Instructions for Model Training and Inference

1.模型架構(Model Definition)

- 原始完整程式碼: 114552029_DL_lab1

1.1 損失函數:AdvancedLoss (Huber + L1)

為何使用？

標準的 MSE(均方誤差)損失對離群值非常敏感。在房價預測中，一筆極高價的豪宅樣本就可能產生巨大的誤差，進而主導整個訓練過程並導致梯度爆炸。為了處理這種極端的價格分布，因此使用了 AdvancedLoss。

技術原理 (對應程式碼):

- **Huber Loss**:它在誤差較小 (`|error| < delta`) 時等同於 MSE，保持平滑的梯度;在誤差較大時 (`|error| >= delta`) 等同於 MAE，降低離群值的影響，使模型更穩健 (robust)。
- **L1 正則**:並額外加入 0.05 倍的 L1 損失 (MAE)，進一步增強模型對絕對誤差的懲罰，有助於優化 MAPE 這類相對指標。

```
class AdvancedLoss(nn.Module):
    def __init__(self, smoothing: float = 0.0, l1_alpha: float = 0.05, huber_delta: float = 1.0):
        super().__init__()
        self.smoothing = smoothing
        self.l1_alpha = l1_alpha
        self.huber_delta = huber_delta

    def forward(self, pred, target):
        # 1. Huber 損失:對離群值穩健
        huber_loss = F.huber_loss(pred, target, delta=self.huber_delta)
        # 2. L1 損失 (MAE):增強穩健性
        l1_reg = torch.mean(torch.abs(pred - target))
        # 3. 結合兩者
        return huber_loss + self.l1_alpha * l1_reg
```

1.2 神經網路架構:AdvancedHousePrNN

設計了一個 6 層的深度網路(輸入層 -> 1024 -> 512 -> 256 -> 128 -> 64 -> 輸出層)。為了使這個深度網路能穩定訓練並具備強大的泛化能力，在架構中整合了 `BatchNorm`、`SiLU` 和 `Dropout` 。

```
class AdvancedHousePriceNN(nn.Module):
    def __init__(self, n_features, y_mean=0.0):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_features, 1024),
            nn.BatchNorm1d(1024),
            nn.SiLU(),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.BatchNorm1d(512),
            nn.SiLU(),
            nn.Dropout(0.25),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.SiLU(),
            nn.Dropout(0.2),
```

```
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.SiLU(),
            nn.Dropout(0.15),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.SiLU(),
            nn.Dropout(0.1),
            nn.Linear(64, 1)
        )
        with torch.no_grad():
            self.net[-1].bias.fill_(float(y_mean))

    def forward(self, x):
        return self.net(x)
```

關鍵技術 **1**：**BatchNorm + Dropout (正則化組合)**

為何使用？

- `nn.BatchNorm1d` (批次標準化): 在 6 層深的網路中，如果沒有 BatchNorm，梯度很容易消失或爆炸。BatchNorm 透過標準化每層的輸出 (均值為0, 標準差為1)，極大地加速收斂並穩定訓練過程 。
- `nn.Dropout` (隨機失活): 正則化技術之一。它在訓練時隨機「關閉」一定比例（例如 30%）的神經元，強迫網路學習更穩健的特徵，防止模型過度擬合 (Overfitting) 。

關鍵技術 **2**：**SiLU 激活函數 (為何不用 ReLU？)**

為何使用 **`SiLU()`**？

- 解決「**ReLU**」問題：ReLU 在 `x < 0` 時梯度為 0。如果一個神經元不幸更新後，其輸入恆為負，它將永遠停止學習。SiLU 在負值區仍有非零梯度，避免了此問題。

---

2. 模型訓練詳細流程 (Model Training Process)

2.1 策略：K-Fold 交叉驗證

為何使用？

為了最大化利用 26 萬筆資料，並得到一個可靠、泛化能力強的模型，查找資料後使用 5-Fold 交叉驗證，而非單次 80/20 切分。這能確保每筆資料都被當作驗證集一次，提供的 OOF 評估分數可能比單次驗證更可靠 。

