

Министерство образования и науки РФ
Санкт-Петербургский Политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ
«Vision Transformer»
по дисциплине «Машинное обучение»

Выполнили:

студентки гр. 3540201/20301

_____ Климова О. А.

_____ Сысоева П. А.

подпись, дата

Проверил:

д.т.н., проф.

_____ Уткин Л. В.

подпись, дата

Санкт-Петербург

2023

Содержание

1. Теория	3
2. Пример работы с Vision Transformer	4
3. Задания	16
3.1 Задание № 1	16
3.2 Задание № 2	16
3.3 Задание № 2	16

1. Теория

Vision Transformer (ViT) — это преобразователь, предназначенный для решения задач машинного зрения, таких как распознавание изображений. В данной модели изображение разбивается на патчи фиксированного размера, которые далее линейно выстраиваются, проецируются, и к ним добавляются позиции, что формирует последовательность векторов, которые далее передаются в стандартный преобразователь энкодер (рис. 1).

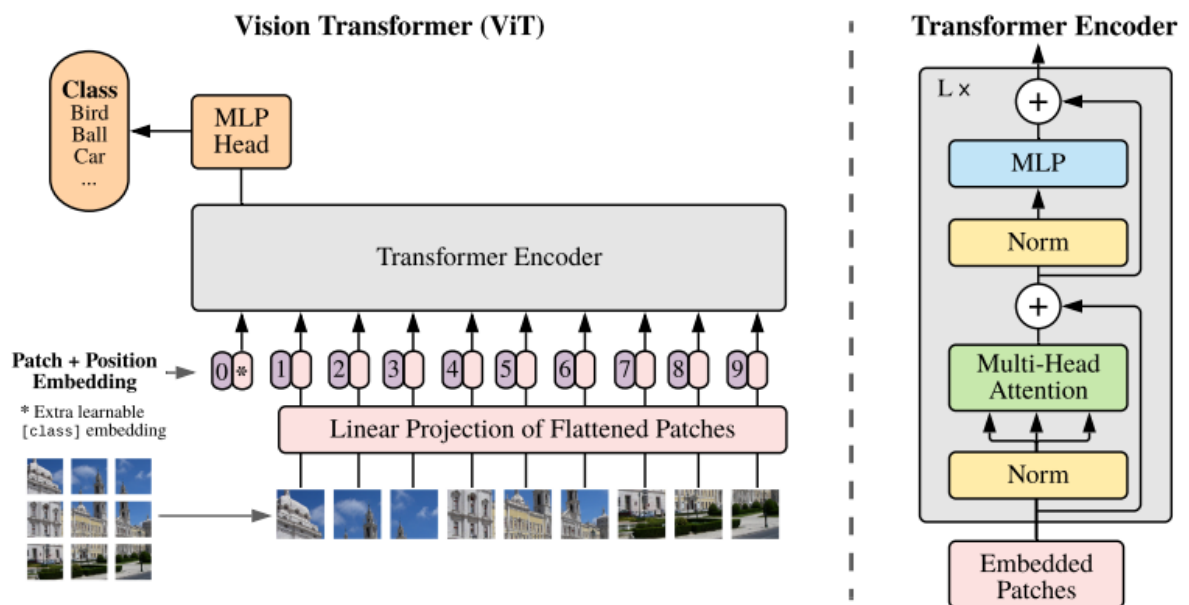


Рисунок 1 – Vision Transformer (ViT)

Трансформеры изначально лишены присущих CNN индуктивных смещений, таких как локальность, и плохо обобщают, когда обучаются на недостаточном количестве данных. Тем не менее, они достигают или превосходят уровень CNN в нескольких тестах распознавания изображений при обучении на больших наборах данных.

2. Пример работы с Vision Transformer

Реализация Vision Transformer осуществлялась в Pytorch. Для проверки работы модели использовалось предварительно загруженное изображение (рис. 2).

