A Reading in EBMM

A.1 Projected gradient descent

We described the basic reading procedure in section 2, however, there is a number of extensions we found useful in practice.

Since in all experiments we work with data constrained to the $[0, 1]$ interval, one has to ensure that the read data also satisfies this constraint. One strategy that is often used in literature is to model the output as an argument to sigmoid function (logits). This may not work well for values close to the interval boundaries due to vanishing gradient, so instead we adopted a projected gradient descent, i.e.

$$\mathbf{x}^{(k+1)} = \text{proj}(\mathbf{x}^{(k)} - \gamma^{(k)} \nabla_{\mathbf{x}} E(\mathbf{x}^{(k)})),$$

where the proj function clips data to the $[0, 1]$ interval.

Quite interestingly, this formulation allows more flexible behavior of the energy function. If a stored pattern $\mathbf{x}$ has one of the dimensions exactly on the feasible interval boundary, e.g. $x_j = 0$, then $\nabla_{x_j} E(\mathbf{x})$ does not necessarily have to be zero, since x_j will not be able to go beyond zero. We provide more information on the properties of storied patterns in further appendices.

A.2 Nesterov momentum

Another extension we found useful is to employ Nesterov momentum [22] into the optimization scheme and we use it in all our experiments.

$$\hat{\mathbf{x}}^{(k)} = \text{project}(\mathbf{x}^{(k)} + \psi^{(k)} v^{(k-1)}), \quad v^{(k)} = \psi^{(k)} v^{(k-1)} - \gamma^{(k)} \nabla E(\hat{\mathbf{x}}^{(k)})$$
$$\mathbf{x}^{(k)} = \text{project}(v^{(k)}).$$

A.3 Step sizes

To encourage learning converging attractor dynamics we constrained step sizes $\boldsymbol{\gamma}$ to be a non-increasing sequence:

$$\gamma^{(k)} = \gamma^{(k-1)} \sigma(\eta^{(k)}), \quad k > 1$$

Then the actual parameters to meta-learn is the initial step size $\gamma^{(1)}$ and the logits $\boldsymbol{\eta}$. We apply a similar parametrization to the momentum learning rates $\boldsymbol{\psi}$.

A.4 Step-wise reconstruction loss

As it is often found helpful in literature [5, 3] we apply the reconstruction loss (2) not just to the final iterate of the gradient descent, but to all iterates simultaneously:

$$\mathcal{L}_K(X, \boldsymbol{\theta}) = \sum_{k=1}^{K} \frac{1}{N} \sum_{i=1}^{N} \mathbb{E} \left[||\mathbf{x}_i - \mathbf{x}_i^{(k)}||_2^2 \right].$$

B Architecture details

Below we provide pseudocode for computational graphs of models used in the experiments. All modules containing memory parameters are specifically named as `memory`.

B.1 Gated RNN

We used a fairly standard recurrent architecture only equipped with an update gate as in [6]. We unroll the RNN for 5 steps and compute the energy value from the last hidden state.

```
hidden_size = 1024
input_size = 128
```

```
# 128 * (128 - 1) / 2 + 128 parameters in total
dynamic_size = (input_size - 1) // 2

state = repeat_batch(zeros(hidden_size))
memory = Linear(input_size, dynamic_size)

gate = Sequential([
    Linear(input_size + hidden_size, hidden_size),
    sigmoid
])

static = Linear(input_size + hidden_size, hidden_size - dynamic_size)

for hop in xrange(5):
    z = concat(x, state)

    dynamic_part = memory(x)
    static_part = static(z)
    c = tanh(concat(dynamic_part, static_part))
    u = gate(z)
    state = u * c + (1 - u) * state

energy = Linear(1)(state)
```

