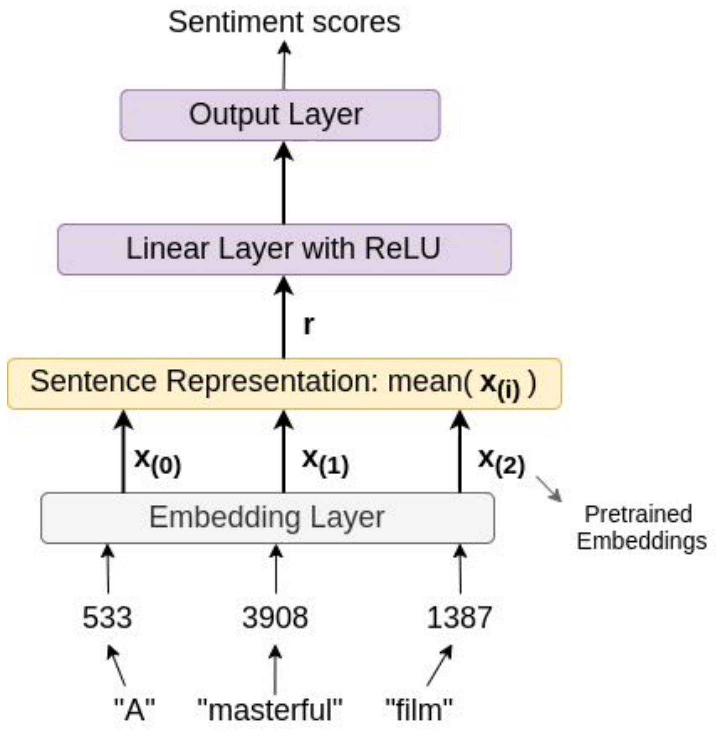
SLP-Lab3

NLP-RNN-LSTM-Attention-
Transfer Learning

Preparation Exercise - Architecture



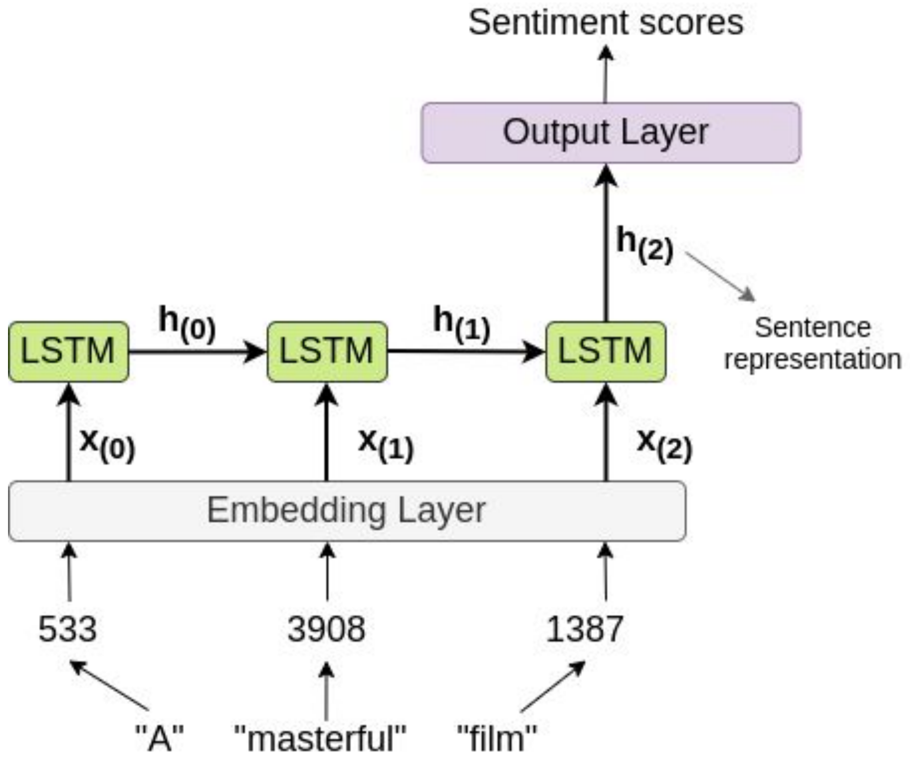
- **Task**: Sentiment Analysis of Movie Reviews or Tweets
- **Embedding Layer**: Each word is initially represented by its ID (e.g. 1387 for the word “film”). The *embedding layer* returns the word embedding.
- **Pretrained word embeddings**: We download and use word vectors trained on huge datasets (eg. GloVe embeddings).

