Use weighted loss function to solve imbalanced data classification problems

hengtao tantai · Follow
8 min read · Feb 27

Imbalanced datasets are a common problem in classification tasks, where number of instances in one class is significantly smaller than number of instances in another class. This will lead to biased models that perform poorly on minority class.

generated by midjourney

A weighted loss function is a modification of standard loss function used in training a model. The weights are used to assign a higher penalty to mis classifications of minority class. The idea is to make model more sensitive to minority class by increasing cost of mis classification of that class.

The most common way to implement a weighted loss function is to assign higher weight to minority class and lower weight to majority class. The weights can be inversely proportional to frequency of classes, so that minority class gets higher weight and majority class gets lower weight.

In this post, I will show you how to add weights to pytorch's common loss functions

Binary Classification

`torch.nn.BCEWithLogitsLoss` function is a commonly used loss function for binary classification problems, where model output is a probability value between 0 and 1. It combines a sigmoid activation function with a binary cross-entropy loss.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right]$$

For imbalanced datasets, where number of instances in one class is significantly smaller than other, `torch.nn.BCEWithLogitsLoss` function can be modified by adding a weight parameter to loss function. The weight parameter allows to assign different weights for the positive and negative classes.

The weight parameter is a tensor of size `[batch_size]` that contains weight value for each sample in the batch.

Here is an example:

```
import torch
 import torch.nn as nn

 # Define the BCEWithLogitsLoss function with weight parameter
 weight = torch.tensor([0.1, 0.9]) # higher weight for positive class
 criterion = nn.BCEWithLogitsLoss(weight=weight)

 # Generate some random data for the binary classification problem
 input = torch.randn(3, 1)
 target = torch.tensor([[0.], [1.], [1.]])

 # Compute the loss with the specified weight
 loss = criterion(input, target)

 print(loss)
```

we set weight of positive class to 0.9 and weight of negative class to 0.1. `input` tensor contains logits predicted by model. `target` tensor contains ground-truth labels for binary classification problem.

Note: Set weights manually to 0.1 and 0.9, based on assumption that positive class has only 10% of the samples.

To calculate weights, you can compute weight for each class as:

```
weight_for_class_i = total_samples / (num_samples_in_class_i * num_classes)
```

where `total_samples` is total number of samples in dataset, `num_samples_in_class_i` is number of samples in class i, and `num_classes` is total number of classes (in the case of binary classification, `num_classes` is 2).

If you have a binary classification problem with 1000 samples, where 900 samples belong to class 0 and 100 samples belong to class 1, calculate weights as:

```
weight_for_class_0 = 1000 / (900 * 2) = 0.5556
 weight_for_class_1 = 1000 / (100 * 2) = 5.0000
```

and the weight parameter of `torch.nn.BCEWithLogitsLoss` to a tensor of size 2 weights is:

```
import torch
 import torch.nn as nn

 weight = torch.tensor([0.5556, 5.0000]) # higher weight for class 1
 criterion = nn.BCEWithLogitsLoss(weight=weight)
```

Note: specific formula and method for calculating weights can depend on problem and dataset, and there may be other approaches to consider, such as using resampling techniques or other loss functions that are designed to handle class imbalance.

In addition to `weight` parameter, `torch.nn.BCEWithLogitsLoss` also has a `pos_weight` parameter, which is a simpler way to specify weight for positive class in a binary classification problem.

The `pos_weight` parameter is a scalar that represents weight for positive class. It is equivalent to setting `weight` parameter to `[1, pos_weight]`, where weight for negative class is 1.

```
import torch
 import torch.nn as nn

 # Define the BCEWithLogitsLoss function with pos_weight parameter
 pos_weight = torch.tensor([3.0])  # higher weight for positive class
 criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)

 # Generate some random data for the binary classification problem
 input = torch.randn(3, 1)
 target = torch.tensor([[0.], [1.], [1.]])

 # Compute the loss with the specified pos_weight
 loss = criterion(input, target)

 print(loss)
```

Set `pos_weight` parameter to 3.0, means weight for positive class is 3 times of negative class. We compute loss using `criterion` function with specified `pos_weight`.

If you specify both `weight` and `pos_weight` parameters, `pos_weight` parameter takes precedence over weight for positive class.

