KAN Kolmogorov Arnold Network Vision Transformer

Kolmogorov Arnold Network

相關參考資料

- 原始論文：**KAN: Kolmogorov-Arnold Network**
- 原始論文網址：https://arxiv.org/abs/2404.19756
- 原始開源代碼：https://github.com/KindXiaoming/pykan

BIBTEX 供論文引用

論文內容

📄 Summary

KAN（Kolmogorov-Arnold Network）論文基本概念。

1. Research Background
   - Kolmogorov-Arnold Representation Theorem（KART）：此定理說明任何多變量的連續函數都可以表示為單變量連續函數和加法操作的有限組合。
   - Limitations of Multi-Layer Perceptron（MLP）：傳統由 MLP 形成的神經網路雖然具有強大的表達能力，但在某些應用中存在固定的 Activation Function，使得其解釋性較差且參數效率低下。
2. Kolmogorov-Arnold Network（KAN）
   - Network Structure：與 MLP 不同，KAN 在 Edge（即權重）上使用可學習的 Activation Function，而不是在 Node（即神經元）上使用固定的 Activation Function。
   - Activation Function：KAN 中的每個權重參數被替換為一個參數化為樣條函數的單變量函數。節點只進行簡單的信號相加操作，不應用任何非線性操作。
3. Advantages of KAN
   - Higher Precision：KAN 在數據擬合和偏微分方程求解方面比 MLP 更準確。例如，在偏微分方程求解中，一個兩層且寬度為十的 KAN 比一個四層且寬度為一百的 MLP 準確度高百倍。
   - Explainability：KAN 可以直觀地可視化，並能與人類用戶進行互動，有助於科學家重新發現數學和物理定律。

📄 Summary

KAN（Kolmogorov-Arnold Network）和 MLP（Multi-Layer Perceptron）之間的主要差異與比較。

1. 激活函數的位置和特性
   - MLP：激活函數固定，位於節點（神經元）上；且激活函數一般是非線性函數，例如 ReLU、Sigmoid 等。
   - KAN：激活函數是可學習的，位於邊（權重）上；且每個權重參數被替換為一個參數化為樣條函數的單變量函數。
2. 網路結構和權重表示
   - MLP：使用線性權重矩陣進行計算，然後應用固定的非線性激活函數；節點進行非線性變換。
   - 結構公式：

$$MLP(x) = ((W3 \circ \sigma2 \circ W2 \circ \sigma1 \circ W1)(x))$$

   - KAN：沒有線性權重矩陣，所有權重都被樣條函數替代；節點僅進行簡單的信號相加操作，不進行非線性變換。

- 結構公式：

$$KAN(x) = ((\Phi3 \circ \Phi2 \circ \Phi1)(x))$$

3. 訓練方法和參數優化
   - MLP：權重矩陣通過梯度下降法進行訓練；訓練過程需要調整大量的線性權重參數。
   - KAN：核心在於樣條函數的參數化和學習；樣條函數通過調整其參數進行優化。
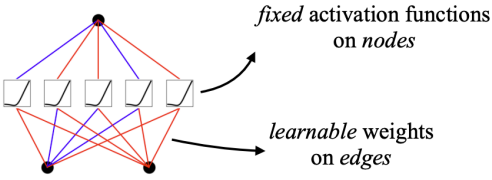4. 表達能力和適用範圍
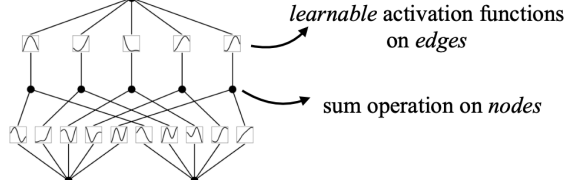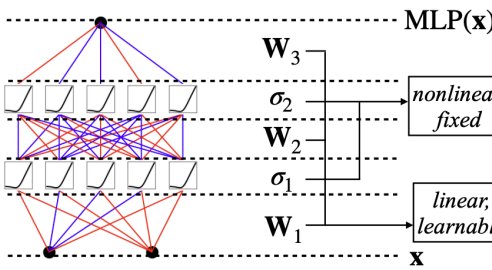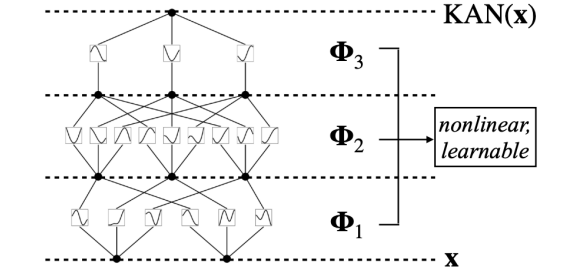   - MLP：基於普適近似定理，MLP 能夠逼近任意連續函數；常用於各種回歸和分類問題，但在高維數據下可能效率低下。
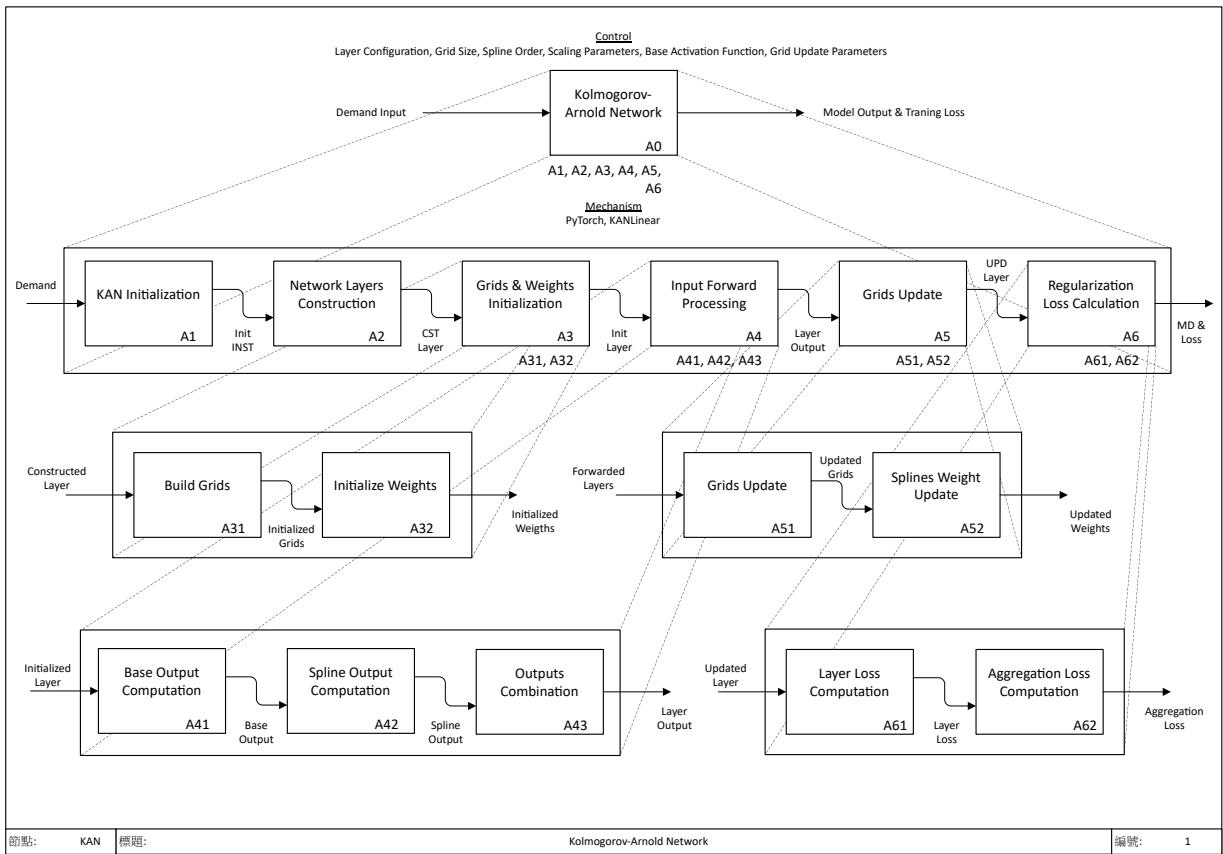   - KAN：基於 Kolmogorov-Arnold 表示定理，能夠表達高維數據的組合結構和單變量函數；對於需要高準確度和解釋性的應用，如數學模型和物理模型，有顯著優勢。
5. 可解釋性和直觀性
   - MLP：由於固定的激活函數和複雜的權重矩陣，MLP 在解釋性方面較為薄弱；解釋模型需要額外的工具和方法，如 SHAP、LIME 等。
   - KAN：由於激活函數是可學習的單變量函數，KAN 的結構更易於直觀理解；KAN 的節點僅進行信號相加，使得整體網路更易於可視化和解釋。
6. 計算和資源需求
   - MLP：訓練和推理過程中，計算資源需求較大，特別是在高維數據和大模型情況下。
   - KAN：由於樣條函數的引入，KAN 在同樣準確度下所需的參數和計算資源相對較少；能夠在較小的計算圖上達到與大型 MLP 相同甚至更好的準確度。

Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a) *fixed* activation functions on *nodes* / *learnable* weights on *edges*	(b) *learnable* activation functions on *edges* / sum operation on *nodes*
Formula (Deep)	$MLP(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$KAN(\mathbf{x}) = (\mathbf{\Phi}_3 \circ \mathbf{\Phi}_2 \circ \mathbf{\Phi}_1)(\mathbf{x})$
Model (Deep)	(c) MLP(x) $\mathbf{W}_3$ σ_2 $\mathbf{W}_2$ σ_1 $\mathbf{W}_1$ x / *nonlinear, fixed* / *linear, learnable*	(d) KAN(x) $\mathbf{\Phi}_3$ $\mathbf{\Phi}_2$ $\mathbf{\Phi}_1$ x / *nonlinear, learnable*

IDEF0 設計

Control
Layer Configuration, Grid Size, Spline Order, Scaling Parameters, Base Activation Function, Grid Update Parameters

