# 90-minütige Klausur zur Vorlesung „Deep Learning for NLP / Profilierungsmodul II“ (WS 23/24),

H. Schütze, M. Assenmacher, L. Weissweiler

HAUPTKLAUSUR / MAIN EXAM (12.02.2024)

---

| | |
|---|---|
| Vorname (First name): | |
| Nachname (Last name): | |
| Matrikelnummer: | |
| Studiengang (Studies): | ☐ M.Sc. Computerlinguistik, ☐ M.Sc. Informatik, ☐ Magister |
| | ☐ M.Sc. Statistics and Data Science, ☐ M.Sc. ESG Data Science |
| | ☐ M.Sc. Statistik/WiSo-Statistik/Biostatistik |
| | ☐ anderer/other: |

**Points:**

| Aufgabe (Task) | mögliche Punkte (possible points) | erreichte Punkte (achieved points) |
|---|---|---|
| 1. Neural Networks | 23 | |
| 2. Pre-trained models | 21 | |
| 3. Transformer parameters | 20 | |
| 4. Hallucinations | 16 | |
| 5. PyTorch | 10 | |
| Summe (Sum) | 90 | |
| Note (Grade) | | |

**Deutsch:**

- Die Klausur besteht aus **5 Aufgaben** auf **18** Seiten.
- Die Punktzahl ist bei jeder Aufgabe angegeben. Die Bearbeitungsdauer beträgt **90 Minuten** (*Tipp: Die Punkte orientieren sich ungefähr an der Anzahl der Minuten, die man brauchen sollte, um eine Aufgabe zu lösen.*).
- Bitte überprüfen Sie, ob Sie ein vollständiges Exemplar erhalten haben.
- Nur die in den Kästen eingetragenen Ergebnisse werden bepunktet; falls der Platz in einem der Kästen nicht ausreicht, benutzen Sie bitte die Zusatzblätter an Ende Klausur! Im betreffenden Kasten muss ein Verweis zur Lösung zu finden sein.
- Verwenden Sie einen dokumentenechten Kugelschreiber oder Füller, **keine** Bleistifte.
- Als Hilfsmittel ist ein Taschenrechner/Lexikon zugelassen.
- **Sie können Fragen auf Deutsch oder Englisch bearbeiten**.
- Bitte tragen Sie **zuerst**, d.h., bevor Sie die Aufgaben lösen, auf **allen** Seiten Ihren Namen ein und füllen Sie die Titelseite aus.

**English:**

- The exam consists of **5 tasks** on **18** pages.
- The score is given for each task. The given time is **90 minutes** (*Hint: The points are approximately based on the number of minutes it should take to solve a task*).
- Please check that you have received a complete copy.
- Only the results entered in the boxes will be scored; if there is not enough space in one of the boxes, please use the additional sheets at the end of the exam! There must be a reference to the solution in the relevant box.
- Use a document-safe ballpoint pen or fountain pen, **no** pencils.
- A calculator/dictionary is permitted as an aid.
- **You can work on questions in German or English**.
- **First**, i.e. before you solve the tasks, please write your name on **all** pages and fill in the title page.

NAME:

# Aufgabe 1 Neural Networks

(a) Explain how a vanilla[1] seq2seq encoder-decoder model is constructed (2-3 sentences). (2 Punkte)

(b) Explain what the unique ability of a vanilla seq2seq encoder-decoder is (1 sentence) and name two NLP tasks for which it is therefore well suited. Further, briefly explain its major flaw as well as "attention", a mechanism to alleviate this flaw (1-2 sentences each). (4 Punkte)

(c) Explain (i) the architectural differences between the encoder and the decoder part of the transformer (1 sentence) and (ii) how the attention mechanisms differ between the two parts (2-3 sentences). (3 Punkte)

[1] "vanilla" means a standard, usual, or unmodified version of something

NAME:

Let $H$ be the output representation of a (self-attention) transformer head for a sentence $x$ of length $J$. Let the parameters of the head be $W^q \in \mathbb{R}^{d_{model} \times d_q}, W^k \in \mathbb{R}^{d_{model} \times d_k}$ and $W^v \in \mathbb{R}^{d_{model} \times d_v}$ and let $X \in \mathbb{R}^{J \times d_{model}}$ be the input representation of $x$ for the transformer head.

