Position-Wise Feed-Forward Network (FFN)

Hunter Phillips · Following

9 min read · May 9

This is the fourth article in The Implemented Transformer series. The Position-wise Feed-Forward Network is an expand-and-contract network that transforms each sequence using the same dense layers.

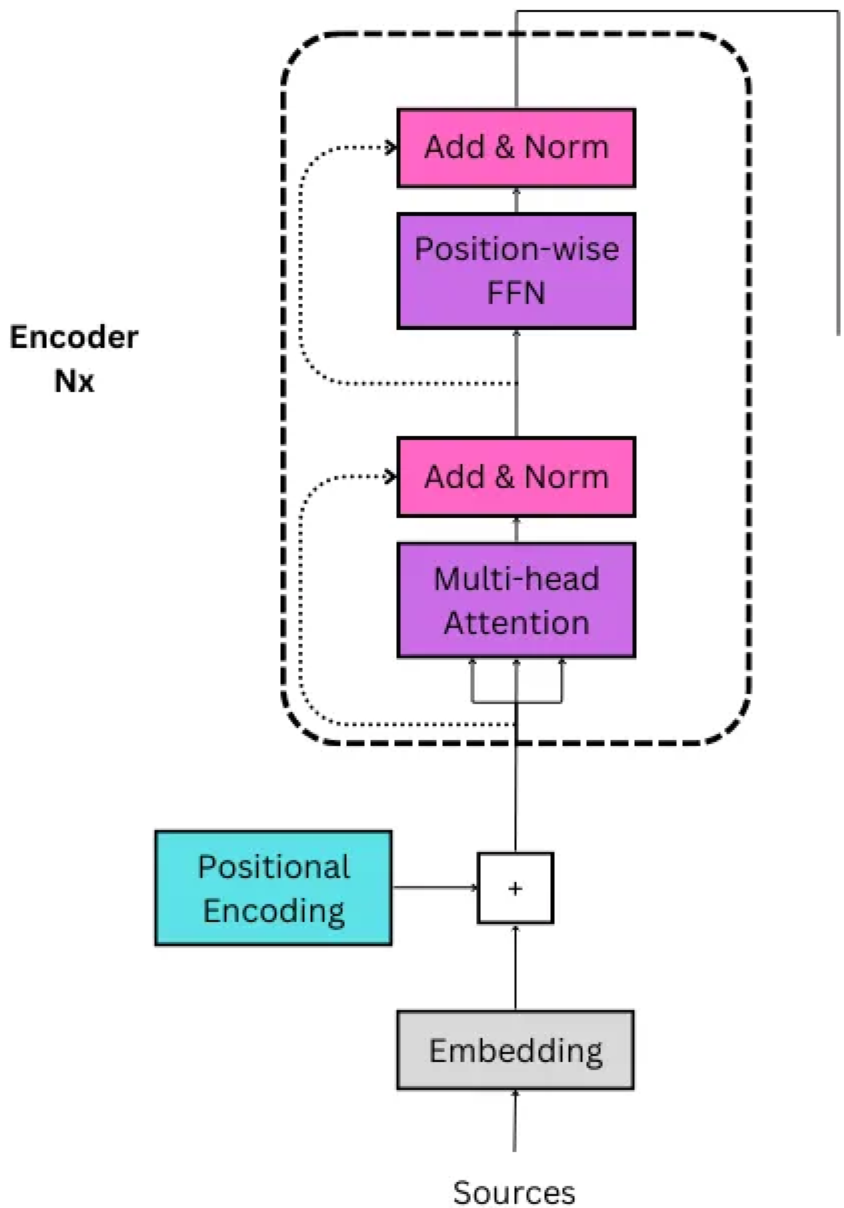


Image by Author

Background

The Position-Wise Feed-Forward Network (FFN) consists of two fully connected dense layers, or a multi-layer perceptron (MLP). The hidden layer, which is known as ***d_ffn,*** is generally set to a value about four times that of ***d_model***. This is why it is sometimes known as an expand-and-contract network.

