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

H. Schütze, M. Assenmacher, L. Weissweiler

NACHKLAUSUR / RETAKE EXAM (26.03.2024)

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. RLHF	20	
2. Multilingual LLMs	18	
3. Embeddings	16	
4. Pre-Trained models	17	
5. MLP in Transofrmer	8	
6. PyTorch	11	
Summe (Sum)	90	
Note (Grade)		

Deutsch:

- Die Klausur besteht aus **6 Aufgaben** auf **27** 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 **6 tasks** on **27** 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 RLHF

(a) There are three different models in the InstructGPT RLHF approach. Give each of these three models a name and explain in one sentence per model (i) what they are trained on and (ii) what they produce. (6 Punkte)

(b) Which of the three models is trained with the following objective? Explain in one sentence how you were able to map the model to the objective. (2 Punkte)

$$\operatorname{loss}(\theta) = -\frac{1}{\binom{K}{2}} E_{(x,y_w,y_l)\sim D}\left[\log\left(\sigma\left(r_\theta\left(x,y_w\right) - r_\theta\left(x,y_l\right)\right)\right)\right]$$

(c) Which of the three models is trained with the following objective? Explain in one sentence how you were able to map the model to the objective. (2 Punkte)

$$\text{objective}(\phi) = E_{(x,y)\sim D_{\pi_\phi^{\mathrm{RL}}}}\left[\underset{r_\theta}{\underline{r_\theta(x,y)}} - \underset{\beta}{\underline{\beta\log\left(\pi_\phi^{\mathrm{RL}}(y \mid x)/\pi^{\mathrm{SFT}}(y \mid x)\right)}}\right] + \underset{\gamma}{\underline{\gamma E_{x\sim D_{\text{pretrain}}}\left[\log(\pi_\phi^{\mathrm{RL}}(x))\right]}}$$

NAME:

(d) There are three terms in the following objective, marked as r_θ, β, and γ in the figure. Explain the role each of the three terms plays in the objective function in one or two sentences each. (6 Punkte)

$$\text{objective}(\phi) = E_{(x,y)\sim D_{\pi_\phi^{\mathrm{RL}}}}\left[\underbrace{r_\theta(x,y)}_{r_\theta} - \underbrace{\beta \log\left(\pi_\phi^{\mathrm{RL}}(y \mid x)/\pi^{\mathrm{SFT}}(y \mid x)\right)}_{\beta}\right] + \underbrace{\gamma E_{x\sim D_{\text{pretrain}}}\left[\log(\pi_\phi^{\mathrm{RL}}(x))\right]}_{\gamma}$$

(e) There are two objectives that are shown above for two of the three models. What is the objective of the third model? Explain the objective in words and write it down in formal mathematical notation. (4 Punkte)
(Hint: The objective occurs as a subpart of one of the formulas shown above, so for the mathematical part of your answer, you can just copy this subpart.)

6+2+2+6+4=20 PUNKTE

NAME:

Aufgabe 2 Multilingual LLMs

Please read the entire problem first. Then start working on the subproblems. Don't worry about the architecture unless the question explicitly asks about it.

You are given a multilingual corpus C_m. C_m contains 500 languages, 30,000 sentences each for 490 low-resource languages and 10^9 sentences each for 10 high-resource languages. We also have a corpus C_e of English that contains 10^{11} tokens. We are interested in creating a multilingual language model M_m that has at least some competence for these 500 languages. We will compare it to an English language model M_e trained on C_e.

(a) Describe the simplest way we can train the two models M_m and M_e by filling out the table below. We are using a BPE tokenizer. Describe the objective in words or with a formula. (3 Punkte)

	English	multilingual
training set for BPE		
training set for M		
objective for training M		

(b) Comparing the two BPE tokenizers (the one for M_m and the one for M_e): the BPE tokenizer for M_e is likely to work well. But the BPE tokenizer for M_m is not likely to work well; why is this the case (1-2 sentences)? (3 Punkte)

(c) An alternative to learning M_m from scratch is to start with an existing high-resource model M_h that has already learned some high-resource languages and perform what is called continued pretraining on C_m. What disadvantage does this setup have? (1-2 sentences) What advantage does this setup have? (1-2 sentences) (4 Punkte)

NAME:

(d) Compare two scenarios. (i) The 500 languages are all African languages. (ii) The 500 languages are the 500 languages with the highest number of speakers.

