22 Attention

Goal

Introduction to transformer, attention, BERT, GPTs, and the exploding related.
A nice code tutorial is available: https://nlp.seas.harvard.edu/2018/04/03/attention.html.

Alert 22.1: Convention

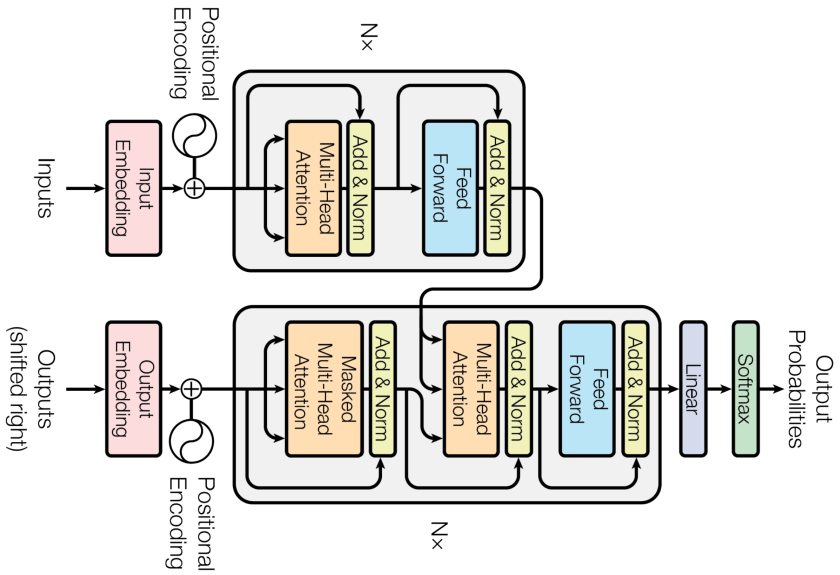
Gray boxes are not required hence can be omitted for unenthusiastic readers. Tokens are arranged row-wise.
This note is likely to be updated again soon.

Remark 22.2: The input sequential price of RNNs

A lot of natural language processing (NLP) techniques rely on recurrent neural networks, which are unfortunately sequential in nature (w.r.t. input tokens). Our main goal in this lecture is to use a hierarchical, parallelizable attention mechanism to trade the input sequential part in RNNs with that in network depth.

Definition 22.3: Transformer (Vaswani et al. 2017)

In a nutshell, a transformer (in machine learning!) is composed of multiple blocks of components that we explain in details below. It takes an input sequence and outputs another sequence, much like an RNN:



Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30*, pp. 5998–6008.

Definition 22.4: Input and output embeddings

Typically, the tokens in an input sequence are one-hot encoded over a dictionary with size say p. We add and learn a distributed representation $W_e \in \mathbb{R}^{p\times d}$ of the tokens. The same W_e is also used to decode the output tokens, and its transpose $W_e^\top$ is used to compute the `softmax` output probability (over tokens).

Definition 22.5: Positional encoding

The order of tokens in an input sequence matters. To encode this information, we may simply add a positional vector $\mathfrak{p}_t \in \mathbb{R}^d$ for each fixed position t:

$$\mathfrak{p}_{t,2i} = \sin\left(t/10000^{2i/d}\right), \quad \mathfrak{p}_{t,2i+1} = \cos\left(t/10000^{2i/d}\right), \quad i = 0, \ldots, \tfrac{d}{2} - 1.$$

It is clear that $\mathfrak{p}_{t+k}$ is an orthogonal transformation of $\mathfrak{p}_t$, since

$$\begin{bmatrix} \mathfrak{p}_{t+k,2i} \\ \mathfrak{p}_{t+k,2i+1} \end{bmatrix} = \begin{bmatrix} \cos(k/10000^{2i/d}) & \sin(k/10000^{2i/d}) \\ -\sin(k/10000^{2i/d}) & \cos(k/10000^{2i/d}) \end{bmatrix} \begin{bmatrix} \sin\left(t/10000^{2i/d}\right) \\ \cos\left(t/10000^{2i/d}\right) \end{bmatrix}$$
$$= \begin{bmatrix} \cos(k/10000^{2i/d}) & \sin(k/10000^{2i/d}) \\ -\sin(k/10000^{2i/d}) & \cos(k/10000^{2i/d}) \end{bmatrix} \begin{bmatrix} \mathfrak{p}_{t,2i} \\ \mathfrak{p}_{t,2i+1} \end{bmatrix}.$$

The periodic functions chosen here also allows us to handle test sequences that are longer than the training sequences. Of course, one may instead try to directly *learn* the positional encoding $\mathfrak{p}_t$.

Definition 22.6: Residual connection, layer-normalization and dropout

In each layer we add residual connection (He et al. 2016) and layer-wise normalization (Ba et al. 2016) to ease training. We may also add dropout layers (Srivastava et al. 2014).

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). "Layer Normalization".

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958.