According to d2l, it is known as a "position-wise" network because it "transforms the representation at all the sequence positions using the same MLP."

In other words, the FNN has a size of ***(d_model, d_ffn)*** in its first layer, which means it has to be broadcast across each sequence during tensor multiplication. This means each sequence is multiplied by the same weights. If identical sequences are input, the outputs will also be identical. This same logic applies to the second dense layer of size ***(d_ffn, d_model),*** which returns the tensor to its original size.

The ReLU activiation function, ***max(0, X)***, is used between the layers. Any values greater than 0 remain the same, and any values less than or equal to 0 become 0. It introduces non-linearity and helps prevent vanishing gradients.

Basic Implementation

The following code relies on the previous modules of the transformers model. The output of the layers up to this point is *(3,6,8)*. There are 3 sequences of 6 tokens with 8 dimensional embeddings.

```
torch.set_printoptions(precision=2, sci_mode=False)

# convert the sequences to integers
sequences = ["I wonder what will come next!",
```

```
             "This is a basic example paragraph.",
             "Hello, what is a basic split?"]

# tokenize the sequences
tokenized_sequences = [tokenize(seq) for seq in sequences]

# index the sequences
indexed_sequences = [[stoi[word] for word in seq] for seq in tokenized_sequences

# convert the sequences to a tensor
tensor_sequences = torch.tensor(indexed_sequences).long()

# vocab size
vocab_size = len(stoi)

# embedding dimensions
d_model = 8

# create the embeddings
lut = Embeddings(vocab_size, d_model) # look-up table (lut)

# create the positional encodings
pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=10)

# embed the sequence
embeddings = lut(tensor_sequences)

# positionally encode the sequences
X = pe(embeddings)

# set the n_heads
n_heads = 4

# create the attention layer
attention = MultiHeadAttention(d_model, n_heads, dropout=0.1)

# pass X through the attention layer three times to create Q, K, and V
output, attn_probs = attention(X, X, X, mask=None)

output
```

```
tensor([[[-0.87, -1.03,  0.92, -0.21,  0.85,  0.80,  0.18, -0.67],
         [-1.10, -1.13,  1.03, -0.11,  0.88,  0.59, -0.03, -0.51],
         [-0.68, -1.29,  0.76, -0.47,  0.70,  0.89, -0.25, -0.42],
         [-0.21, -0.92,  0.51, -0.52,  0.43,  0.87, -0.12,  0.21],
         [-1.23, -1.60,  1.36,  0.31,  0.69, -0.22, -1.17, -1.49],
```

```
         [-1.13, -1.68,  1.35,  0.23,  0.66, -0.08, -1.23, -1.21]],

        [[-1.12, -0.46,  0.34, -0.13,  0.57, -0.23,  0.33, -0.58],
         [-1.09, -0.01,  0.31,  0.06,  0.06, -0.51,  0.68, -0.44],
         [-0.86,  0.09,  0.17, -0.27, -0.26, -0.24, -0.14,  0.32],
         [ 0.88,  1.22,  0.14,  0.32, -0.97, -0.50, -0.47, -0.56],
         [-0.49, -0.62, -0.46, -0.70, -0.10, -0.52, -0.45, -1.04],
         [-0.06,  0.09,  0.48, -0.15, -0.29,  0.28, -0.62, -0.16]],

        [[-2.44, -3.36, -0.28, -1.39,  1.83, -0.52, -1.41, -1.68],
         [ 0.00, -1.52, -0.39, -0.81, -0.15, -0.53, -1.37, -2.45],
         [ 0.52, -0.80,  0.54, -0.31, -0.46,  0.41, -1.35, -1.54],
         [-0.37, -1.55, -0.22, -1.01, -0.09, -0.19, -2.06, -1.40],
         [ 1.10,  0.80,  0.11, -0.04, -0.76,  0.22, -0.20, -1.63],
         [ 1.21,  0.05,  0.07, -0.26, -0.75,  0.20, -0.93, -2.64]]],
      grad_fn=<ViewBackward0>)
```