If you set `pos_weight` to a value other than 1.0, weight for positive class in `weight` tensor will be ignored.

Multi-class Classification

Cross-Entropy Loss is commonly used in multi-class classification problems. It calculates negative log-likelihood of predicted class distribution compared to true class distribution. When dealing with imbalanced datasets, some classes might have more or fewer samples than others, and this can lead to a bias in model's predictions towards the more frequent classes.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

To address this issue, `weight` parameter in `torch.nn.CrossEntropyLoss` can be used to apply a weight to each class. The weights can be specified as a 1D Tensor or a list and should have same length as the number of classes. The loss is then calculated as follows:

```
loss(x, y) = -weight[y] * log(exp(x[y]) / sum(exp(x)))
```

where `x` is model's output, `y` is target class, `exp` is exponential function, and `sum(exp(x))` is sum of exponentials over all classes. The weight for true class is multiplied with negative log-likelihood of true class, so loss is up weighted for underrepresented classes.

Here's an example of using `weight` parameter in `torch.nn.CrossEntropyLoss` for a classification problem with three imbalanced classes.

Suppose we have a dataset with 1000 samples, and target variable has three classes: Class A, Class B, and Class C. The distribution of samples in dataset is as follows:

- Class A: 100 samples
- Class B: 800 samples
- Class C: 100 samples

To address class imbalance, we can assign a weight to each class that inversely proportional to its frequency. The weight of each class can be calculated as follows:

- Class A: 1000 / 100 = 10
- Class B: 1000 / 800 = 1.25
- Class C: 1000 / 100 = 10

We can create a weight tensor that reflects these weights as follows:

```
import torch
```

```
weights = torch.tensor([10, 1.25, 10])
```

And use this weight tensor in `torch.nn.CrossEntropyLoss` as follows:

```
loss_fn = torch.nn.CrossEntropyLoss(weight=weights)
```

When we use this loss function to train a model , Classes A and C will contribute more to loss than Class B, due to up weighted effect of weight tensor. This helps to balance effect of different classes on model's training and can improve model's performance on the underrepresented classes.

In PyTorch's `torch.nn.CrossEntropyLoss` , `label_smoothing` parameter is used to smooth one-hot encoded target values toencourage model to be less confident in its predictions and prevent overfitting to training data.

This smoothing is achieved by adding a small value (i.e., smoothing factor) to off-diagonal elements of one-hot encoded target values and subtracting this same value from diagonal elements. This has effect of reducing confidence of model's predictions and encouraging it to explore a wider range of solutions.

`label_smoothing` parameter take a value between 0 and 1, 0 means no smoothing is applied, and 1 means maximum smoothing is applied. A typical value for `label_smoothing` is around 0.1. The loss is then calculated as follows:

```
loss(x, y) = -((1 - label_smoothing) * log(exp(x[y]) / sum(exp(x))) + (label_sm
```

x is model's output, y is target class, exp is exponential function, sum(exp(x)) is sum of exponentials over all classes, and K is number of classes. The smoothing factor is multiplied by a uniform prior probability for each class, which is equal to 1/K, and achieve more generalized model.

Both `label_smoothing` and `weight` parameters can be used to address issues related to class imbalance and overfitting in multi-class classification problems. However, these two parameters work in different ways and have different use cases.

`weight` parameter is used to apply a weight to each class in loss calculation, which can be useful when dealing with imbalanced datasets. The weight for true class is multiplied with negative log-likelihood of true class, so loss is upweighted for underrepresented classes. `weight` parameter does not affect predicted probabilities or model's confidence in its predictions, but rather adjusts weight assigned to each class in loss calculation.

`label_smoothing` parameter is used to smooth one-hot encoded target values to encourage model to be less confident in its predictions and prevent overfitting to training data. Smoothing is achieved by adding a small value (i.e., smoothing factor) to off-diagonal elements of one-hot encoded target values and subtracting this same value from diagonal elements. This has effect of reducing confidence of the model's predictions and encouraging it to explore a wider range of solutions. `label_smoothing` parameter does not affect weight assigned to each class in loss calculation, but rather adjusts the target values used in loss calculation.