Definition 22.7: Attention (e.g. Bahdanau et al. 2015)

Given a query vector $\mathbf{q} \in \mathbb{R}^{d_k}$ and a database consisting of m key-value pairs $(K, V) : K = [\mathbf{k}_1, \ldots, \mathbf{k}_m]^\top \in \mathbb{R}^{m \times d_k}, V = [\mathbf{v}_1, \ldots, \mathbf{v}_m]^\top \in \mathbb{R}^{m \times d_v}$, a natural way to guess/retrieve the value of the query is through comparison to the known key-value pairs:

$$\texttt{att}(\mathbf{q}; K, V) = \sum_{i=1}^{m} \pi_i \cdot \mathbf{v}_i = \boldsymbol{\pi}^\top V, \tag{22.1}$$

namely the value of the query is some convex combination of the values in the database.

The coefficient vector $\boldsymbol{\pi}$ can be determined as follows:

$$\operatorname*{argmin}_{\boldsymbol{\pi} \in \Delta_{m-1}} \sum_{i=1}^{m} \pi_i \cdot \text{dist}_k(\mathbf{q}, \mathbf{k}_i) + \lambda \cdot \pi_i \log \pi_i, \tag{22.2}$$

where $\text{dist}_k(\mathbf{q}, \mathbf{k}_i)$ measures the dissimilarity between the query $\mathbf{q}$ and the key $\mathbf{k}_i$, and $\pi_i \log \pi_i$ is the so-called entropic regularizer. As you can verify in Exercise 22.8:

$$\pi_i = \frac{\exp(-\text{dist}_k(\mathbf{q}, \mathbf{k}_i)/\lambda)}{\sum_{j=1}^{m} \exp(-\text{dist}_k(\mathbf{q}, \mathbf{k}_j)/\lambda)}, \quad i.e., \quad \boldsymbol{\pi} = \texttt{softmax}(-\text{dist}_k(\mathbf{q}, K)/\lambda). \tag{22.3}$$

A popular choice for the dissimilarity function is $\text{dist}_k(\mathbf{q}, \mathbf{k}) = -\mathbf{q}^\top \mathbf{k}$.

Now, with the above $\boldsymbol{\pi}$, we solve a similar weighted regression problem to retrieve the value of the query:

$$\operatorname*{argmin}_{\mathbf{a}} \sum_{i=1}^{m} \pi_i \cdot \text{dist}_v(\mathbf{a}, \mathbf{v}_i).$$

With $\text{dist}_v(\mathbf{a}, \mathbf{v}_i) := \|\mathbf{a} - \mathbf{v}_i\|_2^2$, we easily verify the convex combination in (22.1).

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural machine translation by jointly learning to align and translate". In: *International Conference on Learning Representations*.

Exercise 22.8: KL leads to `softmax`

Prove that the optimal solution of (22.2) is given by the `softmax` in (22.3).

Example 22.9: Popular attention mechanisms

Popular choices for the dissimilarity function dist_k include:

- (negated) dot-product: we have already mentioned this choice $\text{dist}_k(\mathbf{q}, \mathbf{k}) = -\mathbf{q}^\top \mathbf{k}$. We further point out that when normalized, i.e. $\|\mathbf{q}\|_2 = \|\mathbf{k}\|_2 = 1$, then the (negated) dot-product essentially measures the angular distance between the query and the key. This choice is particularly efficient, as we can easily implement the matrix-product $QK^\top$ (for a set of queries and keys) in parallel. One typically sets $\lambda = \sqrt{d_k}$ so that if $\mathbf{q} \perp \mathbf{k} \sim \mathcal{N}(0, I_{d_k})$, then $\text{Var}(\mathbf{q}^\top \mathbf{k}/\sqrt{d_k}) = 1$.
- additive: more generally we may parameterize the dissimilarity function as a feed-forward network $\text{dist}_k(\mathbf{q}, \mathbf{k}; \mathbf{w})$ whose weight vector $\mathbf{w}$ is learned from data.

Exercise 22.10: Self-attention in the limit

Let $V \in \mathbb{R}^{m \times d}$ be arbitrary and consider applying self-attention repeatedly to it:

$$V \leftarrow \mathsf{A}_\lambda(V) := \texttt{softmax}(VV^\top/\lambda)V,$$

where of course the `softmax` operator is applied row-wise. What is the limiting behaviour of $\mathsf{A}_\lambda^\infty(V)$?