Preparation exercise - Implementation

```python
class DNN(nn.Module):
    ...
    def forward(self, x, lengths):
        embeddings = self.embeddings(x)

        representations = torch.sum(embeddings,dim=1)
        for i in range(lengths.shape[0]):
            representations[i] = representations[i] / lengths[i]

        representations =self.relu(self.linear(representations))

        logits = self.output(representations)
        return logits
```

```python
def train_dataset(_epoch, dataloader, model, loss_function, optimizer):
   model.train()
   running_loss =0.0
   device = next(model.parameters()).device

   for index, batch in enumerate(dataloader, 1):
       inputs, labels, lengths = batch
       inputs, labels, lengths = inputs.to(device), labels.to(device), lengths.to(device)

       model.zero_grad()
       outputs = model(inputs, lengths)

       loss = loss_function(outputs, labels)
       loss.backward()
       optimizer.step()

       running_loss += loss.data.item()
       ...
```

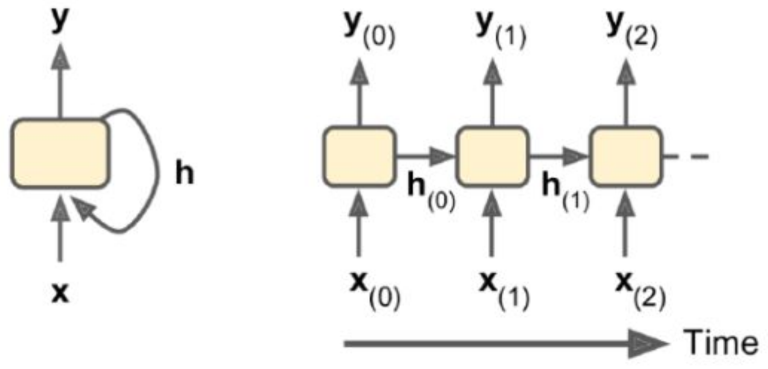
Lab exercise - Architecture (LSTM)



Differences:

- We feed the word embeddings to an **LSTM** model**.**
- We use as the **Sentence representation** the final hidden state of the model.
- (Minor) We don't use a *Linear Layer with ReLU* before the Output one (but we could).

RNNs

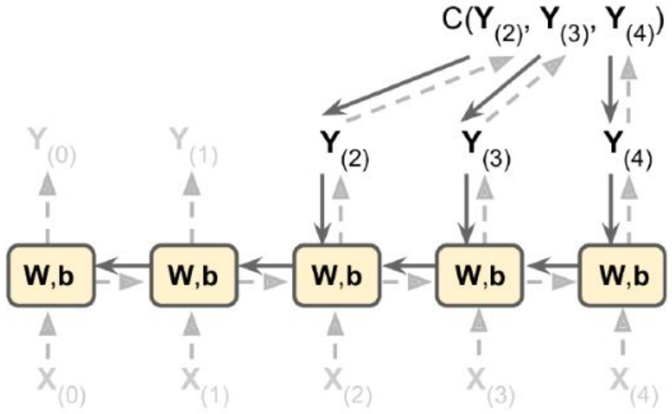


- **RNNs** can work on **sequences of arbitrary lengths.**
- They have a form of **memory** since the *output* of a recurrent neuron at each time step is a *function of all the inputs from previous timesteps*.
- The part that preserves some state across time steps is called a **memory cell** (or simply a cell).

