Introduction to Deep Learning - Homework 5

Prajwal Kumar
prajwalk

Question 1: Position Embeddings Exploration

(a) Permuting the Input

(i) Showing that $\boldsymbol{Z}_{\text{perm}} = \boldsymbol{P}\boldsymbol{Z}$

We start with an input sequence $X \in \mathbb{R}^{T \times d}$ and its permutation

$$X_{\text{perm}} = PX,$$

where $P \in \mathbb{R}^{T \times T}$ is a permutation matrix. Recall that the Transformer processes the input in two stages:

1. Self-Attention Layer: The layer computes:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

and then:

$$H = \text{softmax}\Big(\frac{QK^\top}{\sqrt{d}}\Big)V.$$

For the permuted input:

- **Compute:**
$$Q_{\text{perm}} = X_{\text{perm}}W_Q = (PX)W_Q = P(XW_Q) = PQ.$$
Similarly,
$$K_{\text{perm}} = PK \quad \text{and} \quad V_{\text{perm}} = PV.$$

- **Softmax Transformation:** Consider the product:
$$Q_{\text{perm}}K_{\text{perm}}^\top = (PQ)(PK)^\top.$$
Since $(PK)^\top = K^\top P^\top$, we have:
$$Q_{\text{perm}}K_{\text{perm}}^\top = P\,QK^\top\,P^\top.$$
Dividing by $\sqrt{d}$ and applying the softmax using the property
$$\text{softmax}\Big(PAP^\top\Big) = P\,\text{softmax}(A)P^\top,$$

with $A = \frac{QK^\top}{\sqrt{d}}$, it follows that:

$$\operatorname{softmax}\Big(\frac{Q_{\text{perm}}K_{\text{perm}}^\top}{\sqrt{d}}\Big) = P\operatorname{softmax}\Big(\frac{QK^\top}{\sqrt{d}}\Big)P^\top.$$

Then, the output of the self-attention layer for the permuted input is:

$$H_{\text{perm}} = \operatorname{softmax}\Big(\frac{Q_{\text{perm}}K_{\text{perm}}^\top}{\sqrt{d}}\Big)V_{\text{perm}} = \Big[P\operatorname{softmax}\Big(\frac{QK^\top}{\sqrt{d}}\Big)P^\top\Big](PV).$$

Since $P^\top P = I$, we simplify to:

$$H_{\text{perm}} = P\operatorname{softmax}\Big(\frac{QK^\top}{\sqrt{d}}\Big)V = PH.$$

2. Feed-Forward Network Layer: The feed-forward layer applies the transformation:

$$Z = \operatorname{ReLU}(HW_1 + \mathbf{1}\,b_1)W_2 + \mathbf{1}\,b_2,$$

where $\mathbf{1} \in \mathbb{R}^{T\times 1}$ is a vector of ones.

For the permuted hidden state $H_{\text{perm}} = PH$, we have:

$$S_{\text{perm}} = H_{\text{perm}}W_1 + \mathbf{1}\,b_1 = PHW_1 + \mathbf{1}\,b_1.$$

Since permutation matrices reorder rows and $\mathbf{1}$ remains unchanged under permutation ($P\mathbf{1} = \mathbf{1}$), we can factor out P:

$$S_{\text{perm}} = P\,(HW_1 + \mathbf{1}\,b_1).$$

Applying ReLU and using the property

$$\operatorname{ReLU}(PA) = P\operatorname{ReLU}(A),$$

gives:

$$\operatorname{ReLU}(S_{\text{perm}}) = P\operatorname{ReLU}(HW_1 + \mathbf{1}\,b_1).$$

Finally, the output is:

$$Z_{\text{perm}} = \operatorname{ReLU}(S_{\text{perm}})W_2 + \mathbf{1}\,b_2 = P\operatorname{ReLU}(HW_1 + \mathbf{1}\,b_1)W_2 + \mathbf{1}\,b_2.$$

Again, since $\mathbf{1}$ is invariant under P,

$$Z_{\text{perm}} = P\Big[\operatorname{ReLU}(HW_1 + \mathbf{1}\,b_1)W_2 + \mathbf{1}\,b_2\Big] = PZ.$$

Thus, we have shown:

$$\boxed{Z_{\text{perm}} = PZ.}$$

(ii) Implications for Processing Text

The result $Z_{\text{perm}} = PZ$ demonstrates that the Transformer's operations—both the self-attention and feed-forward layers—are *permutation equivariant*. That is, shuffling the input tokens simply shuffles the output in the same way.

