A. Appendix

A.1. Experimental details

CIFAR-100 Given the low resolution of CIFAR-100 images, we do not downsample feature maps before the attention operation and instead resort to a smaller batch size. We train all networks for 500 epochs using synchronous SGD with momentum 0.9 distributed across 8 TESLA V100 GPUs. The learning rate is linearly scaled from 0 to $0.2B/256$, where B is the total batch size, for the first 5% training epochs and then annealed with cosine decay [30]. We use standard CIFAR preprocessing: mean normalizing, random flipping and cropping [55, 10, 48]. For the non-augmented architectures, we use a batch size of 1024 and a weight decay of 2e-4. When using Attention Augmentation, the batch size is set to 256 and the weight decay is set to 5e-4.

ImageNet classification with ResNet We train all ResNet architectures for 100 epochs using synchronous SGD with momentum 0.9 across 8 TESLA V100 GPUs and weight decay of 1e-4. We use the largest batch size per worker $B \in \{32, 64, 128, 256\}$ that fits in a minibatch. The initial learning rate is scaled linearly according to the total batch size using a base learning rate of 0.128 for total batch size of 256. During training, we linearly scale the learning rate from 0 to this value for the first 5% of training epochs and divide it by 10 at epochs 30, 60, 80 and 90. We use standard Inception data augmentation as described in [41].

ImageNet classification with MnasNet We follow the training setup described in [42] and train all networks for 350 epochs with the RMSProp optimizer using exponential learning rate decay. When training our augmented MnasNets, we divide the learning rate by 2 and adjusted the learning rate decay so that the final learning rate stays the same.

Object Detection with COCO dataset We follow the setup described in [26, 11] and train the RetinaNet from scratch for 150 epochs without using ImageNet pretraining for the ResNet backbone. We use the preprocessing pipeline described in [26]. We apply multiscale jitter, randomly resize images from [512, 768] and crop to a max dimension of 640 during training. All images are horizontally flipped with a 50% probability.

A.2. Computational & Memory costs

Table 9 provides the breakdown of self-attention related computational costs per image. All parameter counts and FLOPS are obtained with the TensorFlow Profiler. These consider all parameters/computations, including the ones needed to compute the attention maps, thus allowing for a fair comparison. Storing attention maps in each layer induces a memory cost of $N_h(HW)^2$ `bfloat16`. At inference, the memory cost for storing attention maps is only 1.2% of the memory required to store model parameters (49MB).

Layer	Memory	Params	FLOPS
{Stage 2 - H=W=14} * 4	600KB	43k	22M
{Stage 3 - H=W=14} * 6	600KB	90k	40M
{Stage 4 - H=W=7} * 3	37.5KB	190k	19M
Training	6MB (total)	1.3M	390M
Inference	600KB (max)	1.3M	390M

Table 9. Computational costs associated with self-attention in the forward pass of the ResNet50. During inference, we only consider the largest memory cost since activations are not stored.

Figures 5 and 6 show the accuracies of our attention augmented networks across FLOPS counts, which correlate with running times across hardware platforms.

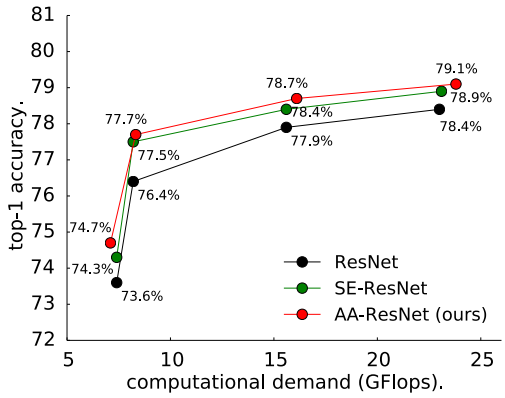

Figure 5. ImageNet top-1 accuracy as a function of computational demand for variety of ResNet architectures [14]. From left to right: ResNet-34, ResNet-50, ResNet-101 and ResNet-152.

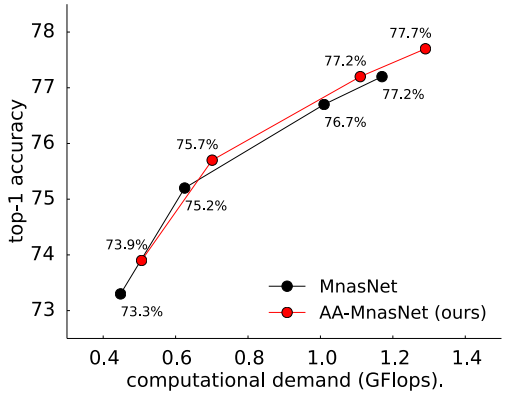

Figure 6. ImageNet top-1 accuracy as a function of computational demand for MnasNet (black) and Attention-Augmented-MnasNet (red) with depth multipliers 0.75, 1.0, 1.25 and 1.4.

