# Layer Normalization

Hunter Phillips · Following
9 min read · May 9

This is the fifth article in The Implemented Transformer series. Layer normalization directly follows the multi-head attention mechanism and the position-wise feed-forward network from the previous articles.

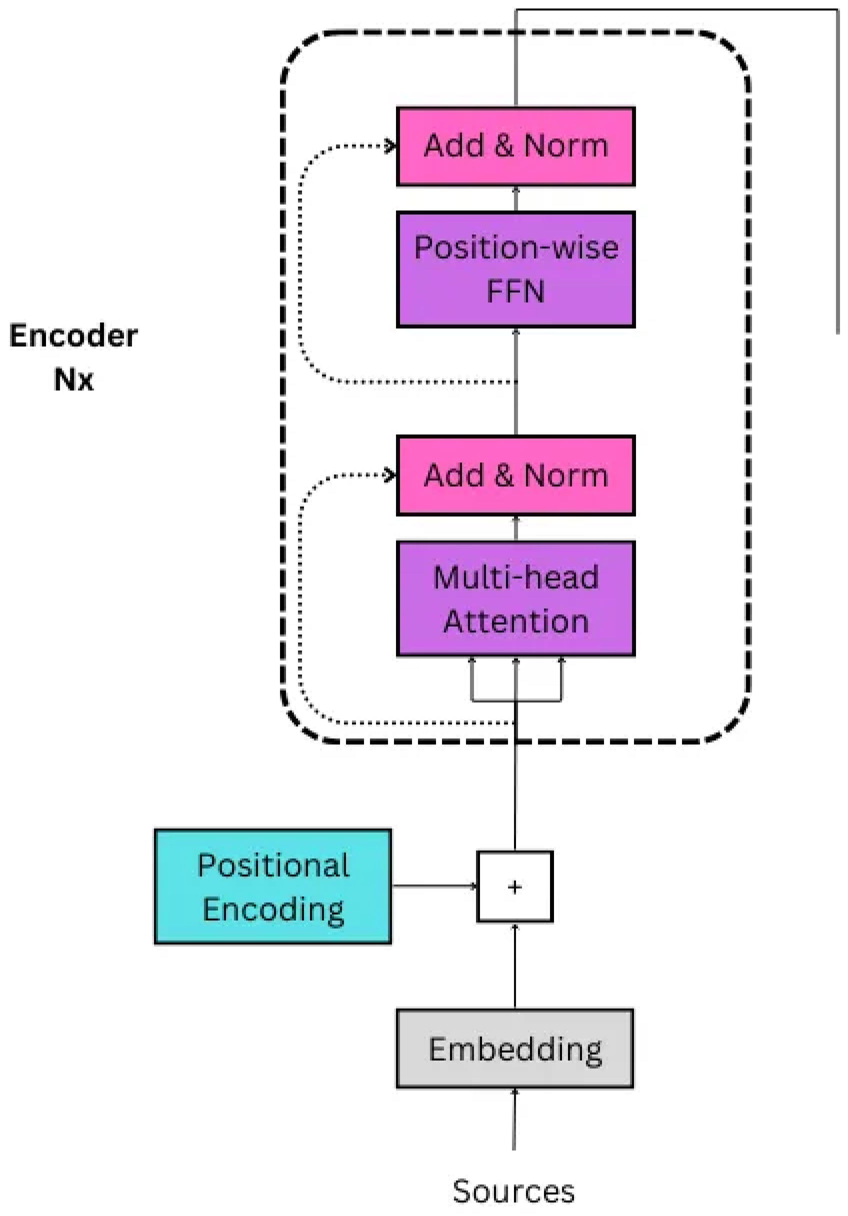


Image by Author

## Background

In general, normalization is the process of transforming features to be on a comparable scale. There are numerous ways to normalize features, including the standard score and min-max feature scaling.

