【資料準備】

- 放壓縮檔在 data\raw\Simple_chinastell_data.zip

```
Expand-Archive -Force data\raw\Simple_chinastell_data.zip data\raw
```

- 解壓縮後的資料放在 data\raw\

```
data/raw/
├── 1_hole/
├── 2_blackscratch_long/
├── 3_whitescratch/
└── ...（共10類）
```

- 產生訓練與測試集

```
python -m src.dataprep.split
```

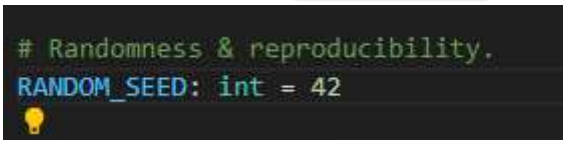
- 輸出：

```
data/splits/train.csv
data/splits/test.csv
```

- 分割程式碼

```python
"""Generate stratified 8:2 train/test CSV splits from class-folder dataset."""

from __future__ import annotations

from pathlib import Path

import pandas as pd
from sklearn.model_selection import train_test_split

from src.utils.constants import RANDOM_SEED

# Comments go above the code they describe:
# Root folders for input dataset and output CSV files.
ROOT = Path(__file__).resolve().parents[2]
DATA_DIR = ROOT / "data" / "raw"
SPLIT_DIR = ROOT / "data" / "splits"
SPLIT_DIR.mkdir(parents=True, exist_ok=True)

IMG_EXTS = {".jpg", ".jpeg", ".png", ".bmp", ".tif", ".tiff"}


def scan_dataset() -> pd.DataFrame:
    """Scan class-folder dataset and return a DataFrame with (path, label)."""
    rows: list[dict[str, str]] = []
    for class_dir in sorted(DATA_DIR.iterdir()):
        if not class_dir.is_dir():
            continue
        label = class_dir.name
        for p in class_dir.iterdir():
            if p.suffix.lower() in IMG_EXTS:
                rows.append({"path": p.as_posix(), "label": label})
    return pd.DataFrame(rows)


def main() -> None:
    """Create stratified 8:2 train/test CSV files using the global RANDOM_SEED."""
    df = scan_dataset()
    assert not df.empty, f"Dataset is empty under {DATA_DIR}"
    print("Total samples:", len(df))
    print(df["label"].value_counts())

    train_df, test_df = train_test_split(
        df, test_size=0.2, stratify=df["label"], random_state=RANDOM_SEED
    )
    train_df.to_csv(SPLIT_DIR / "train.csv", index=False)
    test_df.to_csv(SPLIT_DIR / "test.csv", index=False)
```

```
    print("Saved:", SPLIT_DIR / "train.csv", "and", SPLIT_DIR / "test.csv")


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

- 程式碼解說
  - 使用 `train_test_split` 來分割訓練與測試集
  - 訓練集與測試集的比例為 8:2
  - random_state 設定為 `RANDOM_SEED`
  - RANDOM_SEED 定義在 `constants.py`
    - 

  - 訓練集與測試集分別存放在 `data\splits\train.csv` 以及 `data\splits\test.csv`

【模型定義】

- MLP 模型定義
- 使用 `torch.nn` 定義網路結構
- 程式碼

```python
"""MLP classifier for flattened images (baseline)."""

from __future__ import annotations

import torch.nn as nn

from src.models.registry import register


class MLP(nn.Module):
    """A simple multilayer perceptron for image classification.

    Notes:
        Input expects a flattened tensor of shape (B, in_dim).
        You should handle resize/gray/flatten in your Dataset/Transforms.
    """

    def __init__(self, in_dim: int, num_classes: int, hidden: list[int], dropout: float = 0.2)
        """Initialize the MLP."""
        super().__init__()
        layers: list[nn.Module] = []
        prev = in_dim
        for h in hidden:
            layers += [nn.Linear(prev, h), nn.ReLU(inplace=True), nn.Dropout(dropout)]
            prev = h
        layers += [nn.Linear(prev, num_classes)]
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        """Forward pass returning logits of shape (B, num_classes)."""
        return self.net(x)


@register("mlp")
def build_mlp(in_dim: int, num_classes: int, hidden: list[int], dropout: float = 0.2) -> nn.Mod
    """Factory function used by the registry to create an MLP."""
    return MLP(in_dim=in_dim, num_classes=num_classes, hidden=hidden, dropout=dropout)
```

- 程式碼解說
  - *init*_() 建構MLP的Layers
    - 傳入：
      - in_dim: 輸入向量的維度
      - hidden: 隱藏層配置

        - num_classes: 最後輸出的分類數
        - dropout: 防止 overfitting
    - 用 nn.Sequential() 把它們包起來
    - forward() 前向傳遞
    - register() 註冊模型到 registry
- vgg16 模型定義
- 使用 `torchvision.models` 定義網路結構
- 程式碼

```python
"""VGG16 transfer-learning model with a configurable classifier head."""