- There are factors that make the 500 African languages easier to learn than the 500 most spoken languages. Describe one such factor. (1 Punkt)

- There are factors that make the 500 African languages harder to learn than the 500 most spoken languages. Describe one such factor. (1 Punkt)

- One could argue that the the model for the 500 most spoken languages is more useful than the model for 500 African languages. Why? (1 Punkt)

(e) For how many epochs would you train a very large language model (say 10^{12} parameters) on C_e, the English corpus described above? Why? (1 Punkt)

NAME:

(f) For how many epochs would you train a very large language model (say 10^{12} parameters) on C_m, the multilingual corpus described above? Why? (2 Punkte)

(g) Per the description given above, C_m contains 500 languages, 30,000 sentences each for 490 low-resource languages and 10^9 sentences each for 10 high-resource languages. Before training a model on C_m, how would you change this distribution of C_M to improve the result of training? (2 Punkte)

3+3+4+1+1+1+1+1+2+2=18 PUNKTE

NAME:

Aufgabe 3 Embeddings

The word embedding models can be seen as bigram language models in which we estimate the conditional probability of some word $word_i$ appearing in the context of some other word $word_j$:

$$P(word_i|word_j) = f(\mathbf{w}^{(i)}, \mathbf{v}^{(j)})$$

with $\mathbf{w}^{(i)}$, $\mathbf{v}^{(j)} \in \mathbb{R}^d$ as the embeddings (i.e., d-dimensional vectors) of the i-th and j-th vocabulary words, respectively (we use $\mathbf{w}^{(i)}$ when predicting the i-th vocabulary word and $\mathbf{v}^{(i)}$ when the i-th word is the context for predicting some other word). embeddings and language models.

(a) What function f that produces a probability distribution is commonly used in word embedding models? Write the actual expression with which we would compute $P(word_i|word_j)$, if predicting $word_i$ by considering **all vocabulary words**. (3 Punkte)

(b) What is negative sampling and why do the embedding models like Skip-gram resort to it? Write the Skip-gram objective for a window of size $2c+1$ (i.e., c context words before and after the center token t), with T negatives for each pair of context and center word. (4 Punkte)

NAME:

(c) Specify all basic units into which the FastText tokenizer with a character n-gram range `{4,6}` would split the word **python**. (4 Punkte)

(d) How does the objective of FastText relate to the skip-gram and/or the CBOW objective? (2 Punkte)

(e) How do we typically measure the semantic similarity between two words with static word embeddings such as word2vec? What is the difficulty with this similarity metric when dealing with pre-trained models like BERT? (2 Punkte)

(f) What is one of the main disadvantages of static word embeddings, such as word2vec, regarding homonymy (two words with identical spelling and pronunciation, but different meanings)? How is this alleviated in pre-trained models like BERT? (1 Punkt)

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

NAME:

Aufgabe 4 Pre-Trained models

(a) **Name** and **describe** (1-2 sentences per objective) the self-supervised pre-training objectives of BERT and T5. Contrast them to the ordinary Language Modeling objective (i.e. what benefit do they bring beyond it). (5 Punkte)

(b) Explain the difference between Fine-Tuning and In-Context Learning. (2 Punkte)

(c) Which one (Fine-Tuning or In-Context Learning) is advantageous if you only have very little examples for a specific task? Which one if you have a large training set? Why? (2 Punkte)

NAME:

(d) Briefly describe the Chain-of-Thought prompting mechanism (1-2 sentences). (2 Punkte)

(e) Illustrate Chain-of-Thought prompting with one example. (3 Punkte)

(f) Briefly explain two benefits of Chain-of-Thought prompting. (3 Punkte)

5+2+2+2+3+3=17 PUNKTE

NAME:

Aufgabe 5 MLP in Transformer

The Multilayer Perceptron (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.