Implications:

- **Lack of Order Sensitivity:** In natural language, the meaning of a sentence critically depends on the order of words. A model that is permutation equivariant would treat sentences like "The cat sat on the mat" and "On the mat sat the cat" as equivalent, thereby ignoring the sequential structure.

- **Need for Positional Information:** To address this, Transformers add *positional embeddings* to the input word embeddings, thus injecting order information that breaks the permutation symmetry.

- **Potential Misinterpretation:** Without positional information, even small changes in word order might lead to misinterpretations. Positional embeddings ensure that the model can distinguish between different token orders.

(b) The Role and Uniqueness of Position Embeddings

Position embeddings encode the position of each token in the sequence and are added to the input word embeddings:

$$X_{\text{pos}} = X + \Phi,$$

where the position embeddings $\Phi \in \mathbb{R}^{T\times d}$ are defined as:

$$\Phi(t, 2i) = \sin\left(\frac{t}{10000^{2i/d}}\right), \quad \Phi(t, 2i+1) = \cos\left(\frac{t}{10000^{2i/d}}\right),$$

with $t \in \{0, 1, \dots, T-1\}$ and $i \in \{0, 1, \dots, d/2-1\}$.

(i) Do Positional Embeddings Help?

Yes, positional embeddings help address the issue identified in part (a). Since the Transformer layers are inherently permutation equivariant, without positional information the model would treat all input tokens as unordered. By adding position embeddings:

- Each token's embedding is augmented with a unique signal indicating its position in the sequence.

- This additional information breaks the permutation symmetry, allowing the model to capture the sequential order of words—a critical property for understanding natural language.

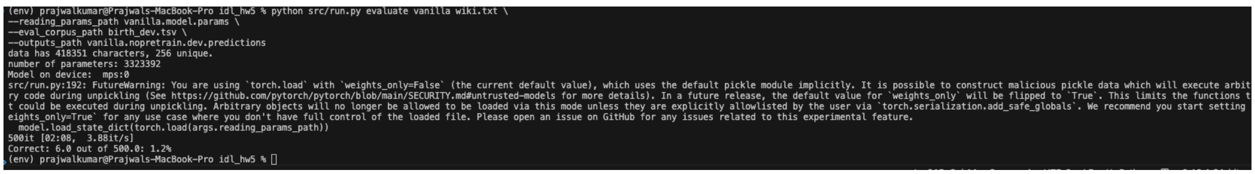
(ii) Can Two Different Tokens Have the Same Position Embedding?

No, In the standard formulation, the position embedding for each token is entirely determined by its position t in the sequence. Therefore, for two tokens at different positions ($t \neq t'$), the position embeddings $\Phi(t, \cdot)$ and $\Phi(t', \cdot)$ are computed using different values, and they are designed to be unique.

Question 2: Pretrained Transformer mdoels and knowledge access

(d) Predictions

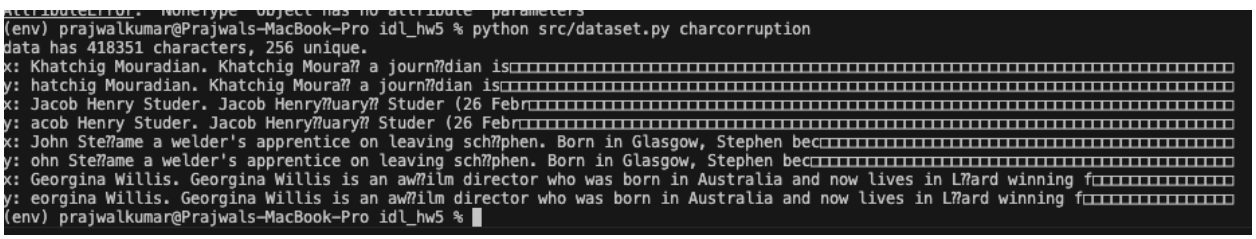
- **My model's accuracy on the dev set Correct: 6.0 out of 500.0: 1.2%**
- **My London Baseline accuracy is 5% (25/500)**



```
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 % python src/run.py evaluate vanilla wiki.txt \
--reading_params_path vanilla.model.params \
--eval_corpus_path birth_dev.tsv \
--outputs_path vanilla.nopretrain.dev.predictions
data has 418351 characters, 256 unique.
number of parameters: 3323392
Model on device:  mps:0
src/run.py:192: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(args.reading_params_path))
500it [02:08,  3.88it/s]
Correct: 6.0 out of 500.0: 1.2%
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 %
```