from __future__ import annotations

import torch.nn as nn
from torchvision import models

from src.models.registry import register


def _load_vgg16(pretrained: bool) -> nn.Module:
    """Create a VGG16-BN backbone, optionally with pretrained ImageNet weights."""
    if pretrained:
        try:
            weights = models.VGG16_BN_Weights.IMAGENET1K_V1
            model = models.vgg16_bn(weights=weights)
        except AttributeError:
            model = models.vgg16_bn(pretrained=True)
    else:
        if hasattr(models, "VGG16_BN_Weights"):
            model = models.vgg16_bn(weights=None)
        else:
            model = models.vgg16_bn(pretrained=False)
    return model


def _build_classifier(
    in_features: int, num_classes: int, hidden: int, dropout: float
) -> nn.Sequential:
    """Return a compact classifier head starting from the flattened feature size."""
    return nn.Sequential(
        nn.Linear(in_features, hidden),
        nn.ReLU(inplace=True),
        nn.Dropout(p=dropout),
        nn.Linear(hidden, num_classes),
    )


@register("vgg16")
def build_vgg16(
    num_classes: int,
    pretrained: bool = True,
    freeze_features: bool = False,
    dropout: float = 0.5,
    classifier_hidden: int = 512,
) -> nn.Module:
```

```
    """Build a VGG16-BN model with a small classifier head.

    Notes:
        The original VGG16-BN classifier expects a flattened input of 25088
        elements (7x7x512). The custom head must start from this size.
    """
    model = _load_vgg16(pretrained=pretrained)

    if freeze_features:
        for p in model.features.parameters():
            p.requires_grad = False

    # The first linear layer of the original classifier takes 25088 inputs.
    in_features = model.classifier[0].in_features

    # Replace the entire classifier with a compact head that starts at 25088.
    model.classifier = _build_classifier(
        in_features=in_features,
        num_classes=num_classes,
        hidden=classifier_hidden,
        dropout=dropout,
    )
    return model
```

- 程式碼解說
  - _load_vgg16()
    - 載入 VGG16-BN 模型，可選擇 pretrained=True（載入 ImageNet 檔案）
  - freeze_features
    - 是否凍結 feature extractor 的卷積層
  - _build_classifier()
    - 建立自己的 classifier head（Linear + ReLU + Dropout + Linear）
  - build_vgg16()
    - 把 backbone + classifier head 合成完整模型
  - @register("vgg16")
    - 讓 config.yaml 可以用 "vgg16" 來建立這個模型
  - 模型定義完成

【訓練模型】

- MLP 模型訓練

```python
"""Train an MLP baseline for steel-coil classification."""

from __future__ import annotations

import importlib
from pathlib import Path
from typing import Tuple

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import yaml
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

from src.dataprep.dataset import create_dataloader_from_cfg
from src.models.registry import build
from src.training.losses import build_criterion
from src.training.optim import build_optimizer, build_scheduler
from src.utils.constants import RANDOM_SEED
from src.utils.seed import set_global_seed

# ensure @register("mlp")
importlib.import_module("src.models.mlp")

def evaluate(
    model: nn.Module, loader: torch.utils.data.DataLoader, device: torch.device
) -> Tuple[float, float, float]:
    """Evaluate accuracy, precision and recall on a given dataloader."""
    model.eval()
    all_preds, all_targets = [], []
    with torch.no_grad():
        for xb, yb in loader:
            xb = xb.to(device)
            yb = yb.to(device)
            logits = model(xb)
            preds = torch.argmax(logits, dim=1)
            all_preds.append(preds.cpu())
            all_targets.append(yb.cpu())
    y_true = torch.cat(all_targets).numpy()
    y_pred = torch.cat(all_preds).numpy()
    acc = float(accuracy_score(y_true, y_pred))
    precision, recall, _f1, _ = precision_recall_fscore_support(
        y_true, y_pred, average="macro", zero_division=0
    )
    return acc, float(precision), float(recall)
```

```python
def main() -> None:
    """Train the MLP model using parameters from configs/base.yaml and save artifacts."""
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Training on device: {device}")

    set_global_seed(RANDOM_SEED)

    cfg = yaml.safe_load(open("configs/base.yaml", "r", encoding="utf-8"))
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    train_loader, class_names = create_dataloader_from_cfg(cfg, phase="train")
    test_loader, _ = create_dataloader_from_cfg(cfg, phase="test")
    num_classes = len(class_names)

    img_size = int(
        cfg.get("data", {}).get("image_size") or cfg.get("data", {}).get("img_size") or 128
    )
    in_dim = img_size * img_size
    mlp_cfg = cfg.get("model", {}).get("mlp", {}) or {}
    hidden = list(mlp_cfg.get("hidden", [512, 256]))
    dropout = float(mlp_cfg.get("dropout", 0.2))

    model = build("mlp", in_dim=in_dim, num_classes=num_classes, hidden=hidden, dropout=dropout
        device
    )

    epochs = int(cfg.get("train", {}).get("epochs", 10))

    criterion = build_criterion(cfg)
    optimizer = build_optimizer(model, cfg)
    scheduler = build_scheduler(optimizer, cfg)

    out_root = Path(cfg.get("paths", {}).get("outputs", "outputs"))
    model_dir = out_root / "models"
    metrics_dir = out_root / "metrics"
    model_dir.mkdir(parents=True, exist_ok=True)
    metrics_dir.mkdir(parents=True, exist_ok=True)

    best_acc = -1.0
    history_train, history_test = [], []

    for epoch in range(1, epochs + 1):
        model.train()
        running_loss = 0.0
        for xb, yb in train_loader:
```