執行指令 **(對應程式碼)**：

```
print(f"\n Starting K-Fold cross-validation training...")

n_splits = 5
kfold = KFold(n_splits=n_splits, shuffle=True, random_state=42)
epochs = 300
batch_size = 256
accumulation_steps = 4

oof_preds = np.zeros(X.shape[0], dtype=np.float32)
test_preds_folds = []
val_scores = []
val_metrics = []
fold_history = []

for fold, (train_idx, val_idx) in enumerate(kfold.split(X, y_log)):
```

```
    print(f"\n===== Fold {fold+1}/{n_splits} =====")

    # Data splitting (資料切分)
    X_train_fold, y_train_fold = X.iloc[train_idx], y_log[train_idx]
    X_val_fold, y_val_fold = X.iloc[val_idx], y_log[val_idx]

    # Scaling (標準化)
    # 重要：StandardScaler 在 Fold 內部 fit 訓練集，防止資料洩漏
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_fold)
    X_val_scaled = scaler.transform(X_val_fold)
    X_test_scaled = scaler.transform(X_test)

    # Data loaders (資料載入器)
    train_dataset = TensorDataset(torch.FloatTensor(X_train_scaled), torch.FloatTensor(y_train_fold).unsqueeze(1))
    val_dataset = TensorDataset(torch.FloatTensor(X_val_scaled), torch.FloatTensor(y_val_fold).unsqueeze(1))
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=False)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, drop_last=False)

    # Model setup (模型設定)
    model = AdvancedHousePriceNN(X.shape[1], y_mean=y_train_fold.mean()).to(device)
    criterion = AdvancedLoss(smoothing=0.0, l1_alpha=0.05, huber_delta=1.0)
    optimizer = optim.AdamW(model.parameters(), lr=0.003, weight_decay=1e-4, betas=(0.9, 0.999), eps=1e-8)

    # ... (訓練迴圈開始) ...
```

2.2 訓練配置：AMP、梯度累積與學習率調度

在每個 Fold 內部，使用的訓練配置：

1. 自動混合精度 (AMP) + 梯度累積 (Accumulation)為何使用？ 為了在 CUDA GPU 上加速訓練（26萬資料 * 300 epochs）並節省顯存 。

2. 學習率調度：CosineAnnealingWarmRestarts 為何使用？ 讓學習率週期性重啟，幫助模型跳出局部最優點

(對應程式碼)：

```
scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts( optimizer, T_0=50 * len(train_loader), T_mult=2,
eta_min=1e-7, last_epoch=-1)


# 1. AMP：初始化 GradScaler
scaler_amp = GradScaler()

# ... (訓練變數初始化) ...
best_val_loss = float('inf')
best_val_r2 = float('-inf')
patience_counter = 0
best_model_state = None
train_loss_log, val_loss_log, val_r2_log = [], [], []

# Training loop
for epoch in range(1, epochs + 1):
    model.train()
    train_loss_sum = 0.0
```

```
    batch_idx = 0
    optimizer.zero_grad(set_to_none=True)

    for i, (xb, yb) in enumerate(train_loader):
        xb, yb = xb.to(device), yb.to(device)
        # 2. AMP：啟用 autocast
        with autocast():
            preds = model(xb)
            # 3. 梯度累積：損失均分
            loss = criterion(preds, yb) / accumulation_steps
        # 4. AMP：縮放梯度並反向傳播
        scaler_amp.scale(loss).backward()

        # 5. 梯度累積：每 4 步更新一次
        if (i + 1) % accumulation_steps == 0:
            scaler_amp.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            scaler_amp.step(optimizer)
            scaler_amp.update()
            optimizer.zero_grad(set_to_none=True)

        train_loss_sum += loss.item() * accumulation_steps
        scheduler.step(batch_idx) # 每個 step 更新 LR
        batch_idx += 1
```

2.3 驗證與提早停止 (Early Stopping)

為何使用？

為了防止過擬合並節省訓練時間，使用雙指標（Val Loss 和 Val R^2）進行監控。只要任一指標有改善，就繼續訓練；若連續 30 個 epoch 均未改善，則停止。