Now, this can be passed through the FFN. This will change the 8-dimensional embeddings to 32-dimensional embeddings. This is also passed through the ReLU activation function. The new shape will be ***(3, 6, 8) x (8, 32) → (3, 6, 32)***.

```
d_ffn = d_model * 4  # 32

w_1 = nn.Linear(d_model, d_ffn)  # (8, 32)
w_2 = nn.Linear(d_ffn, d_model)  # (32, 8)

ffn_1 = w_1(output).relu()
```

```
tensor([
        # sequence 0
        [[    0.00,     0.00,     0.58,     0.00,     0.86,     0.00,     0.00,
         [    0.00,     0.00,     0.62,     0.00,     0.90,     0.00,     0.00,
         [    0.00,     0.00,     0.28,     0.00,     0.81,     0.00,     0.00,
         [    0.06,     0.00,     0.11,     0.00,     0.60,     0.00,     0.00,
         [    0.00,     0.12,     0.40,     0.00,     0.63,     0.00,     0.00,
         [    0.00,     0.13,     0.29,     0.00,     0.67,     0.00,     0.00,

        # sequence 1
```

```
        [[    0.00,     0.00,     0.89,     0.00,     0.51,     0.00,     0.00,
         [    0.00,     0.00,     0.96,     0.00,     0.39,     0.00,     0.00,
         [    0.07,     0.00,     0.56,     0.00,     0.22,     0.00,     0.00,
         [    0.68,     0.00,     0.01,     0.31,     0.00,     0.18,     0.00,
         [    0.00,     0.00,     0.31,     0.32,     0.00,     0.00,     0.00,
         [    0.24,     0.00,     0.12,     0.00,     0.00,     0.00,     0.00,

        # sequence 2
        [[    0.00,     1.00,     0.67,     0.07,     1.18,     0.00,     0.00,
         [    0.00,     0.10,     0.00,     0.68,     0.00,     0.00,     0.42,
         [    0.00,     0.00,     0.00,     0.14,     0.00,     0.00,     0.30,
         [    0.00,     0.08,     0.00,     0.22,     0.00,     0.00,     0.45,
         [    0.35,     0.00,     0.00,     0.55,     0.00,     0.13,     0.03,
         [    0.01,     0.00,     0.00,     0.79,     0.00,     0.01,     0.42,
```

Then, the tensor can be passed through the second dense layer to return to its normal size, *(3, 6, 32) x (32, 8) = (3, 6, 8)*. The values have changed according to the weights and activation function.

```
ffn_2 = w_2(ffn_1)
ffn_2
```

```
tensor([[[ 0.09,  0.37, -0.20, -0.12,  0.30, -0.24,  0.13,  0.33],
         [ 0.13,  0.30, -0.23, -0.22,  0.25, -0.38,  0.02,  0.22],
         [ 0.05,  0.35, -0.11, -0.20,  0.32, -0.26,  0.16,  0.37],
         [ 0.08,  0.28, -0.04, -0.30,  0.34, -0.15,  0.18,  0.24],
         [ 0.08,  0.27, -0.43, -0.28,  0.43, -0.05, -0.03,  0.24],
         [ 0.14,  0.23, -0.30, -0.22,  0.19, -0.39, -0.04,  0.30]],

        [[ 0.18,  0.43, -0.23, -0.05,  0.22, -0.15,  0.13,  0.11],
         [ 0.13,  0.41, -0.20, -0.13,  0.10, -0.04,  0.10,  0.01],
         [ 0.08,  0.31, -0.08, -0.09,  0.13, -0.13,  0.16,  0.08],
         [-0.20,  0.18, -0.20,  0.21,  0.19, -0.00,  0.26,  0.25],
         [ 0.17,  0.21, -0.08, -0.07,  0.10, -0.01,  0.09,  0.08],
         [-0.02,  0.25, -0.08, -0.11,  0.23, -0.12,  0.19,  0.23]],

        [[-0.21,  0.71, -0.29, -0.08,  0.21,  0.12, -0.33,  0.49],
         [ 0.19,  0.19, -0.06,  0.19,  0.33,  0.02,  0.25,  0.29],
```

```
         [-0.04,  0.25, -0.10, -0.12,  0.21, -0.11,  0.19,  0.42],
         [ 0.05,  0.35, -0.02,  0.05,  0.27, -0.06,  0.17,  0.29],
         [-0.16,  0.22, -0.13,  0.15,  0.13,  0.00,  0.31,  0.34],
         [-0.05,  0.23, -0.18,  0.22,  0.13, -0.03,  0.26,  0.40]]],
       grad_fn=<ViewBackward0>)
```