Demand Input → Kolmogorov-Arnold Network A0 → Model Output & Traning Loss

A1, A2, A3, A4, A5, A6

Mechanism
PyTorch, KANLinear

Demand → KAN Initialization A1 → Init INST → Network Layers Construction A2 → CST Layer → Grids & Weights Initialization A3 → Init Layer → Input Forward Processing A4 → Layer Output → Grids Update A5 → UPD Layer → Regularization Loss Calculation A6 → MD & Loss

A31, A32 · A41, A42, A43 · A51, A52 · A61, A62

Constructed Layer → Build Grids A31 → Initialized Grids → Initialize Weights A32 → Initialized Weigths

Forwarded Layers → Grids Update A51 → Updated Grids → Splines Weight Update A52 → Updated Weights

Initialized Layer → Base Output Computation A41 → Base Output → Spline Output Computation A42 → Spline Output → Outputs Combination A43 → Layer Output

Updated Layer → Layer Loss Computation A61 → Layer Loss → Aggregation Loss Computation A62 → Aggregation Loss

| 節點: | KAN | 標題: | Kolmogorov-Arnold Network | 編號: | 1 |

GRAFCET 設計

0 — Instance Creation Complete
1 — KAN Initialization — Initialization Complete
2 — Network Layers Construction — KANLinear Initialization — Layers Construction Complete
3 — Grids & Weights Initialization — Grids & Weights Initialization Complete
4 — Input Forward Processing — All Layers Data Forward Processing Complete
5 — Grids Update — Grids Update Skip or Complete
6 — Regularization Loss Calculation — Regularization Loss Computation Complete

30 — Layers Construction Complete
31 — Build Grids — All Layers Grids Initialization Complete
32 — Initialize Weights — All Layers Weights Initialization Complete

40 — Grids & Weights Initialization Complete
41 — Base Output Computation — Base Output Calculation Complete
42 — Spline Output Computation — Spline Output Calculation Complete
43 — Outputs Combination — All Layers Outputs Combination Complete

50 — All Layers Data Forward Processing Complete
51 — Grids Update — New Grid Update Complete
52 — Splines Weight Update — New Spline Weight Update Complete

60 — Grids Update Complete
61 — Layer Loss Computation — All Layers Regularization Losses Calculation Complete
62 — Aggregation Loss Computation — Regularization Loss Calculation Complete

| 節點: | KAN | 標題: | Kolmogorov-Arnold Network | 編號: | 2 |

Python PyTorch 模擬驗證

- 基於 IDEF0 和 Grafcet 重構後之 Kolmogorov-Arnold Network

> **ⓘ Info**
>
> 基於 Grafcet 設計後的一些調整
>
> - Original Implementation 的 Performance 問題主要在於需要展開所有中間變數來執行不同的 Activation Function。對於具有 `in_features` 輸入和 `out_features` 輸出的層,原始實現需要將輸入展開為形狀為 `(batch_size, out_features, in_features)` 的 Tensor 來執行 Activation Function。然而,所有的 Activation Function 都是一組固定 Base Function (B-Spline) 的線性組合,也就可以將計算重新整理為使用不同的 Base Function 激活輸入,然後將它們線性組合。

```python
import math

import torch
import torch.nn.functional as F


class KANLinear(torch.nn.Module):
    def __init__(
            self,
            in_features,
            out_features,
            grid_size=5,
            spline_order=3,
            scale_base=1.0,
            scale_spline=1.0,
            enable_standalone_scale_spline=True,
            base_activation=torch.nn.SiLU,
            grid_eps=0.02,
            grid_range=[-1, 1],
    ):
        super(KANLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.grid_size = grid_size
        self.spline_order = spline_order

        # 構建網格點
        self.grid = self.build_grid(grid_range, grid_size, spline_order)

        # 初始化基礎權重和樣條權重
        self.base_weight, self.spline_weight, self.spline_scaler = self.initialize_weights(
            out_features, in_features, grid_size, spline_order, scale_base, scale_spline,
enable_standalone_scale_spline
        )

        self.scale_base = scale_base
        self.scale_spline = scale_spline
        self.enable_standalone_scale_spline = enable_standalone_scale_spline
        self.base_activation = base_activation()
        self.grid_eps = grid_eps

    def build_grid(self, grid_range, grid_size, spline_order):
        h = (grid_range[1] - grid_range[0]) / grid_size
        grid = (
            (
                    torch.arange(-spline_order, grid_size + spline_order + 1) * h
                    + grid_range[0]
            )
            .expand(self.in_features, -1)
            .contiguous()
        )
        return grid

    def initialize_weights(self, out_features, in_features, grid_size, spline_order, scale_base, scale_spline,
                            enable_standalone_scale_spline):
        base_weight = torch.nn.Parameter(torch.Tensor(out_features, in_features))
        spline_weight = torch.nn.Parameter(
            torch.Tensor(out_features, in_features, grid_size + spline_order)
        )
```

```python
        if enable_standalone_scale_spline:
            spline_scaler = torch.nn.Parameter(
                torch.Tensor(out_features, in_features)
            )
        else:
            spline_scaler = None
        torch.nn.init.kaiming_uniform_(base_weight, a=math.sqrt(5) * scale_base)
        torch.nn.init.kaiming_uniform_(spline_weight, a=math.sqrt(5) * scale_spline)
        if enable_standalone_scale_spline:
            torch.nn.init.kaiming_uniform_(spline_scaler, a=math.sqrt(5) * scale_spline)
        return base_weight, spline_weight, spline_scaler

    def b_splines(self, x: torch.Tensor):
        bases = self.calculate_b_spline_bases(x)
        return bases.contiguous()

    def calculate_b_spline_bases(self, x: torch.Tensor):
        grid: torch.Tensor = (
            self.grid
        )  # (in_features, grid_size + 2 * spline_order + 1)
        x = x.unsqueeze(-1)
        bases = ((x >= grid[:, :-1]) & (x < grid[:, 1:])).to(x.dtype)
        for k in range(1, self.spline_order + 1):
            bases = (
                        (x - grid[:, : -(k + 1)])
                        / (grid[:, k:-1] - grid[:, : -(k + 1)])
                        * bases[:, :, :-1]
                ) + (
                        (grid[:, k + 1:] - x)
                        / (grid[:, k + 1:] - grid[:, 1:(-k)])
                        * bases[:, :, 1:]
                )
        return bases

    def curve2coeff(self, x: torch.Tensor, y: torch.Tensor):
        A = self.b_splines(x).transpose(
            0, 1
        )  # (in_features, batch_size, grid_size + spline_order)
        B = y.transpose(0, 1)  # (in_features, batch_size, out_features)
        solution = torch.linalg.lstsq(
            A, B
        ).solution  # (in_features, grid_size + spline_order, out_features)
        result = solution.permute(
            2, 0, 1
        )  # (out_features, in_features, grid_size + spline_order)
        return result.contiguous()

    @property
    def scaled_spline_weight(self):
        if self.enable_standalone_scale_spline:
            return self.spline_weight * self.spline_scaler.unsqueeze(-1)
        else:
            return self.spline_weight

    def forward(self, x: torch.Tensor):
        base_output = self.compute_base_output(x)
        spline_output = self.compute_spline_output(x)
        return base_output + spline_output

    def compute_base_output(self, x: torch.Tensor):
        return F.linear(self.base_activation(x), self.base_weight)

    def compute_spline_output(self, x: torch.Tensor):
        return F.linear(
            self.b_splines(x).view(x.size(0), -1),
            self.scaled_spline_weight.view(self.out_features, -1),
        )

    @torch.no_grad()
    def update_grid(self, x: torch.Tensor, margin=0.01):
        batch = x.size(0)
```

```python
        splines = self.b_splines(x)  # (batch, in, coeff)
        splines = splines.permute(1, 0, 2)  # (in, batch, coeff)
        orig_coeff = self.scaled_spline_weight  # (out, in, coeff)
        orig_coeff = orig_coeff.permute(1, 2, 0)  # (in, coeff, out)
        unreduced_spline_output = torch.bmm(splines, orig_coeff)  # (in, batch, out)
        unreduced_spline_output = unreduced_spline_output.permute(
            1, 0, 2
        )  # (batch, in, out)

        x_sorted = torch.sort(x, dim=0)[0]
        grid_adaptive = x_sorted[
            torch.linspace(
                0, batch - 1, self.grid_size + 1, dtype=torch.int64, device=x.device
            )
        ]

        uniform_step = (x_sorted[-1] - x_sorted[0] + 2 * margin) / self.grid_size
        grid_uniform = (
                torch.arange(
                    self.grid_size + 1, dtype=torch.float32, device=x.device
                ).unsqueeze(1)
                * uniform_step
                + x_sorted[0]
                - margin
        )

        grid = self.grid_eps * grid_uniform + (1 - self.grid_eps) * grid_adaptive
        grid = torch.concatenate(
            [
                grid[:1]
                - uniform_step
                * torch.arange(self.spline_order, 0, -1, device=x.device).unsqueeze(1),
                grid,
                grid[-1:]
                + uniform_step
                * torch.arange(1, self.spline_order + 1, device=x.device).unsqueeze(1),
            ],
            dim=0,
        )

        self.grid.copy_(grid.T)
        self.spline_weight.data.copy_(self.curve2coeff(x, unreduced_spline_output))

    def regularization_loss(self, regularize_activation=1.0, regularize_entropy=1.0):
        l1_fake = self.spline_weight.abs().mean(-1)
        regularization_loss_activation = l1_fake.sum()
        p = l1_fake / regularization_loss_activation
        regularization_loss_entropy = -torch.sum(p * p.log())
        return (
                regularize_activation * regularization_loss_activation
                + regularize_entropy * regularization_loss_entropy
        )


class KAN(torch.nn.Module):
    def __init__(
            self,
            layers_hidden,
            grid_size=5,
            spline_order=3,
            scale_base=1.0,
            scale_spline=1.0,
            base_activation=torch.nn.SiLU,
            grid_eps=0.02,
            grid_range=[-1, 1],
    ):
        super(KAN, self).__init__()
        self.grid_size = grid_size
        self.spline_order = spline_order
```

```python
        # 構建 KAN 的層
        self.layers = self.build_layers(
            layers_hidden, grid_size, spline_order, scale_base, scale_spline, base_activation, grid_eps,
grid_range
        )

    def build_layers(self, layers_hidden, grid_size, spline_order, scale_base, scale_spline, base_activation,
grid_eps,
                     grid_range):
        layers = torch.nn.ModuleList()
        for in_features, out_features in zip(layers_hidden, layers_hidden[1:]):
            layers.append(
                KANLinear(
                    in_features,
                    out_features,
                    grid_size=grid_size,
                    spline_order=spline_order,
                    scale_base=scale_base,
                    scale_spline=scale_spline,
                    base_activation=base_activation,
                    grid_eps=grid_eps,
                    grid_range=grid_range,
                )
            )
        return layers

    def forward(self, x: torch.Tensor, update_grid=False):
        for layer in self.layers:
            if update_grid:
                layer.update_grid(x)
            x = layer(x)
        return x

    def regularization_loss(self, regularize_activation=1.0, regularize_entropy=1.0):
        return sum(
            layer.regularization_loss(regularize_activation, regularize_entropy)
            for layer in self.layers
        )
```

