Topik : 4.3. Evaluasi Trade-off

Objective : Uji beberapa nilai noise_multiplier → analisis akurasi vs ε

Task : Plot hasil trade-off

Source :
https://www.tensorflow.org/federated/tutorials/federated_learning_with_differential_privacy

Differential Privacy in TFF

Differential Privacy (DP) Adalah metode yang banyak digunakan untuk membatasi dan mengukur kebocoran privasi data sensitive Ketika melakukan tugas pembelajaran. Melatih model dengan DP Tingkat pengguna menjamin bahwa model tidak akan belajar sesuatu yang signifikan tentang data individu tertentu, tetapi tetap dapat mempelajari pola yang ada pada data dari banyak client.

Kita akan melatih model pada dataset **federated EMNIST**. Terdapat trade-off yang melekat antara utilitas dan privasi, sehingga mungkin sulit untuk melatih model dengan tingkat privasi tinggi yang memiliki performa setara dengan model *non-private* terbaik. Demi kecepatan dalam tutorial ini, kita hanya akan melatih selama **100 putaran**, sehingga mengorbankan sebagian kualitas demi mendemonstrasikan cara melatih dengan privasi tinggi. Jika kita menggunakan lebih banyak putaran pelatihan, kita tentu bisa mendapatkan model privat dengan akurasi yang agak lebih tinggi, tetapi tidak akan setinggi model yang dilatih tanpa DP.

1. Import

fl_df.py
```python
#================================ IMPORT ================================
import collections
import dp_accounting
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_federated as tff
import matplotlib.pyplot as plt
import seaborn as sns

```

2. Download and pre-process the federated EMNIST dataset.

```python
#================================ Download and preprocess the federated EMNIST dataset. ================================
def get_emnist_dataset() :
    emnist_train, emnist_test = tff.simulation.datasets.emnist.load_data(
        only_digits=True
    )

    def element_fn(element):
        return  collections.OrderedDict(
            x=tf.expand_dims(element['pixels'], -1), y=element['label']
        )

    def preprocess_train_dataset(dataset):
        # Use buffer_size same as the maximum client dataset size,
        # # 418 for Federated EMNIST
        return (dataset.map(element_fn).shuffle(buffer_size=418).repeat(1).batch(32, drop_remainder=False))

    def preprocess_test_dataset(dataset) :
        return dataset.map(element_fn).batch(128, drop_remainder=False)

    emnist_train = emnist_train.preprocess(preprocess_train_dataset)
    emnist_test = preprocess_test_dataset(emnist_test.create_tf_dataset_from_all_clients())
    return emnist_train, emnist_test
train_data, test_data = get_emnist_dataset()
```

3. Define Our Model

```python
#================================ Define our model. ================================
def my_model_fn():
    model = tf.keras.models.Sequential([
        tf.keras.layers.Reshape(input_shape=(28,28,1), target_shape=(28 * 28)),
        tf.keras.layers.Dense(200, activation=tf.nn.relu),
        tf.keras.layers.Dense(200, activation=tf.nn.relu),
        tf.keras.layers.Dense(10),
    ])
    return tff.learning.models.from_keras_model(
        keras_model = model,
        loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        input_spec = test_data.element_spec,
        metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
    )

```

4. Determine the noise sensitivity of the model

Untuk mendapatkan jaminan Differential Privacy (DP) di level pengguna, kita perlu mengubah algoritma dasar Federated Averaging dengan dua cara :

1. Update model dari klien harus di-clip sebelum dikirim ke server, sehingga membatasi pengaruh maksimum dari setiap klien.
2. Server harus menambahkan noise yang cukup pada jumlah total update pengguna sebelum melakukan averaging, agar pengaruh terburuk dari satu klien dapat tersamarkan.

Untuk proses *clipping*, kita menggunakan metode *adaptive clipping* dari Andrew dkk. (2021), *Differentially Private Learning with Adaptive Clipping*, sehingga tidak perlu menetapkan norma clipping secara eksplisit.

Penambahan noise umumnya akan menurunkan utilitas model, tetapi kita bisa mengontrol jumlah noise pada setiap rata – rata update dengan dua parameter : standar deviasi dari noise Gaussian yang ditambahkan pada jumlah total , serta jumlah klien yang dilibatkan dalam rata – rata. Strategi kita Adalah pertama – tama menentukan seberapa besar noise yang masih bisa ditoleransi model dengan jumlah klien per ronde yang relative kecil tanpa kehilangan kualitas model yang signifikan. Kemudian , untuk melatih model final, kita bisa meningkatkan jumlah noise pada aggregasi, sambil secara proposional menambah jumlah klien per ronde (dengan asumsi dataset cukup besar untuk mendukung jumlah klien tersebut). Hal ini tidak mungkin berdampak signifikan pada kualitas model , karena efeknya hanya mengurangi variansi akibat *sampling* klien (dan memang akan kita lakukan verifikasi bahwa ini tidak memengaruhi kualitas pada kasus kita).