(d) Provide the formulas to calculate query $Q$, key $K$ and value $V$ representations and write down their dimensions (3 Punkte)

$Q =$ , $Q \in$

$K =$ , $K \in$

$V =$ , $V \in$

(e) Complete the formula for the calculation of $H$ and state which requirement regarding $d_q$, $d_k$, and $d_v$ need to be fulfilled so that this works. (2 Punkte)

$$H = \text{softmax}\left(\frac{1}{\sqrt{d_k}} \right)$$

Requirement regarding $d_q$, $d_k$, and $d_v$:

(f) Which of the following would lead to the same output at each position of a sequence, that is to $H_i = H_j$ for all $1 \leq i, j \leq J$ (you only get points with the correct reason why). (6 Punkte)

(a) Setting all positional encodings to $\vec{0}$ (all zeros).
☐ yes; ☐ no
Why/why not? (1 sentence)

(b) Setting all query weights $W^q$ to $\vec{0}$.
☐ yes; ☐ no
Why/why not? (1 sentence)

(c) Adding a constant $\vec{a}$ to all query weights $W^q$.
☐ yes; ☐ no
Why/why not? (1 sentence)

(d) Setting all value weights $W^v$ to $\vec{0}$.
☐ yes; ☐ no
Why/why not? (1 sentence)

(g) Let $n$ be the sequence length, and $d$ be the representation dimension. What is the **computational** complexity (in terms of $n$ and $d$) per layer for the following? (2 Punkte)

- Number of operations (float values to compute) for the Transformer:

- Number of operations for a Recurrent Neural Network:

(h) What is the **time**-complexity of the transformer layer when executed on a GPU, assuming that matrix multiplication and softmax normalization take constant time (and not taking into consideration reading in the data): (1 Punkt)

2+4+3+3+2+6+2+1=23 PUNKTE

NAME:

# Aufgabe 2 Pre-trained models

(a) Explain, why simply applying the language modeling objective would not work for the BERT architecture (2-3 sentences). (2 Punkte)

(b) Explain briefly why positional encodings are utterly necessary in transformer models. Then, discuss the advantages and disadvantages of learned vs. fixed positional encodings in transformer models. (3-4 sentences) (3 Punkte)

(c) Let's consider a simple vanilla transformer encoder that consists of a self-attention layer on top of only a simple word embedding lookup layer (i.e. no positional encodings).

We have two sentences:

"*Paul ist der Vater von Peter*" *and* "*Peter ist der Vater von Paul*"

What will be the difference between the model's representation of "Paul" in the first and second sentence? Explain why! (1 sentence) (1 Punkt)

(d) Explain each term from the list below such that the conceptual differences between them are clear. Further, state for each of them *(i) from which* parameter values the process is started and *(ii) which parameters are updated* during the process. Use a transformer decoder model (e.g., a model from the GPT family) as an example. Each explanation should encompass 1-2 short sentences.

- pre-training
- fine-tuning
- zero-shot learning
- one-shot learning

(6 Punkte)

(e) For the task of classifying whether the review "I love this pizza!" has a positive or negative sentiment, create a "zero-shot" and a "one-shot" prompt, also including the sentence itself.

(2 Punkte)

(f) In T5 (text-to-text-transfer transformer) task-specific prefixes are added to the samples before they are fed to the model. Explain the following points: (1-2 short sentences each)

- Specify the dimensionality and the activation function of the output layer. (1 Punkt)
- How do the labels have to be formatted? (1 Punkt)
- In which way does this extend T5's capabilities beyond those of e.g. BERT? (1 Punkt)
- In how far does this differ from prompting? (2 Punkte)

(g) Neural networks are almost exclusively trained in a supervised fashion (input/label pairs for computing the loss, etc.). But BERT is pre-trained without labeled data and you can say that 'no labels = unsupervised'. Can you explain this terminological discrepancy? (2-3 short sentences) (2 Punkte)

2+3+1+6+2+1+1+1+2+2=21 PUNKTE

NAME:

# Aufgabe 3 Transformer parameters

In your answers, use the notation given below:

| Notation | |
|---|---|
| $D$ | size of training corpus |
| $d_{head}$ | head/attention dimension |
| $d_{model}$ | model/embedding dimension |
| $mem_{model}$ | memory needs of model in bytes |
| $n_{heads}$ | number of heads |
| $n_{layers}$ | number of layers |
| $n_{voc}$ | size of vocabulary |

Parameters that do not contribute to the primary model scaling can be neglected.