## Min-Max Feature Scaling

Min-max feature scaling transforms values into the range *[0,1]*. This is also known as a unity-based normalization. It can be calculated with the following equation:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

The top of the equation shifts every value by ***X_min***; the numerator becomes 0 when ***X*** = ***X_min***. When this is divided by the denominator, the output is 0.

Likewise, the new maximum occurs when the numerator is ***X_max — X_min***. When this value is divided by ***X_max — X_min***, it becomes 1. This is how the range shifts to between 0 and 1.

The example below demonstrates how the values are calculated, step by step.

```
import torch
X = torch.Tensor([22, 5, 6, 8, 10, 19,2])
X_max = X.max() # 22
X_min = X.min() # 2

# [22-2, 5-2, 6-2, 8-2, 10-2, 19-2, 2-2] =
numerator = X-X_min # [20, 3, 4, 6, 8, 17, 0]

denominator = X_max-X_min # 22 - 2 = 20
```

```
    # [20/20, 3/20, 4/20, 6/20, 8/20, 17/20, 0/20]
    X_new = numerator/denominator
```

```
    tensor([1.00, 0.15, 0.20, 0.30, 0.40, 0.85, 0.00])
```

## Standard Score

During standardization, each value is converted to its standard score. The standard score is also known as the z-score. This is done by subtracting the mean from each value and dividing by the standard deviation.

$$z = \frac{x - \mu}{\sigma}$$

The ***μ*** represents the mean, or average, of the data. It is calculated by summing all the data points in a dataset and dividing by the number of data points, ***n***:

$$\mu = \sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

***σ*** represents the standard deviation of the data, which is the average dispersion of the values from the mean. If a dataset has a low standard deviation, the values are likely closer to the mean. If it has a high standard deviation, this likely means the values are distributed over a larger range. It can be calculated with the following formula.

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

The first step is to find the deviation of each point from the mean. This is done by simply subtracting the mean from each point. These values can then be squared to remove any negatives. Finally, they can be summed and divided by the number of values, $n$. The standard deviation can then be calculated by taking the square root. If the square root is not taken, $\sigma^2$ is the variance.

The example below shows these steps.

```
import torch
X = torch.Tensor([22, 5, 6, 8, 10, 19,2])
n = len(X)

mean = X.sum()/n # X.mean()

std = (((X-mean)**2).sum()/n).sqrt() # X.std(unbiased=False)

z_scores = (X - mean)/std

print(mean, std, z_scores, sep="\n")
```

```
tensor(10.2857)
tensor(6.9016)
tensor([ 1.6973, -0.7659, -0.6210, -0.3312, -0.0414,  1.2626, -1.2005])
```

As can be seen, the values are distributed around 0. This is expected since the mean is subtracted from each value. Now, the standard scores convey similar information without needing large values. Since the mean is 10.2857, it is easy to see that the standard score of 10 is just below 0 at -0.0414.

### Why Normalization?

Normalization is used in machine learning because models with features on different scales take longer to train; this occurs because gradient descent will need more time to converge.

According to Pinecone, a lack of normalization can lead to large error gradients that eventually explode, making the model unstable.

In many cases, data should therefore be normalized before it is inserted into a model.

### Layer Normalization

According to Pinecone, layer normalization ensures "all neurons in a particular layer effectively have the same distribution across all features for a given input."

Normalization is performed on the last ***D*** dimensions; ***D*** is the number of dimensions that will be normalized. For instance, if the goal is to normalize a one-dimensional vector with 10 elements, ***D*** would be 1. If the goal is to normalize a matrix with a shape of *(2,3)*, ***D*** would be 2. Similarly, if the goal is to normalize a tensor with a shape of *(2,5,3)*, ***D*** would be 3.