```
            xb = xb.to(device)
            yb = yb.to(device)
            optimizer.zero_grad(set_to_none=True)
            logits = model(xb)
            loss = criterion(logits, yb)
            loss.backward()
            optimizer.step()
            running_loss += float(loss.item())
        if scheduler is not None:
            scheduler.step()
        train_acc, train_prec, train_rec = evaluate(model, train_loader, device)
        test_acc, test_prec, test_rec = evaluate(model, test_loader, device)
        history_train.append(train_acc)
        history_test.append(test_acc)

        avg_loss = running_loss / max(1, len(train_loader))
        print(
            f"Epoch {epoch:02d}/{epochs} | loss {avg_loss:.4f} | "
            f"train acc {train_acc:.3f} prec {train_prec:.3f} rec {train_rec:.3f} | "
            f"test acc {test_acc:.3f} prec {test_prec:.3f} rec {test_rec:.3f}"
        )

        if test_acc > best_acc:
            best_acc = test_acc
            torch.save(
                {"model_state": model.state_dict(), "config": cfg, "classes": class_names},
                model_dir / "best_mlp.pt",
            )

    fig = plt.figure(figsize=(6, 4))
    plt.plot(range(1, len(history_train) + 1), history_train, label="Train Acc")
    plt.plot(range(1, len(history_test) + 1), history_test, label="Test Acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.tight_layout()
    fig.savefig(metrics_dir / "mlp_accuracy_curve.png", dpi=150)
    plt.close(fig)

    print("Saved:", model_dir / "best_mlp.pt")
    print("Saved:", metrics_dir / "mlp_accuracy_curve.png")


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

- 程式碼解說

```
讀 YAML config
     ↓
建立 train/test dataloader
     ↓
依照 YAML 參數建立 MLP 模型
     ↓
依照 YAML 建立 loss / optimizer / scheduler
     ↓
開始訓練迴圈
     ↓
每個 epoch 計算 train/test 的 acc/precision/recall
     ↓
若 test acc 變好 → 儲存 best_mlp.pt
     ↓
畫出 accuracy 曲線 mlp_accuracy_curve.png
```

