

# Appendices

## A Model details

We now discuss the logical implementation of both forward and backward passes by focusing on the local interactions that each node has. For instance, instead of describing procedures in terms of a weight matrix  $W$ , we will show what happens to each individual synapse  $w_{ij}$  locally. The actual implementation computes updates in bulks; Figure 1 shows a higher level example of dataflow.

In the **forward pass**, we construct nodes for each operation happening in affine transforms, activations, and loss functions. Each weight and bias is stored in the state of the respective node. We define a forward arrow to compute one-dimensional outputs from one-dimensional inputs, just like what occurs in a traditional forward pass of neural networks; for instance, a forward pass of a weight is  $y_{ij} = w_{ij} \cdot x_i$ , and stores  $x_i$  for future use in the backward pass.  $y_{ij}$  is further aggregated to  $y_j = \sum_k y_{kj}$ . The result of a matrix multiplication can therefore be constructed by these local operations to obtain the more familiar  $y = Wx$ . Likewise, a sigmoid layer  $y = \sigma(x)$  can be deconstructed locally for each scalar  $x_i \in x$  and  $y_i \in y$  as follows:  $y_i = \sigma(x_i)$ , storing  $x_i$  for further use in the backward pass.

In the **backward pass**, every node computes a message to send back, given its stored forward input, the message being passed from the successive layer, and any internal states. We refer to this function as the *message passing rule*, or  $g$ . For a given node indexed  $i$ , we compute the message  $m_i$  from message  $m_j$ , forward input  $x_i$  and internal state  $h_i$ , which consists of any parameters specific to the node's forward pass as well as any recurrent hidden state.

$$m_i = g(m_j, x_i, h_i) \quad (5)$$

Operations such as the loss function and activation functions have no parameters, so the internal states of these nodes only store the pre-activation inputs<sup>3</sup> and a hidden recurrent state.

Seeding the backward pass requires passing an initial message to the loss node. We typically pass multidimensional messages between nodes, however we treat the message being externally passed into the loss node as a special case of one-dimensional message:

$$m_{loss_i} = [loss_i] \quad (6)$$

The output message of a loss node being passed backwards is multidimensional.

Nodes representing parameterized operations, such as an affine transform, further undergo a *weight update rule* defined by  $f$  during the backwards pass.

$$\Delta w_{ij} = f(m_j, x_i, h_i) \quad (7)$$

For an input with a batch of size  $B$ , at step  $t$ , the update is the average of the update computed over every element in the batch.

$$w_{ij}^{t+1} = w_{ij}^t + \frac{\sum^B \Delta w_{ij}^t}{|B|} \quad (8)$$

Given a traditional dense layer with a weight matrix of cardinality  $N \cdot M$ , and forward and backward arrows for each individual weight and bias, it is evident some of the messages have to be replicated and others aggregated. Each of the  $M$  neurons will have their own bias, which will send the same message to each of its  $N$  inputs, as they are connected to  $N$  incoming nodes and associated weights. This is not unlike how the backwards flowing gradient is sent along all paths. Similarly, the output of the backward pass of a dense layer is expected to be  $N$  messages, one for each input. We accomplish that by averaging the messages being passed backwards to each input.

$$m_i = \frac{\sum^K m_{ik}}{K} \quad (9)$$

Other strategies may be worth exploring, such as summing backward flowing messages, or even more complex and stateful aggregation such as feeding messages into an RNN.