B.2 ResNet, fully-connected memory

```
channels = 32
hidden_size = 512
representation_size = 512
static_size = representation_size - dynamic_size

state = repeat_batch(zeros(hidden_size))

encoder = Sequential([
    ResBlock(channels * 1, kernel=[3, 3], stride=2, downscale=False),
    ResBlock(channels * 2, kernel=[3, 3], stride=2, downscale=False),
    ResBlock(channels * 3, kernel=[3, 3], stride=2, downscale=False),
    flatten,
    Linear(256),
    LayerNorm()
])

gate = Sequential([
    Linear(hidden_size),
    sigmoid
])

hidden = Sequential([
    Linear(hidden_size),
    tanh
])

x = encoder(x)

memory = Linear(input_size, dynamic_size)

dynamic_part = memory(x)
static_part = Linear(static_size)(x)
x = tanh(concat(dynamic_part, static_part))
```

```
for hop in xrange(3):
    z = concat(x, state)
    c = hidden(z)
    c = LayerNorm()(c)
    u = gate(z)
    state = u * c + (1 - u) * c

h = tanh(Linear(1024)()(state))
energy = Linear(1)(h)
```

The encoder module is also shared with all baseline models together with its transposed version as a decoder.

B.3 ResNet, convolutional memory

```
channels = 32
x = ResBlock(channels * 1, kernel=[3, 3], stride=2, downscale=True)(x)
x = ResBlock(channels * 2, kernel=[3, 3], stride=2, downscale=True)(x)

def resblock_bottleneck(x, channels, bottleneck_channels, downscale=False):
  static_size = channels - dynamic_size

  z = x

  x = Conv2D(bottleneck_channels, [1, 1])(x)
  x = LayerNorm()(x)
  x = tanh(x)

  if downscale:
    memory_part = Conv2D(dynamic_size, kernel=[3, 3], stride=2, downscale=True)(x)
    static_part = Conv2D(static_size, kernel=[3, 3], stride=2, downscale=True)(x)
  else:
    memory_part = Conv2D(dynamic_size, kernel=[3, 3], stride=1, downscale=False)(x)
    static_part = Conv2D(static_size, kernel=[3, 3], stride=1, downscale=False)(x)
    x = concat([static_part, memory_part], -1)
    x = LayerNorm)(x)
    x = tanh(x)

  z = Conv2D(channels, kernel=[1, 1])(z)
  if downscale:
    z = avg_pool(z, [3, 3] + [1], stride=2)
    x += z
  return x

x = resblock_bottleneck(x, channels * 4, channels * 2, False)
x = resblock_bottleneck(x, channels * 4, channels * 2, True)

recurrent = Sequential([
  Conv2D(hidden_size, kernel=[3, 3], stride=1),
  LayerNorm(),
  tanh
])

update_gate = Sequential([
  Conv2D(hidden_size, kernel=[1, 1], stride=1),
  LayerNorm(),
  sigmoid
])
```

```
hidden_size = 128
hidden_state = repeat_batch(zeros(4, 4, hidden_size))

for hop in xrange(3):
  z = concat([x, hidden_state], -1)
  candidate = recurrent(z)
  u = update_gate(z)
  hidden_state = u * candidate + (1. - u) * hidden_state

  x = Linear(1024)(x)
  x = tanh(x)
  energy = Linear(1)
```

B.4 ResNet, ImageNet

This network is effectively a slightly larger version of the ResNet with convolutional memory described above.

```
channels = 64
dynamic_size = 8

x = ResBlock(channels * 1, kernel=[3, 3], stride=2, downscale=True)(x)
x = ResBlock(channels * 2, kernel=[3, 3], stride=2, downscale=True)(x)

x = resblock_bottleneck(x, channels * 4, channels * 2, True)
x = resblock_bottleneck(x, channels * 4, channels * 2, True)

recurrent = Sequential([
  Conv2D(hidden_size, kernel=[3, 3], stride=1),
  LayerNorm(),
  tanh
])

update_gate = Sequential([
  Conv2D(hidden_size, kernel=[1, 1], stride=1),
  LayerNorm(),
  sigmoid
])

hidden_size = 256
hidden_state = repeat_batch(zeros(4, 4, hidden_size))

for hop in xrange(3):
  z = concat([x, hidden_state], -1)
  candidate = recurrent(z)
  u = update_gate(z)
  hidden_state = u * candidate + (1. - u) * hidden_state

  x = Linear(1024)(x)
  x = tanh(x)
  energy = Linear(1)
```

