Modified Backpropagation for Idempotent Neural Networks

Nikolaj B. Jensen

September 2024

1 Development of the method

1.1 Why $3\mathbf{A}^2 - 2\mathbf{A}^3$ is special

We know that the expression $3\mathbf{A}^2 - 2\mathbf{A}^3$ has the particular property that it is a recurrence relation which finds idempotent $\mathbf{A}$; that is, $\mathbf{A}' = 3\mathbf{A}^2 - 2\mathbf{A}^3$ leads to an $\mathbf{A}^*$ whose eigenvalues are either 0 or 1 after sufficiently many steps.

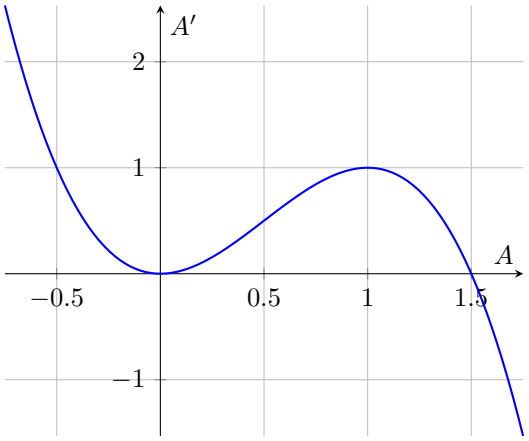

Figure 1: Plot of $\mathbf{A}' = 3\mathbf{A}^2 - 2\mathbf{A}^3$ in the case where $\mathbf{A}$ is a scalar.

We now desire to transfer this recurrence property into our learning algorithm. This is fundamentally different from ordinary backpropagation, which relies on the gradient of some function to find a local minimum. Here, the function $h(\mathbf{A}) = 3\mathbf{A}^2 - 2\mathbf{A}^3$ tells us all we need to find solutions we want (either 0 or 1), so we do not need to take any gradients.

Another way to look at this is to consider how we can use ordinary backpropagation to find idempotent matrices. Since backpropagation finds local minima, a reasonable way to do this is to minimize $f(\mathbf{A}) = \|\mathbf{A}^2 - \mathbf{A}\|$. Note, however, that f does *not* have the particular recurrence property that we want. In other words, $\mathbf{A}' = \|\mathbf{A}^2 - \mathbf{A}\|$ does not lead to an $\mathbf{A}^*$ whose eigenvalues are 0 or 1 after sufficiently many steps[1].

This brings the first important realization:

Realization 1 It is not trivial to find idempotent solutions to $h(\mathbf{A}) = 3\mathbf{A}^2 - 2\mathbf{A}^3$ using ordinary backpropagation with h as a meaningful part of the loss function, as the types of solutions (namely local minima) are entirely irrelevant to $h(\mathbf{A})$. Instead, we need to exploit the recurrence property of $h(\mathbf{A})$ which our previous analysis already shows finds the solutions we want. **To use $h(\mathbf{A})$ in finding idempotent neural networks therefore requires a fundamental modification to backpropagation.**

[1] Actually, it does. For any input $\mathbf{A}$ the result of applying this recurrence relation sufficiently many times will be that its eigenvalues all become 0 (that is, it will be the null matrix). But a method to find null matrices is not particularly interesting.

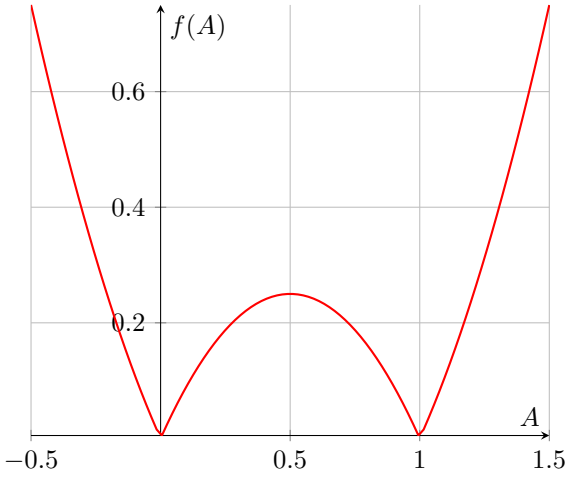

Figure 2: Plot of $f(\mathbf{A}) = \|\mathbf{A}^2 - \mathbf{A}\|$ in the case where $\mathbf{A}$ is a scalar.