Transformer

相關參考資料

- 原始論文：**Attention is All You Need**
- 原始論文網址：https://arxiv.org/abs/1706.03762
- 原始開源代碼：https://github.com/huggingface/transformers

BIBTEX 供論文引用（HuggingFace Library）

```
@inproceedings{wolf-etal-2020-transformers,
    title = "Transformers: State-of-the-Art Natural Language Processing",
    author = "Thomas Wolf and Lysandre Debut and Victor Sanh and Julien Chaumond and Clement Delangue and Anthony
Moi and Pierric Cistac and Tim Rault and Rémi Louf and Morgan Funtowicz and Joe Davison and Sam Shleifer and
Patrick von Platen and Clara Ma and Yacine Jernite and Julien Plu and Canwen Xu and Teven Le Scao and Sylvain
Gugger and Mariama Drame and Quentin Lhoest and Alexander M. Rush",
    booktitle = "Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System
Demonstrations",
    month = oct,
    year = "2020",
    address = "Online",
    publisher = "Association for Computational Linguistics",
    url = "https://www.aclweb.org/anthology/2020.emnlp-demos.6",
    pages = "38--45"
}
```

論文內容

- 全局分析
- 位置編碼
- 多頭注意力機制
- 殘差
- Batch Normal
- Layer Normal
- Decoder

Vision Transformer (ViT)

相關參考資料

- 原始論文：`An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale`
- 原始論文網址：https://arxiv.org/abs/2010.11929
- 原始開源代碼：https://github.com/google-research/vision_transformer

BIBTEX 供論文引用

```
@article{dosovitskiy2020vit,
  title={An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale},
  author={Dosovitskiy, Alexey and Beyer, Lucas and Kolesnikov, Alexander and Weissenborn, Dirk and Zhai, Xiaohua
and Unterthiner, Thomas and  Dehghani, Mostafa and Minderer, Matthias and Heigold, Georg and Gelly, Sylvain and
Uszkoreit, Jakob and Houlsby, Neil},
  journal={ICLR},
  year={2021}
}
```

論文內容

ViT PyTorch 重構

相關說明

> **ⓘ Info**
>
> 代碼使用流程介紹

1. 下載好資料集（Dataset），目前代碼預設（Default）用的是花的分類（`tf_flowers`）：
   https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
2. 在 `train.py` 中把 `--data-path` 設定為解壓縮後的 `flower_photos` 文件夾的絕對路徑。
3. 下載預訓練權重（Pre-Trained Weight），在 `vit_model.py` 中，每個模型都有提供預訓練權重的下載地址，根據選擇的模型下載對應的預訓練權重。
4. 在 `train.py` 腳本中將 `--weights` 參數設定為下載好的預訓練權重路徑。
5. 設定好資料集的路徑 `--data-path` 以及預訓練權重的路徑 `--weights`，即可使用 `train.py` 開始訓練（訓練過程中會自動生成 `class_indices.json` 文件）。
6. 在 `predict.py` 中導入與訓練代碼中相同的模型，並將 `model_weight_path` 設定為訓練好的模型權重路徑（預設保存在 `weights` 文件夾下）。
7. 在 `predict.py` 中將 `img_path` 設定為你需要預測的圖片的絕對路徑。
8. 設定好權重路徑 `model_weight_path` 和預測圖片路徑 `img_path`，即可使用 `predict.py` 進行預測。
9. 如果要使用其他資料集，就比照預設採用的「花的分類」之文件結構進行擺放（即一個類別對應一個文件夾），並且將訓練及預測代碼中的 `num_classes` 設定為你自己的數據的類別數。

> **❞Quote**

額外優化參考論文：**Bag of Tricks for Image Classification with Convolutional Neural Networks**
額外優化參考論文網址：https://arxiv.org/abs/1812.01187

```
@misc{1812.01187,
Author = {Tong He and Zhi Zhang and Hang Zhang and Zhongyue Zhang and Junyuan Xie and Mu Li},
Title = {Bag of Tricks for Image Classification with Convolutional Neural Networks},
Year = {2018},
Eprint = {arXiv:1812.01187},
}
```

代碼實現

> 💧 **Important**
>
> PyTorch 重構後之所有 Vision Transformer 代碼
>
> - 代碼倉庫：https://github.com/toby0622/Kolmogorov-Arnold-Network-Vision-Transformer (**Private**)

- my_dateset.py

```python
import torch
from PIL import Image
from torch.utils.data import Dataset


# 自定義資料集
class MyDataSet(Dataset):
    # 初始化資料集
    def __init__(self, images_path: list, images_class: list, transform=None):
        self.images_path = images_path
        self.images_class = images_class
        self.transform = transform

    # 返回資料集圖像的數量
    def __len__(self):
        return len(self.images_path)

    # 返回資料集中的圖像和標籤
    def __getitem__(self, item):
        img = Image.open(self.images_path[item])
        # RGB 為彩色圖片，L 為灰度圖片
        if img.mode != "RGB":
            raise ValueError("image: {} isn't RGB mode.".format(self.images_path[item]))

        label = self.images_class[item]

        if self.transform is not None:
            img = self.transform(img)

        return img, label

    # 將資料集的圖像和標籤組合成一個 batch
    @staticmethod
    def collate_fn(batch):
        # PyTorch 官方實現的 default_collate 參考下列網址
        # https://github.com/pytorch/pytorch/blob/67b7e751e6b5931a9f45274653f4f653a4e6cdf6/torch/utils/data/_utils/collate.py
        images, labels = tuple(zip(*batch))

        images = torch.stack(images, dim=0)
        labels = torch.as_tensor(labels)

        return images, labels
```

