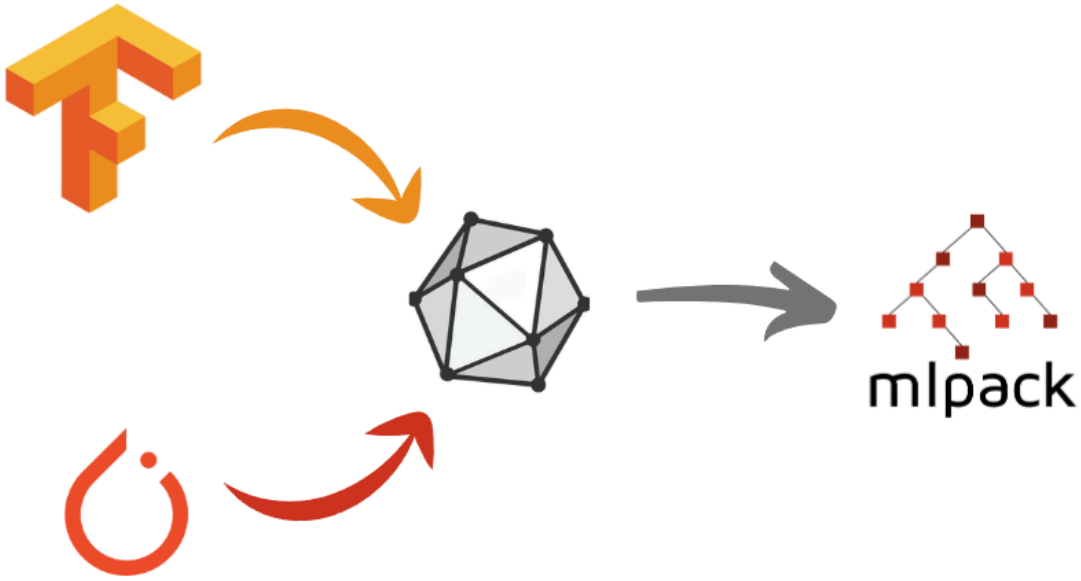
Making an ONNX-mlpack Converter



Basic Information

Name: Kumar Utkarsh

University: Indian Institute of Technology (BHU), Varanasi

Field of study: Mechanical Engineering

Time study was started: October 2022

Expected graduation date: July 2026

Degree: Bachelor of Technology

Year: Sophomore

Github: kumarutkarsh1248

LinkedIn: Kumar Utkarsh

IRC nickname: kumarutkarsh1248

Email Address: nvnukumarutkarsh@gmail.com

Phone number: (+91) 8102226322

CV: https://drive.google.com/file/d/1IIDP1ccv981y13XrtvCc1r91KlBGLx7x/view?usp=sharing

TimeZone: Indian Standard Time (UTC +5:30)

Table of content

Project Proposal

Objective

Training intricate machine learning models to achieve the desired level of proficiency can be time-consuming and require high computational resources. These challenges restrict the utilization of highly complex models in mlpack. To tackle this issue, one solution is to utilize pre-trained models available in the ONNX format. ONNX offers a standardized method for developers to define and exchange deep learning models across various frameworks like TensorFlow, PyTorch, and more.

Here are the specific goals I aim to achieve by the end of the summer.

- Analyze all internal layers and their order in the ONNX model, and generate an error if a layer is missing in mlpack.
- Develop a mapping container capable of extracting weights and layer sizes from layers of ONNX model and linking them to corresponding mlpack model layers.
- Make the converter support all existing mlpack layers and kernels
- Creating detailed docs for all the above implementations.
- Writing test cases for each of the implementations.
- Creating necessary environments, for proper testing of algorithms above (after
- discussion with mentor)
- putting several conversion examples for linear, convolution, MobileNet SSD and Tiny Yolo v3 from ONNX format to mlpack binary.

Understanding protocol buffer and onnx

Protocol Buffers (protobuf) is a method for serializing structured data developed by Google. ONNX utilizes Protocol Buffers as its serialization format for representing neural network models. Understanding protobuf is crucial because ONNX model files are protobuf (.pb) files. ONNX itself doesn't offer any method to extract model information from the ONNX file in C++. Instead, it's the Protocol Buffer compiler that provides various classes and functions from which we can import the model and extract all kinds of information from it.

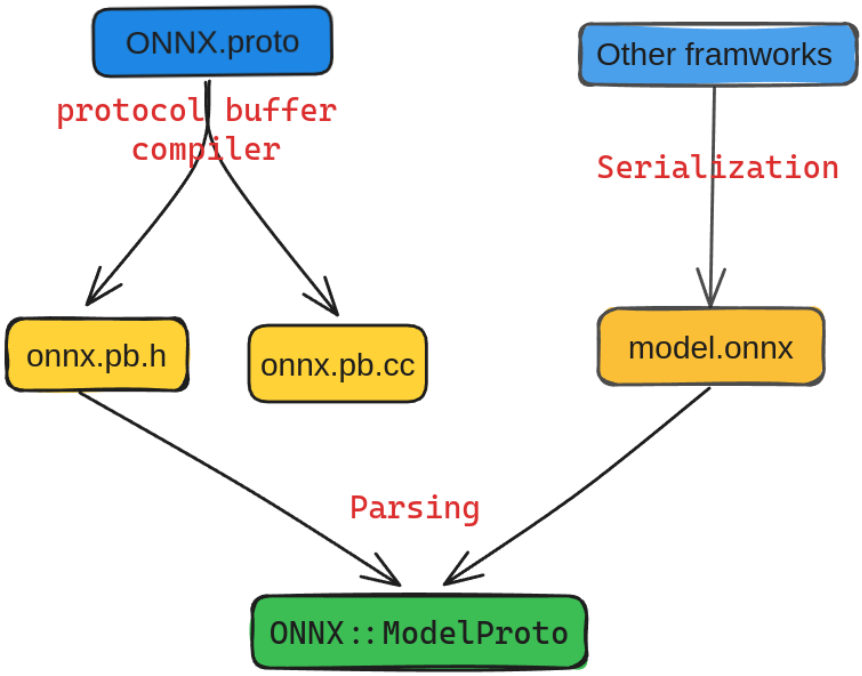
Google provides comprehensive documentation on how data structures are defined in protobuf files and how the protobuf compiler generates classes and functions from these protobuf files.

ONNX uses Protobuf to define the structure of its models. Protobuf majorly provides two types of data structure “message” and “field”.

- **Message:** For simplicity it can be considered as c++ structs without member function.
- **Fields**: They can be considered as member variables.

Note: The message can contain multiple other messages nested within it. ONNX defines its own set of messages and fields from which any neural network model can be represented. These messages and fields are defined in the onnx.proto file, and these data structures can be visualized in **onnx.png**.

The Protocol Buffer compiler compiles the onnx.proto file and generates classes and functions that we can use to parse the serialized model(model.onnx) in our program and further extract information from the Model.

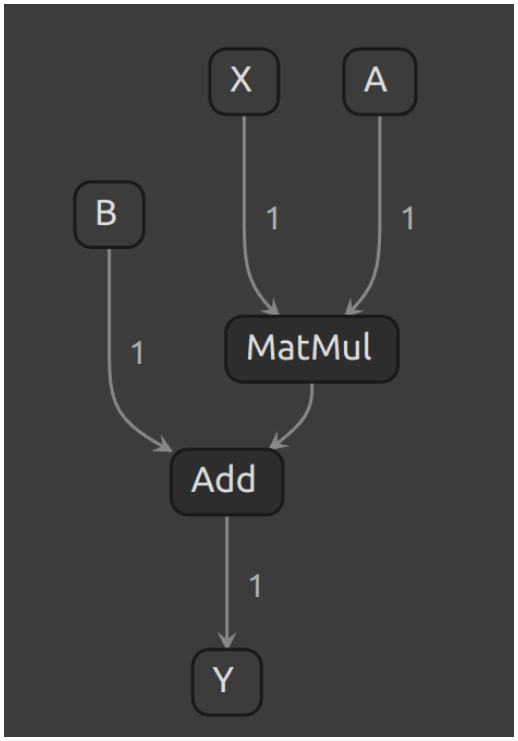


Model.onnx : Different frameworks save their trained model in the .onnx file.
ONNX::ModelProto : This is an object containing the parsed ONNX model within the current program.