B.5 The role of skip-connections in energy models

Gradient-based meta-learning and EBMM in particular rely on the expressiveness of not just the forward pass of a network, but also the backward pass that is used to compute a gradient. This may require special considerations about the network architecture.

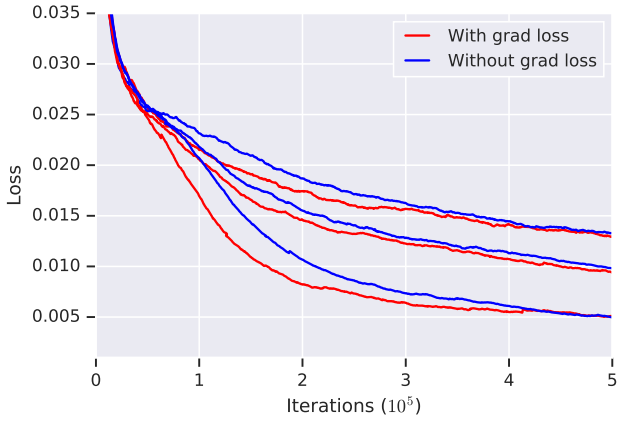

Figure 4: Effect of including the $||\nabla_{\mathbf{x}} E(\mathbf{x})||^2$ term in the writing loss (3) on Omniglot.

One may notice that all energy models considered above have an element of recurrency of some sort. While the recurrency itself is not crucial for good performance, skip-connections, of which recurrency is a special case, are.

We can illustrate this by considering an energy function of the following form:

$$E(x) = o(h(x)), \quad h(x) = f(x) + g(f(x)).$$

Here we can think of h as a representation from which the energy is computed. We allow the representation to be first computed as $f(x)$ and then to be refined by adding $g(f(x))$.

During retrieval, we use gradient of the energy with respect to x which can be computed as

$$\frac{d}{dx}E(x) = \frac{do}{dh}\frac{dh}{dx} = \frac{do}{dh}(\frac{df}{dx} + \frac{dg}{df}\frac{df}{dx}).$$

One can see, that with a skip-connection the model is able to *refine the gradient* together with the energy value.

A simple way of incorporating such skip-connections is via recurrent computation. We allow the model to use a gating mechanism that can modulate the refinement and prevent from unnecessary updates. We found that usually a small number of recurrent steps (3-5) is enough for good performance.

C Explanations on the writing loss

Our setting deviates from the standard gradient-based meta-learning as described in [8]. In particular, we are not using the same loss function (naturally defined by the energy function) in adaptation and inference phases. As we explain in section 3, writing loss (3) besides just the energy term also contains the gradient term and the prior term.

Even though we found it sufficient to use just the energy value as the writing loss, perhaps not surprisingly, minimizing the gradient norm appeared to help optimization especially in the early training (see figure 4) and lead to better final results.

We use an individual learning rate per each writable layer and each of the three loss terms, initialized at 10^{-4} and learned together with other parameters. We used softplus function to ensure that all learning rates remain non-negative.

D Experimental details

We train all models using AdamW optimizer [19] with learning rate 5×10^{-5} and weight decay 10^{-6}, all other parameters set to Adam defaults. We also apply gradient clipping by global norm at 0.05.

All models were allowed to train for 2×10^6 gradient updates or 1 week whichever ended first. All baseline models always made more updates than EBMM.

One instance of each model has been trained. Error bars showed on the figures correspond to 5- and 95-percentiles computed on a 1000 of random batches.

In all experiments we used initialization scheme proposed by He et al. [10].