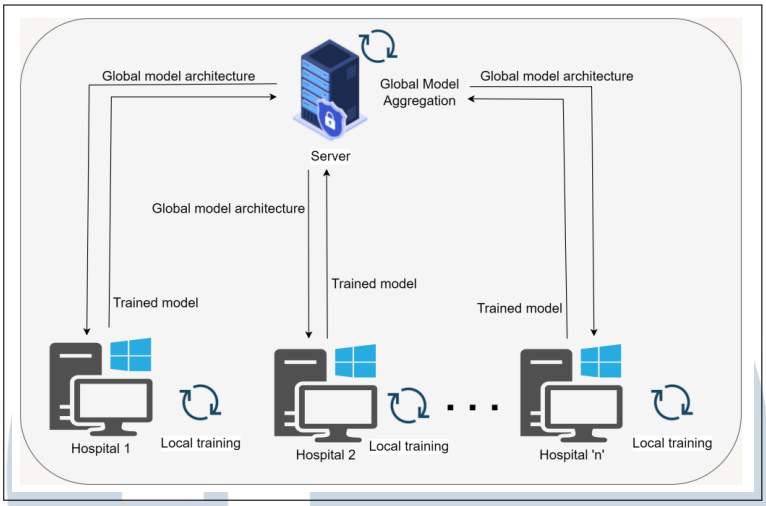
BAB 2
LANDASAN TEORI

2.1 Federated Learning

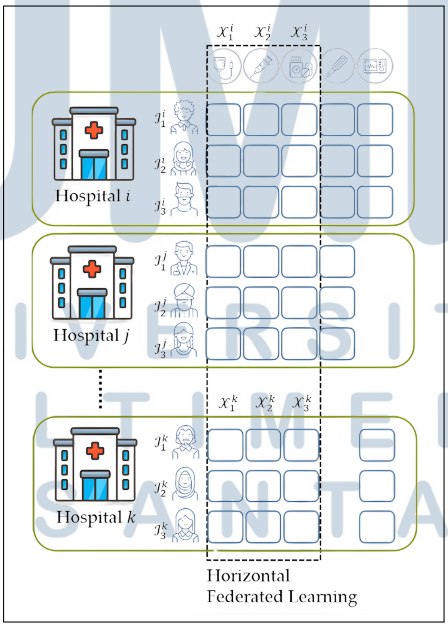
Federated Learning (FL) merupakan paradigma *machine learning* terdistribusi yang memungkinkan banyak klien melakukan proses pelatihan model secara lokal pada perangkat atau institusi masing-masing tanpa perlu memindahkan data mentah ke server pusat [5, 6, 7, 15, 16, 17]. Pendekatan ini muncul sebagai solusi terhadap keterbatasan pembelajaran terpusat yang sering kali menimbulkan risiko kebocoran privasi serta permasalahan kepemilikan data [15, 16, 18]. Melalui FL, setiap klien melatih model lokal menggunakan dataset internal, kemudian hanya parameter atau gradien hasil pelatihan yang dikirimkan ke server untuk dilakukan agregasi [15, 19, 20]. Server kemudian memperbarui model global berdasarkan kontribusi masing-masing klien sehingga dihasilkan model yang lebih representatif tanpa mengorbankan kerahasiaan data lokal [16, 17, 20].

Struktur umum *Federated Learning* (FL) terdiri atas satu server pusat dan sejumlah klien yang melatih model secara lokal menggunakan data masing-masing tanpa perlu membagikan data mentah [17, 20]. Server mengirimkan parameter model global awal kepada klien untuk dilatih secara paralel di perangkat lokal mereka. Setelah proses pelatihan selesai, setiap klien mengirimkan hasil pembaruan modelnya kembali ke server. Server kemudian menggabungkan seluruh pembaruan tersebut menggunakan mekanisme agregasi berbobot untuk memperbarui model global. Proses ini berlangsung secara iteratif hingga model mencapai konvergensi dan performa yang stabil [16, 17, 20]. Gambar 2.1 memperlihatkan alur kerja umum FL yang mencakup seleksi klien, pelatihan lokal, dan agregasi model global.



Gambar 2.1. Alur kerja umum *Federated Learning* [21].

Horizontal Federated Learning (HFL) merupakan bentuk kolaborasi terdistribusi di mana beberapa institusi dengan struktur fitur data yang serupa melatih model secara bersama tanpa perlu berbagi data mentah [17, 18]. Melalui mekanisme ini, HFL memungkinkan pembelajaran kolektif yang menjaga privasi data sekaligus meningkatkan generalisasi model terhadap variasi populasi pasien [22]. Gambar 2.2 memperlihatkan ilustrasi HFL pada skenario rumah sakit, di mana masing-masing institusi memiliki fitur data yang identik namun pasien yang berbeda.



Gambar 2.2. Arsitektur *Horizontal Federated Learning* [22].

Algoritma *Federated Averaging* (FedAvg) yang diperkenalkan oleh McMahan *et al.* [19] merupakan metode agregasi paling umum dalam *Federated Learning*. Algoritma ini menggabungkan model lokal dari tiap klien melalui rata-rata berbobot untuk membentuk model global baru yang kemudian didistribusikan kembali ke seluruh klien. Proses dasarnya ditunjukkan pada *pseudocode* berikut.

Algorithm 1 Federated Averaging (FedAvg) [19]

Require: Number of clients K, local batch size B, local epochs E, learning rate η
Ensure: Global weights w

Initialize global model w_0
for $t = 1, 2, \ldots, T$ **do**
    for all $k = 1, 2, \ldots, K$ **(in parallel) do**
        $w_{t+1}^k \leftarrow \text{CLIENTUPDATE}(k, w_t)$
    end for
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{\sum_{j=1}^{K} n_j} w_{t+1}^k$
end for
function CLIENTUPDATE(k, w)
    Split local dataset P_k into batches of size B
    for $i = 1$ **to** E **do**
        for each batch $b \in P_k$ **do**
            $w \leftarrow w - \eta \nabla \mathscr{L}(w; b)$
        end for
    end for
    return w
end function

Pada Algoritma 1, w_t menyatakan parameter model global pada ronde ke-t, sedangkan w_{t+1}^k adalah parameter model hasil pembaruan lokal pada klien ke-k yang dilakukan dengan inisialisasi w_t. Data lokal pada klien ke-k dinotasikan sebagai P_k dan diproses dalam *batch* $b \in P_k$. Pembaruan parameter lokal pada setiap *batch* dilakukan dengan aturan penurunan gradien, yakni parameter w diperbarui ke arah negatif gradien fungsi kerugian $\mathscr{L}(w; b)$ dengan besar langkah yang ditentukan oleh η, sehingga diperoleh pembaruan $w \leftarrow w - \eta \nabla \mathscr{L}(w; b)$. Setelah seluruh klien menyelesaikan pelatihan lokal, server memperbarui model global melalui agregasi rata-rata berbobot $w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{\sum_{j=1}^{K} n_j} w_{t+1}^k$, dengan n_k menyatakan jumlah sampel pada klien ke-k sehingga kontribusi masing-masing klien proporsional terhadap

ukuran data lokalnya. Proses ini diulang selama T ronde komunikasi hingga diperoleh model global akhir.

Jiménez-Sánchez *et al.* (2023) [8] mengembangkan pendekatan *Federated Learning* dengan memperluas algoritma FedAvg melalui tiga mekanisme utama, yaitu penambahan *noise*, penjadwalan *curriculum learning* yang adaptif, serta penyelarasan fitur lintas domain menggunakan *adversarial alignment*. Model diinisialisasi menggunakan bobot awal hasil *pretrained* dari Wu *et al.* (2020) [23] untuk mempercepat konvergensi dan menjaga stabilitas awal pelatihan federatif.