Untuk itu, kita akan melatih serangkaian model dengan 50 klien per ronde, dengan jumlah noise yang meningkat. Secara khusus, kita meningkatkan nilai *noise_multiplier,* yaitu rasio standar deviasi noise terhadap clipping norm. kerena kita menggunakan *adaptive clipping,* maka besar actual dari noise akan berubah dari ronde ke ronde.

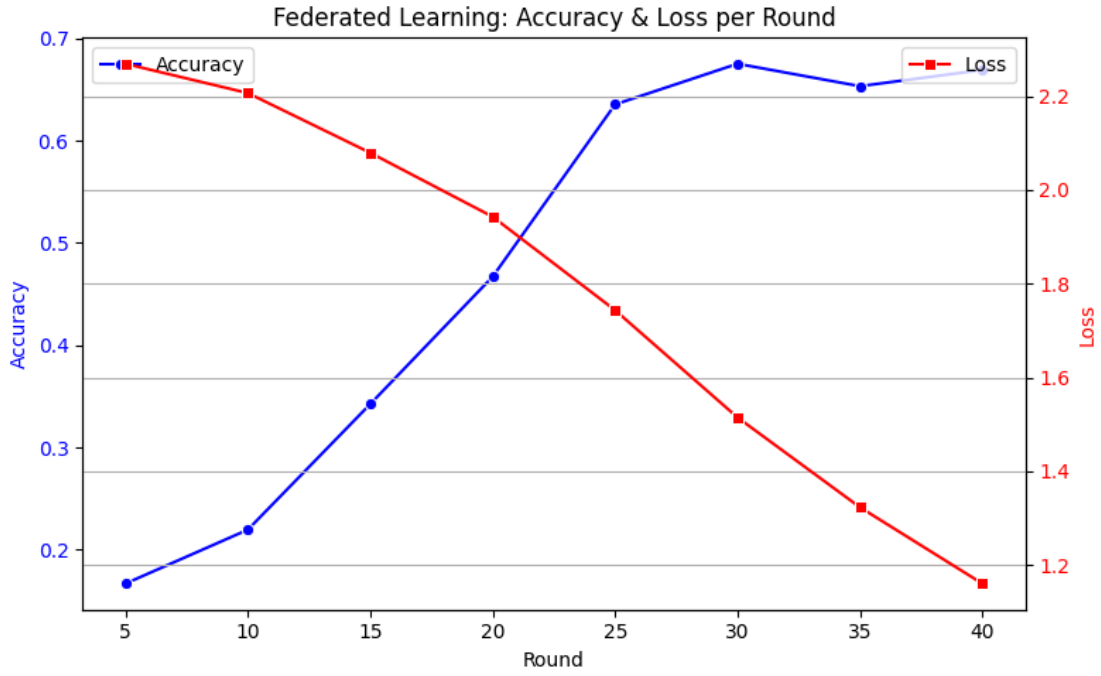
```python

total_clients = len(train_data.client_ids)

def train(rounds, noise_multiplier, clients_per_round, data_frame):
  # Using the `dp_aggregator` here turns on differential privacy with adaptive
  # clipping.
  aggregation_factory = tff.learning.model_update_aggregator.dp_aggregator(
      noise_multiplier, clients_per_round)

  # We use Poisson subsampling which gives slightly tighter privacy guarantees
  # compared to having a fixed number of clients per round. The actual number of
  # clients per round is stochastic with mean clients_per_round.
  sampling_prob = clients_per_round / total_clients

  # Build a federated averaging process.
  # Typically a non-adaptive server optimizer is used because the noise in the
  # updates can cause the second moment accumulators to become very large
  # prematurely.
  learning_process = tff.learning.algorithms.build_unweighted_fed_avg(
        my_model_fn,
        client_optimizer_fn=tff.learning.optimizers.build_sgdm(0.01),
        server_optimizer_fn=tff.learning.optimizers.build_sgdm(
            1.0, momentum=0.9),
        model_aggregator=aggregation_factory)

  eval_process = tff.learning.build_federated_evaluation(my_model_fn)

  # Training loop.
  state = learning_process.initialize()
  for round in range(rounds):
    if round % 5 == 0:
      model_weights = learning_process.get_model_weights(state)
      metrics = eval_process(model_weights, [test_data])['eval']
      if round < 25 or round % 25 == 0:
        print(f'Round {round:3d}: {metrics}')
      data_frame = data_frame.append({'Round': round,
                                      'NoiseMultiplier': noise_multiplier,
                                      **metrics}, ignore_index=True)

    # Sample clients for a round. Note that if your dataset is large and
    # sampling_prob is small, it would be faster to use gap sampling.
    x = np.random.uniform(size=total_clients)
    sampled_clients = [
        train_data.client_ids[i] for i in range(total_clients)
        if x[i] < sampling_prob]
    sampled_train_data = [
        train_data.create_tf_dataset_for_client(client)
        for client in sampled_clients]

    # Use selected clients for update.
    result = learning_process.next(state, sampled_train_data)
    state = result.state
    metrics = result.metrics

  model_weights = learning_process.get_model_weights(state)
  metrics = eval_process(model_weights, [test_data])['eval']
  print(f'Round {rounds:3d}: {metrics}')
  data_frame = data_frame.append({'Round': rounds,
                                  'NoiseMultiplier': noise_multiplier,
                                  **metrics}, ignore_index=True)

  return data_frame
```

```python
#Logging Akurasi
data_frame = pd.DataFrame()
round = 100
clients_per_round = 50

for noise_multiplier in [0.0, 0.5, 0.75, 1.0]:
    print(f'Starting training with noise multiplier: {noise_multiplier}')
    data_frame = train(round, noise_multiplier, clients_per_round, data_frame)
    print()
```