- vgg16 模型訓練

```python
"""Train a VGG16 transfer-learning model for steel-coil classification."""

from __future__ import annotations

import importlib
from pathlib import Path
from typing import Tuple

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import yaml
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

from src.dataprep.dataset import create_dataloader_from_cfg
from src.models.registry import build
from src.training.losses import build_criterion
from src.training.optim import build_optimizer, build_scheduler

BEST_CKPT = "best_vgg16.pt"
CURVE_PNG = "vgg16_accuracy_curve.png"


def evaluate(
    model: nn.Module, loader: torch.utils.data.DataLoader, device: torch.device
) -> Tuple[float, float, float]:
    """Evaluate accuracy, macro precision and macro recall."""
    model.eval()
    preds, targets = [], []
    with torch.no_grad():
        for xb, yb in loader:
            xb = xb.to(device)
            yb = yb.to(device)
            logits = model(xb)
            pred = torch.argmax(logits, dim=1)
            preds.append(pred.cpu())
            targets.append(yb.cpu())
    y_true = torch.cat(targets).numpy()
    y_pred = torch.cat(preds).numpy()
    acc = float(accuracy_score(y_true, y_pred))
    precision, recall, _f1, _ = precision_recall_fscore_support(
        y_true, y_pred, average="macro", zero_division=0
    )
    return acc, float(precision), float(recall)
```

```
def main() -> None:
    """Train VGG16 using config values and save artifacts."""
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Training on device: {device}")

    cfg = yaml.safe_load(open("configs/base.yaml", "r", encoding="utf-8"))
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    importlib.import_module("src.models.vgg")

    train_loader, class_names = create_dataloader_from_cfg(cfg, phase="train")
    test_loader, _ = create_dataloader_from_cfg(cfg, phase="test")
    num_classes = len(class_names)

    vgg_cfg = cfg.get("model", {}).get("vgg16", {}) or {}
    model = build(
        "vgg16",
        num_classes=num_classes,
        pretrained=bool(vgg_cfg.get("pretrained", True)),
        freeze_features=bool(vgg_cfg.get("freeze_features", False)),
        dropout=float(vgg_cfg.get("dropout", 0.5)),
        classifier_hidden=int(vgg_cfg.get("classifier_hidden", 512)),
    ).to(device)

    criterion = build_criterion(cfg)
    optimizer = build_optimizer(model, cfg)
    scheduler = build_scheduler(optimizer, cfg)

    out_root = Path(cfg.get("paths", {}).get("outputs", "outputs"))
    model_dir = out_root / "models"
    metrics_dir = out_root / "metrics"
    model_dir.mkdir(parents=True, exist_ok=True)
    metrics_dir.mkdir(parents=True, exist_ok=True)

    epochs = int(cfg.get("train", {}).get("epochs", 10))
    best_acc = -1.0
    hist_tr, hist_te = [], []

    for epoch in range(1, epochs + 1):
        model.train()
        running = 0.0
        for xb, yb in train_loader:
            xb = xb.to(device)
            yb = yb.to(device)
            optimizer.zero_grad(set_to_none=True)
            logits = model(xb)
```

```
            loss = criterion(logits, yb)
            loss.backward()
            optimizer.step()
            running += float(loss.item())
        if scheduler is not None:
            scheduler.step()

        tr_acc, tr_prec, tr_rec = evaluate(model, train_loader, device)
        te_acc, te_prec, te_rec = evaluate(model, test_loader, device)
        hist_tr.append(tr_acc)
        hist_te.append(te_acc)

        avg_loss = running / max(1, len(train_loader))
        print(
            f"Epoch {epoch:02d}/{epochs} | loss {avg_loss:.4f} | "
            f"train acc {tr_acc:.3f} prec {tr_prec:.3f} rec {tr_rec:.3f} | "
            f"test acc {te_acc:.3f} prec {te_prec:.3f} rec {te_rec:.3f}"
        )

        if te_acc > best_acc:
            best_acc = te_acc
            torch.save(
                {"model_state": model.state_dict(), "config": cfg, "classes": class_names},
                model_dir / BEST_CKPT,
            )

    fig = plt.figure(figsize=(6, 4))
    plt.plot(range(1, len(hist_tr) + 1), hist_tr, label="Train Acc")
    plt.plot(range(1, len(hist_te) + 1), hist_te, label="Test Acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.tight_layout()
    fig.savefig(metrics_dir / CURVE_PNG, dpi=150)
    plt.close(fig)

    print("Saved:", model_dir / BEST_CKPT)
    print("Saved:", metrics_dir / CURVE_PNG)


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

- 程式碼解說

```
讀 YAML config（configs/base.yaml）
     ↓
import VGG16 模型並註冊（@register("vgg16")）
     ↓
建立 train / test dataloader（依據 data 設定與 augmentation）
     ↓
依照 YAML 參數建立 VGG16 transfer-learning 模型
  └→ 是否使用 pretrained ImageNet 權重（pretrained）
  └→ 是否凍結卷積特徵層（freeze_features）
  └→ 建立自訂 classifier head（classifier_hidden, dropout）
     ↓
依照 YAML 建立 loss / optimizer / scheduler
  └→ loss：cross_entropy
  └→ optimizer：adam / adamw / sgd
  └→ scheduler：step / onecycle / 或關閉
     ↓
開始訓練迴圈（共 epochs 輪）
  └→ forward：VGG16(xb) 得到 logits
  └→ 計算 loss = criterion(logits, yb)
  └→ backward：loss.backward()
  └→ optimizer.step() 更新權重
  └→ scheduler.step()（如果有設定）
     ↓
每個 epoch 計算 train / test 的 acc / precision / recall
     ↓
若 test acc 變好 → 儲存最佳模型為 outputs/models/best_vgg16.pt
     ↓
畫出 Train / Test accuracy 曲線 → 存成 outputs/metrics/vgg16_accuracy_curve.png
```

【損失函數/優化器設定】

- 損失函數
- python code

```python
"""Criterion (loss function) factory."""

from __future__ import annotations

from typing import Iterable

import torch
import torch.nn as nn


def _tensor_or_none(x: Iterable[float] | None) -> torch.Tensor | None:
    """Convert a Python list to a float tensor on CPU, or return None."""
    if x is None:
        return None
    t = torch.tensor(list(x), dtype=torch.float32)
    return t


def build_criterion(cfg: dict) -> nn.Module:
    """Build a PyTorch loss function from a config dictionary.

    Supported:
        - cross_entropy: nn.CrossEntropyLoss (supports class weights)
        - focal (stub): raises NotImplementedError for now

    Args:
        cfg: Configuration with a 'train.loss' section.