Algoritma FedAvg konvensional menghasilkan AUC sebesar 0.75 dan PR-AUC sebesar 0.77. Penambahan *noise* diterapkan selama proses agregasi parameter untuk menyeimbangkan kontribusi antar klien dan menjaga kestabilan pembaruan model global pada data lintas domain. Mekanisme ini berperan sebagai strategi regularisasi ringan yang mencegah dominasi model dari domain dengan distribusi data yang lebih besar serta meningkatkan kemampuan generalisasi model federatif [8]. Proses agregasi berbobot dengan penambahan *noise* ditunjukkan pada cuplikan berikut, yang sekaligus memperlihatkan penggunaan parameter `pace` dan `nsteps` dalam penjadwalan komunikasi federatif.

```
if (count % params.pace == 0) or t == params.nsteps - 1:
    # communication - weights update
    with torch.no_grad():
        for key in global_model.state_dict().keys():
            # num_batches_tracked is a non trainable LongTensor and
            # num_batches_tracked are the same for all clients
            if local_models[0].state_dict()[key].dtype == torch.int64:
                global_model.state_dict()[key].data.copy_(local_models[0].state_dict()[key])
            else:
                temp = torch.zeros_like(global_model.state_dict()[key])
                # add noise
                for s in range(n_sites):
                    if params.noise_type == 'G':
                        nn = tdist.Normal(
                            torch.tensor([0.0]),
                            params.noise * torch.std(
                                local_models[s].state_dict()[key].detach().cpu()
                            )
                        )
                    else:
                        nn = tdist.Laplace(
                            torch.tensor([0.0]),
                            params.noise * torch.std(
                                local_models[s].state_dict()[key].detach().cpu()
```

```
                                )
                            )
                        noise = nn.sample(local_models[s].state_dict()
    [key].size()).squeeze(-1)
                        noise = noise.to(device)
                        temp += w[s] * (local_models[s].state_dict()[
    key] + noise)
                # update global model
                global_model.state_dict()[key].data.copy_(temp)
                # update local models
                for s in range(n_sites):
                    local_models[s].state_dict()[key].data.copy_(
    global_model.state_dict()[key])
```

Kode 2.1: Proses agregasi model global dengan regularisasi berbasis *noise* dan penjadwalan komunikasi [8]

Pada Kode 2.1, variabel `count` merupakan penghitung langkah pelatihan lokal dalam satu epoch, sedangkan `params.nsteps` menentukan jumlah total langkah (*mini-steps*) yang dijalankan setiap epoch pada masing-masing situs. Kondisi `(count % params.pace == 0) or t == params.nsteps - 1` memastikan bahwa agregasi global hanya dilakukan secara periodik setiap sejumlah langkah tertentu yang diatur oleh `pace`, atau dipaksa terjadi pada akhir epoch ketika `t = params.nsteps - 1`. Dengan demikian, `pace` mengendalikan frekuensi komunikasi federatif, sementara `nsteps` mengatur granularitas pelatihan lokal dalam satu epoch. Jiménez-Sánchez *et al.* [8] menunjukkan bahwa pengaturan ritme komunikasi ini penting untuk menjaga keseimbangan antara stabilitas konvergensi dan efisiensi pelatihan, terutama ketika digabung dengan mekanisme *adversarial alignment* dan *curriculum learning*.

2.1.1 Adversarial Alignment

Mekanisme *adversarial alignment* digunakan untuk menyelaraskan distribusi representasi fitur antar domain dengan prinsip pembelajaran *adversarial*. Arsitektur terdiri atas dua komponen utama, yaitu *encoder* untuk mengekstraksi fitur dan *discriminator* untuk membedakan asal domain representasi tersebut. Proses pelatihan berlangsung secara kompetitif agar *encoder* menghasilkan fitur yang tidak dapat dibedakan oleh *discriminator*, sehingga tercipta representasi yang bersifat *domain-invariant* [8]. Fungsi kerugian yang digunakan dijelaskan sebagai berikut.

```
def advDloss(d1, d2):
    res = -torch.log(d1).mean() - torch.log(1 - d2).mean()
    return res

```

```
def advGloss(d1, d2):
    res = -torch.log(d1).mean() - torch.log(d2).mean()
    return res.mean()
```

Kode 2.2: Fungsi kerugian pada mekanisme *adversarial alignment* [8]

Proses pelatihan berlangsung dengan membandingkan representasi antar domain untuk memperkecil perbedaan distribusi fitur. Pembaruan bobot dilakukan secara bergantian antara *encoder* dan *discriminator* agar keseimbangan pembelajaran terjaga. Cuplikan berikut menggambarkan bagian utama dari proses pelatihan *adversarial* sekaligus menunjukkan bagaimana parameter `nsteps` digunakan untuk mengulang proses optimisasi klasifikasi dan penyelarasan fitur pada setiap epoch.

```
for t in range(params.nsteps):
    # feature space
    fs = []

    # optimize classifier
    for i in range(n_sites):
        optimizers[i].zero_grad()
        inputs, labels, domain, idx = next(data_inters[i])  # get mini-batch for site i
        num_data[i] += labels.size(0)
        inputs = inputs.to(device)
        labels = labels.to(device)
        probs, logits = local_models[i](inputs)  # get output of model i
        loss = class_criterion(logits, labels)  # compute loss

        loss_all[i] += loss.item() * labels.size(0)
        loss.backward(retain_graph=True)
        optimizers[i].step()

        fs.append(local_models[i].encoder(inputs))  # feature space

    # optimize alignment
    nn = []
    noises = []
    for i in range(n_sites):
        nn = tdist.Normal(torch.tensor([0.0]), 0.001 * torch.std(fs[i].detach().cpu()))
        noises.append(nn.sample(fs[i].size()).squeeze().to(device))

    for i in range(n_sites):
        for j in range(n_sites):
            if i != j:
                optimizerDs[i].zero_grad()
                optimizerGs[i].zero_grad()
                optimizerGs[j].zero_grad()

                d1 = discriminators[i](fs[i].detach() + noises[i])
                d2 = discriminators[i](fs[j].detach() + noises[j])
```

```
                num_dataG[i] += d1.size(0)
                num_dataD[i] += d1.size(0)
                lossD = advDloss(d1, d2)
                lossG = advGloss(d1, d2)

                lossD_all[i] += lossD.item() * d1.size(0)
                lossG_all[i] += lossG.item() * d1.size(0)
                lossG_all[j] += lossG.item() * d2.size(0)
                lossD = 0.1 * lossD

                if epoch >= params.n_epochs_adversarial:
                    lossG.backward(retain_graph=True)
                    optimizerGs[i].step()
                    optimizerGs[j].step()

                    lossD.backward(retain_graph=True)  # retain_graph=True works too
                    optimizerDs[i].step()

    count += 1
```

Kode 2.3: Loop pelatihan lokal dengan nsteps dan *adversarial alignment* [8]

Pada Kode 2.3, parameter `params.nsteps` menentukan berapa kali proses pelatihan lokal diulang dalam satu *epoch* pada setiap situs. Setiap iterasi langkah t terdiri atas dua bagian utama: (1) optimisasi *classifier* berbasis *loss* klasifikasi untuk memperbaiki kemampuan prediksi lokal, dan (2) proses *adversarial alignment* yang menambahkan *noise* kecil pada representasi fitur (`fs`) dan melatih pasangan *encoder–discriminator* agar distribusi fitur antar domain menjadi lebih selaras. Dalam konteks ini, `nsteps` berperan sebagai pengatur granularitas pembaruan model lokal: semakin besar `nsteps`, semakin banyak *mini-batch* yang diproses dan semakin halus perubahan parameter sebelum dilakukan agregasi global yang dijadwalkan oleh `pace` [8].

Integrasi *adversarial alignment* menghasilkan AUC sebesar 0.78 dan PR-AUC sebesar 0.80, atau peningkatan sekitar 3% dibandingkan FedAvg. Hasil ini menunjukkan bahwa penyelarasan fitur antar domain efektif dalam mengurangi perbedaan distribusi representasi dan memperkuat generalisasi model global [8].

