The Ico-Unet

Kilian Liss

December 13, 2022

1 Context

1.1 Graph Neural Networks (GNN) in the Physical Sciences

Meshes are very popular for representing 3D objects digitally. They consist of vertices, edges and faces (or cells), and so they are fundamentally a graph with additional attributes. Simulations in the physical sciences commonly operate on mesh type data, however, most algorithms are domain specific, and computationally slow.

Machine learning methods have been used in previous works to greatly improve the run time of physical simulations. Pfaff et. al. [2] used GNNs to simulate cloths, structural mechanics, and fluid mechanics with promising results. They also claim that their GNN model runs 1-2 orders of magnitudes faster compared to the simulations that they were trained on, whilst also being capable of generalizing and scaling. Similarly, Gonzalez et. el [3] used GNNs to simulate water, goop and sand, which can also simulate orders of magnitude more particles simultaneously compared to the simulation it was trained on. Note that in both of their works [2, 3], it is safe to assume that the dynamics of vertices can be predicted using only data obtained from neighbouring vertices, and so it is sufficient to use a few graph convolutions and linear layers to get great results.

1.2 The Traditional Unet Model

The traditional Unet model is a fully convolutional neural network architecture that takes grid structured data as inputs and outputs. It has commonly been used for image to image generative tasks, such as image segmentation [4], and are also commonly used in the context of Generative Adversarial Networks (GANs) [5, 6] where the loss function is a classifier network attempting to distinguish real images from generated ones, which the generator (Unet) is trying to fool.

The Unet model itself is has an encoder and decoder stage. The encoder applies a few image convolutions, followed by a pooling stage that reduces the resolution of the image, which is repeated until the bottom of the Unet. The decoder uses unpooling to increase the image resolution, concatenates it with

the output of the encoder of the same depth, applies a few more convolutions, and this process is repeated until we reach the original input image resolution.

1.3 GOSPL

GOSPL [1] is a python numerical model that simulates landscape dynamics over the course of millions of years, which operates on an icospherical mesh. As inputs, it takes initial elevations, tectonic plate movements, rainfall and tectonic uplift as vertex attributes, and it outputs elevations, soil deposition and water flow accumulation. It is worth investigating if we can use machine learning methods to also significantly improve the run-time of GOSPL simulations.

2 Problem Statement

In our previous works, we have used the traditional Unet architecture to replicate GOSPL simulations, with significant performance improvements. Our traditional Unet model can replicate 60 simulation iterations in 1 second that would otherwise take 10 minutes to run. Since the traditional Unet model can only take grid like data as inputs, we had to interpolate our training data to a UV sphere, and unwrap it as a preprocessing step. However, this model can not scale to larger resolutions.

Since GNN models can take any graph (or mesh) as inputs during training and evaluation, it is worth investigating whether they can replicate GOSPL and generalize to a variety of resolutions. If a high resolution icosphere is too large to be stored on GPU memory during training, we could perhaps break it into chunks and still train it on the very same model that is also concurrently learning on lower icosphere resolutions.

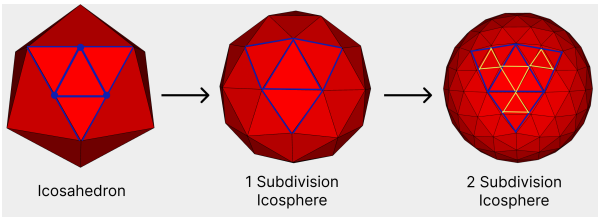
Unlike the above mentioned works [2, 3], it is not safe to assume that the dynamics of vertex attributes can be predicted based on neighbouring vertices alone. The change of elevation at vertices depends on how much soil is being eroded or deposited, which depends on the water flow, which depends depends on the topology and rainfall at distant vertices, and so simple graph convolutions and linear layers alone may not be sufficient.

This led to the idea of replicating the Unet model, but using GNNs on an icosphere. If we can get this architecture to work well, then the very same model could be capable of operating on any mesh that has been created by means of subdivisions, which can be applicable in many fields of study outside of this work.