A.3. 2D Relative Self-Attention implementation

While our method is simple and only requires matrix multiplication, addition and the softmax operation (Equations 3 and 4), our implementation relies on non-trivial operations (e.g. tiling, transposing and reshaping) because no low-level kernels currently exist for hardware platforms. Future work may develop specialized kernels as previously done for convolutions. Therefore, we believe that current latency times (Table 2) reflect the lack of dedicated engineering as opposed to inefficiency in the proposed method.

```
def shape_list(x):
  """Return list of dims, statically where possible."""
  static = x.get_shape().as_list()
  shape = tf.shape(x)
  ret = []
  for i, static_dim in enumerate(static):
    dim = static_dim or shape[i]
    ret.append(dim)
  return ret

def split_heads_2d(inputs, Nh):
  """Split channels into multiple heads."""
  B, H, W, d = shape_list(inputs)
  ret_shape = [B, H, W, Nh, d // Nh]
  split = tf.reshape(inputs, ret_shape)
  return tf.transpose(split, [0, 3, 1, 2, 4])

def combine_heads_2d(inputs):
  """Combine heads (inverse of split_heads_2d)."""
  transposed = tf.transpose(inputs, [0, 2, 3, 1, 4])
  Nh, channels = shape_list(transposed)[-2:]
  ret_shape = shape_list(transposed)[:-2] + [Nh * channels]
  return tf.reshape(transposed, ret_shape)

def rel_to_abs(x):
  """Converts tensor from relative to aboslute indexing."""
  # [B, Nh, L, 2L-1]
  B, Nh, L, _ = shape_list(x)
  # Pad to shift from relative to absolute indexing.
  col_pad = tf.zeros((B, Nh, L, 1))
  x = tf.concat([x, col_pad], axis=3)
  flat_x = tf.reshape(x, [B, Nh, L * 2 * L])
  flat_pad = tf.zeros((B, Nh, L-1))
  flat_x_padded = tf.concat([flat_x, flat_pad], axis=2)
  # Reshape and slice out the padded elements.
  final_x = tf.reshape(flat_x_padded, [B, Nh, L+1, 2*L-1])
  final_x = final_x[:, :, :L, L-1:]
  return final_x

def relative_logits_1d(q, rel_k, H, W, Nh, transpose_mask):
  """Compute relative logits along one dimenion."""
  rel_logits = tf.einsum('bhxyd,md->bhxym', q, rel_k)
  # Collapse height and heads
  rel_logits = tf.reshape(
      rel_logits, [-1, Nh * H, W, 2 * W-1])
  rel_logits = rel_to_abs(rel_logits)
  # Shape it and tile height times
  rel_logits = tf.reshape(rel_logits, [-1, Nh, H, W, W])
  rel_logits = tf.expand_dims(rel_logits, axis=3)
  rel_logits = tf.tile(rel_logits, [1, 1, 1, H, 1, 1])
  # Reshape for adding to the logits.
  rel_logits = tf.transpose(rel_logits, transpose_mask)
  rel_logits = tf.reshape(rel_logits, [-1, Nh, H*W, H*W])
  return rel_logits
```

Figure 7. Helper functions in Tensorflow for 2D relative self-attention.

```
def relative_logits(q, H, W, Nh, dkh):
  """Compute relative logits."""
  # Relative logits in width dimension first.
  rel_embeddings_w = tf.get_variable(
      'r_width', shape=(2*W - 1, dkh),
      initializer=tf.random_normal_initializer(dkh**-0.5))
  # [B, Nh, Hw, HW]
  rel_logits_w = relative_logits_1d(
      q, rel_embeddings_w, H, W, Nh, [0, 1, 2, 4, 3, 5])

  # Relative logits in height dimension next.
  # For ease, we 1) transpose height and width,
  # 2) repeat the above steps and
  # 3) transpose to eventually put the logits
  # in their right positions.
  rel_embeddings_h = tf.get_variable(
      'r_height', shape=(2 * H - 1, dkh),
      initializer=tf.random_normal_initializer(dkh**-0.5))
  # [B, Nh, Hw, HW]
  rel_logits_h = relative_logits_1d(
      tf.transpose(q, [0, 1, 3, 2, 4]),
      rel_embeddings_h, W, H, Nh, [0, 1, 4, 2, 5, 3])

  return rel_logits_h, rel_logits_w

def self_attention_2d(inputs, dk, dv, Nh, relative=True):
  """2d relative self-attention."""
  _, H, W, _ = shape_list(inputs)
  dkh = dk // Nh
  dvh = dv // Nh
  flatten_hw = lambda x, d: tf.reshape(x, [-1, Nh, H*W, d])

  # Compute q, k, v
  kqv = tf.layers.conv2d(inputs, 2 * dk + dv, 1)
  k, q, v = tf.split(kqv, [dk, dk, dv], axis=3)
  q *= dkh ** -0.5 # scaled dot-product

  # After splitting, shape is [B, Nh, H, W, dkh or dvh]
  q = split_heads_2d(q, Nh)
  k = split_heads_2d(k, Nh)
  v = split_heads_2d(v, Nh)

  # [B, Nh, HW, HW]
  logits = tf.matmul(flatten_hw(q, dkh), flatten_hw(k, dkh),
        transpose_b=True)

  if relative:
    rel_logits_h, rel_logits_w = relative_logits(q, H, W, Nh,
          dkh)
    logits += rel_logits_h
    logits += rel_logits_w

  weights = tf.nn.softmax(logits)
  attn_out = tf.matmul(weights, flatten_hw(v, dvh))
  attn_out = tf.reshape(attn_out, [-1, Nh, H, W, dvh])
  attn_out = combine_heads_2d(attn_out)
  # Project heads
  attn_out = tf.layers.conv2d(attn_out, dv, 1)
  return attn_out

def augmented_conv2d(X, Fout, k, dk, dv, Nh, relative):
  conv_out = tf.layers.conv2d(inputs=X, filters=Fout - dv,
       kernel_size=k, padding='same')
  attn_out = self_attention_2d(X, dk, dv, Nh, relative=
       relative)
  return tf.concat([conv_out, attn_out], axis=3)
```