2.1.2 Curriculum Learning

Mekanisme *curriculum learning* yang diterapkan bersifat *memory-aware*, mengatur prioritas pelatihan berdasarkan perubahan prediksi antar ronde. Sampel yang kembali salah setelah pembaruan model global dilatih terlebih dahulu pada iterasi berikutnya. Strategi ini menjaga konsistensi antara pembelajaran lokal dan global serta mencegah model melupakan informasi penting pada domain tertentu

[8]. Pembobotan dihitung berdasarkan perbandingan hasil prediksi antar ronde sebagaimana diformulasikan berikut.

```
def get_curriculum_weights(preds_back, preds_recent):
    comparison = preds_back > preds_recent
    weights = comparison.astype(np.float) + 1.0
    return weights
```

Kode 2.4: Perhitungan bobot kurikulum lokal [8]

Proses pembentukan kurikulum dilakukan setelah fase *adversarial* selesai. Setiap situs menghitung bobot kurikulum berdasarkan perubahan prediksi antara *epoch* sebelumnya dan saat ini. Bobot ini digunakan sebagai probabilitas sampling untuk menyusun ulang data pelatihan menggunakan *Weighted Random Sampling*, sebagaimana ditunjukkan pada cuplikan berikut.

```
if epoch > params.n_epochs_adversarial:
    curriculum_weights = []
    for i in range(n_sites):
        weights = get_curriculum_weights(track_preds[epoch - 1][i
    ],
                                         track_preds[epoch][i])
        curriculum_weights.append(weights)

    train_sampler0 = WeightedRandomSampler(weights=
    curriculum_weights[0],
                                           num_samples=len(
    trainset0))
    train_loader0 = DataLoader(trainset0,
                               batch_size=len(trainset0) // params
    .nsteps,
                               shuffle=False,
                               num_workers=params.num_workers,
                               sampler=train_sampler0)

    train_sampler1 = WeightedRandomSampler(weights=
    curriculum_weights[1],
                                           num_samples=len(
    trainset1))
    train_loader1 = DataLoader(trainset1,
                               batch_size=len(trainset1) // params
    .nsteps,
                               shuffle=False,
                               num_workers=params.num_workers,
                               sampler=train_sampler1)

    train_sampler2 = WeightedRandomSampler(weights=
    curriculum_weights[2],
                                           num_samples=len(
    trainset2))
    train_loader2 = DataLoader(trainset2,
                               batch_size=len(trainset2) // params
    .nsteps,
                               shuffle=False,
                               num_workers=params.num_workers,
                               sampler=train_sampler2)
```

```
    data_inters = [iter(train_loader0),
                   iter(train_loader1),
                   iter(train_loader2)]
```

Kode 2.5: Implementasi pembaruan bobot kurikulum pada tiap situs [8]

Pada Kode 2.5, *curriculum learning* diintegrasikan dengan skema *batching* yang sama seperti pada loop utama, yaitu menggunakan `batch_size = len(trainset0)//params.nsteps`. Hal ini memastikan bahwa urutan sampel yang diprioritaskan oleh kurikulum tetap konsisten dengan ritme pelatihan lokal yang diatur oleh `nsteps`. Hasil eksperimen menunjukkan bahwa penerapan *curriculum learning* menghasilkan AUC sebesar 0.75 dan PR-AUC sebesar 0.78, yang merepresentasikan peningkatan sekitar 1% dibandingkan FedAvg standar [8]. Peningkatan ini menandakan bahwa pengaturan urutan pelatihan berdasarkan tingkat kesulitan data berkontribusi pada stabilitas proses pembelajaran federatif.

Kombinasi kedua mekanisme, yaitu *alignment* dan *curriculum learning* (FedAlign-CL), menghasilkan performa terbaik dengan AUC sebesar 0.79 dan PR-AUC sebesar 0.82, atau peningkatan masing-masing sekitar 4% dan 5% dibandingkan FedAvg dasar. Integrasi *Curriculum Learning* dan *Adversarial Alignment* yang dijalankan dalam loop pelatihan berbasis `nsteps` serta disinkronkan secara periodik menggunakan `pace` meningkatkan efektivitas pembelajaran federatif serta memperkuat stabilitas dan kemampuan generalisasi model global pada data medis multisentra [8]. Tabel 2.1 merangkum penelitian terkait penggunaan *Federated Learning*.

Tabel 2.1. Ringkasan penelitian Federated Learning

Peneliti	**Metode**	**Hasil**	**Judul**
Jiménez-Sánchez *et al.* (2023) [8]	FL-Align-CL	AUC 79%, PR-AUC 82%	*Memory-aware curriculum federated learning for breast cancer classification*
Nan Yan *et al.* (2024) [14]	FL + HE	Acc 71,37%	*Efficient and straggler-resistant homomorphic encryption for heterogeneous federated learning*
Lanjut pada halaman berikutnya			

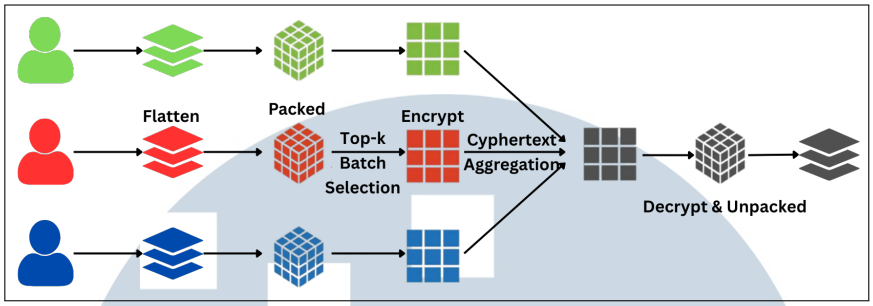
Tabel 2.1 Ringkasan penelitian Federated Learning (lanjutan)

Peneliti	Metode	Hasil	Judul
Shukla *et al.* (2025) [24]	FL + DP	Acc 96,1%	*Federated learning with differential privacy for breast cancer diagnosis enabling secure data sharing and model integrity*
Adnan *et al.* (2022) [25]	FL + DP	AUC $\approx$ 0,9	*Federated learning and differential privacy for medical image analysis*
Lessage *et al.* (2024) [26]	FL + HE	Acc 73,2%	*Secure federated learning applied to medical imaging with fully homomorphic encryption*
Zhang *et al.* (2020) [27]	FL + HE	Acc 74,04%	*BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning*

2.2 Packed CKKS Homomorphic Encryption

Skema *Packed CKKS Homomorphic Encryption* merupakan pengembangan dari CKKS (Cheon–Kim–Kim–Song) yang memungkinkan operasi aritmetika dilakukan langsung di atas data terenkripsi tanpa proses dekripsi. Mekanisme ini mendukung representasi bilangan riil dan komputasi paralel, menjadikannya sangat relevan untuk sistem pembelajaran terdistribusi yang membutuhkan privasi tinggi seperti *Federated Learning* [11, 14]. Prinsip utama *Packed CKKS* adalah konsep *packing*, yaitu pengemasan sejumlah besar nilai *plaintext* ke dalam satu *ciphertext* agar dapat diproses secara serentak. Dengan demikian, efisiensi komputasi meningkat tanpa mengurangi keamanan data.

Gambar 2.3 memperlihatkan alur umum *Packed CKKS* pada skenario *Federated Learning*. Setiap klien melakukan pelatihan lokal dan menghasilkan pembaruan parameter, kemudian parameter tersebut dikemas (*packing*) ke dalam slot-slot CKKS dan dienkripsi menjadi *ciphertext*. Server hanya menerima *ciphertext* dari seluruh klien dan melakukan agregasi secara homomorfik untuk membentuk *ciphertext* global tanpa mengakses nilai *plaintext*. Hasil agregasi selanjutnya dikirim kembali ke klien untuk didekripsi, sehingga klien memperoleh parameter global terbaru untuk ronde berikutnya [14].



Gambar 2.3. Ilustrasi *Packed CKKS* [14].

Inisialisasi konteks dan kunci CKKS dilakukan satu kali pada tahap awal sistem. Implementasi Nan Yan *et al.* [14] membangkitkan konteks CKKS dengan menetapkan parameter kriptografi dan menghasilkan pasangan kunci publik serta kunci privat. Proses serialisasi menyimpan parameter konteks (dan kunci sesuai kebutuhan implementasi) ke dalam *file*, sehingga konteks yang sama dapat dipakai kembali selama seluruh ronde pelatihan [14].

Kunci publik pada konteks digunakan untuk melakukan enkripsi parameter model di sisi klien. Kunci privat digunakan di sisi klien untuk melakukan dekripsi hasil agregasi global. Server hanya memproses *ciphertext* melalui operasi homomorfik dan tidak memiliki akses ke kunci privat, sehingga isi model tidak dapat dibaca dalam bentuk *plaintext*. Pola ini konsisten dengan skenario FedPHE yang mengasumsikan distribusi kunci dilakukan melalui kanal aman dan server tidak berkolusi dengan klien [14]. Kode 2.6 disajikan sebagai kutipan langsung dari implementasi Nan Yan *et al.* [14] untuk menggambarkan proses inisialisasi konteks CKKS.

```
import os, tenseal as ts

def ckks_init(data_dir):
    ckks_ctx = ts.context(ts.SCHEME_TYPE.CKKS, poly_modulus_degree
   =8192, coeff_mod_bit_sizes=[60, 40, 40, 60])
    ckks_ctx.global_scale=2**40
    ckks_ctx.generate_galois_keys()
    params = ckks_ctx.serialize(save_secret_key=True)
    ckks_file = os.path.join(data_dir + 'context_params')
    with open(ckks_file, "wb") as f:
        f.write(params)
```

Kode 2.6: Inisialisasi konteks CKKS [14]

Berdasarkan implementasi yang dikemukakan oleh Nan Yan *et al.* [14], mekanisme umum pada Gambar 2.3 dirumuskan kembali dalam bentuk *pseudocode* pada Algoritma 2. Algoritma ini menjelaskan proses kerja inti Packed CKKS di sisi klien, meliputi pelatihan lokal, pengemasan *batch*, seleksi parameter signifikan, dan

enkripsi ke domain homomorfik.