Note: When a .proto file is compiled with a protobuf compiler, all the generated classes will be placed in a namespace matching the name of the .proto file. Therefore, the ModelProto class is generated by the protobuf compiler, but it is placed inside the namespace "onnx" because it is generated after the compilation of the onnx.proto file.
*** These details are described very clearly in protobuf c++ documentation.*

Visualizing ModelProto

Before defining the internal components of ModelProto, let's first start with a simple example:




```
ir_version: 9
graph {
        node {
                input: "X"
                input: "A"
                output: "XA"
                op_type: "MatMul"
        }
        node {
            input: "XA"
            input: "B"
            output: "Y"
            op_type: "Add"
        }
        input {
                name: "X"
                type {...}
        }
        input {
                name: "A"
                type {...}
        }
        input {
                name: "B"
                type {...}
        }
        output {
                name: "Y"
                type {...}
        }
        initializer {
                dims: 1
                dims: 1
                float_data: 3
                name: "A"
        }
        initializer {
                dims: 1
                dims: 1
                float_data: 5
                name: "B"
        }
}
```

***For the input that serves as the entry point for the entire model, I will be calling it "model input"*

In the above example model input is X.
Suppose the model is performing a simple linear operation: y = 3*x + 5.

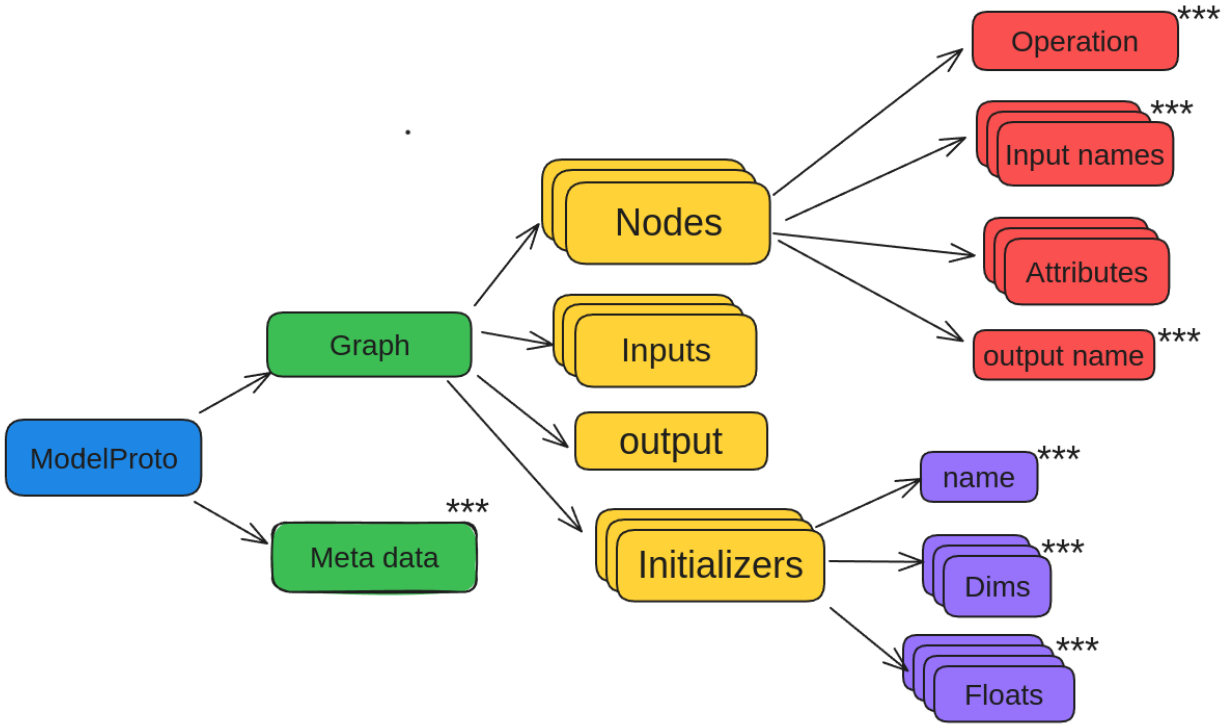
For this simple linear model, the onnx graph will have:
2 nodes, 3 inputs, 1 output, 2 initializer messages.

Data flow through model:

- The first node has two input fields: 'X' and 'A', and performs matrix multiplication between them.
- The name of the first graph initializer is 'A', and there is no initializer for 'X'. This indicates that during matrix multiplication, the value for 'A' is already present in the graph, while the value for 'X' will come from model input.
- After the matrix multiplication, the first node produces the output 'XA'.
- The second node has two input fields: 'XA' and 'B'. Once again, there is an available initializer for 'B', while 'XA' is the output of the previous node.

Note: the input messages inside the graph are the pure inputs ("X", "A", "B"). This means either these inputs will correspond to one of the initializers, or they will be the model inputs. They can't be the output of any of the nodes.

Internal structure of the Modelproto can be visualized from the block diagram shown below:



Note: All blocks with *** marks on top of them represent fields, while other blocks represent messages. Blocks stacked on top of each other indicate that the parent message contains multiple instances of these messages/fields.

- **Modelproto**: This is an object containing the parsed ONNX model within the current program. Each piece of information needed to generate the mlpack model can be extracted from the ModelProto.
- **Metadata**: It contains information such as the ONNX version and the framework from which the model is developed.
- **Graph:** the core computational structure of the model is defined within the graph.
- **Node**: It represents an operation or a single computational unit within a computational graph. It can be considered as a layer of the FFN model.

  Node contains various field and messages:
  1. Operation: Operation that the node performs on the input.
  2. Input names: Contains the names of all the inputs that the node takes to execute the operation
  3. Attribute: Node attributes specify details about how the operation represented by the node should be carried out; for instance, in convolution, attributes may include parameters such as padding, kernel size, and stride.
  4. Output name: Contains the name of the output that the node is generating after performing operation on the input data.

