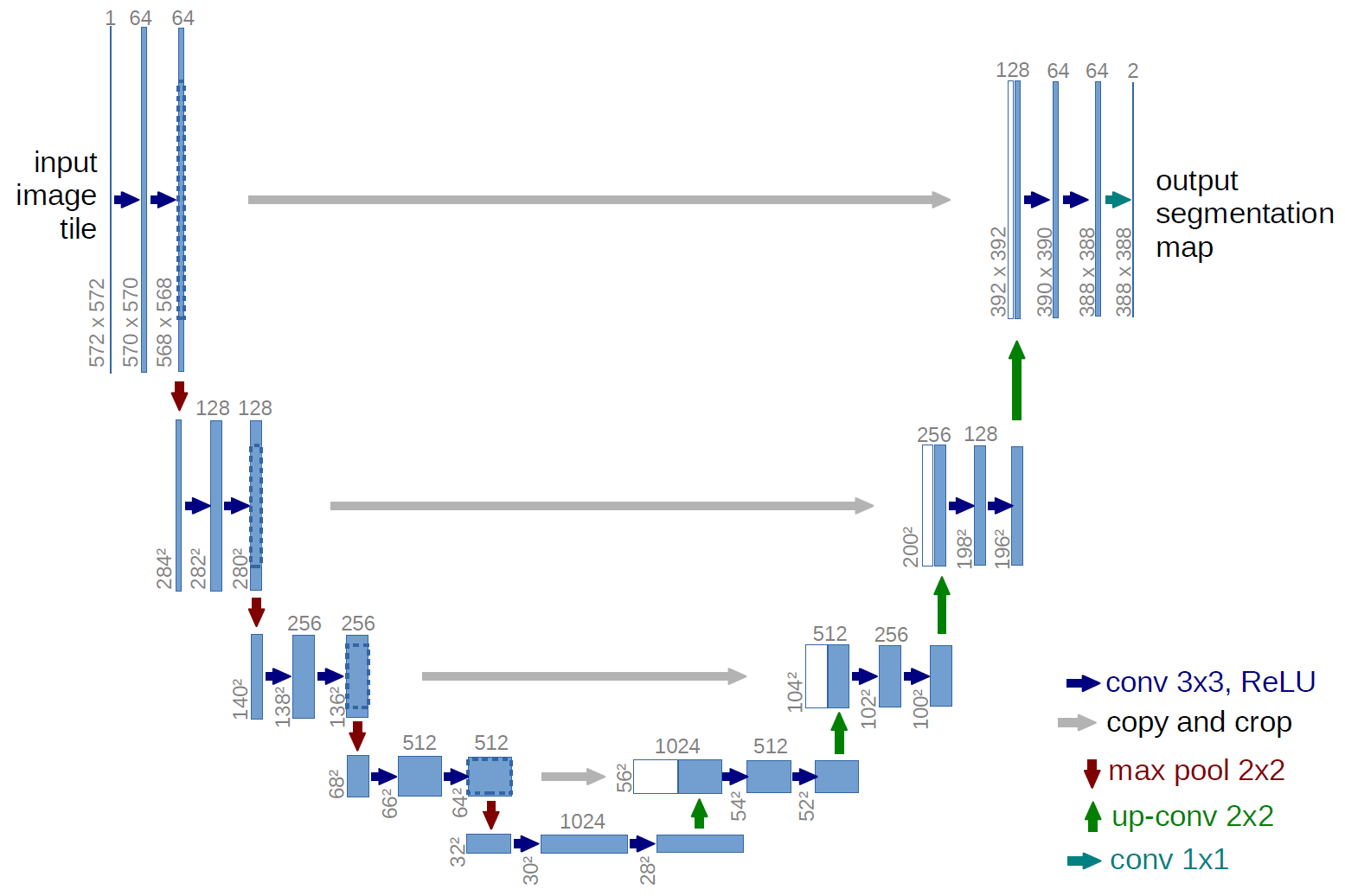
UNet Model Report

# General Description

UNets were originally developed for biomedical image segmentation by Ronneberger et al. This architecture works by downsampling (halving the size of the input size) the image repeatedly, called the encoding path, to capture context and then upsampling (doubling the size of the image again), called the decoding path, back up to the original resolution of the input image. Included below is an image of what a Unet looks like (although this one is for 2d images rather than 1d time series).



