

## Homework 3

6.S898 Deep Learning  
Fall 2023

**Instructions:** There are a total of 32 points for this homework. Each question is marked with its corresponding points. Some questions are **not graded**, including all bonus questions. But you are encouraged to think about and attempt them.

**Submission:** Upload a **PDF** of your response through **Gradescope** by **10/24 at 1:00PM ET**. This assignment is **coding heavy** and will take some time. All of the code for this homework (Problems 2, 3, and 4) will be in the following colab notebook [here](#). For this assignment, do not include your code inline. Instead, export the notebook as a pdf (with the outputs) and attach the PDF to your submission. However, written answers should be included as usual.

**Notation:** We will use [this set of math notation specified on course website](#), whose  $\text{\LaTeX}$  source is available on Canvas. For example,  $c$  is a scalar,  $\mathbf{b}$  is a vector and  $\mathbf{W}$  is a matrix. You are encouraged (although not enforced) to follow this notation in your typeset submission, or to the best of your ability with a handwritten response (bolding may be difficult :))

### Problem 1: RNNs versus transformers (8 pt)

Recurrent neural networks, also known as RNNs, are a type of neural network used for sequence modelling. In this question, we will think conceptually about how an RNN processes information, and compare this to transformers.

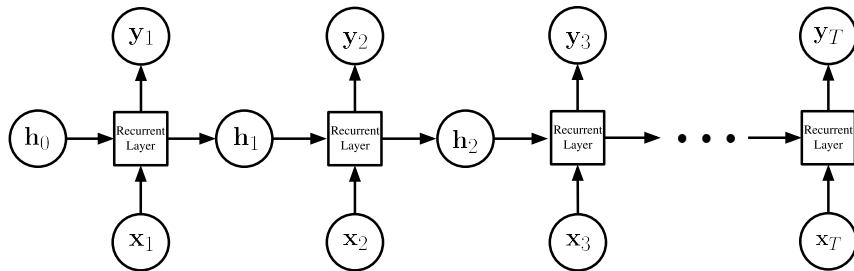


Figure 1: A simple RNN architecture. At time  $t$ , an RNN computes a hidden state  $\mathbf{h}_t$  based on the current input  $\mathbf{x}_t$  and prior hidden state  $\mathbf{h}_{t-1}$ . The RNN also spits out an output  $\mathbf{y}_t$ .

Consider the simple RNN architecture shown in Figure 1. The inputs  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$  are vectors in  $\mathbb{R}^{d_{\text{in}}}$ , the hidden states  $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T$  are vectors in  $\mathbb{R}^{d_{\text{hidden}}}$ , and the outputs  $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_T$  are vectors in  $\mathbb{R}^{d_{\text{out}}}$ . The hidden states and outputs are given by recurrence relations:

$$\mathbf{h}_t = \phi_h(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}_h); \quad (1)$$

$$\mathbf{y}_t = \phi_y(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y). \quad (2)$$

## Homework 3

The functions  $\phi_h(\cdot)$  and  $\phi_y(\cdot)$  are arbitrary element-wise non-linearities. The RNN has three weight matrices  $\mathbf{W}_h$ ,  $\mathbf{W}_x$  and  $\mathbf{W}_y$  and two bias vectors  $\mathbf{b}_h$  and  $\mathbf{b}_y$ .

For simplicity, in this question we will set the initial hidden state  $\mathbf{h}_0 = \mathbf{0}$ , we will set the non-linearity  $\phi_h$  to the identity  $\phi_h(\mathbf{h}) = \mathbf{h}$  and we will set the biases  $\mathbf{b}_h = \mathbf{0}$  and  $\mathbf{b}_y = \mathbf{0}$ . Under this simplification, after one time step  $T=1$ :  $\mathbf{h}_1 = \mathbf{W}_x \mathbf{x}_1$  and  $y_1 = \phi_y(\mathbf{W}_y \mathbf{W}_x \mathbf{x}_1)$ .

- (a) **(1pt)** Derive formulae for hidden state  $\mathbf{h}_2$  and output  $y_2$  in terms of the weight matrices  $\mathbf{W}_y$ ,  $\mathbf{W}_x$ ,  $\mathbf{W}_h$  and inputs  $\mathbf{x}_1, \mathbf{x}_2$ .
- (b) **(1pt)** Derive formulae for hidden state  $\mathbf{h}_3$  and output  $y_3$  in terms of the weight matrices  $\mathbf{W}_y$ ,  $\mathbf{W}_x$ ,  $\mathbf{W}_h$  and inputs  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ .
- (c) **(1pt)** Derive formulae for hidden state  $\mathbf{h}_T$  and output  $y_T$  in terms of the weight matrices  $\mathbf{W}_y$ ,  $\mathbf{W}_x$ ,  $\mathbf{W}_h$  and inputs  $\mathbf{x}_1, \dots, \mathbf{x}_T$ .
- (d) **(1pt)** Suppose the sequence length  $T$  is very long. What do you notice about the contribution of the first input  $\mathbf{x}_1$  to the last output  $y_T$  of the RNN?