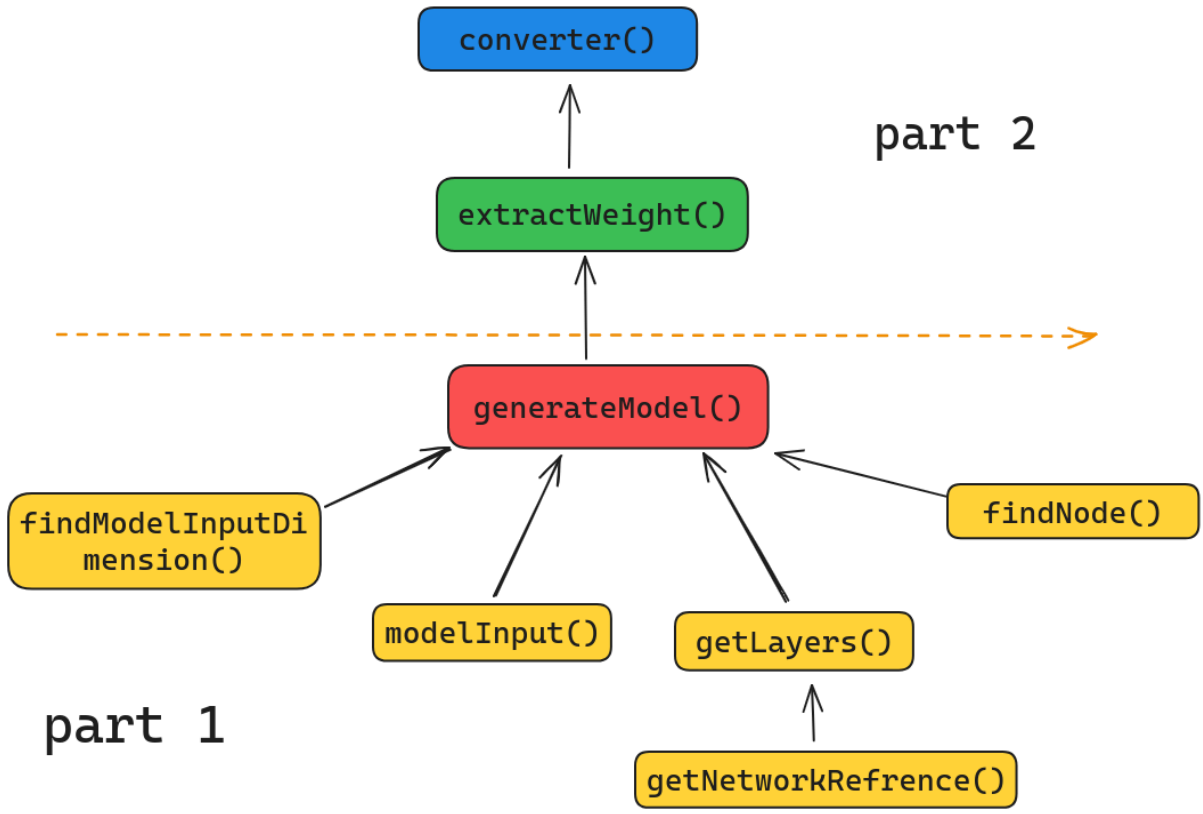
- **Inputs**: The input messages inside the graph are the pure inputs. This means that either these inputs will correspond to one of the initializers, or they will be the model inputs.
- **Initializer**: It contains the trained weights and biases of the layer and its dimensions. For each initializer, there will be a corresponding input, and the name of both the initializer and input will be the same.

```
initializer {
  dims: 35
  dims: 20
  data_type: 1
  float_data: -0.37362504
  float_data: -0.352708191
  float_data: 0.202972785
  float_data: 0.876093447
  float_data: 0.257881105
  float_data: 0.0852852687
  float_data: -0.274584472
  float_data: 0.249078721
  float_data: -0.46734941
  float_data: -0.158601716
  float_data: -0.607757807
```

```
node {
  input: "ReLU114_Output_0"
  output: "Pooling160_Output_0"
  name: "Pooling160"
  op_type: "MaxPool"
  attribute {
    name: "kernel_shape"
    ints: 3
    ints: 3
    type: INTS
  }
  attribute {
    name: "strides"
    ints: 3
    ints: 3
    type: INTS
  }
  attribute {…
  }
}
```

Background Information

Generating an mlpack model from the ONNX model involves two major steps. First, creating a feedforward network by analyzing the graph of the ONNX model and mapping ONNX node attributes to the mlpack FFN layers. Second, transferring the weights from the ONNX model to the feedforward network generated in the first part.



Part 1: Analyzing the ONNX model and mapping ONNX node attribute to mlpack model

The most important aspect of generating the mlpack model from the ONNX model is to iterate through the nodes in the correct order. ONNX models are defined according to the ONNX Intermediate Representation (IR) specification and operator schemas. According to the IR specification, ONNX nodes are topologically sorted, which means the output of the current node will be the input of the next node. To move to the next node, we check which one of the nodes is taking the output of the current node.

Once we are in the current node, we can determine which layer needs to be added in the mlpack feedforward neural network (FFN). It is also crucial to properly set the behavior of that layer by setting the parameters of the mlpack layer. ONNX defines in the operator schema how

parameters are stored in the ONNX graph. However, the way layer parameters are stored in the ONNX model differs significantly from how mlpack requires them to create a layer. Therefore, the most challenging aspect of this ONNX-mlpack converter is to map ONNX layer parameters to mlpack layer parameters. We will use various special cases and methods to achieve this parameter mapping.

Part 2: Transferring ONNX weights to mlpack model

In ONNX, weights are stored in initializers, and whether these weights are in row-major or column-major format depends on the framework from which the ONNX model is generated.

For the feedforward neural network (FFN) generated from the first part, all its weights are randomly initialized. We iterate through its layers and transfer the weights to these layers. We extract the weights from the initializer, transform them to row-major format, and then give them the proper dimensions before transferring these weights to the layer.

Part 1: Analyzing the ONNX model and mapping ONNX node attribute to mlpack model

Abstract

A. **Generating FFN:**
To generate an FFN from an ONNX model with all input and output dimensions set and weights randomly initialized, we will define a function called generateModel(). This function will iterate through the nodes of the graph in topological order, and for each node, it will add a new mlpack layer to the FFN one by one.

B. **Finding the model input**
To iterate through the nodes in topological order, it's important to obtain the model input. This input is crucial because it allows us to identify the first node of the ONNX model. Then, the output of the first node becomes the input of the second node, facilitating the progression to the subsequent nodes.
To achieve this, we will define a function called findNode, which will take the input name as an argument and then search for the node that is using that input.

C. **Finding the model input dimension:**
While making a FFN in mlpack we have to explicitly set the input dimension of our model. For that purpose we explicitly define a function findModelInputDimension(), which will return the input dimension of the model input in vector form.

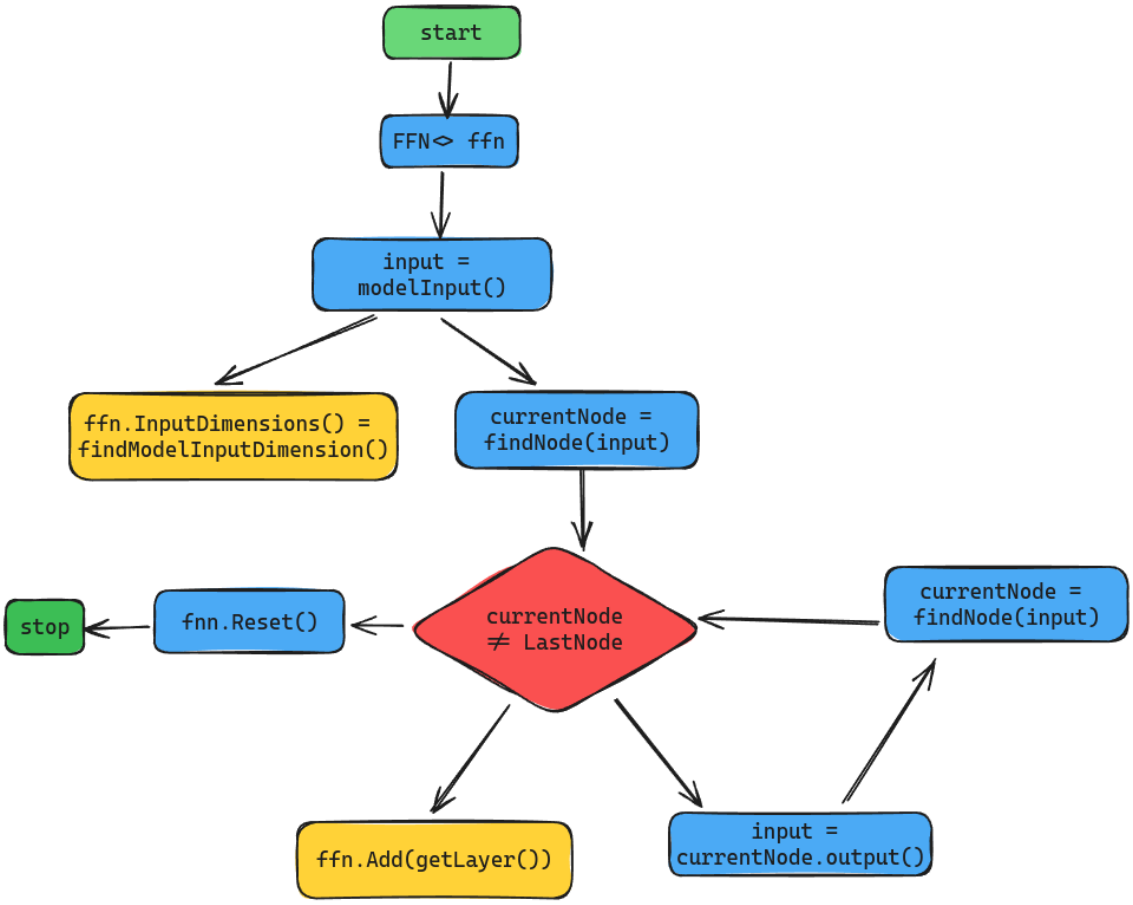
D. **Find node:**
To iterate through the nodes in topological order, we take the output of the current node and search for the node that is using this output as its input. For this purpose, we define a function called findNode(), which takes the input name as an argument and returns the node that is consuming that input.