### The Normalization Equation

For each input, notated $x$, layer normalization can be calculated using a modified z-score equation:

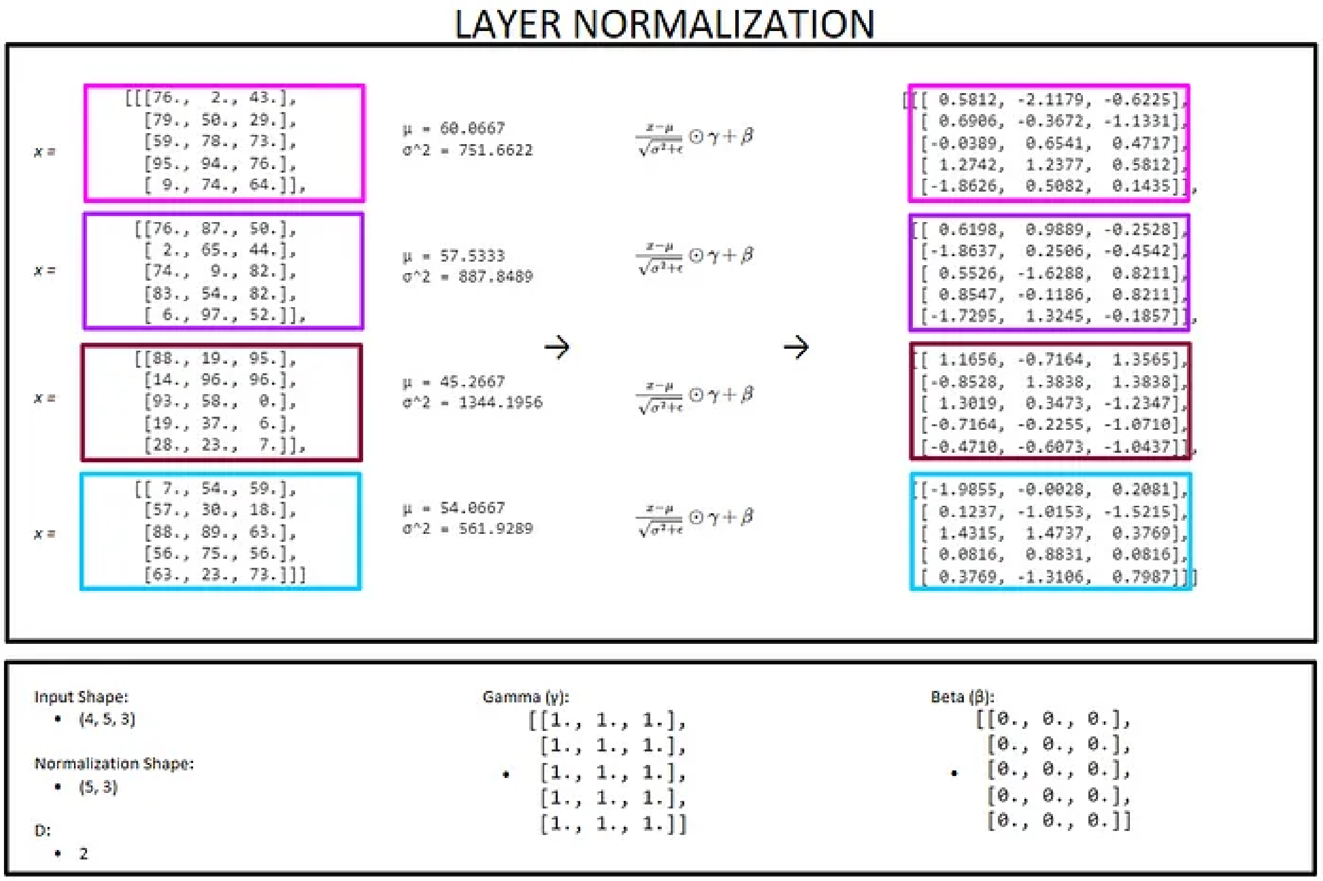
$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta$$

- $\mu$ represents the mean of the last $D$ dimensions
- $\sigma^2$ represents the variance of the last $D$ dimensions
- $\varepsilon$ is an extremely small value that helps when $\sigma^2$ is small
- $\gamma$ and $\beta$ are learnable parameters.

According to Pinecone, $\gamma$ and $\beta$ are used because “forcing all the pre-activations to be zero and unit standard deviation…can be too restrictive. It may be the case that the fluctuant distributions are necessary for the network to learn certain classes better.”