**Stateful and stateless learners.** We parameterize the backwards arrows,  $f$  and  $g$ , using deep neural networks. We experiment with both stateful and stateless implementations of these arrows. We consider the backward

---

<sup>3</sup>Some nodes take more inputs. This is purely done to reduce the total number of nodes necessary for complex operations. For instance, Softmax nodes store every intermediate operation result and the common denominator across all inputs.

network to be stateless in the case when there is not recurrent hidden state between messages or updates being computed - the backwards pass has no memory of previous iterations. To implement the backwards pass with a memory, we incorporate a hidden state at each node and implement  $f$  and  $g$  as using a Gated Recurrent Unit (GRU). We observed that a deep network is beneficial for the computation of the next state update:

$$u = \sigma(x * W_{ux} + c^t * W_{uc} + b_u) \quad (10)$$

$$r = \sigma(x * W_{rx} + c^t * W_{rc} + b_r) \quad (11)$$

$$c^{t+1} = u \cdot c^t + (1 - u) \cdot \tanh(MLP_x(x) + MLP_c(c^t \cdot r) + b_n) \quad (12)$$

The MLP have two hidden layers of size 80 and 40 respectively with ReLu activations. We consider some of the carry states as output messages, and weight update for weights and biases. In Experiment 4.1.2, we used a stateless version of the above, where we use a MLP for  $f$ , and a MLP for  $g$ , both of them with two hidden layers of size 80 and 40 respectively with ReLu activations, and a tanh activation for the final layer. We did not explore the possible space of architectures further.

**Normalization.** Meta-learning is notoriously hard to train. We found initializations to be critical for successful training. In particular, for any  $f$  and  $g$  inputs, it is evident that input messages, carry states, inputs to the forward pass, and optional weights have all different means and magnitudes. We mitigated this problem by standardizing these inputs individually over initial minibatches. The mean and standard deviation from these initial minibatches are recorded, and reused thorough the network’s lifetime to standardize subsequent batches. Outputs need to be translated and scaled as well: the outputs of  $f$  are inherently bounded in  $(-1, 1)$  by the tanh activation function, while the typical magnitude and standard deviation of weights and biases is usually orders of magnitude smaller. This empirically results in too large weight and bias updates during early training, rendering meta-training particularly unstable. Moreover, having a mean output significantly different from zero causes suboptimal training. To mitigate these problems, before starting training, for each dense layer, we store the output mean of  $f_w$  and  $f_b$  (respectively for the weights and biases) and use them to have initial mean zero outputs. Furthermore, we scale outputs down to be at most 0.2 times the standard deviation of the weight matrix<sup>4</sup>. We keep all these normalization variables fixed during meta-training, except for the output scaling, which we meta-train as we observed it to be beneficial for fast adaptation. We believe other standardization techniques such as batch normalization may prove fruitful, but we do not explore them in this work.

**Parameter sharing.** Across all our experiments, all weight nodes for a given weight matrix, and all bias nodes for a given bias vector, share  $f$  and  $g$ <sup>5</sup>. Likewise, all nodes for a given activation function, and all nodes for a given loss, share  $g$ . In addition, we experimented with two more configurations: sharing  $f$  and  $g$  across all layers of the same kind (so we have one  $f$  and one  $g$  for all weight nodes in the network, and another set of  $f$  and  $g$  for all bias nodes), and sharing  $f$ ,  $g$  and the standardization parameters described above.

## B Sinusoidal Ablation study

We performed ablations on message size and amount of shared parameters (see Appendix A for the meaning of the possible configurations). Unfortunately, we observed a great amount of variance for each experiment, and it was too computationally prohibitive to perform several repetitions for each instance. What therefore follows is anecdotal evidence. Ablating the message size (we tried 1, 4, and 8), we observed that training converges faster and the final result is better with a larger message size. Ablating the amount of shared parameters, we observed the models to converge faster with no shared parameters, but the end results are all comparable even if we share all parameters. A note on reliable results: the least powerful the models, the more likely they are to get stuck in some local minima, and we observed every model to get stuck, in some runs; the most common problem we observed was it getting stuck to classifying a center arc only, while regressing to classifying a constant value outside of the center.

<sup>4</sup>This means that weights and biases of the same dense layer are initialized to have  $f$ s output similar magnitudes. This is of particular importance, since we zero-initialize biases, and therefore we could not compute their initial standard deviations.

<sup>5</sup>Weights and biases **do not** share  $f$  and  $g$  among themselves. The same reasoning applies with loss functions and activations: a cross-entropy loss node **does not** share  $g$  with a ReLU node.