Another way of handling sequential data is to use a self-attention layer, à la transformers. Given a sequence of inputs  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T$ . A self-attention layer computes:

$$\text{attention scores:} \quad \alpha_t = \frac{1}{\sqrt{T}} \sum_{j=1}^T (\mathbf{Q} \mathbf{x}_t)^\top (\mathbf{K} \mathbf{x}_j) \quad \text{for } t = 1, \dots, T; \quad (3)$$

$$\text{outputs:} \quad \mathbf{y}_t = \frac{\sum_{j=1}^T e^{\alpha_j} \mathbf{V} \mathbf{x}_j}{\sum_{j=1}^T e^{\alpha_j}} \quad \text{for } t = 1, \dots, T. \quad (4)$$

The matrices  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V}$  are given the names *query*, *key* and *value* matrices.

- (e) **(3pt)** For the first two questions, your answer only needs to indicate the asymptotic scaling with sequence length  $T$ . Use big- $\mathcal{O}$  notation, and ignore any other factors.
  - For RNNs, how many floating point operations are needed for a forward pass?
  - For a self-attention layer, how many floating point operations are needed for a forward pass?
  - With sufficient parallel hardware, performing a forward pass on a transformer is typically much faster than for an RNN. Why is this the case?

**Hint:** Think about different ways to arrange the computation of Equations 3 and 4.

- (f) **(1pt)** If you wanted to train and deploy a neural network that operates on very long sequences  $T \rightarrow \infty$ , would you rather use an RNN or a transformer? Why?

## Homework 3

6.S898 Deep Learning  
Fall 2023

**Hint:** There are different possible answers here, and we are just looking for some short sensible commentary that reflects on memory and time complexity mentioned in previous problems.

### Problem 2: Implementing a Transformer (10 pt)

In this problem, you'll implement a Transformer (from scratch!). We'll first focus on writing a self-attention module. In the next two problems we'll explore how to use transformers for two entirely different domains: image classification and language. Throughout this problem, let  $B$  be the batch size and  $T$  be the sequence length.

These problems will be implemented using the following colab notebook [here](#). These problems are coding heavy and will take some amount of time. For the submission, **download the entire colab notebook as a PDF document** and attach this PDF to your solution at the end. No need to copy and paste the code in-line. However, if we ask for a non-coding response, answer the question here.

- (a) **(4pt)** Our first task is to implement single-headed self-attention. Take a look at the class `AttentionHead`. For an input  $x \in \mathbb{R}^{T \times d}$ , we want to setup three linear layers  $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d_k}$ . Use these to create your queries, keys, and values.

$$\mathbf{Q} = \mathbf{x}\mathbf{W}_Q \quad \mathbf{K} = \mathbf{x}\mathbf{W}_K \quad \mathbf{V} = \mathbf{x}\mathbf{W}_v$$

Then compute your attention weights

$$\mathbf{A} = \text{softmax}(\mathbf{Q}\mathbf{K}^T / \sqrt{d_k})$$

Here  $\mathbf{A}_{ij}$  indicates the weight of token  $j$  when computing the new representation for token  $i$ . Finally use your attention weights to compute your output as a weighted combination of the values:

$$\text{Out} = \mathbf{A}\mathbf{V}$$

**Deliverable** Implement `AttentionHead`. Notice that the forward function takes an argument called `attn_mask`  $\in \{0, 1\}^{T \times T}$ . If `attn_maski,j` = 0 token  $i$  should not attend on token  $j$  (e.g., `Aij` = 0). Modify your code accordingly.

**Hint:** If you want token  $i$  not to attend on token  $j$  what value should you set in the input of the softmax?)

- (b) **(3pt)** We now implement multi-headed attention.

Take a look at the class `MultiHeadedAttention`. Our output should separately compute the attention outputs for each of the heads, then concatenate the outputs together

## Homework 3

6.S898 Deep Learning  
Fall 2023

and project them into the output dimension. Specifically for parameter matrix  $W_O$  (what size should  $W_O$  be?)

$$\text{MultiHead}(x) = \text{Concat}(\text{head}_1(x), \dots, \text{head}_n(x))W_O$$

**Deliverable** Implement `MultiHeadedAttention`.<sup>1</sup>

(c) **(3pt)** Now let's put it all together!

Take a look at the feed-forward-network module `FFN` and the residual model `AttentionResidual`. We've already implemented this bit for you (you're welcome!). The `FFN` performs normalization along with linear layers interspersed by GELUs<sup>2</sup> (similar to the MLPs you've seen in previous PSETs). `AttentionResidual` then sends the input through both multiheaded attention and the `FFN`, adding a residual after each step.