FFN in Transformers

$$FFN(x, W_1, W_2, b_1, b_2) = \max(0, xW_1 + b_1)W_2 + b_2$$

The implementation for the FFN is straightforward. It requires two linear, or dense, layers. The first dense layer has a size of ***(d_model, d_ffn)***, and the second layer has a size of ***(d_ffn, d_model)***.

The input to the model, ***X***, will have a size of ***(batch_size, seq_length, d_model)***. The input will therefore go through the following transformations:

1. ***(batch_size, seq_length, d_model) x (d_model, d_ffn) = (batch_size, seq_length, d_ffn)***
2. ***max(0, (batch_size, seq_length, d_ffn)) = (batch_size, seq_length, d_ffn)***
3. ***(batch_size, seq_length, d_ffn) x (d_ffn, d_model) = (batch_size, seq_length, d_model)***

```
class PositionwiseFeedForward(nn.Module):
  def __init__(self, d_model: int, d_ffn: int, dropout: float = 0.1):
    """
    Args:
        d_model:      dimension of embeddings
```

```
        d_ffn:        dimension of feed-forward network
        dropout:      probability of dropout occurring
    """
    super().__init__()

    self.w_1 = nn.Linear(d_model, d_ffn)
    self.w_2 = nn.Linear(d_ffn, d_model)
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    """
    Args:
        x:            output from attention (batch_size, seq_length, d_model)

    Returns:
        expanded-and-contracted representation (batch_size, seq_length, d_model)
    """
    # w_1(x).relu(): (batch_size, seq_length, d_model) x (d_model,d_ffn) -> (bat
    # w_2(w_1(x).relu()): (batch_size, seq_length, d_ffn) x (d_ffn, d_model) ->
    return self.w_2(self.dropout(self.w_1(x).relu()))
```