In general, `weight` parameter is useful when dealing with imbalanced datasets, where the cost of misclassification is not same for all classes. On other hand, `label_smoothing` parameter is useful when dealing with overfitting and highly confident predictions, where model is too confident in its predictions and is not exploring a wide range of solutions.

both `weight` and `label_smoothing` parameters can be used together to address both class imbalance and overfitting issues in multi-class classification problems.

However, two parameters may not always be used together and their use depends on specific characteristics of the dataset and problem at hand. It's important to experiment with different combinations of hyperparameters, including `label_smoothing` and `weight`, to determine best approach for a given problem.

Multilabel classification

Multilabel classification is a type of classification problem where an object or instance can belong to one or more classes simultaneously. In other words, instead of assigning a single label to each instance, multiple labels can be assigned to it.

This is in contrast to traditional classification problems, where each instance is assigned a single label.

Binary Cross-Entropy Loss commonly used in binary classification problems, but can also be used in multilabel classification by treating each label as a separate binary classification problem. It measures the difference between the predicted probabilities and actual labels for each class separately.

we can set the `weight` parameter, and `pos_weight` like Binary Classification do.

In summary

Imbalanced datasets are a common problem in machine learning, where one class of data may be significantly more prevalent than another.This can lead to issues when trying to train a classification model, as model may become biased towards more prevalent class.One solution to this problem is to use a weighted loss function during training.A weighted loss function assigns a higher weight to errors made on the minority class, thereby helping model to pay more attention to these samples.By using a weighted loss function, model can learn to better distinguish between the minority and majority classes, leading to improved performance on imbalanced datasets.

If this post is helpful to you, please clap👏 and follow me😊.

Artificial Intelligence | Machine Learning | Data Science | Pytorch | Deeplearing

Follow

Written by hengtao tantai

29 Followers

Independent Researcher.I post the AI content that I am interested in.Hope you like it too