Figure 8. Tensorflow code for 2D relative self-attention.

A.4. Attention visualizations.

In Figure 10, we present attention maps visualizations for the input image shown in Figure 9. We see that attention heads learn to specialize to different content and notably can delineate object boundaries.

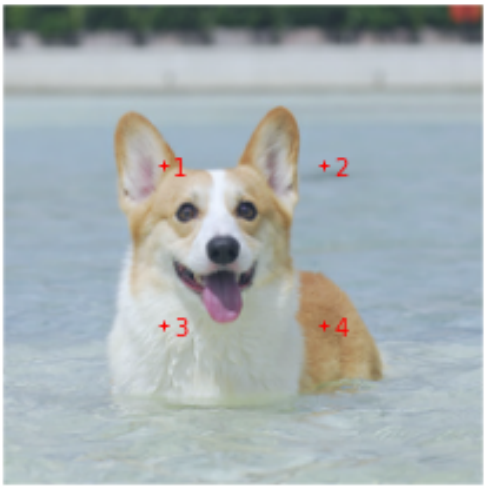

Figure 9. An input image. The red crosses indexed 1 to 4 represent the pixel locations for which we show the attention maps in Figure 10.

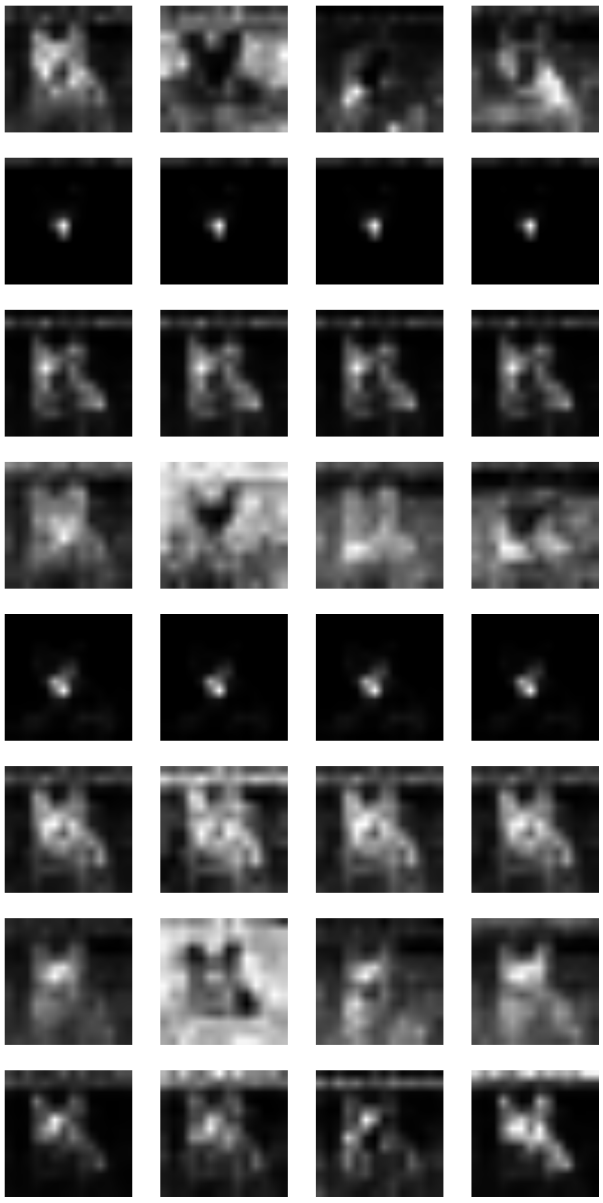

Figure 10. Visualization of attention maps for an augmented convolution in the Attention-Augmented-ResNet50. Rows correspond to the 8 different heads and columns correspond to the 4 pixel locations depicted in the input image (See Figure 9).