E. **Getting layer parameters:**
While adding a layer to the feedforward network (FFN) in mlpack, it's important to set the behavior of that layer. This can be achieved by appropriately mapping the ONNX layer parameters to the mlpack layer, taking into consideration the ONNX operator schema.
To accomplish this, we define a function called getLayer, which properly maps the ONNX parameters to the corresponding mlpack layer.

F. **Creating the layer:**
Once we know which layer needs to be added in the feedforward neural network (FFN) and we have all the required parameters, we can proceed to create the actual mlpack layer. For this purpose, we define a function called getNetworkReference which creates an mlpack layer, sets all its parameters, and returns a reference to that layer.

Implementation detail

A. generateModel():



Pseudocode:

1. Generating the model begins with finding the model input and creating a FFN object.
2. Find the dimensions of the model input with the help of function modelInput() and set it as the input dimensions of the FFN.
3. Since the ONNX model works based on topological order, the node taking the model input will be the first node and we treat it as the current node.
4. We get the first node by passing model input in the function findNode().
5. Check whether the current node is the last node or not. If it's the last node, then reset the network and terminate the program.
6. Add a layer to the feedforward neural network corresponding to the current node.
7. Find the output generated by this current node and use it to determine the next node.

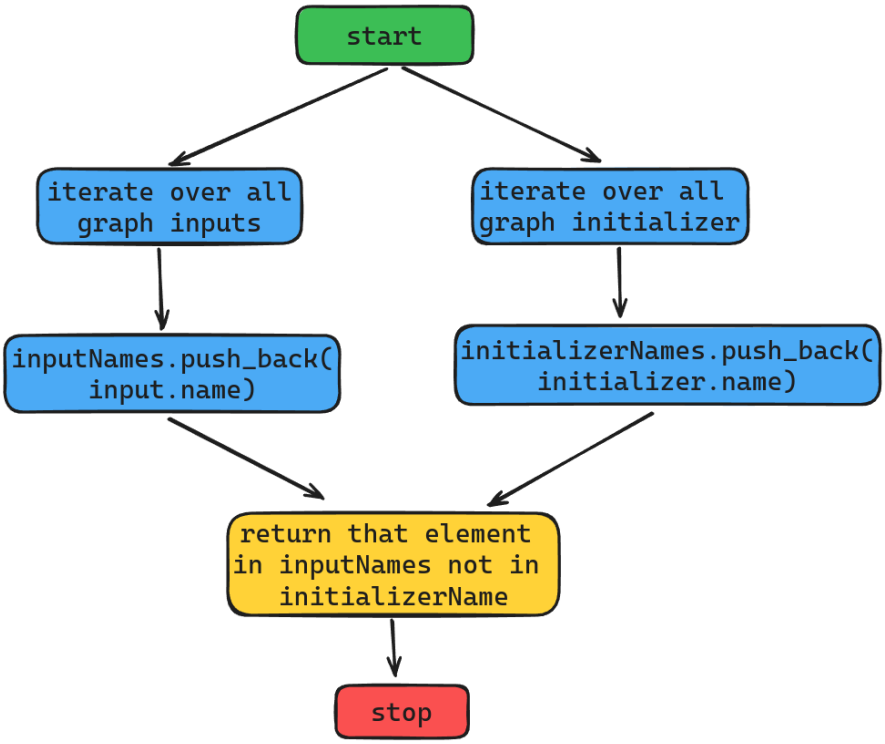
8. Repeat from step 5 until the current node becomes the last node of the ONNX model.

I have implemented the generateModel() function **here**.

***Note**: We can't directly iterate through the nodes in the order returned by onnxModel.node() because I have analyzed several ONNX models and realized that the order of nodes returned by onnxModel.node() is in the actual order only in the case of linear models. However, in models involving convolution operations, the order of nodes returned by onnxModel.node() is almost random. This is just a general observation, and I have raised a query regarding this in the ONNX repository.*

B. modelInput():

This relies on the fact that among all input messages within the graph, there will be one single input message for which no initializer will be available. That input message will represent the model input, and we need to obtain the name of that input message.



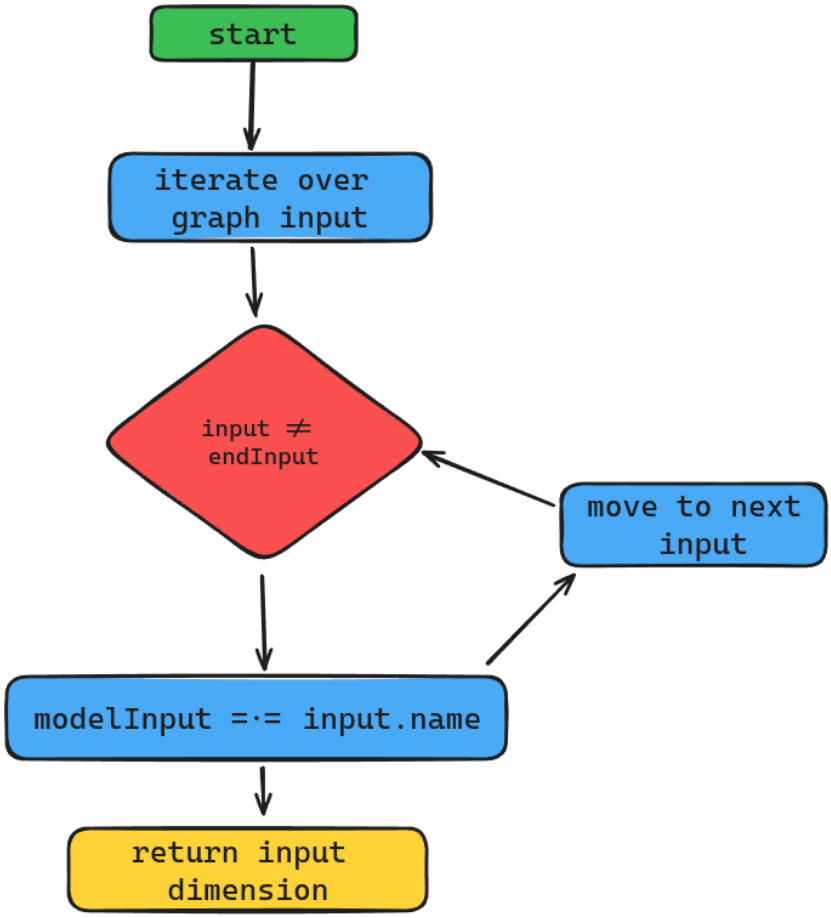
Pseudocode:

1. First, we iterate through all the graph inputs and store the name of each input in a vector<string> called inputNames.
2. Then, we iterate through all the graph initializers and store the name of each initializer in a vector<string> called initializerNames.

3. Next, we search for elements in inputNames that are not present in initializerNames.
4. The element we find will be the model input.

I have implemented the modelInput() function **here**.