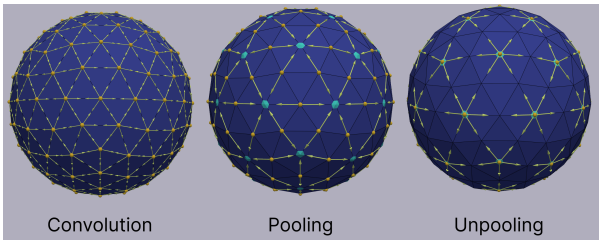
3 Methods

3.1 Generating an Icosphere

To understand how we can build a Unet like model using GNNs on an icosphere, we first have to understand how the icosphere is generated in the first place,



(a) The subdivisioning of icospheres. New vertices are added to the center of each edge during subdivision.



(b) The message passing involved for convolution, pooling and unpooling operations. The cyan dots represent vertices from a lower subdivision icosphere, and the arrows represent directional edges used for message passing.

Figure 1

which is illustrated in figure 1a. The algorithm begins with an icosahedron, where the vertices and edges are specified manually. New vertices are created on the centres of each edge, and their position is adjusted based on the radius of the sphere. New edges are then added to form triangles as shown in figure 1a, and the process is repeated for each subdivision of the icosphere. Notice that for every new vertex in a subdivision, it has exactly two neighbouring vertices that belonged to the previous subdivision icosphere. This property will allow the use of message passing to form the pooling and unpooling layers required by the Unet architecture.

3.2 The General Ico-Unet

In order to replicate the traditional Unet model, we will need to replicate layers for convolutions, pooling and unpooling, which is illustrated in figure 1b. Image convolutions can easily be replicated with bidirectional graph convolutions. To replicate pooling, we pass directional messages from all vertices to their nearest vertex belonging to a lower subdivision icosphere. To replicate unpooling, we pass directional messages from vertices belonging from the lower subdivision icosphere outwards.

To form the encoding stage of the Ico-Unet, we perform a few graph convolutions, pool messages to a lower level icosphere and repeat until we have reached

the bottom depth of the Unet. To form the decoding stage, we unpool messages to higher subdivision icospheres, concatenate embeddings from the encoder of the same depth, perform a few graph convolutions and repeat until we have reached the original subdivision icosphere.

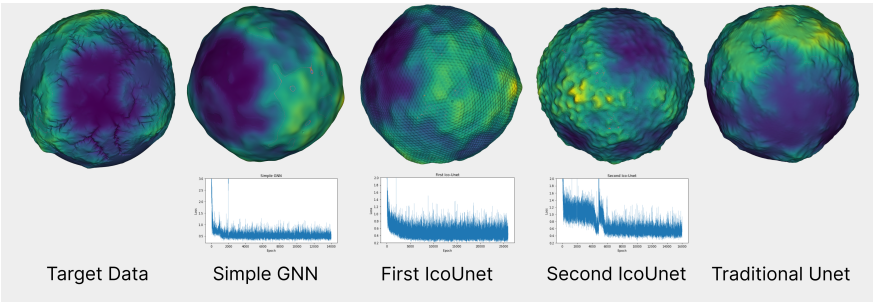


Figure 2: Comparing the resulting terrains of models with target terrains. A plot of training loss is also included where applicable

3.3 GNN Models

We build multiple GNN models with different architectures and compare their performances on our GOSPL training data set. In all our models, the input data will contain 5 features, and 3 output target features for each vertex in the mesh. The input features are (x, y, z, h_i, μ), where (x, y, z) are the cartesian coordinates of a unit radius icosphere, h_i is the initial heightmap of the landscape in meters above sea level, and μ is the tectonic uplift in meters per year. The target features are (h_f, d, a) where h_f is the final heightmap, d is the soil deposition and a is the water flow accumulation. All our data is obtained after running GOSPL for 60 million years, at time steps of 1 million years, however to keep things simple we will only be using the first and last iteration as training data.