Information is transferred across the U (grey arrows) and also between layers (not pictured in this image). This flow of information, called skip connections, can help the model to consider information at many levels of abstraction. (Note to Dr. Cooper: if we decide it is worth it, we can produce an image ourselves that is more specific to our application).

The (en/de)coding paths both consist of blocks, followed by either an (down/up)sample denoted by (red/green) arrows in the diagram. Each block consists of two convolutions (recall, a convolution allows the network to apply a learned filter to the data) followed by some other steps to clean up the output (normalization and activation, plus potentially dropout). The blocks allow the network to process the information at that level of abstraction, and then it is (down/up) sampled to be processed at the next level of abstraction.

At the lowest point of the U (called the bottleneck), we can either apply another block or use a more complex transformer model. The transformer, developed by Vaswani et al, was originally developed for processing text. It is powerful because it allows the model to look at entire sequence at once using a mechanism called self-attention. However, even after being downsized through the encoding path of the UNet the data is still too large for a transformer to be run in a reasonable time. To remedy this, we use windowed attention (also sometimes called sliding window attention), similar to the implementation from Beltagy et al. This allows a UNet with windowed attention to be similarly fast to a UNet with a more standard block in the bottleneck.

UNets were originally designed for segmentation, which is very similar to the waveform labeling problem, giving them a strong theoretical reason to perform well. They are a particularly good choice for our task since they can look at the entire input sequence, avoiding some of the issue present in windowing like abruptly cutting our data. They offer an exponentially growing receptive field, similar to a TCN, and are reasonably fast to run.

Our implementation differs from the standard UNet primarily because our problem is a 1d time series, as opposed to the 2d images UNets were originally designed for. For this reason, we recommend a growth factor of 1 both theoretically (explained in the architecture section) and empirically, as opposed to the more standard 2 in image segmentation problems. There are several low-level issues introduced by variable input sizes, but these are fairly straightforward and discussed in the technical notes.

# Model

The architecture, at a high level, is:

* in\_conv, from the size of the input data (1) to *starting\_features* size, with kernel size 1
* *num\_layers* EncoderBlocks, each followed by a MaxPool (to downasmple) with *up\_down\_sample\_kernel\_size* and stride=2
* an appropriate *bottleneck\_type* (either a block or WindowedAttention)
* *num\_layers* DecoderBlocks, each preceded by a ConvTranspose1d (to upsample) with both kernel size and stride *up\_down\_sample\_kernel\_size*
* out\_conv, from the size of the *starting\_features* size to the number of classes (6), with kernel size 1

Each block consists of *n\_conv\_steps\_per\_block* repetitions of: convolution, InstanceNorm, GELU, Dropout (only in the encoder).

There are skip connections across the U and between layers (before and after the block).

We also leave the capability for the user to select a *growth\_factor* if they want the amount of information in the UNet to decrease as it reaches the bottleneck (which can be desirable if they want the UNet to compress information more), by setting *growth\_factor*=1. We empirically find that *growth\_factor*=1 (compressing information) performs better than *growth\_factor*=2 (not compressing information).

# References

Vaswani, Ashish, et al. *"Attention Is All You Need."* 2017. *arXiv*,<http://arxiv.org/abs/1706.03762>.

Beltagy, Iz, et al. *Longformer: The Long-Document Transformer.* 2020, arXiv,<https://arxiv.org/abs/2004.05150>.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation.” *arXiv*, 2015, arXiv:1505.04597,<https://arxiv.org/abs/1505.04597>.

Akiba, Takuya, et al. "Optuna: A Next-generation Hyperparameter Optimization Framework." *arXiv* (2019), arXiv:1907.10902.<https://arxiv.org/abs/1907.10902>.