Starting training with noise multiplier: 0.0

```python
#============================== Logging Akurasi ==============================
data_frame = pd.DataFrame()
rounds = 40
clients_per_round = 40

# Hanya 1 noise multiplier: 0.0
noise_multiplier = 0.0
print(f'Starting training with noise multiplier: {noise_multiplier}')
data_frame = train(rounds, noise_multiplier, clients_per_round, data_frame)
print("Training selesai.")
```

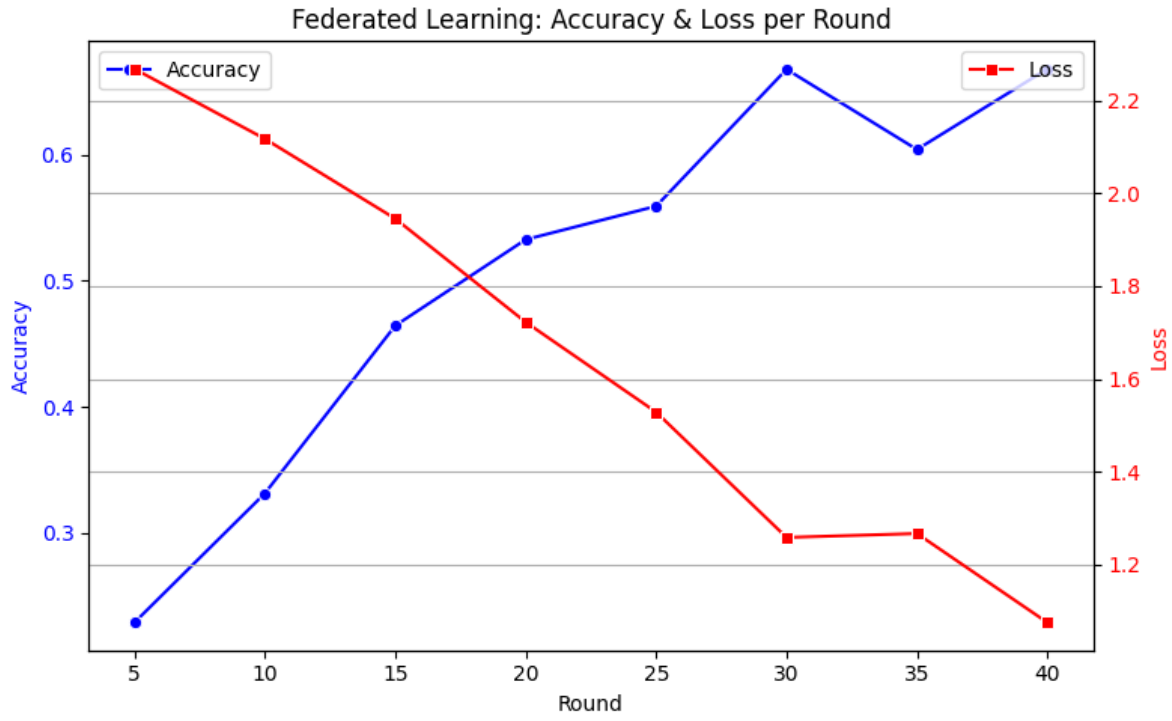
```
Skipping registering GPU devices...
Round   5: {'loss': 2.2691152095794678, 'sparse_categorical_accuracy': 0.16692790389060974}
Round  10: {'loss': 2.2066309452056885, 'sparse_categorical_accuracy': 0.21963165700435638}
Round  15: {'loss': 2.079280376434326, 'sparse_categorical_accuracy': 0.342476487159729}
Round  20: {'loss': 1.9429410696029663, 'sparse_categorical_accuracy': 0.46701115369796753}
Round  25: {'loss': 1.7436435222625732, 'sparse_categorical_accuracy': 0.6353350281715393}
Round  30: {'loss': 1.5149896144866943, 'sparse_categorical_accuracy': 0.6750832796096802}
Round  35: {'loss': 1.3235301971435547, 'sparse_categorical_accuracy': 0.6531886458396912}
Round  40: {'loss': 1.1607935428619385, 'sparse_categorical_accuracy': 0.6690585613250732}
Training selesai.
```