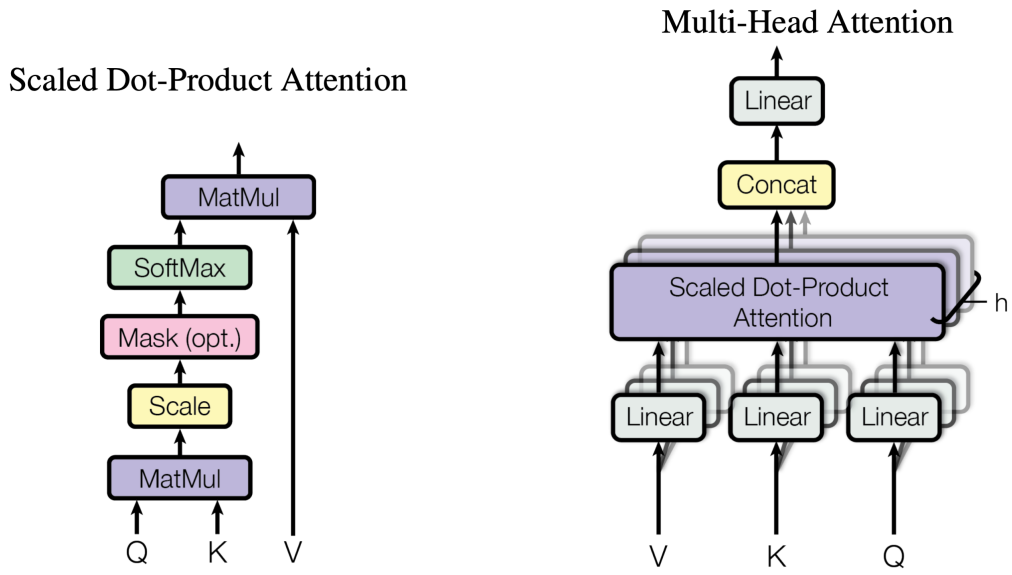
Definition 22.11: Multi-head attention

Recall that in CNNs, we employ a number of filters to extract different feature maps. Similarly, we use multi-head attention so that our model can learn to attend to different parts of the input simultaneously:

$$H = [A_1, \dots, A_h]W, \text{ with } A_i = \texttt{att}(QW_i^Q; KW_i^K, VW_i^V), \quad i = 1, \dots, h,$$

where $W \in \mathbb{R}^{(hd_v) \times d}, W_i^Q \in \mathbb{R}^{d \times d_k}, W_i^K \in \mathbb{R}^{d \times d_k}, W_i^V \in \mathbb{R}^{d \times d_v}$ are weight matrices to be learned, and `att` is an attention mechanism in Definition 22.7 (applied to each row of QW_i^Q; see also Example 22.9).

If we set $d_k = d_v = d/h$, then the number of parameters in h-head attention is on par with single-head attention. The choice of dimensions here also facilitates the implementation of residual connections above.



Remark 22.12: implicit distance learning?

Another way to interpret the linear projections W_i^Q and W_i^K is through distance learning. Indeed, the distance between the query and key, after linear projection, is

$$-(QW_i^Q)(KW_i^K)^\top = -Q\big(W_i^Q(W_i^K)^\top\big)K^\top =: -QM_iK^\top,$$

where the low-rank matrix $M_i \in \mathbb{R}^{d\times d}$ "distorts" the dot-product: $\langle \mathbf{q},\mathbf{k}\rangle_{M_i} := \mathbf{q}^\top M_i\mathbf{k}$. This explanation suggests tying together W_i^Q, W_i^K, and possibly also W_i^V.

Definition 22.13: Self- and context- attention

The transformer uses multi-head attention along with an encoder-decoder structure, and employs self-attention (e.g. Cheng et al. 2016) as a computationally efficient way to relate different positions in an input sequence:

- encoder self-attention: In this case $Q = K = V$ all come from the input (of the current encoder layer). Each output can attend to all positions in the input.
- context attention: In this case Q comes from the input of the current decoder layer while (K, V) come from the output of (the final layer of) the encoder, namely the context. Again, each output can attend to all positions in the input.
- decoder self-attention: In this case Q comes from the input (of the current decoder layer), $K = Q \odot M$ and $V = Q \odot M$ are masked versions of Q so that each output position can only attend to positions up to and including the current position. In practical implementation we can simply reset *illegal* dissimilarities:

$$\text{dist}_k(\mathbf{q}_i, \mathbf{q}_j) \leftarrow \infty \quad \text{if} \quad i < j.$$

The input to the transformer is a sequence $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m]^\top \in \mathbb{R}^{m\times p}$ and it generates an output sequence $Y = [\mathbf{y}_1, \ldots, \mathbf{y}_l]^\top \in \mathbb{R}^{l\times p}$. Note that we shift the output sequence 1 position to the right, so that combined with the decoder self-attention above, each output symbol only depends on symbols before it (not including itself).

Cheng, Jianpeng, Li Dong, and Mirella Lapata (2016). "Long Short-Term Memory-Networks for Machine Reading". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 551–561.

Definition 22.14: Position-wise feed-forward network

In each encoder and decoder layer, we also apply a feed-forward network at each position. For instance, let $\mathbf{h}_t \in \mathbb{R}^d$ be the (row) representation for position t (at some layer), then we compute

$$\texttt{FFN}(\mathbf{h}_t) = \sigma(\mathbf{h}_t^\top \mathsf{W}_1)\mathsf{W}_2.$$

The weights $\mathsf{W}_1 \in \mathbb{R}^{d\times 4d}$ and $\mathsf{W}_2 \in \mathbb{R}^{4d\times d}$ are shared among different positions but change from layer to layer.