More from hengtao tantai

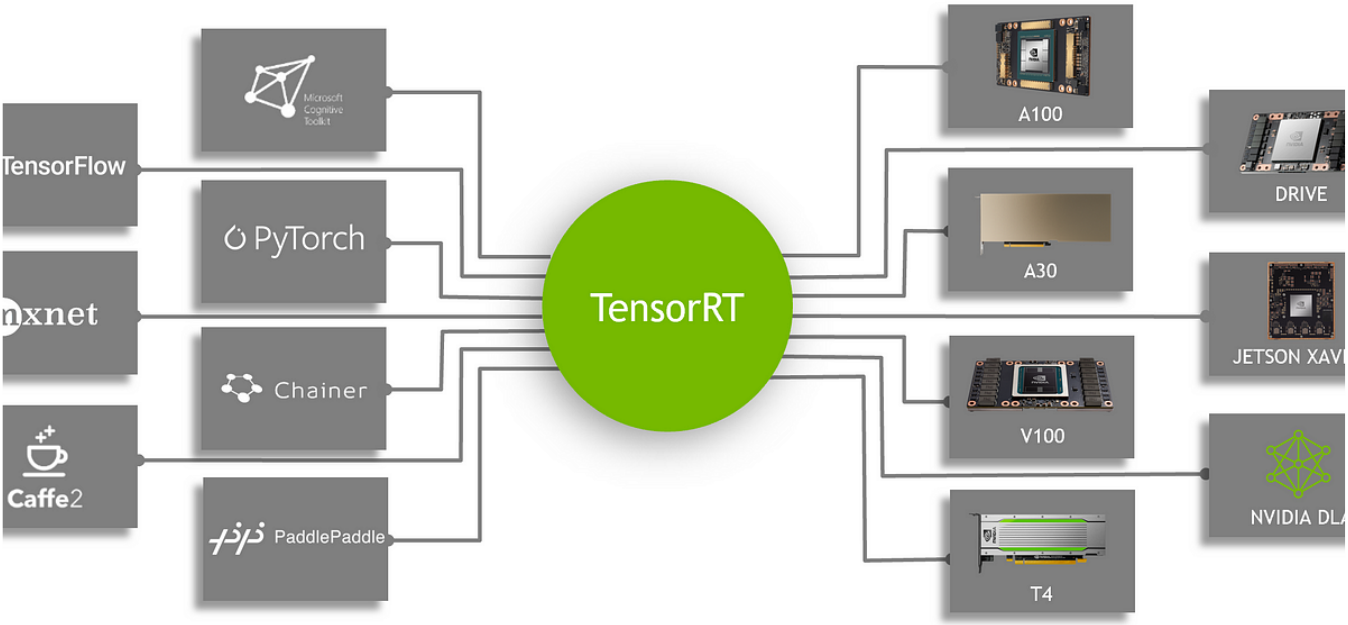
hengtao tantai

What is Safetensors and how to convert .ckpt model to .safetensors

If you often download model weight file, you will often see the .safetensors file."Safetensors" is a new file format for storing tensors...

5 min read · Apr 7

15 3



hengtao tantai

Accelerating Model inference with TensorRT: Tips and Best Practices for PyTorch Users

TensorRT is a high-performance deep-learning inference library developed by NVIDIA. It is designed to optimize and accelerate the inference...

10 min read · Apr 1

hengtao tantai

Improving Control and Reproducibility of PyTorch DataLoader with Sampler Instead of Shuffle...

DataLoader is a class that provides an iterable over a given dataset in Pytorch, which can be used to efficiently load data in parallel...

