Data 188 Introduction to Deep Learning
Spring 2026 Eric Kim

Discussion 10

This week we will cover Visual Transformers.

1. Decoder "Shift Right"

In this question, we will learn why it is important for encoder-decoder models to use *right-shifted* decoder inputs, i.e. why we need to prepend a `<BOS>` token to the decoder input sequence like below:

```
X_src = ["I", "like", "to", "sing"]
X_tgt = [<BOS>, "J'aime", "chanter"]
Y_tgt = ["J'aime", "chanter", <EOS>]
```

where `X_src` is the encoder input, `X_tgt` is the decoder input, `Y_tgt` are the prediction targets, and `<BOS>` and `<EOS>` are the "beginning of sequence" and "end-of-sequence" tokens, respectively.

For simplicity, assume that the above `X_src`, `X_tgt`, `Y_tgt` are already tokenized, i.e. the above are string representations of integer token IDs, something like: `{"I": 829, "like": 9001, "to": 642, "sing": 543, "J'aime": 10282, "chanter": 12343, <BOS>: 1, <EOS>: 2, <PAD>: 0}`. In your answers for this question, you can just refer to tokens as their string representation.

Recall that in order to perform text generation, the decoder is trained to autoregressively predict the next token in the target sequence. In other words, the decoder repeatedly predicts one token at a time, taking the output of the previous task as the input for the next task.

(a) Fill in the blanks in the table below **without** using right-shifted decoder inputs (i.e. if we set `X_tgt = Y_tgt = ["J'aime", "chanter", <EOS>]`).

Task	Decoder Input Tokens	Target Prediction Token
1		
2		
3		

Solution:

Task	Decoder Input Tokens	Target Prediction Token
1	`["J'aime"]`	`"J'aime"`
2	`["J'aime", "chanter"]`	`"chanter"`
3	`["J'aime", "chanter", <EOS>]`	`<EOS>`

(b) Fill in the blanks in the table below **with** right-shifted decoder inputs.

Task	Decoder Input Tokens	Target Prediction Token
1		
2		
3		

Solution:

Task	Decoder Input Tokens	Target Prediction Token
1	`[<BOS>]`	`"J'aime"`
2	`[<BOS>, "J'aime"]`	`"chanter"`
3	`[<BOS>, "J'aime", "chanter"]`	`<EOS>`

(c) Observe the differences between the two tables above. Why is it important that we prepend `<BOS>` to `X_tgt`, and not to `Y_tgt`?

Solution: It's important to prepend `<BOS>` to `X_tgt` so that the prediction tasks are aligned correctly. If we didn't do this `<BOS>` prepend to `X_tgt`, then the tasks are out-of-sync in a nefarious way: the decoder gets to see the target ground truth token! This would trivialize the prediction task: it can learn to always output the final decoder input token, without actually learning anything.

2. Cross-Token Interactions

(a) Suppose I removed the multi-head self attention (MHA) blocks from a transformer encoder, and replaced it with a no-op (say, an Identity layer). What remains are blocks like: position encoding, Linear layers, LayerNorms, residual connections, and MLP/FFNs. Will this "hollowed out" transformer encoder be able to learn cross-token interactions? Why or why not?

Solution: No: MHA is the only component that calculates cross-token interaction terms. Every other layer (ex: Linear, FFN, etc) operate on each token embedding independently ("point wise").

(b) Aside from MHA, what is another way of calculating cross-token interactions? **Hint:** Use an MLP, but on a modified version of the input token embeddings `X_src` (with original shape `[n_src, d]`).

Solution: If you flatten the `X_src` into shape `[n_src * d]`, an MLP can learn cross-token features. This is because an MLP can learn cross-feature interactions, but only at each position/sample at a time. In transformer models, since each Linear layer is performed on each token position independently, the Linear layers won't learn cross-token interactions, but flattening all tokens into a single "position" will allow learning cross-token interactions.

However, this is an inefficient way of learning cross-token (and even cross-feature) interactions: hence, why MHA is an effective technique for learning cross-token interactions.