Remark 22.15: Comparison between attention, RNN and CNN

Layer type	per-layer complexity	sequential operations	max path length
Self-attention	$O(m^2d)$	$O(1)$	$O(1)$
Recurrent	$O(md^2)$	$O(m)$	$O(m)$
Convolution	$O(kmd^2)$	$O(1)$	$O(\log_k m)$
Self-attention (restricted)	$O(rmd)$	$O(1)$	$O(m/r)$

Let m be the length of an input sequence and d the internal representation dimension (as above). In self-attention, the dot-product $QQ^\top$ costs $O(m^2d)$ (recall that $Q \in \mathbb{R}^{m\times d}$). However, this matrix-matrix multiplication can be trivial parallelized using GPUs. We define the maximum path length to be the maximum number of sequential operations for any output position to attend to any input position. Clearly, for the transformer, each output position can attend to each input position hence its maximum path length is $O(1)$.

In contrast, for RNNs, computation is sequential in terms of the tokens in the input sequence. Each recurrence, e.g. $\mathbf{h} \leftarrow W_2\sigma(W_1(\mathbf{h}, \mathbf{x}))$, costs $O(d^2)$, totaling $O(md^2)$. For CNNs (e.g. Gehring et al. 2017) with filter size k, each convolution costs $O(kd)$ and for a single output filter we need to repeat m times while we have d output filters, hence the total cost $O(kmd^2)$. Convolutions can be trivially parallelized on GPUs, and if we employ dilated convolutions (Kalchbrenner et al. 2016) the maximum path length is $O(\log_k m)$ (and $O(m/k)$ for usual convolutions with stride k). Separable convolutions (Chollet 2017) can reduce the per-layer complexity to $O(kmd + md^2)$.

Finally, if we restrict attention to the r neighbors (instead of all m tokens in the input sequence), we may reduce the per-payer complexity to $O(rmd)$, at the cost of increasing the maximum path length to $O(m/r)$.

From the comparison we see that transformer (with restricted attention, if quadratic time/space complexity is a concern) is very suitable for modeling long-range dependencies.

Gehring, Jonas, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin (2017). "Convolutional Sequence to Sequence Learning". In: *Proceedings of the 34th International Conference on Machine Learning*, pp. 1243–1252.

Kalchbrenner, Nal, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu (2016). "Neural Machine Translation in Linear Time". In:

Chollet, F. (2017). "Xception: Deep Learning with Depthwise Separable Convolutions". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1800–1807.

Remark 22.16: Attention as interpretation?

One advantage of the attention mechanism is visualization: we can inspect the attention distribution over layers or positions and try to interpret the model; see (Jain and Wallace 2019; Wiegreffe and Pinter 2019) for some discussions.

Jain, Sarthak and Byron C. Wallace (2019). "Attention is not Explanation". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 3543–3556.

Wiegreffe, Sarah and Yuval Pinter (2019). "Attention is not not Explanation". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 11–20.

Definition 22.17: Training using machine translation

Vaswani et al. (2017) trained the transformer using the supervised machine translation problem on the WMT 2014 dataset that consists of pairs (X, Y) of sentences, one (X) from the source language (say English) and the other (Y) from the target language (say French or German). The usual (multi-class) cross-entropy loss is used as the objective function:

$$\min \quad -\hat{\mathsf{E}}\left[\left\langle Y, \log \hat{Y} \right\rangle\right], \quad \hat{Y} = [\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_\ell]^\top,$$

where $\hat{\mathbf{y}}_j$ depends on the input sentence $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m]^\top$ and all previous target tokens $Y_{<j} := [\mathbf{y}_1, \ldots, \mathbf{y}_{j-1}]^\top$. Note that the sequence lengths m and l vary from one input to another. In practice, we "bundle" sentence pairs with similar lengths to streamline the parallel computation.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30*, pp. 5998–6008.

Remark 22.18: Some practicalities about Transformer

We mention some training choices in the original work:

- optimizer: Adam (Kingma and Ba 2015) was used with $\beta_1 = 0.9, \beta_2 = 0.98$ and $\epsilon = 10^{-9}$.
- learning rate:

  $$\eta_{\texttt{iter}} = \frac{1}{\sqrt{d}} \cdot \min\{\frac{1}{\sqrt{\texttt{iter}}}, \texttt{iter} \cdot \frac{1}{\tau\sqrt{\tau}}\},$$

  where $\tau = 4000$ controls the warm-up stage where the learning rate increases linearly. After that, the learning rate decreases inverse proportionally to the square root of the iteration number.
- regularization: (a) dropout (with rate 0.1) was added after the positional encoding layer and after each attention layer; (b) residual connection and layer normalization is performed after each attention layer and feed-forward layer; (c) label smoothing (Szegedy et al. 2016):

  $$\mathbf{y} \leftarrow (1-\alpha)\mathbf{y} + \alpha\frac{1}{C},$$

  where $\mathbf{y}$ is the (original, typically one-hot encoded) label vector, C is the number of classes, and $\alpha = 0.1$ controls the amount of smoothing.