Starting training with noise multiplier: 0.5

```
134
135   #================================ Logging Akurasi ================================
136   data_frame = pd.DataFrame()
137   rounds = 40
138   clients_per_round = 40
139
140   # Hanya 1 noise multiplier: 0.5
141   noise_multiplier = 0.5
142   print(f'Starting training with noise multiplier: {noise_multiplier}')
143   data_frame = train(rounds, noise_multiplier, clients_per_round, data_frame)
144   print("Training selesai.")
```

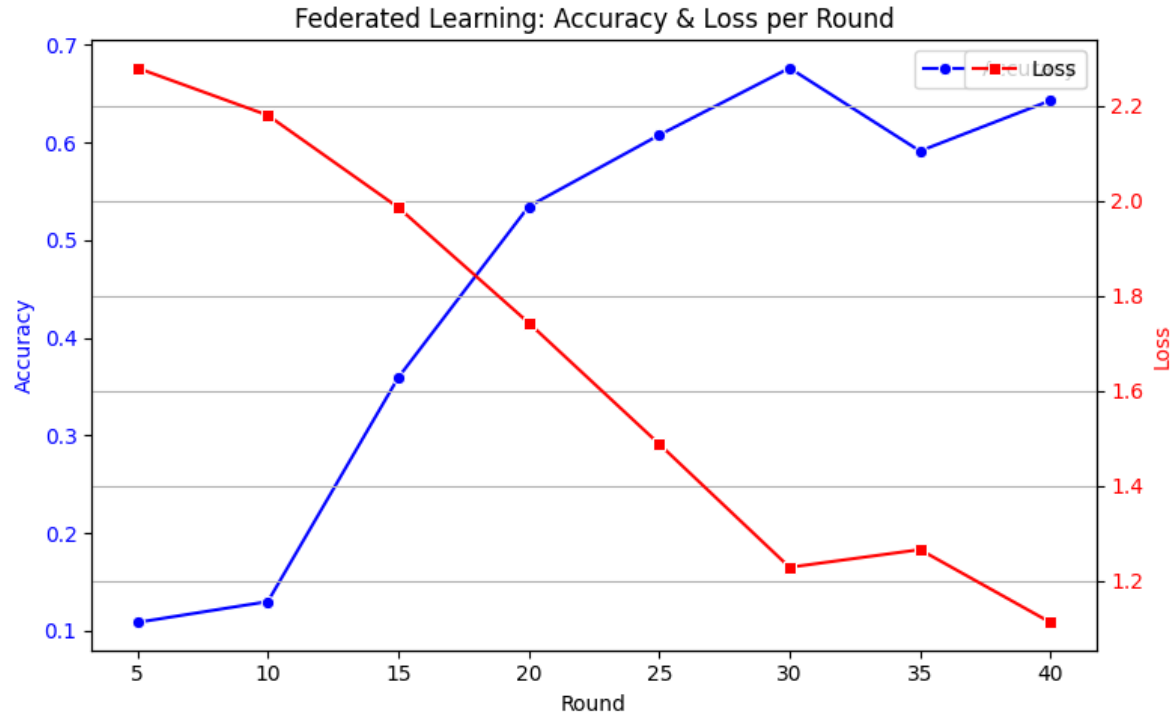
```
Id like to use GPU. Follow the guide at https://www.tensorflow.org/install/gpu for how to download and setup the require
Skipping registering GPU devices...
Round   5: {'loss': 2.2679574489593506, 'sparse_categorical_accuracy': 0.22918298840522766}
Round  10: {'loss': 2.117283582687378, 'sparse_categorical_accuracy': 0.3315047025680542}
Round  15: {'loss': 1.945177435874939, 'sparse_categorical_accuracy': 0.46441516280174255}
Round  20: {'loss': 1.7222167253494263, 'sparse_categorical_accuracy': 0.5326459407806396}
Round  25: {'loss': 1.528311729431523, 'sparse_categorical_accuracy': 0.5590713024139404}
Round  30: {'loss': 1.2581448554992676, 'sparse_categorical_accuracy': 0.6675401926040649}
Round  35: {'loss': 1.266601324081421, 'sparse_categorical_accuracy': 0.6040115356445312}
Round  40: {'loss': 1.0747275352478027, 'sparse_categorical_accuracy': 0.6678585410118103}
Training selesai.
Grafik disimpan sebagai fl_metrics.png
(venv) ezranahumury@DESKTOP-8003BIM:/mnt/c/KP/MATERI/4.3 Evaluasi Trade off$
```