8 min read · Feb 22

7

hengtao tantai

Convert Pytorch model to tf-lite with onnx-tf

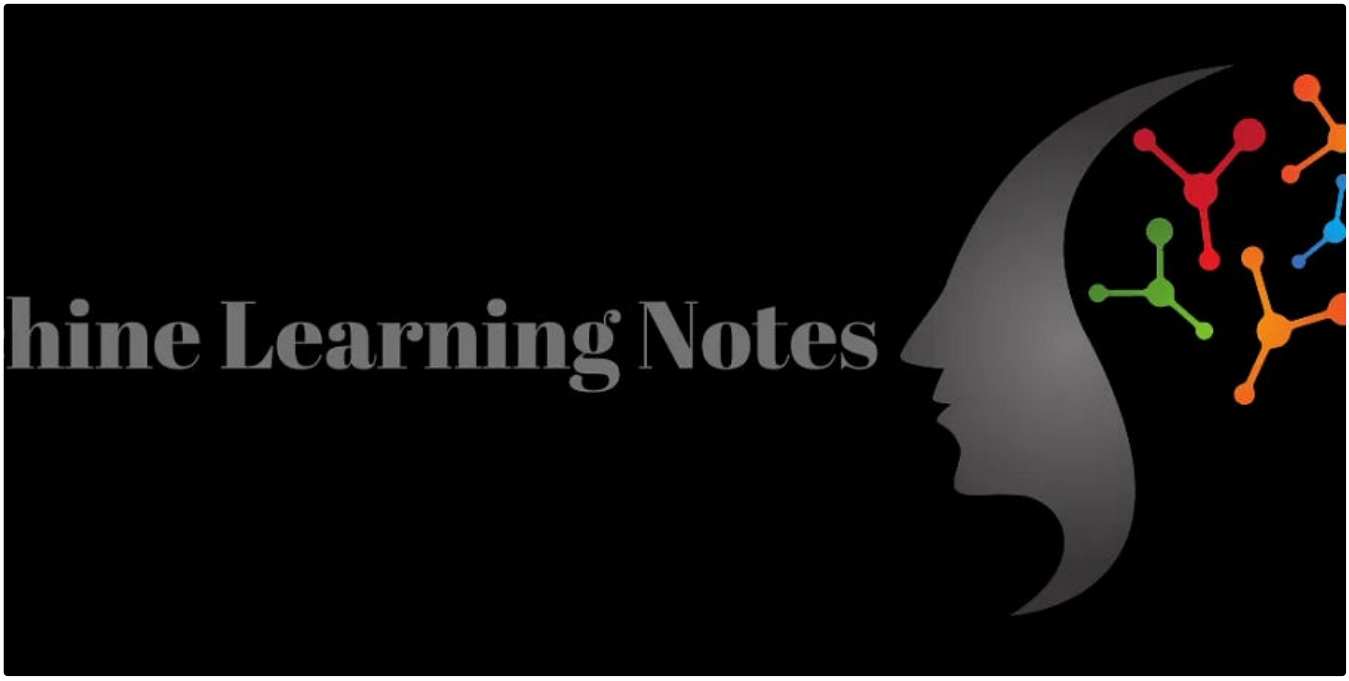
What is TensorFlow Lite

3 min read · Mar 6

1 1

See all from hengtao tantai

Recommended from Medium



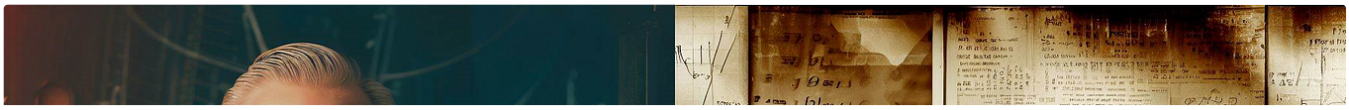
Rahul S

Machine Learning: Cross Entropy and Cross-Entropy Loss

Cross Entropy and Cross-Entropy Loss are closely related concepts, but they serve different purposes in the realm of probability theory and...

· 2 min read · Oct 3

3

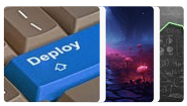


Somayyeh Gholami

The Bayesian Approach: An Efficient Way For Hyperparameters Tuning (Optimizing)

16 min read · Aug 24

Lists



Predictive Modeling w/ Python

20 stories · 537 saves

Practical Guides to Machine Learning

10 stories · 620 saves

Natural Language Processing

762 stories · 346 saves

ChatGPT prompts

27 stories · 572 saves

Siladittya Manna in The Owl

Weighted Categorical Cross-Entropy Loss in Keras

In this article, we will be looking at the implementation of the Weighted Categorical Cross-Entropy loss.