- `vit_model.py`

```python
import torch
import torch.nn as nn
from functools import partial
from collections import OrderedDict


def drop_path(x, drop_prob: float = 0.0, training: bool = False):
    if drop_prob == 0.0 or not training:
        return x

    keep_prob = 1 - drop_prob
    shape = (x.shape[0],) + (1,) * (
        x.ndim - 1
    )  # 適用於不同維度的張量，而不僅僅是 2D 卷積網絡
    random_tensor = keep_prob + torch.rand(shape, dtype=x.dtype, device=x.device)
    random_tensor.floor_()  # 二值化
    output = x.div(keep_prob) * random_tensor

    return output


# Drop paths (Stochastic Depth) per sample (when applied in main path of residual blocks)
class DropPath(nn.Module):
    def __init__(self, drop_prob=None):
        super(DropPath, self).__init__()
        self.drop_prob = drop_prob

    def forward(self, x):
        return drop_path(x, self.drop_prob, self.training)


# 2D Image to Patch Embedding
class PatchEmbed(nn.Module):
    def __init__(
        self, img_size=224, patch_size=16, in_c=3, embed_dim=768, norm_layer=None
    ):
        super().__init__()
        img_size = (img_size, img_size)
        patch_size = (patch_size, patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.grid_size = (img_size[0] // patch_size[0], img_size[1] // patch_size[1])
        self.num_patches = self.grid_size[0] * self.grid_size[1]

        self.proj = nn.Conv2d(
            in_c, embed_dim, kernel_size=patch_size, stride=patch_size
        )
        self.norm = norm_layer(embed_dim) if norm_layer else nn.Identity()

    def forward(self, x):
        B, C, H, W = x.shape
        assert (
            H == self.img_size[0] and W == self.img_size[1]
        ), f"輸入圖像大小 ({H}*{W}) 與模型大小 ({self.img_size[0]}*{self.img_size[1]}) 不匹配。"

        # flatten: [B, C, H, W] → [B, C, HW]
        # transpose: [B, C, HW] → [B, HW, C]
        x = self.proj(x).flatten(2).transpose(1, 2)
        x = self.norm(x)
        return x


class Attention(nn.Module):
    def __init__(
        self,
        dim,  # 輸入 token 的維度
        num_heads=8,
        qkv_bias=False,
```

```python
            qk_scale=None,
            attn_drop_ratio=0.0,
            proj_drop_ratio=0.0,
    ):
        super(Attention, self).__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop_ratio)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop_ratio)

    def forward(self, x):
        # [batch_size, num_patches + 1, total_embed_dim]
        B, N, C = x.shape

        # qkv(): → [batch_size, num_patches + 1, 3 * total_embed_dim]
        # reshape: → [batch_size, num_patches + 1, 3, num_heads, embed_dim_per_head]
        # permute: → [3, batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        qkv = (
            self.qkv(x)
            .reshape(B, N, 3, self.num_heads, C // self.num_heads)
            .permute(2, 0, 3, 1, 4)
        )
        # [batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        q, k, v = (
            qkv[0],
            qkv[1],
            qkv[2],
        )  # make torchscript happy (cannot use tensor as tuple)

        # transpose: → [batch_size, num_heads, embed_dim_per_head, num_patches + 1]
        # @: multiply → [batch_size, num_heads, num_patches + 1, num_patches + 1]
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        # @: multiply → [batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        # transpose: → [batch_size, num_patches + 1, num_heads, embed_dim_per_head]
        # reshape: → [batch_size, num_patches + 1, total_embed_dim]
        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x


# Vision Transformer 使用的 MLP
class Mlp(nn.Module):
    def __init__(
        self,
        in_features,
        hidden_features=None,
        out_features=None,
        act_layer=nn.GELU,
        drop=0.0,
    ):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
```

```python
        return x


class Block(nn.Module):
    def __init__(
        self,
        dim,
        num_heads,
        mlp_ratio=4.0,
        qkv_bias=False,
        qk_scale=None,
        drop_ratio=0.0,
        attn_drop_ratio=0.0,
        drop_path_ratio=0.0,
        act_layer=nn.GELU,
        norm_layer=nn.LayerNorm,
    ):
        super(Block, self).__init__()
        self.norm1 = norm_layer(dim)
        self.attn = Attention(
            dim,
            num_heads=num_heads,
            qkv_bias=qkv_bias,
            qk_scale=qk_scale,
            attn_drop_ratio=attn_drop_ratio,
            proj_drop_ratio=drop_ratio,
        )
        # 注意：隨機深度的丟棄路徑，我們將看看這是否比 dropout 更好
        self.drop_path = (
            DropPath(drop_path_ratio) if drop_path_ratio > 0.0 else nn.Identity()
        )
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(
            in_features=dim,
            hidden_features=mlp_hidden_dim,
            act_layer=act_layer,
            drop=drop_ratio,
        )

    def forward(self, x):
        x = x + self.drop_path(self.attn(self.norm1(x)))
        x = x + self.drop_path(self.mlp(self.norm2(x)))
        return x


class VisionTransformer(nn.Module):
    def __init__(
        self,
        img_size=224,
        patch_size=16,
        in_c=3,
        num_classes=1000,
        embed_dim=768,
        depth=12,
        num_heads=12,
        mlp_ratio=4.0,
        qkv_bias=True,
        qk_scale=None,
        representation_size=None,
        distilled=False,
        drop_ratio=0.0,
        attn_drop_ratio=0.0,
        drop_path_ratio=0.0,
        embed_layer=PatchEmbed,
        norm_layer=None,
        act_layer=None,
    ):
        """
        Args:
            img_size (int, tuple): input image size
```

```
            patch_size (int, tuple): patch size
            in_c (int): number of input channels
            num_classes (int): number of classes for classification head
            embed_dim (int): embedding dimension
            depth (int): depth of transformer
            num_heads (int): number of attention heads
            mlp_ratio (int): ratio of mlp hidden dim to embedding dim
            qkv_bias (bool): enable bias for qkv if True
            qk_scale (float): override default qk scale of head_dim ** -0.5 if set
            representation_size (Optional[int]): enable and set representation layer (pre-logits) to this value
if set
            distilled (bool): model includes a distillation token and head as in DeiT models
            drop_ratio (float): dropout rate
            attn_drop_ratio (float): attention dropout rate
            drop_path_ratio (float): stochastic depth rate
            embed_layer (nn.Module): patch embedding layer
            norm_layer: (nn.Module): normalization layer
        """
        super(VisionTransformer, self).__init__()
        self.num_classes = num_classes
        self.num_features = self.embed_dim = (
            embed_dim  # 為了與其他模型一致 · 使用 num_features
        )
        self.num_tokens = 2 if distilled else 1
        norm_layer = norm_layer or partial(nn.LayerNorm, eps=1e-6)
        act_layer = act_layer or nn.GELU

        self.patch_embed = embed_layer(
            img_size=img_size, patch_size=patch_size, in_c=in_c, embed_dim=embed_dim
        )
        num_patches = self.patch_embed.num_patches

        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.dist_token = (
            nn.Parameter(torch.zeros(1, 1, embed_dim)) if distilled else None
        )
        self.pos_embed = nn.Parameter(
            torch.zeros(1, num_patches + self.num_tokens, embed_dim)
        )
        self.pos_drop = nn.Dropout(p=drop_ratio)

        dpr = [
            x.item() for x in torch.linspace(0, drop_path_ratio, depth)
        ]  # 隨機深度衰減規則
        self.blocks = nn.Sequential(
            *[
                Block(
                    dim=embed_dim,
                    num_heads=num_heads,
                    mlp_ratio=mlp_ratio,
                    qkv_bias=qkv_bias,
                    qk_scale=qk_scale,
                    drop_ratio=drop_ratio,
                    attn_drop_ratio=attn_drop_ratio,
                    drop_path_ratio=dpr[i],
                    norm_layer=norm_layer,
                    act_layer=act_layer,
                )
                for i in range(depth)
            ]
        )
        self.norm = norm_layer(embed_dim)

        # 表示層
        if representation_size and not distilled:
            self.has_logits = True
            self.num_features = representation_size
            self.pre_logits = nn.Sequential(
                OrderedDict(
                    [
                        ("fc", nn.Linear(embed_dim, representation_size)),
```

```python
                    ("act", nn.Tanh())),
                ]
            )
        )
    else:
        self.has_logits = False
        self.pre_logits = nn.Identity()

    # 分類頭
    self.head = (
        nn.Linear(self.num_features, num_classes)
        if num_classes > 0
        else nn.Identity()
    )
    self.head_dist = None
    if distilled:
        self.head_dist = (
            nn.Linear(self.embed_dim, self.num_classes)
            if num_classes > 0
            else nn.Identity()
        )

    # 權重初始化
    nn.init.trunc_normal_(self.pos_embed, std=0.02)
    if self.dist_token is not None:
        nn.init.trunc_normal_(self.dist_token, std=0.02)

    nn.init.trunc_normal_(self.cls_token, std=0.02)
    self.apply(_init_vit_weights)

def forward_features(self, x):
    # [B, C, H, W] → [B, num_patches, embed_dim]
    x = self.patch_embed(x)  # [B, 196, 768]
    # [1, 1, 768] → [B, 1, 768]
    cls_token = self.cls_token.expand(x.shape[0], -1, -1)
    if self.dist_token is None:
        x = torch.cat((cls_token, x), dim=1)  # [B, 197, 768]
    else:
        x = torch.cat(
            (cls_token, self.dist_token.expand(x.shape[0], -1, -1), x), dim=1
        )

    x = self.pos_drop(x + self.pos_embed)
    x = self.blocks(x)
    x = self.norm(x)
    if self.dist_token is None:
        return self.pre_logits(x[:, 0])
    else:
        return x[:, 0], x[:, 1]

def forward(self, x):
    x = self.forward_features(x)
    if self.head_dist is not None:
        x, x_dist = self.head(x[0]), self.head_dist(x[1])
        if self.training and not torch.jit.is_scripting():
            # 推理期間,返回兩個分類器預測的平均值
            return x, x_dist
        else:
            return (x + x_dist) / 2
    else:
        x = self.head(x)
    return x


# ViT 權重初始化
def _init_vit_weights(m):
    # parameter m = module
    if isinstance(m, nn.Linear):
        nn.init.trunc_normal_(m.weight, std=0.01)
        if m.bias is not None:
            nn.init.zeros_(m.bias)
```

```python
        elif isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode="fan_out")
            if m.bias is not None:
                nn.init.zeros_(m.bias)
        elif isinstance(m, nn.LayerNorm):
            nn.init.zeros_(m.bias)
            nn.init.ones_(m.weight)


def vit_base_patch16_224(num_classes: int = 1000):
    # ViT-Base model (ViT-B/16) ImageNet-1k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=None,
        num_classes=num_classes,
    )

    return model


def vit_base_patch16_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Base model (ViT-B/16) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=768 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_base_patch32_224(num_classes: int = 1000):
    # ViT-Base model (ViT-B/32) ImageNet-1k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=32,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=None,
        num_classes=num_classes,
    )

    return model


def vit_base_patch32_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Base model (ViT-B/32) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=32,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=768 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_large_patch16_224(num_classes: int = 1000):
    # ViT-Large model (ViT-L/16) ImageNet-1k weights @ 224x224
```

```python
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        representation_size=None,
        num_classes=num_classes,
    )

    return model


def vit_large_patch16_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Large model (ViT-L/16) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        representation_size=1024 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_large_patch32_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Large model (ViT-L/32) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=32,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        representation_size=1024 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_huge_patch14_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Huge model (ViT-H/14) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=14,
        embed_dim=1280,
        depth=32,
        num_heads=16,
        representation_size=1280 if has_logits else None,
        num_classes=num_classes,
    )

    return model
```

