KAN Kolmogorov Arnold Network Note

研究任務

1. 閱讀 KAN Kolmogorov-Arnold Network 論文
2. 設計 KAN 網路系統階層式架構 IDEF0
3. 設計 KAN 網路系統每個功能模組離散事件建模 Grafcet
4. 以 MIAT 方法論合成每個 Grafcet 控制器電路
5. 以 ChatGPT 合成每個 Grafcet Datapath 電路
6. FPGA 整合驗證

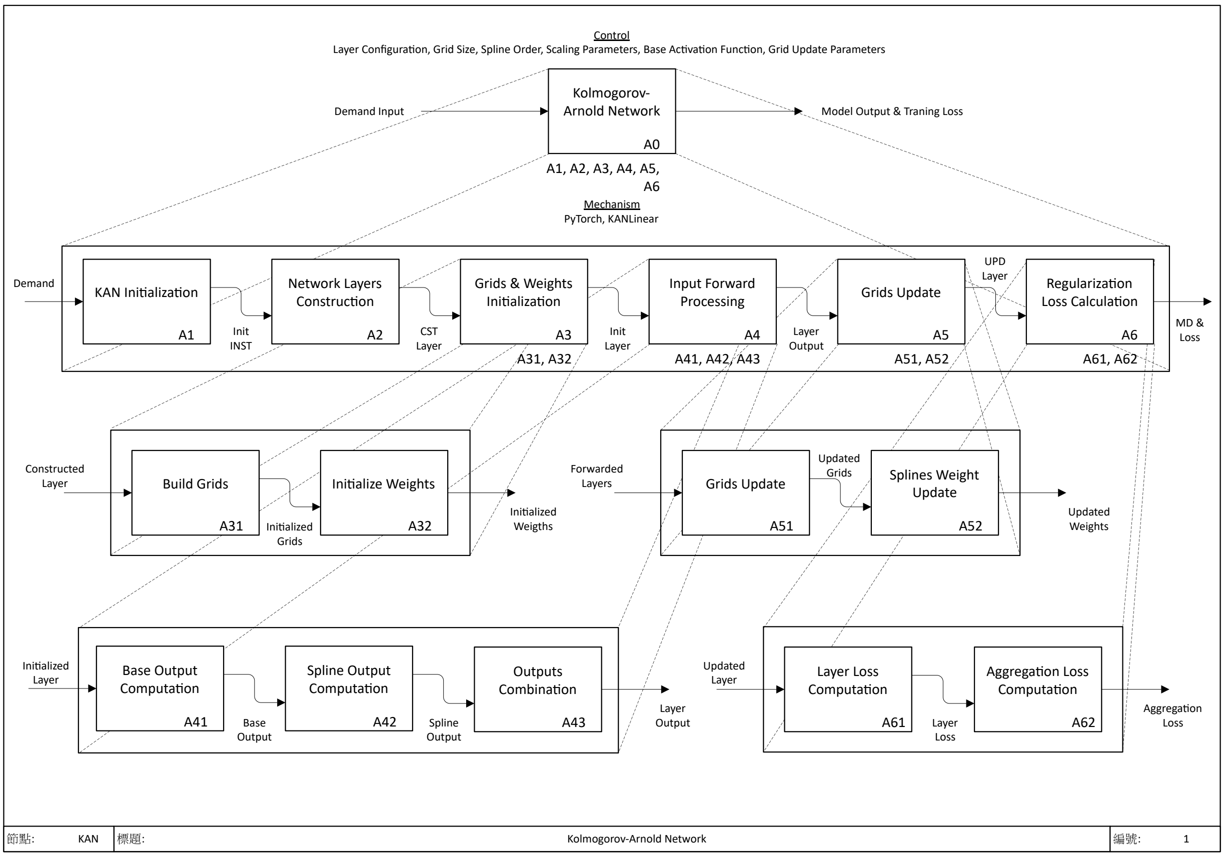
> ⚠ Attention
>
> Kolmogorov-Arnold Network 之研究任務須於六月底完成，作為暑期實習前置條件。

> ✎ Seealso
>
> 目標預定之所有（第一到第六點）研究任務，當前初版皆已完成，接下來主要目標為模型量化和優化相關電路設計及輸出結果。
>
> - Integer Quantization 部分已經完成

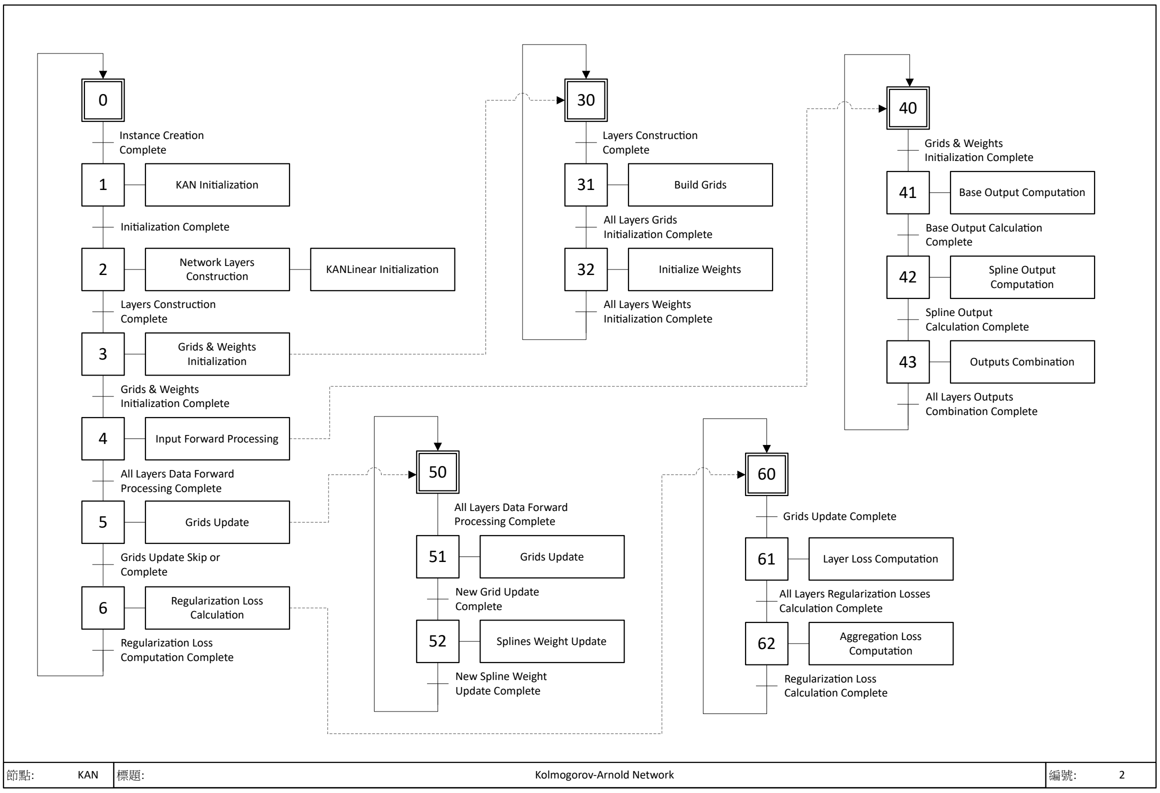
開放原始碼參考

- Code: https://github.com/KindXiaoming/pykan
- Reference: https://arxiv.org/abs/2404.19756

設計階層式架構 IDEF0



設計功能模組離散事件建模 Grafcet



Python 模擬驗證

- 基於 IDEF0 和 Grafcet 重構後之 Kolmogorov-Arnold Network (PyTorch)

```
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(
```

```python
            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)
        )
        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(
```

```python
            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)

        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

        # 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
        )
```