Algorithm 2 Packed CKKS Client Process [14]

Require: Model lokal w_i, jumlah epoch lokal E, rasio sparsifikasi $s\%$, ukuran batch B, konteks enkripsi ctx

Ensure: Ciphertext terenkripsi C_i^t dan mask M_i^t

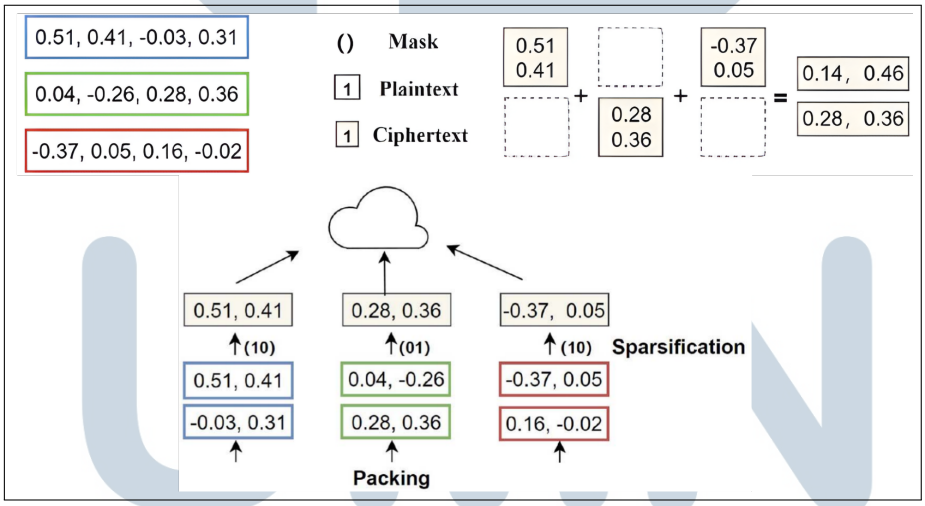
for setiap *global round* $t = 0, 1, \ldots, T-1$ **do**
  // Pelatihan lokal
  // Packed CKKS dan sparsifikasi *top-k*
  $P_i \leftarrow$ *Flatten and pack* w_i^t menjadi batch $\{P_i^1, \ldots, P_i^K\}$
  $\theta \leftarrow$ ambang batas $s\%$ tertinggi dalam P_i
  for setiap batch $P_i[l] \in P_i$ **do**
    if $|P_i[l]| \geq \theta$ **then**
      $M_i^t[l] \leftarrow 1$
      $C_i^t[l] \leftarrow E(P_i[l])$
    else
      $M_i^t[l] \leftarrow 0$
    end if
  end for
  Kirim pasangan (C_i^t, M_i^t) ke server
  Terima hasil agregasi global (C^t, M^t) dari server
  Dekripsi C^t menggunakan kunci privat dan *mask* M^t
  $\tilde{w}_{t+1} \leftarrow$ hasil dekripsi ciphertext global
  Unpack $\tilde{w}_{t+1}$ ke dalam struktur model: $w_{t+1}^i \leftarrow$ params_tomodel$(\tilde{w}_{t+1})$
end for

Indeks t menyatakan *global round* ke-t. Indeks i menyatakan klien ke-i. Setiap ronde global menghasilkan parameter model lokal terbaru w_i^t melalui pelatihan lokal. Parameter w_i^t diubah menjadi vektor satu dimensi melalui proses *flattening* dan dipadatkan ke dalam slot-slot CKKS (*packing*) untuk membentuk representasi terkemas P_i. Mengingat kapasitas slot CKKS terbatas, P_i dipartisi menjadi beberapa *batch* $\{P_i^1, \ldots, P_i^K\}$ agar setiap *batch* dapat diproses dan dienkripsi secara efisien. Tahap ini memungkinkan banyak elemen parameter dimuat dalam satu *ciphertext* sesuai karakteristik skema *Packed* CKKS.

Sparsifikasi *top-k* diterapkan dengan menentukan ambang θ yang merepresentasikan $s\%$ nilai tertinggi berdasarkan magnitudo pada P_i. Setiap *batch* $P_i[l]$ yang memenuhi kondisi $|P_i[l]| \geq \theta$ dipertahankan, diberi *mask* biner $M_i^t[l] =$

1, dan dienkripsi menggunakan operator enkripsi $E(\cdot)$ sehingga menghasilkan *ciphertext* $C_i^t[l] = E(P_i[l])$. *Batch* yang tidak memenuhi ambang diabaikan dengan $M_i^t[l] = 0$. Pasangan (C_i^t, M_i^t) dikirim ke server untuk dilakukan agregasi dalam domain *ciphertext*. Hasil agregasi global (C^t, M^t) diterima klien pada akhir ronde. Klien mendekripsi C^t menggunakan kunci privat dan menerapkan *mask* M^t untuk merekonstruksi vektor parameter global terdekripsi $\tilde{w}_{t+1}$. Vektor $\tilde{w}_{t+1}$ di-*unpack* dan dipetakan kembali ke struktur parameter model melalui fungsi `params_tomodel` sehingga diperoleh parameter klien pada ronde berikutnya, yaitu w_{t+1}^i.

Gambar 2.4 menunjukkan ilustrasi mekanisme *top-k sparsification* pada skema *Packed CKKS*. Pembaruan parameter yang telah dikemas dipilih berdasarkan nilai magnitudo tertinggi, kemudian ditandai menggunakan *mask* biner. Hanya *batch* terpilih yang dienkripsi dan dikirimkan ke server, sedangkan elemen yang tidak terpilih tidak dikirim sehingga volume komunikasi dapat ditekan tanpa mengubah alur agregasi terenkripsi [14].



Gambar 2.4. Ilustrasi *top-k sparsification* dengan *mask* pada proses *packing* dan agregasi terenkripsi [14].

Konteks ini menjadi landasan utama bagi seluruh proses enkripsi dan dekripsi dalam skema *Packed* CKKS. Parameter model diubah dari struktur tensor multidimensi menjadi representasi vektor satu dimensi agar dapat dikemas dalam *batch*. Transformasi ini dilakukan melalui fungsi `params_tolist()` berikut.

```
import numpy as np, torch

def params_tolist(model):
    model.to('cpu')
```

```
    local_state = model.state_dict()
    params_list, layer_shape, params_num = [], {}, {}
    for key in local_state.keys():
        layer_shape[key] = local_state[key].shape
        params_num[key] = int(np.prod(local_state[key].shape))
        layer_values = local_state[key].reshape(params_num[key]).
    tolist()
        params_list.extend(layer_values)
    return params_list, params_num, layer_shape
```

Kode 2.7: Konversi parameter model ke vektor satu dimensi [14]

Fungsi tersebut berperan sebagai tahap *pra-packing* (*flattening*) yang menyusun seluruh parameter model dalam satu daftar panjang (*flat list*). Setiap elemen parameter disusun secara berurutan agar dapat dibagi menjadi *batch* tetap pada tahap enkripsi berikutnya.

Proses *packing* aktual dilakukan bersamaan dengan enkripsi pada mekanisme CKKS. Setiap *batch* parameter berukuran `enc_batch_size` dikemas menjadi satu *ciphertext*. Penelitian Nan Yan *et al.* [14] menerapkan teknik *top-k sparsification* untuk mengoptimalkan efisiensi komunikasi dengan hanya mengenkripsi sebagian *batch* dengan magnitudo tertinggi.