Starting training with noise multiplier: 0.75

```python
#================================ Logging Akurasi ================================
data_frame = pd.DataFrame()
rounds = 40
clients_per_round = 40

# Hanya 1 noise multiplier: 0.76
noise_multiplier = 0.75
print(f'Starting training with noise multiplier: {noise_multiplier}')
data_frame = train(rounds, noise_multiplier, clients_per_round, data_frame)
print("Training selesai.")

```

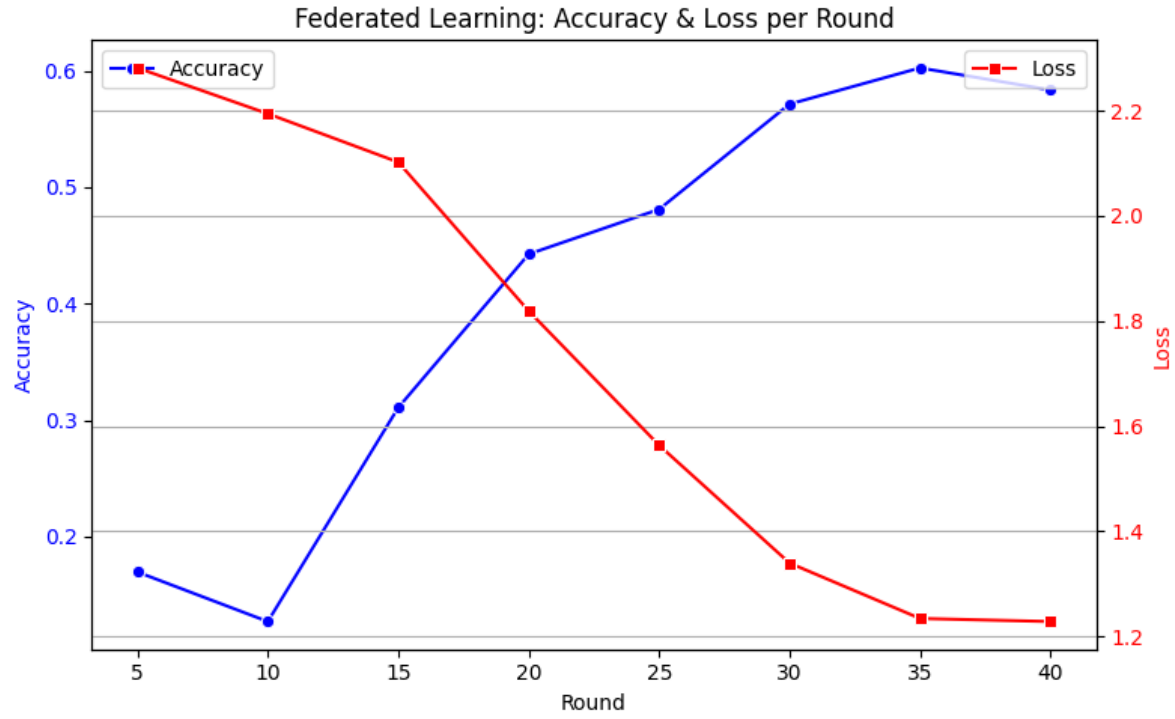
```
Skipping registering GPU devices...
Round   5: {'loss': 2.280210018157959, 'sparse_categorical_accuracy': 0.10834639519453049}
Round  10: {'loss': 2.1805832386016846, 'sparse_categorical_accuracy': 0.1296776980161667}
Round  15: {'loss': 1.9863053560256958, 'sparse_categorical_accuracy': 0.3591056168079376}
Round  20: {'loss': 1.7430737018585205, 'sparse_categorical_accuracy': 0.5344092845916748}
Round  25: {'loss': 1.4877409934997559, 'sparse_categorical_accuracy': 0.6074647307395935}
Round  30: {'loss': 1.2277956008911133, 'sparse_categorical_accuracy': 0.6761853694915771}
Round  35: {'loss': 1.2650632858276367, 'sparse_categorical_accuracy': 0.59112948179245}
Round  40: {'loss': 1.1116344928741455, 'sparse_categorical_accuracy': 0.6428781151771545}
Training selesai.
```