- 創建 Kolmogorov-Arnold Network 進行測試，資料集使用 MNIST 手寫數字辨識

```python
from EfficientKAN import KAN

# Train on MNIST
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from tqdm import tqdm

# Load MNIST
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)
trainset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
valset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
valloader = DataLoader(valset, batch_size=64, shuffle=False)

# Define model
model = KAN([28 * 28, 64, 10])
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
# Define optimizer
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
# Define learning rate scheduler
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.8)

# Define loss
criterion = nn.CrossEntropyLoss()
for epoch in range(10):
    # Train
    model.train()
    with tqdm(trainloader) as pbar:
        for i, (images, labels) in enumerate(pbar):
            images = images.view(-1, 28 * 28).to(device)
            optimizer.zero_grad()
            output = model(images)
            loss = criterion(output, labels.to(device))
            loss.backward()
            optimizer.step()
            accuracy = (output.argmax(dim=1) == labels.to(device)).float().mean()
            pbar.set_postfix(loss=loss.item(), accuracy=accuracy.item(), lr=optimizer.param_groups[0]['lr'])

    # Validation
    model.eval()
    val_loss = 0
    val_accuracy = 0
    with torch.no_grad():
        for images, labels in valloader:
            images = images.view(-1, 28 * 28).to(device)
            output = model(images)
            val_loss += criterion(output, labels.to(device)).item()
            val_accuracy += (
                (output.argmax(dim=1) == labels.to(device)).float().mean().item()
            )
    val_loss /= len(valloader)
    val_accuracy /= len(valloader)

    # Update learning rate
    scheduler.step()

    print(
        f"Epoch {epoch + 1}, Val Loss: {val_loss}, Val Accuracy: {val_accuracy}"
    )
```

```python
# Print model weights
print("Trained Model Weights:")

for i, layer in enumerate(model.layers):
    print(f"Layer {i + 1}:")
    print("Spline Weights:")
    print(layer.spline_weight)
    print("Base Weights:")
    print(layer.base_weight)
    print()

# Save model weights (need to create KAN instance then "torch.load")
torch.save(model.state_dict(), "kan_mnist_weights.pth")
# Save the entire model (just get with "torch.load")
torch.save(model, "kan_mnist_model.pth")
```

- Model 於 MNIST 之訓練結果

```
100%|██████████| 938/938 [00:40<00:00, 23.07it/s, accuracy=1, loss=0.031, lr=0.000134]
Epoch 10, Val Loss: 0.08577567627701677, Val Accuracy: 0.9750199044585988
```

- 創建 Kolmogorov-Arnold Network 進行測試，函數擬合簡易乘法 a x b

```python
import torch
import torch.nn as nn
from tqdm import tqdm

from EfficientKAN import KAN


def test_mul():
    kan = KAN([2, 3, 3, 1], base_activation=nn.Identity)
    optimizer = torch.optim.LBFGS(kan.parameters(), lr=0.001)

    with tqdm(range(200)) as pbar:
        for i in pbar:
            loss, reg_loss = None, None

            def closure():
                optimizer.zero_grad()
                x = torch.rand(1024, 2)
                y = kan(x, update_grid=(i % 20 == 0))
                assert y.shape == (1024, 1)
                nonlocal loss, reg_loss
                u = x[:, 0]
                v = x[:, 1]
                loss = nn.functional.mse_loss(y.squeeze(-1), u * v)
                reg_loss = kan.regularization_loss(1, 0)
                (loss + 1e-5 * reg_loss).backward()
                return loss + reg_loss

            optimizer.step(closure)
            pbar.set_postfix(mse_loss=loss.item(), reg_loss=reg_loss.item())

    for layer in kan.layers:
        print(layer.spline_weight)

    torch.save(kan, 'model/kan_multiple_model.pth')
    torch.save(kan.state_dict(), "model/kan_multiple_weights.pth")

    # Test the trained model
    test_model(kan)


def test_model(model):
    model.eval()
    with torch.no_grad():
        test_x = torch.rand(1024, 2)
```

```python
        test_y = model(test_x)
        u = test_x[:, 0]
        v = test_x[:, 1]
        expected_y = u * v
        test_loss = nn.functional.mse_loss(test_y.squeeze(-1), expected_y)
        print(f"Test Loss: {test_loss.item():.4f}")


test_mul()
```

- Model 於函數擬合簡易乘法之訓練結果

```
100%|██████████| 200/200 [01:03<00:00,  3.13it/s, mse_loss=0.0938, reg_loss=9.12e-7]
```

