Dangers of a simple 'if' condition

You might be thinking what is wrong with ‘if’ conditions?

It’s a straightforward syntax. Do you want to do some activity if the number is even? `if number%2 == 0` if the number is odd then we provide an `else`. Let's level it up. Do an activity when the number is even and is a multiple of 5. `if number%2 == 0 and number%5 == 0`. What if the number cannot be a multiple of 10. `if number%2 == 0 and number%5 == 0 and not number%10 == 0`. This can be either in a single sentence or nested. These are still easier to read and understand. I am not going into more detail on how if can be used but believe me as we work with production systems with literally N variables and M possibilities and each has a branch then the condition takes the form of worst-case `MxN`.

Human registry

Humans can only track 3 to 4 items in mind at a time. We are programmers, this is like very small for our registers. Even 2 variables with 3 possibilities are 6 items and handling this in one go is confusing. A simple `if` but confused programmer. I always have the theory that the code should be written to read and maintained not completing a functionality(this is important, without functionality there is no product). Luckily we have ways to make it happen.

Avoid the `if` conditions if the conditions/branching goes above 4 items.

Let me make it a little more simple on where I am going with it. I will coin this term ==perceived linearity==. What do I mean by this? Basically, the code should look linear. Don’t get me wrong if the algorithm has a branch(if condition) then we cannot avoid it, we can either transform it or move it along the repo to make it seem the code doesn’t have branches. There are a set of ways to do it that is out of the scope of this article but I can show you some of it.

Data as we see it can be categorical, countable, or continuous. For each of these types we can have specific implementations.

Categorical

The data is categorical if the value is distinct and the set of values is limited. A variable that has layer-type of a deep learning network is categorical. Numbers can be categorical if each number is a symbol of an object, in our case the special layers that always have the same layer id. Here is a condition looking at these variables `if layer_type == 'CONVOLUTION'`, `if layer_id == 101` etc.

One common way to avoid this is using a map/dict. For each of these categorical types, we will create a map having the category as key and code block(functions) as values. We can include additional variations to this base map version. We can create a registry function that can register functions with a unique id as a key as shown below.

```python
class MethodMap:
    method_map = None

    @classmethod
    def register(cls, key, calculator_method):
        if cls.method_map == None:
            cls.method_map = {}
        cls.method_map[key] = calculator_method

    @classmethod
    def get_map(cls):
        if cls.method_map == None:
            cls.method_map = {}
        return cls.method_map
```

```python

def register_calculator(name):
    def register_func(calc_class):
        logger = MLogger.getLogger(__name__, mode='a')
        logger.debug("Registering calculator {}".format(name))
        MethodMap.register(name, calc_class())
    return register_func
```

```python
@register_calculator('CONVOLUTION')
class ConvAddressCalculator(AddressCalculator):
    def get_address(self, layer, *addr_info):
        memory_requirement = layer.get_mem_required()
        assert layer.get_stride() in [1, 2], "Stride other
        free_input_mem = not layer.sublayering
        if addr_info[0]['input'].is_type_inside_sch():
            free_input_mem = False
        if layer.get_stride() == 1:
            #computed_addr = self.memory_calculator.calcula
            computed_addr = self.memory_calculator.calculat
        else:
            computed_addr = self.memory_calculator.calculat
put_mem =free_input_mem)

        layer.set_mem_address(computed_addr)
        out_addr_info = fill_addr_info(layer, addr_info[0],
        # if layer.scratch_enable:
            # layer.scratch.forward(out_addr_info)
        return out_addr_info

```

Another example of categorical data. The set of nodes called ‘top_split_nodes’, ‘middle_split_nodes’, and ‘bottom_split_nodes' could have been merged into a single list and processed using if condition, say `if node_type == top_split_node`. But for every specific operation this `if` statement will interfere. Or we can separate the nodes and name it at a higher level and use those levels to manage the code. This is essentially what we do with any high-level programming language(C++, Python) without us knowing it.

```
# A Naive way
node_list = [set of nodes]# some are top, middle, bottom
#character of the top is not change in row
#character of the middle change the rows by 2
# character of the bottom change the rows by 3

for node in node_list:
  if self.node_type(node) == 'TOP_SPLIT':
    node.rows = node.rows
  elif self.node_type(node) == 'MIDDLE_SPLIT':
    node.rows /= 2
  elif self.node_type(node) == 'BOTTOM_SPLIT':
    node.rows /= 3

# Set of operations

for node in node_list:
  if self.node_type(node) == 'TOP_SPLIT':
    node.rows = node.rows
  elif self.node_type(node) == 'MIDDLE_SPLIT':
    node.rows /= 2
  elif self.node_type(node) == 'BOTTOM_SPLIT':
    node.rows /= 3

top_split_nodes = []
middle_split_nodes = []
bottom_split_nodes = []

for node in node_list:
  if self.node_type(node) == 'TOP_SPLIT':
    top_split_nodes.append(node)
  elif self.node_type(node) == 'MIDDLE_SPLIT':
    middle_split_nodes.append(node)
  elif self.node_type(node) == 'BOTTOM_SPLIT':
    bottom_split_nodes.append(node)

for node in top_split_nodes:
  # Do all operations needed

for node in middle_split_nodes:
  # Do all operations needed

for node in bottom_split_nodes:
  # Do all operations needed
```