**Deliverable** Implement `Transformer`, which passes the input through successive `AttentionResidual` layers.

Make sure you can pass the given test cases (they just check output sizes). In particular, test case 3 checks whether you've dealt with `attn_mask` correctly. The provided attention mask indicates that each token should only attend to itself and the token before it. Your returned `A` should reflect this attention pattern.

### Problem 3: Vision Transformers (5 pt)

So far, the best performing model we've seen for image classification has been CNNs. Recently, transformer architectures have been shown to have similar to better performance. One such model called Vision Transformer (ViT) splits up images into regularly sized patches (Dosovitskiy et al. [2020]). The patches are treated as a sequence and attention weights are learned as in a standard transformer model.

In detail the ViT has a few steps (see Figure 2).

- First we embed the patches using into a sequences of embeddings.
- We add a positional encoding to the embedding which captures the position of each patch in the image.

---

<sup>1</sup>Note, in most industrial settings, rather than looping over the heads sequentially, all the heads are computed at the same time in a vectorized fashion. This provides a speedup by letting you compute the heads in parallel, but don't worry about this here.

<sup>2</sup>a Gaussian Error Linear Unit (GELU) functions here as a smoother version of a ReLU.

## Homework 3

6.S898 Deep Learning  
Fall 2023

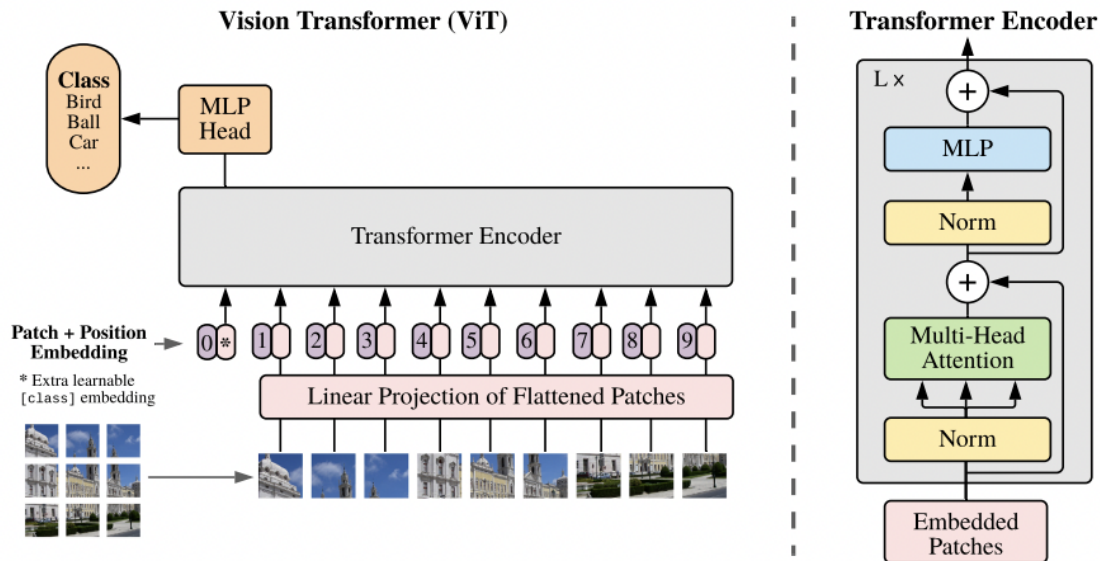


Figure 2: ViT Architecture

- We prepend an extra learned class embedding to our sequence and pass the entire sequence through a transformer.
- We extract the final representation of the class embedding and learn a linear layer (MLP Head) to predict the probability of each class.
- We supervise the class with cross entropy loss.

Now that we've implemented a transformer, we can use it to implement a ViT! Make sure you've already done the previous section.

- (a) **(2pt)** We first implement our patch embedding. Take a look at the class `PatchEmbed`. For a given image, we want to split the image into square patches. Each patch should then be flattened and linearly projected with some weight.

For example, suppose we want embeddings of size 128. If your image is size  $(3, 32, 32)$  and your patches are  $4 \times 4$ , you should end up with 64 patches. Flattened, each patch contains  $3 \times 4 \times 4 = 48$  elements. We want to learn a linear projection from those 48 elements to our output dimension 128. We'll then end up with a sequence of 64 inputs of 128 elements each to pass into our transformer!

**Deliverable** Implement `PatchEmbed`.

## Homework 3

6.S898 Deep Learning  
Fall 2023

**Hint:** Splitting up the patches manually and then using `nn.Linear` will be painful. Instead, look at `nn.Conv2d`. How can you use this to implement the patch embedding?