1.2 A naive approach

Let us then consider how $3\mathbf{A}^2 - 2\mathbf{A}^3$ can be used to learn linear idempotent neural networks. Consider an $n_{\mathbf{W}}(\mathbf{x}) = \mathbf{W}\mathbf{x}$, where $\mathbf{W} \in \mathbb{R}^{n\times m}$ is a weight matrix, and $\mathbf{x} \in \mathbb{R}^{m\times 1}$ is an "input" such that $n_{\mathbf{W}}(\mathbf{x}) \in \mathbb{R}^{n\times 1}$ is the application of a neural network to some input. Clearly, we must have $m = n$ to satisfy idempotency.

In this situation, our recurrence relation could be redefined as $3n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})) - 2n_{\mathbf{W}}(n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})))$.

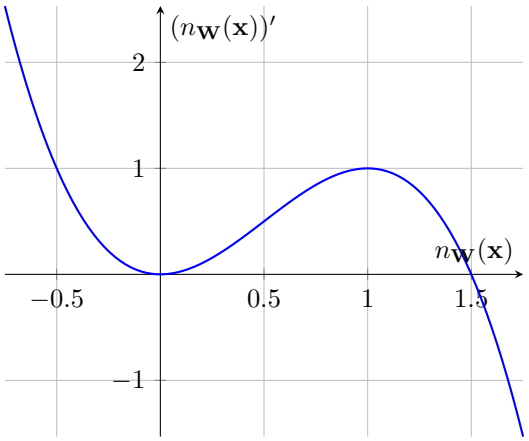

Figure 3: Plot of $(n_{\mathbf{W}}(\mathbf{x}))' = 3n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})) - 2n_{\mathbf{W}}(n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})))$.

Note that this recurrence relation now operates in $n_{\mathbf{W}}(\mathbf{x}) \in \mathbb{R}^{n\times 1}$ space whereas $\mathbf{W} \in \mathbb{R}^{n\times n}$ space. Thus, this recurrence relation does not directly allow us to update the weights of n, but only describes how n **applied to some** $\mathbf{x}$ should be updated.

Realization 2 We need to find a "meaningful" link between updating $n_{\mathbf{W}}(\mathbf{x})$ and updating $\mathbf{W}$.

$$\Delta n_{\mathbf{W}}(\mathbf{x}) \quad \rightsquigarrow \quad \Delta\mathbf{W}$$

We have found such a link (described below), but we are yet to prove why it is meaningful.

We may interpret Figure 3 as describing the difference between $(n_{\mathbf{W}}(\mathbf{x}))'$ and $n_{\mathbf{W}}(\mathbf{x})$:

$$\begin{aligned}\Delta n_{\mathbf{W}}(\mathbf{x}) &= (n_{\mathbf{W}}(\mathbf{x}))' - n_{\mathbf{W}}(\mathbf{x}) \\ &= 3n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})) - 2n_{\mathbf{W}}(n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x}))) - n_{\mathbf{W}}(\mathbf{x})\end{aligned}$$

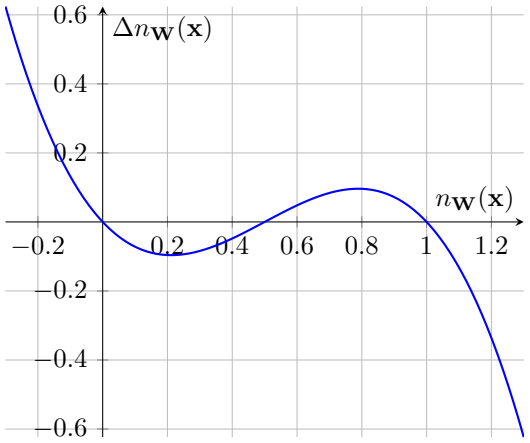

Figure 4: Plot of $\Delta n_{\mathbf{W}}(\mathbf{x}) = 3n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})) - 2n_{\mathbf{W}}(n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x}))) - n_{\mathbf{W}}(\mathbf{x})$.