In our code base

```
for node in top_split_nodes + middle_split_nodes + bottom_split_nodes:
    available_node_map[node.id_] = 1

valid_start_rows = valid_rows_list[0]
for i, node in enumerate(top_split_nodes):
    if node.is_meta_parent:
        continue
    assert node.ir.get_stride() < 3
    #if node.ir.get_stride() == 2:
    if node.ir.get_stride() == 2 and node.ir.get_filter_size()%2 != 0:
        additional_stride_rows = node.ir.get_filter_size()-1
    if node.is_subgraph_end_node:
        self.subgraph_change_details[node.id_]['num_rows'] = valid_start_
        if self.subgraph_change_details[node.id_]['num_rows'] % 2 != 0:
           self.subgraph_change_details[node.id_]['num_rows'] += 1

valid_mid_rows = (valid_rows_list[1] - valid_rows_list[0])
middle_split_offset_rows = {}
middle_split_layer_levels = {}
for i, node in enumerate(middle_split_nodes):
    if node.is_meta_parent:
        continue
    self.subgraph_change_details[node.id_]['parent_input_offset'] = {}
    assert node.ir.get_stride() < 3

    max_level_offset_rows = 0
    for p in node.parents:
        if p.id_ in middle_split_offset_rows:
            max_level_offset_rows = max(max_level_offset_rows, middle_spli
```

Countable data

An example of countable data is a set of integer numbers, a set of even numbers, set of odd numbers. In our deep learning the `filter_size` (that can be any number from 1 to N), or `stride` (can be any number).

Sometimes if the countable size is less and each value has a specific code block we can use the same method as the categorical data(substituting equal-to condition ex: `if filter_size==3)`. Make a map of `filter_size` and use the corresponding function as a key.

For other types of conditions where there is operations such as less-than, greater-than etc are involved. We actually move the `if` condition to a separate function. This is same for the continuous values also so have given an example on next heading. Below is a sample code.

```
class Convgreaterthan3x3Preprocessing(PreprocessHandler):
    def handle(self, data_node):
        node = data_node.data[0]
        parents = node.parents
        children = node.children
        act_node = None
        if node.ir.get_filter_size() > 3 and node.ir.get_layer_type() in ['CONVOL
            self.activation_layer = Activation(node.ir.layer_dict, activation_typ
            self.activation_layer.offset_select = 1
            act_node = GraphNode(self.activation_layer)
            act_node.addChildren(node.children)
            node.removeAllChild()
            node.addChild(act_node)
            act_node.addParentEdges(node)
            act_node.relinkChildren()
            parents[0].ir.auto_left_right = False
            node.ir.pre_pad = 1
            for child in act_node.children:
                child.replace_parent_edge(node.id_, act_node.id_)
            #node.ir.reset_output_channels(parents[0].ir.num_output_channels)
            data_node.update([node, act_node], mutate_more=False)
```

Continuous data

Continuous data are those that can have any value. Whether it is countable or uncountable is a separate matter. We mostly use comparative conditions like greater-than, less-than etc for the conditions involving these data. For these a common or acceptable way is we move the `if` condition to a separate function. Remember I said we cannot avoid the branching but we try to perceive it as linear. An algorithm defined definitely needs its branching the choice we have as programmers are where to place it. Below is an example of how we can handle it. We give a context to the condition using function syntax. If the below-written code didn't have this syntax it would basically be another condition in a set of conditions. We gave a context to that condition, basically saying this condition is for handling ddr_tags.

```
def handle_ddr_tags(self, nodes):
    for node in nodes:
        self.ddr_tag_node[node.id_] = 0
        if self.get_output_memory_required(node) > self.stop_memory and self.tag_node[node.id_] == self.LAST_SPLIT:
            self.ddr_tag_node[node.id_] = self.OUT_DDR
```

How to handle `if` conditions?

Removing `if` condition is basically a deliberate transformation to something we can handle. When you see an `if` condition first ask yourself, Does it confuse you? Does it confuse others? How can I remove the confusion? What tricks do I have that can make this better?

Writing that one `if` condition and solving the problem will be very easy and tempting but if it causes confusion then the first person that would affect is yourself, if not now then probably when you pass on the code to teammates and you move forward for career and company advancing work. So make a deliberate attempt to change it. Remember the whole idea is for the code to be readable and maintainable.