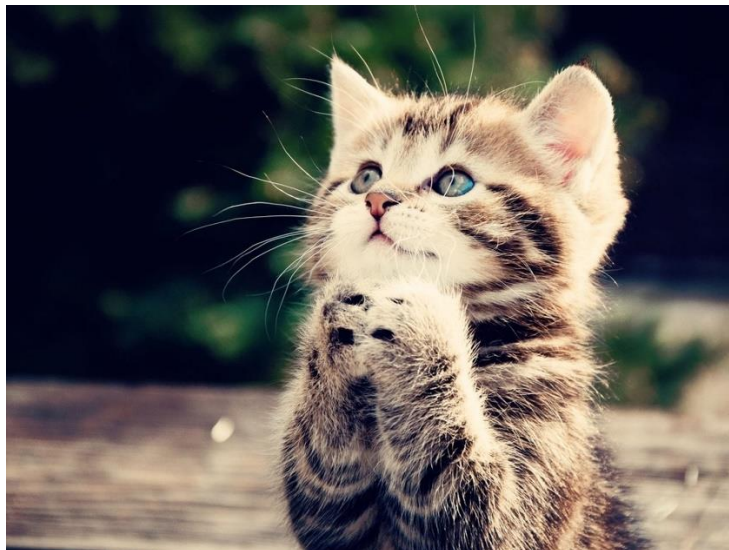


Рисунок 2 – Изображение для проверки работы модели

В результате реализации модели была получена краткая сводка о ее характеристиках (рис. 3).

```
Total params: 86,415,592
Trainable params: 86,415,592
Non-trainable params: 0
-----
Input size (MB): 0.57
Forward/backward pass size (MB): 364.33
Params size (MB): 329.65
Estimated Total Size (MB): 694.56
```

Рисунок 3 – Характеристики ViT

Также была продемонстрирована классификация рассматриваемого изображения – отнесение к одному из 1000 классов (рис. 4).

```
# Определение класса изображения

model = ViT()
model(x).argmax().item()

855
```

Рисунок 4 – Классификация с использованием ViT

Код, демонстрирующий реализацию Vision Transformer в Pytorch представлен далее.

Скачиваем пакет einops – пакет для обработки тензоров:

```
!pip install einops
```

Импортируем необходимые пакеты:

```
import matplotlib.pyplot as plt #пакет для построения графиков
from PIL import Image          #пакет для работы с изображениями
import torch
import torch.nn.functional as F
from torch import Tensor, nn    #пакет для построения слоев модели
from torchsummary import summary
#для аугментации изображений
from torchvision.transforms import Compose, Resize, ToTensor
from einops import rearrange, reduce, repeat
from einops.layers.torch import Rearrange, Reduce
```

Загружаем изображение для проверки:

```
img = Image.open('Cat.jpg') #картинка кота, представленная выше
fig = plt.figure() #создает новую фигуру или активирует существующую
plt.imshow(img)    #отображение данных в виде изображения, т. е. в виде
                   #обычного двумерного раstra
plt.show()         #показывает все открытые фигуры
```

Производим препроцессинг (обрезаем картинку до размера 224 на 224):

```
#функция для обрезания картинки под размер 224 на 224
transform = Compose([
    Resize((224, 224)),
    ToTensor(),
])
x = transform(img)
x = x.unsqueeze(0)
#выводим размерность картинки
print(x.shape)
torch.Size([1, 3, 224, 224])
```

Проецирование (каждый кусочек картинки сглаживается и ему присваивается порядковый номер относительно целого изображения):

```
#размер изображение
img_size = 224
```

```

class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16,
emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        #последовательность шагов для сглаживания
        self.projection = nn.Sequential(
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),) # разбиение и
сглаживание картинки
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size))
        self.positions = nn.Parameter(torch.randn((img_size //
patch_size)**2 + 1, emb_size))
        #прямой проход модели
    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        # подготовка токенов для передачи в модель
        x = torch.cat([cls_tokens, x], dim=1)
        x += self.positions
        return x

