The Forward-Forward Algorithm: Some Preliminary Investigations

Geoffrey Hinton
Google Brain
geoffhinton@google.com
What is wrong with backpropagation

1. the perceptual system needs to perform inference and learning in real time without stopping to perform backpropagation

2. it requires perfect knowledge of the computation performed in the forward pass in order to compute the correct derivatives. (It is possible to resort to reinforcement learning)
The Forward-Forward Algorithm

- Replace the forward and backward passes of backpropagation by two forward passes.

- **Positive pass** on real data adjusts the weight to increase the goodness in every hidden layer.

- **Negative pass** on negative data adjusts the weight to decrease the goodness in every hidden layer.

- **The aim of the learning** is to make the goodness be well **above** \( \theta \) for real data and well **below** \( \theta \) for negative data.

\[
\text{Goodness} = \sum_j y_j^2 \quad y_j \text{ is the activity of hidden unit } j
\]

\[
p(\text{positive}) = \sigma \left( \sum_j y_j^2 - \theta \right)
\]

\( \theta \): Hyperparameter
\( \sigma \): logistic function
Where does the real & negative data from?

The second way to get negative data:
Negative data is generated by doing a single forward pass through the net to get probabilities for all the classes and then choosing between the incorrect classes in proportion to their probabilities. This makes the training much more efficient.
The Forward-Forward Algorithm

- **Positive pass** on real data adjusts the weight to **increase the goodness** in every hidden layer.

- **Negative pass** on negative data adjusts the weight to **decrease the goodness** in every hidden layer.

The aim of the learning is to make the goodness be well above some threshold value for real data and well below that value for negative data.
How to predict

Goodness = \sum_j y_j^2
Another way to predict (quick but sub-optimal)

Goodness = \sum_j y_j^2

784 x 1000

ReLU

Layer normalization

FC

ten entries of 0.1

softmax

N x 10
The advantage of FF

• It can be used when the precise details of the forward computation are unknown.

• It can learn while pipelining sequential data through a neural network ever without storing the neural activities or stopping to propagate error derivatives.
What is wrong with FF

- FF is somewhat slower than backpropagation

- FF does not generalize quite as well as backpropagation

- What is learned in later layers cannot affect what is learned in earlier layers.
How to resolve the third disadvantage?

What is learned in later layers cannot affect what is learned in earlier layers.

Activity vector $b = 0.3 \times \text{Activity vector } a + 0.7 \times \ln(\text{Activity vector } c)$
### Experiments with CIFAR-10

<table>
<thead>
<tr>
<th>learning procedure</th>
<th>testing procedure</th>
<th>number of hidden layers</th>
<th>training % error rate</th>
<th>test % error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td></td>
<td>2</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>FF min ssq</td>
<td>compute goodness for every label</td>
<td>2</td>
<td>20</td>
<td>41</td>
</tr>
<tr>
<td>FF min ssq</td>
<td>one-pass softmax</td>
<td>2</td>
<td>31</td>
<td>45</td>
</tr>
<tr>
<td>FF max ssq</td>
<td>compute goodness for every label</td>
<td>2</td>
<td>25</td>
<td>44</td>
</tr>
<tr>
<td>FF max ssq</td>
<td>one-pass softmax</td>
<td>2</td>
<td>33</td>
<td>46</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td>3</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>FF min ssq</td>
<td>compute goodness for every label</td>
<td>3</td>
<td>24</td>
<td>41</td>
</tr>
<tr>
<td>FF min ssq</td>
<td>one-pass softmax</td>
<td>3</td>
<td>32</td>
<td>44</td>
</tr>
<tr>
<td>FF max ssq</td>
<td>compute goodness for every label</td>
<td>3</td>
<td>21</td>
<td>44</td>
</tr>
<tr>
<td>FF max ssq</td>
<td>one-pass softmax</td>
<td>3</td>
<td>31</td>
<td>46</td>
</tr>
</tbody>
</table>

**Conclusion:** Although the test performance of FF is worse than backpropagation it is only slightly worse, even when there are complicated confounding backgrounds.

Max(min) ssq: maximize(minimize) the sum of the squared activities
Other interesting points

- An energy efficient way to multiply an activity vector by a weight matrix is to implement activities as voltages and weights as conductances.
- There are many other possible activation function to explore in the context of FF.
- Mortal Computation makes trillion parameter neural net to consume a few watts.
```python
class Layer(nn.Linear):
    def __init__(self, in_features, out_features,
                 bias=True, device=None, dtype=None):
        super().__init__(in_features, out_features, bias, device, dtype)
        self.relu = torch.nn.ReLU()
        self.opt = Adam(self.parameters(), lr=0.03)
        self.threshold = 2.0
        self.num_epochs = 500

    def forward(self, x):
        x_direction = x / (x.norm(2, 1, keepdim=True) + 1e-4)
        return self.relu(
            torch.mm(x_direction, self.weight.T) +
            self.bias.unsqueeze(0))

    def train(self, x_pos, x_neg):
        for i in tqdm(range(self.num_epochs)):
            g_pos = self.forward(x_pos).pow(2).mean(1)
            g_neg = self.forward(x_neg).pow(2).mean(1)

            loss = torch.log(1 + torch.exp(torch.cat([
                -g_pos + self.threshold,
                g_neg - self.threshold])))
            self.opt.zero_grad()
            loss.backward()
            self.opt.step()

        return self.forward(x_pos).detach(), self.forward(x_neg).detach()
```