(a) How many parameters does **one individual** attention matrix of a transformer, e.g. the query matrix, have? Give one sentence of explanation. (2 Punkte)

(b) How many parameters does the multilayer perceptron (MLP) of a transformer have? Give one sentence of explanation. (2 Punkte)

(c) In a transformer, the output of the attention operation is mapped to the input vector of the MLP. How many parameters are needed for this part? Give one sentence of explanation. (2 Punkte)

(d) Write down a formula of the number of parameters of one layer of the transformer considering only the three components we have covered so far (the attention matrices, the MLP, the matrix mapping from attention matrix output to MLP input). Do not yet simplify the formula. Give one sentence of explanation. (2 Punkte)

(e) To simplify the formula from the last subproblem as shown in the lecture, there is one key assumption you have to make. Which assumption is this? Making this assumption, simplify the formula from the last subproblem as we did in the lecture. (3 Punkte)

(f) What is the number of parameters of a transformer with $n_{layers}$ layers, taking into account only the components we have explicitly mentioned so far? (1 Punkt)

(g) The actual number of parameters of a transformer is larger than the correct answer to the previous question. Why? Give one single reason. (1 Punkt)

(h) The MLP in a standard transformer processes a vector that represents a subword at a particular position. We will now change the architecture as follows. The MLP still takes as input the vector of its own position, but also the vectors of the two preceding subwords and the following subword. The MLP output configuration doesn't change. Derive and simplify the number of parameters this new MLP configuration has. (4 Punkte)

(i) In this modified architecture what is the total number of parameters of a transformer with $n_{layers}$ layers? Give one sentence of explanation. (3 Punkte)

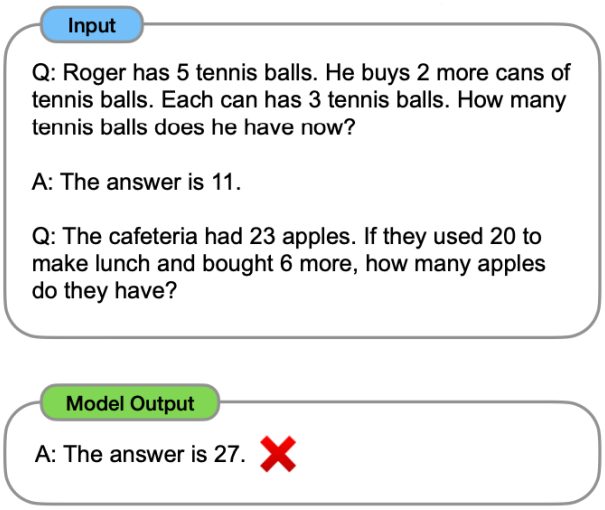
2+2+2+2+3+1+1+4+3=20 PUNKTE

NAME:

# Aufgabe 4 Hallucinations

Below you will be asked to explain why LLM models hallucinate in particular situations. **For the last four subproblems [(d) – (g)], attempt to make your explanations complementary. It is sufficient to give one reason that has caused the hallucination per subproblem as long as the reasons you give for the different subproblems are distinct.**

(a) In the example shown below, the LLM gives an incorrect answer. This is not a hallucination. Why? Explain in one sentence. (1 Punkt)




(b) Define what an LLM hallucination is (1 sentence). According to your definition, is the example given in the previous subproblem a hallucination? Why (1 sentence)?
An LLM generates: "I will now tell you a fairy tale. . . . Under her eyes, the frog transformed into a handsome prince. . . . ". According to your definition, is this a hallucination? Why (1 sentence)? (3 Punkte)

(c) Hallucinations are caused by several overlapping, but distinguishable causes. Name **four** distinct causes and write one sentence of explanation for each. Number these reasons. **Recommendation: Do this subproblem last. First solve the next four subproblems, then come back to this one.** If you answer subproblems (d) – (g) correctly, then all you have to do for this subproblem is to copy four distinct reasons for hallucinations from (d) – (g) to this subproblem. (4 Punkte)

(d) In response to "Please write a short bio of Oskar Fischer", GPT4 responds on 2024-01-30: "Oskar Fischer (19 March 1923 – 2 January 2020) was a prominent Czechoslovak and Czech diplomat and politician, best known for his role as the Minister of Foreign Affairs of Czechoslovakia from 1980 to 1990." This is a hallucination. Oskar Fischer was an obscure East-German politician; most people who are not East-German of a certain generation will not have heard of him. He was born (as an ethnic German) in Czechoslawakia. What is the reason for this hallucination? (2 Punkte)