Starting training with noise multiplier: 1.0

```python
#================================ Logging Akurasi ================================
data_frame = pd.DataFrame()
rounds = 40
clients_per_round = 40

# Hanya 1 noise multiplier: 1.0
noise_multiplier = 1.0
print(f'Starting training with noise multiplier: {noise_multiplier}')
data_frame = train(rounds, noise_multiplier, clients_per_round, data_frame)
print("Training selesai.")
```

```
Skipping registering GPU devices...
Round   5: {'loss': 2.2818074226379395, 'sparse_categorical_accuracy': 0.1695239096879959}
Round  10: {'loss': 2.194612979888916, 'sparse_categorical_accuracy': 0.1269347518682479}
Round  15: {'loss': 2.102410316467285, 'sparse_categorical_accuracy': 0.31068769097328186}
Round  20: {'loss': 1.8190611600875854, 'sparse_categorical_accuracy': 0.44247159361839294}
Round  25: {'loss': 1.5633248090744019, 'sparse_categorical_accuracy': 0.4809707999229431}
Round  30: {'loss': 1.3390660285949707, 'sparse_categorical_accuracy': 0.5713165998458862}
Round  35: {'loss': 1.2335984706878662, 'sparse_categorical_accuracy': 0.6024196743965149}
Round  40: {'loss': 1.2277140617370605, 'sparse_categorical_accuracy': 0.5834884643554688}
Training selesai.
```



Biasanya ada pertukaran antara kualitas model dan privasi. Semakin tinggi noise yang kita gunakan, semakin banyak privasi yang bisa kita dapatkan untuk jumlah waktu pelatihan dan jumlah klien yang sama. Sebaliknya, dengan noise yang lebih sedikit, kita mungkin mendapatkan model yang lebih akurat, tetapi kita harus melatih dengan lebih banyak klien per putaran untuk mencapai tingkat privasi yang kita targetkan.

Sekarang kita dapat menggunakan fungsi dp_accounting untuk menentukan berapa banyak klien yang diharapkan per putaran yang kita butuhkan untuk mendapatkan privasi yang dapat diterima. Praktik standar adalah memilih delta yang agak lebih kecil dari satu dibagi jumlah rekaman dalam kumpulan data. Kumpulan data ini memiliki total 3383 pengguna pelatihan, jadi mari kita targetkan (2, 1e-5)-DP.

Kami menggunakan dp_accounting.calibrate_dp_mechanism untuk mencari jumlah klien per putaran. Akuntan privasi (RdpAccountant) yang kami gunakan untuk memperkirakan privasi yang diberikan dp_accounting.DpEvent didasarkan pada Wang et al. (2018) dan Mironov et al. (2019).

```python
# ============================== Konfigurasi Global & Kalibrasi DP ==============================
# Tentukan jumlah round training lebih dulu (dipakai saat komposisi DP).
rounds = 40

# Total klien dari dataset (umumnya =3383 untuk EMNIST only_digits)
total_clients = len(train_data.client_ids)
print(f"Total clients (detected): {total_clients}")

# Target privasi (samakan dengan snippet yang kamu minta)
noise_to_clients_ratio = 0.01    # rasio untuk menghitung noise_multiplier dari clients_per_round
target_delta = 1e-5
target_eps = 2.0

# No-arg callable that returns a fresh accountant.
make_fresh_accountant = dp_accounting.rdp.RdpAccountant

# Buat fungsi yang menerima clients_per_round dan membangun DpEvent komposisi penuh sepanjang 'rounds'
def make_event_from_param(clients_per_round: int):
    q = clients_per_round / total_clients
    noise_multiplier = clients_per_round * noise_to_clients_ratio
    gaussian_event = dp_accounting.GaussianDpEvent(noise_multiplier)
    sampled_event = dp_accounting.PoissonSampledDpEvent(q, gaussian_event)
    composed_event = dp_accounting.SelfComposedDpEvent(sampled_event, rounds)
    return composed_event

# Rentang pencarian untuk clients_per_round: [1, total_clients]
bracket_interval = dp_accounting.ExplicitBracketInterval(1, total_clients)

# Cari nilai clients_per_round terkecil yang mencapai (eps, delta) target.
clients_per_round = dp_accounting.calibrate_dp_mechanism(
    make_fresh_accountant,
    make_event_from_param,
    target_eps,
    target_delta,
    bracket_interval,
    discrete=True,
)

# Hitung noise multiplier dari rasio
noise_multiplier = clients_per_round * noise_to_clients_ratio
print(
    f"To get ({target_eps}, {target_delta})-DP, use {clients_per_round} clients "
    f"per round with noise multiplier {noise_multiplier:.6f}."
)
```

```
Total clients (detected): 3383
To get (2.0, 1e-05)-DP, use 100 clients per round with noise multiplier 1.000000.
Starting training with nm=1.000000, clients_per_round=100
```

```
Skipping registering GPU devices...
Round   5: {'loss': 2.2276828289031982, 'sparse_categorical_accuracy': 0.16651156544685364}
Round  10: {'loss': 2.1150102615356445, 'sparse_categorical_accuracy': 0.3380926847457886}
Round  15: {'loss': 1.9374396800994873, 'sparse_categorical_accuracy': 0.3779388666152954}
Round  20: {'loss': 1.7043087482452393, 'sparse_categorical_accuracy': 0.5826312899589539}
Round  25: {'loss': 1.468660831451416, 'sparse_categorical_accuracy': 0.6514988541603088}
```