- utils.py

```python
import os
import sys
import json
import pickle
import random
import torch
import matplotlib.pyplot as plt
from tqdm import tqdm


def read_split_data(root: str, val_rate: float = 0.2):
```

```python
    random.seed(112522083)  # 保證每次運行的隨機結果一致
    assert os.path.exists(root), "dataset root: {} does not exist.".format(root)

    # 遍歷所有文件夾，每個文件夾代表一個類別
    flower_class = [
        cla for cla in os.listdir(root) if os.path.isdir(os.path.join(root, cla))
    ]

    # 排序，保證各平台順序一致
    flower_class.sort()

    # 生成類別名稱與對應的索引
    class_indices = dict((k, v) for v, k in enumerate(flower_class))
    json_str = json.dumps(
        dict((val, key) for key, val in class_indices.items()), indent=4
    )

    with open("class_indices.json", "w") as json_file:
        json_file.write(json_str)

    train_images_path = []   # 儲存訓練集的所有圖片路徑
    train_images_label = []  # 儲存訓練集圖片對應索引
    val_images_path = []   # 儲存驗證集的所有圖片路徑
    val_images_label = []  # 儲存驗證集圖片對應索引
    every_class_num = []  # 儲存每個類別的樣本數
    supported = [".jpg", ".JPG", ".png", ".PNG"]  # 支持的圖片格式

    # 遍歷每個 Folder 的 File
    for cla in flower_class:
        cla_path = os.path.join(root, cla)

        # 遍歷獲取 supported 支持的所有文件路徑
        images = [
            os.path.join(root, cla, i)
            for i in os.listdir(cla_path)
            if os.path.splitext(i)[-1] in supported
        ]

        # 排序，保證各平台順序一致
        images.sort()

        # 獲取該類別對應的索引
        image_class = class_indices[cla]

        # 紀錄該類別的樣本數
        every_class_num.append(len(images))

        # 依照比例隨機抽樣驗證集樣本
        val_path = random.sample(images, k=int(len(images) * val_rate))

        for img_path in images:
            # 如果該路徑在採樣的驗證集樣本中則存入驗證集
            if img_path in val_path:
                val_images_path.append(img_path)
                val_images_label.append(image_class)
            # 否則存入訓練集
            else:
                train_images_path.append(img_path)
                train_images_label.append(image_class)

    print("{} images were found in the dataset.".format(sum(every_class_num)))
    print("{} images for training.".format(len(train_images_path)))
    print("{} images for validation.".format(len(val_images_path)))
    assert len(train_images_path) > 0, "number of training images must greater than 0."
    assert len(val_images_path) > 0, "number of validation images must greater than 0."

    plot_image = False

    if plot_image:
        # 繪製每個類別的樣本數柱狀圖
        plt.bar(range(len(flower_class)), every_class_num, align="center")
```

```python
        # 將橫坐標 0, 1, 2, 3, 4 替換為相應的類別名稱
        plt.xticks(range(len(flower_class)), flower_class)

        # 在柱狀圖上添加數值標籤
        for i, v in enumerate(every_class_num):
            plt.text(x=i, y=v + 5, s=str(v), ha="center")

        # 設置 x 坐標
        plt.xlabel("image class")
        # 設置 y 坐標
        plt.ylabel("number of images")
        # 設置柱狀圖的標題
        plt.title("flower class distribution")
        plt.show()

    return train_images_path, train_images_label, val_images_path, val_images_label


def plot_data_loader_image(data_loader):
    batch_size = data_loader.batch_size
    plot_num = min(batch_size, 4)

    json_path = "./class_indices.json"
    assert os.path.exists(json_path), json_path + " does not exist."
    json_file = open(json_path, "r")
    class_indices = json.load(json_file)

    for data in data_loader:
        images, labels = data
        for i in range(plot_num):
            # [C, H, W] → [H, W, C]
            img = images[i].numpy().transpose(1, 2, 0)
            # 反 Normalize 操作
            img = (img * [0.229, 0.224, 0.225] + [0.485, 0.456, 0.406]) * 255
            label = labels[i].item()
            plt.subplot(1, plot_num, i + 1)
            plt.xlabel(class_indices[str(label)])
            plt.xticks([])  # 去除 x 軸刻度
            plt.yticks([])  # 去除 y 軸刻度
            plt.imshow(img.astype("uint8"))
        plt.show()


def write_pickle(list_info: list, file_name: str):
    with open(file_name, "wb") as f:
        pickle.dump(list_info, f)


def read_pickle(file_name: str) → list:
    with open(file_name, "rb") as f:
        info_list = pickle.load(f)
        return info_list


def train_one_epoch(model, optimizer, data_loader, device, epoch):
    model.train()
    loss_function = torch.nn.CrossEntropyLoss()
    accu_num = torch.zeros(1).to(device)  # 累計預測正確的樣本數
    accu_loss = torch.zeros(1).to(device)  # 累計損失
    optimizer.zero_grad()

    sample_num = 0
    data_loader = tqdm(data_loader, file=sys.stdout)
    for step, data in enumerate(data_loader):
        images, labels = data
        sample_num += images.shape[0]

        pred = model(images.to(device))
        pred_classes = torch.max(pred, dim=1)[1]
        accu_num += torch.eq(pred_classes, labels.to(device)).sum()
```

```python
            loss = loss_function(pred, labels.to(device))
            loss.backward()
            accu_loss += loss.detach()

            data_loader.desc = "[train epoch {}] loss: {:.3f}, acc: {:.3f}".format(
                epoch, accu_loss.item() / (step + 1), accu_num.item() / sample_num
            )

            if not torch.isfinite(loss):
                print("WARNING: non-finite loss, ending training ", loss)
                sys.exit(1)

            optimizer.step()
            optimizer.zero_grad()

    return accu_loss.item() / (step + 1), accu_num.item() / sample_num


@torch.no_grad()
def evaluate(model, data_loader, device, epoch):
    loss_function = torch.nn.CrossEntropyLoss()

    model.eval()

    accu_num = torch.zeros(1).to(device)   # 累計預測正確的樣本數
    accu_loss = torch.zeros(1).to(device)   # 累計損失

    sample_num = 0
    data_loader = tqdm(data_loader, file=sys.stdout)
    for step, data in enumerate(data_loader):
        images, labels = data
        sample_num += images.shape[0]

        pred = model(images.to(device))
        pred_classes = torch.max(pred, dim=1)[1]
        accu_num += torch.eq(pred_classes, labels.to(device)).sum()

        loss = loss_function(pred, labels.to(device))
        accu_loss += loss

        data_loader.desc = "[valid epoch {}] loss: {:.3f}, acc: {:.3f}".format(
            epoch, accu_loss.item() / (step + 1), accu_num.item() / sample_num
        )

    return accu_loss.item() / (step + 1), accu_num.item() / sample_num
```

- train.py

```python
import os
import math
import argparse
import torch
import torch.optim as optim
import torch.optim.lr_scheduler as lr_scheduler
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
from my_dataset import MyDataSet
from vit_model import vit_base_patch16_224_in21k as create_model
from utils import read_split_data, train_one_epoch, evaluate


def main(args):
    device = torch.device(args.device if torch.cuda.is_available() else "cpu")

    if os.path.exists("./weights") is False:
        os.makedirs("./weights")

    tb_writer = SummaryWriter()

    train_images_path, train_images_label, val_images_path, val_images_label = (
```

```python
        read_split_data(args.data_path)
    )

    data_transform = {
        "train": transforms.Compose(
            [
                transforms.RandomResizedCrop(224),
                transforms.RandomHorizontalFlip(),
                transforms.ToTensor(),
                transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
            ]
        ),
        "val": transforms.Compose(
            [
                transforms.Resize(256),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
            ]
        ),
    }

    # 訓練資料集實例
    train_dataset = MyDataSet(
        images_path=train_images_path,
        images_class=train_images_label,
        transform=data_transform["train"],
    )

    # 驗證資料集實例
    val_dataset = MyDataSet(
        images_path=val_images_path,
        images_class=val_images_label,
        transform=data_transform["val"],
    )

    batch_size = args.batch_size

    nw = min(
        [os.cpu_count(), batch_size if batch_size > 1 else 0, 8]
    )  # DataLoader使用的進程數

    print("Using {} dataloader workers every process".format(nw))

    train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        pin_memory=True,
        num_workers=nw,
        collate_fn=train_dataset.collate_fn,
    )

    val_loader = torch.utils.data.DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        pin_memory=True,
        num_workers=nw,
        collate_fn=val_dataset.collate_fn,
    )

    model = create_model(num_classes=args.num_classes, has_logits=False).to(device)

    if args.weights ≠ "":
        assert os.path.exists(args.weights), "weights file: '{}' not exist.".format(
            args.weights
        )
        weights_dict = torch.load(args.weights, map_location=device)
        # 刪除不需要的權重
        del_keys = (
```

```python
            ["head.weight", "head.bias"]
            if model.has_logits
            else [
                "pre_logits.fc.weight",
                "pre_logits.fc.bias",
                "head.weight",
                "head.bias",
            ]
        )
        for k in del_keys:
            del weights_dict[k]
        print(model.load_state_dict(weights_dict, strict=False))

    if args.freeze_layers:
        for name, para in model.named_parameters():
            # 除了 head 和 pre_logits 之外，其他權重全部凍結
            if "head" not in name and "pre_logits" not in name:
                para.requires_grad_(False)
            else:
                print("training {}".format(name))

    pg = [p for p in model.parameters() if p.requires_grad]
    optimizer = optim.SGD(pg, lr=args.lr, momentum=0.9, weight_decay=5e-5)

    # Scheduler 優化來自論文
    # Bag of Tricks for Image Classification with Convolutional Neural Networks (2018)
    lf = (
        lambda x: ((1 + math.cos(x * math.pi / args.epochs)) / 2) * (1 - args.lrf)
        + args.lrf
    )  # 餘弦
    scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)

    for epoch in range(args.epochs):
        # 訓練
        train_loss, train_acc = train_one_epoch(
            model=model,
            optimizer=optimizer,
            data_loader=train_loader,
            device=device,
            epoch=epoch,
        )

        scheduler.step()

        # 驗證
        val_loss, val_acc = evaluate(
            model=model, data_loader=val_loader, device=device, epoch=epoch
        )

        tags = ["train_loss", "train_acc", "val_loss", "val_acc", "learning_rate"]
        tb_writer.add_scalar(tags[0], train_loss, epoch)
        tb_writer.add_scalar(tags[1], train_acc, epoch)
        tb_writer.add_scalar(tags[2], val_loss, epoch)
        tb_writer.add_scalar(tags[3], val_acc, epoch)
        tb_writer.add_scalar(tags[4], optimizer.param_groups[0]["lr"], epoch)

        torch.save(model.state_dict(), "./weights/model-{}.pth".format(epoch))


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--num_classes", type=int, default=5)
    parser.add_argument("--epochs", type=int, default=10)
    parser.add_argument("--batch-size", type=int, default=8)
    parser.add_argument("--lr", type=float, default=0.001)
    parser.add_argument("--lrf", type=float, default=0.01)

    # 資料集根目錄
    parser.add_argument("--data-path", type=str, default="data/flower_photos")
    parser.add_argument("--model-name", default="", help="create model name")
```

```python
    # 預訓練權重路徑，如果不想載入就設置為空字符串
    parser.add_argument(
        "--weights",
        type=str,
        default="",
        help="initial weights path",
    )

    # 權重是否凍結
    parser.add_argument("--freeze-layers", type=bool, default=True)
    parser.add_argument(
        "--device", default="cuda:0", help="device id (i.e. 0 or 0,1 or cpu)"
    )

    opt = parser.parse_args()

    main(opt)
```