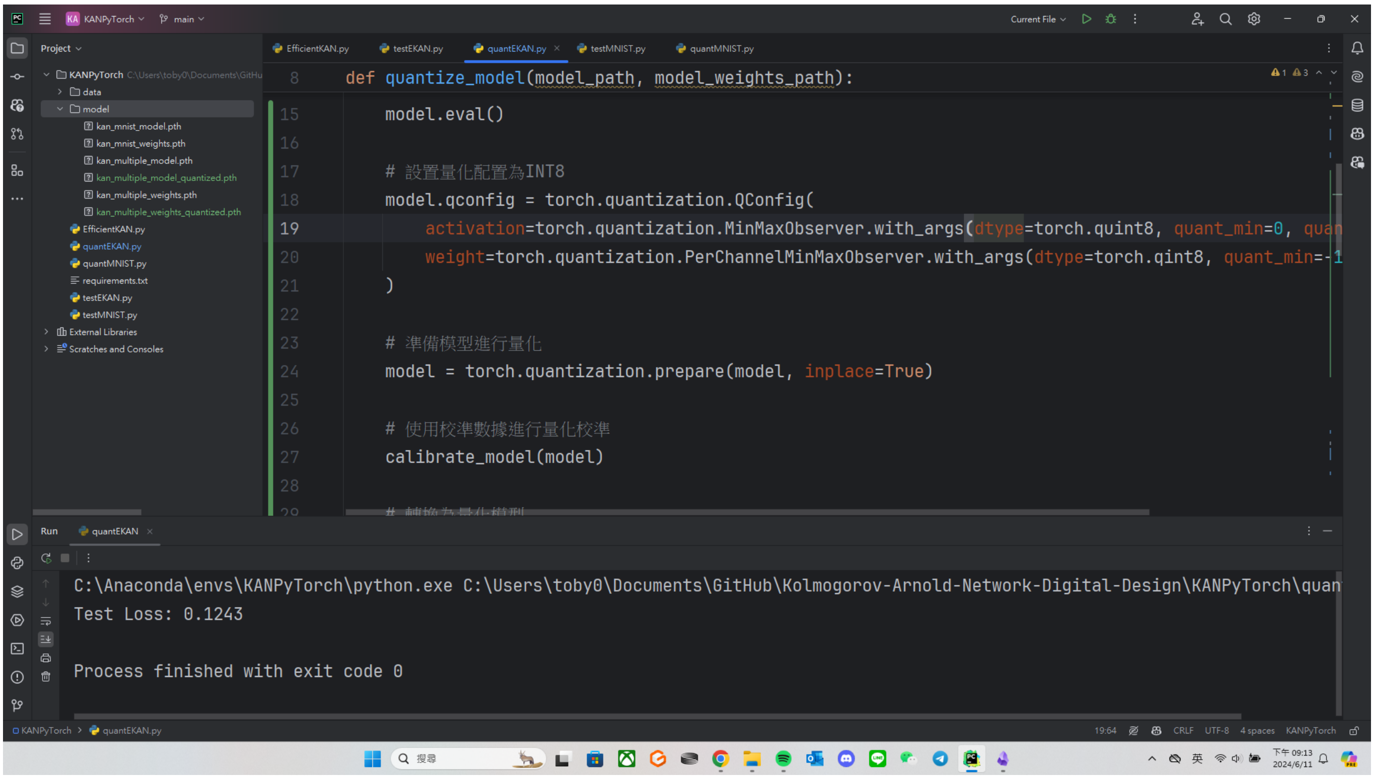
訓練模型 Quantization 處理

> Info
>
> 最基本的 Quantization 實現，使用 PyTorch 庫進行 INT8 Quantization 操作並保存模型。

- PyTorch INT8 Quantization 代碼實現

```python
import torch
import torch.nn as nn
import torch.quantization

from EfficientKAN import KAN


def quantize_model(model_path, model_weights_path):
    # 加載模型架構
    model = torch.load(model_path)

    # 加載模型權重
    model.load_state_dict(torch.load(model_weights_path))

    model.eval()

    # 設置量化配置為INT8
    model.qconfig = torch.quantization.QConfig(
        activation=torch.quantization.MinMaxObserver.with_args(dtype=torch.quint8, quant_min=0, quant_max=255),
        weight=torch.quantization.PerChannelMinMaxObserver.with_args(dtype=torch.qint8, quant_min=-128,
quant_max=127)
    )

    # 準備模型進行量化
    model = torch.quantization.prepare(model, inplace=True)

    # 使用校準數據進行量化校準
    calibrate_model(model)

    # 轉換為量化模型
    model = torch.quantization.convert(model, inplace=True)

    # 保存量化後的模型
    torch.save(model, 'model/kan_multiple_model_quantized.pth')
    torch.save(model.state_dict(), "model/kan_multiple_weights_quantized.pth")

    # 測試量化後的模型
    test_model(model)


def calibrate_model(model):
    # 使用隨機數據進行校準
    with torch.no_grad():
        for _ in range(100):
```

```
            test_x = torch.rand(1024, 2)
            model(test_x)


def test_model(model):
    model.eval()
    with torch.no_grad():
        test_x = torch.rand(1024, 2)
        test_y = model(test_x)
        u = test_x[:, 0]
        v = test_x[:, 1]
        expected_y = u * v
        test_loss = nn.functional.mse_loss(test_y.squeeze(-1), expected_y)
        print(f"Test Loss: {test_loss.item():.4f}")


# 使用訓練好的模型路徑和權重文件路徑
model_path = 'model/kan_multiple_model.pth'
model_weights_path = 'model/kan_multiple_weights.pth'

quantize_model(model_path, model_weights_path)
```



ℹ️ Info

進一步將 Quantization 後的 Weights 導出成 CSV，提供給後續推論調用進行使用。

⚠️ Warning

發現導出的 CSV 結果依舊是浮點數，需要再次確認為何是 CSV 是吃到原始的 Weights 而非 Quantization 後的 Weights。

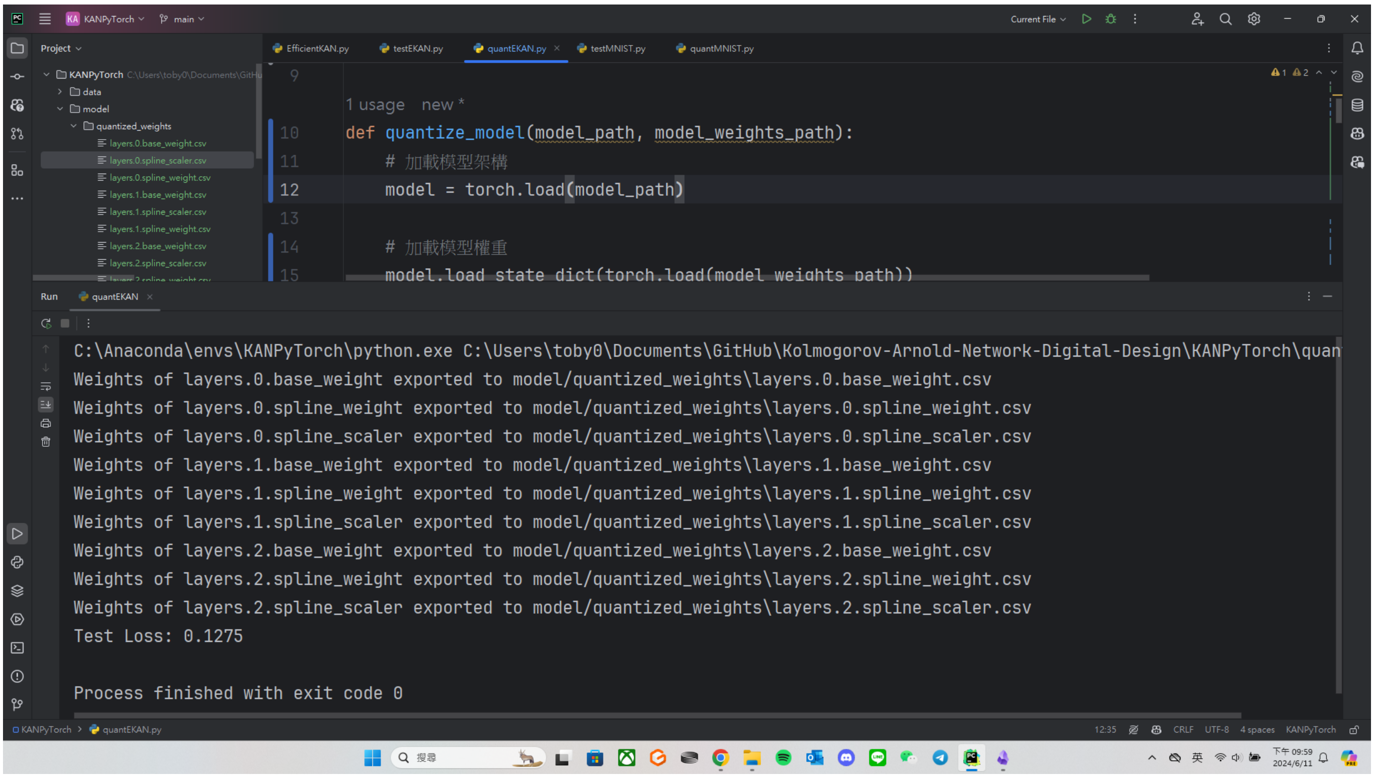
✅ Check

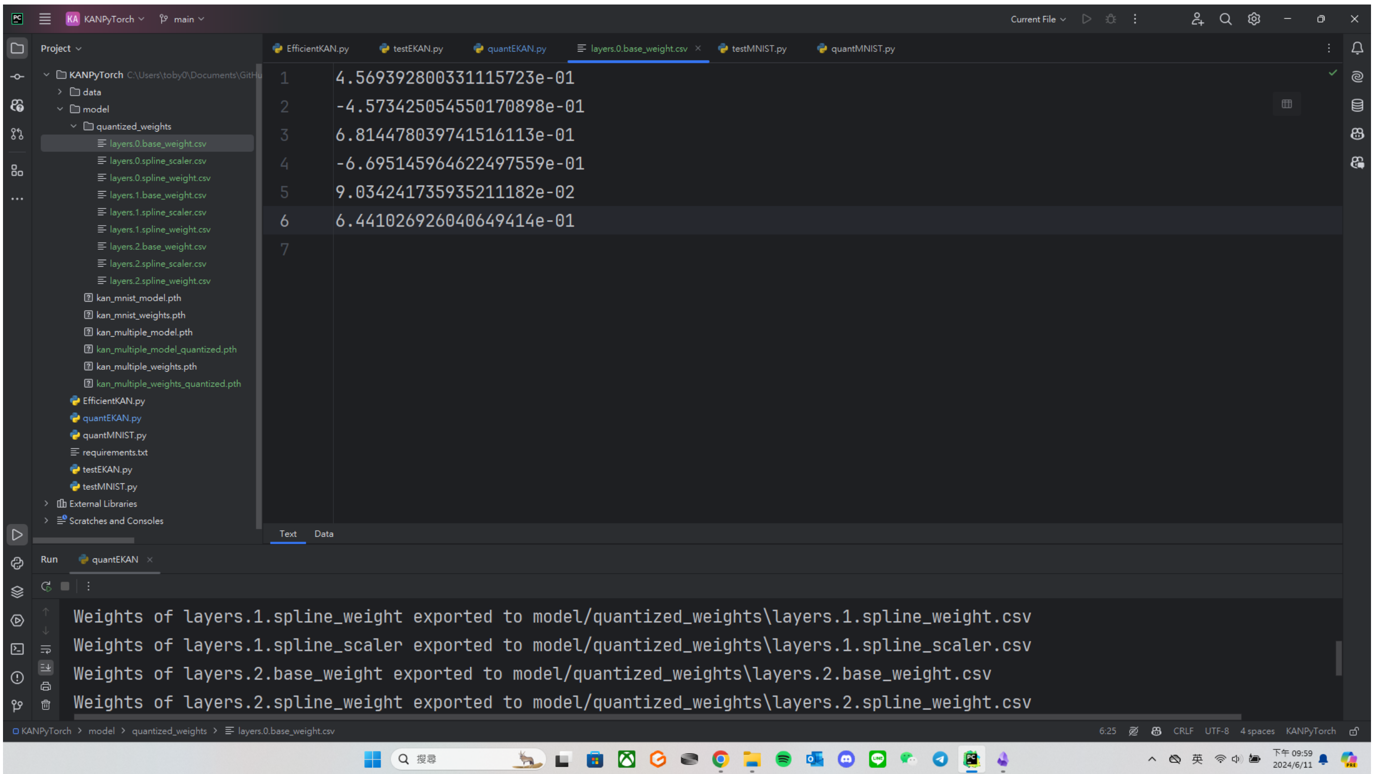
在 PyTorch 中進行量化時，模型的權重並不會直接在 `state_dict` 中顯示為 INT8。這是因為 PyTorch 量化模型的權重會被存儲為浮點數，但在推理時會被視為 INT8。具體來說，量化過程會引入 `FakeQuantize` 模塊來模擬 INT8 行為，但權重仍然以浮點數形式存在。