To form a $\Delta\mathbf{W} \in \mathbb{R}^{n\times n}$ using $\Delta n_{\mathbf{W}}(\mathbf{x}) \in \mathbb{R}^{n\times 1}$ we can take the outer product with the input $\mathbf{x} \in \mathbb{R}^{n\times 1}$:

$$\Delta\mathbf{W} = (\Delta n_{\mathbf{W}}(\mathbf{x}))\mathbf{x}^T$$

We may then consider taking the average of such $\Delta\mathbf{W}$ for many $\mathbf{x}$ from a dataset, and take a small step in this direction:

$$\mathbf{W}' = \mathbf{W} + \eta\Delta\mathbf{W}$$

In this document we call this approach M1. Note that this approach is restricted to the following ways:

- Fully connected neural networks (no CNNs, RNNs, etc.)
- Must have exactly one weight matrix (no multi-layer networks)
- Does not modify biases

1.3 Scaling up the naive approach

We want to solve the problems with the naive approach as outlined in the previous section. Here we show that backpropagation can be modified to use the idea captured by the previous section. First, note that backpropagation is a special case of gradient descent which updates weights using the recurrence relation

$$\mathbf{W}' = \mathbf{W} + \eta\frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial\mathbf{W}}$$

where E is an unspecified loss function. Letting $\mathbf{y} = n_{\mathbf{W}}(\mathbf{x})$ we see that the chain rule gives us

$$\frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial\mathbf{W}} = \frac{\partial(-E(\mathbf{y}))}{\partial\mathbf{y}}\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial\mathbf{W}}$$

Since $\frac{\partial(-E(\mathbf{y}))}{\partial\mathbf{y}}$ describes a change in the loss for a small change in the output of n, we can substitute

$$\begin{aligned}\frac{\partial(-E(\mathbf{y}))}{\partial\mathbf{y}} &= \Delta n_{\mathbf{W}}(\mathbf{x}) \\ &= 3n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x})) - 2n_{\mathbf{W}}(n_{\mathbf{W}}(n_{\mathbf{W}}(\mathbf{x}))) - n_{\mathbf{W}}(\mathbf{x})\end{aligned}$$

The quantity $\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial\mathbf{W}}$ can be left for optimized backpropagation algorithms to find, as it is entirely dependent on the architecture of the underlying network n. Thus, we have:

$$\frac{\partial E(n_{\mathbf{W}}(\mathbf{x}))}{\partial\mathbf{W}} = \Delta n_{\mathbf{W}}(\mathbf{x})\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial\mathbf{W}}$$

In this document we call this approach "modified backpropagation".

Note 1 In the case where $n_{\mathbf{W}}(\mathbf{x}) = \mathbf{W}\mathbf{x}$, M1 is a special case of the modified backpropagation approach. This can be seen by observing that $\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}} = \mathbf{x}^T$, hence $\frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial \mathbf{W}} = (\Delta n_{\mathbf{W}}(\mathbf{x}))\mathbf{x}^T$.

2 Algorithmic differences from ordinary backpropagation

In the last section we introduced approaches M1 and modified backpropagation as ways of leveraging the recurrence property of $3\mathbf{A}^2 - 2\mathbf{A}^3$ in the context of neural networks. We showed that M1 is a special case of modified backpropagation. In this section, we shall compare the algorithmic differences between modified backpropagation and ordinary backpropagation.

Here are the approaches side by side:

Modified Backpropagation	**Ordinary Backpropagation**
Update rule: $$\mathbf{W}' = \mathbf{W} + \eta \frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial \mathbf{W}}$$ where $$\frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial \mathbf{W}} = \frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}} \frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}}$$ $$\frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}} := \Delta n_{\mathbf{W}}(\mathbf{x})$$ $\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}}$ is found ordinarily.	Update rule: $$\mathbf{W}' = \mathbf{W} + \eta \frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial \mathbf{W}}$$ where $$\frac{\partial(-E(n_{\mathbf{W}}(\mathbf{x})))}{\partial \mathbf{W}} = \frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}} \frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}}$$ $\frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}}$ and $\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}}$ are found ordinarily.

Note that the only major difference between the two approaches is to do with $E : \mathbb{R}^{m \times k} \to \mathbb{R}$. Both modified backpropagation and ordinary backpropagation *evaluate* E the same way:

$$E(\mathbf{y}) = \frac{1}{k} \sum_{i=1}^{k} (n_{\mathbf{W}}(\mathbf{y})^{(i)} - \mathbf{y}^{(i)})^2$$

but they differ in how they calculate the quantity $\frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}}$:

- **Modified Backpropagation** defines $\frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}}$ to be $\Delta n_{\mathbf{W}}(\mathbf{x})$.
- **Ordinary Backpropagation** defines $\frac{\partial(-E(\mathbf{y}))}{\partial \mathbf{y}}$ to be the usual value.