Kingma, D. P. and J. Ba (2015). "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations*.

Szegedy, C., V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna (2016). "Rethinking the Inception Architecture for Computer Vision". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826.

Definition 22.19: Beam search during inference time

After we have trained the transformer, during test time we are given an input sentence and asked to decode its translation. We use beam search:

- We keep $b = 4$ currently best candidate translations;
- We augment each candidate translation Y_k with a next word $\mathbf{y}$:

  $$\texttt{score}_k \leftarrow \texttt{score}_k - \log \hat{P}(\mathbf{y}|X, Y_k).$$

  We may prune the next word by considering only those whose score contribution lies close to the best one (up to some threshold, say b).

- We keep only the best b (in terms of score) augmented translations.
- If some candidate translation ends (either by outputting the stop word or exceeding maximum allowed length, say input length plus 50), we compute its normalized score by dividing its length to the power of $\alpha = 0.6$ (to reduce bias towards shorter translations).
- We prune any candidate translation that lies some threshold (say b) below the best normalized score.

Note that *beam search is highly sequential and subject to possible improvements.*

Definition 22.20: Embedding from Language Models (ELMo) (Peters et al. 2018)

Unlike conventional word embeddings that assign each word a *fixed* representation, ELMo is contextualized (Melamud et al. 2016; McCann et al. 2017), where the representation for each word depends on the entire sentence (context) it lies in. ELMo is applied for down-stream tasks in the following way:

- **Two-Stage** (**TS**, not to be confused with few-shot learning in Example 22.27): We fix the parameters in ELMo and use it as an (additional) feature extractor, on top of which we tune a task-specific architecture. For the latter, Peters et al. (2018) also found that stacking ELMo with original token representation $\mathbf{x}$ in its input layer and with its output before passing to a `softmax` layer appears to improve performance.

ELMo trains a bidirectional LM:

$$\min_{\Theta,\Phi} \ \hat{\mathsf{E}} - \log \overrightarrow{\mathsf{p}}\ (X|\Theta) - \log \overleftarrow{\mathsf{p}}\ (X|\Phi), \quad \text{where}$$

$$\overrightarrow{\mathsf{p}}\ (X|\Theta) = \prod_{j=1}^{m} \mathsf{p}(\mathbf{x}_j|\mathbf{x}_1,\ldots,\mathbf{x}_{j-1};\Theta), \quad \overleftarrow{\mathsf{p}}\ (X|\Phi) = \prod_{j=1}^{m} \mathsf{p}(\mathbf{x}_j|\mathbf{x}_{j+1},\ldots,\mathbf{x}_m;\Phi),$$

and the two probabilities are modeled by two LSTMs with shared embedding and `softmax` layers but different hidden parameters.

Given an input sequence, the ELMo representation of any token $\mathbf{x}$ is computed as:

$$\texttt{ELMo}(\mathbf{x};\mathsf{A}) = \mathsf{A}(\mathbf{h}_0, \{\overrightarrow{\mathbf{h}}_l\}_{l=1}^L, \{\overleftarrow{\mathbf{h}}_l\}_{l=1}^L) =: \mathsf{A}(\mathbf{h}_0, \{\mathbf{h}_l\}_{l=1}^L), \quad \mathbf{h}_l := [\overrightarrow{\mathbf{h}}_l; \overleftarrow{\mathbf{h}}_l],$$

where recall that the embedding $\mathbf{h}_0$ (e.g. $W_e\mathbf{x}$, or extracted from any context insensitive approach) is shared between the two L-layer LSTMs, whose hidden states are $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$, respectively. Here A is an aggregation function. Typical choices include:

- Top: $\mathsf{A}(\mathbf{h}_0, \{\overrightarrow{\mathbf{h}}_l\}_{l=1}^L, \{\overleftarrow{\mathbf{h}}_l\}_{l=1}^L) = \mathbf{h}_L$, where only the top layers are retained.
- ELMo as in (Peters et al. 2018):

$$\texttt{ELMo}(\mathbf{x};\mathbf{s},\gamma) = \gamma \sum_{l=0}^{L} s_l \mathbf{h}_l,$$

where the layer-wise scaling parameters $\mathbf{s}$ and global scaling parameter γ are task-dependent.

ELMo employs 3 layers of representation: context-insensitive embedding through character-level CNN, followed by the forward and backward LSTMs. With $L = 2$, Peters et al. (2018) found that the lower layers tend to learn syntactic information while semantic information is captured in higher layers (through e.g. testing on tasks that require syntactic/semantic information), justifying the aggregation scheme in ELMo.

Melamud, Oren, Jacob Goldberger, and Ido Dagan (2016). "context2vec: Learning Generic Context Embedding with Bidirectional LSTM". In: *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pp. 51–61.

McCann, Bryan, James Bradbury, Caiming Xiong, and Richard Socher (2017). "Learned in Translation: Contextualized Word Vectors". In: *Advances in Neural Information Processing Systems 30*, pp. 6294–6305.
Peters, Matthew, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, and Luke Zettlemoyer (2018). "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 2227–2237.

