Programming of Neural Networks

Today: (Reverse-Mode) Automatic Differentation

Current Way

- Currently our Layers have to implement 3 functions

  - Forward(…)
  - Backward(…)
  - calculateDeltaWeights(…)

➔This can get very annoying very quickly, because:

1. We have to manually derive the equations (error prone!)
2. We have to implement all these equations (even more error prone)

Revisting the Math

- If we recall the different calculations:

- Fully Connected
  - $y = x \cdot W + b$

- A gate in a Recurrent Layer
  - $g = x \cdot W_x + h_{t-1} \cdot W_h + b_g$

➔ Somehow these are all multiplications and additions!

Revisting the Math

- In general, the math we need can be decomposed into a set of **elementary operations**, e.g.

1. Activation-Functions (Sigmoid, Tanh, Relu,…)
2. Matrix-Multiplications
3. Elementwise-Additions
4. Elementwise-Multiplications

➔ Let us revisit backpropagation with this is mind!

Revisting the Math

- What we could do, is as follows:
  1. Instead of directly defining Layers such as an LSTM-Layer , we could define Layers that represent **Elementary Operations**
  2. We can then define complex layers out of the building blocks of the elementary layers
  3. Backpropagation would work out of the box!! (Instead of stacking layers, for which the differentiation works „automatically“, we stack operations)
     ➔Computation Graph

Reverse-Mode Automatic Differentiation

- Algorithms for Automatic Differentation exist for a long time already, and have been developed separately from backpropagation

- There are multiple ways for a computer to perform an automatic derivation:
  1. By sheer Numeric calculations $f'(x_0) = \lim_{h \to 0} \frac{f(x_0+h) - f(x_0)}{h}$
  2. By symbolic differentation ➔ see Mathematica
  3. By Forward-Mode and Backward-Mode Automatic Differentation
  4. (And you can always do the math by hand and just evaluate your equations)

Reverse-Mode Automatic Differentiation

- Example: Reverse-Mode Automatic Differentation:

$$y = x_1 \cdot x_2 + \sin(x_1)$$

- Step 1: Introduce forward Variables for the user input
  $v_0 = x_1$ and $v_1 = x_2$,
  let us assume, $v_0 = 2$ and $v_1 = 3$

- Step 2: Forward
  - $v_2 = v_0 \cdot v_1 = 2 \cdot 3 = 6$
  - $v_3 = \sin(v_0) = \sin(2) = 0.9$
  - $v_4 = v_2 + v_3 = 6.9$
  - $v_5 = y = 6.9$

Reverse-Mode Automatic Differentiation

- Step 3: Backward
  - $\frac{\partial y}{\partial v_4} = 1$

$v_0 = x_1$ and $v_1 = x_2$,
let us assume, $v_0 = 2$ and $v_1 = 3$

Step 2: Forward

$$v_2 = v_0 \cdot v_1 = 2 \cdot 3 = 6$$
$$v_3 = \sin(v_0) = \sin(2) = 0.9$$
$$v_4 = v_2 + v_3 = 6.9$$
$$v_5 = y = 6.9$$

Reverse-Mode Automatic Differentiation

- Step 3: Backward
  - $\frac{\partial y}{\partial v_4} = 1$
  - $\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot 1$

$v_0 = x_1$ and $v_1 = x_2$,
let us assume, $v_0 = 2$ and $v_1 = 3$

Step 2: Forward

$$v_2 = v_0 \cdot v_1 = 2 \cdot 3 = 6$$
$$v_3 = \sin(v_0) = \sin(2) = 0.9$$
$$v_4 = v_2 + v_3 = 6.9$$
$$v_5 = y = 6.9$$

Reverse-Mode Automatic Differentiation

- Step 3: Backward
  - $\frac{\partial y}{\partial v_4} = 1$
  - $\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot 1$
  - $\frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot 1$

$v_0 = x_1$ and $v_1 = x_2$,
let us assume, $v_0 = 2$ and $v_1 = 3$

Step 2: Forward

$$v_2 = v_0 \cdot v_1 = 2 \cdot 3 = 6$$
$$v_3 = \sin(v_0) = \sin(2) = 0.9$$
$$v_4 = v_2 + v_3 = 6.9$$
$$v_5 = y = 6.9$$

Reverse-Mode Automatic Differentiation

- Step 3: Backward
  - $\frac{\partial y}{\partial v_4} = 1$
  - $\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot 1$
  - $\frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot 1$
  - $\frac{\partial y}{\partial v_1} = \frac{\partial y}{\partial v_2} \cdot \frac{\partial v_2}{\partial v_1} = 1 \cdot v_0 = 2$
  - $\frac{\partial y}{\partial v_0} = 1$

$v_0 = x_1$ and $v_1 = x_2$,
let us assume, $v_0 = 2$ and $v_1 = 3$

Step 2: Forward

$$v_2 = v_0 \cdot v_1 = 2 \cdot 3 = 6$$
$$v_3 = \sin(v_0) = \sin(2) = 0.9$$
$$v_4 = v_2 + v_3 = 6.9$$
$$v_5 = y = 6.9$$