Task : Plot Hasil Trade-off

```python
# ------------------------------------------------------------
# 6) Federated Training Loop (fungsi)
# ------------------------------------------------------------
def clip_by_l2_norm(tensors, clip):
    s = 0.0
    for t in tensors: s += np.sum(np.square(t))
    norm = float(np.sqrt(s)) + 1e-12
    if norm <= clip: return tensors, 1.0
    factor = clip / norm
    return [t * factor for t in tensors], factor

def federated_train(noise_multiplier, rounds=5):
    DP_CLIENT_LR = 0.05
    DP_L2_NORM_CLIP_CLIENT = 1.5
    SERVER_CLIP = 5.0
    SERVER_LR = 1.0

    tf.keras.utils.set_random_seed(42)
    np.random.seed(42)

    global_model = build_model(len(FEATURE_COLS))
    global_model.compile(optimizer="sgd", loss="binary_crossentropy", metrics=["accuracy"])
    global_weights = get_weights(global_model)

    acc_log, loss_log = [], []

    for round_idx in range(1, rounds+1):
        client_ds = make_client_datasets(local_epochs=LOCAL_EPOCHS, shuffle=True)
        clipped_weighted_sums = None
        total_weight = 0.0

        for k, ds in enumerate(client_ds):
            local_model = build_model(len(FEATURE_COLS))
            set_weights(local_model, global_weights)
            opt = DPKerasSGDOptimizer(
                l2_norm_clip=DP_L2_NORM_CLIP_CLIENT,
                noise_multiplier=noise_multiplier,
                num_microbatches=BATCH_SIZE,
                learning_rate=DP_CLIENT_LR,
                momentum=0.9
            )
            loss = tf.keras.losses.BinaryCrossentropy(from_logits=False, reduction=tf.losses.Reduction.NONE)
            local_model.compile(optimizer=opt, loss=loss, metrics=["accuracy"])
            local_model.fit(ds, epochs=1, verbose=0)

            w_local = local_model.get_weights()
            delta = [wl - wg for wl, wg in zip(w_local, global_weights)]
            delta_clipped, _ = clip_by_l2_norm(delta, SERVER_CLIP)

            weight_k = float(client_sizes[k])
            if clipped_weighted_sums is None:
                clipped_weighted_sums = [d * weight_k for d in delta_clipped]
            else:
                for i in range(len(delta_clipped)):
                    clipped_weighted_sums[i] += delta_clipped[i] * weight_k
            total_weight += weight_k

        avg_delta = [cw / (total_weight+1e-12) for cw in clipped_weighted_sums]
        avg_delta = [SERVER_LR * d for d in avg_delta]
        global_weights = [wg + d for wg, d in zip(global_weights, avg_delta)]
        set_weights(global_model, global_weights)

        gl_loss, gl_acc = evaluate_global(global_model)
        acc_log.append(gl_acc)
        loss_log.append(gl_loss)
        print(f"[nm={noise_multiplier:.3f}] Round {round_idx} | acc={gl_acc:.4f} | loss={gl_loss:.4f}")

    try:
        n = max(client_sizes)
        eps, _ = compute_dp_sgd_privacy_lib.compute_dp_sgd_privacy(
            n=n,
            batch_size=BATCH_SIZE,
            noise_multiplier=noise_multiplier,
            epochs=LOCAL_EPOCHS,
            delta=1e-5
        )
    except Exception:
        eps = np.nan

    return np.mean(acc_log), np.mean(loss_log), eps, acc_log, loss_log
```

```python

# ------------------------------------------------------------
# 7) Evaluasi Trade-off
# ------------------------------------------------------------
noise_list = [0.01, 0.05, 0.1, 0.5, 1.0]
results = []
rounds = 5

for nm in noise_list:
    acc, loss, eps, acc_log, loss_log = federated_train(noise_multiplier=nm, rounds=rounds)
    results.append({"NoiseMultiplier": nm, "Accuracy": acc, "Loss": loss, "Epsilon": eps})

df_results = pd.DataFrame(results)
print("\nHasil Evaluasi Trade-off:")
print(df_results)

# ------------------------------------------------------------
# 8) Plot Trade-off (Accuracy vs Epsilon)
# ------------------------------------------------------------
plt.figure(figsize=(7,5))
sns.lineplot(data=df_results, x="Epsilon", y="Accuracy", marker="o")
for i, row in df_results.iterrows():
    plt.text(row["Epsilon"], row["Accuracy"], f"nm={row['NoiseMultiplier']}", fontsize=9)

plt.xlabel("Epsilon (ε) → Privasi ↓")
plt.ylabel("Accuracy → Utility ↑")
plt.title("Trade-off Federated Learning (DP-SGD)")
plt.grid(True)
plt.tight_layout()
plt.savefig("tradeoff_fl_accuracy_epsilon.png", dpi=150)
print("Grafik trade-off disimpan sebagai tradeoff_fl_accuracy_epsilon.png")