Definition 22.21: Bidirectional Encoder Representation from Transformers (Devlin et al. 2019)

BERT followed up on the pre-training path in GPT, and added some interesting twists:

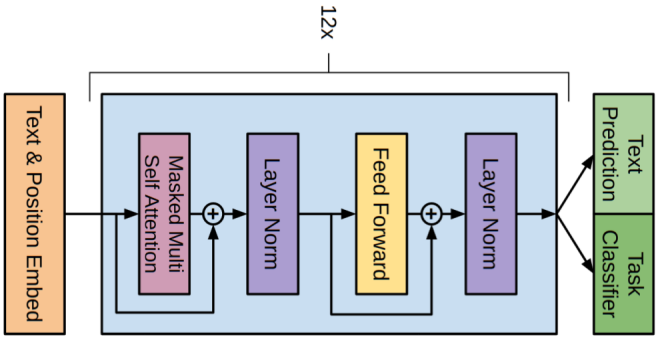
- The input to BERT is a concatenation of two sequences, starting with a special symbol `[CLS]`, followed by sequence A, the special separator `[SEP]`, sequence B, and ending again with `[SEP]`. (It obviously still works in the absence of a sequence B.) The final representation of `[CLS]` is a sentence-level abstraction and can be used for various downstream sentence classification tasks, while the two-sequence input format can be extremely useful in question answering tasks (e.g. question for A and passage for B).
- Apart from token positional embedding, we also add a sequence positional embedding, where tokens in the first sequence share an embedding vector while those in the second sequence share another one.
- **Masked language model (MLM)**: As in ELMo (Definition 22.20), Devlin et al. (2019) argue that in certain tasks it is important to exploit information from both directions, instead of the left-to-right order in usual language models (LMs, e.g. GPT). Thus, BERT aims at training an MLM: On each input, it randomly replaces 15% tokens with the special symbol `[Mask]`. The modified input then goes through the Transformer encoder where each position can attend to any other position (hence bidirectional). The final hidden representations are passed to a `softmax` layer where we predict only the masked tokens with the usual cross-entropy loss. Note that our predictions on the masked tokens are in parallel, unlike in usual LMs where tokens are predicted sequentially and may affect each other (at test time).
- **Hack**: At test time, the input never includes the special symbol `[Mask]` hence creating a mismatch between training and testing in BERT. To remedy this issue, of the 15% tokens chosen during training, only 80% of them will actually be replaced with `[Mask]`, while 10% of them will be replaced with a random token and the remaining 10% will remain unchanged.
- Next sequence prediction (NSP): Given the first sequence, we choose the second sequence as its next sequence 50% of the time (labeled as `true`) and as a random sequence the remaining time (labeled as `false`). Then, BERT pre-training includes a binary classification loss besides MLM, based on the (binary) `softmax` applied on the final hidden representation of `[CLS]`.
- The training loss of BERT is the sum of averaged MLM likelihood and NSP likelihood.
- **Fine-tuning (FT)**: during fine-tuning, depending on the task we may proceed differently. For sequence classification problems, we add a `softmax` layer to the final hidden representation of `[CLS]`, while for token-level predictions we add `softmax` layers to the final hidden representations of all relevant input tokens. For instance, we add two (independent) `softmax` layers (corresponding to `start` and `end`) in span prediction where during test time we use the approximation

$$\log \Pr(\texttt{start} = i, \texttt{end} = j) \approx \log \Pr(\texttt{start} = i) + \log \Pr(\texttt{end} = j),$$

  considering of course only $i \leq j$. Note that all parameters are adjusted in FT, in sharp contrast to the TS approach in ELMo. Of course, BERT can also be applied in the TS setting, where the performance may be slightly worse (Devlin et al. 2019).
- BERT showed that scaling to extreme model sizes leads to surprisingly large improvements on very small scale tasks, provided that the model has been sufficiently pre-trained.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 4171–4186.

Definition 22.22: Generative Pre-Training (GPT) (Radford et al. 2018)



GPT works in two stages:

- unsupervised pre-training through learning a language model:

$$\min_{\Theta} \quad -\hat{\mathsf{E}} \log \mathsf{p}(X|\Theta), \quad \text{where} \quad \mathsf{p}(X|\Theta) = \prod_{j=1}^{m} \mathsf{p}(\mathbf{x}_j|\mathbf{x}_1, \ldots, \mathbf{x}_{j-1}; \Theta).$$

  Namely, given the context consisting of previous tokens $\mathbf{x}_1, \ldots, \mathbf{x}_{j-1}$, we aim to predict the current token $\mathbf{x}_j$. The conditional probability is computed through a multi-layer transformer decoder (Liu et al. 2018):

$$\begin{aligned} H^{(0)} &= XW_e + W_p \\ H^{(\ell)} &= \texttt{transformer\_decoder\_block}(H^{(\ell-1)}), \quad \ell = 1, \ldots, L \\ \mathsf{p}(\mathbf{x}_j|\mathbf{x}_1, \ldots, \mathbf{x}_{j-1}; \Theta) &= \texttt{softmax}(\mathbf{h}_j^{(L)} W_e^\top), \end{aligned}$$

  where $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m]^\top$ is the input sequence consisting of m tokens (m may vary from input to input), L is the number of transformer blocks, W_e is the token embedding matrix and W_p is the position embedding matrix.