C. findModelInutDimension():



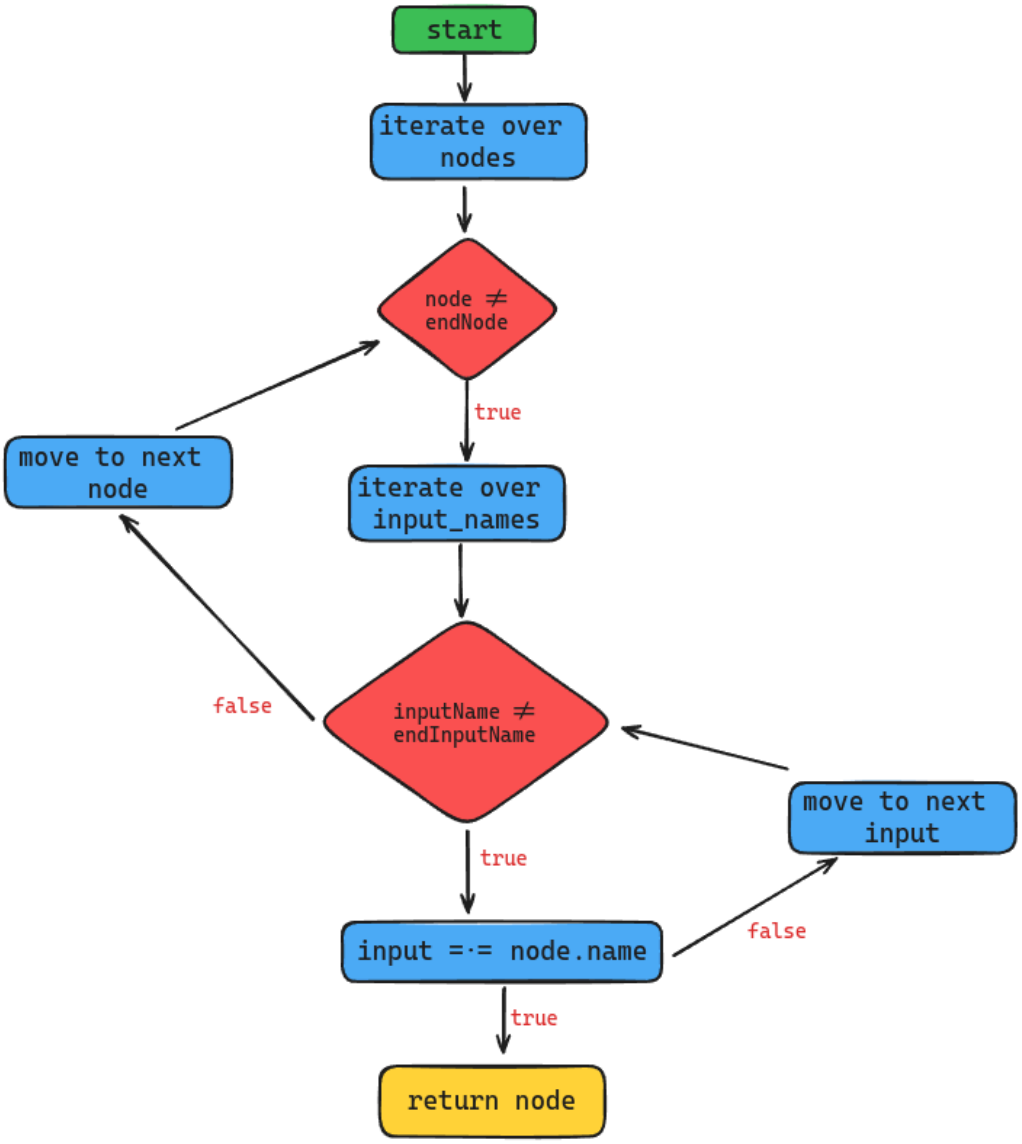
Pseudocode:

1. In the previous section we got the model input.
2. Now we iterate through the graph input and find the one whose name is the same as model input.
3. Input message inside the graph also contains the dimension of that input so we return the dimension of that input through that.

I have implemented the findModelInutDimension() function **here**.

D. findNode():

This function will take arguments, graph and inputName. In the whole graph it will search for the node which is taking the input as inputName.



Pseudocode:

1. First we start iterating over nodes. A single node can contain multiple input fields.
2. Then we start iterating over the input fields inside the current node.
3. If any of the input fields has a name common to inputName.
4. We return the current node as the required node.

I have implemented the findNode() function **here**.

E. getLayer():

Inside this function multiple mapping container are initialized: operatorMap, onxxAttributes, onnxLayerAttribute, mlpackLayerAttribute.

```
map<string, string> operatorMap = {
    {"Conv", "convolution"},
    {"Reshape", "reshape"},
    {"MaxPool", "maxpooling"},
    {"MatMul", "linearnobias"},
    {"Add", "add"},
    {"Relu", "relu"},
    {"Softmax", "softmax"},
    {"Identity", "identity"}
};
```

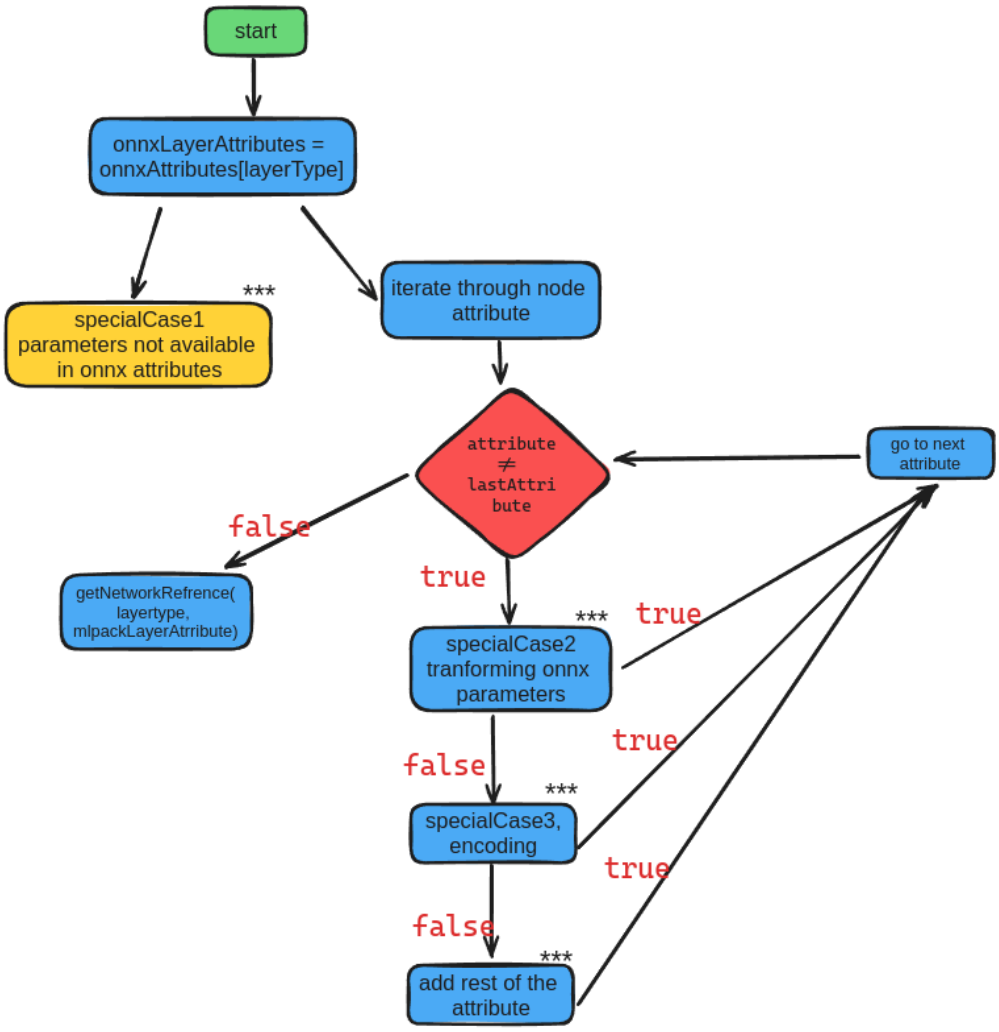
```
map<string, map<string, vector<string>>>
onnxAttributes;
onnxAttributes["Conv"] = {
    {"kernel_shape", {"kh", "kw"}},
    {"pads", {"padh", "padw"}},
    {"strides", {"dh", "dw"}}};
onnxAttributes["MaxPool"] = {
    {"kernel_shape", {"kh", "kw"}},
    {"strides", {"dh", "dw"}}};
onnxAttributes["MatMul"];
onnxAttributes["LeakyRelu"] = {
    {"alpha", {"alpha"}}};
onnxAttributes["Identity"];
onnxAttributes["Relu"];
```