- predict.py

```python
import os
import json
import torch
import matplotlib.pyplot as plt
from PIL import Image
from torchvision import transforms
from vit_model import vit_base_patch16_224_in21k as create_model


def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    data_transform = transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
        ]
    )

    # 加載圖片
    img_path = "../tulip.jpg"
    assert os.path.exists(img_path), "文件: '{}' 不存在.".format(img_path)
    img = Image.open(img_path)
    plt.imshow(img)

    # [N, C, H, W]
    img = data_transform(img)

    # 擴展批次維度
    img = torch.unsqueeze(img, dim=0)

    # 讀取 class_indict
    json_path = "./class_indices.json"
    assert os.path.exists(json_path), "文件: '{}' 不存在.".format(json_path)

    with open(json_path, "r") as f:
        class_indict = json.load(f)

    # 創建模型
    model = create_model(num_classes=5, has_logits=False).to(device)

    # 加載模型權重
    model_weight_path = "./weights/model-9.pth"
    model.load_state_dict(torch.load(model_weight_path, map_location=device))
    model.eval()

    with torch.no_grad():
        # 預測類別
```

```python
        output = torch.squeeze(model(img.to(device))).cpu()
        predict = torch.softmax(output, dim=0)
        predict_cla = torch.argmax(predict).numpy()

    print_res = "類別: {}   機率: {:.3}".format(
        class_indict[str(predict_cla)], predict[predict_cla].numpy()
    )

    plt.title(print_res)

    for i in range(len(predict)):
        print(
            "類別: {:10}   機率: {:.3}".format(class_indict[str(i)], predict[i].numpy())
        )

    plt.show()


if __name__ == "__main__":
    main()
```

⚠️ **Attention**

沒有預訓練權重載入的情況下，基於論文所優化的 `Learning Rate Scheduler` 會導致學習率過小，需要嘗試把一開始給定的 `lr` 以及 `lrf` 調高進行驗證看看結果

ViT KAN 替換

相關說明

ℹ️ **Info**

代碼使用流程介紹

1. 下載好資料集（Dataset），目前代碼預設（Default）用的是花的分類（tf_flowers）：
   https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
2. 在 `train.py` 中把 `--data-path` 設定為解壓縮後的 `flower_photos` 文件夾的絕對路徑。
3. 下載預訓練權重（Pre-Trained Weight），在 `vit_model.py` 中，每個模型都有提供預訓練權重的下載地址，根據選擇的模型下載對應的預訓練權重。
4. 在 `train.py` 腳本中將 `--weights` 參數設定為下載好的預訓練權重路徑。
5. 設定好資料集的路徑 `--data-path` 以及預訓練權重的路徑 `--weights`，即可使用 `train.py` 開始訓練（訓練過程中會自動生成 `class_indices.json` 文件）。
6. 在 `predict.py` 中導入與訓練代碼中相同的模型，並將 `model_weight_path` 設定為訓練好的模型權重路徑（預設保存在 `weights` 文件夾下）。
7. 在 `predict.py` 中將 `img_path` 設定為你需要預測的圖片的絕對路徑。
8. 設定好權重路徑 `model_weight_path` 和預測圖片路徑 `img_path`，即可使用 `predict.py` 進行預測。
9. 如果要使用其他資料集，就比照預設採用的「花的分類」之文件結構進行擺放（即一個類別對應一個文件夾），並且將訓練及預測代碼中的 `num_classes` 設定為你自己的數據的類別數。

> 💬 **Quote**
>
> 原先設計的 Kolmogorov-Arnold Network 雖然有進行部分優化，但在訓練速度上還是令人捉急，為了比較方便地進行比對，改為使用「柴比雪夫多項式」（Chebyshev Polynomials） 取代原先的「B 樣條曲線」（B-Splines）進行函數擬合

- 應用於代碼中的 ChebyKANLayer Class

```python
class ChebyKANLayer(nn.Module):
    def __init__(self, input_dim, output_dim, degree):
        super(ChebyKANLayer, self).__init__()
        self.inputdim = input_dim
        self.outdim = output_dim
        self.degree = degree

        self.cheby_coeffs = nn.Parameter(torch.empty(input_dim, output_dim, degree + 1))
        nn.init.normal_(self.cheby_coeffs, mean=0.0, std=1 / (input_dim * (degree + 1)))
        self.register_buffer("arange", torch.arange(0, degree + 1, 1))

    def forward(self, x):
        # x 形狀: (batch_size, seq_len, inputdim)
        batch_size, seq_len, _ = x.shape

        # 將 x 展平以便與切比雪夫多項式一起使用: (batch_size * seq_len, inputdim)
        x = x.view(-1, self.inputdim)

        # 使用 tanh 將輸入歸一化到 [-1, 1]
        x = torch.tanh(x)

        # 重新塑形並重複輸入 degree + 1 次
        x = x.view((-1, self.inputdim, 1)).expand(-1, -1, self.degree + 1)

        # 應用 acos 並乘以 arange [0 .. degree]
        x = x.acos() * self.arange

        # 應用 cos 以獲取切比雪夫多項式值
        x = x.cos()

        # 計算切比雪夫插值
        y = torch.einsum("bid,iod->bo", x, self.cheby_coeffs)

        # 重新塑形回原始序列格式
        y = y.view(batch_size, seq_len, self.outdim)

        return y
```

- 修改後的 `Block` 設計

```python
class Block(nn.Module):
    def __init__(
        self,
        dim,
        num_heads,
        mlp_ratio=4.0,
        qkv_bias=False,
        qk_scale=None,
        drop_ratio=0.0,
        attn_drop_ratio=0.0,
        drop_path_ratio=0.0,
        act_layer=nn.GELU,
        norm_layer=nn.LayerNorm,
        degree=3,  # 添加 degree 作為 ChebyKAN 的參數
    ):
        super(Block, self).__init__()
        self.norm1 = norm_layer(dim)
        self.attn = Attention(
            dim,
            num_heads=num_heads,
            qkv_bias=qkv_bias,
            qk_scale=qk_scale,
            attn_drop_ratio=attn_drop_ratio,
            proj_drop_ratio=drop_ratio,
        )
        self.drop_path = (
            DropPath(drop_path_ratio) if drop_path_ratio > 0.0 else nn.Identity()
        )
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)

        # 使用兩層 ChebyKANLayer 模擬 MLP 結構
        self.chebykan1 = ChebyKANLayer(
            input_dim=dim, output_dim=mlp_hidden_dim, degree=degree
        )
        self.chebykan2 = ChebyKANLayer(
            input_dim=mlp_hidden_dim, output_dim=dim, degree=degree
        )

    def forward(self, x):
        x = x + self.drop_path(self.attn(self.norm1(x)))
        x = x + self.drop_path(self.chebykan2(self.chebykan1(self.norm2(x))))
        return x
```
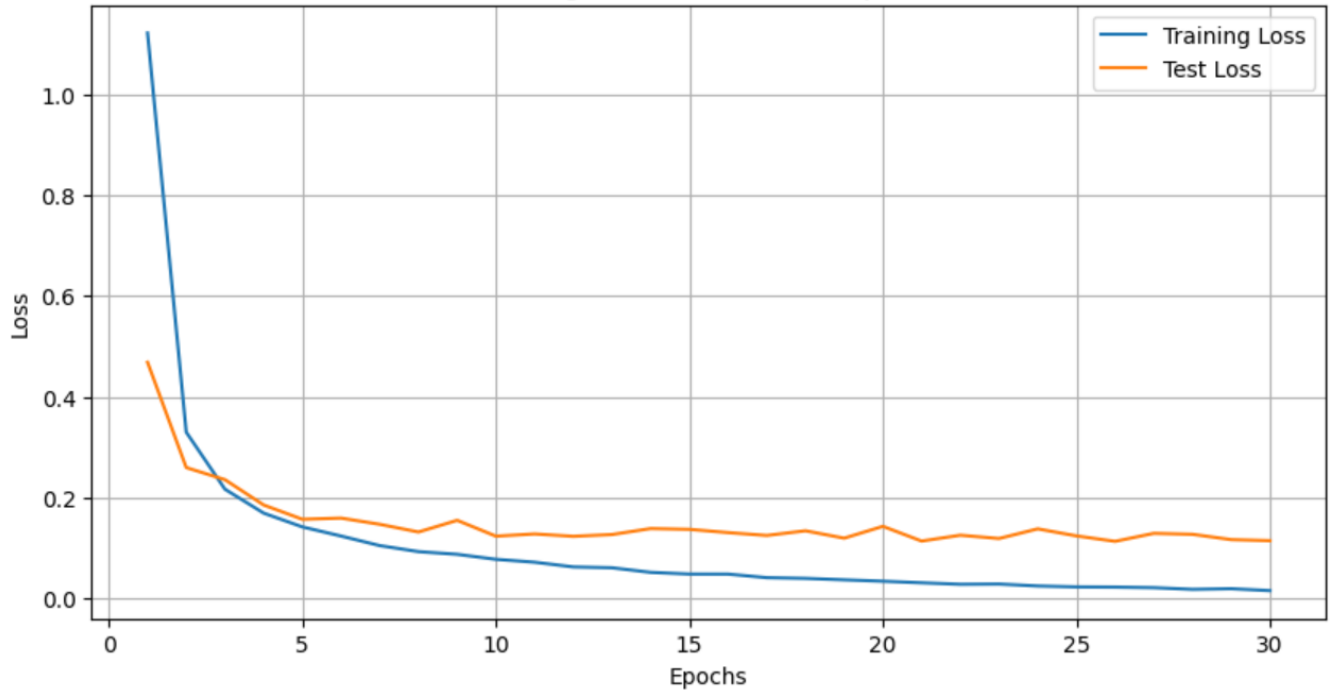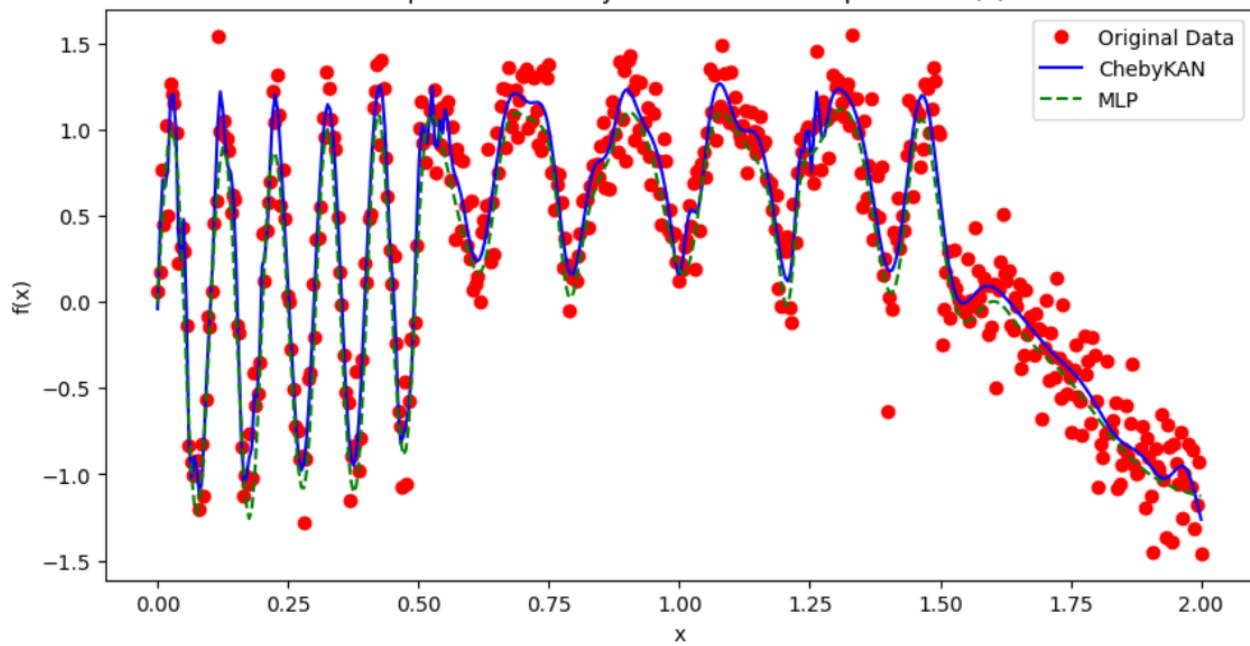
> ✏️ Note
>
> 下列為可行性驗證的各式檢驗圖表