(optional) If you're curious, Deep and Cross Networks (DCN) are a popular way of learning cross-feature interactions in a more efficient manner than a standard MLP.

3. Attention Cost

Recall that the cost for a naive transformer implementation scales quadratically with the input sequence length. Which transformer component primarily contributes to this quadratic cost?

Solution: Multi-head self attention (MHA) is the primary reason for the quadratic cost, due to calculating the `[n_tgt, n_src]` attention score matrix.

4. Visual Transformer

(a) How do we pass an input image into a visual transformer encoder?

Solution: We patchify the image by gridding up the image into non-overlapping 16×16 (pixel) patches, then create a sequence out of the 16×16 patches in raster order (left-to-right, top-to-bottom).

(b) Suppose I wanted to pass higher-resolution images to my visual transformer model. I double the input image resolution (ex: $224 \times 224 \rightarrow 448 \times 448$), but keep the same patchify step. This results in quadrupling the input sequence length. Will I be able to run inference with the larger 448×448 images on a model trained on the smaller 224×224 images?

Solution: It depends!

Inference will work, in that it won't generate an error and will produce an output. However, we are changing the input distribution of the model by passing in higher-resolution images: generally, we want the train and test conditions to match as closely as possible. Performance may degrade due to this domain shift: but, it might also improve, as the increased image resolution may be helpful for the task.

This image resolution domain shift can be mitigated by doing image data augmentation during training. Ex: during training, randomly crop/resize images to different image resolutions, so that the model is robust to changes in input image size.

Regarding `max_seq_len`: depending on what the model's `max_seq_len` is, the model may truncate large portions of the larger input image, which can lead to task degradation, as the model is now seeing less of the image.

(c) Recall that the *inductive bias* of a model architecture is the space of possible hypothesis functions that a particular model architecture restricts itself to. The **Visual Transformer** model is said to have fewer inductive biases than ConvNets. Describe why visual transformer models can learn both spatially-global and local features, while ConvNets learn only spatially-local features.

Solution: Visual transformer models can learn both spatially-global and local features because the self-attention blocks allow mixing information from all parts of the image (via cross-token interactions). Each visual token is a different spatial part of the image.

On the other hand, in ConvNets, a particular activation from an intermediate spatial activation feature map is only a function of nearby spatial locations (ex: the output of a 3×3 kernel). Interestingly, as you get deeper into the ConvNet, each spatial feature map location "sees" more of the input image (in pixel space), due to the downsampling that happens after each block. But, these features are still spatially-local.

There are ConvNet techniques that try to learn more global features: for instance, dilated convolutions increase the filter size while keeping the number of parameters/computations fixed.

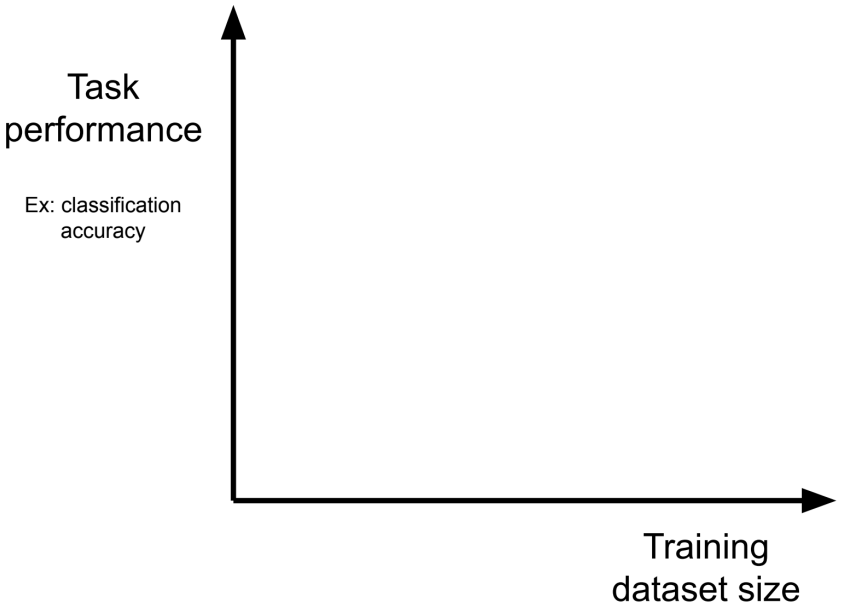
5. Large-Scale Pretraining

(a) Recall that the Visual Transformer paper pretrained on a separate larger image classification dataset (e.g. JFT-300M) before finetuning on ImageNet-1k image classification.

