Attention and Transformers
Recap: Inference and Applications

1. Idea
2. PM
3. Collecting Data
4. Train Model
5. Inference
6. Deploy
SpeakUp

Join by URL: https://web.speakup.info/room/join/80740

Join by key: web.speakup.info → use key 80740

Join by saved session: web.speakup.info

Join by QR:
Outline

• Remember the RNN encoder/decoder architecture

• Motivate the need for attention

• Explore the transformer architecture
Translation Encoder/Decoder RNN

Encoder

Decoder

$x_t$ → Embedding Layer
Embedding Space

Cross-domain sentiment aware word embeddings for review sentiment analysis

Jun Liu¹ • Shuang Zheng¹ • Guangxu Xu¹,²,³ • Mingwei Lin¹
Translation Encoder/Decoder RNN
Translation Encoder/Decoder RNN
Translation Encoder/Decoder RNN

Unrolled Version

Encoder

Decoder
Bottleneck. How do we know which parts of the context are important?

RNN are also slow to train.
The agreement on the European Economic Area was signed in August 1992.

Let's talk about attention in terms of transformers.
Transformer Model Introduced In...

Attention Is All You Need

No recurrence. No convolutions.
Better output. Faster to train.
A Lot Happened All at Once

Attention is All you Need
Part of Advances in Neural Information Processing Systems 30 (NIPS 2017)

https://huggingface.co/docs/transformers/model_summary
Coreference Resolution

The animal didn’t cross the street because it was too tired.
L’animal n’a pas traversé la rue parce qu’il était trop fatigué.

The animal didn’t cross the street because it was too wide.
L’animal n’a pas traversé la rue parce qu’elle était trop large.
Common Transformer Applications

• Audio
  • Classification
  • Speech to text

• Vision
  • Classification
  • Object detection
  • Segmentation
  • Depth estimation
  • Captioning

• Text (NLP)
  • Classification
  • Question answering
  • Summarization
  • Translation
  • Named entity recognition
  • Language modeling
    • Masked (fill in the blank)
    • Generation
    • Completion
Example Application: Dialogue Completer

“But that is not for them to decide. All we have to decide is what to do with the time that is given us.”

“But that is not for them to decide. All we have to decide is what to do with the time that is given us.”

INPUT: But that is not for them to decide. All we have
OUTPUT: to decide is what to do with the time that is given us.
But that is not for them to decide. All we have to decide is what to do with the time that is given us.
def make_transformer(
    src_vocab_size: int,  
    tgt_vocab_size: int,  
    d_model: int = 512,  
    num_head: int = 8,  
    num_layers: int = 6,  
    d_feedforward: int = 2048,  
    dropout_prob: float = 0.1,  
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),  
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),  
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),  
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),  
        Generator(d_model, tgt_vocab_size),
    )

    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return model

https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py
Adapted from: The Annotated Transformer
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )

    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return model

https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py
Adapted from: The Annotated Transformer
PositionalEmbedding

class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout_prob: float, max_len: int):
        super().__init__()
        self.dropout = nn.Dropout(dropout_prob)
        # Compute the positional encodings once in log space
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * -(log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)

class PositionalEmbedding(nn.Module):
    "Scaled embedding followed by a positional encoding."

    def __init__(self, vocab_size: int, d_model: int, dropout_prob: float, max_len: int):
        super().__init__()
        self.embed = ScaledEmbedding(d_model, vocab_size)
        self.positional_encoding = PositionalEncoding(d_model, dropout_prob, max_len)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.positional_encoding(self.embed(x))
#### PositionalEmbedding

<table>
<thead>
<tr>
<th>Index</th>
<th>Embed + Pos. Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>I → 1</td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>am → 2</td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>a → 3</td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Student → 14</td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Hyperparameter

\[ p_i = \sin \left( \frac{i}{10000 \sqrt{\theta}} \right) \quad \text{∀ even numbers} \]

\[ p_i = \cos \left( \frac{i}{10000 \sqrt{\theta}} \right) \quad \text{∀ odd numbers} \]
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )

    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return model

https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py
Adapted from: The Annotated Transformer
class EncoderLayer(nn.Module):
    "An encoder layer is composed of self-attention and feed forward components."
    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = (MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)
        def forward(self, x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
            x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, mask))
            return self.sublayer2(x, self.feed_forward)

class Encoder(nn.Module):
    "An encoder block comprising N encoder layers."
    def __init__(self, d_model: int, num_head: int, num_layers: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_layers):
            self.layers.append(EncoderLayer(d_model, num_head, ff_dim, dropout_prob))
        self.norm = nn.LayerNorm(d_model)

def forward(self, x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
    for layer in self.layers:
        x = layer(x, mask)
    return self.norm(x)
The animal didn’t cross the street because it was too tired.
L’animal n’a pas traverse la rue parce qu’il était trop fatigue.