All GNN models will start with a trainable feature normalization layer, and are trained with the Adam optimizer. ReLU activations are used because they do not pose an upper or lower bound on the layer's outputs. A custom loss function is used which first normalizes each output feature to a normal distribution and then applies the Mean Square Error loss. All models were trained with 2000 training examples.

Note that the list of vertices used to represent icospheres just so happened to be ordered based on which subdivision icosphere they belong to. This made it easy to identify the vertice's subdivisions by simply using vertex list indices.

Code for generating just the models alone is provided in the appendix, for the entire project code, please see the attached ZIP files to this assignment's submission.

3.4 Simple GNN

To compare our Unet model with already existing architectures, we have built a simple GNN model using graph convolution layers. The simple GNN is composed of a trainable normalization layer, 4 graph convolution layers with ReLU activations, followed by 2 fully connected layers where only the first has ReLU activations.

All layers had 128 channels as inputs and outputs, except for the first which had an input dimension of 5, and the last, which had an input dimension of 3.

3.5 First Ico-Unet

In our first attempt of the Ico-Unet, all our message passing was done with bidirectional graph convolutions. Each down convolution consists of a single bidirectional graph convolution and ReLU activations. Slicing of embedding tensors was used to make them compatible with lower level icospheres, which takes advantage of the fact that the list of vertices just so happened to be ordered based on which subdivision they belong to.

In the up-convolution layers, the embeddings tensors were padded with zeros so that their dimensions match the higher subdivision icospheres. The embedding of the encoder of the same depth was then concatenated along the embedding dimension, which was then passed to another graph convolution layer. Similarly to before, apart from the first and last layers, all layers had dimensions of 128 channels.

3.6 Second Ico-Unet

In our second attempt of the Ico-Unet, we introduced directional message passing for pooling and unpooling. We created two custom layers for the down and up convolutions. The down convolution layers had two bidirectional graph convolutions followed by ReLU and dropouts, and one directional graph convolution to pool data to a lower subdivision icosphere. The up convolution layers use zero padding, followed by outwards graph convolutions for unpooling, followed by concatenation of embedding from the encoding stage, followed by two more bidirectional graph convolutions.

The main model made use of the two custom layers, with 6 down convolution layers, and 6 up convolution layers and one last bidirectional graph convolution. Again, most layers had 128 channels each, apart from the first and last layers.

4 Results

Figure 2 shows the terrains of the target data, examples of the output of our models, and the terrain of results using the traditional Unet model. Plots of the training loss is also shown where applicable.

4.1 Simple GNN

The simple GNN did a decent job at approximating the general heightmap of the target terrain, however it's output is much too smooth, and resembles the shape of the input tectonic uplift data. From the training loss, we can see that the simple GNN initially learnt fairly quickly, however it also stopped learning after around 2000 epochs. The resulting loss was also fairly noisy and unstable, which indicates that it did not learn well.

4.2 First Ico-Unet

The first ico-Unet actually performed worse than the simple GNN. The resulting elevations were significantly affected by whether or not they were present in the first two depths of the Unet. This lead to the idea that pooling and unpooling layers should probably be using directional message passing.

4.3 Second Ico-Unet

The results of our second Ico-Unet seem to fix the issue of heights being significantly effected by which subdivision the vertices belong too. The resulting terrain is significantly less flat compared to the simple GNN, but it also failed at properly replicating the target data.

4.4 The traditional Unet

I've been avoiding showing my results of the traditional Unet, as it is part of my primary research project. Nonetheless, the resulting terrain is included in figure 2 to show that it is indeed possible to replicate GOSPL using the Unet architecture. Here the data had to be interpolated onto a UV sphere, which was then unwrapped to make grid structured data.

5 Conclusion

Although we have not managed to get great results with our Ico-Unet, there are still various improvements that could be made to better replicate the traditional Unet architecture. Instead of using directional graph convolutions for pooling and unpooling, we could better replicate the max pooling and unpooling operations. Instead of constantly using 128 channels in each layers, it might be better to gradually increase the channels the deeper you go into the Unet. Perhaps we could also improve the convolution function itself to more closely resemble image convolutions.