- They have the same shape as the given tensor to be normalized.
- $\gamma$ is initialized as ones, and $\beta$ is initialized as zeros.

## A Generic Example


LAYER NORMALIZATION

```
x = [[[76.,  2., 43.],        μ = 60.0667       → [[[ 0.5812, -2.1179, -0.6225],
      [79., 50., 29.],        σ^2 = 751.6622        [ 0.6986, -0.3672, -1.1331],
      [59., 78., 73.],                              [-0.0389,  0.6541,  0.4717],
      [95., 94., 76.],                              [ 1.2742,  1.2377,  0.5812],
      [ 9., 74., 64.]],                             [-1.8626,  0.5082,  0.1435]],

x =  [[76., 87., 50.],        μ = 57.5333           [[ 0.6198,  0.9889, -0.2528],
      [ 2., 65., 44.],        σ^2 = 887.8489        [-1.8637,  0.2506, -0.4542],
      [74.,  9., 82.],                              [ 0.5526, -1.6288,  0.8211],
      [83., 54., 82.],                              [ 0.8547, -0.1186,  0.8211],
      [ 6., 97., 52.]],                             [-1.7295,  1.3245, -0.1857]],

x =  [[88., 19., 95.],        μ = 45.2667           [[ 1.1656, -0.7164,  1.3565],
      [14., 96., 96.],        σ^2 = 1344.1956       [-0.8528,  1.3838,  1.3838],
      [93., 58.,  0.],                              [ 1.3019,  0.3473, -1.2347],
      [19., 37.,  6.],                              [-0.7164, -0.2255, -1.0710],
      [28., 23.,  7.]],                             [-0.4710, -0.6073, -1.0437]],

x =  [[ 7., 54., 59.],        μ = 54.0667           [[-1.9855, -0.0028,  0.2081],
      [57., 30., 18.],        σ^2 = 561.9289        [ 0.1237, -1.0153, -1.5215],
      [88., 89., 63.],                              [ 1.4315,  1.4737,  0.3769],
      [56., 75., 56.],                              [ 0.0816,  0.8831,  0.0816],
      [63., 23., 73.]]]                             [ 0.3769, -1.3106,  0.7987]]]
```

$$\frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} \odot \gamma + \beta$$

Input Shape:
- (4, 5, 3)

Normalization Shape:
- (5, 3)

D:
- 2

Gamma (γ):

-
  ```
  [[1., 1., 1.],
   [1., 1., 1.],
   [1., 1., 1.],
   [1., 1., 1.],
   [1., 1., 1.]]
  ```

Beta (β):

-
  ```
  [[0., 0., 0.],
   [0., 0., 0.],
   [0., 0., 0.],
   [0., 0., 0.],
   [0., 0., 0.]]
  ```

Image by Author

To demonstrate how layer normalization is calculated, a tensor with a shape of *(4,5,3)* will be normalized across its matrices, which have a size of *(5,3)*. This means ***D*** is 2.

In the image above, it is clear that the values of each matrix are standardized based on the other values within the same matrix.

Layer normalization can be implemented using PyTorch's statistical capabilities.

```
# Input Tensor: 4 matrices of 5 rows and 3 columns
X = torch.randint(0, 100, (4, 5, 3)).float()

# Shape to be Normalized: 5 rows, 3 columns
normalized_shape = (5, 3)
```

```
# Number of Dimensions in the Shape to be Normalized
D = len(normalized_shape)

# Set the Default Values for Epsilon, Gamma, and Beta
eps = 1e-5
gamma = torch.ones(normalized_shape)
beta = torch.zeros(normalized_shape)

X
```

```
tensor([[[[76.,  2., 43.],
          [79., 50., 29.],
          [59., 78., 73.],
          [95., 94., 76.],
          [ 9., 74., 64.]],

         [[76., 87., 50.],
          [ 2., 65., 44.],
          [74.,  9., 82.],
          [83., 54., 82.],
          [ 6., 97., 52.]],

         [[88., 19., 95.],
          [14., 96., 96.],
          [93., 58.,  0.],
          [19., 37.,  6.],
          [28., 23.,  7.]],

         [[ 7., 54., 59.],
          [57., 30., 18.],
          [88., 89., 63.],
          [56., 75., 56.],
          [63., 23., 73.]]]])
```