Difficulties that RNN face:

- **Unstable gradients problem**: it may take forever to train, or training may be unstable.
- **A limited short-term memory**: when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence.

Training RNNs

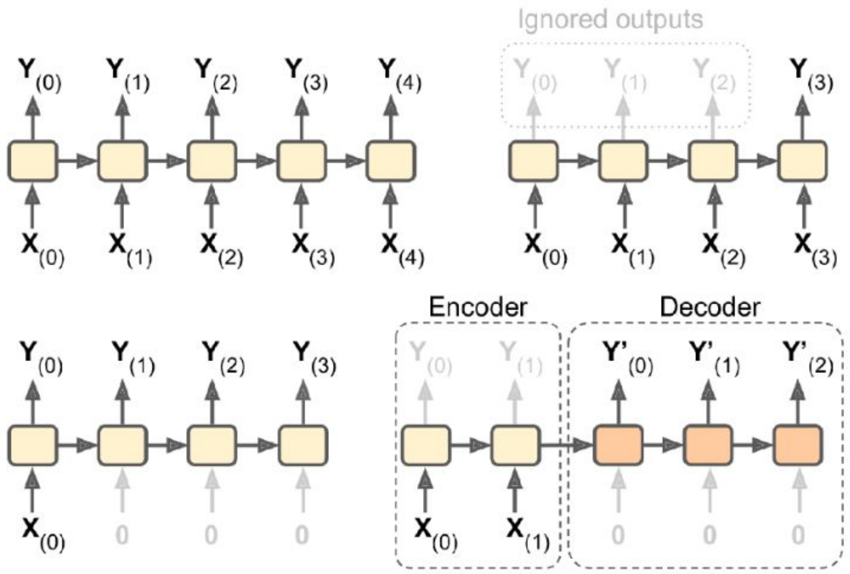


- Note that the cost function may ignore some outputs depending on the *type* of the RNN.

- **Unroll RNN through time** and then simply use regular backpropagation. This strategy is called **backpropagation through time** (BPTT).

- The *gradients* of the cost function (C) are *propagated backward through the unrolled network* (represented by the solid arrows).

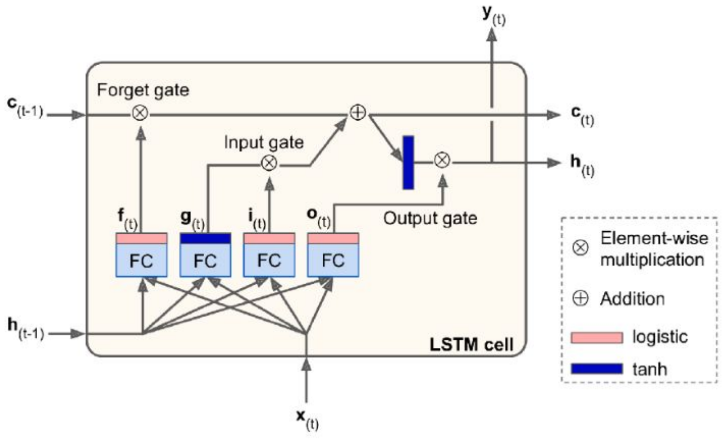
Types of RNNs



Ignored outputs
$Y_{(0)}$ $Y_{(1)}$ $Y_{(2)}$ $Y_{(3)}$ $Y_{(4)}$
$X_{(0)}$ $X_{(1)}$ $X_{(2)}$ $X_{(3)}$ $X_{(4)}$
Encoder
Decoder
$Y'_{(0)}$ $Y'_{(1)}$ $Y'_{(2)}$
0

- **Sequence-to-Sequence** (top left), eg. *translate an english sentence to french*
- **Sequence-to-Vector** (top right), eg. *create a text representation* - our task
- **Vector-to-Sequence** (bottom left), eg. *create a caption using an image as the input*
- **Encoder-Decoder** (bottom right) is also a Seq-to-Seq model that consists of a *Sequence-to-Vector* followed by a *Vector-to-Sequence*

LSTM cells (1/2)



- **LSTM state** is split into two vectors: short-term state **h**(t) and long-term state **c**(t).