Implementasi kombinasi *packing* dan *top-k sparsification* menggunakan pustaka `TenSEAL` ditunjukkan pada potongan kode berikut. Proses ini membagi parameter menjadi beberapa *batch* berukuran tetap, menghitung rata-rata absolut setiap *batch*, lalu mengenkripsi *batch* dengan skor tertinggi menjadi *ciphertext* CKKS.

```
import tenseal as ts, numpy as np

def ckks_enc(plain_list, ckks_ctx, isBatch, batch_size, topk_ratio
    , is_spars='topk'):
    batch_num = int(np.ceil(len(plain_list) / batch_size))
    if isBatch:
        if len(plain_list) % batch_num != 0:
            pad = batch_num * batch_size - len(plain_list)
            plain_list.extend([0] * pad)
        if is_spars == 'topk':
            plain_batches = [plain_list[i * batch_size : (i + 1) *
     batch_size] for i in range(batch_num)]

            avg_list = [np.average(np.abs(batch)) for batch in
    plain_batches]
            k = int(np.ceil(batch_num * topk_ratio))
            top_idx = np.argsort(avg_list)[-k:]
            mask = [1 if i in top_idx else 0 for i in range(
    batch_num)]

            cipher_list = []
            for i in top_idx:
                cipher = ts.ckks_vector(ckks_ctx, plain_batches[i
    ])
```

```
                cipher_list.append(cipher.serialize())
            return cipher_list, mask

    else:
        cipher = [ts.ckks_vector(ckks_ctx, [i]).serialize() for i
    in plain_list]
        return cipher
```

Kode 2.8: Proses *packing* dan enkripsi menggunakan skema *Packed* CKKS [14]

Fungsi `enc_params()` digunakan untuk menjalankan proses *packing* sekaligus enkripsi parameter lokal di sisi klien. Fungsi ini membaca konteks enkripsi publik (*public context*) yang berisi parameter CKKS, kemudian memanggil `ckks_enc()` dengan parameter sistem yang telah ditetapkan, seperti ukuran *batch* (`enc_batch_size`) dan rasio *top-k sparsification*.

```
import os, tenseal as ts

def enc_params(params_list, args, epoch=0):
    with open(os.path.join(args.data_dir, "context_params"), "rb")
     as f:
        ckks_ctx = ts.context_from(f.read())
    return ckks_enc(params_list, ckks_ctx, isBatch=args.isBatch,
                    batch_size=args.enc_batch_size,
                    topk_ratio=args.topk, is_spars=args.isSpars)
```

Kode 2.9: Pemanggilan fungsi enkripsi di sisi klien [14]

Proses dekripsi dijalankan di sisi klien menggunakan kunci privat yang tidak pernah dibagikan. Setiap *ciphertext* didekripsi menjadi vektor numerik dan dinormalisasi sesuai jumlah *batch* aktif. Mekanisme tersebut diimplementasikan menggunakan fungsi `ckks_dec()` berikut.

```
import tenseal as ts
import numpy as np

def ckks_dec(cipher_list, ckks_ctx, sk, isBatch=True, sum_masks
    =[], batch_size=0):
    if isBatch:
        plain_list = []
        for idx, cipher_serial in enumerate(cipher_list):
            if cipher_serial == 0:
                zero_pad = [0] * batch_size
                plain_list.extend(zero_pad)
            else:
                plain = ts.CKKSVector.load(ckks_ctx, cipher_serial
    ).decrypt(sk)
                if len(sum_masks) > 0 and sum_masks[idx] > 0:
                    plain = np.array(plain) / sum_masks[idx]

                plain_list.extend(plain)

        return np.array(plain_list)
```

Kode 2.10: Dekripsi dan normalisasi hasil agregasi *Packed* CKKS [14]

Fungsi `dec_params()` digunakan untuk mendekripsi hasil agregasi di sisi klien menggunakan konteks privat yang berisi kunci rahasia (*secret key*). Fungsi ini memanggil `ckks_dec()` untuk mengubah *ciphertext* global menjadi representasi numerik sebelum dilakukan proses *unpacking*.

```
import os, tenseal as ts

def dec_params(cipher_list, sum_masks, args):
    with open(os.path.join(args.data_dir, "context_params"), "rb")
     as f:
        ckks_ctx = ts.context_from(f.read())
    sk = ckks_ctx.secret_key()
    return ckks_dec(cipher_list, ckks_ctx, sk,
                    isBatch=args.isBatch,
                    sum_masks=sum_masks,
                    batch_size=args.enc_batch_size)
```

Kode 2.11: Pemanggilan fungsi dekripsi di sisi klien [14]

Langkah akhir dalam mekanisme *Packed* CKKS adalah proses *unpacking*, yaitu pengembalian hasil dekripsi ke struktur parameter model semula. Setelah *ciphertext* agregat didekripsi menjadi vektor numerik, nilai-nilai tersebut dipetakan kembali ke tensor parameter model berdasarkan bentuk aslinya. Implementasinya dilakukan melalui fungsi `params_tomodel()` seperti berikut.

```
import numpy as np, torch
from collections import OrderedDict

def params_tomodel(model, global_list, params_num, layer_shape,
    args, params_list):
    update_state = OrderedDict()
    model.to('cpu')
    idx_cnt = 0
    if args.isSpars == 'topk':
        for key in model.state_dict().keys():
            layer_size = int(params_num[key])
            tmp = global_list[idx_cnt : idx_cnt + layer_size]
            for j in range(len(tmp)):
                if tmp[j] == 0 and (j == len(tmp) - 1 or tmp[j +
    1] == 0):
                    tmp[j] = params_list[idx_cnt + j]
            update_state[key] = torch.from_numpy(np.array(tmp).
    reshape(layer_shape[key]))
            idx_cnt += layer_size
    model.load_state_dict(update_state)
```

Kode 2.12: Proses *unpacking* hasil dekripsi ke struktur model [14]

Fungsi tersebut memastikan hasil dekripsi (*global_list*) dikembalikan ke bentuk tensor sesuai dimensi parameter model awal. Mekanisme ini menjadi komponen akhir dari *Packed* CKKS, yang menjamin model hasil agregasi dapat digunakan kembali pada iterasi berikutnya tanpa kehilangan struktur internal

maupun presisi numeriknya [14]. Kombinasi konsep *packing* CKKS dan *top-k sparsification* sebagaimana diusulkan oleh Nan Yan *et al.* [14] mampu meningkatkan efisiensi komunikasi dan keamanan tanpa mengorbankan akurasi. Pendekatan ini menurunkan ukuran data terenkripsi hingga lebih dari 16 kali dibanding skema CKKS konvensional, dengan performa model yang tetap stabil, sehingga menjadi solusi efektif untuk sistem pembelajaran terdistribusi yang aman dan efisien.

Mekanisme pembobotan dalam FedPHE pada implementasi resmi Nan Yan *et al.* [14] tidak menggunakan metrik kemiripan model maupun divergensi sebagaimana dijelaskan dalam formulasi teoritis, melainkan mengandalkan skema pembobotan sederhana berbasis jumlah sampel. Pembobotan ini diimplementasikan secara eksplisit pada fungsi `init_prop()` pada kode sumber resmi, yang menghitung proporsi sampel pelatihan dan pengujian untuk setiap klien. Potongan kode berikut menunjukkan cara perhitungan bobot agregasi tersebut.

```
def init_prop(train_dataset,test_dataset,n_clients):
    """
    Initialize weights of aggregation according to the samples of clients.

    Args:
        train_dataset (`dict`):
            Training dataset.
        test_dataset (`dict`):
            Test dataset.
        n_clients (`int`):
            Number of clients to participate.
    Returns:
        train_props (`list`):
            Training weight for each client.
        test_props (`list`):
            Test weight for each client.
    """
    client_n_samples_train = []
    client_n_samples_test = []
    for idx in range(n_clients):
        client_n_samples_train.append(len(train_dataset[idx]))
        client_n_samples_test.append(len(test_dataset[idx]))
    samples_sum_train = np.sum(client_n_samples_train)
    samples_sum_test = np.sum(client_n_samples_test)
    test_props = []
    train_props = []
    for idx in range(n_clients):
        train_props.append(client_n_samples_train[idx]/samples_sum_train)
        test_props.append(client_n_samples_test[idx]/samples_sum_test)
    return train_props,test_props
```

Kode 2.13: Perhitungan bobot agregasi berbasis jumlah sampel [14]

Fungsi `init_prop()` menghitung jumlah sampel pelatihan dan pengujian pada setiap klien, kemudian menormalkannya terhadap total sampel seluruh klien. Hasil normalisasi tersebut disimpan dalam `train_props` dan digunakan sebagai bobot agregasi *(training weights)* pada sisi server. Dengan demikian, kontribusi parameter klien yang memiliki lebih banyak data akan secara proporsional lebih besar dalam pembentukan model global. Skema ini sejalan dengan pendekatan *weighted Federated Averaging*, namun diintegrasikan dengan skema enkripsi homomorfik CKKS sehingga seluruh proses agregasi tetap dilakukan di domain terenkripsi.