Each of these matrices can be standardized using a loop. In this loop, the mean and variance are calculated. These values are then inserted into the layer normalization equation to calculate the normalized values for the matrix.

```
# Normalize
for i in range(0,4):                  # loop through each matrix
  mean = X[i].mean()                  # mean
  var = X[i].var(unbiased=False)      # variance
  layer_norm = (X[i]-mean)/(torch.sqrt(var+eps))*gamma + beta

  print(f"μ = {mean:.4f}")
  print(f"σ^{2} = {var:.4f}")
  print(layer_norm)
  print("="*50)
```

```
μ = 60.0667
σ^2 = 751.6622
tensor([[ 0.5812, -2.1179, -0.6225],
        [ 0.6906, -0.3672, -1.1331],
        [-0.0389,  0.6541,  0.4717],
        [ 1.2742,  1.2377,  0.5812],
        [-1.8626,  0.5082,  0.1435]])
==================================================
μ = 57.5333
σ^2 = 887.8489
tensor([[ 0.6198,  0.9889, -0.2528],
        [-1.8637,  0.2506, -0.4542],
        [ 0.5526, -1.6288,  0.8211],
        [ 0.8547, -0.1186,  0.8211],
        [-1.7295,  1.3245, -0.1857]])
==================================================
μ = 45.2667
σ^2 = 1344.1956
tensor([[ 1.1656, -0.7164,  1.3565],
        [-0.8528,  1.3838,  1.3838],
        [ 1.3019,  0.3473, -1.2347],
        [-0.7164, -0.2255, -1.0710],
        [-0.4710, -0.6073, -1.0437]])
==================================================
μ = 54.0667
σ^2 = 561.9289
tensor([[-1.9855, -0.0028,  0.2081],
        [ 0.1237, -1.0153, -1.5215],
        [ 1.4315,  1.4737,  0.3769],
        [ 0.0816,  0.8831,  0.0816],
        [ 0.3769, -1.3106,  0.7987]])
==================================================
```

By examining the image above and the output of the code, it is apparent that layer normalization calculates a variation of the z-score for the values in each matrix in this example.

In the first matrix, the mean is about 60. It is thus reasonable that 59 would have a z-score of -0.0389. Likewise, 2, the value furthest from the mean, has a z-score of -2.1179.

The same answer as above can be calculated using PyTorch's ***LayerNorm*** module. This module requires the shape-to-be-normalized to be initialized. Then, the tensor can be passed to the module, and each matrix will be normalized accordingly.

```
layer_normalization = nn.LayerNorm(normalized_shape) # nn.LayerNorm((5,3))
layer_normalization(X)
```

```
tensor([[[ 0.5812, -2.1179, -0.6225],
         [ 0.6906, -0.3672, -1.1331],
         [-0.0389,  0.6541,  0.4717],
         [ 1.2742,  1.2377,  0.5812],
         [-1.8626,  0.5082,  0.1435]],

        [[ 0.6198,  0.9889, -0.2528],
         [-1.8637,  0.2506, -0.4542],
         [ 0.5526, -1.6288,  0.8211],
         [ 0.8547, -0.1186,  0.8211],
         [-1.7295,  1.3245, -0.1857]],

        [[ 1.1656, -0.7164,  1.3565],
         [-0.8528,  1.3838,  1.3838],
         [ 1.3019,  0.3473, -1.2347],
         [-0.7164, -0.2255, -1.0710],
         [-0.4710, -0.6073, -1.0437]],

        [[-1.9855, -0.0028,  0.2081],
         [ 0.1237, -1.0153, -1.5215],
         [ 1.4315,  1.4737,  0.3769],
         [ 0.0816,  0.8831,  0.0816],
         [ 0.3769, -1.3106,  0.7987]]],
         grad_fn=<NativeLayerNormBackward0>)
```