Reverse-Mode Automatic Differentiation

- Step 3: Backward
  - $\frac{\partial y}{\partial v_4} = 1$
  - $\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot 1$
  - $\frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot 1$
  - $\frac{\partial y}{\partial v_1} = \frac{\partial y}{\partial v_2} \cdot \frac{\partial v_2}{\partial v_1} = 1 \cdot v_0 = 2$
  - $\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2} \cdot \frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_0} = 1 \cdot v_1 + 1 \cdot \cos(v_0) = 3 + \cos(2) = 2.58$

$v_0 = x_1$ and $v_1 = x_2$,
let us assume, $v_0 = 2$ and $v_1 = 3$

Step 2: Forward

$$v_2 = v_0 \cdot v_1 = 2 \cdot 3 = 6$$
$$v_3 = \sin(v_0) = \sin(2) = 0.9$$
$$v_4 = v_2 + v_3 = 6.9$$
$$v_5 = y = 6.9$$

Reverse-Mode Automatic Differentiation

- Is this just the same as Backpropagation??

➔No, but almost, the algorithm is also called **Generalized Backpropagation**

➔Reverse A-D also covers the case where the final output is not a scalar!

Our Loss

➔ I bet you didnt know that you know you knew it!

Final thoughts:

- When implementing Neural Networks, thinking in layers (which appeared rather intuitive at first glance) might be hindering.

➔ Better to think of a layer as an operation that modifies the computation graph

➔ After all layers added their elementary operations to the computation graph, it can be **compiled** (and maybe some optimizations applied!)

➔ Ram is allocated due to this graph, and the graph is executed on the GPU

What about the framework?

- I am giving some details now, as of how you should/could (if you want to) modify your framework.

- The resulting API is pretty close to that of PyTorch
- Be aware, that this is alot of work to get it done correctly

Revisiting the data structures

- We will still use Tensors to store the data of our computations

➔ But Tensors should now have access to the computation graph

➔ Easiest is to store the operation that created them

Tensor

elements: FloatArray
shape: Shape
producingOp: ElementaryOperation

The computation Graph

- Analogous to the previous Network
- Can forward, backward through operations
- Have access to a cache