```

Определяется механизм внимания:

```

class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size: int = 768, num_heads: int = 8, dropout:
float = 0):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        # queries, keys и values
        self.qkv = nn.Linear(emb_size, emb_size * 3)
        self.att_drop = nn.Dropout(dropout)
        self.projection = nn.Linear(emb_size, emb_size)
    def forward(self, x : Tensor, mask: Tensor = None) -> Tensor:
        # разбиваем keys, queries и values по количеству heads внимания
        qkv = rearrange(self.qkv(x), "b n (h d qkv) -> (qkv) b h n d",
h=self.num_heads, qkv=3)
        queries, keys, values = qkv[0], qkv[1], qkv[2]
        energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
        if mask is not None:
            fill_value = torch.finfo(torch.float32).min
            energy.mask_fill(~mask, fill_value)
        scaling = self.emb_size ** (1/2)
        att = F.softmax(energy, dim=-1) / scaling
        att = self.att_drop(att)
        out = torch.einsum('bhal, bhlv -> bhav ', att, values)
        out = rearrange(out, "b h n d -> b n (h d)")

```

```

out = self.projection(out)
return out

```

Оболочка для выполнения residual addition:

```

class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn
    def forward(self, x, **kwargs):
        res = x                                #residual
        x = self.fn(x, **kwargs)
        x += res                                #добавляем residual
        return x

```

Прямой проход полносвязной сети:

```

class FeedForwardBlock(nn.Sequential):
    def __init__(self, emb_size: int, L: int = 4, drop_p: float = 0.):
        super().__init__(
            nn.Linear(emb_size, L * emb_size), #линейный слой
            nn.GELU(),                          #функция активации
            nn.Dropout(drop_p),                 #dropout
            nn.Linear(L * emb_size, emb_size), #линейный выходной слой
        )

```

Реализуем блок энкодера: он состоит из residual connection, линейного слоя, части отвечающей за внимание, и дропаута:

```

class TransformerEncoderBlock(nn.Sequential):
    def __init__(self, emb_size: int = 768, drop_p: float = 0.,
forward_expansion: int = 4,
                forward_drop_p: float = 0., **kwargs):
        super().__init__(
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                MultiHeadAttention(emb_size, **kwargs),
                nn.Dropout(drop_p)
            )),
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                FeedForwardBlock(
                    emb_size, L=forward_expansion, drop_p=forward_drop_p),
                nn.Dropout(drop_p)
            ))
        )

```

Проверяем корректность реализации:

```
patches_embedded = PatchEmbedding()(x)
print(TransformerEncoderBlock()(patches_embedded).shape)
```

Реализуем классификатор, который состоит из стандартного линейного слоя, на выходе получаем распределение по классам:

```
class ClassificationHead(nn.Sequential):
    def __init__(self, emb_size: int = 768, n_classes: int = 1000):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.LayerNorm(emb_size),
            nn.Linear(emb_size, n_classes))
```

Объединяем все части в полную структуру ViT:

```
class ViT(nn.Sequential):
    def __init__(self,
                 in_channels: int = 3,
                 patch_size: int = 16,
                 emb_size: int = 768,
                 img_size: int = 224,
                 depth: int = 12,
                 n_classes: int = 1000,
                 **kwargs):
        super().__init__(
            PatchEmbedding(in_channels, patch_size, emb_size),
            TransformerEncoder(depth, emb_size=emb_size, **kwargs),
            ClassificationHead(emb_size, n_classes))
```

Смотрим на полученную модель:

```
summary(ViT(), (3, 224, 224))
```

Определяем класс изображения:


```
model = ViT()
model(x).argmax().item()
```


Далее рассмотрим полный цикл обучения модели Vit на примере Dataset for Snacks. Так как созданную нами модель тяжело обучать с нуля из-за потребности большого количества ресурсов и данных, воспользуемся уже предобученной моделью Vit на ImageNet-21K, который состоит примерно из 14 миллионов изображений и 21 843 классов, где каждое изображение в наборе данных имеет размер 224 x 224 пикселя.