- supervised fine-tuning with task-aware input transformations:

$$\min_{W_y} \min_{\Theta} \quad -\hat{\mathsf{E}} \log \mathsf{p}(\mathbf{y}|X, \Theta) - \lambda \cdot \hat{\mathsf{E}} \log \mathsf{p}(X|\Theta), \quad \text{where} \quad \mathsf{p}(\mathbf{y}|X, \Theta) = \left\langle \mathbf{y}, \texttt{softmax}(\mathbf{h}_m^{(L)} W_y) \right\rangle,$$

  and we include the unsupervised pre-training loss to help improving generalization and accelerating convergence. Note that for different tasks, we only add an extra `softmax` layer to do the classification. Unlike ELMo, GPT avoids task-specific architectures and aims to learn a *universal* representation (through language model pre-training) for *all* tasks.

Sequence pre-training has been explored earlier in (Dai and Le 2015; Howard and Ruder 2018) using LSTMs.

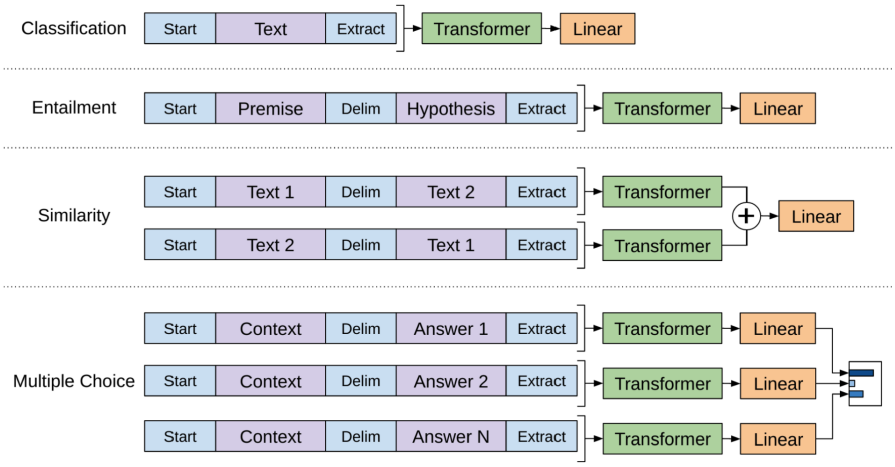
Liu, Peter J., Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer (2018). "Generating Wikipedia by Summarizing Long Sequences". In: *International Conference on Learning Representations*.

Dai, Andrew M and Quoc V Le (2015). "Semi-supervised Sequence Learning". In: *Advances in Neural Information Processing Systems 28*, pp. 3079–3087.

Howard, Jeremy and Sebastian Ruder (2018). "Universal Language Model Fine-tuning for Text Classification". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pp. 328–339.

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever (2018). "Improving Language Understanding by generative pre-training".

Remark 22.23: Input transformations



GPT employs task-specific input transformations so that its fine-tuning strategy in Definition 22.22 is applicable:

- **Textual entailment**: Given `premise`, decide if `hypothesis` holds. We simply concatenate `premise` with `hypothesis`, delimited by a special token, as input to GPT and reduce to 3-class classification: `entailment, neutral, contradiction\verb`.
- **Similarity**: Similarly, we concatenate the two input sentences in both orders and add the resulting representation.
- **Question answering and reasoning**: We concatenate the context, question and each possible answer to obtain multiple input sequences and reduce to multi-class classification.

Remark 22.24: Some practicalities about GPT

We mention the following implementation choices in GPT:

- Byte pair encoding (BPE): Current character-level language models (LMs) (e.g. Gillick et al. 2016) are not as competitive as word-level LMs (e.g. Al-Rfou et al. 2019). BPE (e.g. Sennrich et al. 2016) provides a practical middle ground, where we start with UTF-8 *bytes* (256 base vocabs instead of the overly large 130k codes) and repeatedly merge pairs with the highest frequency in our corpus. We prevent merging between different (UTF-8 code) categories, with an exception on spaces (to reduce similar vocabs such as `dog` and `dog!` but allow `dog cat`). BPE (or any other character-level encoding) allows us to compute probabilities even over words and sentences that are not seen at training.
- Gaussian Error Linear Unit (GELU) (Hendrycks and Gimpel 2016):

$$\sigma(x) = x\Phi(x) = \mathsf{E}(x \cdot m|x), \quad \text{where} \quad m \sim \text{Bernoulli}(\Phi(x)),$$

  i.e., on the input x, with probability $\Phi(x)$ we drop it. Unlike Relu where we drop any negative input, GELU drops the input more probably as the latter decreases.
- fine-tuning uses a smaller batch size 32, and converges after 3 epochs in most cases. $\lambda = \frac{1}{2}$; warm-up $\tau = 2000$ during pre-training and 0.2% during fine-tuning.