(a) We learned about two main types of parallelism in distributed training. Name these two types of parallelism. (2 Punkte)

(b) Give a brief explanation of what each type of parallelism does and how they differ. (2 Punkte)

(c) Which of these two types is interfering with the modified MLP architecture and which one is not? Give a brief explanation. (4 Punkte)

2+2+4=8 PUNKTE

NAME:

Aufgabe 6 PyTorch

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

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

        self.mha = MultiHeadAttention(emb_dim, forward_dim)

        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, num_heads),
            nn.ReLU(),
            nn.Linear(num_heads, emb_dim),
        )

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

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

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

class Encoder(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.embedding = nn.Embedding(vocab_size, emb_dim)
        self.sinusoid_table = get_sinusoid_table(max_len + 1, emb_dim)
        self.pos_encoding = nn.Embedding.from_pretrained(self.sinusoid_table, freeze=True)
        self.dropout = nn.Dropout(dropout)

        self.layers = nn.ModuleList([
            TransformerBlock(emb_dim,num_heads,dropout,forward_dim)
```

NAME:

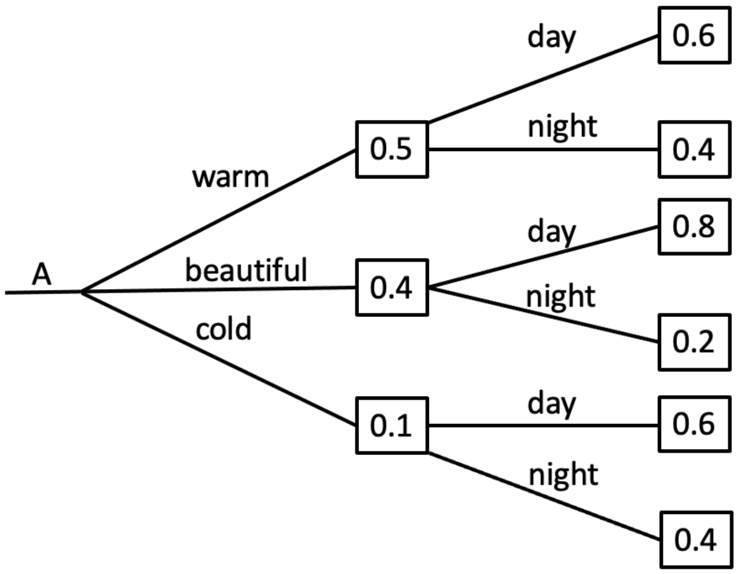
```
            for _ in range(num_layers)])

    def forward(self, x, mask):
        batch_size, seq_len = x.shape
        device = x.device

        positions = ((torch.arange(seq_len).expand(batch_size, seq_len) + 1).to(device))
        sum_emb = self.embedding(x) + self.pos_encoding(x)
        out = self.dropout(sum_emb)

        for layer in self.layers:
            out = self.dropout(layer(out, out, mask, mask)

        return out
```

NAME:

NAME:

(b) Explain shortly in 2 - 3 sentences what is early stopping and what is it used for. (1 Punkt)

(c) In the following, a prediction diagram is shown where e.g. the 0.5 box after "warm" means $P(\text{warm}|A) = 0.5$, i.e. $P(x_i|x_{<i})$. Give the predictions for the Beam searches of beam size 1 and 2 and shortly explain your reasoning. (2 Punkte)



NAME:

8+1+2=11 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 (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))`). γ and β 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 normalized_shape when `elementwise_affine` is set to `True`. The values are initialized to 1.
- **bias** – the learnable bias of the module of shape 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

Appendix F: Sequential

Table of Contents

SEQUENTIAL

CLASS torch.nn.Sequential(*args: Module) [SOURCE]

CLASS torch.nn.Sequential(arg: OrderedDict[str, Module])

A sequential container.

Modules will be added to it in the order they are passed in the constructor. Alternatively, an `OrderedDict` of modules can be passed in. The `forward()` method of `Sequential` accepts any input and forwards it to the first module it contains. It then "chains" outputs to inputs sequentially for each subsequent module, finally returning the output of the last module.

The value a `Sequential` provides over manually calling a sequence of modules is that it allows treating the whole container as a single module, such that performing a transformation on the `Sequential` applies to each of the modules it stores (which are each a registered submodule of the `Sequential`).

What's the difference between a `Sequential` and a `torch.nn.ModuleList` ? A `ModuleList` is exactly what it sounds like–a list for storing `Module` s! On the other hand, the layers in a `Sequential` are connected in a cascading way.

Example:

```
# Using Sequential to create a small model. When `model` is run,
# input will first be passed to `Conv2d(1,20,5)`. The output of
# `Conv2d(1,20,5)` will be used as the input to the first
# `ReLU`; the output of the first `ReLU` will become the input
# for `Conv2d(20,64,5)`. Finally, the output of
# `Conv2d(20,64,5)` will be used as input to the second `ReLU`
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Using Sequential with OrderedDict. This is functionally the
# same as the above code
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

append(*module*) [SOURCE]

Append a given module to the end.