The animal didn’t cross the street because it was too wide.
L’animal n’a pas traverse la rue parce qu’elle était trop large.
Self-Attention

• Self-attention processes the entire sequence at once (not one at a time like an RNN)
MultiHeadedAttention

Scaled Dot-Product Attention

Multi-Head Attention

\[ h \]
class PositionWiseFeedForward(nn.Module):
    """Implements feedforward neural network equation."""

def __init__(self, d_model: int, ff_dim: int, dropout_prob: float):
    super().__init__()
    self.w1 = nn.Linear(d_model, ff_dim)
    self.w2 = nn.Linear(ff_dim, d_model)
    self.dropout = nn.Dropout(dropout_prob)

def forward(self, x: torch.Tensor) -> torch.Tensor:
    return self.w2(self.dropout(self.w1(x).relu()))
class SublayerConnection(nn.Module):
    "A residual connection followed by a LayerNorm."

    def __init__(self, d_model: int, dropout_prob: float):
        super().__init__()
        self.layernorm = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, x: torch.Tensor, sublayer: nn.Module) -> torch.Tensor:
        return x + self.dropout(sublayer(self.layernorm(x)))

class EncoderLayer(nn.Module):
    "An encoder layer is composed of self-attention and feed forward components."

    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)

    def forward(self, x: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, mask))
        return self.sublayer2(x, self.feed_forward)
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return model
class DecoderLayer(nn.Module):
    "A decode layer comprises self and source attention and feed forward components."
    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.src_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)
        self.sublayer3 = SublayerConnection(d_model, dropout_prob)

        m = memory
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, tgt_mask))
        x = self.sublayer2(x, lambda x: self.src_attention(x, m, m, src_mask))
        return self.sublayer3(x, self.feed_forward)

class Decoder(nn.Module):
    "A decoder block comprising N layers with masking."
    def __init__(self, d_model: int, num_head: int, num_layers: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_layers):
            self.layers.append(DecoderLayer(d_model, num_head, ff_dim, dropout_prob))

        self.norm = nn.LayerNorm(d_model)

        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
class DecoderLayer(nn.Module):
    "A decode layer comprises self and source attention and feed forward components."
    def __init__(self, d_model: int, num_head: int, ff_dim: int, dropout_prob: float):
        super().__init__()
        self.self_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.src_attention = MultiHeadedAttention(d_model, num_head, dropout_prob)
        self.feed_forward = PositionWiseFeedForward(d_model, ff_dim, dropout_prob)
        self.sublayer1 = SublayerConnection(d_model, dropout_prob)
        self.sublayer2 = SublayerConnection(d_model, dropout_prob)
        self.sublayer3 = SublayerConnection(d_model, dropout_prob)

        m = memory
        x = self.sublayer1(x, lambda x: self.self_attention(x, x, x, tgt_mask))
        x = self.sublayer2(x, lambda x: self.src_attention(x, m, m, src_mask))
        return self.sublayer3(x, self.feed_forward)

class Decoder(nn.Module):
    "A decoder block comprising N layers with masking."
    def __init__(self, d_model: int, num_head: int, num_layers: int, ff_dim: int, dropout):
        super().__init__()
        self.layers = nn.ModuleList()
        for _ in range(num_layers):
            self.layers.append(DecoderLayer(d_model, num_head, ff_dim, dropout))
        self.norm = nn.LayerNorm(d_model)

        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
INPUT: <START>But that is not for them to decide. All we have<END>

OUTPUT: <START>to decide is what to do with the time that is given us.<END>
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model

https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py
Adapted from: The Annotated Transformer
class Generator(nn.Module):
    "Define standard linear + softmax generation step."

    def __init__(self, d_model: int, vocab_size: int):
        super().__init__()
        self.linear = nn.Linear(d_model, vocab_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return log_softmax(self.linear(x), dim=-1)
def make_transformer(
    src_vocab_size: int,
    tgt_vocab_size: int,
    d_model: int = 512,
    num_head: int = 8,
    num_layers: int = 6,
    d_feedforward: int = 2048,
    dropout_prob: float = 0.1,
    max_len: int = 5000,
):
    model = Transformer(
        PositionalEmbedding(src_vocab_size, d_model, dropout_prob, max_len),
        Encoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        PositionalEmbedding(tgt_vocab_size, d_model, dropout_prob, max_len),
        Decoder(d_model, num_head, num_layers, d_feedforward, dropout_prob),
        Generator(d_model, tgt_vocab_size),
    )

    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return model

https://github.com/anthonyjclark/cs152sp23/blob/main/Lectures/13-Transformer.py
Adapted from: The Annotated Transformer
Summary

• Recurrent neural networks were the best performing for sequence tasks.

• They have an advantage of arbitrary input and output sequence lengths.

• Transformers have fixed input and output sequence lengths (with padding).

• But, transformers outperform RNNs on (probably) all sequence tasks.

• Transformers do not maintain any internal memory.

• Most (if not all) large language models (LLMs) are built on transformers.
• **Visualizing A Neural Machine Translation Model**
• **Transformer: A Novel Neural Network Architecture for Language Understanding**
• **A Visual Guide to Transformers Neural Networks: step by step explanation - YouTube**
• **L19.4.1 Using Attention Without the RNN -- A Basic Form of Self-Attention - YouTube**
• **karpathy/nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs.**
• **karpathy/minGPT: A minimal PyTorch re-implementation of the OpenAI GPT (Generative Pretrained Transformer) training**
• **karpathy/minGPT: A minimal PyTorch re-implementation of the OpenAI GPT (Generative Pretrained Transformer) training**
• **The Annotated Transformer**