    Returns:
        An `nn.Module` loss instance.
    """
    loss_cfg = dict(cfg.get("train", {}).get("loss", {}) or {})
    name = str(loss_cfg.get("name", "cross_entropy")).lower()

    if name == "cross_entropy":
        weight_list = loss_cfg.get("weight", None)
        weight = _tensor_or_none(weight_list)
        return nn.CrossEntropyLoss(weight=weight)

    if name == "focal":
        raise NotImplementedError("Focal loss is not implemented yet.")

    raise ValueError(f"Unknown loss: {name}")
```

- 程式碼解說
  - _tensor_or_none()
    - 將一個 Python list 轉換成 float tensor on CPU，或者回傳 None
  - build_criterion()
    - 根據 config.yaml 設定損失函數
    - 目前只支援 cross_entropy 損失函數
- 損失函數定義完成
- 優化器定義

```python
"""Optimizer and scheduler factories."""

from __future__ import annotations

import torch
import torch.optim as optim
from torch.optim.lr_scheduler import _LRScheduler
from torch.optim.optimizer import Optimizer


def build_optimizer(model: torch.nn.Module, cfg: dict) -> Optimizer:
    """Build an optimizer from 'train.optimizer' config."""
    opt_cfg = dict(cfg.get("train", {}).get("optimizer", {}) or {})
    name = str(opt_cfg.get("name", "adam")).lower()
    lr = float(opt_cfg.get("lr", 1e-3))
    weight_decay = float(opt_cfg.get("weight_decay", 0.0))

    if name == "adam":
        betas = tuple(opt_cfg.get("betas", (0.9, 0.999)))
        return optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay, betas=betas)

    if name == "adamw":
        betas = tuple(opt_cfg.get("betas", (0.9, 0.999)))
        return optim.AdamW(model.parameters(), lr=lr, weight_decay=weight_decay, betas=betas)

    if name == "sgd":
        momentum = float(opt_cfg.get("momentum", 0.9))
        return optim.SGD(
            model.parameters(), lr=lr, weight_decay=weight_decay, momentum=momentum, nesterov=
        )

    raise ValueError(f"Unknown optimizer: {name}")


def build_scheduler(optimizer: Optimizer, cfg: dict) -> _LRScheduler | None:
    """Build an LR scheduler from 'train.scheduler' config. Returns None if disabled."""
    sch_cfg = dict(cfg.get("train", {}).get("scheduler", {}) or {})
    name = sch_cfg.get("name", None)
    if not name:
        return None
    name = str(name).lower()

    if name == "step":
        step_size = int(sch_cfg.get("step_size", 5))
        gamma = float(sch_cfg.get("gamma", 0.1))
        return optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)
```

```
    if name == "onecycle":
        max_lr = float(sch_cfg.get("max_lr", 1e-3))
        total_steps = int(sch_cfg.get("total_steps", 0))
        if total_steps <= 0:
            raise ValueError("OneCycle requires 'total_steps' > 0.")
        return optim.lr_scheduler.OneCycleLR(optimizer, max_lr=max_lr, total_steps=total_steps)

    raise ValueError(f"Unknown scheduler: {name}")
```

- 程式碼解說
  - build_optimizer()
    - 根據 config.yaml 設定優化器
    - 目前只支援 Adam/AdamW/SGD
  - build_scheduler()
    - 根據 config.yaml 設定學習率調整器
    - 目前只支援 StepLR/OneCycleLR
- 優化器與學習率調整器定義完成

【訓練模型相關超參數】

- Data相關

```yaml
data:
  image_size: 128
  num_workers: 0
  pin_memory: true
  augment:
    enabled: true
    hflip_prob: 0.5
    rotation_deg: 5
    brightness: 0.10
    contrast: 0.10
    saturation: 0.00
    hue: 0.00
```

- 超參數說明
  - image_size: 輸入影像的尺寸
    - VGG16-BN 預訓練模型通常用 224

        - MLP 可以用 96~128
    - num_workers: DataLoader 的 worker 個數
    - pin_memory: 是否 pin memory
    - augment: 是否啟用增強訓練
        - hflip_prob: 水平翻轉機率
        - rotation_deg: 旋轉角度
        - brightness: 調整亮度
        - contrast: 調整對比
        - saturation: 調整飽和度
        - hue: 調整色調
- Model相關

```yaml
model:
# Switch between "mlp" and "vgg16"
name: "mlp"