Загружаем набор данных со снеками:

```
!pip install -q datasets

from datasets import load_dataset
dataset = load_dataset("Matthijs/snacks")
print(dataset)

Dataset snacks downloaded and prepared to /root/.cache/huggingface/datasets/Matthijs/
100%  3/3 [00:00<00:00, 114.31it/s]

DatasetDict({
  train: Dataset({
    features: ['image', 'label'],
    num_rows: 4838
  })
  test: Dataset({
    features: ['image', 'label'],
    num_rows: 952
  })
  validation: Dataset({
    features: ['image', 'label'],
    num_rows: 955
  })
})
```

Создаем словарь, в котором сопоставим название снека с числом (для проведения классификации):

```
labels = dataset["train"].features["label"].names
num_labels = len(dataset["train"].features["label"].names)
label2id, id2label = dict(), dict()
for i, label in enumerate(labels):
    label2id[label] = i
    id2label[i] = label

print(label2id)
print(id2label)
```

```
{'apple': 0, 'banana': 1, 'cake': 2, 'candy': 3, 'carrot': 4,
{0: 'apple', 1: 'banana', 2: 'cake', 3: 'candy', 4: 'carrot',
```

Демонстрируем на созданном примере процесс разбиения входного изображения на патчи со свёрточным слоем:

```

import torch
import torch.nn as nn

# Создадим изображение для прогонки через модель, чтобы проверить что все
размерности корректны
toy_img = torch.rand(1, 3, 48, 48)

# Определим параметры для сверточной сети
num_channels = 3
hidden_size = 768
patch_size = 16

# Conv 2D - слой свертки
projection = nn.Conv2d(num_channels, hidden_size, kernel_size=patch_size,
                        stride=patch_size)

# Подаем картинку в модель, чтобы посмотреть, что она определена корректно
out_projection = projection(toy_img)

print(f'Original image size: {toy_img.size()}')
print(f'Size after projection: {out_projection.size()}')

Original image size: torch.Size([1, 3, 48, 48])
Size after projection: torch.Size([1, 768, 3, 3])

```

Распрямляем изображение после прогонки через сверточный слой:

```

patch_embeddings = out_projection.flatten(2).transpose(1, 2)
print(f'Patch embedding size: {patch_embeddings.size()}')

Patch embedding size: torch.Size([1, 9, 768])

# Определим токен [CLS], он необходим для корректной подачи данных в
модель трансформера
batch_size = 1
cls_token = nn.Parameter(torch.randn(1, 1, hidden_size))
cls_tokens = cls_token.expand(batch_size, -1, -1)

# Добавим токен к вектору изображения
patch_embeddings = torch.cat((cls_tokens, patch_embeddings), dim=1)
print(f'Patch embedding size: {patch_embeddings.size()}')

Patch embedding size: torch.Size([1, 10, 768])

# Определим эмбединги с информацией о позиции патча (кусочка картинки) в
исходном изображении
position_embeddings = nn.Parameter(torch.randn(batch_size, 10,
hidden_size))

# Добавим эмбединг к вектору изображения
input_embeddings = patch_embeddings + position_embeddings
print(f'Input embedding size: {input_embeddings.size()}')

```

```

# Определим параметры для модели ViT
num_heads = 12
num_layers = 12

# Реализуем один блок энкодера
transformer_encoder_layer = nn.TransformerEncoderLayer(
    d_model=hidden_size, nhead=num_heads,
    dim_feedforward=int(hidden_size * 4),
    dropout=0.1)

# Объединим несколько блоков для построения модели
transformer_encoder = nn.TransformerEncoder(
    encoder_layer=transformer_encoder_layer,
    num_layers=num_layers)

# Подадим тестовую картинку в модель для проверки корректности
output_embeddings = transformer_encoder(input_embeddings)
print(f' Output embedding size: {output_embeddings.size()}')