- The **main layer** that is equivalent to the Simple RNN cell is the one that outputs **g**(t). In contrast, here, this layer's output *does not go straight out*, but instead its **most important parts are stored in the long-term state** (and the rest is dropped).

The three other layers are **gate controllers**. Since they use the logistic activation function, their outputs range from 0 (close the gate) to 1 (open it):

- **forget gate**, **f**(t), controls which parts of the long-term state should be erased
- **input gate**, **i**(t), controls which parts of **g**(t) should be added to the long-term state
- **output gate**, **o**(t), controls which parts of the long-term state should be read and output at this time step

LSTM cells (2/2)

In short, an LSTM cell can learn to:

- recognize an important input and store it in the long-term state (that’s the role of the input gate),
- preserve it for as long as it is needed (that’s the role of the forget gate),
- and extract it whenever it is needed (using the output gate).

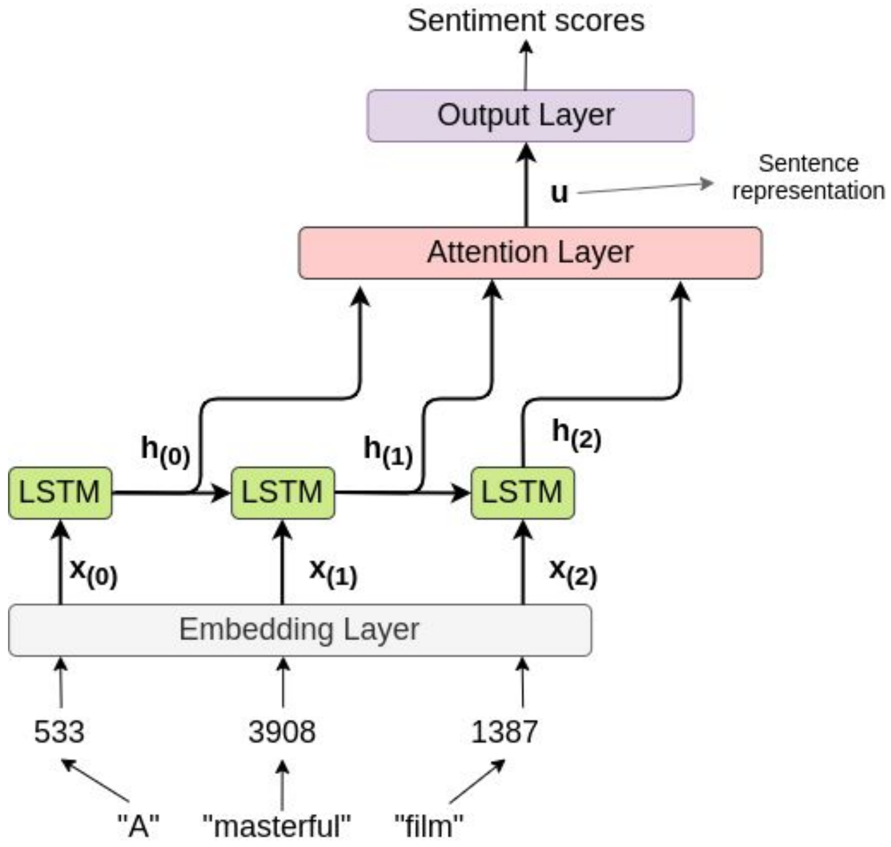
Bidirectional LSTMs

- A each time step, a *regular LSTM* (and RNN in general) only looks at *past and present inputs* before generating its output. In other words, it is **“causal”**, meaning it cannot look into the future. This type of model makes sense when forecasting time series, but **for many NLP tasks**, such as Neural Machine Translation, it is often **preferable to look ahead at the next words** before encoding a given word
- To implement this, we run **two LSTMs on the same inputs**, **one** reading the words **from left to right** and the **other** reading them **from right to left**. Then, we simply **combine their outputs at each timestep**, typically by **concatenating** them. This is called a **bidirectional LSTM**.