- ChebyKAN Training & Test Loss Over Epochs

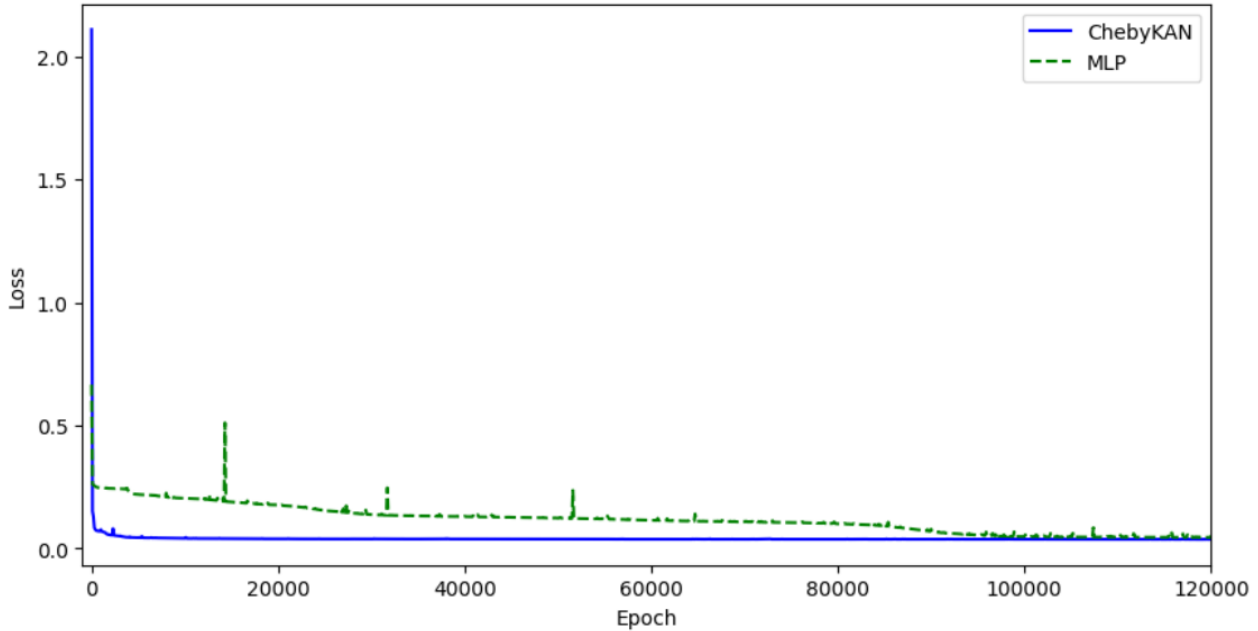Training and Test Loss Over Epochs

- ChebyKAN Comparison with MLP Interpolations

Comparison of ChebyKAN and MLP Interpolations f(x)