(e) In the continuation of the above exchange, we asked: "Can you say a little bit more about his life before he became Minister of Foreign Affairs of Czechoslovakia in 1980?" The response included: "Fischer joined the Communist Party of Czechoslovakia". This is a hallucination. In the context of the beginning of the conversation shown in the previous subproblem, what is the reason for this hallucination? (2 Punkte)

(f) In response to the question: "What is better: tricep dip or triceps dip?", PALM2 responds with a table in which it compares "tricep dips" with "triceps dips". For example, the table has a row labeled "Suitability" which states that tricep dips are suitable for everyone whereas triceps dips are for people who have injuries or limitations that prevent them from doing bodyweight dips. In reality, there is no difference between a tricep dip and a triceps dip. What is the reason for this hallucination? (2 Punkte)

(g) Michael Wooldridge is an Oxford professor. Some time ago, he asked ChatGPT about himself. ChatGPT wrote a pretty good bio of Wooldridge, but it contained one hallucination. It said: "Wooldridge received his undergraduate degree from Cambridge". This is false: he received his undergraduate degree from a different university. What is the reason for this hallucination? Relate your argument to ChatGPT's training objective. (2 Punkte)

1+3+4+2+2+2+2=16 PUNKTE

NAME:

# Aufgabe 5 PyTorch

(a) In the appendix of this exam, you can find several excerpts from the official PyTorch documentation. The code for the TransformerBlock and Multihead Attention are also given in the appendix. Using those, find eight errors hidden in the following code for a Vanilla Transformer Decoder. Highlight the places with errors (e.g. by drawing circles around them) and, using footnotes, write down the corrections. (8 Punkte)

```
class DecoderBlock(nn.Module):
    def __init__(self, emb_dim, num_heads, forward_dim, dropout):
        super().__init__()

        self.norm = nn.LayerNorm(emb_dim, eps=1e-6)
        self.mha = MultiHeadAttention(emb_dim, num_heads)
        self.transformer_block = TransformerBlock(emb_dim, num_heads, dropout, forward_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, value, key, src_mask, tgt_mask):
        attention = self.mha(x, x, x, src_mask)
        query = self.norm(self.dropout(attention))
        out = self.transformer_block(query, key, value, tgt_mask)
        return out

class Decoder(nn.Module):
    def __init__(self, vocab_size, emb_dim, num_layers, num_heads, forward_dim, dropout, max_len):
        super().__init__()

        self.emb_dim = emb_dim
        self.dropout = nn.Dropout(dropout)


        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.rel_pos_embedding = nn.Embedding(vocab_size, emb_dim)

        self.layers = nn.ModuleList([
        DecoderBlock(emb_dim, num_heads, forward_dim, dropout)
                for _ in range(num_layers)])

        self.linear_out = nn.Linear(emb_dim, vocab_size)

    def forward(self, x, encoder_out, src_mask, tgt_mask):
        seq_len, batch_size = x.shape
        device = x.device

        positions = torch.arange(seq_len).expand(batch_size, seq_len).to(device)
        embedded = self.embedding(x)
        rel_pos_emb = self.rel_pos_embedding(positions)
        out = embedded + rel_pos_emb

        for layer in self.layers:
            out = layer(out, out, encoder_out, src_mask, tgt_mask)
        return out
```

NAME:

(b) You are using the HuggingFace Transformers library, specifically the Seq2SeqTrainer, to fine-tune a T5 model for Summarization, but you face an issue: Your GPU runs out of memory. **Name and briefly describe (1 sentence) two arguments/strategies the SeqSeqTrainingArguments provides to reduce the amount of GPU memory.**
*Note: You do not need to give the exact name of the training argument, e.g. "num_layers vs. number of layers*". (2 Punkte)

8+2=10 PUNKTE

# Zusatzblatt

# Zusatzblatt

# Appendix A: Linear Layer

Table of Contents

## LINEAR

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

**Parameters**

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

**Variables**

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape $(\text{out\_features})$. If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Examples:

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

‹ Previous Next ›

# Appendix B: Embedding Layer

Table of Contents

## EMBEDDING

CLASS torch.nn.Embedding(*num_embeddings*, *embedding_dim*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*, *_weight=None*, *_freeze=False*, *device=None*, *dtype=None*) [SOURCE]