```

```
TensorFlow: 2.14.1
TF-Privacy: OK (DP-SGD tersedia)
[nm=0.010] Round 1 | acc=0.7449 | loss=0.5289
[nm=0.010] Round 2 | acc=0.7449 | loss=0.4330
[nm=0.010] Round 3 | acc=0.7984 | loss=0.3482
[nm=0.010] Round 4 | acc=0.8860 | loss=0.2732
[nm=0.010] Round 5 | acc=0.9276 | loss=0.2166
WARNING:absl:`compute_dp_sgd_privacy` is deprecate
practice. Please use `compute_dp_sgd_privacy_state
te_dp_sgd_privacy_statement`, call the `dp_account
[nm=0.050] Round 1 | acc=0.7449 | loss=0.5292
[nm=0.050] Round 2 | acc=0.7449 | loss=0.4334
[nm=0.050] Round 3 | acc=0.7969 | loss=0.3503
[nm=0.050] Round 4 | acc=0.8833 | loss=0.2743
[nm=0.050] Round 5 | acc=0.9276 | loss=0.2164
WARNING:absl:`compute_dp_sgd_privacy` is deprecate
practice. Please use `compute_dp_sgd_privacy_state
te_dp_sgd_privacy_statement`, call the `dp_account
[nm=0.100] Round 1 | acc=0.7449 | loss=0.5295
[nm=0.100] Round 2 | acc=0.7449 | loss=0.4339
[nm=0.100] Round 3 | acc=0.7940 | loss=0.3533
[nm=0.100] Round 4 | acc=0.8800 | loss=0.2767
[nm=0.100] Round 5 | acc=0.9280 | loss=0.2169
WARNING:absl:`compute_dp_sgd_privacy` is deprecate
practice. Please use `compute_dp_sgd_privacy_state
te_dp_sgd_privacy_statement`, call the `dp_account
[nm=0.500] Round 1 | acc=0.7449 | loss=0.5338
[nm=0.500] Round 2 | acc=0.7449 | loss=0.4375
[nm=0.500] Round 3 | acc=0.7769 | loss=0.3766
[nm=0.500] Round 4 | acc=0.8480 | loss=0.3035
[nm=0.500] Round 5 | acc=0.9187 | loss=0.2340
WARNING:absl:`compute_dp_sgd_privacy` is deprecate
practice. Please use `compute_dp_sgd_privacy_state
te_dp_sgd_privacy_statement`, call the `dp_account
[nm=1.000] Round 1 | acc=0.7449 | loss=0.5381
[nm=1.000] Round 2 | acc=0.7449 | loss=0.4515
[nm=1.000] Round 3 | acc=0.7593 | loss=0.4160
[nm=1.000] Round 4 | acc=0.8276 | loss=0.3590
[nm=1.000] Round 5 | acc=0.8927 | loss=0.2720
WARNING:absl:`compute_dp_sgd_privacy` is deprecate
```

```
Hasil Evaluasi Trade-off:
   NoiseMultiplier  Accuracy      Loss        Epsilon
0             0.01  0.820356  0.359983  149665.029213
1             0.05  0.819511  0.360726    5665.029213
2             0.10  0.818356  0.362040    1165.029981
3             0.50  0.806667  0.377074      12.198622
4             1.00  0.793867  0.407317       2.311752
Grafik trade-off disimpan sebagai tradeoff_fl_accuracy_epsilon.png
```