INFR10086 Machine Learning (MLG) Semester 1, 2022/23

Tutorial 4: Computation Graphs

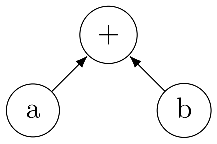
In this tutorial, we will write a simple program that does automatic differentiation on computation graphs.

1 Computation Graphs

Computation graphs are the main data structure for automatic differentiation. Automatic differentiation is central to training neural networks.[1] Before tools for automatic differentiation exist, we have to derive and implement gradients every time we come up with a new network architecture. Implementing gradients is not only time consuming but also error prone.

A computation graph is a regular graph with nodes and edges. Each node has a specified computation `type`, e.g., addition. Each node also stores the `value` after the computation. The edges signal where each node takes its arguments from. For every edge, we have a parent node and a child node, and the child points to the parent.

Imagine a use case where we want to represent the following graph.



Ideally, we want to write

```
a = ComputeNode('var', 0)
b = ComputeNode('var', 1)
c = ComputeNode('add', 2, a, b)
```

to represent the graph above, where the first argument is a `type`, the second argument is an `id` for a node useful for debugging, and the rest of the arguments are the children of a node. Given such a use case, we can write the following `ComputeNode` class.

```
class ComputeNode:
    def __init__(self, type, id, *children):
        self.type = type
        self.id = id
        self.children = []
        self.children.extend(children)
        self.value = None
        self.grad = None
```

[1]The two mainstream tools for running and training neural networks are pytorch `https://pytorch.org` and tensorflow `https://www.tensorflow.org`. We will write a mini version of these tools in less than 150 lines of python.

The members `value` and `grad` will get assigned later when the computation actually happens.

Once `ComputeNode` class is ready, we need a `Graph` class to keep track of all the nodes. For example, we can write

```
g = Graph()
a = g.var(1)
b = g.var(2)
c = g.add(a, b)
```

to represent the same computation of addition, except that now we have the value 1 assigned to `a` and the value 2 assigned to `b`. Instead of manually assigning ids to nodes, we will let the `Graph` class automatically increase ids for us. To achieve these, the `Graph` class could look like the following.

```
class Graph:
    def __init__(self):
        self.nodes = []

    def var(self, value):
        a = ComputeNode('var', len(self.nodes))
        a.value = value
        self.nodes.append(a)
        return a

    def add(self, a, b):
        v = ComputeNode('add', len(self.nodes), a, b)
        self.nodes.append(v)
        return v

    def mul(self, a, b):
        v = ComputeNode('mul', len(self.nodes), a, b)
        self.nodes.append(v)
        return v
```

We can extend this to other computation types, such as the multiplication above.

1.1 Forward Computation

Note that the classes `ComputeNode` and `Graph` merely store the computations needed. No actual computation has happened yet. To populate the values, we go over the nodes in the order they are created, and compute the values given the arguments from the children. This can be implemented with a for loop. Since the computation follows the directions of the edges, we call this the forward computation.

```
def forward(graph):
    for n in graph.nodes:
        if n.type == 'var':
            # do nothing
            pass
        elif n.type == 'add':
            n.value = n.children[0].value + n.children[1].value
        elif n.type == 'mul':
            n.value = n.children[0].value @ n.children[1].value
        else:
            print('unknown node type: {}'.format(n.type))
```

Discussion. How do we know that when a parent node computes its value, the children already have their values computed?

Discussion. When we go over the nodes in topological order the children are guaranteed to be ready before the parents. What is a topological order?

Discussion. Why do we not need to run topological sort? Why is `graph.nodes` already in topological order?

Now that the `forward` is in place, we can write

```
g = Graph()
a = g.var(1)
b = g.var(2)
c = g.add(a, b)
forward(graph)
print(c.value) # => 3
```

to compute the values of the computations.

1.2 Backward Computation

Remember that the goal is to automate the computation of gradients. In particular, we need to compute gradient `grad` to every `ComputeNode` in the `Graph`. In the end, we will use the computation graph to compute a loss function L, since this is a machine learning problem. We need the gradients to the parameters, and use them to do stochastic gradient descent (SGD).

As an example, suppose addition $f(a, b) = a + b$ is involved in computing L. We will need to compute $\frac{\partial L}{\partial f}$ first and then compute $\frac{\partial L}{\partial a}$ and $\frac{\partial L}{\partial b}$. Formally, by chain rule, we have

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial a} = \frac{\partial L}{\partial f}. \tag{1}$$

The expression $\frac{\partial f}{\partial a} = 1$ because $f(a, b) = a + b$. Similarly, we have

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial b} = \frac{\partial L}{\partial f}. \tag{2}$$

In words, a and b merely copy the gradient stored in f.