```

```
Output embedding size: torch.Size([1, 10, 768])
```

На данном этапе мы самостоятельно построили модель ViT, однако так как модель достаточно большая для ее обучения потребуется значительное количество вычислительных ресурсов и данных, поэтому для классификации рассматриваемого датасета со снеками возьмем предобученную модель, которую сможем дообучить на нашем датасете.

```

!pip install transformers

from transformers import ViTModel

# Загружаем модель
model_checkpoint = 'google/vit-base-patch16-224-in21k'
model = ViTModel.from_pretrained(model_checkpoint,
    add_pooling_layer=False)

# Пример входного изображения для проверки корректности скачивания
input_img = torch.rand(batch_size, num_channels, 224, 224)

# Прогоняем картинку через модель
output_embedding = model(input_img)
print(output_embedding)
print(f"Output embedding size:
{output_embedding['last_hidden_state'].size()}")

```

```

num_labels = 20
# Определяем классификатор, который будет состоять из одного линейного
слоя (всего классифицируем на 20 классов)
classifier = nn.Linear(hidden_size, num_labels)

```

```
# Проверяем, что задали слой верно
output_classification =
classifier(output_embedding['last_hidden_state'][:, 0, :])
print(f"Output embedding size: {output_classification.size()}")
```

```
Output embedding size: torch.Size([1, 20])
```

Импортируем библиотеки для работы с массивами, тензорами, моделью ViT и изображениями:

```
import numpy as np
import torch
import cv2
import torch.nn as nn
from transformers import ViTModel, ViTConfig
from torchvision import transforms
from torch.optim import Adam
from torch.utils.data import DataLoader
from tqdm import tqdm
```

Создаем класс Dataset, с помощью которого проведем предобработку изображений (обрежем и нормализуем), а также для формирования данных типа Dataset для более удобной подачи в модель:

```
class ImageDataset(torch.utils.data.Dataset):

    def __init__(self, input_data):
        self.input_data = input_data
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Resize((224, 224), antialias=True),
            transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                std=[0.5, 0.5, 0.5])
        ])

    def __len__(self):
        return len(self.input_data)

    def get_images(self, idx):
        return self.transform(self.input_data[idx]['image'])

    def get_labels(self, idx):
        return self.input_data[idx]['label']

    def __getitem__(self, idx):
        train_images = self.get_images(idx)
        train_labels = self.get_labels(idx)
        return train_images, train_labels
```

Добавим к предобученной части модели ViT построенный нами линейный слой классификатора:

```
class ViT(nn.Module):

    def __init__(self, config=ViTConfig(), num_labels=20,
                  model_checkpoint='google/vit-base-patch16-224-in21k'):

        super(ViT, self).__init__()

        self.vit = ViTModel.from_pretrained(model_checkpoint,
add_pooling_layer=False)
        self.classifier = (
            nn.Linear(config.hidden_size, num_labels)
        )
    def forward(self, x):

        x = self.vit(x)['last_hidden_state']
        output = self.classifier(x[:, 0, :])

        return output
```

Функция обучения для модели:

```
def model_train(dataset, epochs, learning_rate, bs):

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    # Загружаем модель на device (необходимо при использовании GPU),
определяем функцию потерь и оптимизатор
    model = ViT().to(device)
    criterion = nn.CrossEntropyLoss().to(device)
    optimizer = Adam(model.parameters(), lr=learning_rate)

    # Загружаем батч с изображениями
    train_dataset = ImageDataset(dataset)
    train_dataloader = DataLoader(train_dataset, num_workers=1,
batch_size=bs, shuffle=True)

    # Цикл дообучения (дообучение, так как мы взяли предобученную модель)
    for i in range(epochs):
        total_acc_train = 0
        total_loss_train = 0.0

        for train_image, train_label in tqdm(train_dataloader):
            output = model(train_image.to(device))
            loss = criterion(output, train_label.to(device))
            acc = (output.argmax(dim=1) ==
train_label.to(device)).sum().item()
```

```

        total_acc_train += acc
        total_loss_train += loss.item()

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    print(f'Epochs: {i + 1} | Loss: {total_loss_train /
len(train_dataset): .3f} | Accuracy: {total_acc_train /
len(train_dataset): .3f}')
```