Proses penggabungan parameter terenkripsi kemudian dilaksanakan pada sisi server melalui fungsi `aggregatie_weights()`, yang menerima *ciphertext* CKKS dan *mask* dari setiap klien, lalu mengalikan *ciphertext* tersebut dengan bobot `weights_client` (yang berasal dari `train_props`) sebelum dilakukan penjumlahan homomorfik. Mekanisme ini menghasilkan model global terenkripsi yang merupakan kombinasi berbobot dari kontribusi seluruh klien tanpa membuka nilai *plaintext*. Algoritma berikut merangkum prosedur agregasi dalam FedPHE berdasarkan implementasi tersebut.

Algorithm 3 *Packed CKKS Server Aggregation Process* dengan bobot proporsi sampel [14]

Require: Ciphertext dan mask klien $\{(C_i^t, M_i^t)\}_{i=1}^N$, bobot sampel $\{p_i\}$, ukuran batch B, konteks enkripsi ctx

Ensure: Ciphertext global C^t dan mask agregat M^t

for setiap ronde federasi $t = 0, \ldots, T-1$ **do**
  Terima (C_i^t, M_i^t) dari setiap klien
  for setiap batch aktif l **do**
    $C^t[l] \leftarrow \sum_{i=1}^N p_i C_i^t[l]$
    $M^t[l] \leftarrow \sum_{i=1}^N p_i M_i^t[l]$
  end for
  Kirimkan (C^t, M^t) ke seluruh klien
end for

Algoritma 3 menggambarkan proses agregasi global pada sisi server menggunakan skema *Packed CKKS Homomorphic Encryption* dengan agregasi berbobot. Skema ini mengadopsi konsep *secure weighted aggregation* pada *ciphertext* sebagaimana diperkenalkan oleh Nan Yan *et al.* [14].

Pada setiap ronde federasi t, server menerima pasangan *ciphertext* dan

mask (C_i^t, M_i^t) dari seluruh klien yang berpartisipasi. Setiap C_i^t merepresentasikan parameter model klien yang telah melalui proses *packing*, *sparsification*, dan enkripsi CKKS, sedangkan M_i^t digunakan untuk menandai *batch* aktif yang dikirim oleh klien.

Agregasi dilakukan pada tingkat *batch* terenkripsi. Untuk setiap *batch* aktif l, server menghitung *ciphertext* global $C^t[l]$ sebagai penjumlahan linier berbobot dari *ciphertext* klien, yaitu $C^t[l] = \sum_{i=1}^{N} p_i C_i^t[l]$. Bobot p_i mencerminkan proporsi kontribusi klien terhadap model global dan dijumlahkan tanpa melakukan dekripsi, sehingga kerahasiaan parameter klien tetap terjaga. Proses yang sama diterapkan pada *mask* agregat $M^t[l]$ untuk memastikan konsistensi struktur *batch* pada tahap dekripsi di sisi klien.

Penggunaan bobot p_i secara konsisten pada sisi klien dan server mengikuti pendekatan *weighted aggregation* dalam *Federated Learning*. Klien mengalikan parameter model lokal dengan bobot p_i sebelum proses enkripsi, sementara server menggunakan bobot yang sama untuk melakukan agregasi homomorfik. Pendekatan ini ekuivalen dengan melakukan agregasi berbobot pada domain *plaintext*, namun seluruh operasi dilakukan pada domain *ciphertext*, sehingga server tidak memiliki akses terhadap nilai parameter asli.

Dengan mekanisme ini, agregasi global dapat dilakukan secara aman tanpa mengorbankan prinsip dasar *Federated Learning*, serta memungkinkan penggabungan kontribusi klien yang tidak seragam dalam skema terenkripsi.

```
def aggregatie_weights(rec,recv_list,weights_client,total_sum,
                       batch_num,id_list,args,enc_tools = {},
    rep_num = []):
    if args.enc:
        global_cipher = [0] * batch_num
        if args.isSpars == 'topk':
            sum_mask = [0] * batch_num
    for idx,value in enumerate(rec.values()):
        c_id = value[0]
        if args.algorithm == 'ckks':
            ckks_file = os.path.join(args.data_dir + '
    context_params')
            with open(ckks_file, "rb") as f:
                params = f.read()
            ckks_ctx = ts.context_from(params)
            frac = ts.ckks_vector(ckks_ctx,[weights_client[c_id]])
            if args.isSpars == 'topk':
                mask = value[1]
```

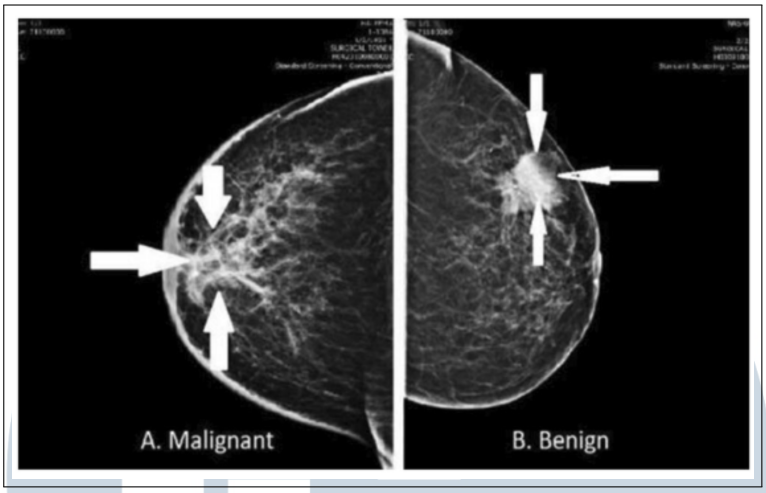
```
                cipher = value[2]
                for batch in range(batch_num):
                    if mask[batch]:
                        cnt = sum(mask[:batch])
                        res = ts.CKKSVector.load(ckks_ctx,cipher[
    cnt]) * frac
                        sum_mask[batch] += weights_client[c_id]
                        if global_cipher[batch]:
                            res += ts.CKKSVector.load(ckks_ctx,
    global_cipher[batch])
                        global_cipher[batch] = res.serialize()
     return sum_mask,global_cipher
```

Fungsi di atas menunjukkan bahwa *ciphertext* setiap *batch* dimuat kembali ke dalam konteks CKKS, dikalikan dengan bobot klien, kemudian dijumlahkan secara homomorfik. Variabel `sum_mask` menyimpan total bobot yang berkontribusi pada setiap *batch* dan digunakan selama proses dekripsi untuk menormalkan parameter global. Pendekatan ini menghasilkan agregasi yang setara dengan *Federated Averaging* berbobot tetapi dilakukan sepenuhnya pada domain terenkripsi menggunakan *Packed CKKS*.

2.3 Klasifikasi Kanker Payudara

Kanker payudara menjadi penyebab utama kematian akibat kanker pada perempuan di seluruh dunia dan menempati peringkat pertama dalam tingkat insidensi global [28, 29, 1]. Deteksi dini berperan penting dalam keberhasilan terapi, sehingga pengembangan metode klasifikasi berbasis citra medis menjadi fokus utama dalam sistem *computer-aided detection* (CAD) [30, 31, 32, 33]. Tujuan klasifikasi adalah membedakan jaringan jinak (*benign*) dan ganas (*malignant*) pada citra *mammogram* untuk mendukung diagnosis klinis yang akurat [31, 30, 33].

Gambar 2.5 memperlihatkan contoh citra *mammogram* dengan label *benign* dan *malignant*. Citra *benign* biasanya menunjukkan pola jaringan yang lebih homogen dan batas lesi yang halus, sedangkan citra *malignant* cenderung memiliki tepi tidak beraturan dan intensitas kontras yang lebih tinggi [34].



Gambar 2.5. Klasifikasi *benign* dan *malignant* [34].

Pendekatan modern berbasis *deep learning*, khususnya *Convolutional Neural Networks* (CNN), memungkinkan ekstraksi fitur secara otomatis tanpa rekayasa manual [1, 33, 35]. CNN mempelajari representasi spasial dan semantik melalui operasi konvolusi bertingkat untuk mengenali pola kompleks seperti tepi, tekstur, dan bentuk lesi [8, 31, 35]. Arsitektur populer seperti VGGNet, ResNet, dan DenseNet menunjukkan performa tinggi pada dataset CBIS-DDSM, INbreast, dan BreakHis [36, 33]. Model-model tersebut umumnya dievaluasi menggunakan metrik akurasi, AUC, dan PR-AUC sebagai ukuran utama kinerja klasifikasi [8, 33].