If we could get this model to work well, it's applications would be much broader than just replicating GOSPL. Our Unet model takes edges as inputs, and so we could dynamically change the resolution of data during training and evaluation. The model could also be used on any mesh that has been generated

by means of subdivisions, as long as we keep track of which subdivision each vertex belongs to.

References

[1] Salles Tristan, Mallard Claire and Zahirovic Sabin: *GOSPL: Global Scalable Paleo Landscape Evolution* Journal of Open Source Software, 2020

[2] Tobias Pfaff Meire Fortunato, Alvaro Sanchez-Gonzalez, Peter Battaglia: *Learning mesh-based simulation with Graph Networks*' International Conference on Learning Representations (ICLR), 2021

[3] Alvaro Sanchez Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, Peter W. Battaglia *Learning to Simulate Complex Physics with Graph Networks*, CoRR, abs/2002.09405, 2020

[4] Olaf Ronneberger, Philipp Fischer, Thomas Brox *U-Net: Convolutional Networks for Biomedical Image Segmentation*, CoRR, 1505.04597, 2015

[5] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros *Image-to-Image Translation with Conditional Adversarial Networks* CoRR, 1611.07004, 2016

[6] Guérin, Éric, Julie Digne, Éric Galin, Adrien Peytavie, Christian Wolf, Bedrich Benes, and Benoît Martinez. *Interactive Example-Based Terrain Authoring with Conditional Generative Adversarial Networks.* ACM Transactions on Graphics 36, no. 6 (2017): 1–13.

6 Appendix: Summary of Models

6.1 Simple GNN

```
# GCN Model
class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Layers
        self.norm = nn.BatchNorm1d(5)
        self.relu = nn.ReLU()
        self.conv1 = gnn.GCNConv(5, 128)
        self.conv2 = gnn.GCNConv(128, 128)
        self.conv3 = gnn.GCNConv(128, 128)
        self.conv4 = gnn.GCNConv(128, 128)
        self.lin1 = nn.Linear(128, 128)
        self.lin2 = nn.Linear(128, 128)
        self.lin3 = nn.Linear(128, 3)
```

```
    # Forward pass of GNN
    def forward(self, x, edge_index):
        h = self.norm(x)
        h = self.conv1(h, edge_index)
        h = h.relu()
        h = self.conv2(h, edge_index)
        h = h.relu()
        h = self.conv3(h, edge_index)
        h = h.relu()
        h = self.conv4(h, edge_index)
        h = h.relu()
        h = self.lin1(h)
        h = h.relu()
        out = self.lin3(h)
        return out
```