- PyTorch INT8 Quantization 含 CSV 權重導出

```python
import os

import numpy as np
import torch
import torch.nn as nn
import torch.quantization


def quantize_model(model_path, model_weights_path):
    # 加載模型架構
    model = torch.load(model_path)

    # 加載模型權重
    model.load_state_dict(torch.load(model_weights_path))

    model.eval()

    # 設置量化配置為INT8
    model.qconfig = torch.quantization.default_qconfig

    # 準備模型進行量化
    torch.quantization.prepare(model, inplace=False)

    # 使用校準數據進行量化校準
    calibrate_model(model)

    # 轉換為量化模型
    torch.quantization.convert(model, inplace=False)

    # 保存量化後的模型
    torch.save(model.state_dict(), "model/kan_multiple_weights_quantized.pth")
    torch.save(model, 'model/kan_multiple_model_quantized.pth')

    # 導出量化後的權重
    export_weights_to_csv(model, "model/quantized_weights")

    # print(model.state_dict())

    # 測試量化後的模型
    test_model(model)


def calibrate_model(model):
    # 使用隨機數據進行校準
    with torch.no_grad():
        for _ in range(100):
            test_x = torch.rand(1024, 2)
            model(test_x)


def test_model(model):
    model.eval()
    with torch.no_grad():
        test_x = torch.rand(1024, 2)
        test_y = model(test_x)
        u = test_x[:, 0]
        v = test_x[:, 1]
        expected_y = u * v
        test_loss = nn.functional.mse_loss(test_y.squeeze(-1), expected_y)
        print(f"Test Loss: {test_loss.item():.4f}")


def export_weights_to_csv(model, folder_path):
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

    # 遍歷模型中的每一層，並將權重存儲在單獨的 CSV 文件中
    for name, param in model.named_parameters():
        if param.requires_grad:
            weight_array = param.detach().cpu().numpy()
```

```python
            file_path = os.path.join(folder_path, f"{name}.csv")
            np.savetxt(file_path, weight_array.flatten(), delimiter=",")
            print(f"Weights of {name} exported to {file_path}")


# 使用訓練好的模型路徑和權重文件路徑
model_path = 'model/kan_multiple_model.pth'
model_weights_path = 'model/kan_multiple_weights.pth'

quantize_model(model_path, model_weights_path)
```





