3.4. Transformers are RNNs

In literature, transformer models are considered to be a fundamentally different approach to recurrent neural networks. However, from the causal masking formulation in § 3.3 and the discussion in the previous section, it becomes evident that any transformer layer with causal masking can be written as a model that, given an input, modifies an internal state and then predicts an output, namely a Recurrent Neural Network (RNN). Note that, in contrast to Universal Transformers (Dehghani et al., 2018), we consider the recurrence with respect to time and not depth.

In the following equations, we formalize the transformer layer of equation 1 as a recurrent neural network. The resulting RNN has two hidden states, namely the attention memory s and the normalizer memory z. We use subscripts to denote the timestep in the recurrence.

$$s_0 = 0, \tag{16}$$

$$z_0 = 0, \tag{17}$$

$$s_i = s_{i-1} + \phi\left(x_i W_K\right)\left(x_i W_V\right)^T, \tag{18}$$

$$z_i = z_{i-1} + \phi\left(x_i W_K\right), \tag{19}$$

$$y_i = f_l\left(\frac{\phi\left(x_i W_Q\right)^T s_i}{\phi\left(x_i W_Q\right)^T z_i} + x_i\right). \tag{20}$$

In the above equations, x_i denotes the i-th input and y_i the i-th output for a specific transformer layer. Note that our formulation does not impose any constraint on the feature function and it can be used for representing *any transformer* model, in theory even the ones using softmax attention. This formulation is a first step towards better understanding the relationship between transformers and popular recurrent networks (Hochreiter & Schmidhuber, 1997) and the processes used for storing and retrieving information.

Algorithm 1 Linear transformers with causal masking

function `forward`$(\phi(Q), \phi(K), V)$**:**
  $V' \leftarrow 0, S \leftarrow 0$
  for $i = 1, \ldots, N$ **do**
    $S \leftarrow S + \phi(K_i) V_i^T$ equation 10
    $\bar{V}_i \leftarrow \phi(Q_i) S$
  end
  return $\bar{V}$
end

function `backward`$(\phi(Q), \phi(K), V, G)$**:**
  `/* G is the gradient of the loss with respect to the output of forward */`
  $S \leftarrow 0, \nabla_{\phi(Q)}\mathcal{L} \leftarrow 0$
  for $i = 1, \ldots, N$ **do**
    $S \leftarrow S + \phi(K_i) V_i^T$
    $\nabla_{\phi(Q_i)}\mathcal{L} \leftarrow G_i S^T$ equation 13
  end
  $S \leftarrow 0, \nabla_{\phi(K)}\mathcal{L} \leftarrow 0, \nabla_V \mathcal{L} \leftarrow 0$
  for $i = N, \ldots, 1$ **do**
    $S \leftarrow S + \phi(Q_i) G_i^T$
    $\nabla_{V_i}\mathcal{L} \leftarrow S^T \phi(K_i)$ equation 15
    $\nabla_{\phi(K_i)}\mathcal{L} \leftarrow S V_i$ equation 14
  end
  return $\nabla_{\phi(Q)}\mathcal{L}, \nabla_{\phi(K)}\mathcal{L}, \nabla_V \mathcal{L}$
end

4. Experiments

In this section, we analyze experimentally the performance of the proposed *linear transformer*. Initially, in § 4.1, we evaluate the linearized attention in terms of computational cost, memory consumption and convergence on synthetic data. To further showcase the effectiveness of *linear transformers*, we evaluate our model on two real-world applications, image generation in § 4.2 and automatic speech recognition in § 4.3. We show that our model achieves competitive performance with respect to the state-of-the-art transformer architectures, while requiring significantly less GPU memory and computation.

Throughout our experiments, we compare our model with two baselines, the full transformer with softmax attention and the Reformer (Kitaev et al., 2020), the latter being a state-of-the-art accelerated transformer architecture. For the Reformer, we use a PyTorch reimplementation of the published code and for the full transformer we use the default PyTorch implementation. Note that for Reformer, we do not use the reversible layers, however, this does not affect the results as we only measure the memory consumption with respect to the self attention layer. In all experiments, we use **softmax** (Vaswani et al., 2017) to refer to the standard transformer architecture, **linear** for our proposed *linear transformers* and **lsh-X** for Reformer (Kitaev et al., 2020), where *X* denotes the hashing rounds.

For training the *linear transformers*, we use the feature map of equation 7. Our PyTorch (Paszke et al., 2019) code with documentation and examples can be found at `https://linear-transformers.com/`. The constant memory gradient computation of equations 13-15 is implemented in approximately 200 lines of CUDA code.

4.1. Synthetic Tasks

4.1.1. Convergence Analysis

To examine the convergence properties of *linear transformers* we train on an artifical copy task with causal masking. Namely, the transformers have to copy a series of symbols similar to the sequence duplication task of Kitaev et al. (2020). We use a sequence of maximum length 128 with 10