Forward Pass

This forward pass assumes the data has been passed through the embedding, positional encoding, and multi-head attention layers. It does not use layer normalization or residual addition, which will be implemented before and after this network in the Encoder in article six.

```
torch.set_printoptions(precision=2, sci_mode=False)

# convert the sequences to integers
sequences = ["I wonder what will come next!",
             "This is a basic example paragraph.",
             "Hello, what is a basic split?"]

# tokenize the sequences
tokenized_sequences = [tokenize(seq) for seq in sequences]

# index the sequences
```

```
indexed_sequences = [[stoi[word] for word in seq] for seq in tokenized_sequences

# convert the sequences to a tensor
tensor_sequences = torch.tensor(indexed_sequences).long()

# vocab size
vocab_size = len(stoi)

# embedding dimensions
d_model = 8

# create the embeddings
lut = Embeddings(vocab_size, d_model) # look-up table (lut)

# create the positional encodings
pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=10)

# embed the sequence
embeddings = lut(tensor_sequences)

# positionally encode the sequences
X = pe(embeddings)

# set the n_heads
n_heads = 4

# create the attention layer
attention = MultiHeadAttention(d_model, n_heads, dropout=0.1)

# pass X through the attention layer three times to create Q, K, and V
output, attn_probs = attention(X, X, X, mask=None)

# calculate the d_ffn
d_ffn = d_model*4 # 32

# pass the tensor through the position-wise feed-forward network
ffn = PositionwiseFeedForward(d_model, d_ffn, dropout=0.1)

ffn(output)
```

```
tensor([[[ 0.19, -0.09,  0.05,  0.34, -0.11,  0.24,  0.03, -0.01],
         [ 0.29, -0.23, -0.19,  0.23, -0.33,  0.02, -0.08,  0.17],
         [-0.01, -0.16, -0.21,  0.23, -0.11,  0.12,  0.09, -0.02],
         [ 0.04, -0.14, -0.50,  0.12, -0.39,  0.04,  0.21,  0.12],
         [ 0.25, -0.19, -0.07,  0.47, -0.27,  0.16, -0.00,  0.39],
         [ 0.21, -0.17, -0.01,  0.46, -0.30,  0.28,  0.05,  0.23]],
```

```
          [[ 0.11, -0.16, -0.34,  0.12, -0.25, -0.01,  0.06,  0.04],
           [ 0.03, -0.16, -0.39, -0.01, -0.35, -0.07,  0.18, -0.02],
           [-0.00, -0.29, -0.44,  0.13, -0.07,  0.08, -0.01,  0.23],
           [ 0.17, -0.33, -0.49, -0.14, -0.45, -0.18,  0.22, -0.10],
           [ 0.02, -0.20, -0.53,  0.18, -0.43,  0.07,  0.32,  0.13],
           [ 0.03, -0.19, -0.50,  0.15, -0.38,  0.15,  0.20,  0.27]],

          [[ 0.07, -0.38, -0.51, -0.05, -0.48,  0.15,  0.20,  0.16],
           [ 0.03, -0.40, -0.61, -0.16, -0.59,  0.16,  0.52,  0.13],
           [-0.02, -0.18, -0.32,  0.13, -0.19,  0.08, -0.01,  0.11],
           [ 0.08, -0.52, -0.53,  0.11, -0.37,  0.31,  0.26,  0.35],
           [-0.02, -0.12, -0.50,  0.07, -0.22,  0.03,  0.26, -0.01],
           [ 0.02, -0.11, -0.29,  0.17, -0.25,  0.04, -0.00,  0.14]]],
        grad_fn=<ViewBackward0>)
```

The output is similar to the manually implemented one, but the values are different due to the dropout layer being utilized. Otherwise, the same transformations occur.

The next article briefly covers Layer Normalization before it is implemented in article six.

Please don't forget to like and follow for more! :)

References

1. The Annotated Transformer
2. d2l's Transformer Implementation

Transformers | NLP | Machine Learning | Ffn | Neural Networks

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

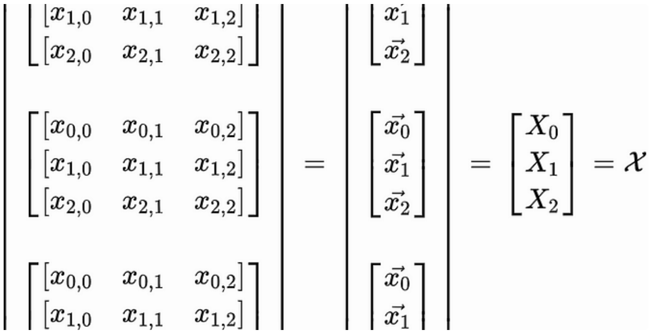
Written by Hunter Phillips

Following

219 Followers

Machine Learning Engineer and Data Scientist

More from Hunter Phillips



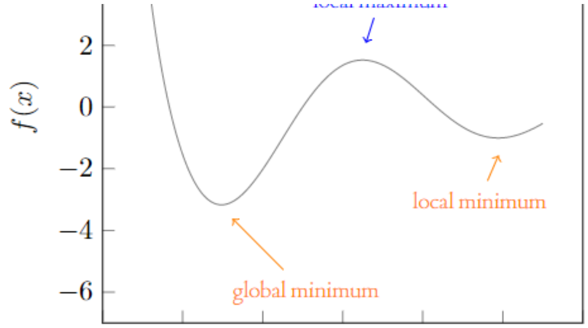
Hunter Phillips

A Simple Introduction to Tensors

A tensor is a generalization of vectors and matrices to n dimensions. Understanding ho…