Keterbatasan utama pada klasifikasi citra medis terletak pada jumlah data yang terbatas dan kebijakan privasi pasien [3, 7]. *Federated Learning* (FL) memberikan solusi dengan memungkinkan pelatihan model secara kolaboratif antar institusi tanpa perlu berbagi data mentah [3, 19]. Setiap institusi melatih model lokal menggunakan data internal dan hanya mengirimkan pembaruan parameter ke server pusat untuk agregasi [8, 1]. Pendekatan ini menjaga kerahasiaan data pasien sekaligus memastikan kepatuhan terhadap regulasi privasi seperti HIPAA dan GDPR [4, 3, 7].

2.3.1 Metrik Evaluasi untuk Klasifikasi Kanker Payudara

Kinerja model dievaluasi menggunakan tiga metrik utama, yaitu *Accuracy*, *Area Under the Receiver Operating Characteristic Curve* (AUC), dan *Area Under the Precision–Recall Curve* (PR-AUC). Ketiga metrik ini digunakan untuk memberikan evaluasi yang komprehensif terhadap performa model klasifikasi kanker payudara, khususnya pada kondisi data medis dengan distribusi kelas yang

tidak seimbang.

Accuracy mengukur proporsi prediksi yang diklasifikasikan secara benar terhadap keseluruhan sampel, dan didefinisikan sebagai:

$$\text{Akurasi} = \frac{TP+TN}{TP+TN+FP+FN}, \tag{2.1}$$

di mana TP, TN, FP, dan FN masing-masing menyatakan *true positive*, *true negative*, *false positive*, dan *false negative* [37].

Perhitungan nilai akurasi dilakukan pada tahap validasi model dengan membandingkan hasil prediksi kelas dan label sebenarnya pada setiap *batch* data. Prediksi kelas diperoleh dengan memilih kelas dengan probabilitas tertinggi dari keluaran model. Implementasi penghitungan akurasi mengikuti mekanisme evaluasi yang digunakan oleh Sánchez *et al.* [8], sebagaimana ditunjukkan pada potongan kode berikut.

```
preds = torch.argmax(probs, dim=1)
correct += preds.eq(labels.view(-1)).sum().item()

accuracy = correct / len(dataloader.dataset)
```

Kode 2.14: Perhitungan akurasi pada tahap validasi model [8]

Kode 2.14 menunjukkan bahwa akurasi dihitung dengan membandingkan prediksi kelas hasil model dan label sebenarnya. Prediksi kelas diperoleh dari probabilitas keluaran model menggunakan operator argumen maksimum, sedangkan nilai akurasi diperoleh dengan menormalisasi jumlah prediksi yang benar terhadap total sampel pada dataset validasi [37].

AUC merepresentasikan luas area di bawah kurva *Receiver Operating Characteristic* (ROC), yang menggambarkan *trade-off* antara *True Positive Rate* (TPR) dan *False Positive Rate* (FPR) pada berbagai ambang keputusan. Kurva ROC memplot hubungan TPR dan FPR yang didefinisikan sebagai berikut [38]:

$$\text{TPR} = \frac{TP}{TP+FN} \tag{2.2}$$

$$\text{FPR} = \frac{FP}{FP+TN} \tag{2.3}$$

Berdasarkan kurva tersebut, AUC diperoleh sebagai integral luas area di

bawah kurva ROC dan dapat dirumuskan sebagai [38]:

$$\text{AUC} = \int_0^1 \text{TPR}\, d(\text{FPR}) = 1 - \int_0^1 \text{FPR}\, d(\text{TPR}) \tag{2.4}$$

Nilai AUC yang semakin mendekati 1 menandakan kemampuan model yang semakin baik dalam membedakan kelas positif dan kelas negatif.

PR-AUC (*Precision–Recall AUC*) digunakan untuk mengevaluasi performa model, terutama pada data dengan distribusi label yang tidak seimbang. Kurva *Precision–Recall* memplot hubungan antara *Precision* dan *Recall*, yang didefinisikan sebagai berikut [38]:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.5}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.6}$$

Nilai PR-AUC diperoleh dari luas area di bawah kurva *Precision–Recall* dan dirumuskan sebagai berikut [38]:

$$\text{AUPRC} = \int_0^1 \text{Precision}\, d(\text{Recall}) \tag{2.7}$$

Nilai PR-AUC yang tinggi mengindikasikan bahwa model mampu mengenali sebagian besar sampel positif dengan tingkat kesalahan prediksi yang rendah [38].

Implementasi penghitungan kedua metrik mengikuti langsung kode evaluasi yang digunakan oleh Sánchez *et al.* [8]. Pada tahap evaluasi, vektor label sebenarnya dan probabilitas keluaran model yang tersimpan dalam `targets` dan `probabilities` terlebih dahulu diratakan menjadi `y_true` dan `y_prob`. Nilai `y_prob` diambil dari komponen kelas positif (malignant), yaitu elemen ke-2 dari vektor probabilitas, kemudian dikonversi dari log-probabilitas menggunakan fungsi eksponensial. Kurva *precision–recall* dan nilai PR-AUC dihitung menggunakan fungsi `precision_recall_curve` dan `auc`, sedangkan AUC diperoleh melalui `roc_auc_score`, sebagaimana ditunjukkan pada potongan kode berikut.

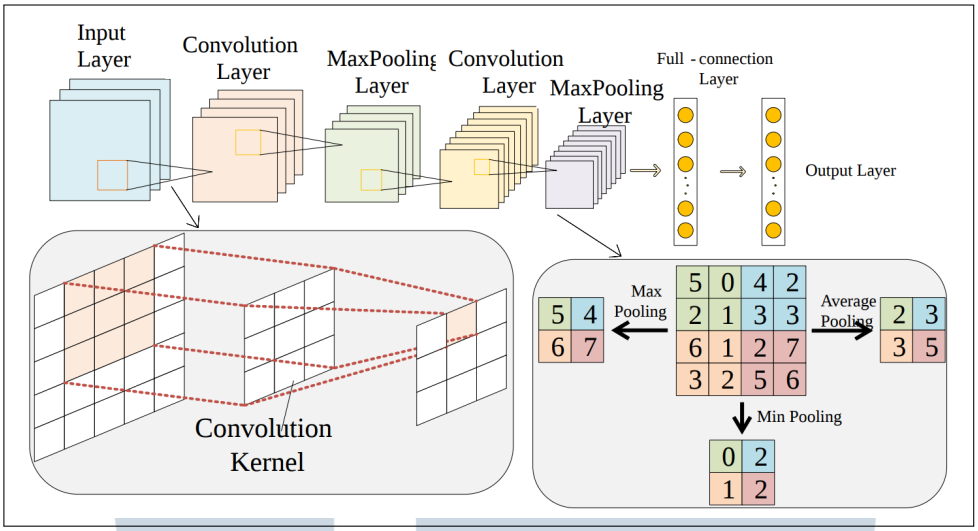
```python
from sklearn.metrics import precision_recall_curve, auc,
    roc_auc_score, confusion_matrix
import numpy as np

# calculate precision and recall for each threshold
y_true = np.asarray([val for sublist in targets for val in sublist
    ])
y_prob = np.asarray([np.exp(val[1]) for sublist in probabilities
    for val in sublist])
y_pred = np.asarray([np.argmax(val) for sublist in probabilities
    for val in sublist])
lr_precision, lr_recall, _ = precision_recall_curve(y_true, y_prob
    )
pr_auc = auc(lr_recall, lr_precision)
roc_auc = roc_auc_score(y_true, y_prob)

print('AUC: {:.4f}'.format(roc_auc))
print('PR AUC: {:.4f}'.format(pr_auc))
print(confusion_matrix(y_true, y_pred))
```

Kode 2.15: Perhitungan AUC dan PR-AUC pada tahap evaluasi model [8]

Kode pada Listing 2.15 menunjukkan bahwa `roc_auc_score(y_true, y_prob)` digunakan untuk menghitung luas di bawah kurva ROC, sedangkan kombinasi `precision_recall_curve(y_true, y_prob)` dan `auc(lr_recall, lr_precision)` menghasilkan nilai PR-AUC. Vektor `y_pred` dibentuk dari argumen maksimum probabilitas (`np.argmax`) untuk memperoleh prediksi kelas diskret, yang kemudian digunakan dalam `confusion_matrix(y_true, y_pred)` guna mengevaluasi distribusi prediksi benar dan salah model.