In this case, the pretraining task and finetuning task are the same: image classification. What is an example of an effective pretraining strategy where the pretraining task is **different** from the finetuning task?

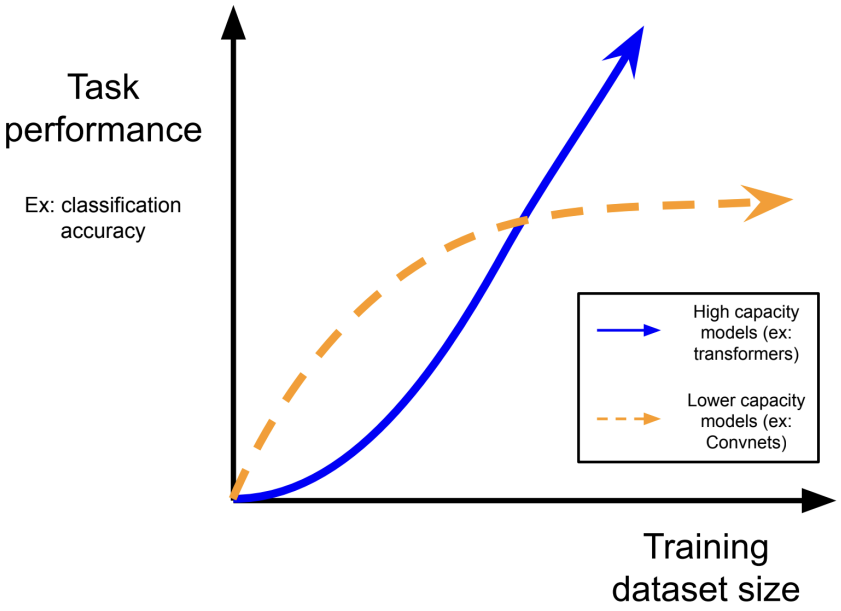
Solution: In masked autoencoder (MAE), the pretraining task is to "fill in the image patch," while the finetuning task is image classification.

(b) Consider the following chart. Note that a common (yet imperfect) proxy for model capacity is the number of model parameters.



Based on learnings from the field (and this class), draw the expected behavior of (1) high-capacity models like vision transformer models and (2) lower-capacity models like ConvNets in the chart. Your answer should have two curves, one for each type of model. Describe what's going on and why. How does inductive bias fit into this?

Solution:



At small/medium training dataset sizes, lower capacity models like ConvNets will outperform high capacity models like transformer models. However, at large training dataset sizes (ex: on the order of JFT-300M), transformer models will surpass ConvNets, while ConvNets will start seeing diminishing returns on training dataset size increases ("plateauing").

One explanation of this phenomenon has to do with inductive bias. ConvNets have a strong inductive bias towards models that learn spatially-local features, due to the properties of `Conv2d`. This is useful at first for small/medium datasets, as spatially-local features are indeed useful for many vision tasks.

However, at a certain point the restrictions of the ConvNet inductive bias start getting in the way of higher task performance. Spatially-global features are often indeed useful for many downstream vision tasks, but the ConvNet model architecture makes it awkward/difficult to learn spatially-global features.

On the other hand, transformer models have fewer inductive biases than ConvNets: in particular, transformer models can learn both spatially local and global features. This is because the self-attention blocks allow mixing information from all parts of the image (via cross-token interactions). However, the tradeoff with this modeling flexibility (i.e. fewer inductive biases) is: the model needs to see more data to learn strong representations.

Contributors:

- Eric Kim.
- Rebecca Dang.