執行指令 (對應程式碼)：

```
        # Validation
        model.eval()
        val_loss_sum = 0.0
        val_preds_list = []
        with torch.no_grad():
            for xb, yb in val_loader:
                # ... (驗證迴圈) ...
                val_preds_list.append(val_preds.detach().cpu().numpy().flatten())

        val_loss = val_loss_sum / max(1, len(val_loader))
        val_preds_epoch = np.concatenate(val_preds_list)
        val_r2 = r2_score(y_val_fold, val_preds_epoch)

        # Log metrics
        train_loss_log.append(train_loss_sum/len(train_loader))
        val_loss_log.append(val_loss)
        val_r2_log.append(val_r2)

        if epoch % 50 == 0 or epoch == 1:
            lr = optimizer.param_groups[0]['lr']
            print(f'Epoch {epoch:3d}/{epochs} | Train: {train_loss_log[-1]:.4f} | Val: {val_loss:.4f} | R²: {val_r2:.4f} | LR:
{lr:.7f}')

        # Early stopping with dual criteria
```

```
        improved = False
        if val_loss < best_val_loss - 1e-6:
            best_val_loss = val_loss
            improved = True
        if val_r2 > best_val_r2 + 1e-5:
            best_val_r2 = val_r2
            improved = True

        if improved:
            patience_counter = 0
            best_model_state = {k: v.detach().cpu().clone() for k, v in model.state_dict().items()}
        else:
            patience_counter += 1
            if patience_counter >= 30:
                print(f"Fold {fold+1} early stopped at epoch {epoch}")
                break

    # Load best model and evaluate
    if best_model_state is not None:
        model.load_state_dict(best_model_state)

    print(f"Fold {fold+1} Best Val Loss: {best_val_loss:.4f}, Best R²: {best_val_r2:.4f}")
    val_scores.append(best_val_loss)
    fold_history.append({'train_loss': train_loss_log, 'val_loss': val_loss_log, 'val_r2': val_r2_log})
```

---

3. 推論 (Inference) 與評估

訓練結束後，腳本會自動執行推論並產出最終結果。推論分為兩部分：對驗證集的推論（用於 OOF 評估）和對測試集的推論（用於 Kaggle 提交）。

3.1 OOF (Out-of-Fold) 預測與評估

為何使用 OOF？

OOF 預測是組合了 5 個模型各自對其「驗證集」的預測。這能提供一個對整體訓練資料集最公平、最接近真實泛化能力的評估分數，因為每個樣本的預測都來自「未訓練過它」的模型 。

執行指令 **(對應程式碼)**：

```
    # OOF predictions (此程式碼在 K-Fold 迴圈內部)
    model.eval()
    with torch.no_grad():
        oof_preds[val_idx] = model(torch.FloatTensor(X_val_scaled).to(device)).cpu().numpy().flatten()

    # Calculate fold metrics (此程式碼在 K-Fold 迴圈內部)
    fold_oof_original = np.expm1(oof_preds[val_idx]) # 反向轉換 log1p
    fold_val_original = np.expm1(y_val_fold)
    eps = 1e-9
    fold_rmse = np.sqrt(mean_squared_error(fold_val_original, fold_oof_original))
    fold_mae = mean_absolute_error(fold_val_original, fold_oof_original)
    fold_mape = np.mean(np.abs((fold_val_original - fold_oof_original)/(fold_val_original+eps))) * 100
    fold_r2 = r2_score(fold_val_original, fold_oof_original)

    val_metrics.append({
        'fold': fold+1, 'rmse': fold_rmse, 'mae': fold_mae, 'mape': fold_mape, 'r2': fold_r2
    })
```

```
    # ... (迴圈繼續) ...

# ========== Final Evaluation ==========
# (此程式碼在 K-Fold 迴圈外部)
print("\n" + "="*60)
# 打印 dL_lab1.pdf (Page 3) 中的 K-Fold 平均損失 [cite: 340]
print(f"K-Fold Mean Validation Loss: {np.mean(val_scores):.4f}")

# 計算總體 OOF 指標
y_original = np.expm1(y_log)
oof_preds_original = np.expm1(oof_preds)
val_rmse = np.sqrt(mean_squared_error(y_original, oof_preds_original))
val_mae = mean_absolute_error(y_original, oof_preds_original)
val_mape = np.mean(np.abs((y_original - oof_preds_original)/(y_original+1e-9))) * 100
val_r2 = r2_score(y_original, oof_preds_original)


print(f"\nOverall OOF Metrics:")
print(f"RMSE: ${val_rmse:,.0f}")
print(f"MAE:  ${val_mae:,.0f}")
print(f"MAPE: {val_mape:.2f}%")
print(f"R²:   {val_r2:.4f}")
print("="*60)


for metric in val_metrics:
    print(f"Fold {metric['fold']}: RMSE=${metric['rmse']:,.0f}, MAE=${metric['mae']:,.0f}, R²={metric['r2']:.4f}")
```