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

**Parameters**

- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **padding_idx** (*int*, *optional*) – If specified, the entries at `padding_idx` do not contribute to the gradient; therefore, the embedding vector at `padding_idx` is not updated during training, i.e. it remains as a fixed "pad". For a newly constructed Embedding, the embedding vector at `padding_idx` will default to all zeros, but can be updated to another value to be used as the padding vector.
- **max_norm** (*float*, *optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- **norm_type** (*float*, *optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*bool*, *optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool*, *optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

**Variables**

**weight** (*Tensor*) – the learnable weights of the module of shape (num_embeddings, embedding_dim) initialized from $\mathcal{N}(0, 1)$

Shape:

- Input: $(*)$, IntTensor or LongTensor of arbitrary shape containing the indices to extract
- Output: $(*, H)$, where * is the input shape and $H = \text{embedding\_dim}$

• NOTE

Keep in mind that only a limited number of optimizers support sparse gradients: currently it's `optim.SGD` (*CUDA* and *CPU*), `optim.SparseAdam` (*CUDA* and *CPU*) and `optim.Adagrad` (*CPU*)

• NOTE

When `max_norm` is not `None`, `Embedding`'s forward method will modify the `weight` tensor in-place. Since tensors needed for gradient computations cannot be modified in-place, performing a differentiable operation on `Embedding.weight` before calling `Embedding`'s forward method requires cloning `Embedding.weight` when `max_norm` is not `None`. For example:

```
n, d, m = 3, 5, 7
embedding = nn.Embedding(n, d, max_norm=True)
W = torch.randn((m, d), requires_grad=True)
idx = torch.tensor([1, 2])
a = embedding.weight.clone() @ W.t()  # weight must be cloned for this to be differentiable
b = embedding(idx) @ W.t()  # modifies weight in-place
out = (a.unsqueeze(0) + b.unsqueeze(1))
loss = out.sigmoid().prod()
loss.backward()
```

Examples:

# Appendix C: Dropout Layer

Table of Contents

## DROPOUT

CLASS torch.nn.Dropout(*p=0.5*, *inplace=False*) [SOURCE]

During training, randomly zeroes some of the elements of the input tensor with probability `p`.

The zeroed elements are chosen independently for each forward call and are sampled from a Bernoulli distribution.

Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors.

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

**Parameters**

- **p** (*float*) – probability of an element to be zeroed. Default: 0.5
- **inplace** (*bool*) – If set to `True`, will do this operation in-place. Default: `False`

Shape:

- Input: $(*)$. Input can be of any shape
- Output: $(*)$. Output is of the same shape as input

Examples:

```
>>> m = nn.Dropout(p=0.2)
>>> input = torch.randn(20, 16)
>>> output = m(input)
```

‹ Previous Next ›

# Appendix D: Layer Norm

Table of Contents

## LAYERNORM

CLASS torch.nn.LayerNorm(*normalized_shape*, *eps=1e-05*, *elementwise_affine=True*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies Layer Normalization over a mini-batch of inputs.

This layer implements the operation as described in the paper Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated over the last *D* dimensions, where *D* is the dimension of `normalized_shape`. For example, if `normalized_shape` is `(3, 5)` (a 2-dimensional shape), the mean and standard-deviation are computed over the last 2 dimensions of the input (i.e. `input.mean((-2, -1))`). $\gamma$ and $\beta$ are learnable affine transform parameters of `normalized_shape` if `elementwise_affine` is `True`. The standard-deviation is calculated via the biased estimator, equivalent to *torch.var(input, unbiased=False)*.

• NOTE

Unlike Batch Normalization and Instance Normalization, which applies scalar scale and bias for each entire channel/plane with the `affine` option, Layer Normalization applies per-element scale and bias with `elementwise_affine`.

This layer uses statistics computed from input data in both training and evaluation modes.

**Parameters**

- **normalized_shape** (*int* or *list* or *torch.Size*) –
  input shape from an expected input of size

  $$[* \times \text{normalized\_shape}[0] \times \text{normalized\_shape}[1] \times \ldots \times \text{normalized\_shape}[-1]]$$

  If a single integer is used, it is treated as a singleton list, and this module will normalize over the last dimension which is expected to be of that specific size.
- **eps** (*float*) – a value added to the denominator for numerical stability. Default: 1e-5
- **elementwise_affine** (*bool*) – a boolean value that when set to `True`, this module has learnable per-element affine parameters initialized to ones (for weights) and zeros (for biases). Default: `True`.
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias (only relevant if `elementwise_affine` is `True`). Default: `True`.