1. operatorMap: It maps the ONNX node op_type name to the mlpack layer name.
2. onnxAttributes: This will contain the attributes for all possible ONNX operations stored according to the ONNX operator schema.
3. onnxLayerAttribute: This will contain the attributes for some specific ONNX operations stored according to the ONNX operator schema. Example: onnxLayerAttribute = onnxAttributes[“Conv”].
4. mlpackLayerAttribute: This will be the final mapping container, which will contain all the parameters required to generate the mlpack layer.

```
map<string, vector<string>>
onnxLayerAttribute = {
  {"kernel_shape", {"kh", "kw"}},
  {"pads", {"padh", "padw"}},
  {"strides", {"dh", "dw"}}
}
```

```
map<string, double>
mlpackLayerAttribute =
{
  {"maps", NAN},
  {"kw", NAN},
  {"kh", NAN},
  {"dw", 1},
  {"dh", 1},
  {"padw", 0},
  {"padH", 0},
  {"paddingtype", 0}
}
```

It should be noted that in mlpackLayerAttribute, the paddingType is given the value 0.This value is encoded, and later, when we actually create the mlpack layer in getNetworkReference, these encoded values will be decoded.



*In the above block diagram, the blocks having *** in its top are responsible for adding parameters in mlpackLayerAttribute.*

Pseudocode:

1. Find the onnxLayerAttribute corresponding to layerType from *onnxAttribute*.
2. Handle special case 1: There are some mlpack layers that require certain parameters not present in the node attribute. We handle such layer parameters here. For example, we use the linearNoBias layer for the MatMul node. The mlpack linearNoBias layer requires the parameter outSize, which is not present in the node attribute. Thus, we define a special function like findOutputDimension() to extract this outSize information from the graph initializer. This has been implemented in the code here.

```
// special layer parameters specific to given type of layer
if (node.op_type() == "MatMul")
{
    mlpackLayerAttribute["outsize"] = findOutputDimension(graph, node);
}
if (node.op_type() == "Conv")
{
    mlpackLayerAttribute["maps"] = findConvMap(graph, node);
}
```

3. Start iterating through the attributes. If it's the last attribute, then call the getNetworkReference().
4. Handle special case 2: There are some ONNX node attributes that need to be transformed before mapping them to mlpackLayerAttribut. For example, for the conv node, ONNX stores padding parameters like padTop, padBottom, padLeft, and padRight, but the mlpack convolution layer accepts only padH and padW. Thus, we need to transform these parameters to make them compatible with mlpack. This special case is implemented in the code **here**. If this special case is matched, then update the mlpackLayerAttribute and move to the next attribute.
5. Handle special case 3: If the node attribute does not fall under special case 2, then we check for special case 3. Our mlpackLayerAttribute can only take string-double key-value pairs, but there are layer parameters that are boolean or string. We handle this by encoding such parameters here in numeric form and later decoding them when generating the mlpack layer. If this special case is matched, then update the mlpackLayerAttribute and move to the next attribute.

```
else if (attrName == "auto_pad")
{
    if (attribute.s() == "SAME_UPPER" || attribute.s() == "SAME_LOWER")
    {
        mlpackLayerAttribute["paddingType"] = 0; // same
    }
    else if (attribute.s() == "VALID")
    {
        mlpackLayerAttribute["paddingType"] = 1; // valid
    }
    else if (attribute.s() == "NOTSET")
    {
        mlpackLayerAttribute["paddingType"] = 2; // none
    }
}
```

In the above pic, I have shown how encoding will done for conv paddingType

6. If none of the special cases are matched, it means that the ONNX attributes can be directly mapped to `mlpackLayerAttribute`, so update the mlpackLayerAttribute and move to the next attribute.
7. Repeat from step 3.

***I understand that the implementation has become a bit complicated due to these special cases, but we can't avoid them because the operator schema for ONNX and mlpack differ*

significantly. As we add more mlpack layers to our converter, there will likely be even more special cases. Therefore, in the future, I plan to split this function into several other functions so that the getNetwork function can call these sub-functions to handle these special cases. This approach will make the getNetwork function appear less complicated.

I have implemented the getLayer() function **here**.

E. getNetworkRefrence():

Pseudocode:

1. First, it generates a temporary mapping container called origParams.
2. Then, based on the given layerType, it begins populating origParams with all the required parameters along with their default values.
3. Next, it utilizes the updateParams function, which replaces the default values with the corresponding values from mlpackLayerAttribute.
4. Next a new layer object is created, and all the required parameters from origParams are assigned to it with all necessary decoding.
5. Finally, this layer object is returned.

```
Layer<> *getNetworkReference(const std::string &layerType,
                             std::map<std::string, double> &layerParams)
{
  std::map<std::string, double> origParams;
  Layer<> *layer;

  if (layerType == "linearnobias")
  {
    NoRegularizer regularizer;
    origParams["outsize"] = NAN;
    updateParams(origParams, layerParams);
    layer = new LinearNoBias(origParams["outsize"], regularizer);
  }
  else if (layerType == "convolution")
  {
    origParams["maps"] = NAN;
    origParams["kw"] = NAN;
    origParams["kh"] = NAN;
    origParams["dw"] = 1;
    origParams["dh"] = 1;
    origParams["padw"] = 0;
    origParams["padH"] = 0;
    origParams["paddingtype"] = 0; // None = 0 , Valid = 1, Same = 2
    updateParams(origParams, layerParams);
    std::string padding = decodePadType(origParams["paddingtype"]);
    layer = new Convolution(origParams["maps"], origParams["kw"],
                            origParams["kh"], origParams["dw"],origParams["dh"],
                            origParams["padw"], origParams["padh"],padding);
  }
  return layer;
}
```

I have implemented the getNetworkRefrence() function **here**.

Special method specific to layer type

---

1. **findConvMap:**
   In the first special of getLayer, this findConvMap method is used to get the number of convolution maps.
   This method is implemented **here**.

2. **findOutputDimension**:
   In the first special of getLayer, this findOutputDimension method is used to get the outSize of LinearNoBIas or MatMul layer.
   This method is implemented **here**.

Inside the getLayer function, these above two methods are called like:

```
// special layer parameters specific to given type of layer
if (node.op_type() == "MatMul")
{
    mlpackLayerAttribute["outsize"] = findOutputDimension(graph, node);
}
if (node.op_type() == "Conv")
{
    mlpackLayerAttribute["maps"] = findConvMap(graph, node);
}
```

3. **Decoding encoded parameters**:
   The decoding methods are called inside the getNetworkRefrence function when mlpack layers are made.

```
std::string decodePadType(double val)
{
  if (val==0)
    return "None";
  else if (val==1)
    return "Valid";
  else
    return "Same";
}
```