2.1 PyTorch implementation

In PyTorch, the mathematical function E can be implemented as a module:

```
class ELoss(nn.Module):
def __init__(self, net):
super(ELoss, self).__init__()
self.net = net

def forward(self, y):
y2 = self.net(y)
return torch.mean((y2 - y) ** 2)
```

This is all that is required to use ordinary backpropagation. To implement the modified gradient used for modified backpropagation, we define the following PyTorch Autograd Function:

```
class E(torch.autograd.Function):
@staticmethod
def loss_fn(y, net):
loss = torch.mean((net(y) - y) ** 2)
return loss

@staticmethod
def forward(ctx, y, net):
ctx.save_for_backward(y)
ctx.net = net

return E.loss_fn(y, net)

@staticmethod
def backward(ctx, grad_output):
y, = ctx.saved_tensors
net = ctx.net

y2 = net(y)
y3 = net(y2)
e = 3*y2 - 2*y3 - y
grads = -e / e.shape[0]
return grads * grad_output, None

class ELoss(torch.nn.Module):
def __init__(self, net, use_mod = False):
super(ELoss, self).__init__()
self.net = net
self.use_mod = use_mod

def forward(self, y):
if self.use_mod:
return E.apply(y, self.net)
else:
return E.loss_fn(y, self.net)
```

This allows us to train the neural network using modified backpropagation in the following way:

```
net = Net()
opt = torch.optim.SGD(net.parameters(), lr=0.1)
criterion = ELoss(net, use_mod=True)

opt.zero_grad()
y = net(x)
loss = criterion(y)
loss.backward()
opt.step()
```

The `use_mod` flag for `ELoss` configures the loss function to use either the `E.backward` or default implementations when computing the gradient, resulting in modified backpropagation or ordinary backpropagation, respectively.

2.2 Verification of PyTorch implementation

In PyTorch, most operations between PyTorch tensors implicitly construct a computational graph which is used to efficiently compute gradients for backpropagation. We can sanity-check the PyTorch implementation above by inspecting the computational graph for both modified backpropagation and ordinary backpropagation.

The computational graphs are constructed using the same single-layer neural network without biases and no activation function. We therefore have $n_{\mathbf{W}}(\mathbf{x}) = \mathbf{W}\mathbf{x}$. **PyTorch, however, represents the input $\mathbf{x}$ transposed, hence the following use $n_{\mathbf{W}}(\mathbf{x}) = \mathbf{W}\mathbf{x}^T = \mathbf{x}\mathbf{W}^T$ as the definition.** We show the graphs describing the computation of gradients for $\mathbf{W}$ for a single optimization step. Below is the code used to generate the computational graphs.

```
    def sanity_check(net, x, mode):
    net_copy = deepcopy(net)
    opt = torch.optim.SGD(net_copy.parameters(), lr=0.1)
    criterion = ELoss(net_copy, use_mod=(mode == Algorithm.MOD_AUTODIFF))

    opt.zero_grad()
    y = net_copy(x)
    loss = criterion(y)
    loss.backward(create_graph=True)

    grads = net_copy.main[0].weight.grad
    print(f"Grads: \n{grads}")
    dot = make_dot(grads, show_attrs=True, show_saved=True, params=dict(list(net_copy.
                                                named_parameters())))
    display(dot)
    return dot


    net = Net(dims=5)
    inp = torch.randn((3, 5))
    sanity_check(net, inp, Algorithm.AUTODIFF).render('autodiff')
    sanity_check(net, inp, Algorithm.MOD_AUTODIFF).render('mod_autodiff')
```

In these graphs, the weight matrix $\mathbf{W}$ is the blue box, and the green box at the bottom of the graph is the gradient matrix $\frac{\partial E(n_{\mathbf{W}}(\mathbf{x}))}{\partial \mathbf{W}}$. Yellow boxes of size 3×5 denote the input $\mathbf{x}$ consisting of 3 samples with 5 features each.

The red line in Figure 5 denotes the intuitive split by the chain rule. Computation above the line corresponds to $\frac{\partial E(\mathbf{y})}{\partial \mathbf{y}}$ and computation below the line corresponds to $\frac{\partial n_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}}$, which altogether gives $\frac{\partial E(n_{\mathbf{W}}(\mathbf{x}))}{\partial \mathbf{W}}$. Chasing through the graph shows that it indeed does use exactly the algorithm outlined above.