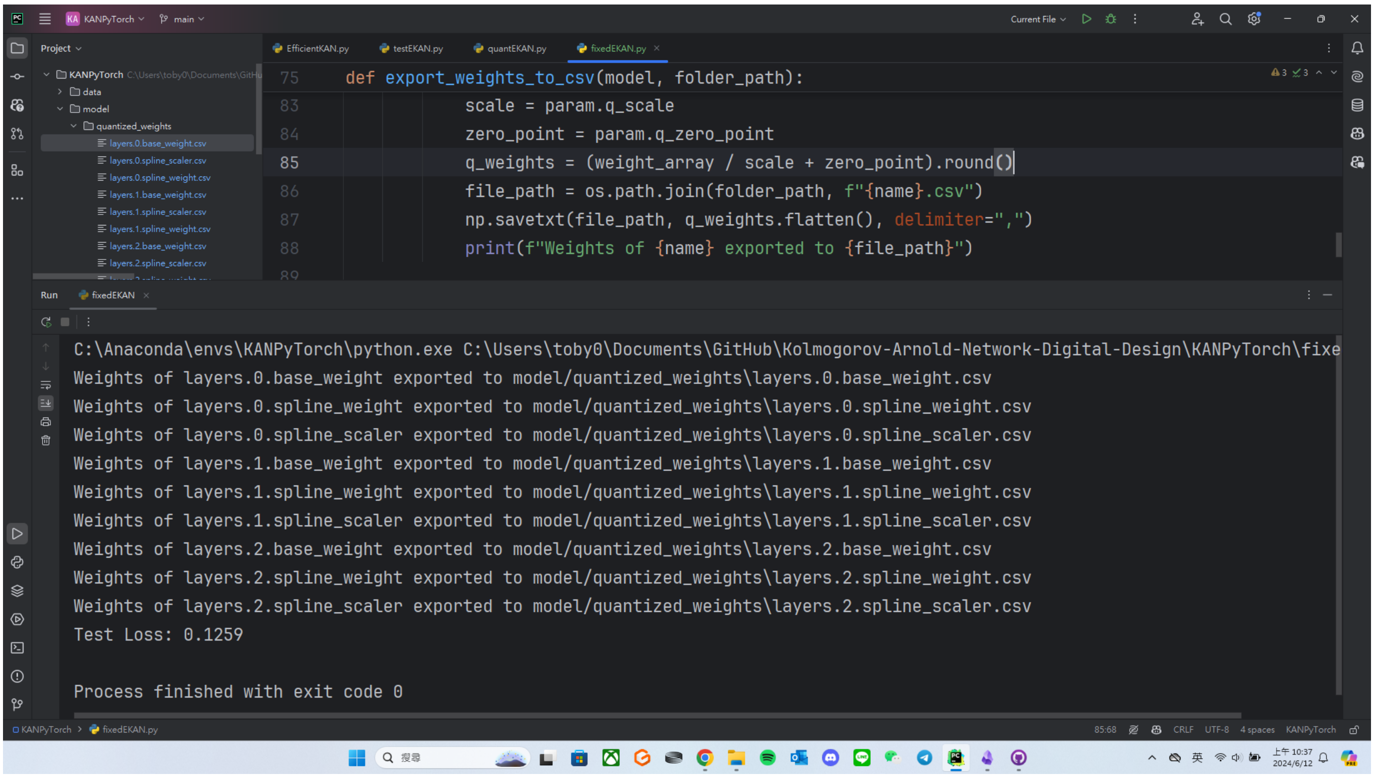
ℹ Info

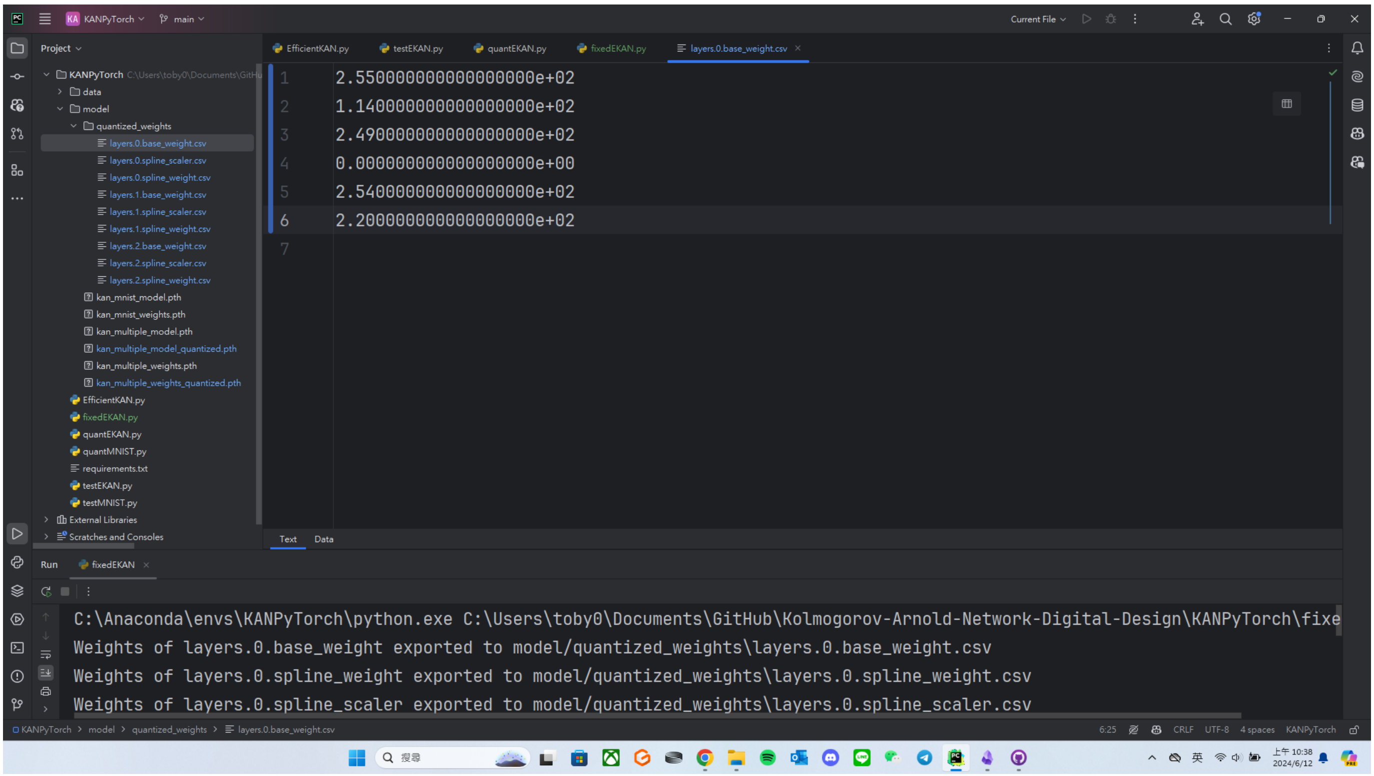
由於 PyTorch Quantization 的性質，改為使用手動實現 Quantization 工具。

- 手動實現 Quantization

```
import os

import numpy as np
import torch
import torch.nn as nn
import torch.quantization


def quantize_tensor(tensor, num_bits=8):
    qmin = 0.
    qmax = 2. ** num_bits - 1.

    min_val, max_val = tensor.min(), tensor.max()
    scale = (max_val - min_val) / (qmax - qmin)
    zero_point = qmin - min_val / scale

    zero_point = int(zero_point)
    q_tensor = (tensor / scale + zero_point).round().clamp(qmin, qmax)

    return q_tensor, scale, zero_point


def dequantize_tensor(q_tensor, scale, zero_point):
    return scale * (q_tensor - zero_point)


def quantize_model(model_path, model_weights_path):
    # 加載模型架構
    model = torch.load(model_path)

    # 加載模型權重
    model.load_state_dict(torch.load(model_weights_path))

    model.eval()

    # 手動量化模型中的權重
    for name, param in model.named_parameters():
        if param.requires_grad:
            q_param, scale, zero_point = quantize_tensor(param.data)
            param.data = dequantize_tensor(q_param, scale, zero_point)
            param.q_scale = scale
            param.q_zero_point = zero_point

    # 保存量化後的模型
    torch.save(model.state_dict(), "model/kan_multiple_weights_quantized.pth")
    torch.save(model, 'model/kan_multiple_model_quantized.pth')

    # 導出量化後的權重
    export_weights_to_csv(model, "model/quantized_weights")

    # 測試量化後的模型
    test_model(model)


def calibrate_model(model):
    # 使用隨機數據進行校準
    with torch.no_grad():
        for _ in range(100):
            test_x = torch.rand(1024, 2)
            model(test_x)


def test_model(model):
    model.eval()
    with torch.no_grad():
        test_x = torch.rand(1024, 2)
        test_y = model(test_x)
        u = test_x[:, 0]
```

```python
        v = test_x[:, 1]
        expected_y = u * v
        test_loss = nn.functional.mse_loss(test_y.squeeze(-1), expected_y)
        print(f"Test Loss: {test_loss.item():.4f}")


def export_weights_to_csv(model, folder_path):
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

    # 遍歷模型中的每一層，並將權重存儲在單獨的 CSV 文件中
    for name, param in model.named_parameters():
        if param.requires_grad:
            weight_array = param.detach().cpu().numpy()
            scale = param.q_scale
            zero_point = param.q_zero_point
            q_weights = (weight_array / scale + zero_point).round()
            file_path = os.path.join(folder_path, f"{name}.csv")
            np.savetxt(file_path, q_weights.flatten(), delimiter=",")
            print(f"Weights of {name} exported to {file_path}")


# 使用訓練好的模型路徑和權重文件路徑
model_path = 'model/kan_multiple_model.pth'
model_weights_path = 'model/kan_multiple_weights.pth'

quantize_model(model_path, model_weights_path)
```





Info

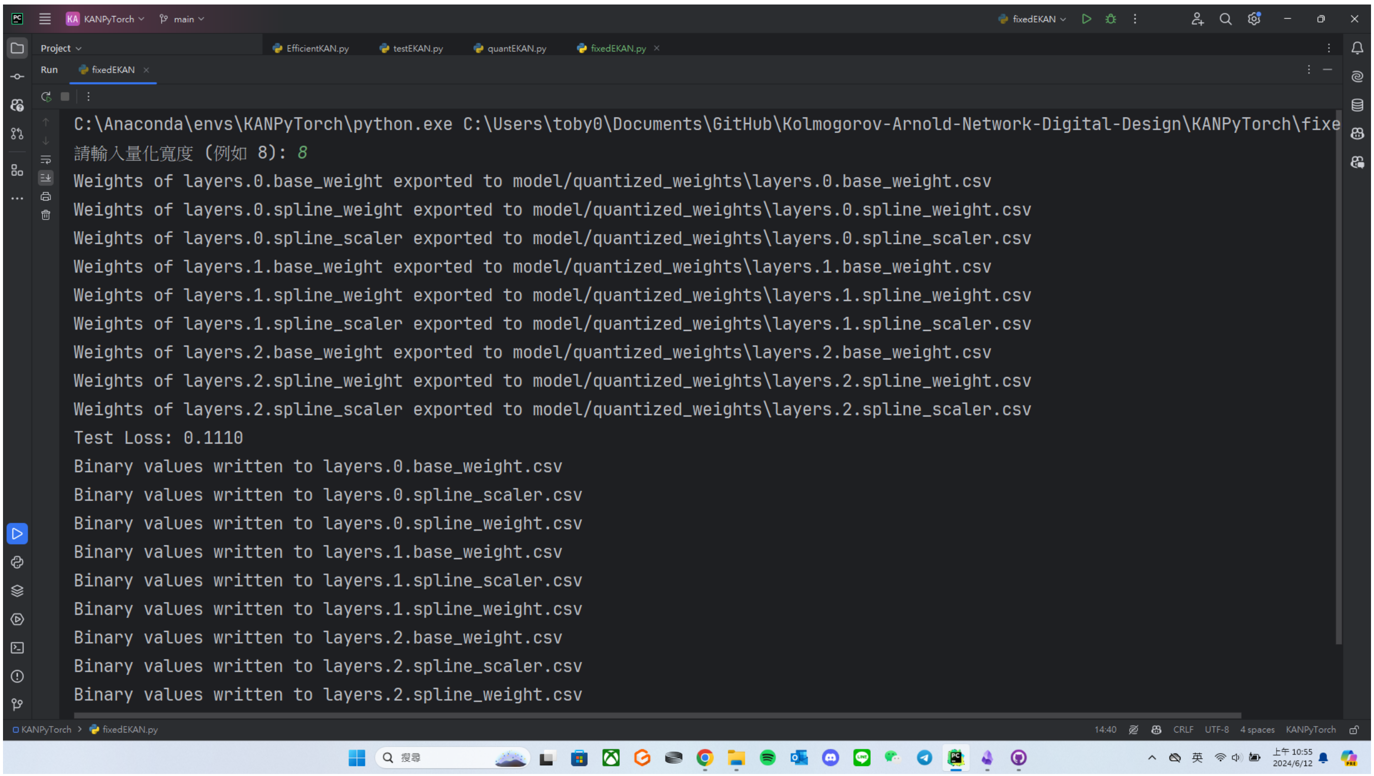
將 Quantization 後的 Weights 進一步處理為二進制輸出（同時自定義 Quantization 寬度）