3 min read · Aug 28

55

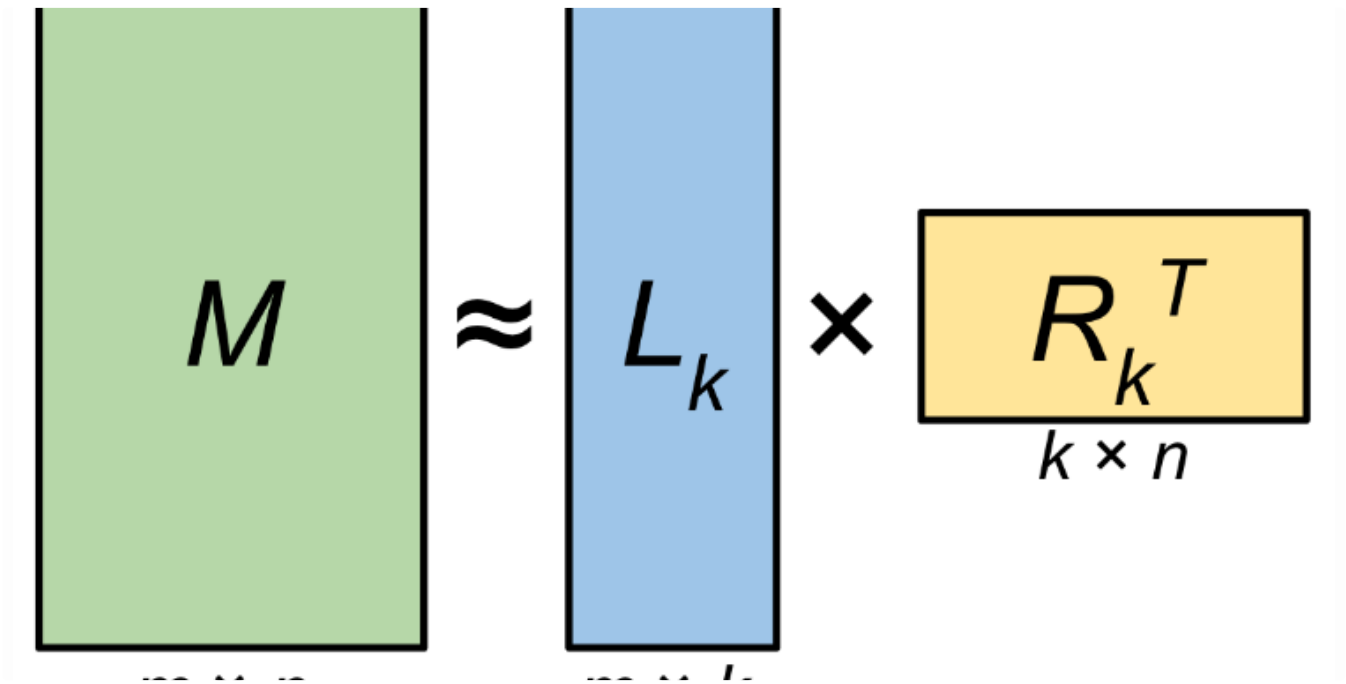
Ruman

Part 1: Ultimate Guide to Fine-Tuning in PyTorch : Pre-trained Model and Its Configuration

Master model fine-tuning: Define pre-trained model, Modifying model head, loss functions, learning rate, optimizer, layer freezing, and...

16 min read · Jul 17

151 2



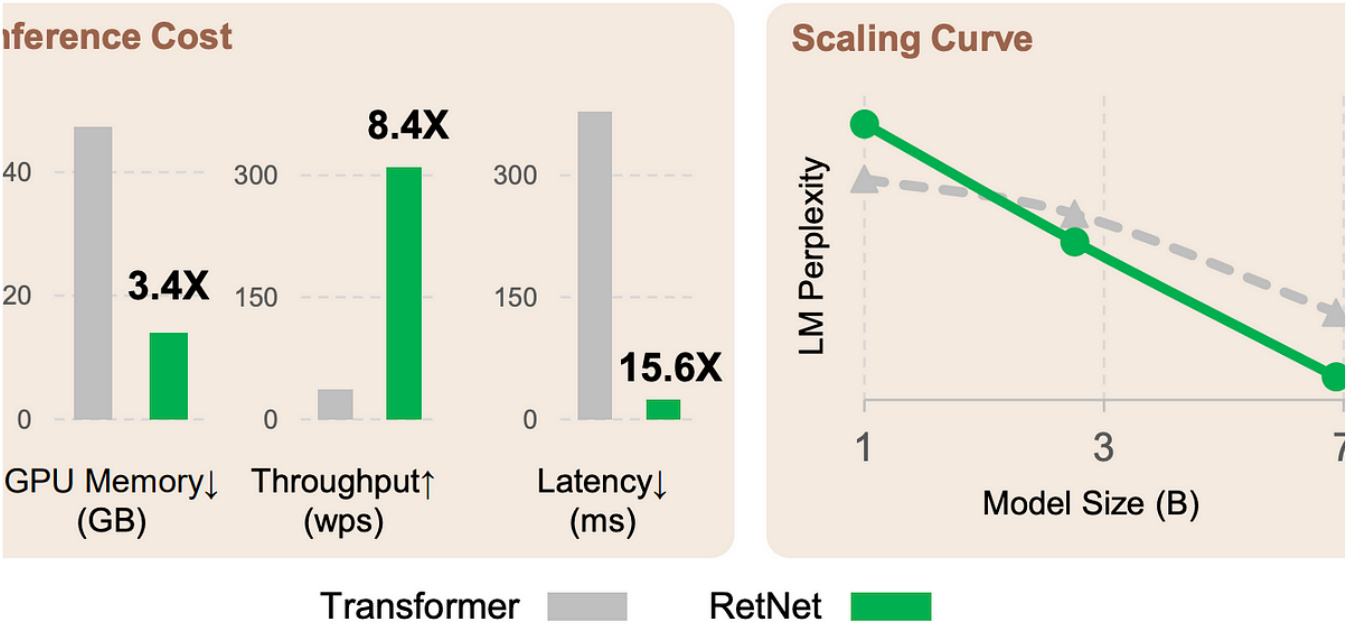
Zeeba Navas

LoRA and LLMs

ChatGPT has changed the way the world's interaction with Deep Learning Models. There is wide adoption of ChatGPT from writing stories to...

5 min read · Sep 7

3



Freedom Preetham in Autonomous Agents

What Next after Transformers? A Rigorous Mathematical Examination of the Retentive Network (RETNET)

The landscape of deep learning is punctuated by the emergence of novel architectures, each aiming to address the multifaceted challenges...

10 min read · Sep 25

-- 1