## NLP Example

In natural language processing, layer normalization occurs across the embedding dimensions of each token. For a batch with 2 sequences, 3 tokens, and 5-element embeddings, the shape is *(2, 3, 5)*. *D* would be 1 since the last dimension will be normalized. The shape of the embedding will be *(5,)*. It has to be initialized in a tuple to ensure its value can be extracted by the *LayerNorm* module. Alternatively, *X.shape[-1]* could be used.

```
# Input Tensor: 2 sequences of 3 tokens with 5 dimensional embeddings
X = torch.randint(2, 3, 5)

# Shape to be Normalized: 5 dimensional embedding
normalized_shape = (5,)

# Number of Dimensions in the Shape to be Normalized
D = len(normalized_shape) # 1

# Create the LayerNorm
layer_normalization = nn.LayerNorm(normalized_shape)

# view the beta and gamma and beta
layer_normalization.state_dict()
```

```
OrderedDict([('weight', tensor([1., 1., 1., 1., 1.])),
             ('bias',   tensor([0., 0., 0., 0., 0.]))])
```

Above are the values of $\gamma$ and $\beta$. Below, it can be seen that the values of $X$ were initialized as integers to easily demonstrate how layer normalization impacts them.

```
X
```

```
tensor([[[49., 90., 29., 76., 33.],
         [86., 42., 20., 56., 79.],
         [40., 49., 72., 16., 85.]],

        [[44., 62., 14., 46.,  5.],
         [22., 45.,  8., 47., 78.],
         [96., 17.,  7., 56., 60.]]])
```

The mean value of each row can also be calculated using PyTorch instead of a for-loop:

```
X.mean(2, keepdims=True) # maintains the dimensions of X
```

```
tensor([[[55.4000],
         [56.6000],
         [52.4000]],

        [[34.2000],
         [40.0000],
         [47.2000]]])
```

These values can be referred to after normalization is applied to see how far each value was from the mean.

```
layer_normalization(X)
```

```
tensor([[[-0.2675,  1.4464, -1.1036,  0.8611, -0.9364],
         [ 1.2167, -0.6042, -1.5147, -0.0248,  0.9270],
         [-0.5116, -0.1403,  0.8087, -1.5018,  1.3450]],

        [[ 0.4601,  1.3051, -0.9483,  0.5539, -1.3708],
         [-0.7518,  0.2088, -1.3366,  0.2924,  1.5872],
         [ 1.5204, -0.9409, -1.2525,  0.2742,  0.3988]]],
       grad_fn=<NativeLayerNormBackward0>)
```

For the first token, 90 has the largest z-score at 1.4464 since it is the furthest from the mean of 55.4, which would have a z-score of 0.

Now that layer normalization has been explained from a generic and natural language processing point of view, it is time to show how it is used in transformers.

## Layer Normalization in Transformers

Although PyTorch has its built in *LayerNorm* module, it can be recreated for a better understanding of its use in the transformers model. The implementation is relatively straightforward and follows the code from the explanation.

```
class LayerNorm(nn.Module):

  def __init__(self, features, eps=1e-5):
    super().__init__()
    # initialize gamma to be all ones
    self.gamma = nn.Parameter(torch.ones(features))
    # initialize beta to be all zeros
    self.beta = nn.Parameter(torch.zeros(features))
    # initialize epsilon
    self.eps = eps

  def forward(self, src):
    # mean of the token embeddings
    mean = src.mean(-1, keepdim=True)
    # variance of the token embeddings
    var = src.var(-1, keepdim=True,unbiased=False)
    # return the normalized value
    return self.gamma * (src - mean) / torch.sqrt(var + self.eps) + self.beta
```

The layer normalization will be utilized in the Encoder, so its usage will be demonstrated with residual addition in the next article, The Encoder. For now, its important that its implementation is understood.