Parameters

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

Return type

Sequential

Previous Next

Appendix G: ReLU

Table of Contents

RELU

CLASS `torch.nn.ReLU(inplace=False)` [SOURCE]

Applies the rectified linear unit function element-wise:

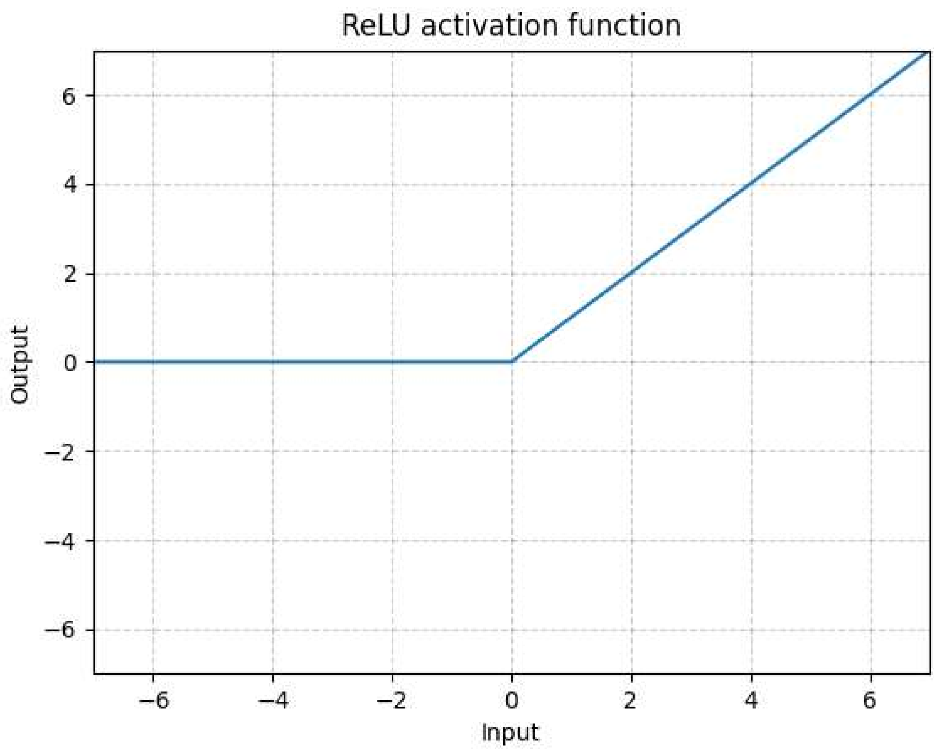
$\text{ReLU}(x) = (x)^+ = \max(0, x)$

Parameters

inplace – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.ReLU()
>>> input = torch.randn(2)
>>> output = m(input)


An implementation of CReLU - https://arxiv.org/abs/1603.05201

>>> m = nn.ReLU()
>>> input = torch.randn(2).unsqueeze(0)
>>> output = torch.cat((m(input),m(-input)))
```

Appendix H: Multihead Attention and Sinusoid Table

```
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)
```

```
def get_sinusoid_table(max_len, emb_dim):
    def get_angle(pos, i, emb_dim):
        return pos / 10000 ** ((2 * (i // 2)) / emb_dim)

    sinusoid_table = torch.zeros(max_len, emb_dim)
    for pos in range(max_len):
        for i in range(emb_dim):
            if i % 2 == 0:
                sinusoid_table[pos, i] = math.sin(get_angle(pos, i, emb_dim))
            else:
                sinusoid_table[pos, i] = math.cos(get_angle(pos, i, emb_dim))
    return sinusoid_table
```