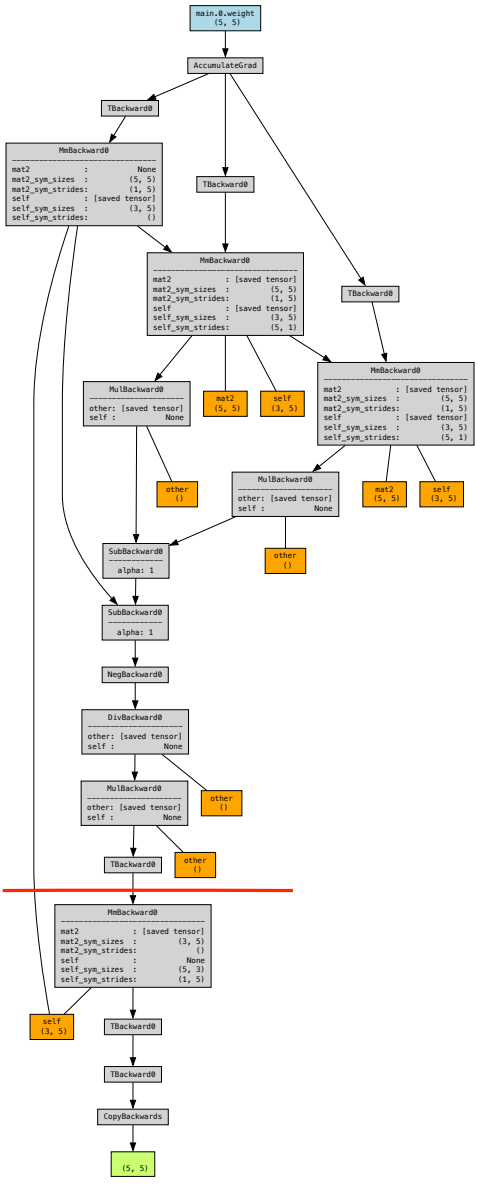

Figure 5: Computational graph for gradient calculation using **modified** backpropagation.

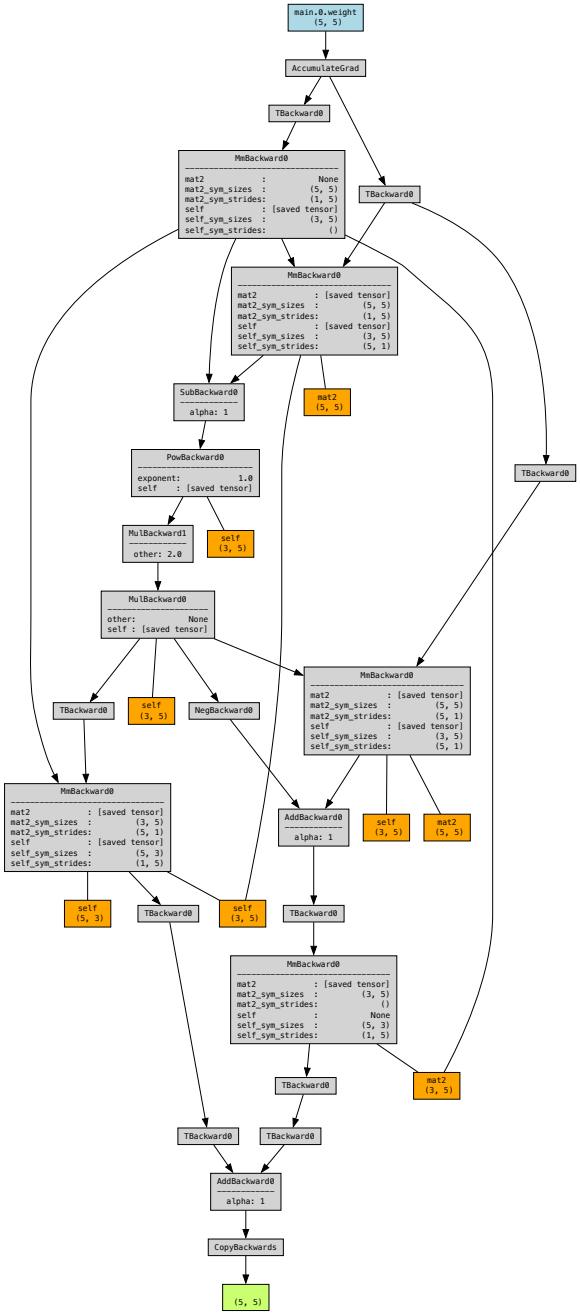

Figure 6: Computational graph for gradient calculation using **ordinary** backpropagation.

