Analysis

1. Jika menggunakan model MLP dengan 3 hidden layer (256-128-64) menghasilkan underfitting pada dataset ini, modifikasi apa yang akan dilakukan pada arsitektur? Jelaskan alasan setiap perubahan dengan mempertimbangkan bias-variance tradeoff!

   Untuk mengatasi underfitting dengan 3 hidden layer dapat dimodifikasi pada bagian layer yaitu :

   a. Ubah banyaknya neuron per layer, yang tadinya menggunakan (256-128-64) ubah menjadi missal 512-256-128. Gunakan lebih banyak layer missal yang sebelumnya menggunakan 3 layer ubah menjadi 4-5 layer Dengan lebih banyak unit dan layer, model punya kekuatan representasi lebih tinggi untuk memetakan pola kompleks di data.
   b. Menurunkan nilai dropout rate, hal ini dikarenakan Dropout membantu mencegah overfitting, tapi terlalu tinggi akan merusak sinyal penting, bikin underfitting.
   c. Gunakan aktivasi lebih kompleks Misal LeakyReLU atau ELU daripada ReLU murni.hal ini dikarenakan Aktivasi ini bisa membantu gradien tidak mati, memfasilitasi pembelajaran di jaringan yang lebih dalam.
   d. Optimasi Learning rate, Jika learning rate terlalu kecil, bobot bergerak lambat dan mungkin belum mencapai minima global dalam jumlah epoch yang sama.

   Contoh implementasi

```
def build_clf_model(input_dim):
    model = models.Sequential([
        layers.Input(shape=(input_dim,)),
        layers.Dense(512),
        layers.LeakyReLU(alpha=0.1),
        layers.BatchNormalization(),
        layers.Dropout(0.1),

        layers.Dense(256),
        layers.LeakyReLU(alpha=0.1),
        layers.BatchNormalization(),
        layers.Dropout(0.1),

        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.1),

        layers.Dense(64, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.1),

        layers.Dense(2, activation='softmax')
    ])
    model.compile(
        optimizer=optimizers.Adam(learning_rate=1e-3),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
```

```
    )
    return model

clf_model = create_clf_model_stronger(X_tr_clf_p.shape[1])
```

2. Selain MSE, loss function apa yang mungkin cocok untuk dataset ini? Bandingkan kelebihan dan kekurangannya, serta situasi spesifik di mana alternatif tersebut lebih unggul daripada MSE!

LOSS	KELEBIHAN	KEKURANGAN	WAKTU UNGGUL
MAE (Mean Absolute Error)	• Tidak mudah terpengaruh terhadap faktor outlier • Interpretasi langsung (median)	Gradien konstan → bisa lebih sulit konvergensi halus	Ketika data dengan outlier tak dihapus semua
Huber Loss	Kombinasi MSE & MAE: kuadratik di dekat error kecil, linear untuk outlier	Perlu tuning parameter delta	Ketika ada outlier tapi ingin gradien tetap halus
Log-cosh	Mirip Huber, smooth	Komputasi lebih berat	Butuh fungsi loss yang sangat smooth dan tidak terpengaruh outlier

Contoh implementasi

MAE
model.compile(loss='mae', optimizer=...)
Huber Loss
model.compile(loss=tf.keras.losses.Huber(delta=1.0), optimizer=...)

3. Jika salah satu fitur memiliki range nilai 0-1, sedangkan fitur lain 100-1000, bagaimana ini memengaruhi pelatihan MLP? Jelaskan mekanisme matematis (e.g., gradien, weight update) yang terdampak!
    a. Gradient Magnitude
       Bobot terkait fitur besar (skala 100–1000) akan menghasilkan gradien jauh lebih besar daripada fitur 0–1, sehingga update bobot dominan pada satu fitur saja.
    b. Conditioning Matrix
       Matriks Hessian (atau aproksimasi diagonal) jadi ill-conditioned, memperlambat konvergensi.
    c. Weight Update

$$\mathrm{w} \leftarrow \mathrm{w} - \eta \frac{\partial \mathrm{w}}{\partial \mathrm{L}}$$

       $\frac{\partial \mathrm{w}}{\partial \mathrm{L}}$ dari fitur skala besar besar juga → bobot menyesuaikan terlalu drastis.

Mitigasi yang dapat dilakukan adalah menggunakan StandardScaler() di create_preprocessing_pipeline untuk menyetarakan mean≈0 dan var≈1, sehingga gradien dan update bobot menjadi seimbang antar-fitur.