3.2 訓練歷史視覺化

此步驟用於驗證 5-Fold 訓練的健康度。需要檢查：

1. 損失曲線：訓練 (Train) 和驗證 (Val) 損失是否都穩定下降並收斂？
2. **R² 進展**：驗證集的 R² 是否穩定上升並達到平穩？
3. 過擬合：Train Loss 和 Val Loss 之間是否存在巨大差距？

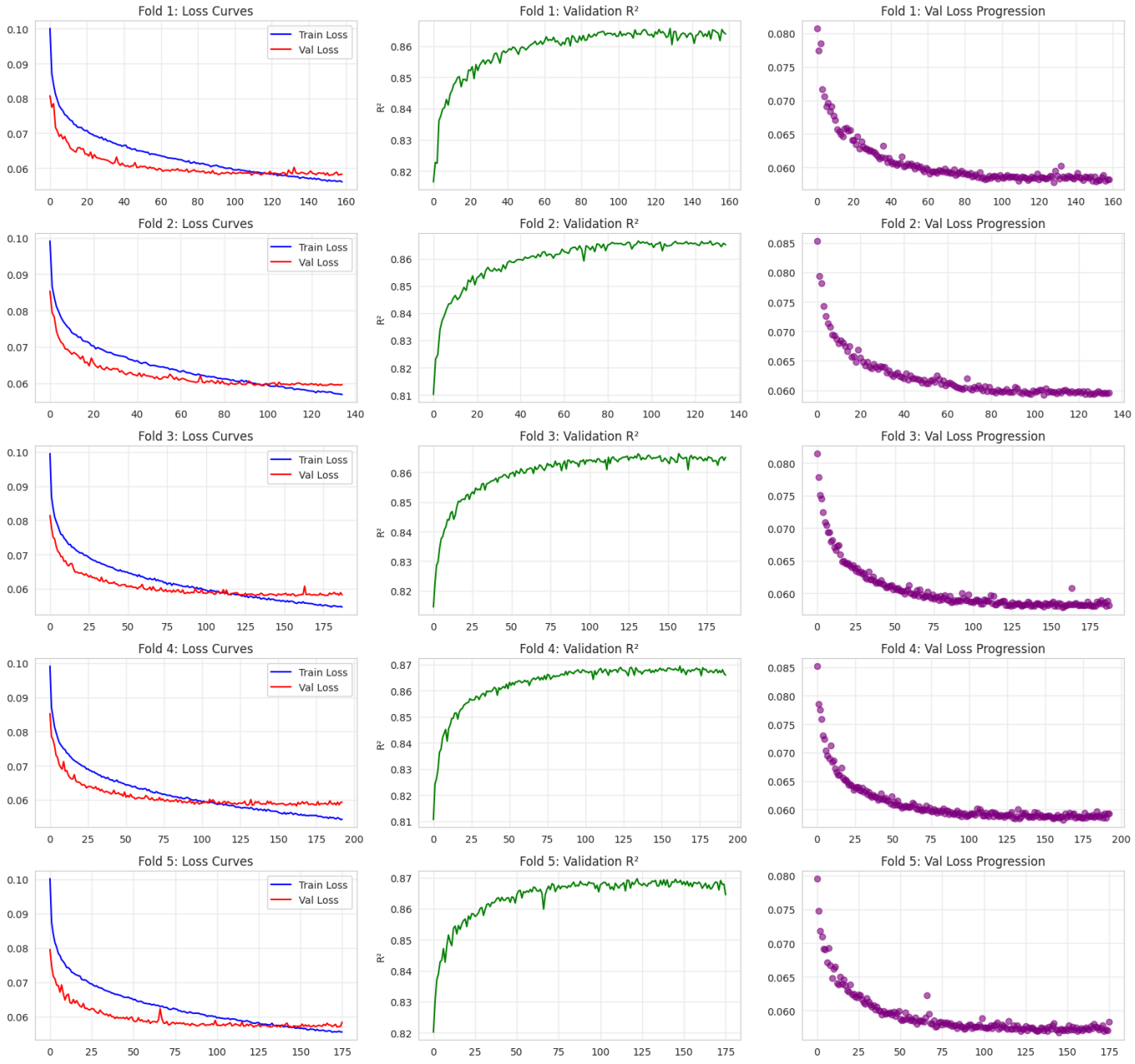
執行指令 (對應程式碼)：

```
# ========== VISUALIZATION 4: Training History ==========
print(f"\nStep 6: Training history visualization...")

def plot_fold_metrics(fold_history):
    fig, axs = plt.subplots(n_splits, 3, figsize=(16, 3*n_splits))
    for i, fh in enumerate(fold_history):
        # Loss curves
        axs[i,0].plot(fh['train_loss'], label='Train Loss', color='blue')
        axs[i,0].plot(fh['val_loss'], label='Val Loss', color='red')
        axs[i,0].set_title(f'Fold {i+1}: Loss Curves')
        axs[i,0].legend()
        axs[i,0].grid(True, alpha=0.3)

        # R² progression
        axs[i,1].plot(fh['val_r2'], color='green')
        axs[i,1].set_title(f'Fold {i+1}: Validation R²')
        axs[i,1].set_ylabel('R²')
```

```
        axs[i,1].grid(True, alpha=0.3)

        # Validation loss scatter
        axs[i,2].scatter(range(len(fh['val_loss'])), fh['val_loss'], alpha=0.6, color='purple')
        axs[i,2].set_title(f'Fold {i+1}: Val Loss Progression')
        axs[i,2].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
    print("→ Training curves show model learning and early stopping effectiveness")

plot_fold_metrics(fold_history)
```