# MLP options (used only when name == "mlp")
mlp:
  hidden: [512, 256]
  dropout: 0.5

# VGG16 options (used only when name == "vgg16")
vgg16:
  pretrained: true
  freeze_features: false
  classifier_hidden: 512
  dropout: 0.5
```

- 超參數說明
    - name: 模型名稱
        - mlp: 模型為多層感知器
        - vgg16: 模型為 VGG16-BN 預訓練模型
    - mlp:
        - hidden: 隱藏層配置
        - dropout: 防止 overfitting
    - vgg16:
        - pretrained: 是否使用預訓練模型
        - freeze_features: 是否冻結特徵層
        - classifier_hidden: 最後輸出的分類數
        - dropout: 防止 overfitting
- Training相關

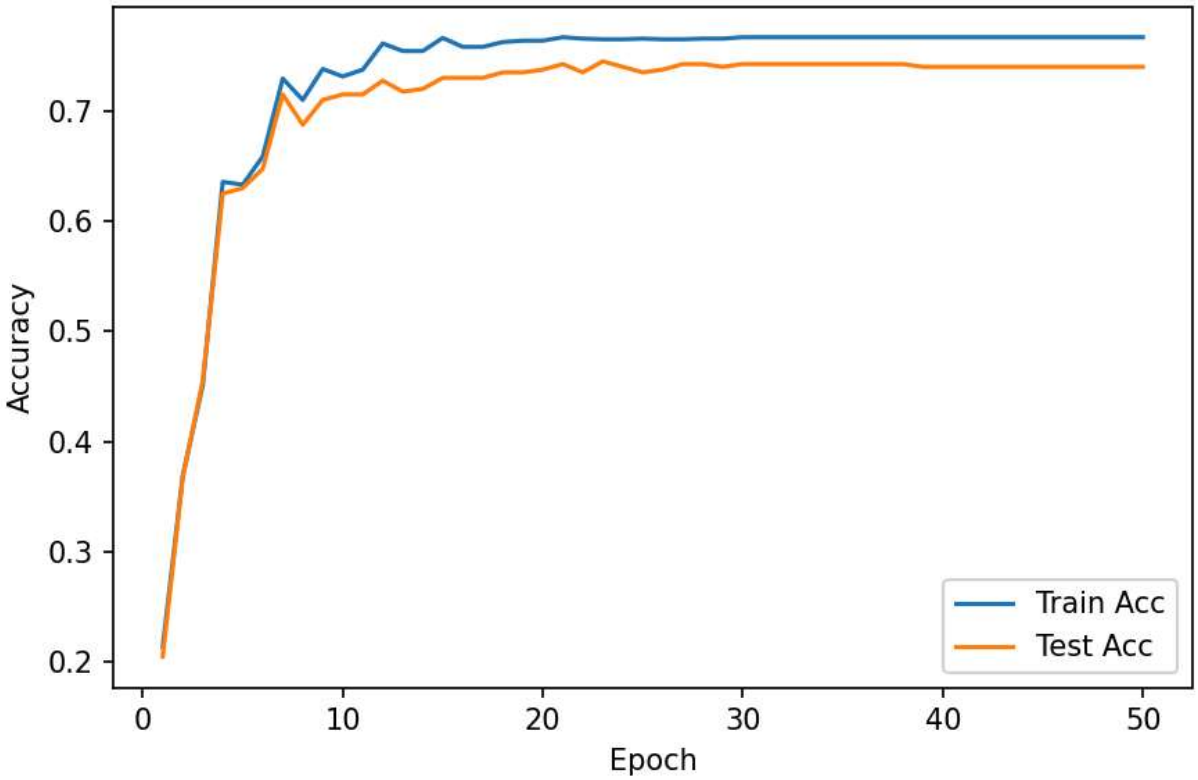
```yaml
train:
  epochs: 10
  batch_size: 32

  loss:
    name: cross_entropy
    weight: null

  optimizer:
    name: adamw
    lr: 1e-4
    weight_decay: 1e-4
    betas: [0.9, 0.999]
    momentum: 0.9