Gillick, Dan, Cliff Brunk, Oriol Vinyals, and Amarnag Subramanya (2016). "Multilingual Language Processing From Bytes". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 1296–1306.
Al-Rfou, Rami, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones (2019). "Character-Level Language Modeling with Deeper Self-Attention". In: *AAAI*.
Sennrich, Rico, Barry Haddow, and Alexandra Birch (2016). "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 1715–1725.

Hendrycks, Dan and Kevin Gimpel (2016). "Gaussian Error Linear Units (GELUs)".

Alert 22.25: "What I cannot create, I do not understand." — Richard Feynman

A popular hypothesis to support pre-training (through say language modeling) is that the underlying generative model learns to perform many of the downstream (supervised) tasks in order to improve its language modeling capability, and the long-range dependencies allowed by transformers assists in transfer compared to LSTMs (or other RNNs).

Example 22.26: GPT-2 (Radford et al. 2019)

Most current ML systems are narrow in the sense that they specialize exclusively on a single task. This approach, however successful, suffers from generalizing to other domains/tasks that are not encountered during training. While unsupervised pre-training combined with supervised fine-tuning (such as in GPT) proves effective, GPT-2 moves on to the completely unsupervised zero-shot setting.

McCann et al. (2018) showed that many NLP tasks (input and output) can be specified purely by language, and hence can be solved through training a *single* model. Indeed, the target output to a natural language task is just one of the many possible *next sentences*. Thus, training a sufficiently large *unsupervised* language model may allow us to perform well on a range of tasks (not explicitly trained with supervision), albeit in a much less data-efficient way perhaps.

GPT-2 also studied the effect of test set contamination, where the large training set accidentally includes near-duplicates of part of the test data. For example CIFAR-10 has 3.3% overlap between train and test images (Barz and Denzler 2020).

Apart from some minor adjustments, the main upgrade from GPT to GPT-2 is a (sharp) increase in model capacity, a larger training set, and the exclusive focus on zero-shot learning. Conceptually, GPT-2 demonstrated the (surprising?) benefit of training (excessively?) large models, with near-human performance on some NLP tasks (such as text generation).

McCann, Bryan, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher (2018). "The Natural Language Decathlon: Multitask Learning as Question Answering".

Barz, Björn and Joachim Denzler (2020). "Do We Train on Test Data? Purging CIFAR of Near-Duplicates". *Journal of Imaging*, vol. 6, no. 41, pp. 1–8.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever (2019). "Language models are unsupervised multitask learners".

Example 22.27: GPT-3 (Brown et al. 2020)

The main contribution of GPT-3 is perhaps its crystallization of different evaluation schemes:

- Fine-Tuning (FT): this was explored in GPT-1 and involves task-dedicated datasets and gradient updates on both the LM and the task-dependent classifier.
- Few-Shot (FS): a natural language description of the task (e.g. `Translate English to French`) and a few example demonstrations (e.g. English-French sentence pairs), followed by a final context (e.g. English sentence) that will be completed by GPT-3 (e.g. a French translation).
- One-Shot (1S): same as above with the number of demonstrations restricted to 1.
- Zero-Shot (0S): just the task description (in natural language form) is provided.

GPT-3 focuses exclusively on the last 3 scenarios and never performs gradient updates on the LM. The Common Crawl dataset that GPT-3 was built on is so large that no sequence was ever updated twice during training.

Brown, Tom B. et al. (2020). "Language Models are Few-Shot Learners".

Alert 22.28: The grand comparison

model	data	en	de	input	embedding	heads	params	batch	steps	GPUs	time
transformer	WMT14	6	6	-	512	8	213M	50k tokens	100k	8×P100	12h
ELMo	1B Word Bench	-	2×2	2048c	512	-	?	?	10e	?	?
GPT-1	BooksCorp	0	12	512×40k	768	12	100M	64	100e	8×P600	1m
$\text{BERT}_{\text{base}}$	BooksCorp+Wiki	12	0	512×30k	768	12	110M	256	1M	4×cTPU	4d
$\text{BERT}_{\text{large}}$		24			1024	16	340M			16×cTPU	
GPT-2	WebText	0	12	1024× 50k	768	12	117M	512	?	?	?
			24		1024		345M				
			36		1280		762M				
			48		1600		1542M				
GPT-3-S	CommonCrawl	0	12	2048× 50k	768	12	125M	0.5M	?	?×V100	?
GPT-3-M			24		1024	16	350M	0.5M			
GPT-3-L			24		1536	16	760M	0.5M			
GPT-3-XL			24		2048	24	1.3B	1M			
GPT-3-SS			32		2560	32	2.7B	1M			
GPT-3-MM			32		4096	32	6.7B	2M			
GPT-3-LL			40		5140	40	13B	2M			
GPT-3			96		12288	96	175B	3.2M			