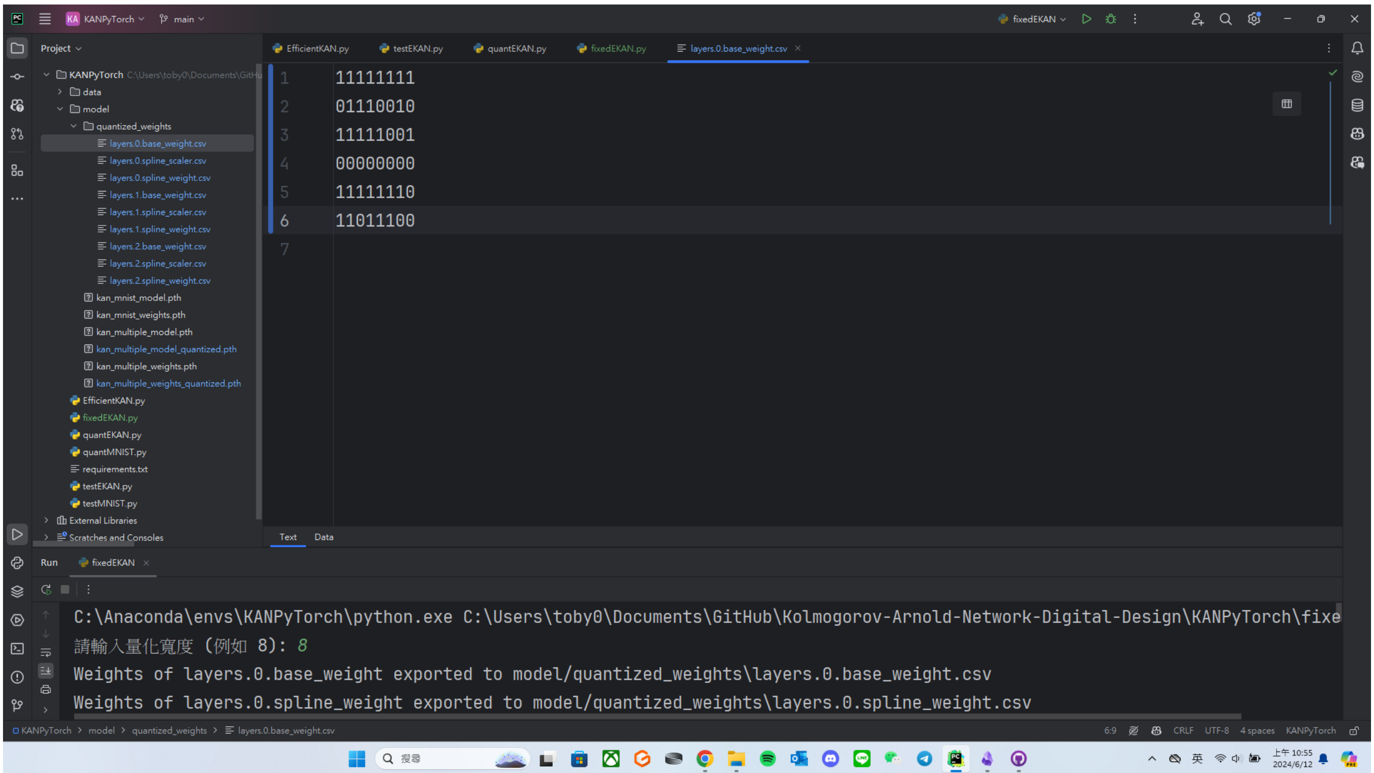
- 二進制輸出及自定義量化寬度

```
import os
import numpy as np
import torch
import torch.nn as nn
import torch.quantization


def quantize_tensor(tensor, num_bits):
    qmin = 0.
    qmax = 2. ** num_bits - 1.

    min_val, max_val = tensor.min(), tensor.max()
    scale = (max_val - min_val) / (qmax - qmin)
    zero_point = qmin - min_val / scale

    zero_point = int(zero_point)
    q_tensor = (tensor / scale + zero_point).round().clamp(qmin, qmax)

    return q_tensor, scale, zero_point


def dequantize_tensor(q_tensor, scale, zero_point):
    return scale * (q_tensor - zero_point)


def quantize_model(model_path, model_weights_path, num_bits):
    # 加載模型架構
    model = torch.load(model_path)

    # 加載模型權重
    model.load_state_dict(torch.load(model_weights_path))

    model.eval()

    # 手動量化模型中的權重
    for name, param in model.named_parameters():
```

```python
        if param.requires_grad:
            q_param, scale, zero_point = quantize_tensor(param.data, num_bits)
            param.data = dequantize_tensor(q_param, scale, zero_point)
            param.q_scale = scale
            param.q_zero_point = zero_point

    # 保存量化後的模型
    torch.save(model.state_dict(), "model/kan_multiple_weights_quantized.pth")
    torch.save(model, 'model/kan_multiple_model_quantized.pth')

    # 導出量化後的權重
    export_weights_to_csv(model, "model/quantized_weights", num_bits)

    # 測試量化後的模型
    test_model(model)


def calibrate_model(model):
    # 使用隨機數據進行校準
    with torch.no_grad():
        for _ in range(100):
            test_x = torch.rand(1024, 2)
            model(test_x)


def test_model(model):
    model.eval()
    with torch.no_grad():
        test_x = torch.rand(1024, 2)
        test_y = model(test_x)
        u = test_x[:, 0]
        v = test_x[:, 1]
        expected_y = u * v
        test_loss = nn.functional.mse_loss(test_y.squeeze(-1), expected_y)
        print(f"Test Loss: {test_loss.item():.4f}")


def export_weights_to_csv(model, folder_path, num_bits):
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

    # 遍歷模型中的每一層，並將權重存儲在單獨的 CSV 文件中
    for name, param in model.named_parameters():
        if param.requires_grad:
            weight_array = param.detach().cpu().numpy()
            scale = param.q_scale
            zero_point = param.q_zero_point
            q_weights = (weight_array / scale + zero_point).round()
            file_path = os.path.join(folder_path, f"{name}.csv")
            np.savetxt(file_path, q_weights.flatten(), delimiter=",")
            print(f"Weights of {name} exported to {file_path}")


def read_convert_and_write_binary(folder_path, num_bits):
    # 遍歷文件夾中的所有CSV文件
    for file_name in os.listdir(folder_path):
        if file_name.endswith(".csv"):
            file_path = os.path.join(folder_path, file_name)
            # 讀取CSV文件
            data = np.loadtxt(file_path, delimiter=",")
            # 將數據轉換為二進制格式
            binary_data = np.vectorize(np.binary_repr)(data.astype(int), width=num_bits)
            # 將二進制數據寫回CSV文件
            with open(file_path, 'w') as f:
                for binary_value in binary_data:
                    f.write(f"{binary_value}\n")
            print(f"Binary values written to {file_name}")


if __name__ == "__main__":
    # 從終端獲取量化寬度
```

```
    num_bits = int(input("請輸入量化寬度 (例如 8): "))

    # 使用訓練好的模型路徑和權重文件路徑
    model_path = 'model/kan_multiple_model.pth'
    model_weights_path = 'model/kan_multiple_weights.pth'

    quantize_model(model_path, model_weights_path, num_bits)

    # 調用函數，讀取並轉換文件夾中的CSV文件
    folder_path = "model/quantized_weights"
    read_convert_and_write_binary(folder_path, num_bits)
```