  scheduler:
    name: step
    step_size: 3
    gamma: 0.5
```

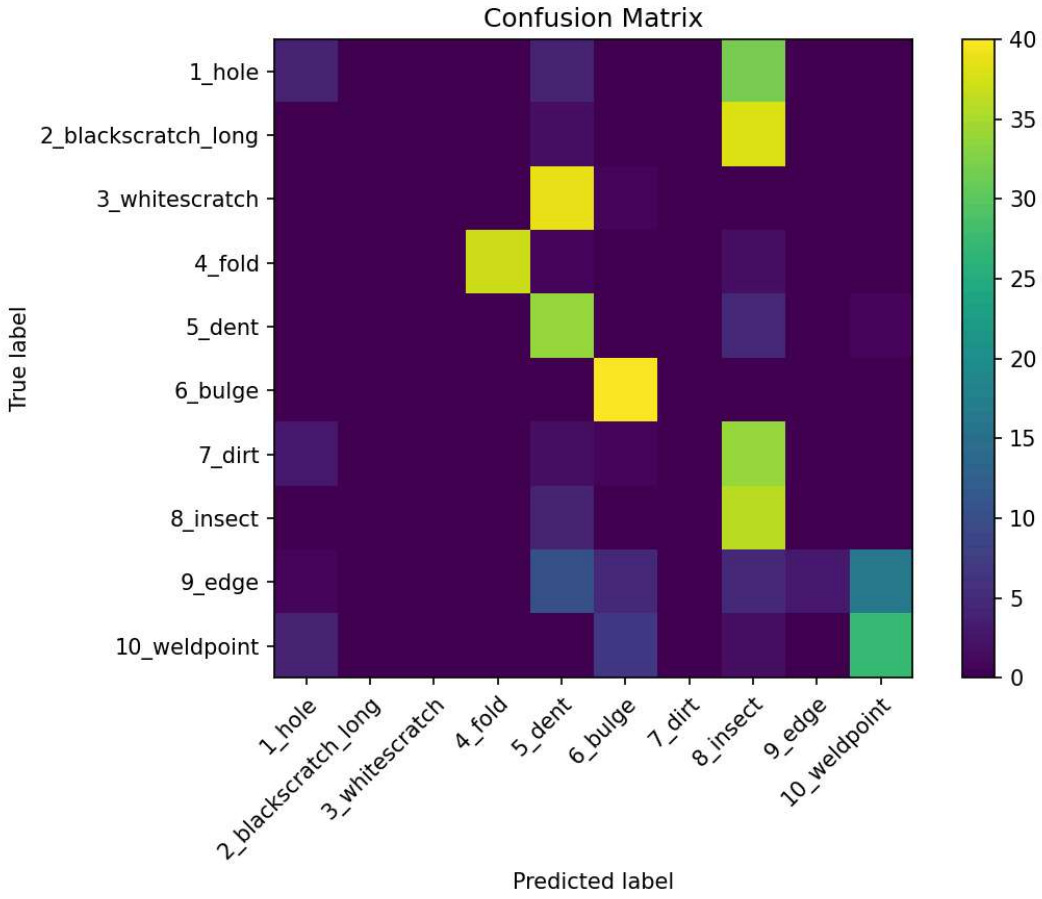
- 超參數說明
  - epochs: 訓練次數
  - batch_size: 批次大小
  - loss:
    - name: 損失函數名稱
    - weight: 損失函數權重
  - optimizer:
    - name: 優化器名稱
    - lr: 學習率
    - weight_decay: 權重衰減
    - betas: Adam/AdamW 的參數
    - momentum: SGD 的參數
  - scheduler:
    - name: 學習率調整器名稱
    - step_size: StepLR 的參數