**Variables**

- **weight** – the learnable weights of the module of shape $\text{normalized\_shape}$ when `elementwise_affine` is set to `True`. The values are initialized to 1.
- **bias** – the learnable bias of the module of shape $\text{normalized\_shape}$ when `elementwise_affine` is set to `True`. The values are initialized to 0.

Shape:

- Input: $(N, *)$
- Output: $(N, *)$ (same shape as input)

Examples:

```
>>> # NLP Example
>>> batch, sentence_length, embedding_dim = 20, 5, 10
>>> embedding = torch.randn(batch, sentence_length, embedding_dim)
>>> layer_norm = nn.LayerNorm(embedding_dim)
>>> # Activate module
>>> layer_norm(embedding)
>>>
>>> # Image Example
>>> N, C, H, W = 20, 5, 10, 10
>>> input = torch.randn(N, C, H, W)
>>> # Normalize over the last three dimensions (i.e. the channel and spatial dimensions)
>>> # as shown in the image below
>>> layer_norm = nn.LayerNorm([C, H, W])
>>> output = layer_norm(input)
```

# Appendix E: Module List

Table of Contents

## MODULELIST

CLASS torch.nn.ModuleList(*modules=None*) [SOURCE]

Holds submodules in a list.

`ModuleList` can be indexed like a regular Python list, but modules it contains are properly registered, and will be visible by all `Module` methods.

**Parameters**

**modules** (*iterable, optional*) – an iterable of modules to add

Example:

```
class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

append(*module*) [SOURCE]

Append a given module to the end of the list.

**Parameters**

**module** (*nn.Module*) – module to append

**Return type**

*ModuleList*

extend(*modules*) [SOURCE]

Append modules from a Python iterable to the end of the list.

**Parameters**

**modules** (*iterable*) – iterable of modules to append

**Return type**

*Self*

insert(*index*, *module*) [SOURCE]

Insert a given module before a given index in the list.

**Parameters**

- **index** (*int*) – index to insert.
- **module** (*nn.Module*) – module to insert

‹ Previous Next ›

# Appendix F: Transformer Block and Multihead Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()

        self.emb_dim = emb_dim
        self.num_heads = num_heads
        self.head_dim = emb_dim // num_heads

        self.Q_linear = nn.Linear(self.emb_dim, num_heads * self.head_dim)
        self.K_linear = nn.Linear(self.emb_dim, num_heads * self.head_dim)
        self.V_linear = nn.Linear(self.emb_dim, num_heads * self.head_dim)
        self.linear_out = nn.Linear(num_heads * self.head_dim, emb_dim)

    def forward(self, query, key, value, mask=None):
        batch_size = query.shape[0]

        Q = self.Q_linear(query).view(batch_size, -1, self.num_heads, self.head_dim)
        K = self.K_linear(key).view(batch_size, -1, self.num_heads, self.head_dim)
        V = self.V_linear(value).view(batch_size, -1, self.num_heads, self.head_dim)

        key_out = torch.einsum("nqhd, nkhd -> nhqk", Q, K)


        if mask is not None:
            key_out = key_out.masked_fill(mask == 0, -1e20)

        attn = F.softmax(key_out / (self.head_dim ** (1 / 2)), dim=3)


        out = torch.einsum("nhql, nlhd -> nqhd", attn, V).reshape(
            batch_size, query.shape[1], self.num_heads * self.head_dim
        )

        return self.linear_out(out)
```

```
class TransformerBlock(nn.Module):
    def __init__(self, emb_dim, num_heads, dropout, forward_dim):
        super().__init__()
        self.mha = MultiHeadAttention(emb_dim, num_heads)

        self.dropout = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(emb_dim, eps=1e-6)
        self.norm2 = nn.LayerNorm(emb_dim, eps=1e-6)

        self.ffn = nn.Sequential(
            nn.Linear(emb_dim, forward_dim),
            nn.ReLU(),
            nn.Linear(forward_dim, emb_dim),
        )

    def forward(self, query, key, value, mask):
        attention = self.mha(query, key, value, mask)

        x = self.norm1(self.dropout(attention + query))

        ffn = self.ffn(x)
        out = self.norm2(self.dropout(ffn + x))
        return out
```