ComputationGraph

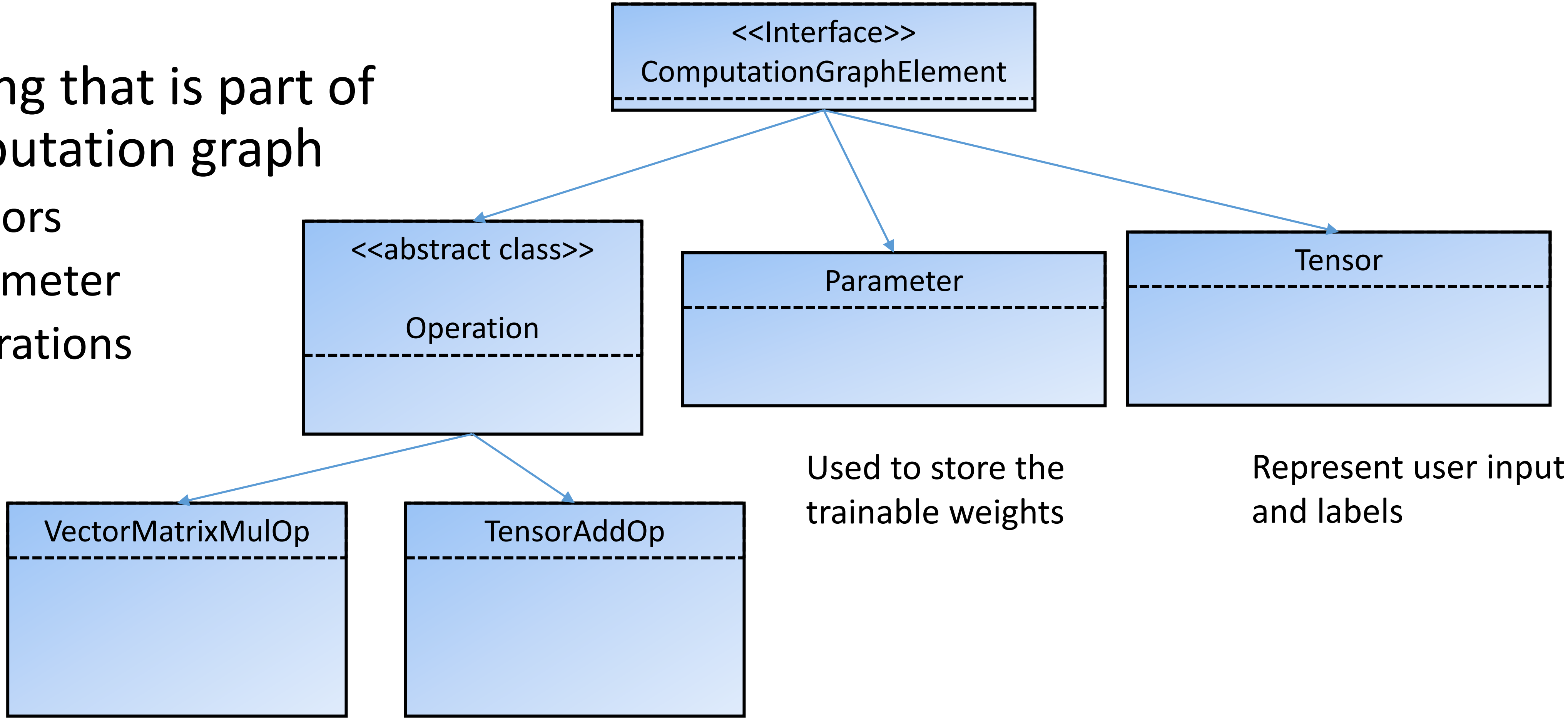
---

graphElements: List<CompGraphElement>

forward:
backward:
backprop:

Computation Graph Elements

- Anything that is part of a computation graph
  - Tensors
  - Parameter
  - Operations

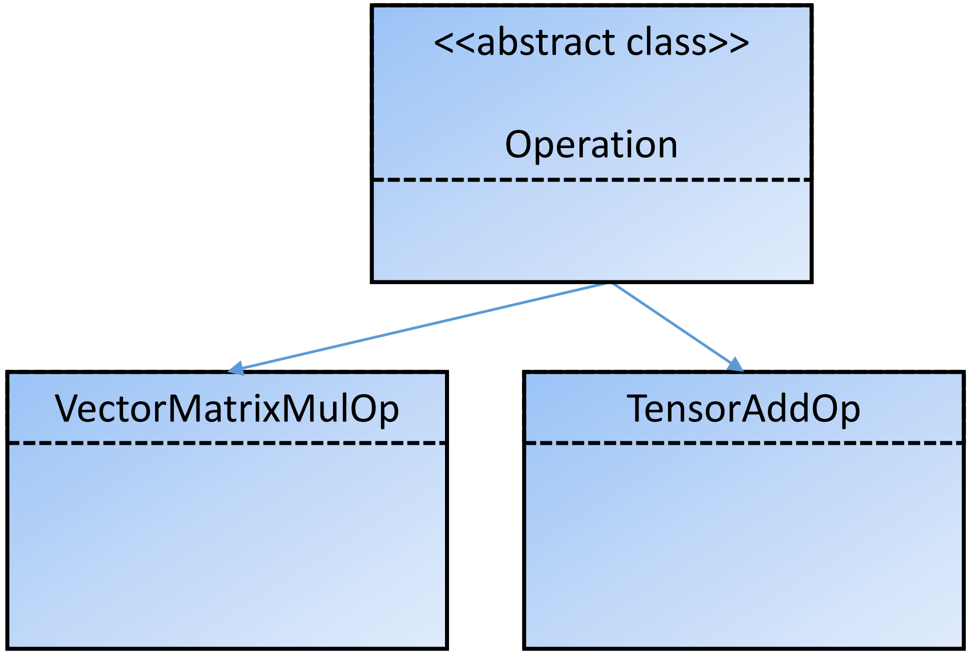


Used to store the trainable weights

Represent user input and labels

Operations

- Operations replace what we modelled as layers
- They store the tensor(s) they forward
- Have access to all computation graph elements that they get as input
- Know all successors



Layer

- A layer is now syntactic sugar, that adds operations into a computation graph
- Can hide the operation manipulation using operator overloading
- No more backward/paramUpdates required

```
class FullyConnectedLayer(inLayer: NetworkLayer, val hiddenUnits: Int) : NetworkLayer(inLayer) {

    lateinit var weights: Parameter
    lateinit var bias: Parameter

    override fun forward(inTensors: List<FloatTensor>): List<FloatTensor> {
        require(inTensors.size == 1)
        val inVec : Vector = inTensors.first() as Vector

        val result : FloatTensor = (inVec * weights) + bias
        return listOf(result)
    }
}
```

Recap

- We started modelling out neural network using layers
- Learned backpropagation and got a feeling for the chain rule
- Struggled through Convolution and Recurrent Layers
- Got frustrated about the annoying backward functions
- Realized that we always use the same building blocks
- Generalized backpropagation into Reverse Mode Automatic Differentiations
- Gave some hints as of how one can create a nice API

Outlook

- Things we have not talked about:
  1. How to exactly integrate batches
  2. What is different, when we calculate on the GPU
  3. How to model traditional inputs (text, audio, video)
  4. More strategies to prevent overfitting (e.g regularization, data augmentation)
  5. More advanced Neural Building Blocks:
     1. Neural Comparator
     2. Neural Conditional Random Fields
     3. Attention and Transformer
     4. Sequence-To-Sequence models
     5. Tensor-To-Tensor models (e.g. Graph Neural Networks)