LSTM implementation (example)

```
class LSTM(nn.Module):
    ...
    def forward(self, x, lengths):
        batch_size, max_length = x.shape
        embeddings = self.embeddings(x)

        X = torch.nn.utils.rnn.pack_padded_sequence(embeddings, lengths, batch_first=True, enforce_sorted=False)
        ht, _ = self.lstm(X)
        ht, _ = torch.nn.utils.rnn.pad_packed_sequence(ht, batch_first=True)

        # Sentence representation as the final hidden state of the model
        representations = torch.zeros(batch_size, self.hidden_size).float()
        for i in range(lengths.shape[0]):
            last = lengths[i] - 1 if lengths[i] <= max_length else max_length - 1
            representations[i] = ht[i, last, :]

        logits = self.linear(representations)

        return logits
```

Lab exercise - Architecture (LSTM-Attention)



Differences:

- We feed the encoded words by the **LSTM** (**h**(0), **h**(1), ..., **h**(N)) to an Attention Layer**.**
- Each **Sentence representation** is a weighted sum of all LSTM *hidden states* (that carry the “meaning” of each word) These weights are computed at the *Attention layer*.
- So, for a sentence we get:

$$\mathbf{u} = \Sigma_i(\alpha_i \cdot \mathbf{h}_{(i)})$$
where $\Sigma_i \alpha_i = 1$

Attention

Attention mechanism revolutionized NLP, because of its ability to identify the information in an input which is the most relevant to accomplish a task.

- It was introduced by the authors of the 2014 paper *"Neural Machine Translation by Jointly Learning to Align and Translate"*, Bahdanau et al., as a **technique that allowed the decoder to focus on the appropriate words** at each time step. For example, at the time step where the decoder needs to output the word “lait”, it will focus its attention on the word “milk”. This means that the path from an input word to its translation was much shorter, so the short-term memory limitations of RNNs had much less impact.

Attention Is All You Need: The Transformer Architecture

- In a groundbreaking 2017 paper, Vaswani et al. suggested that *“Attention Is All You Need”*. They managed to create an architecture called the **Transformer**, which significantly improved the state of the art in NMT without using any recurrent or convolutional layers, **just attention mechanisms** (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). Moreover, this architecture was also much faster to train and easier to parallelize, so they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

LSTM-Attention implementation (example)

```
class LSTMAttention(nn.Module):
    ...
    def forward(self, x, lengths):
       batch_size, max_length = x.shape
       embeddings = self.embeddings(x)

       X = torch.nn.utils.rnn.pack_padded_sequence(embeddings, lengths,batch_first=True, enforce_sorted=False)
       ht, _ = self.lstm(X)
       ht, _ = torch.nn.utils.rnn.pad_packed_sequence(ht,batch_first=True)

       # apply attention to get Sentence representation
       representations, att_scores = self.attention(ht, lengths)

       logits = self.linear(representations)

       return logits
```

** you can use the already developed implementation of Attention here.*

Transfer Learning (bonus)

Transfer learning refers to a **technique for predictive modeling** on a different (but somehow similar) problem that can then be **reused partly or wholly** to accelerate the training and improve the performance of a model on the problem of interest.
In *deep learning*, this means **reusing the weights in one or more layers from a pre-trained network model in a new model** and either keeping the weights fixed, fine tuning them, or adapting the weights entirely when training the model.

Example implementation:

```
# target model
model = LSTMRegression(embeddings, bidirectional=True)

# load source model
source_model_path = os.path.join(SAVE_PATH, "Semeval2017A_LSTMAttention.pt")
source_model = torch.load(source_model_path)

# remove weights of final layer
del source_model['linear.weight']
del source_model['linear.bias']

# initialize target model with the state of source model
model.load_state_dict(source_model, strict=False)

...
```