# Technical Notes (for appendix)

## Architecture

class Model():

def \_\_init\_\_(self, epochs, lr, num\_layers, growth\_factor, features, n\_conv\_steps\_per\_block, block\_kernel\_size, up\_down\_sample\_kernel\_size, block\_padding, weight\_decay, dropout\_rate, bottleneck\_type, transformer\_window\_size, embed\_dim, transformer\_layers, transformer\_nhead):

The architecture, at a high level, is:

* in\_conv, from the size of the input data (1) to *starting\_features* size, with kernel size 1
* *num\_layers* EncoderBlocks, each followed by a MaxPool with *up\_down\_sample\_kernel\_size* and stride=2
* an appropriate *bottleneck\_type* (either a block or WindowedAttention)
* *num\_layers* DecoderBlocks, each preceded by a ConvTranspose1d with both kernel size and stride *up\_down\_sample\_kernel\_size*
* out\_conv, from the size of the *starting\_features* size to the number of classes (6), with kernel size 1

Additionally, in the decoding path at each step we take the corresponding output from the encoding path, pad it up to the same size if needed, and add it for the skip connection across the U. We take the output after Encoding on the i-th level of the encoder, and add it before the Decoder on the i-th level of the decoder.

If the bottleneck is a standard block, then it is an Encoder/Decoder block without any up/down sampling and no dropout. If the bottleneck is WindowedAttention, that is described in detail in its own section below.

Note that

## Encoder/Decoder Block

class EncoderBlock(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels, n\_conv\_steps\_per\_block, dropout\_rate, block\_kernel\_size, block\_padding, growth\_factor):

There is one block of conv(*in\_channels*,*out\_channels*), Instancenorm, GELU, Dropout followed by *n\_conv\_steps\_per\_block-1* blocks of conv(*out\_channels*,*out\_channels*), Instancenorm, GELU, Dropout. That is, if in\_channels != out\_channels then the first conv\_step goes from in\_channels to out\_channels and the rest go from out\_channels to out\_channels. The conv layers all have *block\_kernel\_size* and *block\_padding*

*growth\_factor* is the parameter that controls the multiple by which the number of channels grows by each step up/down in the Unet. If *growth\_factor* is 1, then do an additive skip connection from before the EncoderBlock to after. If *growth\_factor* is not 1 (2), then do no skip since the shapes are complicated.

The DecoderBlock is the same, except no dropout is included.

## Windowed Attention Bottleneck

WindowedTransformerBotleneck(nn.Module):

def \_\_init\_\_(self, in\_channels, window\_size, embed\_dim, transformer\_layers, nhead)

First, we pad the input sequence up to be a multiple of *transformer\_window\_size*, then cut into windows. After, the added padding is cut out to get a sequence that is the same size as the input sequence.

If *in\_channels* != *embed\_dim*, then a linear layer is used before and after the transformer to project in\_channels into *embed\_dim*, and then out again after the transformer to be used downstream in the UNet.

Positional encoding are added to the windowed data, and then the transformer itself is run with appropriate *transformer\_layers* and *transformer\_nhead*.

## Training

The model is trained with Adam with CrossEntropy loss, using the specified *lr* and *weight\_decay* and run for *epochs*.

As with all other models, the data is first split into probes.

## Parameter Search

Parameter search was performed with Optuna (Akiba et al) for 100 trials over the following space:

epochs = [8, 16, 32, 64]

lr = [5e-3, 5e-4, 5e-5, 5e-6]

dropout\_rate = [0, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

weight\_decay = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4]

num\_layers = [2, 4, 6, 8]

features = [8, 16, 32, 64]

With attention:

transformer\_window\_size = [50, 100, 150, 200, 250]

embed\_dim = [features] # force to be same size as features

transformer\_layers = [2,4,6]

transformer\_channels\_per\_nhead = [16,32]

## Best Parameters

The best parameters found by this search were:

* Without attention: {'epochs': 64, 'lr': 0.0005, 'dropout\_rate': 0.1, 'weight\_decay': 1e-06, 'num\_layers': 8, 'features': 32}
* With attention: {'epochs': 64, 'lr': 0.0005, 'dropout\_rate': 1e-05, 'weight\_decay': 1e-05, 'num\_layers': 8, 'features': 64, 'transformer\_window\_size': 150, 'transformer\_layers': 2, ‘nhead’:2}