- (b) **(1pt)** Read through the `VisionTransformer` (implemented for you). Take a look at the *positional embedding*. Positional embeddings encode the position of each element in the sequence. In this case, the positional embeddings for every position in the sequence is *learned*. However, this creates a strict limit on how many tokens can be passed to the transformer (if you only had 64 position embeddings, the positional embedding of the 65th token is undefined!)

Suppose you wanted to implement a transformer that can take arbitrarily long inputs (ignore any memory or time constraints). Describe a way to implement the positional embedding such that there is no maximum sequence size.

- (c) **(1pt)** Train the Vision Transformer on CIFAR-10! We've implemented the training loop for you. Run the cells to train a model and **report your validation accuracy here** (it should be in the high 80s). This should take about 5 minutes.
- (d) **(1pt)** The attention maps for transformers tell us which patch relied on which other patch. Let's take a look at the attention heatmap of the class token (averaged over all heads and layers). At a high level this can give us a sense of which parts of the image the model is relying on. We provide code to visualize this heatmap for 10 validation images.

Include the images with their heatmaps here. What do you observe about the heatmaps?

### Problem 4: DialogueGPT (9 pt)

Now let's use our Transformer to train a language model! We're going to train a language model on some Shakespeare. Run the cell to download `input.txt` which contains some Shakespeare text. Each element in `all_dialogues` will be one example in our dataset.

- (a) **(2pt)** Our first step is to build a *tokenizer*, which splits line of dialogue into individual tokens and assigns each token to an ID. We're going to use NLTK's `word_tokenize`, which splits words and punctuation into their own tokens.

Take a look at `MyTokenizer`. This tokenizer has three special tokens: `start` (which will start every example), `pad` (used to pad examples to the same length), and `unk` (used when encountering a word not in our vocabulary). We've initialized the tokenizer for you.

## Homework 3

6.S898 Deep Learning  
Fall 2023

**Deliverable** Implement the following functions in MyTokenizer:

- encode: convert a string to a series of token ids and prepend the token id for the start token. Use word\_tokenize to split the string.
- decode: convert an array of token ids back into tokens. Join them with a space.

Make sure that your tokenizer fulfills the test case.

(b) Read over and make sure you understand how we create the DialogueDataset and data loader. The data loader pads the list of tokens such that they are all the same length. It outputs a dictionary with two elements:

- input\_ids contains the input ids for each element in the batch (padded to the right to the max length)
- input\_mask indicates which tokens are pad tokens (and should thus be ignored).

**Deliverable** You don't need to do anything for this question.

(c) **(2pt)** Time to implement DialogueGPT! Review the lecture on language models if you haven't already.

**Deliverable** Fill out TODOs in the \_\_init\_\_ and forward methods. The forward call should:

- Given the token ids, retrieve the corresponding token embeddings. Add to this a learned positional embedding.
- Generate a *causal attention mask*. Remember that for GPT, every token only depends on itself and the tokens before it
- Pass the embeddings and the attention mask to the transformer and the language model head. Output logits of size  $(T \times V)$  where  $T$  is the number of tokens and  $V$  is the vocabulary size. This step is implemented for you

(d) **(2pt)** Let's implement the loss. Remember that a language model is trained to predict the next token given a prefix of tokens.

Suppose our vocab size is  $V$  and we have  $T$  tokens in our training example (including the start token). Then our model will output logits  $O \in \mathbb{R}^{T \times V}$ . Our loss for this example will be  $T - 1$  individual classification losses, where using logit vector  $O[i]$  we will predict the token id for token  $i + 1$  via cross entropy loss. This means we will not supervise the start token, nor will we use the last logit vector  $O[-1]$ . An illustration of this is shown below.

## Homework 3

6.S898 Deep Learning  
Fall 2023



Figure 3: Illustration of GPT Loss

**Deliverable** Implement `DialogueLoss`. Remember to take into account the `inp_mask` to ignore supervising tokens that correspond to padding.

- (e) **(1pt)** Go back to `DialogueGPT` read the `generate` function, which we have implemented for you. The `generate` function takes in a prefix of token ids and autoregressively generates `num_tokens` more tokens, by greedily picking the most likely next token and adding it back to the input.

Generating  $T$  tokens is often much slower than training on an input with  $T$  tokens. Comment on why this is the case.

- (f) **(1pt)** Now its time to train `DialogueGPT`! Run the cells to train the model. This step will take you around 30 minutes, so budget accordingly. Note: if you're having trouble debugging, try overfitting to just a few examples (e.g., make your training dataset size 10 or so).

The training code generates some text after every epoch. What do you notice about the generations as the epochs progress?

- (g) **(1pt)** Generate 50 tokens of input text. Our language model is pretty small and hasn't been trained for very long, but you should still get something approximating english. Paste the output of your model here.

## References

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. URL <https://arxiv.org/abs/2010.11929>.