訓練歷程視覺化

1. 展示了 5-Fold 交叉驗證中，每一折的「損失曲線」(Loss Curves)、「R^2」和「驗證損失散點圖」。
2. 分析：如 → Training curves show model learning and early stopping effectiveness 如圖所示，所有fold的訓練損失（藍線）和驗證損失（紅線）均穩定下降並收斂，R^2（綠線）也快速上升至平穩，證明模型學習有效且提早停止機制成功防止了過度擬合。



3.3 OOF 預測分析視覺化

此步驟使用 OOF 預測來診斷模型的偏差 (Bias) 和方差 (Variance)。

1. 預測 **vs** 真實：理想情況下，所有點應落在 y=x 的紅線上。
2. 殘差分布：理想情況下，殘差 (True - Predicted) 應呈常態分布，且中心在 0。
3. 執行指令 **(**對應程式碼**)**：

```
print(f"\n Step 7: Prediction analysis...")
def plot_pred_vs_true(y_true, y_pred):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5)
    # Predictions vs Actual
    ax1.scatter(y_true, y_pred, alpha=0.4, s=20)
    min_val, max_val = y_true.min(), y_true.max()
    ax1.plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2, label='Perfect Prediction')
    ax1.set_xlabel('True Price')
    ax1.set_ylabel('OOF Predicted Price')
    ax1.set_title('Predictions vs Actual Values')
    ax1.legend()
    ax1.grid(True, alpha=0.3)
    # Residual distribution
    residual = y_true - y_pred
    sns.histplot(residual, bins=50, kde=True, color='royalblue', ax=ax2)
    ax2.axvline(0, color='red', linestyle='--', linewidth=2, label='Zero Error')
    ax2.set_title('Residual Distribution')
    ax2.set_xlabel('True - Predicted')
    ax2.legend()
    ax2.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()
    print("→ Prediction analysis shows model accuracy and error distribution")
```

預測分析

- 左圖 **(Predictions vs Actual Values)**：OOF 預測值（藍點）緊密圍繞在「完美預測線」（紅色虛線）周圍，顯示模型預測與真實值高度相關。
- 右圖 **(Residual Distribution)**：殘差（真實值 - 預測值）的分布呈現一個極窄且高峰的鐘形，中心集中在 0，表明模型沒有系統性偏差 (Bias)。
- 分析：如 下圖所示，模型整體準確度高，誤差分布健康。