6.2 First Ico-Unet

```
def __init__(self, vertsLengths):
        super().__init__()
        self.norm = nn.BatchNorm1d(5)

        # Downwards convolution layers
        self.conv0 = gnn.GCNConv(128, 128)
        self.conv1 = gnn.GCNConv(128, 128)
        self.conv2 = gnn.GCNConv(128, 128)
        self.conv3 = gnn.GCNConv(128, 128)
        self.conv4 = gnn.GCNConv(128, 128)
        self.conv5 = gnn.GCNConv(128, 128)
        self.conv6 = gnn.GCNConv(5, 128)

        # Padding layers
        self.pad0 = nn.ConstantPad2d((0, 0, 0, vertsLengths[1] - vertsLengths[0]), 0)
        self.pad1 = nn.ConstantPad2d((0, 0, 0, vertsLengths[2] - vertsLengths[1]), 0)
        self.pad2 = nn.ConstantPad2d((0, 0, 0, vertsLengths[3] - vertsLengths[2]), 0)
        self.pad3 = nn.ConstantPad2d((0, 0, 0, vertsLengths[4] - vertsLengths[3]), 0)
        self.pad4 = nn.ConstantPad2d((0, 0, 0, vertsLengths[5] - vertsLengths[4]), 0)
        self.pad5 = nn.ConstantPad2d((0, 0, 0, vertsLengths[6] - vertsLengths[5]), 0)

        # Upwards convolution layers
        self.upConv0 = gnn.GCNConv(256, 128)
        self.upConv1 = gnn.GCNConv(256, 128)
        self.upConv2 = gnn.GCNConv(256, 128)
        self.upConv3 = gnn.GCNConv(256, 128)
        self.upConv4 = gnn.GCNConv(256, 128)
```

```
        self.upConv5 = gnn.GCNConv(256, 128)
        self.upConv6 = gnn.GCNConv(256, 3)

    # Forward pass of GNN
    def forward(self, x, vertsLengths, edgeDict):

        # Down convolution layers
        h = self.norm(x)
        h6 = self.conv6(h, edgeDict[6])
        h6 = h6.relu()

        h5 = h6[0:vertsLengths[5]]
        h5 = self.conv5(h5, edgeDict[5])
        h5 = h5.relu()

        h4 = h5[0:vertsLengths[4]]
        h4 = self.conv4(h4, edgeDict[4])
        h4 = h4.relu()

        h3 = h4[0:vertsLengths[3]]
        h3 = self.conv3(h3, edgeDict[3])
        h3 = h3.relu()

        h2 = h3[0:vertsLengths[2]]
        h2 = self.conv2(h2, edgeDict[2])
        h2 = h2.relu()

        h1 = h2[0:vertsLengths[1]]
        h1 = self.conv1(h1, edgeDict[1])
        h1 = h1.relu()

        h0 = h1[0:vertsLengths[0]]
        h0 = self.conv0(h0, edgeDict[0])
        h0 = h0.relu()

        # Up convolution layers
        h0Padded = self.pad0(h0)
        h1 = torch.cat((h1, h0Padded), 1)
        h1 = self.upConv1(h1, edgeDict[1])
        h1 = h1.relu()

        h1Padded = self.pad1(h1)
        h2 = torch.cat((h2, h1Padded), 1)
        h2 = self.upConv2(h2, edgeDict[2])
        h2 = h2.relu()
```

```
        h2Padded = self.pad2(h2)
        h3 = torch.cat((h3, h2Padded), 1)
        h3 = self.upConv3(h3, edgeDict[3])
        h3 = h3.relu()

        h3Padded = self.pad3(h3)
        h4 = torch.cat((h4, h3Padded), 1)
        h4 = self.upConv4(h4, edgeDict[4])
        h4 = h4.relu()

        h4Padded = self.pad4(h4)
        h5 = torch.cat((h5, h4Padded), 1)
        h5 = self.upConv5(h5, edgeDict[5])
        h5 = h5.relu()

        h5Padded = self.pad5(h5)
        h6 = torch.cat((h6, h5Padded), 1)
        h6 = self.upConv6(h6, edgeDict[6])
        return h6
```