方法論合成 Grafcet 控制器電路

Success

更換測試資料及 Network 設計以求可以將電路設計容納進去（學習函數擬合簡易乘法，KAN 的架構為 [2, 3, 3, 1]）。

- KANLayer 實現（KANLayer.v）

```
module KANLinear #(parameter IN_FEATURES = 2, OUT_FEATURES = 3, integer LAYER_NUM = 0) (
    input clk,
    input reset,
    input [15:0] data_in [IN_FEATURES-1:0],
    output reg [15:0] data_out [OUT_FEATURES-1:0]
);

    reg [15:0] base_weight [OUT_FEATURES*IN_FEATURES-1:0];
    reg [15:0] spline_weight [OUT_FEATURES*IN_FEATURES-1:0];
    reg [15:0] base_weight_2d [OUT_FEATURES-1:0][IN_FEATURES-1:0];
    reg [15:0] spline_weight_2d [OUT_FEATURES-1:0][IN_FEATURES-1:0];

    integer i, j;

    initial begin
        if (LAYER_NUM == 0) begin
            $readmemh("C:/intelFPGA_lite/18.1/KAN2/base_weight_layer_0.txt", base_weight);
            $readmemh("C:/intelFPGA_lite/18.1/KAN2/spline_weight_layer_0.txt", spline_weight);
        end else if (LAYER_NUM == 1) begin
            $readmemh("C:/intelFPGA_lite/18.1/KAN2/base_weight_layer_1.txt", base_weight);
            $readmemh("C:/intelFPGA_lite/18.1/KAN2/spline_weight_layer_1.txt", spline_weight);
        end else if (LAYER_NUM == 2) begin
            $readmemh("C:/intelFPGA_lite/18.1/KAN2/base_weight_layer_2.txt", base_weight);
            $readmemh("C:/intelFPGA_lite/18.1/KAN2/spline_weight_layer_2.txt", spline_weight);
        end

        for (i = 0; i < OUT_FEATURES; i = i + 1) begin
            for (j = 0; j < IN_FEATURES; j = j + 1) begin
                base_weight_2d[i][j] = base_weight[i * IN_FEATURES + j];
                spline_weight_2d[i][j] = spline_weight[i * IN_FEATURES + j];
            end
        end
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (i = 0; i < OUT_FEATURES; i = i + 1) begin
                data_out[i] <= 16'd0;
            end
        end else begin
            for (i = 0; i < OUT_FEATURES; i = i + 1) begin
                data_out[i] <= 16'd0;
                for (j = 0; j < IN_FEATURES; j = j + 1) begin
                    data_out[i] <= data_out[i] + base_weight_2d[i][j] * data_in[j];
                end
                if (data_out[i] < 16'd0) begin
                    data_out[i] <= 16'd0;
                end
            end
        end
    end

endmodule
```

Caution

MNIST 版本之 Reference

- KANLayer 實現（KANLayer.v）

```
module KANLayer #(
    parameter IN_FEATURES = 784,
    parameter OUT_FEATURES = 64,
    parameter SCALE = 256,  // Quantization scale factor
    parameter BASE_WEIGHT_FILE = "C:/intelFPGA_lite/18.1/KAN/base_weight_layer_0.txt",
    parameter SPLINE_WEIGHT_FILE = "C:/intelFPGA_lite/18.1/KAN/spline_weight_layer_0.txt"
)(
    input wire clk,
    input wire reset,
    input wire [7:0] in_data [0:IN_FEATURES-1],  // Input data
    output reg [7:0] out_data [0:OUT_FEATURES-1] // Output data
);
    // Weights stored in on-chip memory (BRAM)
    reg signed [15:0] base_weights [0:OUT_FEATURES*IN_FEATURES-1];
    reg signed [15:0] spline_weights [0:OUT_FEATURES*IN_FEATURES-1];

    // Load weights from memory (initialization)
    initial begin
        $readmemh(BASE_WEIGHT_FILE, base_weights);
        $readmemh(SPLINE_WEIGHT_FILE, spline_weights);
    end

    // Output registers
    reg signed [31:0] base_output [0:OUT_FEATURES-1];
    reg signed [31:0] spline_output [0:OUT_FEATURES-1];
    reg signed [31:0] total_output [0:OUT_FEATURES-1];

    integer i, j;

    // Forward pass
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            for (i = 0; i < OUT_FEATURES; i = i + 1) begin
                base_output[i] <= 0;
                spline_output[i] <= 0;
                total_output[i] <= 0;
            end
        end else begin
            for (i = 0; i < OUT_FEATURES; i = i + 1) begin
                base_output[i] <= 0;
                spline_output[i] <= 0;
                for (j = 0; j < IN_FEATURES; j = j + 1) begin
                    base_output[i] <= base_output[i] + in_data[j] * base_weights[i*IN_FEATURES + j];
                    spline_output[i] <= spline_output[i] + in_data[j] * spline_weights[i*IN_FEATURES + j];
                end
                total_output[i] <= (base_output[i] + spline_output[i]) / SCALE;  // Combine and scale the outputs
                out_data[i] <= total_output[i][15:8];  // Convert to 8-bit output
            end
        end
    end
endmodule
```

ChatGPT 合成 Grafcet Datapath 電路

Success

更換測試資料及 Network 設計以求可以將電路設計容納進去（學習函數擬合簡易乘法，KAN 的架構為 [2, 3, 3, 1]）。

- 完整 Network 實現（KAN.v）

```
module KAN #(parameter IN_FEATURES = 2, L1_FEATURES = 3, L2_FEATURES = 3, OUT_FEATURES = 1) (
    input clk,
    input reset,
    input [15:0] input_data [IN_FEATURES-1:0],
    output [15:0] output_data
);
```

```
    wire [15:0] layer1_out [L1_FEATURES-1:0];
    wire [15:0] layer2_out [L2_FEATURES-1:0];
    wire [15:0] layer3_out [OUT_FEATURES-1:0];

    KANLinear #(.IN_FEATURES(IN_FEATURES), .OUT_FEATURES(L1_FEATURES), .LAYER_NUM(0)) layer1 (
        .clk(clk),
        .reset(reset),
        .data_in(input_data),
        .data_out(layer1_out)
    );

    KANLinear #(.IN_FEATURES(L1_FEATURES), .OUT_FEATURES(L2_FEATURES), .LAYER_NUM(1)) layer2 (
        .clk(clk),
        .reset(reset),
        .data_in(layer1_out),
        .data_out(layer2_out)
    );

    KANLinear #(.IN_FEATURES(L2_FEATURES), .OUT_FEATURES(OUT_FEATURES), .LAYER_NUM(2)) layer3 (
        .clk(clk),
        .reset(reset),
        .data_in(layer2_out),
        .data_out(layer3_out)
    );

    assign output_data = layer3_out[0];

endmodule
```