A more interesting example would be matrix multiplication $g(x, A) = xA$. Note that x here is a row vector, not a column vector. By chain rule, we have

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial g}\frac{\partial g}{\partial x} = \frac{\partial L}{\partial g} A^\top. \tag{3}$$

Note again that $\frac{\partial L}{\partial g}$ is a row vector, not a column vector. For the other argument, we have

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial g}\frac{\partial g}{\partial A} = x^\top \frac{\partial L}{\partial g}. \tag{4}$$

Because both are row vectors, $x^\top \frac{\partial L}{\partial g}$ is the outer product of the two, not the inner product.

When computing gradients, note that $\frac{\partial L}{\partial g}$ needs to be ready before computing $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial A}$. The computation is going in the opposite direction of the edges. Instead of following the order `graph.nodes`, we go in the reversed order. We can write the computation again as a for loop.

```
def backward(graph):
    for n in reversed(graph.nodes):
        if n.type == 'var':
            # do nothing
            pass
        elif n.type == 'add':
            init_grad(n.children[0])
            init_grad(n.children[1])
            n.children[0].grad += n.grad
            n.children[1].grad += n.grad
        elif n.type == 'mul':
            init_grad(n.children[0])
            init_grad(n.children[1])
            n.children[0].grad += n.grad @ n.children[1].value.T
            n.children[1].grad += numpy.outer(n.children[0].value, n.grad)
        else:
            print('unknown node type: {}'.format(n.type))
```

Since the gradients for the children need to be initialized first, we have the following function to save us from repeating it over and over.

```
def init_grad(n):
    if n.grad is None:
        n.grad = numpy.zeros_like(n.value)
```

The loop is called the backward computation, or better known as backpropagation.

Discussion. When going in the reversed order, are the gradients of the parents guaranteed to be ready before computing the gradients of the children?

Discussion. Why do we not need to initialize in the forward computation?

Discussion. Note that the gradients are accumulated with +=. Why is this necessary?

To complete the library, we need more computation types and loss functions. A few minimal ones are implemented in `nn.py`. We will use digit classification as an example to demonstrate how to use the library.

2 Linear Classification

We will use the data set MNIST. The input is a 28×28 image, and we will simply represent it as a $\mathbb{R}^{784}$ vector. Loading images and labels are already written in `mnist.py`. Since we are doing SGD, we sample `img` and `label` from the data set. Below is a snippet of computing $w^\top x + b$ and the cross entropy loss for that single sample.

```
g = nn.Graph()
w = g.var(param_w)
b = g.var(param_b)
x = g.var(img)
y = g.var(nn.one_hot(label, 10))
score = g.add(g.mul(x, w), b)
loss = g.cross_entropy(score, y)
```

If we have a closer look at the cross entropy loss in the file `nn.py`, the cross entropy loss is implemented by first computing the log probability using softmax

```
    # in class Graph
    def cross_entropy(self, pred, gold):
        logprob = self.logsoftmax(pred)
        v = ComputeNode('cross-entropy', len(self.nodes), logprob, gold)
        self.nodes.append(v)
        return v
```

and then take the expectation (a weighted sum) using a dot product

```
        # in function forward
        elif n.type == 'cross-entropy':
            n.value = (-1) * numpy.dot(n.children[0].value, n.children[1].value)
```

Implementation. Run

```
  $ ./lin-train.py lin-param-0 lin-param-1 > log-1
```

Discussion. Read `lin-train.py`. Where is SGD implemented?

Implementation. Write a short script to average the loss values in `log-1`.

Implementation. Repeat and run `lin-train.py` to produce `lin-param-2`, . . . , `lin-param-20` and `log-2`, . . . , `log-20`.

Implementation. Plot the losses on the y-axis and epochs on the x-axis.