Figure 1: Dev set accuracy

(e) Span corruption function



```
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 % python src/dataset.py charcorruption
data has 418351 characters, 256 unique.
x: Khatchig Mouradian. Khatchig Moura⁇ a journ⁇dian is
y: hatchig Mouradian. Khatchig Moura⁇ a journ⁇dian is
x: Jacob Henry Studer. Jacob Henry⁇uary⁇ Studer (26 Febr
y: acob Henry Studer. Jacob Henry⁇uary⁇ Studer (26 Febr
x: John Ste⁇ame a welder's apprentice on leaving sch⁇phen. Born in Glasgow, Stephen bec
y: ohn Ste⁇ame a welder's apprentice on leaving sch⁇phen. Born in Glasgow, Stephen bec
x: Georgina Willis. Georgina Willis is an aw⁇ilm director who was born in Australia and now lives in L⁇ard winning f
y: eorgina Willis. Georgina Willis is an aw⁇ilm director who was born in Australia and now lives in L⁇ard winning f
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 %
```

Figure 2: Output of python src/dataset.py charcorruption

(f) Pretrain

My model's accuracy on the dev set Correct: 108.0 out of 500.0: 21.6%

```
--writing_params_path vanilla.finetune.params \
--finetune_corpus_path birth_places_train.tsv
data has 418351 characters, 256 unique.
number of parameters: 3323392
Model on device:  mps:0
src/run.py:182: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(args.reading_params_path))
Loaded pretrained parameters. Finetuning for 10 epochs.
epoch 1 iter 7: train loss 0.77357. lr 5.999664e-04: 100%|| 8/8 [00:06<00:00,  1.26it/s]
epoch 2 iter 7: train loss 0.53933. lr 5.998600e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 3 iter 7: train loss 0.50591. lr 5.996808e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 4 iter 7: train loss 0.44335. lr 5.994287e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 5 iter 7: train loss 0.38640. lr 5.991038e-04: 100%|| 8/8 [00:05<00:00,  1.34it/s]
epoch 6 iter 7: train loss 0.36352. lr 5.987063e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 7 iter 7: train loss 0.28722. lr 5.982362e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 8 iter 7: train loss 0.27668. lr 5.976936e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 9 iter 7: train loss 0.22702. lr 5.970787e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
epoch 10 iter 7: train loss 0.16861. lr 5.963916e-04: 100%|| 8/8 [00:05<00:00,  1.35it/s]
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 %
```

Figure 3: Finetuning the model

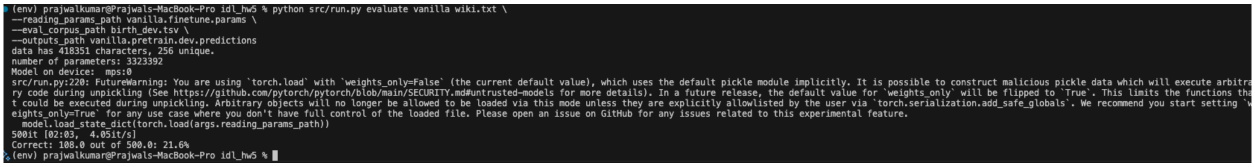


Figure 4: Dev set accuracy

(g) Rope

(i) RoPE via Complex Multiplication

Concept: Each pair of features $[x_t^{(1)}, x_t^{(2)}]$ is thought of as a complex number

$$z_t = x_t^{(1)} + i\, x_t^{(2)}.$$

Rotating this complex number by an angle $t\theta$ means multiplying it by

$$e^{i\,t\theta} = \cos(t\theta) + i\, \sin(t\theta).$$

Thus,

$$e^{i\,t\theta}\, z_t = \big(\cos(t\theta) + i\, \sin(t\theta)\big) \times \big(x_t^{(1)} + i\, x_t^{(2)}\big).$$

Writing out the multiplication and then taking the real and imaginary parts gives:

- **Real part:** $\cos(t\theta)x_t^{(1)} - \sin(t\theta)x_t^{(2)}$
- **Imaginary part:** $\sin(t\theta)x_t^{(1)} + \cos(t\theta)x_t^{(2)}$

This is exactly the effect of multiplying by the 2×2 rotation matrix:

$$\begin{pmatrix} \cos(t\theta) & -\sin(t\theta) \\ \sin(t\theta) & \cos(t\theta) \end{pmatrix} \begin{pmatrix} x_t^{(1)} \\ x_t^{(2)} \end{pmatrix}.$$

Thus, by grouping pairs of dimensions into complex numbers, Equation 2 (the element-wise multiplication by complex exponentials) produces the same rotated vectors as Equation 1 (the block diagonal rotation matrix) after appropriate reshaping.