11 min read · May 10

273 5



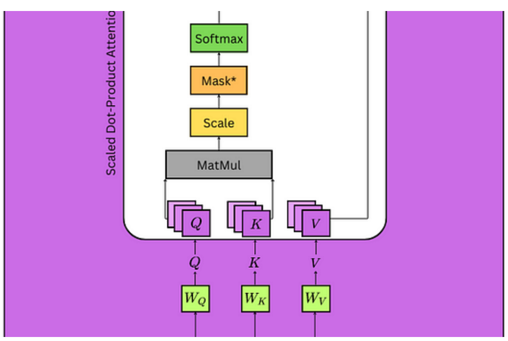
Hunter Phillips

A Simple Introduction to Gradient Descent

Gradient descent is one of the most common optimization algorithms in machine learning.…

10 min read · May 18

108



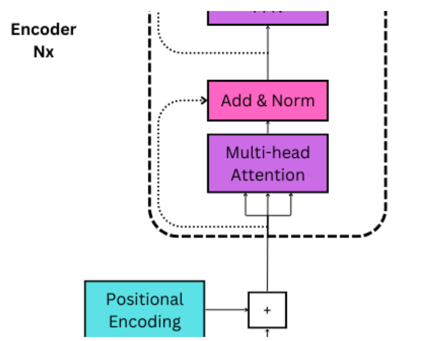
Hunter Phillips

Multi-Head Attention

This article is the third in The Implemented Transformer series. It introduces the multi-…

25 min read · May 9

167



Hunter Phillips

Layer Normalization

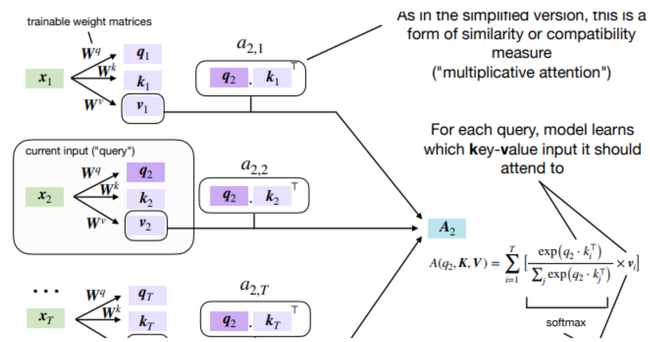
This is the fifth article in The Implemented Transformer series. Layer normalization…

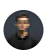
9 min read · May 9

160

See all from Hunter Phillips

Recommended from Medium





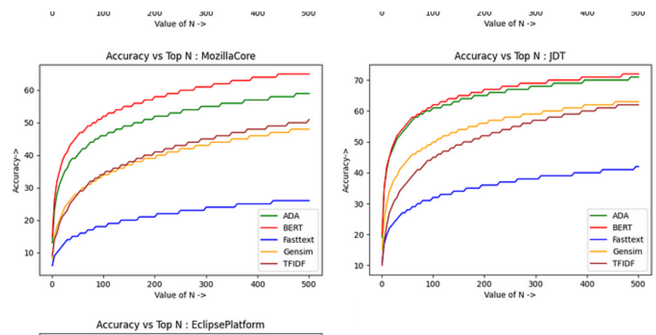
Zain ul Abideen

Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

6 min read · Jun 26

144



Avinash Patil

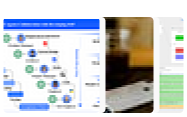
Embeddings: BERT better than ChatGPT4?