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

## References

1. Pinecone's Batch and Layer Normalization
2. PyTorch's LayerNorm Module
3. Wikipedia's Normalization Article

Normalization Layer Normalization Transformers NLP Machine Learning

## 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

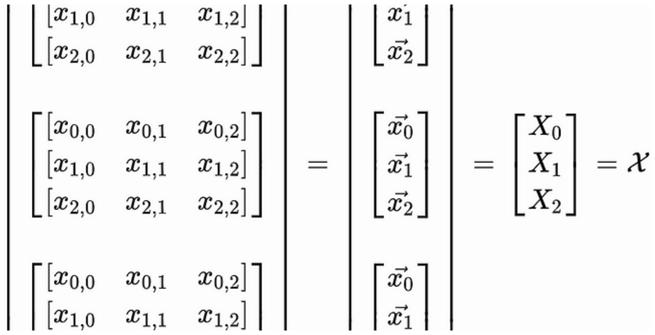
## Written by Hunter Phillips

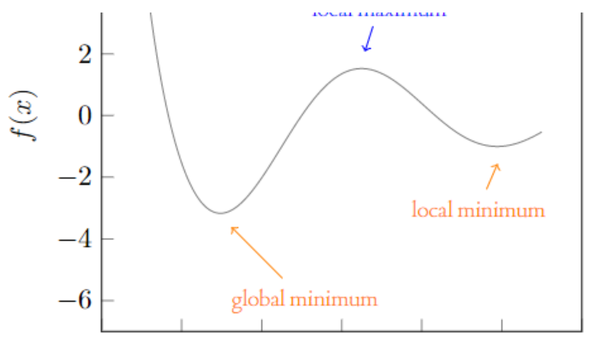
Following

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

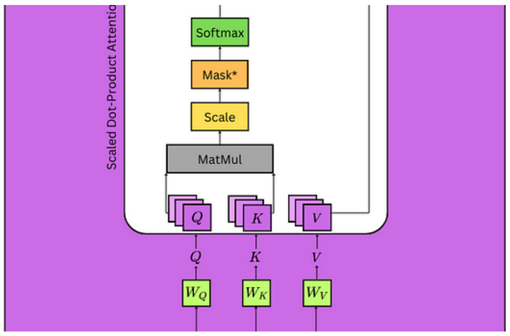
Hunter Phillips

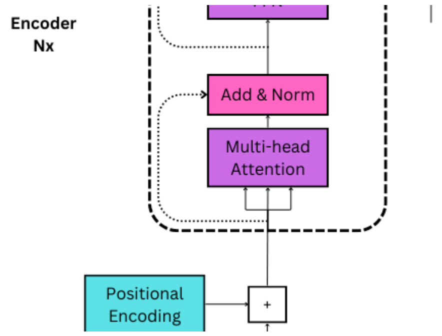
## A Simple Introduction to Gradient Descent

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

## Position-Wise Feed-Forward Network (FFN)

This is the fourth article in The Implemented Transformer series. The Position-wise Feed-…

9 min read · May 9

155

See all from Hunter Phillips

## Recommended from Medium

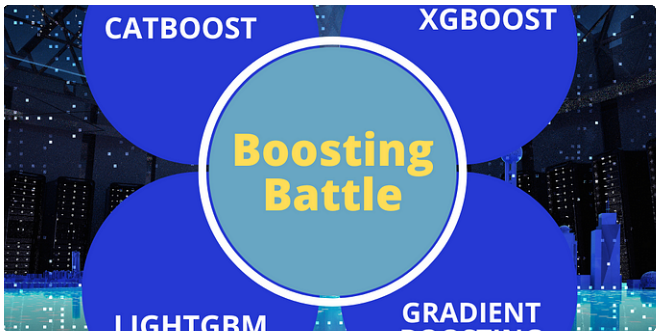
Walter Sperat

**Using Optuna the wrong way**

If you have lived under a rock for the last couple of years, Optuna is a Python library…