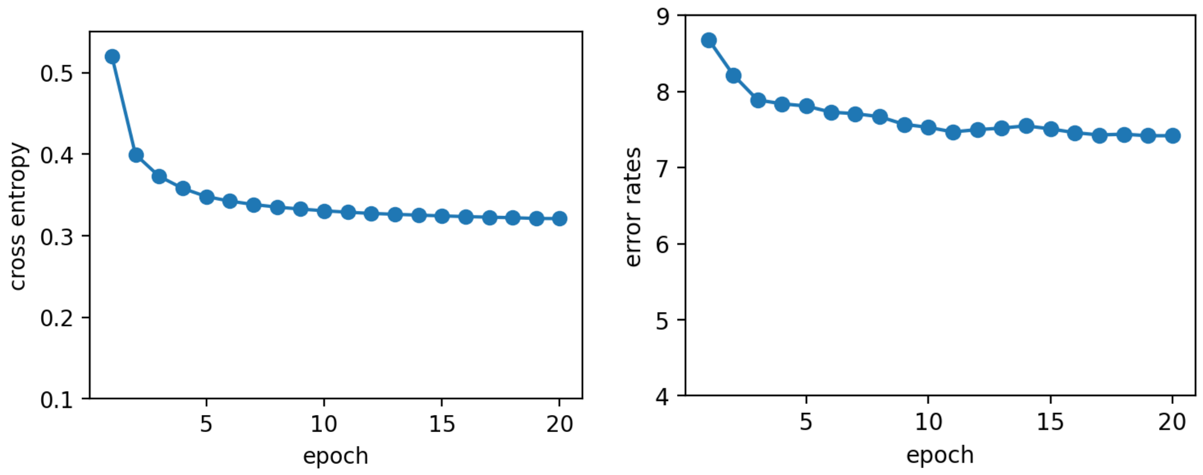


Figure 1: The cross entropy during training and error rates on the test set for linear classification.

Implementation. Run

```
$ ./lin-test.py lin-param-20
```

to get the error rate on the test set.

Implementation. Compute the error rates for all epochs. Plot the error rates on the y-axis and epochs on the x-axis.

The results are shown in the Figure 1.

Discussion. Why do we plot the error rates on the test set?

Discussion. Is it a good practice to measure performance on the test set?

We have written a short program to do automatic differentiation on computation graphs. This framework can be easily extended to multilayer perceptrons (MLP). Below we show the snippet for computing the loss with an MLP.

```
g = nn.Graph()
w1 = g.var(param_w1)
b1 = g.var(param_b1)
w2 = g.var(param_w2)
b2 = g.var(param_b2)
x = g.var(img)
y = g.var(nn.one_hot(label, 10))
score = g.add(g.mul(g.logistic(g.add(g.mul(x, w1), b1)), w2), b2)
```

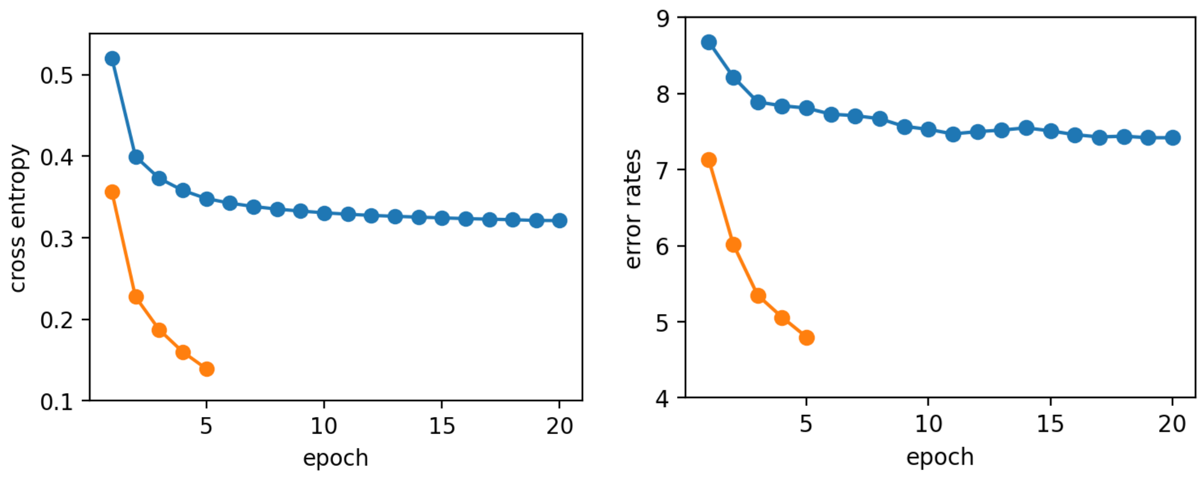


Figure 2: The cross entropy during training and error rates on the test set for MLPs.

```
loss = g.cross_entropy(score, y)
```

Implementation. Run

```
$ ./mlp-train.py mlp-param-0 mlp-param-1 > log-1
```

Implementation. Repeat and run `mlp-train.py` to produce `mlp-param-2`, . . . , `mlp-param-5` and `log-2`, . . . , `log-5`.

Implementation. Overlay the losses of MLPs and linear classifiers.

Implementation. Run

```
$ ./mlp-test.py mlp-param-5
```

to get the error rate on the test set.

Implementation. Overlay error rates of MLPs and linear classifiers.

The results are shown in the Figure 2. Even if we only run 5 epochs, it is pretty clear that 1) neural networks fit the data better than linear classifiers without using a feature function[2] and 2) training neural networks requires significantly more computation than a linear classifier. Dedicated hardware, such as GPUs, is needed to train large neural networks, and is beyond the scope of this course.

[2]Linear classifiers with the proper feature function can actually perform quite well, if not better than neural networks on this data set.