(ii) Relative Position via Dot Product

Goal: Show that

$$\langle \mathrm{RoPE}(z_1, t_1),\, \mathrm{RoPE}(z_2, t_2) \rangle = \langle \mathrm{RoPE}(z_1, t_1 - t_2),\, \mathrm{RoPE}(z_2, 0) \rangle.$$

Explanation: In the complex formulation, define

$$\mathrm{RoPE}(z,t) = e^{i\,t\theta}\,z.$$

Then the dot product between two rotated vectors (interpreted as the real part of the product with the conjugate) is given by

$$\langle \mathrm{RoPE}(z_1,t_1),\, \mathrm{RoPE}(z_2,t_2)\rangle = \mathrm{Re}\left(e^{i\,t_1\theta} z_1\,\overline{e^{i\,t_2\theta} z_2}\right) = \mathrm{Re}\left(e^{i\,(t_1-t_2)\theta} z_1\,\overline{z_2}\right).$$

Since the term $e^{i\,(t_1-t_2)\theta}$ depends only on the difference $t_1 - t_2$, we can equivalently write:

$$\langle \mathrm{RoPE}(z_1, t_1 - t_2),\, \mathrm{RoPE}(z_2, 0)\rangle,$$

which shows that the dot product depends only on the relative position $t_1 - t_2$.

(iii) RoPE Implementation

My model's accuracy on the dev set Correct: 160.0 out of 500.0: 32%

```
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 % python src/run.py evaluate rope wiki.txt \
--reading_params_path rope.finetune.params \
--eval_corpus_path birth_dev.tsv \
--outputs_path rope.pretrain.dev.predictions
data has 418351 characters, 256 unique.
number of parameters: 3290624
Model on device:  cpu
src/run.py:220: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses
ry code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details).
t could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they a
eights_only=True` for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for
  model.load_state_dict(torch.load(args.reading_params_path))
500it [00:33, 14.76it/s]
Correct: 160.0 out of 500.0: 32.0%
(env) prajwalkumar@Prajwals-MacBook-Pro idl_hw5 %
```

Figure 5: RoPE accuracy output

Question 3: Considerations in Pretrained Knowledge (Revised)

(a)

The pretrained (vanilla) model benefits from large-scale pretraining on diverse text, which allows it to learn robust language representations. This prior knowledge helps the model achieve an accuracy well above 10% on the downstream task (e.g., our pretrained model reached 21.6% accuracy on the dev set), whereas a non-pretrained model, learning only from limited task-specific data, achieves very poor performance (e.g., 1.2% accuracy). In other words, pretraining provides a strong prior that facilitates generalization when fine-tuning on smaller datasets.

(b)

Even though the pretrain+finetuned vanilla model produces both correct predictions (e.g., in some cases achieving up to 32% accuracy with RoPE) and errors, its output does not indicate whether a predicted birth place has been accurately retrieved from factual knowledge or is simply fabricated. Two main concerns for user-facing systems are:

1. **Misinformation:** Users might accept an output as correct without knowing that it could be a made-up fact. For instance, if a chatbot returns a plausible but incorrect birth place for a celebrity, users may be misled.

2. **Lack of Accountability:** Without clear evidence of the source—retrieved fact versus generated guess—it becomes difficult to audit or correct errors in sensitive applications. This opacity can be problematic in domains where accuracy is critical, such as legal, educational, or medical contexts.

(c)

For names that were unseen during both pretraining and fine-tuning, the model must still produce a prediction. A likely strategy is to default to a statistical heuristic, such as predicting the most frequent birth place observed in the training data. Although this offers a way to respond when data is sparse, it is problematic because it may propagate biases and lead to systematic errors; the prediction is essentially a best-guess rather than a data-driven, factual response. This lack of specificity and reliability is concerning, especially in applications where precise and trustworthy information is required.