    - gamma: StepLR 的參數

【評估與推論結果】

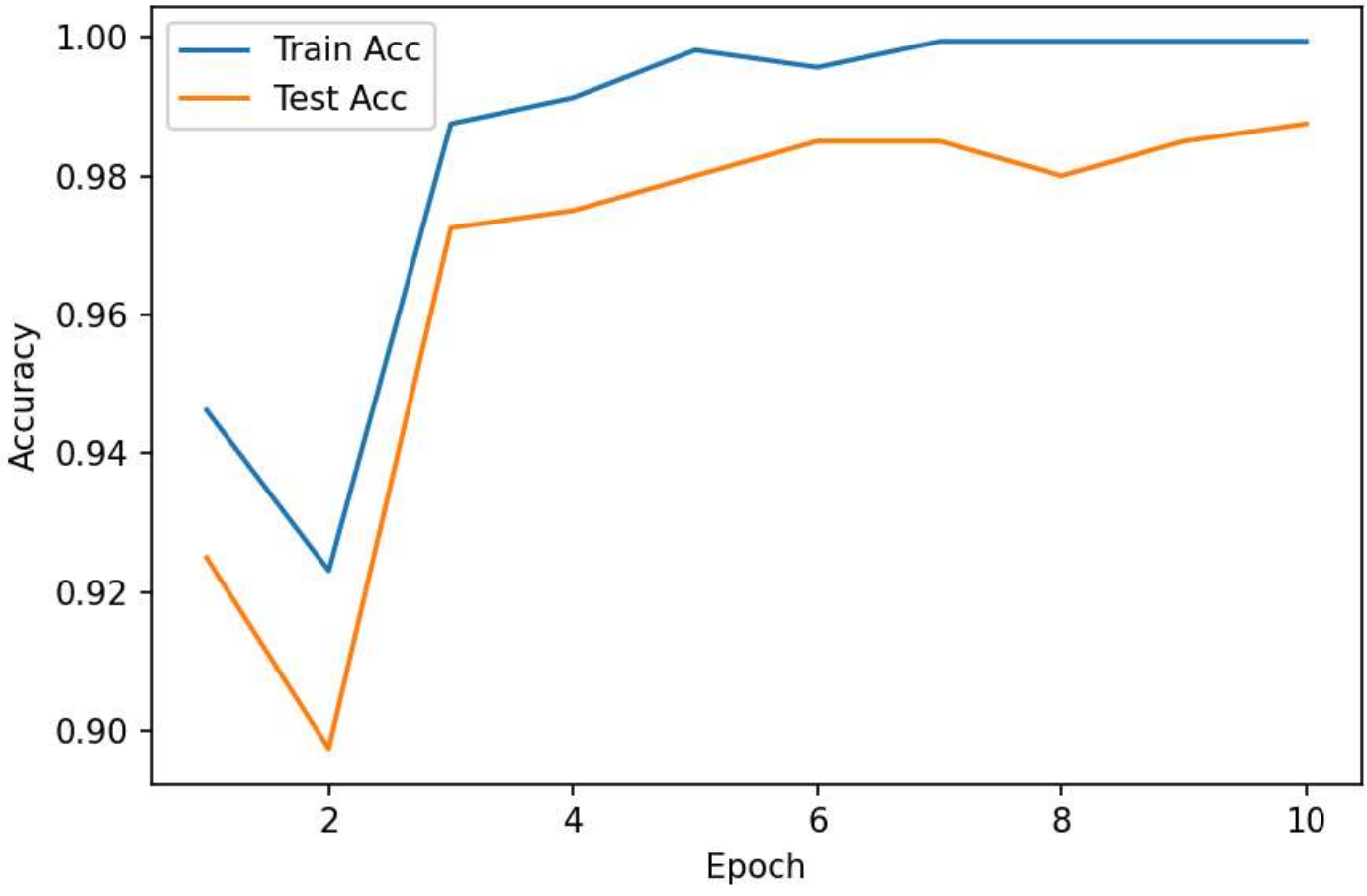
- MLP

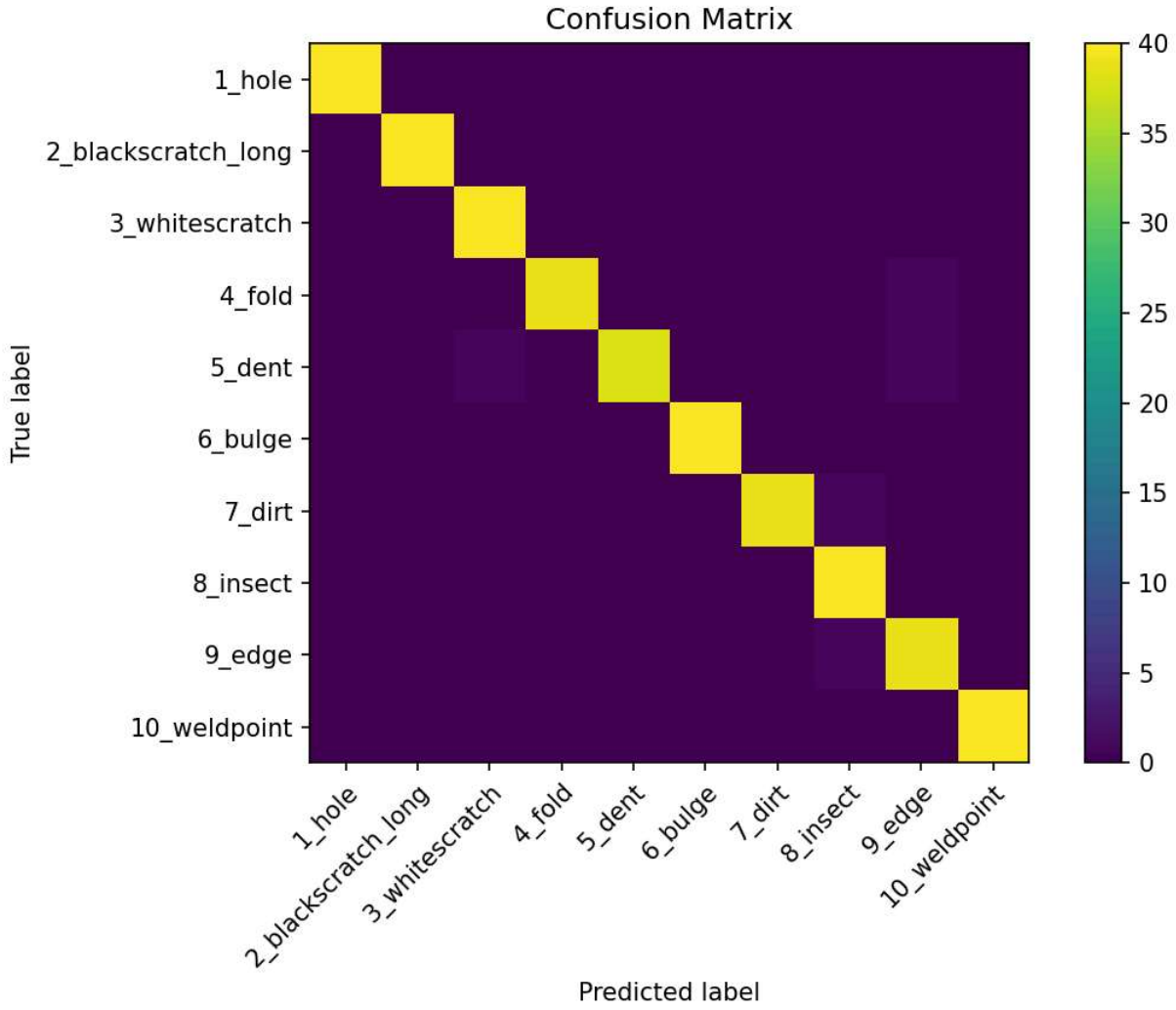
- 原本訓練結果很爛，所以做了一點調整
- Epochs: 50
  - 原本設定 epochs: 10
- lr: 3e-4
  - 原本設定 lr: 1e-4
- hidden: [1024, 512, 256]
  - 原本設定 hidden: [512, 256]
- dropout: 0.3
  - 原本設定 dropout: 0.5





- 看結果調整後大概epochs設定25就可以了
- VGG16





- 前3個epoch就建立高performance