2.4 CNN ResNet-22

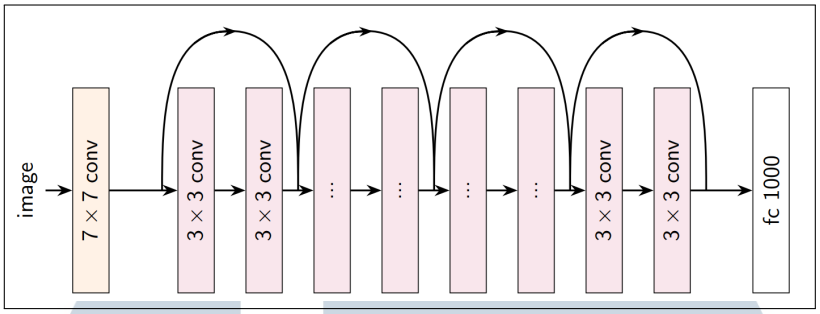
Convolutional Neural Network (CNN) merupakan jenis jaringan saraf tiruan yang dirancang khusus untuk mengolah data berbentuk citra. CNN terdiri atas beberapa lapisan utama yaitu *convolution layer*, *pooling layer*, dan *fully connected layer*. Lapisan konvolusi berfungsi mengekstraksi fitur spasial seperti tepi, bentuk, dan tekstur, sedangkan lapisan *pooling* bertugas mereduksi dimensi fitur tanpa menghilangkan informasi penting. Lapisan *fully connected* berperan melakukan klasifikasi berdasarkan fitur yang telah diekstraksi. Pendekatan ini telah terbukti efektif untuk berbagai tugas pengenalan pola, termasuk klasifikasi kanker payudara menggunakan citra *mammogram* [39, 40].



Gambar 2.6. Arsitektur CNN [41].

Gambar 2.6 menggambarkan proses kerja CNN yang diawali dengan masukan citra pada lapisan input, diikuti operasi konvolusi menggunakan kernel untuk mengekstraksi fitur lokal. Hasilnya kemudian diproses melalui *pooling* layer untuk mereduksi ukuran fitur, dan diakhiri dengan lapisan klasifikasi yang menghasilkan keluaran berupa label kelas citra. CNN mampu mempelajari representasi hierarkis secara otomatis dari fitur sederhana menuju fitur kompleks tanpa rekayasa manual.

Arsitektur CNN yang sangat dalam sering menghadapi masalah *vanishing gradient* yang menghambat proses pelatihan. Untuk mengatasi hal ini, He et al. [40] memperkenalkan arsitektur *Residual Network* (ResNet). Ciri khas ResNet adalah adanya *skip connection* yang memungkinkan sinyal input diteruskan langsung ke lapisan yang lebih dalam dan dijumlahkan dengan keluaran fungsi residual. Struktur ini memungkinkan jaringan yang sangat dalam dilatih tanpa mengalami degradasi performa. ResNet terbukti meningkatkan akurasi pada berbagai tugas klasifikasi dan menjadi fondasi bagi banyak model visi komputer modern. Gambar 2.7 memperlihatkan contoh arsitektur ResNet dengan koneksi residual antar blok konvolusi.



Gambar 2.7. Arsitektur *Residual Network* (ResNet) [41].

Jiménez-Sánchez *et al.* [8] mengusulkan arsitektur *ResNet-22* sebagai *encoder* ringan untuk klasifikasi citra medis, khususnya citra *mammogram*. Arsitektur ini terdiri atas lima tahap blok residual berturut dengan konfigurasi [2, 2, 2, 2, 2], yang diakhiri dengan tiga lapisan klasifikasi penuh (*fully connected*) berukuran 128, 64, dan 2 neuron. Pendekatan ini menjaga kedalaman jaringan tetap efisien sambil mempertahankan kemampuan ekstraksi fitur kompleks. Implementasi struktur *feature extractor*, *classifier*, dan *domain discriminator* pada model tersebut dirangkum pada Tabel 2.2.

Tabel 2.2. Arsitektur ResNet-22 CNN [8]

Layer	**Configuration**
	F: Feature Extractor
1	Conv(1, 16, 3, 1, 1), MaxPool(3, 2, 0, 1)
2.1	Block(16, 16, 3, s_1=1, s_2=1, 1), Conv(16, 16, 1, 1)
2.2	Block(16, 32, 3, s_1=2, s_2=1, 1)
3.1	Block(16, 32, 3, s_1=2, s_2=1, 1), Conv(16, 32, 1, 2)
3.2	Block(16, 32, 3, s_1=2, s_2=1, 1)
4.1	Block(32, 64, 3, s_1=2, s_2=1, 1), Conv(32, 64, 1, 2)
4.2	Block(32, 64, 3, s_1=2, s_2=1, 1)
5.1	Block(64, 128, 3, s_1=2, s_2=1, 1), Conv(64, 128, 1, 2)
5.2	Block(64, 128, 3, s_1=2, s_2=1, 1)
6.1	Block(128, 256, 3, s_1=2, s_2=1, 1), Conv(128, 256, 1, 2)
6.2	Block(128, 256, 3, s_1=2, s_2=1, 1)
	Cls: Classifier

Lanjut ke halaman berikutnya

Tabel 2.2. Arsitektur ResNet-22 CNN [8] (lanjutan)

Layer	Configuration
1	FC(256, 128), BN, ReLU, Dropout(0.5)
2	FC(128, 64), BN, ReLU, Dropout(0.5)
3	FC(64, 2), Sigmoid
D: Domain Discriminator	
1	FC(256, 4), ReLU
2	FC(4, 2), Sigmoid

Implementasi *feature extractor* (*encoder*) dan *classifier* pada model Jiménez-Sánchez *et al.* [8] ditulis menggunakan pustaka `PyTorch`. Struktur berikut memperlihatkan bahwa modul `Classifier` terdiri atas tiga lapisan linear dengan aktivasi ReLU dan normalisasi *batch*, sementara bagian `Encoder` memanfaatkan arsitektur `ResNet-22` untuk mengekstraksi fitur dari citra *mammogram*.

```
class Classifier(nn.Module):
    def __init__(self, num_classes=2):
        super(Classifier, self).__init__()
        self.encoder = Encoder()
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, num_classes)
        self.bn1 = nn.BatchNorm1d(128)
        self.bn2 = nn.BatchNorm1d(64)
        self.drop1 = nn.Dropout()
        self.drop2 = nn.Dropout()

    def forward(self, input):
        logits = self.encoder(input)
        logits = F.relu(self.bn1(self.fc1(logits)))
        logits = self.drop1(logits)
        logits = F.relu(self.bn2(self.fc2(logits)))
        logits = self.drop2(logits)
        logits = self.fc3(logits)
        probs = F.softmax(logits, dim=1)
        return probs, logits
```

Kode 2.16: Struktur utama *Classifier* dan *Encoder* pada model Jiménez-Sánchez [8]

Selain struktur klasifikasi, arsitektur *ResNet-22* diimplementasikan menggunakan blok residual yang diadaptasi dari *ResNet v2*. Setiap blok residual terdiri atas dua lapisan konvolusi dengan normalisasi *batch* dan fungsi aktivasi ReLU, sebagaimana ditunjukkan pada potongan berikut.

```
class BasicBlockV2(nn.Module):
    def __init__(self, inplanes, planes, stride=1, downsample=None
    ):
```

```
        super(BasicBlockV2, self).__init__()
        self.bn1 = nn.BatchNorm2d(inplanes)
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv2 = conv3x3(planes, planes)
        self.downsample = downsample
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        residual = x
        out = self.relu(self.bn1(x))
        if self.downsample is not None:
            residual = self.downsample(out)
        out = self.conv1(out)
        out = self.relu(self.bn2(out))
        out = self.conv2(out)
        return out + residual
```

Kode 2.17: Blok residual dasar (*BasicBlockV2*) pada arsitektur ResNet-22 [8]

Fungsi `resnet22()` membangun jaringan dengan lima blok residual berturut, masing-masing terdiri atas dua lapisan konvolusi. Implementasi ringkasnya ditunjukkan pada potongan berikut.

```
def resnet22(input_channels, activation):
    return ViewResNetV2(
        input_channels=input_channels,
        activation=activation,
        num_filters=16,
        blocks_per_layer_list=[2, 2, 2, 2, 2],
        block_strides_list=[1, 2, 2, 2, 2],
        block_fn=BasicBlockV2,
    )
```

Kode 2.18: Implementasi fungsi pembentuk arsitektur *ResNet-22* [8]

Arsitektur ResNet-22 dirancang agar ringan secara komputasi namun tetap mampu mengekstraksi fitur kompleks dari citra medis resolusi tinggi. Struktur residual memungkinkannya mempertahankan stabilitas gradien selama pelatihan, sehingga model dapat beradaptasi terhadap variasi data lintas institusi [8].