6.3 Second Ico-Unet

```
# Define down convolution layer
class DownConvLayer(torch.nn.Module):
    def __init__(self, dims=[128, 128, 128]):
        super().__init__()

        # Layers
        self.conv1 = gnn.GCNConv(dims[0], dims[1])
        self.conv2 = gnn.GCNConv(dims[1], dims[1])
        self.conv3 = gnn.GCNConv(dims[1], dims[2])

    # Forward pass of the layer
    def forward(self, x, biEdges, inEdges, numVertsInLowerSub):

        # Bidirectional Convolution
        h = self.conv1(x, biEdges)
        h = h.relu()
        h = dropout(h, p=0.5, training=self.training)

        # Another bidirectional convolution
        h = self.conv2(h, biEdges)
        h = h.relu()
        h = dropout(h, p=0.5, training=self.training)

        # Inwards convolution (pooling)
```

```
        hPool = self.conv3(h, inEdges)
        hPool = hPool.relu()
        hPool = hPool[:numVertsInLowerSub]
        #hPool = dropout(hPool, p=0.5, training=self.training)
        return h, hPool

# Define down convolution layer
class UpConvLayer(torch.nn.Module):
    def __init__(self, paddingSize, dims=[128, 128, 128]):
        super().__init__()

        # paddingSize = vertsLengths[1] - vertsLengths[0]

        # Layers
        self.conv1 = gnn.GCNConv(dims[0], dims[0])
        self.conv2 = gnn.GCNConv(2*dims[0], dims[1])
        self.conv3 = gnn.GCNConv(dims[1], dims[2])
        self.pad = nn.ConstantPad2d((0, 0, 0, paddingSize), 0)

    # Forward pass of the layer
    def forward(self, x, xSkip, biEdges, outEdges):

        # Outwards convolution (unpooling)
        h = self.pad(x)
        h = self.conv1(h, outEdges)
        h = h.relu()
        h = dropout(h, p=0.5, training=self.training)

        # Convolution layer
        h1 = torch.cat((h, xSkip), 1)
        h1 = self.conv2(h1, biEdges)
        h1 = h1.relu()
        h1 = dropout(h1, p=0.5, training=self.training)

        # Another convolution layer
        h1 = self.conv3(h1, biEdges)
        h1 = h1.relu()
        return h1

# Define down convolution layer
class IcoUnet(torch.nn.Module):
    def __init__(self, vertsLengths):
        super().__init__()

        dim = 128
        l = vertsLengths
```

```
        self.norm = nn.BatchNorm1d(5)

        # Down convolution layers
        self.downConv6 = DownConvLayer(dims=[5, dim, dim])
        self.downConv5 = DownConvLayer(dims=[dim, dim, dim])
        self.downConv4 = DownConvLayer(dims=[dim, dim, dim])
        self.downConv3 = DownConvLayer(dims=[dim, dim, dim])
        self.downConv2 = DownConvLayer(dims=[dim, dim, dim])
        self.downConv1 = DownConvLayer(dims=[dim, dim, dim])

        # Upcovolution layers
        self.upConv1 = UpConvLayer(l[1] - l[0], dims=[dim, dim, dim])
        self.upConv2 = UpConvLayer(l[2] - l[1], dims=[dim, dim, dim])
        self.upConv3 = UpConvLayer(l[3] - l[2], dims=[dim, dim, dim])
        self.upConv4 = UpConvLayer(l[4] - l[3], dims=[dim, dim, dim])
        self.upConv5 = UpConvLayer(l[5] - l[4], dims=[dim, dim, dim])
        self.upConv6 = UpConvLayer(l[6] - l[5], dims=[dim, dim, dim])

        self.lastConv = gnn.GCNConv(dim, 3)

    # Forward pass of IcoUnet model
    def forward(self, x, biEdges, inEdges, outEdges, vertsLengths):
        h = self.norm(x)

        # Encoder layers (down convolutions)
        hSkip6, h = self.downConv6(h, biEdges[6], inEdges[6], vertsLengths[5])
        hSkip5, h = self.downConv5(h, biEdges[5], inEdges[5], vertsLengths[4])
        hSkip4, h = self.downConv4(h, biEdges[4], inEdges[4], vertsLengths[3])
        hSkip3, h = self.downConv3(h, biEdges[3], inEdges[3], vertsLengths[2])
        hSkip2, h = self.downConv2(h, biEdges[2], inEdges[2], vertsLengths[1])
        hSkip1, h = self.downConv1(h, biEdges[1], inEdges[1], vertsLengths[0])

        # Decoder layers (up convolutions)
        h = self.upConv1(h, hSkip1, biEdges[1], outEdges[1])
        h = self.upConv2(h, hSkip2, biEdges[2], outEdges[2])
        h = self.upConv3(h, hSkip3, biEdges[3], outEdges[3])
        h = self.upConv4(h, hSkip4, biEdges[4], outEdges[4])
        h = self.upConv5(h, hSkip5, biEdges[5], outEdges[5])
        h = self.upConv6(h, hSkip6, biEdges[6], outEdges[6])
        h = self.lastConv(h, biEdges[6])
        return h
```