Part 2: Transferring ONNX weights to mlpack model

Abstract

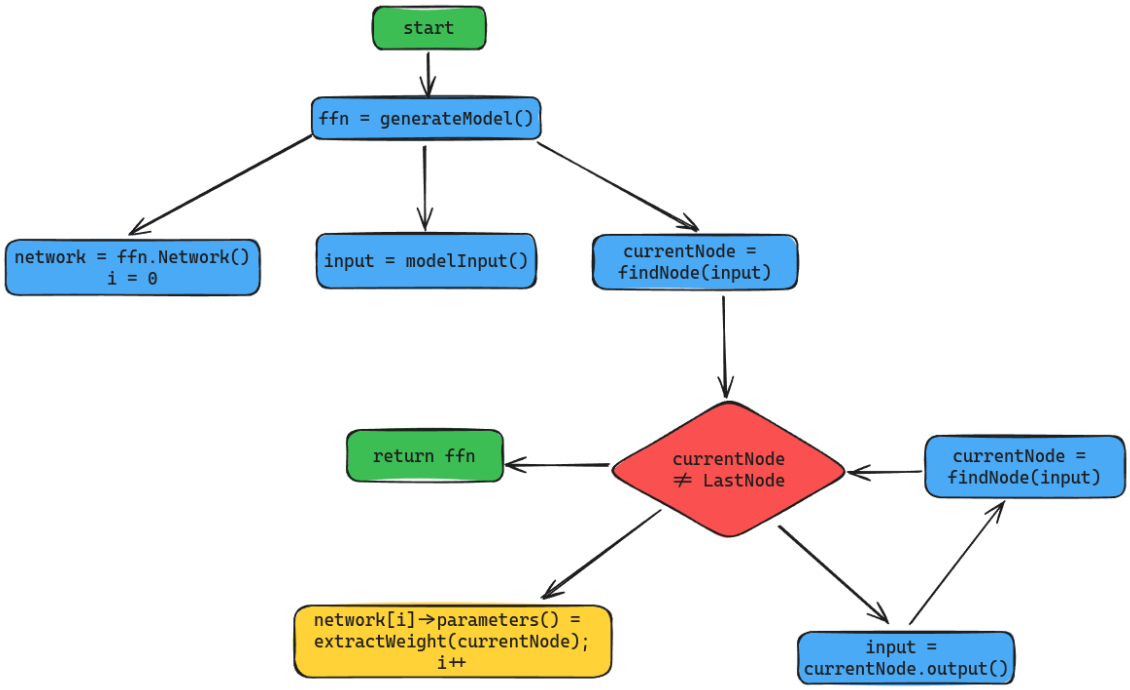
A. **Converter:**
Once the FFN has been generated with the generateModel function, the next step would be to set the weights of the layers for this generated model to get the desired prediction from the model.

B. **Weight extraction**
This function would take care of the row-major to column-major conversion between the mlpack and ONNX models. By knowing the node for which the weight is to be extracted, it will extract the weight, give it the proper dimensions, and return it in the arma::Mat format, which can be directly mapped with the parameter of the mlpack layer.

Implementation detail

---

1. **Converter:**
This will be the ultimate function that will finally return the model with all its weights set so that the model can make predictions.



Pseudocode:

1. The converter will first generate an FFN (Feedforward Neural Network) with all input and output dimensions set and randomly initialized.
2. Then, all the internal layers of the FFN will be stored in a vector<Layer<>*> called network.
3. Next, we find the model input and with the help of this we find the initial node of the graph with findNode()
4. Take the initial node as currentNode.
5. Then, we check whether the current node is the last node or not. If it is the last node, we return the FFN, and the program terminates.
6. We extract the weight of the current node with the help of the extractWeight() function and transfer this weight to the i-th layer of the network.
7. After that, we find the output of the current node and find the next node with the help of this output.
8. Repeat from step 5.

2. **extractWeight:**
   This function would take care of the row major- column major between mlpack and onnx model, and ny knowing the node for which weight is to be extracted, it will extract the weight, give it proper dimension and return in arma::Mat format which can be directly map with the parameter of mlpack layer.

   Pseudo code:

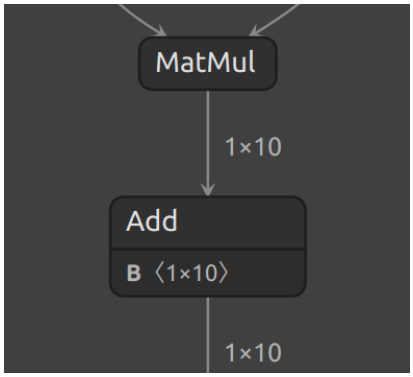
1. First major step would be to know whether the weights are stored in row major or column major format.
   Many deep learning frameworks like TensorFlow and PyTorch use row-major storage for weights, where elements are stored row by row. However, the `extractWeight` *function will be designed in such a way that it can handle both row major and column major formats. To determine the storage format of weights in the ONNX model, it is necessary to know from which framework this ONNX model was generated.*

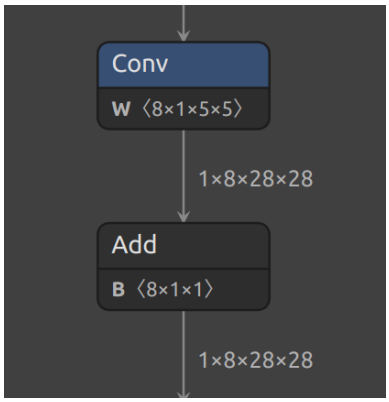
2. For the corresponding node, the converter will search for the initializer containing all the weights.

```
6255  initializer {
6256    dims: 1
6257    dims: 10
6258    data_type: 1
6259    float_data: -0.0448560268
6260    float_data: 0.00779166119
6261    float_data: 0.0681008175
6262    float_data: 0.0299937408
6263    float_data: -0.126409635
6264    float_data: 0.14021875
6265    float_data: -0.0552849025
6266    float_data: -0.0493838154
6267    float_data: 0.0843220502
6268    float_data: -0.0545404144
6269    name: "Parameter194"
6270  }
```

3. It will calculate the size of the weights and create an Armadillo matrix of that size.
4. Subsequently, it will extract the weights and place them all into the matrix.
5. By considering the dimensions specified in the initializer, it will reshape the matrix, taking care of row-major/column-major order.

Note: ONNX graph has a separate node for adding a bias term. Therefore, instead of adding a linear layer in the mlpack model, I will be performing this in two steps. First, using the LinearNoBias layer, and then adding the bias term using the Add layer. Similarly, for the convolution layer, we will first set all the bias terms to zero, and then the bias term will be added with a separate Add layer. This step is necessary as weights and biases are stored in different initializers, so dealing with them separately will make the whole transferring process a lot easier.





Expected Timeline with Proposed Deliverables

I don't have any other major commitments during the coding period. Because of the extended time period of my project, I will have exams around August - September. Exact dates and details will be communicated to my mentor inorder to avoid any deadline or progress issues.

Community Bonding Period **May 1 - 26**

➔ I will discuss project in more detail with the mentor.
➔ I will try to raise issues in the ONNX repository to clear all my current doubts and establish a bond with the ONNX community so that any future queries can be resolved promptly.
➔ I will get more acquainted with the codebase, especially mlpack ANN layers and FFN, along with that I plan on getting to know the armadillo library as it will be used extensively.
➔ I'll try to fix issues related to Add class that applies a bias term to the incoming data as this layer will be extremely important when I will be working on transferring weight from the onnx model to the mlpack model.

Week 1 **27 May - 3 June**

➔ If the "Add class Issue" has been resolved, I would attempt to implement a weight transfer function in the current converter in order to achieve the desired output from the linear and MNIST models.
➔ Even if the issue with the Add class still persists, I have another approach in mind to merge the MatMul and Add nodes and use a single linear layer in place of them. However, this approach has its own complexity during weight transferring, and I may need to take a few more days from next week to implement this approach.

Week 2 & 3 **4 - 17 June**

➔ This period would be spent on analyzing some more advanced 15-20 ONNX models and deeply understanding the mlpack ANN layers required to implement for translating these ONNX models into mlpack models.
➔I would also spend more time understanding the ONNX operator schema and mapping them to mlpack layer parameters.

Week 4 & 5 **18 - 30 June**

➔ This period would be spent implementing all those ,mlpack layers in the getNetwork and getNetworkReference functions, so the converter becomes capable of generating models from those 15-20 ONNX models.
➔I would also try to improve the extractWeight function so that our converter can generate models with all their learnable values set.

**If the goals have been achieved up to this week, then we can call our model somewhat generalized, and we will be very close to achieving our target for this summer.

Week 6 & 7 **1- 14 July**

➔ I will attempt to translate MobileNet SSD and Tiny Yolo v3 from ONNX format to mlpack binary.
➔ I would also spend more time understanding the different versions of ONNX models and how these versions affect the model's graph.
➔ To achieve the future goal of bidirectional translation between ONNX and mlpack, I will start exploring how an ONNX model can be generated in C++.

Week 8 & 9 **15 - 28 July**

➔I will make suitable changes to the converter so that it can recognize the ONNX model version and behave accordingly.
➔ I'll complete writing tests for the converter and put some example on MobileNet SSD and Tiny Yolo v3.
➔ I will continue exploring the generation of ONNX models in C++ and search for some references for this bidirectional translation.