- Covergence Speed Comparison Between ChebyKAN & MLP

Convergence Speed Comparison Between ChebyKAN and MLP

代碼實現

> 💧 **Important**
>
> 經過調整後的 `vit_model.py`（其餘沒有變動）

```python
import torch
import torch.nn as nn
from functools import partial
from collections import OrderedDict


class ChebyKANLayer(nn.Module):
    def __init__(self, input_dim, output_dim, degree):
        super(ChebyKANLayer, self).__init__()
        self.inputdim = input_dim
        self.outdim = output_dim
        self.degree = degree

        self.cheby_coeffs = nn.Parameter(torch.empty(input_dim, output_dim, degree + 1))
        nn.init.normal_(self.cheby_coeffs, mean=0.0, std=1 / (input_dim * (degree + 1)))
        self.register_buffer("arange", torch.arange(0, degree + 1, 1))

    def forward(self, x):
        # x 形狀: (batch_size, seq_len, inputdim)
        batch_size, seq_len, _ = x.shape

        # 將 x 展平以便與切比雪夫多項式一起使用: (batch_size * seq_len, inputdim)
        x = x.view(-1, self.inputdim)

        # 使用 tanh 將輸入歸一化到 [-1, 1]
        x = torch.tanh(x)

        # 重新塑形並重複輸入 degree + 1 次
        x = x.view((-1, self.inputdim, 1)).expand(-1, -1, self.degree + 1)

        # 應用 acos 並乘以 arange [0 .. degree]
        x = x.acos() * self.arange

        # 應用 cos 以獲取切比雪夫多項式值
        x = x.cos()

        # 計算切比雪夫插值
        y = torch.einsum("bid,iod→bo", x, self.cheby_coeffs)
```

```python
        # 重新塑形回原始序列格式
        y = y.view(batch_size, seq_len, self.outdim)

        return y


def drop_path(x, drop_prob: float = 0.0, training: bool = False):
    if drop_prob == 0.0 or not training:
        return x

    keep_prob = 1 - drop_prob
    shape = (x.shape[0],) + (1,) * (
        x.ndim - 1
    )  # 適用於不同維度的張量，而不僅僅是 2D 卷積網絡
    random_tensor = keep_prob + torch.rand(shape, dtype=x.dtype, device=x.device)
    random_tensor.floor_()  # 二值化
    output = x.div(keep_prob) * random_tensor

    return output


# 每個樣本的 drop path（隨機深度，應用於殘差塊的主路徑時）
class DropPath(nn.Module):
    def __init__(self, drop_prob=None):
        super(DropPath, self).__init__()
        self.drop_prob = drop_prob

    def forward(self, x):
        return drop_path(x, self.drop_prob, self.training)


# 2D 影像 Patch Embedding
class PatchEmbed(nn.Module):
    def __init__(
        self, img_size=224, patch_size=16, in_c=3, embed_dim=768, norm_layer=None
    ):
        super().__init__()
        img_size = (img_size, img_size)
        patch_size = (patch_size, patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.grid_size = (img_size[0] // patch_size[0], img_size[1] // patch_size[1])
        self.num_patches = self.grid_size[0] * self.grid_size[1]

        self.proj = nn.Conv2d(
            in_c, embed_dim, kernel_size=patch_size, stride=patch_size
        )
        self.norm = norm_layer(embed_dim) if norm_layer else nn.Identity()

    def forward(self, x):
        B, C, H, W = x.shape
        assert (
            H == self.img_size[0] and W == self.img_size[1]
        ), f"輸入圖像大小 ({H}*{W}) 與模型大小 ({self.img_size[0]}*{self.img_size[1]}) 不匹配。"

        # flatten: [B, C, H, W] → [B, C, HW]
        # transpose: [B, C, HW] → [B, HW, C]
        x = self.proj(x).flatten(2).transpose(1, 2)
        x = self.norm(x)
        return x


class Attention(nn.Module):
    def __init__(
        self,
        dim,  # 輸入 token 的維度
        num_heads=8,
        qkv_bias=False,
        qk_scale=None,
        attn_drop_ratio=0.0,
```

```python
        proj_drop_ratio=0.0,
    ):
        super(Attention, self).__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim**-0.5
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop_ratio)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop_ratio)

    def forward(self, x):
        # [batch_size, num_patches + 1, total_embed_dim]
        B, N, C = x.shape

        # qkv(): → [batch_size, num_patches + 1, 3 * total_embed_dim]
        # reshape: → [batch_size, num_patches + 1, 3, num_heads, embed_dim_per_head]
        # permute: → [3, batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        qkv = (
            self.qkv(x)
            .reshape(B, N, 3, self.num_heads, C // self.num_heads)
            .permute(2, 0, 3, 1, 4)
        )
        # [batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        q, k, v = (
            qkv[0],
            qkv[1],
            qkv[2],
        )  # make torchscript happy (cannot use tensor as tuple)

        # transpose: → [batch_size, num_heads, embed_dim_per_head, num_patches + 1]
        # @: multiply → [batch_size, num_heads, num_patches + 1, num_patches + 1]
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        # @: multiply → [batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        # transpose: → [batch_size, num_patches + 1, num_heads, embed_dim_per_head]
        # reshape: → [batch_size, num_patches + 1, total_embed_dim]
        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x


# Vision Transformer 使用的 MLP
class Mlp(nn.Module):
    def __init__(
        self,
        in_features,
        hidden_features=None,
        out_features=None,
        act_layer=nn.GELU,
        drop=0.0,
    ):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x
```

```python
class Block(nn.Module):
    def __init__(
        self,
        dim,
        num_heads,
        mlp_ratio=4.0,
        qkv_bias=False,
        qk_scale=None,
        drop_ratio=0.0,
        attn_drop_ratio=0.0,
        drop_path_ratio=0.0,
        act_layer=nn.GELU,
        norm_layer=nn.LayerNorm,
        degree=3,  # 添加 degree 作為 ChebyKAN 的參數
    ):
        super(Block, self).__init__()
        self.norm1 = norm_layer(dim)
        self.attn = Attention(
            dim,
            num_heads=num_heads,
            qkv_bias=qkv_bias,
            qk_scale=qk_scale,
            attn_drop_ratio=attn_drop_ratio,
            proj_drop_ratio=drop_ratio,
        )
        self.drop_path = (
            DropPath(drop_path_ratio) if drop_path_ratio > 0.0 else nn.Identity()
        )
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)

        # 使用兩層 ChebyKANLayer 模擬 MLP 結構
        self.chebykan1 = ChebyKANLayer(
            input_dim=dim, output_dim=mlp_hidden_dim, degree=degree
        )
        self.chebykan2 = ChebyKANLayer(
            input_dim=mlp_hidden_dim, output_dim=dim, degree=degree
        )

    def forward(self, x):
        x = x + self.drop_path(self.attn(self.norm1(x)))
        x = x + self.drop_path(self.chebykan2(self.chebykan1(self.norm2(x))))
        return x


class VisionTransformer(nn.Module):
    def __init__(
        self,
        img_size=224,
        patch_size=16,
        in_c=3,
        num_classes=1000,
        embed_dim=768,
        depth=12,
        num_heads=12,
        mlp_ratio=4.0,
        qkv_bias=True,
        qk_scale=None,
        representation_size=None,
        distilled=False,
        drop_ratio=0.0,
        attn_drop_ratio=0.0,
        drop_path_ratio=0.0,
        embed_layer=PatchEmbed,
        norm_layer=None,
        act_layer=None,
    ):
        """
        Args:
            img_size (int, tuple): input image size
```

```
            patch_size (int, tuple): patch size
            in_c (int): number of input channels
            num_classes (int): number of classes for classification head
            embed_dim (int): embedding dimension
            depth (int): depth of transformer
            num_heads (int): number of attention heads
            mlp_ratio (int): ratio of mlp hidden dim to embedding dim
            qkv_bias (bool): enable bias for qkv if True
            qk_scale (float): override default qk scale of head_dim ** -0.5 if set
            representation_size (Optional[int]): enable and set representation layer (pre-logits) to this value
if set
            distilled (bool): model includes a distillation token and head as in DeiT models
            drop_ratio (float): dropout rate
            attn_drop_ratio (float): attention dropout rate
            drop_path_ratio (float): stochastic depth rate
            embed_layer (nn.Module): patch embedding layer
            norm_layer: (nn.Module): normalization layer
        """
        super(VisionTransformer, self).__init__()
        self.num_classes = num_classes
        self.num_features = self.embed_dim = (
            embed_dim  # 為了與其他模型一致，使用 num_features
        )
        self.num_tokens = 2 if distilled else 1
        norm_layer = norm_layer or partial(nn.LayerNorm, eps=1e-6)
        act_layer = act_layer or nn.GELU

        self.patch_embed = embed_layer(
            img_size=img_size, patch_size=patch_size, in_c=in_c, embed_dim=embed_dim
        )
        num_patches = self.patch_embed.num_patches

        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.dist_token = (
            nn.Parameter(torch.zeros(1, 1, embed_dim)) if distilled else None
        )
        self.pos_embed = nn.Parameter(
            torch.zeros(1, num_patches + self.num_tokens, embed_dim)
        )
        self.pos_drop = nn.Dropout(p=drop_ratio)

        dpr = [
            x.item() for x in torch.linspace(0, drop_path_ratio, depth)
        ]  # 隨機深度衰減規則
        self.blocks = nn.Sequential(
            *[
                Block(
                    dim=embed_dim,
                    num_heads=num_heads,
                    mlp_ratio=mlp_ratio,
                    qkv_bias=qkv_bias,
                    qk_scale=qk_scale,
                    drop_ratio=drop_ratio,
                    attn_drop_ratio=attn_drop_ratio,
                    drop_path_ratio=dpr[i],
                    norm_layer=norm_layer,
                    act_layer=act_layer,
                )
                for i in range(depth)
            ]
        )
        self.norm = norm_layer(embed_dim)

        # 表示層
        if representation_size and not distilled:
            self.has_logits = True
            self.num_features = representation_size
            self.pre_logits = nn.Sequential(
                OrderedDict(
                    [
                        ("fc", nn.Linear(embed_dim, representation_size)),
```

```python
                            ("act", nn.Tanh())),
                        ]
                    )
                )
            else:
                self.has_logits = False
                self.pre_logits = nn.Identity()

            # 分類頭
            self.head = (
                nn.Linear(self.num_features, num_classes)
                if num_classes > 0
                else nn.Identity()
            )
            self.head_dist = None
            if distilled:
                self.head_dist = (
                    nn.Linear(self.embed_dim, self.num_classes)
                    if num_classes > 0
                    else nn.Identity()
                )

            # 權重初始化
            nn.init.trunc_normal_(self.pos_embed, std=0.02)
            if self.dist_token is not None:
                nn.init.trunc_normal_(self.dist_token, std=0.02)

            nn.init.trunc_normal_(self.cls_token, std=0.02)
            self.apply(_init_vit_weights)

    def forward_features(self, x):
        # [B, C, H, W] → [B, num_patches, embed_dim]
        x = self.patch_embed(x)  # [B, 196, 768]
        # [1, 1, 768] → [B, 1, 768]
        cls_token = self.cls_token.expand(x.shape[0], -1, -1)
        if self.dist_token is None:
            x = torch.cat((cls_token, x), dim=1)  # [B, 197, 768]
        else:
            x = torch.cat(
                (cls_token, self.dist_token.expand(x.shape[0], -1, -1), x), dim=1
            )

        x = self.pos_drop(x + self.pos_embed)
        x = self.blocks(x)
        x = self.norm(x)
        if self.dist_token is None:
            return self.pre_logits(x[:, 0])
        else:
            return x[:, 0], x[:, 1]

    def forward(self, x):
        x = self.forward_features(x)
        if self.head_dist is not None:
            x, x_dist = self.head(x[0]), self.head_dist(x[1])
            if self.training and not torch.jit.is_scripting():
                # 推理期間，返回兩個分類器預測的平均值
                return x, x_dist
            else:
                return (x + x_dist) / 2
        else:
            x = self.head(x)
        return x


# ViT 權重初始化
def _init_vit_weights(m):
    # parameter m = module
    if isinstance(m, nn.Linear):
        nn.init.trunc_normal_(m.weight, std=0.01)
        if m.bias is not None:
            nn.init.zeros_(m.bias)
```

```python
        elif isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode="fan_out")
            if m.bias is not None:
                nn.init.zeros_(m.bias)
        elif isinstance(m, nn.LayerNorm):
            nn.init.zeros_(m.bias)
            nn.init.ones_(m.weight)


def vit_base_patch16_224(num_classes: int = 1000):
    # ViT-Base model (ViT-B/16) ImageNet-1k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=None,
        num_classes=num_classes,
    )

    return model


def vit_base_patch16_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Base model (ViT-B/16) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=768 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_base_patch32_224(num_classes: int = 1000):
    # ViT-Base model (ViT-B/32) ImageNet-1k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=32,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=None,
        num_classes=num_classes,
    )

    return model


def vit_base_patch32_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Base model (ViT-B/32) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=32,
        embed_dim=768,
        depth=12,
        num_heads=12,
        representation_size=768 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_large_patch16_224(num_classes: int = 1000):
    # ViT-Large model (ViT-L/16) ImageNet-1k weights @ 224x224
```

```python
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        representation_size=None,
        num_classes=num_classes,
    )

    return model


def vit_large_patch16_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Large model (ViT-L/16) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=16,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        representation_size=1024 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_large_patch32_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Large model (ViT-L/32) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=32,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        representation_size=1024 if has_logits else None,
        num_classes=num_classes,
    )

    return model


def vit_huge_patch14_224_in21k(num_classes: int = 21843, has_logits: bool = True):
    # ViT-Huge model (ViT-H/14) ImageNet-21k weights @ 224x224
    model = VisionTransformer(
        img_size=224,
        patch_size=14,
        embed_dim=1280,
        depth=32,
        num_heads=16,
        representation_size=1280 if has_logits else None,
        num_classes=num_classes,
    )

    return model
```

✅ **Done**

10 個 Epoch 的訓練結果（當前基準），無預訓練權重

⚠️ **Attention**

Learning Rate 過小的問題在應用 Kolmogorov-Arnold Network 後更為明顯，因為沒有足夠的學習量能讓函數進行擬合，如純 PyTorch 實現一樣需要重新調整

實驗結果

未經優化

- 原始 Vision Transformer

- KAN 重構後 Vision Transformer

  - 比對的表格

訓練優化後

- 原始 Vision Transformer
- KAN 重構後 Vision Transformer

- 比對的表格