The graph in Figure 6 shows a computation of $\frac{\partial E(n_{\mathbf{W}}(\mathbf{x}))}{\partial \mathbf{W}}$ without using `E.backward`. Although the graph is less obvious, chasing the graph shows that it computes the following quantity:

$$(2(\mathbf{x}\mathbf{W}^T\mathbf{W}^T - \mathbf{x}\mathbf{W}^T)\cdot\mathbf{x})^T\mathbf{x}\mathbf{W}^T + ((2(\mathbf{x}\mathbf{W}^T\mathbf{W}^T - \mathbf{x}\mathbf{W}^T)\cdot\mathbf{x})\mathbf{W} - 2(\mathbf{x}\mathbf{W}^T\mathbf{W}^T - \mathbf{x}\mathbf{W}^T)\cdot\mathbf{x})^T\mathbf{x}$$

Which is equivalent to the ordinary analytical solution to $\frac{\partial E(n_{\mathbf{W}}(\mathbf{x}))}{\partial \mathbf{W}}$.

Thus, we can conclude that the training scheme outlined indeed correctly implements modified and ordinary backpropagation, as required.

3 Modified backpropagation in practice

We compare modified backpropagation with ordinary backpropagation in a variety of settings. We consider the following neural networks:

Identifier	Architecture
N1	Linear(5, 5, bias=False)
N2	Linear(5, 10, bias=False) Linear(10, 5, bias=False)
N3	Linear(5, 10, bias=True) Tanh() Linear(10, 5, bias=True) Tanh()
N4	Linear(784, 1024, bias=True) Tanh() Linear(1024, 2048, bias=True) Tanh() Linear(2048, 784, bias=True) tanh()

Identifier	Architecture
B1	Linear(5, 5, bias=True)
B2	Linear(100, 250, bias=True) Linear(250, 250, bias=True) Linear(250, 250, bias=True) Linear(250, 250, bias=True) Linear(250, 100, bias=True)
B3	Linear(4096, 1024, bias=True) Linear(1024, 4096, bias=True)
B4	Linear(784, 1024, bias=True) Linear(1024, 2048, bias=True) Linear(2048, 784, bias=True)

Batching of data has been used when training the network. Specifically, each network is trained on n total number of samples grouped into batches of size m. All samples are drawn from a normal distribution with mean 0 and variance 1. Hence, at each training iteration, the matrix $\mathbf{x} \in \mathbb{R}^{m \times n}$. A training "epoch" is then $\lfloor \frac{n}{m} \rfloor$ iterations of updates to the network. In the case where $m = n$ there is exactly one iteration per epoch, where $\mathbf{x}$ is the entire dataset. Usually, when training neural networks, the dataset remains fixed between epochs. **In this setup, however, we draw an entirely new dataset at each epoch in order to eliminate possible overfitting. As we show below, this in turn requires that larger networks need impractically long to train. We should run additional experiments with a fixed dataset between epochs.**

For each network we present the idempotent error, the norm (of $n_{\mathbf{W}}(\mathbf{I})$), and the trace (of $n_{\mathbf{W}}(\mathbf{I})$). We train 5 randomly initialized networks with both methods. The dataset contains $n = 10000$ samples. We conduct the experiment twice; once with full-batch ($m = n = 10000$, one iteration per epoch) and small-batch ($m = 2000$ and $n = 10000$).

3.1 Commentary on results

Some immediate comments about the results

- For each network configuration, modified backpropagation finds as good or better solutions (in terms of idempotent error) than ordinary backpropagation.
- For each network configuration, modified backpropagation finds solutions in fewer epochs than ordinary backpropagation.

We also note these general observations:

- We need to devise better experiments which compare the approaches more fairly.
  - There are many factors; learning rate, batch configuration, network initialization, network architecture, etc. There is a lot of variance between each run, which makes "average and standard deviation" a less good descriptor of overall performance. This is not trivial.
- We need to create experiments on real datasets which **do not change between epochs**.
  - As complexity in network architecture increases (N3, for instance), training on a newly sampled random dataset at each epoch might not be a good idea. It takes forever to train (even for ordinary backpropagation with high learning rate) since the input/output space is so huge. It would be more meaningful to train on the same dataset at each epoch, which is similar to real-life neural network training. The potential problem with a constant dataset is that the network might not learn a meaningful latent space (i.e., it does not learn "what it means to be an idempotent mapping"). But we have no reason to believe this will happen, and we can perform latent space analysis to verify that the network is not just becoming an elaborate lookup table.