Week 10 & 11 **29 July - 11 August**

➔ I'll provide detailed documentation for converter and also a tutorial, if time remains.
➔ I'll Make sure that the converter reaches mlpack standard.
➔ I will attempt to serialize a simple linear mlpack model to ONNX format, and if I am able to do this successfully, I will take this bidirectional mlpack-ONNX conversion forward.

Ultimate Week **12 - 19 August**

➔ I would wrap up my implementations by completing all pending work (if any) from before, including implementation, tests and documentation.
➔Finally, I will discuss with the mentor regarding future improvements and making the converter bidirectional.

Personal Details

I am Kumar Utkarsh pursuing graduation in Mechanical engineering at Indian Institute of Technology (BHU), Varanasi.

I started coding primarily 2 years ago and stepped into open source just to try something new.

I have extensively used deep learning frameworks like pytorch and keras with tensorflow for my projects. Creating such a massive library which made machine learning so easy is not an easy task and I have always wondered what goes into making such easy to use APIs. Since my engagement with mlpack for the last couple months, I have had the opportunity to observe it. Now with GSoC 24, I would like to be a part of this amazing team.

I have been using Ubuntu 20.04. Generally, I use Visual Studio Code for working on development projects.

Technical proficiency and coding skills:

---

1. **What languages do you know? Rate your experience level (1-5: rookie-guru) for each**

   ➨ C++ : 3.5
   ➨ Python : 4
   ➨ Arduino : 4
   ➨ JavaScript : 3
   ➨ C : 2

2. **How long have you been coding in those languages?**

   I have been coding for two years, starting with Python as my first programming language. From the very beginning, I have had an interest in game development. After becoming somewhat familiar with Python, I created a Snake game. Later, I shifted my focus to web development and creative coding with P5, where I developed a game called **Tomb Runner**. Over the past year, I have been delving into machine learning with Python, TensorFlow, and PyTorch. I've even developed my own **neural network library using just NumPy**. Additionally, I have an interest in robotics and have been working with Arduino and controls for the past year. One of my recent Arduino projects was a **micromouse**, where I tried to make its codebase similar to mlpack :) Recently, I've also started learning C++ with the goal of contributing to open source projects. Over the past 4-5 months, I have been improving my C++ skills, and I am now familiar with most object-oriented programming concepts.

3. **Are you a contributor to other open-source projects?**

   I haven't had any accepted pull requests yet, but I did submit a pull request to the **mlpack-pytorch-weight-translator** repository, where I created a custom loss function for the YOLO Tiny model in PyTorch.

4. **Do you have a link to any of your work (i.e. github profile)?**

   - Neural network from numpy : **code**
   - Micromouse **code**

- Yolo in pytorch with custom loss function **colab**
- Tomb Runner game **web game**
- Personal portfolio **website**
- Other objects can be found here: **github account**

5. **What areas of machine learning are you familiar with?**

   ➢ **ONNX-protocol buffer** I have been working on ONNX-Mlpack converter for a month and for that i have explored so much that now i feel very comfortable with ONNX and protocol buffer.

   ➢ **Neural network architecture:** I started exploring the machine learning field when I received a project from my college club. After using some TensorFlow methods like a black box, I was amazed at how those backpropagation and optimization methods enabled our model to make such complicated predictions with such high accuracy. Then, I decided to create my own library from NumPy to have a better understanding of machine learning architecture.

   ➢ **YOLO:** A few months ago, I was considering adding an example of YOLO Tiny to the example zoo. For that purpose, I created a YOLO Tiny model in PyTorch. I intended to use Kartik Dutta's mlpack-pytorch-translator to transfer these weights from PyTorch to mlpack. However, this project is still pending because the PyTorch-mlpack-translator is not compatible with mlpack4.

6. **Have you taken any coursework relevant to machine learning?**

   ➢ I have taken IIT Madras lectures on deep learning **CS7015**.

   ➢ In addition to that, I have studied course on **Probability and Statistics, linear algebra and CSO101** at my institute.
   ➢ For my personal project on machine learning, I rely on Medium blogs, reddit, StackOverflow discussions and Research Papers.

Other open ended question

---

1. **What are your long-term plans, if you have figured those out yet? Where do you hope to see yourself in 10 years?**
   ➢ I have been exploring machine learning for one year, and I still feel that there is a lot more for me to discover. I also have a great interest in robotics because these hardware can interact with the real world and solve physical problems. Therefore, I hope to see

myself working with an organization in 10 years, implementing machine learning outputs in robotics.

➢ I am really enjoying my journey into open source. My experience across organizations has been amazing. Now that I am getting familiar with the workflow, I have no plans on quitting. So, I would like to contribute to open-source projects whenever I get time from my work, for fun, learning, and to have a sense of belonging, as I have been experiencing for quite some days now I think OpenSource development will occupy a long portion of my life.

➢ While I am currently very comfortable with machine learning and intend to stick with it for the long run, I also love exploring new technologies. I believe in expanding my knowledge beyond my primary domain to avoid being completely unfamiliar with other areas. Additionally, I enjoy delivering my concepts through YouTube tutorials and receiving compliments from people across the world. My aspiration is to become a successful YouTuber, creating tutorials on emerging software technologies.

2. **Describe the most interesting application of machine learning you can think of, and then describe how you might implement it.**
   ➢ So far, most of my machine learning projects have revolved around computer vision. A few months back, I was working on a project for college surveillance in which we aimed to create a system capable of detecting individuals based on their body posture and movement style. One might think that we could have directly used face detection, but most college events happen at night, making it very difficult to detect someone's face. Therefore, we focused on detecting individuals through their body movements. If implemented, this could have greatly aided in college surveillance by having data on each individual entering and exiting a particular area.

   ➢ I also thought of simulating the early Earth environment to observe the emergence and evolution of life from atomic particles to complex organisms. Although I have not yet started working in the reinforcement learning domain, after reading many articles and applications of reinforcement learning, I believe this idea is achievable for sure. These are dreams that are not that far away. I believe this is achievable with the combination of reinforcement learning and evolutionary strategies.

3. **Both algorithm implementation and API design are important parts of mlpack. Which is more difficult? Which is more important? Why?**
   Although both are important parts of mlpack, I personally find API design to be more challenging and important at the same time. This is because implementation of an algorithm could be done straight forward, but for a proper API design, the developer has to consider a lot of things, including the ease of usage and understandability both for

fellow/future developers and for end users. Also, the codebase needs to be structured in such a way that it is easy for adding new features in the future.

Communication

I'm flexible with my schedule and typically prefer to work during the daytime. However, I believe time zone differences won't be an issue because while you're asleep, I may have made some progress on the problem we're discussing. So, we can continue our discussions seamlessly regardless of the time difference.

I'm comfortable with any of the communication mediums I mentioned above.
I can work full-time on weekdays and am usually available between 1 PM IST to 6 PM IST. On weekends, I would love to spend time communicating with the team to learn from them, while working on whatever issues occur at that time.

I'll responsibly keep my mentor updated in case of any emergency that occurs with suitable details.

Post GSoC

If there are any remaining tasks left unimplemented, I'll try to complete them post-GSoC and will continue contributing to mlpack.

ONNX has no API or documentation for its use in C++, which makes it quite challenging for beginners to start working on it in C++. Therefore, I believe if we can create a foundational structure for the mlpack to ONNX conversion by the end of this summer, then achieving the goal of bidirectional compatibility will definitely be attainable by next summer.

I also enjoy creating YouTube tutorial videos, so if the mlpack community allows me, I would love to make tutorials specifically for the ONNX-mlpack-converter. These tutorials would help new contributors start quickly and understand my approach.

Contributions

Opened pull request

- **ONNX-mlpack-translator** Until now, this converter has been capable of converting linear and MNIST ONNX models to mlpack models. (I'm developing this repository on my personal account.)
- **Custom loss function in pytorch for yolo-tiny** in mlpack-pytroch-translator.