In this study, we compared the effectiveness of semantic textual similarity methods for…

4 min read · Sep 19

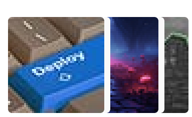
3 1

Lists



Natural Language Processing

669 stories · 283 saves



Predictive Modeling w/ Python

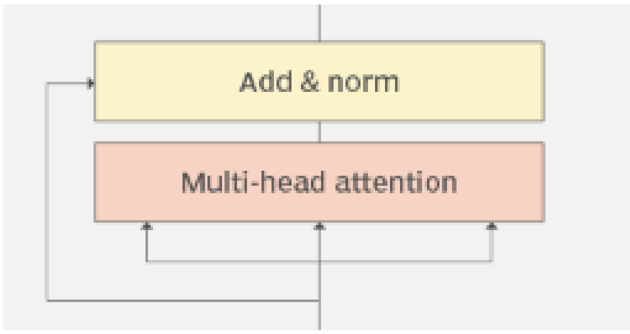
20 stories · 452 saves

Practical Guides to Machine Learning

10 stories · 519 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves



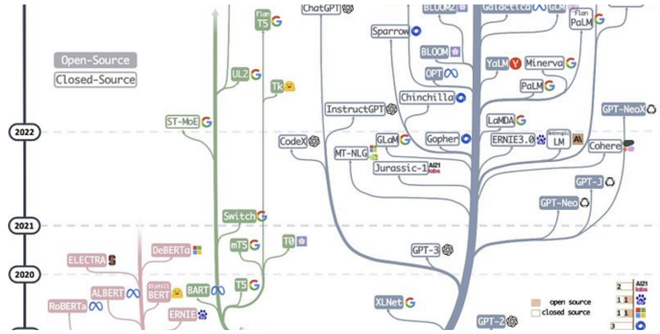
Eugene Ku

Transformer Architecture (Part 3— Scaling Self-Attention)

In part 2, we talked about how Self-Attention (Multi-Head Attention) takes attention…

8 min read · Aug 23

10



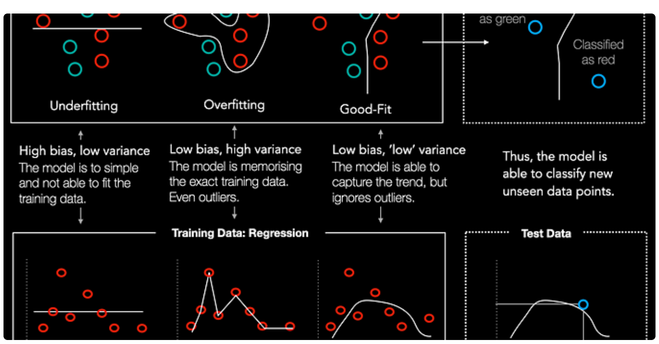
Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's…

15 min read · Sep 14

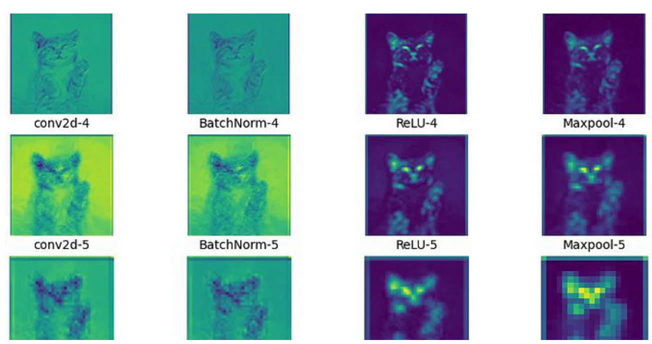
372



Frederik vl in Advanced Deep Learning

Understanding Bias and Variance in Machine Learning

The terms bias and variance describe how well the model fits the actual unknown data…



Sharath S Hebbar

Softmax for Intermediate CNN Layers

When you are writing an Image classification CNN model code you often don't think for a…

3 min read · Sep 15

44

3 min read · Sep 17

5

See more recommendations