4. Tanpa mengetahui nama fitur, bagaimana Anda mengukur kontribusi relatif setiap fitur terhadap prediksi model? Jelaskan metode teknikal (e.g., permutation importance, weight analysis) dan keterbatasannya!
   Terdapat beberapa metode
   a. Permutation Importance
      - Acak (shuffle) satu kolom pada data validasi, lihat drop di metrik (misal AUC). Namun terdapat keterbatasan dimana komputasi per kolomnya mahal, mengabaikan interaksi kompleks.
   b. SHAP (SHapley Additive exPlanations)
      - Memecah prediksi ke kontribusi fitur berdasarkan teori Shapley. Namun DeepExplainer untuk MLP memerlukan banyak memori, dan hasil bisa bias jika data distribusi tak sesuai asumsi.
   c. Weight Analysis
      - Lihat bobot pada layer pertama yaitu fitur dengan bobot input lebih besar → diasumsikan lebih penting. Tetapi, Hanya relevan untuk model linier di awal; tidak mengambil efek non-linear dan interaksi.

Contoh implementasi
explainer = shap.DeepExplainer(best_model, X_te_clf_p[:100]) shap.summary_plot(...)

5. Bagaimana Anda mendesain eksperimen untuk memilih learning rate dan batch size secara optimal? Sertakan analisis tradeoff antara komputasi dan stabilitas pelatihan!
   a. Grid/Random Search atas kombinasi $\{\eta\} \times \{B\}$
   b. Learning Rate Range Test (Leslie Smith):
      - Jalankan beberapa batch, naikkan η secara eksponensial, cari η di mana loss mulai meledak → pilih ∼ 1/10 dari nilai itu.
   c. Batch Size Trade-off:
      - Kecil (32–64): estimasi gradien stochastik, lebih cepat iterasi, namun mungkin noisy.
      - Besar (256–512): estimasi gradien lebih stabil → learning rate bisa lebih besar, tapi memerlukan memori lebih banyak dan setiap epoch lebih lambat.
   d. Stabilitas vs Komputasi:
      - Learning Rate terlalu tinggi → loss divergen; terlalu rendah → pelatihan lambat.
      - Batch size besar → overhead IO kecil sehingga tiap epoch cepat, tapi butuh GPU memory besar.

   Contoh Implementasi

Learning Rate Range Test

class LRFinder(callbacks.Callback):
  def __init__(self, start_lr=1e-5, end_lr=1):
  super().__init__()
  self.start_lr, self.end_lr = start_lr, end_lr
  self.lrs, self.losses = [], []

```
            def on_batch_end(self, batch, logs):
                lr = self.start_lr * (self.end_lr/self.start_lr)**(batch/self.params['steps'])
                tf.keras.backend.set_value(self.model.optimizer.lr, lr)
                self.lrs.append(lr)
                self.losses.append(logs['loss'])

        # jalankan range test
        lr_finder = LRFinder()
        model_lr = create_clf_model_stronger(X_tr_clf_p.shape[1])
        model_lr.fit(
            X_tr_clf_p, y_tr_clf,
            epochs=1,
            batch_size=128,
            callbacks=[lr_finder]
)
```

Hyperparameter Tuning dengan Keras Tuner

```
        def build_model_hp(hp):
            inputs = layers.Input(shape=(X_tr_clf_p.shape[1],))
            x = inputs
            for i in range(hp.Int('n_layers', 2, 5)):
                units = hp.Int(f'units_{i}', min_value=64, max_value=512, step=64)
                x = layers.Dense(units, activation='relu')(x)
                x = layers.Dropout(hp.Float(f'dropout_{i}', 0.1, 0.5, step=0.1))(x)
            outputs = layers.Dense(2, activation='softmax')(x)
            model = models.Model(inputs, outputs)
            model.compile(
                optimizer=optimizers.Adam(
                    hp.Float('learning_rate', 1e-4, 1e-2, sampling='log')
                ),
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy']
            )
            return model

        tuner = kt.Hyperband(
            build_model_hp,
            objective='val_accuracy',
            max_epochs=30,
            factor=3,
            directory='hp_dir',
            project_name='clf_tuning'
        )
        tuner.search(X_tr_clf_p, y_tr_clf, validation_data=(X_te_clf_p, y_te_clf),
        epochs=30, batch_size=hp.Choice('batch_size', [32,64,128]))
        best_hp = tuner.get_best_hyperparameters()[0]
print(best_hp.values)
```