> ⚠ Caution
>
> MNIST 版本之 Reference

- 完整 Network 實現（KAN.v）

```
module KAN (
    input wire clk,
    input wire reset,
    input wire [7:0] in_data [0:783],   // 28x28 = 784 pixels, 8-bit each
    output wire [7:0] out_data [0:9]    // 10 classes, 8-bit each
);
    // Internal signals for each layer
    wire [7:0] layer1_out [0:63];
    wire [7:0] layer2_out [0:9];

    // Instantiate layers
    KANLayer #(
        .IN_FEATURES(784),
        .OUT_FEATURES(64),
        .BASE_WEIGHT_FILE("C:/intelFPGA_lite/18.1/KAN/base_weight_layer_0.txt"),
        .SPLINE_WEIGHT_FILE("C:/intelFPGA_lite/18.1/KAN/spline_weight_layer_0.txt")
    ) layer1 (
        .clk(clk),
        .reset(reset),
        .in_data(in_data),
        .out_data(layer1_out)
    );

    KANLayer #(
        .IN_FEATURES(64),
        .OUT_FEATURES(10),
        .BASE_WEIGHT_FILE("C:/intelFPGA_lite/18.1/KAN/base_weight_layer_1.txt"),
        .SPLINE_WEIGHT_FILE("C:/intelFPGA_lite/18.1/KAN/spline_weight_layer_1.txt")
    ) layer2 (
        .clk(clk),
        .reset(reset),
        .in_data(layer1_out),
        .out_data(layer2_out)
```

```
    );

    // Connect the final output
    assign out_data = layer2_out;
endmodule
```

FPGA 整合驗證

Success

更換測試資料及 Network 設計以求可以將電路設計容納進去（學習函數擬合簡易乘法，KAN 的架構為 [2, 3, 3, 1]）。

- Testbench 實現（Testbench.v）

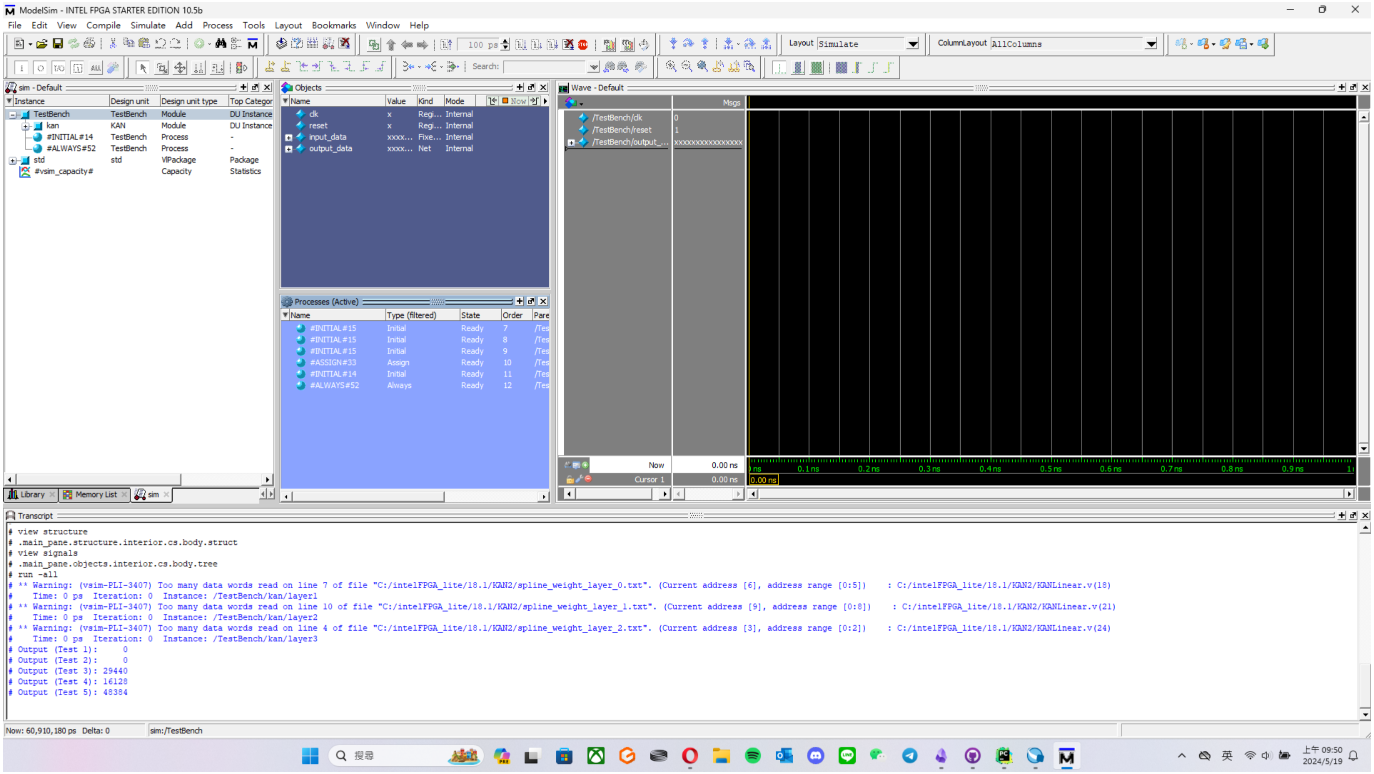
```
module TestBench;
    reg clk;
    reg reset;
    reg [15:0] input_data [0:1];
    wire [15:0] output_data;

    KAN #(.IN_FEATURES(2), .L1_FEATURES(3), .L2_FEATURES(3), .OUT_FEATURES(1)) kan (
        .clk(clk),
        .reset(reset),
        .input_data(input_data),
        .output_data(output_data)
    );

    initial begin
        clk = 0;
        reset = 1;
        input_data[0] = 16'd0;
        input_data[1] = 16'd0;
        #10 reset = 0;

        // test data 1
        input_data[0] = 16'd50;
        input_data[1] = 16'd30;
        #10;
        $display("Output (Test 1): %d", output_data);

        // test data 2
        input_data[0] = 16'd100;
        input_data[1] = 16'd200;
        #10;
        $display("Output (Test 2): %d", output_data);

        // test data 3
        input_data[0] = 16'd150;
        input_data[1] = 16'd250;
        #10;
        $display("Output (Test 3): %d", output_data);

        // test data 4
        input_data[0] = 16'd75;
        input_data[1] = 16'd125;
        #10;
        $display("Output (Test 4): %d", output_data);

        // test data 5
        input_data[0] = 16'd175;
        input_data[1] = 16'd225;
        #10;
        $display("Output (Test 5): %d", output_data);
    end

    always #5 clk = ~clk;
```

```
endmodule
```

> ⚠ **Warning**
>
> 硬體設計完仿真的結果如下，誤差老實說非常大，基本上是 Quantization 後的 Weight 發生狀況。

> ℹ **Todo**
>
> 目前主要的問題在於原生的 Verilog 不支持浮點運算，而這剛好是 KAN 最大的痛點，畢竟 KAN 與 MLP 最大的不同就在於是使用曲線去做合成，基本上出來的權重都會是小數。就算完成 Quantization，也只是 Weight 本身的值是 INT，依舊會需要處理浮點數的 Scaler 才能進行正確的權重還原，也因為可以進行 Weight 還原，使得 PyTorch 在 Quantization 後依舊可以有好結果 (畢竟 Python 是支援浮點運算的) 。目前還沒有想到甚麼比較好的解決方法，只是單純去查了一下說 HLS 可以支持單精度浮點和雙精度浮點運算，那之後可能會朝相關方向前進，暫時就不調整目前 Verilog 的結果。

- ModelSim 仿真結果



Pipeline 架構處理