return model

Гиперпараметры

EPOCHS = 10

LEARNING_RATE = 1e-4

BATCH_SIZE = 8

Обученная модель

trained_model = model_train(dataset['train'], EPOCHS, LEARNING_RATE, BATCH_SIZE)

```

100%|██████████| 605/605 [02:44<00:00, 3.67it/s]
Epochs: 1 | Loss: 0.123 | Accuracy: 0.838
100%|██████████| 605/605 [02:49<00:00, 3.56it/s]
Epochs: 2 | Loss: 0.030 | Accuracy: 0.945
100%|██████████| 605/605 [02:53<00:00, 3.50it/s]
Epochs: 3 | Loss: 0.018 | Accuracy: 0.966
100%|██████████| 605/605 [02:53<00:00, 3.48it/s]
Epochs: 4 | Loss: 0.013 | Accuracy: 0.972
100%|██████████| 605/605 [02:53<00:00, 3.48it/s]
Epochs: 5 | Loss: 0.012 | Accuracy: 0.974
100%|██████████| 605/605 [02:53<00:00, 3.48it/s]
Epochs: 6 | Loss: 0.008 | Accuracy: 0.981
100%|██████████| 605/605 [02:55<00:00, 3.44it/s]
Epochs: 7 | Loss: 0.015 | Accuracy: 0.968
100%|██████████| 605/605 [02:57<00:00, 3.41it/s]
Epochs: 8 | Loss: 0.005 | Accuracy: 0.991
100%|██████████| 605/605 [02:57<00:00, 3.41it/s]
Epochs: 9 | Loss: 0.009 | Accuracy: 0.980
100%|██████████| 605/605 [02:58<00:00, 3.40it/s] Epochs: 10 | Loss: 0.011 | Accuracy: 0.978
```

Функция позволяющая получить класс картинки, который предсказала обученная модель:

```

def predict(img):

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize((224, 224)),
        transforms.Normalize(mean=[0.5, 0.5, 0.5],
                               std=[0.5, 0.5, 0.5])
```

```
    ])\n\n    img = transform(img)\n    output = trained_model(img.unsqueeze(0).to(device))\n    prediction = output.argmax(dim=1).item()\n\n    return prediction
```

Проверка точности классификации на тестовой выборке:

```
sum = 0\nfor i in range(952):\n    if predict(dataset['test'][i]['image']) == dataset['test'][i]['label']:\n        sum += 1\n\nprint(f'Тестовая точность:{sum/952}')
```

Тестовая точность:0.8203781512605042

3. Задания

3.1 Задание № 1

Запустить описанную в примере модель и проверить ее работу на любом выбранном самостоятельно изображении.

3.2 Задание № 2

Реализовать цикл обучения рассматриваемой модели с использованием набора данных Dataset for Snacks, как было продемонстрировано в примере, сравнить результаты и сделать выводы о точности классификации:

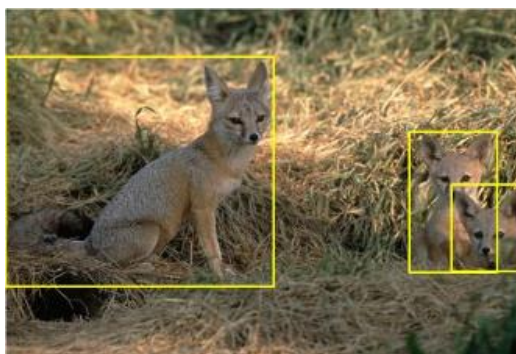


Dataset for Snacks

<https://huggingface.co/datasets/Matthijs/snacks>

3.3 Задание № 2

Реализовать цикл обучения рассматриваемой модели с использованием набора данных ImageNet:



Loading ImageNet

<https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data>

Для лучшего понимания структуры и работы Vision Transformer:

<https://arxiv.org/pdf/2010.11929.pdf>