7 min read · Jun 8

23



Uttam Kumar

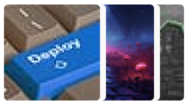
**Boosting Techniques Battle: CatBoost vs XGBoost vs LightGB…**

Machine learning models have revolutionized the way we approach and solve complex…

9 min read · Apr 4

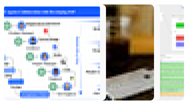
115

### Lists



**Predictive Modeling w/ Python**

20 stories · 452 saves



**Natural Language Processing**

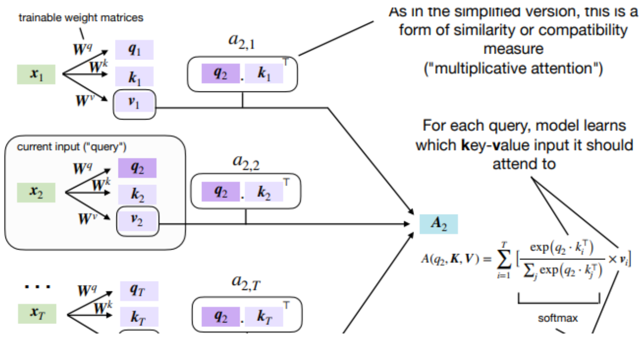
669 stories · 283 saves

Practical Guides to Machine Learning

10 stories · 519 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves



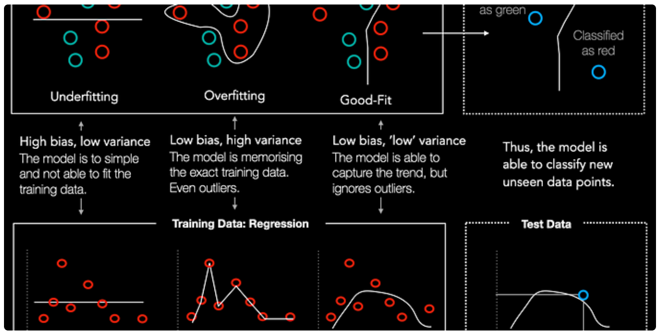
Zain ul Abideen

**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



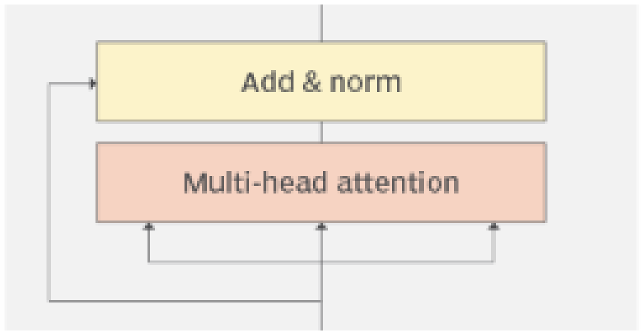
Frederik vl in Advanced Deep Learning

**Understanding Bias and Variance in Machine Learning**

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

3 min read · Sep 15

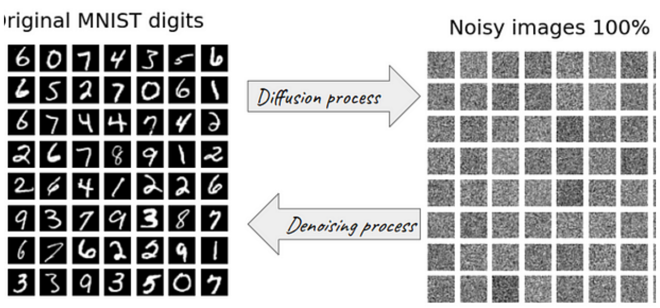
44



Eugene Ku

**Transformer Architecture (Part 3— Scaling Self-Attention)**

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



Antony M. Gitau

**A friendly Introduction to Denoising Diffusion Probabilistic…**

I recently attended a Nordic probabilistic AI school, ProbAI 2023, which inspired my…

8 min read · Aug 23

10

9 min read · Jul 9

53

See more recommendations